Guidewire PolicyCenter®

PolicyCenter Product Model Guide

PolicyCenter Release 8.0.3

Guidewire®
Deliver insurance your way™

Product Release Information
Product Name: Guidewire PolicyCenter
Product Release: 8.0.3
Document Name: *PolicyCenter Product Model Guide*
Document Revision: 18-November-2014

Contents

About PolicyCenter Documentation

The following table lists the documents in PolicyCenter documentation.

Document	Purpose
InsuranceSuite Guide	If you are new to Guidewire InsuranceSuite applications, read the *InsuranceSuite Guide* for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications.
Application Guide	If you are new to PolicyCenter or want to understand a feature, read the *Application Guide*. This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with PolicyCenter.
Upgrade Guide	Describes how to upgrade PolicyCenter from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing PolicyCenter application extensions and integrations.
New and Changed Guide	Describes new features and changes from prior PolicyCenter versions. Intended readers are business users and system administrators who want an overview of new features and changes to features. Consult the “Release Notes Archive” part of this document for changes in prior maintenance releases.
Installation Guide	Describes how to install PolicyCenter. The intended readers are everyone who installs the application for development or for production.
System Administration Guide	Describes how to manage a PolicyCenter system. The intended readers are system administrators responsible for managing security, backups, logging, importing user data, or application monitoring.
Configuration Guide	The primary reference for configuring initial implementation, data model extensions, and user interface (PCF) files. The intended readers are all IT staff and configuration engineers.
Globalization Guide	Describes how to configure PolicyCenter for a global environment. Covers globalization topics such as global regions, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who localize PolicyCenter.
Rules Guide	Describes business rule methodology and the rule sets in PolicyCenter Studio. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu.
Contact Management Guide	Describes how to configure Guidewire InsuranceSuite applications to integrate with ContactManager and how to manage client and vendor contacts in a single system of record. The intended readers are PolicyCenter implementation engineers and ContactManager administrators.
Best Practices Guide	A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers.
Integration Guide	Describes the integration architecture, concepts, and procedures for integrating PolicyCenter with external systems and extending application behavior with custom programming code. The intended readers are system architects and the integration programmers who write web services code or plugin code in Gosu or Java.
Gosu Reference Guide	Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration.
Glossary	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.

Document	Purpose
Product Model Guide	Describes the PolicyCenter product model. The intended readers are business analysts and implementation engineers who use PolicyCenter or Product Designer. To customize the product model, see the *Product Designer Guide*.
Product Designer Guide	Describes how to use Product Designer to configure lines of business. The intended readers are business analysts and implementation engineers who customize the product model and design new lines of business.

Conventions in This Document

Text style	Meaning	Examples
italic	Emphasis, special terminology, or a book title.	A *destination* sends messages to an external system.
bold	Strong emphasis within standard text or table text.	You **must** define this property.
narrow bold	The name of a user interface element, such as a button name, a menu item name, or a tab name.	Next, click **Submit**.
`monospaced`	Literal text that you can type into code, computer output, class names, URLs, code examples, parameter names, string literals, and other objects that might appear in programming code. In code blocks, bold formatting highlights relevant sections to notice or to configure.	Get the field from the `Address` object.
`monospaced italic`	Parameter names or other variable placeholder text within URLs or other code snippets.	Use `getName(`*`first, last`*`)`. `http://`*`SERVERNAME`*`/a.html`.

Support

For assistance with this software release, contact Guidewire Customer Support:

- At the Guidewire Resource Portal – `http://guidewire.custhelp.com`
- By email – `support@guidewire.com`
- By phone – +1-650-356-4955

part I

PolicyCenter Product Model

chapter 1

Configuring the Product Model

The product model identifies the types of products and policies that your PolicyCenter configuration offers. For each product, the product model specifies the choices about the items that can be covered. Each product model you define is a *product configuration*.

This topic includes:

Product Model Patterns

The product model consists of set of templates called *patterns*. PolicyCenter uses these patterns during policy transactions to generate specific instances of policies and policy objects. The product model provides a large number of patterns. The following patterns are the core patterns you use to start to define a new product. Most of the patterns contain the word "pattern" in their name. However, one exception is `Product`, which does not.

Pattern	See
`Product` – Creates an instance of a product, which is a policy type available to an applicant in the **Submission Manager** screen in PolicyCenter. A product is a pattern that creates new policy instances. PolicyCenter lists each product on a separate row of the **New Submissions** screen. Each product pattern contains at least one PolicyLinePattern.	• "Working with Products" on page 15
`PolicyLinePattern` – Creates an instance of a policy line for a specific line of business, such as businessowners or personal auto. Each policy line pattern contains any number of *clause patterns*. A clause pattern is a generic term that refers to a coverage pattern, exclusion pattern, or condition pattern.	• "Working with Policy Lines" on page 23
`CoveragePattern` – Creates an instance of a coverage, which is a type of loss covered by a policy. A coverage pattern creates an instance of a coverage on a specific policy line.	• "Adding Coverages to a Policy Line" on page 24

Pattern	See
`ExclusionPattern` – Creates an instance of an exclusion, which is a type of loss explicitly not covered by a policy line. An exclusion pattern creates an instance of an exclusion on a specific policy.	• "Adding Exclusions to a Policy Line" on page 37
`ConditonPattern` – Creates an instance of a condition, which is a contractual obligation that is neither providing nor excluding coverage. A condition pattern creates an instance of a condition on a specific policy.	• "Adding Conditions to a Policy Line" on page 37
`CoverageTermPattern` – Creates an instance of a value that specifies the extent, degree, or attribute of coverage, exclusion, or condition. Coverage terms measure or further define a specific clause pattern. One example of a coverage term is a deductible.	• "Defining Coverage Terms" on page 32
`ModifierPattern` – Creates an instance of a modifier that affects the calculation of the policy premium.	• "Modifiers Page" on page 17 • "Adding Quote Modifiers to a Policy Line" on page 40

Creating or modifying a line of business requires working with the product model with two different skill sets: programming skills and business domain skills. In some organizations, a single person performs all requires steps. In other organizations, software developers perform the programming steps while business analysts perform product model design steps. Guidewire provides separate tools for each aspect:

- Guidewire Studio
  - Creates entities and defines their physical implementation in the PolicyCenter database
  - Defines business rules
  - Creates and modifies the PolicyCenter user interface
- Guidewire Product Designer
  - Configures the product model patterns that define a line of business
  - Defines system tables that support the business logic needed by the line of business

Both tools can run on a single computer together with a development instance of PolicyCenter. Alternatively, Product Designer can run on a separate application server enabling multiple business analysts to edit the product mode at one time. The *Product Designer Guide* describes both installation scenarios.

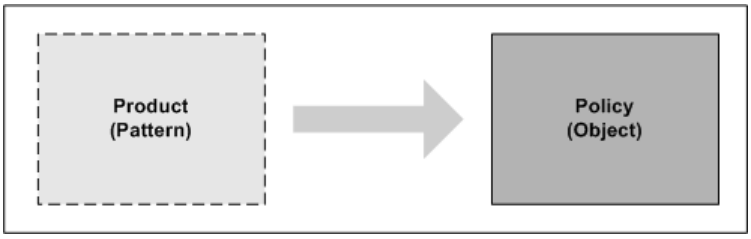
Another part of a complete line of business is the set of forms that accompany each product. PolicyCenter can be integrated with a forms issuance system and each line of business can be configured to infer the forms required for each policy instance. For information about form patterns, see "Policy Form Pattern Administration" on page 697 in the *Application Guide* for information about adding form patterns to products and policy lines.

Product and Policy Objects

The `Product` pattern creates `Policy` instances. The `Product` pattern captures top-level information about the product, such as:

- Whether the product relevant to companies, individuals, or both.
- Whether an applicant is qualified to purchase this product.

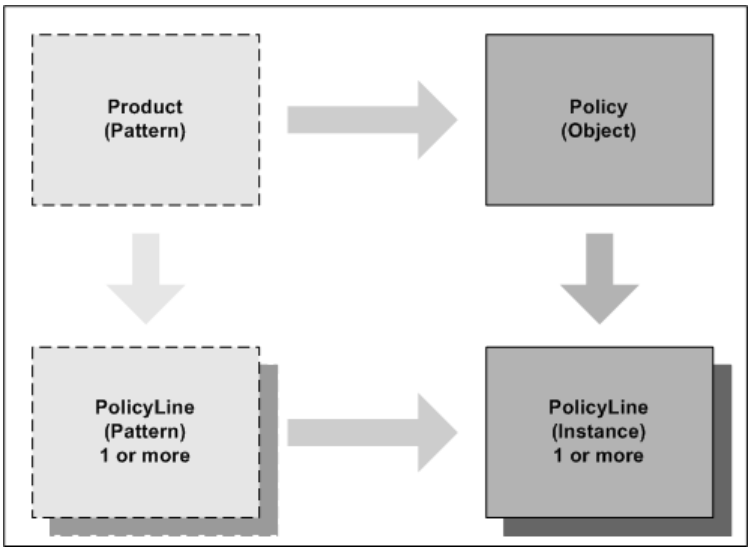
The following graphic shows the relationship between the `Product` pattern and the `Policy` object.



The `PolicyLinePattern` creates `PolicyLine` instances. A policy line pattern is a group of legal and binding information about the product, such as:

- The exposures thbase configurationat can be covered on the policy
- The coverages that are available on the policy

The following graphic shows the relationship between the policy line pattern and an instance of that policy. Notice that a `Product` (pattern) relates to a `PolicyLinePattern` as an instance of a `Policy` relates to an instance of a `PolicyLine`. Each product pattern can have one or more policy line patterns. Likewise, an instance of a policy can have more or more instances of a policy line.



Working with Products

A *Product* represents a type of policy that is available. Each product appears as a separate row in the PolicyCenter **New Submissions** screen.

- A single *product* can have multiple policy line patterns. For example, commercial package policy is a product that includes multiple policy lines: general liability, commercial property, inland marine, and crime.
- A single *policy line pattern* can be used by multiple products. For example, you can use the same general liability policy line pattern in both a general liability product and a commercial package product.

With the exception of commercial package policy, all of the products provided in the base configuration contain only a single policy line pattern.

Topics in this section include:

- “Associating a Policy Line with a Product” on page 20
- “Specifying Policy Terms” on page 20
- “Specifying Advanced Settings for a Product” on page 20
- “Adding Document Templates to a Product” on page 21
- “Adding an Initialization Script” on page 21

Products Pages

To access the PolicyCenter product model, log in to Product Designer and click **Products** in the navigation panel to display the **Products** page. To add a new product, click **Add**. To edit an exiting product, select a product in the **Products** page to display the product page for that product pattern. For example, click **Businessowners** to display the **Businessowners** page. The product page for the selected product has the following sections:

Section	See
Main – Specify product name, description, abbreviation, default policy term, product type, account type, and whether an offering is required to write this type of policy.	• “Defining a Product” on page 19 • “Question Sets Page” on page 16
Policy Lines – Add one or more policy lines to the product. Unless you are defining a package, each product typically maps to one policy line. Each policy line added to a policy adds a link to the **Policy Lines** section. Click a policy line link to jump directly to the corresponding **Policy Line** page.	• “Associating a Policy Line with a Product” on page 20
Policy Terms – Add or remove policy terms. Typical terms include **Annual**, **6 months**, and **Other**.	• “Specifying Policy Terms” on page 20
Advanced – Define advanced settings, such as quote rounding, days until quote needed, integration reference code, and document templates.	• “Specifying Advanced Settings for a Product” on page 20 • “Adding an Initialization Script” on page 21 • “Adding Document Templates to a Product” on page 21

Note: The **Translate** icon appears at the end of fields that can be translated. Click the icon to display the **Display Key Values by Locale** dialog box where you can view or add translated text for each product locale. For example, you can specify that the name of the *Personal Auto* line will be *Automobile Personnelle* in French. For more information, see “Localizing Product Model String Resources” on page 51 in the *Globalization Guide*.

Use links under **Go to** to display other pages related to the selected product. Other product pages are described in the following topics:

- “Question Sets Page” on page 16
- “Modifiers Page” on page 17
- “Offerings Page” on page 17
- “Availability Page” on page 17

See also

- “Policy Form Pattern Administration” on page 697 in the *Application Guide* for information about adding form patterns to products.

Question Sets Page

Question sets are lists of questions that can be used to determine an applicant’s eligibility and to further assess the applicant’s risk potential.

After selecting or adding a product, click **Question Sets** under **Go to** to display the **Questions Sets** page for the selected product. To associate a question set with a product, click **Add** to display the available question sets, and then click the question set to associate.

See also

- "Question Sets" on page 57 to learn how to define and manage question sets.

Modifiers Page

Modifiers are factors that affect rating and typically result in an increase or decrease in the premium for a policy. There are multiple types of modifiers, many of which are specific to a jurisdiction. Modifiers can be added to a product or a policy line. Modifiers added to a product affect rating for all lines in that product. Modifiers added to a policy line affect only that line.

See also

- "Quote Modifiers" on page 51.

Offerings Page

Offerings define product variations, enabling carriers to provide products that vary depending on target customers or other marketing criteria. For example, a Standard personal auto offering defines a standard set of coverages, limits, and deductibles. A Beginning Driver offering has different limits and deductibles to satisfy the needs of drivers who are just starting to drive as well as their financially-responsible parents.

To define or edit offerings, select or add a product, and then click **Offerings** under **Go to** to display the product's **Offerings** page.

See also

- "Configuring Offerings" on page 99
- "Understanding Offerings" on page 481 in the *Application Guide* to learn about offerings and why you use them.

Availability Page

Availability is the product model mechanism that captures whether or not a product model pattern can be used. Specific product model patterns in PolicyCenter can be made available only:

- As of, or until, a given date
- Within (or never within) a given jurisdiction
- When specific underwriting companies are used to write the policy
- Other, more complex conditions that can be expressed in Gosu code.

To view or edit availability for a product, select the product and then click **Availability** under **Go to** to display the **Availability** page. To add a new availability row to the set of availability conditions, click **Add**.

See also

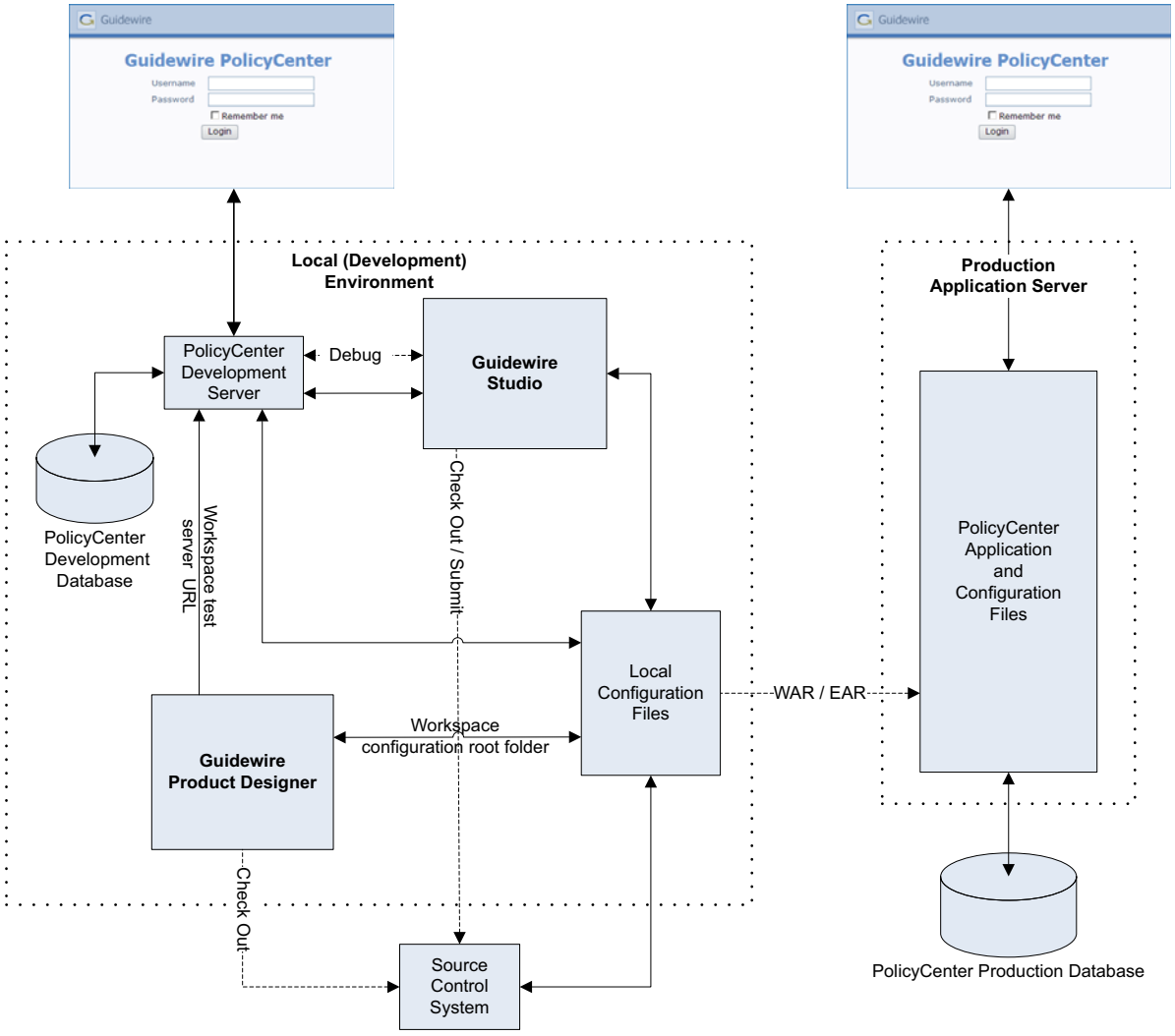
- "Configuring Availability" on page 83

Working with Workspaces and Change Lists

When you edit the PolicyCenter product model in Product Designer, your changes are stored in a *change list* that is associated with a *workspace*.

- A *workspace* is a named association with a set of PolicyCenter configuration files. Product Designer refers to the location of the PolicyCenter configuration files as the *configuration root folder*. Your administrator defines Product Designer workspaces by following the instructions in “Managing Workspaces” on page 25 in the *Product Designer Guide*.
- A *change list* is a named collection of changes that are held by Product Designer until you decide either to commit or revert the changes. Committing your changes means moving them to the application’s configuration files on the PolicyCenter server specified in the workspace with which the change list is associated. Because your changes are in a change list, you can safely make changes and then later choose to commit or not commit some or all of those changes. But until you commit the change list, the changes it contains do not become part of the active PolicyCenter configuration.

Each change list is associated with a workspace. Therefore, each change list, when committed, modifies the configuration files in a specific PolicyCenter instance. The following illustration shows the relationship between Product Designer and other PolicyCenter components.



You can create multiple change lists, and then select the active change list from among them. All changes you make are saved in the active change list. Using multiple change lists enables you to group your changes into separate sets and independently commit each change list at the appropriate time. Unless you assign it to another user, each change list and the changes it contains belong to you. Other users have their own change lists. A change list remains under your control unless you or your administrator reassigns it to another Product Designer user.

A workspace optionally can be configured with a source control system. When configured in this way, Product Designer automatically checks out individual files as each user makes changes to them. However, it does not check them in again. You must manually check in your changed files using the source control system.

In addition to a configuration root folder, your administrator also can designate a *test server URL* for each workspace. **Test Server URL** specifies a running PolicyCenter instance to which to deploy product model changes when you select the **Synchronize Product Model** or **Synchronize System Tables** command in Product Designer.

The **Test Server URL** need not designate the same PolicyCenter instance as the **Configuration Root Folder** for the workspace, although typically they are the same. However, the **Test Server URL** must point to an instance of PolicyCenter that is running in development mode. As indicated in the previous illustration, you cannot deploy the product model to a running instance of PolicyCenter in production mode. Instead, you must instead create a WAR or EAR file and manually deploy the file using the application server's deployment commands. For more information on server modes, see "Working with the QuickStart Development Server" on page 87 in the *Configuration Guide*.

Defining a Product

Use the Product home page to set the parameters that define the product pattern. For specific information about each field, click **Help** to display the **Help** panel. As you work, the changes, additions, and deletions you make are saved in a change list.

Product Model Deployment

After creating a product and committing your changes, you can use Product Designer to deploy the product model to a running PolicyCenter development instance. Or, you can manually deploy your changes to a development or production server.

To deploy your product model updates to a running development instance of PolicyCenter, use the **Synchronize Product Model** command on the **Options** menu. Unless pre-configured by your administrator, to complete this operation you must specify the user name and password of a PolicyCenter user who belongs to a role that has `soapadmin` permission.

After deploying the new product, it is immediately available in the PolicyCenter **Submission Manager**. Although your new product is available, you cannot use it to create submissions, because the product does not have a policy line pattern associated with it. If you attempt to create a Quick Quote or Full Application using the new product, PolicyCenter generates a PCF error. Before you can use the new product to create submissions, you must be able to associate a policy line pattern with it on the **Policy Lines** tab within PolicyCenter. However, before you can do that, you must define a policy line pattern in Product Designer. In addition, before using a new product to create submissions, you must use Studio to either create the wizard screens or to associate the new product with existing wizard screens.

Deleting a Product

You can delete a product or policy line from PolicyCenter. For example, you can delete lines of business that you intend never to use. Before deleting a product, keep in mind that every product must have at least one policy line, and every policy line must be associated with at least one product.

WARNING Deleting a product or line of business can have unanticipated and unintended consequences. Guidewire strongly recommends that you fully understand the product model configuration before taking this action. For example, if you do not correctly remove references to a deleted policy line from your product definition, PolicyCenter cannot synchronize the product model at server startup. As a consequence, it can be impossible to start the server thereafter until the problem is corrected. For more information, see "Product Model Immutable Field Verification" on page 112.

Rather than deleting products and policy lines, consider using availability to set unwanted products to **Unavailable**. Doing so hides the product in the PolicyCenter **Submission Manager** as well as in other areas of the user interface.

Associating a Policy Line with a Product

You associate a policy line pattern with a product in **Policy Lines** section of the Product home page. The policy line pattern must exist before you can add it to the product. You can add one or more policy lines to a product. The Commercial Package product is an example of a product that contains multiple policy lines.

Note: Every product must have at least one associated policy line. Conversely, every policy line pattern must be associated with at least one product. If this requirement is not met, Product Designer validation fails and prevents you from committing your changes.

See also

- "Working with Policy Lines" on page 23 for information on how to create and manage policy line patterns.

Specifying Policy Terms

In Product Designer, the **Policy Terms** section of the product page enables you to select one or more available terms that apply to the product. The terms that you enable under **Policy Terms** are the terms available in PolicyCenter when creating a submission for this product. Set the default term in the **Default Policy Term** drop-down list in the main section of the product page. Most of the options are self-explanatory. The **Other** term enables you create a term specific to a risk.

Note: You define the available term types in Studio in the `TermTypes` typelist.

Specifying Advanced Settings for a Product

In the **Advanced** section of the Product home page, you can specify settings such as quote rounding level, initialization script, and document templates.

Quote Rounding Level Within a Product

Quote rounding determines how fractional monetary amounts are rounded in the quote returned from the rating engine. In addition to specifying the **Quote Rounding Level**, set the **Quote Rounding Mode** to specify how to round fractional amounts.

For example, in the base configuration for personal auto, the quote rounding level is set to 0 for the product. A value of 0 causes PolicyCenter to round quotes to the nearest whole monetary amount. With **Quote Rounding Mode** set to **Half Up**, a value of 231.50 is rounded up to 232.00.

Specific portions of a quote, such as state taxes, can be rounded within the rating engine. For example, the rounding level for taxes is 2 in New York and Florida. A rounding level of two rounds the amount to two decimal places.

If you have Guidewire Rating Management, the **State Tax Calculation** rate routine for personal auto sets the quote rounding mode for taxes in specified states. You can use this rate routine as an example for making similar changes. If you do not have Guidewire Rating Management, you can make this change in Studio. The `rateTaxes` method in `gw.lob.pa.rating.PARatingEngine` sets the quote rounding mode for taxes in specified states. You can use this method as an example for making similar changes.

See also

- "Rate Routines" on page 550 in the *Application Guide*

Adding an Initialization Script

An initialization script specifies Gosu code that PolicyCenter executes every time someone creates a policy from the product pattern. PolicyCenter runs the initialization script when the policy object is instantiated, not when the policy is bound and issued. For example, every time PolicyCenter creates a Personal Auto policy, the following initialization script could set the amount of deposit collected for the policy to zero:

```
PolicyPeriod.DepositCollected = Obd.ofCurrencyPC(PolicyPeriod.PreferredSettlementCurrency)
```

Note: Although it provides a place to type a Gosu script, Product Designer does not perform syntax checking or other coding assistance functions that are common in integrated development environments. Therefore, Guidewire recommends that you define all but the simplest scripts in Studio.

Adding Document Templates to a Product

A *document* is a electronic file, such as a PDF, Microsoft Word, or plain text file, that contains information relevant to a policy or account. Documents, unlike forms, are not contractual parts of a policy. *Document templates* are the frameworks that, after appropriate business data is inserted, become individual document instances.

For example, one document template might be a letter notifying an account holder of the details of three quotes that you prepared. The account holder's name, address, account number, and other personal details are retrieved from PolicyCenter and combined with the document template to create an instance of the document. The resulting document is then delivered to the account holder.

Document templates can be associated with accounts as well as products. Document templates that have been associated with products enable PolicyCenter to generate policy-level documents such as quote and binder documents.

Note: You define the available document template types in Studio in the `DocumentTemplateType` typelist.

See also

- "Creating a Document Template" on page 461 in the *Application Guide*
- "Document Management" on page 191 in the *Integration Guide*
- "Document Creation" on page 103 in the *Rules Guide*

chapter 2

Configuring Policy Lines

This topic describes how to configure policy lines for PolicyCenter.

This topic includes:

Working with Policy Lines

Note: Even though the Product Designer has a node named **Policy Lines**, you are actually working with policy line patterns.

To access the policy line patterns

In the Product Designer home page, click **Product Model**, or select **Product Model** in the navigation panel. In the **Policy Lines** page, select an existing policy line pattern by clicking the link in the **Name** column, or define a new policy line pattern by clicking **Add**.

See also

- "Policy Form Pattern Administration" on page 697 in the *Application Guide* for information about defining form patterns associated with policy lines.

PolicyLine Objects

The `PolicyLine` entity has multiple subtypes. When PolicyCenter creates policy line from a policy line pattern, PolicyCenter needs to know which subtype within the `PolicyLine` entity to use for the new policy line instance. Therefore, PolicyCenter always ties a policy line pattern to one of the policy line subtypes. In the base configuration, the policy line subtypes are:

- Personal auto
- Commercial auto
- Commercial property
- Workers' compensation
- General liability
- Businessowners
- Inland marine

A `PolicyLine` object in Gosu is aware of the product model configuration of its pattern. For example, the coverages defined within the policy line pattern can be referenced from an instance of the `PolicyLine` object. Therefore, the following expression is valid:

```
var x = WCLine.WCEmpLiabCov
```

However, the following expression causes a type error because the businessowners policy (BOP) line does not include workers' compensation coverages:

```
var x = BOPLine.WCEmpLiabCov
```

As another example, consider the `WorkersCompLine` pattern, whose policy line subtype is `WorkersCompLine`. (It is possible—but not required—for a policy line pattern to have the same code as the subtype it creates.) The following definition occurs in `WorkersCompLine.eti`:

```
<subtype entity="WorkersCompLine" displayName="Workers' Comp"
  desc="Workers' Comp" supertype="PolicyLine" generateCode="true">
    ...
    <typekey name="OtherStatesOpt" typelist="OtherStates" desc="Other states option for the coverage"/>
    ...
  </subtype>
```

Therefore, the following Gosu is valid:

```
WCLine.OtherStatesOpt = "None"    //Comes from the configuration dm files
var x = WCLine.WCEmpLiabCov        //Comes from the product model configuration
```

The Gosu compiler merges the type information from the subtype definition and the product configuration to form the final type of the WorkersComp PolicyLine.

Note: No product model configuration type information is available in Java, only the information provided by the configuration `dm` files. Therefore, Guidewire recommends that you exercise extra care with `Policy` objects in Java.

Adding Coverages to a Policy Line

A coverage is a type of loss covered by a policy for a specific property or liability exposure. Coverages are the fundamental building blocks of a policy. Coverages determine what the policy actually covers, and, for the most part, how much the policy costs. In general:

- Liability coverages typically attach at the `PolicyLine` level.
- Property coverages typically attach to other coverable objects.

In some cases, a coverage specifies not only the type of loss but the cause. For example, collision coverage covers damage to a car from auto accidents. Comprehensive coverage covers damage from incidents other than auto accidents (weather, fire, or theft, for example). Suppose that a policy covers a given car for collision but does not provide comprehensive coverage. If a windshield becomes damaged, the cause of loss is relevant to determining whether the policy covers the loss.

Note: Within the insurance industry, different carriers use the term *coverage* differently.

Coverages and Coverables

A *coverage* is protection from a specific risk. A *coverable* is a covered object such as a house or vehicle. In PolicyCenter, coverages attach only to coverables.

Coverable

A coverable is an exposure to risk that can be protected by the policy. A coverable may be a tangible property item, a location, a jurisdiction, or the policy itself. Within PolicyCenter, Guidewire makes the policy line a coverable to represent the named insureds. Coverages attach only to coverables. You can further subdivide coverables into property coverables and liability coverables.

- *Property* coverables are things with physical attributes (height, weight, value, construction type, and age, for example).
- *Liability* coverables are operations that you typically represent with class codes (coal mining or personal auto operation, for example).

Coverage

A coverage is protection from a specific risk. A coverage must be attached to a coverable. Coverages, like coverables, also are divided into two types: property and liability. For example, on an auto policy, a collision property coverage protects the insured's vehicle and a liability coverage protects the driver for damage done to someone else's vehicle. Automobile liability coverage does not insure the vehicle. Rather, it insures its operation.

Using a car as an example to illustrate the different types of coverage:

Risk	Coverage	Coverable
Theft	Property	The coverable is the car and the type of loss is theft.
Collision	Property	The coverable is the car and the type of loss is damage from collision to the vehicle owned by the insured.
	Liability	The coverable is the policy. Liability coverages covers damage to other vehicles and their occupants.

Each `PolicyLine` contains one or more coverable entities and one or more coverages.

Covered Objects

Coverables have the following characteristics:

- All covered objects are coverables.
- Some policy entities are not coverables.
- A coverable entity implements the `CoverableDelegate` interface.
- A `Coverable` delegate encapsulates coverage behavior.

The term *coverable* refers to any covered object. For example, insurance terminology refers to a covered object, such as a house, as a coverable.

- For liability, the insured is the coverable.
- For liability coverages, PolicyCenter designates the policy line as the coverable to represent the insured.
- For coverages that attach at a location, the location is a coverable. Do not use `PolicyLocation` as the coverable, but instead, create a separate coverable entity.

Other Policy Entities

Entities within a policy need not be coverables. For example, general liability stores rating basis information in location-based exposures. This exposure entity is not coverable. Scheduled equipment in the businessowners line is another example. To provide a coverage for set of scheduled items, attach a coverage to a building or location that contains an aggregate limit or deductible as a coverage term. Then create a scheduled item array on the building or location. You can use the stated values of scheduled items to determine a coverage limit. Unless you need separate declared limits or deductibles associated with each scheduled item, you need not attach a coverage to each item in the list. In cases where the set of scheduled items is covered by a single coverage limit, the scheduled item entity is not a coverable.

Coverables and Coverage Tables

Each coverable has at least one coverage table. This coverage table defines the data entity that contains the coverages. Typically, you define one coverage table for each coverable. However, you can associate more than one coverage table with a coverable. For example, if some coverage patterns have characteristics that are different from all other coverages, you can define a second coverage table. Be sure to include in the definition of the coverage table any values needed for all coverage patterns that can be used for the coverable.

Note: In general, Guidewire recommends that you use one coverage table for each coverable, unless there is a clear use case for a second coverage table.

Persisting Coverage Data to the Database

PolicyCenter records the selection of a coverage in the database. It does not store information about electable or suggested coverages that are declined. Therefore, the database stores one row for each coverage that exists on a policy. If you do not select a coverage, PolicyCenter does not store information about it.

Coverables and Delegates

Each coverable delegates to the `Coverable` interface. A delegate is a reusable component that contains an interface and a default implementation of that interface. Using a delegate enables an entity to implement an interface while delegating the implementation of that interface to another class, that of the delegate. Therefore, Guidewire recommends that you use a delegate with objects that share code. The delegate then implements the code rather than each class duplicating the shared code. For example, a delegate encapsulates behaviors that attach a coverage to a coverable or a modifier to a modifiable. You can implement the shared delegate class either in Java or in Gosu.

See also

- “Delegate Data Objects” on page 161 in the *Configuration Guide*

CoveragePattern and Coverage Objects

Within the PolicyCenter product model are two sets of objects that track coverages: the `CoveragePattern` object and the `Coverage` object.

CoveragePattern Objects

A `CoveragePattern` object describes how to create coverages for a given policy line. During policy creation, the coverage patterns determine which coverages must be created, which coverages are optional, and which coverages must not be created.

Much like `PolicyLinePattern` objects and `PolicyLine` objects, the configuration of a coverage pattern affects the Gosu type information of the `Coverage` objects that the pattern creates. The following Gosu is valid, because the BOP `PolicyLinePattern` object has a `BOPAcctRecvCov` coverage pattern configured for it.

```
BOPLine.BOPAcctRecvCov
```

Coverage Objects

A `Coverage` object describes the actual coverage instances for each and every policy. PolicyCenter stores information from this object in the coverage table. During policy creation, the coverage table stores the actual coverage instances created from the coverage pattern. PolicyCenter creates one row for every instance of a coverage for a policy.

In the base configuration, Product Designer has the following behavior:

- The optional **Blanket Group Type** field appears only for coverages in the commercial property policy line.
- The optional **Coverage Symbol Group** page appears only in the commercial auto policy line.
- The optional **Official IDs** page appears only for the workers' compensation policy line.
- The optional **Split Rating Period** appears only for modifiers on the workers' compensation policy line.

To add or remove optional screens or fields on a policy line pattern

1. In Studio, open the property file for the policy line pattern by navigating to **configuration** → **config** and opening:
   `resources/productmodel/policylinepatterns/`*code*`Line/`*code*`Line.properties`
   The properties file is named *code*`Line.properties` where *code* is the value that appears in **Code** at the top of the Policy Line home page. For example, the commercial property properties file is `CPLine.properties`.
2. Set the property to `true` to add the page or field or `false` to remove the page or field. The properties are:

Property	When true, enables
`line.usesBlankets`	**Blanket Group Type** field
`line.usesCoverageSymbolGroups`	**Coverage Symbol Group** page
`line.usesOfficialIDs`	**Official ID** page
`line.usesSplitRatingPeriod`	**Split Rating Period** check box in modifiers

See also

- "Step 6: Add Optional Features to a Policy Line" on page 158 for an example of adding the optional **Blanket Group Type** field to a new policy line pattern.

Terms Page for Coverages

For each coverage pattern, the **Terms** page displays the coverage terms defined for that coverage. On this page, you can **Add** or **Remove** coverage terms. A **Terms** page is available for coverages, conditions, and exclusions.

Reinsurance Page for Coverages

For each coverage pattern, the Reinsurance page enables you to select the reinsurance coverage group, if any, that applies to the coverage. It also enables you to specify a script that returns an `RICoverageGroupType` typekey that can programmatically select a coverage group. If the script returns null, PolicyCenter uses the manually-selected reinsurance group.

The **Reinsurance** page appears only for coverages—not for conditions or exclusions.

Availability Page for Coverages

You can specify the conditions that make a coverage available in the **Availability** page. Coverage availability is based on a variety of factors, including dates, jurisdiction, underwriting company, and job type. You also can write Gosu code in an **Availability Script** to determine availability based upon various factors, including answers to question sets.

Grandfathering enables you to continue to offer a coverage to existing customers, even though the coverage is not otherwise available. Under **Grandfather States**, you can specify a jurisdiction, underwriting company, and end effective date.

Availability pages are provided for products, coverages, coverage terms, options, packages, conditions, exclusions, modifiers, and question sets.

See also

- "Configuring Availability" on page 83

Offerings Page for Coverages

By default, each coverage is included in all offerings and appears in the **Included or implied in these offerings** column in the **Offerings** page. To disable a coverage in an offering, select the offering and click the right arrow to move the offering to the **Disabled in these offerings** column.

Offerings are defined on products. **Offerings** pages are provided for coverages, coverage terms, options, packages, conditions, exclusions, modifiers, and question sets. On the **Offerings** pages, you can define whether a newly-added item is automatically included in the new offering by selecting or clearing the **Include in all new offerings** check box.

Coverages and Schedules

Schedules are lists that contain detailed information about an insured's coverables. Any coverage can be configured to have one or more schedules to collect information about individual covered items. Schedules can be configured to display any number of columns to collect the needed information.

See also

- "Generic Schedules" on page 361 in the *Application Guide*
- "Configuring Generic Schedules" on page 41

Adding a New Coverage

To add a new coverage

1. In Product Designer, navigate to the Policy Line home page for the policy line in which you want to add a coverage.
2. Under **Go to**, click **Coverages**.
3. On the **Coverages** page, click **Add** to display the **Add Coverage** dialog box.
4. Fill in the required fields. If you need help understanding the fields in the **Add Coverage** dialog box, cancel the operation and click **Help**  .

After creating a new coverage, Product Designer displays the Coverage home page where you can define additional coverage details, including:

- Existence
- Coverage Symbol Group
- Reference Date By
- Integration parameters
- Scripts
- Terms
- Reinsurance
- Availability

- Offerings

Configuring Coverages in PCF Files

A typical line of business has many coverages. To improve usability, you can divide coverages into commonly used, and less commonly used, categories. You then can display the commonly used coverages on the **Coverages** tab in PolicyCenter, and the less commonly used coverages on the **Additional Coverages** tab. If you display coverages on the default PCF pages, the coverage category determines on which tab it appears, and whether it always appears or only appears after searching for it.

To summarize:

- Put coverages that appear together into the same category.
- Separate coverages that show automatically from those for which you must search. (Put each type in a separate coverage category.)
- Display the most commonly used coverages on the **Coverages** tab.
- Display the less commonly used coverages on the **Additional Coverages** tab.

