Cúram 8.1.2

Inside Eligibility and Entitlement Using Cúram Express Rules Guide

Note

Before using this information and the product it supports, read the information in Notices on page 243

Edition

This edition applies to Cúram 8.1, 8.1.1, and 8.1.2.

Contents

1 Developing with Eligibility and Entitlement by using Cúram Express Rules

Use this information to learn how to set up and use Cúram Express Rules to return eligibility and entitlement results for cases. Detailed examples of eligibility and entitlement results are provided as well as descriptions of the processing that returns these results; and instructions on how to use this processing.

Note: The key decision factors feature is deprecated. For more information, see Deprecated features.

1.1 Introduction

Purpose

The purpose of this guide is twofold:

- To provide an inside look at how a case's eligibility and entitlement is assessed using Cúram Express Rules (CER)
- To provide instructions on how to set up and use the application to return eligibility and entitlement results

To achieve this purpose, the guide provides detailed examples of eligibility and entitlement results, descriptions of the processing that returns these results, and instructions on how to use this processing.

The guide promotes a "designing in stages" approach, focusing first on returning the basic eligibility and entitlement results (i.e. eligible periods and entitlement amounts), and then on designing solutions that return more complex eligibility and entitlement results (e.g. returning eligibility and entitlement results with detailed explanations of how those results were derived).

Note: The assessment of cases is carried out by the Eligibility and Entitlement Engine, which for the sake of brevity is referred to as simply "the Engine" throughout this guide.

Audience

This guide is intended for a technical audience interested in understanding how the Engine fits into case processing. Designers and developers responsible for building and customizing products will find this document useful in conjunction with reading the `How to Build a Product` guide.

This guide is also intended to help system operators understand the operational requirements of the Engine, particular with regard to bulk case reassessment.

Related Reading

There are several related documents, some of which provide helpful background information, others that provide more detailed information on topics covered in this guide. The following provides a brief description of the related reading materials available:

Table 1: Description of Related Reading

Document Name	Description
Integrated Case Management Guide	This provides a business overview of Cúram Integrated Case Management, focusing specifically on the needs-to-delivery side of case processing.
Integrated Case Management Configuration Guide	This describes the configuration settings for cases including product configuration, configuration options for case pages, and a description of the application properties relating to case processing.
How to Build a Product	This provides sample-based instructions on how to build a product. It starts off with a simple product sample, and builds upon complexity with varying product samples.
Cúram Dynamic Evidence Configuration Guide	This describes the configuration settings for Cúram Dynamic Evidence and includes instructions on how to use the Cúram Dynamic Evidence Editor to manage case evidence.
Cúram Express Rules Reference Manual	This describes the Cúram Express Rules language and provides instructions on how to design rule sets using the Cúram Express Rules Editor.

Chapters in this Guide

The following list describes the chapters within this guide:

- **1.2 Eligibility and Entitlement Processing at a Glance on page 13**
  This chapter provides an overview of the end-to-end eligibility/entitlement processing performed by the Engine. It provides an 'at a glance' view of this processing, with the remaining chapters in this guide examining eligibility and entitlement processing 'under the microscope'.
- **1.3 Navigating Determinations on page 26**
  This chapter describes the different types of determinations and how case workers can view the details of those determinations.
- **1.4 Calculating and Displaying Eligibility and Entitlement on page 28**
  This chapter describes in detail how the Engine calculates and displays core eligibility/entitlement information for a case.
- **1.5 (deprecated) Calculating and Displaying Key Decision Factors on page 56**
  This chapter describes in detail how the Engine calculates and displays "key decision factors" to help case workers understand a case's eligibility/entitlement
- **1.6 Calculating and Displaying Decision Details on page 70**
  This chapter describes in detail how the Engine calculates and displays free-form "decision details" to help case workers understand a case's eligibility/entitlement
- **1.7 Understanding Rule Object Converters and Propagators on page 97**
  This chapter describes how the Engine uses configurable "converters" and "propagators" to make custom entities and evidence types available for CER rules calculations.

- **1.8 How Determinations Are Stored on page 147**
  This chapter describes how the Engine stores determination results so that they can be retrieved later for viewing by case workers and as inputs into financial processing.
- **1.9 Scheduling Financials on page 156**
  This chapter describes how the Engine interacts with Cúram Financials to make payments, deductions and corrections.
- **1.10 Reassessment - Handling Changes in Circumstance on page 170**
  This chapter describes how the Engine processes changes of circumstances and reassesses cases.
- **1.11 Incremental Design and Evolution on page 191**
  This chapter provides advice on how to implement complex products and how to maintain them when legislation changes.
- **1.12 Compliancy for eligibility and entitlement on page 206**
  This appendix describes how to develop with the Engine in a compliant manner.
- **1.13 The Eligibility and Entitlement Engine API and Customizability on page 210**
  This appendix describes the API and extension points for the Engine.
- **1.14 Extensions to Cúram Express Rules on page 213**
  This appendix describes the assessment-specific CER expressions and annotations contributed by the Engine.
- **1.16 Environment Variables on page 238**
  This appendix lists the Cúram Environment Variables that you can set to change the Engine's behavior.

1.2 Eligibility and Entitlement Processing at a Glance

Introduction

At a glance, the main objective of the Engine is to work with Cúram Express Rules (CER) to determine case eligibility and entitlement over the lifetime of the case. CER is responsible for applying rules logic to real world data in order to make decisions regarding eligibility and entitlement.

The starting point for case eligibility and entitlement is the Product. A Product contains all the configuration details which specify which CER rules to use when determining eligibility entitlement. Once a Product has been *configured*, its configuration can be used to *calculate* and *store* a *determination result* based on *input* data.

This determination result is used to *generate financials* and is *retrieved* when a case worker user views eligibility and entitlement details for the case. When circumstances change which affect an *assessment* for an active case, the Engine can automatically *reassess* the case.

The Engine manages the reassessment of case eligibility and entitlement through the use of a Dependency Manager. The Dependency Manager stores dependencies when they are identified during the calculation of eligibility and entitlement, and then identifies items that have changed and queues them for later processing, typically in deferred processing. When the deferred process is executed, the Dependency Manager examines its stored dependency records to identify the cases that require reassessment, and for each identified case it uses to CER to re-calculate the determination result.

The below list describes each of the stages in eligibility and entitlement processing, incorporating the above-mentioned terms.

- **Configure**

  A product must be configured before it can be used to create product delivery cases. Over its lifetime, a product's configuration can be updated, e.g. in response to changes in legislation.
- **Input**

  Any input data which affects the case's eligibility and entitlement must be gathered so that it can be converted into CER Rule Objects that are used by CER to perform calculations on that data.
- **Calculate**

  The Engine requests that CER performs calculations to provide an initial determination result for a case. The dependencies on input data used during the calculations are stored and thereafter, if any input data changes which has a bearing on the determination result, the Dependency Manager will automatically use the stored dependencies and run the CER calculations again to see if the overall result has changed.
- **Determination Result**

  A Determination Result is the overall output from CER eligibility/entitlement rules calculations, and contains the "three Es" (Eligibility, Entitlement, Explanation) over the full lifetime of the case.
- **Store**

  The Engine takes a snapshot of the Determination Result and stores it so that it can be used to generate financials, display details to users and act as an audit of how decisions regarding a case were arrived at.
- **Schedule Financials**

  The Engine integrates with the application's Financial Engine to schedule financials (typically, payments). If cases are retrospectively reassessed, then over and under-payments are automatically handled.
- **Retrieve**

  Case workers can view the full history of a case. They can see the eligibility, entitlement and explanation for a case over its full lifetime (including cases which are open-ended). A full history of determinations is kept, so that the user can see how corrections have been made to determinations as circumstances have changed in the real world and/or corrections have been made to input data held on the system.
- **Assessment**

  The first determination for an active case is recorded as its initial assessment. This assessment is based on the circumstances of the case at the time it was activated and gives rise to the initial financials for the case.
- **Reassessment**

  When circumstances change for an active case, or there is a change to the product which affects many cases, then the Engine reassesses the case, possibly leading to a new determination result which in turn can affect the financials for the case.

The remaining sections in this chapter provide more detail on eligibility and entitlement processing and include an example story that spans the sections.

Product Configuration

An agency must decide what products to offer its clients, and must create a Product in their application.

There are several details that must be configured for a product (for details, see the *`How to Build a Product`* guide). Some important configuration items highlighted here are:

- the types of evidence available on integrated cases and product delivery cases;
- the types of input data that can affect eligibility and entitlement calculations;
- the business rules for how to calculate eligibility and entitlement to the product (and explanations), based on input data; and
- how the product's lifetime divides up into different "product periods", with different business rules for legislation changes.

Once a product has been set up in the application, case workers are able to create individual product delivery cases and progress those cases through a case life cycle, including eligibility and entitlement processing. For more details on the case life cycle, see the *`Cúram Integrated Case Management Guide`*.

Example

An agency decides to offer a new product to provide financial assistance to lone parents (the "Lone Parent" Benefit).

The business rules for the product dictate that eligibility decisions are based on a number of criteria, including:

- the age of the child;
- evidence that the child normally resides with the parent, and that the parent has no live-in partner; and
- a means test, of the family's income.

Product-design users in the agency design a new product configuration, which includes:

- New types of evidence for personal circumstances details pertinent to the Lone Parent Benefit product (relationships from the claimant to their child or children), living arrangements, employments and incomes, etc.);
- Eligibility/Entitlement/Explanation rules that calculate based on the parent's living arrangements and their child or children's ages; and
- Rule Object data configurations that allow the personal details of the parent and child, and the new types of evidence, to be made available to CER for rules processing.

The product-design users notify the agency's case workers that the new product is now available for use in the creation of product delivery cases to deliver benefits.

Recording of Input Data

Changes to data which may affect case eligibility and entitlement decisions can come in many forms, including:

- new evidence records for events that have occurred in the real world;
- corrections to data which were recorded incorrectly or declared fraudulently; and
- product-wide data common to all cases.

Changes to such data can affect both:

- eligibility/entitlement decisions already made; and/or
- eligibility/entitlement decisions to be made in the future.

To accommodate both these needs, the Engine knows the types of data required by rules processing, and uses the Dependency Manager to manage changes made to this data through the use of "precedent change sets". When a change is made to data that can affect eligibility/entitlement decisions, an item is added to a precedent change set. The points in the data capture life cycle where this typically occurs are below:

Table 2: When Do Precedent Change Set Items Get Created?

What input data is recorded?	When is an item added to a precedent change set?
Product Delivery Case	When the case is created
Personal details	When the personal details are recorded on the system, but only for types of data configured to be pertinent to eligibility/entitlement calculations (any other details are ignored).
Evidence records for a case	When the evidence is activated, for the types of evidence configured to be pertinent to eligibility/entitlement calculations (NB not when the evidence is first recorded, as only when the evidence is activated is it deemed trustworthy for eligibility/entitlement calculations).
Product-wide configuration information	When an administrator publishes changes to rate tables, rule sets, data configurations, or product configurations, for those items configured to be pertinent to eligibility/entitlement calculations.

For all input data aside from product-wide configuration information, when an item is written to a precedent change set this results in the creation of a deferred process that when executed uses the dependencies stored by the Dependency Manager to determine if any dependents exist for the precedent change set item, and if so re-calculates those dependents. For changes to product-wide configuration information an item is written to a precedent change set that is executed in batch mode.

Example

A case worker interviews a claimant who is a single father of a young daughter. The case worker advises the father to apply for Lone Parent Benefit, and the father agrees to make such an application.

The case worker checks whether the father is already registered on the system, and finds that he is not registered, and so the case worker must register the father's personal details before a case can be created.

When the case worker completes the registration of the father's personal details, the system stores those details. The system also creates a precedent change set item which results in the execution of a deferred process. Because no dependencies on the father's personal details have yet been established, no recalculations occur.

Having registered the father on the system, the case worker can now proceed to create a case for the Lone Parent Benefit for the father. The case worker creates a product delivery case for the father's claim, and the system stores a case record. The system also creates a precedent change set item for the creation of the case that results in the execution of a deferred process, but again because no dependencies have yet been established, no recalculations occur.

The case worker asks the father for information which the agency needs in order to make a determination on the case. Initially, the case worker asks for personal details of the daughter (including her date of birth). The case worker searches for the daughter on the system, but finds no registration records and proceeds to register the daughter's personal details and also records the daughter as an additional member of the case. The system stores these details and creates a precedent change set item for each, but again deferred processing results in no recalculation.

The case worker goes on to ask the father for details of his living arrangement (i.e. partners and spouses) and income for the household. The case worker records this information (which can change over time) as evidence. The evidence is "in edit" and so has not yet become trusted data, and so at this point no precedent change set items are created.

The case worker satisfies herself that the evidence presented by the father is indeed correct, and goes on to activate the evidence. The system marks the evidence records as active.

Rules Calculations and Determination Results

When eligibility is determined within a product delivery case (for CER-based products) the system retrieves any input data which affects the case's eligibility and entitlement from the relevant entities and creates CER rule objects in memory which are used by CER to perform calculations on that data. This includes the creation of a "case" rule object responsible for calculating the determination result for a case. A determination result includes the eligibility, entitlement and explanation for the lifetime of a case, and a request to calculate this result can occur in either a "active" or "reactive" way.

The following sections give a high-level summary of:

- what a determination result contains;
- what triggers the calculation of a determination result; and
- how a determination result is calculated.

What a Determination Result Contains

Each determination result contains information regarding the eligibility and entitlement of a case over the case's lifetime and can include detailed explanations on the case eligibility and entitlement.

The Three Es: Eligibility, Entitlement, and Explanation

A determination result includes a range of decision information (provided by business rules) which can be broken down into the categories referred to as the 'three Es': eligibility, entitlement, and explanation.

Note: The key decision factors feature is deprecated. For more information, see Deprecated features.

- **Eligibility**

The overall "yes" or "no" for whether the claimant is eligible for the product. Typically there will be business rules which either "rule in" or "rule out" the case according to details of the case (including personal details of the members of the case and evidence of their circumstances).

- **Entitlement**

  The objectives which the claimant (and possibly other parties) are entitled to. For benefit products, typically there will be monetary objectives, perhaps broken down into separate components. The case's entitlement is often an answer to the question "how much should the claimant receive?", but objectives can be used for other purposes too. Note that a case's entitlement only applies during periods of eligibility - whenever the case is ineligible, there is no entitlement.
- **Explanation**

  The explanation (aimed at the case worker) for *why* the eligibility and entitlement calculation results are what they are. For periods of eligibility, the explanation typically contains a description of why the case is eligible, and for periods of ineligibility the explanation typically contains one or more reasons why the case is ineligible. The explanation (when shown to the case worker) contains a number of tabs for explaining entitlement calculations broken down into different categories, and also a display of important events which have a bearing on the case's eligibility and entitlement.

Visually, the Engine presents explanations in these ways:

- a graphical view showing key decision factors. Each factor is shown as a date on which either an important event occurred (for instance, the date of birth of a new child) or an important quantity changes value (for instance, a rise in income); and
- display of arbitrary *decision details* laid out on a dynamic UIM page, for a period during the case's lifetime for which the explanation is unchanging.

It can be useful to observe that the requirements that underpin eligibility/entitlement and explanation calculations can come from very different sources:

- **Eligibility and Entitlement**

  The requirements for eligibility and entitlement calculation typically have their roots in legislation or policy documents, and thus are more-or-less "set in stone" when it comes to the implementation of CER rules for providing those calculations. The job of the rules analyst and developers is the *science* of translating "legalese" into CER rules, clarifying any uncertainties along the way. The acid test of the implemented CER rules is whether they meet the proscribed legislation and/or policy. The rules developer can exercise ingenuity in implementing the rules in the simplest way possible, but in the main there is little creativity involved in the implementation task.
- **Explanation**

  By contrast, the requirements for explanation are much looser, and typically center around "whatever can be displayed in order to help the case worker understand the case, and/or help the case worker answer questions from claimants about the case". As such, the analysis and development of CER rules for explaining a determination are much more akin to *art* than science. The initial implementation of explanation rules may be based around the best guesses of what questions might be asked of case workers, and so it is recommended that explanation rules be implemented in such a way that they can be easily enhanced later *without* needing any changes to the underlying (set-in-stone) eligibility and entitlement rules.

The Determination Result Covers the Lifetime of a Case

The eligibility, entitlement and explanation may vary over the *lifetime* of a case. The lifetime of a case is the period of time between the case's start and end dates, inclusive. Each case may bear an actual or expected start date; if an actual start date is present then it is used as the start of the case, otherwise the expected start date will be used. Similarly, a case may have an actual end date and/or an expected end date, which governs the end date of the case's lifetime (with the actual date taking precedence over any expected date).

Important: The end date for a product delivery case is optional. A case without an end date is known as an "open-ended" case.

Each product is configured to specify whether or not it allows open-ended cases.

Open-ended cases may give rise to open-ended decisions and, ultimately, open-ended financial schedules (i.e. "pay until further notice").

In practice, each open-ended case will ultimately end due to some real-world event (and an end date will be recorded, and the case will eventually be closed).

The lifetime of a case may include:

- **Periods in the past**

  These are periods which have already occurred, and (assuming that the agency has a correct record of all pertinent real-world events) will have been correctly calculated and assessed. For financial components, these past periods will have already been paid (or billed). Any retrospective reassessment of past periods (arising from corrections, or a lag between events occurring in the real world and their subsequent notification to the agency) may result in changes to the determination for a past period, and for these periods the system may need to make corrections to financials, e.g. through the use of over/under payment cases.
- **Periods in the future**

  These are periods which have yet to occur, and represent the system's "best guess" regarding determination according to what is already known about the real world. New events which come to light may cause this "best guess" to change, but in general a change in prediction of future eligibility/entitlement will not require corrections to financials (except, perhaps, if payments are made in advance rather than in arrears).

Because a case's determination is a value calculated for the case's lifetime, any change in the case's start or end date (e.g. the extension of a case's expected end date) will cause the case to be reassessed. From a CER perspective, the case's start and end dates are simply input data in the same way that evidence, personal data and rates are.

In general, the eligibility, entitlement and explanation for a case will tend to change on the same dates. However, in some cases, not all will change - for example, it is possible for a case to remain constantly ineligible, but the reason for *why* the case is ineligible may vary over time, and hence the case's explanation may change value on a date even though the case's eligibility does not change value on that date.

Of course, the future gradually becomes the past at every passing moment; and so if "left unchecked" (i.e. there are no changes to input data), the predictions made about future eligibility/entitlement will gently flow into past "actual" eligibility/entitlement, and will be used as the basis for new financials.

What Triggers the Calculation of a Determination Result

The calculation of a determination is triggered at various points of the case life cycle, in one of two ways:

- **Active**

  For various user-initiated case events, the Engine explicitly requests the value of a determination result from CER. CER rule objects are created in memory through the use of a converter that retrieves data from the relevant database entities and rules are then executed against this data.
- **Reactive**

  Whenever the Dependency Manager detects that the values used in a case's assessment determination result have changed, the Dependency Manager invokes the Engine to reassess the case.

The following sections explain Active and Reactive determinations in more detail.

Active Determination Calculation Requests

The Engine explicitly requests the value of a determination result from CER at these points in the case life cycle:

- **Prior to evidence activation**

  A case worker can manually request an interim determination for a case, based on either the active evidence only, or the active and in-edit evidence on the case. This kind of determination is made available on a "what if" basis, and the results are essentially disposable once seen by the case worker who requested the determination. A request for an interim determination does not result in the storage of any dependencies by the Dependency Manager.
- **Approval**

  The system automatically records a snapshot of a case's determination result when the case is submitted for approval, approved or rejected. This snapshot provides an audit trail of the state of the case's eligibility/entitlement at the point at which important decisions are made about the case (e.g. the information available to a supervisor at the point when the supervisor chose to accept or reject the case). Determinations created during the approval process do not result in the storage of any dependencies by the Dependency Manager.
- **Activation**

  When a case is first activated, the system automatically records a snapshot of the case's determination result, and this result is used as an input into financials processing. The Engine also uses the Dependency Manager to store the dependencies that were identified and used by CER during its calculations.
- **Manual Redetermination**

  To support the ability to take a snapshot of a case's determination result at key points in the life cycle of a case, for example when an appeal is created, a facility exists for a case worker to manually request a redetermination for a case. For CER-based cases, the system will in most situations be up-to-date with changes that affect a case and so any manual request to redetermine the case will likely not result in a change to the case's determination. Determinations created during a manual request for redetermination may cause changes to dependencies to be stored by the Dependency Manager.
- **Generating Payments**

> When the application is generating payments for a case, it can be configured to reassess the case's determination first in order to ensure that payments are never generated for out of date determinations. This behaviour is controlled via the application property `curam.batch.generateinstructionlineitems.dontreassesscase`.

At a CER level, a request for the value of a determination result is a request for the value of a particular rule attribute on the CER rule object for the product delivery case. As described above, upon activation of the case and creation of the initial assessment determination, the Engine will store the dependencies that were identified and used by CER during its calculations. This includes all of the dependencies of the rule attribute that holds the determination result. Once identified, these dependencies are now established so that the system can react to changes in input data which may affect its value (and may cause reactive re-determinations).

Reactive Determination Calculations

Once a case has been activated and an initial assessment determination has been created, then the items upon which the determination result depends will be established as dependents so that the system may react to changes which may affect it. This is in sharp contrast to traditional case processing which is centered around having to write processing to identify affected cases; instead, the paradigm is much closer to that of a "spreadsheet", which automatically calculates results whenever input values change.

When any data that affects eligibility and entitlement is subsequently changed, an item will be written to a precedent change set, and a request for a deferred process made. When executed, the deferred process will then use the stored dependencies to determine the dependents for the precedent items that were changed and recalculate those dependents. A dependent which relates to a case[1]will cause that case to be reassessed.

The Dependency Manager does not really "understand" different types of input data; rather, it just knows that a dependency exists from a determination result value to the various input data values used to calculate it (in the same way as a spreadsheet does not understand the purpose of the data typed into it). From a business perspective, the types of input data change that can affect determination results will typically include:

- changes to case data, e.g. start and end dates;
- changes to personal data, e.g. the correction of a date of birth or the recording of a date of death;
- changes to rate data, e.g. an increase in a benefit payment rate; and
- changes to product configuration, e.g. the introduction of a new period of legislation.

The Dependency Manager's lack of understanding of types of input data is the strength underpinning how reactive determinations work. In a spreadsheet-like way, the Dependency Manager will reassess all cases that are dependent upon the data that has been changed, regardless of whether that change ends up affecting zero, one or very many cases. For some changes to input data, the Engine may reassess cases but find that the overall determination result value has not changed, and in these circumstances no new determination will be stored.

---

[1] The Dependency Manager manages dependencies for items other than cases, too - e.g. Advice.

How a Determination Result Is Calculated

A determination result is calculated by the execution of a chain of Cúram Express Rules (CER) rules.

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

These rules fall into two categories:

- **Fixed Rules**

  The Engine includes some fixed rules which cannot be customized. These rules are responsible for calculating the determination over different legislative periods, and for creating product-specific rule objects for the execution of product-specific rules.
- **Product-Specific Rules**

  Each product will have rules which contain the business and technical logic for the calculation of eligibility/entitlement and explanations for that product. These rules also effectively link the product to types of evidence, personal data and rates to be used in determination calculations for the product.

A high-level summary of the chain of CER rules execution for a determination is as follows:

- the case's lifetime is calculated, with reference to its actual and expected start and end dates;
- the case's lifetime is checked against the product periods for the product, to see which of these periods overlap with the case's lifetime. These product periods contribute to the case's determination (any other product periods lying wholly outside the case's lifetime are ignored);
- for each contributing product period, product-specific rule objects for the case are created (one rule object for the case's eligibility/entitlement, optionally[2]one rule object for key decision factors, and one rule object for each category of decision details[3]), according to the configuration of the product period;
- for each product-specific rule object created, its output attribute values are calculated to get the eligibility/entitlement/explanation results for a portion of the case's lifetime;
- the calculation of eligibility/entitlement/explanation results will involve the execution of lower-level rules, which may perform searches to retrieve personal data, evidence and/or rate data. These searches effectively link the case to the input data on which the case determination result ultimately depends.
- the results arising from different product periods are then "spliced together" to arrive at an overall determination result which covers the full lifetime of the case.

Example

When the case worker has gathered evidence for the father's Lone Parent Benefit case, the father asks how much benefit he is likely to receive. The case worker requests a manual determination based on the in-edit evidence gathered (the evidence has not yet been approved).

The system actively calculates a determination result that:

---

[2] It is optional whether to configure the product period for key decision factors. If key decision factors are not configured for the product period then no key decision factors rule object is created.

[3] It is optional whether to configure the product period for any decision details categories. If no decision details categories are configured for the product period then no decision details rule objects are created.

- shows the father is *eligible* for benefit from 1st January 2001, up until his daughter is expected to reach the age of majority on 14th December 2010 (which, at the time the claim is created, is many years in the future);
- shows the father is *entitled* to $20 per week, but that this amount will rise to $25 per week from 1st June 2002, when there is a planned increase in benefit payment rates;
- *explains* that the father meets all the eligibility criteria but has been means tested and thus is entitled to a reduced rate of benefit due to his income.

At this point, no dependencies are stored by the Dependency Manager because the case has not yet been activated.

The case worker activates the evidence, which results in items being written to a precedent change set, and a deferred process being created. Because no dependencies yet exist for the case, no recalculation occurs when the deferred process is executed.

The case worker then submits the case for approval. The system creates a snapshot of the determination result (recalculated above) for audit purposes, and routes the case to a supervisor. The supervisor reviews the case and approves it, and the system creates another snapshot of the determination result, and links it to the record of the case approval. The case is activated and the system creates the initial assessment determination result. The dependencies that were identified during the calculation of the determination result are stored.

On 1st March 2003, the father remarries, and thus stops being a lone parent. The father is somewhat lax with regard to informing the agency, and only gets around to telling a case worker about his new marriage three months after it occurred. The case worker records the change in the father's personal circumstances and activates the change in evidence. An item is written to a precedent change set, and because the case is dependent upon the evidence in the case, CER recalculates a new determination for the case. The determination shows that the father's eligibility stopped three months previously (on 1st March 2003, his date of marriage) and that he has been overpaid in the meantime. The system initiates overpayment processing to recover the amount overpaid. The case worker can see from the latest determination that the reason that eligibility ended was due to the father no longer satisfying the "lone" condition of the product's business rules.

On 1st May 2008, the father's wife dies, and the father notifies the agency that he is again a lone parent. The change in his personal circumstances again results in an item being written to a precedent change set, and the case once again being recalculated. It transpires that over the last few years, increases in the father's income have pushed him out of the low-income bracket, and so despite now being "lone", he is ineligible for the Lone Parent Benefit due to his income level. The determination viewed by the case worker shows that when his wife died, the father continued to be ineligible for benefit, but the reason for ineligibility changed (prior to his wife's death, the reason was that he was not "lone", after her death the reason was that his means test failed due to his income level). The determination continues to show that when his daughter finally reaches the age of majority (which is still in the future), he will continue to be ineligible, but for a different reason again (namely that the father has no minor dependent). The determination also continues to show all the historical changes in eligibility, entitlement and explanation since the start of the case.

On 14th December 2010, the daughter becomes an adult. A case worker, who periodically reviews cases, notices that the case has been inactive for some time and closes the case, recording an end date. The open-ended determination result is replaced with a closed-period determination result.

Throughout the evolving history for the case, each determination shows key decision factors such as the date that the daughter was born, the date that the daughter became an adult, and changes in the household's total income. For any constant period of explanation, there are different categories of explanation, one showing a summary of which eligibility criteria have been met, and another showing how the means test was applied during that period.

Determination Storage

For both active and reactive determinations, the Engine stores a snapshot of a determination.

There are two main groups of database tables used to store the determination data:

- CREOLECaseDetermination and its related tables - stores full details of the determination for later viewing by a case worker; and
- CaseDecision and its related tables - stores details of the eligibility/entitlement result for input into financial processing (typically to make benefit payments to the claimant).

For more details on these database tables and the processing that occurs around the storage of determinations, see 1.8 How Determinations Are Stored on page 147.

Example

When the father's Lone Parent Benefit case was initially activated (i.e. prior to his marriage and other changes of circumstances), the father's entitlement was $20 per week from 1st January 2001, rising to $25 per week from 1st June 2002, with eligibility halting when his daughter reaches the age of majority on 14th December 2010.

This initial assessment determination would be stored as a single "current" record on the CREOLECaseDetermination family of database tables, with these related records on the CaseDecision family of database tables:

- eligible and entitled to $20 per week from 1st January 2001 to 31st May 2002 inclusive;
- eligible and entitled to $25 per week from 1st June 2002 to 13th December 2010 inclusive; and
- ineligible from 13th December 2010 until further notice.

When the evidence on the case and/or payment rates change, the records above are superseded and replaced with a new set of "current" records.

Scheduling Financials

The financial scheduler is responsible for scheduling financial transactions based on the eligibility and entitlement results and the case deductions. These financial schedules, known as financial components, are used by the Financial Manager to create financial instruction line items.
The financial scheduler sits between the Eligibility and Entitlement Engine and the Financial Manager, translating eligibility and entitlement results, as well as case deductions, into financial schedules that can be processed into actual payments or bills.

Financial schedules will only be created from assessment determinations (for more details on the different types of case determinations, see 1.3 Navigating Determinations on page 26).

Example

The case determination calculated earlier shows that from 1st January 2001 to 31st May 2002 inclusive the father is entitled to $20 per week and that from 1st June 2002 to 13th December 2010 inclusive the father is entitled to $25 per week.

Using this information the financial scheduler will produce one financial component to deliver $20 weekly from the 1st January 2001 to 31st May 2002 inclusive and a second financial component to deliver $25 weekly from the 1st June 2002 to 13th December 2010 inclusive.

A few months later, the new case determination created following the budgetary review shows that the father will be entitled to $28 from 1st January 2003.

Using this information the financial scheduler cancels the existing financial components and creates two new ones. The first will continue to deliver $25 weekly until the current rate expires. The second will deliver $28 weekly from the date that the revised rate comes into effect until the daughter reaches the age of majority.

The following year, the new determination created following the father's marriage shows that the father's eligibility stopped three months earlier (on the date of his marriage) and that the entitlement from that date onwards was $0.

Using this information the financial scheduler cancels the existing financial component (for $28 weekly) and creates a payment correction case for the amount overpaid to the father. The financial schedule created within the payment correction case indicates that the father is liable for a once-off liability to allow the agency to recoup the amount he has been overpaid.

Five years later, the new determination created after the father's wife dies shows that the father is still ineligible, due to his income level. Since no eligible case decisions exist no financial schedules will be created.

The daughter becomes an adult and the case is closed. When a case is closed any live financial components on the case are also cancelled. However in this example no live financial components exist so there is nothing for the financial scheduler to do.

The actual number of financial components required to represent these various financial schedules depends on a number of factors including the nominee component assignments, the nominee delivery patterns, the period to which the schedule applies and the case decision objective tags which have been specified.

For more details on scheduling financials, see Scheduling Financials on page 24.

Determination Retrieval

A case worker can view determination snapshots for a product delivery case, including:

- the current assessment determination, if any - i.e. the determination which is the basis of the actual delivery of the product (e.g. the basis for financials for benefit payments);
- historical assessment determinations, each of which was at one time current but has since been superseded;
- snapshots taken when a case was submitted for approval, approved or rejected; and
- the result of requesting a manual determination.

When the case worker views a determination snapshot, then the system will analyze the determination result to find the dates on which one or more of the eligibility/entitlement/explanation changes, and will use these dates to carve up the case lifetime into coverage periods.

Each coverage period is shown with its from and to dates. If the case is open-ended, then the last coverage period will be open-ended. If the case worker chooses to see decision details for a coverage period, then the system will display a "vertical slice" through the determination data for that date.

Example

A determination for the father's case establishes the following:

- the claim is eligible from 1st January 2001 up to 30th June 2005 inclusive
- the father is entitled to:
    - $20 per week from 1st January 2001 to 31st May 2002 inclusive; and
    - $25 per week from 1st June 2002 to 30th June 2005 inclusive;
- the explanation of the case varies:
    - father passes all eligibility criteria from 1st January 2001 to 30th June 2005 inclusive;
    - father fails the "lone" condition from 1st July 2005 to 31st January 2010 inclusive; and
    - father fails the income means test from 1st February 2010 until further notice.

The dates on which items change can be combined to carve up the case's lifetime into these coverage periods (where the eligibility, entitlement and explanation is constant throughout each coverage period):

- 1st January 2001 to 31st May 2002 inclusive;
- 1st June 2002 to 30th June 2005 inclusive;
- 1st July 2005 to 31st January 2010 inclusive; and
- 1st February 2010 until further notice.

1.3 Navigating Determinations

Introduction

The Engine calculates and stores different types of determination results at various points of the case lifecycle. This section describes how a case worker user can navigate the determinations using the screens included with the Engine.

The Engine supports these types of determination:

- Manual eligibility check determinations;
- Snapshot determinations; and
- Assessment determinations.

Once the Engine displays a determination, then the case worker can drill into details of the determination. The different types of details shown in the determination are described in later chapters:

- **Eligibility/entitlement calculation results**

  See How It Looks on page 29;
- **Key decision factors**

  See (deprecated) How It Looks on page 57; and

- **Decision details**

  See How It Looks on page 71.

Manual Check Determinations

The case worker can request a manual eligibility check on a product delivery case and choose whether that check should include "in edit" evidence changes.

If the case worker chooses to base the eligibility check on in-edit evidence, then the Engine provides a determination as if pending evidence changes had already been applied (see Temporary Access to In-Edit Evidence Changes on page 141), i.e. as if:

- newly-added evidence had been activated;
- corrections to existing evidence had been activated; and
- pending-removal evidence had been removed.

If the case worker instead chooses to use active evidence only, then any pending changes to evidence are ignored.

The Engine displays the determination result for the manual eligibility check, noting whether or not in-edit evidence was used. Once a manual eligibility check determination has been created, then the case worker can navigate to the last manual eligibility check created.

Snapshot Determinations

The Engine automatically records a snapshot of a case's determination result when the case is:

- submitted for approval; and
- approved/rejected.

Snapshot determinations are displayed on the Case Calendar and a decision summary can be accessed from the calendar view.

Assessment Determinations

The Engine creates an assessment determination whenever there is an active or reactive request to assess a case. The Current Determination page displays the active assessment determination result. The active assessment determination feeds through to financial processing, typically to dictate the amount payable on a case.

The Determination History displays the list of all assessment determination results, the active assessment determination result and superseded assessment determination results. There is also an option to manually force the reassessment of an active case and view these results.

Current Assessment Determination

When a case worker clicks on the Determinations tab, the Engine displays the current assessment determination, if any. This current assessment determination is the determination which currently governs how the product is being delivered (typically, how much is payable).

If there is no current assessment determination (for example if the case has not yet been activated), then the Engine displays a message explaining that no current assessment determination is available.

Historical Assessment Determinations

Within the Determinations tab, when a case worker clicks on the Determination History option, the Engine displays a list of assessment determinations on the case (if any). The most recent (current) assessment determinations (if any) are displayed at the top and the remainder are superseded determinations (if any) - i.e. determinations which were "current" at some time in the past, but have been replaced due to a change which affected the case's determination result.

Recall that each determination covers the complete case lifetime, and typically includes past periods based on real-world events and predictions based on expected events. Thus each superseded determination contains the predictions about a case that the Engine made based on the known facts at the time; if facts change, then the prediction must be superseded and replaced with a more accurate prediction based on more accurate facts. The current determination represents the best prediction about the case's future eligibility and entitlement based on facts known today.

Manual Reassessments

To support the ability to take a snapshot of a case's determination result at key points in the life cycle of a case, for example when an appeal is created, the case worker can manually request that an active case be reassessed. The case worker must confirm the request to manually reassess a case:

When a case is manually reassessed, the Engine displays the new determination result and notes that it was due to a manual reassessment request.

1.4 Calculating and Displaying Eligibility and Entitlement

Introduction

Eligibility and entitlement results are the core data calculated by the Engine and underpin how the case is treated by financial processing. The Engine contains features to calculate additional "explanation" results (see 1.5 (deprecated) Calculating and Displaying Key Decision Factors on page 56 and 1.6 Calculating and Displaying Decision Details on page 70), but it is critical to understand how the core eligibility and entitlement results are calculated by the system and displayed to the user.

This chapter describes the flow of processing that allows eligibility and entitlement data to be displayed to the user. The processing is intentionally described in reverse-chronological order; firstly we describe the end results, followed by the Engine processing that produces those results, before finally describing how the data was calculated.

This "backwards" perspective will echo how your rules designers will need to think when designing your product; they will need to start with the end in mind (namely how case workers and financial processing will use the eligibility and entitlement results to perform subsequent work on the case).

This chapter is structured as follows:

- **How it looks**

  Describes how eligibility and entitlement information is displayed to a case worker.
- **How it works**

Describes how fixed processing by the Engine and custom processing combine to calculate and display eligibility/entitlement results.

- **How to use it**

  Describes the steps you will need to follow to implement eligibility/entitlement calculations for your product.

How It Looks

This section describes how eligibility and entitlement information is displayed (in summary form) to a case worker.

The structure of eligibility and entitlement data is intentionally restrictive; it contains the information required to allow Cúram Financials to subsequently process the case, such as whether the case can be paid, and if so how much and to whom.

Because the structure of the eligibility and entitlement data is fixed by the Engine, the Engine knows how to display this information for any CER-based case, and so includes standard screens capable of displaying summaries of a case's eligibility and entitlement. (For the display of more flexible data specific to your product, see 1.5 (deprecated) Calculating and Displaying Key Decision Factors on page 56 and 1.6 Calculating and Displaying Decision Details on page 70.)

Eligibility/Entitlement data is also used as input into Cúram Financials; for more details on this use of the data, see 1.8 How Determinations Are Stored on page 147 and 1.9 Scheduling Financials on page 156.

Viewing a Determination's Coverage Periods

The case worker can navigate to an assessment determination for a product delivery case as described in Assessment Determinations on page 27.

A determination covers the full lifetime of the case (as known at the time the determination was made - the case's lifetime may have subsequently changed and a newer determination recorded). The determination is open-ended because the case itself is open-ended, so the latest decision for this determination applies until further notice.

The determination is divided into a number of contiguous "Coverage Periods". Broadly each coverage period is a period of constant eligibility, entitlement and explanation for a determination.

For each coverage period, the system shows the eligibility result for that period, and a summary of the amount payable. This coverage period data is standard eligibility/entitlement information and the Engine ships with a standard screen for displaying this information for any CER-based case. You should not need to write your own coverage period screens unless you have special requirements.

Basic Eligibility/entitlement Decision Details

The Engine also provides the ability to present a case's entitlement information in a very basic way. The details are presented in a somewhat technical way, as a list of objectives and their respective tags.

The Engine does *not* automatically show this information for your product; however it is relatively easy to configure your product to show this information, typically as a stepping stone on the way to fully developing your product.

These "basic" eligibility/entitlement details are a simple implementation of Decision Details rules (see 1.6 Calculating and Displaying Decision Details on page 70); typically these details are too simple to be useful to a case worker, but may be useful to your developers during the development of your product, and so may be used as "scaffolding" during your development cycle. See Re-use the Basic Decision Details before Writing Your Own on page 198 for a description of how to enable these basic details for your product.

How It Works

This section describes how fixed processing by the Engine and product-specific processing combine to calculate and display eligibility/entitlement results.

Note: The key decision factors feature is deprecated. For more information, see Deprecated features.

The calculation and display of a determination involves a mixture of:

- fixed processing contributed by the Engine; and
- custom product-specific processing contributed by you (i.e. the implementation of your product).

The Engine follows the following high-level steps to arrive at eligibility and entitlement data that can be displayed to a case worker:

- At Determination Calculation time:
  - An action occurs which triggers the determination of a case (either an active or reactive determination);
  - The Engine identifies the product periods (configured for the product) that cover the case's lifetime;
  - The Engine uses CER rules (specific to the product) to calculate the eligibility and entitlement for each contributing product period;
  - The Engine calculates the eligibility and entitlement across the lifetime of the case by "splicing together" the eligibility and entitlement from each contributing product period;
  - The Engine calculates the eligibility and entitlement across the lifetime of the case by "splicing together" the eligibility and entitlement from each contributing product period;
  - The Engine stores (on the database) a determination result containing the eligibility and entitlement data (as well as key decision factors and decision details, covered elsewhere in this document).
- At Determination View time:
  - A case worker requests to view a determination on a case;
  - The Engine retrieves the determination result from the database and identifies its coverage periods; and
  - The Engine lists the coverage periods for the case, and uses the decision summary display strategy configured for the product to produce a summary of the entitlement for each coverage period.

The data interfaces and implementation for calculation of eligibility/entitlement, and subsequent display of eligibility/entitlement data are described in more detail below.

Calculation of Eligibility and Entitlement

The responsibilities for calculating a case's eligibility and entitlement are divided between fixed implementations provided by the application and product specific implementations for a product (some of which must adhere to application-provided interfaces). This section describes the important interfaces and implementations involved in the calculation of the eligibility/entitlement results that form part of a determination result.

Sections A, B, and I describe a layer of fixed implementations that are responsible for calculating and storing the overall determination result.

The processing described in sections C, D, and E represents a combination of fixed interfaces and custom product-specific processing that is responsible for determining the eligibility and entitlement for the case across product periods.

Section C describes a fixed interface included with the Engine that serves as the interface between the fixed eligibility/entitlement processing provided by the Engine and the product-specific rules for the calculation of eligibility and entitlement results.

Sections D and E both describe custom product-specific processing.

The final layer described in sections F, G, and H represents a mixture of fixed implementation and fixed interfaces contributed by the engine, as well as custom product-specific processing that is responsible for retrieving the data from entities, evidence and rate tables used by the custom product-specific processing described in section D.

(deprecated) A) `ProductEligibilityEntitlementRuleSet ProductDeliveryCase` rule object

When the calculation of a determination is triggered, a rule object `ProductEligibilityEntitlementRuleSet.ProductDeliveryCase` is created automatically in memory and populates it with case data, including the case ID, start date, expected start date, end date, and expected end date. This processing is critical to the Engine and cannot be customized. The rule object is responsible for calculating the overall determination result (including the calculation of eligibility/entitlement information).

In particular, the rules for the rule object's calculated attributes are fixed and cannot be customized:

- the value of `contributingProductPeriods` is provided by the static method `curam.core.sl.infrastructure.assessment.impl.Statics.contributingProductPeriods` (which is included with the Engine), which returns a timeline of `ProductPeriod` rule objects (see below); and
- the value of `determinationResult` is provided by the static method `curam.core.sl.infrastructure.assessment.impl.Statics.determinationResult` (which is included with the Engine), which returns a `curam.core.sl.infrastructure.assessment.impl.DeterminationResult` Java data object (see below).

B) `ProductEligibilityEntitlementRuleSet.ProductPeriod` rule objects

The Engine also retrieves product period information from the database, including the start date and end date, and creates a `ProductPeriod` rule object in memory for each published product period in the system. For a new product, there is typically only one period covering the lifetime of the product, but as new legislation is introduced, then it is possible (depending on design approach) for the product to have more than one period. For more information on adding product periods over time, see 1.11 Incremental Design and Evolution on page 191.

These product periods allow the Engine to identify which ones will contribute to a determination on a case - namely the periods that overlap in any way with the case's lifetime. Other product periods outside the case's lifetime (i.e. before the case started or after it ended) do not contribute. The structure and maintenance of the `ProductPeriod` rule objects is fixed by the Engine and cannot be customized.

C) `ProductEligibilityEntitlementRuleSet AbstractCase` rule class

The Engine creates an Abstract Case rule object in memory for each Product Period rule object stored in memory. The `AbstractCase` rule class acts as the interface between the fixed eligibility/entitlement processing provided by the Engine, and the product-specific rules for the calculation of eligibility and entitlement results.

This "interface" rule class ensures that concrete sub-rule-classes have implementations for the following calculated rule attributes, which provide the fixed-structure eligibility and entitlement results required by the Engine:

- **isEligibleTimeline**

  Responsible for calculating the intervals of time during which the case is/is not eligible during the product period; and
- **objectiveTimelines**

  Responsible for calculating the objectives that the case may be entitled to, and for each objective, the intervals of time when the case is/is not entitled to the objective, and the tag values available (see Write the Case Eligibility/Entitlement Calculation Rule Classes on page 45 for more details).

The Engine-supplied interface rule set has no implementations for these attributes - rule classes must be written (see below) to provide implementations for the business requirements of the product; rather the interface rule class allows the Engine to communicate with the rule classes by specifying a contract for the data structures that it requires.

D) Custom rule classes for eligibility/entitlement

When the Engine calculates eligibility/entitlement results for a product period, the Engine first asks the product period which rule class should be used (which is recorded on the product period as part of setting up a product).

The rule class specified on the product period must ultimately extend from the `ProductEligibilityEntitlementRuleSet.AbstractCase` interface rule class. For ease of upgrades, it is recommended that the rule class extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultCase` rule class which provides default implementations.

The rule class must provide implementations of the `isEligibleTimeline` and `objectiveTimelines` rule attributes, which is where the bulk of the implementation effort for eligibility/entitlement calculations will lie (and in fact this effort may form the bulk of the overall product implementation). See Write the Case Eligibility/Entitlement Calculation Rule Classes on page 45 for more details on the work involved in implementing these attributes.

If the product has different product periods, due to a change in legislation that affects how eligibility and/or entitlement is calculated, then each product period will be configured to use a different rule class.

E) Custom rule classes for calculations

Typically the calculations of eligibility and entitlement stored in the `isEligibleTimeline` and `objectiveTimelines` rule attributes may involve complex business calculations, and CER rule classes can be created to structure these calculations in line with business requirements.

Any such rule classes do not need to adhere to any application-supplied interfaces (although the CER "extends" mechanism may be used to create a hierarchy of rule classes to treat rule objects in a polymorphic way if useful).

Some of the calculations will involve accessing stored data, by retrieving rule objects of these types:

- rate cells (see F) `RateRuleSet.RateCell` rule objects and propagation configuration on page 33);
- custom entities (see G) Custom Entity rule objects on page 33); and /or
- custom evidence (see H) Custom Evidence rule objects on page 33.

Note: These retrievals cause a dependency to be stored between the case's `determinationResult` and the underlying stored data, which is how the Engine handles reassessment of cases when data changes (see 1.10 Reassessment - Handling Changes in Circumstance on page 170.

F) `RateRuleSet.RateCell` rule objects and propagation configuration

The Engine stores and maintains `RateCell` rule objects on CER's database tables for cells from certain rate tables. The data from the rate table cells is propagated to these rule objects when modified and is stored in a `valueTimeline` rule attribute that stores the value of the rate over time. The list of rate tables that are propagated to rule objects is configurable (see Configuration on page 105), but otherwise the processing is supplied by the Engine and cannot be customized.

Typically the eligibility and/or entitlement calculations for a product will involve comparisons with or multiplications by rates (in addition to case-specific data); and rules for calculations will access such rates via `RateCell` rule objects.

G) Custom Entity rule objects

When the calculation of a determination is triggered, the Engine queries rule object data configurations to determine what entities contain data that is required by CER for rules processing and supports the population of CER rule objects in memory for each of the relevant entities. The Engine contains a generic mechanism for populating the rule objects with the data from these custom entities (see Entity Rule Objects on page 107).

H) Custom Evidence rule objects

When the calculation of a determination is triggered, the Engine queries rule object data configurations to determine what custom evidence types (both static and dynamic evidence) contain data that is required by CER for rules processing and supports the population of CER rule objects in memory for each of the relevant evidence types. The Engine contains a generic mechanism for populating the rule objects with the data from these custom evidence types (see

Active Succession Set Rule Objects on page 115 and Active Evidence Row Rule Objects on page 130).

Each rule class for a custom evidence type must ultimately extend the Engine-supplied `PropagatorRuleSet.ActiveSuccessionSet` or `PropagatorRuleSet.ActiveEvidenceRow` rule classes.

The population of CER rule objects for dynamic evidence types is handled automatically (see the section on "Generated Rule Sets" in the *Cúram Dynamic Evidence Configuration Guide*).

I) `DeterminationResult`

The ultimate determination output from the `ProductEligibilityEntitlementRuleSet.ProductDeliveryCase.determinationResult` attribute is an immutable instance of the `curam.core.sl.infrastructure.assessment.impl.DeterminationResult` interface.

This object holds data only (the methods on the interface are accessors only). The determination result is created by the Engine when splicing together the individual eligibility and entitlement results across the case's division into different product periods and is stored on the database. Note that in the simple case (typically for recently-created products) where the product has only one product period, the entire eligibility and entitlement results are exactly those contributed by the single product period which covers the entire case lifetime.

The `DeterminationResult` data holds the following:

- **productDeliveryID**

  The unique identifier of the product delivery case;
- **determinationDateRange**

  The "lifetime" of the determination, which was the lifetime of the case at the time the determination was taken;
- **determinationEligibilityEntitlementTimeline**

  The varying eligibility/entitlement data for the case, spliced together from the `isEligibleTimeline` and `objectiveTimelines` values from the rule objects created for each of the contributing product periods; and
- Other data not directly relevant to eligibility and entitlement (and so covered elsewhere in this document - see (deprecated) E) `DeterminationResult` on page 60 and G) `DeterminationResult` on page 76).

The `DeterminationResult` structure is provided by the Engine and cannot be customized in any way.

Display of Eligibility and Entitlement

The vast majority of data processing for eligibility and entitlement occurs at the time when the determination result was calculated, as described in the previous section.

However, some data processing for eligibility and entitlement occurs at the time that the data is viewed; and this section describes the display-time processing of eligibility and entitlement data.

When a case worker views a determination then the Engine automatically:

- divides the determination into coverage periods; and
- displays a summary of entitlement for each coverage period.

Dividing the Determination into Coverage Periods

When a case worker views a determination, the Engine automatically divides up the case lifetime into a number of coverage periods.

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

Each coverage period is a period within the determination where the following are constant:

- The eligibility (yes/no);
- The entitlement (which objectives, their targets and references, and their tag values); and
- The decision details data (see 1.6 Calculating and Displaying Decision Details on page 70).

Or, to put it another way, any change in eligibility, entitlement and/or decision details along the lifetime of the determination causes a new coverage period to come into effect on that date.

> **Note:** A change in a key decision factor on a particular date does not in itself cause a new coverage period to start on that date.
>
> Typically, though, the fact that there is a change to a key decision factor on that date tends to mean that the eligibility/entitlement and/or decision details are likely to change on that date, too (and the change in eligibility/entitlement and/or decision details will cause a new coverage period to start on that date).

The division of the determination into contiguous coverage periods occurs automatically on the Engine's standard screens for viewing a determination, and cannot be customized.

Displaying a Summary of Entitlement for a Coverage Period

The Engine includes an interface for providing a summary description of the entitlement for a coverage period: `curam.core.sl.infrastructure.assessment.impl.DeterminationIntervalSummarizerStrate` interface. See the JavaDoc for this interface for more details.

When you set up your product, you may specify an implementation of this interface that the Engine will use when displaying a coverage period. The Engine invokes this strategy implementation whenever it displays a coverage period for a determination, and the strategy implementation is responsible for returning an appropriate summary line of text (in the user's locale).

Note that:

- if you do not specify a strategy implementation for your product, then no summary for the coverage period will be displayed; and
- if you subsequently change the strategy implementation for your product, then only the display output shown to users is affected; the system will not reassess cases nor cause any changes in financial output.

As such, specifying a strategy implementation for entitlement summaries is not critical to getting your product up and running; this task can be deferred until later in the development cycle if need be.

How to Use It

Most of the high-level processing for eligibility and entitlement is fixed logic provided by the Engine. However, you will have to provide implementations for certain lower-level logic. In order to do this, you must understand the basic concepts of eligibility and entitlement.

In addition to providing an understanding of these concepts, this section describes the work you will need to do to complete the eligibility and entitlement logic for your product, as follows:

- Analysis;
- Implementation; and
- Testing.

Note: This section describes the complete work for eligibility/entitlement logic; however, for short-cuts you can take to get your product up-and-running quickly, see Incremental Design on page 194.

Understanding Eligibility and Entitlement Concepts

The Engine has a fixed data structure for eligibility and entitlement data. Understanding the structure of this data is critical as you must map your business requirements to the concepts in this structure.

Case Lifetime

The lifetime of a case is the period of time between the case's start and end dates, inclusive.

Each case may bear an actual or expected start date; if an actual start date is present then it is used as the start of the case, other- wise the expected start date will be used.

Similarly, a case may have an actual end date and/or an expected date, which governs the end date of the case's lifetime (with the actual date taking precedence over any expected date).

A case without an end date is known as an "open-ended" case. A determination for an open-ended case will thus be an open-ended determination; and the final coverage period in that determination will be open-ended too.

Eligibility

On any given day in the case's lifetime, the case is either *eligible* or *ineligible* for delivery (depending on the circumstances of the case).

The Engine thus treats the case's eligibility as a Boolean value which can vary over the lifetime of the case (and thus in CER terms is a Timeline of Boolean values).

For example, a claim for child benefit may be eligible today but will cease to be eligible once the child being claimed for reaches the age of majority.

Objectives

An objective is something "delivered" by a product delivery case to a particular target. Commonly an objective is a payment of some kind of benefit or allowance paid to a client, but an objective could also be (for example) an amount to bill, or a non-monetary outcome such as a recommendation to the client.

The Engine has separate concepts for:

- the *types* of objective supported by your product, e.g. your product supports the concept of a personal benefit allowance; each type of objective defines its name and other fixed data such as the type of client that it targets; vs.
- an *instance* of an objective on a particular product delivery case, e.g. on case 123 for your product, claimant John Smith is entitled to personal benefit allowance from 1st Jan 2011 until 15th Feb 2011; each objective instance describes the particular target to deliver to and the period(s) for which that target is entitled to the objective.

On any given day in the case's lifetime, if the case is eligible then the case is either *entitled* or *not entitled* to any particular objective instance. Equivalently, we say that an objective instance was *attained* or *not attained* on that day.

The rule classes for objective types and instances described later in this section show the full details available.

Objective Tags

An objective tag is a frequency at which an objective may be delivered. Commonly for payment objectives, the objective will support a mixture of delivery frequencies, e.g. daily and weekly, to support:

- Different frequencies of payment offered to the client; and
- Payments for ramp-up and ramp-down periods. These are described more in Calculating Financial Component Cover Periods on page 158

The Engine has separate concepts for:

- the *types* of tag supported by a particular type of objective, e.g. the personal benefit allowance (supported by your product) may be paid either weekly or daily; vs.
- an *instance* of a tag on a particular objective instance, e.g. on case 123 for your product, claimant John Smith can be paid $10 per week, when receiving the personal benefit allowance to which he is entitled.

The rule classes for objective tag types and instances described later in this section show the full details available.

Analysis

You must understand the requirements for your product, and analyze how these requirements broadly map to the Engine's eligibility/objective/tag concepts before starting implementation.

Typically, the requirements for eligibility/entitlement are enshrined in legislation, and so the task of implementing the requirements is, to a certain extent, a translation exercise (albeit a complex one). We note this here as the tasks described later in this document (for key decision factors and decision details) differ in nature.

The following steps should aid your analysis.

Identify the product periods for your product

You must understand whether there are any significant changes in legislation already in place for your product, and whether you will implement these changes using multiple product periods.

Typically, for new products there is (to date) only a single version of the legislation, and so usually a new product will initially have only a single product period set up. Subsequent changes in legislation may occur once your product matures, of course.

See Handling Legislation Change on page 200 for how to decide whether your product should have multiple product periods for legislation changes, or rules that branch based on legislation change. This will help you analyze how many periods your product is split up into.

Each product period will typically have its own special rule class for eligibility and entitlement calculations.

For a new benefit product created via the dynamic product wizard a default product period is automatically inserted and has a default eligibility and entitlement rule set associated with it. Although the default product period is published upon creation of the benefit product, the default eligibility and entitlement rule set is left in an in-edit state following completion of the dynamic product wizard. The rule set appears in the list of rule sets available for publication on the Cúram Express Rule Sets page of the Administration Workspace and is a generic rule set which is not suitable for product use prior to update. The rule set should be edited within the Cúram Express Rules editor to meet product requirements prior to use of the newly created product. See the section "Configuring the Product in the Administration Workspace" in the *How to Build a Product Guide* for additional information about new products created using the dynamic product wizard.