Rendering Common Coverages

As PolicyCenter renders a page containing the common coverages, it does the following:

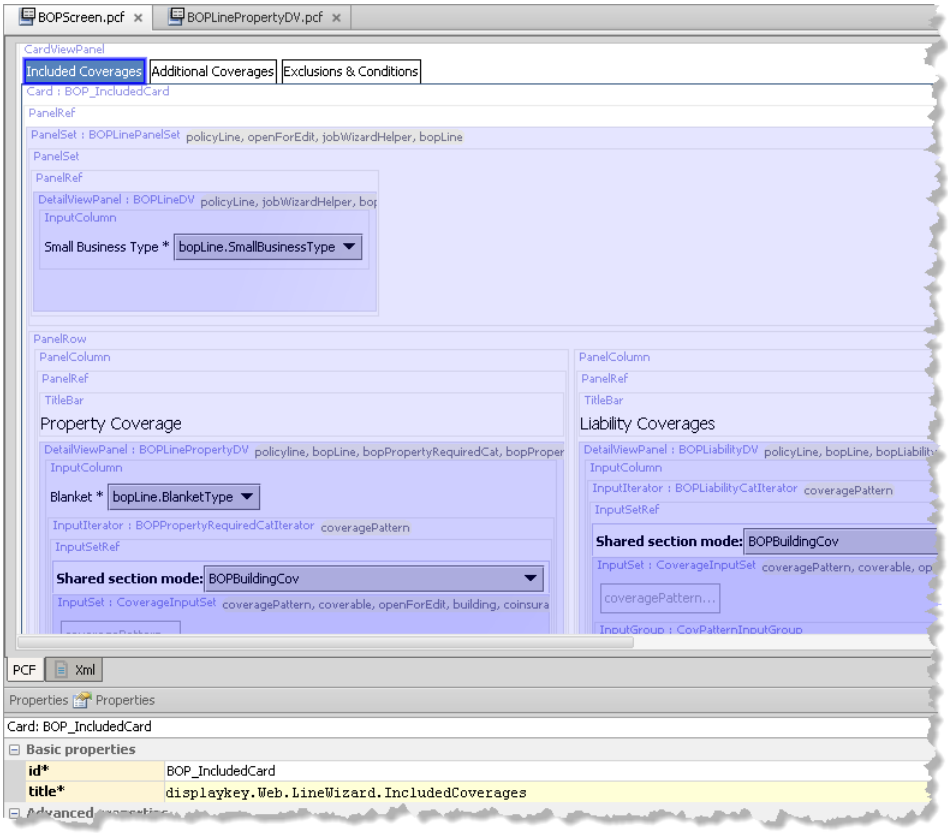
1. Determines which card or cards to render on the page. For example, for `BOPScreen`, PolicyCenter renders (among others) a detail view card named `BOPLinePropertyDV`.
2. Within the each included card, PolicyCenter iterates across the coverage categories that are selected or available. For example, for `BOPLinePropertyDV`, PolicyCenter uses an input iterator with the following ID:

   ```
   id = BOPPropertyRequiredCatIterator
   ```

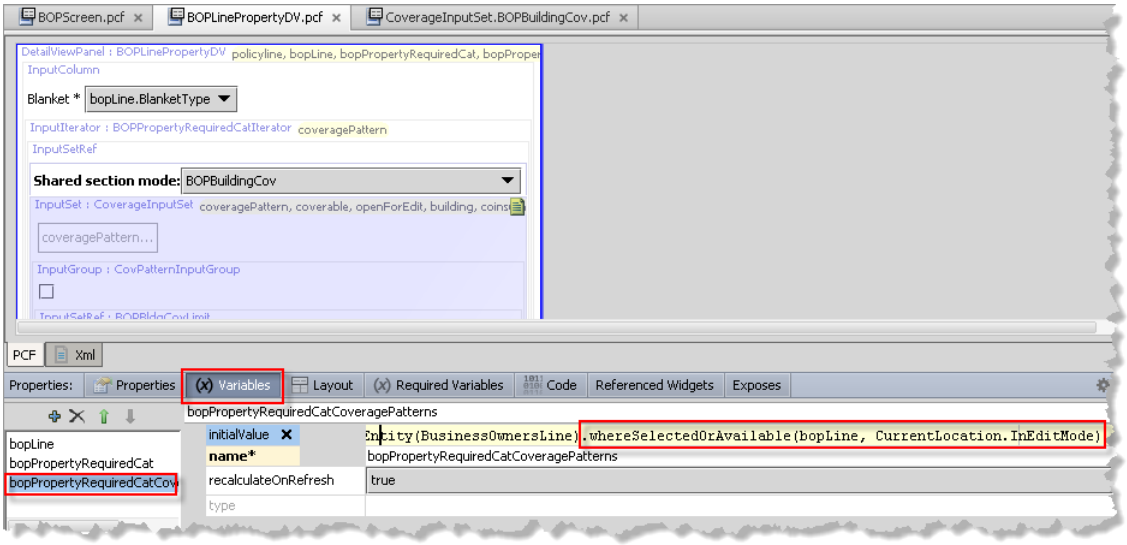
   This iterator has the following `initialValue` that determines whether to include a coverage:

   ```
   bopPropertyRequiredCat.coveragePatternsForEntity(BusinessOwnersLine)
           .whereSelectedOrAvailable(bopLine, CurrentLocation.InEditMode)
   ```
3. Within the coverage category iterator, PolicyCenter uses an `InputSetRef` to render the resulting coverages in PolicyCenter. For example, `BOPPropertyRequiredCatIterator` contains an `InputSetRef` with the iterator `CoverageInputSet()` that iterates across each resulting coverage category and renders its coverages on the page.

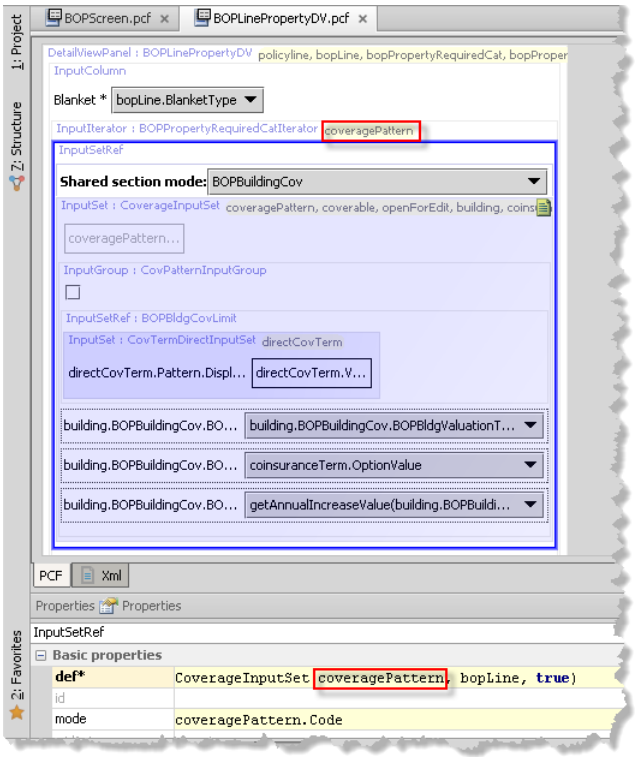
You define this behavior in Studio in `BOPScreen.pcf`. To view this file, press `CTRL+N` and type the file name.



You define this behavior in Studio in `BOPLinePropertyDV.pcf`. In the **Variables** tab of the **Properties** for the `BOPLinePropertyDV.pcf`, the `bopPropertyRequiredCatCov` variable selects the coverages to include.

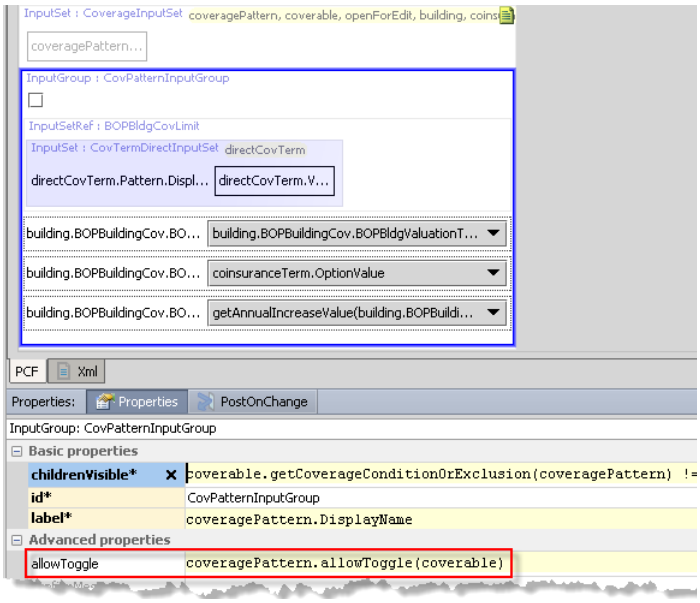


As PolicyCenter builds the interface, it passes the categories for the common categories as a `CoverageInputSet()` parameter (`coveragePattern`) to the `BOPLinePropertyDV` Detail View panel.



PolicyCenter renders a coverage on the screen only if you set the `Exists` bit on the `Coverable`. The `allowToggle` setting in the `CoverageInputSet.BOPBuildingCov` PCF file governs this bit. The `allowToggle` property sets whether the coverage pattern exists on this particular LOB. If you select a coverage (check the check box), then this value toggles on.

For example:



Rendering Less Common Coverages

As PolicyCenter renders a page containing the additional or less common coverages, the following occurs:

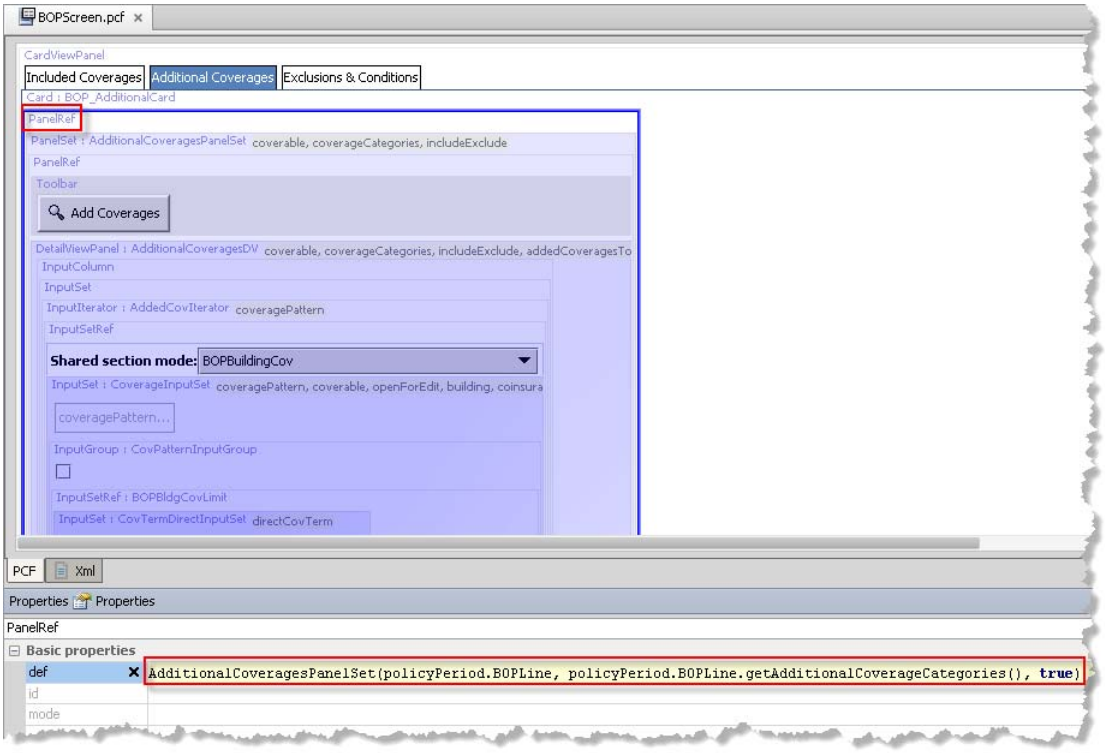
- Iterator `AddedCovIterator` iterates across all the coverage categories.
- `CoverageInputSet` iterates across each coverage category and renders its coverages in the PolicyCenter interface.

While rendering the interface, PolicyCenter passes the categories for the less common categories to the `Additional Coverages` panel set. PolicyCenter uses a method on `BOPLineEnhancement` to determine which coverages to add. The `def` property on the BOP line `PanelRef` illustrates how additional coverages are added.

```
AdditionalCoveragesPanelSet(policyPeriod.BOPLine,
    policyPeriod.BOPLine.getAdditionalCoverageCategories(), true)
```

Note: To change which additional coverages PolicyCenter renders on the screen, modify the list of returned coverages in `BOPLineEnhancement.getAdditionalCoverageCategories()`.

In Studio, `BOPScreen.pcf` provides an example of configuring additional coverages:



Defining Coverage Terms

A *coverage term* is a value that specifies the extent, degree, or attribute of coverage. Coverage terms usually are limits or deductibles. However, coverage terms also can be other terms of coverage, such as a coverage election or scope of coverage. Following are examples of these types of coverage terms:

coverage election	Does Boiler and Machinery coverage include coverage for air conditioning failure?
coverage scope	*How many licensed beauticians are covered by this liability coverage?*

A *coverage term pattern* holds all configuration information for the terms of a coverage. Each coverage pattern has an associated coverage term pattern. PolicyCenter uses the coverage term pattern to create coverage term instances for coverage instances.

A coverage can have zero, one, or many coverage terms:

- Loss of Use coverage in the base commercial auto line is an example of a coverage pattern with no coverage term patterns. After you add the coverage, the extent of the coverage is the same for all policies.
- Hired Auto Collision coverage in the base commercial auto line is an example of a coverage pattern with one term pattern. It has a deductible term, but no other terms, such as a limit.
- The Employee Dishonesty coverage in the base businessowners line is an example of a coverage pattern with multiple coverage term patterns. It has terms for limit, number of covered employees, and number of covered locations.

When you add a coverage term pattern, you specify, among other parameters, its term type and its model type.

Term type	Convention for selecting the value of the coverage term within PolicyCenter. For example, do you choose from a drop-down list, enter a numeric value, or select from a predefined set of packaged values, such as **100/200/300**?
Model type	What the value measures. For example, is the value a limit or a deductible? This information can be used when integrating PolicyCenter with other systems to correctly interpret the coverage term pattern information.

Coverage term patterns are added and configured in Product Designer.

- You set the coverage **Term Type** in the **Add Term** dialog box as you create the coverage term.
- You set the coverage term **Model Type** in the **Integration** section of the Coverage Term home page, after you create the coverage term.

Coverage Term Types

The available coverage term pattern types include:

Type	Description	See
Direct	Numeric value entered directly by the PolicyCenter user, subject to the bounds specified in the `CoverageTermPattern`.	“Direct Coverage Term Type” on page 33
Option	Numeric value selected from a predefined range of coverage term options. Define the list of options in the **Coverage Term Options** page in Product Designer.	“Option Coverage Term Type” on page 34
Package	Compound collection of limits or deductibles selected as a group from a list of predefined choices.	“Package Coverage Term Type” on page 34
Typekey	Value selected from a typelist of predefined choice, optionally filtered through use of a type filter.	“Typekey Coverage Term Type” on page 35
Generic	Custom value, for example, a date or Boolean value.	“Generic Coverage Term Type” on page 35

Direct Coverage Term Type

For Direct coverage term types, you type a numeric values directly into a text box in PolicyCenter. Any numeric value is valid within the bounds specified on the `CoverageTermPattern`. For example, the limit for building coverage relates to the value of the building. The value you enter in PolicyCenter could be 100000 or 1592410.

Direct coverage terms are similar to ordinary numbers. Guidewire provides syntax support in Gosu that you can use to directly compare `DirectCoverageTerm` objects with number literals. For example, consider the following `DirectCoverageTerm`:

```
BOPLine.NewCov.DirectTerm
```

You can assign a value to this term and compare it directly to a numerical value. For example:

```
BOPLine.NewCov.DirectTerm = 100   //Assign a value of 100 to the term
BOPLine.NewCov.DirectTerm > 10    //true
BOPLine.NewCov.DirectTerm < 10    //false
BOPLine.NewCov.DirectTerm == 100  //true
```

Thus, you can treat a Direct coverage term as a number and use natural syntax in any Gosu code.

Option Coverage Term Type

For Option coverage term types, you select a single numeric value from a predefined list of options in PolicyCenter. For example, options such as 100, 500, and 1000 enables you to select only those specific deductible limits. Option coverage term types enable you to choose specific values and do not let you freely specify any number.

You can assign Option coverage term type objects to, and compare them with, the string literals that are the `OptionCode` values of the coverage term options from their patterns. For example:

```
BOPLine.NewCov.OptionTerm = "100"   //Assign value of the term to the CovTermOpt with the value "100"
BOPLine.NewCov.OptionTerm == "100"  //true
BOPLine.NewCov.OptionTerm == "1000" //false
```

Package Coverage Term Type

A package coverage term type is a compound collection of limits or deductibles selected as a group, rather than a single value as with an Option coverage term type. You select a package in PolicyCenter from a predefined list of values.

After you define the multiple compound values in a Package coverage term type, you must also provide information to identify the meaning of each value. For example, you must identify whether a value is a limit, and, if so, to what it applies.

Each of the compound values in a Package coverage term type represents one set of multiple terms for a coverage that must be selected as unit. For example, a medical payments liability coverage for a commercial auto policy might have the following limits and permissible values:

- A limit for each person – 10000 and 25000.
- A limit for each vehicle – 20000 and 50000.
- A limit for each accident – 25000, 50000, and 100000.

However, only some combinations of values make sense from a business or legal perspective. For example, selecting the combination 25000/50000/25000 does not make sense. This combination sets the limit for each vehicle to a higher value than the limit for each accident. If permitted, this combination would result in ineffective coverage if two vehicles are involved in a single accident.

To simplify selecting a valid and meaningful combination of coverage terms, PolicyCenter displays the limit combinations in a single list. In this list, each value is actually a set of the permissible values. For example, an automobile liability package might have the following coverages:

- Each claimant Bodily Injury
- Each accident Bodily Injury
- Each accident Property Damage

The permissible values for these coverages then appear as the following package limits:

- 10000/20000/50000
- 10000/20000/100000
- 25000/50000/100000

You can assign Package coverage terms to, and compare them with, string literals that are the `PackageCode` objects of the coverage term packages from their patterns. For example:

```
BOPLine.NewCov.PackageTerm = "10/50/100" //Assign the value with the code "10/50/100"
BOPLine.NewCov.PackageTerm == "10/50/100" //true
BOPLine.NewCov.PackageTerm == "20/60/200" //false
```

Typekey Coverage Term Type

For Typekey coverage term types, you select a value from a list of choices known as a typelist. Optionally, the list of choices can be filtered through the use of a typefilter. For information on typelists and typefilters, see "Working with Typelists in Studio" on page 269 in the *Configuration Guide*.

Generic Coverage Term Type

You define Generic coverage term types as custom values which must be a string, date and time, or Boolean value. A date and time value is formatted as `yyyy-MM-dd HH:mm:ss.SSS`. You cannot customize the Generic coverage term.

Coverage Term Model Types

You set the coverage term model type in Product Designer by expanding the **Integration** section on the Coverage Term home page. (after you select the coverage term). In the base application, every coverage term has one of the following possible model type values:

- **Coinsurance**
- **Deductible**
- **Exclusion**
- **Limit**
- **Peril**
- **Other** – Information about the coverage that is not a deductible, exclusion, limit, or coinsurance.

Each of these model types defines how the coverage term measures the coverage legally, and, how the rating engine uses the coverage term value.

Adding New Coverage Term Patterns

Use Product Designer to add a new coverage term pattern.

To add a new coverage term pattern

1. In the navigation panel, expand the tree to locate the coverage pattern to which to add the new coverage term.
2. Select the coverage pattern to open the Coverage home page
3. Under **Go to**, click **Terms** to display the Coverage Terms page.
4. Click **Add Term** to display the **Add Term** dialog box.

The following table describes the fields in the dialog.

Field	Description
Code	Coverage term code, which must begin with a letter and be composed only of letters, numbers, and underscores. Established when you add a new term and thereafter cannot be changed.
Name	Name of the coverage term as it appears in PolicyCenter.
Type	Type of coverage term, one of **Direct**, **Option**, **Package**, **Typekey**, or **Generic**. See "Coverage Term Types" on page 33 for descriptions of each type.
Typelist	Appears only if the **Type** is **Typekey**. Typelist from in which the term type is defined.
Required	Whether the coverage requires this coverage term. If selected, PolicyCenter displays an error if you do you supply a value for the coverage term.
Column	Database table column to use for the coverage term.

IMPORTANT Guidewire provides availability logic and validation for Money value types in Direct, Option, and Package term types, and therefore recommends that you use these term times for monetary values. If you instead choose to use a Typelist to hold monetary values, you must provide all needed implementation and validation required to handle multiple currencies.

It is important to understand that from a configuration standpoint:

- The choices of coverage term type and options (if relevant) are immediate and apparent. These choices control the look of the widget on the interface and the options that appear in a list.
- The choice of coverage term model type is neither immediate nor apparent. The behavior of a coverage term in the interface is the same regardless of the model type. Instead, the model type has implications when passed to the rating engine, and when you import a policy from a legacy policy system into PolicyCenter. Therefore, even though it has no impact on configuration behavior, the choice of model type is as important as the choice of coverage term type.

The coverage input widget includes one embedded coverage term input for each coverage term associated with the coverage. Therefore, in most cases you need make no additional PCF configuration to render coverage terms.

Coverage Term Availability

Each coverage term has an Availability page that defines the time period and other conditions that determine whether the coverage term is available.

In addition, each coverage term option and coverage term package has an Availability page that controls whether the individual option or package item is available.

For example, Collision coverage in the personal auto line has an Option coverage term called **Collision Deductible** with five options. You can set availability separately on each of the five options, and you can set availability for the entire coverage term.

Note: Guidewire recommends that you use the individual item availability feature sparingly due to its impact on performance.

Adding Exclusions to a Policy Line

Exclusions define causes of loss that are explicitly not covered by the policy, so that the carrier has no exposure to claims in those areas. PolicyCenter supports exclusions in the following ways:

- **Boolean values at the policy line level** – The majority of exclusions exclude something from the policy line. As with coverages, each exclusion of this type can have its own terms, availability, and offerings.

  You use Product Designer to add policy line-level exclusion patterns in the same way you add coverage patterns. For instructions, see "Adding Coverages to a Policy Line" on page 24.

- **Schedules of excluded items** – Exclusions can be lists of excluded items that get included on a schedule form.

  An example of this type of exclusion can be seen in the base workers' compensation line. The **WC Options** screen displays a list of excluded workplaces. The excluded workplace is a separate entity with a few fields defined, and the policy line has an array of these entities. This type of exclusion requires custom configuration and need not be included in the product model.

- **Restrictions on a particular coverage** – Some exclusions are related to a particular coverage and optionally exclude items from that coverage.

  You can model these exclusions as coverage terms in the product model. The base personal auto line has an example of this type of exclusion. The `PIP - NY` coverage has a typelist coverage term for `Exclude Medical` that enables you to optionally exclude medical coverage from personal injury protection in New York.

Adding exclusions to a policy line is similar to adding coverages. However, in most cases, exclusions are not rated, so they do not have associated cost subtypes. To add exclusions to a line of business, follow the same steps that you would follow for a coverage but omit costs. For instructions, see "Adding Coverages to a Policy Line" on page 24.

Exclusions and Schedules

Schedules are lists that contain detailed information about an insured's coverables. Any exclusion can be configured to have one or more schedules to collect information about individual excluded items. Schedules can be configured to display any number of columns to collect the needed information.

See also

- "Generic Schedules" on page 361 in the *Application Guide*
- "Configuring Generic Schedules" on page 41

Adding Conditions to a Policy Line

Policy conditions are contractual obligations defined in the insurance contract. Policy conditions neither provide nor exclude coverage.

Unlike coverages and exclusions, policy conditions are diverse and cannot be easily characterized. Any variety of additional data might need to be associated with a policy condition. Conditions vary from simple and obligatory such as, "You will pay your bill," to complex concepts such as retrospective rating in workers' compensation policies.

Simple conditions that are related to coverages can be modeled as coverage terms that are not limits or deductibles. Other conditions can be modeled as optional values that apply to all the coverages on a policy, such as an aggregate policy limit or a maximum covered item value. A common use of a condition is a deductible that is shared by many different coverages, where the coverages are selected individually and might be attached to different coverables.

Specify optional conditions in the Conditions page of Product Designer. Do not configure standard conditions that apply to all policies such as, "You will pay your bill," in Product Designer. Instead, configure these as text directly on the standard policy forms.

Adding conditions to a policy line is similar to adding coverages. However, in most cases, conditions are not rated, so they do not have associated cost subtypes. To add conditions to a line of business, follow the same steps that you would follow for a coverage but omit costs. For instructions, see "Adding Coverages to a Policy Line" on page 24.

Conditions and Schedules

Schedules are lists that contain detailed information about an insured's coverables. Any condition can be configured to have one or more schedules to collect information about individual items to which the condition applies. Schedules can be configured to display any number of columns to collect the needed information.

See also

- "Generic Schedules" on page 361 in the *Application Guide*
- "Configuring Generic Schedules" on page 41

Defining Categories

PolicyCenter groups coverages, exclusions, and policy conditions into categories. Categories are only a grouping mechanism.

You define categories for each product line in Product Designer in the Categories page for the selected policy line. You assign a category to a coverage, exclusion, or condition in the corresponding Coverage, Exclusion, or Condition home page.

Defining a Coverage Symbol Group

Coverage symbol groups apply only to the commercial auto line. A commercial auto policy prints these symbols on the policy declarations page to indicate the types of coverages in effect.

During the submission process, PolicyCenter uses a set of rules to assign the initial coverage symbols from the selected coverages. Some carriers use the automatic symbol assignment as specified by the rules, without changes. Other carriers allow certain individuals to amend the automatically-generated coverage symbols. In PolicyCenter, only someone with the **Edit covered auto symbols** permission (`editautosymbol`) can modify the default symbol assignment.

You add coverage symbol groups in Product Designer in the Coverage Symbol Groups page of policy lines that have the necessary backing data to support them.

In the base configuration, the following appear only for the commercial auto policy line pattern:

- In Product Designer, the **Coverage Symbol Groups** link under **Go to** and the corresponding Coverage Symbol Groups page
- In PolicyCenter, the **Covered Vehicles** screen that displays a list of coverages and coverage symbols that can be edited by a user with the appropriate permissions, as previously explained

To add or remove coverage symbol groups on a policy line pattern

1. In Studio, open the property file for the policy line pattern by navigating to **configuration** → **config** and opening:
   `resources/productmodel/policylinepatterns/`*`code`*`Line/`*`code`*`Line.properties`
   The properties file is named *`code`*`.properties` where *`code`* is the code of the policy line. For example, the commercial auto properties file is `BusinessAutoLine.properties`.

2. Set the `line.usesCoverageSymbolGroups` property to `true` to add the tab, or `false` to remove the tab.

See also

- "Step 6: Add Optional Features to a Policy Line" on page 158 for more information about adding the optional **Coverage Symbol Groups** tab to a new policy line pattern.

Symbols

Symbols are specific to the line of business. The insurance industry sets the definition of symbols. For copyright reasons, PolicyCenter does not include the industry standard symbols, but you can add them in. In the base configuration, the `CoverageSymbolType` typelist defines the following symbols:

Code	Meaning
ANY	Any Vehicle
OVO	Owned Vehicles Only
OPV	Owned Private Passenger Vehicles Only
OCV	Owned Commercial Vehicles
SRC	Compulsory State Requirement
DVO	Designated Vehicles Only
HVO	Hired Vehicles Only
NOV	Non Owned Vehicles
CUS	Custom Definition

PolicyCenter includes an extra coverage symbol designated as a custom symbol (`CUS`). It is only available to someone with the **Edit covered auto symbols** permission (`editautosysmbol`). If you select this coverage symbol, you can enter a free-form text description.

Business Purpose of Symbols

Your choice of symbols is important because not only do they describe selected coverages, they also trigger coverages. For example, suppose the insured purchased a new vehicle that was in an accident prior to an endorsement request. A covered auto symbol of `ANY` or `OVO` on liability coverage would dictate that the new vehicle automatically had coverage. Therefore, a liability claim would be paid. However, if instead you chose `HVO`, the new car would not have liability coverage because it had been purchased. The vehicle would require a backdated endorsement for the claim to be covered.

PolicyCenter Usage

Within PolicyCenter, you create coverage symbol groups and specify coverage symbols within the policy line. You then can specify the coverage symbol group to which each coverage pattern belongs.

Configure coverage symbols in the Gosu enhancement `gw.lob.ba.BusinessAutoLineEnhancement`, which enhances business auto line objects. The enhancement method `setCoveredAutoSymbols` configures which coverage auto symbols to select by default in PolicyCenter.

Defining Official IDs

Use an official ID to specify an identifier associated with a particular line of business. In the base configuration, workers' compensation is the only line of business that uses official IDs. For example, in workers' compensation, you use a bureau ID to identify an entity for statistical reporting.

You set the any official ID values in Product Designer using the **Official IDs** page in the workers' compensation policy line. The **Official IDs** link appears under **Go to** only in the workers' compensation line of business.

When defining an official ID, the **Scope** attribute has the following meanings:

- An **Insured and State** official ID applies to additional named insureds. The base configuration of workers' compensation captures the tax ID of each additional named insured as an official ID on the policy.
- A **State** official ID applies to the primary named insured only. Generally, you must specify a different ID for each state in which this ID applies. The Bureau ID in workers' compensation is an example of an ID that varies by state. The system uses the bureau ID to track statistics reporting between NCCI and non-NCCI states.

In the base configuration, the **Official IDs** page in Product Designer appears only for the workers' compensation policy line pattern.

To add or remove the optional Official Ids tab on a policy line pattern

1. In Studio, open the property file for the policy line pattern by navigating to **configuration** → **config** and opening:

   `resources/productmodel/policylinepatterns/`*`code`*`Line/`*`code`*`Line.properties`

   The properties file is named *`code`*`.properties` where *`code`* is the code of the policy line. For example, the workers' compensation properties file is `WorkersCompLine.properties`.
2. Set the `line.usesOfficialIDs` property to `true` to add the tab, or `false` to remove the tab.

See also

- "Step 6: Add Optional Features to a Policy Line" on page 158 for more information about adding the optional **Official IDs** tab to a new policy line pattern.

Adding Quote Modifiers to a Policy Line

Modifiers are factors that are applied as part of the rating algorithm. They frequently result in an increase or decrease in the premium for a policy. There are multiple types of modifiers, with most being jurisdiction specific. Modifiers are specified by the user.

Modifiers can be added to the product or policy line. Modifiers added to the product affect rating for all lines in that product. Modifiers added to a policy line affect only that line.

For details on working with quote modifiers, see "Quote Modifiers" on page 51.

chapter 3

Configuring Generic Schedules

Schedules are lists that contain detailed information about an insured's coverables. PolicyCenter implements two distinct types of schedules:

- Schedules that capture information per scheduled item. Schedules of this type are not used directly for rating, but are often taken into consideration during underwriting and usually are included in forms.
- Schedules that include coverage terms per scheduled item. In schedules of this type, each scheduled item's coverage terms are passed to the rating engine and potentially affect the cost of the policy.

Generic schedules are instances of schedules that are implemented by using the generic schedule data model in PolicyCenter.

This topic explains how to configure generic schedules.

This topic includes:

See also

- "Generic Schedules" on page 361 in the *Application Guide*

Generic schedules provide a template that enables you to quickly configure new schedules. Generic schedules avoid the need to define a new data model, new user interface, and new configuration elements each time you

want to add a new schedule. Using the generic schedule feature, you can quickly adapt the generic structure to represent many different schedules.

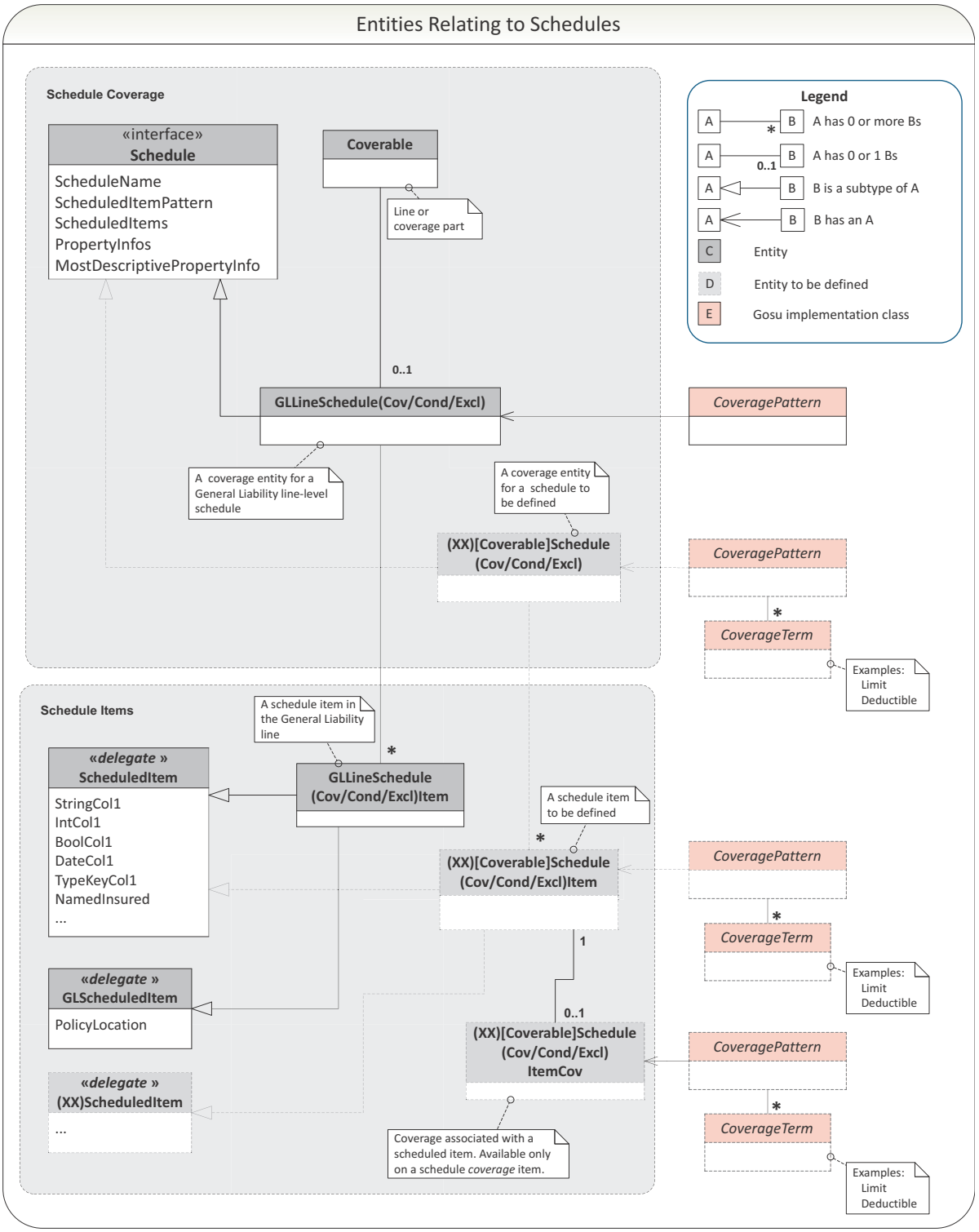
Generic Schedule Data Model

The important characteristic of generic schedules is their flexible data model. The generic schedule data model includes one set of entity types that have many columns. Predefined columns include StringCol1, StringCol2, TypeKeyCol1, PosIntCol1, and so forth. These columns are not inherently descriptive, but instead provide a generic way to store information. When configuring a new schedule, you use these generic columns and provide a description for them in the context of the schedule. For example, the schedule **Exclude Y2K Computer Related And Other Electronic Problems** contains:

- A **Description** column that is stored in `StringCol1`
- A **Type** column that is stored in `TypeKeyCol1`
- A **Bodily Injury Excluded** column that is stored in `BoolCol1`
- A **Property Damage Excluded** column that is stored in `BoolCol2`

The following data model displays the key entities related to schedules. Examples of schedules are provided in the general liability line that is part of the base PolicyCenter product. Schedules can be implemented in other lines by configuring the product model and the generic schedule data model.

See the *Data Dictionary* for a complete list of entities and properties. The entities in the object model are not a complete list.



The generic schedule data model defines a schedule coverage and a set of schedule items.

Schedule Coverage

The Schedule Coverage portion of the data model defines the `Schedule` interface that describes the properties of the schedule. This interface is implemented by `AbstractScheduleImpl`. You extend this implementation class for each schedule you want to define. You can find an example of a schedule implementation in the `gw.lob.gl.schedule.GLLineScheduleExclImpl` class.

Schedule Items

Schedule items are the individual rows of a schedule, one row per item to be captured by the schedule. The data captured for each schedule item is defined in columns that appear in the schedule user interface. The Schedule Items portion of the data model defines the `ScheduledItem` delegate, which defines the following a set of column types that are common to most schedules:

`ScheduleNumber`	Column of type `integer` that automatically numbers the items in a schedule.
`StringCol1, StringCol2`	Two instances of a `shorttext` type column.
`IntCol1`	Column of type `integer`.
`PosIntCol1`	Column of type `positiveinteger`.
`BoolCol1, BoolCol2`	Two instances of a `bit` type column.
`DateCol1`	Column of type `dateonly`.
`TypeKeyCol1, TypeKeyCol2`	Two instances of a `patterncode` type column.
`PolicyNamedInsured`	Foreign key to a named insured.

If any schedules within a coverage, condition, or exclusion requires other column types, you must create a new scheduled item delegate that defines those column types. To see an example of extending the column type definitions, in Studio navigate to **configuration** → **config** → **Metadata** → **Entity** and open `GLScheduledItem.eti`. This example shows how General Liability defines a `PolicyLocation` foreign key that can be used as a column in a schedule.

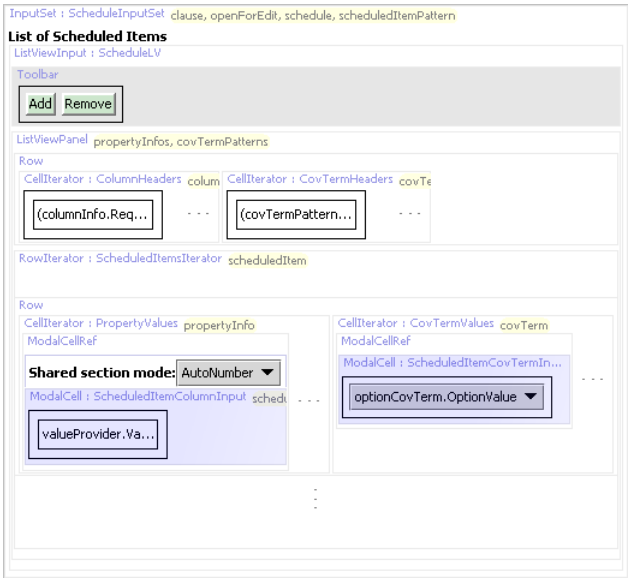
For each coverage, condition, or exclusion in which you want to define one or more schedules, you must define new entities that implement the patterns in the delegates. To see an example of implementing the delegates for General Liability, in the Studio **Project** window, navigate to **configuration** → **config** → **metadata** → **entity** and then open `GLLineScheduleCovItem.eti`.

See also

- “Generic Schedule Data Model” on page 42
- “Modifying the Base Data Model” on page 211 in the *Configuration Guide*
- “Extending a Base Configuration Entity” on page 215 in the *Configuration Guide*

Generic Schedule User Interface

To display different types of schedules without using a different PCF for each one, PolicyCenter uses a single, flexible user interface. This single user interface uses a generic PCF page, `ScheduleInputSet.pcf`, that requires a set of column information for each schedule. The PCF uses the column information to define the input columns for the schedule instance.



For example, in the schedule **Exclude Y2K Computer Related And Other Electronic Problems** schedule, one of the columns is labeled **Description** and another is labeled **Type**. Column labels and types are mapped in Gosu schedule adapter classes. The mapping process varies depending on whether the column is a coverage term. The PCF uses the column mapping information to determine the correct type of cell and creates a modal cell input. For example, a **Description** column requires a text cell and a **Type** column requires a range input that lists the values in a typekey. The following topics show examples of how each type of scheduled item is configured.

Defining Schedules

This topic explains how to define a typical coverage schedule in which none of the schedule items have coverage terms.

The example presented in these topics explains how to create a schedule named **Surveillance Equipment Schedule** in the businessowners line. This topic is presented in two parts:

- “Implementing Schedules in Lines That Do Not Have Schedules” on page 45
- “Configuring a Schedule” on page 49

Implementing Schedules in Lines That Do Not Have Schedules

To begin, you must implement generic schedules for the line of business and clause type (coverage, condition, or exclusion) as needed. Implementation includes creating new entities and classes that take advantage of the built-in framework that PolicyCenter provides for adding generic schedules to any line of business.

Step 1: Create a Subtype Entity that Implements the Schedule Interface

The first step in implementing a generic schedule for a coverable that does not yet have schedules is to define an entity that is implements the `Schedule` interface. Within a given coverable, you need only a single instance of this entity per clause (coverage, condition, or exception). The **Surveillance Equipment Schedule** example applies to a coverage.

In Studio, in the Project window, navigate to **configuration** → **config** → **metadata** → **entity** and create a new file named `BOPLineScheduleCov.etx`. The entity must specify an implementation class that extends `AbstractScheduleImpl`, and it must create an array to store the scheduled items, as shown in the following example:

```
<?xml version="1.0"?>
<subtype
  xmlns="http://guidewire.com/datamodel"
  desc="BOP Line coverage with a schedule"
  entity="BOPLineScheduleCov"
  final="false"
  supertype="BusinessOwnersCov">
  <implementsInterface
    iface="gw.api.productmodel.Schedule"
    impl="gw.lob.bop.schedule.BOPLineScheduleCovImpl"/>
  <implementsEntity
    name="ScheduleAutoNumberSequence"/>
  <array
    arrayentity="BOPLineScheduleCovItem"
    cascadeDelete="true"
    desc="Scheduled Items"
    name="BOPScheduledItems"
    setterScriptability="external"/>
</subtype>
```

Step 2: Create a Delegate for Additional Column Types

If all of the columns needed in the line of business can be mapped to the column types defined in the `Schedule` interface, skip this step.

If any schedules in the line require columns that are not defined in the `Schedule` interface, create an *`LOB`*`ScheduleItem` delegate entity that defines the additional column types. In the **Surveillance Equipment Schedule** example, the **Location** column needs to reference a policy location. Policy location is not defined in the `Schedule` interface.

In Studio, in the Project window, navigate to **configuration** → **config** → **metadata** → **entity** and create a new file named `BOPScheduledItem.etx`. Define a delegate as shown in the following example:

```
<?xml version="1.0"?>
<delegate
  xmlns="http://guidewire.com/datamodel"
  effdatedOnly="true"
  extendable="true"
  name="BOPScheduledItem">
  <fulldescription>A BOP scheduled item</fulldescription>
  <foreignkey
    desc="BOP Location"
    fkentity="BOPLocation"
    exportable="true"
    name="BOPLocation"/>
</delegate>
```

Step 3: Define a Schedule Implementation Class

The implementation class must extend `AbstractScheduleImpl<entity.`*`schedule_subtype`*`>`, where *`schedule_subtype`* is the entity that you defined in Step 1. This implementation class must override the methods and properties that the schedule requires, including the `PropertyInfos` property that maps column types to schedule columns and their column heading display keys.

All schedules for a clause type (coverage, condition, or exclusion) are defined in this single class. In the **Surveillance Equipment Schedule** example, the schedule applies to a coverage. In Studio, in the **Project** window, navigate to **gsrc** → **gw** → **lob** → **bop**. Create a new package named `schedule`, and in that package, define the new class as shown in the following example:

```
package gw.lob.bop.schedule
uses gw.lob.common.AbstractScheduleImpl
uses gw.api.productmodel.SchedulePropertyInfo
uses gw.lang.reflect.IPropertyInfo
uses gw.api.productmodel.ScheduleStringPropertyInfo

@Export
class BOPLineScheduleCovImpl extends AbstractScheduleImpl<entity.BOPLineScheduleCov> {

  construct(delegateOwner : entity.BOPLineScheduleCov) {
    super(delegateOwner)
  }

  override property get ScheduledItems() : ScheduledItem[] {
    return Owner.BOPLineScheduledItems
  }

  override function createAndAddScheduledItem() : ScheduledItem {
    var scheduledItem = new BOPLineScheduleCovItem(Owner.Branch)
    createAutoNumber(scheduledItem)
    Owner.addToBOPLineScheduledItems(scheduledItem)
    return scheduledItem
  }

  override property get PropertyInfos() : SchedulePropertyInfo[] {
    switch (typeof Owner) {
      case BOPSurveillanceEquipmentScheduleCov:
        return {
          new BOPScheduledItemPropertyInfoString("StringCol1",
                  displaykey.Web.Policy.BOP.Schedule.Name, true),
          new ScheduleBOPPolicyLocationPropertyInfo("PolicyLocation",
                  displaykey.Web.Policy.BOP.ScheduleLocation,
                  new IValueRangeGetter() {
                    override property get ValueRange() : KeyableBean[] {
                      return Owner.BOPLine.Branch.PolicyLocations
                    }},
                  true),
             new BOPScheduledItemColumnInfoTypeKey<BOPSurveillanceType>("TypeKeyCol1",
                 displaykey.Web.Policy.BOP.BOPSurveillanceType, true) }
         case anotherSchedule:
              .
              .
              .
        }
        default:
          return super.PropertyInfos
    }
  }

  override function removeScheduledItem(item : ScheduledItem) {
    Owner.removeFromBOPLineScheduledItems(item as BOPLineScheduleCovItem)
    renumberAutoNumberSequence()
}

  override property get CurrentAndFutureScheduledItems() : KeyableBean[] {
    var schedItems = Owner.ScheduledItems.toList()

    Owner.Branch.OOSSlices
      .where(\ p ->  p.BOPLine != null)
      .each(\ p ->  {
        var matchingSlicedScheduleCov = p.BOPLine.CoveragesFromCoverable.firstWhere(\ c -> c.FixedId
                == Owner.FixedId) as BOPLineScheduleCov
        if (matchingSlicedScheduleCov != null){
          matchingSlicedScheduleCov.ScheduledItems.each(\ s -> {
            if(!schedItems.contains(s)) {
              schedItems.add(s)
            }
          })
        }
      })
    return schedItems.map(\ item -> item as BOPLineScheduleCovItem).toTypedArray()
  }

  override property get ScheduleNumberPropInfo() : IPropertyInfo {
```

```
    return BOPLineScheduleCovItem.Type.TypeInfo.getProperty("ScheduleNumber")
  }
}
```

The last column in the example schedule column mapping implements a column type of `TypeKeyCol1`. You must create this typekey and add a list of representative items to display in the schedule's **Type** column.

IMPORTANT The syntax for a typekey column is different than the syntax for other column definitions, in that the constructor must be parameterized against the typekey's type. For example,

```
new BOPScheduledItemColumnInfoTypeKey<BOPSurveillanceType>
```

Incorrectly declaring or omitting the typekey's type results in a runtime failure.

Step 4: Create a Scheduled Item Entity

The scheduled item entity is the concrete implementation of the scheduled items for a particular clause (coverage, condition, or exclusion). It defines the database columns where schedule items are actually stored. The scheduled item entity must implement the `Coverable` entity as well as the `ScheduledItem` entity and in each case specify the line-specific adapter class to be defined in Step 5. If your schedule requires additional columns that have been defined in the *LOB*`ScheduledItem` delegate entity, the scheduled item entity also must implement that delegate. In addition, the scheduled item entity must have a foreign key that links to the `Schedule` delegate. As with the other entities, all scheduled items for a given clause type (coverage, condition, or exclusion) are defined by a single entity.

In Studio, in the Project window, navigate to **configuration** → **config** → **metadata** → **entity** and create a new file named `BOPLineScheduleCovItem.etx`. Create a new entity as shown in the following example:

```
<?xml version="1.0"?>
<entity
  xmlns="http://guidewire.com/datamodel"
  entity="BOPLineScheduleCovItem"
  table="boplineschedcovitem"
  desc="BOP Line level coverage scheduled item"
  exportable="true"
  final="false"
  loadable="false"
  type="effdated"
  platform="false"
  effDatedBranchType="PolicyPeriod">
  <implementsEntity
    name="BOPScheduledItem"/>
  <implementsEntity
    name="ScheduledItem"
    adapter="gw.lob.bop.BOPLineScheduleCovItemCoverableAdapter"/>
  <implementsEntity
    name="Coverable"
    adapter="gw.lob.bop.BOPLineScheduleCovItemCoverableAdapter"/>
  <foreignkey
    name="Schedule"
    fkentity="BOPLineScheduleCov"
    nullok="false"/>
</entity>
```

Step 5: Define a Line-specific Schedule Coverable Adapter Class

The line-specific schedule coverable adapter class overrides the generic properties and methods of the `AbstractScheduledItemAdapter` class that it extends. The properties and methods to override depend on the clause type (coverage, condition, or exclusion) to which your schedule applies. The following adapter overrides the properties and methods that apply to a coverage.

In the **Surveillance Equipment Schedule** example, define a new class named `gw.lob.gl.BOPLineScheduleCovItemCoverableAdapter`, as follows:

```
package gw.lob.bop
uses gw.lob.common.AbstractScheduledItemAdapter
uses gw.api.productmodel.Schedule
```

```
uses gw.api.domain.Clause

@Export
class BOPLineScheduleCovItemCoverableAdapter extends AbstractScheduledItemAdapter{
  var _owner : BOPLineScheduleCovItem  as readonly Owner

  construct(item : BOPLineScheduleCovItem) {
    _owner = item
  }

  override property get ScheduleParent() : Schedule {
    return _owner.Schedule
  }

  override property get PolicyLine() : PolicyLine {
    return _owner.Schedule.BOPLine
  }

  override property get AllCoverages() : Coverage[] {
    return _owner.ScheduledItemClause == null ? {} : {_owner.ScheduledItemClause as Coverage}
  }

  override function addCoverage( cov : Coverage) {
    _owner.ScheduledItemClause = cov as BOPLineSchCovItemCov
  }

  override function removeCoverage( cov : Coverage ) {
    _owner.ScheduledItemClause = null
  }

  override property get Clause() : Clause {
    var result = _owner.ScheduledItemClause
    return result
  }

  override function hasClause() : boolean {
    return (_owner.ScheduledItemClause != null)
  }

}
```

Configuring a Schedule

After defining the generic schedule implementation for the line of business, you must perform several additional steps to configure the schedule as needed.

Step 1: Define a Schedule Helper Class

The schedule helper class overrides the generic methods of the `AbstractScheduleHelper` class that it extends. The methods in this class handle out-of-sequence and preemption conditions within schedules.

In typical implementations, you can copy an existing helper class and change its references from the original line of business to the line in which you are defining schedules. The following steps provide an example of copying an existing class in this way:

1. In the **Surveillance Equipment Schedule** example, copy the existing `gw.lob.gl.GLScheduleHelper` class from **configuration/gsrc/gw/lob/gl/schedule** to **configuration/grsc/gw/lob/bop/schedule**. Rename the class file `BOPScheduleHelper.gs`.
2. In `BOPScheduleHelper.gs`:
   - Replace `GeneralLiability` with `BusinessOwners`.
   - Replace `GL` with `BOP`.
   - Replace `gl` with `bop`.

Step 2: Modify the Line-specific Policy Line Methods Class to Handle Schedules

The line-specific policy line methods class. Each line-specific definition of this class extends `PolicyLineMethodsDefaultImpl` and provides overrides for the specific functionality required by that line of business. To handle generic schedules, your line-specific class must override the following methods:

- `AllSchedules`
- `AllCurrentAndFutureScheduledItems`
- `canSafelyDeleteLocation`
- `checkLocationInUse`
- `cloneAutoNumberSequences`
- `resetAutoNumberSequences`
- `bindAutoNumberSequences`
- `renumberAutoNumberSequences`

As a starting point, you can examine the overrides from an existing line-specify policy line methods class. For example, examine the `GLPolicyLineMethods` class and implement similar overrides in the `BOPPolicyLineMethods` class.

Step 3: Define a Coverage Pattern

Next, define a coverage pattern for the schedule coverage. You define coverage patterns in Product Designer. The example configures the **Surveillance Equipment Schedule** on the businessowners line and defines the schedule as a coverage. Therefore, the coverable must be `BusinessOwnersLine`, and the coverage entity must be `BOPLineScheduleCov`.

In the **Surveillance Equipment Schedule** example, add a new coverage pattern. In the **Add Coverage** dialog box, specify the following settings:

Code	**BOPSurveillanceEquipmentScheduleCov**
Name	**Surveillance Equipment Schedule**
Category	Select a category in which to display the schedule. Example: **Building - Other**. This selection determines where in the user interface the schedule appears. It has no meaning relative to the structure of the schedule.
Covered Object	**BusinessOwnersLine** This selection must match the coverable for which the schedule is configured. The example is configured on the line.
Covered Object Type	**BOPLineScheduleCov** This selection must be the entity that implements the Schedule interface.

Step 4: Deploy Your Product Model Changes

To update PolicyCenter with the new schedule, deploy your configuration changes. See “Deploying Configuration Files” on page 31 in the *Configuration Guide* for instructions.

chapter 4

Quote Modifiers

Modifiers are factors that are applied as part of the rating algorithm. They frequently result in an increase or decrease in the premium for a policy. There are multiple types of modifiers, with most being jurisdiction specific.

Modifiers can be added to the product or policy line. Modifiers added to the product affect rating for all lines in that product. Modifiers added to a policy line affect only that line.

This topic includes:

See also

- "Modifiers Page" on page 17
- "Adding Quote Modifiers to a Policy Line" on page 40

Terms Associated with Modifiers

The following table lists some terms associated with modifiers.

Term	Description
Modifier	The most general term used to encompass the superset of premium-bearing factors including, but not limited to, the following: • Experience modifier (exp mods) • Schedule rates (credits and debits) • Individual Risk Premium Modification (IRPM) • Package modifiers • Multi-location dispersion credit Modifiers appear on the policy (unlike, for example, commissions).
Experience Modifier (exp mod)	An experience factor that indicates whether the insured has a less or more favorable loss history than peers in its industry. Various rating bureaus publish Workers' Compensation experience modifiers. In other lines, each carrier calculates experience modifiers as part of the rating process.

Term	Description
Schedule Rate	A specific type of modifier that provides credits or debits within established value ranges. Use them for factors such as management, property, and other intangibles that you cannot quantify as part of the submission.
IRPM	Individual Risk Premium Modification is a specific example of a schedule rate modifier. Used primarily in property and liability polices, the rating and quoting engine takes the following factors, among others, into account in determining the premium: • Size of premium • Spread of risk • Superior building construction • Quality of management

Configuring Modifiers in Product Designer

To configure modifiers, use the Product Designer Navigation panel to select **Modifiers** under **Products** or **Policy Lines** to open the Product or Policy Line **Modifiers** page. The page displays a list of existing modifiers.

- To add a new modifier, click **Add** to open the **Add Modifier** dialog box.
- To edit an existing modifier, click the modifier **Name** link to open the Product or Policy Line Modifier home page.

Rate Modifiers

When adding a modifier, if you set the modifier **Data Type** to **Rate**, the **Add Modifier** dialog box displays a **Schedule Rate** check box. Select this check box to create a schedule rate modifier.

If you did not select **Schedule Rate**, the **Modifier** home page displays a **Rate is Relative** field:

- **0** – Modify rate as an offset relative to 0. For example, 0.10 represents a 10% increase in the rate, -0.15 represents a 15% reduction. Sets `renderRateAsMultiplier` to `false` on the modifier pattern object, (`ModifierPattern`).
- **1** – Default. Modify rate as a multiplier. For example, 1.1 represents a 10% increase on the rate, 0.85 represents a 15% reduction. Sets `renderRateAsMultiplier` to `true` on the modifier pattern object (`ModifierPattern`). If you do not specify the `renderRateAsMultiplier` attribute in the XML definition of `ModifierPattern`, `1` is the default value.

For rate modifiers, the Product or Policy Line Modifier home page displays an additional **State Min/Max** link under **Go to**. For each state, you can define minimum and maximum rate modifiers.

Split Rating Period

In some lines of business, such as Workers' Compensation, rating can be split into multiple rating periods. PolicyCenter creates multiple rating periods around an anniversary rating date or split dates specified by the user. Anniversary rating dates and split dates are specified by jurisdiction.

For anniversary rating dates and user-defined split dates, the **Split Rating Period** modifier check box determines whether PolicyCenter applies the same or separate modifier values for each rating period. If the **Split Rating Period** check box is selected, then PolicyCenter establishes the rating periods and splits the exposures for each. It then calculates the policy premium separately for each rating period.

If a split rating period applies, regulatory authorities for the jurisdiction notify the carrier. PolicyCenter stores this information as part of the policy information for that jurisdiction.

Using the **ExpMod** modifier as an example, assume that **Split Rating Period** is selected. For this modifier, the user can specify separate experience modifier values for each rating period. The following example illustrates this use of experience modifiers.

Sample Policy	
Policy Term	1/1/2013 to 1/1/2014
Anniversary rating date	7/1/2012 - translated as an anniversary on July 1, 2013
Modifier	ExpMod
Expmod modifiers provided by Jurisdiction	1. Effective 7/1/2012 = 1.10 2. Effective 7/1/2013 = 1.05
Modifier pattern	**Split Rating Period** selected
Policy Rating	
1/1/2012 to 7/1/2013	ExpMod 1.10 applies
7/1/2013 to 1/1/2014	ExpMod 1.05 applies

To add or remove the optional Split Rating Period check box in modifier patterns for a line of business

1. In Studio, open the properties file for the policy line pattern by navigating to **configuration** → **config** and opening:
   `resources/productmodel/policylinepatterns/XXLine/XXLine.properties`
   The properties file is named *code*`.properties` where *code* is the code of the policy line. For example, the workers' compensation properties file is `WorkersCompLine.properties`.
2. Change the `line.usesSplitRatingPeriod` property to `true` to add the check box, or `false` to remove the check box.

See also

- "Step 6: Add Optional Features to a Policy Line" on page 158 for more information about adding the optional **Split Rating Period** check box to a new policy line pattern.
- "Multiple Rating Periods" on page 282 in the *Application Guide*
- "Rating Periods" on page 289 in the *Application Guide*

Display Section

Expand the **Display** section of the Product or Policy Line **Modifiers** page to configure various display parameters for the modifier, including:

Display Justification	Specifies whether to display the **Justification** field in PolicyCenter.
Display Range	Specifies whether to show the range of values for this modifier in PolicyCenter. Appears only if the modifier data type is **rate**.
Default Minimum	Specifies the minimum value that can be entered absent any jurisdiction-specific minimum value defined in the **State Min/Max** page. Appears only if the modifier date type is **rate**.
Default Maximum	Specifies the maximum value that can be entered absent any jurisdiction-specific maximum value defined in the **State Min/Max** page. Appears only if the modifier date type is **rate**.
Display Value Final	Displays an option in PolicyCenter that lets you specify whether the entered modifier value is an estimate or final value at time of issuance. Appears only if the modifier data type is **rate**.
Manually Enabled	Modifier is not automatically included on the policy line. Appears only if modifier data type is **rate** and **Schedule Rate** is false.

Note: The default minimum and maximum values set the overall possible range for this modifier if no jurisdiction-specific minimum and maximum values are defined. If the **Schedule Rate** field is set to **Yes** and rate factors are entered, each rate factor must have a value within its own minimum and maximum. In addition, the sum of all rate factors must be within the overall modifier minimum and maximum values.

Rate Factors Page

During modifier creation, Product Designer adds a **Rate Factors** link under **Go to** provided you do both of the following:

- Set the data type to **rate**.
- Select the **Schedule Rate** check box.

You use this page to add one or more rate factors that combine to give the value of the schedule credit.

You set the **Default Minimum** and **Default Maximum** values for this individual rate factor in **Display** section of this page. These minimum and maximum values constrain the amount of the credit you can enter in PolicyCenter for that rate factor, absent any state-specific minimum and maximum values.

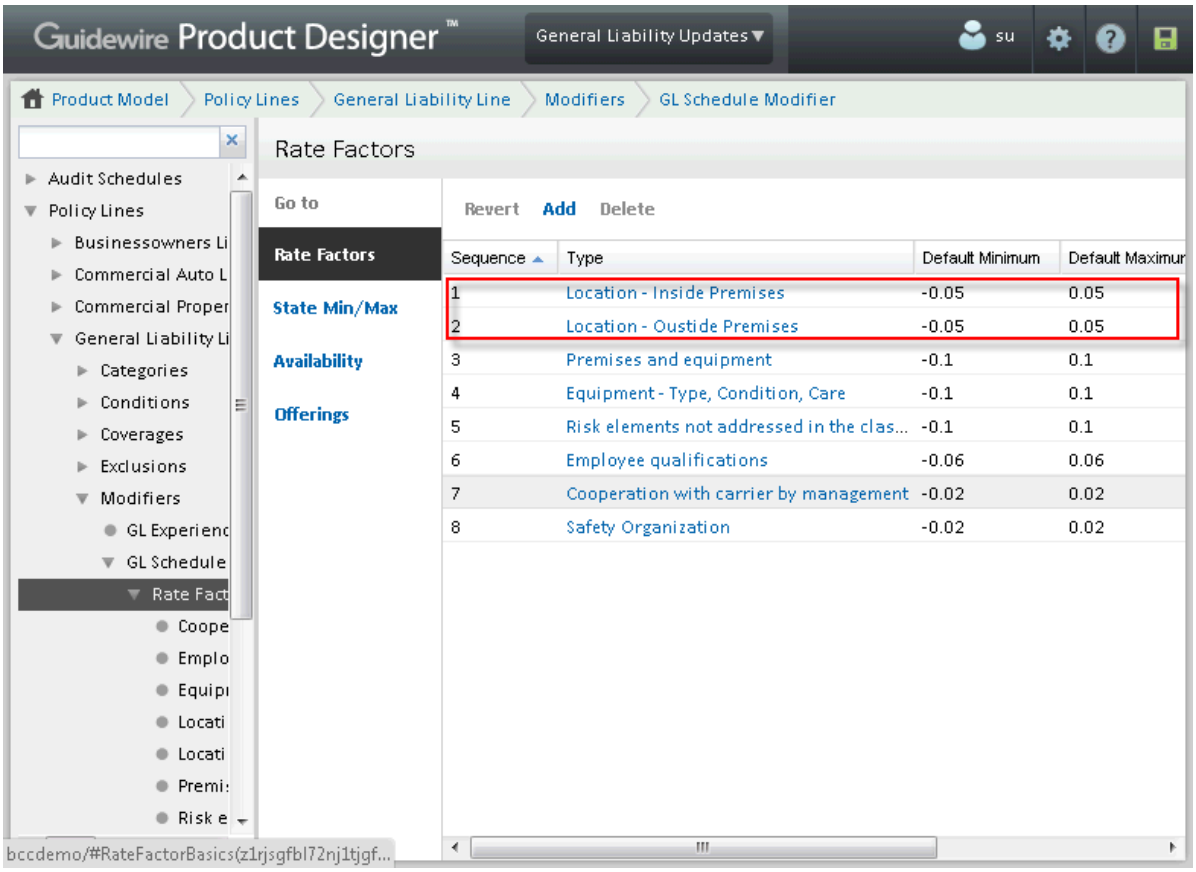
On the modifier rate factors **State Min/Max** page, use the links under **Go to** to configure the following behaviors for individual rate factors:

- **State Min/Max** to configure jurisdiction-specific minimum and maximum rate factor values.
- **Availability** to configure availability for individual rate factors.
- **Offerings** to add or remove individual rate factors from offerings.

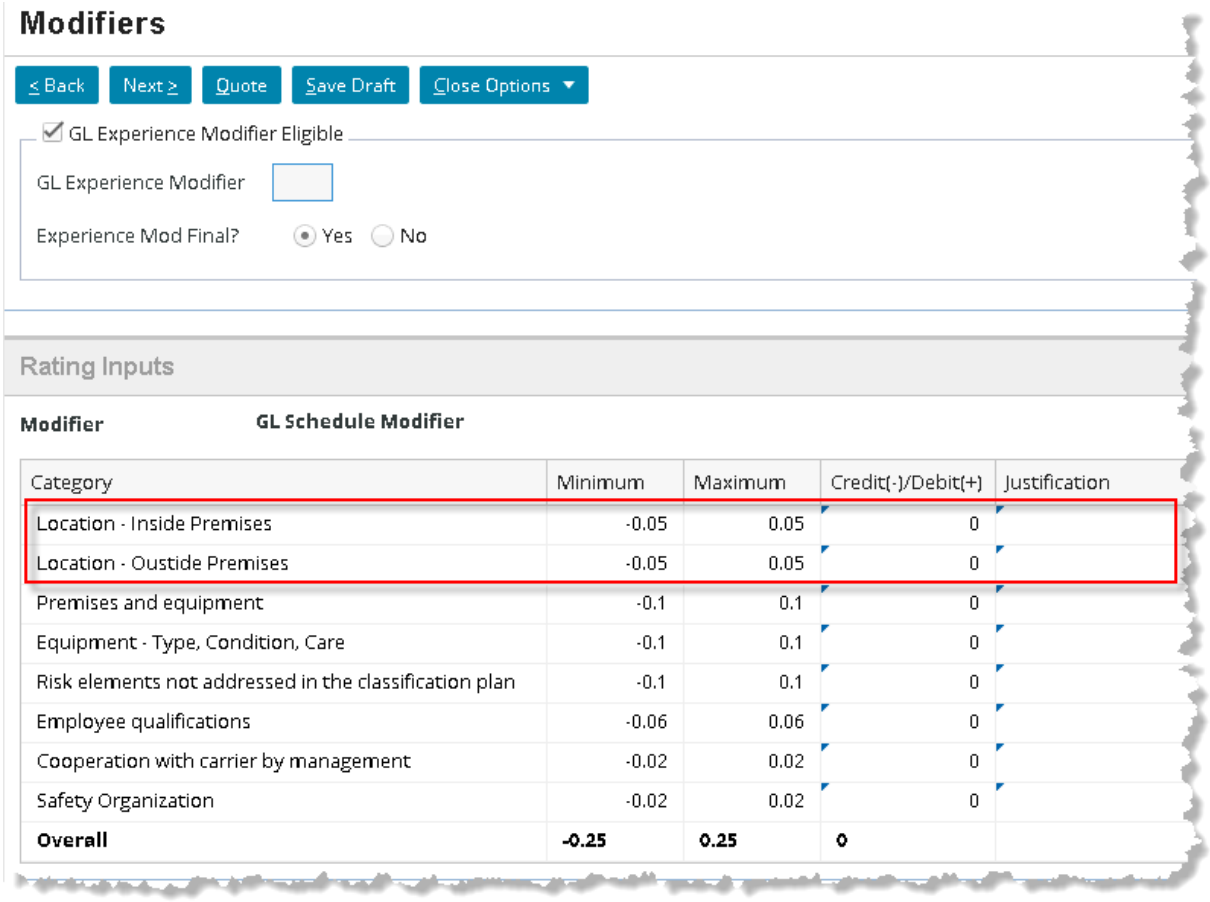
Note: The values available in the **Rate Factors** page are restricted to the overall range of values specified in the **Display** section of the Product or Policy Line home page. The sum of all rate factor values must be within the overall minimum and maximum for the schedule rate modifier. The values are also subject to the overall jurisdiction-specific minimum and maximum values for the modifier.

Example of Schedule Credits

In Product Designer, the General Liability line defines a number of schedule rate modifiers on the **Rate Factors** page.



You access these values in PolicyCenter through the General Liability **Modifiers** screen.



Category	Minimum	Maximum	Credit(-)/Debit(+)	Justification
Location - Inside Premises	-0.05	0.05	0	
Location - Oustide Premises	-0.05	0.05	0	
Premises and equipment	-0.1	0.1	0	
Equipment - Type, Condition, Care	-0.1	0.1	0	
Risk elements not addressed in the classification plan	-0.1	0.1	0	
Employee qualifications	-0.06	0.06	0	
Cooperation with carrier by management	-0.02	0.02	0	
Safety Organization	-0.02	0.02	0	
Overall	**-0.25**	**0.25**	**0**	

Notice that in the PolicyCenter screen, the bold **Overall** row displays the minimum and maximum allowable values for the individual columns. These values in bold are configured in Product Designer in the **Display** section of the Product or Policy Line Modifier home page. The values correspond to the **Default Minimum** and **Default Maximum** fields if no jurisdiction-specific minimum and maximum have been specified.

State Min/Max Page

You use the Product or Policy Line modifier **State Min/Max** page to set minimum and maximum modifier values that apply to a specific jurisdiction for the modifier pattern. As elsewhere, values typically range between -0.05 to +0.05. See “Display Section” on page 54 for more information.

Availability Page

You use the Product or Policy Line modifier **Availability** page to set availability for the modifier pattern. Availability functionality for modifier patterns is similar to that of the other patterns, and is explained in “Defining Availability” on page 85.

Offerings Page

You use the Product or Policy Line modifier **Offerings** page to set which offerings have this modifier enabled. You can also specify whether newly created offerings have this modifier enabled by default. See “Offerings Page” on page 17 and “Configuring Offerings” on page 99 for more information.

chapter 5

Question Sets

A question set is a collection of questions presented within PolicyCenter that gathers information about an applicant. You use the answers to these questions to evaluate the risk associated with a policy applicant. PolicyCenter has a number of predefined question set types. You can also add your own question set types.

This topic includes:

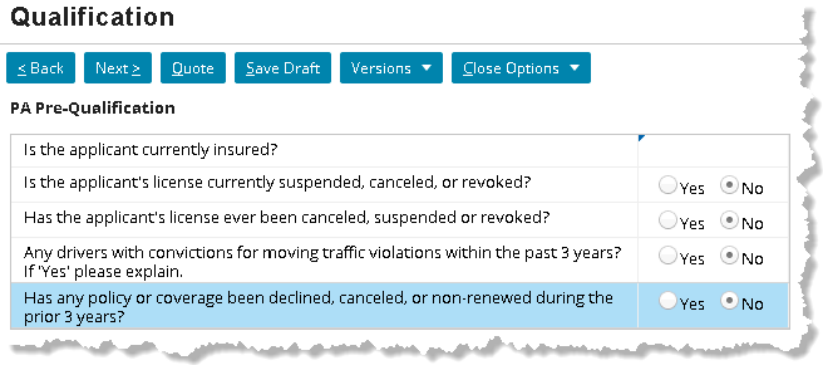
Question Set Basics

You use question sets to assess applicants in various contexts. For example, you can use question sets to:

- Collect information about an insured. This type of information is not directly used to calculate premiums.
- Collect information to determine whether an applicant qualifies for a particular product.
- Collect information to help an underwriter evaluate risk.

Multiple questions are typically used to assess risk. Therefore, the answer to a single question often does not determine whether or not the applicant qualifies for a product. Instead, the answers to a series of questions typically determine eligibility. Multiple incorrect answers can be used to flag an applicant who is too high a risk for the current product. Because of a greater risk, other options must be selected. For example, you could qualify the applicant by selecting a different underwriting company, a different modifier value, or you could reject the applicant altogether.

The **Qualification** screen for Personal Auto displays questions that you can ask to qualify or reject a policy applicant.



PolicyCenter:

- Automatically renders a question set in a list view with an optional default answer specified in Product Designer.
- Stores logic to hide or display a question with the question itself.
- References a question set as a single element in its PCF page.

When to Use Question Sets

Questions are best suited to capture information used by underwriting or to qualify risk. Do not use question sets in place of standard data model fields to capture information needed to rate a policy or to pass to other integrated components. The way that PolicyCenter stores and retrieves answers makes question sets unsuitable for rating and integrations. Likewise, question sets are not well suited for capturing information directly related to a particular object, such as a building or a vehicle.

The advantage of question sets is that you can quickly and easily configure them in Product Designer, often with no need to perform user interface configuration in Studio. Questions also provide the product model availability mechanism to control when a question is visible.

The disadvantages of question set concerns their storage and retrieval. To enable question sets to be quickly configured through the product model, their answers are stored in generic database tables and columns. These tables can become large enough to cause performance problems, particularly when many questions are combined with complicated availability logic. Additionally, answer values must be retrieved by question code value, rather than by using a specific type-safe symbol as with other product model patterns. An answer value then must be cast to the correct type, introducing the possibility for errors in any programmatic statements that use the answer.

PolicyCenter provides the following two alternatives to question sets to consider, depending on your data collection needs:

- For data used by a rating engine, modifiers provide many of the same advantages as question sets. However, modifiers are specifically designed for fields that are used in rating algorithms.
- For fields associated with specific objects, such as buildings or vehicles, use standard datamodel fields tather than question sets. Datamodel fields are type-safe and properly normalized. You can design the label for a datamodel field to look the same as a question, so the PolicyCenter user is not aware of a difference.

Associating Question Sets with Multiple Products

Individual questions are not reusable in multiple question sets. To use the same question in two question sets, you must define the question in each question set. However, you can associate the same question set with multiple products. For example, to ask the same questions in multiple products, put the questions into a single question set and then associate that question set with each product.

Questions and Answers

After you initiate a policy transaction, PolicyCenter creates a list of questions based on the product model definition. It then renders these questions in a question set. PolicyCenter stores the answer to each question permanently in the PolicyCenter database. PolicyCenter stores the answer to each question for a particular policy transaction as a separate row in the database. The question set answer container determines the database table used to store the answers.

Each answer points back to its question through the `QuestionCode`, which in Product Designer is the `Code` value of the question. You can think of answers as instances of a question pattern.

You can access and manipulate the answers to a question set through delegates. For more information, see "Question Set Object Model" on page 69.

Incorrect Answers

When defining a question, you can define a correct answer. You then can configure the question set such that if the applicant answers one or more questions incorrectly, PolicyCenter can take various actions. A sufficient number of incorrect answers can result in:

- **An underwriting issue** – PolicyCenter can raise an underwriting issue to block quote or bind. For example, you cannot quote the policy until the underwriting issue is approved by an underwriter with sufficient authority.
- **A number of risk points** – Risk points can be used in many ways. In the base configuration, if the sum of risk points from the pre-qualification question set is 200 or more, PolicyCenter raises an underwriting issue that blocks quote. The medium risk point threshold of 200 points is set in Studio by segmentation. For more information, see "Selecting the Underwriting Company through Segmentation" on page 595 in the *Configuration Guide*.
- **A blocking action** – You can specify whether the incorrect answer blocks you from advancing or just displays a warning before advancing to the next page.

You can mix and match these incorrect answer consequences. For example, you can specify that an incorrect answer to a question does all of the following:

- Displays a warning.
- Raises an underwriting issue.
- Contributes a specified number of risk points needed to raise a different type of underwriting issue.

For example, in the base configuration, the applicant is asked:

"Has any policy or coverage been declined, canceled, or non-renewed during the prior 3 years?"

If the answer is **`Yes`** (the incorrect answer), the question creates an underwriting issue of type `Question blocking bind`. The incorrect answer also adds 50 risk points. If another 150 risk points have been added by incorrect answers to other questions, then the threshold of 200 causes PolicyCenter to create an underwriting issue that blocks bind. This underwriting issue prevents you from binding the policy.

In another example using a Personal Auto policy submission, the applicant admits to having a suspended license in the pre-qualification questions. PolicyCenter raises an underwriting issue when you try to quote the policy, because the correct answer for the suspended license question is **`No`**. The incorrect answer blocks quoting. PolicyCenter displays a message when you click **`Quote`**. The policy cannot be quoted until the issue is approved by a user with sufficient authority to approve issues of this type. At this point, you have the following options:

- Save the submission as a draft.
- Close the submission as withdrawn, declined, or not taken.
- Change the answer to the suspended license question.