Identify what types of objective are delivered by your product, and at what frequencies

For each product period, you must identify the types of objectives that your product supports.

Identify what it is that your product "delivers". A common type of delivery is that of a benefit payment to a person or a household. There may be more than one type of thing delivered by your product, for example your product may deliver two different financial components (aimed at different types of nominees - for example a personal allowance and a child allowance), and also a recommendation.

Once you have identified what it is that your product delivers, you need to map these deliverable things to types of objectives. You may have a design decision to make regarding whether to have a smaller number of complex objective types, or a larger number of simple objective types.

To answer this you may need to consider such things as whether each case needs to have the flexibility to pay certain types of benefits to one nominee and other types of benefit to different types of nominee:

- if so, you may well need to have separate types of objectives for the different types of benefit that your product delivers; however
- if benefit payments are always to a single nominee, then your choice of whether to implement separate objective types may rest on whether the total benefit paid is a simple aggregation of all objectives achieved, or whether there are complex rules to come up with the total amount payable (if the latter, then one complex objective type is likely to be more suitable).

You may find it useful to come up with a trial design of objectives, and walk through different business scenarios and how they are treated by the Engine and financial processing, to ensure that your business requirements mapping to objective types allows your business requirements for your product to be met.

Once you are happy with your types of objectives, then for each type of objective, identify at what frequency it can be delivered. Typically a daily rate is always required (to handle ramp up and ramp down periods) but you can implement longer periods such as weekly or monthly too. These longer periods can be useful if your daily rate represents a rounded-up fraction of a weekly or monthly rate, or if your eligibility results are always exactly some larger unit, e.g. if you have business requirements that say each case is either entitled to an objective for an entire month or not at all.

At this point, you have identified the types of things delivered by your product in general; now you need to turn your attention to how the system will decide HOW to deliver a particular product delivery CASE.

Identify the rules that govern when a case is eligible

You must analyze the rules for how the system should determine a case's eligibility. The structure of these rules can vary very much from product to product.

It is common for eligibility rules to center around the claimant or household falling into a particular category, where each category is determined by the claimant or household meeting a number of conditions. The categories and the rules for each typically form the "highest level" of rules for eligibility.

For some products, the case's overall eligibility can be determined without reference to the entitlement to any objectives; for other products the eligibility and entitlement calculations are much more intertwined (e.g. for some products, you may only want the case to be deemed eligible if one or more objectives are attained). You will need to visit your business requirements to understand the interplay (if any) between eligibility and entitlement calculations for your product.

The eligibility result calculated by the Engine is a Timeline of Boolean results, and thus allows for the eligibility to vary over time. For some simple products, the eligibility on any given day is purely determined by the case's circumstances on that day; however, for some more complex products, the eligibility on one day may be influenced by events that have occurred on other days (e.g. an event that occurred in on a different day in the same month). You should take care to understand your business requirements for applying events that occur on one day to an overall eligibility result for different days.

For more information on CER's support for Timelines, (see "4.5 Handling Data that Changes Over Time" in the *Cúram Express Rules Reference Manual*)

Identify the rules that govern the objectives for each case

You must analyze which objective instances are required for each case. Typically the creation of objectives falls into one of these patterns:

- **Single Objective**

  For a given objective type, there is always exactly one objective instance for the case; and/or
- **Multiple Objective**

  For a given objective type, there is one objective instance for each member of the case who meets certain criteria.

For example, a product aimed at a household with children might always deliver exactly one "basic household allowance" objective to the overall household, but also deliver a "child allowance" objective to each child. Thus a household with 3 children would be have 4 objectives created - one for the household and one for each child. A household with no children would have only a "basic household allowance" objective created.

Note that the Engine does not impose these single/multiple objective patterns; the creation of objectives can be as complex as your business requirements dictate.

Identify the rules that determine when an objective has been attained and its target

For each objective instance that will be created for a case, you must analyze the rules for deciding at which points in the case lifetime the case is entitled to the objective.

Whether or not a particular objective has been attained can vary over time - e.g. an allowance payable whenever a parent is absent from the household will be attained when a parent leaves and will cease to be attained once the parent returns; if the parent leaves and returns many times over the case's lifetime, then the case's entitlement to the objective will similarly vary many times.

For some simple products, the case is always entitled to an objective by dint of the case being eligible for that period of time, and thus there is no real entitlement calculation required. For other more complex products, the rules for entitlement to a given objective can rival those for overall case eligibility in terms of their complexity. It can be useful to have an understanding of the relative complexity of your overall case eligibility rules vs. the entitlement rules for your various objective types.

You must also analyze the requirements for the target of each objective instance, and any useful related reference information. The Engine allows the values these to vary over time, but for typical products the values are constant.

Identify the rules that determine the values at which an objective can be delivered

For each objective, you must analyze the value(s) to be used when it is delivered (i.e. during periods when the overall case is eligible, and also that the case is entitled to the objective).

Earlier you analyzed each type of objective to understand the frequencies at which instances may be delivered; now for each these frequencies you must analyze how the Engine will calculate value of an objective instance at that frequency.

For some simple products, the value may be a fixed rate (possibly retrieved from a rate table); for other more complex products, the value may be a complex calculation involving the case's circumstances.

It is common to see a pattern whereby the value for one frequency provided by a calculation, and then values for other frequencies based on it. For example, on one product there may be a complex calculation to provide a daily rate for an entitlement, but the calculation for the weekly rate is simple 7 x <daily rate>; or on another product, there may be a complex calculation for a monthly rate, and then the daily rate is calculated as 12 x <monthly rate> / 365, rounded up to the nearest cent (in favor of the claimant).

Note that the Engine does not enforce these patterns; the calculation of each frequency's value can be as complex as business requirements dictate.

Implementation

Having analyzed your business requirements, you are now in a position to start the implementation of your eligibility and entitlement calculations for your product. A default eligibility and entitlement rule set is automatically created for benefit products by the dynamic product wizard, and this rule set should be edited in line with the guidelines below to suit your product needs. The default name of this default eligibility and entitlement rule set is `*ProductNameWithBlankSpacesRemoved*` `DefaultEligibilityEntitlementRuleSet`, but it is possible to override the name of this rule set on the Eligibility Determination page of the dynamic product wizard. It is also possible to create a new eligibility and entitlement rule set for your product, which should follow the guidelines below.

If your product is complex, then consider sketching out the implementation rule classes on paper or in an object-oriented modelling tool prior to creating CER rule sets. You can then use the sketch to validate your rule classes against various business scenarios.

The rule classes that you create will end up interrelating so it will be useful to keep these interrelationships in mind as you implement your rule classes. The sections below this provide a step-by-step path to implement your eligibility/entitlement calculations.

Write the Product Structure Rule Classes

For each product period, you must create a rule class which is responsible for describing the structure of your product, i.e. which types of objectives it supports. Typically the structure of your product will be identical across product periods and you need only create one set of rule classes which will be used on each product period.

For a new benefit product created via the dynamic product wizard, a product structure rule class will be included in the default eligibility and entitlement rule set that is associated with the default product period created for the product at the completion of the dynamic product wizard. This default product structure rule class should be edited to meet your product requirements prior to product use.

You will require the following rule classes:

- one Product rule class for your overall Product;
- one or more Objective Type rule classes, one for each type of objective supported by your product; and
- one or more Objective Tag Type rule classes, one for each frequency supported for each of the types of objective supported by your product.

You will also relate these rule classes to each other by implementing rules that return instances of your rule classes; thus the product rule class has an attribute for identifying its types of objectives, and each objective type has an attribute for identifying its supported tags. The default eligibility and entitlement rule set created for the benefit product by the dynamic product wizard will already contain each of these rule classes, but they should be edited to meet your product requirements prior to product use.

The following sections describe the creation of these rule classes and their interrelations in detail.

Write the Product rule class

Your rule class to describe your product's structure must ultimately extend from the `ProductEligibilityEntitlementRuleSet.AbstractProduct` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultProduct` rule class which provides default implementations. The default eligibility and entitlement rule set that is automatically created for benefit products by the dynamic product wizard will contain a Product rule class which extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultProduct` rule class.

Here is a description of the attributes inherited from `AbstractProduct`:

Table 3: Rule attributes inherited from `ProductEligibilityEntitlementRuleSet. AbstractProduct`

Rule Attribute name	Data type	Description
`objectiveTypes`	List of `AbstractObjectiveType`	The types of objective supported by this product.

To write the Product rule class, create a rule class which extends `DefaultProductEligibilityEntitlementRuleSet.DefaultProduct`. The rule class should be named in line with your product, e.g. `*ProductName* Product` (the Engine does not have any technical constraint on the rule class name - rather a good name for your rule class may make it easier to develop and maintain your rule sets). The default name of the Product rule class within the default eligibility and entitlement rule set that is automatically created for a benefit product by the product wizard is `*ProductNameWithBlankSpacesRemoved* Product`.

The inherited implementation of `objectiveTypes` returns an empty list; leave this implementation for now and you will return to it once you have created your objective type and tag rule classes.

Write the Objective Type rule classes

For each type of objective supported by your product (for example, a Personal Benefit Allowance or a Child Benefit Allowance), you must create a rule class which must ultimately extend from the `ProductEligibilityEntitlementRuleSet.AbstractObjectiveType` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultObjectiveType` rule class which provides default implementations. The default eligibility and entitlement rule set created for benefit products by the dynamic product wizard will contain two Objective Type rule classes each of which extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultObjectiveType` rule class.

Here is a description of the attributes inherited from `AbstractObjectiveType`:

Table 4: Rule attributes inherited from *`ProductEligibilityEntitlementRuleSet.AbstractObjectiveType`*

Rule Attribute name	Data type	Description
`objectiveTypeID`	String	Identifier of the object type, which must be unique within the product. The length of this identifier must be no more than the number of characters dictated by the RULES_OBJECTIVE_ID domain (which by default is 16 characters).
`name`	Code from the `RulesComponentType` code table	The code for the display name of this objective type.
`financialComponentType`	Code from the `RulesComponentFCType` code table	The financial component type associated with this objective, CT1 if this is a benefit, CT2 if it is a liability.
`rateTarget`	Code from the `RulesComponentTarget` code table	The target for this objective, Client, Product Provide, Service Supplier, Employer, etc..
`tagTypes`	List of `AbstractTagType`	The tag types available for this objective type, indicating the frequencies at which this objective can be delivered
`description`	Localizable message	A description of this type of objective
`comments`	Localizable message	Comments describing this type of objective
`isDeductionAllowable`	Boolean	Whether case workers are allowed to select objective instances of this type when creating case deductions.

For each objective type supported by your product, create a rule class which extends `DefaultProductEligibilityEntitlementRuleSet.DefaultObjectiveType`. The default eligibility and entitlement rule set that is automatically created for benefit products by the dynamic product wizard will already contain two rule classes which extend the `DefaultProductEligibilityEntitlementRuleSet.DefaultObjectiveType` rule class. These two default rule classes are named `PersonalBenefitAllowanceObjectiveType` and `ChildBenefitAllowanceObjectiveType`. If you are writing a new rule class, the name of your rule class should be in line with the name of your objective type (*`ObjectiveTypeName`* `ObjectiveType`), to ease development and maintenance, e.g. `PersonalBenefitAllowanceObjectiveType` or `ChildBenefitAllowanceObjectiveType`.

The inherited implementation of `tagTypes` returns an empty list; leave this implementation for now and you will return to it once you have created your objective tag rule classes.

For all other inherited rule attributes, use your analysis of your business requirements to implement rules to return values appropriate to your objective type. The default eligibility and entitlement rule set should be edited to reference values suitable to your product. If you are writing a new eligibility and entitlement rule set rather than using the default rule set created by the product wizard, typically you will add a new value to the `RulesComponentType` code table to implement the name for your objective type, but you will use one of the values provided from the `RulesComponentFCType` and `RulesComponentTarget` code tables when implementing the rules for `financialComponentType` and `rateTarget` respectively.

Write the Objective Tag Type rule classes

For each of your product's supported objectives (e.g. a Personal Benefit Allowance), your analysis determined frequencies at which that objective may be delivered (e.g. a Personal Benefit Allowance might be payable either daily or weekly, at the choice of the claimant or to handle ramp-up/ramp-down periods).

For each frequency for an objective type, you must create a rule class which must ultimately extend from the `ProductEligibilityEntitlementRuleSet.AbstractTagType` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultTagType` rule class which provides default implementations. The default eligibility and entitlement rule set created for benefit products by the dynamic product wizard will already contain three objective tag type rule classes each of which extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultTagType` rule class.

Here is a description of the attributes inherited from `AbstractTagType`:

Table 5: Rule attributes inherited from `ProductEligibilityEntitlementRuleSet.AbstractTagType`

Rule Attribute name	Data type	Description
`tagTypeID`	Long	Identifier of the tag type, which must be unique within the product.
`name`	Localizable message	The display name of this tag type.
`pattern`	Frequency Pattern	The frequency at which this tag type is delivered. For more information, see the JavaDoc for `curam.util.type.FrequencyPattern`.

Rule Attribute name	Data type	Description
`valueType`	Code from the `RulesTagType` code table	The type of value held in instances of this tag type.
`description`	Localizable message	A description of this type of objective tag.

The three objective tags defined in the default eligibility and entitlement rule set created for benefit products by the dynamic product wizard are daily, weekly and monthly tags. If you are extending this default rule set to include additional objective tags, or if you are creating a new eligibility and entitlement rule set, for each objective tag supported by your product create a rule class which extends `DefaultProductEligibilityEntitlementRuleSet.DefaultTagType`. Again the name of your rule class should be in line with the name of your objective tag type (`*TagTypeName* TagType`), to ease development and maintenance, e.g. `PersonalBenefitAllowanceWeeklyDeliveryTagType` or `PersonalBenefitAllowanceDailyDeliveryTagType`.

For all the inherited rule attributes, use your analysis of your business requirements to implement rules to return values appropriate to your objective tag type. Typically you will use one of the values provided from the `RulesTagType` code table when implementing the rules for `valueType`.

Relate each Objective Type to its supported Objective Tag Types

Now that you have created rule classes for your objective types and objective tag types, you must relate each objective type to its list of supported objective tag types.

For each objective type rule class that you created, you must now implement its `tagTypes` attribute to create a list of instances of the rule classes which represent its tag types. Typically this list is a `<fixedlist>` where each member in the list is a simple `<create>` expression.

For example, for the `PersonalBenefitAllowanceObjectiveType`, its implementation of `tagTypes`, in pseudo-code, would be:

- Create a list of `AbstractTagType`, with members:
  - Create an instance of `PersonalBenefitAllowanceWeeklyDeliveryTagType`; and
  - Create an instance of `PersonalBenefitAllowanceDailyDeliveryTagType`.

Relate the Product to its supported Objective Types

Now that you have created rule classes for your product and objective types, you must relate the product to its list of supported objective tag types. This association will already be in place if you are using the default eligibility and entitlement rule set that is automatically created for benefit products by the dynamic product wizard.

For your product rule class, you must now implement its `objectiveTypes` attribute to create a list of instances of the rule classes which represent its objective types. Typically this list is a `<fixedlist>` where each member in the list is a simple `<create>` expression.

For example, for a product which supports Personal Benefit Allowance and Child Benefit Allowance objective types, its implementation of `objectiveTypes`, in pseudo-code, would be:

- Create a list of `AbstractObjectiveType`, with members:
  - Create an instance of `PersonalBenefitAllowanceObjectiveType`; and

- Create an instance of `ChildBenefitAllowanceObjectiveType`.

Write the Case Eligibility/Entitlement Calculation Rule Classes

Now that you have implemented rule classes to describe the structure of your product, you can start to implement the rule classes which calculate the eligibility and entitlement results for a given product delivery case.

For each product period that you are creating, you must create a rule class which is responsible for calculating the eligibility and entitlement for a case, i.e. which objectives must be created on the case and their entitlement.

Typically the eligibility and entitlement calculations for your product will be different across product periods (because it was these differences which led you to identify multiple product periods in the first place). However, it is likely that large swathes of the eligibility/entitlement logic are identical between product periods and you should consider factoring your rule classes so that common logic is implemented and maintained in a single place, just as you would do in most kinds of development work.

In a development environment, initially it is recommended that you do just enough implementation work for eligibility and entitlement calculations so that you can view your product in a development database and assure yourself that you can create cases against it and see early eligibility/entitlement results.

You can then return to the non-trivial task of fleshing out your eligibility/entitlement calculation rules to provide the full logic required by your business requirements.

You will require the following rule classes:

- one Case rule class responsible for the overall eligibility/entitlement calculations for a particular case;
- for each Objective Type supported by your product, one Objective Instance rule class; and
- for each Objective Tag Type supported by each of your Objective Types, one Objective Tag Instance rule class.

The default eligibility and entitlement rule set created for the benefit product by the dynamic product wizard will already contain each of these rule classes, but they should be edited to meet your product requirements prior to product use.

Write the Case rule class

Your rule class to calculate a case's eligibility/entitlement must ultimately extend from the `ProductEligibilityEntitlementRuleSet.AbstractCase` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultCase` rule class which provides default implementations. The default eligibility and entitlement rule set that is automatically created for benefit products by the dynamic product wizard will already contain a rule class which extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultCase` rule class.

Here is a description of the attributes inherited from `AbstractCase`:

Table 6: Rule attributes inherited from `ProductEligibilityEntitlementRuleSet.AbstractCase`

Rule Attribute name	Data type	Description
`productDeliveryCase`	`ProductDeliveryCase`	The controlling rule object which is responsible for splicing together the determination result from the contributions made by the product period. Passed in when the instance of `AbstractCase` is created.
`isEligibleTimeline`	Timeline of Boolean	The varying overall eligibility of the case.
`objectiveTimelines`	List of `AbstractObjectiveTimeline`	The objectives created for this case.

Create a rule class which extends `DefaultProductEligibilityEntitlementRuleSet.DefaultCase`. The default eligibility and entitlement rule set that is automatically created for benefit products by the dynamic product wizard will name the rule class in line with the newly created product. The default name of the rule class is *`ProductNameWithBlankSpacesRemoved`* `Case`. If you are creating a new eligibility and entitlement rule set, the rule class should be named in line with your product, e.g. *`ProductName`* `Case` (the Engine does not have any technical constraint on the rule class name - rather a good name for your rule class may make it easier to develop and maintain your rule sets).

The value of `productDeliveryCase` will be automatically set to be the Engine's rule object for the overall case. This rule object can be used to access information about the case, such as its unique identifier.

You must provide an implementation for `isEligibleTimeline`. The `isEligibleTimeline` is created by default within the default eligibility and entitlement rule set that is automatically created for benefit products by the dynamic product wizard, but should be edited prior to use in line with your product requirements. Typically this implementation will be non-trivial (and in fact its implementation may well be the bulk of the effort in the development of your product), as it is responsible for ultimately calculating your case's overall eligibility during the product period, and its full implementation will need to access data about the case (e.g. using rule objects for evidence recorded on the case) and/or product-wide data such as rates. The implementation may involve the creation of many "calculator" rule classes which provide interim calculated results required to perform the complex calculations dictated by your business requirements for case eligibility on your product.

In a development environment, you may choose to initially implement a very simple cut-down version of your eligibility rules, e.g. that the case is eligible whenever the claimant fits into one of a number of very broad categories, or (even more simply) that the case is always eligible. Once you have completed the skeleton implementation of your eligibility rules and checked that cases can be created against it, then you can return to the much-longer development task of implementation your full eligibility rules, which will involve the creation of other rule classes, e.g. which map to your custom evidence types, or which provide intermediate calculation results.

The inherited implementation of `objectiveTimelines` returns an empty list (i.e. your case never has any objectives at all); leave this implementation for now and you will return to it once you have created your objective instance and tag instance classes.

Write the Objective Instance rule classes

Each objective rule object created for your case will be an `AbstractObjectiveTimeline`, which is responsible for calculating the periods when the case is entitled to the objective.

For each type of objective supported by your product (for example, a Personal Benefit Allowance or a Child Benefit Allowance), you must create a rule class which must ultimately extend from the `ProductEligibilityEntitlementRuleSet.AbstractObjectiveTimeline` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultObjectiveTimeline` rule class which provides default implementations. This rule class will already have been created for each type of objective within the default eligibility and entitlement rule set that is automatically created for benefit products by the dynamic product wizard, but they should be edited in line with your product requirements prior to product use.

Here is a description of the attributes inherited from `AbstractObjectiveTimeline`:

Table 7: Rule attributes inherited from *`ProductEligibilityEntitlementRuleSet.AbstractObjectiveTimeline`*

Rule Attribute name	Data type	Description
`objectiveType`	`AbstractObjectiveType`	The type of this objective timeline.
`isEntitledTimeline`	Timeline of Boolean	The varying entitlement to this objective. The value of this timeline is only taken into account during periods when the case is eligible.
`targetIDTimeline`	Timeline of Long	The varying ID of the target participant (e.g. Person, Service Supplier, Product Provider, Employer, etc.) which is targeted by this objective.
`tagTimelines`	List of `AbstractTagTimeline`	The frequencies at which this objective can be delivered.
`relatedReferenceTimeline`	Timeline of String	The varying reference to additional business-specific information relating to this objective. The `relatedReference` attribute can be used to store information that will help distinguish the difference between one instance of a rules objective and another, which may be important for financial processing to explain the breakdown of a payment. The length of this identifier must be no more than the number of characters dictated by the RELATED_REFERENCE_TEXT domain (which by default is 4000 characters).
`description`	Localizable message	A description of this objective instance

For each objective type supported by your product, create a rule class which extends `DefaultProductEligibilityEntitlementRuleSet.DefaultObjectiveTimeline`. The default eligibility and entitlement rule set that is automatically created for benefit products by the dynamic product wizard will already contain two rule classes which extend the `DefaultProductEligibilityEntitlementRuleSet.DefaultObjectiveTimeline` rule class. These two default rule classes are named `PersonalBenefitAllowanceObjectiveTimeline` and `ChildBenefitAllowanceObjectiveTimeline`. If you are writing a new rule class, again the name of your rule class should be in line with the name of your objective type (*`ObjectiveTypeName`* `ObjectiveTimeline`), to ease development and maintenance, e.g. `PersonalBenefitAllowanceObjectiveTimeline` or `ChildBenefitAllowanceObjectiveTimeline`.

The inherited implementation of `tagTimelines` returns an empty list (i.e. your objective has no supported frequencies at which it can be delivered); leave this implementation for now and you will return to it once you have created your objective tag instance rule classes.

Implement the other rule attributes on the rule class. Typically the bulk of the work is in the implementation of `isEntitledTimeline`, the complexity of which depends on your requirements. For some types of objective, the objective is always attained whenever the case is eligible; for other types of objective, the case is only entitled to the objective if the circumstances of the case allow it.

Some types of objective will require additional context in order to calculate their entitlement. For any additional context required, create extra rule attributes to hold the context. Later when you create instances of your objective rule class, the `<create>` expressions will need to pass in the values for these "context" rule attributes.

Tip: In particular, multiple objectives (i.e. multiple instances of your rule class) will need some sort of context to distinguish them, e.g. the person to which the multiple objective relates.

Write the Objective Tag Instance rule classes

Each objective rule object created for your case needs to list its supported tags (as instances of `AbstractTagTimeline` which are responsible for calculating the values at which the objective can be delivered).

For each type of tag supported by each type of objective supported by your product (for example, a weekly delivery of a Personal Benefit Allowance or a daily delivery of a Child Benefit Allowance), you must create a rule class which must ultimately extend from the `ProductEligibilityEntitlementRuleSet.AbstractTagTimeline` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultTagTimeline` rule class which provides default implementations. The default eligibility and entitlement rule set created for benefit products by the dynamic product wizard will already contain four such rule classes each of which extends the `DefaultProductEligibilityEntitlementRuleSet.DefaultTagTimeline` rule class. These default rule classes should be edited or added to in line with your product requirements prior to product use.

Here is a description of the attributes inherited from `AbstractTagTimeline`:

Table 8: Rule attributes inherited from `ProductEligibilityEntitlementRuleSet.AbstractTagTimeline`

Rule Attribute name	Data type	Description
`tagType`	`AbstractTagType`	The type of this tag timeline.
`valueTimeline`	Timeline of Object	The varying value of this tag timeline. When converted to a String, the length of this value must be no more than the number of characters dictated by the RULES_OBJECT_TAG_VALUE domain (which by default is 1024 characters).

For each tag type supported by your product, create a rule class which extends `DefaultProductEligibilityEntitlementRuleSet. DefaultTagTimeline`. Again the name of your rule class should be in line with the name of your tag type (*`TagTypeName`* `TagTimeline`), to ease development and maintenance, e.g. `PersonalBenefitAllowanceWeeklyDeliveryTagTimeline` or `ChildBenefitAllowanceDailyDeliveryTagTimeline`. The four default tag timeline rule classes that are defined in the default eligibility and entitlement rule set created for benefit products by the dynamic product wizard are `PersonalBenefitAllowanceDailyDeliveryTagTimeline`, `PersonalBenefitAllowanceWeeklyDeliveryTagTimeline`, `PersonalBenefitAllowanceMonthlyDeliveryTagTimeline` and `ChildBenefitAllowanceDailyDeliveryTagTimeline`.

The inherited implementation of `tagTimelines` returns an empty list; leave this implementation for now and you will return to it once you have created your objective tag instance rule classes.

Implement the `tagType` rule attribute, typically to just `<create>` an instance of the appropriate tag type rule class (see Write the Product Structure Rule Classes on page 41).

Implement the `valueTimeline` rule attribute to calculate the varying value of the tag. The complexity of the implementation will hinge on the complexity of your requirements; some objectives have fixed value tags (e.g. the payment value on any attained objective is identical across cases - the payment amount does not take into account any circumstances on the case), whereas for other objectives, the amount to pay for an attained objective varies according to the circumstances of the case (e.g. reductions in payment amounts due to means tests). It is also possible for the implementation of a tag's value for one frequency to lean on the calculation for a related tag for a different frequency (see the example in Identify the rules that determine the values at which an objective can be delivered on page 40).

The implementation of `valueTimeline` may require that extra context is passed in when the tag timeline is created. If required, create additional rule attributes to hold this context. Later when you create instances of your tag rule class, the `<create>` expressions will need to pass in the values for these "context" rule attributes.

Create tag instances from your objective rule classes

Now that you have created rule classes for your objective instances and tag instances, you must implement how each objective instance will create its tag instances.

The default implementation of `tagTimelines` inherited from `DefaultProductEligibilityEntitlementRuleSet.DefaultObjectiveTimeline` returns an empty list - i.e. no tags are supported for the objective.

For each objective instance rule class that you created, you must now override and implement its inherited `tagTimelines` attribute to create a list of instances of the rule classes which represent its tag instances. Typically this list is a `<fixedlist>` where each member in the list is a simple `<create>` expression.

For example, for the `PersonalBenefitAllowanceObjectiveTimeline` rule class, its implementation of `tagTimelines`, in pseudo-code, would be:

- Create a list of `AbstractTagTimeline`, with members:
  - Create an instance of `PersonalBenefitAllowanceWeeklyDeliveryTagTimeline`; and
  - Create an instance of `PersonalBenefitAllowanceWeeklyDeliveryTagTimeline`.

When creating tags, you will need to pass in any additional context required by that tag (by specifying values to set in the `<create>` expressions). In turn, this additional context may give rise to additional "context" rule attributes being required on the objective rule class itself.

For example, if the value of an objective's tag depends on total income of the person targeted by the objective, then the tag instance rule class may require the person to be set as a context rule attribute; in turn, the objective instance rule class will need such a rule attribute in order to pass it to the tag instance at creation time.

Create objective instances from your case rule class

Now that you have created rule classes for your objective instances, you must implement how the case calculates which objectives are available.

The default implementation of `objectiveTimelines` inherited from `DefaultProductEligibilityEntitlementRuleSet.DefaultCase` returns an empty list - i.e. the case has no objective instances at all.

For your case rule class that you created, you must now override and implement its inherited `objectiveTimelines` attribute to create a list of instances of the rule classes which represent its objective instances. This attribute will already have been created within the default eligibility and entitlement rule set that is created for benefit products by the dynamic product wizard, but it should be edited prior to product use.

If your product contains only single objectives, your implementation of `objectiveTimelines` will typically be a `<fixedlist>` where each member in the list is a `<create>` expression, e.g. (in pseudo-code):

- Create a list of `AbstractObjectiveTimeline`, with members:
  - Create an instance of `BasicHouseholdAllowanceObjectiveTimeline`, setting `productDeliveryCase` = `this.productDeliveryCase`; and
  - Create an instance of `ColdWeatherPaymentObjectiveType`, setting `productDeliveryCase` = `this.productDeliveryCase`.

If your product has multiple objective instances of a given type, your implementation of `objectiveTimelines` will typically be a `<dynamiclist>` to create an objective instance for each object of a particular kind, e.g.

- For each child in the household:
  - Create an instance of `ChildBenefitAllowanceObjectiveType`, setting `child` = current child.

Your product may contain a mixture of single objective instances and multiple objective instances (and indeed many different types of multiple objectives for different types), in which case you will need to nest the `<fixedlist>` and `<dynamiclist;` creations within a `<joinlists>` expression.

It may be clearer to factor out the creation of different types of objectives to their own rule attribute before joining the lists together, e.g.

- Create an attribute called `singleObjectiveTimelines`, which creates a fixed list of all the single objective instances for your product;
- Create an attribute called `childBenefitAllowanceObjectiveTimelines`, which creates a dynamic list containing one objective instance per child on the case;
- Implement `objectiveTimelines` to join together the `singleObjectiveTimelines` and `childBenefitAllowanceObjectiveTimelines` lists.

> **Important:** The list of object timelines for your case is a simple list which does *not* vary over time; rather, the case's entitlement to each objective is the thing that varies over time.
>
> You must create an objective instance for any objective which *could in theory* be attained at some point in the case lifetime, even if for some or all of the case lifetime it is not attained.
>
> For example, if a product has an objective type is aimed at paying child benefit for each child on the case, then at some point those children will each become adults but possibly remain resident in the household. At that point, the objective for that person (who was a child, but now an adult) will not longer be attained; but it still must be listed in the simple list of objective timelines for the case. As such, typically each *person* (rather than just each *child*) in the household should have a child benefit objective instance created for them; however, for a person who was already an adult when the case began will never be entitled to that objective.

When creating objective instances, you will need to pass in any additional context required by that objective instance (by specifying values to set in the `<create>` expressions).

For example, if the case requires there to be one objective targeted at each person in the household, then the objective instance rule class may require the person to be set as a context rule attribute; when creating the multiple objectives, the implementation of `objectiveTimelines` will have to create an objective for each person in the household, and pass that person to the objective instance so that it can use that person as its target.

> **Tip:** It can be useful to pass the case rule object's `productDeliveryCase` value to objective and tag instance classes, so that they can access its value of `caseID` and other data.

A note on manipulating Timelines in CER

Your output data from eligibility/entitlement calculations are centered around CER Timelines. For example, the `isEligibleTimeline` for the case and the `isEntitledTimeline` for your objectives are both timelines of Boolean values.

In general, most of your input data into eligibility/entitlement calculations is *already* in timeline format, as populated by the Active Succession Set Rule Object Converter (see Active Succession Set Rule Objects on page 115).

Typically, then, your rules for eligibility/entitlement calculations will generally transform input timelines into output timelines (via intermediate timelines). CER contains a number of expressions for manipulating timelines, but given the nature of eligibility/entitlement calculations, the expressions you should expect to see most commonly are `<timelineoperation>` and `<intervalvalue>`.

Use of other CER expressions for creating timelines is rare, but may be useful:

- to create timeline data from non-timeline data, such as custom entities (as opposed to custom evidence); and/or
- as "scaffolding" to hard-code timeline data (such as a rate that changes over time) during the early days of your product's implementation.

Write the Product Periods

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

For each period in your product, you must create a product period record and link it to the rule classes you created to:

- Describe the product structure
- Calculate eligibility/entitlement results for cases

The way you create and link these records differs depending on whether you are working in a development environment or a running system. A default product period is automatically inserted for a benefit product that is created via the dynamic product wizard, and this product period will have been automatically linked to the rule classes of the default eligibility and entitlement rule set that is also automatically inserted by the product wizard for the benefit product.

Working in a Development Environment

Create Data Mining Extensions (DMX) entries for any new rule sets you created for your rule classes (see section D.5.1. in the *Cúram Express Rules Reference Manual*).

For each product period, perform the following steps in the custom component:

- Create an entry in a *`CREOLERuleClassLink.dmx`* file, which points to the rule class for your product structure:

  Table 9: DMX data for `CREOLERuleClassLink` for your product structure rule class

Attribute Name	Value
`creoleRuleClassLinkID`	A unique ID from your custom key range.
`creoleRuleSetID`	The value of `CREOLERuleSet.creoleRuleSetID` you assigned above for the rule set containing your product structure rule class.
`ruleClassName`	The unqualified name of your product structure rule class.
`versionNo`	1

  > **Important:** You must create a separate record for use by each product period, even if multiple product periods point to the same product structure rule class.

- Create an entry in a *`CREOLERuleClassLink.dmx`* file, which points to the rule class for your case eligibility/entitlement calculations:

  Table 10: DMX data for `CREOLERuleClassLink` for your eligibility/entitlement rule class

Attribute Name	Value
`creoleRuleClassLinkID`	A unique ID from your custom key range.
`creoleRuleSetID`	The value of `CREOLERuleSet.creoleRuleSetID` you assigned above for the rule set containing your eligibility/entitlement rule class.
`ruleClassName`	The unqualified name of your eligibility/entitlement rule class.
`versionNo`	1

  > **Important:** You must create a separate record for use by each product period, even if multiple product periods point to the same eligibility/entitlement rule class.

- Create an entry in a *`CREOLEProductPeriod.dmx`* file:

Table 11: DMX data for `CREOLEProductPeriod`

Attribute Name	Value
`creoleProductPeriodID`	A unique ID from your custom key range.
`productID`	The ID of your CER-based product.
`startDate`	The start date of this product period. If your product has a single period, typically this start date should be the same as `Product.startDate`.
`endDate`	The end of this product period. If your product has a single period, typically this end date should be blank.
`productStructureRCLID`	The value of `CREOLERuleClassLink.creoleRuleClassLinkID` you assigned for your product structure rule class.
`decisionRCLID`	The value of `CREOLERuleClassLink.creoleRuleClassLinkID` you assigned for your case eligibility/entitlement calculation rule class.
`otherKeyDataRCLID`	Leave blank. (You will change this later if you implement key decision factor rules for your product.)
`nameID`	The value of `LocalizableText.localizableTextID` you assigned above.
`versionNo`	1

See the core data dictionary for a full description of these database columns.

Working in a Running System

Publish your rule sets containing your new rule classes.

Start the admin application and navigate to Product Delivery Cases, select your product, choose Rule Sets and copy the product for edit (if it is not already in edit).

For each product period in your analysis, perform the following steps:

- Create a product period;
- Set the value of "Product Structure Rule" to be the rule class you created for your product's structure; and
- Set the value of "Eligibility/Entitlement Rule" to the rule class you created for the calculation of a case's eligibility/entitlement results for your product.
- (Leave the value of "Key Decision Factors Rule" blank - it is not required for eligibility/entitlement rules.)

Do not set up any display categories - they are not required for eligibility/entitlement rules.

Publish your changes to the product.

Choose or Create a Summarizer Strategy

Displaying a Summary of Entitlement for a Coverage Period on page 35 described how the Engine can summarize the entitlement for a coverage period within a determination.

To use this feature, you must configure your Product to specify a strategy implementation to use. You must either:

- in development, change your *`CREOLEProduct.dmx`* file to populate your product's `detIntSummarizerStrategyType` column with the code (from the `DetIntSummarizerStrategy` code table) for your chosen strategy implementation; or
- in a running system, start the admin application and navigate to Product Delivery Cases, select your product, choose Rule Sets and choose Eligibility Determination, change "Decision Summary Display Strategy" to be your chosen strategy implementation.

When choosing a strategy implementation to use, you can either:

- use a strategy implementation included with the Engine (described below); or
- develop your own strategy implementation (described below).

Strategy Implementations Included with the Engine

The Engine includes these implementations which are suitable for most products:

Table 12: Summarizer Strategy Implementations Included with the Engine

Display/code	Implementation Class
blank	`curam.core.sl.infrastructure.assessment.impl.BlankDeterminationIntervalSummarizerS`
Total daily monetary entitlement	`curam.core.sl.infrastructure.assessment.impl.TotalDailyMonetaryEntitlementDetermin`
Total weekly monetary entitlement	`curam.core.sl.infrastructure.assessment.impl.TotalMonthlyMonetaryEntitlementDeterm`
Total monthly monetary entitlement	`curam.core.sl.infrastructure.assessment.impl.TotalWeeklyMonetaryEntitlementDetermi`

See the JavaDoc for the above classes for more details on the behavior of each strategy implementation.

Developing your own Strategy Implementation

If you have custom requirements not met by the implementations, you may develop your own strategy implementation(s) for use in your products as follows:

- Add a new entry to the `DetIntSummarizerStrategy` code table (using custom.ctx files);
- Create an implementation class which implements the `DeterminationIntervalSummarizerStrategy` interface; implement the required method to return an appropriate summary of the coverage period;
- Bind the code table entry to your implementation, in your custom Guice Module:

```
{
                      // Register your custom determination
 interval summarizer strategies
```

```
                 final
MapBinder<DETERMINATIONINTERVALSUMMARIZERSTRATEGYEntry,
                   DeterminationIntervalSummarizerStrategy>
                 determinationIntervalSummarizerStrategies =
MapBinder
                   .newMapBinder(binder(),

DETERMINATIONINTERVALSUMMARIZERSTRATEGYEntry.class,

DeterminationIntervalSummarizerStrategy.class);


determinationIntervalSummarizerStrategies.addBinding(

DETERMINATIONINTERVALSUMMARIZERSTRATEGYEntry.YOUR_STRATEGY).to(

YourDeterminationIntervalSummarizerStrategy.class);
                 }
```

(replacing *YOUR_STRATEGY* with the constant for your new code table code and *YourDeterminationIntervalSummarizerStrategy* with your strategy implementation class as appropriate)

- Build your application;
- Configure your product to use your new strategy (see instructions above).

Testing

For a complex product created in a development environment, you should create unit tests for individual parts of your product's eligibility/entitlement rules, using CER's support for rules testing.

You might consider creating end-to-end unit tests that test full scenarios involving the creation and activation of evidence, and the creation and activation of product delivery cases, to test that the overall eligibility and entitlement results are calculated as expected.

You might also perform manual testing of the online system to check that your eligibility/entitlement scenarios are handled as expected.

The Engine will be unable to calculate eligibility/entitlement results for a period in the case's lifetime if there is no product period covering part of the case's lifetime - to fix this you must change the product periods configured for your product so that all cases created have their entire lifetimes covered by exactly one product period. In particular, if your product allows open-ended cases, then typically the last product period for your product should be open-ended too (unless you intend your product to reach its end-of-life soon).

If the Engine detects a missing product period for a particular period in the case's lifetime, then:

- at determination calculation time, the Engine will:
  - store one or more instances of `DeterminationProblem` in the determination result (each problem stores a message and a stack trace of the underlying error, if any); and
  - write out the problem to the application logs (according to the setting of the Cúram Environment Variable *curam.creole.log.case.determination.problems*); and
- at determination display time, the Engine will display to the case worker an eligibility result of "Eligibility could not be determined" (as opposed to "Eligible" or "Not Eligible") for the coverage period within the case.

If there is a runtime error in the calculation of a CER attribute value for eligibility/entitlement, such as a reference not found (analogous to a `NullPointerException` in Java), or a division by zero, or any other calculation problem, then the Engine will throw an exception. The application logs will contain details of this exception including its stack trace. For CER calculation errors, the stack trace can include important information regarding the location within a CER rule set where the error occurred. To fix this, you will need to debug and retest your rules.

1.5 (deprecated) Calculating and Displaying Key Decision Factors

(deprecated) Introduction

The Engine contains features to calculate and display key decision factors - pieces of data that were important in arriving at eligibility and entitlement results.

When you design your product, you can choose to output such factors. The structure of the output data for key decision factors is imposed by the Engine; and since the structure is fixed, the Engine contains a generic set of screens to display the key decision factors to case worker users.

Typically, the rules for calculation of key decision factors will "sit on top of" the rules for calculating a case's eligibility and entitlement. If you follow the best practice recommendation and layer your rules this way, then you can make changes to the output of key decision factors for cases while guaranteeing not to affect any case's underlying eligibility/entitlement results. Data calculated for key decision factors never affects financial processing or any other processing - the data is used for display purposes only.

This chapter describes the flow of processing that allows key decision factors to be displayed to the user. The processing is intentionally described in reverse-chronological order; firstly we describe the end results, followed by the Engine processing that produces those results, before finally describing how the data was calculated.

This "backwards" perspective will echo how your rules designers will need to think when designing your product; they will need to start with the end in mind (namely how case workers will benefit from the additional ability to comprehend a case's details through the medium of key decision factors).

This chapter is structured as follows:

- **How it looks**

  Describes how key decision factors are displayed to a case worker.
- **How it works**

  Describes how fixed processing by the Engine and custom processing combine to calculate and display key decision factors.
- **How to use it**

  Describes the steps you will need to follow to implement key decision factors for your product.

(deprecated) How It Looks

The **How It Looks** section describes the win which key decision factors are displayed to a caseworker.

The structure of eligibility and entitlement data is intentionally restrictive; it contains very basic information such as the name of a key decision factor, the dates on which its value changes and/or text describing certain events that occurred.

Because the structure of the key decision factor data is fixed by the Engine, the Engine knows how to display this information for any Cúram Express Rules (CER)-based case, and so includes standard screens capable of displaying key decision factors. When you require to display data outside of this fixed structure, consider implementing decision details rules instead (see 1.6 Calculating and Displaying Decision Details on page 70).

(deprecated) Viewing Key Decision Factors Graphically

A graphical view of the key decision factors for a determination is provided through the **Graphical View** tab, for a case on a product which has key decision factors enabled.

A mixture of different types of event are displayed:

- Case events, which are automatically generated by the Engine - e.g. "Case Started";
- Eligibility/Entitlement events, which are automatically generated by the Engine - e.g. "Eligible $150 Weekly", "Not Eligible";
- Custom events specific to this product, for changes in a data value, e.g. "Total Income $50.00 Weekly", "Total Income $60.00 Weekly"; and
- Custom events specific to this product, for important events, e.g. "Client Turned 65: James Smith turned 65 and is no longer eligible.".

The events are displayed with earlier dates on the left and later dates to the right. A filter is provided to allow the user to restrict the events displayed to those of a particular type or to narrow the date range.

(deprecated) Viewing Key Decision Factors in a List

A list view of key decision factors for a determination is also provided as an alternative to the graphical view (described in the previous section), for a case on a product which has key decision factors enabled.

The list displays the events with the latest event shown first (which may be in the future). The list can be sorted in the standard way, by clicking on the column headings.

The list displays the date that the event occurred (or is expected to occur, for future events), the type of event and its text description. For Decision-type events, the user can click on the action to view the coverage period containing the decision event.

(deprecated) How It Works

The How It Works section describes how fixed processing by the Engine and custom processing combine to calculate and display key decision factors.

The calculation and display of a determination involves a mixture of:

- fixed processing contributed by the Engine; and
- custom product-specific processing contributed by the implementation of the product.

Note that it is not mandatory to configure custom key decision factors for your product periods; any or all of the product periods can opt not to use key decision factors, and if so no custom key decision factors will be displayed for those periods.

The Engine follows the following high-level steps to arrive at key decision factors that can be displayed to a case worker:

- At Determination Calculation time:
  - An action occurs which triggers the determination of a case (either an active or reactive determination);
  - The Engine identifies the product periods (configured for the product) that cover the case's lifetime;
  - The Engine uses CER rules (specific to the product) to calculate the custom key decision factors for each contributing product period;
  - The Engine calculates the key decision factors across the lifetime of the case by "splicing together" the key decision factors from each contributing product period;
  - The Engine stores (on the database) a determination result containing the key decision factors data (as well as eligibility/entitlement results and decision details, covered elsewhere in this document).
- At Determination View time:
  - A case worker requests to view a determination on a case;
  - The Engine retrieves the determination result from the database and interrogates the determination result to obtain its key decision factors;
    - the fixed case and decision factors for the determination, using the summarizer strategy configured for the product to produce a summary of the case's entitlement for each decision; and
    - the custom key decision factors for the determination (if any).

The data interfaces and implementation for calculation of key decision factors, and subsequent display of key decision factors are described in more detail below.

(deprecated) Calculation of Key Decision Factors

The system of interfaces and implementations involved in the calculation of the key decision factors that form part of a determination result follow a similar pattern to that for eligibility/entitlement calculations.

For more information, (see Calculation of Eligibility and Entitlement on page 31).

The responsibilities for calculating a case's key decision factors are divided between fixed implementations provided by the application and custom implementations for a product (some of which must adhere to interfaces included with the application).

Sections A and E describe a layer of fixed implementations similar to that of the eligibility/entitlement calculations, and contribute to calculating and storing the overall determination result, which includes the key decision factors. Although not described below, this layer also includes the calculation of contributing product periods.

The processing described in sections B, C, and D represents a combination of fixed interfaces and custom product-specific processing that is responsible for determining the key decisions factors across product periods.

Similar to eligibility/entitlement calculations, section B describes a fixed interface included with the Engine and sections C and D both describe custom product-specific processing.

Although not described below, the fixed implementation and fixed interfaces contributed by the engine that are responsible for retrieving the data from entities, evidence, and rate tables for eligibility or entitlement calculations are also used by the custom product-specific processing that is described in section D.

(deprecated) A) `ProductEligibilityEntitlementRuleSet.ProductDeliveryCase` rule object

This is the single rule object that controls the overall determination for the case.

For more information, see (deprecated) A) `ProductEligibilityEntitlementRuleSet ProductDeliveryCase` rule object on page 31

(deprecated) B) `ProductKeyDataRuleSet AbstractCase` rule class

The `AbstractCase` rule class acts as the interface between the fixed key decision factor processing provided by the Engine, and the product-specific rules for the calculation of key decision factors for a case.

This "interface" rule class ensures that concrete sub-rule-classes have an implementation for the following rule attribute, which provides the fixed-structure key decision factors required by the Engine:

- `keyDataTimelines` - responsible for calculating the named key data items, and their dates when the key decision factor changes value or undergoes a significant event.

The Engine-supplied interface rule set has no implementations for this attribute - rule classes must be written (see below) to provide an implementation for the business requirements of the product; rather the interface rule class allows the Engine to communicate with the rule classes by specifying a contract for the data structures that it requires.

(deprecated) C) Custom rule classes for key decision factors

When the Engine calculates key decision factors for a product period, the Engine first asks the product period which rule class should be used (which is recorded on the product period as part of setting up the product).

The rule class specified on the product period must ultimately extend from the `ProductKeyDataRuleSet.AbstractCase` interface rule class. For ease of upgrades, it is recommended that the rule class extends the `DefaultProductKeyDataRuleSet.DefaultCase` rule class which provides default implementations.

The rule class must provide an implementation of `keyDataTimelines`, which is where the bulk of the implementation effort for the key decision factor work will lie.

(deprecated) D) Custom rule classes for calculations

The calculation of the key decision factors may require complex business recalculation, which may be served by custom "calculator" rule classes.

Typically the implementation of `keyDataTimelines` will reuse calculation rules which you have already implemented for eligibility/entitlement calculations, although on rare occasions some refactoring of the existing rules may be required to make the common rules suitable for both eligibility/entitlement and key decision factor purposes.

The calculations will typically ultimately retrieve (and thus depend on) entity, evidence or rate data. These dependencies behave in a similar way to those for eligibility/entitlement calculations (see E) Custom rule classes for calculations on page 33).

(deprecated) E) `DeterminationResult`

The Determination Result holds the key decision factors for the case, as `determinationKeyDataTimelines`.

For more information, see (described in I) `DeterminationResult` on page 34).

When splicing together key decision factors across different product periods, the Engine matches contributions from different product periods by the description of each key decision factor. For each key decision factor found, the Engine draws event dates from the different product periods; naturally enough, only the dates that fall within each product period's effective range are used.

The Engine shows the key decision factors in the order they are listed in the `keyDataTimelines` rule attribute value. Typically each product period will define the same key decision factors, in the same order, but in the rare situation where the ordering of key decision factors is different across contributing product periods, then the order is determined by the latest product period to contain that key decision factor.

(deprecated) Display of Key Decision Factors

The vast majority of data processing for key decision factors occurs at the time when the determination result was calculated, as described in the previous section.

However, some data processing for key decision factors occurs at the time that the data is viewed; and this section describes the display-time processing of key decision factors.

When a case worker views key decision factors for a determination (whether using the Graphical View or the List View), then the Engine retrieves the custom key decision factors stored in the case determination but also automatically adds in:

- case lifetime events; and
- case decision events.

(deprecated) Adding Case Lifetime Events

The Engine automatically adds key decision events for:

- the case's start date (with the actual start date taking precedence over an expected start date); and
- the case's end date (if the case has an end date, with the actual end date taking precedence over an expected end date).

These key decision events are part of the Engine's processing and cannot be customized.

(deprecated) Adding Case Decision Events

The Engine automatically adds key decision events for each decision period within a determination. The event shows whether the case is eligible or not during a decision period, and if eligible shows a summary of entitlement.

These key decision events are part of the Engine's processing and cannot be customized.

The Engine uses the Summarizer Strategy configured for the product to calculate the text to show on the event for the decision period. For more details, see Displaying a Summary of Entitlement for a Coverage Period on page 35.

(deprecated) How to Use It

Most of the high-level processing for key decision factors is fixed logic provided by the Engine. However, you will have to provide implementations for certain lower-level logic. In order to do this, you must understand the basic concepts of key decision factors.

In addition to providing an understanding of these concepts, this section describes the work you will need to do to complete the key decision factors logic for your product, as follows:

- Analysis;
- Implementation; and
- Testing.

> **Note:** This section describes the complete work for key decision factors logic; however, for short-cuts you can take to get your product up-and-running quickly, see Incremental Design on page 194.

(deprecated) Understanding Key Decision Factor Concepts

The Engine has a fixed data structure for key decision factors. Understanding the structure of this data is critical as you must map your business requirements to the concepts in this structure. Each case (for a product where key decision factors have been implemented) will calculate a list of key decision factors when a determination is stored.

Key decision factors can represent a value which changes over time, e.g. a person's income. They can also represent key events that happen once over the lifetime of a case, e.g, a person's date of birth. Once-off key events must be explicitly in a logical fashion that makes sense to a case worker.

(deprecated) Fixed Data Structure for Key Decision Factors

Each key decision factor is an object with:

- a localizable description (displayed as the name of the key decision factor);
- a value which changes over time such as a person's income (optional); and
- a list of named key events such as a person's date of birth (optional and described in more detail below).

Typically each key decision factor uses exactly one of these optional features - i.e. the key decision factor is centered around a value which changes or a list of key events. It is possible to combine both features if you require. (Technically it is also possible to use neither feature; however, such a key decision factor would not be displayed and would be somewhat pointless.)

(deprecated) Explicitly Named Key Events

In contrasts to key decision factors with values that change over time, certain key events do not represent changes in a single piece of data, but rather a once-off significant event

These key events are objects with:

- a localizable description, describing the event, e.g. person's date of birth; and
- the date on which the event occurred.

It is possible to combine a key decision factor that changes over time, e.g. an employment with earnings that vary over time, *and* the named events such as when the employment started and ended. For named events, you should analyze whether each event *always* occurs for the key decision factor, or just *may* occur. For example, an employment key decision factor will always have a start date but may or may not have an end date.

(deprecated) Analysis

You must understand the requirements for your product, and analyze how these requirements broadly map to the Engine's key decision factor/key event concepts before starting implementation.

Unlike the requirements for eligibility/entitlement, typically requirements for key decision factors are *not* enshrined in legislation, but instead are based off informed predictions of the kinds of information that may be useful to a case worker, when the case worker is trying to understand a complex case (and perhaps answer questions about the case from customers.)

As such, producing requirements for your key decision factors is perhaps more art than science. For a new product, you might consider revisiting your key decision factor requirements once a product has been live for some time, in light with the kinds of information that case workers are attempting to understand when they view determinations. If you layer your requirements and implementation according to the recommendations, you should be able to implement and deploy changes to your key decision factor rules without affecting any underlying eligibility/entitlement calculations.

The following steps should aid your analysis.

(deprecated) Identify which decision factors are "key"

For a non-trivial product, the eligibility/entitlement calculations are likely to be complex, with many layers and interactions.

Ultimately each of these calculations has a bearing on the overall eligibility/entitlement result; however, you must decide which of these results are "key" to aiding a case worker's understanding of a case. You might prepare a candidate list of key decision factors and discuss them with your business experts and/or senior end-users.

If you identify too few key decision factors, then case workers might not be able to readily understand a determination. By contrast, if you identify too many key decision factors, then the key decision view may become too cluttered for case workers to easily use. In particular, take care to distinguish candidates for key decision factor output from requirements for decision details (which are amenable to more detailed display - see 1.6 Calculating and Displaying Decision Details on page 70).

For example, let's say you have a product where a household is means tested, by comparing the total household income against a set of income thresholds.

The total household income is calculated by adding up the total household income for each person in the household. Moreover, each person can have many concurrent employments (e.g. a day job, a night job and/or a weekend job) and so the total income for each person is calculated by adding up the earnings from each income.

In this example, the following are candidates for key decision factors:

- which income threshold the household falls into;
- the income thresholds themselves (which happen to be constant across all products - still potentially useful to show to the case worker, though);
- the total household income;
- for each person in the household, the total person income;
- for each income, the earnings for that income, and the dates that the income started and ended.

An implementation which included all of the above key decision factors could well be too cluttered to use, so business analysts producing requirements for key decision factors must choose carefully.

(deprecated) Identify the cardinality and descriptions for your key decision factors

Each type key decision factor that you have identified will typically fall into one of these patterns:

- **Single key decision factor**

  Each case has exactly one instance of this key decision factor, e.g. a "total household income" key decision factor for the case;
- **Multiple key decision factor**

  Each case has a number of instances of this key decision factor, depending on the case's circumstances, e.g. a "total person income" key decision factor for each of the people in the case's household.

Note that the Engine does not impose these single/multiple key decision factor patterns; the creation of key decision factors can be as complex as your business requirements dictate.

For a single key decision factor, the description of the factor can be a fixed piece of (localized) text, e.g. "Total Household Income".

For a multiple key decision factor, each instance of the factor in a determination must have a unique name - this uniqueness is required by the Engine and also is necessary for the case worker to distinguish between multiple instances of the key decision factor. Thus the description for a multiple key decision factor typically requires a calculation involving some fixed text and some variable data from the case, e.g. "Total Person Income for <person-full-name>". Your analysis should include the calculation requirements for these kinds of descriptions.

(deprecated) Identify the data type for each key decision factor

For each key decision factor identified, you must analyze whether the key decision factor will display:

- the changes in value of a single piece of data; and/or
- important named events.

Each of these relay events to the case worker, as each change in value of an important piece of data (such as total household income) is a kind of event in itself - with the pattern that visually, reporting the new value of the data (such as $100) is enough information to communicate the event (since text for a named event such as "The total household income is now $100" could well be overly-wordy).

By contrast, events such as a person being born or dying are not changes in a single piece of data, and must be explicitly named events.

(deprecated) Implementation

Having analyzed your business requirements, you are now in a position to start the implementation of your key decision factors for your product.

You are likely to re-use calculation results already implemented for your eligibility/entitlement calculations, as typically the factors that identify as "key" are those already part of your eligibility and entitlement logic. As such, the effort required to implement key decision factors is typically far smaller than that required for eligibility/entitlement logic calculations.

It is recommended that you implement your key decision factors rule classes in a rule set separate from your eligibility/entitlement rule set(s), but allow your key decision factor rule classes to depend on your eligibility/entitlement rule classes (but not the other way around).

This approach means that you can evolve your key decision factor implementation in the future without having to retest your eligibility/entitlement implementation; this can be important since key decision factors are merely "view" data to aid the case worker, whereas eligibility/entitlement results may affect more critical business functions such as how much a client is actually paid.