You can also configure a question to block the progress or display a warning message in response to an incorrect answer. In some cases, a carrier wants the message to be explicit so that users understand the eligibility requirements for the product. This type of warning message helps to prevent users from continually initiating applications that do not meet the criteria. In other cases, a carrier wants the warning message to be less explicit so that users do not understand the eligibility requirements. This type of warning message helps to prevent users from learning how to pre-qualify undesirable candidates by knowing all the correct answers.

Types of Question Sets

Question sets can facilitate collecting information at different points during policy processing. The following question set types are defined in the base configuration:

- **PreQual** – Questions that determine whether or not the insured meets the basic requirements for coverage by your carrier. You can configure the actions to take when the questions are answered incorrectly, such as blocking the user or raising underwriting issues. Therefore, these questions typically appear early in the submission process. In the base configuration, PolicyCenter displays pre-qualification questions on the **Qualification** screen. The base configuration uses pre-qualification question sets exclusively in submission transactions.
- **Product Qualification** – Questions to assess whether or not an applicant is qualified for a particular product.
- **Location** – Questions that link to a particular location and used for underwriting. PolicyCenter is configured to display an additional tab when location questions are present. You can further configure PolicyCenter to display these questions when the location fits a particular profile or has certain associated coverages. For example, a question about whether the location serves alcohol could be used in businessowners products where the location has an industry code associated with restaurants. The answer then could be associated with liquor liability coverage. For another example, a question about whether the location has a swimming pool could be use if the location has an industry code associated with apartments or motels.
- **Offering Selection** – Questions associated with products that have offerings. Such products have an **Offerings** screen that appears at the beginning of a job wizard. The offerings available in the **Offerings Selection** list can be determined by answers to these questions.
- **Supplemental** – Questions that collect additional underwriting information needed to assess risk. For example, in a workers' compensation policy, a question asking if any employees are under age 16 or over age 60 could add a certain number of risk points. Another question about whether any employees travel out of state on business could add a smaller number of risk points.
- **Underwriting** – Questions that generally help to determine the quality of a risk. A higher-quality risk (one that is less likely to have a loss) could qualify for automated approval and a favorable rating. A lower-quality risk could be charged more for the policy, or possibly not qualify for the policy at all. Underwriting questions are typically evaluated as a set.

In addition to the question set types provided in the base configuration, you can define your own question set types. For example, a carrier wants to have question sets that assess an applicant's suitability for upselling. You can define a new type for such question sets so that you can track them and categorize their answers accordingly.

Question Set Type and Answer Container

Question sets can be included in the job wizards based upon their type, answer container, and the product with which they are associated. The question set type is often, but not required to be, related to the entity where the answers are stored. In the base configuration, the `PolicyLine`, `PolicyPeriod`, and `PolicyLocation` entities are answer containers. In the base configuration, the questions on the **Qualification** screen have a `QuestionSetType` set of `Pre-Qual` and an answer container type of `PolicyPeriod`. You also can configure question sets for other entities as needed.

See also

- "The Answer Container Delegate" on page 69
- "Defining Answer Containers and Question Sets for Other Entities" on page 70

Configuring Question Sets in Product Designer

This topic describes how to configure question sets in Product Designer.

- Adding a New Question Set
- Configuring Offerings for Question Sets
- Defining New Questions
- Configuring Question Behavior
- Configuring Question Dependencies
- Configuring Incorrect Answer Behavior
- Configuring Question Choices
- Configuring Question Help Text

Adding a New Question Set

PolicyCenter does not display individual questions, only question sets. Therefore, you must create questions within a question set. Consequently, the first step in creating questions is to add a question set. After you add the question set, you must associate the question set with the products that will use it.

To add a new question set

1. In the Product Designer navigation panel, select the **Question Sets** node.
2. At the top of the **Question Sets** page, click **Add** to open the **Add Question Set** dialog box.
3. Enter the code, name, question set type, and the answer container type for the new question set, then click **OK**. PolicyCenter adds the new question set to the navigation panel.

At this point, the question set exists, but contains no questions and does not link to a product.

To associate a question set with a product

1. In the Product Designer navigation panel, expand **Products** and select the product to which to add the question set.
2. In the product's home page, under **Go to**, click **Question Sets**.
3. At the top of the **Question Sets** page, click **Add** and select the question set from the list of choices. The list shows only question sets that have not already been added to the product.

Question Set Pages

After you create a question set, you can select it in the navigation panel and then use the links under **Go to** to visit any of the following related pages:

- **Questions** – Define the individual questions within the question set, as explained in "Defining New Questions" on page 62.
- **Availability** – Define the conditions that control whether the entire question set is available. Availability functionality for question sets is similar to that of the other patterns, and is explained in "Defining Availability" on page 85.
- **Offerings** – Add or remove the entire question set from offerings. See "Offerings Page" on page 17 and "Configuring Offerings" on page 99 for more information.

Configuring Offerings for Question Sets

Use the question set **Offerings** page to specify which offerings include the question set, and whether the question set is automatically included in new offerings.

By default, each question set added to the product model is included in all offerings and appears in the **Included or implied in these Offerings** column in the **Offerings** page. To disable a question set in an offering, select the offering and click the right arrow to move the offering to the **Disabled in these Offerings** column.

> **Note:** If the **Question Set Type** is **Offering Selection**, the question set is always enabled in all offerings. Offering selection question sets cannot be moved to the **Disabled in these Offerings** column.

See also

- "Offerings Page for Coverages" on page 28 for instructions on how to use this tab
- "Offerings and Question Sets" on page 102

Defining New Questions

After you create a question set, add questions to it.

To add a question to a question set

1. In the Product Designer navigation panel, expand the **Question Sets** node and then select the question set to which to add questions.
2. At the top of the **Questions** page, click the icon that represents the type of question set to add.

Icon	Question type	Format
	String	• String Field • String Text Box
	Integer	• Integer Field
	Boolean	• Boolean Checkbox • Boolean Radio • Boolean Select
	Choice	• Choice Radio • Choice Select
	Date	• Date Field

3. Clicking a question set type icon opens the **Add Question** dialog box. Specify the following properties:

Property	Value
Code	`QuestionCode` that answers use to point back to their questions.
Label	Text of the question as it appears in PolicyCenter.
Format	Listed in the preceding table. Determines how the question is presented in PolicyCenter.

4. Click **OK** to add the new question to the list of questions for this question set.
5. Arrange the order of questions in the **Questions** page by dragging them up or down. The order of questions in the **Questions** page determines the order of questions in PolicyCenter.

Question Pages

After you add a question, you can select it in the navigation panel and then use the links under **Go to** to visit any of the following related pages:

- **Dependent On** – Create dependencies that determine whether specific questions are visible in PolicyCenter, as explained in "Configuring Question Dependencies" on page 64.
- **Incorrect Answer** – Specify a correct answer, failure message, blocking action, underwriting issue type, and risk points to add, as explained in "Configuring Incorrect Answer Behavior" on page 65.
- **Availability** – Define the conditions that control whether an individual question is available. Availability functionality for questions is similar to that of the other patterns, and is explained in "Defining Availability" on page 85.
- **Choices** – Appears for Choice type questions only. Define the choices available for answering the question, as explained in "Configuring Question Choices" on page 67.

Configuring Question Behavior

You can configure the behavior of each question in Product Designer either in:

- The **Questions** page, where you can see it in context with other questions.
- An individual question's home page, where you see only a single question at a time.

To configure a question's behavior on the Questions page

1. In the navigation panel, expand the **Question Sets** node and then expand the node for the question set you want to configure. Click the **Questions** node to open the **Questions** page.
2. Click a question to select it. The background of the question is highlighted to indicate that it is selected. In addition, the Question Editor appears to the right of the question.
3. With the question selected and the Question Editor visible, you can do any of the following:
   - Drag and drop the question to change the order in which it appears.
   - Specify a default answer by entering or selecting the answer, or remove a default answer by clicking Reset to Default  .
   - Delete a question by clicking Delete  .
   - View the question's **Code**.
   - Edit the question **Label** and click Translate  to enter the label in other languages.
   - Specify a question **Indent**. Use this feature to indicate, for example, that this question is dependent on the non-indented one above it.
   - Indicate that the question is **Required**. Required questions must be answered before you can move to another screen in PolicyCenter.
   - Configure **Post** behavior as **Always** or **Automatic**, enabling you to determine whether answering this question updates the other questions or fields in the PolicyCenter screen.
   - Enter one or more lines of **Help Text** to display below the question in PolicyCenter.
   - Jump to other pages to configure **Dependent On**, **Incorrect Answer**, and **Availability** for this individual question.

To configure a question's behavior on the question's home page

1. In the navigation panel, expand the **Question Sets** → *Question Set Name* → **Questions**, and then select the individual question you want to configure to open that question's home page. The title of the question home page is the question itself.
2. In the individual question's home page, you can do any of the following:

- Specify a default answer by entering or selecting the answer, or remove a default answer by clicking Reset to Default  .
- Delete a question by clicking **Delete** in the upper right corner of the page.
- View the question's **Code**.
- Edit the question **Label** and click Translate  to enter the label in other languages.
- Specify a question **Indent**. Use this feature to indicate, for example, that this question is dependent on the non-indented one above it.
- Indicate that the question is **Required**. Required questions must be answered before you can move to another screen in PolicyCenter.
- Configure **Post** behavior as **Always** or **Automatic**, enabling you to determine whether answering this question updates the other questions or fields in the PolicyCenter screen.
- Enter one or more lines of **Help Text** to display below the question in PolicyCenter.

3. Use the Go to links to jump to other pages where you can configure **Dependent On**, **Incorrect Answer**, and **Availability** for this individual question.

Configuring Question Dependencies

In Product Designer, configure question dependencies by opening the **Dependent On** page in one of the following ways:

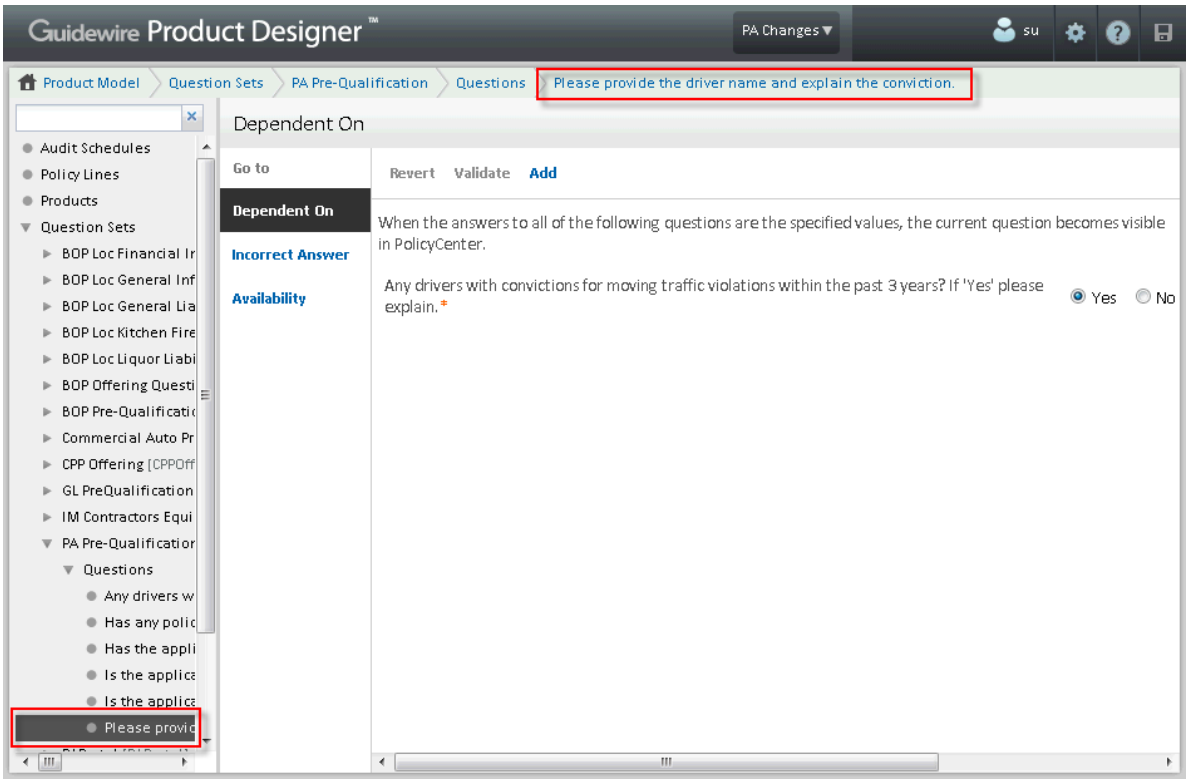
- After selecting a question in the **Questions** page, click the **Dependent On** link in the Question Editor.
- After selecting an individual question in the navigation panel, click the **Dependent On** link under **Go to** on the question's home page.

You can define any number of questions on which a selected question is dependent. If the applicant answers all of the dependent questions with the values specified in the **Dependent On** page, PolicyCenter displays the question you are currently configuring. The Dependent On feature enables you to display follow-up questions that appear only when specific answers have been specified. The dependent on and dependent question can be of any question type.

To configure a question dependency

1. Click **Add** at the top of the **Dependent On** page to open the **Add Question to Depend On** dialog box.
2. From the **Question** list, select the question whose answer is to control the visibility of the question you are configuring, then click **OK**. The question is added to the **Dependent On** page.
3. Enter the answer to this question that is to cause the question you are configuring to appear.
4. Repeat steps 1–3 to add more dependent questions. The question you are configuring appears only if the answers the user enters for all questions match exactly the answers you supply on the **Dependent On** page.

In the following illustration, the dependent question is the one selected in the navigation panel, **Please provide the driver name and explain the conviction.** This name also appears in the breadcrumbs at the top of the page.



The question and answer that determine whether the dependent question appears is added to the Dependent On page. In this example, it is a Boolean type, and its answer is set to **Yes**. Therefore, in a Personal Auto pre-qualification, if the dependent-on question is answered **Yes**, the dependent question appears prompting for additional details about the conviction.

Configuring Incorrect Answer Behavior

In Product Designer, configure how to handle incorrect answers by opening the **Incorrect Answer** page in one of the following ways:

- After selecting a question in the **Questions** page, click the **Incorrect Answer** link in the Question Editor.
- After selecting an individual question in the navigation panel, click the **Incorrect Answer** link under **Go to** on the question's home page.

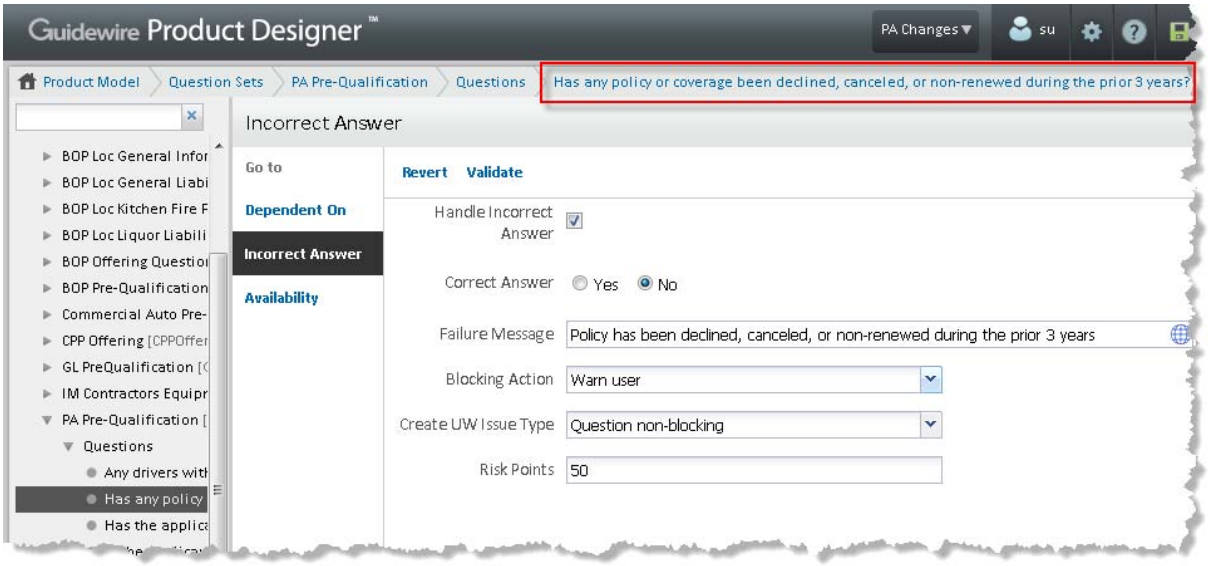
To configure incorrect answer behavior

1. Select the **Handle Incorrect Answer** check box to enable the other controls on the **Incorrect Answer** page.
2. Specify the **Correct Answer**. Guidewire recommends that you configure incorrect answers only for questions that have a restricted set of answers, such as Boolean, Choice, or Date. A special type of underwriting issue, **Question with number**, can handle incorrect answers for Integer types by creating appropriate underwriting issues. This type of underwriting issue is described in the following table. Attempting to exactly match a String question type typically is unreliable.
3. Configure any combination of the following options:
   - **Failure message** – Enter a message that appears in PolicyCenter when the answer does not match the **Correct Answer**. If needed, click Translate to enter the message in other languages.

- **Blocking Action** – Select whether to block or warn the user, or leave blank to do nothing when the answer does not match the **Correct Answer.**
- **Risk Points** – Specify the number of risk points to add when the answer does not match the **Correct Answer.**
- **Create UW Issue Type** – Select the type of underwriting issue to raise, or leave blank to do nothing when the answer does not match the **Correct Answer.** The following table explains the available options.

Underwriting issue type	Description
Question blocking bind	An incorrect answer allows quoting the policy but generates an underwriting issue that does not allow binding it unless an underwriter approves the issue.
Question blocking quote	An incorrect answer generates an underwriting issue and blocks quoting the policy unless an underwriter approves the issue.
Question non-blocking	An incorrect answer generates an underwriting issue but does not block quoting or binding.
Question with number blocking bind	An answer to an Integer type question with a number greater than the specified **Correct Answer** generates an underwriting issue that blocks binding unless an underwriter approves the issue.
Question with number blocking quote	An answer to an Integer type question with a number greater than the specified **Correct Answer** generates an underwriting issue that blocks quoting unless an underwriter approves the issue.
Question with number non-blocking	An answer to an Integer type question with a number greater than the specified **Correct Answer** generates an underwriting issue bud does not block quoting or binding.

In the following illustration, the selected question, **Has any policy or coverage been declined, canceled, or non-renewed during the prior 3 years?**, has all possible **Incorrect Answer** parameters enabled.



In this example, the **Correct Answer** to the Boolean question is **No**. Any other answer triggers the incorrect answer behavior, which includes:

- Displaying the **Failure Message**.
- Warning the user, but allowing them to proceed to the next wizard step.
- Creating a non-blocking underwriting issue, which notifies underwriting, but does not block quoting or binding the policy.
- Adding 50 risk points. In the base configuration, if this brings the total risk points to a value of 200 or greater, PolicyCenter raises an underwriting issue and blocks quote, regardless other settings.

Configuring Question Choices

In Product Designer, when you add a Choice type question, you must configure the available answer choices by opening the **Choices** page in one of the following ways:

- After selecting a question in the **Questions** page, click the **Choices** link in the Question Editor.
- After selecting an individual question in the navigation panel, click the **Choices** link under **Go to** on the question's home page.

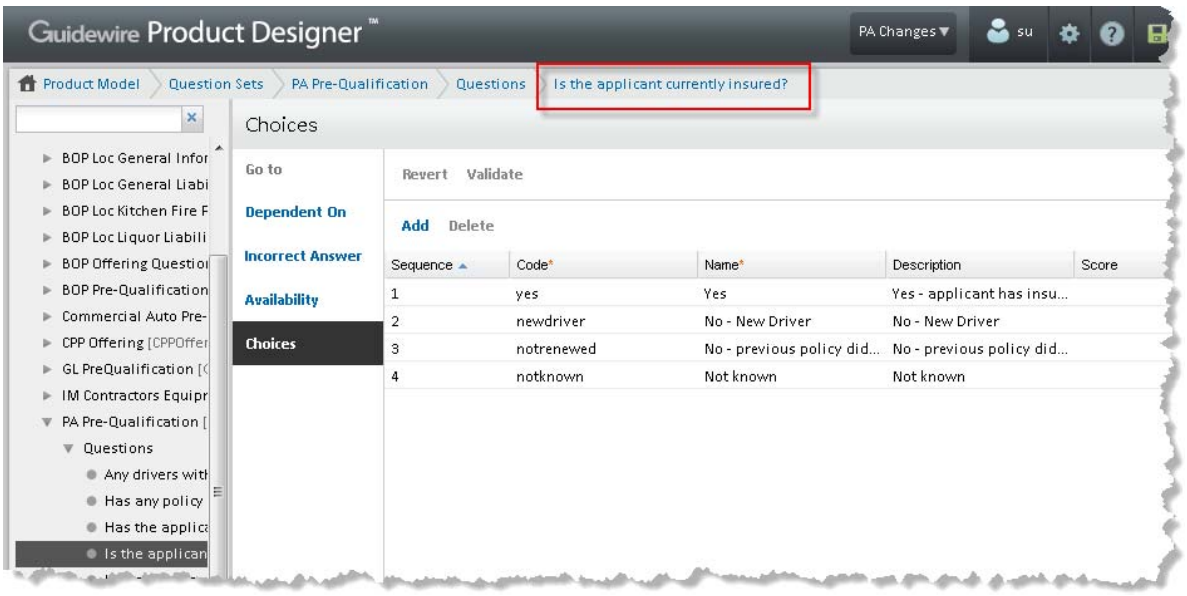
To configure question choices

1. Click **Add** at the top of the **Choices** page to add a new row to the list of choices.
2. Supply the following information:

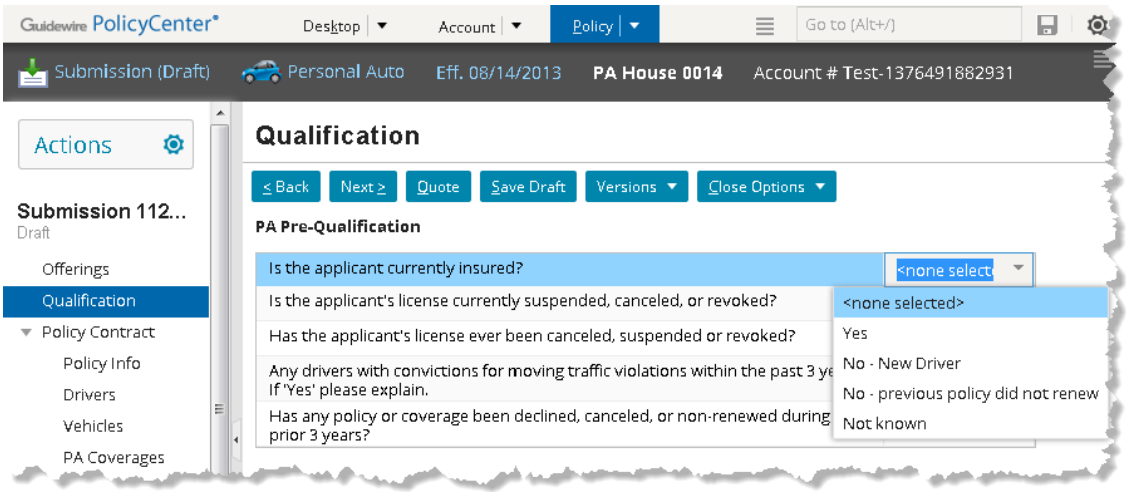
Property	Value
Code	Code that is used internally to represent the answer choice.
Name	Text of the answer choice that appears in PolicyCenter.
Description	Optional description that only appears in Product Designer.
Score	Not used in the default PolicyCenter configuration.

3. Repeat steps 1 and 2 to add more answer choices.
4. Arrange the choices as you want them to appear in PolicyCenter. To arrange the choices, you can:
   - Drag and drop each choice to a new row, but only if the list of choices is sorted by **Sequence**. Click the **Sequence** column heading to sort by that column.
   - Change the **Sequence** number to indicate a new order. Product Designer automatically renumbers other rows to accommodate the number you specify.

The following illustration shows an example of the choices configured for the PA Pre-Qualification question, **Is the applicant currently insured?**



In this example, you can see that the question has four possible answer choices, arranged in the order in which they appear in PolicyCenter. PolicyCenter renders the answers to the `PACurrentlyInsured` question as follows:



A Choice type question can appear either as a drop-down list or as a set of option buttons by setting the **Format** to **Choice Select** or **Choice Radio**. However, after you add the question to the question set, you cannot change its format.

Configuring Question Help Text

In Product Designer, you can add supplemental help text that appears below a question to help the user understand how to answer the question.

- After selecting a question in the **Questions** page, click Add a line of help text in the Question Editor.
- After selecting an individual question in the navigation panel, click Add a line of help text in the Question Editor.

To configure question help text

1. After selecting a question in the **Questions** page or selecting an individual question in the navigation panel, click Add a line of help text in the Question Editor.
2. Type the question text. Regardless of the number of lines of text you enter in the text box, the text you enter appears on a single line in PolicyCenter.
3. If needed, click Translate to enter the help text in other languages.
4. Repeat steps 1–3 to add more lines of help text, as needed.
5. Drag and drop lines of help text in the Question Editor to change their order.

For example, suppose that you have a pre-qualification question on a Personal Auto policy submission that asks if the applicant meets certain conditions.

Boolean Question: **Does the applicant meet the criteria for a Good Driver?**

Help text line 1: `The applicant must have a clean violation record for the last three years.`

Help text line 2: `The applicant must have a clean accident record for the last five years.`

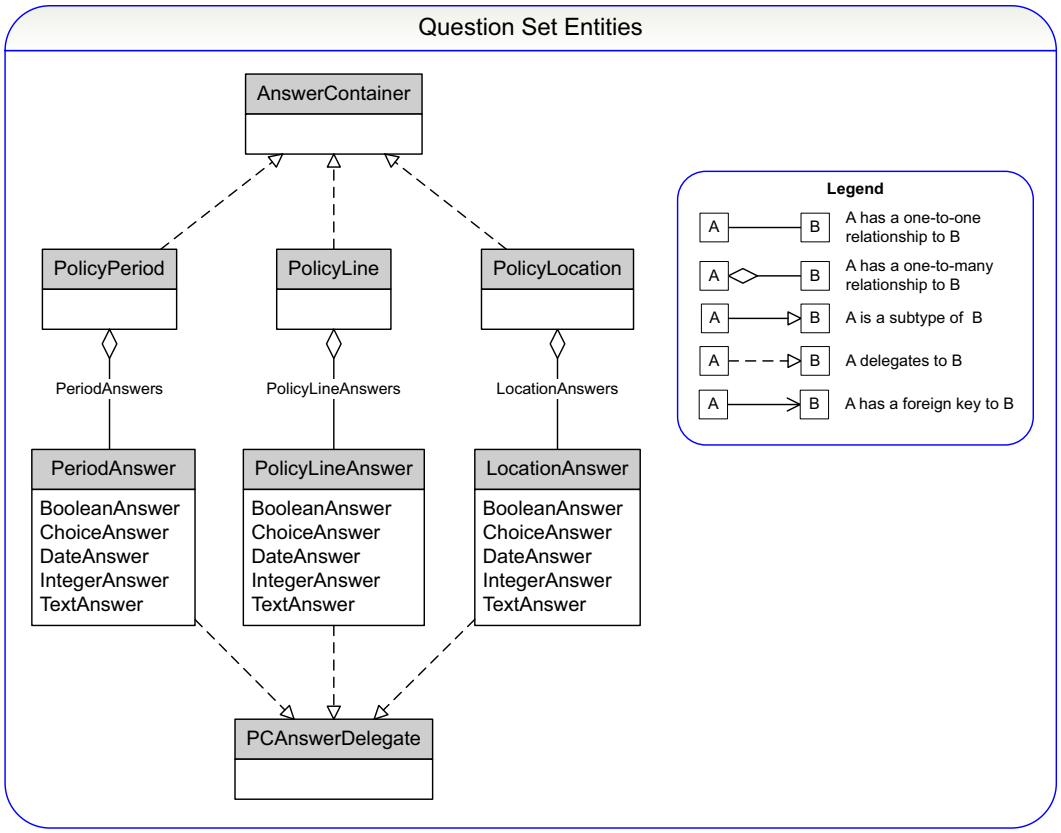
Help text line3: `The applicant cannot drive a red car.`

Question Set Object Model

In the base configuration, the `PolicyLine`, `PolicyPeriod`, and `PolicyLocation` entities delegate to `AnswerContainer`. Consequently, each of these entities has an array to access the answers.

The `PolicyLineAnswer`, `PeriodAnswer`, and `LocationAnswer` entities delegate to `PCAnswerDelegate`.

The following illustration shows some of the key entities associated with question sets.



The Answer Container Delegate

The `AnswerContainer` delegate has a number of methods and properties. Some of the methods are:

- `createAnswer` – Creates an answer to the specified question.
- `getAnswer` – Gets the answer to the specified question.
- `removeAnswer` – Removes the answer to the specified question.
- `syncQuestions` – Synchronizes answers that match the type, the `AnswerContainer` delegate, and the product. You can call this method before displaying question sets in a PCF file. There are several examples of using this method in the job wizards. For example, `IssuanceWizard` makes an indirect call to `syncQuestions` in the `onEnter` property in the `Offering` step.

The **Answer Container Type** of the question set specifies the entity that delegates to `AnswerContainer`. In Product Designer, for example, use the navigation pane to go to **Question Sets** → **PA Pre-Qualification**. In the question set's home page, you can see that the **Question Set Type** is **PreQual** and that the `PolicyPeriod` entity delegates to `AnswerContainer`.

Defining Answer Containers and Question Sets for Other Entities

The default PolicyCenter configuration provides three answer container types: `PolicyLocation`, `PolicyPeriod`, and `PolicyLine`. These three types enable you to define question sets pertaining to policy locations, policy periods and the policy line itself. To define questions sets pertaining to a different entity, you must configure a new answer container type.

Note: The configuration information in the following topics are general guidelines that must be adapted to your specific requirements.

This topic describes how to define question sets for entities other than `PolicyLocation`, `PolicyPeriod`, and `PolicyLine`.

Step 1: Create an Answer Container for the Entity

The first step in defining a question set in another entity is to create a new answer container entity of the corresponding type.

To create an answer container for an entity

1. Create a new entity that has a foreign key to the target entity and implements `PCAnswerDelegate`.

   For example, to configure an answer container for an account-level question set, create an entity named `AccountAnswer.etx` with a foreign key to `Account` that implements `PCAnswerDelegate`.

   For guidance, examine one of the three base product answer containers, such as `LocationAnswer.eti`.

2. Add an answer array to the base entity in which you are creating an answer container

   For example, to add an answer array for account-level answers, add an array named `AccountAnswers` to `Account.etx`, as follows:

   ```
   Account.etx
     <array
       arrayentity="AccountAnswer"
       cascadeDelete="true"
       desc="Set of answers for this account."
       name="AccountAnswers"
       owner="false"/>
   ```

   For guidance, examine one of the three base product entities that implements answer containers, such as `PolicyLocation.eti`.

3. Define an answer container adapter in the `gw.question` package that implements the `AnswerContainerAdapter` interface.

   For example, to define an answer container adapter for an account-level question set, create a new class file: `gw.question.AccountAnswerContainerAdapter.gs` that implements `gw.api.domain.AnswerContainerAdapter` and overrides appropriate methods and properties. The following code demonstrates a way to define an answer container for account-level question sets:

   ```
   package gw.question
   uses gw.api.domain.AnswerContainerAdapter
   uses java.util.Date
   uses gw.api.productmodel.QuestionSet

   @Export
   class AccountAnswerContainerAdapter implements AnswerContainerAdapter {
   ```

```
  var _owner : Account
  construct(owner : Account) {
    _owner = owner
  }

  override property get Answers() : PCAnswerDelegate[] {
    return _owner.AccountAnswers
  }

  override property get AssociatedPolicyPeriod() : PolicyPeriod {
    return null
  }

  override property get Locked() : boolean {
    return false
  }

  override function addToAnswers(answer : PCAnswerDelegate) {
    _owner.addToAccountAnswers(answer as AccountAnswer)
  }

  override function createRawAnswer() : PCAnswerDelegate {
    return new AccountAnswer(_owner)
  }

  override function getQuestionSetLookupReferenceDate(p0 : QuestionSetType) : Date {
    return Date.Today
  }

  override function removeFromAnswers(answer : PCAnswerDelegate) {
    _owner.removeFromAccountAnswers(answer as AccountAnswer)
  }

  override property get QuestionSets() : QuestionSet[] {
    return QuestionSet.getAll().where(\ q -> q.AnswerContainerType == Account.Type).toTypedArray()
  }
}
```

Note: As shown in the last override of the preceding example, when getting question sets associated with an answer container, filter on `AnswerContainerType` rather than `QuestionSetType`. This technique is preferred because there can be multiple question set types associated with any one answer container type.

For more guidance, examine one of the three base product entities that implements answer containers, such as `PolicyLocationAnswerContainerAdapter.gs`.

4. Update the definition of your entity to implement the answer container using the answer container adapter class you just defined.

   For example, if you are defining an account-level question set and your answer container class is named `AccountAnswerContainerAdapter`:

```
<implementsEntity name="AnswerContainer" adapter="gw.question.AccountAnswerContainerAdapter"/>
name="AnswerContainer" />
```

Step 2: Add a typekey to define the question set type

If you want to define a new question set type, you must add a new typekey to the `QuestionSetType` typelist. The question set type is often, but not necessarily, related to the entity where the answers are stored. However, a new question set type is not required and the question sets for the new answer container can use one of the existing question set types.

To add the typekey

1. Use the **Project** window in Studio to navigate to **configuration** → **config** → **extensions** → **typelist** and open `QuestionSetType.ttx`.
2. Right-click anywhere in the typelist in the left panel of the editor, and then select **Add new** → **typecode** to add a new row.
3. With the new typekey row selected in the left panel of the editor, fill in the new typelist values as appropriate in the right panel of the editor.

For example, for an account-level question set type, add a typekey with a code of `account`, a name of `Account`, and a description of `Account-level question set`.

Step 3: Define Question Set and Question Lookup Tables

Next you must add lookup tables: one for the answer container's question sets and one for the answer container's questions.

To define question set and question lookup tables

1. Use the Project window in Studio to navigate to **configuration** → **config** → **lookuptables** and open `lookuptables.xml`.
2. Add two new `LookupTable` elements in the `Question/QuestionSet Lookup Tables` section of the file, one for question sets and one for questions.

   For example, if you are configuring account-level questions, add `<AccountQuestionSetLookup ... />` and `<AccountQuestionLookup... />` elements. Set the `root` element to `"Account"` and add the appropriate `Dimension` element attributes.

   ```
   <LookupTable code="AccountQuestionSetLookup" entityName="QuestionSetLookup" root="Account">
     <Dimension field="State" valuePath="Account.AccountHolderContact.PrimaryAddress.State"
             precedence="0"/>
     <Dimension field="IndustryCode" valuePath="Account.IndustryCode" precedence="1"/>
     <DistinguishingField field="QuestionSetCode"/>
   </LookupTable>

   <LookupTable code="AccountQuestionLookup" entityName="QuestionLookup" root="Account">
     <Dimension field="State" valuePath="Account.AccountHolderContact.PrimaryAddress.State"
             precedence="0"/>
     <DistinguishingField field="QuestionCode"/>
   </LookupTable>
   ```

   For guidance, examine the question set and question set elements already defined in `lookuptables.xml`, such as `LocationQuestionSetLookup` and `LocationQuestionLookup`.
3. Restart Studio to load the new lookup table elements.

Step 4: Define Question Set and Questions

Now that you have configured an answer container and the necessary lookup tables, you can define the actual question set and questions to use.

To define a question set and questions

1. In the Product Designer navigation panel, select the **Question Sets** node.
2. At the top of the **Question Sets** page, click **Add** to open the **Add Question Set** dialog box.
3. Add a new question set, following the detailed instructions in "Adding a New Question Set" on page 61.

   For example, if you are defining a new account-level question set, select an answer container type of **Account**.
4. Continue following the detailed instructions in "Configuring Question Sets in Product Designer" on page 61 to define individual questions and configure all other aspects of your new question set.

Step 5: Define the User Interface to Display the Question Set

At this point, you have defined an answer container of the appropriate type, added the needed lookup tables, and defined at least one question set and the questions it contains. Next you must define a PCF page to display the question set to the user.

To define a PCF page to display the question set

1. Configure a new PCF page to display the question set.

2. Add a variable to the PCF page to reference the question set to display.

   For example, to display an account-level question set named `accountQuestionSet`, add a variable named `accountQuestionSets` with an initial value of `account.getQuestionSets()`.

3. In the `afterEnter` attribute of the PCF page, call

   ```
   ProductModelSyncIssueHandler.syncQuestions
   ```

   For example, to display an account-level question set, enter the following code in the `afterEnter` attribute:

   ```
   gw.web.productmodel.ProductModelSyncIssuesHandler.syncQuestions({account as AnswerContainer},
           accountQuestionSets, null)
   ```

4. Add a `PanelRef` on the PCF page that points to the `QuestionSetsDV` detail view.

   For example, to display an account-level question set, add a `PanelRef` that points to

   ```
   QuestionSetsDV(accountQuestionsSets, account, null)
   ```

If the new question set is only for information-gathering purposes, you are finished. If you want the new question set to trigger underwriting issues, follow the instructions in “Step 6: Configuring an Answer Container to Trigger Underwriting Issues” on page 73.

Step 6: Configuring an Answer Container to Trigger Underwriting Issues

Question sets can be used for several different purposes, one of which is raising underwriting issues. If you are configuring a question set to raise underwriting issues, you must perform some additional steps.

To configure an answer container to trigger underwriting issues

1. In the `autoRaiseIssuesForQuestions` method in `QuestionIssueAutoRaiser` class, get all available question sets for all the instances of the new answer container type.

2. For each instance of the new answer container type, call the following function:

   ```
   raiseIssuesForQuestionSets(entity.getAvailableQuestionSets(instance), instance, context)
   ```

   For example, to trigger underwriting issues for an account level answer container where there is only one instance of account in a given context (job):

   ```
   raiseIssuesForQuestionSets(getAvailableAccountQuestionSets(account), account, context)
   ```

   In this example, `account` is an instance of type `Account`.