It can be helpful to track the dependencies between your rule sets so that as your product evolves, you have an insight into how changes in one rule set might affect other rule sets that depend on it.

For each product period, you must create a rule class which is responsible for identifying and calculating the key decision factors for the case.

It is possible that your key decision factors are calculated in an identical way across product periods, in which case you may be able to re-use one case rule class for many product periods. Your factoring of common calculated eligibility/entitlement results may affect how you must factor your case rule classes for key decision factors.

You will write the following rule classes:

- one case rule class (per product period) to hold the overall identification and calculation of key decision factors;
- one or more key decision factor rule classes, one for each type key decision factor supported by your product; and
- zero or more key event rule classes, one for each type of named event supported by any of your key decision factors.

The sections below detail a step-by-step path to implement your key decision factors.

(deprecated) Write the Case rule class

Your rule class to identify and calculate key decision factors for a case must ultimately extend from the `ProductKeyDataRuleSet.AbstractCase` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductKeyDataRuleSet.DefaultCase` rule class which provides default implementations.

Here is a description of the attributes inherited from `AbstractCase`:

Table 13: Rule attributes inherited from `ProductKeyDataRuleSet.AbstractProduct`

Rule Attribute name	Data type	Description
`productDeliveryCase`	`ProductDeliveryCase`	The controlling rule object which is responsible for splicing together the determination result from the contributions made by the product period. Passed in when the instance of `AbstractCase` is created.
`keyDataTimelines`	List of `AbstractKeyDataTimeline`	The list of key decision factors for the case.

Create a rule class which extends `DefaultProductKeyDataRuleSet.DefaultCase`. The rule class should be named in line with your product, e.g. *`ProductName`* `KeyDecisionFactors` (the Engine does not have any technical constraint on the rule class name - rather a good name for your rule class may make it easier to develop and maintain your rule sets).

The inherited implementation of `keyDataTimelines` returns an empty list; leave this implementation for now and you will return to it once you have created your key decision factor classes.

(deprecated) Write the Key Decision Factor rule classes

For each type of key decision factor that you have identified, you must create a rule class which must ultimately extend from the `ProductKeyDataRuleSet.AbstractKeyDataTimeline` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductKeyDataRuleSet.DefaultKeyDataTimeline` rule class which provides default implementations.

Here is a description of the attributes inherited from `AbstractKeyDataTimeline`:

Table 14: Rule attributes inherited from `ProductKeyDataRuleSet.AbstractKeyDataTimeline`

Rule Attribute name	Data type	Description
`description`	Localizable message	The identifying description of this key decision factor (as displayed to the user)
`timeline`	Timeline of Object	The single varying value whose changes will be reported as events for this key decision factor.
`keyEvents`	List of `AbstractKeyEvent`	The named events for this key decision factor.

Create a rule class which extends `DefaultProductKeyDataRuleSet.DefaultKeyDataTimeline`. The rule class should be named in line with your key decision factor (*`KeyDecisionFactorName`* `KeyDecisionFactor`), e.g. `TotalHouseholdIncomeKeyDecisionFactor` (the Engine does not have any technical constraint on the rule class name - rather a good name for your rule class may make it easier to develop and maintain your rule sets).

Implement a meaningful `description` attribute for your rule class. For a single key decision factor, a fixed localizable message (typically from a resource file, using CER's ResourceMessage expression) may suffice, e.g. "Total Household Income".

For a multiple key decision factor, the implementation of the description attribute will need to have some variable text, e.g. "Total Person Income for <person-full-name>", typically using CER's `<ResourceMessage>` expression to substitute variable text for placeholder in a message from a resource file, e.g. "Total Person Income for {0}".

Your key decision factor rule class will typically need some context in order to calculate its events and its description (for key decision factors which support multiple instances). This context will typically be a rule object which will be passed in when an instance of your rule class is created (see sections below). You should identify the context and model rule attribute(s) for the context in your rule class.

For example, if your key decision factor shows the total income for a person, then the context required may be a rule object for that person, so that your key decision factor rule class can use that person's data to calculate total income.

Leave the `keyEvents` with its inherited implementation; you may revisit it later if required.

The inherited implementation of `timeline` returns a Timeline with a constant value (and thus has no change events); if your key decision factor has only named events (as supplied by the keyEvents attribute), then do not create an implementation for the `timeline` attribute.

If your key decision factor does require events for a single varying value, then you must implement the `timeline` attribute to obtain that value, typically retrieving an existing Timeline attribute from another rule class, perhaps creating a rule object instance based off the context passed in to your key decision factor rule class.

For example, if your key decision factor displays the total income for a person, then:

- your key decision factor rule class will have a `person` attribute, of type `Person` (another rule class);
- your key decision factor rule class will have a `personCalculator` attribute, which creates a `PersonCalculator` instance (`PersonCalculator` is another rule class), passing in the person instance;
- your implementation of the timeline attribute on your key decision rule class will retrieve the `personCalculator.totalIncome` rule attribute value (which is already a Timeline).

Write the Key Event rule classes

For each type of key decision factor that supports named events, you must create a rule class for each type of event. For example, if your key decision factor describes an employment, you might write these rule classes:

- `EmploymentStartedEvent`; and
- `EmploymentEndedEvent`.

For each type of key event that you have identified (if any), you must create a rule class which must ultimately extend from the `ProductKeyDataRuleSet.AbstractKeyEvent` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductKeyDataRuleSet.DefaultKeyEvent` rule class which provides default implementations.

Here is a description of the attributes inherited from `AbstractKeyEvent`:

Table 15: Rule attributes inherited from `ProductKeyDataRuleSet.AbstractKeyEvent`

Rule Attribute name	Data type	Description
`description`	Localizable message	The description of this key event (as displayed to the user)
`date`	Date	The date on which the event occurred (or is expected to occur).

Create a rule class which extends `DefaultProductKeyDataRuleSet.DefaultKeyEvent`. The rule class should be named in line with your key event (*`EventName`* `Event`), e.g. `EmploymentStartedEvent` (the Engine does not have any technical constraint on the rule class name - rather a good name for your rule class may make it easier to develop and maintain your rule sets).

Implement a `description` for your event, which may or may not require context data to be passed into your rule class, e.g.:

- (no context required) "Employment started"; or
- (age context required) "Customer turned <new age on birthday>".

If the calculation of the date is complex, you may wish to implement a derivation for the date attribute. Otherwise, the value of date can be set by the calling rules when implementing the `keyEvents` rule attribute (see below).

Tip: Events which have a date of `null` will not be displayed. This can be useful for optional events such as those for an end date. If you have an `EmploymentEndedEvent` which applies only if an employment has an end date, then you can simply create an `EmploymentEndedEvent` instance and set its date to the end date of the employment; if the end date is null (i.e. the employment is ongoing), the event will not display, but if the employment has an end date recorded then the event will have a non-null date and will display.

This treatment of null dates typically results in more maintainable rule logic that the alternative approach whereby there is conditional logic governing which event instances to create.

Relate each Key Decision Factor to its supported Key Events

For each key decision factor rule class, you must consider whether to implement the `keyEvents` attribute. The inherited implementation of keyEvents returns an empty list; if your key decision factor has only events for a single varying value (as supplied by the timeline attribute), then do not create an implementation for the `keyEvents` attribute.

If your key decision factor does require named events, then your implementation of `keyEvents` must create a list of event objects. Depending on your implementation of the key event rule class, the each key event rule object may require additional context to be passed in when it is created.

For a fixed set of events (e.g. a start event and an end event), your implementation of `keyEvents` will typically be a `<fixedlist>` where each member in the list is a `<create>` expression, e.g. (in pseudo-code):

- Create a list of `AbstractKeyEvent`, with members:
  - Create an instance of `EmploymentStartedEvent`, setting `date` = `myEmployment.startDate`; and
  - Create an instance of `EmploymentEndedEvent`, setting `date` = `myEmployment.endDate` (see tip above about how a null date prevents an event from being displayed).

For some events, there may be an arbitrary number depend on other conditions, such as a number of birthdays. For these types of events, you will need to use other constructs such as `<dynamiclist>` or `<joinlists>` to create your list of key events for the key decision factor.

Relate the Case to its supported Key Decision Factors

You must implement the `keyDataTimelines` attribute on your case rule object to return a list of key decision factors for the case.

If your product contains only single key decision factors, your implementation of `keyDataTimelines` will typically be a `<fixedlist>` where each member in the list is a `<create>` expression, e.g. (in pseudo-code):

- Create a list of `AbstractKeyDataTimeline`, with members:
  - Create an instance of `TotalHouseholdIncomeKeyDecisionFactor`, setting `productDeliveryCase` be `this.productDeliveryCase`; and
  - Create an instance of `ClaimantLifeEvents`, setting `productDeliveryCase` be `this.productDeliveryCase`.

If your product has multiple key decision factor instances of a given type, your implementation of `keyDataTimelines` will typically be a `<dynamiclist>` to create a key decision factor for each object of a particular kind, e.g.:

- For each person in the household:
    - Create an instance of `PersonIncomeKeyDecisionFactor`, setting `person` = current person.

Typically your product may contain a mixture of single key decision factors, and multiple key decision factors (and indeed many different types of multiple key decision factors), in which case you will need to nest the `<fixedlist>` and `<dynamiclist;` creations within a `<joinlists>` expression.

It may be clearer to factor out the creation of different types of events to their own rule attribute before joining the lists together, e.g.:

- Create an attribute called `singleKeyDecisionFactors`, which creates a fixed list of all the single key decision factors for your product;
- Create an attribute called `personIncomeKeyDecisionFactors`, which creates a dynamic list containing one key decision factor per person on the case;
- Implement `keyDataTimelines` to join together the `singleKeyDecisionFactors` and `personIncomeKeyDecisionFactors` lists.

Update the Product Periods

For each period in your product, you must modify the product periods you created for eligibility/entitlement calculations (see Write the Product Periods on page 51) to link each period to your case rule class for key decision factors.

The way you link these records differs depending on whether you are working in a development environment or a running system.

Working in a Development Environment

Create DMX entries for any new rule sets you created for your rule classes (see section D.5.1. in the *Cúram Express Rules Reference Manual*).

For each product period, perform the following steps in the custom component:

- Create an entry in a *CREOLERuleClassLink.dmx* file, which points to the rule class for your case rule class for key decision factors:

Table 16: DMX data for `CREOLERuleClassLink` for your key decision factors rule class

Attribute Name	Value
`creoleRuleClassLinkID`	A unique ID from your custom key range.
`creoleRuleSetID`	The value of `CREOLERuleSet.creoleRuleSetID` you assigned above for the rule set containing your key decision factors rule class.
`ruleClassName`	The unqualified name of your key decision factors rule class.
`versionNo`	1

> **Important:** You must create a separate record for use by each product period, even if multiple product periods point to the same key decision factors rule class.

- Update your entry in your *`CREOLEProductPeriod.dmx`* file, setting the following attribute:

Table 17: DMX data for `CREOLEProductPeriod`

Attribute Name	Value
`otherKeyDataRCLID`	The value of `CREOLERuleClassLink.creoleRuleClassLinkID` you assigned for your key decision factors rule class.

Working in a Running System

Publish your rule sets containing your new rule classes.

Start the admin application and navigate to Product Delivery Cases, select your product, choose Rule Sets and copy the product for edit (if it is not already in edit).

For each product period, set the value of "Key Decision Factors Rule" to be the rule class you created for your case's key decision factors.

Publish your changes to the product. Note that if the product has existing cases, these cases will be reassessed if batch processes are run to reassess all potentially affected cases, so that the new determination contains key decision factor data. If you have followed the recommendations regarding the dependencies between your rule sets, then no eligibility/entitlement changes should result from the reassessment (and thus no payments for cases will be affected).

(deprecated) Testing

For a complex product created in a development environment, you should create unit tests for individual parts of your product's key decision factor rules, using Cúram Express Rules's (CER) support for rules testing.

You might consider creating end-to-end unit tests that test full scenarios involving the creation and activation of evidence, and the creation and activation of product delivery cases, to test that the overall key decision factor results are calculated as expected.

You might also perform manual testing of the online system to check that your overall key decision factor scenarios are handled as expected.

The Engine may encounter runtime problems when calculating key decision factors, due to calculation errors in CER attribute values.

If there is a runtime error in the calculation of a CER attribute value for a key decision factor, such as a reference not found (analogous to a `NullPointerException` in Java), or a division by zero, or any other calculation problem, then the Engine will throw an exception. The application logs will contain details of this exception including its stack trace. For CER calculation errors, the stack trace can include important information regarding the location within a CER rule set where the error occurred. To fix this, you will need to debug and retest your rules.

1.6 Calculating and Displaying Decision Details

Introduction

The Engine contains features to calculate and display decision details on free-form screens for displaying the detailed breakdown of eligibility/entitlement calculations.

Note: The key decision factors feature is deprecated. For more information, see Deprecated features.

When you design your product, you can choose to output such decision details. Unlike the structure of eligibility/entitlement results and key decision factors, the structure of the output data for decision details is product-specific and so the content and layout of the data shown on screens must be defined (using dynamic Cúram User Interface Metadata (UIM) screens). The Engine uses these product-specific UIM screens to display the decision details to case worker users.

Typically, the rules for calculation of decision details will "sit on top of" the rules for calculating a case's eligibility and entitlement. If you follow the best practice recommendation and layer your rules this way, then you can make changes to the output of decision details for cases while guaranteeing not to affect any case's underlying eligibility/entitlement results. Data calculated for decision details never affects Cúram financials or other processing - the data is used for display purposes only.

This chapter describes the flow of processing that allows decision details to be displayed to the user. The processing is intentionally described in reverse-chronological order; firstly we describe the end results, followed by the Engine processing that produces those results, before finally describing how the data was calculated.

This "backwards" perspective will echo how your rules designers will need to think when designing your product; they will need to start with the end in mind (namely how case workers will navigate decision details, which details to display and how they are laid out on the screen).

This chapter is structured as follows:

- **How it looks**

  Describes how decision details are displayed to a case worker.
- **How it works**

  Describes how fixed processing by the Engine and custom processing combine to calculate and display decision details.
- **How to use it**

  Describes the steps you will need to follow to implement decision details for your product.

How It Looks

The **How It Looks** section describes how decision details are displayed to a caseworker.

Note: The key decision factors feature is deprecated. For more information, see Deprecated features.

Unlike eligibility/entitlement and key decision factor data, the structure of decision details data is intentionally very flexible; rules designers can include all different types of information in decision details.

Because the structure of the decision details data is very flexible, the Engine cannot know by itself how to display this information, and product designers must create dynamic Cúram User Interface Metadata (UIM) screens which can extract their required data from the decision details and format it appropriately for a case worker user.

8.1.2.0 *Multi-locale support for CER*

You can view the decision details data in your preferred language by enabling the application property `curam.display.rules.multi.locale`. For more information about multi-locale support, see the section on "Multi-locale support for CER" in the *Cúram Express Rules Reference Manual*.

For information about configuring localizable display rules, see the section "Configuring localizable display rules" in the *System Administration Guide*.

Summary Display Category

A decision details tab is provided for a coverage period within a determination.

The header area of the screen displays summary information about the case (its start and end dates) and the coverage period selected (its date range and the eligibility decision during that date range). This header area is supplied automatically by the Engine.

Below the header area is a strip of tabs, one for each display category configured for the product. It is recommended that the first tab shown should be an overall summary of the case's eligibility and entitlement calculation, because:

- This tab is displayed by default when the coverage period is shown; and
- The details from this first tab are also shown when the user expands a coverage period listed on the overall determination.

The main body of the screen contains "summary" details for the coverage period, using standard UIM features such as headers, tabular labels and values, and formatting such as bold text and a total line.

Decision Comparison

The Engine contains features that allow a decision details page to also show data from the previous case decision within the determination. This feature can be useful to see how details of the case have changed along the determination.

Sub-screens

The Engine contains features that allow rows of data on a decision details page to be expanded to show further details on a sub-screen. This feature can be useful to allow a case worker to drill down into further detail.

Basic Eligibility/entitlement Information

The Engine includes "basic" screens and decision detail rules to display:

- the overall case eligibility; and
- (if eligible) the objectives and tags that the case is entitled to.

This information is too basic to be of genuine use to a case worker, however you may find it useful to re-use this output in the early days of developing your product (with a view to removing this "basic" category once you have implemented custom decision details rules and screens for your product).

The screen definitions and decision details rules included with the application also serve as examples which may be useful when you come to implement your custom decision details; and these "basic" artifacts are also used as examples in the remainder of this chapter.

How It Works

The **How It Works** section describes how fixed processing by the Engine and custom processing combine to calculate and display decision details.

Note: The key decision factors feature is deprecated. For more information, see Deprecated features.

The calculation and display of a determination involves a mixture of:

- fixed processing contributed by the Engine; and
- custom product-specific processing contributed by you (i.e. the implementation of your product).

The list of display categories are configured at a product level; in other words, the strip of display category tabs available for a determination are the same no matter which product period contributes to a coverage period.

Note that for any display category, it is not mandatory to configure decision details rules for your product periods; any or all of your product periods can opt not to use decision details rules for a display category, and if so the Engine will display a message on the screen to say that no details are available. This situation can arise if a change in legislation means that a new display category must be introduced for a product, yet only new (later) product periods need to display details for that category (because that category of details are simply not relevant to an earlier period).

For example, let's say that a product is implemented which has a means test against a household's income. The product is configured to show an "Income" decision details tab which provides details of how the total income for the household was derived. However, in 2010 legislation changed so that from 2010 onwards, the means test includes the value of a household's assets as well as total income. To implement the new legislation, the following steps would be taken:

- a new evidence type would be created to record `Asset` details for people in a household;

- the product's configuration would be changed to include a new "Assets" tab;
- the product's lifetime would be divided into two product periods ("pre-2010" and "2010 onwards"); in other words the existing single product period would be ended and a new product period created;
- the "2010 onwards" product period would implement decision details rules for the "Assets" tab, but the "pre-2010" product period would not implement any new decision details rules, as the household assets have no relevance to that period on the case (and in any case there would be no historical asset evidence recorded prior to 2010, because the evidence type is new and did not exist at the time that older cases were created).

The Engine follows the following high-level steps to arrive at decision details that can be displayed to a case worker:

- At Determination Calculation time:
  - An action occurs which triggers the determination of a case (either an active or reactive determination);
  - The Engine identifies the product periods (configured for the product) that cover the case's lifetime;
  - For each display category configured for the product:
    - The Engine uses CER rules (specific to the product) to calculate the decision details for each contributing product period;
    - The Engine calculates the decision details across the lifetime of the case by "splicing together" the decision details from each contributing product period;
  - The Engine stores (on the database) a determination result containing the decision details for each display category (as well as eligibility/entitlement results and key decision factors, covered elsewhere in this document).
- At Determination View time:
  - A case worker requests to view the details for a display category on a coverage period within a determination on a case;
  - The Engine retrieves the determination result from the database, and extracts from the determination the decision details for the required category (the value of which may vary over the lifetime of the case).
  - The Engine gets the value of the decision details for the coverage period required, and creates an XML document containing the decision details data;
  - The Engine retrieves the dynamic UIM page configured to display decision details for the required display category;
  - The Engine displays header details for the case and coverage period; and
  - The dynamic UIM page extracts data from the XML document and formats that data to display the body of the decision details.
- If the screen contains expandable rows of data, then there is further processing at the time a row is expanded:
  - A case worker expands a row on a decision details page (which displays details for a display category on a coverage period within a determination on a case);
  - As above, the Engine retrieves the determination result from the database, and extracts from the determination the decision details for the required category. The Engine gets the value of the decision details for the coverage period required.

- The Engine uses the *subscreenName* passed in from the screen to identify the required attribute from the case rule object. The Engine obtains its value, and then looks through the list of rule objects to match on the `businessObjectID` passed in from the screen.
- The Engine creates an XML document for the matching rule object details.
- The top-level dynamic UIM page opens an inner dynamic UIM page; the inner page receives the XML document, extracts data from it and formats it to display the expanded details.

The data interfaces and implementation for calculation of decision details, and subsequent display of decision details are described in more detail below.

Calculation of Decision Details

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

The system of interfaces and implementations follows a similar pattern to that for eligibility/entitlement calculations (see Calculation of Eligibility and Entitlement on page 31) and key decision factors (deprecated) Calculation of Key Decision Factors on page 58; however, given the free-form structure of decision details data, there are some important differences which are described throughout this section.

The responsibilities for calculating a case's decision details are divided between fixed implementations provided by the application and custom implementations for a product (some of which must adhere to application-shipped interfaces).

Sections A and G describe a layer of fixed implementations similar to that of the eligibility/entitlement calculations, and contribute to calculating and storing the overall determination result, which also holds the decision details. Although not described below, this layer also includes the calculation of contributing product periods.

The processing described in sections B, C, D, E, and F represents a layer that results in rule objects that are created in-memory only and are not stored on the database. Similar to eligibility/entitlement calculations, section B and D describes a fixed interface shipped by the Engine and sections C, E, and F describe custom product-specific processing.

Although not described below, there is also a final layer similar to that described in Calculation of Eligibility and Entitlement on page 31 that is responsible for the creation of rule objects that are retrieved by the custom product-specific processing described in section F.

This section describes the important interfaces and implementations involved in the calculation of the decision details that form part of a determination result.

A) `ProductEligibilityEntitlementRuleSet.ProductDeliveryCase` rule object

This is the single rule object that controls the overall determination for the case, described in (deprecated) A) `ProductEligibilityEntitlementRuleSet ProductDeliveryCase` rule object on page 31.

B) `ProductDecisionDetailsRuleSet AbstractCase` rule class

The `AbstractCase` rule class acts as the interface between the fixed decision details processing provided by the Engine, and the product-specific rules for the calculation of decision details for a case (to provide details for a top-level display category screen; the interface for sub-screens is described below).

Unlike the interface rule classes for eligibility/entitlement and key decision factors, the `ProductDecisionDetailsRuleSet.AbstractCase` interface rule class does not mandate a fixed data structure for concrete sub-rule-classes to implement. Later we will see how the Engine "walks" free-form data to calculate decision details for the case.

C) Custom rule classes for decision details

When the Engine calculates decision details for a display category on a product period, the Engine first asks the product period which rule class should be used for that display category (which is recorded on the product period as part of setting up your product).

The rule class specified on the product period/display category must ultimately extend from the `ProductDecisionDetailsRuleSet.AbstractCase` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductDecisionDetailsRuleSet.DefaultCase` rule class which provides default implementations.

Your rule class will have one or more attributes annotated with `Display` or `DisplaySubscreen`, which is where the bulk of the implementation effort for your decision details work will lie.

D) `ProductDecisionDetailsRuleSet.AbstractCaseSubscreenDisplay` rule class

The `AbstractCase` rule class acts as the interface between the fixed decision details processing provided by the Engine, and the product-specific rules for the calculation of decision details for a case (to provide details for a sub-screen expanded from within a top-level display category screen).

This "interface" rule class ensures that concrete sub-rule-classes have an implementation for the following rule attribute:

- `businessObjectID` - identifies the object being expanded on the screen, so that details relevant to that object only can be retrieved and displayed.

E) Custom rule classes for sub-screen details

When the Engine accumulates decision details for a determination, the data accumulated may include data to be included on a sub-screen, i.e. a panel shown when a row on a dynamic UIM screen is expanded by a user.

Data for sub-screens must be specified by a rule class which must ultimately extend from the `ProductDecisionDetailsRuleSet.AbstractCaseSubscreenDisplay` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductDecisionDetailsRuleSet.DefaultCaseSubscreenDisplay` rule class which provides default implementations.

Your rule class will have one or more attributes annotated with `Display`, which is where the bulk of the implementation effort for your decision details work will lie.

F) Custom rule classes for calculations

The calculation of your decision details (and/or sub-screen data) may require complex business recalculation, which may be served by custom "calculator" rule classes.

Typically the implementation of your attributes annotated with `Display` or `DisplaySubscreen` will reuse calculation rules which you have already implemented for eligibility/entitlement calculations, although on rare occasions some refactoring of your existing rules may be required to make the common rules suitable for both eligibility/entitlement and decision details purposes.

You may also create new rule classes to accumulate data for display and/or transform data into a format more suitable for display. Your rule classes will have one or more attributes annotated with `Display`, but may also have non-annotated attributes to hold interim calculation results.

The calculations will typically ultimately retrieve (and thus depend on) entity, evidence or rate data. These dependencies behave in a similar way to those for eligibility/entitlement calculations (see E) Custom rule classes for calculations on page 33).

G) `DeterminationResult`

The Determination Result (described in I) `DeterminationResult` on page 34) also holds the decision details for the case, as `determinationDecisionDetailsTimelines`. `determinationDecisionDetailsTimelines` is a map from each display category to a timeline of XML data (which holds the varying decision detail data for that display category).

Each product period can contribute to the XML data for a particular display category. If a product period does not have rules configured for a particular display category, then for the period of time that the product period contributes to the case's determination, the XML data will be empty and no decision details can be displayed for that period.

The Engine calculates the XML data for a product period's contribution to a display category by starting with a rule object created for that product period/display category, and then "walking" the values on that rule object to gather XML data as follows:

- The Engine walks all the displayable data to find out all the "change dates" on which any timeline values change (because the eventual overall XML will change on each of these change dates). The Engine walks the displayable data in a recursive fashion as follows:
    - Create an instance of the decision details rule class (see C) Custom rule classes for decision details on page 75);
    - Find every attribute on the rule object which is annotated with the `Display` annotation (other non- `Display` attributes are ignored).
    - For each `Display` attribute, processing differs according to the type of the attribute:
        - **Timeline**

          If the value is a timeline, then inspect the timeline value to find the dates on which it changes value, and contribute these dates to the overall change dates.
        - **Rule Object**

          If the value is another rule object (which has not yet been walked), then recurse to inspect its `Display` attributes and add their change dates to the overall change dates.
        - **List**

          If the value is a list of rule objects or Timelines, then each item in the list is checked to identify change dates to contribute to the overall change dates.
        - **Other**

          If the value type is not a Timeline, Rule Object or List then it does not contribute to the overall change dates.
- For each date on which any display data changes, the Engine creates an XML document by re-walking the data above, to find the value of each displayable item on that date (see the example below in Basic Eligibility/Entitlement example XML output on page 77). There are special cases for sub-screen data and their business object IDs to automatically include them.

- The Engine combines the XML documents for each change date into a timeline of XML documents. This timeline holds all the displayable data for a display category on the case, and the Engine stores it in the map of display categories to XML timelines.

Basic Eligibility/Entitlement example XML output

Here is an example of an XML document for a particular change date, produced by the `AbstractBasicProductDecisionDetailsRuleSet.AbstractBasicCase` rule class included with the Engine (to display basic eligibility/entitlement details):

```
<DecisionDetails>
  <BasicCase>
    <isEligibleTimeline domain="SVR_BOOLEAN">true</
isEligibleTimeline>
    <displayObjectiveTimelines>
      <Item>
        <relatedReferenceTimeline domain="SVR_UNBOUNDED_STRING" /
>
        <index domain="SVR_INT64">0</index>
        <objectiveTypeID
           domain="SVR_UNBOUNDED_STRING">CREOLE 1</
objectiveTypeID>
        <targetIDTimeline domain="SVR_INT64">123</
targetIDTimeline>
        <isEntitledTimeline
           domain="SVR_BOOLEAN">true</isEntitledTimeline>
      </Item>
    </displayObjectiveTimelines>
    <displayObjectiveTimelineSubscreens>
      <Item>
        <displayTagTimelines>
          <Item>
            <pattern domain="FREQUENCY_PATTERN">000120100</
pattern>
            <valueTimeline
               domain="SVR_UNBOUNDED_STRING">0</valueTimeline>
          </Item>
        </displayTagTimelines>
        <businessObjectID domain="SVR_INT64">0</businessObjectID>
      </Item>
    </displayObjectiveTimelineSubscreens>
  </BasicCase>
</DecisionDetails>
```

Figure 1: Example XML document for Basic Eligibility/Entitlement data for a coverage period

The XML has the following structure:

- A top-level `DecisionDetails` element; this is a fixed-named element provided by the Engine;
- A `BasicCase` element; the name of this element is that of a rule class used to provide decision details rules (in this example, for the "basic" objectives/tags output); this element represents a `BasicCase` rule object and the child elements represent the point-in-time values for attribute values from the rule object (for attributes which have been annotated with `Display`, to command them to be recorded in this XML);
- A `isEligibleTimeline` element for the value of the `BasicCase. isEligibleTimeline` timeline during this coverage period (in this case "true"); also specifies the domain type of the

value so that the dynamic UIM understands the type of the data and can display or process it appropriately.

> **Note:** The domain type is automatically provided by the Engine by inspecting the data type of the rule attribute; usually this is sufficient but in some data conversion scenarios, you may explicitly specify the domain to use within the `Display` annotation.

- A `displayObjectiveTimelines` element for another attribute on `BasicCase` which is annotated with `Display`. This attribute returns a list of rule objects, and so this XML element has a child Item element (because in this example the list holds a single rule object - there will be one Item child element for each entry in the list value).
- The child elements of Item correspond to the attributes of the rule object in the `displayObjectiveTimelines` list value (in this case, attributes on the AbstractBasicProductDecisionDetailsRuleSet.DisplayObjectiveTimeline rule class)
- A `displayObjectiveTimelineSubscreens` element for an attribute on `BasicCase` which is annotated with `DisplaySubscreen`. This data will be used to allow the user to expand a row on the screen to drill down its detail by displaying a sub-screen of a data, keyed on the `businessObjectID` of the required item.

Display of Decision Details

Some data processing for decision details occurs at the time when the determination result was calculated, as described in the previous section.

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

However, in contrast to eligibility/entitlement and key decision factors, display of decision details also involves some considerable processing, described in this section.

A case worker can choose to view Decision Details in these ways:

- When viewing a determination's list of coverage periods, the user can expand a row for a coverage period, to display the decision details for the first display category configured on the product; and/or
- The user can click on the date range shown for a coverage period to display the full decision details pages, which show a clickable tab for each display category (with the first display category shown by default).

Either of these actions instructs the Engine to display decision details for a particular date and display category on a determination for a case. When instructed, the Engine performs the following steps:

- The Engine looks up the Dynamic UIM page to display, based on the display category requested. The configuration for which page to display is stored on the `CREOLEProductDecisionDispCat.displayPageName` database column.
- The web server loads the Dynamic UIM page, which will specify an appropriate server interface to call to retrieve data for the screen. The supported server interfaces are as follows:

Table 18: Supported server interfaces for decision details UIM pages

Server interface	Description
`CaseDetermination.viewDecisionDisplayRulesCategoryXML`	Retrieves XML data for the top-level decision details screen for a single coverage period.
`CaseDetermination.viewDecisionDisplayRuleCategorySubscreenXML`	Retrieves XML data for a sub-screen of decision details screen for a single coverage period and business object ID (identifying the sub-screen being expanded).
`CaseDetermination.viewPreviousDecisionDisplayRulesCategoryXML`	Retrieves XML data for the case decision period succeeded by the coverage period requested. Typically used in addition to `CaseDetermination.viewDecisionDisplayRulesCategoryXML` to provide data for a comparison screen.
`CaseDetermination.viewPreviousDeterminationDecisionDisplayRulesCategoryXML`	Retrieves XML data for the same period in the determination superseded by the requested determination - reserved for use by Engine screens to compare under/over-payment determinations only.

- The Dynamic UIM page calls its server interface(s) to retrieve XML data for the coverage period.
- The Dynamic UIM page contains XPath-like expressions for querying the XML data returned. The Dynamic UIM page executes these XPath-like expressions against the XML data obtained from the calls to server interface(s), and obtains data to use in UIM constructs (such as field values and conditions).
- The web server displays the formatted page of decision details.

Basic Eligibility/Entitlement UIM examples

The Engine includes with a set of screens to display decision details for basic objective and tag information (see Basic Eligibility/entitlement Decision Details on page 29). This section uses those screens to illustrate the mechanisms used for displaying decision details.

You should refer to the full Dynamic UIM page definitions and associated *`.properties`* files for the following pages:

- `CREOLEDisplayRules_basicCaseDisplay`; and
- `CREOLEDisplayRules_basicCaseDisplay_objectiveTagSubscreen`.

Use of data in a condition

`CREOLEDisplayRules_basicCaseDisplay.uim` makes use of a Boolean value in a condition, which governs which cluster is displayed when the case is eligible or ineligible:

```
<SERVER_INTERFACE
    CLASS="CaseDetermination"
    NAME="DISPLAY"
    OPERATION="viewDecisionDisplayRulesCategoryXML"
  />
...
  <CLUSTER
    NUM_COLS="1"
    SHOW_LABELS="FALSE"
    TITLE="Cluster.Title.Eligibility"
  >


    <CONDITION>
      <IS_FALSE
        EXTENDED_PATH=
"/DecisionDetails/BasicCase/isEligibleTimeline"
        NAME="DISPLAY"
        PROPERTY="xmlData"
      />
    </CONDITION>
...
```

The XPath-like syntax of `/DecisionDetails/BasicCase/isEligibleTimeline` retrieves the value from the XML returned. In the example XML shown in Basic Eligibility/Entitlement example XML output on page 77, the value retrieved would be "true" (as a SVR_BOOLEAN) from the xmlData, which enables it to be used in the condition for the cluster.

Displaying a list of data

`CREOLEDisplayRules_basicCaseDisplay.uim` makes use of a list to display the list of objectives to which an eligible case is entitled. The Item[] syntax is used to refer to an item in the list.

```
...
<CLUSTER
  NUM_COLS="1"
  TITLE="Cluster.Title.Entitlement"
>
  <LIST>

    <DETAILS_ROW>
      ...
    </DETAILS_ROW>

    <FIELD
      DOMAIN="SVR_UNBOUNDED_STRING"
      LABEL="Field.Label.ObejctiveTypeID"
    >
      <CONNECT>
        <SOURCE
          EXTENDED_PATH=
"/DecisionDetails/BasicCase/displayObjectiveTimelines/Item[]/
objectiveTypeID"
          NAME="DISPLAY"
          PROPERTY="xmlData"
        />
      </CONNECT>
    </FIELD>


    <FIELD
      DOMAIN="SVR_BOOLEAN"
      LABEL="Field.Label.Entitled"
    >
      <CONNECT>
        <SOURCE
```

```
          EXTENDED_PATH=
"/DecisionDetails/BasicCase/displayObjectiveTimelines/Item[]/
isEntitledTimeline"
          NAME="DISPLAY"
          PROPERTY="xmlData"
        />
      </CONNECT>
    </FIELD>


    <FIELD
      DOMAIN="SVR_INT64"
      LABEL="Field.Label.Target"
    >
      <CONNECT>
        <SOURCE
          EXTENDED_PATH=
"/DecisionDetails/BasicCase/displayObjectiveTimelines/Item[]/
targetIDTimeline"
          NAME="DISPLAY"
          PROPERTY="xmlData"
        />
      </CONNECT>
    </FIELD>


    <FIELD
      DOMAIN="SVR_UNBOUNDED_STRING"
      LABEL="Field.Label.RelatedReference"
    >
      <CONNECT>
        <SOURCE
          EXTENDED_PATH=
"/DecisionDetails/BasicCase/displayObjectiveTimelines/Item[]/
relatedReferenceTimeline"
          NAME="DISPLAY"
          PROPERTY="xmlData"
        />
      </CONNECT>
    </FIELD>
  </LIST>
</CLUSTER>
...
```

Connecting a top-level screen to a sub-screen

`CREOLEDisplayRules_basicCaseDisplay.uim` allows each objective in its list to be expanded to show makes use of a list to display the list of objectives to which an eligible case is entitled. The Item[] syntax is used to refer to an item in the list. This is the top-level, or "outer" screen in the example:

```
<CLUSTER
  NUM_COLS="1"
  TITLE="Cluster.Title.Entitlement"
>
  <LIST>
```

```
    <DETAILS_ROW>
      <INLINE_PAGE PAGE_ID=

 "CREOLEDisplayRules_basicCaseDisplay_objectiveTagSubscreen">
        <CONNECT>
          <SOURCE
            NAME="PAGE"
            PROPERTY="determinationID"
          />
          <TARGET
            NAME="PAGE"
            PROPERTY="determinationID"
          />
        </CONNECT>
        <CONNECT>
          <SOURCE
            NAME="PAGE"
            PROPERTY="displayDate"
          />
          <TARGET
            NAME="PAGE"
            PROPERTY="displayDate"
          />
        </CONNECT>
        <CONNECT>
          <SOURCE
            EXTENDED_PATH=
"/DecisionDetails/BasicCase/displayObjectiveTimelines/Item[]/
index"
            NAME="DISPLAY"
            PROPERTY="xmlData"
          />
          <TARGET
            NAME="PAGE"
            PROPERTY="businessObjectID"
          />
        </CONNECT>
      </INLINE_PAGE>
    </DETAILS_ROW>
    ...

  </LIST>
</CLUSTER>
```

This is the "inner" screen, `CREOLEDisplayRules_basicCaseDisplay_objectiveTagSubscreen`:

```
<PAGE_PARAMETER NAME="determinationID"/>
  <PAGE_PARAMETER NAME="displayDate"/>
  <PAGE_PARAMETER NAME="businessObjectID"/>
```

The inner screen receives the parameters from the outer screen.

```
<SERVER_INTERFACE
  CLASS="CaseDetermination"
  NAME="DISPLAY"
  OPERATION="viewDecisionDisplayRuleCategorySubscreenXML"
/>
```

The inner screen calls the `CaseDetermination.viewDecisionDisplayRuleCategorySubscreenXML` bean to get details for the `businessObjectID` passed. The bean returns XML for the required "Item".

```
<CLUSTER NUM_COLS="1">
  <LIST>


    <FIELD
      DOMAIN="SVR_UNBOUNDED_STRING"
      LABEL="Field.Label.Value"
    >
      <CONNECT>
        <SOURCE
          EXTENDED_PATH=
"/Item/displayTagTimelines/Item[]/valueTimeline"
          NAME="DISPLAY"
          PROPERTY="xmlData"
        />
      </CONNECT>
    </FIELD>
    ...
```

The inner screen contains XPath-like expressions to query the sub-screen XML to populate its sublist.

How to Use It

Most of the high-level processing for decision details is fixed logic provided by the Engine. However, you will have to provide implementations for certain lower-level logic. In order to do this, you must understand the basic concepts of decision details.

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

Note that for eligibility/entitlement and key decision factor work, you needed only to provide server-side logic, because the Engine contains fixed screens to display eligibility/entitlement and key decision factor output. By contrast, you must implement not only server-side logic but also Dynamic UIM screens for your decision details logic, and ensure that these implementations integrate correctly.

In addition to providing an understanding of decision details, this section describes the work you will need to do to complete the decision details logic for your product, as follows:

- Analysis;
- Implementation; and
- Testing.

> **Note:** This section describes the complete work for decision details logic; however, for shortcuts you can take to get your product up-and-running quickly, see Incremental Design on page 194.

Understanding Decision Details Concepts

The Engine contains these high-level concepts:

- **Top-level screen**

  A screen of decision details that is displayed when the user views a display category for a coverage period, showing a point-in-time view of the details of a case.
- **Sub-screen**

  A panel of details which is displayed when the user expands a row on a top-level screen (or parent sub-screen).
- **Case Rule Class**

  The rule class configured by a product period on a display category, responsible for identifying all the data to be made available for display.
- **Display attribute**

  A rule attribute annotated with the `Display` annotation, indicating that its data should be made available in decision details XML;
- **DisplaySubscreen attribute**

  A rule attribute on a Case Rule Class annotated with the `DisplaySubscreen` annotation, indicating that it returns a list of rule objects which can be queried for display on an expanded sub-screen.
- **businessObjectID**

  A unique numerical identifier for data displayed when a row of data is expanded to show a sub-screen.

Analysis

You must understand the requirements for your product, and analyze how these requirements broadly map to the Engine's decision details concepts before starting implementation.

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

Unlike the requirements for eligibility/entitlement, typically requirements for decision details are not enshrined in legislation, but instead are based off informed predictions of the kinds of information that may be useful to a case worker, when the case worker is trying to understand a complex case (and perhaps answer questions about the case from customers.)

As such, producing requirements for your decision details is perhaps more art than science (in a similar way as for key decision factors; however, requirements for decision details tend to be more complex than those for key decision factors, given the flexibility of the Engine's support for decision details and the wealth of data items that are candidates for display).

For a new product, you might consider revisiting your decision details requirements once a product has been live for some time, in light with the kinds of information that case workers are attempting to understand when they view determinations. If you layer your requirements and implementation according to the recommendations, you should be able to implement and deploy changes to your decision details rules without affecting any underlying eligibility/entitlement calculations.

The following steps should aid your analysis.

Identify the Display Categories

You must identify and name your display categories, and consider the order in which the categories will be displayed to a case worker.

Recall that the first category will be displayed by default when the user is presented with a row of tabs, and moreover the first category will also be used when the user expands a coverage period row when viewing details of a determination. It is recommended that you design your first display category to show overall "summary" details of the case's eligibility and entitlement calculations.

Sketch out the Screens

For anything other than the most trivial of screens, it can be helpful to sketch out an example of the data to be shown on each of your display category screens.

For each display category that you have identified, sketch out an example of the screen with some realistic data laid out appropriately, giving particular attention to:

- whether any expandable sub-screens are required to display further details on any data; and indeed whether the sub-screens themselves require further sub-screens (possibly creating separate sketches if warranted by the complexity of the sub-screens);
- any data which is only to be displayed if a particular condition is met (and possibly create a separate sketch for how the screen should look if this condition is not met); and/or
- any non-standard layout such as bold characters, total lines, etc.

Map displayed data to eligibility/entitlement data

For each data item displayed on your screen examples (and any data item used as a condition), you must identify where the data will come from.

Some data may be sourced from existing data used or derived during eligibility/entitlement calculations. Other data may need further transformation before being suitable for display.

For example, your rules for eligibility may state that, under certain circumstances, the household must be means-tested in order to determine eligibility (whereas under other circumstances, the household must fulfill other conditions to be eligible, but those conditions do not include a means-test).

You may require to display a "Means test" item on your decision details screen with values "Passed", "Failed" and "Not applicable". To populate this item, you may need a calculation specific to decision details to translate the condition of whether or not a means test is required, and if so whether it passed. This calculated "means test status" value is probably of no relevance to your underlying eligibility/entitlement rules and thus will require implementing specifically to support your decision details screen.

It may be helpful to keep track of which data for your screens is already available directly from eligibility/entitlement rules vs. which data requires screen-specific calculations.

Identify keys for sub-screens

For any expandable sub-screens, you must identify a numerical key (`businessObjectID`) that can be used to uniquely identify the row being expanded (and thus also uniquely identifies the sub-screen to display for the row).

Identify comparison data

Each of your screens will show data from a coverage period within a determination. However it also possible for a top-level screen to display additional data from the previous case decision.

For each top-level screen, you must identify whether (in addition to data from the "current" coverage period), the top-level screen will also display data from the previous case decision period (if any). Typically most top-level screens do *not* display comparison data.

Implementation

Having analyzed your business requirements, you are now in a position to start the implementation of your decision details for your product.

Note: The key decision factors feature is deprecated. For more information, see Deprecated features.

You are likely to re-use some of the calculation results already implemented for your eligibility/entitlement calculations, as some decision details are already part of your eligibility and entitlement logic. Other decision details may require additional calculations to make them suitable for display.

It is recommended that you implement your decision details rule classes in a rule set separate from your eligibility/entitlement rule set(s), but allow your decision details rule classes to depend on your eligibility/entitlement rule classes (but not the other way around).

This approach means that you can evolve your decision details implementation in the future without having to retest your eligibility/entitlement implementation; this can be important since key decision details are merely "view" data to aid the case worker, whereas eligibility/entitlement results may affect more critical business functions such as how much a client is actually paid.

It can be helpful to track the dependencies between your rule sets so that as your product evolves, you have an insight into how changes in one rule set might affect other rule sets that depend on it.

For each product period and display category, you must create:

- a rule class which is responsible for identifying and calculating the decision details for the case; and
- a dynamic Cúram User Interface Metadata (UIM) page which is responsible for retrieving the details and formatting them for display to a caseworker.

It is possible that your decision details are calculated in an identical way across product periods, in which case you may be able to re-use one case rule class for many product periods. Your factoring of common calculated eligibility/entitlement results may affect how you must factor your case rule classes for decision details.

You may also create an arbitrary number of rule classes to model the data for display and/or provide intermediate calculations for decision details data. The flexibility of decision details data means that there are no fixed data structures to adhere to (unlike the steps for implementing eligibility/entitlement and key decision factor rules).

The sections below detail a step-by-step path to implement your decision details.

Write the Case rule class

Your rule class to identify and calculate decision details for a particular display category on a case must ultimately extend from the `ProductDecisionDetailsRuleSet.AbstractCase` interface rule class. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductDecisionDetailsRuleSet.DefaultCase` rule class which provides default implementations.

Here is a description of the attributes inherited from `AbstractCase`:

Table 19: Rule attributes inherited from `ProductDecisionDetailsRuleSet.AbstractCase`

Rule Attribute name	Data type	Description
`productDeliveryCase`	`ProductDeliveryCase`	The controlling rule object which is responsible for splicing together the determination result from the contributions made by the product period. Passed in when the instance of `AbstractCase` is created.

Create a rule class which extends `DefaultProductKeyDataRuleSet.DefaultCase`. The rule class should be named in line with your product and display category, e.g. *`ProductNameDisplayCategoryName`* (the Engine does not have any technical constraint on the rule class name - rather a good name for your rule class may make it easier to develop and maintain your rule sets).

The inherited implementation from `DefaultCase` means that you have a valid rule class for use with decision details, but as yet your rule class will not gather any useful data for display.

Implement attributes to return top-level screen data

You must add new `Display` attributes to your case rule class to identify data that must be made available for display.

For calculated data which is already implemented via eligibility/entitlement calculations, you should implement an attribute to obtain this data. Typically you will use CER's <create> expression to create an instance of the eligibility/entitlement rule object (specifying the `productDeliveryCase` value passed in), and then use CER's `<reference>` expression to obtain data from the eligibility/entitlement rule object. The eligibility/entitlement rule object will tend to be required by many `Display` attributes and can be created in its own rule attribute which is *not* annotated with `Display`.

For data which requires display-specific calculations, then you must implement these calculations. Typically any intermediate steps in these calculations are not required for display, and so any attributes you implement for intermediate-only steps should *not* be annotated with `Display`. You may also create your own display-specific rule classes which contain a mixture of Display/non-Display rule attributes.

For a list of data items, you will typically use CER's <dynamiclist> expression to transform a list of data into a list of "wrapper" rule objects, where each rule object has annotated attributes which identify which values to display and which provide any display-specific calculations.

For example, you may require a summary display category to show the following:

- The name of the claimant;
- The number of people in the household; and
- A list of assets owned by the claimant.

You identify that all these details are available by manipulation of data already used in eligibility/entitlement calculations, and so your decision details need to create an instance of your eligibility/entitlement rule class (as is typically the case).

The name of the claimant can be obtained by navigating to the person rule object already retrieved by your eligibility/entitlement rules, and so create a `claimantName Display` attribute on your case rule object, which navigates the eligibility/entitlement rules to obtain the required data.

The number of people in the household is not directly relevant to eligibility/entitlement rules, so you implement a `householdCount Display` attribute which retrieves the list of household members from your eligibility/entitlement rules and then provides display-specific processing to count the number of items in the list.

To display the list of assets, you implement a rule class named `AssetDisplay` with the responsibility for display each asset. You implement an assets rule attribute on your case rule class which retrieves the eligibility/entitlement Asset instances and for each one creates an `AssetDisplay` instance, passing in the eligibility/entitlement `Asset` instance.

In pseudo-code, your case rule class ends up containing rule attributes like this:

- `eligibilityEntitlementCase` (*not* annotated with `Display`):
    - Create an instance of the eligibility/entitlement rule class for the product, specifying the value of `productDeliveryCase`
- `claimantName` (annotated with `Display`):
    - Retrieve the value of `personName` from the value of `claimant` from the value of `eligibilityEntitlementCase` worked out above (or, in a more Java-like notation, `eligibilityEntitlementCase.claimant.personName`)
- `householdCount` (annotated with `Display`):
    - Retrieve the value of `householdMembers` from the value of `eligibilityEntitlementCase` worked out above; and
    - Count the number of items in the `householdMembers` list.
- `assets` (annotated with `Display`):
    - Retrieve the value of `assets` from the value of `eligibilityEntitlementCase` worked out above.
    - For each asset, create an instance of `AssetDisplay`, passing in the current `asset`.
    - Return a list of `AssetDisplay` instances.

Tip: To aid parallel development of dynamic UIM screens while rules are being developed, initially you can create your Display attributes with dummy implementations which return fixed values; the real attribute implementations can then be developed while another developer creates the Dynamic UIM screens and tests them against the dummy rule attribute implementations.

Implement attributes and rule classes for sub-screen data

If your top-level screen allows rows to be expanded to show sub-screens, then you must create an attribute on your case rule class that returns a list of rule objects and annotate that attribute with the `DisplaySubscreen` annotation. This annotation allows the data to be searched by the Engine when a user expands a row of data. It is recommended that you name your attribute *`datatype`* `Subscreens` .

The rule objects returned in the list from your `DisplaySubscreen` attribute must ultimately extend the `ProductDecisionDetailsRuleSet.AbstractCaseSubscreenDisplay` rule class shipped by the Engine. For ease of upgrades, it is recommended that your rule class extends the `DefaultProductDecisionDetailsRuleSet.DefaultCaseSubscreenDisplay` rule class which provides default implementations.

Here is a description of the attributes inherited from `AbstractCaseSubscreenDisplay`:

Table 20: Rule attributes inherited from *`ProductDecisionDetailsRuleSet.AbstractCaseSubscreenDisplay`*

Rule Attribute name	Data type	Description
`businessObjectID`	Number	Identifier of the row being expanded (must be unique amongst all sibling rows).

Create a rule class which extends `DefaultProductDecisionDetailsRuleSet.DefaultCaseSubscreenDisplay` rule. It is recommended that you name your rule class *`Datatype`* `Subscreen`, in line with the attribute you created above.

Typically, instances of your rule class will be constructed passing some underlying eligibility/ entitlement rule object to be displayed on the sub-screen. You must provide an implementation of `businessObjectID` on your sub-screen rule class, and typically this implementation will refer to some identifier on the underlying eligibility/entitlement rule object.

The implementation of the attribute on your case rule class will typically use CER's <dynamiclist> expression to create an instance of your rule class to wrap some eligibility/ entitlement rule object.

For example, let's say that the `AssetDisplay` rule class in the previous section is used to populate a list of assets on the summary decision details screen, and that furthermore we want the user to be able to expand each asset to see further details such as purchase date and a list of valuations for that asset.

You would write the following:

- A `ValuationDisplay` rule class, with the following attributes:
  - `valuation` (the underlying eligibility/entitlement rule class), not annotated;
  - `valuationAmount`, annotated with `Display`, which retrieves the valuationAmount value from the underlying valuation;
- An `AssetSubscreen` rule class, extending `DefaultProductDecisionDetailsRuleSet.DefaultCaseSubscreenDisplay`, with the following attributes:
  - `asset` (the underlying eligibility/entitlement rule class), not annotated;
  - `businessObjectID`, not annotated, which retrieves the `assetID` from the underlying asset;
  - `purchaseDate`, annotated with `Display`, which retrieves the `purchaseDate` from the underlying asset;
  - `valuations`, annotated with `Display`, which retrieves the underlying valuations for the asset and for each one creates a ValuationDisplay to wrap it;
- An `assetSubscreens` attribute on your case rule object, annotated with `DisplaySubscreen`, which retrieves all the assets on the case and for each one creates an `AssetSubscreen` to wrap it.

It is possible that your top-level screen allows multiple types of expansion (i.e. your top-level screen shows multiple lists, each of which may be expanded). For each type of expansion, you must create separate `DisplaySubscreen` attributes in your case rule class.

It is also possible that your sub-screens also allow further drill down into sub-screens. In this situation, all data for the lower sub-screens still needs to be returned by a `DisplaySubscreen` attribute on your case rule class.

Tip: CER's `<joinlists>` expression can sometimes be useful to aggregate lists of lists into a single list in such a situation.

Write the Dynamic UIM screens

For each top-level screen and sub-screen, you must write a Dynamic UIM file and associated *`.properties`* file, and then store these files on the database.

Top-level screens

For each top-level screen, write UIM including the following:

- Accept these page parameters:

  ```
  <PAGE_PARAMETER NAME="determinationID"/>
  <PAGE_PARAMETER NAME="displayDate"/>
  ```
- Call the standard interface to retrieve XML data for a display category for a coverage period, and connect the parameters required by the call:

  ```
  <SERVER_INTERFACE
    CLASS="CaseDetermination"
    NAME="DISPLAY"
    OPERATION="viewDecisionDisplayRulesCategoryXML"
  />
  <CONNECT>
    <SOURCE
      NAME="PAGE"
      PROPERTY="determinationID"
    />
    <TARGET
      NAME="DISPLAY"
      PROPERTY="key$determinationID"
    />
  </CONNECT>
  <CONNECT>
    <SOURCE
      NAME="PAGE"
      PROPERTY="displayDate"
    />
    <TARGET
      NAME="DISPLAY"
      PROPERTY="key$date"
    />
  </CONNECT>
  <CONNECT>
    <SOURCE
      NAME="TEXT"
      PROPERTY="CategoryRef"
    />
    <TARGET
      NAME="DISPLAY"
      PROPERTY="key$categoryRef"
    />
  </CONNECT>
  ```

- Decide on a "category reference" for your display category (which will be used later when configuration your product's display categories). Create a `.properties` entry for the *CategoryRef* of the display category:

  ```
  CategoryRef=MY_CATEGORY_REF
  ```
- If your top-level screen requires data from the previous decision period to compare to, then write a call to perform additional retrievals and connect the parameters:

  ```
  <SERVER_INTERFACE
    CLASS="CaseDetermination"
    NAME="DISPLAY_PREV"
    OPERATION="viewPreviousDecisionDisplayRulesCategoryXML"
  />
  <CONNECT>
    <SOURCE
      NAME="PAGE"
      PROPERTY="determinationID"
    />
    <TARGET
      NAME="DISPLAY_PREV"
      PROPERTY="key$determinationID"
    />
  </CONNECT>
  <CONNECT>
    <SOURCE
      NAME="PAGE"
      PROPERTY="displayDate"
    />
    <TARGET
      NAME="DISPLAY_PREV"
      PROPERTY="key$date"
    />
  </CONNECT>
  <CONNECT>
    <SOURCE
      NAME="TEXT"
      PROPERTY="CategoryRef"
    />
    <TARGET
      NAME="DISPLAY_PREV"
      PROPERTY="key$categoryRef"
    />
  </CONNECT>
  ```
- Write custom UIM to query the *xmlData* returned from the server call(s) via XPath-like expressions, and lay out this data on the screen. (This step is typically where the bulk of your screen implementation effort will lie.)
- If the screen allows the expansion of rows to show sub-screens, write UIM to connect to your sub-screen.

See the screen `CREOLEDisplayRules_basicCaseDisplay.uim` included with the Engine for a full example of a decision details top-level screen (including connection to a sub-screen).

Sub-Screens

For each sub-screen, write UIM including the following:

- Accept these page parameters:

  ```
  <PAGE_PARAMETER NAME="determinationID"/>
  <PAGE_PARAMETER NAME="displayDate"/>
  <PAGE_PARAMETER NAME="businessObjectID"/>
  ```

- Call the standard interface to retrieve XML data for a business object within a display category for a coverage period, and connect the parameters required by the call:

  ```
  <SERVER_INTERFACE
    CLASS="CaseDetermination"
    NAME="DISPLAY"
    OPERATION="viewDecisionDisplayRuleCategorySubscreenXML"
  />
  <CONNECT>
    <SOURCE
      NAME="PAGE"
      PROPERTY="determinationID"
    />
    <TARGET
      NAME="DISPLAY"
      PROPERTY="key$determinationID"
    />
  </CONNECT>
  <CONNECT>
    <SOURCE
      NAME="PAGE"
      PROPERTY="displayDate"
    />
    <TARGET
      NAME="DISPLAY"
      PROPERTY="key$date"
    />
  </CONNECT>
  <CONNECT>
    <SOURCE
      NAME="TEXT"
      PROPERTY="CategoryRef"
    />
    <TARGET
      NAME="DISPLAY"
      PROPERTY="key$categoryRef"
    />
  </CONNECT>
  <CONNECT>
    <SOURCE
      NAME="PAGE"
      PROPERTY="businessObjectID"
    />
    <TARGET
      NAME="DISPLAY"
      PROPERTY="key$businessObjectID"
    />
  </CONNECT>
  <CONNECT>
    <SOURCE
      NAME="TEXT"
      PROPERTY="SubscreenName"
    />
  ```

```
  <TARGET
    NAME="DISPLAY"
    PROPERTY="key$subscreenName"
  />
</CONNECT>
```

- Create a `.properties` entry for the *CategoryRef* of the display category, and the SubscreenName (which is the name of the attribute on the case rule object which is annotated with DisplaySubscreen):

  ```
  CategoryRef=MY_CATEGORY_REF
  SubscreenName=myCaseRuleAttributeWithDisplaySubscreenAnnotation
  ```
- Write custom UIM to query the *xmlData* returned from the server call via XPath-like expressions, and lay out this data on the screen. (This step is typically where the bulk of your screen implementation effort will lie.)
- If the screen allows the expansion of rows to show further sub-screens, write UIM to connect to your sub-screen.

See the screen *`CREOLEDisplayRules_basicCaseDisplay_objectiveTagSubscreen.uim`* included with the Engine for a full example of a decision details sub-screen.

Storing your screens

Refer to the *Cúram Web Client Reference Manual* for more details.

Configure the Product

For each display category identified for your product, you must create a display category record and link your product period(s) to it.

The way you create these records differs depending on whether you are working in a development environment or a running system.

Working in a Development Environment

Create DMX entries for any new rule sets you created for your rule classes (see section D.5.1. in the *`Cúram Express Rules Reference Manual`*).

For each display category, perform the following steps in the custom component:

- Create an entry in a *`LocalizableText.dmx`* file with the following attributes (to point to the name of your display category):

  Table 21: DMX data for `LocalizableText`

Attribute Name	Value
`localizableTextID`	A unique ID from your custom key range.
`richTextInd`	0
`versionNo`	1
- Create an entry in a *`TextTranslation.dmx`* file with the following attributes (to name your display category in your default locale):

Table 22: DMX data for `TextTranslation`

Attribute Name	**Value**
`textTranslationID`	A unique ID from your custom key range.
`localizableTextID`	The value of `LocalizableText.localizableTextID` you assigned above.
`localeCode`	Your default locale code, e.g. "en".

- Create an entry in a *`CREOLEProductDecisionDispCat.dmx`* file with the following attributes:

Table 23: DMX data for `CREOLEProductDecisionDispCat`

Attribute Name	**Value**
`creoleProductDecisionDispCatID`	A unique ID from your custom key range.
`categoryRef`	The category reference you assigned to your display category (and used in your UIM *`.properties`* files).
`displayOrder`	The placement[4] of this display category amongst those for the same product.
`displayPageName`	The name of your Dynamic UIM page that you created for this display category.
`nameID`	The value of `LocalizableText.localizableTextID` you assigned above.
`productID`	The ID of your CER-based product.
`versionNo`	1

For each product period, you must decide whether the product period will support display of decision details for your display category.

For each product period that supports your display category, perform the following steps in the custom component to link your product period to your display category:

- Create an entry in a *`CREOLERuleClassLink.dmx`* file, which points to the rule class for your case rule class for decision details:

Table 24: DMX data for `CREOLERuleClassLink`

Attribute Name	**Value**
`creoleRuleClassLinkID`	A unique ID from your custom key range.
`creoleRuleSetID`	The value of `CREOLERuleSet.creoleRuleSetID` you assigned above for the rule set containing your decision details rule class.
`ruleClassName`	The unqualified name of your decision details rule class.

---

[4] Thus the display category with the lowest `displayOrder` value for the product will be displayed first (and also displayed when a coverage period row is expanded).

Attribute Name	Value
`versionNo`	1

Important: You must create a separate record for use by each product period, even if multiple product periods point to the same decision details rule class.

- Create an entry in a *`CREOLEProductPeriodDispCat.dmx`* file with the following attributes:

Table 25: DMX data for `CREOLEProductPeriodDispCat`

Attribute Name	Value
`creoleProductPeriodDispCatID`	A unique ID from your custom key range.
`creoleProductPeriodID`	The ID of your product period.
`creoleProductDecisionDispCatID`	The value of `CREOLEProductDecisionDispCat.creoleProductDecisionDisp` you assigned above.
`decisionDetailsRCLID`	The value of `CREOLERuleClassLink.creoleRuleClassLinkID` you assigned above.
`versionNo`	1

See the core data dictionary for a full description of these database columns.

Working in a Running System

Publish your rule sets containing your new rule classes.

Start the admin application and navigate to Product Delivery Cases, select your product, choose Rule Sets and copy the product for edit (if it is not already in edit).

Click on "Display Categories", and for each display category in your analysis, perform the following steps:

- Create a new display category;
- Set the value of "Name" to be the name of the display category in the user's locale;
- Set the value of "Display Order" to be the placement of this display category amongst those for the same product;
- Set the value of "Display Page" to be the placement of this display category amongst those for the same product;
- Set the value of "Display Order" to be the name of your Dynamic UIM page that you created for this display category; and
- Set the value of "Category Reference" to be the category reference you assigned to your display category (and used in your UIM *`.properties`* files).

Click on "Product Periods" and for each product period perform the following steps:

- For each display category that the product period must support, choose "Associate Decision Details Rule...";
- Choose the Display Category; and
- Search for your case rule class that you created for the display category.

Publish your changes to the product.

Testing

For a complex product created in a development environment, you should create unit tests for individual parts of your product's decision details rules, using Cúram Express Rules's (CER) support for rules testing.

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

You might consider creating end-to-end unit tests that test full scenarios involving the creation and activation of evidence, and the creation and activation of product delivery cases, to test that the overall decision details results are calculated as expected.

You might also perform manual testing of the online system to check that your overall decision details scenarios are handled as expected. This step is particularly important for decision details, as the implementation of decision details involves custom screens (unlike the display of eligibility/entitlement and key decision factor results, which use screens included with the Engine).

It is particularly important to ensure that the XPath-like expressions in your Dynamic UIM screens matches the structure of the XML data produced by the Engine when it creates a determination; the structure of the XML data is in turn mandated by the annotations on your case rule object for the product period/display category.

> **Tip:** In a development environment, it can be useful to set breakpoints on the `CaseDetermination` facade method(s) called from your Dynamic UIM pages, so that you can inspect the XML returned. In a situation where a screen value is failing to display as intended, it can be useful to see if:
>
> - the value is present in the returned XML, in which case the error most likely lies with your XPath-like expression in the Dynamic UIM; or
> - The value is not present in the returned XML, in which case the error most likely lies with the annotations in your rule classes for decision details.

Note that it is possible for the XML returned from the server to contain data which is not used on any decision details screen. Under rare circumstances, it is possible for a case worker to view a determination, shown as a number of coverage periods, with the decision details for those coverage periods seeming (to the case worker) to appear as identical. In this situation, it is possible that the Engine is splitting the determination into coverage periods based on some data in the XML which is not shown to the user. You should take care to ensure that there is no such extraneous data in your XML - i.e. that all `Display` attributes do in fact appear on one or more decision detail screens or subscreens.

The Engine may encounter runtime problems when calculating decision details for a particular display category, due to calculation errors in CER attribute values.

If there is a runtime error in the calculation of a CER attribute value for a display category, such as a reference not found (analogous to a `NullPointerException` in Java), or a division by zero, or any other calculation problem, then the Engine will throw an exception. The application logs will contain details of this exception including its stack trace. For CER calculation errors, the stack trace can include important information regarding the location within a CER rule set where the error occurred. To fix this, you will need to debug and retest your rules.

1.7 Understanding Rule Object Converters and Propagators

Introduction

Cúram eligibility and entitlement processing relies on CER to calculate determination results based on details of the case, evidence recorded against the case and stand-alone data such as personal details and rates. CER is able to access such data via the use of *rule object converters*, which are responsible for retrieving data from business tables and converting that data into rule objects that can be used by CER.

Cúram's eligibility and entitlement processing also relies on the Dependency Manager to track the data that a determination result depends on, and for the Dependency Manager to request a reassessment whenever such data changes. The Engine contains a number of *rule object propagators* which are responsible for listening for changes in data which could affect calculations, and for informing the Dependency Manager that such data has changed.

Typically, for each type of data that is used in determination calculations, there is a corresponding converter and propagator pair which complement each other; the converter retrieves data of that type when requested by CER calculations while the propagator listens for changes to that data of that type.

For some of these converter/propagator pairs, there are configuration options to set exactly which business data will be converted/propagated; and where there are such configuration options, both the converter and the propagator use the same underlying configuration data.

This chapter describes in details how the rule object converters and propagators work and how you can configure them for your business needs.

An Initial Assessment Example

This section walks through an example of how business data on the database is converted to CER rule objects throughout the lifecycle of a product and case and how changes in that data are propagated to the Dependency Manager.

The example follows these steps:

- A system administrator creates and publishes rule set information for a Product;
- A system administrator creates and publishes a new Rate Table;
- A case worker registers a Person;
- A case worker creates a new Case for that Person;
- A case worker adds an additional member to the Case;
- A case worker captures and activates some Income evidence; and
- A case worker activates the Case.

As these steps progress, we will see how data from various parts of the system is converted into CER rule objects.

Propagation processing (to inform the Dependency Manager of changes) occurs later in the case lifecycle when changes of circumstances occur (see 1.10 Reassessment - Handling Changes in Circumstance on page 170).

A System Administrator Creates and Publishes Rule Set Information for a Product

Before any kind of eligibility and entitlement processing can occur for a Product, that Product must first be created and rule set information for that Product published. For details on how a Product is built, see the see the *`How to Build a Product`* guide.

In this example, the Product is for a Sickness Benefit. The system administrator sets up the new Product details, associates rule set information to the Product, and submits the rule set information for publication.

When the rule set information is published, rows are written to a number of database tables, and in particular the data from the database tables listed here will be required later, during the calculation of each case's determination result:

- CREOLEProductPeriod;
- CREOLEProductPeriodDispCat; and
- CREOLERuleClassLink.

> **Important:** These database tables are configured to be converted and propagated by the Entity converter and propagator respectively (see Entity Rule Objects on page 107).
>
> The configuration is included by the application in the *`.../EJBServer/components/core/data/initial/blob/EntityPropagatorConfiguration.xml`* file.
>
> This configuration is critical to the correct operation the Engine, and must not be modified or removed by customers.
>
> Note that the data on the CREOLEProductDecisionDispCat database table is not required by rules processing and is *not* present in the configuration included in the application.

Later, during the calculation of a case's determination result, the Entity converter will populate rule object instances of these rule classes from the `ProductEligibilityEntitlementRuleSet` rule set:

- `ProductPeriod`;
- `ProductPeriodDisplayCategory`; and
- `RuleClassLink`.

> **Important:** The rule `ProductEligibilityEntitlementRuleSet` rule set provided by the application is critical to the correct operation of Cúram Eligibility and Entitlement, and must not be modified or removed by customers.

A System Administrator Creates and Publishes a New Rate Table

The rules for the new Sickness Benefit require a household's total income to be tested against a maximum threshold (i.e. if the total household income exceeds this threshold, then the case is not eligible).

This threshold will vary over time (as policy makers decide from time to time), so rather than hard-code the threshold into rules, the rule set designer has required that the threshold be stored in a new "Income Limits" rate table. The rates for this year and next year have already been decided.

The administrator creates the new rate table including an effective-dated change when this year's limit will increase to next year's limit.

When finished, the administrator adds the new rate table to the configuration for the Rate Rule Object Propagator (see Rate Rule Objects on page 105), and chooses to "Apply Changes" to the new rates.

The system is now configured to create rule objects for the cells in the new Income Limits rate table, and so creates a CER rule object (of class `RateRuleSet.RateCell`) for each distinct cell in the new rate table. CER stores these new rule objects on its database tables, for later retrieval during the calculation of a case's determination result.

The income threshold rate varies over time, and so the `valueTimeline` in the rule object reflects both this year's and next year's rate.

A Case Worker Registers a Person

An unmarried father applies for Sickness Benefit. Before his claim can be registered, the case worker user must register the father as a person on the system.

When the system registers the father, a row is inserted onto the Person database table.

The Entity Rule Object Propagator (see Entity Rule Objects on page 107) is configured to listen for changes to the Person database table, because that table is mapped to a rule class (`ParticipantEntitiesRuleSet.Person`), and so the Entity Rule Object Propagator informs the Dependency Manager that data that could potentially affect calculations has changed.

In this situation, the Dependency Manager identifies that the new data for the father does not affect any existing calculations and thus takes no further action.

A Case Worker Creates a New Case for that Person

Now that the father has been registered, the case worker goes on to register a Sickness Benefit case for the father.

When the system registers the case, it writes a row to each of these database tables:

- CaseHeader; and
- ProductDelivery.

Later, during case determination, the data on these rows will be converted to a `ProductEligibilityEntitlementRuleSet.ProductDeliveryCase` rule object by the Product Delivery Rule Object Converter (see Product Delivery Rule Objects on page 102). This rule object will be used to calculate the determination result for the case.

When the system registers the case, it also records the father as a member of the case, by writing a row to the CaseParticipantRole database table. Later, during case determination, the Entity Rule Object Converter will convert the data on this row to a `CaseEntitiesRuleSet.CaseParticipantRole` rule object for the father's role on the case.

At this point we now have:

- `RateCell` rule objects stored on CER's database tables rule objects for the new rate table; and
- business data stored on Cúram's application database tables, ready to be converted into CER rule objects during later processing:
    - product data;
    - non-case business data (personal details); and
    - case-specific data (product delivery/case details and case participant role details).

A Case Worker Adds an Additional Member to the Case

The father has a daughter who must be added to the case. The case worker registers the daughter on the system and adds her to the case.

The system creates additional rows on Person and CaseParticipantRole, and as before these rows are ready to be converted to rule objects during case determination processing later.

A Case Worker Captures and Activates Some Income Evidence

The rules for Sickness Benefit rely on calculating the total income for a household, and comparing that total income to a predetermined threshold.

The total income for the household can vary over time (as can the threshold, shown earlier as a rate table). In order that the system can calculate the total income (later), a record of the father's varying income must be stored on the system.

The case worker captures the history of the father's income since the start of the case, and the system stores the income data using evidence (see *`Designing Cúram Evidence Solutions`*).

The father has received pay rises over the lifetime of the income, and so a number of income records are stored within the same evidence succession set. Each evidence record bears the income effective from a particular date (i.e. the date of the pay rise).

While the income evidence is being recorded, each evidence record is "in edit", and so generally is not available to case determination calculations (i.e. would not be converted into rule objects by the active evidence converters). For eligibility and entitlement processing, rule objects represent the "best known truth" about real-world data, and while the evidence is in-edit, it has not yet become "truth", so to speak.

After all the income evidence has been captured and verified, the case worker chooses to activate the evidence, at which point its data becomes part of the system's "best known truth". Later, during case determination, evidence in the "active" state is converted into rule objects so that the evidence data can be used in CER's calculation of the determination result.

Later, during case determination, CER will make a request for the rule object for the father's income. The Active Succession Set Rule Object Converter will retrieve all the active evidence versions for the father's income (which form a single "succession set" of evidence) and create a single rule object representing the father's income changing over time. The rule object created has:

- static values for non-changing details regarding the `Income` (e.g. its owning `caseParticipantRoleID` and its `startDate`, which cannot vary over time); and
- timeline values for details which may change over time (e.g. its `amount`).

At this point we have all the data required in order to calculate the case's eligibility and entitlement (but no such calculation has yet occurred for this case).

A Case Worker Activates the Case

The members of the case and the case evidence have now been recorded, and so the case worker progresses the case through its approval and activation steps (see the *`Integrated Case Management Guide`*).

When the case is activated, the system assesses the case. The system:

- searches for the `ProductDeliveryCase` rule object for the case, matching on its `caseID` (in this example, search for a caseID value of 1234). The Product Delivery Rule Object Converter (see Product Delivery Rule Objects on page 102) reads the CaseHeader and ProductDelivery rows for caseID 1234 and forms a `ProductDeliveryCase` rule object which is returned to the Engine;
- requests the value of the `determinationResult` attribute value on the `ProductDeliveryCase` rule object, which causes CER to perform calculations in order to calculate the requested value. During the calculation, there are calculation rules which require data for the:
  - **Product**

    Data regarding the structure of the product (e.g. the product periods which make up the product) is retrieved by the Entity Rule Object Converter and converted into CER rule objects;
  - **Personal Details**

    Data from stand-alone entities such as personal details are also retrieved by the Entity Rule Object Converter;
  - **Rates**

    Rule objects for rates are retrieved directly from CER's database tables for stored rule objects; and
  - **Evidence**

    Data for evidence is retrieved by the Active Succession Set Rule Object Converter and converted into CER rule objects.
- gathers dependencies by analyzing the kinds of data retrieved by CER (above). These dependencies are then passed to the Dependency Manager for storage, so that there is now a record of which data the case's determination result depends on. The record of these dependencies will be checked whenever data (such as personal details or evidence) changes, so that the case can be automatically reassessed.
- stores the varying determination result on the CREOLECaseDetermination and CREOLECaseDeterminationData database tables, and stores the point-in-time financial decisions on the CaseDecision, CaseDecisionObjective and CaseDecisionObjectiveTag database tables (see 1.8 How Determinations Are Stored on page 147).

Note: CER does not store on the database any of the rule objects populated by rule object converters. The rule objects are brought into memory as required and are released when no longer required in memory.

The Framework for Converters and Propagators

The application maintains a registry of converter and propagators that allow application components to contribute to rule object conversion and rule object propagation processing.

At runtime, the application invokes the appropriate converter or propagator implementation from amongst those registered.

Some converters and propagators have fixed behavior, whereas others have configurable behavior. Typically each type of data is handled by a converter/propagator pair, which are responsible for managing their own configuration.

Important: The application will validate that each rule class is mapped at most once by the data configurations for a particular converter/propagator.

Do not attempt to use the same rule class in data configurations across different converter/propagator implementations.

The following types of rule objects are handled by the converters and propagators included with the application:

- Product Delivery Rule Objects;
- Rate Rule Objects;
- Entity Rule Objects;
- Active Succession Set Rule Objects; and
- Active Evidence Row Rule Objects.

Important: Some other application components also contribute their own rule object converters and propagators targeted at their own functional needs.

These other rule object converters and propagators are *not* suitable for eligibility/entitlement processing are so are not listed in this guide.

The sections below cover:

- rule objects for use with eligibility and entitlement processing;
- data configuration problems;
- data access points;
- logging; and
- supported domain types.

Rule Objects for Use with Eligibility and Entitlement Processing

This section describes the types of rule objects which can be used with eligibility and entitlement processing.

The Engine includes a number of rule object converters which are responsible for reading data from the database and populating rule objects so that CER can access their data in calculations.

The Engine also includes a number of rule object propagators which are responsible for listening for changes in data and notifying the Dependency Manager that data has changed, so that the Dependency Manager can request the Engine to reassess cases which may be affected by the data changes.

Product Delivery Rule Objects

Overview

The Product Delivery Rule Object Converter is responsible for populating the rule object for a product delivery case for a CER-based product. The Product Delivery Rule Object Propagator is responsible for detecting when changes in data occur that would affect the data populated on the case rule object and for interacting with the Dependency Manager.

Note: Any product delivery case for a product which is configured to use Cúram Rules is ignored by this converter and propagator.

Configuration

The behavior of the propagator can be configured via the "Reassessment Strategy" option on each CER-based product. See Propagation Processing on page 103 below.

Conversion Processing

The details of a CER-based product delivery case are converted to a `ProductEligibilityEntitlementRuleSet.ProductDeliveryCase` rule object. This rule object contains a calculated attribute `determinationResult` which has the responsibility of calculating the overall determination result for the case (see (deprecated) A) `ProductEligibilityEntitlementRuleSet.ProductDeliveryCase` rule object on page 31).

The rule object will be converted on demand whenever assessment processing requires it. For case assessment determinations (see Assessment Determinations on page 27), the Engine will analyze which input values were accessed by CER during the calculation of the determination result and record dependencies in the Dependency Manager. In particular, the determination result calculation will access the case's start and end dates (both actual dates and expected dates), and the dependencies recorded on these dates gives rise to CER's ability to automatically recalculate the determination result when these dates change.

The following case-level data items are used to populate the attributes on the rule object:

- from the CaseHeader database row:
  - caseID;
  - concernRoleID;
  - integratedCaseID;
  - startDate;
  - endDate;
  - expectedStartDate; and
  - expectedEndDate; and
- from the ProductDelivery database row:
  - productID.

Precedents Identified

Access to Product Delivery Rule Objects during CER calculations does not give rise to any precedents being identified.

Product Delivery Rule Objects are intended to be used to calculate determination results only.

Propagation Processing

Whenever the CaseHeader or ProductDelivery data changes for a CER-based product delivery, the propagator requests the `ProductEligibilityEntitlementRuleSet.ProductDeliveryCase` rule object for the case and manipulates it in memory.

For a product which has its reassessment strategy set to "Do not reassess closed cases", then:

- when a product delivery case for the product is closed, then the propagator removes the rule object in memory and also deletes all dependency records for the case's determination result, to prevent automatic reassessment of the case; and
- when a product delivery case for the product is re-opened and subsequently re-activated, then the Engine will calculate the case's determination result and will also build back up the dependency records for the case's determination result, to allow automatic reassessment of the case if data changes in the future.

For a product which has its reassessment strategy set to "Automatically reassess all cases", then no such processing occurs and the continuing presence of the dependency records in the Dependency Manager while a case is closed means that the case will continue to be reassessed due to data changes even if it is closed.

See Reassessment Strategy on page 206 for details of the processing that occurs if an administrator chooses to change the reassessment strategy for an existing product that has product delivery cases created against it.

Example

A CER-based product is set up in the system with a reassessment strategy set of "Do not reassess closed cases".

A case worker registers a new product delivery case for the product, with `concernRoleID` 2345, and an actual start date of 1st January 2001, but with no expected or actual end date (i.e. an open-ended case). When the case is activated, the Engine requests the rule object for the case, and CER invokes the Product Delivery Rule Object Converter to convert the case data into the required CER rule object. The Engine requests the rule object's `determinationResult` and also requests CER to determine the data dependencies for the calculation. The Engine passes these dependencies to the Dependency Manager for storage.

Some time later, a user maintains the case and enters an expected end date of 31st December 2001. The Entity Rule Object Propagator notifies the Dependency Manager that a CaseHeader row has changed, and the Dependency Manager identifies that the case requires reassessment. The Engine reassesses the case, and when it requests the rule object for the case the Product Delivery Rule Object Converter retrieves the latest data for the case and forms a rule object that has its expected end date set. The Engine gets the determination result from the rule object, which now takes into account the expected end date. The Engine detects that the determination result has changed (due to the expected end date now set) and stores the new determination.

The case comes to a natural end and is closed by the case worker. The Product Delivery Rule Object Propagator removes the dependency records for the case's determination result (because the product is set not to reassess closed cases).

Later, new evidence comes to light which requires the case to be re-opened. When the case worker re-activates the case, the Product Delivery Rule Object Propagator triggers the Engine to reassess the case to build back up the dependency records for the case's determination result. The Engine finds that there is no change in the case's determination result since it was last assessed and so does not store a new determination. The case worker records and activates the changes in evidence, and the Active Succession Set Rule Object Propagator notifies the Dependency Manager that evidence on the case has changed. The Dependency Manager identifies that the case requires reassessment and triggers the Engine to reassess the case, which will now take into account the recently-activated changes in evidence. The Engine finds that the determination result has changed and stores a new determination.

Rate Rule Objects

Overview

The Rate Rule Object Propagator is responsible for the creation, maintenance and removal of rate cell rule objects for nominated rate tables. In contrast to the other propagators, this propagator stores rule objects on CER's database tables.

The Engine contributes a `rate` expression to CER (see `rate` on page 225) which can be used to retrieve a rule object populated by the Rate Rule Object Converter.

Configuration

This propagator accepts configurations which adhere to the following structure:

- propagator type must be "ROPT2006" (the code for 'Rate' from the `RuleObjectPropagatorType` code table); and
- each rate table to be propagated must be listed in a `ratetable` element with a type matching the rate table's type code.

Configurations are cumulative, i.e. there may be many configurations of type "ROPT2006", and if a rate table's code is present in any of those configurations then the rate table will be propagated; otherwise, the rate table will be ignored.

The following type of configuration problem will be detected by the Rate Rule Object Converter:

- Rate table type code not specified in the `ratetable` element.

Any configuration problems detected will be processed according to Data Configuration Problems on page 139.

Conversion Processing

There is no converter for rate data - CER rule objects for rates are read directly from CER's database tables when retrieved during CER calculations.

Precedents Identified

If Rate Rule Objects are accessed during a CER calculation, and the CER utility is used to identify precedents, then internal IDs for the rate values on the CER rule objects will be identified directly by CER.

See the *`Cúram Express Rules Reference Manual`*.

Propagation Processing

This propagator creates CER rule objects for rate data, which are stored by CER on its database tables.

In contrast to other propagators, rate data is *not* propagated incrementally. This is because typically a user may want to change several rate tables before applying all the changes in one go, at which point the system will identify all the cases affected.

As such, rates are only propagated during initial/full propagation (as for the other propagators), or when the user explicitly chooses to apply rate changes to CER products. When any of these propagation triggers occur, the system will:

- read the configurations for the Rate Rule Object Propagator (i.e. those of type "ROPT2006");
- read all the rate cells for the configured rate tables;

- identify each rate cell's varying value over the history of the rate table (ignoring any range data);
- for each rate cell's varying value, create or update the corresponding `RateCell` rule object;
- remove any `RateCell` rule objects for rate cells which have been removed; and
- for any rates which have been created, updated or removed, inform the Dependency Manager of the changes in data so that the Dependency Manager can identify which cases require reassessment. Dependencies on rate data are stored at the Attribute Value level (i.e. per rate cell).

Important: Only rate cells which are in top-level rows and columns are propagated. Any rate cell which belongs to a sub-row and/or sub-column is ignored.

Only the value of a rate cell is used during propagation. Any min or max range data for the rate cells is ignored during propagation.

For more information on rate tables, see "Implementing Rate Tables" in the *Integrated Case Management Configuration Guide*.

Each rate cell will only be valid as far back as its earliest rate header record. Before this date, the timeline for the rate's varying value will be defaulted to zero.

Example

Let's say that a user creates a new rate table named "Income Limits", with a single column named "Maximum Allowable Limit" and a single row named "Sickness Benefit". Furthermore, the user specifies that the initial rate for this single column/row is 10,000 valid from 1st January 2000, rising to 12,000 effective 1st January 2001 (until further notice).

The administrator configures the Rate Rule Object Propagator to propagate the new "Income Limits" rate table, and then chooses the "Apply Changes" option for rate tables.

The Rate Rule Object Propagator will create a single `RateCell` rule object with a `valueTimeline` populated as follows:

- Beginning of time - 31st December 1999: 0 (the default value for rates for periods before the rate comes into effect);
- 1st January 2000 - 31st December 2000: 10,000; and
- 1st January 2001 - End of time: 12,000.

A case is registered and assessed, and the calculation of the determination result involves the use of the `rate` on page 225 expression to retrieve the "Maximum Allowable Limit" and "Sickness Benefit" rate timelines.

The Engine invokes the CER utility to identify these dependencies (which are stored using the Dependency Manager):

Table 26: Example Dependency Storage for Rate Rule Objects

Dependent		Precedent
Case 453's Entitlement	depends on	Attribute 'valueTimeline' on rule object ID '23423456' (RateRuleSet.RateCell)
Case 453's Entitlement	depends on	Attribute 'valueTimeline' on rule object ID '9879872342' (RateRuleSet.RateCell)

Entity Rule Objects

Overview

The Entity Rule Object Converter is responsible for converting a generic database row into a CER rule object. You should use this converter to populate CER rule objects from your CRUD-style entity data.

> **Important:** You should not use the Entity Rule Object Converter to populate rule objects from:
>
> - **Rate tables**
>
>   See Rate Rule Objects on page 105 instead; or
> - **Evidence**
>
>   See Active Succession Set Rule Objects on page 115 instead.

The Entity Rule Object Propagator is responsible for listening for database writes that occur in the application's processing and for notifying the Dependency Manager that data for entity rule objects has changed.

Configuration

The converter and propagator share a common set of configuration data, and accept configurations which adhere to the following structure:

- propagator type must be "ROPT2003" (the code for 'General CRUD entity' from the `RuleObjectPropagatorType` code table);
- each entity to be converted or propagated must be listed in a `table` element with a name exactly matching the database table's name, as per the application model;
- each propagation target must be listed as a `ruleset` element (within the `table` element), specifying the name of the rule set to target and optionally the rule class (if the name of the rule class differs from that of the database table);
- each `table` element may optionally specify a column/value combination which identifies the row as "canceled"; any row on the table which has this column/value combination will not be propagated - this feature allows physically and logically deleted rows to behave the same (i.e. to cause the Entity Rule Object Converter to not populate a rule object where the underlying database row is marked as logically deleted); and
- each `ruleset` element may optionally specify a `filter` element to specify a custom filter class which determines whether each row should be included or excluded from the Entity Rule Object Converter and Propagator. The `filter` element must specify the `name` of the filter implementation, and may include a `filterconfig` element to provide filter configuration specific to the filter implementation. The Engine includes an `EntityAttributePropagationFilter` which provides a simple ANT-like filtering mechanism based on `include` and `exclude` elements.

Configurations are cumulative, i.e. there may be many configurations of type "ROPT2003", and if an entity is present in any of those configurations then the entity will be converted and propagated; otherwise, the entity will be ignored.

> **Note:** Any database tables which are on the "exclude" list (see Propagation Processing on page 111) cannot be propagated (and will be ignored).

Tip: You are free to create your own rule classes to match database entities whose data you require in CER rules. However, you should first check whether existing rule classes are already suitable for your needs - while it is possible to convert each database row to many different rule classes, this flexibility comes with a potential maintenance cost for your rule sets.

In particular, the application includes configurations to convert and propagate data which is particularly often used in eligibility/entitlement processing to rule classes included with the Engine in these rule sets:

- `ParticipantEntitiesRuleSet`; and
- `CaseEntitiesRuleSet`.

You should consider reusing these rule classes when creating your own rule sets, in order to avoid maintenance overhead for your rule sets.

When you create your own rule classes for propagated data, it is recommended that you:

- implement a meaningful calculation of the `description` rule attribute (which can be useful for debugging); but
- do not implement any other calculated attributes on the rule class (in order to promote re-use of your rule classes across products).

If you require to map all rows from a database table to one rule class, but only the non-canceled rows to a different rule class, then you should create separate `table` elements which name the same database table, but only specify the `canceledValue` and `statusColumn` attributes on one of the `table` elements. Similarly, if you require different filters for different rule classes, create separate `table` elements which name the same database table.

The following types of configuration problems will be detected by the Entity Rule Object Converter/Propagator processing:

- Entity name not specified in the `table` element;
- The modeled database table with the specified name could not be found;
- The modeled database table does not have a single-field primary key;
- The modeled database table is propagated to a rule class which does not contain an attribute corresponding to the primary key of the rule class.
- A rule class is targeted by more than one source entity;
- The filter class specified for a filter is invalid; and
- The name or value is missing when using the `EntityAttributePropagationFilter`.

Any configuration problems detected will be processed according to Data Configuration Problems on page 139.

Conversion Processing

When a database row is converted to a rule object, then the values of the database columns are used to map to identically-named rule attributes on the rule object. Any database column without a corresponding rule attribute is ignored. The key column for the database table *must* have a corresponding rule attribute, as this rule attribute will be used to identify the rule object.

Rows from a database table may only be converted if the database table has been modeled to have a primary key which contains a single column, with a data type supported by the converter (see Supported Domain Types on page 143). If a database table has no primary key, or has a primary key made up of more than one column, then any attempt to use that database table in this converter's configuration will result in a warning being logged and the configuration for the database table will be ignored.

Restrictions on Access

In your CER rule sets you will use CER's `<readall>/<match>` expression to access rule objects converted from entity data.

You may only specify a `retrievedattribute` which matches a database column on the underlying database row used to populate the CER rule object. The data type of the attribute must be `Number` or `String`.

If you attempt to specify a `retrievedattribute` to be the name of an attribute which is calculated by CER, or which is of a data type other than `Number` or `String`, then the Entity Rule Object Converter will throw a runtime exception when the CER `<readall>/<match>` expression is executed.

Do not attempt to search for a `retrievedattribute` passing a value which would be stored as a NULL by the application's Data Access Layer. No rule objects will be found for such values, e.g. for a unique identifier value of 0 or an empty string.

You may specify the `ruleset` and `ruleclass` for the `<readall>` expression to be a rule class mapped by the data configuration. If you attempt to specify a rule class which is not directly mapped (e.g. a base rule class that you have created from which your concrete rule classes inherit) then no rule objects will be found.

Important: You can use `<readall>` without a `<match>` to retrieve *all* rule objects converted from the entity, but you should do so with care, because:

- there may be a large number of instances of rule objects for that entity; and/or
- the dependent will be recalculated every time a new row is stored for that entity or an existing row removed.

A `<readall>` without a `<match>` is likely to useful only to convert rule objects from a "control" entity which holds only a small number of rows and it is expected that additions to or removals from those control rows should cause dependents to be recalculated.

Precedents Identified

If Entity Rule Objects are accessed during a CER calculation, and the CER utility is used to identify precedents, then the following precedents will be identified:

Table 27: Precedents Identified for Entity Rule Objects

Name	When Identified	Trigger for Recalculation
Entity Row	Identifies any entity row for which: • a search was executed against the row's primary key (regardless of whether a row with that primary key value was found); and/or • one or more attribute values were accessed for the rule object populated data from the entity row with that primary key value. The precedent ID refers to the name of the entity, the name of the entity's primary key field and the primary key value sought.	A precedent change item for the entity row will be written to a precedent change set if: • any data on the entity row changes; • a new entity row is inserted onto the database[5]; and/or • an existing entity row is removed from the database.
'readall' search	Identifies any searches to retrieve all Entity Rule Objects for a given rule class. The precedent ID refers to the name of the rule class sought by the `readall` expression.	A precedent change item for the rule class will be written to a precedent change set if: • a new entity row is inserted into the database; • an existing entity row is removed from the database; • the data on the entity row changes in such a way that it is now considered logically deleted, or ceases to be considered logically deleted (if the logical deletion feature of the Entity data configuration is in use); and/or • the data on the entity row changes in such a way that it now passes the filter, or ceases to pass the filter (if the filter feature of the Entity data configuration is in use).

---

[5] This precedent change caters for an edge case whereby previously a CER expression had sought a row matching a primary key value, but none was found; but now a new row matching that primary key value is being inserted.

Name	When Identified	Trigger for Recalculation
'readall/match' search	Identifies any `readall/match` searches against Entity Rule Objects other than by primary key. The precedent ID refers to the name of the rule class sought by the `readall/match` expression, together with the attribute name and value used as the search criterion.	A precedent change item for the rule class and its attribute name and match value will be written to a precedent change set if, for the entity that is mapped to the rule class by the data configurations: • a new entity row is inserted into the database; • an existing entity row is removed from the database; • the data on the entity row changes in such a way that it is now considered logically deleted, or ceases to be considered logically deleted (if the logical deletion feature of the Entity data configuration is in use); • the data on the entity row changes in such a way that it now passes the filter, or ceases to pass the filter (if the filter feature of the Entity data configuration is in use); and/or • the value of the attribute's data on the entity row changes.
Rule Object Data Configurations	Identifies the use of the configuration for the Entity Rule Object Converter if any Entity Rule Object is accessed during the calculation.	If changes to the data configuration for the Entity Rule Object Converter are published, then a precedent change item for the converter's data configuration will be written to a precedent change set.

Propagation Processing

When a row of data changes for a table which is configured for Entity rule objects, then the Entity Rule Object Propagator requests the corresponding rule object and manipulates it in memory.

A rule object may be created, modified or removed, according to the nature of the change to the underlying database row (and taking into account configurations for logical deletions and/or filters).

The Entity Rule Object Propagator informs the Dependency Manager of any rows that have changed so that the Dependency Manager can determine the effects of those changes. Dependencies on entity data are stored at the entity row level, by recording a dependency on the entity's name and the row's key value.

Each database row may map to a number of target rule classes, according to the configurations for the Entity Rule Object Propagator held on the system. However, for the sake of clarity, the rest of this section describes the behavior of the Entity Rule Object Propagator in the situation where an entity is configured to be propagated to a single rule class only.

Support for Entity Operation Stereotypes

Cúram database tables are modeled as entities in the application model. Each modeled entity may contain several database operations, each with their own stereotype. Not all of these operations stereotypes can be reliably propagated. The table below shows the support for propagating data from invocations of methods for these stereotypes:

Table 28: Support for Entity Operation Stereotypes

Entity Operation Stereotype	Support in Entity Rule ObjectPropagator
(unstereotyped)	Ignored - no data written.
batchinsert	Supported. Each row in the batch will be propagated, if possible (as per nsinsert).
batchmodify	Supported[6]. Each row in the batch will be propagated, if possible (as per nsmodify).
insert	Supported.
modify	Supported.
nkmodify	Supported.
nkread	Ignored - no data written.
nkreadmulti	Ignored - no data written.
nkremove	Supported.
ns	For read operations, ignored. For write operations, not supported, as the operation can contain arbitrary SQL and in general it is not possible to reliably detect which database rows have been affected by the operation.
nsinsert	Supported, as long as the details being written to the database include a value for the primary key (i.e. no reliance on database-level key assignment).
nsmodify	Supported.
nsmulti	Ignored - no data written.
nsread	Ignored - no data written.
nsreadmulti	Ignored - no data written.
nsremove	Supported.
read	Ignored - no data written.
readmulti	Ignored - no data written.
remove	Supported.

Any operation that has the "no generated SQL" option set will not be supported by the rule object propagators included with the application.

---

[6] There is one edge-case which is not supported, which is where a batchmodify operation contains a number of modify operations, each of which change the primary key value of an entity, and which form a chain of changes to the same underlying row, e.g. the batchmodify contains two operations:

- a modify which changes a row's key from keyValue1 to keyValue2; and
- another modify which changes the same row's key from keyValue2 to keyValue3.

The net effect of these chained modify operations is to change the database row's key from keyValue1 to keyValue3. Such a change cannot be reliably propagated by the Entity Rule Object Propagator and should be avoided. (It is good practice to avoid creating modify operations that change a row's key anyway.)

If the `EntityRuleObjectPropagator` detects that an unsupported operation has occurred, then its behavior is governed by the value of the `curam.ruleobjectpropagation.nonpropagatableoperation.errorlevel` environment variable:

Table 29: Behavior when non-propagatable operations are invoked

Value of `curam.ruleobjectpropagation.n`	Behavior of Entity Rule Object Propagator	Comments
warn (default value)	The Entity Rule Object Propagator writes a warning to the application logs.	This is the default behavior and should be suitable for most environments.
ignore	The Entity Rule Object Propagator ignores the non-propagatable operation.	Consider using this setting if you have very many invocations of non-propagatable operations, and you already have in place procedures to identify and recalculate any dependents potentially affected.
error	The Entity Rule Object Propagator raises an exception with the details of the non-propagatable operation (which will typically result in the overall database transaction being rolled back).	Consider using this setting if you do not expect to have *any* non-propagatable operations.

If a non-propagatable operation occurs (and the Entity Rule Object Propagator is configured to warn the operator), then the operator should follow your procedures to identify and recalculate any dependents potentially affected.

The "Exclude" List for Entity Propagation

Some database tables are unsuitable for consideration by the Entity Rule Object Propagator because:

- they contain non-business data only;
- they have abnormally high numbers of data writes; and/or
- access to their data occurs before the propagation framework has been initialized.

The application allows components to register such database tables on an "exclude" list, and writes to these database tables are never considered for propagation to the Dependency Manager. The application's components may contribute to the exclude list, and customers may add further entries to the list by adding a binding inside a Guice module as follows:

```
import com.google.inject.AbstractModule;
import curam.core.sl.infrastructure.propagator.impl.
    RuleObjectDatabaseWriteListener.ExcludedTable;

public class MyModule extends AbstractModule {

  //
  @Override
  public void configure() {

    ...
    {
      // register excluded tables

      final Multibinder<ExcludedTable> excludedTables =
 Multibinder
          .newSetBinder(binder(), ExcludedTable.class);

      excludedTables
          .addBinding()
          .toInstance(
              new ExcludedTable(
                  curam.core.sl.infrastructure.assessment.intf.
                  MyEntity1.class));
      excludedTables
          .addBinding()
          .toInstance(
              new ExcludedTable(
                  curam.core.sl.infrastructure.assessment.intf.
                  MyEntity2.class));

    }
    ...

  }
}
```

For information on writing Guice modules, see the *Persistence Cookbook*.

Example

Let's say that a user creates a new rule set (`MyRuleSet`) with a `MyPerson` rule class. The user adds some rule attributes to the rule class named after some columns on the Person database table:

- `concernRoleID` (the primary key of the database table);
- `dateOfBirth`; and
- `dateOfBirthVerInd`.

The user creates a new configuration to propagate Person database rows to the new `MyRuleSet.MyPerson` rule class. There also continues to be an existing configuration (provided by the application) to propagate Person database rows to the `ParticipantEntitiesRuleSet.Person` rule class.

After a new row is inserted into the Person database table, then if a `MyRuleSet.MyPerson` rule object is requested for that person row during CER calculations, then the Entity Rule Object Converter will create a rule object in memory from the data on that row. The `MyRuleSet.MyPerson` rule object has its values for `concernRoleID`, `dateOfBirth` and `dateOfBirthVerInd` populated from the database row. It does not have any value for `dateOfDeath` because there is no such rule attribute on the `MyPerson` rule class.

By contrast, if a `ParticipantEntitiesRuleSet.Person` rule object is requested for that person row during CER calculations, then the Entity Rule Object Converter will populate the `ParticipantEntitiesRuleSet.Person` rule object from the same underlying database row. `ParticipantEntitiesRuleSet.Person` does have an attribute for `dateOfDeath` but not for `dateOfBirthVerInd`. Both rule classes have a `concernRoleID` rule attribute, which is required because `concernRoleID` is the primary key of the Person database table.

If the person row is subsequently modified on the database, then the Entity Rule Object Propagator notifies the Dependency Manager that the row has changed, and the Dependency Manager identifies all case determination results that depend on that row and requests that those cases be reassessed.

Let's say there are rules which require access to a claimant's personal details, and also requires access to details of other people related to the claimant, such as spouses and relatives. The rules would:

- use the claimant's `concernRoleID` to search for the `ParticipantEntitiesRuleSet.Person` rule object by `concernRoleID` (a primary key search) and access the claimant's personal details;
- use the claimant's `concernRoleID` to search for `ParticipantEntitiesRuleSet.ConcernRoleRelationship` rule objects by `concernRoleID` (not a primary key search); and
- for each `ParticipantEntitiesRuleSet.ConcernRoleRelationship` found, use the `relConcernRoleID` to search for the `ParticipantEntitiesRuleSet.Person` rule object by `concernRoleID` (a primary key search) and access the related person's details.

Claimant Joe (concernRoleID 392) has relationships to wife Mary (concernRoleID 393) and brother Frank (concernRoleID 394) stored as concern role relationships (concernRoleRelationshipIDs 773 and 774 respectively). These details are retrieved during the eligibility/entitlement calculation for Joe's case (caseID 453).

The Engine invokes the CER utility to identify these dependencies (which are stored using the Dependency Manager):

Table 30: Example Dependency Storage for Entity Rule Objects

Dependent		Precedent
Case 453's Entitlement	depends on	Entity row from table 'Person' where attribute 'concernRoleID' has value '392'
Case 453's Entitlement	depends on	<readall>/<match> expression against rule attribute 'ParticipantEntitiesRuleSet.ConcernRoleRelationship.concernRoleID matching value 392
Case 453's Entitlement	depends on	Entity row from table 'ConcernRoleRelationship' where attribute 'concernRoleRelationshipID' has value '773'
Case 453's Entitlement	depends on	Entity row from table 'Person' where attribute 'concernRoleID' has value '393'
Case 453's Entitlement	depends on	Entity row from table 'ConcernRoleRelationship' where attribute 'concernRoleRelationshipID' has value '774'
Case 453's Entitlement	depends on	Entity row from table 'Person' where attribute 'concernRoleID' has value '394'
Case 453's Entitlement	depends on	Data configuration for conversion of Entity rule objects

Active Succession Set Rule Objects

Overview

The Active Succession Set Rule Object Converter is responsible for converting a succession set of active evidence records into a CER rule object.

The Active Succession Set Rule Object Propagator is responsible for listening for changes to active evidence records and for notifying the Dependency Manager that active evidence data for succession set rule objects has changed.

> **Important:** Do not confuse the Active Succession Set Rule Object Converter and Propagator (suitable for use in eligibility/entitlement processing) with the Active/In-Edit Succession Set Rule Object Converter and Propagator class (which is *not* suitable for eligibility/entitlement processing, and thus not described in this guide - see the *Cúram Advisor Configuration Guide*).

Evidence (see *Designing Cúram Evidence Solutions*) allows developers to store records of evidence that can change over time. When circumstances change in the real world, a user can record those changes in the system by "succeeding" an earlier version of evidence. These versions of evidence make up a "succession set" which describe the history of some real-world evidence.

By contrast, in CER rules it is typically far easier to treat each piece of changeable evidence as one rule object, which has timeline-based attributes (see "4.5 Handling Data that Changes Over Time" in the *Cúram Express Rules Reference Manual*). The Active Succession Set Rule Object Converter automates the conversion of a succession set of evidence rows into a single rule object with a mixture of timeline and non-timeline attributes.

Changes to evidence go through an edit/activate lifecycle. Only evidence changes which have been activated are considered by the Active Succession Set Rule Object Converter; any pending additions, changes or removals are ignored.

Each rule class targeted by the Active Succession Set Rule Object Converter must extend the `PropagatorRuleSet.ActiveSuccessionSet` rule class included by the Engine. This rule class contains a `successionID` rule attribute which is used as a unique identifier, since the rule object represents the entire succession set of evidence.

Both dynamic and non-dynamic evidence types can be used with the Active Succession Set Rule Object Converter.

The Active Succession Set Rule Object Converter also populates relationships between rule objects for parent and child succession sets, if required.

Configuration

This converter and propagator share a common set of configuration data, and accept configurations which adhere to the following structure:

- propagator type must be "ROPT2005" (the code for 'Active succession set' from the `RuleObjectPropagatorType` code table);
- each evidence type to be converted or propagated must be listed in an `evidence` element with a type exactly matching the evidence's type from the `EvidenceType` code table; and
- each conversion/propagation target must be listed as a `ruleset` element (within the `evidence` element), specifying the name of the rule set to target and optionally the rule class (if the name of the rule class differs from that of the database table).

Configurations are cumulative, i.e. there may be many configurations of type "ROPT2005", and if an evidence type is present in any of those configurations then the evidence type will be converted and propagated; otherwise, the evidence type will be ignored.

The following types of configuration problems will be detected by the Active Succession Set Rule Object Converter:

- Evidence type not specified in the `evidence` element;
- The evidence type with the specified type code could not be found;
- The targeted rule class does not extend the `PropagatorRuleSet.ActiveSuccessionSet` rule class; and
- A rule class is targeted by more than one source evidence type.

Any configuration problems detected will be processed according to Data Configuration Problems on page 139.

Conversion Processing

Each evidence type may map to a number of target rule classes, according to the configurations for the Active Succession Set Rule Object Converter held on the system. However, for the sake of clarity, the rest of this section describes the behavior of the Active Succession Set Rule Object Converter in the situation where an evidence type is mapped to a single rule class only.

When an Active Succession Set Rule Object is requested during a CER calculation, the Active Succession Set Rule Object Converter is invoked to populate that rule object. The Active Succession Set Rule Object Converter will retrieve all the active evidence rows in the succession set and use them to populate the attribute values on the rule object.

The values of the evidence fields are used to map to identically-named rule attributes on the rule class. Any evidence field without a corresponding rule attribute is ignored. Evidence fields are defined by:

- **Dynamic evidence**

  The evidence fields available are those defined by the dynamic evidence metadata for the evidence type (see the *Cúram Dynamic Evidence Configuration Guide*); and
- **Non-dynamic evidence**

  The evidence fields available are those defined on the evidence-specific database table modeled for the static evidence type (see the *Cúram Evidence Generator* guide).

When populating a particular attribute value on a rule object, the behavior of the Active Succession Set Rule Object Converter differs according to whether the rule attribute's type is a Timeline. The Active Succession Set Rule Object Converter also contains special processing to populate rule attributes to point to rule objects for related succession sets.

The following sections detail how the Active Succession Set Rule Object Converter populates timeline and non-timeline data, and relationship data, along with some useful rule attributes inherited from the `ActiveSuccessionSet` rule class.

Timeline-based data types

If the data type of the attribute is `Timeline<` *some data type* `>`, then the Active Succession Set Rule Object Converter allows the possibility of the evidence value to differ (across different evidence versions in the succession set).

The identification of the "lifetime" of the succession set depends on the use of the `SuccessionSetPopulation` annotation on the rule class (see `SuccessionSetPopulation` on page 234).

The Active Succession Set Rule Object Converter will form a timeline value from the evidence versions as follows:

- **Calculate the start of the evidence lifetime**

  The Active Succession Set Rule Object Converter inspects the `SuccessionSetPopulation` annotation on the rule class to get the name of the attribute which the rule designer has designated as holding the start date of the evidence's lifetime. If no such rule attribute has been designated, or the rule attribute holds a blank value, then the evidence is deemed to have started from the beginning of time.
- **Calculate the end of the evidence lifetime**

  The Active Succession Set Rule Object Converter inspects the inspects the `SuccessionSetPopulation` annotation on the rule class to get the name of the attribute which the rule designer has designated as holding the end date of the evidence's lifetime. If no such rule attribute has been designated, or the rule attribute holds a blank value, then the evidence is deemed to continue to exist until the end of time (i.e. "until further notice").
- **Accumulate the varying values for the timeline**

  The Active Succession Set Rule Object Converter uses the evidence versions in the succession set to work out which values apply from which dates. The succession set will typically hold a single EvidenceDescriptor with a blank effective date, and the value of the evidence field on this evidence version will be used from the start of the evidence lifetime. The other EvidenceDescriptor rows will each have a populated effective date (on which the change of circumstances occurred), and the value of the evidence field will be used from that date in the timeline.
- **Assemble the timeline**

  The Active Succession Set Rule Object Converter inspects the effective dates for the varying values. Any dates which occur outside of the lifetime start and end are discarded. For periods before the lifetime start and after the lifetime end (if any), a default value is used (see Supported Domain Types on page 143). A timeline value is built from the values and used to populate the rule attribute value.

  Note: As for all timelines, any contiguous values which are identical are combined into a single interval.

  This can occur when two neighboring versions of evidence have the same value for an evidence field - the second version of the evidence has been recorded because a *different* evidence field has changed value.

  See the example where two neighboring evidence rows have the same value for `amount`, because only the `employmentStatus` has changed.

Tip: Because only values from within the evidence lifetime are used, then typically the timeline value will change to the default value on the day *after* the end date for the evidence (if any).

Non-timeline data types

If the data type of the attribute is not `Timeline<` *`some data type`* `>`, then the Active Succession Set Rule Object Converter does not allow the possibility of the evidence value to differ (across different evidence versions in the succession set). Ordinarily each version of evidence in the succession set should bear the same data value for the evidence field, and this single data value will be used to populate the rule attribute value.

However, it is possible that for such fields, either the rules designer has made an incorrect assumption that the data should not change over time, and/or a case worker has not consistently applied a correction to data consistently across the versions in the succession set. In these circumstances, the value of the evidence field may be inconsistent across the versions in the succession set, and if so the Active Succession Set Rule Object Converter will:

- propagate the value from the evidence version with the *latest* effective date (on the grounds that this is the version that a case worker is most likely to have maintained); and
- write a warning to the application logs saying that inconsistent evidence data was found.

Tip: If you see many warnings in your application logs pertaining to inconsistent evidence data, you should revisit your evidence and/or rule class design to resolve the inconsistency.

It is especially important to investigate any warnings which pertain to evidence fields which are designated as the start or end date of your succession set's lifetime, as any errors in such dates can lead to unreliable timelines being used to populate CER attribute values.

For more information about whether data should be modeled as a Timeline or not, see "What Is Timeline Data?" in the *`Cúram Express Rules Reference Manual`*.

Population of relationships to rule objects for other succession sets

When a succession set of active evidence is converted into a rule object, then any rule attributes which are annotated with `relatedSuccessionSet` on page 236 will be automatically populated with rule objects for related succession sets:

- **parent**

  the attribute will be populated with the rule objects for the succession sets for the parent(s) of the evidence; or
- **child**

  the attribute will be populated with the rule objects for the succession sets for the children of the evidence.

The type of the related evidence is identified from the type of the attribute, which can either be a rule class (extending `ActiveSuccessionSet`) or a list of such rule classes. The behavior of the Active Succession Set Rule Object Converter differs according to whether a list is used:

Table 31: Population of related `ActiveSuccessionSet` rule objects

Number of related versions found	Value propagated when attribute type is a rule class	Value propagated when attribute type is a List<rule class>
0	null	Empty list

Number of related versions found	Value propagated when attribute type is a rule class	Value propagated when attribute type is a List<rule class>
1	The rule object for the related succession set found	A list with a single item (the rule object for the related succession set found)
many	(An exception is thrown at population time)	A list with the rule objects for all the related succession sets found

Note: Typically in evidence design, a parent evidence item has 0, 1 or many child items (of a given type), while each child evidence item relates to exactly 1 parent.

However, the evidence infrastructure does not impose any constraints in this area, and so on occasion you may encounter child evidence types which relate to *multiple* parent types, and/or parent evidence types which expect only a *single* child evidence type.

Tip: The relationships between parent and child rule objects are not stored as timelines - a relationship between a parent and child rule object holds "for all time", even outside the parent or child's lifetime.

Typically any rules for the parent rule object, which retrieve the related children, will make reference to the lifetime of the child rule object (i.e. will use the value of the child's `exists` timeline (see Conversion Processing on page 117).

Rule attributes inherited from `ActiveSuccessionSet`

Each rule class targeted by the Active Succession Set Rule Object Converter must ultimately extend the `PropagatorRuleSet.ActiveSuccessionSet` rule class, and so will inherit the following rule attributes:

- **successionID**

  Populated from the successionID value on the EvidenceDescriptor rows, and used to uniquely identify the rule object (amongst other rule objects of the same rule class);
- **caseID**

  Populated from the caseID value on the EvidenceDescriptor row. If the evidence relates to an integrated case, the case ID will be that of an integrated case; if the evidence relates to a product delivery case, the case ID will be that of the particular product delivery that holds the evidence;
- **description**

  Contains a default rule to derive a description for the succession set rule object; sub-classes are free to override this `description` if required;
- **exists**

  A Boolean timeline which indicates the period of time for which the succession set rule object "exists", i.e. true for the dates between the designated start and end dates (inclusive), and false for dates before the start of the lifetime or after its end, if any; and
- **evidenceDescriptorID**

  A Number timeline, populated from the evidenceDescriptorID value on the EvidenceDescriptor rows which make up the succession set. The values vary according to the evidence row "in effect" at various points along the lifetime of the succession set rule object. Each value uniquely identifies the active EvidenceDescriptor row which contains the source of the data in effect on a particular date on the timeline-based attributes on the rule object. Note that these values will change when an evidence correction is activated, because at that point a different evidence row becomes an active member of the succession set.

Tip: The `exists` rule attribute can be useful for certain eligibility/entitlement calculations.