   In situations where there are multiple instances of the answer container type in a given context, the function needs to raise issues for each instance. For example, when triggering underwriting issues on a policy line question set:

   ```
   for (line in period.Lines) {
     raiseIssuesForQuestionSets(product.getAvailableQuestionSets(line), line, context)
  }
   ```

For guidance, examine the functions for handling period, line, and location questions sets in `QuestionIssueAutoRaiser.gs`. For more information about how to configure answers to raise underwriting issues, see “Incorrect Answers” on page 59.

Triggering Actions when Incorrect Answers are Changed

Carriers sometimes want to know when a user changes an answer from incorrect to correct, especially when the change enables the policy to pass validation or unblocks its progress.

You can configure the `IncorrectAnswerChangedAction` class to trigger an action when a user changes an incorrect answer. You also can examine, but not change, the code that identifies answers with values that have been changed by opening the read-only `IncorrectAnswerProcessor` class.

The base configuration of PolicyCenter creates a custom history event for transactions in which answers are changed that had previously blocked the user. The custom history event is used on the **Pre-Qualification** screen of the `Submission` wizard and on the **Qualification** screen of the `RewriteNewAccount` wizard.

To enable other wizard steps to trigger an action when incorrect answers are changed

1. Define a variable in the wizard PCF file to store a map of incorrect answers.

   For example

   ```
   <Variable
     initialValue="new java.util.HashMap<gw.api.productmodel.Question, String>()"
     name="incorrectAnswerMap"
     type="java.util.Map<gw.api.productmodel.Question, String>"/>
   ```

2. Call the incorrect answer processor in the `beforeSave` attribute of the wizard screen. Be sure to call this function before the changed answer's bundle is committed to the database. The function compares the answers with their original values. If the answer is now correct, the function calls the `IncorrectAnswerChangedAction` class to perform the action you specified.

   For example, to check for changes in answers associated with the questions stored in `incorrectAnswerMap`.

   ```
   <JobWizardStep
     beforeSave="gw.question.IncorrectAnswerProcessor.processIncorrectAnswers(policyPeriod,
             incorrectAnswerMap);
             gw.policy.PolicyPeriodValidation.validatePreQualAnswers(policyPeriod)"
     ... >
   </JobWizardStep>
   ```

chapter 6

System Tables

This topic covers the use and configuration of system tables, which are tables that you use to support your business logic.

This topic includes:

What Are System Tables?

System tables are database tables that support business logic in PolicyCenter lines of business. Developers who use Guidewire Studio define system tables with needed columns as entities in the data model. Business analysts who use Product Designer can then examine, edit, and enter values for the system tables columns.

System tables provide additional metadata beyond the capacity of typelists. System tables typically provide storage for information that must be maintained periodically by non-developers. Examples include:

- Class codes
- Territory codes
- Industry codes
- Reason codes
- Reference dates
- Rate factors
- Underwriting companies

You view and manage system tables in Product Designer.

Configuring System Tables

Access system tables in Studio if you are configuring new system tables. Access system tables in Product Designer, if you are adding, removing, or changing data in an existing system table.

This topic includes:

Adding a System Table in Studio

To add a new system table

1. Create the system table entity definition in the same manner as defining or extending any other entity. In the resulting entity or entity extension file, define the columns needed to hold the required system table values.
2. Define the new system table in the file `systables.xml` in conformance with the schema defined in `systables.xsd`. These files are located in **configuration** → **config** → **resources** in Studio.
3. Restart the PolicyCenter server. When the server starts, PolicyCenter creates the needed system table XML file in **configuration** → **config** → **resources** → **systables** based on the information you defined in `systables.xml`.
4. Use Product Designer to add, edit, and delete system table entries. As with other product model changes, you must restart Product Designer or switch change lists before the new system table appears. After committing your changes, the newly-created system table XML file is populated accordingly.

   Note: All system tables must be defined in `systables.xml`. Only system tables listed in `systables.xml` appear in Studio and Product Designer. PolicyCenter loads system tables into the database at system startup. If you do not add the system table XML file name to the `systables.xml` file:

   - Studio does not recognize the file to be a resource.
   - Studio does not load the system table into the database.
   - Studio does not display the contents of the system table in the PolicyCenter interface.

   If you want to control when PolicyCenter loads a particular system table, do not add the system table XML file name to `systables.xml`. You then can write your own code to load the system table into the database when the system table is needed.

Within `systables.xml`, you can specify the order in which to load the system table XML files by using the `FileDefinition Priority` attribute. As PolicyCenter loads the product model, it loads files with a lower priority value before files with a higher priority value. PolicyCenter loads files with the same priority value concurrently.

The loading order is critical if there are dependencies between system tables. For example, the line-specific class code system tables must be loaded prior to the industry codes system table. Therefore, Guidewire sets the priority value for the class code files lower than the priority value for the industry code file.

For *class codes*, the priority is:

```
<FileDefinition Name="gl_class_codes.xml" Priority="1">
        <Entity Type="GLClassCode"/>
    ...
</FileDefinition>
```

For *industry codes*, the priority is:

```
<FileDefinition Name="industry_code_class_code.xml" Priority="2">
        <Entity Type="IndustryCodeClassCode"/>
    ...
</FileDefinition>
```

Configuring File Loading of System Tables

In addition to the `Priority` attribute, the `FileDefinition` element also has a Boolean attribute, `ExternallyManaged`, which sets whether Product Designer or an external system manages a particular system table. It takes the following form:

```
<FileDefinition Name="..." Priority="..." ExternallyManaged="true">
```

Possible values of the `ExternallyManaged` attribute and their meanings are:

- `true` – You cannot view or edit the system table in Product Designer. You must use an external editor to view or make changes to the system table.
- `false` (or not-specified) – You can edit and manage the system table values in Product Designer.

A typical use case for setting `ExternallyManaged` to `true` is when dealing with very large system tables. Very large system tables are not efficient to edit in Product Designer due to its page-at-a-time view of system table data.

Territory codes provide an example use case. Territory codes are a way of encoding a given geographical location for the purpose of rating. Therefore, the territory code system table, `territory_codes.xml`, can become very large, containing as many as a million entities. If you find that editing this system table is not practical using Product Designer, set the `ExternallyManaged` attribute for the territory code system table to `true`, as follows:

```
<FileDefinition Name="territory_codes.xml" Priority="0" ExternallyManaged="true">
    <Entity Type="DB_Territory"/>
</FileDefinition>
```

Setting `ExternallyManaged` to `true` affects only the ability of Product Designer to load the system table for viewing and editing. It does not affect the way in which PolicyCenter loads the system table during system startup.

Verifying System Tables

No two system table files can have the same entity types in them. PolicyCenter loads system tables, unlike other product model entities, into a single database table. Therefore, if the same entity exists in more than one system table, an `IllegalStateException` occurs at system startup.

If you use Studio to change entity definitions and those entities reference system tables, Product Designer does not validate the changes for consistency. To validate system tables, do one of the following:

- Start the PolicyCenter development instance in which you have made the changes.
- Run the command-line validator `gwpc.bat verify-types`.

PolicyCenter can be configured to verify other system table data consistency rules. You can do this configuration by using the `Verifier` parameter to add a verifier class to the system table definition in `systables.xml`. For example:

```
<FileDefinition Name="cancelrefund.xml" Priority="0">
    <Entity Type="CancelRefund" Verifier="gw.systable.verifier.CancelRefundVerifier"/>
```

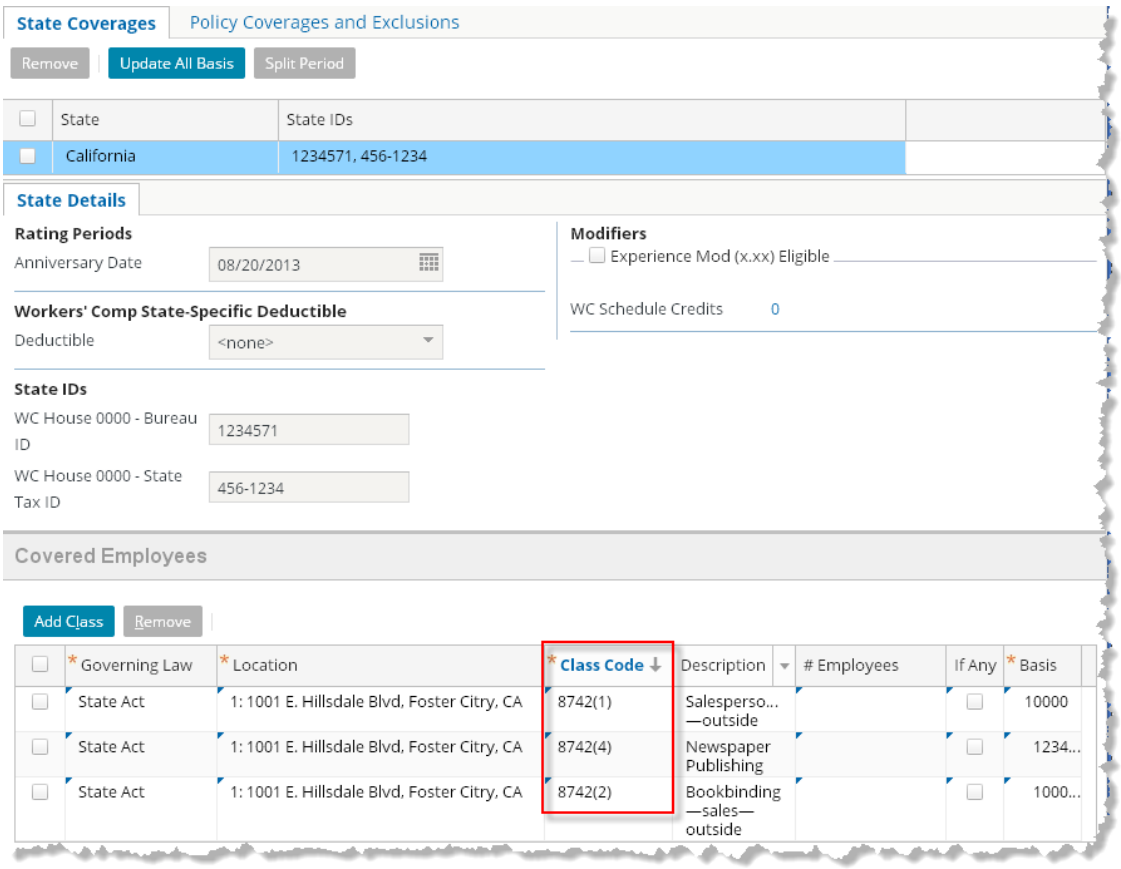
You then must write a `Verifier` class that implements the `Verifier` interface. Include a single method named `verify()` in the class. In the base configuration, verifiers are defined for many common data consistency checks in the system tables, such as:

- All `PublicID` values are valid.
- The end effective date is later than the start effective date for each row.
- Effective dates on multiple rows do not overlap in system class code and industry code system tables.
- Unique indexes defined for the system table entity are unique.

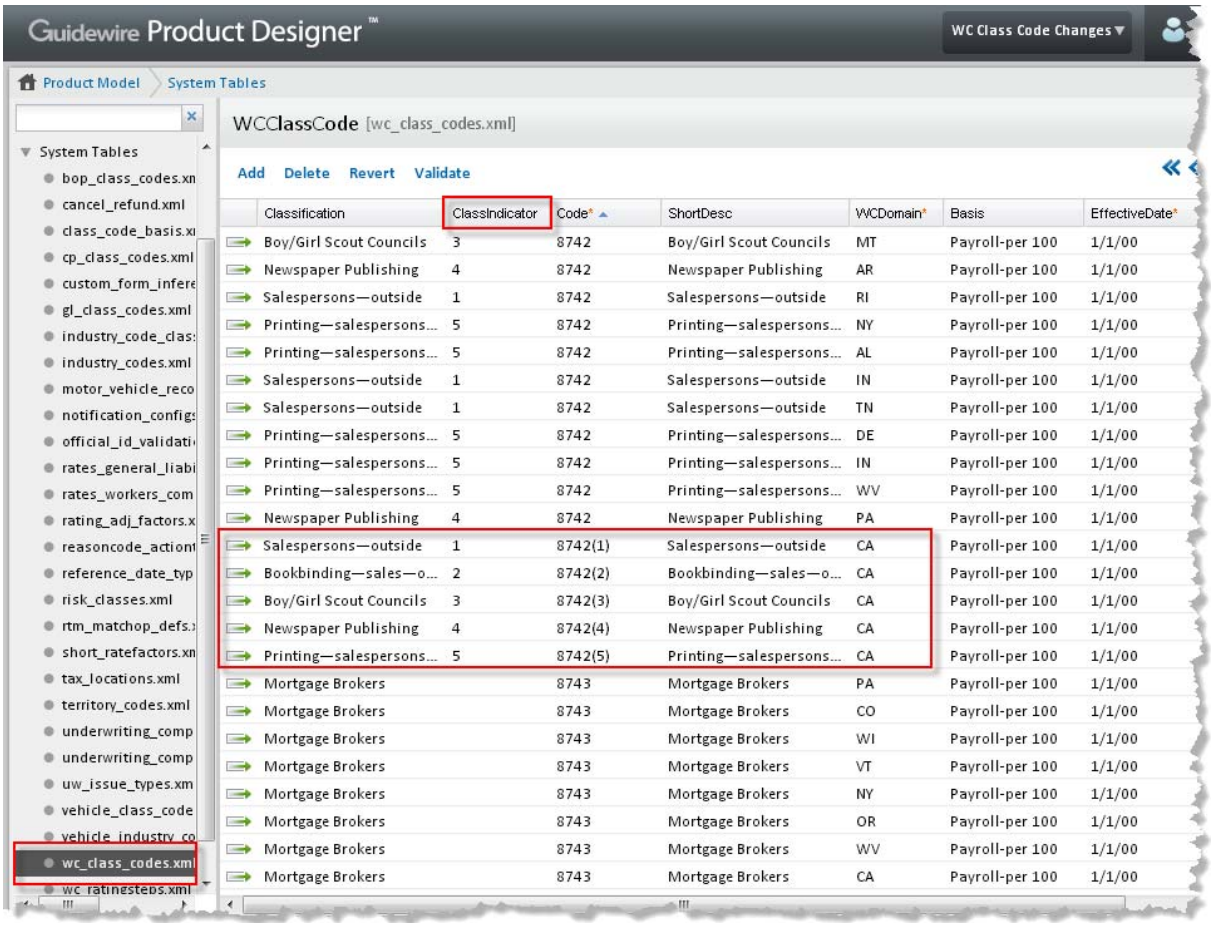
Note: In very large system tables, such as territory codes, complex data verifications can take a long time to execute, which can have a significant impact on server startup time.

Class Codes with Multiple Descriptions

A single class code can have multiple descriptions. Multiple descriptions are important in printing policies and audits and for class code search. You see these multiple descriptions, for example, in the Workers' Comp **Coverages** tab, for class code 8742.



PolicyCenter supports multiple descriptions by including a sequence value in the `ClassIndicator` column of the system table. You can see an example of the sequence value in the workers' compensation line by using Product Designer to open the `wc_class_codes.xml` system table and locating code `8742` for California.



Adding a New System Table

Use Studio to add a new system table.

To add a new system table

1. In Studio, create a file named *`entity`*`.eti`.
   a. In the **Project** window, navigate to **configuration** → **config** → **Metadata** → **Entity.**
   b. Right click the **Entity** node, and then select **New** → **File** from menu.
   c. Enter the name of the entity, including the `.eti` extension.
2. Define the following attributes in your new entity:

```
<entity
    xmlns="http://guidewire.com/datamodel"
    abstract="true"
    desc="..."
    effDatedBranchType="..."
    entity="..."
    exportable="true"
    extendable="true"
    final="false"
    platform="false"
```

```
table="..."
type="effdated">
    <fulldescription> ... </fulldescription>
    <column ... />
    ...
```

Examine an entity definition for an existing system table in the same folder for an example of how to define your system table entity.

3. In Studio, create a file named *entity*.xml.
   a. In the **Project** window, navigate to **configuration** → **config** → **resources** → **systables**.
   b. Right click the **systables** node, and then select **New** → **File** from menu.
   c. Enter the name of the entity, including the .xml extension.
   d. Add the following xml elements to your new XML file:

```
<?xml version="1.0"?>
<import>
</import>
```

   e. Save the XML file.
4. In Studio, add a new FileDefinition element for your new system table to systables.xml:
   a. In the **Project** window, navigate to **configuration** → **config** → **resources** and open systables.xml.
   b. Add a FileDefinition element that provides a link between the system table XML file and the system table entity file. For example:

```
<FileDefinition Name="my_class_codes.xml" Priority="0">
    <Entity Type="MyClassCodes"/>
</FileDefinition>
```

   c. If needed, add parameters to specify a verifier class and whether to manage the system table externally.
5. Use Product Designer to fill in system table values, as follows:
   a. Log into Product Designer. If you are already logged in, log out and log in again to reload the PolicyCenter configuration values.
   b. Navigate to **System Tables**, and then locate and open your new system table XML file.
   c. Add rows to the system table using Product Designer.

See also

- For complete information on data entities and how to create them, see "The PolicyCenter Data Model" on page 149 in the *Configuration Guide*.
- For information on adding localized columns to data entities, including system tables, see "Localizing Administration Data" on page 61 in the *Globalization Guide*.

Notification Config System Table

This topic describes the NotificationConfig system table provided in the base configuration of PolicyCenter. PolicyCenter uses this table to return a notification lead time for the following events:

- Renewal process
- Nonrenewal by carrier
- Cancellation effective date

See also

- "Changing the Cancellation Effective Date on an Open Cancellation" on page 113 in the *Application Guide*

- "Configuring Batch Process Renewals" on page 610 in the *Configuration Guide*

Lead Time in NotificationConfig System Table

The `NotificationPlugin` plugin retrieves the lead time from the `NotificationConfig` system table. The following table describes the columns in the `NotificationConfig` system table.

Column	Description
`EffectiveDate`	Required. Date the row becomes effective. The row is effective on or after this date.
`ExpirationDate`	Date the row expires. The row is effective until the end of the day prior to this date. If omitted, the row is effective indefinitely.
`LeadTime`	Required. Lead time in days.
`ActionType`	Type of action to which this row applies. Click **Search** to display the **ActionType (NotificationActionType)** dialog box where you can select a value from the `NotificationActionType` typelist. The `NotificationActionType` typelist can also specify the `NotificationCategory` that each action type belongs to. If the `Category` column in the `NotificationConfig` system table is `null`, the `getMaximumLeadTime` method uses the `NotificationCategory`.
`Category`	Category of action to which this row applies. The action is a value from the `NotificationCategory` typelist. For example, if the value is `Renewal`, this row applies to any type of renewal.
`Jurisdiction`	Value from the `Jurisdiction` typelist representing the jurisdiction in which this row applies.
`LineOfBusiness`	String matching the pattern code of the line of business to which this row applies.
`PremiumIncreaseThreshold`	Not used in lead time calculation.
`RateIncreaseThreshold`	Not used in lead time calculation.

Wildcards that match any value are supported for `ActionType`, `Category`, `Jurisdiction`, and `LineOfBusiness`. If the value is `null`, a row matches all inputs for the column. Because this situation can lead to multiple matches for any particular combination of criteria, an order of precedence determines which row is returned.

The order of precedence for determining rows that match is as follows (listing highest precedence first). The priority of the columns is `Jurisdiction`, `LineOfBusiness`, and `Category`. The `ActionType` column is not used. However, if the `Category` column is `null`, the plugin checks the value of `NotificationCategory` on the `ActionType` column.

For more information on the Notification plugin, see "Notification Plugin" on page 166 in the *Integration Guide*.

Precedence Example

This example shows precedence during a renewal job. The same precedence rules apply to cancellation.

Consider the following hypothetical rows from the `NotificationConfig` table. The rows are sorted from highest to lowest priority. An asterisk (*) indicates a wildcard that matches any value.

Lead time	Jurisdiction	Line of business	Action type	Category
1	CA	BOP	Material Change Renewal	Renewal
2	CA	BOP	Other Renewal	*
3	CA	BOP	*	Renewal
4	CA	BOP	*	*
5	CA	*	Material Change Renewal	Renewal
6	CA	*	Other Renewal	*

Lead time	Jurisdiction	Line of business	Action type	Category
7	CA	*	*	Renewal
8	CA	*	*	*
9	*	BOP	Material Change Renewal	Renewal
10	*	BOP	Other Renewal	*
11	*	BOP	*	Renewal
12	*	BOP	*	*
13	*	*	Material Change Renewal	Renewal
14	*	*	Other Renewal	*
15	*	*	*	Renewal
16	*	*	*	*

The `PolicyRenewalPlugin` plugin calls the `getMaximumLeadTime` method that takes a `NotificationCategory`. The input parameters are:

Parameter	Value
LineOfBusiness	`WorkersCompLine`
Jurisdiction	`California [CA]`
Category	`Renewal [renewal]`

The `ActionType` column is ignored. The filter matches eight different rows: the ones with lead times 5 through 8 and 13 through 16. Of these values, two rows tie for most specific: 5 and 7. Because the `PolicyRenewalPlugin` plugin requested the maximum lead time, the plugin method returns 7.

Action Type and Notification Category Example

This example shows precedence when the action type uses the notification category in a renewal job. The same precedence rules apply to cancellation.

In the `NotificationActionType` typelist, each action type can also specify the `NotificationCategory` that the action type belongs to. If the `Category` column is `null`, then the `getMaximumLeadTime` method uses the `NotificationCategory` on `ActionType`.

The input parameters are:

Parameter	Value
`LineOfBusiness`	`WorkersCompLine`
`Jurisdiction`	`California [CA]`
`Category`	`Renewal [renewal]`

The row with lead time 2 matches because the `NotificationCategory` is `renewal`. The row with lead time 1 is not a match because the `NotificationCategory` is `cancel`.

Lead time	Jurisdiction	Line of business	Action type	Category
1	CA	*	Other Cancellation `NotificationCategory=cancel`	*
2	CA	*	Other Renewal `NotificationCategory=renewal`	*

chapter 7

Configuring Availability

This topic discusses how you can configure PolicyCenter to control the availability of an entity through the use lookup tables, availability scripts, grandfathering, and offerings. Availability is determined by start and end effective dates. The reference date is compared against the start and end effective dates for a pattern.

This topic includes:

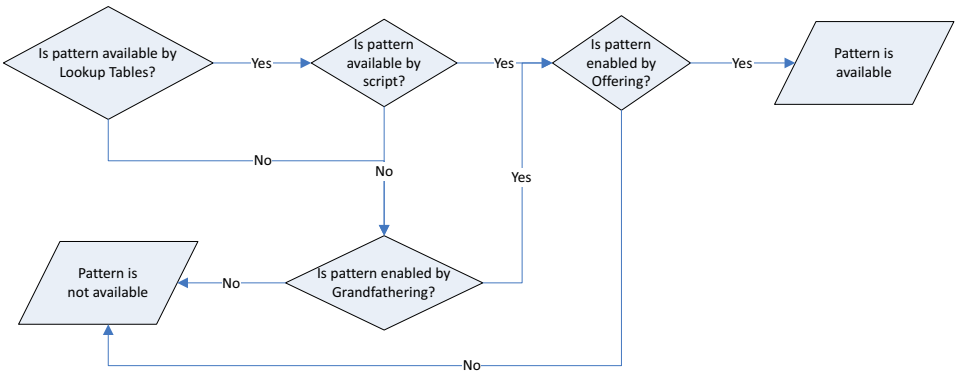
What is Availability?

If a pattern is unavailable, PolicyCenter does not expose that pattern and does not permit you to create instances from the pattern. In some cases, for example in the **Submission Manager**, PolicyCenter applies other criteria as well.

All of the product model patterns except policy line patterns have a configurable availability framework, including all of the core coverage patterns (`CoveragePattern`, `CovTermPattern`, `CovTermOpt`, and `CovTermPack`), exclusions, and conditions. However, PolicyCenter does not apply availability to policy line patterns, but instead controls policy line availability at the product level. There is never a need for a product pattern to be available and one of its underlying policy line patterns to be unavailable. Define other product patterns to make different policy line patterns available.

The availability calculation is a multiple step process as shown in the following illustration.

Determining the Availability of a Pattern



PolicyCenter controls availability using following mechanisms:

- Lookup tables
- Availability scripts
- Grandfathering
- Offerings

Not all mechanisms are available for every pattern type. For example, you cannot specify an availability script for a product pattern.

To determine whether a particular product model pattern is available, PolicyCenter first consults the appropriate availability lookup table to determine the initial set of available entities. If needed, this set can be a very restricted subset of all the possible entities named in the product model. After PolicyCenter determines the initial set of available entities, it applies an availability script, if one exists, to each member of the set to possibly filter out more elements. The availability script is a Gosu expression that returns a Boolean value. Use the availability script to further limit the available elements. If the availability script is empty, PolicyCenter makes all elements of the set available. The availability script only removes available entities from the set—it cannot add them. If the availability lookup table and an availability script set a pattern to unavailable, grandfathering rules are applied that can keep the pattern available if it was already in use. After all of these rules are applied, a pattern that is otherwise available can become unavailable if it is disabled in the selected offering.

The product model entities that are available or unavailable is determined by values store on the policy itself. For example, the set of specified jurisdictions can restrict the availability of a particular coverage. If a coverage is available in California and Nevada only, then the coverage cannot be selected for a building in New York. By using an availability script, any policy characteristic can restrict the availability of a particular product model pattern.

Grandfathering and Offerings

Grandfathering provides ways to continue to make various patterns available even if lookup tables and availability scripts determine that they are unavailable. Typically, grandfathering is used to maintain ongoing policy features to existing customers after those features are no longer offered to new customers. Grandfathering can be applied to the following patterns:

- Coverages
- Coverage term options
- Coverage term packages
- Offerings
- Exclusions
- Conditions
- Modifiers
- Modifier rate factors

Offerings can be used to tailor a product for a particular use case, such as a business-specific product or tiers of coverage. PolicyCenter applies offering logic after grandfathering logic, which means that offerings can make a pattern unavailable even if grandfathering makes it available. Offerings can control the availability of:

- Products
- Policy lines, in package policies only
- Policy terms
- Coverages
- Coverage terms
- Coverage term options and packages
- Exclusions
- Conditions
- Modifiers
- Question sets
- Individual questions within a question set

See also

- "Grandfathering Overview" on page 480 in the *Application Guide*
- "Defining Grandfathering" on page 88
- "Understanding Offerings" on page 481 in the *Application Guide*
- "Configuring Offerings" on page 99

Defining Availability

Typically, some patterns in PolicyCenter are available only:

- As of, or until, a given date
- Within (or never within) a given jurisdiction
- When certain underwriting companies are used to write the policy
- Other, more complex conditions that must be expressed in Gosu code.

Availability is the product model mechanism that captures this logic.

Performance Considerations for Availability

Use the following mechanisms to specify availability logic:

- Availability table
- Availability script
- Grandfathering
- Offerings

Performance of availability lookup tables is much better than availability scripts. Consequently, Guidewire recommends that you specify availability through the availability tables whenever possible. Many examples of available in the base configuration use availability tables only and do not use availability scripts at all.

Defining Availability in Lookup Tables

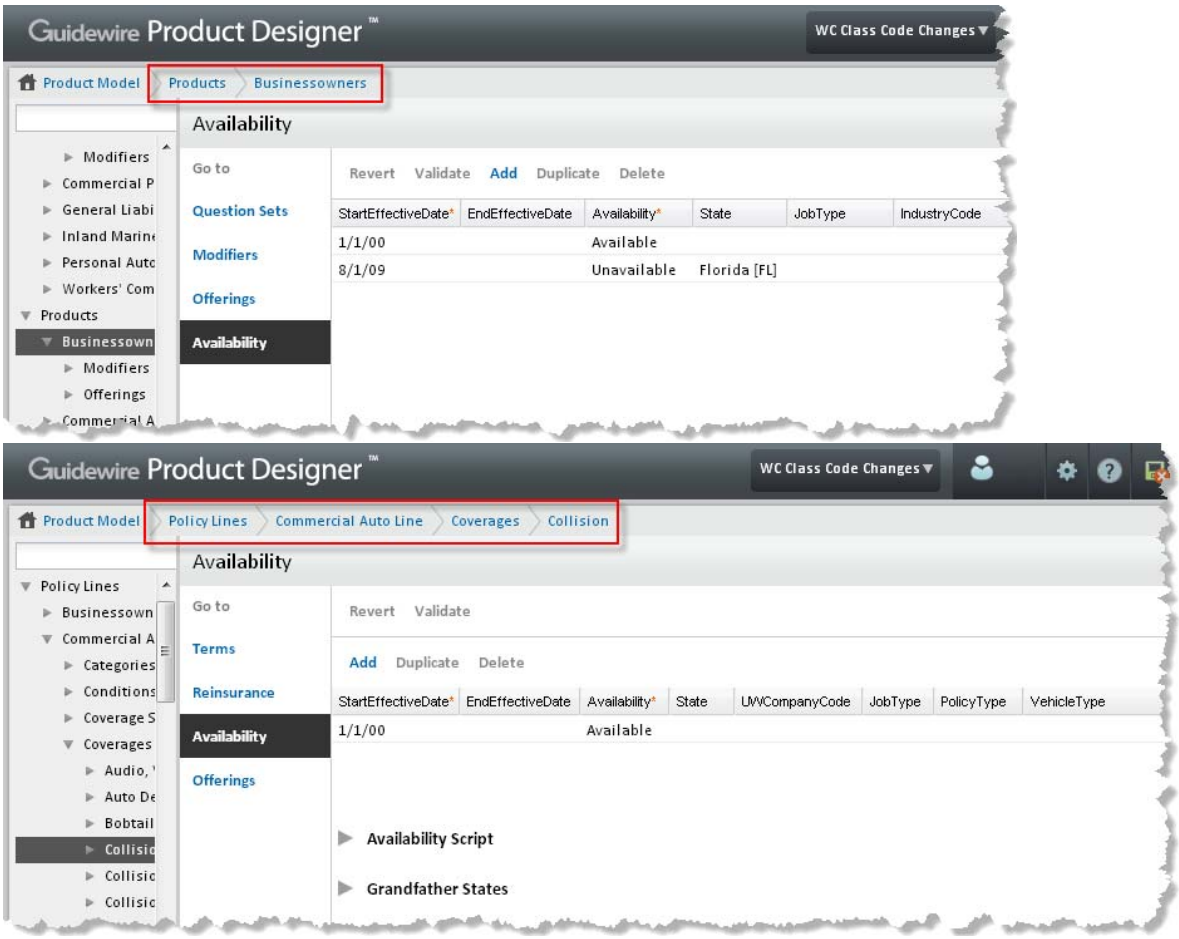
An availability lookup table is a set rows. Each row defines a rule for whether or not the pattern is available. Availability lookup tables are available on the following patterns:

- Products
- Options
- Coverages
- Exclusions
- Rate factors
- Question sets

- Packages
- Modifiers
- Conditions
- Coverage terms
- Questions
- Offerings

Availability in all of these patterns can be based on date and jurisdiction. Product availability can also be based on industry code. Availability for all patterns except products can based on underwriting company and job type. You can extend availability tables with additional columns to handle other availability criteria.

The following illustrations show two availability lookup tables. The first illustration shows the lookup table for the businessowners product. The second illustration shows the lookup table for collision coverage on the commercial auto line. Notice that the two lookup tables have some same and some different columns.



The columns in an availability table, such as start effective date, end effective date, and job type, are called dimensions. In addition, a column called **Availability** designates whether a pattern that matches the row is **Available** or **Unavailable**. When evaluating a pattern, PolicyCenter finds the row in the availability table that matches best, and then uses the state of the **Availability** cell in that row. The best matching row is the row that matches the most dimensions. If multiple rows match the same number of dimensions, the precedence of the dimensions determines the row that determines availability.

Omitted dimensions are wildcards that match any value. If no rows match any defined dimension, the pattern is made unavailable.

IMPORTANT Every availability lookup table must contain at least one row. Product Designer displays validation errors and refuses to commit your changes if it detects any availability tables that do not have at least one row. If you define an availability table without at least one row by, for example, using Studio to edit the XML file, the PolicyCenter server refuses to start.

Defining a Pattern as Available on a Job-by-Job Basis

The **JobType** column specifies the policy transaction type on which to base availability. One use of this field is to make a coverage or other pattern available on a policy transaction basis. For example, you can make a pattern available for new business only. For more information, see "Making a Pattern Available by Policy Transaction Type" on page 480 in the *Application Guide*.

Defining Availability in Scripts

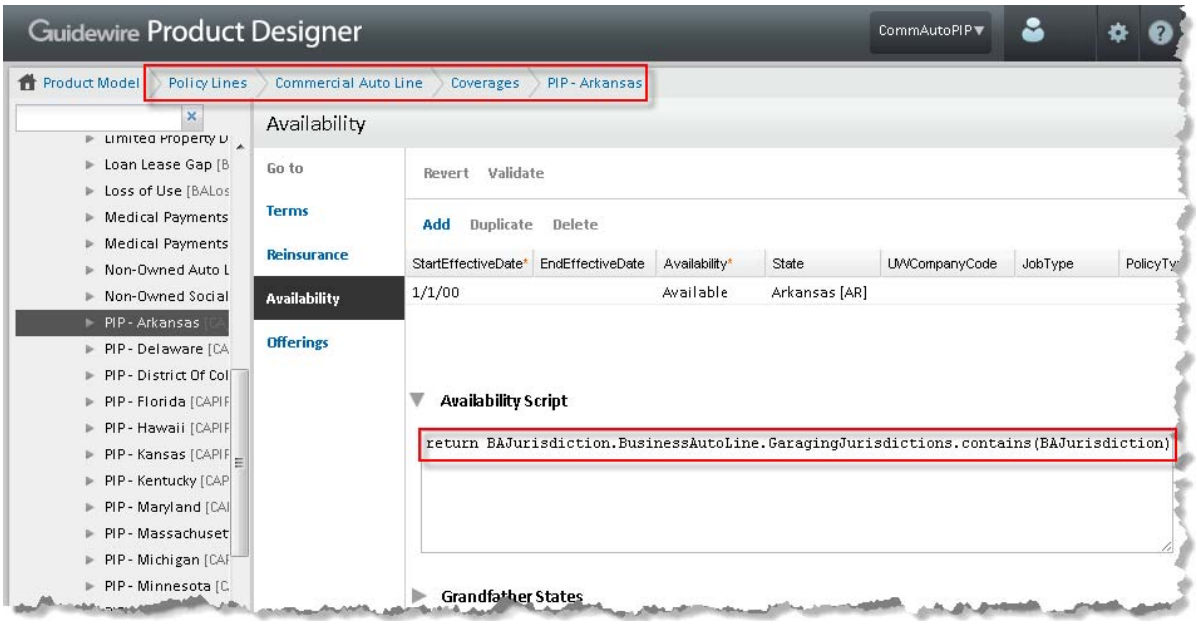
Availability scripts return true/false values. They are written in Gosu and therefore can capture complex logic. However, they are more resource intensive than availability tables.

IMPORTANT Extensive use of availability scripts or the use of complicated scripts can have a detrimental effect on the system performance as PolicyCenter recalculates availability.

PolicyCenter evaluates an availability script only when the availability table determines that a pattern is available. Therefore, an availability script can only reduce the number of available patterns.

The following illustration shows the availability script in the base configuration of personal injury protection coverage in the commercial auto line.

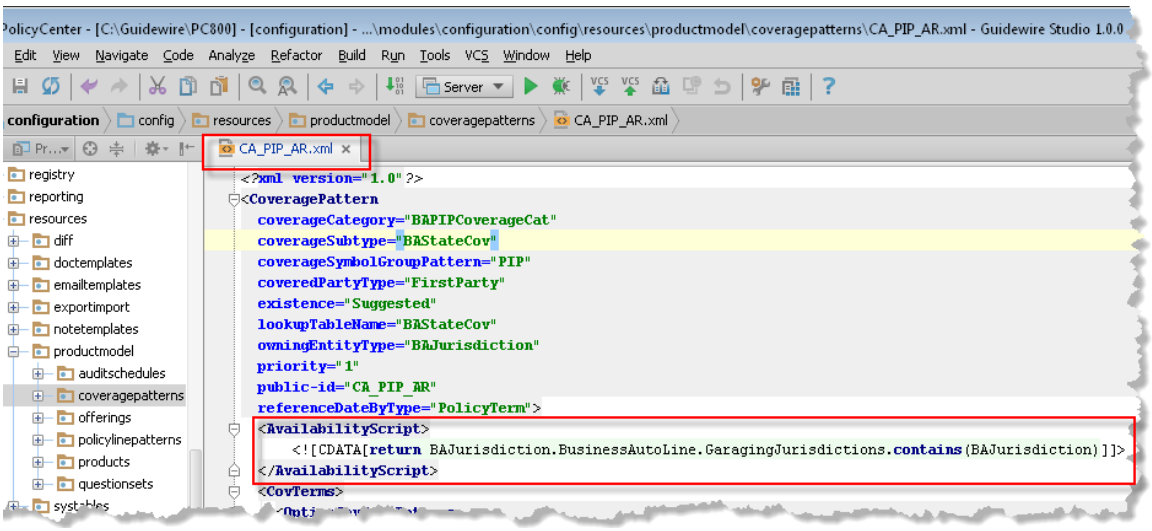
Note: Because the line was previously known as business auto, some of is artifacts continue to use the abbreviation `BA`, while others have been updated to `CA`.



Note: Although Product Designer enables you to write availability scripts, it does not validate or compile the scripts that you write. Therefore, Guidewire recommends that you use Product Designer only to examine scripts, and use the Studio XML editor to write, verify, and compile scripts.

To define or edit a script in Studio

1. Use the **Project** window to navigate to **configuration** → **config** and go to the `resources/productmodel/codeLine` directory.
2. Locate and open the folder that corresponds to the pattern in which you want to write or edit the script. In the preceding example, go to the `BusinessAutoLine/coveragepatterns` folder.
3. Open the XML file that corresponds to the pattern **Code**. The code appears in brackets to the right of the selected item in the Product Designer Navigation panel. In the preceding example, open `CA_PIP_AR.xml`.
4. Locate the `CoveragePattern` element in which you want to write or edit the script, and then locate the `AvailabilityScript` element within that `CoveragePattern`.
5. Define your script as XML CDATA, as shown in the following example. Notice that as you type, Studio provides auto-completion and validates your code.