For example, if a particular objective is attained whenever one of the parents of a minor is absent from the household, and periods of Absence are recorded as succession sets of evidence (with start and end dates naturally mapping to the start and end of the period of absence), then the value of the objective's `isEntitledTimeline` is (in pseudo-code, using appropriate timeline operations):

- Get the periods of absence for the minor's parents;
- Entitled when *any*:
  - (iterate through the periods of absence)
    - the current period of absence "exists"

Handling of in-edit evidence changes

In general, the Active Succession Set Rule Object Converter ignores in-edit pending changes to evidence.

However, during manual determinations using in-edit evidence, the Active Succession Set Rule Object Converter supports a special processing mode to allow in-edit pending changes to be taken into account. See Temporary Access to In-Edit Evidence Changes on page 141 for more details.

Restrictions on Access

In your CER rule sets you will use CER's `<readall>/<match>` expression to access rule objects converted from active succession set data.

You may only specify the `retrievedattribute` to be the `caseID`.

If you attempt to specify a `retrievedattribute` to be the name of any other attribute, then the Active Succession Set Rule Object Converter will throw a runtime exception when the CER `<readall>/<match>` expression is executed.

Tip: If you require only some of the active evidence row evidence of a given type for a case, then consider wrapping the `<readall>/<match>` expression within a `<filter>` expression to return only the data you require, e.g. use `<readall>/<match>` matching on `caseID` to find all the `Income` active succession set rule objects for a case, and then use a `<filter>` to restrict the rule objects to just those for a particular member of the case.

You may specify the `ruleset` and `ruleclass` for the `<readall>` expression to be a rule class mapped by the data configuration. If you attempt to specify a rule class which is not directly mapped (e.g. a base rule class that you have created from which your concrete rule classes inherit) then no rule objects will be found.

Important: Do not use a `<readall>` without a `<match>`.

Such an unqualified `<readall>` would typically retrieve a very large number of rule objects and no dependency on the overall set of rule objects will be stored.

Precedents Identified

If Active Succession Set Rule Objects are accessed during a CER calculation, and the CER utility is used to identify precedents, then the following precedents will be identified:

Table 32: Precedents Identified for Active Succession Set Rule Objects

Name	When Identified	Trigger for Recalculation
Active Evidence	Identifies any case for which[7]: • a search was executed to retrieve Active Succession Set Rule Objects; and/or • one or more attribute values were accessed for one or more Active Succession Set Rule Objects for the case's evidence The precedent ID refers to the caseID which owns the evidence that was accessed.	If in-edit evidence changes for a case are activated, then a precedent change item for the case will be written to a precedent change set.
Rule Object Data Configurations	Identifies the use of the configuration for the Active Succession Set Rule Object Converter if any Active Succession Set Rule Object is accessed during the calculation.	If changes to the data configuration for the Active Succession Set Rule Object Converter are published, then a precedent change item for the converter's data configuration will be written to a precedent change set.

Propagation Processing

When evidence changes are applied for an evidence type that is configured for Active Succession Set Rule Objects, then the Active Succession Set Rule Object Propagator listens to internal events from the Evidence Controller, requests the corresponding rule object and manipulates it in memory.

A rule object may be created, modified or removed, according to whether evidence is being activated for the first time, is undergoing corrections or changes of circumstances, or is being canceled.

The Active Succession Set Rule Object Propagator informs the Dependency Manager of active evidence data that has changed so that the Dependency Manager can determine the effects of those changes. Dependencies on active evidence are stored at the case level, by recording a dependency on the caseID of the case that owns the evidence.

---

[7] In practice these two conditions amount to the same thing - that Active Succession Set Rule Objects for the case's evidence were accessed in some way. Generally, a search will be executed to retrieve rule objects in order that one or more attribute values can be accessed on those rule objects anyway.

Example

Let's say that a person's Income from an employment is modeled as temporal evidence. The income starts when a person starts an employment, and ends if the employment is subsequently terminated. The name of the employer is constant throughout the period of income, because the designer of the evidence structure made a design decision that if a person moves from one job to another, then the first employment comes to an end and a separate employment starts.

Over the lifetime of an employment, the income amount (i.e. the per annum pay) can vary, as the employee receives pay rises. Similarly, but independently, the person can be employed on a permanent or temporary basis, and this "employment status" can change over the lifetime of the employment. It is possible for the income's amount to change on the same date as the employment status, but a change in income amount can occur without a change in employment status, and vice versa.

The evidence designer designs an Income evidence entity as follows:

- **startDate**

  The date that the income (i.e. the overall employment) started;
- **endDate**

  The date that the income (i.e. the overall employment) ended, if any;
- **employer**

  Identifier of the employer, constant throughout the income (see the design decision described above);
- **amount**

  The per-annum pay amount; and
- **employmentStatus**

  Code for whether the employment status is permanent or temporary.

A rules designer then models an new `Income` rule class, extending the `ActiveSuccessionSet` rule class, and adds rule attributes, identifying which have values that change over time (i.e those which should be allowed to vary across different records in the same succession set):

- Should be constant across records in the succession set:
  - `startDate`;
  - `endDate`;
  - `employer`; and
- Should be allowed to vary across records in the succession set:
  - `amount`; and
  - `employmentStatus`.

The rules designer also identifies which rule attributes identify the "lifetime" of the `Income` :

- `startDate`; and
- `endDate`;

and annotates the rule class to identify these rule attributes.

An administrator publishes the rule set changes, and then publishes a data configuration for Active Succession Set Rule Object Converter to map the `Income` evidence type to the new rule class.

A case worker records some new Income evidence (for an employment which started on 1st January 2000). Initially the evidence is "in edit".

The details stored on the data are as follows (not all evidence details are included here, only those of interest to the converter):

Table 33: Database Details Stored for New Evidence

Database Column	Evidence Version Record 1
EvidenceDescriptor.evidenceDescriptorID	978
EvidenceDescriptor.caseID	453
EvidenceDescriptor.successionID	376
EvidenceDescriptor.effectiveFrom	(blank)
Income.startDate	1st January 2000
Income.endDate	(blank)
Income.employer	Acme Ind.
Income.amount	$10,000
Income.employmentStatus	Temporary (code)

When evidence capture is complete, the case worker activates the evidence and activates the case.

During the calculation of the case's determination result, the Active Succession Set Rule Object Converter retrieves the succession set for the newly-activated Income evidence and populates a rule object for it, with values as follows:

Table 34: Active Succession Set Rule Object after Initial Activation of Evidence

Rule Attribute Name	Value
ActiveSuccessionSet.description	"Income, successionID 376"
ActiveSuccessionSet.caseID	453
ActiveSuccessionSet.successionID	376
ActiveSuccessionSet.exists	Timeline: • Beginning of time - 31st December 1999: false • 1st January 2000 - End of time: true
ActiveSuccessionSet.evidenceDescriptorID	Timeline: • Beginning of time - 31st December 1999: 0 (default) • 1st January 2000 - End of time: 978
Income.startDate	1st January 2000
Income.endDate	(blank)
Income.employer	Acme Ind.
Income.amount	Timeline: • Beginning of time - 31st December 1999: $0 (default) • 1st January 2000 - End of time: $10,000

Rule Attribute Name	Value
Income.employmentStatus	Timeline: • Beginning of time - 31st December 1999: (blank) (default) • 1st January 2000 - End of time: Temporary

The Engine invokes the CER utility to identify these dependencies (which are stored using the Dependency Manager):

Table 35: Example Dependency Storage for Active Succession Set Rule Objects

Dependent		Precedent
Case 453's Entitlement	depends on	Active Evidence for case 453
Case 453's Entitlement	depends on	Data configuration for conversion of Active Succession Set rule objects

Over time, real-world circumstances change:

- on 1st January 2001, the income amount increases; and
- on 1st May 2002, the employment status changes from "temporary" to "permanent".

The agency is informed of these evidence changes and a case worker records new versions of the Income evidence, leading to the system storing new EvidenceDescriptor / Income pairs of rows for the evidence data effective from each change date:

Table 36: Database Details Stored for Changes of Circumstances

Database Column	Evidence Version Record 1	Evidence Version Record 2	Evidence Version Record 3
EvidenceDescriptor.evidenceDescriptorID	978	979	980
EvidenceDescriptor.caseID	453	453	453
EvidenceDescriptor.successionID	376	376	376
EvidenceDescriptor.effectiveFrom	(blank)	1st January 2001	1st May 2002
Income.startDate	1st January 2000	1st January 2000	1st January 2000
Income.endDate	(blank)	(blank)	(blank)
Income.employer	Acme Ind.	Acme Ind.	Acme Ind.
Income.amount	$10,000	$12,000	$12,000
Income.employmentStatus	Temporary (code)	Temporary (code)	Permanent (code)

Note how each version of the evidence shows a snapshot of all the evidence data as it was at a point in time. A different version of the evidence must be stored for a date on which *any* of the data items change on that evidence.

When the case worker activates the change-of-circumstances evidence changes, the Evidence Controller notifies the Active Succession Set Rule Object Propagator of the evidence changes, and the Active Succession Set Rule Object Propagator in turn notifies the Dependency Manager that evidence for a case has changed. The Dependency Manager identifies the product delivery case that depends on the changed evidence and requests that the Engine reassesses the case. During reassessment, the Engine invokes CER to calculate the determination result, and as part of this calculation the Active Succession Set Rule Object Converter is called upon to populate the rule objects for the changed evidence, which are populated as follows:

Table 37: Active Succession Set Rule Objects after Changes of Circumstances

Rule Attribute Name	Value
ActiveSuccessionSet.description	"Income, successionID 376"
ActiveSuccessionSet.caseID	453
ActiveSuccessionSet.successionID	376
ActiveSuccessionSet.exists	Timeline: • Beginning of time - 31st December 1999: false • 1st January 2000 - End of time: true
ActiveSuccessionSet.evidenceDescriptorID	Timeline: • Beginning of time - 31st December 1999: 0 (default) • 1st January 2000 - 31st December 2000: 978 • 1st January 2001 - 30th April 2001: 979 • 1st May 2001 - End of time: 980
Income.startDate	1st January 2000
Income.endDate	(blank)
Income.employer	Acme Ind.
Income.amount	Timeline: • Beginning of time - 31st December 1999: $0 (default) • 1st January 2000 - 31st December 2000: $10,000 • 1st January 2001 - End of time: $12,000
Income.employmentStatus	Timeline: • Beginning of time - 31st December 1999: (blank) (default) • 1st January 2000 - 30th April 2001: Temporary • 1st May 2001 - End of time: Permanent

Note how, in contrast to the way evidence versions are stored, each data item that can vary over time has its own timeline independently of other data items which may or may not change value on the same dates.

On 30th June 2002, the employment comes to an end and a case worker records the end date on the latest record in the succession set:

Table 38: Database Details Stored for Ended Evidence

Database Column	Evidence Version Record 1	Evidence Version Record 2	Evidence Version Record 3
EvidenceDescriptor.evidenceDescriptorID	978	979	981
EvidenceDescriptor.caseID	453	453	453
EvidenceDescriptor.successionID	376	376	376
EvidenceDescriptor.effectiveFrom	(blank)	1st January 2001	1st May 2002
Income.startDate	1st January 2000	1st January 2000	1st January 2000
Income.endDate	(blank)	(blank)	30th June 2002
Income.employer	Acme Ind.	Acme Ind.	Acme Ind.
Income.amount	$10,000	$12,000	$12,000
Income.employmentStatus	Temporary (code)	Temporary (code)	Permanent (code)

The case worker activates the changes, which causes the existing latest EvidenceDescriptor / Income pair to become "superseded" (evidenceDescriptorID 980) and a new pair to become "active" (evidenceDescriptorID 981).

Again the Active Succession Set Rule Object Propagator causes the case to be reassessed on foot of the evidence changes. During reassessment the Active Succession Set Rule Object Converter populates the rule object with these values:

Table 39: Active Succession Set Rule Objects after Evidence Ended

Rule Attribute Name	Value
ActiveSuccessionSet.description	"Income, correctionSetID 376"
ActiveSuccessionSet.caseID	453
ActiveSuccessionSet.successionID	376
ActiveSuccessionSet.exists	Timeline: • Beginning of time - 31st December 1999: false • 1st January 2000 - 30th June 2002: true • 1st July 2002 - End of time: false

Rule Attribute Name	Value
ActiveSuccessionSet.evidenceDescriptorID	Timeline: • Beginning of time - 31st December 1999: 0 (default) • 1st January 2000 - 31st December 2000: 978 • 1st January 2001 - 30th April 2001: 979 • 1st May 2001 - 30th June 2002: 981 • 1st July 2002 - End of time: 0 (default)
Income.startDate	1st January 2000
Income.endDate	30th June 2002
Income.employer	Acme Ind.
Income.amount	Timeline: • Beginning of time - 31st December 1999: $0 (default) • 1st January 2000 - 31st December 2000: $10,000 • 1st January 2001 - 30th June 2002: $12,000 • 1st July 2002 - End of time: $0 (default)
Income.employmentStatus	Timeline: • Beginning of time - 31st December 1999: (blank) (default) • 1st January 2000 - 30th April 2001: Temporary • 1st May 2001 - 30th June 2002: Permanent • 1st July 2002 - End of time: (blank) (default)

At some time later, a review of the case finds that the entire history of the income has been recorded against the wrong person. All the evidence records for the Income are canceled by the case worker, and the evidence re-recorded against the correct person (in a new succession set). When the case is reassessed, the Active Succession Set Rule Object Converter does not populate a rule object for the evidence because now none of its succession set records are "active".

Some new legislation is introduced which affects how eligibility and entitlement must be calculated, and in order to comply with this legislation, the agency must now capture more details about periods of employment, specifically to capture details of the varying responsibilities that a person had during each employment. An employee may have several responsibilities at the same time during an employment, and each responsibility may begin and end independently of others.

An evidence designer models a new type of temporal evidence named `Responsibility`, which is a child evidence type of the `Income` evidence type:

- **income**

  The parent `Income` evidence of which the `Responsibility` evidence is a child;
- **type**

  Code for the type of responsibility (e.g. management, clerical tasks, financial control, etc.);
- **startDate**

  The date that the responsibility started; and
- **endDate**

  The date that the responsibility ended, if any.

A rules designer creates a new `Responsibility` rule class, and identifies that rules centered around the `Responsibility` will need to navigate to parent `Income` rule objects:

- `parentIncome`, of type `Income` , annotated to mark it to be populated from parent related succession sets; the attribute holds a single rule object of type `Income` , because each `Responsibility` succession set relates to exactly one parent `Income` succession set;
- `type`;
- `startDate`; and
- `endDate`.

The rules designer identifies that none of the `Responsibility` details undergo changes over time, but regardless chooses to use the Active Succession Set Rule Object features to get an automatically populated `exists` timeline for the `Responsibility` rule object (which will be referenced by new rules on the parent `Income` rule class). The rules designer accordingly annotates the `startDate` and `endDate` attributes on `Responsibility`(as these are the attributes which determine the `Responsibility`'s "lifetime").

The rules designer also identifies that rules centered around the `Income` will need to navigate to child `Responsibility` rule objects, and so adds a new rule attribute to the existing `Income` rule class:

- `childResponsibilities`, of type `List<Responsibility>`, annotated to mark it to be populated from child related succession sets; the attribute holds a list of rule objects of type `Income` , because each `Income` succession set may relate to 0, 1 or many child `Responsibility` succession sets

The evidence design changes, rules changes and new data configuration for the `Responsibility` rule class are published.

A case worker records details for an employment where there is a pay rise on 1st January 2005. From the start of the employment, the employee is responsible for clerical tasks, but from 1st July onwards, the employee is also responsible for financial tasks (in addition to still be responsible for clerical tasks).

When the case is assessed, the Active Succession Set Rule Object Converter populates these rule objects:

- A parent `Income` rule object, with an `amount` timeline showing the pay rise and with its `childResponsibilities` value set to be a list containing the two `Responsibility` rule objects below;

- A child `Responsibility` rule object for the clerical tasks, with its `parentIncome` value set to be the `Income` rule object above; and
- Another child `Responsibility` rule object for the financial tasks, with its `parentIncome` value also set to be the `Income` rule object above.

Note that there are two `Responsibility` rule objects, because there are two distinct real-world responsibilities, each stored as different succession sets. The single `Income` rule object, represents the changes in the `Income` evidence item over time (i.e. represents the single succession set of `Income` evidence).

Active Evidence Row Rule Objects

Overview

The Active Evidence Row Rule Object Converter is responsible for converting a row of active evidence into a CER rule object. Each active version of the evidence is converted to its own CER rule object (unlike the Active Succession Set Rule Object Converter which converts all active evidence rows from a single succession set into a single rule object).

Each rule class targeted by the Active Evidence Row Rule Object Converter must extend the `PropagatorRuleSet.ActiveEvidenceRow` rule class included by the Engine. This rule class contains a `correctionSetID` rule attribute which is used as a unique identifier, since in any correction set of evidence there can be at most one active record.

Both dynamic and non-dynamic evidence types can be used with the Active Evidence Row Rule Object Converter.

The Active Evidence Row Rule Object Converter also populates relationships between rule objects for parent and child evidence versions, if required.

> **Tip:** The Active Evidence Row Rule Object Converter is likely to be useful only for evidence which is not temporal in nature, and/or which does not use standard evidence facilities for recording real-world changes of circumstances, because any CER eligibility/entitlement rules which handle changes of circumstances will typically need to manipulate the separate rule objects into timelines.
>
> If you have evidence which does use standard evidence facilities, see Active Succession Set Rule Objects on page 115 instead.

Configuration

The converter and propagator share a common set of configuration data, and accept configurations which adhere to the following structure:

- propagator type must be "ROPT2004" (the code for 'Active evidence row' from the `RuleObjectPropagatorType` code table);
- each evidence type to be converted or propagated must be listed in an `evidence` element with a type exactly matching the evidence's type from the 'EvidenceType' code table; and
- each conversion/propagation target must be listed as a `ruleset` element (within the `evidence` element), specifying the name of the rule set to target and optionally the rule class (if the name of the rule class differs from that of the database table).

Configurations are cumulative, i.e. there may be many configurations of type "ROPT2004", and if an evidence type is present in any of those configurations then the evidence type will be converted and propagated; otherwise, the evidence type will be ignored.

The following types of configuration problems will be detected by the Active Evidence Row Rule Object Converter/Propagator:

- Evidence type not specified in the `evidence` element;
- The evidence type with the specified type code could not be found;
- The targeted rule class does not extend the `PropagatorRuleSet.ActiveEvidenceRow` rule class; and
- A rule class is targeted by more than one source evidence type.

Any configuration problems detected will be processed according to Data Configuration Problems on page 139.

Conversion Processing

Each evidence type may map to a number of target rule classes, according to the configurations for the Active Evidence Row Rule Object Converter held on the system. However, for the sake of clarity, the rest of this section describes the behavior of the Active Evidence Row Rule Object Converter in the situation where an evidence type is mapped to a single rule class only.

When an Active Evidence Row Rule Object is requested during a CER calculation, the Active Evidence Row Rule Object Converter is invoked to populate that rule object. The Active Evidence Row Rule Object Converter will retrieve the active row for the evidence's correction set and use it populate the attribute values on the rule object.

The values of the evidence fields are used to map to identically-named rule attributes on the rule object. Any evidence field without a corresponding rule attribute is ignored. Evidence fields are defined by:

- **Dynamic evidence**

  The evidence fields available are those defined by the dynamic evidence metadata for the evidence type (see the *`Cúram Dynamic Evidence Configuration Guide`*); and
- **Non-dynamic evidence**

  The evidence fields available are those defined on the evidence-specific database table modeled for the static evidence type (see the *`Cúram Evidence Generator`* guide).

Population of relationships to rule objects for other evidence rows

When a row of active evidence is converted to a rule object, then any rule attributes which are annotated with `relatedEvidence` on page 235 will be automatically populated with rule objects for related evidence versions:

- **parent**

  the attribute will be populated with the rule objects for the version(s) for the parent evidence item(s) of the evidence version; or
- **child**

  the attribute will be populated with the rule objects for the version(s) for the child evidence item(s) of the evidence version.

The type of the related evidence is identified from the type of the attribute, which can either be a rule class (extending `ActiveEvidenceRow`) or a list of such rule classes. The behavior of the Active Evidence Row Rule Object Converter differs according to whether a list is used:

Table 40: Propagation of related `ActiveEvidenceRow` rule objects

Number of related versions found	Value populated when attribute type is a rule class	Value populated when attribute type is a List<rule class>
0	null	Empty list
1	The rule object for the related instance found	A list with a single item (the rule object for the related instance found)
many	(An exception is thrown at propagation time)	A list with the rule objects for all the related instances found

> **Important:** Remember that the related `ActiveEvidenceRow` rule objects are each a *version* of evidence.
>
> Even if a real-world child object can only have one real-world parent, if that parent has data that changes over time, then each child version may relate to many parent versions.
>
> For a rule attribute that holds related parent or child rule objects, you should model that rule attribute as a list of rule classes, unless you can guarantee that there will only ever be one active version of the related parent/child evidence (which would generally only be the case if the related evidence type does not store data which can undergo a change of circumstances).

Rule attributes inherited from `ActiveEvidenceRow`

Each rule class targeted by the Active Evidence Row Rule Object Converter must ultimately extend the `PropagatorRuleSet.ActiveEvidenceRow` rule class, and so will inherit the following rule attributes:

- **correctionSetID**

  Populated from the correctionSetID value on the EvidenceDescriptor row, and used to uniquely identify the rule object (amongst other rule objects of the same rule class);
- **caseID**

  Populated from the caseID value on the EvidenceDescriptor row. If the evidence relates to an integrated case, the case ID will be that of an integrated case; if the evidence relates to a product delivery case, the case ID will be that of the particular product delivery that holds the evidence;
- **description**

  Contains a default rule to derive a description for the evidence rule object; sub-classes are free to override this `description` if required;
- **effectiveDate**

  Populated from the effectiveFrom value on the EvidenceDescriptor row; will be null for evidence effective from the start of the case;
- **evidenceDescriptorID**

  Populated from the evidenceDescriptorID value on the EvidenceDescriptor row; uniquely identifies the active EvidenceDescriptor row which contains the source of the data on the rule object. Note that this value will change when an evidence correction is activated, because at that point a different evidence row becomes the only active row in the correction set; and
- **successionID**

Populated from the successionID value on the EvidenceDescriptor row.

Handling of in-edit evidence changes

In general, the Active Evidence Row Rule Object Converter ignores in-edit pending changes to evidence.

However, during manual determinations using in-edit evidence, the Active Evidence Row Rule Object Converter supports a special processing mode to allow in-edit pending changes to be taken into account. See Temporary Access to In-Edit Evidence Changes on page 141 for more details.

Restrictions on Access

In your CER rule sets you will use CER's `<readall>/<match>` expression to access rule objects converted from active evidence row data.

You may only specify the `retrievedattribute` to be the `caseID`.

If you attempt to specify a `retrievedattribute` to be the name of any other attribute, then the Active Evidence Row Rule Object Converter will throw a runtime exception when the CER `<readall>/<match>` expression is executed.

Tip: If you require only some of the active evidence row evidence of a given type for a case, then consider wrapping the `<readall>/<match>` expression within a `<filter>` expression to return only the data you require, e.g. use `<readall>/<match>` matching on `caseID` to find all the `Income` active evidence row rule objects for a case, and then use a `<filter>` to restrict the rule objects to just those for a particular member of the case.

You may specify the `ruleset` and `ruleclass` for the `<readall>` expression to be a rule class mapped by the data configuration. If you attempt to specify a rule class which is not directly mapped (e.g. a base rule class that you have created from which your concrete rule classes inherit) then no rule objects will be found.

Important: Do not use a `<readall>` without a `<match>`.

Such an unqualified `<readall>` would typically retrieve a very large number of rule objects and no dependency on the overall set of rule objects will be stored.

Precedents Identified

If Active Evidence Row Rule Objects are accessed during a CER calculation, and the CER utility is used to identify precedents, then the following precedents will be identified:

Table 41: Precedents Identified for Active Evidence Row Rule Objects

Name	When Identified	Trigger for Recalculation
Active Evidence	Identifies any case for which[8]: • a search was executed to retrieve Active Evidence Row Rule Objects; and/or • one or more attribute values were accessed for one or more Active Evidence Row Rule Objects for the case's evidence The precedent ID refers to the caseID which owns the evidence that was accessed.	If in-edit evidence changes for a case are activated, then a precedent change item for the case will be written to a precedent change set.
Rule Object Data Configurations	Identifies the use of the configuration for the Active Evidence Row Rule Object Converter if any Active Evidence Row Rule Object is accessed during the calculation.	If changes to the data configuration for the Active Evidence Row Rule Object Converter are published, then a precedent change item for the converter's data configuration will be written to a precedent change set.

Propagation Processing

When evidence changes are applied for an evidence type that is configured for Active Evidence Row Rule Objects, then the Active Evidence Row Rule Object Propagator listens to internal events from the Evidence Controller, requests the corresponding rule object and manipulates it in memory.

A rule object may be created, modified or removed, according to whether evidence is being activated for the first time, is undergoing corrections or changes of circumstances, or is being canceled.

The Active Evidence Row Rule Object Propagator informs the Dependency Manager of active evidence data that has changed so that the Dependency Manager can determine the effects of those changes. Dependencies on active evidence are stored at the case level, by recording a dependency on the caseID of the case that owns the evidence.

Example

Let's say that a person's Income from an employment is modeled as evidence. The income starts when a person starts an employment, and ends if the employment is subsequently terminated. (This example is intentionally similar to that for Active Succession Set Rule Objects on page 115.)

Over the lifetime of an employment, the income amount (i.e. the per annum pay) can vary, as the employee receives pay rises. Similarly, but independently, the person can be employed on a permanent or temporary basis, and this "employment status" can change over the lifetime of the employment. It is possible for the income's amount to change on the same date as the employment status, but a change in income amount can occur without a change in employment status, and vice versa.

---

[8] In practice these two conditions amount to the same thing - that Active Evidence Row Rule Objects for the case's evidence were accessed in some way. Generally, a search will be executed to retrieve rule objects in order that one or more attribute values can be accessed on those rule objects anyway.

The evidence designer designs an Income evidence entity as follows:

- **startDate**

  The date that the income (i.e. the overall employment) started;
- **endDate**

  The date that the income (i.e. the overall employment) ended, if any;
- **employer**

  Identifier of the employer;
- **amount**

  The per-annum pay amount; and
- **employmentStatus**

  Code for whether the employment status is permanent or temporary.

A rules designer then models an new `Income` rule class, extending the `ActiveEvidenceRow` rule class, and adds rule attributes:

- `startDate`;
- `endDate`;
- `employer`;
- `amount`; and
- `employmentStatus`.

An administrator publishes the rule set changes, and then publishes a data configuration for Active Evidence Row Rule Object Converter and Propagator to map the `Income` evidence type to the new rule class.

A case worker records some new Income evidence (for a temporary employment which started on 1st January 2000, salary $10,000). Initially the evidence is "in edit".

The details stored on the data are as follows (not all evidence details are included here, only those of interest to the converter):

Table 42: Database Details Stored for New Evidence

Database Column	Evidence Version Record 1
EvidenceDescriptor.evidenceDescriptorID	978
EvidenceDescriptor.caseID	453
EvidenceDescriptor.correctionSetID	476
EvidenceDescriptor.effectiveFrom	(blank)
Income.startDate	1st January 2000
Income.endDate	(blank)
Income.employer	Acme Ind.
Income.amount	$10,000
Income.employmentStatus	Temporary (code)

When evidence capture is complete, the case worker activates the evidence and activates the case.

During the calculation of the case's determination result, the Active Evidence Row Rule Object Converter retrieves the data for the newly-activated Income evidence and populates a rule object for it, with values as follows:

Table 43: Active Evidence Row Rule Object after Initial Activation of Evidence

Rule Attribute Name	Value for Rule Object 1
ActiveEvidenceRow.description	"Income, correctionSetID 476"
ActiveEvidenceRow.caseID	453
ActiveEvidenceRow.correctionSetID	476
ActiveEvidenceRow.evidenceDescriptorID	978
ActiveEvidenceRow.effectiveDate	(blank)
Income.startDate	1st January 2000
Income.endDate	(blank)
Income.employer	Acme Ind.
Income.amount	$10,000
Income.employmentStatus	Temporary

The Engine invokes the CER utility to identify these dependencies (which are stored using the Dependency Manager):

Table 44: Example Dependency Storage for Active Evidence Row Rule Objects

Dependent		Precedent
Case 453's Entitlement	depends on	Active Evidence for case 453
Case 453's Entitlement	depends on	Data configuration for conversion of Active Evidence Row rule objects

Over time, real-world circumstances change:

- on 1st January 2001, the income amount increases; and
- on 1st May 2002, the employment status changes from "temporary" to "permanent".

The agency is informed of these evidence changes and a case worker records new versions of the Income evidence, leading to the system storing new EvidenceDescriptor / Income pairs of rows for the evidence data effective from each change date:

Table 45: Database Details Stored for Changes of Circumstances

Database Column	Evidence Version Record 1	Evidence Version Record 2	Evidence Version Record 3
EvidenceDescriptor.evidenceDescriptorID	978	979	980
EvidenceDescriptor.caseID	453	453	453
EvidenceDescriptor.correctionSetID	476	477	478
EvidenceDescriptor.effectiveDate	(blank)	1st January 2001	1st May 2002
Income.startDate	1st January 2000	1st January 2000	1st January 2000
Income.endDate	(blank)	(blank)	(blank)
Income.employer	Acme Ind.	Acme Ind.	Acme Ind.

Database Column	Evidence Version Record 1	Evidence Version Record 2	Evidence Version Record 3
Income.amount	$10,000	$12,000	$12,000
Income.employmentStatus	Temporary (code)	Temporary (code)	Permanent (code)

When the case worker activates the change-of-circumstances evidence changes, the Evidence Controller notifies the Active Evidence Row Rule Object Propagator of the evidence changes, and the Active Evidence Row Rule Object Propagator in turn notifies the Dependency Manager that evidence for a case has changed. The Dependency Manager identifies the product delivery case that depends on the changed evidence and requests that the Engine reassesses the case. During reassessment, the Engine invokes CER to calculate the determination result, and as part of this calculation the Active Evidence Row Rule Object Converter is called upon to populate the rule objects for the changed evidence, which are populated as follows:

Table 46: Active Evidence Row Rule Objects after Changes of Circumstances

Rule Attribute Name	Value for Rule Object 1	Value for Rule Object 2	Value for Rule Object 3
ActiveEvidenceRow.description	"Income, correctionSetID 476"	"Income, correctionSetID 477"	"Income, correctionSetID 478"
ActiveEvidenceRow.caseID	453	453	453
ActiveEvidenceRow.correctionSetID	476	477	478
ActiveEvidenceRow.evidenceDescriptorID	978	979	980
ActiveEvidenceRow.effectiveDate	(blank)	1st January 2001	1st May 2002
Income.startDate	1st January 2000	1st January 2000	1st January 2000
Income.endDate	(blank)	(blank)	(blank)
Income.employer	Acme Ind.	Acme Ind.	Acme Ind.
Income.amount	$10,000	$12,000	$12,000
Income.employmentStatus	Temporary	Temporary	Permanent

On 30th June 2002, the employment comes to an end and a case worker records the end date on the latest version of the evidence:

Table 47: Database Details Stored for Ended Evidence

Database Column	Evidence Version Record 1	Evidence Version Record 2	Evidence Version Record 3
EvidenceDescriptor.evidenceDescriptorID	978	979	981
EvidenceDescriptor.caseID	453	453	453
EvidenceDescriptor.correctionSetID	476	477	478
EvidenceDescriptor.effectiveFrom	(blank)	1st January 2001	1st May 2002
Income.startDate	1st January 2000	1st January 2000	1st January 2000
Income.endDate	(blank)	(blank)	30th June 2002
Income.employer	Acme Ind.	Acme Ind.	Acme Ind.
Income.amount	$10,000	$12,000	$12,000
Income.employmentStatus	Temporary (code)	Temporary (code)	Permanent (code)

The case worker activates the changes, which causes the existing latest EvidenceDescriptor / Income pair to become "superseded" (evidenceDescriptorID 980) and a new pair to become "active" (evidenceDescriptorID 981).

Again the Active Evidence Row Rule Object Propagator causes the case to be reassessed on foot of the evidence changes. During reassessment the Active Evidence Row Rule Object Converter populates the rule object with these values:

Table 48: Active Evidence Row Rule Objects after Evidence Ended

Rule Attribute Name	Value for Rule Object 1	Value for Rule Object 2	Value for Rule Object 3
ActiveEvidenceRow.description	"Income, correctionSetID 476"	"Income, correctionSetID 477"	"Income, correctionSetID 478"
ActiveEvidenceRow.caseID	453	453	453
ActiveEvidenceRow.correctionSetID	476	477	478
ActiveEvidenceRow.evidenceDescriptorID	978	979	981
ActiveEvidenceRow.effectiveDate	(blank)	1st January 2001	1st May 2002
Income.startDate	1st January 2000	1st January 2000	1st January 2000
Income.endDate	(blank)	(blank)	30th June 2002
Income.employer	Acme Ind.	Acme Ind.	Acme Ind.
Income.amount	$10,000	$12,000	$12,000
Income.employmentStatus	Temporary	Temporary	Permanent

At some time later, a review of the case finds that the entire history of the income has been recorded against the wrong person. All the evidence records for the Income are canceled by the case worker, and the evidence re-recorded against the correct person (in a new correction sets). When the case is reassessed, the Active Evidence Row Rule Object Converter does not populate rule objects for the evidence because now none of its evidence records are "active".

Some new legislation is introduced which affects how eligibility and entitlement must be calculated, and in order to comply with this legislation, the agency must now capture more details about periods of employment, specifically to capture details of the varying responsibilities that a person had during each employment. An employee may have several responsibilities at the same time during an employment, and each responsibility may begin and end independently of others.

An evidence designer models a new type of evidence named `Responsibility`, which is a child evidence type of the `Income` evidence type:

- **income**

  The parent `Income` evidence of which the `Responsibility` evidence is a child;
- **type**

  Code for the type of responsibility (e.g. management, clerical tasks, financial control, etc.);
- **startDate**

  The date that the responsibility started; and
- **endDate**

  The date that the responsibility ended, if any.

A rules designer creates a new `Responsibility` rule class, and identifies that rules centered around the `Responsibility` will need to navigate to parent `Income` rule objects:

- `parentIncomeVersions`, of type `List<Income>`, annotated to mark it to be populated from parent related evidence;
- `type`;
- `startDate`; and
- `endDate`.

The rules designer also identifies that rules centered around the `Income` will need to navigate to child `Responsibility` rule objects, and so adds a new rule attribute to the existing `Income` rule class:

- `childResponsibilityVersions`, of type `List<Responsibility>`, annotated to mark it to be populated from child related evidence;

The evidence design changes, rules changes and new data configuration for the `Responsibility` rule class are published.

A case worker records details for an employment where there is a pay rise on 1st January 2005. From the start of the employment, the employee is responsible for clerical tasks, but from 1st July onwards, the employee is also responsible for financial tasks (in addition to still be responsible for clerical tasks).

When the case is assessed, the Active Evidence Row Rule Object Converter populates these rule objects:

- A parent `Income` rule object effective from the start of the case, with an `amount` of $15,000 and with its `childResonsibilityVersions` value set to be a list containing the two `Responsibility` rule objects below;
- Another parent `Income` rule object effective from 1st January 2005, with an `amount` of $16,000 and with its `childResonsibilityVersions` value also set to be a list containing the two `Responsibility` rule objects below;
- A child `Responsibility` rule object for the clerical tasks, with its `parentIncomeVersions` value set to be a list containing the two `Income` rule object above; and
- Another child `Responsibility` rule object for the financial tasks, with its `parentIncomeVersions` value also set to be a list containing the two `Income` rule object above.

Note that there are two `Responsibility` rule objects, because there are two distinct real-world responsibilities, each stored as different succession sets. There are two `Income` rule objects, one for each version of the evidence as it changed over time.

Data Configuration Problems

For converters and propagators which are configurable, there may be problems detected in the data configurations.

The behavior of converters and propagators which encounter configuration problems is governed by the value of the `curam.ruleobjectpropagation.configuration.errorlevel` environment variable:

Table 49: Behavior when configuration problems are found

Value of `curam.ruleobjectpropagation.configuration.er`	**Behavior of configurable converters and propagators**
warn (default value)	The converter or propagator writes a warning to the application logs, and ignores the problematic configuration.
ignore	The converter or propagator ignores the problematic configuration.
error	The converter or propagator raises an exception with the details of the configuration problem, and does not allow processing (typically, application startup) to continue.

Configuration problems may be detected:

- whenever changes to configurations are published (see Rule Object Data Configurations on page 176); and/or
- when the data configurations are initially loaded, if configured to do so. The application property, `curam.ruleobjectpropagation.configuration.validateonload`, is used to dictate whether or not configurations are validated when they are initially loaded. The default value is 'NO'. If this value is set to 'YES' , then configurations will be validated when they are initially loaded, shortly after application start-up (as soon as database writes for non-excluded tables are detected). For optimized performance it is recommended that this value is set to 'NO'.

Data Access Points

This section gives an overview of the various points at which the rule object converters and propagators interact with data from the application database.

Normal Conversion

During normal processing, rule object converters are invoked by CER whenever CER is instructed to perform a search against rule objects.

CER looks up the appropriate rule object converter based on the rule class being searched, and invokes the rule object converter to gather the appropriate data from the application database and populate CER rule objects in memory, which can then be used in further CER calculations.

In particular, during normal processing, these converters access the active evidence records on the application database:

- Active Succession Set Rule Objects on page 115; and
- Active Evidence Row Rule Objects on page 130.

The retrieval of active evidence records is used when the Engine requests CER to calculate one of the following types of determination:

- an assessment determination (see Assessment Determinations on page 27);
- a snapshot determination (see Snapshot Determinations on page 27);
- a manual check determination *where the user has chosen the option to include active evidence only* (see Manual Check Determinations on page 27);

In contrast, the evidence converters also support a special data access mode to provide a calculation based off in-edit evidence data (see Temporary Access to In-Edit Evidence Changes on page 141).

Temporary Access to In-Edit Evidence Changes

The converters for these CER rule objects read data from evidence:

- Active Succession Set Rule Objects on page 115; and
- Active Evidence Row Rule Objects on page 130.

These converters support a special mode to allow the population of rule object data from in-edit evidence changes whenever a case worker requests a manual determination based on *in-edit pending changes* to one or more evidence items (see Manual Check Determinations on page 27). This is in contrast to normal processing (see Normal Conversion on page 140) whereby these converters have access to active evidence data only.

When the case worker requests a manual determination based on in-edit pending changes to evidence, then the Engine:

- starts a new CER session;
- instructs these converters to temporarily use in-edit evidence changes to populate rule objects during the session, i.e. to:
  - take into account any pending addition of new evidence;
  - take into account any pending modification to existing evidence; and
  - disregard evidence data for any pending removal of existing evidence; and
- requests the `determinationResult` value from CER. The calculation will invoke the rule object converters to access rule objects for evidence, which will take into account the in-edit evidence changes when populating those rule objects.

Incremental Propagation

During normal running of the application, the system detects changes to data which may have been used to populate CER rule objects, by listening for these internal events:

- changes to evidence, such as the activation of in-edit evidence changes; and
- changes to entity rows, for entities which are mapped in data configurations for the Entity Rule Object Converter.

The processing of these internal events as they occur is known as "incremental propagation". Incremental propagation is used to inform the Dependency Manager of changes to precedent data, so that the Dependency Manager can take care of identifying dependents to recalculation.

There are situations where incremental propagation cannot automatically detect changes to precedent data, namely:

- a non-propagatable data write operation is executed (see Propagation Processing on page 111); and/or
- data is written to the database outside the control of standard modeled entity operations, e.g. via an SQL script or another system connected to the application's database.

If either of these occur you must take manual steps to identify and reassess cases which may be potentially affected.

Bulk Maintenance of Rate Rule Objects

The Engine uses CER to store rule objects for rate table data on CER's database tables. These stored rule objects act as a "mirror" copy of the rate table data in a form that can be accessed during CER calculations.

The CER rule objects may not accurately reflect the latest rate table data for a number of reasons:

- An administrator has changed rates in the application but not yet applied the rate changes to CER.

  The administration application contains an "Apply Changes" action, which will request a deferred process to execute which will incrementally make changes to the affected CER rule objects.
- The system has been initially deployed into production and no CER rule objects have yet been created for rate table data.

  A system operator must arrange to run the `RateCreateInitialRuleObjects` process (see the *Propagating Non Cúram Data For Cúram Express Rules* guide).
- A database for a development system has been built and no CER rule objects have yet been created for rate table data.

  The developer can run the **`build prepare.application.data`** target prior to starting the application, or else the creation of rule objects will be performed automatically at application start-up[9].
- Changes to rate table data have been made outside of the application's APIs.

  Depending on the number of changes to rate table data, a system operator must arrange to either:

  - choose the "Apply Changes" action (for small numbers of changes); or
  - run the `FullPropagationToRuleObjects` batch process (for larger numbers of changes - see the *Cúram Operations Guide*).

---

[9] Shortly after start-up, the system checks the RuleObjectPropagatorControl table and runs initial propagation if it has not already been run. In a development environment, there is typically no discernible effect; however, in a production environment, the higher data volumes can mean that any initial propagation run after start-up (typically during user login) will cause the database transaction to timeout, depending on application server timeout settings.

To avoid this problem, be sure to run **`build prepare.application.data`** prior to starting the application.

Initial propagation is controlled by a single control row on the RuleObjectPropagatorControl table, which is populated by the DMX file included by the application. This control row ensures that initial propagation is only run once in an environment where many JVM instances attach to the database (e.g. during repeated runs of JUnit tests in a development environment, or when many application servers are used in a production environment).

The *`EJBServer/components/core/data/initial/RULEOBJECTPROPAGATORCONTROL.dmx`* file included by the application is required for the correct behavior of initial propagation, and so this file must not be customized or removed by customers. The control row populated by this DMX file must be reflected in any production database.

Logging

Configuration problems encountered by rule object propagators are automatically written to the application logs. You should monitor the logs and correct any warnings reported.

On occasion, it can be useful to log the detailed actions taken by the rule object converters and propagators.

The logging behavior of the rule object propagators is governed by these Cúram environment variables:

- `curam.trace.ruleobjectpropagation` (specific to rule object propagation); and
- `curam.trace` (general Cúram trace level).

The amount of logging performed by rule object converters and propagators can be controlled by setting either of these variables to one of the following (if both are setting, the more verbose setting takes precedence):

- trace_off;
- trace_on;
- trace_verbose; or
- trace_ultra_verbose.

Types of actions logged include:

- the details of a database write operation that has occurred;
- the details of a database write operation that is/is not of interest to a particular propagator;
- the details of a search for existing rule objects that match a database row which has changed or has been removed;
- details of each rule object created, modified or removed; and
- the value that a converter sets on a CER rule attribute.

Supported Domain Types

The conversion of business data into CER rule objects supports the use of the majority of the application's fundamental domain types.

The table below shows the correct CER data type to use for a CER rule attribute, which at rule object conversion time will be populated from a database or evidence field based on a domain. The table also describes any logic which is applied at data conversion time and the default value that the rule object converters will use for any values which are not sourced directly from data sources (e.g. when populating before-start or after-end values in a timeline):

Table 50: Mapping from Cúram Domain Types to CER Rule Attribute Types

Cúram Domain Type	CER Rule Attribute Type	Data conversion logic	Default value
Numerical types: • SVR_DOUBLE; • SVR_FLOAT; • SVR_INT8; • SVR_INT16; • SVR_INT32; • SVR_INT64; and • SVR_MONEY.	Java class - Number	Converted to CER's own numerical format.	0
Character types: • SVR_STRING; and • SVR_CHAR.	• Java class - String, for text data other than a code table code; or • Code table entry (specifying the appropriate code table) for text data which is a code table code		An empty String ("").
SVR_BLOB	Not supported.		
SVR_BOOLEAN	Java class - Boolean		false
SVR_DATE	Java class - `curam.util.value.Date`	The "zero date" (blank) is converted to a null value.	null
SVR_DATETIME	Java class - `curam.util.value.DateTime`	The "zero date time" (blank) is converted to a null value.	null

Tip: If there is a mismatch between the database column domain type and the CER rule attribute type, then at conversion time CER will report an error that the value set on the rule attribute does not match its expected type. This error points to the incorrect attribute data type having been modeled on the target rule class.

Important: Each CER rule attribute automatically has a `description` rule attribute (of type `curam.creole.value.Message`), inherited from the `RootRuleClass`.

Rule attributes of this `curam.creole.value.Message` data type do not map to a domain type, and so cannot be populated by the rule object converter. Any data named "description" in the source data will be ignored.

Propagator Filter Hook

A hook mechanism exists for CER propagators to enable propagation events to be filtered (i.e. ignored). A propagation event denotes a change to a piece of rules data, such as modifying a record. Ignoring a propagation event has the effect of hiding it from CER and the Dependency

Manager, meaning that a Precedent Change Item (PCI) will not get generated for that change. This can be used to prevent certain types of data changes from triggering reassessments.

To use the hook you must:

1. Implement the `PropagatorFilterHook` interface, and
2. Bind it to one or more propagator types

The `PropagatorFilterHook` interface is described below:

```
/**
 * Gets called for each propagator filter to indicate whether
 propagation
 * should be ignored.
 */
@curam.util.type.AccessLevel(curam.util.type.AccessLevelType.EXTERNAL)
public interface PropagatorFilterHook {

  /**
   * Indicates whether propagation should be ignored for the
 given propagator
   * and database details. Multiple filter instances can be
 defined for each
   * propagator type, so it is up to the filter to decide whether
 propagation
   * should be ignored for each given instance.
   *
   * @param prop The propagator instance, whose type should be
 checked to
   * ensure that it is of interest.
   * @param propagationListener The propagator listener.
   * @param session The current rules session.
   * @param databaseWriteDetails the details which are to be
 written to
   * the database.
   *
   * @return true if we wish to exclude the precedent changes
 from propagation.
   *
   * @throws AppException Standard signature.
   * @throws InformationalException Standard signature.
   */
  public boolean shouldIgnore(final RuleObjectPropagator prop,
    final PropagationListener propagationListener, final Session
 session,
    final DatabaseWriteDetails databaseWriteDetails) throws
 AppException,
      InformationalException;

}
```

Implementation notes:

- A single implementation can be bound to one or many propagator types. Similarly, many implementations can be bound to one or more propagator types. Other components of the product may also have bound implementations, so be aware that a particular implementation may not be the only one present in the product. If any one of these implementations returns 'true' then that propagation event is ignored.

- The hook can get called for multiple propagator types, depending on how the product has been configured, so the implementation should check the `prop` parameter to ensure that it is one for which the code should execute.
- The `databaseWriteDetails` parameter contains information about the propagation event such as the table name, operation name, and the contents of the details parameter being written to the database. It does not include the 'old' values for the data from the row, nor does it explicitly indicate which fields were changed by the transaction. However the details struct information can be used to re-read the original record from the database so that it can be compared against the 'new' record in order to detect which fields have changed.

Binding the implementation:

The example below shows class 'MyCustomHookImpl' being bound to the interface.

```
/*
   * When binding a propagator type, you must subclass
 PropagatorFilterHookDelegateImpl
   * and bind to it. Your subclass can be empty. This is due to a
 Guice 2.0 limitation
   * which requires that mapBinders bind to unique implementation
 classes.
   * So PropagatorFilterHookDelegateImplEntity is essentially an
 alias for
   * PropagatorFilterHookDelegateImpl.
   */
  public class PropagatorFilterHookDelegateImplEntity
  extends PropagatorFilterHookDelegateImpl {
    // Deliberately left empty, nothing to override.
  }

  /*
   * As above PropagatorFilterHookDelegateImplPDCHeader is
 essentially an alias for
   * PropagatorFilterHookDelegateImpl.
   */
  public class PropagatorFilterHookDelegateImplPDCHeader
  extends PropagatorFilterHookDelegateImpl {
    // Deliberately left empty, nothing to override.
  }

  public class Module extends AbstractModule {
    public void configure() {

    // Enable filtering for 2 types of propagator.
    final MapBinder<RULEOBJECTPROPAGATORTYPEEntry,
 PropagatorFilterHook> propagatorFilterHookMap =
 MapBinder.newMapBinder(
      binder(), RULEOBJECTPROPAGATORTYPEEntry.class,
 PropagatorFilterHook.class);


 propagatorFilterHookMap.addBinding(RULEOBJECTPROPAGATORTYPEEntry.ENTITY).
      PropagatorFilterHookDelegateImplEntity.class);

 propagatorFilterHookMap.addBinding(RULEOBJECTPROPAGATORTYPEEntry.PRODUCTD
      PropagatorFilterHookDelegateImplPDCHeader.class);
    // Add to the overall list of propagator hooks:
    final Multibinder<PropagatorFilterHook> propagatorFilterHook
 = Multibinder.newSetBinder(
      binder(), PropagatorFilterHook.class);

    // Bind your implementation to the hooks. It will now get
 invoked for each propagator
    // type to which subclasses of
 PropagatorFilterHookDelegateImpl were bound above.
    // This hook will now get called up to twice per propagation
 event, because
    // two propagator types have been specified above.
    propagatorFilterHook.addBinding().to(MyCustomHookImpl.class);
  }
}
```

1.8 How Determinations Are Stored

Introduction

After the Engine has calculated a determination result, the Engine must choose whether to store the determination result, and if so, how to store it.

The choice of whether to store a determination result hinges on a number of factors, described later. Typically each new determination result will end up being stored.

The Engine stores the determination result by writing a row to the CREOLECaseDetermination database table (and also some other data written to child database tables). The data stored includes the full details of the determination result and optionally also a "snapshot" of all the CER rule objects used in the calculation of the determination result.

The Engine also stores rows of eligibility and entitlement data CaseDecision and its child tables, so that this data can be used later byCúram Financials to generate financial components for the case. The Engine links the CaseDecision rows to the CREOLECaseDetermination row by storing rows on CREOLECaseDecision.

Note: In general, you should not need to access the data on any of these tables; this chapter merely provides a reference to the data stored.

This chapter is structured as follows:

- **The Database Tables**

  A description of the tables that the Engine uses to store determination data;
- **Decision Periods**

  How the Engine splits a determination into periods of constant eligibility/entitlement; and
- **Determination Comparison Strategies**

  How the Engine decides whether a new determination is "different" from an existing determination.

The Database Tables

The database tables below are used by the Engine to store determination data:

- CREOLECaseDetermination;
- CREOLECaseDeterminationData;
- CaseDecision;
- CaseDecisionObjective;
- CaseDecisionObjectiveTag; and
- CREOLECaseDecision.

CREOLECaseDetermination

This is the main table which owns the record of each determination.

The Engine stores a single row on this table for each determination result which ends up getting stored.

The details stored vary slightly depending on the type of determination being stored (Manual Check /Snapshot/Case Assessment). For data which is common to all types of determination, see the core Entity Relationship Diagram. The tables below show:

- the CREOLECaseDetermination data which is populated regardless of the type of determination; and
- the CREOLECaseDetermination data which varies according to the type of determination.

Table 51: Population of common CREOLECaseDetermination data

Attribute Name	Value
`creoleCaseDeterminationID`	Unique ID assigned by the system.
`caseID`	Identifier of the case which has its eligibility and entitlement determined.
`determinationDateTime`	The date and time that the determination was made.
`type`	The type of this determination. The value of this attribute governs the varying data stored, as shown in the following table.
`createdByUser`	The user who created this determination.
`determinationResultDataID`	The ID of the record which stores the XML document for the overall determination result.

Table 52: Population of CREOLECaseDetermination data, according to the type of determination

attribute name/ determination type	Manual Check	Snapshot	Case Assessment
`assessmentReason`	*blank*	*blank*	Value from the `CaseAssessmentDetReason` code table indicating the reason why the case assessment determination was requested. You are permitted to add new values to this code table to contribute your own assessment reasons if required.
`assessmentStatus`	*blank*	*blank*	Value from the `CaseDeterminationStatus` code table indicating whether this determination is: • Current (and thus is being used for deliveries such as financials); or • Superseded (and thus has been replaced by a different Current record). Extensions to this code table are not supported.
`snapshotReason`	*blank*	Value from the `CaseSnapshotDetReason` code table indicating the reason that the snapshot was requested. You are permitted to add new values to this code table to contribute your own snapshot reasons if required.	*blank*
`evidenceUsed`	Value from the `CaseDetEvidenceUsed` code table indicating whether the manual check was based off: • in-edit changes to evidence; or • active evidence only. Extensions to this code table are not supported.	*blank*	*blank*

attribute name/ determination type	Manual Check	Snapshot	Case Assessment
`ruleObjectSnapshotDataID`	*blank*, unless the application has been configured to record snapshots for manual check determinations (using the *curam.creole.manualeligibilitycheckdetermination.store.ruleobjectsnapshot* environment variable), in which case this attribute stores the ID of the record which stores the XML document of the snapshot of the rule object data used in the determination.	The ID of the record which stores the XML document of the snapshot of the rule object data used in the determination.	The ID of the record which stores the XML document of the snapshot of the rule object data used in the determination.

See also the core data dictionary.

CREOLECaseDeterminationData

This table stores XML data for determinations.

The Engine can store two different types of XML data for each determination (and so for each row on CREOLECaseDetermination there are typically two related rows on CREOLECaseDeterminationData):

- **Determination result**

  An XML representation of a determination result, including the full eligibility/entitlement, key decision factor and decision details over the lifetime of the case, and also details of any errors encountered during calculation of the determination. This XML will be used when data from the determination result is subsequently displayed to a case worker; and
- **Snapshot of Rule Objects**

  An XML snapshot of the CER rule objects used in the calculation of the determination result. This snapshot provides a point-in-time view of the CER rule objects and is stored to provide a full technical audit of how the determination result was calculated. The CER rule objects include those for input data (such as evidence, personal details and rates) and also all intermediate calculation steps. The snapshot points to the versions of the rule sets that were in place when the time that the snapshot was taken, so that the subsequent publication of changes to those rule sets do not affect the ability to read the snapshot data. The snapshot can be read using CER's `SnapshotDataStorage` feature. A snapshot is stored for assessment and snapshot determinations, but is only stored for manual check determinations if the application has been configured to do so (using the *curam.creole.manualeligibilitycheckdetermination.store.ruleobjectsnapshot* environment variable).

In the unlikely event that the XML data is too long to fit onto a single CREOLECaseDeterminationData, the data will be truncated to fit and the extra data stored on an "overflow" CREOLECaseDeterminationData row (or chain of overflow rows).

Important: The XML format of determination results and CER rule object snapshots is internal to the application and direct access to this XML is not supported.

The data contained in the XML may be accessed via the application's published APIs only.

CaseDecision

A determination result typically contains eligibility and entitlement data that varies over the lifetime of the case.

Note: The key decision factors feature is deprecated. For more information, see Deprecated features.

For assessment determinations, the Engine stores details of the case's varying eligibility and entitlement on the CaseDecision table (and its child tables, CaseDecisionObjective and CaseDecisionObjectiveTag, see below), so that financial processing can use this data to deliver the case's attained objectives.

When the Engine stores a new assessment determination, the Engine first supersedes the existing stored determination (if any) and supersedes any CaseDecision rows linked to that determination. For other types of determination no superseding takes place.

Then, for the new determination, the Engine inspects the varying eligibility and entitlement data to determine the dates on which the eligibility and/or entitlement changes (see Decision Periods on page 154). For each of these eligibility/entitlement change dates, the Engine stores a row on CaseDecision (and rows on its child tables) to detail the eligibility and entitlement results that apply from that date until the next change.

Sometimes it is possible for a determination result to contain key decision factors and/or decision details which change on a particular date where there is no accompanying change in eligibility and entitlement, for example where the case continues to be eligible but for a different business reason than previously. In these circumstances, the Engine will *not* store a CaseDecision record effective from the change date, because there has not been a change in eligibility and entitlement.

The CaseDecision table is used to store eligibility/entitlement data for CER-based cases as described above. However, it continues to be used to store eligibility/entitlement data for cases assessed using Cúram Rules (as opposed to CER). Some of the attributes on CaseDecision are reserved for use by Cúram Rules[10].

The Engine stores the following values in CaseDecision attributes for CER-based cases:

Table 53: Population of CaseDecision rows

Attribute on CaseDecision	Value stored by the Engine
`caseDecisionID`	Primary key
`caseID`	ID of case for which has been determined.
`decisionDate`	Set to the `decisionToDate`.

---

[10] See *Inside Cúram Eligibility and Entitlement Using Cúram Rules*.

Attribute on CaseDecision	Value stored by the Engine
`resultCode`	• "Eligible" if the eligibility result for the period is true; or • "Not eligible" otherwise (if the result is false or cannot be calculated).
`methodCode`	*blank* (reserved for use by Cúram Rules cases)
`initReasonCode`	Value stored depends on the type of determination: • For manual check determinations, "Pre-release". • For snapshot determinations, "Release". • For assessment determinations, "Release".
`typeCode`	Always "Product Delivery Eligibility".
`statusCode`	Value stored depends on the type of determination: • For manual check determinations, "Superseded". • For snapshot determinations, "Superseded". • For assessment determinations, "Current" if the CREOLECaseDetermination is Current, otherwise "Superseded".
`decisionFromDate`	Start of period from which the decision applies.
`decisionToDate`	End of period to which the decision applies - *blank* if the decision period is open-ended, i.e. is the last decision period in an open-ended case
`runMode`	*blank* (reserved for use by Cúram Rules cases)
`decisionFlow`	*blank* (reserved for use by Cúram Rules cases)
`decisionResult`	*blank* (reserved for use by Cúram Rules cases)
`tagValue`	*blank* (reserved for use by Cúram Rules cases)
`evidenceUsed`	*blank* (reserved for use by Cúram Rules cases)
`decisionFlowOverflowInd`	Always false (reserved for use by Cúram Rules cases).
`decisionResultOverflowInd`	Always false (reserved for use by Cúram Rules cases).
`evidenceUsedOverflowInd`	Always false (reserved for use by Cúram Rules cases).
`creationDate`	System date at the time the record is created.

CaseDecisionObjective

This describes an objective attained (i.e. entitled) for a parent CaseDecision. Objectives which are not entitled are not stored.

The Engine will create a set of CaseDecisionObjective rows for each change in a case's entitlement during its lifetime, as present in a determination result. Each CaseDecisionObjective row describes a single objective attained for the period of the CaseDecision (a CaseDecision may have zero, one or many attained objectives).

The Engine stores the following values in CaseDecisionObjective attributes for CER-based cases, centered around the determination data drawn from an `AbstractObjectiveTimeline` rule object:

Table 54: Population of CaseDecisionObjective rows

Attribute on CaseDecisionObjective	Value stored by the Engine
`caseDecisionObjectiveID`	Primary key
`objectiveID`	The identifier of the type of objective attained. Set to the CER attribute value `AbstractObjectiveType.objectiveTypeID` from the objective type returned by `AbstractObjectiveTimeline.objectiveType`.
`caseDecisionID`	ID of the parent CaseDecision which owns this CaseDecisionObjective record.
`concernRoleID`	The target of the attained objective. Set to the value during the decision period for the CER attribute value `AbstractObjectiveTimeline.targetIDTimeline`.
`value`	*blank* (reserved for use by Cúram Rules cases)
`relatedReference`	The related reference of the attained objective. Set to the value during the decision period for the CER attribute value `AbstractObjectiveTimeline.relatedReferenceTimeline`.
`overflowInd`	Always false (reserved for use by Cúram Rules cases).

CaseDecisionObjectiveTag

This describes the frequency at which an attained objective can be delivered.

The Engine will create a set of CaseDecisionObjectiveTag rows for each row on CaseDecisionObjective. These tags are used to calculate the amounts on the financial schedules. For more information see Calculating Financial Component Amounts on page 159

The Engine stores the following values in CaseDecisionObjective attributes for CER-based cases, centered around the determination data drawn from an `AbstractTagTimeline` rule object:

Table 55: Population of CaseDecisionObjectiveTag rows

Attribute on CaseDecisionObjectiveTag	Value stored by the Engine
`caseDecisionObjectiveTagID`	Primary key
`caseDecisionID`	ID of the parent CaseDecision which owns this CaseDecisionObjectiveTag record.
`objectiveTagID`	The identifier of the type of tag which can be delivered. Set to the CER attribute value `AbstractTagType.tagTypeID` from a tag type returned by `AbstractTagTimeline.tagType`.
`value`	The value of the objective if delivered at the frequency of this tag. Set to the value during the decision period for the CER attribute value `AbstractTagTimeline.valueTimeline`, with appropriate numeric-to-String conversions if required.
`description`	*blank* (reserved for use by Cúram Rules cases)

Attribute on CaseDecisionObjectiveTag	Value stored by the Engine
`type`	The type of data held in the value attribute. Set to the CER attribute value `AbstractTagType.valueType` from the tag type returned by `AbstractTagTimeline.tagType`.
`pattern`	The frequency at which this tag is delivered. Set to the CER attribute value `AbstractTagType.pattern` from the tag type returned by `AbstractTagTimeline.tagType`.
`objectiveID`	The type of objective for the parent objective which owns this tag.
`caseDecisionObjectiveID`	The ID of the parent objective which owns this tag.
`relatedReference`	*blank* (reserved for use by Cúram Rules cases)
`moneyValue`	Holds the precise string representation of a money objective tag.

CREOLECaseDecision

This is the link between a CREOLECaseDetermination and a CaseDecision which records a period of constant eligibility and entitlement within the determination.

The Engine stores a row on CREOLECaseDecision for every CaseDecision row which forms part of the determination written to CREOLECaseDetermination.

Decision Periods

The Engine stores details on CaseDecision and its child tables whenever there is a "change" in eligibility and/or entitlement over the lifetime of a case. In other words, the Engine splits a determination into "decision periods" of constant eligibility/entitlement, and stores each of those period as a row on CaseDecision (and links those rows back to the CREOLECaseDetermination).

The CER rule objects for eligibility and entitled objectives/tags contain a mixture of fixed data and data which changes over time. For the sake of clarity, this section describes each of the types of data changes which the Engine considers a "change" in eligibility and/or entitlement:

- Each date on which the eligibility result changes;
- Each date on which the set of attained objectives changes; or
- Each date on which any of these values change for an attained objective:
  - The target for the objective;
  - The related reference for the objective; and/or
  - The value of any type of tag for the objective.

Each change to any of the above will result in a CaseDecision record (plus child records) for that period during a determination.

Determination Comparison Strategies

When the Engine calculates a determination result, then the Engine will store that determination result if the new result is "different" from the previous result stored. Each CER-based Product can be configured to set just how "different" a new determination result needs to be in order to be stored (and thus to supersede the existing determination).

When the Engine stores a new determination snapshot (i.e. stores a new row on CREOLECaseDetermination), it will either:

- if the new determination result is "different" from the previously stored determination result, store a new row for the determination's XML documents on CREOLECaseDeterminationData, and link the new CREOLECaseDetermination to the new rows for the XML documents; or
- if the new determination result is not "different" from the previously stored determination result, link the new CREOLECaseDetermination to the existing rows for the XML documents stored against the previous determination.

For example, if a case worker repeatedly requests manual eligibility checks on the same data, each request will result in a (small) row being written to CREOLECaseDetermination, but only one pair of (large) rows being written to CREOLECaseDeterminationData.

Each CER-based Product must specify a strategy that the Engine will use when comparing determinations. You must either:

- in development, change your *`CREOLEProduct.dmx`* file to populate your product's `determinationCompStrategyType` column with the code (from the `DeterminationCompStrategy` code table) for your chosen strategy implementation; or
- in a running system, start the admin application and navigate to Product Delivery Cases, select your product, choose Rule Sets and choose Eligibility Determination, change "Determination Comparison Strategy" to be your chosen strategy implementation.

The interface for the strategy is `curam.core.sl.infrastructure.assessment.impl.DeterminationComparisonStrategy`. You must choose an existing comparison strategy or implement one of your own.

Tip: In the early stages of developing your product, it may be useful to initially use the "Compare all user-facing data" strategy included with the Engine, and then later in your development revisit whether this strategy meets your requirements.

Depending on the strategy in place for a product, it is possible for a new determination to be stored even though its eligibility and entitlement are identical to the that for the previous determination (i.e. the new CaseDecision and child records contain effectively the same data as the old records.).

This situation can arise when two determinations differ in *explanation only*. For example, a case may be determined to be ineligible forever because the claimant is not a citizen. If the claimant acquires citizenship, the case will be reassessed but the case may still be ineligible forever because the claimant fails a means test. In this way, the underlying eligibility and entitlement is unchanged (namely, "ineligible forever"), yet the explanation for the ineligibility has changed, and may cause a new determination to be stored (depending on the determination comparison strategy in place for the product).

Strategy Implementations Included with the Engine

The Engine includes these implementations which are suitable for most products:

Table 56: Determination Comparison Strategy Implementations Included with the Engine

Display/code
Compare all user-facing data

Display/code
Compare eligibility/entitlement data only

Developing your own Strategy Implementation

If you have custom requirements not met by the implementations included with the Engine, you may develop your own strategy implementation(s) for use in your products as follows:

- Add a new entry to `DeterminationCompStrategy` code table (using custom `.ctx` files);
- Create an implementation class which implements the `DeterminationComparisonStrategy` interface; implement the required method to return whether the two determinations passed in are considered to have equal data;
- Bind the code table entry to your implementation, in your custom Guice Module:

  ```
  {
    // Register your custom determination comparison strategies
    final MapBinder<DETERMINATIONCOMPARISONSTRATEGYEntry,
                       DeterminationComparisonStrategy>
      determinationComparisonStrategies
        = MapBinder.newMapBinder(binder(),
            DETERMINATIONCOMPARISONSTRATEGYEntry.class,
            DeterminationComparisonStrategy.class);

    determinationComparisonStrategies.addBinding(
      DETERMINATIONCOMPARISONSTRATEGYEntry.YOUR_STRATEGY.to(
        YourDeterminationComparisonStrategy.class));
  }
  ```

  (replacing *YOUR_STRATEGY* with the constant for your new code table code and *YourDeterminationComparisonStrategy* with your strategy implementation class as appropriate)
- Build your application;
- Configure your product to use your new strategy (see instructions above).

1.9 Scheduling Financials

Introduction

The financial scheduler is responsible for scheduling financial transactions based on eligibility and entitlement results and case deductions. These financial schedules, known as financial components, are used by the Financial Manager to create financial instruction line items. The financial scheduler sits between the Eligibility and Entitlement Engine and the Financial Manager, translating eligibility and entitlement results as well as case deductions into financial schedules that can be processed into actual payments or bills.

This chapter covers the scheduling of financials for eligible case decisions, case deduction items, and payment corrections. The approach used to describe each of these financial schedules is the same followed throughout this guide. For each financial schedule here is a description of how it looks, how it works, and how to use it.

Financial components are schedules of transactions to be realized into actual financial transactions. A financial component encompasses all of the elements that constitute a financial schedule, e.g. amount, cover period, frequency, validity period, effective date and so on.

Scheduling Financials for Eligible Case Decisions

The case determination information produced by the Engine is used to create the financial schedules for the case. Only eligible case decisions are considered and each eligible decision will have one or more associated case decision objectives. Each of these case decision objectives represents a component for which financials must be scheduled. One or more financial components are created for each case decision objective, and these financial components are used to create one or more instruction line items that represent the actual financial transactions.

How It Looks

This section describes how financial information is displayed to a case worker. A financial instruction representing a benefit payment to a nominee is generated and displayed. From this, the case worker can view the total amount due, the instruction line items which make up the total payment, the nominee and the delivery details.

For example, a financial instruction has been generated and is displayed for the period 22nd June to 10th July, for the amount of $407.15. This financial instruction is comprised of three payment instruction line items. The nominee was determined eligible to receive an Income Assistance component and entitled to a weekly amount of $150. The financial scheduler uses the one case decision objective produced by the Engine to create two financial components.

The first financial component serves as a ramp-up financial component to cover a partial payment period. Because the nominee is assigned a delivery pattern of 'Weekly by Check in Advance on a Monday' and becomes eligible starting on Wednesday 22nd of June 2011, the financial scheduler creates a one-time ramp-up financial component that is processed by the Financial Manager into one instruction line item for a partial week payment of $107.15 that covers the period from Wednesday 22nd of June 2011 through Sunday 26th of June 2011.

The second financial component serves as a recurring financial component that is processed by the Financial Manager into two instruction line items, one for a full weekly payment of $150 for the week starting Monday 27th of June 2011 and the second for the week starting Monday 4th of July 2011.

How It Works

There are a number of factors taken into consideration when deciding how to represent a specific case decision as a financial schedule. These include the case decision objectives, the nominee component assignments, the nominee delivery patterns, the period to which the decision applies and the case decision objective tags which have been specified. The following sections provide more information on each of these factors.

Considering Case Decision Objectives

An eligible case decision is first turned into a set of virtual components, one virtual component for each of associated case decision objectives. The start and end dates of these virtual components will match the start and end dates of the decision. If the decision is open-ended the virtual components will also be open-ended.

For example, a case decision which indicates eligibility for the Income Assistance component from 13th June 2011 until 29th July 2011 would generate a virtual component starting on 13th June 2011 and ending on 29th July 2011.

Considering Nominee Component Assignments

Once an initial virtual component has been created the next step is to examine the nominee component assignments. Any changes to the component assignment during the cover period of the virtual component are identified and are used to split the initial virtual component into multiple, nominee specific parts.

For example, the Income Assistance component is assigned to nominee James from 13th June 2011 to 5th July 2011 and is then assigned to nominee Linda from 6th July 2011 onwards. Using this information the initial virtual component is split up, giving one virtual component for nominee James, which covers the period 13th June 2011 to 5th July 2011 and one virtual component for nominee Linda, which covers the period 6th July 2011 to 29th July 2011.

Considering Nominee Delivery Patterns

Once the nominee specific virtual components have been created, the next step is to examine the nominee delivery patterns. Any changes to the nominee's delivery pattern during the cover period of their virtual component are identified and are used to split it again into multiple, delivery pattern specific parts.

For example, nominee James is paid using the pattern 'Weekly by EFT in Advance on a Monday' from 13th June 2011 to 3rd July 2011 and then paid 'Daily by EFT' from 4th July 2011 onwards. Nominee Linda is paid using the pattern 'Weekly by Check in Advance on a Monday' from 6th July 2011 onwards. In this situation the virtual component for nominee James will be split into two parts. One virtual component for the period of the 'Weekly by EFT in Advance on a Monday' delivery pattern and once virtual component for the period of the 'Daily by EFT' delivery pattern.

At the end of this processing we will have three virtual components that must be realized into appropriate financial schedules as follows:

- Monday 13th of June 2011 until Sunday 3rd July 2011 for James, Weekly by EFT in Advance on a Monday
- Monday 4th of July 2011 until Tuesday 5th July 2011 for James, Daily by EFT
- Wednesday 6th of July 2011 until Friday 29th of July 2011 for Linda, Weekly by Check in Advance on a Monday

Calculating Financial Component Cover Periods

Each of the virtual components that we have at this stage is then further split up based on how its cover period matches with the frequency of the nominee delivery pattern. For each virtual component up to three financial components may be created. Each financial component will apply for a specific interval during the period to which the virtual component applies.

If the virtual component does not begin on a day that can initiate a complete delivery period, a financial component is created to cover the period between the virtual component start date and the beginning of the first complete delivery period. This is known as a ramp-up financial component.

A second financial component may be generated to cover all complete cycles of the delivery frequency that can be achieved within the virtual component cover period. This is known as a recurring financial component.

A final financial component may be created to cover the period between the end of the last complete delivery period and the virtual component end date. This is known as a ramp-down financial component.

For example, the virtual component for nominee Linda covers the period from Wednesday 6th of July 2011 to Friday 29th of July 2011 and uses the delivery frequency Weekly by Check on Monday. Using the frequency of this nominee delivery pattern to split up the virtual component means that a separate financial component is generated to cover each of the periods listed below:

1. From Wednesday 6th July 2011 to Sunday 10th July 2011. This financial component covers the interval from the start of the virtual component to the start of the first complete delivery cycle, i.e. Monday 11th July.
2. From Monday 11th July 2011 to Sunday 24th July 2011. This financial component covers the period of the virtual component that contains all of the complete delivery frequency cycles.
3. From Monday 25th July 2011 to Friday 29th July 2011. This financial component covers the period between the end of the last complete delivery cycle and the virtual component end date.

Calculating Open Ended Financial Component Cover Periods

If a product supports open ended cases, it means that the last decision on a case may be open ended. When the last decision is eligible, then it results in the creation of a recurring, open-ended financial component. This means that a ramp-down financial component is not required.

For example, an eligible, open ended decision has been created starting on Wednesday 6th of July 2011 which applies until further notice. For a case with one nominee component assignment and one nominee delivery pattern which has the delivery frequency Weekly by Check on Monday, one open ended virtual component will be created. Using the frequency of the nominee delivery pattern to split up the open ended virtual component means that a separate financial component is generated to cover each of the periods listed below:

1. From Wednesday 6th July 2011 to Sunday 10th July 2011. This financial component covers the interval from the start of virtual component to the start of the first complete delivery cycle, i.e. Monday 11th July.
2. From Monday 11th July 2011 until further notice. This financial component covers the period of the virtual component that contains complete delivery frequency cycles.

At some point in the future, an end date will be added to the case. This might be because of a change of circumstance or because the case is being closed. When this occurs, the case will be reassessed and the previous open ended decision will be replaced by one which has an end date. At that point the open ended financial component will be replaced by a bounded financial component which applies until the case end date.

Calculating Financial Component Amounts

After the required financial components have been identified, the amount and effective date must be determined for each one. The amount is determined by examining the period for which the financial component applies. For a ramp-up or ramp-down financial component, this will be the cover period of the financial component. For a recurring or open-ended financial component, this will be cover period of the first complete delivery cycle.

For example, if we take the three financial components identified above for nominee Linda, the periods specified would be:

1. From Wednesday 6th July 2011 to Sunday 10th July 2011.
2. From Monday 11th July 2011 to Sunday 24th July 2011.
3. From Monday 25th July 2011 to Friday 29th July 2011.

The amount used for each individual financial component depends on the case decision objective tags which have been specified in the rule set for the product. Each tag has an associated frequency. The available frequencies are daily, weekly, bi-monthly, monthly and yearly.

When calculating the amount for a recurring financial component typically the frequency will match one of the available tags. When calculating the amount for a ramp-up or ramp-down financial component the available tags are applied largest first.

For example, if the rule set for the product specified a 'daily' case decision objective tags with an amount of $10 (indicating that for a single day the amount of 10 should be paid) and a 'weekly' case decision objective tag with an amount of $65 (indicating that for a full week the amount of 65 should be paid), the amounts calculated for the three financial components would be as follows:

1. From Wednesday 6th July 2011 to Sunday 10th July 2011 the amount would be $50 (five days at $10 per day).
2. From Monday 11th July 2011 to Sunday 24th July 2011 the amount would be $65 (one full week at $65 per week).
3. From Monday 25th July 2011 to Friday 29th July 2011 the amount would be $50 (five days at $10 per day).

Consider this second example which explains how the tags are applied largest first. If the delivery frequency is monthly and a ramp-up financial component is required for the period from 8th June 2011 until 30th June 2011 (23 days in total), and the following case decision objective tags are specified:

1. A monthly tag at $250 per month
2. A weekly tag at $65 per week.
3. A daily tag at $10 per day.

Using these Tags to calculate the amount for the ramp-up financial component would give a total of $215 (three weekly tags plus two daily tags).

However, if no weekly tag had been specified then the amount calculated for the ramp-up financial component would be $230 (twenty three daily tags).

Calculating Financial Component Effective Dates

The effective date for the financial component is determined using a combination of the cover period type and the delivery pattern. For each of the supported cover period types, the effective date is calculated as follows:

- Issue in Advance

  The effective date is the start date of the first delivery period covered by the financial component. Therefore, for the three financial components defined above for nominee Linda, the effective dates would be Monday 4th July, Monday 11th July and Monday 25th July respectively. It is worth noting that the effective date for the first financial component is not within the cover period for that component.
- Issue in Arrears

  The effective date is the start date of the delivery period following the first delivery period covered by the financial component. Therefore, for the three financial components defined above for nominee Linda, the effective dates would be Monday 11th July, Monday 18th July and Monday 1st August respectively.
- Issue for Full Month

The effective date is the start date of the financial component. Therefore, for the three financial components defined above for nominee Linda, the effective dates would be Wednesday 6th July, Monday 11th July and Monday 25th July respectively.

- Once-off Issue

  The effective date is the start date of the financial component. Therefore, for the three financial components defined above for nominee Linda, the effective dates would be Wednesday 6th July, Monday 11th July and Monday 25th July respectively.
- Issue in Advance - N Days Prior to Issue Date

  The effective date is N days prior to the start date of the first delivery period covered by the financial component. For example, if N equals 2, then for the three financial components defined above for nominee Linda, the effective dates would be Saturday 2nd July, Saturday 9th July and Saturday 23rd July respectively.
- Issue in Arrears - N Days Prior to Issue Date

  The effective date is N days prior to the start date of the delivery period following the first delivery period covered by the financial component. For example, if N equals 2, then for the three financial components defined above for nominee Linda, the effective dates would be Saturday 9th July, Saturday 16th July and Saturday 30th July respectively.
- Issue in Advance - on the Nth Day of Month Prior

  The effective date is the Nth day of the month prior to the start date of the financial component cover period. For example, if N equals 2, then for each of the three financial components defined above for nominee Linda the effective date would be Friday 3rd June.

How to Use It

This section describes the information that must be provided to allow financial schedules to be generated for an eligible case decision. It is divided into two sections, mandatory information and optional information.

Mandatory Information

Case Decision Objectives

These are defined in the rule set assigned to the product on which this case is based. An Objective is anything that can be awarded as part of a determination result calculated by CER rules. An eligibility rule set can have multiple objectives.

For more information see Objectives on page 36.

Case Decision Objective Tags

These are also defined in the rule set assigned to the product on which this case is based. An Objective Tag represents how a particular objective is awarded for a specific period of time (from days to years). An objective may have several objective tags. For example, a Loan Parent benefit product may have two objective tags, one applying 'per day' and one 'per week'. An objective tag can be an amount of money or a formula that evaluates an amount of money for a specific objective.

For more information see Objective Tags on page 37.

Financial Code Tables

There are five code tables which must be customized to enable the financial schedules and subsequent financial transactions to be generated correctly from the eligible case decisions. They are:

1. `RulesComponentType`;
2. `FinComponentType`;
3. `ProductComponentFCConv`;
4. `ILIType`; and
5. `TranslateILIType`.

For more information see the 'Financial Code Tables' section of the *Inside the Cúram Financial Manager* Guide.

Optional Information

Nominee Component Assignments

Every component available on a case must be assigned to a nominee. When a case is first created all the components are initially assigned to the default nominee. However, if an additional nominee has been added to the case they can be assigned a component from a certain date or for a specific period of time.

Assigning a component to multiple nominees during the lifetime of an eligible case decision will result in the creation of separate financial components for each of those nominees. Nominee Component Assignments can be configured via the Transactions page of the case.

For more information see the *Cúram Nominees* Guide.

Nominee Delivery Patterns

Every nominee must have a delivery pattern which indicates how they wish to receive payments and at what frequency. A nominee's delivery pattern can change over time, but no gaps are allowed. The delivery pattern can be specified when the nominee is being added to the case. If one is not explicitly selected, then the nominee is given the delivery pattern currently in use by the default nominee.

Having multiple delivery patterns for a nominee during the lifetime of an eligible case decision will result in the creation of separate financial components for each of the delivery patterns used. Nominee Delivery Patterns can be configured via the Transactions page of the case.

The nominee delivery pattern is also where the delivery frequency and the cover period type are specified, as well as any offsets.

For more information see the *Cúram Nominees* Guide.

Allow Open Ended Cases Indicator

This indicator can be found on the Eligibility Determination tab of the Rule Sets page for a Product. It is set to 'Yes' by default, meaning that cases based on this product may be open ended. Only open ended cases can generate open ended decisions and subsequently open ended financial schedules. Setting this indicator to 'No' ensures that an end date or expected end date must always be set for cases based on this product and ensures that any financial schedules created for such a case will always have an explicit end date.

For more information see the `Integrated Case Management Configuration Guide`.

Scheduling Financials for Case Deductions

The case deduction item information on a case is used to create the deduction financial schedules. Only active case deduction items are considered. Deductions can either be for a fixed amount or for a percentage of the payment amount, with the specific amount being calculated during payment generation.

For a fixed deduction, the value of the deduction financial component is taken directly from the case deduction item. A fixed deduction is applied to the total payment amount of all the deductible components that the specified nominee has received.

For a variable deduction, a rate is used instead. The rate indicates the percentage of the payment amount that should be deducted. A variable deduction is applied to the total payment amount of a specific component if one is selected, otherwise it is applied to the total payment amount of all the deductible components on the case. Since a variable deduction represents a percentage of the total payment, the same percentage is deducted from each nominee receiving one of the applicable components.

Each active case deduction item defined for a case will result in the generation of one or more financial components depending upon the number of nominees affected. One or more deduction instruction line items will then be generated for each financial component.

For more information on configuring deductions, see the `Cúram Deductions` Guide.

How It Looks

This section describes how financial information is displayed to a case worker when a deduction is taken from a payment. A financial instruction representing a benefit payment to a nominee is generated and displayed. From this, the case worker can view the total amount due, the instruction line items and the deduction items which make up the total payment, the nominee and the delivery details.

For example, a financial instruction has been generated and is displayed for the period 22nd June to 10th July, for the amount of $397.15. This financial instruction is comprised of three payment instruction line items and one deduction instruction line item. The nominee was determined eligible to receive an Income Assistance component and entitled to a weekly amount of $150. A fixed deduction of $10 has also been created for the nominee in order to assist the nominee in making payments for a utility bill. The financial scheduler uses the one case decision objective produced by the Engine to create two financial components to represent the payment, and uses the case deduction item information to create one financial component to represent the deduction.

The first payment related financial component serves as a ramp-up financial component to cover a partial payment period. Because the nominee was assigned a delivery pattern of 'Weekly by Check in Advance on a Monday' and became eligible starting on Wednesday 22nd of June 2011, the financial scheduler creates a one-time ramp-up financial component that is processed by the Financial Manager into one instruction line item for a partial week payment of $107.15 to covers the period from Wednesday 22nd of June 2011 through Sunday 26th of June 2011.

The second payment related financial component serves as a recurring financial component that is processed by the Financial Manager into two instruction line items, one for a full weekly payment for $150 for the week starting Monday 27th of June 2011 and the second for the week starting Monday 4th of July 2011.

The deduction financial component is processed by the Financial Manager into one instruction line item for the amount of $10 that is deducted from the payment financial instruction created for the nominee.

How It Works

There are a number of factors taken into consideration when deciding how to represent a specific case deduction item as a financial schedule. These include whether the deduction is fixed or variable, the nominee component assignments, the nominee delivery patterns, the period to which the deduction applies and the latest payment date associated with the case itself. The following sections provide more information on each of these factors.

Considering Case Deduction Items

First the active case deduction items are retrieved. Only a case deduction item with an end date after the last paid to date of the case will be considered. If an active deduction has already ended, it will not be used generate financial schedules.

For example, a case which starts on 13th of June 2011 and has an expected end date of 10th July 2011 delivers two components, Income Assistance and Medical Assistance. The case has two nominees Lisa and Paul, but no payments have been issued yet.

Lisa is the default nominee and so is assigned both components from the case start date. Lisa's nominee delivery pattern is 'Weekly by EFT in Advance on a Monday' and applies from the case start date. Paul's nominee delivery pattern is 'Weekly by Check in Advance on a Monday' and also applies from the case start date. Paul is assigned the Income Assistance component from 27th June 2011 (the start of the third week).

The case also has two case deductions specified as follows:

1. A fixed deduction for the amount $12, assigned to nominee Lisa. The deduction starts on 13th of June 2011 and has no end date specified, meaning that it applies until further notice.
2. A variable deduction for 20% against the Income Assistance component. It starts on 20th of June 2011 and has an end date of 5th July 2011.

In this example, both case deduction items will be retrieved and used to generate financial components.

Considering Deduction Types

Once the case deduction items have been identified, the next step is to check the deduction type to determine how many nominees are affected. A fixed deduction will affect a single nominee while a variable deduction may affect multiple nominees depending on the components it is targeted at and the nominee component assignments. Any changes to the component assignments during the cover period of the deduction are identified and are used to split the initial deduction component into multiple, nominee specific parts. This is done for each component affected by the deduction.

In our example, the fixed deduction is assigned to Lisa, so it can be represented by a single deduction financial component. The variable deduction is against the Income Assistance component which is assigned to Lisa for the first two weeks and then to Paul for the remainder of the case, so the deduction financial component representing this must be split in two.

At the end of this processing we will have three deduction financial components as follows:

1. From Monday 13th of June 2011 until Sunday 10th July 2011 for nominee Lisa, deducting $12 per week from her total payment.

2. From Monday 13th of June 2011 until Sunday 26th June 2011 for nominee Lisa, deducting 20% per week from her total payment.
3. Monday 27th of June 2011 until Sunday 10th July 2011 for nominee Paul, deducting 20% per week from his total payment.

Calculating Deduction Cover Periods

The cover period of the deduction financial component starts on the case deduction item start date, unless that date is before the last date paid out on the case. In that situation, the financial component cover period starts the day after the last date paid.

The cover period of the deduction financial component ends on the case deduction item end date, if specified. If no end date has been specified the expected end date of the case is used instead.

For example, if the case deduction item has a start date of 13th June 2011 and no end date is specified and the case to which it belongs has an expected end date of Sunday 10th July 2011 and has been paid up to Sunday 19th June 2011. The cover period of the deduction financial component would be from Monday 20th June 2011 to Sunday 10th July 2011.

How to Use It

This section describes the information that must be provided to allow financial schedules to be generated for a case deduction item. It is divided into two sections, mandatory information and optional information.

Mandatory Information

Case Deduction Items

The relevant case deduction item must be added to the case. It must have been Activated, and its end date, if specified, must be later than the last date paid out on the case.

For more information see the `Cúram Deductions` Guide.

Optional Information

Nominee Component Assignments

Every component available on a case must be assigned to a nominee. When a case is first created all the components are initially assigned to the default nominee. However, if an additional nominee has been added to the case they can be assigned a component from a certain date or for a specific period of time.

Assigning a component to multiple nominees during the lifetime of a variable case deduction item targeted at that component will result in the creation of separate deduction financial components for each of those nominees. Nominee Component Assignments can be configured via the Transactions page of the case.

For more information see the `Cúram Nominees` Guide.

Scheduling Financials for Payment Corrections

The reassessment information produced by the Engine is used to create the payment correction financial schedules.

When a change of circumstance results in a reassessment over a period which has already been paid, that reassessment may determine that the entitlement amount originally calculated and subsequently issued to the nominee for that period was incorrect. When this occurs the original entitlement must be corrected and a balancing payment or bill must be issued. A payment correction is the mechanism used to do this.

If the original payment was larger that it should have been, an overpayment correction will be created. If the original payment was smaller that it should have been, an underpayment correction will be created. If the original payment was comprised of multiple components and the total amount for the overpaid components was found to be the same as the total amount for the underpaid components, a net zero payment correction will be created.

Reassessment information is stored for each nominee affected by the payment correction in the NomineeOverUnderPayment table. This information is then broken down by component for each nominee and stored in the OverUnderPaymentBreakdown table and is used to create evidence for payment correction. The type and number of evidence records created depends upon the type of product used to deliver the payment correction. Each evidence record results in the creation of one corresponding financial component which in turn is used to create one instruction line item.

How It Looks

This section describes how financial information is displayed to a case worker when an overpayment or underpayment occurs and is processed within a Payment Correction case.

For example, a nominee was initially determined eligible to receive both an Income Assistance and Medical Assistance component and is entitled to a weekly amount of $150 for the Income Assistance component and a weekly amount of $35 for the Medical Assistance component. The financial scheduler uses the two case decision objectives produced by the Engine to create two financial components that are processed by the Financial Manager into a payment.

Due to a change in circumstance for the nominee, reassessment of the nominee's eligibility and entitlement occurs resulting in an overpayment for both components and the creation of a Payment Correction case. The financial scheduler uses the reassessment information produced Engine to create two financial components.

The first financial component is created for the Income Assistance component and is processed by the Financial Manager into one liability instruction line item for $6.41 that covers the period of the original payment, from Wednesday 22nd of June 2011 through Thursday 7th of July 2011.

The second financial component is created for the Medical Assistance component and is processed by the Financial Manager into one liability instruction line item for $10.71 that covers the period of the original payment, from Wednesday 22nd of June 2011 through Thursday 7th of July 2011.

A financial instruction representing a liability for an overpayment to a nominee is displayed within a Payment Correction case for the amount of $17.12. This financial instruction is comprised of the two liability instruction line items created by the Financial Manager.

In the originating Product Delivery case, the Engine has determined that the entitlement amount originally calculated and subsequently issued was larger than it should have been, resulting in the creation of an overpayment.

The overpayment of $17.12 that resulted in the creation of the Payment Correction case is displayed. The overpayment is broken down into an overpayment of $10.71 for the Medical Assistance component and an overpayment of $6.41 for the Income Assistance component.

How It Works

For a payment correction, the generation of the financial schedules is normally a two step process. First a product delivery case is created and the reassessment information is used to create the evidence for the case. When that case is subsequently activated the evidence is used as the basis for the financial schedules.

An alternative process is possible for an underpayment correction. With this approach it is possible to deliver the underpayment on the original benefit case, rather than using a separate underpayment case. In this situation the reassessment information is still used to create the evidence, but the necessary financial component(s) are created immediately afterwards.

There are a number of factors taken into consideration when deciding how to represent a payment correction as a financial schedule. These include the payment correction type determined by the reassessment, the product used to deliver the payment correction, the number of components included in the reassessment, and various administration and product settings. The following sections provide more information on each of these factors.

Considering Payment Correction Types

An overpayment correction is created when the original payment was larger that it should have been. If the original payment was comprised of multiple components then it is possible that some of those component could have been overpaid while others have been underpaid, but when the totals are combined the balance is an overpayment.

An underpayment correction is created when the original payment was smaller that it should have been. If the original payment was comprised of multiple components then it is possible that some of those component could have been overpaid while others have been underpaid, but when the totals are combined the balance is an underpayment.

A net zero payment correction is created when the original payment was comprised of multiple components and the total amount for the overpaid components was found to be the same as the total amount for the underpaid components. When the totals are combined the balance is zero.

In the example, the Engine has determined that an overpayment correction is required. The reassessment information shows that the total overpayment amount is $17.12 and this total was calculated by adding the overpayment of $10.71 for the Medical Assistance component and the overpayment of $6.41 for the Income Assistance component.

Considering Correction Products

There are three products which can be used when creating financial schedules for a payment correction. They are:

- The Payment Correction product which has the ability to produce individual financial schedules for each component which was over or under paid on the original benefit case. This product supports all the payment correction types (overpayment, underpayment and net-zero).
- The Overpayment product which has the ability to produce a single liability financial schedule for the total amount overpaid on the original benefit case. This product supports only the overpayment correction type.
- The Underpayment product which has the ability to produce a single benefit financial schedule for the total amount underpaid on the original benefit case. This product supports only the underpayment correction type.

The payment correction product will be used by default, however an application property is provided that allows the overpayment and underpayment products to be used instead.

In the example, the product associated with the case is configured to use the Payment Correction product, and since this is an overpayment correction a Payment Correction case will be created.

Once the Payment Correction case has been created the appropriate evidence is added to it. In this example, two evidence records will be created:

1. An overpayment record for the amount of $10.71 to correct the original payment for the Medical Assistance component.
2. An overpayment record for the amount of $6.41 to correct the original payment for the Medical Assistance component.

Important: Cases that have been migrated from earlier versions of the application may have existing reassessment information. For case such as this, the granular reassessment information will not be available. Therefore the payment correction product cannot be used. So these cases will continue to use the overpayment and underpayment products as before.

Considering Nominees

For an overpayment correction, the nominee on the Payment Correction case will be the primary client of the original benefit case and the nominee delivery pattern will be 'Once-off by Invoice'.

For an underpayment correction the nominee on the Payment Correction case will be same as the nominee underpaid on the original benefit case. If this is different to the primary client, a second nominee will be added to the Payment Correction case to support this. The nominee delivery pattern will be 'Once-off by X' (where X is the delivery method of that nominee on the original benefit case). A new nominee can also be added to the underpayment case and assigned to the benefit underpayment component, however unlike regular benefit cases, the payment cannot be split between multiple nominees over different period of times because an underpayment is a single once off payment. In order for a different nominee to receive the underpayment, the date range of the component assignment must cover the entire cover period of the benefit underpayment component. If the date range of the component assignment does not cover the entire cover period, then the original nominee will still receive the entire underpayment.

For a net zero payment correction, the nominee on the Payment Correction case will be the primary client of the original benefit case and the nominee delivery pattern will be 'Once-off by X' (where X is the delivery method of that nominee on the original benefit case).

In the example, the Engine has determined that an overpayment correction is required. The nominee on the financial component will be the primary client of the original benefit case and the nominee delivery pattern will be 'Once-off by Invoice'.

How to Use It

This section describes the configuration settings which control how financial schedules are generated for payment corrections. All the configuration options described here have default values which can be changed if required.

The 'Use Rolled Up Reassessment Products' setting is a configuration option which controls the product that will be used to deliver a correction. When this is set to 'NO', it indicates that the Payment Correction product should be used. When it is set to 'YES', it indicates that the legacy, Overpayment and Underpayment products should be used. The default value for this setting is 'NO'.

For more information see the *`Integrated Case Management Configuration Guide`*.

When will an Overpayment Correction case be created?

Depending on the value of the 'Use Rolled Up Reassessment Products' setting, either a payment correction case or an overpayment case may be created to deliver the overpayment correction. The following configuration settings also affect the behaviour:

- Automatic Overpayment Case Processing

  This setting works in conjunction with the 'Use Rolled Up Reassessment Products' setting mentioned above and can be configured in one of three ways. The first option allows an administrator to specify that a separate case should be automatically created when an overpayment correction is detected. This will be either an overpayment case or a payment correction case depending on the value that is specified for the 'Use Rolled Up Reassessment Products' setting. Once the case is created, a user must manually approve, activate, and generate the liability financials required to recoup the overpayment.

  The second option allows an administrator to specify that a separate case should be automatically created and approved, activated, and liability financials generated without the intervention of a user. Note that this option is only available for benefit products for which the value of the 'Use Rolled Up Reassessment Products' setting is 'NO'.

  The third option instructs the system not to automatically create a separate case to correct the overpayment. Instead, a task is generated to alert the user of the overpayment. The user can then manually create and manage a liability case to recoup the overpayment.

When will an Underpayment Correction case be created?

Depending on the value of the 'Use Rolled Up Reassessment Products' setting, either a payment correction case or an underpayment case may be created to deliver the underpayment correction. Alternatively, in some situations it is possible for the underpayment correction to be delivered on the original benefit case. The following configuration settings also affect the behaviour:

- Automatic Underpayment Case Creation

  This setting works in conjunction with the 'Use Rolled Up Reassessment Products' setting mentioned above and can be configured in two ways. The first option allows an administrator to specify that a separate case should be automatically created when an underpayment correction is detected. This will be either an underpayment case or a payment correction case depending on the value that is specified for the 'Use Rolled Up Reassessment Products' setting. Once the case is created, a user must manually approve, activate, and generate the benefit financials required to issue the underpayment.

  The second option instructs the system not to automatically create a separate case to correct the underpayment. Instead, an underpayment financial component should be created on the original benefit case to deliver the underpayment.
- *curam.miscapp.checkforliveliabilities*

  This application property determines whether a check is performed by the system to establish the existence of outstanding liabilities for a client when an underpayment correction has been detected. The default value of this property is 'YES', meaning that a separate case will always be created to deliver the underpayment when an outstanding liability exists for this client.
- *curam.miscapp.underpmtcase.createfornomineediff*

  This application property determines whether a check is performed by the system to establish whether the nominee on the original benefit case, currently assigned the underpaid component, is the same as the underpaid nominee. The default value of this property is 'YES', meaning

  that a separate case will always be created to deliver the underpayment when the nominee currently assigned the component is not the nominee underpaid.
- Invalidate Payments

  This is not a configuration option, rather it is a check that is carried out internally by the financial scheduler. If the underpayment correction is for a period that has been paid, but the payment was cancelled and invalidated, then a separate case will always be created to deliver the underpayment.

When will a Net Zero Correction case be created?

Net zero payment corrections are created when the overpaid components and underpaid components included in a reassessment cancel each other out. Since the creation of a net zero payment correction case is only possible when the Payment Correction product is used, the value of the 'Use Rolled Up Reassessment Products' setting must be 'NO'.

Net zero payment correction cases will not be of great interest to a case worker, but they facilitate fund management and accurate account management.

With this in mind, net zero payment correction cases are automatically created, approved, activated, and the financials generated without the intervention of a user.

1.10 Reassessment - Handling Changes in Circumstance

When a caseworker first activates a product delivery case, the Engine creates a determination for the initial assessment for that case, and uses that determination as input into financial processing.

This initial assessment takes into account:

- Case-specific circumstances known at the time of the initial assessment, including:
  - Personal data, such as dates of birth.
  - Case data, such as case start and end dates.
  - Evidence, such as income levels for household members.
- Product-wide configuration, including:
  - Product periods.
  - The product rules for eligibility/entitlement, key decision factors and decision details.
  - Rate data.

Any of all of these types of data can change as time goes on; families move, they have more children, their income fluctuates, they encounter unforeseen problems that lead to greater need; product legislation and/or policy changes, rates change in line with costs of living.

Depending on the nature of the change, some changes may affect future periods on cases which have not yet been paid. However, other types of change may affect periods on cases which have already been paid, resulting in under- or over-payments (see Scheduling Financials for Payment Corrections on page 165).

Case-level reassessment

Most data changes recorded in the application affect a small number of cases. Product delivery cases are automatically reassessed when certain types of data change.

- When evidence recorded directly against a product delivery case is activated, typically only that product delivery case is affected.
- When evidence that is recorded against an integrated case is activated, typically only the product delivery cases belong to that integrated case are affected.
- When personal data is changed, typically only cases which involve those persons are affected. Such cases might reference the person data directly, or use Evidence Broker to control the use of the personal data in cases.
- When product delivery case data such as case start or end dates are changed, typically only that product delivery case is affected.

Note: The Engine does not enforce any restrictions about sharing evidence or other data across cases.

When changes to data occur, they are written to a precedent change set and processed by the Dependency Manager. The Dependency Manager processes the precedent change set by finding the unique set of dependents affected by any of the items in the precedent set and instructing each affected dependent to recalculate itself. Each type of dependent in the system has a registered 'dependent handler' which, when invoked by the Dependency Manager, is responsible for taking the appropriate steps to recalculate a dependent. If any of the dependents that are affected are a case assessment determination then the appropriate step is to request that the Engine reassesses the case, by invoking CER to recalculate the value of the `determinationResult` attribute value for the case.

The processing of precedent change set items that are likely to affect a small number of cases occurs in deferred processing.

Note: Because the Engine uses the Dependency Manager in such a way as to use either workflow or deferred processing, then a case that is affected by a change in data can be either:

- Reassessed within a workflow.
- Reassessed within a deferred process.
- Before the execution of the deferred process, if reassessment is initiated within the case itself as a result of manual reassessment, a change in evidence, or the generation of financial payments.

Reassessment Aggregation

The Dependency Manager generates a precedent change set when some data changes, and it attempts to process those changes as soon as they are detected. This strategy can result in some instances where more than one reassessment of a case takes place concurrently, which can sometimes lead to performance or contention issues.

Reassessment Aggregation is a case reassessment process that uses the Cúram Workflow Infrastructure. The aim is to coordinate different entry points into the reassessment process to ensure that only one process reassesses a case at any one time. If multiple processes attempt to reassess the same case at the same time, where possible these reassessment requests are aggregated into one single reassessment, reducing unnecessary reassessments and minimizing the more common conflicts caused by online processing.

By default, the Reassessment Aggregation feature is enabled out of the box. To disable the feature, set the *curam.case.reassessment.aggregation* application property to "NO". When disabled, reassessments are not aggregated and are invoked using Deferred Processes.

For any Deferred Process that invokes a case reassessment, including those outside of the Reassessment Aggregation workflow, diagnostic details of the reassessment are stored in the DependencyTrace table. This information is used in the status messages on the Current Determination page to indicate when reassessment last took place on a case, when the reassessment is deferred to batch, when there is a reassessment in progress, and when there is a reassessment in progress with some changes pending. See Diagnostics on page 188.

The Workflow

The process of ensuring that only one process will reassess a particular case at any given time is controlled by a Cúram Workflow. The Reassessment Aggregation workflow uses two tables to coordinate and monitor the aggregated reassessments of cases: ReassessmentQueue and ReassessmentQueueControl.

Source Location

`EJBServer\components\core\workflow\ReassessmentAggregation_v1.xml`

Enactment

When the Dependency Manager processes a precedent change set, it will find the unique set of dependents affected by any of the items in the precedent set and instruct each affected dependent to recalculate itself. Each type of dependent in the system has a registered 'dependent handler' which, when invoked by the Dependency Manager, is responsible for taking the appropriate steps to recalculate a dependent. If any of the dependents affected are a case assessment determination then the appropriate step is to request that the Engine reassesses the case.

If Reassessment Aggregation is enabled, the case assessment determination dependent handler will delegate responsibility for reassessing cases by invoking new instances of the Reassessment Aggregation workflow. At this point the handler will add entries to the ReassessmentQueue table with details of the cases to be reassessed and the precedent change sets involved.

Default Behavior

The process of ensuring that only one process will reassess a particular case at any given time takes place through the following workflow activities:

1. Claim Queue; during this activity the workflow attempts to claim responsibility for the case by inserting a new 'Pending' entry onto the ReassessmentQueueControl table if there is not already a responsible workflow for that case.
2. Reassess Case; at the next stage, the list of 'Pending' precedent change sets on the ReassessmentQueue table is stored in memory and the case is reassessed using normal CER reassessment logic. After successful reassessment, and storing of the determination decision, the change sets read at the start of the transaction are marked as 'Complete' in the ReassessmentQueue table.
3. Release Queue; in the final activity, the workflow releases its claim on the queue by marking 'Complete' the ReassessmentQueueControl table entry. This allows any future workflows to claim the queue and reassess the case.
4. The final step is to check if there are any pending entries in the ReassessmentQueue table for the case. If so, the workflow will loop back to the first activity and attempt to claim the queue

once again. This ensures that any precedent change sets generated while a reassessment was already in progress are guaranteed to be processed.

If at any stage during the workflow an error is encountered, the ReassessmentQueue and ReassessmentQueueControl entries will be marked as 'Deferred To Batch' and the workflow will exit. This means that case is written to the currently-open batch precedent change set for future processing as part of the Dependency Manager batch suite. See the *`Cúram Express Rules Reference Manual`*.

The Database Tables

The database tables below are used by the Reassessment Aggregation workflow to control the aggregation of reassessments for a case:

- ReassessmentQueue
- ReassessmentQueueControl

ReassessmentQueue

Each record in this table represents a request to reassess a case. Multiple requests to reassess the same case may be aggregated into a single reassessment.

Table 57: Population of common ReassessmentQueue data

Attribute Name	Value
`reassessmentID`	Unique ID assigned by the system.
`caseID`	Identifier of the case which needs reassessment.
`creationTS`	The date and time that the reassessment was queued.
`completionTS`	The date and time that the reassessment was completed.
`statusCode`	The current status of the reassessment request for this case: • "Pending" when the reassessment has been queued but not yet processed. • "Complete" when the reassessment has been completed successfully. • "Deferred To Batch" if an error was encountered during reassessment.
`precedentChangeSetID`	The ID of the precedent change set which triggered this reassessment, if triggered by the Dependency Manager.
`queueControlID`	The ID of the ReassessmentQueueControl table entry which claimed ownership of this reassessment.

ReassessmentQueueControl

Each record in this table represents a case for which a reassessment was started - and optionally finished. A case may appear in this table many times, indicating that it was reassessed many times. However only one such entry may have a status of 'Pending', meaning that only one process may be reassessing that case at any one time.

Table 58: Population of common ReassessmentQueueControl data

Attribute Name	Value
`reassessmentQueueControlID`	Unique ID assigned by the system.
`caseID`	Identifier of the case which a workflow has claimed responsibility for reassessing.
`statusCode`	The current status of the reassessment workflow for this case: • "Pending" when a case has been claimed and a workflow has taken responsibility for the reassessment of that case. • "Complete" when the workflow has been completed successfully. • "Deferred To Batch" if an error was encountered during reassessment. • "Backed Off" if the case could not be claimed because another workflow was already responsible for reassessing the case.
`lockTS`	The date and time that the reassessment workflow claimed the case for processing.
`completionTS`	The date and time that the reassessment workflow was completed.
`workflowID`	The ID of the ProcessInstance table entry representing the Reassessment Aggregation workflow instance which controlled this reassessment.

Bulk Reassessment

Certain types of system-wide change can affect many or all of the product delivery cases on the system.

This section describes these types of system-wide change and the facilities available to you to handle the effects of system-wide changes, by identifying and reassessing the affected product delivery cases.

Types of Change that Cause Bulk Reassessment

This section describes the types of change that the Engine treats as causing bulk reassessment, i.e. where the expected effect of the changes is *not* limited to a low number of cases (unlike case-level reassessment - see Case-level reassessment on page 171).

The effects of any of these types of change are handled in batch processing, and so require the execution of batch jobs to be scheduled, either on a regular schedule provided by your third-party scheduling software, or on an ad hoc basic by being manually run by system operator staff.

The Engine treats any change to the following system-wide data as "bulk change" potentially affecting a large number of cases, and thus requiring bulk reassessment processing:

- Product configuration;
- CER Rules used by the Product;
- Rule Object Data Configurations; and
- Rate Tables.

The following sections describe these types of change in more detail, and explain the lifecycle of each type of change. In general terms, each type of change can be worked on by an administrator without causing any effects on the system, followed by a "publish" action where the completed changes start to take effect and are available for case processing.

> **Tip:** When changes to any of these types of data are published, an informational message (advising that a system-wide change has occurred and that bulk reassessment processing is required) is shown on the publication screen and also written to the application logs.
>
> It may be useful to monitor the application logs for this informational message if you choose to run bulk reassessment processing on an ad hoc basis (i.e. only when a bulk change has occurred).

Product Configuration

Any change to the configuration of a product has the potential to affect assessment determinations for the product's cases.

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

In particular, changes to the product periods for your product may affect the product's cases. Product period changes include:

- Adding a new product period (for instance, to implement a change in legislation, using the multiple product periods approach, see Multiple Product Periods for Your Product on page 201);
- Removing an existing product period;
- Changing the start and/or end date of an existing product period;
- Changing the name[11] of the eligibility/entitlement rule class for an existing product period;
- Adding, changing, or removing the name of the key decision factor rule class for an existing product period; and/or
- Adding, changing, or removing the name of the decision details rule class for any display category on an existing product period.

The administration application contains a "sandbox" area where an administrator can accumulate changes to a product (including its product periods) before choosing to "publish" those changes, at which point the changes to the product will start to affect product delivery cases. Unpublished product changes have no effect on case processing.

There is additional processing required if a change is made to the Product's reassessment strategy. See Reassessment Strategy on page 206.

CER Rules used by your Product

Any change to any part of these CER rule sets, for any product period on the product, has the potential to affect assessment determinations for the product's cases:

---

[11] This change refers to changing which rule class the product period "points at" for eligibility/entitlement calculations; i.e. changing the product period from pointing to one rule class to instead point to another; similarly for key decision factors and decision details rule classes.

Changes to the CER rules themselves are described below.

- the rule set containing the eligibility/entitlement rule class;
- the rule set containing the key decision factors rule class (if configured);
- the rule set (s)containing any of the decision details rule classes(if configured); and/or
- any other rule set containing any rule attribute that was encountered during the calculation of a case's determination, e.g. those on "calculator" rule classes and "data" rule classes for custom entity and evidence types, which may be stored in "common" rule sets separate from the rule sets containing the rule classes named by your product period(s).

Note: Other processing in the application (outside the Engine) may also rely on CER rule sets, and so it is possible that CER rule sets are being changed for reasons unrelated to case assessments.

The Dependency Manager does not know which CER rule sets do or do not affect cases, and so for *any* change in CER rule sets, the Dependency Manager will treat the change as one that might affect cases, but will simply identify that no product delivery cases are affected.

The administration application contains a "sandbox" area where an administrator can accumulate changes to CER rules before choosing to "publish" those changes, at which point the changes to CER rules will start to affect product delivery cases. Unpublished rule set changes have no effect on case processing.

Rule Object Data Configurations

Any change to any configuration for any of the configurable rule object data configurations has the potential to affect product delivery cases, because the entities and evidence to be used by CER during the execution of rules may have changed.

Note: Other processing in the application (outside the Engine) may also rely on rule objects created by configurable rule object data configurations, and so it is possible that configurations are being changed for reasons unrelated to case assessments.

The Dependency Manager does not know which configuration changes do or do not affect cases, and so for *any* change in data configurations, the Dependency Manager will treat the change as one that might affect cases. If the data configuration changes are unrelated to case assessments, then the Dependency Manager will simply identify that no product delivery cases are affected.

The administration application contains a "sandbox" area where an administrator can accumulate changes to rule object data configurations before choosing to "publish" those changes, at which point the changes to the data configurations will start to affect product delivery cases. Unpublished data configurations changes have no effect on case processing.

Rate Tables

Any change to a rate table (which is configured to populate `RateCell` rule objects) has the potential to affect product delivery cases, typically when the value of an existing version of a rate is changed or a new effective period of a rate table comes into effect (see "Implementing Rate Tables" in the *Cúram Integrated Case Management Configuration Guide*).

Note: Other processing in the application (outside the Engine) may also rely on the values stored in rate tables, and so it is possible that rate tables are being changed for reasons unrelated to case assessments.

The Dependency Manager does not know which rate tables do or do not affect cases, and so for *any* change in rate table data, the Dependency Manager will treat the change as one that might affect cases. If rate table changes are unrelated to case assessments, then the Dependency Manager will simply identify that no product delivery cases are affected.

In contrast to the other types of data changes described above, there is no system-wide "publication" step for rate table changes.

However, the Engine contains a special "Apply Changes" option which allows an administrator to choose when the changes made to rate tables will start to affect product delivery cases. Until an administrator chooses this option, rate table changes have no effect on case processing (for CER-based cases). Other processing outside the Engine will see the rate table changes immediately, though.

Approaches to Identifying and Reassessing All Affected Cases

Requirements for Bulk Reassessment

When identifying and reassessing cases in response to a system-wide change, typically the requirements for processing fall into one of these categories:

- **Consistency**

  All cases affected by the change must be identified and reassessed (i.e. all reassessments which are *sufficient* to make those cases consistent with the change). It is not acceptable for any case affected by the change to be "missed" during batch processing.
- **Efficiency**

  Only cases affected by the change should be identified and reassessed (i.e. only those reassessments which are *necessary* should occur). It can be wasteful to spend system time either reassessing cases which are not affected and/or reassessing any particular case more than once.
- **Concurrency**

  It must be possible to reassess cases in parallel batch streams (to promote scalability). It must be possible to reassess cases in batch while the online application is being used (to avoid the necessity for system downtime for online users).
- **Business-specific control and processing**

  You may have your own business-specific requirements to control reassessment of cases and/or perform additional processing, e.g.:

  - to reassess cases in a particular order, e.g. by surname of the claimant;
  - to reassess only a particular subset of cases, e.g. those for claimants who have social security numbers in a particular range; or
  - to perform additional business processing for each case, e.g. to send out special correspondence with each reassessed case which is pertinent only to bulk reassessment on this occasion, e.g. an explanatory leaflet sent to each claimant who has a case affected by a particular piece of legislation change.

In practice, there are trade-offs to be made to meet these requirements. Driving the Identification of Affected Cases on page 179 describes the various approaches available for handling a bulk change, i.e. for identifying the cases identified so that they can each be reassessed.

Multiple Reassessments during a Case's Lifetime

At this point it's worth pausing to recall what happens each time a case is reassessed:

- the Engine invokes CER to calculate a new determination result for the case;
- the Engine retrieves the existing determination result for the case from the database;
- the Engine compares the new determination against the existing determination, and *only if the new determination is "different"*:
    - the Engine supersedes the existing determination result and stores the new determination result on the database;
    - the Engine updates the case's financial schedule;
    - the Engine identifies whether there have been any over- or under-payments and if so takes corrective action.