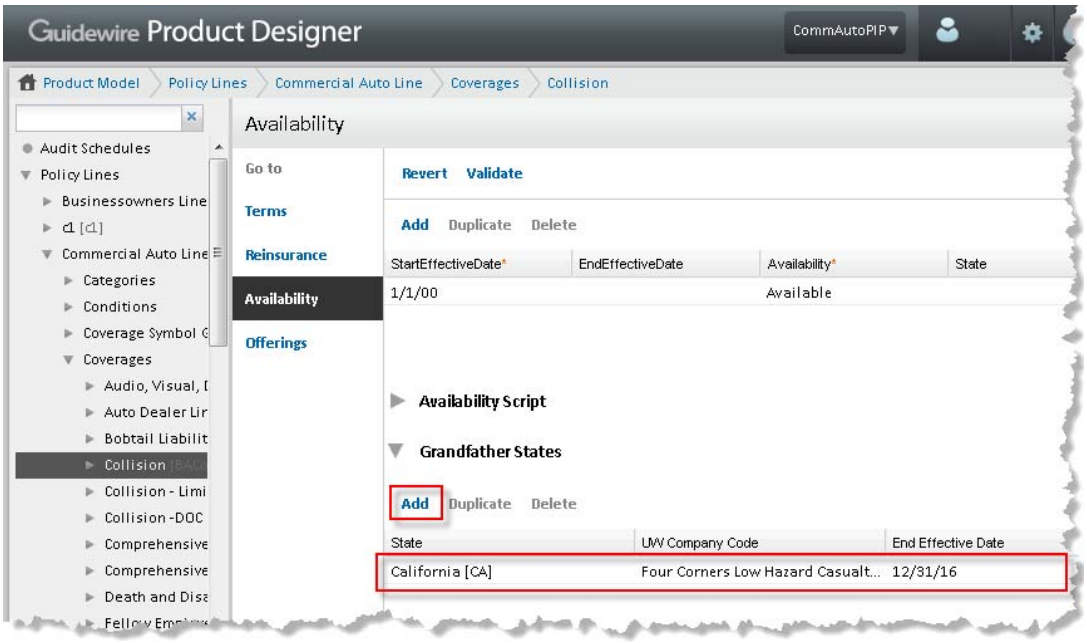


The objects available as symbols in the availability script vary according to the context of the script. For coverages and coverage terms, the script gets a reference to the coverable object. For a coverage term option, the script gets one reference to the parent coverage term and another reference to the specific option value selected by the user. For a modifier, the script gets a reference to a modifiable object.

Defining Grandfathering

Grandfathering enables you to continue to offer a coverage or other pattern to existing customers, even if the pattern is not otherwise available. The pattern is unavailable when writing new business and cannot be added to the policy of an existing customer. However, with grandfathering enabled, the pattern is not removed from existing customers. Grandfathering can be used with the patterns listed in “Grandfathering and Offerings” on page 84.

You configure grandfathering in Product Designer by expanding the **Grandfathering States** section of the **Availability** page for a pattern that supports grandfathering.



To configure grandfathering, specify any or all of the following dimensions:

Field	Description
State	Jurisdiction where grandfathering allows the pattern to be available.
UWCompanyCode	Underwriting company that can offer grandfathering for this pattern.
EndEffectiveDate	Last day this grandfathered pattern can be in force.

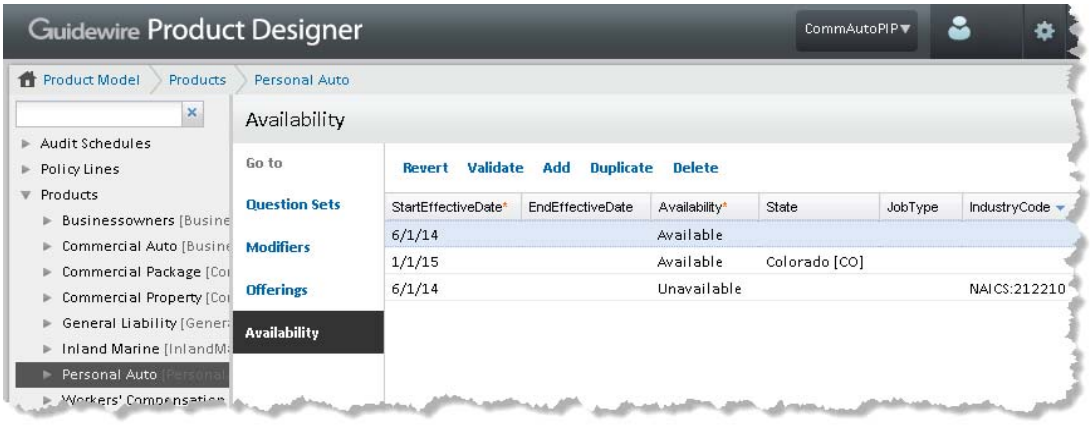
All fields in the **Grandfather States** table can be blank, indicating that grandfathering is allowed, regardless of the values found in the policy for jurisdiction, date, or underwriting company. Therefore, you could add the pattern to the policy even though availability indicates that the pattern is not available, but only if the coverage previously existed on the policy.

See also

- "Grandfathering Overview" on page 480 in the *Application Guide*

Availability Example

For each pattern that supports availability, PolicyCenter creates one condition by default. This condition is available based upon the start effective date set in Product Designer.



Availability is evaluated by finding the row in the table with the most matching columns and then checking whether the pattern is available or unavailable for that row. In this example, the product is available:

- To all non-iron ore (NAICS code 212210) companies as of June 1, 2014.
- To all Colorado companies as of January 1, 2015, even if they are classified as iron-ore.

In this example, an iron ore company in Colorado qualifies for the product beginning on January 1, 2015, as determined by the intersection of the last two availability rows. Because both rows match on the same number of dimensions (columns), the row that decides whether the pattern is available is determined by the precedence attribute on each availability dimension. The precedence attribute ensures that one of the dimensions supersedes other dimensions in the case of a tie. In this example, the **State** column has a lower precedence value (corresponding to higher precedence) than the **Industry Code** column. Therefore the product is available after January 1, 2015 without restriction.

To set the precedence attribute, use the **Project** window in Studio to navigate to **configuration** → **config** → **lookuptables** and open `lookuptables.xml`. You can examine the contents of this file to determine how the precedence attribute is configured by default. For example, for product lookup, you can examine the following elements:

```
<LookupTable code="ProductLookup" entityName="ProductLookup" root="PolicyProductRoot">
  <Dimension field="State" valuePath="PolicyProductRoot.State" precedence="0"/>
  <Dimension field="JobType" valuePath="PolicyProductRoot.JobType" precedence="1"/>
  <Dimension field="IndustryCode" valuePath="PolicyProductRoot.Account.IndustryCode" precedence="2"/>
  <DistinguishingField field="ProductCode"/>
</LookupTable>
```

Notice that the `LookupTable` element defines a field for every dimension in the availability table together with an associated precedence for that dimension. As PolicyCenter calculates availability, availability dimensions with lower precedence numbers override availability dimensions with higher precedence numbers.

Setting the Reference Date

Availability is determined by start and end effective dates. The reference date is compared against the start and end effective dates for a pattern. This section describes how to configure the reference date.

The reference date that is compared against the effective and expiration dates of an availability or rating table is not always the same. It might be the start date of the policy period, the current date, or another date. PolicyCenter has a framework that determines the appropriate reference date before determining the availability of patterns in the product model.

See also

- "Determining the Reference Date" on page 479 in the *Application Guide*

Reference Date Types

The first step to determining the appropriate reference date is to determine the reference date type to be used. The reference date types are:

- **Written date** – The date a transaction is created or the date processing on it starts.
- **Effective date** – The date a transaction is applied to a policy.
- **Rating period date** – For workers' compensation only, the date of the beginning of a split rating period, such as a policy anniversary date.

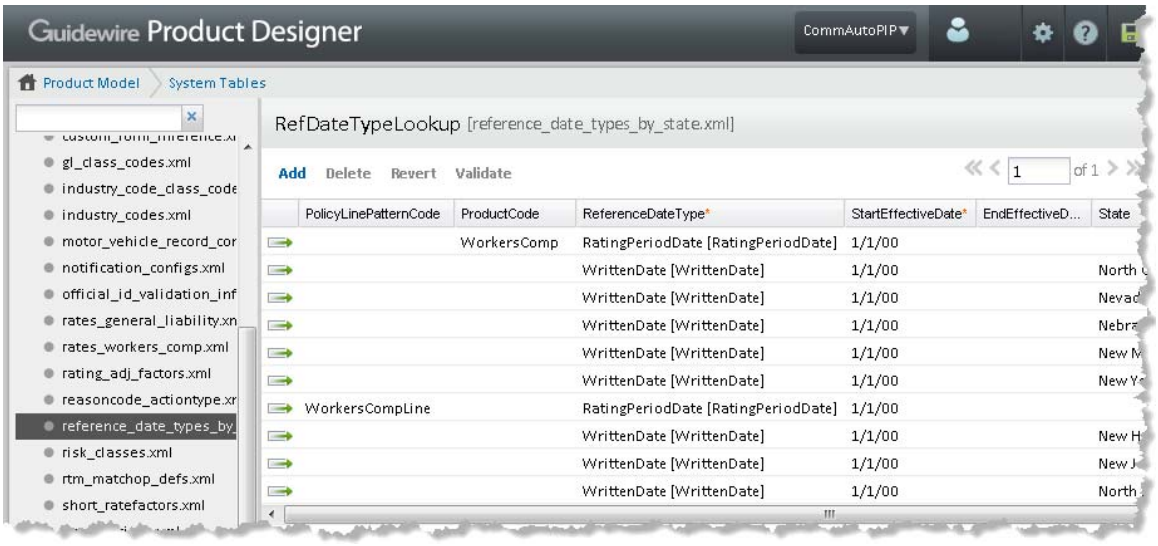
For the `OfferingLookup` and `RefDateTypeLookup` tables, the current system date is used as the reference date. The `RefDateTypeLookup` table determines the reference date type.

The written date field can be made editable in PolicyCenter by permission, if editing is allowed by the carrier.

Specifying the Reference Date Type

The workers' compensation line always uses the rating period date as the reference date. Other lines of business typically use either written or effective date. The appropriate reference date depends on the carrier's filings with the regulatory body and is not consistent across the insurance industry. PolicyCenter uses the `RefDateTypeLookup` system table to determine the appropriate reference date type, based on the jurisdiction, underwriting company, policy line, and product. This system table also contains its own effective dates and expiration dates. The current system date is used to compare against these dates.

To examine the `RefDateTypeLookup` table, use the Product Designer to navigate to **System Tables** and open `reference_date_types_by_state.xml`.



Specifying the Reference Date to Use

After PolicyCenter determines the type of reference date, it must determine which reference date to use.

For example, the policy term has its own written and effective dates. These dates are the date that the policy term was created by a policy transaction and the policy's start effective date, respectively. In contrast, a building added to a property policy mid-term has a written date that matches the written date of the policy change transaction that added the building. It has an effective date that matches the effective date of the policy change transaction that added the building. Further, a coverage added to the policy mid-term would have the written and effective dates associated with the policy transaction that added it.

For each coverage, condition, exclusion, and modifier, you can specify whether to use the written or effective date from the policy term, the coverable object, or the coverage itself. You configure the setting for each applicable object in Product Designer by displaying that object's home page and using the **Reference Date By** list box in the **Advanced** section. Other product model patterns use the associated coverage, exclusion, condition, or modifier reference date when possible. For example, coverage terms use the reference date determined by their parent coverage.

When Reference Dates are Reset

All reference dates are reset at the beginning of a new policy term during renewal or rewrite. Therefore, a building added during a previous policy term has a written date matching the creation date of the renewal. Its effective date is the beginning of the new policy term.

When a coverage or modifier has been bound to a policy, its reference date is set and cannot be changed by future policy change.

For example, a coverage is configured to become unavailable during the course of a policy term. A policy change started after the end date continues to show the previously added coverages and modifiers. These coverages and modifiers continue to be available because the coverage's reference date for the remainder of the policy term is prior to the end of the coverage term's availability. New coverages and modifiers added through subsequent policy changes, however, use new reference dates based upon the transaction that adds them.

Customizing the Reference Date Lookup

You can choose not to use the `RefDateTypeLookup` table for determining the type of reference date. Instead, you can use the `IReferenceDatePlugin` plugin as the starting point for reference date processing. You can add code to do your own reference date calculation, or integrate with an external system that determines reference dates.

See also

- "Reference Date Plugin" on page 168 in the *Integration Guide*

Extending an Availability Lookup Table

PolicyCenter contains several lookup tables that you can use in availability calculations. Often there is no need to create a new lookup table unless you create a new coverage entity. Instead, you usually can achieve needed results by adding new columns to an existing availability lookup table. For example, you could add a column to control availability such that a $500 deductible is available on a monoline product, but not available in a package product.

To add a new column to a lookup table

- Step 1: Extend an Availability Lookup Entity
- Step 2: Define the Column in the Availability Lookup Table
- Step 3: Using the Updated Availability Column

As an example, the following topics explain how to extend the `CovTermOptLookup` entity.

Step 1: Extend an Availability Lookup Entity

To add a new column to an availability lookup table, you first must extend the lookup entity to define the new column. The entity you extend depends on the type of product model pattern you want to change. This scenario extends the `CovTermOptLookup` entity with a new column to capture the product for which the coverage term option is available or unavailable.

To define the lookup entity extension

Follow these instructions to define the lookup entity extension in `CovTermOptLookup.etx`.

1. In the **Project** window in Studio, navigate to **configuration** → **config** → **metadata** → **entity.**
2. Right-click `CovTermOptLookup.eti` and select **New** → **Entity Extension** to display the **Entity Extension** dialog box. Accept the default file name extension by clicking **OK**.

   Studio opens a new extension file named `CovTermOptLookup.etx`.
3. In the Entity editor for `CovTermOptLookup.etx`, select the root element. In the element drop-down list adjacent to the **Add** button, select **column** to add a new `column` to the lookup table.
4. Add the following values to new `column` define the new `ProductCode`:

Property	Value
name	`ProductCode`
type	`varchar`
desc	`code of product`

5. With your new column selected in the Entity editor, select **params** from the element drop-down list to a a `columnParam` element.
6. Add the following values to the new `columnParam`:

Property	Value
name	`size`
value	`50`

Step 2: Define the Column in the Availability Lookup Table

Next, you must modify the lookup table definition to include the new column. In this example, the new availability column is available only for coverage term options for coverages that are associated with commercial property buildings. To modify the lookup table definition, add a `<Dimension>` element to the existing `<LookupTable>` element. You must modify additional nodes to associate the new availability column with coverage term options for other policy lines or with other coverable entities in the commercial property line.

To add a new column to the lookup table

1. In Studio, use the **Projects** window to navigate to **configuration** → **config** → **lookuptables** and open `lookuptables.xml`.
2. Search for `CPBuildingCovOpt` and add the `<Dimension>` element, as shown in the following code fragment.

```
<LookupTable code="CPBuildingCovOpt"
        entityName="CovTermOptLookup"
        root="covTerm"
        appliesTo="CPBuilding">
  <Filter field="CovTermPatternCode" valuePath="covTerm.PatternCode"/>
  <Dimension field="State" valuePath="covTerm.Clause.OwningCoverable.CoverableState" precedence="0"/>
  <Dimension field="UWCompanyCode" valuePath="covTerm.Clause.PolicyLine.PolicyPeriod.UWCompany.Code"
          precedence="1"/>
```

```
    <Dimension field="JobType" valuePath="covTerm.Clause.PolicyLine.PolicyPeriod.Job.Subtype"
            precedence="2"/>
    <Dimension field="ProductCode" valuePath="covTerm.Clause.PolicyLine.PolicyPeriod.Policy.ProductCode"
            precedence="3"/>
    <DistinguishingField field="CovTermOptCode"/>
  </LookupTable>
```

The highlighted `<Dimension>` element is the new element. This element provides information for the `ProductCode` field defined on the lookup entity in "Step 1: Extend an Availability Lookup Entity" on page 93. PolicyCenter decides whether a coverage term is available by comparing the `ProductCode` in use on the policy with the `ProductCode` specified by the row in the lookup table. The `valuePath` attribute designates how PolicyCenter finds the `ProductCode` for comparison. The `valuePath` starts with the entity designated in the `root` attribute. When an availability lookup table has multiple matching rows, the `precedence` attribute determines that rows matching just the `ProductCode` column have lower precedence than rows matching other columns.

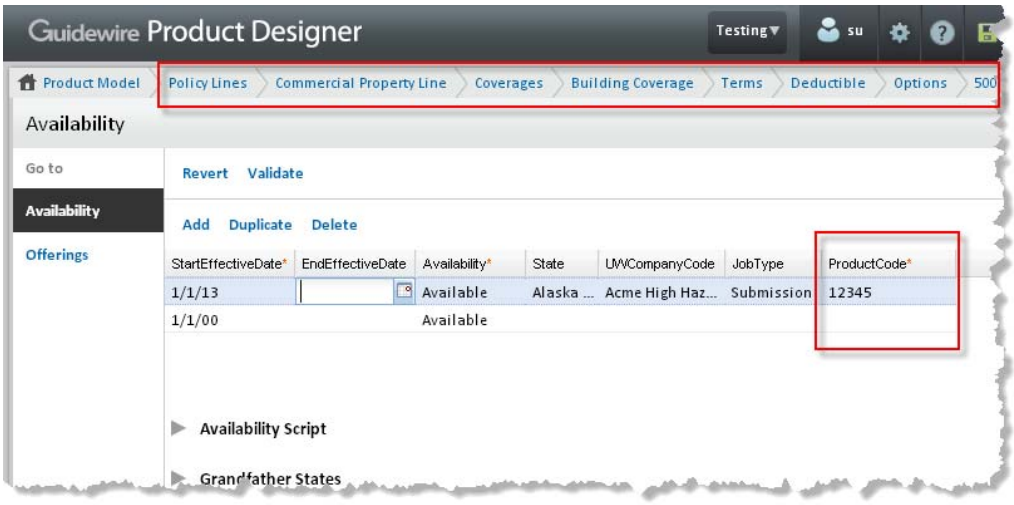
Step 3: Using the Updated Availability Column

After saving your work, log into Product Designer to configure availability for the associated coverage object.

To use the updated availability column

1. Log into Product Designer. If you are already logged in, log out and log in again to reload the PolicyCenter configuration values.
2. Navigate to the affected availability object. In this example scenario, navigate to **Policy Lines** → **Commercial Property Line** → **Coverages** → **Building Coverage** → **Terms** → **Deductible** → **Options** → **500**. Under **Go to**, click **Availability.**
3. Add availability rows as needed.

The following illustration shows the changes to the Commercial Property policy line availability page for commercial property deductible term options. The table on the **Availability** page now has a **ProductCode** column.



Creating a Subtype to Maintain Distinct Column Definitions

When you extend a lookup table entity, the new column is added to all lookup table definitions that use that entity. If the column does not apply to all lookup tables, then define a subtype of the entity, and define the column on that subtype.

For example, if you want to extend the lookup table with a `BodyType` column for personal vehicle coverage terms, you must create a new subtype. Create a new `PAVehicleCovTermLookup` entity extension subtype and add the `BodyType` column. Do not add the `BodyType` column to the `CovTermLookup` entity extension, because it does not apply to all coverage terms.

Therefore, define a new entity in **configuration** → **config** → **extensions** → **entity** as follows:

Property	Value
Entity	PAVehicleCovTermLookup
Entity Type	subtype
Supertype	CovTermLookup

Add a `typekey` element to the new entity with the following parameters:

Property	Value
name	BodyType
typelist	BodyType

In `lookuptables.xml` for `PAVehicleCovTerm`, use `PAVehicleCovTermLookup` instead of `CovTermLookup`:

```
<LookupTable code="PAVehicleCovTerm" entityName="PAVehicleCovTermLookup" root="PersonalVehicle">
  <Filter field="PolicyLinePatternCode" valuePath="PersonalVehicle.PALine.PatternCode"/>
  ...
  <Dimension field="JobType" valuePath="PersonalVehicle.PALine.Branch.Job.Subtype" precedence="2"/>
  <Dimension field="BodyType" valuePath="PersonalVehicle.BodyType" precedence="3"/>
  <DistinguishingField field="CovTermPatternCode"/>
</LookupTable>
```

Reloading Availability Data

You can make changes to availability data in Product Designer and upload these changes to a running PolicyCenter production server or clustered group of servers. The types of availability data you can upload are:

- Lookup tables
- Availability scripts
- Grandfather states

For example, you want to discontinue the personal auto product in Alaska one month from now, or you want to grandfather collision coverage in Texas. You can make these changes to the product model in Product Designer, and then upload the changes dynamically to PolicyCenter without taking the server off line.

IMPORTANT Changes in Studio other than availability changes are not uploaded to PolicyCenter. For example, if you change availability references, PolicyCenter cannot reload availability.

To reload availability, you must copy the product model files to an external directory that is accessible to the application server. Then in PolicyCenter, use the **Server Tools** → **Product Model Info** screen to access the **Reload Availability** command. Alternatively, you can restart the PolicyCenter server to read the new availability data from the external directory.

PolicyCenter loads availability data from the directory specified by the `ExternalProductModelDirectory` parameter in `config.xml` at both server startup time and when a reload is requested. However, it does so only if all of the following are true:

- The parameter specifies a directory accessible to the application server.
- The directory is not in the deployment directory tree.
- The directory contains a complete product model definition that only defines existing patterns.
- The directory contains at least one lookup XML file ending in `-lookups.xml`.
- The files in the directory are valid.

Otherwise, PolicyCenter loads availability data from the standard configuration directory at server startup time, and any attempt to reload fails.

External Product Model Directory

The structure of the external product model directory must match that of the `productmodel` directory located in the `modules/configuration/config/resources` directory of your PolicyCenter installation directory. For example, the XML file containing lookups for the `PACollisionCov` coverage must be `PACollisionCov-lookups.xml` in the `coveragepatterns` subdirectory.

The external directory you specify must contain a complete copy of the product model resources directory. Before reloading, PolicyCenter performs validations on the product model definitions in the external directory. These validations are the same validations that PolicyCenter performs on the product model at startup. The server loads only availability information from the external directory. The rest of the product model definition is taken from the `modules/configuration/config` directory of your PolicyCenter installation directory. Therefore, you cannot, for example, add a new coverage pattern to the product model on a running server.

Availability Reload and Open Transactions

Reloading availability takes effect immediately and can affect open transactions. Changes may become apparent at quote, bind, or issuance. Changes may become apparent when moving between screens or making selections in a transaction.

The Reload Availability Screen

In PolicyCenter, you can reload availability with the server running in either production or development mode. To do so, click **Reload Availability** on the **Server Tools** → **Product Model Info** screen. The server attempts to synchronize the lookup entities and the existing product model availability data with the XML files stored in the external product model directory. If the reload is successful, PolicyCenter displays an informational message. If reload is not successful, PolicyCenter displays an error message. In either case, you can check the server log files for details of the reload operations or problems that occurred.

You can access the **Product Model Info** screen only if you belong to a user role with **View ProductModelInfo tools page** [`toolsProductModelInfoview`] permission. In the base configuration, only the **Superuser** role has this permission.

> **Note:** To reload availability when the server is in development mode, in addition to belonging to a role with the required permission, the `EnableInternalDebugTools` parameter in `config.xml` must be `true`.

Reload Availability in a Clustered Environment

You can reload availability data in a clustered environment. In a cluster, only the batch server performs the actual reload. When you click **Reload Availability**, PolicyCenter starts the reload on the batch server. It does not matter which server your user session is running in.

After reloading lookup tables, the batch server sends a message to all other nodes across the cluster to update the availability data.

All servers in the cluster must have access to the external product model directory. If the batch server cannot access the external directory, PolicyCenter displays an error message after clicking **Reload Availability**. However, if other servers in the cluster cannot access the external directory, PolicyCenter sends an error message to the log file. If a server in the cluster cannot access the external product model directory, it can have different availability data than the batch server. Therefore, you must take care in making sure the external product model directory is available to all servers in a cluster.

Reloading Availability Example

This topic provides step-by-step instructions for reloading availability data. The server can be in development or production mode. The instructions assume the following:

- PolicyCenter is deployed in its base configuration.
- You have loaded the **Large** sample data as described in "PC Sample Data" on page 155 in the *System Administration Guide*.

Step 1: Enable the Configuration Parameter

To set up required configuration parameters for reloading availability

1. Decide on the location for the external product model directory. This directory must:
   - Be accessible to the application server
   - Be named `productmodel`
   - Not be in the deployment directory of the application server

   For example, you could name this directory:

   ```
   c:\PC_Reload_Availability\productmodel
   ```
2. In Studio, use the **Project** window to navigate to **configuration** → **config** and open `config.xml`.
3. Find the `ExternalProductModelDirectory` parameter.
4. Enter the fully qualified path to the reload directory in the `value` attribute. For example, you can define `ExternalProductModelDirectory` as follows:

   ```
   <param name="ExternalProductModelDirectory" value="c:\PC_Reload_Availability\productmodel"/>
   ```
5. If the server is in development mode, find the `EnableInternalDebugTools` parameter and set its value to `true`. To view the **Reload Availability** screen when the server is in development mode, this parameter must be `true`. For more information, see "The Reload Availability Screen" on page 96.
6. Start or restart PolicyCenter to pick up the changes to `config.xml`.

Step 2: Make Changes to Availability in Product Designer

Make changes to lookup tables, availability scripts, and grandfathering in Product Designer. This section provides a few example changes to availability data in the personal auto line.

Note: Changes in Product Designer other than availability changes are not uploaded to PolicyCenter. For example, if you change availability references, PolicyCenter cannot reload availability.

To change the lookup table in the personal auto product

1. In Product Designer, open **Products** → **Personal Auto**.
2. Under **Go to**, click the **Availability** link.
3. Click **Add**. Specify the following properties:

Property	Value
StartEffectiveDate	The current date
Availability	`Unavailable`
State	`Colorado`

4. In a file browser, navigate to:
   PolicyCenter_Installation/modules/configuration/config/resources/productmodel/products/PersonalAuto
   Notice that the modification date of `PersonalAuto-lookups.xml` shows that this file has just been modified.

To change an availability script in the personal auto line

1. In Product Designer, open **Policy Lines** → **Personal Auto Line**.
2. In the navigation panel, expand the nodes to open **Coverages** → **Comprehensive** → **Terms** → **Comprehensive Deductible** → **Options** → **1000**.
3. Under **Go to**, click the **Availability** link.
4. On the **Availability** page, click to expand the **Availability Script** section. Add the following code to remove the value **1000** in California:
   ```
   if (State == "CA") {return false}
   ```
5. In a file browser, navigate in the *PolicyCenter_Installation* directory to:
   modules/configuration/config/resources/policylinepatterns/PersonalAutoLine/coveragepatterns
   Notice that the modification date of `PACollisionCov.xml` shows that this file has just been modified.

Step 3: Copy Changes to External Product Model Directory

1. In a file browser, navigate to:
   PolicyCenter_Installation/modules/configuration/config/resources
2. Copy the `productmodel` directory to the clipboard.
3. Paste the `productmodel` directory to the `ExternalProductModelDirectory` you defined in “Step 1: Enable the Configuration Parameter” on page 97.

Step 4: Reload Availability in PolicyCenter

1. In PolicyCenter, log in as a user who can view the **Server Tools** → **Product Model Info** screen. In the base configuration, login as `su` with password `gw`.
2. Press `ALT+SHIFT+T` to open the Server Tools screen.
3. Select **Server Tools** → **Product Model Info**.
4. Click **Reload Availability** on the **Product Model Info** screen.
5. If the reload succeeded, select **Actions** → **Return to PolicyCenter**.

Step 5: Verify Changes to Availability in PolicyCenter

In PolicyCenter, verify the changes to availability. The following assumes that you made the example changes in “Step 2: Make Changes to Availability in Product Designer” on page 97.

1. Start a submission and set the **Default Base State** to **Colorado**.
   Note that the **Personal Auto** product is not available.
2. Start a submission for personal auto in California.
3. Advance to the **PA Coverages** screen.
   Note that the value **1000** for **Collision Coverage** is not available.

chapter 8

Configuring Offerings

Some carriers offer variations of their policies depending on customer type or sales channel. *Offerings* enable you to define different product types for different situations. If a product provides offerings, you can choose an offering in the submission, issuance, policy change, renewal, or rewrite transaction.

After you define a set of offerings, you then can enable or disable any of the following product model patterns in any combination of offerings:

- Policy lines, in a package policy
- Policy terms
- Coverages
- Coverage terms
- Coverage term options and packages
- Exclusions
- Conditions
- Modifiers
- Question sets

Enabling (including) a product model pattern in an offering causes the pattern to appear when the user has selected the corresponding offering. Conversely, disabling (excluding) a product model pattern in an offering removes the pattern from the available options when the user has selected the corresponding offering. For example, if you include a $100 deductible coverage term option only in an offering named Premium, the $100 deductible option appears only if the user selects the Premium offering.

This topic includes:

See also

- "Understanding Offerings" on page 481 in the *Application Guide*

Working with Offerings in the Product Model

You define offerings in Product Designer by clicking the **Offerings** link under **Go to** on the coverage pattern home page where you want to provide an offering. For example, to define an offering for the commercial package product, display the **Commercial Package** home page, then under **Go to**, click **Offerings** to display the **Offerings** page.

In the base configuration, the following products provide offerings:

- Businessowners
- Commercial Auto
- Commercial Package
- General Liability
- Personal Auto

This topic includes:

Product Offerings Page

In Product Designer, each product has an **Offerings** page where you manage the offerings for that product. On this page, you can examine existing offerings and add or remove offerings as needed. When you add a new offering, you must provide a **Code** and a **Name**. After you have defined a set of offerings, you can arrange the order in which PolicyCenter displays them by specifying a **Sequence**.

After you have defined a set of offerings, you can use either of two methods to define the product model patterns enabled or disabled in each offering:

- By offering in the product **Selections** page – A single view where you can enable or disable all product model patterns in the selected offering. For more information, see "Product Selections Page" on page 100
- By pattern in the product model pattern **Offerings** page – A separate view for each product model pattern where you can enable or disable the single, selected pattern in the set of available offerings. For more information, see "Product Model Pattern Offerings Page" on page 102

Product Selections Page

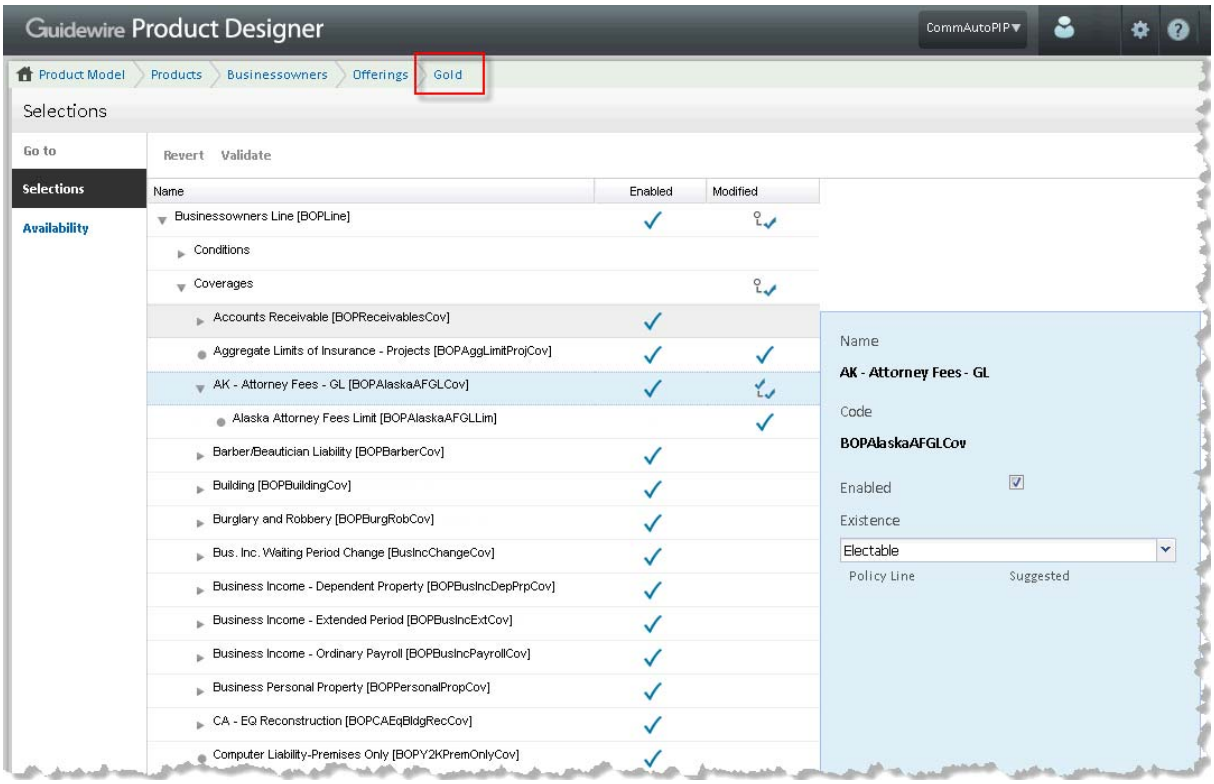
In the offerings **Selections** page for a product, you can enable or disable parts of the product for this offering. The **Selections** page displays an expandable list of the product model patterns that can be enabled or disabled in the selected offering, including:

- Coverages, coverage terms, conditions, exclusions, and modifiers on each policy line included in the product
- Policy terms
- Product modifiers
- Question sets

In each offering, you can enable or disable any of these items. The **Modified** column on the **Selections** page uses icons to help you find items whose enabled or disabled status is different in the selected offering than in the base product:

Modified column	Description
blank (no icon)	This item and all of its descendants are the same as in the base product.
	This enabled/disabled status of this item is different from the base product.
	The enabled/disabled status of this item and one or more of this item's descendants is different from the base product. Expand the descendant nodes to locate the modified items.
	The enabled/disabled status of one or more of this item's descendants is different from the base product. Expand the descendant nodes to locate the modified items.

The following illustration shows a portion of the Gold offering in the Businessowners product, where you can see an example of each of the **Modified** icons.



Configuring Conditional Existence

Existence defines whether a product model clause is required, suggested, or electable, as follows:

- **Required** – The clause is selected and cannot be removed from the offering.
- **Suggested** – The clause is selected but can be removed from the offering.
- **Electable** – The clause can be selected but is initially not selected in the offering.

In addition to setting existence on individual coverages, conditions, and exclusions, you also can define conditional existence based on the offering the user selects. The definition of existence at the offering level overrides the definition of existence at the policy line clause level.

You configure offering existence in the blue offering editor. The **Existence** field can be seen in the preceding illustration, where the **Gold** offering sets the **AK Attorney Fees** coverage to **Electable**.

Product Model Pattern Offerings Page

In the **Offerings** page for a product model pattern, you can choose which offerings are to include the selected pattern from among the entire set of offerings you have defined. You also can define for each product model pattern whether that pattern is automatically included in any new offerings that are defined.

The **Offerings** page consists of the following two lists. Use the arrow buttons between the lists to move the selected offerings from one list to the other.

- **Included or implied in these offerings** – Move the offering to this list to include the product model pattern in the offering.
- **Disabled in these offerings** – Move the offering to this list to exclude the product model pattern from the offering.

To automatically include the selected product model pattern in all new offerings that are defined in the future, select the **Include in all new offerings** check box. Otherwise, if this check box is cleared, new offerings are initially listed in the **Disabled in these offerings** list.

Product Offerings Availability Page

Offerings have their own availability and grandfathering configuration. Availability for offerings is checked last, after all other availability checks for a product model pattern. For this reason, if all other checks enable the pattern to be available, the offering availability check can make it unavailable. However, if any of the other checks makes the pattern unavailable, the offering availability check cannot make it available.

To configure availability and grandfathering for offerings, use Product Designer to navigate to a product's **Offerings** page, and then under **Go to**, click the **Availability** link. Configure availability for the offering in the same way as you configure availability in other patterns, including an **Availability Script** and **Grandfather States**, if needed. For more information, see "Configuring Availability" on page 83.

Because offerings can disable certain product model patterns, the offering selected for a policy period can affect availability for entities within that period. For example, suppose a coverage is enabled in the `Gold` offering but disabled in the `Bronze` offering. That coverage is not available to any policy that selected the `Bronze` offering, even if it would be available according to the lookup table and script.

Offerings can be selected directly by a control in a PCF page or by a question set of type **Offering Selection**.

Offerings and Question Sets

In Product Designer, you can specify:

- Which question sets appear on the **Offerings** screen in PolicyCenter
- Whether an offering includes a question set
- Whether an offering is available based on answers to questions and other criteria

Configuring Questions Sets to Appear on the Offerings Screen

In PolicyCenter, the **Offerings** screen provides the controls that enable you to select an offering. The **Offerings** screen displays question sets that have their type set to **Offering Selection**. In the base configuration, the **BOP Offering Questions** question set is an example of selecting an offering by using a question set. You can examine this question set under **Question Sets** in Product Designer. The **Question Set Type** field appears on the **Question Set** home page.

Configuring Whether an Offering Includes a Question Set

PolicyCenter can display question sets or individual questions based on the selected offering. You can also specify whether a question set is automatically disabled in new offerings.

Note: If the question set type is **Offering Selection**, these actions are unavailable.

See also

- "Configuring Offerings for Question Sets" on page 62

Configuring Whether an Offering is Available

In PolicyCenter, the offerings available from the **Offering Selection** list can be affected by the answers to the offering questions and other factors. For example, the businessowners product uses the Product Designer **Offerings** → **Availability** tab to enable or disable the **Gold** and **Partners** offerings.

Answers to questions is the last check to determine if an offering is available. The offering is available only if the user provides correct answers for all questions.

See also

- "Product Offerings Availability Page" on page 102

chapter 9

Checking Product Model Availability

This topic discusses how PolicyCenter handles missing and unavailable items on a policy transaction.

This topic includes:

What Is Product Model Availability?