Bulk case reassessment is no different from online case lifecycle processing in this regard - i.e. processing can reassess a case any number of times, but it is only when the new assessment determination result differs from the existing assessment determination result that a new determination result is stored and potentially financial impacts are stored (such as changes to the financial schedule, and possibly the identification of over- or under-payments and corrective actions).

As such, a "needless" reassessment which results in no change to the existing assessment determination is not as expensive (in system terms) as a "necessary" reassessment which results in a change to the determination; but on the other hand performing a "needless" reassessment is more expensive than avoiding it. Needless reassessments have no business impact on the case, though. You should consider your performance requirements to determine to what extent needless reassessments are tolerable in your system and bear that in mind when choosing your approach to bulk reassessments.

There are a number of common processing points in the application which will cause a case to be reassessed:

- case lifecycle events - e.g. if the case is closed, and then reopened, and then reactivated, the case will be reassessed at reactivation time;
- when a case worker chooses to manually reassess the case;
- when case-level reassessment occurs, e.g. in response to changes in evidence or personal details;
- if the financials batch programs are configured to force a reassessment of each case prior to generating financials for the case; and
- bulk case reassessment (the subject of this section).

Once system-wide changes to data have been made, then for any particular product delivery case, the first processing point to reassess the case (e.g. one of the events listed above) will calculate a new determination result which takes into account all the system-wide changes to data, and any subsequent processing point will calculate an identical determination result and take no further action (assuming that there have been no other data changes which affect the case in the meantime).

Important: Because the first processing point to reassess the case takes into account all system-wide changes to data, you must consider carefully:

- when to publish the system-wide changes;
- how long after publication the batch processing will run to identify and reassess affected cases;
- whether case workers should be allowed to view and maintain[12] cases during the lag between publication of the system-wide changes and completion of the batch processing; and
- how your business processes will cope with any case that would have had a new determination stored by the batch processing, except that some online processing (e.g. a manual reassessment by a case worker) caused a new determination (taking into account the published system-wide changes) to be stored already.

This last point is especially important if you require to run additional business processing during your batch identification and reassessment of cases (e.g. to send out a particular type of correspondence), as online processes such as a manual reassessment by a case worker will not automatically include that additional processing. In such circumstances it may be important to schedule the publication of the system-wide changes and the full batch processing (to completion) during downtime when caseworkers do not have access to the system, and/or make your batch business processing tolerant to the situation that a case may have already been reassessed by an online action.

Assuming that you have no additional business processing that relies on being the first processing point to reassess a case after system-wide changes have been published, then in general it is safe to run bulk reassessment batch processing concurrently with the online system. You should note though that:

- online users may experience a performance degradation while batch processing is ongoing; and
- if an online user performs an action which leads to a case being reassessed, at the same time that the bulk reassessment batch processing is attempting to reassess the same case, then it is possible for either the online and/or batch transaction to fail:
  - the online user's transaction may fail and the user advised to try again;
  - a deferred transaction to reassess the case triggered from the online user's action may fail and will be retried a set number of times; and/or
  - the batch "chunk" may fail and will be automatically retried.

Driving the Identification of Affected Cases

After system-wide changes to data have been made, you must run batch processing to identify the cases affected, and reassess each case.

---

[12] For example, you might have to carefully communicate to your case workers that a system-wide change has occurred, and to let them know that it will be some time until all affected cases have been reassessed, but that urgent cases can be manually reassessed in the meantime.

Depending on your business needs, this situation may be perfectly acceptable, or on the other hand be needlessly confusing for case workers.

The algorithm(s) that you choose to identify cases will depend on your business requirements. In general there are these types of case-identification algorithm to choose from (or possibly combine):

- **Bottom-up**

  Identify cases that are known by the system to be affected by the change, i.e. use the dependency records stored in the Dependency Manager; and/or
- **Top-down**

  Identify cases by some facet of the case, e.g. all active cases for a particular product, or all cases for claimants in a particular range of social security numbers.

The application includes an implementation of the bottom-up algorithm in the form of the batch suite provided by the Dependency Manager (see the *`Cúram Express Rules Reference Manual`*). The batch suite uses the dependency records to identify cases potentially affected by system-wide changes in data and reassesses them.

Important: The Dependency Manager batch suite has the capability to reassess cases, but it also has the capability to recalculate other types of dependents, such as Advice.

If you choose not to use the Dependency Manager batch suite to reassess cases, you must still use the batch suite to recalculate other types of dependents.

Tooling has been provided to assist with running the Dependency Manager batch suite, please see the 'Dependency Manager Batch Tooling' section of the 'Cúram Operations Guide' for more information.

The application also includes a sample implementation of a top-down algorithm in the form of the `CREOLEBulkCaseChunkReassessmentByProduct` batch process (see the *`Cúram Operations Guide`*), and instructions for writing your own batch process to implement your own top-down algorithm are included below (see Writing your own Bulk Reassessment Batch Process on page 184).

When scheduling batch processing to reassess cases affected by your published system-wide changes, you can choose to schedule one or both of:

- an implementation of a top-down algorithm, either that included with the application or your own custom batch process; and/or
- the bottom-up algorithm included in the Dependency Manager batch suite.

You can choose the most appropriate approach each time you publish one or more system-wide changes, or you can choose to use the same approach each time. Any custom batch processing you require to support bulk case reassessment will need to be implemented and deployed to your production environment, of course.

The benefits and limitations in the choices of approach are outlined in the table below:

Table 59: Benefits and Limitations to Bulk Reassessment Approaches

Approach	Benefits	Limitations
Use the Dependency Manager batch suite only	• The implementation is included with the application - no custom processing needs to be implemented. • The Dependency Manager batch suite provides robust identification of all cases potentially affected by system-wide changes made through the application's APIs. If system-wide changes to data are made outside of the application's APIs, e.g. by an SQL script, then the Dependency Manager cannot automatically identify affected cases. If system-wide changes to data are made outside of the application's APIs, e.g. by an SQL script, then the Dependency Manager cannot automatically identify affected cases. • The Dependency Manager batch suite must be run anyway to recalculate other dependents such as Advice.	• You cannot control the order in which the Dependency Manager reassesses cases. • You cannot perform any additional business processing at the point at which the Dependency Manager batch suite reassesses the case. If you have requirements to perform specific business processing for all the cases that were identified and reassessed by the Dependency Manager and/or those for which new determinations were recorded, you must satisfy yourself that you can identify these cases in some way after the Dependency Manager batch suite has completed and implement your own custom batch program to perform the required post-reassessment business processing.

Approach	Benefits	Limitations
Use an implementation of a top-down case identification/ reassessment algorithm, followed by a run of the Dependency Manager batch suite	• You get to "prioritize" the processing of certain cases (those identified by your top-down algorithm) ahead of any other cases and any other dependent types (such as Advice). • You control the order in which cases are reassessed. • For cases reassessed by your top-down case reassessment implementation, you can perform additional business processing at the point of reassessment. • Any case that your top-down algorithm fails to identify, but which is affected by the system-wide change, will still be identified and reassessed by the Dependency Manager batch suite, so by the end of the batch run all affected cases will have been identified and reassessed. There may be edge cases not obvious during the design of your case identification algorithm. For example, a rate table may be primarily used in the determination calculations for a particular product, and thus it would be possible to drive the identification of cases to reassess by finding all active cases for that product. If, unbeknownst to the administrator, a rules designer has reused that rate table in way that it is used by a handful of unusual cases for another product, then these cases will *not* be identified by the naive "all active cases for a product" algorithm run against the main product which uses the rates. • You can schedule this "clean-up" run of the Dependency Manager batch suite quite some time after the run of the top-down batch processing which identified and reassessed your "priority" cases. For example, you might wish to schedule the "priority" (top-down) case processing to run overnight starting on the evening when the system-wide changes were published, but defer the "clean-up" (bottom-up) case processing until the next weekend. • The Dependency Manager batch suite must be run anyway to recalculate other dependents such as Advice.	• You will have additional development activity to implement and test your custom case identification algorithm and/or post-reassessment business processing (unless you are using the `CREOLEBulkCaseChunkReassessmentByProduct` batch process included with the application). • Some cases identified by the Dependency Manager batch suite will have already been identified and reassessed by your top-down case identification/reassessment algorithm, so some of the reassessments performed by the Dependency Manager may turn out to be needless. • If a case is missed by your top-down algorithm and is instead identified and reassessed by the Dependency Manager batch suite, then the Dependency Manager batch suite cannot perform any additional processing that your top-down algorithm might have done at the point where it would have reassessed the case. If you have requirements to perform specific business processing for the cases that were cleaned-up by the run of the Dependency Manager batch suite, then you must satisfy yourself that you can identify these cases in some way after the Dependency Manager batch suite has completed and implement your own custom batch program to perform the required post-reassessment business processing.

Approach	Benefits	Limitations
Use an implementation of a top-down case identification/ reassessment algorithm only	• You get to "prioritize" the processing of certain cases (those identified by your top-down algorithm) ahead of any other cases and any other dependent types (such as Advice). • You control the order in which cases are reassessed. • For cases reassessed by your top-down case reassessment implementation, you can perform additional business processing at the point of reassessment.	• You will have additional development activity to implement and test your custom case identification algorithm and/or post-reassessment business processing (unless you are using the `CREOLEBulkCaseChunkReassessmentByProduct` batch process included with the application). • You take sole responsibility for the accurate and complete identification of cases affected by the system-wide data change. Any case that your algorithm fails to identify will not be reassessed by the batch processing run, and its stored determination will not be consistent with the system-wide data change until such time as the case is reassessed for some other reason. • The Dependency Manager batch suite must be run anyway to recalculate other dependents such as Advice. Your system operators must be instructed explicitly *not* to run the Dependency Manager batch suite for dependents of type "case assessment determination".

Note: If system-wide changes to data are made outside of the application's APIs, e.g. by an SQL script, then the Dependency Manager cannot automatically identify affected cases.

Note: There may be edge cases not obvious during the design of your case identification algorithm.

Note that it is always possible for the Dependency Manager to identify a case for reassessment, but having reassessed that case the Engine finds out that the reassessment turned out to be needless, due to the granularity at which dependency records are stored for case determination dependents. Examples where such a needless reassessment might occur are:

- a change to column value on an entity row, where other unchanged columns for that entity row have been used as the input to determination calculations;
- a change to a common CER rule set which is used to calculate determinations, and also for some other purpose such as Advice, but the changed rules are not accessed during determination calculations;
- a change to a rate (such as for income thresholds) which does not affect the overall determination result for a case.

> **Note:** There are only two options regarding a reassessment which turns out to be needless:
>
> - either the system performs the reassessment, and only once a new determination is calculated can the system discover that the reassessment turned out to be needless, at the risk of using up processing time; or
> - a human outside the system uses his or her business knowledge to implement processing to identify cases in such as way as to eliminate or minimize needless reassessments, at the risk of human error (i.e. that needless reassessments still occur for some cases, or, more seriously, that some necessary reassessments were *not* identified.
>
> You should weigh up these unavoidable risks as part of your decision as to which approach to use for bulk reassessment processing.

Reassessment Processing

Once bulk processing (whether bottom-up or top-down) has identified cases to reassess, each case can be reassessed by streamed batch processing.

> **Important:** Be careful when implementing extra business processing that occurs at reassessment time, either using:
>
> - hook points provided by the Engine; and/or
> - custom business processing in your implementation of a top-down algorithm.
>
> Any extra business processing that causes database writes may cause new precedent change items to be recorded in the batch precedent change set. A run of the Dependency Manager batch suite will be required to process these new precedent change items.

Writing your own Bulk Reassessment Batch Process

If you decide that you require to implement your own algorithm for identifying cases to reassess, and/or require custom business processing to occur at the time that each case is reassessed, then you must implement your own batch process for bulk reassessment. This section describes how to implement such a custom batch process.

If on the other hand you decide to use the bulk reassessment batch processes included with the application, you can skip this section.

The `CREOLEBulkCaseChunkReassessmentByProduct` Batch Process

This process, included with the application, identifies and reassesses all active cases for a given product.

The `curam.core.sl.infrastructure.assessment.impl.CREOLEBulkCaseChunkReassessmentByProdu` class is a "Chunker" batch job which takes a *productID* as a standard batch parameter and identifies all product delivery cases for that product, with a status of "Active".

The `curam.core.sl.infrastructure.assessment.impl.CREOLEBulkCaseChunkReassessmentStream` class is the corresponding "Stream" job.

The behavior of the batch process can be configured using the following environment variables:

Table 60: Environment Variables for the `CREOLEBulkCaseChunkReassessmentStream` *Batch Process*

Environment variable name	Description	Default value
curam.batch.creolebulkcasechunkreassessment.chunksize	The number of cases in each chunk that will be processed by the CREOLE Bulk Case Chunk Reassessment batch program.	500
curam.batch.creolebulkcasechunkreassessment.dontrunstream	Should CREOLE Bulk Case Chunk Reassessment batch program sleep while waiting for the processing to be completed (rather than run a stream in its context).	NO
curam.batch.creolebulkcasechunkreassessment.chunkkeywaitinterval	The interval (in milliseconds) for which the CREOLE Bulk Case Chunk Reassessment batch program will wait before retrying when reading the chunk key table.	1000
curam.batch.creolebulkcasechunkreassessment.unprocessedchunkwaitinterval	The interval (in milliseconds) for which the CREOLE Bulk Case Chunk Reassessment batch program will wait before retrying when reading the chunk table.	1000
curam.batch.creolebulkcasechunkreassessment.processunprocessedchunks	Should CREOLE Bulk Case Chunk Reassessment batch program process any unprocessed chunks found after all the streams have completed.	NO

Steps to Implement your own Bulk Reassessment Batch Process

Batch processing must be written using the chunked batch processing architecture (see the *`Curam Batch Performance Mechanisms`* document).

Write these batch programs:

- a "Chunker" batch job which identifies[13]the list of cases and passes them, along with appropriate control parameters to the `BatchChunker` to divide them into smaller lots. This process of identifying the cases can be controlled by whatever parameters are required, such as Product ID, case status, etc.
- a "Stream" job which takes one of these chunks of work and performs full reassessment of each case contained in it.

You must decide on appropriate metrics to capture during the stream processing, such as the number of cases processed, and/or the number of cases reassessed which did/did not result in a changed determination result. Your chunker and streamer must share data structures so that the stream processing can capture metrics and the chunker can accumulate them into its report.

The "Stream" job must perform a full reassessment of each CER-based case as follows[14]:

---

[13] You should ensure that each case is identified at most once.

For simple database queries this is unlikely to be challenging; however for non-trivial queries you might consider using SQL's `DISTINCT` keyword, e.g. `SELECT DISTINCT(caseID) FROM....`

[14] You must also include:

```
// class member
            @Inject
            protected DeterminationCalculatorFactory
              determinationCalculatorFactory;

            public void yourBatchMethod(...yourparameters...)
              throws AppException, InformationalException {

            // process an identified case
              final long caseID = ...;

              final DeterminationCalculator
 determinationCalculator =
                determinationCalculatorFactory
                  .newInstanceForCaseID(caseID);

              final DetermineEligibilityKey
 determineEligibiltyKey =
                new DetermineEligibilityKey();

              determineEligibiltyKey.caseID = caseID;

             /*
              * reassess the case and determine whether the
 decision has
              * changed
              */
              final boolean decisionChanged =
               determinationCalculator
                .hasDecisionChanged(
                  determineEligibiltyKey,

 CASEASSESSMENTDETERMINATIONREASONEntry.SOMEREASON);

            }
```

Figure 2: Code example to reassess a CER-based case

Bulk Reassessment for Multiple Simultaneous Changes

It is possible for one business change to require technical changes to a number of system-wide data artefacts. For example, a change in legislation may involve all of:

- changes to existing CER rule sets associated with your product;
- division of the lifetime of the product into several product periods, together with new CER rule sets for the new product periods;
- the capture of new types of evidence which map to new rule classes (i.e. new data configurations for rule object converters); and
- changes in benefit rates.

- processing to capture the metrics for your batch program; and
- appropriate error handling to communicate with the chunked batch processing architecture's support for skipping cases.

This extra code is beyond the scope of this guide.

There are separate publication mechanisms for products, CER rule sets, data configurations and rates, and so changes to different types of system-wide data cannot be published in a single action. However, when bulk reassessment is run, each case affected will be identified only once and the reassessment of each case will take into account all the system-wide changes that have occurred since the case was last assessed.

As such, when you prepare to publish a number of system-wide data changes (whether those changes are inter-related or not), you should consider carefully when bulk reassessment should be run. Depending on your business needs, you might want to run bulk reassessment after each publication or instead hold off until a number of publications have occurred.

> **Note:** When a case is reassessed by the Dependency Manager batch suite, the system chooses a "reassessment reason" for any new determinations that are stored. Each determination can only show a single reassessment reason, so in the situation where there are multiple system-wide changes published which each affect a case, only one related reason will show on that case.
>
> If the case is reassessed using your own top-down algorithm, then that algorithm is responsible for specifying an appropriate reason to store on any new determinations.

Scheduling

Once you have decided your approach for identifying and reassessing cases in batch processing, you must arrange for that batch processing to execute.

Broadly, you can run batch processing either:

- regularly, on a pre-determined schedule, either manually or through the use of third-party scheduling software; or
- on an ad hoc basis, in response to the publication of system-wide data changes.

The Dependency Manager batch suite is amenable to being executed on a pre-determined schedule, because if there have been no system-wide changes to data (written to the batch precedent change set) then the batch suite will quickly identify that there are no cases to reassess. If you use such a pre-determined schedule, then you can ignore the on-screen and application log messages that advise that bulk reassessment processing is required.

It is not recommended to execute top-down case identification algorithms on a pre-determined schedule because those algorithms will identify cases to reassess regardless of whether there have been any system-wide data changes published.

If you are executing batch processing that uses the chunked batch processing architecture (such as the Dependency Manager batch suite, the `CREOLEBulkCaseChunkReassessmentStream` batch process, or the recommended way to implement your own top-down case identification/reassessment batch process), then you have some flexibility when manually executing chunker and streamer processes:

- during the reassessment phase you can start up more instances of the streamer batch processes in order to spread the reassessment work across more physical machines;
- if you find that you need some physical machines for other purposes (e.g. to run the online application in parallel with batch processing), you can manually terminate one or more streamer processes and any uncompleted work for those terminated streamer processes will be automatically picked up by one of the remaining streamer processes; and/or
- if you find that you need to pause the entire batch processing (e.g. you need all your physical machines to be dedicated to the online application) then you can manually terminate the

chunker process and all the streamer processes; when you subsequently re-run the chunker process it will continue running from the point where it left off.

Tip: If you have configured your chunker process to automatically perform streamer processing once the case identification phase is over and the case reassessment phase has begun, *and* you wish to run multiple parallel streamer processes to spread the reassessment load across your physical machines, then you should start your streamer processes *before* starting your chunker process. The streamer processes will simply wait until the chunker process has completed its case identification phase and the case reassessment phase has begun.

If you start your streamer processes *after* the chunker processing, then in a situation where the chunker process identifies only a few cases, it is possible for some of the streamers (including the chunker process itself) to complete reassessment processing on all the cases identified, and the overall batch processing would complete. If this happens, then the other streamer processes will have no work to do but will wait until the chunker process is next run, which could be quite some time later; from an operational perspective, these other streamer processes are just hanging and would need to be manually terminated, which is not ideal under normal operational procedures.

If you are executing the Dependency Manager batch suite, then you must run the `PerformBatchRecalculationsFromPrecedentChangeSet` streamed batch process once per dependent type. You can choose the order of these runs - for example, you may decide that it is more urgent to have your cases reassessed in response to a system-wide data change than it is to have advice recalculated.

If your batch run includes both a top-down case identification/reassessment algorithm and a run of the Dependency Manager batch suite (see Driving the Identification of Affected Cases on page 179), then typically you should run the top-down case identification/reassessment algorithm first so that your priority cases are identified and reassessed.

If your batch run includes the execution of the `ApplyProductReassessmentStrategy` batch process (see Reassessment Strategy on page 206) then typically there are no ordering constraints - but note that cases which could previously not be reassessed will only be able to be reassessed (and identifiable by the Dependency Manager batch suite) once the `ApplyProductReassessmentStrategy` batch process has completed.

If you are planning to publish multiple changes to system-wide data (see Bulk Reassessment for Multiple Simultaneous Changes on page 186), then you may choose to hold off on manually running your preferred approach to case identification/reassessment (or suspend your regular batch schedule, if you have one) until all those system-wide data changes are published. In this way, each case will only be identified and reassessed once in response to the combined system-wide data changes.

Diagnostics

The Dependency Manager can support the capture of additional information to assist with troubleshooting and analysis. This information can be written to the DependencyTrace database table and/or the server log. Suggested uses of this data include:

- being able to link CreoleCaseDetermination records to the PrecedentChangeSet which caused them to be created;

- being able to identify which precedent change sets caused subsequent change sets to be created;
- being able to identify when a precedent change set causes a rules execution without changing a determination.

To enable writing to the DependencyTrace table, create and set the *curam.dependency.monitor.database* application property to "True".

To enable writing to the server log file, create and set the *curam.dependency.monitor.log* application property to "True".

Table 61: Population of common DependencyTrace data

Attribute Name	Value
`dependencyTraceID`	An ascending integer comprising the primary key for this table.
`pcsid`	The precedentChangeSetID of the current precedent change set, if relevant.
`pcsid2`	The precedentChangeSetID of the secondary precedent change set, if relevant.
`timestamp1`	Timestamp when log entry was created.
`status1`	Status code of the precedent change set, if applicable.
`status2`	Secondary status code of the precedent change set, if applicable.
`instdataid`	The deferred processing instance data id, if relevant.
`tracetype`	Indicates the type of trace message: • REASSESS - a case was reassessed. • CADETER_SU - a determination, identified by determinationID, was superseded. • CADETER_IN - a determination was inserted. • CADETER_UN - a determination was unchanged as a result of the recalculation. • PCSBAT_SIZ - a precedent change set was pushed to batch because of its size; the number of affected determinations was above the threshold. • PCSBAT_ERR - a precedent change set was pushed to batch because of an error. • BATCH_FAIL - the case failed to be reassessed by the Dependency Manager batch suite. • PCS_ADD - a new precedent change set, identified by pcsid2, was added while processing the change set identified by pcsID.
`txtype`	Indicates the transaction type: • o - online • d - deferred • b - batch
`txid`	The id of the current transaction.
`ticketid`	The deferred processing ticket id for the transaction, if relevant.
`instanceid`	The deferred processing instance id for the transaction, if relevant.
`caseid`	The case ID, if available.
`determinationid`	The CreoleCaseDetermination identifier, if relevant.

Attribute Name	Value
`count1`	General purpose count, if needed.
`count2`	General purpose count, if needed.

> **Note:** For a record with `tracetype` of 'REASSESS' or 'BATCH_FAIL', a row will always be recorded in the DependencyTrace table, regardless of the *curam.dependency.monitor.database* application property. If the reassessment was processed with the Reassessment Aggregation infrastructure, then the `ticketid` field will be populated with the reassessmentQueueControlID from the ReassessmentQueueControl table and the `instanceid` field with the workflowID which handled the reassessment. If Reassessment Aggregation was not responsible for the reassessment, then the relevant Deferred Process information will instead be stored in the `instdataid` and `ticketid` fields.

If writing diagnostics to the system log is enabled, each log file message consists of the string 'DEPENDENCY_DIAG' followed by the following fields from the DependencyTrace table:

- dependencyTraceID
- txid
- tracetype
- caseid
- pcsid
- pcsid2
- instdataid
- instanceid
- ticketid
- txtype
- determinationid
- status1
- status2
- count1
- count2

Listed below are some sample messages from the system log:

- DEPENDENCY_DIAG 769,768,CADETER_SU,0,0,0,0,0,0,o,7736621209869090816,,,0,0,
- DEPENDENCY_DIAG 770,768,CADETER_IN,0,0,0,0,0,0,o,-1413567333040914432,,,0,0,
- DEPENDENCY_DIAG 776,775,PCSBAT_SIZ,5576019288638095360,-6609769675712626688,0,526,0,0,d,0,,,1,0,
- DEPENDENCY_DIAG 3584,631,CADETER_UN,6256237104964042752,0,4608,4609,d,8028229285741330432,,0,
- DEPENDENCY_DIAG 771,768,REASSESS,4188910603407982592,0,0,0,0,0,o,0,CADR7,RM1,6591,0,
- DEPENDENCY_DIAG 781,779,REASSESS,5576019288638095360,0,0,0,28708,513,d,0,CADR6,RM1,192,0,

The above messages indicate the following events, respectively:

- CreoleCaseDetermination 7736621209869090816 was superseded by an online transaction
- CreoleCaseDetermination -1413567333040914432 was inserted by an online transaction

- PrecedentChangeSet -6609769675712626688 generated on case 5576019288638095360 was deferred to batch due to its size
- PrecedentChangeSet 6256237104964042752 left existing CreoleCaseDetermination 8028229285741330432 unchanged during a deferred processing transaction
- Reassessment occurred for case 4188910603407982592 in an online manual reassessment that took 6591ms
- Reassessment Aggregation handled the reassessment of case 5576019288638095360 with control queue ID 513 and workflow ID 28708

1.11 Incremental Design and Evolution

Introduction

Products can be complex. Their requirements may come from complex legislation and/or policy documents. The explanations available to case workers may need to be very detailed.

Over time, products can become even more complex, as legislation and/or policy is changed.

The Engine supports a rich set of features that allow the implementation of even the most complex of products; however, when starting off the implementation of your product, the complexity of your product combined with the richness of the Engine's features can together be quite daunting.

This chapter offers some advice on how to get started with your product's initial implementation, and describes the options available when your implemented product is required to evolve in the future.

Starting with Rule Sets Included with the Application

A default eligibility and entitlement rule set is automatically created for a benefit product that is created via the dynamic product wizard. You are also free to create your product's rules "from scratch" by starting with empty rule sets. However, if you have purchased a solution, you may wish to use rule sets from these solutions as a starting point for your product.

This section describes the process whereby you can clone certain existing rule sets to come up with rule sets that you are free to customize.

How Rule Sets Inter-relate

The CER rules structure is made up of CER rules artifacts and the relationships between them. Understanding the structure of CER rules is key to the cloning process.

CER Rules Artifacts - Technical Dependencies

CER rules are composed of rule sets which contain rule classes which, in turn, contain rule attributes. CER rules are exposed at the level of rule sets. Each rule set is an independently delivered unit, however there may be interdependencies between the rule classes and rule attributes both within a rule set and across rule sets.

So while a rule set does not have any direct dependencies outside of itself, it can have such dependencies based on the rule classes and rule attributes it contains.

Dependency Types

CER supports both build time dependencies which are based on defined relationships, these are covered in more detail in the following sections, and runtime dependencies which are free form, and hence need to examined in detail when used rather than following well defined patterns.

Rule Class Dependencies

A rule class can extend another rule class; this is a form of implementation inheritance for rule classes. Each rule class can only extend at most one other rule class (single inheritance). The extended rule class can be from the same rule set or from a different rule set from the extending rule class.

Rule Attribute Dependencies

A rule attribute can define its type to be a rule class, or a collection (such as a list or Timeline) of a rule class, or its type can be a built in Java type. The rule class used for an attribute type can be from the same rule set or from a different rule set from the rule attribute.

A rule attribute can be derived:

- as a fixed (externally `<specified>`) value; or
- using rules in the following ways (which can be combined into an arbitrarily complex expression)
  - a direct reference to a rule attribute (using the `<reference` expression), with the rule attribute possibly on a rule class in a different rule set;
  - using the `<readall>` expression to retrieve instances of a rule class, with that rule class possibly in a different rule set;
  - using the `<create>` expression to create an instance of a rule class, with that rule class possibly in a different rule set; and/or
  - using a list of rule class instances, with that rule class possibly in a different rule set.

CER Rules Artifacts - Logical Categorizations

Logically, appropriately structured CER rules can be thought of as being made up of rules classes for two distinct logical functions:

- **Data**

  Data rule classes are those which are an exact mirror for the data held by the application. For example Income Evidence would be a data rules class.
- **Derived Data/Business Logic**

  All other rule classes can be considered as Derived Data/Business Logic rule classes. For example Monthly Income would be such a rule class.

Additionally each rule class, provided out of the box, can be thought of as containing either:

- **Infrastructure**

  Fixed processing which is relied upon by other processing in the application, and which cannot be altered; or
- **Application**

  Processing which can be cloned and used as the starting point for your rule set work.

Cloning CER Rule Sets

This section describes the process you must follow to clone rule sets included with the application so that you can customize the cloned rule sets to meet your product's needs.

If you need to customize any rules element in a rule set included with the application, or if you rely on the current functionality of the version of a rule set included with the application, then you must clone the rule set and all of its dependencies, in terms of application derived data/business logic rules classes (but not infrastructure rule classes).

> **Important:** The application does *not* support the modification of any application-included rule sets "in place".
>
> See the compliancy statement in CER Rule Sets Included with the Application on page 210.

Please refer to the Rule Set Interdependencies developer documentation located alongside the Data Dictionary for more information on which rule sets are Application (clone-able) and which are Infrastructure (not clone-able).

Follow these steps to clone rule sets:

- Identify the rule set(s) that is your entry point into the rules (typically from a rule class).
- For each of these rule sets, follow the arrows on the ruleset inter-dependency documentation, to find any other Application rule sets on which those to customize depend. Repeat this step recursively until there are no more links to Application rule sets.
- For each of the Application rule sets found in the above steps, follow the arrows backwards to pick up any additional Application rule sets that depend on the rule sets found so far. Repeat this step recursively until there are no more links to follow backwards.
- Each of the Application rule sets found during the above steps must now be cloned. Follow the steps below.
  - For each rule set to be cloned, copy its rule set source XML file from *`EJBServer/ components/ component name /CREOLE_Rule_Sets/ someruleset.xml`* to *`EJBServer/components/custom/CREOLE_Rule_Sets/ someruleset Custom.xml`* .
  - For each cloned rule set, create an entry in *`EJBServer/components/custom/ data/initial/CREOLERuleSet.dmx`* (which you must create if it does not already exist). The contents of this entry should be a copy of the entry for the original rule set, with the exception of the 'ruleSetDefinition' attribute. This should be updated to the new location of the cloned rule set in the custom component.
  - Similarly, for each cloned rule set, create an entry in *`EJBServer/components/ custom/data/initial/AppResource.dmx`* (which you must create if it does not already exist). The contents of this entry should be a copy of the entry for the original rule set, with the exception of the 'content' attribute. This should be updated to the new location of the cloned rule set in the custom component.

On completion of rule set cloning, if you wish to add further propagator configuration for your cloned rule sets, you should complete these in the custom directory:

- Propagator configurations should be added to: *`EJBServer/components/ custom / data/ directory /RuleObjectProapgatorConfig.dmx`*
- Propagator configuration XML files should be located in the blob/clob directories within the custom data directories.

Incremental Design

When you are grappling with the complexity of your product, it can be useful to take certain shortcuts so that you can start to see your product "up and running" before it is fully implemented. This section suggests some useful approaches which may help you incrementally design and/or implement your product.

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

> **Important:** There is more to getting a product up and running than is described in this guide. This guide covers only the configuration options specific to CER-based products; for other configuration options and initial set-up tasks, see the `How to Build a Product` guide.
>
> Any short-cuts that you take during the initial development of your product must be recognised for what they are - each short-cut builds up a certain amount of "debt" which must be later repaid. Keep track of which short-cuts you take and plan to place each short-cut with a robust implementation later in your development cycle.

The initial goal of incremental design is to get *something* up and running for your product, even though it is far from fully implemented. This initial version of your product should be able to have cases created against it and determinations made, but those determinations may show fixed eligibility/entitlement information (which does not take into account the case's circumstances) and no key decision factors or decision details.

These chapters earlier in this document describe how to configure your product. The chapters linked below give details for the full implementation of their respective areas:

- **Eligibility and Entitlement Calculations**

  See How to Use It on page 36 in 1.4 Calculating and Displaying Eligibility and Entitlement on page 28.
- **(deprecated) Key Decision Factors**

  See (deprecated) How to Use It on page 61 in 1.5 (deprecated) Calculating and Displaying Key Decision Factors on page 56.
- **Decision Details**

  See How to Use It on page 83 in 1.6 Calculating and Displaying Decision Details on page 70.

Choose Default Configuration Options for Your Product

When you initially set up your product, you must choose certain configuration options. You can use these default options and revisit them later in your development cycle to replace them with more appropriate options:

- **Open-ended cases**

  Allow your product to support open-ended cases.
- **Summarizer Strategy**

  Choose "Total weekly monetary entitlement" (see Choose or Create a Summarizer Strategy on page 54).

- **Determination Comparison Strategy**

  Choose "Compare all user-facing data" (see Determination Comparison Strategies on page 154).
- **Reassessment Strategy**

  Choose "Do not reassess closed cases" (see Product Delivery Rule Objects on page 102).

Implement a Single Product Period First

In the unlikely event that your initial version of product requires multiple product periods (i.e. already has changes of legislation and/or policy to deal with), first work with a single product period only, starting at the start of your overall product, and with no end date. Further product periods can be implemented when this initial product period is up-and-running. A default product period and associated eligibility and entitlement rule set is automatically created and published for a benefit product that is created via the dynamic product wizard. This eligibility and entitlement rule set is intended to be edited in line with your product requirements prior to product use.

When you have implementations for your single product period up-and-running, then you can split your product into multiple product periods, cloning your rule sets from the first period you implemented, as a starting point.

Focus on Eligibility/Entitlement Rules

The core rules to get your product up-and-running are those for eligibility and entitlement calculations. Do not initially implement any rules for key decision factors or decision details.

For a benefit product that has been created via the product wizard, an initial version of the eligibility/entitlement rules will automatically have been created for your product. If you are creating a new eligibility and entitlement rule set, the initial version of your eligibility/entitlement rules can hard-code the following (to be replaced by real implementations later):

- A fixed eligibility result, such as "always eligible", or "eligible for January 2010 to December 2011 only"; and
- Fixed entitlement, i.e. fixed values for all objectives, a fixed number of occurrences for any multiple objectives, and fixed entitlement for each objective (such as "always entitled").

Typically it is easier to work on single objectives before moving on to multiple objectives (see Identify the rules that govern the objectives for each case on page 39).

Spin-off a Task to Write Rule Classes for Custom Entities and/or Evidence Types

While one rule developer creates the initial eligibility/entitlement rules, another developer can (in parallel) write rule sets for "data" rule classes for your:

- custom entities (which are required for rules calculations); and/or
- custom evidence types that you have created for your product.

> **Important:** To maximize re-use of these "data" rule classes, it is recommended that you place them in their own rule set - i.e. not the same rule set as the eligibility/entitlement rule classes.

Unless you have special requirements for handling versions of active evidence, typically you will find the Engine's support for succession sets the easiest option when later working with evidence in your eligibility/entitlement rules (see Active Succession Set Rule Objects on page 115.

The initial implementation of the "data" rule classes should concentrate on creating the rule attributes with the correct data types.

Later, you can add:

- a `description` rule attribute to each data rule class (to aid debugging); and
- rule attributes (suitably annotated) to allow navigation to related parent or child evidence (see Conversion Processing on page 117).

> **Important:** To maximize re-use of these "data" rule classes, there should not be any calculated rule attributes on any rule class other than the `description` rule attribute.
>
> Product-specific calculations should be placed on product-specific "calculator" rule classes later.

Once the "data" rule classes have been written and their rule sets published, the developer can write and publish rule object propagator configurations so that rule objects are automatically created by the Engine. These configurations will need to be in place before the overall product can be tested online, but are not required while the eligibility/entitlement rules are hard-coded, and are not required in order to unit-test parts of implemented eligibility/entitlement rules.

Later you can revisit your data rule classes and check for commonality across your products. You can refactor your rule classes so that any which are common across products are normalized into a common rule set, and thus are propagated to only once (to improve performance and lower database storage requirements).

Top-down Implementations

Once your hard-coded eligibility and entitlement rule implementations are in place, you can use a top-down approach to replace these hard-coded implementations with real logic appropriate to your product. As you drill down, you may create new hard-coded lower-level implementations, which (depending on your factoring of rule sets) might be handed-off to multiple developers for further implementation.

For example, the "Lone Parent" Benefit outlined in Example on page 15 might initially be implemented as "always eligible".

The next step in implementing the eligibility calculation for "Lone Parent" Benefit would be to replace the "always eligible" calculation with (in pseudo-code, and assuming relevant timeline operations):

- `childInEligibleAgeRange` AND
- `childResidingWithParent` AND
- `parentIsLone` AND
- `familyPassesMeansTest`.

These new attributes would each initially be hard-coded to be "true forever". The top-down process can then undergo another iteration and the implementations of these new attributes each changed from the hard-coded "true forever" into real implementations, possibly with further new attributes which are initially hard-coded; e.g. the `childInEligibleAgeRange` could apply date logic to the child's date of birth, but that date of birth could initially be hard-coded instead of coming from a propagated rule class for an entity or evidence type.

Bottom-up Implementations

If you know that there are complex calculations around a propagated rule class, then you can implement and test "calculator" rule classes even though these are not yet used by eligibility/entitlement rules.

Later you can integrate these bottom-up implementations with your top-down implementations and re-test.

Hard-code Rates at First

In the initial implementation of your eligibility/entitlement rules, you may require data which is best implemented as rate tables.

Initially, though, you may find it easier to hard-code the rate information directly in your rule implementations; later you can refactor your rules to move the data to Cúram rate tables, and replace your implementation with the `rate` expression, and create propagator configuration entries to allow your new rate tables to be propagated Rate Rule Objects on page 105.

> **Important:** If your implemented product goes live with rates hard-coded in the rule sets, then any subsequent rate changes will involve rule set changes (with the associated re-testing effort), rather than a rate-only change external to your rule sets.

Keep an Eye on Rule Class Dependencies

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

It is recommended that you keep these rule classes in separate rule sets:

- **Data rule classes**

  Keep the data rule classes for your custom entities and evidence types in a separate rule set to maximize their re-use across different products.
- **Eligibility/entitlement rule classes**

  Keep the rule classes for your eligibility/entitlement calculations in a separate rule set so that display-only changes to rules do not require re-testing of core eligibility/entitlement implementations.
- **(deprecated) Key decision factor rule classes**

  Keep the rule classes for your key decision factors in a separate rule set so that they can be maintained independently of your eligibility/entitlement rule classes.
- **Decision details rule classes**

  Keep the rule classes for your decision details in a separate rule set so that they can be maintained independently of your eligibility/entitlement rule classes.
- **Common calculator rule classes**

As you identify calculator rule classes which are common between your eligibility/entitlement, key decision factor and/or decision detail rule classes, you can consider placing such rule classes in "common" rule sets.

While it is important to adhere to the recommended structure above by the time your product's implementation is complete, you do not have to strictly adhere to it in the early days of your product's development cycle.

You can refactor your rule sets later to match this structure, although such a refactoring task will become more complex the longer it is put off. In any case, to refactor freely you will need to have built up a good bank of unit tests for the behavior of your rule classes.

(deprecated) Try Key Decision Factors before Decision Details

Writing rules for key decision factors is easier than writing rules for decision details, because:

- The data structure for key decision factors is fixed, whereas the data structure for decision details is free-form and requires design work
- The Engine provides screens to display key decision factors, whereas you have to implement your own dynamic UIM screens for decision details.

When starting to implement visualizations for the explanation of a case's determination result, consider implementing key decision factors before embarking on the more onerous task of implementing decision details. Key decision factors may be sufficient to demonstrate that the eligibility/entitlement results for your product's cases are not only "correct", but are "correct for the right reasons".

Re-use the Basic Decision Details before Writing Your Own

The Engine includes support for basic eligibility/entitlement decision details (see Basic Eligibility/entitlement Decision Details on page 29).

Consider re-using this basic display category before implementing your own (more suitable) display categories. This re-use will help you understand how decision details are implemented and will enable your testers to see more details about your cases (albeit only very basic details).

You can remove the Basic display category later when you implement more appropriate display categories.

To re-use the basic decision details, follow these steps (as part of the general steps involved in implementation decision details - see Implementation on page 86):

- When you create your case rule class (see Write the Case rule class on page 86), make your rule class extend `AbstractBasicProductDecisionDetailsRuleSet.AbstractBasicCase`. You must implement the inherited abstract `abstractCase` rule attribute, to create an instance of the underlying rule class for your eligibility/entitlement (i.e. the one you created in Write the Case Eligibility/Entitlement Calculation Rule Classes on page 45), specifying the value for `productDeliveryCase`; and
- When you create you create your dynamic UIM and properties files, clone those for the following pages to your custom component:
  - `CREOLEDisplayRules_basicCaseDisplay`; and
  - `CREOLEDisplayRules_basicCaseDisplay_objectiveTagSubscreen`.

You must provide new unique names for your cloned dynamic UIM and `.properties` files, and update their contents to point to your new names.

Start Slowly with Decision Details

The implementation of decision details (rules and screens) can be a complex area.

Start off with simple screens that do not require subscreens or comparison data. Avoid conditional display in your dynamic UIM until you have all displayable data reliably flowing from rules to determination results to screens.

Later you can start to implement conditional display, and more complex screens that have subscreens or comparison data.

When implementing a dynamic UIM screen, if you require data that is not already available in rules, then initially you can create a new rule class/rule attribute with a hard-coded dummy implementation only; once you have the data flowing to the screen you can revisit the rule attribute to implement real logic (possibly giving the task to another developer to work on in parallel to screen development).

Throughout Your Product's Development

The sections above describe various short-cuts that can be taken during initial development.

However, there are some approaches which are important no matter which part of the development cycle you're in:

- **Understand Timelines**

  CER's concept of Timelines, and its expressions for manipulating them, are widespread in rule sets used by products. Ensure that you are comfortable with how timelines work and when they are used, before you implement any rule sets for your product. See also Write the Case Eligibility/Entitlement Calculation Rule Classes on page 45.
- **Comment as you go**

  Write comments for your rule classes and rule attributes as you implement them, while they are fresh in your mind.
- **Test as you go**

  Once you have simple processing up-and-running, write unit tests for blocks of rules as you implement them. Write integration tests for your product once there are enough end-to-end processing steps implemented for your product.
- **Draw on rule sets included with Cúram Solutions**

  Look at the rule sets included with Cúram Solutions for ideas on how to structure and implement different styles of rules.
- **Refactor**

  Don't be afraid to refactor rules as you discover commonality and better structures. A good bank of unit tests will allow you to refactor with confidence. Draw pictures of complex structures (because any system of a certain complexity cannot be entirely self-documenting).
- **Monitor application logs**

  The Engine tries hard to continue processing even when configuration is not quite correct. Keep an eye on non-fatal warnings and/or errors written to the application logs and console output. Use the Engine's environment variables to increase the logging verbosity to help track down problems (see 1.16 Environment Variables on page 238).

Handling Legislation Change

When a product is first implemented, typically there is an initial version of legislation which takes effect until further notice.

However, over time, for political, policy or budgetary reasons, the legislation underpinning the functional requirements for your product may change. One style of change prevalent amongst social security legislation is whereby the eligibility and entitlement rules for cases change over the lifetime of the case. Typically (but not always):

- the change in rules "takes effect" on a particular date; the case's eligibility and entitlement is calculated according the "old" rules before that date, and according to the "new" rules after that date;
- the date that the rules changes takes effect is the same for all cases for that product, i.e. there is no data on any case which affects the date from which the new rules apply;
- cases that came to a natural end before the change in legislation are unaffected;
- cases that start after the new rules have taken effect use the new rules only; and
- after the new rules have taken effect, any new claims which are registered to *retrospectively* start before that date will use a mixture of old and new rules, just as for claims which were registered *on time* before that date.

The Engine uses the term "legislation era" to refer to a period of constant legislation; in other words, a change in legislation (no matter how big or small) ushers in a new legislation era for a product.

The following example illustrates how a product set-up might need to change from supporting just the "old" rules, to supporting a mixture of "old" and "new" rules, to be applied to different legislative eras.

A client has been receiving an Income Assistance benefit but the case has subsequently ended. Another client has been receiving the same benefit and continues to do so. Both of these cases are using the eligibility and entitlement rules that were implemented to meet the requirements of the single (initial) version of the legislation.

New legislation is enacted resulting in the need for the social services agency to modify the eligibility and entitlement rules. Changes are made to the rule set and made effective from the date on which the legislation went into effect.

For the case that has already ended, eligibility and entitlement continues to be determined using the initial version of the rules because the case ended before the effective date of the legislation changes. For the case that remains open, eligibility and entitlement continues to be determined using the initial version of the rules up to the effective date of the legislation change, but uses the new version of the rules from the effective date onwards.

For any cases that are created after the date on which the new version of the rule set took effect, eligibility and entitlement will be determined using the new version of the rule set. For any cases that are retrospectively created and begin before the effective date of the legislation changes, eligibility and entitlement will be determined using the initial version of the rule set up to the effective date and the new version of the rule set from that date onwards.

The Engine supports two different mechanisms for implementing legislation change:

- **Branching logic in your CER rule sets**

  You can create a new version of your existing rule sets used by your product, and this new version of the rule set can contain branches which apply different logic to the periods on your cases, before and after the date that the legislation changes.
- **Multiple product periods for your product**

  You can change your product's set-up so that the lifetime of the product is carved up into distinct "product periods", and you can assign different eligibility/entitlement CER rule sets to each of the product periods.

Each of these approaches has its own benefits and limitations. The remainder of this chapter describes these two supported mechanisms in greater detail, followed by some important points to consider when choosing which approach to follow when implementing a particular change in legislations.

> **Important:** You should familiarize yourself with the details of these approaches prior to deciding how best to implement changes in legislation which affect your product's requirements.

Branching Logic in Your CER Rule Sets

If you choose to implement branching logic in your rule sets, then you will first need to identify which rule attribute implementations will need to have branching logic. (Indeed, if this identification shows that a very large number of attributes require branching logic, you may be better off switching to the Multiple Product Periods for Your Product on page 201 approach instead.)

For each rule attribute that requires branching logic, use CER's `legislationChange` expression to implement different logic for different "eras" of time. See `legislationChange` on page 221 for a full description of this expression.

Update your rule set tests to test for the behavioral changes introduced by your legislation change.

Once your changes to your existing rule sets are complete, publish the changes, which will result in existing cases being bulk-reassessed, typically leading to new determinations being stored for some existing cases.

Multiple Product Periods for Your Product

If you choose to use multiple product periods for your product, then typically it will be helpful to clone the rule sets for your existing product period (assuming that the structure of the rules is still suitable, and that some significant number of existing rules implementations do not require updating to implement the legislation change).

Clone the rule sets using the process described above in Cloning CER Rule Sets on page 193. You need only clone the rule sets down as far as those rules which are affected by the legislation change. It may be helpful to rename your rule sets in line with your eras of legislation. You may also identify useful refactorings to make common any rule classes which are unaffected by the legislation change.

Update your cloned rule sets to implement the change in legislation.

You must create tests for your rule set changes; because you have cloned rule sets, you may need to clone your bank of rule set tests, and update the cloned tests to use your cloned rule sets. This cloning of tests will help ensure that existing functionality unaffected by the legislation change continues to work as expected. Update your cloned rule set tests to test for the behavioral changes introduced by your legislation change.

Once your changes to the newly-created cloned rule sets change are complete, publish the changes.

Set an end date on the existing product period. Create a new product period (see Write the Product Periods on page 51) and configure the new product period to use your cloned rule sets.

Publish your changes to the product periods, which will result in existing cases being bulk-reassessed, typically leading to new determinations being stored for some existing cases.

Choosing the Right Approach

When you come to implement a change in legislation, there are many factors which you should bear in mind when choosing which approach is better for your change. Sometimes the competing factors will mean that there is no overall clear-cut answer, and you will need to make an informed decision based on the details of the change that you need to make.

> **Note:** The key decision factors feature is deprecated. For more information, see Deprecated features.

Over time, a product may undergo several legislation changes, and each legislation change must be taken on its merits when deciding which approach to take. In other words, a long-lived product may have had several historical legislation changes implemented using branching logic, and also other historical legislation changes implemented using multiple product periods. Indeed, it is possible (but rare) that a single legislation change may be implemented using a combination of the "branching logic" and "multiple product periods" approaches.

This table describes some important consequences to keep in mind when making your choice.

Table 62: Factors involved when choosing the right approach for legislation change

Factor	Consequences under the "branching logic" approach	Consequences under the "multiple product periods" approach
Testing	For existing unit test that test attributes which now contain branching logic, the tests need to be updated to test the output of rule attributes taking into account eras of old legislation and new legislation; these tests will need to deal with both versions of the legislation.	The full bank of unit tests for a cloned rule set must also be cloned and maintained in line with the cloned rule set; in particular, tests for cloned attributes which are updated in line with the legislation change must be updated to test the behavior of the changed attribute. Each test only needs to deal with one version of the legislation, however.
Refactoring	Typically (depending on the complexity of the legislation change) no refactoring will be required to make logic common, as existing common logic should already have been factored to normalize common logic.	The cloning of rule sets may present an opportunity to refactor rule sets so that common logic unaffected by the legislation change can be refactored to exist only once - but this refactoring will take implementation and retesting effort.

Factor	**Consequences under the "branching logic" approach**	**Consequences under the "multiple product periods" approach**
Maintenance	The number of rule classes and rule attributes will typically be unchanged; however, the implementation of some rule attributes will become more complex, as those attributes have to deal with legislation change. Future bug fixes to the rule set (which span legislation eras) will typically require implementation in one rule set only.	The number of rule classes and rule attributes will increase due to cloning; however, the implementation of rule attributes should remain at the existing level of complexity. Future bug fixes to the rule set (which span legislation eras) may require implementation in more than one rule set, depending on whether logic common across rule sets has been refactored.
Complexity	This approach is suited to legislation changes which affect a small number of rule attributes, where the human-readability of the overall rule set is not overly-compromised by the limited introduction of branching logic.	This approach is suited to legislation changes which affect a large number of rule attributes, where the human-readability of the overall rule set would be overly-compromised by the wide-ranging introduction of branching logic.
Guarantees regarding stability of decisions under older legislation	You must test that your implementation of branching logic has correct dates for eras, and does not accidentally affect past periods already determined for cases.	Existing periods of determinations for the old product period (now with an end date) are guaranteed not to be affected by the legislation change introduced by the new product period.
Existing display categories	You must test that the existing dynamic UIM screens for display categories should continue to work with your updated rule sets.	You must test that the existing dynamic UIM screens for display categories work with the different rule sets configured for your different product periods. Either update the existing dynamic UIM screens (if necessary), or if the display output for an existing rule set category is sufficiently different after the legislation change, you may need to clone the display category so that there are different tabs (and thus different dynamic UIM screens) for before/after the legislation change. If you clone a display category, then case workers will need to click on one of two display categories, depending on which one is implemented for the coverage period displayed.
New display categories	If a new display category is introduced for the legislation change, then your existing rules must create appropriate "no output" values for older eras of legislation.	Your cloned rule sets and new product period can introduce support for a new display category which is not supported by older product periods.

Factor	Consequences under the "branching logic" approach	Consequences under the "multiple product periods" approach
Whether the legislation change affects all cases on the same date	Each attribute which is updated to contain legislation change logic can have an implementation which uses different era dates from other attributes (and can even use dates which vary according to circumstances of the case).	The cloned version of the rule sets affects all cases on the same date - namely the date of the new product period. The date cannot vary across cases.
Whether the legislation change date is important enough to flag to case workers on each determination	If the date of legislation change is important in its own right, consider implementing a key decision factor that shows the date that legislation changed.	If the date of legislation change is important in its own right, consider implementing a key decision factor which shows the date that legislation changed. Note that if the format of the decision details data is different between your old product period and your new product period, then when a case worker views a determination which spans these periods, then the Engine will split the determination into different coverage periods anyway (due to the change in decision details across the product periods).

Typically, the choice of approach does not significantly affect performance and/or data storage. The multiple product period approach will lead to more rule objects being created in memory during the calculation of a determination result; but these rule objects are not stored on the database. They are included in rule object snapshots, however, but if common logic has been normalized, then when CER is requested to create a rule object with common logic, CER will re-use a similarly-created rule object that has the same initialization/specified values, and thus the common rule object will appear only once in the rule object snapshot, regardless of which approach is used.

Some types of legislation change require that newly-registered cases are treated differently from cases which already existed at the time that the legislation change came into effect, even if those newly-registered cases are recorded to retrospectively start before the legislation change (i.e. if the case was registered "late" and its start date is back-dated). Under these circumstances, regardless of which approach is chosen, the case's registration date will become an important piece of evidence in its own right, as it will govern how the case is treated. If you are considering the "multiple product period" approach, then you might implement and test the changes to branch logic based on the case registration date *before* cloning your rule sets, as the branching logic may be common to the determination results for both eras of legislation.

Some types of legislation change introduce "transitional" periods whereby claims are treated different. You may need to include branching logic for several dates, or create more than one new product period, to cater for how cases are treated before/during/after any transitional periods.

If a start date of a case is incorrectly recorded, then when it is corrected, there may be a change to the product periods which contribute to that case, and thus the case may now be affected by a legislation change whereas previously it was not, or vice versa. This kind of retrospective correction to case start dates behaves as you would expect and is automatically handled by the Engine.

Legislation changes can be future-only or contain an element of retrospective change, depending on whether the effective date of the legislation change is in the future or the past; indeed, sometimes a legislation change targeted at a future date might, due to lags introduced by legal/ policy departments, end up being in the past by the time it is implemented. The implementation of legislation changes will typically affect a large number of cases, which will undergo changes in their determination results. For future-dated changes, the periods of the determinations affected will be in the future, typically for periods not yet processed by financial processing, and thus there will be no corrective under- or over-payments issued on foot of the change. For retrospective legislation changes though, it is possible to affect determinations for periods already delivered, typically leading to corrective under- or over-payment processing on a number of cases, as is to be expected.

Changing Product Configuration Settings

There are a number of configuration settings at the product level which can be changed at any time, including for a product which has already had product delivery cases created for it:

- decision summary display strategy;
- determination comparison strategy;
- allow open-ended cases; and
- reassessment strategy.

The sections below describe these configuration settings in more detail.

Decision Summary Display Strategy

The strategy in place for the product governs how each coverage period is summarized when shown to the a case worker (see Viewing a Determination's Coverage Periods on page 29, (deprecated) Viewing Key Decision Factors Graphically on page 57 and (deprecated) Viewing Key Decision Factors in a List on page 57).

If you change this setting for your product, then it will affect how all determinations for the product are displayed, including historical determinations already created and stored as well as any new determinations created in the future.

Determination Comparison Strategy

The strategy in place for the product governs how "different" a new assessment determination must be to be worthy of storage (see Determination Comparison Strategies on page 154).

If you change this setting for your product, then it will only affect the processing of new determinations created in the future; all existing determinations which have already been created and stored are entirely unaffected.

Allow Open-Ended Cases

This setting governs whether the Engine allows a new case to be created without an expected or actual end date.

If you change this setting for your product, then it will only affect the processing of new cases created in the future; all existing cases which have already been created and stored are entirely unaffected. Thus it is possible for a product not to allow the creation of open-ended cases, yet have cases registered against which *are* open-ended (because those cases were registered at a time in the past when the product *did* support open-ended cases).

Reassessment Strategy

This setting governs the types of cases that the Engine can reassess.

The supported settings are:

- **Automatically reassess all cases**

  The Engine allows reassessment of all cases and maintains dependency records for each case's assessment for as long as the case is recorded on the system. Over time, depending on the number of cases on your system, the ever-growing number of dependency records can affect the performance of your system.
- **Do not reassess closed cases**

  The Engine allows reassessment of all cases except for those with a status of "Closed". Closing a case removes only the Dependency records relating to eligibility (not advice context, and so forth). The Engine maintains dependency records for each non-closed case. If a case's circumstances change when it is closed, the Engine does not reassess the case (but if the case is reopened, the Engine then reassesses the case).

  This setting is the default for new products you add to your system.

If you change this setting for your product, then all existing cases are checked to see if dependency records should be created or deleted.

Important: The checking of existing cases is performed by the `ApplyProductReassessmentStrategy` batch job.

When a user publishes changes to a product, which include a change to the product's reassessment strategy, then at the point where the user is asked to confirm the publication, the system warns the user that a batch job request is queued up, and that the batch job must be run before the new reassessment strategy takes effect for existing cases.

For each product delivery case for the product:

- if the case was not reassessable under the old strategy but becomes reassessable under the new strategy, then an assessment is performed on the case to build up the dependency records for the case's determination result;
- if the case was reassessable under the old strategy but is no longer reassessable under the new strategy, then the dependency records for the determination result are removed;
- otherwise, no action is performed on the case.

1.12 Compliancy for eligibility and entitlement

Follow the guidelines to develop with the eligibility and entitlement infrastructure in a compliant manner.

The Public API

The Eligibility and Entitlement Infrastructure has a public API which customers may invoke in application code. Nothing will be changed or removed in this public API without following the standards for handling customer impact.

Identifying the Public API

The JavaDoc for the code packages listed in Code Package Restrictions on page 207 is the sole means of identifying which public classes, interfaces and methods form the Eligibility and Entitlement Infrastructure public API.

Code Package Restrictions

The following code packages are restricted:

- `curam.core.facade.infrastructure.assessment.impl`;
- `curam.core.facade.infrastructure.creole.rulesetadmin.impl`;
- `curam.core.facade.infrastructure.paymentcorrection.impl`;
- `curam.core.facade.infrastructure.product.creole.impl`;
- `curam.core.facade.infrastructure.propagator.impl`;
- `curam.core.sl.infrastructure.assessment.event.impl`;
- `curam.core.sl.infrastructure.assessment.impl`;
- `curam.core.sl.infrastructure.creole.extension.impl`;
- `curam.core.sl.infrastructure.creole.impl`;
- `curam.core.sl.infrastructure.paymentcorrection.impl`;
- `curam.core.sl.infrastructure.product.creole.impl`;
- `curam.core.sl.infrastructure.propagator.impl`; and
- `curam.core.sl.infrastructure.rate.impl`.

These packages contain interfaces and classes internal to the Eligibility and Entitlement Infrastructure. Unless explicitly permitted in the JavaDoc, customers must not provide their own implementation of any Java interface nor subclass any implementation Java class contained in these packages. Customers must not place any custom classes or interfaces in these packages.

For convenience the available customization points are described in 1.13 The Eligibility and Entitlement Engine API and Customizability on page 210.

Code Table Restrictions

`CaseAssessmentDetReason`

The `CaseAssessmentDetReason` code table contains codes which indicate why a particular assessment occurred (for example 'Case Activation' or 'Change of Evidence'). It can be found in the *`curam.core.sl.infrastructure.assessment.codetable`* package.

In contrast to the other code tables in the same package, customers are free to add custom codes to the `CaseAssessmentDetReason` code table using the recommended approach.

`CaseSnapshotDetReason`

The `CaseSnapshotDetReason` code table contains codes which indicate why a particular snapshot was taken occurred (for example 'Case Approval' or 'Case Rejection'). It can be found in the *`curam.core.sl.infrastructure.assessment.codetable`* package.

In contrast to the other code tables in the same package, customers are free to add custom codes to the `CaseSnapshotDetReason` code table using the recommended approach.

Restricted Code Table Packages

The following code table code packages are restricted:

- `curam.core.sl.infrastructure.assessment.codetable`;
- `curam.core.sl.infrastructure.codetable`;
- `curam.core.sl.infrastructure.propagator.codetable`; and
- `curam.core.facade.infrastructure.creole.rulesetadmin.codetable`.

With the exception of `CaseAssessmentDetReason` and `CaseSnapshotDetReason`, all code tables contained in these packages are reserved for use by the Eligibility and Entitlement Infrastructure. Customers must not modify these code tables in any way. In particular, the creation of custom versions of these code tables to contain additional codes is *not* supported.

Restricted Code Tables

The following code tables in the *`curam.codetable`* package are also restricted:

- `AssessmentDateListType`;
- `CaseDecisionMethodCode`;
- `CaseDecinitReasonCode`;
- `ReassessmentAmount`;
- `ReassessmentProcMode`; and
- `ReassessmentResult`.

These code tables are reserved for use by the Eligibility and Entitlement Infrastructure. Customers must not modify these code tables in any way. In particular, the creation of custom versions of these code tables to introduce custom codes is *not* supported.

Database Restrictions

The Eligibility and Entitlement Infrastructure includes a number of database tables. In general, these tables are internal to the Eligibility and Entitlement Infrastructure and the data on them may only be read or written via the public API.

For more details on the read/write restrictions for these database tables, see the following subsections.

Note: There are database tables prefixed with CREOLE which are part of other application components (in particular the CER Infrastructure), and which are subject to their own compliancy statements. Only the compliancy statements for the *Eligibility and Entitlement Infrastructure* database tables are described below.

RuleObjectPropagatorControl

RuleObjectPropagatorControl is a single-row control table which is used to control the execution of the initial propagation of application data to CREOLE rule objects.

This table contains a single row which indicates whether initial propagation has been run. This control row ensures that initial rule object propagation is run by exactly one application server instance during Guice initialization.

The data on this table is internal to the Eligibility and Entitlement Infrastructure and may not be read or written by any other component.

The single row on this table is populated via a DMX file provided with the application. Customers must not alter this DMX file nor create any other DMX files which target the RuleObjectPropagatorControl table.

Restricted Database Tables

The following list outlines the remaining database tables that are included in the Eligibility and Entitlement Infrastructure:

- BreakdownInfo
- BulkCaseReassessment
- BulkReassessRecalcGroup
- CaseDecision
- CaseDecisionObjective
- CaseDecisionObjectiveTag
- CaseNomineeProdDelPattern
- ConfigurationChangeItem
- ConfigurationPublication
- CREOLECaseDecision
- CREOLECaseDetermination
- CREOLECaseDeterminationData
- CREOLECaseReassessment
- CREOLEProduct
- CREOLEProductDecisionDispCat
- CREOLEProductPeriod
- CREOLEProductPeriodDispCat
- CREOLEProductPublicationItem
- CREOLEProductSandbox
- CREOLEProductSnapshot
- CREOLERecalcRequestGroup
- CREOLERecalculationRequest
- CREOLERuleClassLink
- CREOLERuleSetCategory
- CREOLERuleSetCategoryLink
- CREOLERuleSetPublicationItem
- NomineeOverUnderPayment
- OverPaymentEvidence
- OverUnderPaymentBreakdown
- OverUnderPaymentHeader
- PaymentCorrectionEvidence
- PaymentCorrectionFCLink
- PropConfigPublicationItem
- ReassessmentAmountInfo
- ReassessmentBalanceInfo
- ReassessmentInfo
- RuleObjPropConfigSandbox
- RuleObjPropConfigSnapshot
- UnderPaymentEvidence

Do not use the extension of the model to customize the database tables. The data on the database tables must not be read or written other than through the public API.

Support of the initial population of the database tables by using DMX files

For most of the preceding database tables, the initial population of the database tables by using DMX files is not supported. However, the following list outlines where the initial population of the database tables by using DMX files is supported:

- CREOLEProduct
- CREOLEProductDecisionDispCat
- CREOLEProductPeriod
- CREOLEProductPeriodDispCat
- CREOLERuleClassLink
- CREOLERuleSetCategory
- CREOLERuleSetCategoryLink

CER Rule Sets Included with the Application

The source of CER rule sets included by the application must not be changed "in place".

If you wish to use CER rule sets included with the application as the basis for your own rule sets, then you must clone those rule sets (and, transitively, their dependent and precedent rule sets) to your custom component. See Cloning CER Rule Sets on page 193 for the steps to follow.

1.13 The Eligibility and Entitlement Engine API and Customizability

Eligibility and Entitlement Engine API

The Eligibility and Entitlement Engine API was developed as a means of clearly identifying the entry points into the eligibility and entitlement engine, which is used in determining a client's eligibility and entitlement. The classes comprising the Eligibility and Entitlement Engine API are as follows:

- `AssessmentEngine`
- `AssessmentEngineImpl`
- `AssessmentEngineEntity`
- `AssessmentEngineEntityImpl`

These classes are located inside the following package:

..\EJBServer\components\core\source\curam\core\sl\infrastructure\assessment\impl

> **Important:** You must not provide your own implementation of any of the Java interfaces above nor subclass any implementation Java class.

Nothing will be changed or removed in this public API without following the standards for handling customer impact.

Customizability

A number of customization points are provided throughout the Eligibility and Entitlement Engine, either through events being raised or through a series of hooks.

Eligibility and Entitlement Engine events

The Eligibility and Entitlement Engine raises several business events that allow customers to add logic at various points in the application.

For information about how to add event listeners, see the *Persistence Cookbook*.

The following table lists the event classes that are available in the Eligibility and Entitlement Engine. For more information, see the Javadoc for the associated class.

Table 63: Eligibility and Entitlement Engine events

Event Class	**Description**
`curam.core.sl.infrastructure.assessment.event.impl.AssessmentEngineEvent`	This class contains events which are raised inside the Eligibility and Entitlement Engine.
`curam.core.impl.ProductDeliveryLifeCycleEvents`	This class contains events that are raised at various product delivery lifecycle events, for example, submitted for approval, approved, activation, closure, reactivation, and rejection.

For the `curam.core.impl.ProductDeliveryLifeCycleEvents` class, customers can provide an implementation of this event class and make it available to the application through the use of Guice bindings, as follows:

```
final Multibinder<ProductDeliveryLifeCycleEvents> productDeliveryLifeCycleEvents =
  Multibinder.newSetBinder(binder(),
    ProductDeliveryLifeCycleEvents.class);

productDeliveryLifeCycleEvents.addBinding()
  .to(CustomProductDeliveryLifeCycleEvents.class);
```

For more information, see the Javadoc for the associated class.

Eligibility and Entitlement Engine Hooks

The Eligibility and Entitlement Engine contains a number of hooks which can be availed of by customers needing to provide custom input to the Eligibility and Entitlement Engine.

The following table lists the hook classes that are available in the Eligibility and Entitlement Engine. For more information, see the Javadoc for the associated class.

Table 64: Eligibility and Entitlement Engine Hooks

Hook Class	**Description**
`curam.core.sl.infrastructure.assessment.impl.AssessmentEngineHooks`	This class contains a number of Eligibility and Entitlement Engine customization/hook points.
`curam.core.sl.infrastructure.assessment.impl.ReassessEligibilityHook`	This class contains a number of product specific hook/customization points called during Eligibility Assessment.

For the `AssessmentEngineHooks`, there is a default implementation of some of these hooks inside `AssessmentEngineHooksImpl`. Customers can provide their own implementation and make these available in the application through the use of Guice bindings, binding `AssessmentEngineHooks.class` to the custom implementation. The detail of how to do this is covered in the *Persistence Cookbook*.

For the `ReassessEligibilityHook`, customers can provide an implementation of these hook points for a specific product, and make them available to the application through the use of Guice bindings, as follows:

```
final MapBinder<Long, ReassessEligibilityHook> reassessEligibilityHooks =
  MapBinder.newMapBinder(binder(), Long.class, ReassessEligibilityHook.class);

reassessEligibilityHooks.addBinding(new Long(<PRODUCT_ID>)).to(<HOOK_IMPL>.class);
```

Product Delivery Effective Date Hook

During the approval of a product delivery case, the effective date is set by default. Customers can implement the `curam.core.hook.impl.ProductDeliveryApprovalStrategyHook` interface to set a custom effective date on the product delivery.

Writing and binding a custom implementation of the `curam.core.hook.impl.ProductDeliveryApprovalStrategyHook` interface replaces the default implementation that sets the effective date on the product delivery during the approval process. The default implementation uses the approval date and the frequency pattern to set the effective date for the product delivery.

The financial component uses the product delivery effective date as the anchor date for payments. The anchor date is the initial date that the system uses to calculate future payment dates during financial processing. For example, if the payment frequency pattern is Every 4 Weeks on a Sunday, the system uses the anchor date to identify which day to start the 4-week count for the first and subsequent payments.

To replace the default implementation, write a new class that implements the `curam.core.hook.impl.ProductDeliveryApprovalStrategyHook` interface.

For example:

```
import curam.core.hook.impl.ProductDeliveryApprovalStrategyHook;
import curam.util.exception.AppException;
import curam.util.exception.InformationalException;
import curam.util.type.Date;

public class CustomProductDeliveryApprovalStrategyHook
  implements ProductDeliveryApprovalStrategyHook {

  @Override
  public void setEffectiveDate(final Long caseID, final Date approvalDate)
    throws AppException, InformationalException {

    // add custom logic to set the case header effective date

  }

}
```

Then, bind the new class in a Guice module.

For example:

```
bind(ProductDeliveryApprovalStrategyHook.class)
  .to(CustomProductDeliveryApprovalStrategyHook.class);
```

For more information about the `curam.core.hook.impl.ProductDeliveryApprovalStrategyHook` interface, see the Javadoc.

1.14 Extensions to Cúram Express Rules

Introduction

The Engine contributes some expressions and annotations to CER. For more information about CER, see the *Cúram Express Rules Reference Manual*.

Important: These expressions and annotations are for use in case eligibility and entitlement processing only, and are not supported for use by any other purpose.

Expressions

The Engine contributes these expressions to CER:

- `combineSuccessionSets` on page 213;
- `legislationChange` on page 221; and
- `rate` on page 225.

combineSuccessionSets

Overview

The Active Succession Set Rule Object Converter (see Active Succession Set Rule Objects on page 115) supports the creation of one rule object per succession set of evidence records.

In some circumstances, a real-world entity (such as a person) can have periods of time with certain characteristics (such as a period of absence from the household). These periods of time may each be recorded as their own succession sets, and in some circumstances it can be beneficial to combine each of these succession sets into a single history of those periods.

The `combineSuccessionSets` expression allows you to "splice together" succession sets which each represent a non-overlapping period, with default values used for the "gaps" between those periods.

Example

Let's say that details of a person's absence from their household are captured as evidence.

Each period of absence has a start date and (optionally) an end date. Each period of absence also has a reason for the person's absence, but notably the reason can change during the absence (i.e. the person can be initially absent for one reason, then from a given date continues to be absent but for a different reason).

For a given person, no two periods of absence can overlap (a person cannot be absent in more than one way at any given time). In particular, the person can have at most one open-ended period of absence.

Each separate period of absence is captured as its own succession set of absence; in other words, if the person is absent from the household, and then returns, and then later leaves the household again, the second period of absence is a different succession set of evidence from the first.

Some eligibility/entitlement rules for the person require to derive information from the person's history of absences. Legislation states that a person is eligible for benefit if present in the household, or is absent for the reason of "Education" only.

To simplify these eligibility/entitlement rules, it is desirable to compute a single history of absence details from the person's absence periods (of which there may be zero, one or many). The `combineSuccessionSets` expression is used to compute this single history of absence details for the person.

Now let's create a history of absences for a `Person` and work through it as an example.

The person John Smith is absent from his household on these occasions:

- John leaves the household on 1st January 2000 to pursue his college education. Unfortunately, on 28th January 2000, John falls seriously ill and is hospitalized. His absence from the household continues but its reason changes from Education to Medical Treatment. John recuperates and returns home on 8th March 2000.
- John's curtailed college education and subsequent illness leave him with personal problems, and after committing a spate of petty crimes he is jailed on 2nd June 2000. A new absence is recorded for John, with a reason of Incarceration throughout (this particular absence does not vary in reason). He is released from prison on 30th June 2000.
- Determined to make a fresh start, John enlists in the army and is then absent from the household from 10th August 2000 until further notice, with an absence reason of Military.

Three `Absence` rule objects are populated for John's three periods of absence by the Active Succession Set Rule Object Converter. Each `Absence` rule object is populated by the succession set of evidence records captured for the related period of absence, e.g. the first `Absence` rule object is populated with a start date of 1st January 2000 and an end date of 8th March 2000. Each `Absence` rule object also has a `reason` attribute with values that vary over time.

A single `CombinedAbsence` rule object is then created by the `combineSuccessionSets` expression. The single `CombinedAbsence` rule object for John has a `reason` attribute with these values that vary over time:

- prior to 1st January 2000, the absence reason is blank (the default value), as John is not absent at this time;
- from 1st January 2000, the absence reason is Education, John's initial reason for his first absence;
- from 28th January 2000, the absence reason changes to Medical Treatment, in line with the change in reason part-way through John's his first absence;
- from 9th March 2000 (the day after John's last day in hospital), the absence reason changes back to blank, because John is no longer absent;
- from 2nd June 2000, John begins his second period of absence, with a reason of Incarceration;
- from 1st July 2000, the absence reason changes back to blank, because John is no longer absent; and
- from 10th August 2000 until further notice, John begins his third period of absence, with a reason of Military.

The single `CombinedAbsence` rule object for John also has an `exists` attribute which simply combines the periods for which the contributing absences each exist.

Detailed Behavior

Each instance of `combineSuccessionSets` must:

- take a single argument which must be a list of rule objects, each of which must ultimately inherit from the `PropagatorRuleSet.ActiveSuccessionSet` rule class;
- nominate a rule class to use as the type of the new rule object created and returned when the expression is evaluated.

When an instance of `combineSuccessionSets` is evaluated, a rule object of the nominated return rule class is created. The attribute values on this return rule object will be set by matching their names to those on the input succession set rule objects being combined.

For periods of time not covered by the input succession set rule objects (i.e. for the "gaps" between periods, or the values used if the input list is empty), then default values will be used as shown in the following table:

Table 65: Mapping from Cúram Domain Types to CER Rule Attribute Types

Data type for succession set attribute used as an input timeline	Default value, used for "gaps" between input succession sets
Timeline<Number>	0
Timeline<Boolean>	false
Timeline<Code table entry>	null (not-specified) entry from the code table
(any other data type)	null

If the input list of succession set of rule objects is empty (e.g. a person who has never been absent from their household), then the default values shown above will be used throughout the entire timelines.

The `combineSuccessionSets` expression applies validations when the rule set is validated, and also when the expression is evaluated, as described in the following sections.

Validation checks made when the rule set is validated

Each instance of `combineSuccessionSets` will be checked to ensure that it meets the following constraints:

- the single argument to the expression must return a list of rule objects, and the rule class for these rule objects must be annotated with the `SuccessionSetPopulation` on page 234 annotation;;
- for the rule class nominated as the return type of the `combineSuccessionSets` expression:
  - the rule class must not be abstract and must not contain any initialized attributes; and
  - each rule attribute on the rule class that has a `specified` derivation must have an identically-named attribute on the rule class for the input list of rule objects.

If the above conditions are not met, then the rule set will not pass validation checks.

Validation checks made when `combineSuccessionSets` is evaluated

When an instance of `combineSuccessionSets` is evaluated at run time, each rule object for a contributing succession set being combined must:

- have a non-blank start date - i.e. the value of the attribute named in the `SuccessionSetPopulation` on page 234 annotation must not be null; and
- not overlap with any other contributing succession set rule objects - i.e. the lifetimes of each contributing succession set (calculated with reference to the start/end date attributes named in the `SuccessionSetPopulation` on page 234 annotation) must each cover distinct periods. In particular, at most one contributing succession set is allowed to be open-ended.

If the above conditions are not met, then `combineSuccessionSets` throws an appropriate exception.

CER Editor reference

The `CombineSuccessionSet` element provides a graphical representation of the `combineSuccessionSet` expression. It is accessed from the Technical Logic pallet.

The steps below describe how to implement the logic described in Example on page 213. For brevity, some steps (such as the best practice of creating description rule attributes) are omitted.

Create an `Absence` rule class for evidence

Create a rule class named `Absence` to represent a `Person` 's succession set of Absence evidence, and add the following attributes:

- `caseParticipantRoleID` (Number);
- `startDate` (Date);
- `endDate` (Date); and
- `absenceReason` (Code table entry, from the `AbsenceReason` code table, then made into a Timeline).

On this `Absence` rule class, select the "Succession Set" check box, set the "Start Date Attribute" to `startDate` and the "End Date Attribute" to `endDate`. Edit the "Extends" to select the "ActiveSuccessionSet" rule class from the PropagatorRuleSet..

See the *`Cúram Express Rules Reference Manual`* for more details on general properties for all rule elements.

Write the `CombinedAbsence` return class

The `CombineSuccessionSet` expression will combine the details from the Absence rule objects, and you need a new rule class to hold the combined details.

Create a `CombinedAbsence` rule class with the following attributes:

- `absenceReason` (Code table entry, from the `AbsenceReason` code table, then made into a Timeline); and
- `exists` (Boolean, then made into a Timeline).

Write the `Person` rule class

Each absence pertains to a person, and thus you must create a `Person` rule class to contain the list of absence details (which will be combined in a single rule object showing the combined absence details for the person).

Add a `caseParticipantRoleID` (Number) attribute.

Add a Rule Attribute for the `Person` 's Absence Succession Sets

On the `Person` rule class, add an absences attribute which retrieves the `Absence` succession set rule objects for the `Person`.

Add a Rule Attribute to combine the `Person` 's Absence Succession Sets

On the `Person` rule class, add an `combinedAbsences` attribute which combines the `Absence` succession set rule objects for the `Person` (using the `CombineSuccessionSets` expression).

Add a Rule Attribute to calculate eligibility based on absence reason

On the `Person` rule class, add an `isEligibleTimeline` attribute (Boolean, made into a Timeline) which tests the absence reason for the combined absence history for the person.

XML Reference

This section provides a reference to the underlying XML representation of `combineSuccessionSets`.

Each `combineSuccessionSets` instance contains:

- a `ruleclass` attribute naming the rule class to be used as the return type;
- optionally, a `ruleset` attribute naming the rule set containing the rule class to be used as the return type (required only if that rule class is in a different rule set); and
- a single child expression, which must return a list of succession set rule objects.

The following XML implements the logic described in Example on page 213.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_combineSuccessionSets"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <!-- Rule class for a person on a case.  -->
  <Class name="Person">

    <Attribute name="caseParticipantRoleID">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Retrieves the absence succession sets (if any) for this
         person on the case. -->
    <Attribute name="absences">
      <type>
        <javaclass name="List">
          <ruleclass name="Absence"/>
        </javaclass>
      </type>
      <derivation>
```

```
        <readall ruleclass="Absence">
          <match retrievedattribute="caseParticipantRoleID">
            <reference attribute="caseParticipantRoleID"/>
          </match>
        </readall>
      </derivation>
    </Attribute>

    <Attribute name="combinedAbsences">
      <type>
        <ruleclass name="CombinedAbsence"/>
      </type>
      <derivation>
        <combineSuccessionSets ruleclass="CombinedAbsence">
          <reference attribute="absences"/>
        </combineSuccessionSets>
      </derivation>
    </Attribute>

    <!-- Eligible if present in the household (i.e. no absence,
         which means the absence reason is blank), or absent
         for the reason of education. -->
    <Attribute name="isEligibleTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Boolean"/>
        </javaclass>
      </type>
      <derivation>
        <timelineoperation>
          <choose>
            <type>
              <javaclass name="Boolean"/>
            </type>
            <test>
              <reference attribute="absenceReason">
                <intervalvalue>
                  <reference attribute="combinedAbsences"/>
                </intervalvalue>
              </reference>
            </test>
            <when>
              <condition>
                <Code table="AbsenceReason">
                  <!-- Not absent - eligible -->
                  <null/>
                </Code>
              </condition>
              <value>
                <true/>
              </value>
            </when>
            <when>
              <condition>
                <Code table="AbsenceReason">
                  <!-- Absent for education - eligible-->
                  <String value="AR001"/>
                </Code>
              </condition>
              <value>
                <true/>
              </value>
            </when>

            <otherwise>
              <!-- Not eligible -->
              <value>
                <false/>
              </value>
            </otherwise>
          </choose>

        </timelineoperation>
      </derivation>
    </Attribute>

  </Class>


  <!-- A rule class for holding a combined history of absences
 (if
       any) for a person. -->
  <Class name="CombinedAbsence">

    <!-- Will be populated from the (varying) absence reason on
         each contributing absence succession set; will be blank
         for periods between absences. -->
    <Attribute name="absenceReason">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <codetableentry table="AbsenceReason"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>


    <!-- Will be populated from the existence period from each
         contributing absence succession set; will be false for
         periods between absences. -->
    <Attribute name="exists">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Boolean"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
  </Class>

  <!-- A rule class for holding a succession set of evidence for
 a
```

```
        single period of absence (perhaps with a changing reason
        for absence).

        The data on this rule class will be used by the example
        combineSuccessionSet expression to populate the
        data on the CombinedAbsence rule object returned.

        Note that this rule class inherits an "exists" rule
        attribute from the ActiveSuccessionSet rule class, which
        will also be used.-->
   <Class extends="ActiveSuccessionSet"
  extendsRuleSet="PropagatorRuleSet" name="Absence">

     <Annotations>
       <SuccessionSetPopulation endDateAttribute="endDate"
  startDateAttribute="startDate"/>
     </Annotations>

     <Attribute name="caseParticipantRoleID">
       <type>
         <javaclass name="Number"/>
       </type>
       <derivation>
         <specified/>
       </derivation>
     </Attribute>

     <Attribute name="startDate">
       <type>
         <javaclass name="curam.util.type.Date"/>
       </type>
       <derivation>
         <specified/>
       </derivation>
     </Attribute>

     <Attribute name="endDate">
       <type>
         <javaclass name="curam.util.type.Date"/>
       </type>
       <derivation>
         <specified/>
       </derivation>
     </Attribute>

     <Attribute name="absenceReason">
       <type>
         <javaclass name="curam.creole.type.Timeline">
           <codetableentry table="AbsenceReason"/>
         </javaclass>
       </type>
       <derivation>
         <specified/>
       </derivation>
     </Attribute>

   </Class>
 </RuleSet>
```

`legislationChange`

Overview

A choice of approaches for implementing changes in legislation is supported (see Handling Legislation Change on page 200).

One approach is to use the `legislationChange` expression to provide "branching" logic in your rule set, by applying different logic contributed by different "legislation eras" to come up with a timeline of values based on the changes in legislation.

Example

Let's say that a person's eligibility for a product is based on whether that person is "low-waged".

Initially, legislation is enacted which lays down that a person is considered to be "low-waged" if that person's total *pre-tax* income is less than $20,000 per annum.

However, after some successful lobbying, the legislation is revisited and it is agreed that from 2001 onwards, a person will instead be deemed to be "low-waged" based on whether that person's *post-tax* income is less than $15,000 per annum.

After a change in administration, the legislation comes under scrutiny again, and from 2002 revised legislation takes effect which broadens the low-income net to cover persons with pre-tax income less than $22,000 per annum *and/or* post-tax income less than $16,000.

The initial eligibility calculation (for a person) is based directly off the pre-tax income (for that person). When the agency implements the changes in legislation, then a rule set designer changes the initial implementation to instead use the `legislationChange` expression to combine contributions from the different legislation "eras". (The implementation of the initial era is just that for the initial implementation; there are new implementations for the subsequent eras.)

John Smith makes a claim for benefit. John's income levels are as follows:

- from 1st January 2000, pre-tax income of $19,500 and post-tax income of $15,500;
- from 1st June 2001, pre-tax income of $18,000 and post-tax income of $14,700;
- from 1st July 2002, pre-tax income of $26,000 and post-tax income of $22,300.

John's eligibility varies not only according to the variations in his income levels, but also according to the changes in legislation that are enacted:

Detailed Behavior

Each instance of `legislationChange` must:

- specify the data type of the intervals in the timeline returned; and
- specify one or more eras of legislation, including an "initial" era (valid from the start of time, i.e. a null date). Each era must return a timeline of the same value type as that returned by the overall `legislationChange` expression

When an instance of `legislationChange` is evaluated, each era is evaluated and the resultant timelines are "spliced together" according to their era dates. An overall return timeline is assembled from these era-contributions, and this return timeline obeys the usual semantics of Timelines in general (in particular, identical contiguous values in the timeline will be amalgamated into a single value).

As such, if a particular change in legislation does *not* affect a calculation, then the resultant timeline will *not* change value on the legislation change date.

CER Editor reference

The Legislation Change element provides a graphical representation of the legislation Change expression. It is accessed from the Business Logic pallet.

The steps below describe how to implement the logic described in Example on page 221. For brevity, some steps (such as the best practice of creating description rule attributes) are omitted.

> **Note:** For brevity and clarity, the cut-off rates are "hard-coded" into this example.
>
> A production-quality rule set would instead externalize these rates using Cúram rate tables. See `rate` on page 225 for further details.

Create a `Person` rule class

Create a rule class named `Person`, and add the following attributes:

- `caseParticipantRoleID` (Number);
- `preTaxIncomeTimeline` (Number, made into a Timeline);
- `postTaxIncomeTimeline` (Number, made into a Timeline); and
- `isEligibleTimeline` (Boolean, made into a Timeline).

> **Note:** No derivations will be given for the `preTaxIncomeTimeline` and `postTaxIncomeTimeline` attributes in this example. These attributes are assumed to be populated from an outside source (such as evidence).

Implement legislation change logic

Implement the derivation of `isEligibleTimeline` to take into account the three eras of legislation:

XML Reference

This section provides a reference to the underlying XML representation of `legislationChange`.

Each `legislationChange` instance contains:

- an `intervaltype` element naming the data type for the intervals in the resultant timeline returned; and
- one or more `era` elements, each containing:
  - a `from` element, which must contain an expression returning a Date (which governs from which date the legislation era takes effect); and
  - a `value` element, which must contain an expression returning a timeline of the same type as that returned by the overall `legislationChange` expression (which provides the values to be used in the resultant timeline, for the portion of that timeline contributed by this era).

The following XML implements the logic described in Example on page 221.

> **Note:** For brevity and clarity, the cut-off rates are "hard-coded" into this example.
>
> A production-quality rule set would instead externalize these rates using Cúram's rate tables. See `rate` on page 225 for further details.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_legislationChange"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <!-- Rule class for a person on a case.  -->
  <Class name="Person">

    <!-- The pre-tax income for this person. -->
    <Attribute name="preTaxIncomeTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Number"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- The pre-tax income for this person. -->
    <Attribute name="postTaxIncomeTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Number"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Eligible if "low-waged", where the definition of
         "low-waged" varies as changes in legislation are
         enacted. -->
    <Attribute name="isEligibleTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Boolean"/>
        </javaclass>
      </type>
      <derivation>
        <legislationChange>
          <!-- The type of the timeline returned is a timeline of
               Boolean values. -->
          <intervaltype>
            <javaclass name="Boolean"/>
          </intervaltype>

          <!-- Initial legislation era. -->
          <era>
            <from>
              <null/>
            </from>
            <value>
              <timelineoperation>
                <!-- Low-waged if pre-tax income is below
```

```
                     20,000. -->
                <compare comparison="&lt;">
                  <intervalvalue>
                    <reference attribute="preTaxIncomeTimeline"/>
                  </intervalvalue>
                  <Number value="20000"/>
                </compare>
              </timelineoperation>
            </value>
          </era>

          <!-- A change in legislation, effective from 1st
 January
               2001, -->
          <era>
            <from>
              <Date value="2001-01-01"/>
            </from>
            <value>
              <timelineoperation>
                <!-- Low-waged if post-tax income is below
                     15,000. -->
                <compare comparison="&lt;">
                  <intervalvalue>
                    <reference attribute="postTaxIncomeTimeline"/
>
                  </intervalvalue>
                  <Number value="15000"/>
                </compare>
              </timelineoperation>
            </value>
          </era>

          <!-- Another change in legislation, effective from 1st
               January 2002. -->
          <era>
            <from>
              <Date value="2002-01-01"/>
            </from>
            <value>
              <timelineoperation>
                <!-- Low-waged if pre-tax income is below 22,000
 OR
                     post-tax income is below 16,000. -->
                <any>
                  <fixedlist>
                    <listof>
                      <javaclass name="Boolean"/>
                    </listof>
                    <members>
                      <compare comparison="&lt;">
                        <intervalvalue>
                          <reference
 attribute="preTaxIncomeTimeline"/>
                        </intervalvalue>
                        <Number value="22000"/>
                      </compare>
                      <compare comparison="&lt;">
```

```
                          <intervalvalue>
                            <reference
 attribute="postTaxIncomeTimeline"/>
                          </intervalvalue>
                          <Number value="16000"/>
                        </compare>
                      </members>
                    </fixedlist>
                  </any>
                </timelineoperation>
              </value>
            </era>

          </legislationChange>



        </derivation>
      </Attribute>

    </Class>
  </RuleSet>
```

`rate`

Overview

The Rate Rule Object Propagator (see Rate Rule Objects on page 105) automatically creates rule objects which mirror the data from cells specified rate tables. The `rate` expression provides a convenience mechanism for searching for the rule object corresponding to a particular cell in a particular rate table.

Example

Let's say that a person's eligibility for a product is based on whether that person's total income is below a certain limit. This limit is revised periodically in line with costs of living and inflation.

The rules for eligibility do not vary as such; however the income limit does vary over time, and so a rules designer models the income limit as a rate table, rather than "hard-coding" the income limit directly in the rule set. This approach allows the income limit to vary independently of rules; changes to the income limit can be effected by publishing changes to the rate table, rather than by changing the rule set. In particular, the rule set itself does not need to be retested when the income limit changes (but of course any users changing rate tables should satisfy themselves that the change being made will have the desired effect, possibly by first trialling the change in a test environment).

The eligibility rules will use the `rate` expression to retrieve the value of the required rate, and use this value to determine whether a person's total income is within the bounds for eligibility.

John Smith makes a claim for benefit. John's total income varies as follows:

- from 1st January 2000, total income of $21,000;
- from 1st June 2001, total income of $23,500; and
- from 1st July 2002, total income of $26,200.

In parallel, the agency varies the income limit applied to eligibility calculations as follows:

- from 1st January 2000, income limit of $20,000;

- from 1st January 2001, total income of $22,000; and
- from 1st January 2002, total income of $24,000.

John's eligibility varies not only according to the variations in his total income level, but also according to the varying income limit rate:

Detailed Behavior

Each instance of `rate` must specify the "co-ordinates" of required cell in from a rate table. The co-ordinates which uniquely identify a cell are:

- the code for the name of the rate table (i.e. the code for the entry from the `RateTableType` code table);
- the code for the name of the row in the rate table (i.e. the code for the entry from the `RateRowType` code table); and
- the code for the name of the column in the rate table (i.e. the code for the entry from the `RateColumnType` code table).

Recall that each rate table can have multiple "version" with values effective from a specified date; whereas the rule object created by the Rate Rule Object Propagator has a timeline of numbers representing a varying rate value. Thus the value returned by the `rate` is the value of the rate cell as it *varies* over different versions of its rate table.

> **Note:** If two neighboring versions of rate table contain the *same* value for a particular rate cell, then the timeline value for that rate cell will not change value on the rate table change data. As per Timeline semantics, contiguous identical values are amalgamated into single unchanging value.
>
> This situation can arise where a new version of a rate table is recorded to change values in *some* of its rate cells only; values in other rate cells may remain unchanged.

When an instance of `rate` is evaluated, a rate rule object matching the required rate cells co-ordinates is sought. Ordinarily the required rate rule object will be found, and its varying rate cell value returned as a timeline of numbers.

However, if the required rate rule object has not been found (e.g. if rate propagation has not been run, for example in a unit test that creates a rate but does not publish changes), then the `rate` expression will create an *internal* rule object to hold the varying value from the rate table (subject to the same constraints that apply to the Rate Rule Object Converter - i.e. cells in sub-rows and sub-columns are not supported). If such internal rule object are created, then if a rate table change is subsequently published, then the `rate` expression will safely recalculate to pick up the *external* rule object created by the propagator. This feature means that it is possible to write unit tests that interact with rates without having to worry about rate propagation.

Important: All uses of the `rate` expression require a database transaction to be in effect.

For normal application processing, a database transaction will be in effect just as for other server logic.

However, for speed of testing, CER promotes the use of in-memory testing of rule sets which do not access the database. If you write any unit tests which cause a `rate` expression to be evaluated, then that test *must* run in the context of database transaction (which is in sharp contrast to the majority of unit tests for CER rule sets, which do *not* require a database transaction).

You must either:

- wrap your test in a database transaction (such as that provided by inheriting from `CuramServerTest`); or
- prevent the evaluation of `rate` expressions by using CER's "specify" mechanism to override the values of any attribute values whose definition includes one or more `rate` expressions.

CER Editor reference

The Rate Table element provides a graphical representation of the rate expression. It is accessed from the Data Types pallet.

The steps below describe how to implement the logic described in Example on page 225. For brevity, some steps (such as the best practice of creating description rule attributes) are omitted.

Create a `Person` rule class

Create a rule class named `Person`, and add the following attributes:

- `totalIncomeTimeline` (Number, made into a Timeline); and
- `isEligibleTimeline` (Boolean, made into a Timeline).

Note: No derivations will be given for the `totalIncomeTimeline` in this example. This attribute is assumed to be populated from an outside source (such as evidence).

Implement rate retrieval logic

Implement the derivation of `isEligibleTimeline` to retrieve the required rate data and compare it to the person's total income.

XML Reference

This section provides a reference to the underlying XML representation of `rate`.

Each `rate` instance contains:

- a `table` attribute naming the code from the RateTableType code table;
- a `row` attribute naming the code from the RateRowType code table;
- a `column` attribute naming the code from the RateColumnType code table;

The following XML implements the logic described in Example on page 225.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_rate"
```

```
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <!-- Rule class for a person on a case.  -->
  <Class name="Person">

    <!-- The total income for this person. -->
    <Attribute name="totalIncomeTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Number"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Eligible if total income is below an income limit (from
 a
         rate table). -->
    <Attribute name="isEligibleTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Boolean"/>
        </javaclass>
      </type>
      <derivation>
        <compare comparison="&lt;">
          <intervalvalue>
            <reference attribute="totalIncomeTimeline"/>
          </intervalvalue>
          <intervalvalue>
            <!-- code table constants for the rate table/rate
 row/rate column -->
            <rate table="RTT_LIMITS" row="RR_INCOME"
 column="RC_AMOUNT"/>
          </intervalvalue>
        </compare>
      </derivation>
    </Attribute>

  </Class>
</RuleSet>
```

Annotations

The Engine contributes these annotations to CER:

- `Display` on page 229;
- `DisplaySubscreen` on page 230;
- `Legislation` on page 233;
- `SuccessionSetPopulation` on page 234;
- `relatedEvidence` on page 235; and
- `relatedSuccessionSet` on page 236.

Display

This marks an attribute for inclusion when the Engine walks rule objects to gather decision details to include in a determination (see 1.6 Calculating and Displaying Decision Details on page 70).

This annotation may be placed on a rule attribute only.

XML Reference

Here is an example rule set with Display and non-Display rule attributes:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Display"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">


  <Class name="HouseholdMember">

    <!-- This attribute will be made available for display in
 decision details -->
    <Attribute name="dateOfBirth">
      <Annotations>
        <Display/>
      </Annotations>
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- This attribute will be made available for display in
         decision details -->
    <Attribute name="fullName">
      <Annotations>
        <Display/>
      </Annotations>
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <XmlMessage>
          <replace>
            <reference attribute="firstName"/>
          </replace>
          <replace>
            <String value=" "/>
          </replace>
          <replace>
            <reference attribute="surname"/>
          </replace>
        </XmlMessage>
      </derivation>
    </Attribute>
    <!-- This attribute is used as an input into the calculated
         fullName, but is not directly required for display -->
    <Attribute name="firstName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- This attribute is used as an input into the calculated
         fullName, but is not directly required for display -->
    <Attribute name="surname">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>
</RuleSet>
```

`DisplaySubscreen`

This marks an attribute as returning data for a subscreen of decision details, which can be displayed by the Engine when a user expands a row on a decision details screen (see Sub-screens on page 72).

This annotation may be placed on a rule attribute only, on a rule class which ultimately extends from `ProductDecisionDetailsRuleSet.AbstractCase`.

XML Reference

Here is an example rule set with `DisplaySubscreen` rule attributes:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_DisplaySubscreen"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">


  <Class name="MyProductSummary" extends="DefaultCase"
    extendsRuleSet="DefaultProductDecisionDetailsRuleSet">

    <!-- Allow the screen to display a list of members for the
         case. -->
    <Attribute name="householdMembers">
      <Annotations>
        <Display/>
      </Annotations>
      <type>
        <javaclass name="List">
          <ruleclass name="HouseholdMember"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Allow the screen to drill down into more details for
each
        household member -->
    <Attribute name="householdMemberSubscreens">
      <Annotations>
        <DisplaySubscreen/>
      </Annotations>
      <type>
        <javaclass name="List">
          <ruleclass name="HouseholdMemberSubscreen"/>
        </javaclass>
      </type>
      <derivation>
        <dynamiclist>
          <list>
            <reference attribute="householdMembers"/>
          </list>
          <!-- Create a wrapper HouseholdMemberSubscreen for
each
              HouseholdMember -->
          <listitemexpression>
            <create ruleclass="HouseholdMemberSubscreen">
              <specify attribute="householdMember">
                <current/>
              </specify>
            </create>
          </listitemexpression>
        </dynamiclist>

      </derivation>
    </Attribute>


  </Class>

  <Class name="HouseholdMember">
    <Attribute name="concernRoleID">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="earnedIncome">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
```

```
   <Attribute name="unearnedIncome">
     <type>
       <javaclass name="Number"/>
     </type>
     <derivation>
       <specified/>
     </derivation>
   </Attribute>


 </Class>

 <Class name="HouseholdMemberSubscreen"
extends="AbstractCaseSubscreenDisplay"
   extendsRuleSet="DefaultProductDecisionDetailsRuleSet">

   <!-- The wrapped household member -->
   <Attribute name="householdMember">
     <type>
       <ruleclass name="HouseholdMember"/>
     </type>
     <derivation>
       <specified/>
     </derivation>
   </Attribute>

   <!-- identifier of the business object -->
   <Attribute name="businessObjectID">
     <type>
       <javaclass name="Number"/>
     </type>
     <derivation>
       <reference attribute="concernRoleID">
         <reference attribute="householdMember"/>
       </reference>
     </derivation>
   </Attribute>

   <!-- Data to display on the subscreen -->

   <Attribute name="totalIncome">
     <Annotations>
       <Display/>
     </Annotations>
     <type>
       <javaclass name="Number"/>
     </type>
     <derivation>
       <arithmetic operation="+">
         <reference attribute="earnedIncome">
           <reference attribute="householdMember"/>
         </reference>
         <reference attribute="unearnedIncome">
           <reference attribute="householdMember"/>
         </reference>
       </arithmetic>
     </derivation>
```

```
    </Attribute>

  </Class>
</RuleSet>
```

`Legislation`

This allows a rule element to be linked to an arbitrary HTML document which describe the legislation underpinning that rule element.

This annotation may be placed on:

- a rule set;
- a rule class;
- a rule attribute; or
- an expression.

The CER Editor can open legislation links in your web browser as follows:

- a legislation link value that starts with *`http://`* or *`https://`* will open the absolute page for that link; or
- any other legislation link value will be used as a path relative to your application.

XML Reference

Here is an example rule set with various legislation link values:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Legislation"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Annotations>
    <!-- Rule set - an absolute link. -->
    <Legislation
 link="http://www.somelegislationsite.com/somepage1"/>
  </Annotations>
  <Class name="Citizen">
    <Annotations>
      <!-- Rule class - another absolute link. -->
      <Legislation
 link="http://www.somelegislationsite.com/somepage2"/>
    </Annotations>

    <Attribute name="dateOfBirth">
      <Annotations>
        <!-- Rule attribute - a relative link. -->
        <Legislation link="somedirectory/onmywebserver/
page.html"/>
      </Annotations>
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified>
          <Annotations>
            <!-- Expression - another relative link. -->
            <Legislation
 link="anotherdirectory/onmywebserver/page.html"/>
          </Annotations>
        </specified>
      </derivation>
    </Attribute>
  </Class>
</RuleSet>
```

`SuccessionSetPopulation`

This annotation indicates to the Active Succession Set Rule Object Converter (see Active Succession Set Rule Objects on page 115) which attributes on a rule class hold the start and end dates which mark the "lifetime" of the succession set of evidence.

This annotation may be placed on a rule class only. The annotated rule class must ultimately extend the `PropagatorRuleSet.ActiveSuccessionSet` rule class included with the application.

The names of the start date attribute and the end date attribute are each optional in this annotation. However, if present, each named attribute must exist on the rule class (i.e. must be declared by or inherited by the rule class), and must return a Date value.

If the start date attribute is not named by the annotation, or is named but at evaluation time is found to have a null value, then the data in the initial version of the succession set is assumed to apply from the beginning of time.

Similarly, if the end date attribute is not named by the annotation, or is named but at evaluation time is found to have a null value, then the data in the final version of the succession set is assumed to apply until the beginning of time.

XML Reference

Here is the XML for an example rule set, with a rule class extending `PropagatorRuleSet.ActiveSuccessionSet`, with its succession set start date and end date attributes set:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_SuccessionSetPopulation">
  <Class name="Employment" extends="ActiveSuccessionSet"
 extendsRuleSet="PropagatorRuleSet">
    <Annotations>
      <SuccessionSetPopulation startDateAttribute="startDate"
        endDateAttribute="terminationDate"/>
    </Annotations>
    <Attribute>
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
    <Attribute>
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
  </Class>
</RuleSet>
```

`relatedEvidence`

When Active Evidence Row Rule Objects are populated (see Active Evidence Row Rule Objects on page 130), indicates that the value of the annotated attribute should be automatically populated with rule object(s) for related evidence. See Conversion Processing on page 131 for full details on the processing the rule object converter performs, taking into account this annotation.

This annotation may be placed on a rule attribute only. Furthermore, the following restrictions apply:

- the annotated attribute must be on a rule class that ultimately extends the `PropagatorRuleSet.ActiveEvidenceRow` rule class; and
- the return type of the annotated attribute must be either:
  - a rule class that ultimately extends the `PropagatorRuleSet.ActiveEvidenceRow` rule class; or
  - a List of such a rule class.

XML Reference

Here is an example rule set with rule attributes for related parent and child evidence rows annotated to be automatically populated using the Active Evidence Row Rule Object Propagator:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_relatedEvidence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">


  <Class name="HouseholdMember" extends="ActiveEvidenceRow"
    extendsRuleSet="PropagatorRuleSet">
    <Attribute name="incomes">
      <Annotations>
        <!-- The Active Evidence Row Rule Object Propagator will
             automatically populate this attribute with a list of
             related Income rule objects.-->
        <relatedEvidence relationship="child"/>
      </Annotations>
      <type>
        <javaclass name="List">
          <ruleclass name="Income"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Income" extends="ActiveEvidenceRow"
    extendsRuleSet="PropagatorRuleSet">
    <Attribute name="householdMembers">
      <Annotations>
        <!-- The Active Evidence Row Rule Object Converter will
             automatically populate this attribute with a list of
             related HouseholdMember rule objects.

             Note that a list is still used because there may be
             multiple versions of the parent household member
             evidence.-->
        <relatedEvidence relationship="parent"/>
      </Annotations>
      <type>
        <javaclass name="List">
          <ruleclass name="HouseholdMember"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>
</RuleSet>
```

`relatedSuccessionSet`

When Active Succession Set Rule Objects are populated (see Active Succession Set Rule Objects on page 115), indicates that the value of the annotated attribute should be automatically populated with rule object(s) for related evidence. See Conversion Processing on page 117 for full details on the processing the rule object converter performs, taking into account this annotation.

This annotation may be placed on a rule attribute only. Furthermore, the following restrictions apply:

- the annotated attribute must be on a rule class that ultimately extends the `PropagatorRuleSet.ActiveSuccessionSet` rule class; and
- the return type of the annotated attribute must be either:
  - a rule class that ultimately extends the `PropagatorRuleSet.ActiveSuccessionSet` rule class; or
  - a List of such a rule class.

XML Reference

Here is an example rule set with rule attributes for related parent and child succession sets annotated to be automatically populated using the Active Succession Set Rule Object Propagator:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_relatedSuccessionSet"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">


  <Class name="HouseholdMember" extends="ActiveSuccessionSet"
    extendsRuleSet="PropagatorRuleSet">
    <Attribute name="incomes">
      <Annotations>
        <!-- The Active Succession Set Rule Object Converter will
             automatically populate this attribute with a list of
             related Income rule objects.-->
        <relatedSuccessionSet relationship="child"/>
      </Annotations>
      <type>
        <javaclass name="List">
          <ruleclass name="Income"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Income" extends="ActiveSuccessionSet"
    extendsRuleSet="PropagatorRuleSet">

    <Attribute name="householdMembers">
      <Annotations>
        <!-- The Active Succession Set Rule Object Converter will
             automatically populate this attribute with the
             related HouseholdMember rule object. -->
        <relatedSuccessionSet relationship="parent"/>
      </Annotations>
      <type>
        <ruleclass name="HouseholdMember"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>
</RuleSet>
```

1.15 Caching

During a reassessment transaction, the results of certain SQL searches from the EvidenceDescriptor table are cached in the current transaction.

Whenever a case ID can be inferred from a search of the table, all the EvidenceDescriptor records for that case are loaded into the cache. Later searches that involve the case are fetched from the cache instead of queried from the database.

The cache mechanism is active for reassessment transactions only. For example, the cache mechanism does not operate for online transactions that modify or edit a piece of evidence.

By default, the cache mechanism for reassessment transactions is enabled.
If required, you can disable it by adding the following application property:`curam.evidence.evdescripcache.disabled=TRUE`.

The following list outlines the three transaction level caches that store the data:

- `curam.cache.transaction-group.evidencedescriptorcache.bycase`
- `curam.cache.transaction-group.evidencedescriptorcache.byrelatedidandtype`
- `curam.cache.transaction-group.evidencedescriptorcache.successionIDToCaseID`

As with all transaction-level caches, you can tune each of the transaction level caches by using the timeToIdle parameter, the timeToLive parameter, or both. You can view the performance of this cache in the JMX statistics.

1.16 Environment Variables

Environment Variables Governing Behavior of Engine

This appendix lists the Environment Variables that you can set to change the Engine's behavior.

Table 66: Environment variables governing the behavior of the Engine

Environment variable name	Description	More Information
curam.trace	General trace setting for the overall application.	Logging on page 143 and the *`Cúram Server Developer's Guide`*.
curam.trace.ruleobjectpropagation	The logging level for rule object propagation log messages. Valid values are: • trace_off; • trace_on; • trace_verbose; and • trace_ultra_verbose.	Logging on page 143.
curam.ruleobjectpropagation.configuration.errorlevel	Whether a problem in the configuration for a rule object propagator is reported as: • an application error (error); • a logged warning (warn); or • ignored (ignore).	Data Configuration Problems on page 139.
curam.ruleobjectpropagation.nonpropagatableoperation.errorlevel	Whether a database operation which cannot be propagated to rule objects is reported as: • an application error (error); • a logged warning (warn); or • ignored (ignore).	Propagation Processing on page 111.
curam.creole.log.case.determination.problems	Whether details of problems encountered during a case determination should be listed as warnings in the application log.	Testing Calculating and Displaying Eligibility and Entitlement, Testing Calculating and Displaying Key Decision Factors and Testing Calculating and Displaying Decision Details.

Environment variable name	Description	More Information
curam.batch.creolebulkcasechunkreassessment.chunksize	The number of cases in each chunk that will be processed by the CREOLE Bulk Case Chunk Reassessment batch program.	The `CREOLEBulkCaseChunkReassessmentByProduct` Batch Process on page 184.
curam.batch.creolebulkcasechunkreassessment.dontrunstream	Should CREOLE Bulk Case Chunk Reassessment batch program sleep while waiting for the processing to be completed (rather than run a stream in its context).	The `CREOLEBulkCaseChunkReassessmentByProduct` Batch Process on page 184.
curam.batch.creolebulkcasechunkreassessment.chunkkeywaitinterval	The interval (in milliseconds) for which the CREOLE Bulk Case Chunk Reassessment batch program will wait before retrying when reading the chunk key table.	The `CREOLEBulkCaseChunkReassessmentByProduct` Batch Process on page 184.
curam.batch.creolebulkcasechunkreassessment.unprocessedchunkwaitinterval	The interval (in milliseconds) for which the CREOLE Bulk Case Chunk Reassessment batch program will wait before retrying when reading the chunk table.	The `CREOLEBulkCaseChunkReassessmentByProduct` Batch Process on page 184.
curam.batch.creolebulkcasechunkreassessment.processunprocessedchunks	Should CREOLE Bulk Case Chunk Reassessment batch program process any unprocessed chunks found after all the streams have completed.	The `CREOLEBulkCaseChunkReassessmentByProduct` Batch Process on page 184.
curam.workflow.gendetermineeligibilityfailureticket	"YES"/"NO" flag which determines whether a workflow ticket is automatically generated whenever a determine product eligibility for a case fails.	
curam.workflow.geneligibilityreassesssuccessticket	"YES"/"NO" flag which determines whether a workflow ticket is automatically generated whenever a case has been reassessed and the case decision is now "eligible".	
curam.trace.productconfiguration.publication	The logging level for product configuration publication log messages. Valid values are: • trace_off; • trace_on; • trace_verbose; and • trace_ultra_verbose.	
curam.trace.caseassessment	The logging level for case assessment log messages. Valid values are: • trace_off; • trace_on; • trace_verbose; and • trace_ultra_verbose.	

Environment variable name	Description	More Information
curam.creole.manualeligibilitycheckdetermination.store.ruleobjectsnapshot	Whether the system will store a snapshot of rule objects used in the calculation of a manual eligibility check determination.	The Database Tables on page 148
curam.ruleobjectpropagation.ignore.caseheader.deferredprocessing	Whether the data propagators should load when the CaseHeader table is being written to where the status value is 'Delayed Processing Pending'. Defaults to 'YES' to reduce processing, as this status is transient and not deemed relevant to rules processing.	
curam.load.datapropagator.userslogin	Whether the data propagators should load when the Users table is being written to during initial login after application start up. The default value is 'NO' to reduce processing, as user login information is not deemed relevant to rules processing.	
curam.case.reassessment.aggregation	Whether reassessments for a particular case are aggregated together into one reassessment workflow process. The default value is 'YES'.	Reassessment Aggregation on page 171
curam.case.reassessment.aggregation.wait.period	The time (in seconds) after which the user will be allowed to manually reassess or activate a case, even if another reassessment is queued or in progress.	See the *`Cúram Integrated Case Management Guide`*.
curam.evidence.evdescripcache.disabled	The variable disables the EvidenceDescriptor transaction level cache.	For more information, see Caching.
curam.evidence.disable.suppress.evidenceapprovalrequeststatus	The variable disables the suppression of the method `EDApprovalRequest.readCurrentApprovalRequestDetails()`. By default, the method is suppressed during a reassessment transaction because it reads data that is not needed by a reassessment. By setting this property to **True**, the method runs even during a reassessment transaction.	

Environment variable name	Description	More Information
curam.evidence.disable.suppress.userforlatestchange	The variable disables the suppression of the method `EvidenceChangeHistory.readUserForLatestChange()`. By default, the method is suppressed during a reassessment transaction because it reads data that is not needed by a reassessment. By setting the property to **True**, the method runs even during a reassessment transaction.	

Notices

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the Merative website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of Merative

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of Merative.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

Merative reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by Merative, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

MERATIVE MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Merative or its licensors may have patents or pending patent applications covering subject matter described in this document. The furnishing of this documentation does not grant you any license to these patents.

Information concerning non-Merative products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Merative has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Merative products. Questions on the capabilities of non-Merative products should be addressed to the suppliers of those products.

Any references in this information to non-Merative websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this Merative product and use of those websites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

The licensed program described in this document and all licensed material available for it are provided by Merative under terms of the Merative Client Agreement.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to Merative, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. Merative, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. Merative shall not be liable for any damages arising out of your use of the sample programs.

Privacy policy

The Merative privacy policy is available at https://www.merative.com/privacy.

Trademarks

Merative ™ and the Merative ™ logo are trademarks of Merative US L.P. in the United States and other countries.

IBM®, the IBM® logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Adobe™, the Adobe™ logo, PostScript™, and the PostScript™ logo are either registered trademarks or trademarks of Adobe™ Systems Incorporated in the United States, and/or other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft™, Windows™, and the Windows™ logo are trademarks of Microsoft™ Corporation in the United States, other countries, or both.

UNIX™ is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.