Checking product model availability refers to the process of comparing the product model data in a policy transaction against the set of currently-defined product model patterns. This process discovers if any product model elements exist on the transaction that are not available according to the product model definition. It also ensures that required elements, such as required coverages that must exist on the policy, exist.

For example, suppose the product model stipulates that a coverage pattern must exist on a particular policy line only in a certain jurisdiction. This set of conditions might be set, for example, through the availability of the coverage pattern. During a policy submission in PolicyCenter, a user realizes that the jurisdiction is set incorrectly and changes it. When the location (the jurisdiction) changes, a coverage that previously was available becomes unavailable. Therefore, the coverage must not be allowed to exist on the policy.

To handle these situations, PolicyCenter periodically checks the current policy data against the product model patterns within the application data model. The checks happen at various points in the wizards. If PolicyCenter detects a discrepancy between the policy data and the product model patterns, it can handle the discrepancy in any of the following ways:

- Ignore the difference, display a message, and continue through the wizard.
- Attempt to correct the issue, for example, by removing an unavailable coverage, as it moves to the next wizard step or as it quotes the policy.

- Display a message, either blocking or non-blocking, that alerts the user to the problem. The user then can resolve the issue. Blocking messages, such as warnings, prevent the user from continuing to the next wizard step until the problem is corrected.

The actions PolicyCenter takes for the various types availability issues is defined in the base configuration, but can be changed. For information on configuring the default behavior, see “Configuring Product Model Availability Checks” on page 108.

Types of Availability Issues

In general, PolicyCenter handles product model availability for both missing items and unavailable items.

- Missing items are items that one expects to be on the policy, but are not.
- Unavailable items are items whose availability has changed.

The following table lists how PolicyCenter handles various availability issues.

Availability issue	Description
Missing Required Coverage	A coverage that is available and required is not present on the appropriate `Coverable` entity. Fixing this issue adds the coverage.
Missing Suggested Coverage	A coverage that is available and suggested is not present on the appropriate `Coverable` entity. Fixing this issue adds the coverage. The default behavior is to neither fix nor display this issue. However, you can configure PolicyCenter to provide user notification for this issue.
Unavailable Coverage	A coverage is present that is currently unavailable. Fixing this issue removes the coverage.
Missing Coverage Term	A `CovTerm` on a coverage is available but not present. Fixing this issue adds the term.
Unavailable Coverage Term	A `CovTerm` is present on a coverage but is currently unavailable. Fixing this issue removes the term.
Unavailable Option Value	An option is selected for an `OptionCovTerm` that is currently unavailable. Fixing this issue resets the term to its default value, or to `null` if no default has been specified.
Unavailable Package Value	A package is selected for a `PackageCovTerm` that is currently unavailable. Fixing this issue resets the term to its default value, or to null if no default has been specified.
Missing Modifier	A modifier is available but not present. Fixing this issue adds the modifier.
Unavailable Modifier	A modifier is present that is currently unavailable. Fixing this issue removes the modifier.
Missing RateFactor	A rate factor is available but not present. Fixing this issue adds the rate factor.
Unavailable RateFactor	A rate factor is present that is currently unavailable. Fixing this issue removes the rate factor.
Missing Question	A question that is currently available does not have an associated answer. Fixing the issue adds an answer object for that question.
Unavailable Question	An answer is present for a question that is currently unavailable. Fixing this issue removes the answer object for that question.

Note: In cases where an coverage pattern contains other coverage patterns, an issue with a parent does not cause PolicyCenter to report potential issues with a descendent. For example, if a required `Coverage` is missing, PolicyCenter does not report any issues with the coverage’s `CovTerm` entities.

Product Model Issue Matrix

The following issue matrix describes the general behavior of product model issues across all lines of business.

The configuration column contains the corresponding Gosu properties defined in `ProductModelSyncIssueWrapper.gs`.

Category	Configuration	Description
Fix Step	Fix during wizard step sync. Generally called as part of entering a page or after making certain types of changes. `ShouldFixDuringNormalSync`	PolicyCenter determines product model availability for the listed item each time you move between wizard steps. If this setting is true, PolicyCenter attempts to correct the problem before moving to the next wizard step.
Fix Quote	Fix during quote `ShouldFixDuringQuote`	PolicyCenter determines product model availability for the listed item after you click **Quote**. If this setting is true, PolicyCenter attempts to correct the problem before quoting the policy.
Display Step	Display during wizard step sync `ShouldDisplayDuringNormalSync`	If this setting is true, PolicyCenter displays information about a problem it discovered as you move between wizard steps.
Display Quote	Display during quote `ShouldDisplayDuringQuote`	If this setting is true, PolicyCenter displays information about a problem it discovered as you attempt to quote a policy.
Severity Step	Severity during wizard step sync `Severity`	If Display Step is true, this value sets the severity level of the message. There are three severity levels: • `Info` - Provides details of the issue only and any possible steps that PolicyCenter took to correct the issue. • `Warning` - Provides details of the issue in the worksheet at the bottom of the screen. You must explicitly cancel the warning message before continuing, but you need not correct the problem at this point. • `Error` - Provides details of the issue in the worksheet (at the bottom of the screen). You must explicitly correct the error condition and cancel the error message before continuing. You cannot continue until you correct the problem.
Severity Quote	Severity during quote `ShouldBlockQuote`	If Display Quote is true, this value sets the severity level of the message. It uses the same severity levels as Severity Step.

Default Issue Matrix Configuration

Issue	Fix Step	Fix Quote	Display Step	Display Quote	Severity Step	Severity Quote
Missing Required Coverage	true	false	true	true	Warning	Error
Missing Suggested Coverage	false	false	false	false	N/A	N/A
Unavailable Coverage	true	true	true	true	Warning	Warning
Missing Coverage Term	true	false	true	true	Info	Error
Unavailable Coverage Term	true	true	true	true	Info	Info
Unavailable Package Value	true	false	true	true	Warning	Error
Unavailable Option Value	true	false	true	true	Warning	Error
Missing Modifier	true	true	false	false	N/A	N/A
Unavailable Modifier	true	true	true	true	Info	Info
Missing Rate Factor	true	true	false	false	N/A	N/A

Issue	Fix Step	Fix Quote	Display Step	Display Quote	Severity Step	Severity Quote
Unavailable Rate Factor	true	true	true	true	Info	Info
Missing Question	true	Pre-Qual = false All others = true	false	false	N/A	N/A
Unavailable Question	true	Pre-Qual = false All others = true	false	false	N/A	N/A

To illustrate, if the missing item is a coverage term, in the base configuration the issue matrix shows the following values:

- Fix Wizard Step – true
- Fix Quote – false
- Display Wizard Step – true
- Display Quote – true
- Severity Wizard Step – Info
- Severity Quote – Error

Therefore, if a coverage term does not exist on a coverage that has been selected, PolicyCenter has the following behavior:

- If you move from a wizard step to the wizard step that displays the coverage, PolicyCenter corrects the problem and displays a corresponding information message.
- If you attempt to quote the policy with the required coverage term missing, PolicyCenter opens a worksheet at the bottom of the screen and indicates which coverage term is missing. It does not complete the quote until you add the required coverage term.

Configuring Product Model Availability Checks

To configure product model availability checking, you use the Gosu `ProductModelSynchIssuesHandler` class in package `gw.web.productmodel` as the initial entry point. The class contains methods that can force availability checking for the following:

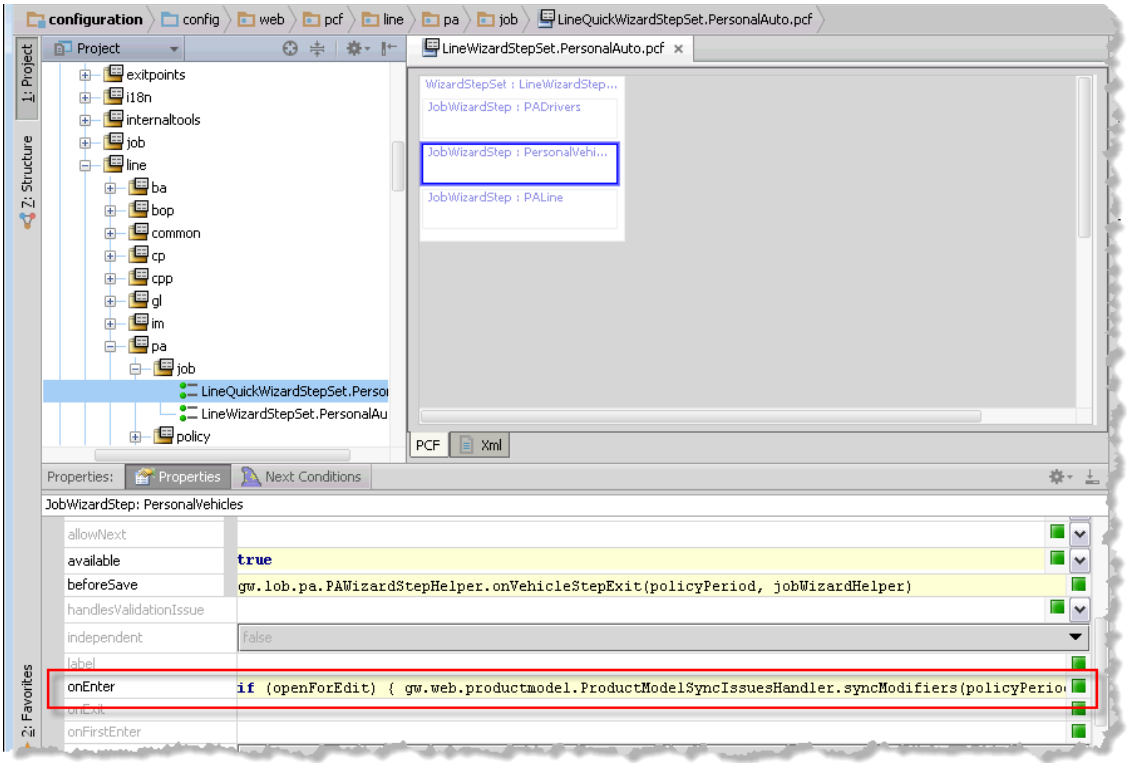
- Coverages
- Modifiers
- Question answers
- Coverables (for whichever type you pass in)

IMPORTANT Product model synchronization can significantly impact performance, particularly if complicated availability scripts are used in the product model. To minimize performance impact, synchronize the smallest possible portion of the product model. For example, do not synchronize modifiers if the screen does not display modifiers. Do not synchronize all coverages if the screen displays only coverages from particular categories. Be aware that you may need to display a smaller set of product model data on a single page to attain adequate performance. Also, do not call synchronization more than necessary. For example, avoid calling synchronization every time the user modifies a field or executes other actions.

You can call the `ProductModelSynchIssuesHandler` class from any Gosu code. Comments in the class provide details on how to use the various class methods. However, you typically call it from a PCF page. For example, you can instruct PolicyCenter to perform a product model availability check before entering the **Vehicles** screen of the Personal Auto line. To trigger a product model availably check in this way, enter the following code on the `onEnter` attribute on the `LineWizardStepSet.PersonalAuto` PCF page.

```
if (rev.OpenForEdit) {
  gw.web.productmodel.ProductModelSyncIssuesHandler.syncModifiers(rev.PersonalAutoLine.AllModifiables,
          jobWizardHelper)
}
```

The following illustration shows how this availability check is configured in the base configuration.



The `ProductModelSynchIssuesHandler` class is a utility class whose purpose is to call other, more specific, classes to handle details of product model availability checking. For example, the call to `ProductModelSyncIssuesHandler.syncModifiers()` in turn calls class `MissingModifierIssueWrapper`, which defines the following behavior:

```
override property get ShouldFixDuringNormalSync()     : boolean { return true }
override property get ShouldDisplayDuringNormalSync() : boolean { return false }
override property get ShouldFixDuringQuote()          : boolean { return true }
override property get ShouldDisplayDuringQuote()      : boolean { return false }
```

To change the default behavior, modify the appropriate value accordingly. For example, if PolicyCenter encounters a missing modifier during availability checking, the default behavior requires that PolicyCenter must correct the issue as it moves between wizard steps. The following code ensures that missing modifiers must be fixed:

```
override property get ShouldFixDuringNormalSync() : boolean { return true }
```

To change this behavior so that the issue need not be corrected at that point in the wizard, set the `return` value to `false`.

Important Classes and Methods Related to Availability Checking

Product model availability checking uses the following classes and methods:

Gosu Classes

- *availabilityissue*IssueWrapper

  Set of Gosu classes in gw.web.productmodel that provide the initial entry point to product model availability checking.

- ProductModelSyncIssuesHandler

  Gosu class in gw.web.productmodel that contains helper methods for synchronizing coverages, modifiers, and questions, and for displaying the results in PolicyCenter.

Gosu Class Methods

- ProductModelSyncIssueWrapper.wrapIssue

  Gosu method that wraps a passed-in ProductModelSyncIssue object in the appropriate ProductModelSyncIssueWrapper class.

Gosu Entity Methods

- AnswerContainer.checkAnswersAgainstProductModel

  Entity method that checks all questions and answers to identify any missing or unavailable questions, and reports them as a list of ProductModelSyncIssue objects. This method does not take any action on these issues.

- AnswerContainer.syncQuestions

  Entity method that calls method checkAnswersAgainstProductModel and wraps the resultant issue objects in ProductModelSyncIssueWrapper classes. It then fixes any issues that must be fixed during standard availability checking and returns the wrappers.

- Coverable.checkCoveragesAgainstProductModel

  Entity method that checks for all coverage, coverage term, and option/package availability issues and reports them as a list of ProductModelSyncIssue objects. This method does not take any action on these issues.

- Coverable.createCoverages

  Entity method that creates the initial set of required and suggested coverages on a Coverable. It throws an exception if the coverages have not been created.

- Coverable.createOrSyncCoverages

  Entity method that calls either the createCoverages if the initial set of coverages has not been created, or syncCoverages if the initial set of coverages already exist.

- Coverable.syncCoverages

  Entity method that checks coverage issues, wraps them in ProductModelSyncIssueWrapper classes, fixes any issues that must be fixed as a result of standard availability checking, and returns the wrappers.

- Modifiable.syncModifiers

  Entity method that calls the updateModifiers method and wraps all the issues in IssueWrapper classes.

- Modifiable.updateModifiers

  Entity method that checks for all modifier and rate factor availability issues, fixes them, and reports them as a list of ProductModelSyncIssues.

chapter 10

Preventing Illegal Product Model Changes

This topic discusses how PolicyCenter verifies that an updated product model does not make illegal changes to the existing product model that is already present on a production server.

This topic includes:

PolicyCenter Product Model Verification

During production mode server startup, PolicyCenter performs the following verifications:

Verification type	Description
1. XML verification	Verifies the XML used product model definition files is both well-formed and valid as defined by the PolicyCenter XML schema.
2. Product model verification	Verifies that no changes in the product model being loaded violate the product model currently in place on the production server. PolicyCenter also performs these checks during product model synchronization with a linked, running development server. See "Verifying the Product Model" on page 117 for more information on verification checking.
3. Illegal product model modification	Verifies that the product model does not contain an illegal modification. For example, it verifies that a required coverage pattern has not been deleted. See "Product Model Immutable Field Verification" on page 112 for details.

If any of these verifications fails, PolicyCenter adds a message to the server log and fails to start.

Product Model Immutable Field Verification

After you move your product model definition to the production server, you must not change it randomly. Unintended changes can—and most likely will—corrupt existing policies. For example, consider the effects of removing a coverage pattern while there is a bound policy that uses that coverage pattern.

It is not important if you change certain fields on the product model. For example, fields such as `Name`, `Description`, or `Priority` can be changed without any bad effects. However, it is extremely important to guard against changes to fields that define the *scaffolding* of the product model. Fields that define scaffolding are fields that define, for example, the coverage terms on a coverage pattern. Guidewire refers to these types of field as being *immutable*, meaning that after you set them you must never change them. Guidewire also uses the term *locked field* interchangeably with the term *immutable field*.

To prevent changes these fields, PolicyCenter verifies the set of immutable fields when you start the server. It checks both for missing items—a missing coverage pattern, for example—and for immutable fields that have been changed. If verification fails, PolicyCenter adds a message to the server log and fails to start.

Locked Entity Fields

PolicyCenter stores information about immutable fields in the production database. During server startup, it reads this information and compares it with the product model definition files stored, or newly loaded, in its local file system. The production locking table contains one row for every product model entity. For example, it contains a row for each coverage, a row for each policy line, and so forth. A separate table stores the locked field name and the locked value. These name-value pairs represent a `Name` and an `EntityPublicID` for a field on the referenced entity.

For example:

```
EntityType
    FieldName 1: FieldValue 1
    FieldName 2: FieldValue 2
    ...
    FieldName n: FieldValue n
```

To illustrate with a more concrete example:

```
CoveragePattern
    PublicID: PADeathCov
    RefCode: 3402
    CoverageSubtype: PersonalAutoCov
    OwningEntityType: PersonalAutoLine
    CoverageCategory: PAPLiabGrp
    ...
```

In this coverage, the `PublicID` and `RefCode` are locked along with the `CoverageSubtype`, `CoverageCategory`, and the fact that this is a line-level coverage.

Locked Array Elements

PolicyCenter also stores any required array elements in the production locking table. The table contains a row for each array element. Therefore, if the array has multiple elements, the table has a row for each element. To illustrate, suppose a particular `CoverageSymbolGroup` entity named `CovSymbGrp` has an array of coverage symbol entities named `CovSymbs` associated with it. In this example, `CovSymbs` has a field with the name `covsymbgrp` that points back to `CovSymbGrp`. In the production locking table, PolicyCenter stores locking table entries as

```
CovSymbs, covsymbgrp : CovSymbGrp, ...
```

This group of locking table entries indicates that there must be an entity of type `CovSymbs` and that the field `covsymbgrp` must have the value of its parent, `CovSymbGrp`. If `CovSymbs` is deleted, PolicyCenter immediately identifies the deletion.

Verifying Locked Fields and Arrays

After PolicyCenter populates the locking table, you can verify whether you have made any illegal edits. During server startup, after loading the product model, PolicyCenter iterates over all the rows in the locking table and tries to find a match for each individual row.

There are two possible scenarios.

- **The entity has a Public ID** – If the entity has a `PublicID`, that `PublicID` always locked. If PolicyCenter does not allow deletions for this entity, PolicyCenter searches the locking table for a product model entity of the same type and with the same `PublicID`. The possible outcomes are:
  - PolicyCenter cannot find a match in the locking table and assumes that the entity was deleted. PolicyCenter adds a message to the server log and fails to start.
  - PolicyCenter finds a match in the locking table and compares all the locked fields against the product model entity. If this comparison fails, PolicyCenter adds a message to the server log and fails to start. In the error case, PolicyCenter issues specific error messages such as: **You changed field *A* on entity *B* to value *C*. Please set it back to *A* .**
- **The entity does not have a Public ID** – The entity has no `PublicID`. A very small subset of the product model entities—approximately eight—do not have a `PublicID`. If deletions for this entity are not allowed, PolicyCenter attempts to find a complete match for the product model entity in the locking table. If it does not find a complete match, PolicyCenter adds an `Entity Deleted` or `Modified` error to the server log and fails to start.

Adding new Entities

You can add new entities to the product model at any time. However, after you add a new entity to the product model on the production server, PolicyCenter locks the entity and does not allow further changes. During server startup, if PolicyCenter finds an entity for which there are no corresponding rows in the locking table, it assumes that the entity is new. If so, it iterates across the entity definition and inserts rows into the locking table as appropriate. During the subsequent production server startups, PolicyCenter verifies its locally stored product model against the product model it is loading.

Product Model Modification Checks

When you start a PolicyCenter server in production mode, it performs the following modification checks on the current product model:

- Deleted Pattern Checks
- Entity Instance Modified Field Checks
- Additional Checks

Deleted Pattern Checks

During production mode server startup, PolicyCenter determines whether any instances of specific product model patterns were deleted when compared to the last server startup. PolicyCenter does this by checking for a `PublicID` in the current product model that matches one in the previous product model.

For most patterns, PolicyCenter does not permit deletions from the product model. For a very small subset of these patterns, it allows deletions. The disallowed and allowed deletions are summarized in the following tables.

Disallowed Deletions

Deleting instances of the following product model patterns is not allowed. If it detects deletions of any of these patterns, PolicyCenter adds a message to the server log and fails to start.

• CoverageCategory	• FinalAuditPattern	• PolicyLinePattern
• CoveragePattern	• FormPattern	• Product
• CoverageSymbolGroupPattern	• GenericCovTermPattern	• Question
• CoverageSymbolPattern	• ModifierPattern	• QuestionChoice
• CovTermOpt	• OfficialIdPattern	• QuestionSet
• CovTermPack	• OptionCovTermPattern	• RateFactorPattern
• DirectCovTermPattern	• PackageCovTermPattern	• TypekeyCovTermPattern

Allowed Deletions

Instances of the following product model patterns can be deleted without interfering with production server startup.

• FormPatternLookup	• ModifierMinMaxLookup
• QuestionSetFilter	• RateFactorMixMaxLookup

Entity Instance Modified Field Checks

During production mode server startup, PolicyCenter determines whether any of the following immutable fields on any of the following entities have been modified when compared to the last server startup. If it detects a modified field, PolicyCenter adds a message to the server log and fails to start.

Entity instance	Immutable field
CoverageCategory	• PolicyLinePattern
CoveragePattern	• OwningEntityType • CoverageSymbolGroupPattern • CoverageSubtype
CoverageSymbolGroupPattern	• PolicyLinePattern
CoverageSymbolPattern	• CoverageSymbolGroupPattern • CoverageSymbolType
CovTermOpt	• CovTermPattern • Value • OptionCode
CovTermPack	• CovTermPattern • PackageCode
DirectCovTermPattern	• CoveragePattern • CoverageColumn • ValueType
GenericCovTermPattern	• CoveragePattern • CoverageColumn
ModifierPattern	• TypeList • ModifierSubtype • ModifierDataType • SplitOnAnniversary • PolicyLinePattern • ScheduleRate
OfficialIdPattern	• PolicyLinePattern • Interstate • OfficialIDType • Scope
OptionCovTermPattern	• ValueType • CoverageColumn • CoveragePattern

Entity instance	Immutable field
PackageCovTermPattern	• CoverageColumn • CoveragePattern
PolicyLinePattern	• PolicyLineSubtype
Product	• Abbreviation • ProductAccountType
Question	• QuestionSet • QuestionType • CorrectAnswer
QuestionChoice	• Question • ChoiceCode
QuestionSet	• QuestionSetType
RateFactorPattern	• ModifierPattern • RateFactorType
TypekeyCovTermPattern	• CoverageColumn • CoveragePattern • Typefilter • Typelist

Additional Checks

During production server startup, PolicyCenter performs the following checks in addition to all of the checks previously described:

- Each product definition contains:
  - `AvailablePolicyTerm` instances with the same `Termtype` values previously loaded.
  - `DocTemplateRef` instances with the same `DocumentTemplateType` values as previously loaded.
  - `ProductPolicyLinePattern` instances with the same `PolicyLinePattern` values as previously loaded.
  - `ProductQuestionSetPattern` instances with the same `QuestionSet` values as previously loaded.
- Each `CovTermPack` definition contains `PackageTerm` instances with the same `Name` and `Value` values as previously loaded.
- Each `Question` definition contains `QuestionFilter` instances with the same `FilterQuestion` value as previously loaded.

If any of these checks fail, PolicyCenter adds a message to the server log and fails to start.

chapter 11

Verifying the Product Model

This topic discusses how PolicyCenter verifies that you have correctly implemented the product model.

This topic includes:

What Is Product Model Verification?

The product model is the part of PolicyCenter configuration that determines the types of policies that you can create. It also determines the coverages, coverage terms, conditions, exclusions, and other objects that are available in each policy. As you customize the product model, it is critical that you adhere to certain product model principles so that the product model remains internally consistent. Adhering to product model principles includes, for example, such rules as the following:

No Product may contain multiple PolicyLinePattern entities of the same type.

To help you make valid changes, Product Designer enables you to validate as much of the product model as possible as you work. All product model pages in Product Designer have a **Validate** link that validates the changes on the current page. Additionally, both the **Commit All** and **Validate Only** links on the **Changes** page perform validation on all of your changes before they are committed to PolicyCenter. However, these validations are only a subset of a complete product model verification.

To perform thorough product model verification and ensure that the product model is consistent and valid, PolicyCenter verifies the product model every time the server starts. Successful product model verification requires the product model to meet a specific list of criteria. PolicyCenter persists the product model through a set of XML configuration files, and checks the content of these XML files under the following circumstances:

- **During server startup** – PolicyCenter performs verification automatically when the application server starts. If it finds an error, it adds a message to the system console and fails to start.

- **During a compile operation from Studio** – PolicyCenter performs verification if you compile one or more product model files in Studio. PolicyCenter displays any errors in the **Log** window at the bottom of Studio.

  Note: Policy validation checks are different from product model verification checks. For information on policy validation, see "Performing Class-Based Validation" on page 69 in the *Rules Guide* and "Performing Rule-based Validation" on page 87 in the *Rules Guide*.

See also

- "Preventing Illegal Product Model Changes" on page 111
- "Checking Product Model Availability" on page 105

Product Model Error Messages

Whenever the product model verification process fails, PolicyCenter stops the process and issues a set of error messages in the server log, as described in the following topics

Product Model Errors During Server Startup

If the product model verification process detects an error during application server startup, it adds error messages to the server log and fails to start.

```
Buildfile: build.xml
verify-checksum:
     [java] [platform] : Calculating module checksum...
     ...
dev-deploy:
[copy-javascript] JavaScript files are up to date - nothing to copy
...
dev-start:
     [java] 2014-11-13 09:35:22,843 INFO Root directory is
             "C:\Guidewire\workspace\pc800\webapps\pc"
     [java] 2014-11-13 09:35:22,843 INFO Temp directory is
             "C:\Documents and Settings\test\Local Settings\Temp\work\Jetty_0_0_0_0_8180__pc"
     [java] 2014-11-13 09:35:22,843 INFO Logging properties found at
             "C:\Guidewire\workspace\pc800\modules\configuration\config\logging\
           logging.properties"
     [java] 2014-11-13 09:35:23,109 INFO ***** PolicyCenter 8.0.0.xxx starting *****
     ...
...
[java] test 2014-11-13 09:35:43,384 INFO Starting parse security config...
     [java] test 2014-11-13 09:35:43,431 INFO Finished parse security config.
  ...
[java] test 2014-11-13 09:35:44,228 INFO System Tables are up to date
     [java] test 2014-11-13 09:35:44,228 INFO ProductModel extending Gosu typesystem...
     [java] test 2014-11-13 09:35:44,228 INFO ProductModel typesystem extended.
     [java] test 2014-11-13 09:35:44,228 INFO Verifying Product Model...
     [java] test 2014-11-13 09:35:46,915 ERROR Errors detected in Product Model:
     [java] test 2014-11-13 09:35:46,915 ERROR Personal Auto [PolicyLinePattern] : -
             Errors in PolicyLinePattern "PersonalAutoLine"
     [java] [1] test [ModifierPattern] : - Errors in ModifierPattern "Rtest"
     [java] [1] gw.api.productmodel.ModifierMinMaxLookup@1944379 [ModifierMinMaxLookup] : -
             Errors in ModifierMinMaxLookup
     [java] [1] gw.api.productmodel.ModifierMinMaxLookup@1944379 [field : Minimum] : -
              ModifierMinMaxLookup.Minimum is required, but is empty
     [java] [2] gw.api.productmodel.ModifierMinMaxLookup@1944379 [field : Maximum] : -
             ModifierMinMaxLookup.Maximum is required, but is empty
...
[java] test 2014-11-13 09:35:46,915 ERROR An exception was thrown while starting a component.
        Setting runlevel to SHUTDOWN [gw.api.webservice.exception.ServerStateException: Errors detected
        in Product Model. First correct the errors and then retry startup.]
[java] java.lang.RuntimeException: gw.api.webservice.exception.ServerStateException: Errors detected
        inProduct Model. First correct the errors and then retry startup.
...
```

Product Model Errors during a Studio Compile Operation

To verify the product model from Studio

1. Make sure all Product Designer users have committed their change lists.
2. In the **Project** window in Studio, navigate to **configuration** → **config** → **Metadata** → **Entity.**
3. Right-click the **Entity** node and select **Compile 'metadata.entity'** from the pop-up menu.
4. Wait for the operation to finish, then check for errors in the **Log** window at the bottom of the Studio window.

Note: Rather than verifying the entire product model, you can verify individual product model files by selecting and compiling a single file. In this case, Studio limits verification to the scope of the selected file. Alternatively, you can rebuild the entire project by selecting **Build** → **Rebuild Project**. In this case, Studio verifies the entire product model as well as the integrity of all other files in PolicyCenter.

Product Model Verification Checks

Guidewire divides the verification checks into the following categories:

Category	Description
Product Model	Constraints on product model entities only, independent of any policies that may use them.
Policy	Constraints on policy entities only, independent of the product model.
Syntactic	Low-level dependencies between policies and the product model that must be enforced for all policies.
Semantic	Business-level constraints between policies and the product model.

Note: The verification checks are built into PolicyCenter. You cannot change or modify them.

Product, Policy, and PolicyLine Verification

During production product model verification, PolicyCenter runs the following checks on each `Product`, `Policy`, and `PolicyLine` in the active product model definition:

Verification category	Description
Product Model	The `PolicyLineSubtype` is defined on every `PolicyLinePattern`.
Product Model	No `Product` contains multiple `PolicyLinePattern` entities of the same type.
Syntactic	The type of each `PolicyLine` agrees with its pattern's `PolicyLineSubtype`.
Syntactic	No `PolicyPeriod` contains multiple lines of the same type.
Syntactic	Each `PolicyLinePattern` is part of the `Product` referenced by its `Policy`.

Coverage Verification

During production product model verification, PolicyCenter runs the following checks on each `Coverage` in the current product model definition:

Verification category	Description
Syntactic	All `Coverage` entities on a policy line map to `CoveragePattern` entities associated with the line's `PolicyLinePattern`.
Syntactic	Each `Coverable` has no two `Coverage` entities with the same `CoveragePattern`.
Product model	The `CoveragePattern.Subtype` field must name a coverage of the appropriate subtype for its owning entity.

CoverageTerm Verification

During production product model verification, PolicyCenter runs the following checks on each `CoverageTerm` in the current product model definition:

Verification category	Description
Semantic	The value of every `DirectCovTerm` is in bounds.
Product Model	Every `OptionCovTermPattern` has at least one `CovTermOpt`.
Product Model	Every `PackageCovTermPattern` has at least one `CovTermPack`.
Product Model	Every `CovTermPack` has at least one `PackageTerm`.
Semantic	The value of every required `OptionCovTerm` is non-null.
Semantic	The value of every required `PackageCovTerm` is non-null.
Semantic	The value of every required `DirectCovTerm` is non-null.

Modifier Verification

During production product model verification, PolicyCenter runs the following checks on each `Modifier` in the current product model definition:

Verification category	Description
Syntactic	All `Modifier` entities on a line are associated to a `ModifierPattern` in the line's `PolicyLinePattern`.
Semantic	`RateModifier` entities have a rate value at least that of the `ModifierPattern` entity's specified lower bound, if any.
Semantic	`RateModifier` entities have a rate value at most that of the `ModifierPattern` entity's specified upper bound, if any.
Product Model	Only rate modifiers can be schedule rates.
Product Model	Only schedule rate modifier patterns have rate factor patterns.
Syntactic	Only schedule rates have rate factors.
Product Model	The `RateFactorPattern` entities for any given `ModifierPattern` all have distinct types.
Syntactic	All `RateFactor` entities on a schedule rate are instantiated from a `RateFactorPattern` associated with the `ModifierPattern`.
Semantic	`RateFactor` entities have a rate value at least that of the `RateFactorPattern` entity's specified lower bound.
Semantic	`RateFactor` entities have a rate value at most that of the `RateFactorPattern` entity's specified upper bound.
Product Model	The `ModifierPattern` entities within any `PolicyLinePattern` all have unique `RefCode` values.
Syntactic	No `PolicyLine` contains two `Modifier` entities of the same type in the same jurisdiction with overlapping effective dates.
Product Model	Each `RateFactorPattern` contains at most one `RateFactorMinMaxLookup` for each jurisdiction.
Product Model	Every `ModifierPattern` entity's default minimum/maximum interval is non-empty
Product Model	Every `ModifierMinMaxLookup` entity's minimum/maximum interval is non-empty.
Product Model	Every `RateFactorPattern` entity's default minimum/maximum interval is non-empty.
Product Model	Every `RateFactorMinMaxLookup` entity's minimum/maximum interval is non-empty.

CoverageSymbol Verification

During production product model verification, PolicyCenter runs the following checks on each `CoverageSymbol` in the current product model definition:

Verification category	Description
Product Model	The `CoverageSymbolGroupPattern` referenced by a `CoveragePattern` must be contained in the same `PolicyLinePattern` as the `CoveragePattern`.
Product Model	The `CoverageSymbolPattern` entities in a `CoverageSymbolGroupPattern` have distinct symbol types.
Syntactic	All `CoverageSymbolGroup` entities in a given line are instantiated from `CoverageSymbolGroupPattern` entities in the line's `PolicyLinePattern`.
Syntactic	No two `CoverageSymbolGroup` entities in a given line are instantiated from the same pattern.
Syntactic	All `CoverageSymbol` entities in a given `CoverageSymbolGroup` are instantiated from `CoverageSymbolPattern` entities in the group's `CoverageSymbolGroupPattern`.
Syntactic	No two `CoverageSymbol` entities in a given `CoverageSymbolGroup` are instantiated from the same pattern.
Syntactic	The `Coverage.CoverageSymbolGroup` field is non-null if and only if the `CoveragePattern` has an associated `CoverageSymbolGroupPattern`.
Syntactic	The `CoverageSymbolGroup` referenced by a `Coverage` must be instantiated from the `CoverageSymbolGroupPattern` referenced by the `Coverage` entity's pattern.
Policy	The `CoverageSymbolGroup` referenced by a `Coverage` must be contained in the same `PolicyLine` as the coverage.
Policy	All `Coverage` entities of the same type in the same `PolicyLine` must reference the same `CoverageSymbolGroup`.

chapter 12

Product Model Loader

The Product Model Loader extracts product model data from the running PolicyCenter server, and stores the information in tables in the PolicyCenter database. Policy data combined with the Product Model Loader data can be extracted, transformed, and loaded (ETL) into a data warehouse or data store for analysis or reporting.

This topic includes:

Overview of Product Model Loader

The PolicyCenter database stores policy data for individual policies. However, the policy data does not contain actual values for coverage terms and other product model information. That is, for an individual policy, the policy data includes a coverage term, but not the value of the coverage term. The data that the Product Model Loader creates enables you to discover that this coverage term is a $1,000 collision deductible. Using an ETL process, you can combine the policy data with the Product Model Loader data and store it in a data warehouse or data store for analysis or reporting.

You can access complete product model information in Product Designer or by browsing through policy transactions in the PolicyCenter application user interface. The PolicyCenter database, however, does not store product model information such as the values of coverage terms. Usually, an ETL process works with the PolicyCenter database, not the PolicyCenter application. To facilitate ETL processing, the Product Model Loader creates and stores product model information in PolicyCenter database tables.

The Product Model Loader extracts product model data for all lines of business. In the base configuration, the Product Model Loader extracts the following information from the product model:

- Coverage term and coverage term option information, including type, name, and value. Common examples of coverage terms include limits and deductibles.
- Clause name and clause type (coverage, exclusion, or condition).
- Currency information on coverage term options.
- Modifier name and, if applicable, associated typelist and associated rate factor names.

The Product Model Loader loads database tables with the product model. This product model information enables you to map the data for individual policies to actual names and values in the product model. The Product Model Loader information provides the decoding or mapping of the pattern identifier of, for example, a coverage term option to the name and/or value of that option. Join the policy information (which only contains the identifier) with the product model information to retrieve the name and value of the options selected in the policy.

If you add policy lines or add clauses and coverage terms to existing policy lines, the Product Model Loader creates ETL product model tables for these additions. You can also extend the Product Model Loader to access other product model data, including product offerings or questions in question sets. These changes may require modifying the ETL entities.

The Product Model Loader is implemented as a plugin that runs when PolicyCenter starts. To disable this plugin, see “Non-operational Plugin Implementation” on page 179 in the *Integration Guide*.

See also

- “ETL Product Model Loader Plugin” on page 179 in the *Integration Guide*

ETL Database Tables and Entities

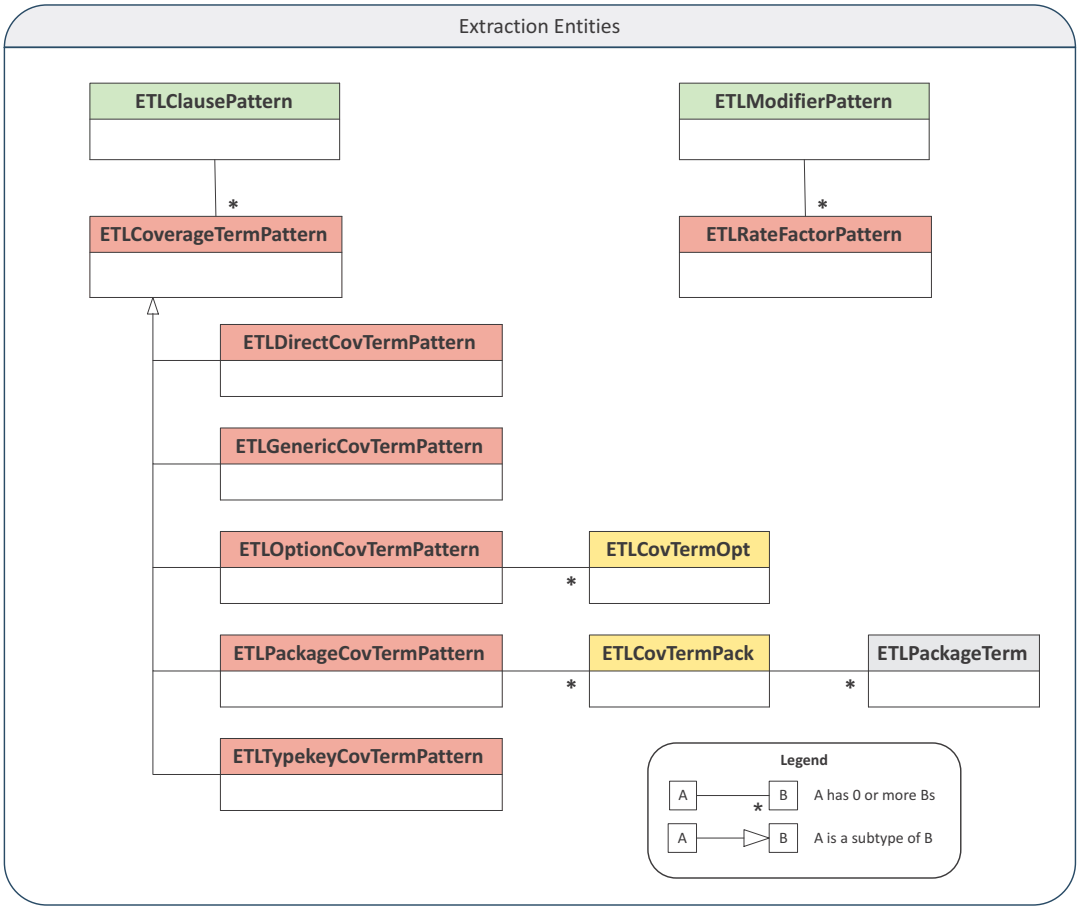
When the Product Model Loader loads the product model information, it first creates entities that contain the product model information, then saves the entity information to database tables. Use the database table names to extract information from the PolicyCenter database. If you modify the Product Model Loader plugin, you may need to use the entity names.

For each type of policy model data, the following table shows the ETL entity and database table.

Policy model data	ETL extraction entity	ETL table name
Clause pattern:	`ETLClausePattern`	`pc_etlclausepattern`
• Coverage pattern		
• Exclusion pattern		
• Condition pattern		
Coverage term pattern	`ETLCoverageTermPattern`	`pc_etlcovtermpattern`
• Direct coverage term pattern	`ETLDirectCovTermPattern`	
• Generic coverage term pattern	`ETLGenericCovTermPattern`	
• Option coverage term pattern	`ETLOptionCovTermPattern`	
• Package coverage term pattern	`ETLPackageCovTermPattern`	
• Typekey coverage term pattern	`ETLTypekeyCovTermPattern`	
Exclusion term pattern	`ETLCoverageTermPattern`	
Condition term pattern		
Coverage term option	`ETLCovTermOpt`	`pc_etlcovtermoption`
Package coverage term option pattern	`ETLCovTermPack`	`pc_etlcovtermpackage`
Package term	`ETLPackageTerm`	`pc_etlpackterm`
Modifier pattern	`ETLModifierPattern`	`pc_etlmodifierpattern`
Rate factor pattern	`ETLRateFactorPattern`	`pc_etlratefactorpattern`

ETL Entities

The following illustration shows the ETL entities that the Product Model Loader creates. See the *Data Dictionary* for complete information. You may need to use these entities if you modify the Product Model Loader plugin.



See also

- "Coverage Pattern Overview" on page 476 in the *Application Guide*
- "Coverage Term Pattern Overview" on page 478 in the *Application Guide*

ETL Database Query Example

You can use database queries to extract product data from policies. The following example queries the PolicyCenter database to extract policy data from Personal Auto policies.

The following SQL statement on the PolicyCenter database returns all instances of Collision (`PACollisionCov`) coverage in Personal Auto.

An instance of Collision coverage is a `PersonalVehicleCov` entity with `PatternCode` of `PACollisionCov`. The `ChoiceTerm1` column contains values like `opt_320` that do not provide actual values for the Collision deductible. Although the Data Dictionary does not show `ChoiceTerm1` on `PersonalVehicleCov`, it is defined in the entity (`PersonalVehicleCov.etx`). The `PersonalVehicleCov` entity is stored in the database in the `pc_personalvehiclecov` table.

The following SQL statement gets the identifier, pattern code, and choice term from all Collision coverages in the database:

```
SELECT
ID, PatternCode, ChoiceTerm1
FROM pc_personalvehiclecov t4
WHERE t4.PatternCode = 'PACollisionCov';
```

The database returns a result set with the following values:

ID	PatternCode	ChoiceTerm1
5000000002	PACollisionCov	opt_319
5000000004	PACollisionCov	opt_320
5000000006	PACollisionCov	opt_321

The `PACollisionCov` pattern code does not convey the name of the coverage. The `opt_319` choice term does not provide name or type of the selected coverage term option.

The Product Model Loader creates database tables that provide access to names of coverages, coverage term names and types, coverage term option names and values, and other product model objects. The tables are in the PolicyCenter database, making the information available to database queries in an ETL process. You can write a query joining the policy data with this product model data. You can store the results of this query in a data warehouse or data store.

The following SQL statement joins the ETL data with the policy data to get the value of the coverage term options. The statement selects all instances of Collision coverage and joins with ETL product model tables. You now have the currency and option values selected for the Collision deductible.

```
SELECT
t4.ID, t4.PatternCode, t3.PatternID, t2.PatternID, t1.Value, t1.Currency
FROM pc_personalvehiclecov t4
JOIN pc_etlclausepattern t3 on t4.PatternCode = t3.PatternID
JOIN pc_etlcovtermpattern t2 on t2.ClausePatternID = t3.ID
JOIN pc_etlcovtermoption t1 on t1.CoverageTermPatternID = t2.ID
AND t1.PatternID = t4.ChoiceTerm1
WHERE t4.PatternCode = 'PACollisionCov';
```

The database returns a result with the following values:

ID	PatternCode	Value	Currency
5000000002	PACollisionCov	250	usd
5000000004	PACollisionCov	500	usd
5000000006	PACollisionCov	1000	usd

part II

Configuring Lines of Business

chapter 13

Adding a New Line of Business

This topic describes how to add a new line of business to PolicyCenter. The base configuration of PolicyCenter includes several reference implementations for lines of business, including personal auto, general liability, and workers' compensation, among others. These lines of business are configurable. If your line of business is not included in the base configuration, this topic describes how to add it to PolicyCenter.

This topic provides general instructions on how to add a new line of business, using homeowners as an example line of business. In general, use the lines of business provided in the base configuration as a guide for developing a new line.

Detailed instructions are provided for creating the `HomeownersLine` entity, its coverage, modifier, and rate factors. Use the `HomeownersLine` entity as an example for creating other coverable entities.

> **Note:** A reference implementation of the homeowners line of business is available as an extension pack. Contact Guidewire support for more information.

This topic includes:

See also

- "Lines of Business" on page 173 in the *Application Guide*

Step 1: Define the Data Model for the New Line of Business

The first step in creating a new line of business is to define the data model. Creating a new line of business is complex, and it needs a well-defined starting point.

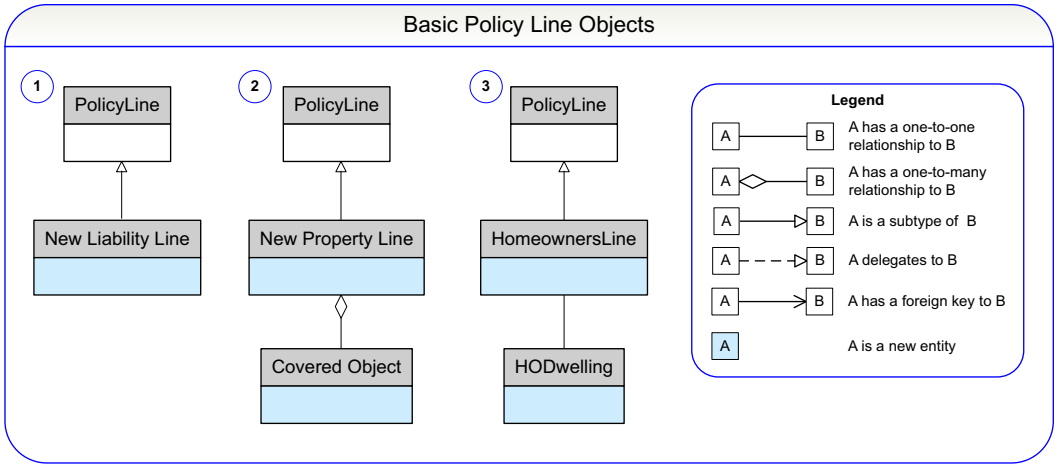
Start by designing the data model. In designing the data model, determine which coverages are for property, which are for liability, and which are for both. Also determine the coverables—the property or persons who are being insured. Creating a data model for a liability-only line is relatively easy, because the only insured object is the line itself. Creating the data model for a new type of property insurance, such as an auto policy, is more complex.

While designing that data model, consider the hierarchy of the data objects. The hierarchy shows relationships between entities including foreign keys, subtypes, and array access to other entities.

Defining the Line

Determine the new entities you need to create for the new line of business. At the top level, you must define the policy line. If there are insured objects, you must define how they relate to the policy line. For example, insured objects might be accessed through arrays off of the policy line, or might be accessed through arrays off of a newly-defined location entity.

The following illustration shows a few examples of basic policy line objects.

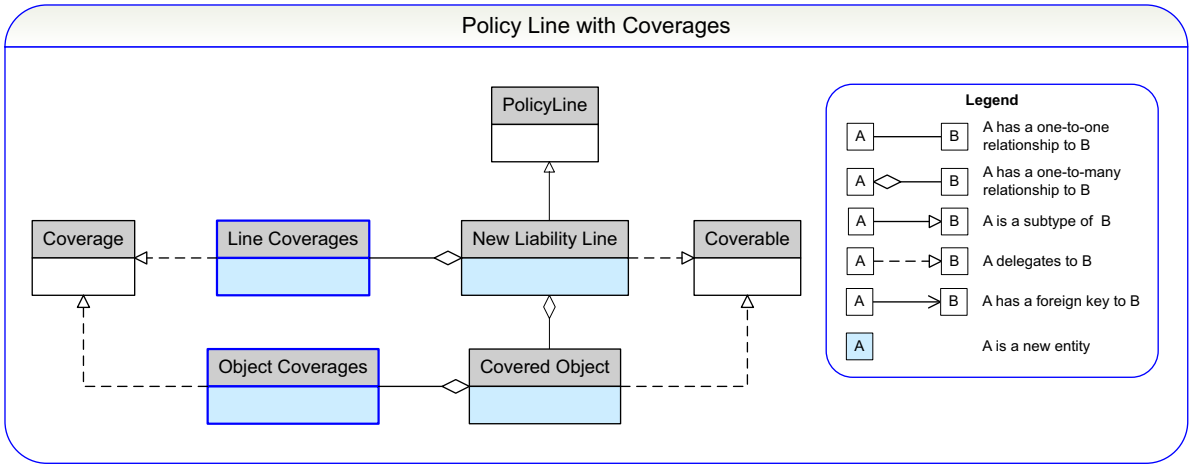


1. A liability line with no insured objects. The entity for the new liability line is a subtype of `PolicyLine`.
2. A simple property line with one or more covered objects that attach directly to the new property line entity, which is a subtype of `PolicyLine`.
3. A specific implementation of case 2, where the new `HomeownersLine` entity is a subtype of `PolicyLine`. The `HODwelling` entity represents an insured object and is directly accessible from the new policy line.

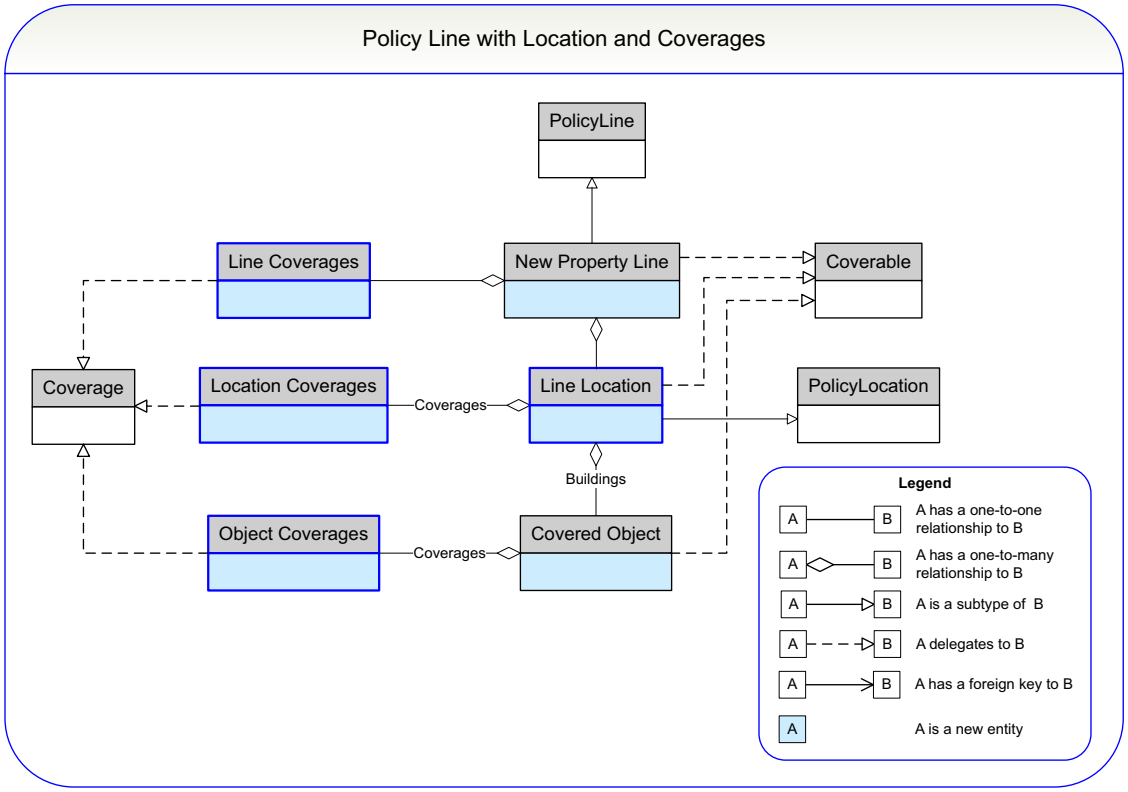
Attaching Coverages

After you define the basic policy line objects, you must determine where to attach coverages. PolicyCenter defines a coverage as a protection from a specific risk. Coverage entities implement the `Coverage` interface. A coverage always attaches to a `Coverable`. A coverable is the covered object, such as a building or a car. For liability coverages, the coverable is the insured. As you create the model for liability coverages, the policy line usually is designated as the coverable that represents the insured. In some cases, coverages attach to jurisdictions. Property coverages attach to a specific coverable object, such as a vehicle, building, or dwelling.

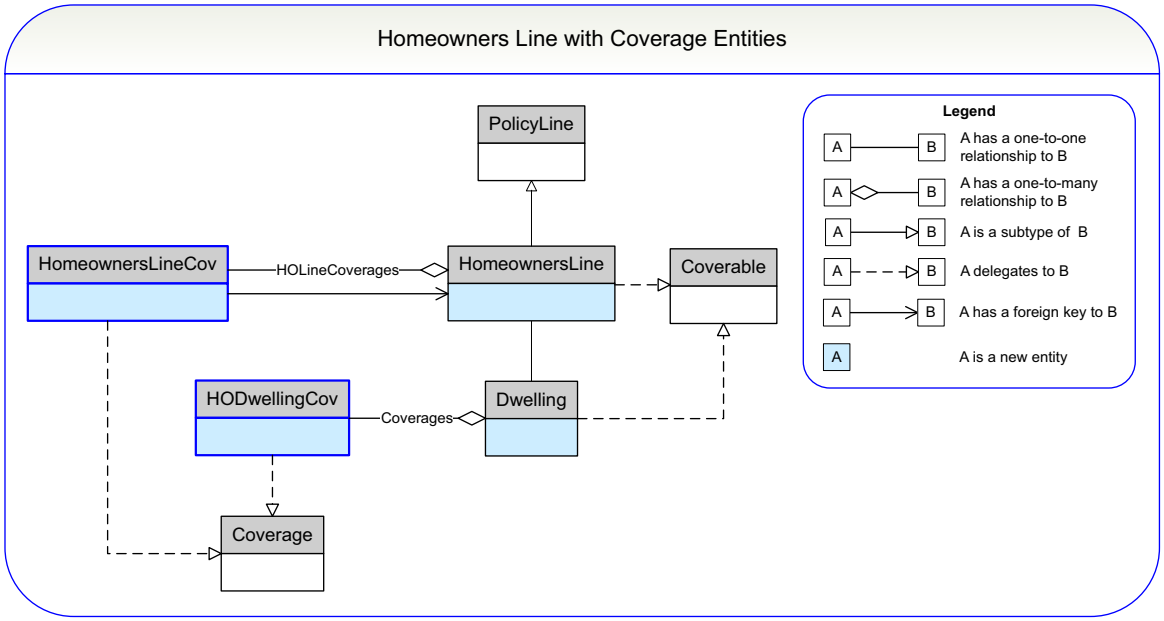
The following illustration shows a liability line with one covered object. Line coverages, usually liability coverages, attach directly to the line. Object, or property, coverages attach to the covered object.



Some coverages are location based. That is, they apply to all the buildings or vehicles at a specific location. In that case, one approach is to create a new location entity as a coverable. In this example, the new location entity is a subtype of `PolicyLocation`. The `PolicyLocation` entity is not a coverable.



The following illustration shows the basic model for the homeowners line use in this example.



Additional Policy Entities

As you follow the steps to develop the line of business, you must add other types of entities to the object model, such as entities for rating and modifiers.

Step 2: Register the New Line of Business

Every new line of business you add to PolicyCenter must be registered by extending the `InstalledPolicyLine` typelist and adding the package name, public ID, and subtype of the policy line. Various parts of the product configuration use one or another of these identifiers.

To register a new line of business

1. Use the Studio **Project** window to navigate to **configuration** → **config** → **extensions**. Right-click the typelist folder, and then select **New** → **Typelist Extension**.
2. In the **Typelist Extension** dialog box, specify the following parameters:

Property	Value
Typelist	`InstalledPolicyLine`
Filename Suffix	`HO`

Studio creates a new typelist extension named `InstalledPolicyLine.HO.ttx`.

3. Add a new `typecode` element to the typelist extension. Specify the following properties:

Property	Value
code	Package name in uppercase, for example, `HO`. This parameter corresponds to the package name that you specify later (in lower case), as explained in "Step 3: Add a Policy Line Package and Configuration Class" on page 133.
name	Public ID of the policy line pattern, for example, `HomeownersLine`. The public ID corresponds to the **Code** of the policy line that you specify later in Product Designer, as explained in "Creating the Policy Line" on page 159. This value cannot contain blanks or special characters and must be unique within a PolicyCenter instance.
desc	Policy line subtype. For example, `HomeownersLine`. This parameter corresponds to the **Entity** parameter you specify later, as explained in "Creating the New Coverable Entities" on page 135.

Note: The PolicyCenter server checks its policy line entities against the parameters in the `InstalledPolicyLine` typelist and refuses to start if it finds a policy line without a corresponding typecode.

Step 3: Add a Policy Line Package and Configuration Class

Every new policy line that you add to PolicyCenter must have its own package containing a `Configuration` Gosu class. The `Configuration` class must extend the `PolicyLineConfiguration` class. In the lines of business supplied in the PolicyCenter base configuration, the line-specific classes override two properties: `RateRoutineConfig` and `AllowedCurrencies`.

To add a policy line package

1. In the Studio **Project** window, navigate to **configuration** → **gsrc** → **gw**.
2. Right-click `lob` and select **New** → **Package**. In the **New Package** dialog box, enter the package name. The package name must be a lower-case duplicate of the **code** specified in "Step 2: Register the New Line of Business" on page 132. For the homeowners example, enter `ho`.

Studio creates a new package named `gw.lob.ho`.

To add a Configuration class

To simplify creating a new Configuration class, copy an existing class from another line of business and make the necessary changes to the code.

1. In the Studio **Project** window, navigate to an existing `Configuration` class file. For example, navigate in **configuration** → **gsrc** to the `gw.lob.gl` package.
2. Right-click the `Configuration` class file. For example, right-click `GLConfiguration`. Then select **Copy**.
3. Navigate back to your new policy line package. For homeowners, navigate in **configuration** → **gsrc** to the `gw.lob` package, and then right-click `ho`. Then select **Paste**.
4. In the **New Name** field of the **Copy Class** dialog box, change the name of the class. Remove the upper-case package name from the source line of business and replace it with the upper-case package name of the new line of business. For example, if you copied the `GLConfiguration` class, for homeowners, change the name to `HOConfiguration`.
5. Verify that the **Destination package** is correct. For homeowners, the destination package is `gw.lob.ho`. When you click **OK**, Studio creates and opens the new class file.
6. Within the `HOConfiguration` class, make any needed changes to the `RateRoutineConfig` property as follows:

- If you are using Guidewire Rating Management, change the `RateRoutineConfig` property to return an appropriate `RateRoutineConfig` class. Examine another line for guidance on implementing the class. For example, examine `PAConfiguration` and `PARateRoutingConfig` and define similar code to implement rating in your new line.
- If you are using the system table rating plugin, `gw.plugin.policyperiod.impl.SysTableRatingPlugin` or a third- party rating solution, set the property to return null.

7. Also within the `HOConfiguration` class, null and change the `AllowedCurrencies` property to specify the correct unlocalized typecode for the policy line. For homeowners, change it to:

```
override property get AllowedCurrencies(): List<Currency> {
    var pattern = PolicyLinePattern.getByCode(InstalledPolicyLine.TC_HO.UnlocalizedName)
    return pattern.AvailableCurrenciesByPriority
```

Note: The PolicyCenter server refuses to start if it finds a policy line without a corresponding *line*`Configuration` class within the policy line package.

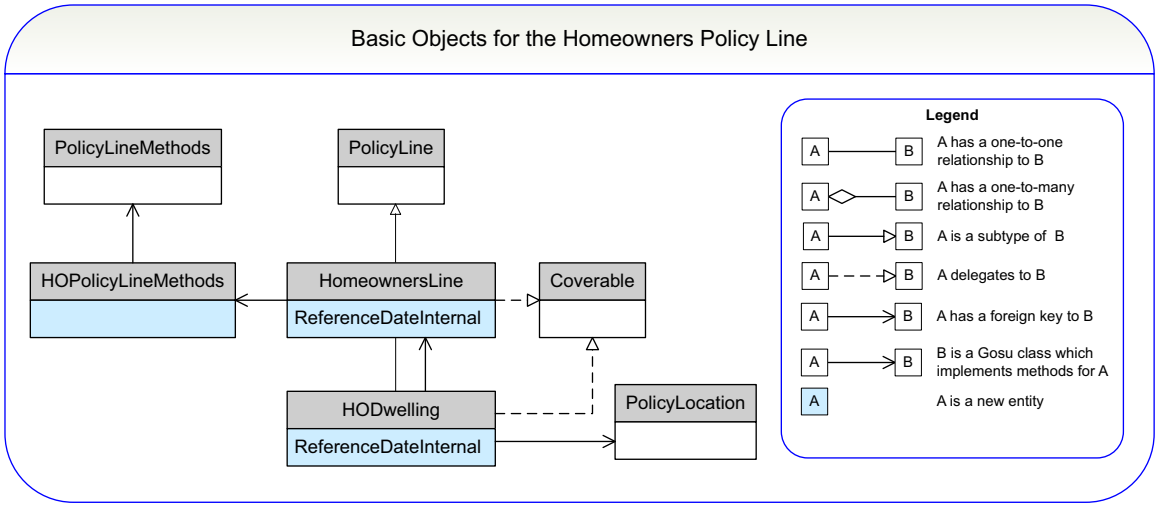
Step 4: Add Coverages to the New Line of Business

This section describes how to add coverages to a new line of business, using the homeowners line as an example. In the example, the `HomeownersLine` entity is a coverable. The following topics provide detailed instructions for creating the `HomeownersLine` entity, its coverages, modifiers, rate factors, and lookup tables.

Creating the Basic Policy Line and Coverable Entities

In "Step 1: Define the Data Model for the New Line of Business" on page 130, you designed the basic policy line and coverable entities and mapped the relationships. This topic provides instructions for creating the policy line and coverable entities. This topic assumes that the policy line is a coverable object. Use these instructions as a guide as you create entities for other coverable objects.

The following illustration shows the basic objects for the homeowners line of business.



This topic includes the following steps:

- Creating the New Coverable Entities
- Creating Policy Line Methods
- Creating the Line Enhancement Methods

Creating the New Coverable Entities

The next step is to create an entity for the new policy line. In this example, the policy line is a coverable object. For the homeowners line of business, the `PolicyLine` subtype is `HomeownersLine`.

> **Note:** For detailed instructions on how to create new entities in “Defining a New Data Entity” on page 214 in the *Configuration Guide*.

To create the new policy line entity

For homeowners, create the `HomeownersLine` entity. These steps create the basic entity definition. Add additional columns and data objects as needed after the verifying the basic definition.

1. In Studio, navigate to **configuration** → **config** → **extensions**. Then right-click **Entity** node and select **New** → **Entity**. In the **Entity** dialog box, enter the following properties:

Property	Value
Entity	This parameter corresponds to the **desc** parameter you specified in “Step 2: Register the New Line of Business” on page 132 when you added a new typelist extension named `InstalledPolicyLine.HO.ttx`. For the homeowners example, enter `HomeownersLine`.
Entity Type	`subtype`
Desc	`Homeowners line of business`
Supertype	`PolicyLine`

2. Add the following child elements to the HomeonwersLine subtype.

   a. If the line is a coverable object, add a ReferenceDateInternal field as a column element. Add the field by adding a new column element with the following properties:

Property	Value
name	ReferenceDateInternal
type	datetime
desc	Internal field stores the reference date of this entity on bound policy periods.

   b. Add an appropriate delegate that implements gw.api.policy.PolicyLineMethods. For homeowners, the delegate is gw.lob.ho.HOPolicyLineMethods. To add the delegate, add an implementsInterface element to the subtype with the following properties:

Property	Value
iface	gw.api.policy.PolicyLineMethods
impl	gw.lob.ho.HOPolicyLineMethods

   For more information, see "<implementsInterface>" on page 194 in the *Configuration Guide*.

   c. Optional. Add additional columns or other elements as needed. For example, as you define the HODwelling entity, add a foreignkey element that points back to the HomeownerLine entity. On the HomeownersLine entity, define a onetoone element that provides access to the HODwelling entity.

If you switch to the **Xml** tab of the Entity editor, the code in the HomeownersLine.eti file appears similar to the following:

```
<?xml version="1.0"?>
<subtype
  xmlns="http://guidewire.com/datamodel"
  desc="Homeowners line of business"
  entity="HomeownersLine"
  supertype="PolicyLine">
  <column
    desc="Internal field for storing the reference date of this entity on bound policy periods"
    name="ReferenceDateInternal"
    nullok="false"
    type="datetime"/>
  <implementsInterface
    iface="gw.api.policy.PolicyLineMethods"
    impl="gw.lob.ho.HOPolicyLineMethods"/>
</subtype>
```

3. Add the other entities needed for the homeowners line, as shown in the preceding object model diagram.

Note: Unlike HomeownersLine, HODwelling is not an entity subtype of another entity. Define HODwelling as an entity rather than as a subtype.

Verifying Your Work

Verify your work in Studio and in PolicyCenter. It is easier to find problems in your implementation if you verify your work often. You can verify your work after the entity definition and classes or other objects referenced by the entity definition exist. You can verify your work now because you have defined the HomeownerLine entity and the HOPOlicyLineMethods interface that it references.

To verify your work

1. Validate the data model by starting the application server. Check for errors in the application console. To avoid database conflicts, drop the database before starting the server:

```
gwpc dev-dropdb dev-start
```

2. Restart Studio to pick up your data model changes. Check for errors in the Studio console.

Creating Policy Line Methods

Within PolicyCenter, each line of business requires a set of policy line methods. For lines of business in the base configuration, these methods are already defined. For a custom policy line, two extra steps are required. The first step is to use a delegate to declare the methods. The second step is to create a Gosu class in Studio that implements the methods. The delegate declaration references the Gosu class. Previously, you defined the policy line element in "Creating the New Coverable Entities" on page 135. At that time, you specified the delegate in the `implementsInterface` element.

Adding the policy line methods is a reasonably simple task. In part, it is simple because the same methods exist for all other lines, including the policy lines included in the base configuration. As you write the code for your policy line, use the code for the existing lines as an example. You can view the existing policy line methods in Studio by navigating to **configuration** → **gsrc** → **gw** → **lob** → *xx* where *xx* is the line abbreviation. For example, the policy line methods for commercial auto are in `gw.lob.ba.BAPolicyLineMethods.gs` located in **configuration** → **gsrc**.

To create the new policy line class

1. In Studio, navigate in **configuration** → **gsrc** to the `gw.lob.`*`line`* package. For homeowners, navigate to the `gw.lob.ho` package.
2. Right-click the `ho` package node that you created in "Step 3: Add a Policy Line Package and Configuration Class" on page 133, and then select **New** → **Gosu Class**. In the **New Gosu Class** dialog box, enter the **class name**. Enter the name from the `impl` attribute of the `implementsInterface` element of the policy line element that you created in "Creating the New Coverable Entities" on page 135. For homeowners, enter `HOPolicyLineMethods`.

   Studio opens the new Gosu class. For homeowners, Studio opens `HOPolicyLineMethods.gs`.
3. Change the class declaration to show that this class extends the policy line methods. For homeowners, modify the class statement as follows:

```
package gw.lob.ho
uses gw.api.policy.AbstractPolicyLineMethodsImpl

class HOPolicyLineMethods extends AbstractPolicyLineMethodsImpl {
  construct() {
  }
}
```

4. Modify the base construct using an existing policy line as an example. For homeowners, modify the base construct as follows:

```
class HOPolicyLineMethods extends PolicyLineMethodsDefaultImpl {
var _line : entity.HomeownersLine

  construct(line : entity.HomeownersLine) {
    _line = line
  }
```

5. Define the `CoveredStates` property, if necessary. For homeowners, define `CoveredStates` as follows:

```
uses java.util.HashSet
override property get CoveredStates() : Jurisdiction[] {
  var covStates : Set<Jurisdiction> = {}
  if (_line.BaseState != null) {
    covStates.add(_line.BaseState)
  }
return covStates.toTypedArray()
}
```

Note: You can revise these properties as needed as you add coverages and other covered objects to the line.

The following is a typical example of a homeowners `gw.lob.ho.HOPolicyLineMethods` class:

```
package gw.lob.ho
```

```
uses gw.api.policy.PolicyLineMethodsDefaultImpl
uses java.util.HashSet

class HOPolicyLineMethods extends PolicyLineMethodsDefaultImpl {
  var _line : entity.HomeownersLine

  construct(line : entity.HomeownersLine) {
    _line = line
  }

  override property get CoveredStates() : Jurisdiction[] {
    var covStates : Set<Jurisdiction> = {}
    if (_line.BaseState != null) {
      covStates.add(_line.BaseState)
    }
    return covStates.toTypedArray()
  }
}
```

Because you have not defined other entities in the line, there are very few methods that you can create at this point.

Note: At this point, you can verify your work. For instructions, see "Verifying Your Work" on page 136.

Creating the Line Enhancement Methods

Depending on the line of business, you can define additional needed properties and methods on the line. Use the `LineEnhancement` classes from other lines to help determine if any methods are needed. You can examine the line enhancements in Studio classes located in the `gsrc.gw.lob.xx` package, where *xx* is the line abbreviation. For example, examine the commercial property `LineEnhancement` class by navigating to `gsrc.gw.lob.cp` and opening `CommercialPropertyLineEnhancement.gsx`.

When you finish, verify your work.

Creating the Coverage Entity for the Coverable Object

The policy line that you created in the previous topic is a coverable. In this topic, you create the `Coverage` entity for the policy line. The policy line is a coverable, and every `Coverable` must have at least one associated `Coverage` entity. Each `Coverage` entity can have zero or more coverage terms. The `Coverage` entity defines the coverage terms in a table. The table limits the number and type of coverage terms that you can add to the coverage. The types of coverage terms are:

- Direct
- Choice
  - Option
  - Package
  - Typekey
- Boolean
- String
- Date

In the base configuration, you can define a `Coverage` entity table with a maximum of:

- Two Direct coverage terms
- Two Option → Package coverage terms
- Two Boolean coverage terms

The base configuration provides no other types. As you add coverages to the product model in Product Designer, the underlying table restricts the number and types of coverage terms that you can add.

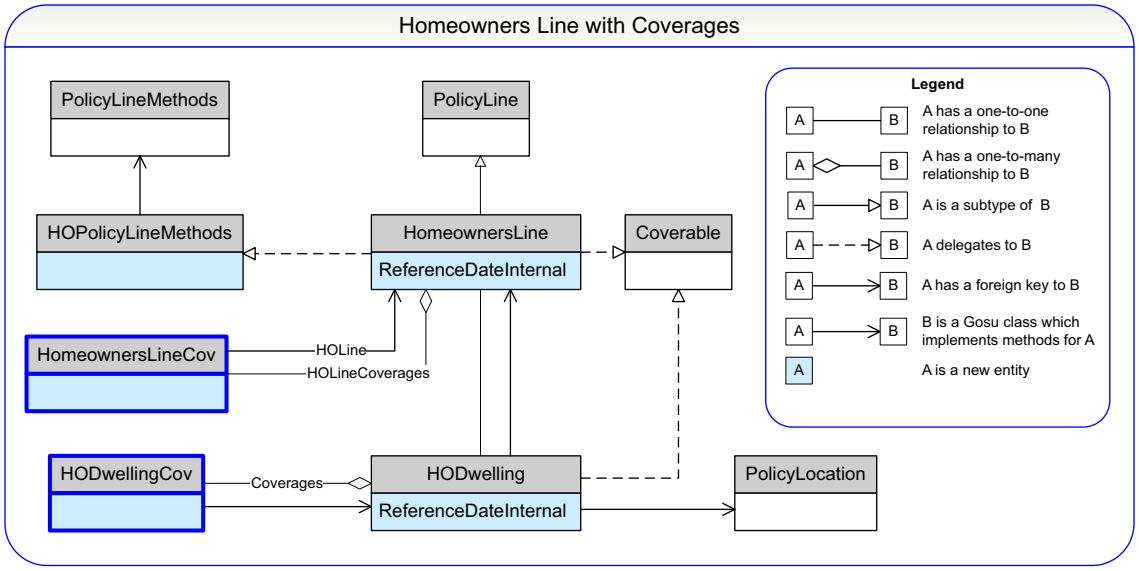
The `CoveragePattern` of a `Coverage` returns the set of coverage terms used by a particular coverage. Each coverage term of each type added to a coverage requires a dedicated column in the database. As you define the coverage, add as many columns as needed for any coverage term that you foresee being used with the coverable.

You define the coverage patterns for a coverage in Product Designer. Most coverages need coverage terms, such as limits, deductibles, retroactive dates, and so forth. You can examine the coverages and coverage terms in the general liability line by navigating to **Policy Lines** → **General Liability Line** → **Coverages** → **Condominiums** → **Terms**. The Condominiums coverage has no coverage terms. Next, select the **Employee Benefits Liability** → **Terms**. The Employee Benefits Liability coverage has several coverage terms. While viewing any term, click **Add** to add more coverage terms. The **Add Term** dialog box enables you to choose from among the coverage term types defined in the table. If the coverage already contains the maximum number of terms of a particular type, the **Column** list is empty and you cannot add a new term of the selected type.

IMPORTANT As you define a `Coverage` entity, provide enough columns of each coverage term type to handle any expected coverage pattern that your coverables require. If a coverable requires a coverage pattern that exceeds these limits, you must add additional columns, and then you must deploy these data model changes to the application server.

The number of coverage term columns you define for each coverage term type applies to a single coverage. If needed, you can define a second coverage entity. Normally, however, you define one coverage entity per coverable.

The following illustration shows the coverages on the homeowners line. There are two coverable objects, `HomeownersLine` and `HODwelling`. Each coverable object has an array of coverages attached to the `HomeownersLineCov` and `HODwellingCov` coverage entities, respectively.



You define coverage terms directly in the coverage entity. The coverage term definition consists of two `column` elements, one that defines the coverage term type and one that stores whether the term was available last time PolicyCenter checked availability. The availability column has the same name of the type definition column, followed by `Avl`. You can define any or all of the following coverage term types in each of your coverage entities:

Coverage term type	Description	column	column
Direct	Integer value that can store any positive number.	• name – `DirectTerm`*n* • type – `decimal`	• name – `DirectTerm`*n*`Avl` • type – `bit`
Choice	Option, Package, and Typekey coverage terms.	• name – `ChoiceTerm`*n* • type – `patterncode`	• name – `ChoiceTerm`*n*`Avl` • type – `bit`
Boolean	True or false value.	• name – `BooleanTerm`*n* • type – `bit`	• name – `BooleanTerm`*n*`Avl` • type – `bit`
String	Textual value.	• name – `StringTerm`*n* • type – `shorttext`	• name – `StringTerm`*n*`Avl` • type – `bit`
Date	Date value.	• name – `DateTerm`*n* • type – `datetime`	• name – `DateTimeTerm`*n*`Avl` • type – `bit`

Defining the Coverage Entity for the Coverable

The coverable objects need a coverage entity. The homeowners line needs two coverable objects, `HomeownersLine` and `HODwelling`. The following steps explain how to define the `HomeownersLineCov` entity that provides coverage for the `HomeownersLine`.

To create the new coverage entity for the coverable

1. Create the new coverage entity for the coverable object. In the `entity` element:
   - Set the `table` attribute to the name of the coverage in lower case letters.
   - Set the `type` attribute to `effdated`.
   - Set the `effDatedBranchType` attribute to `PolicyPeriod`.

   For homeowners, create the `HomeownersLineCov` entity. In the Entity editor window, with the **entity** element selected, set the following properties:

Property	Value
`table`	`homeownerslinecov`
`type`	`effdated`
`effDatedBranchType`	`PolicyPeriod`

2. Add a foreign key to the coverable. For homeowners, add a `foreignkey` element to `HomeownersLine`.

Property	Value
`name`	`HOLine`
`fkentity`	`HomeownersLine`
`nullok`	`false`

3. Add coverage terms as columns in the coverage entity. (The previous topic explained how to specify the number and types of coverage terms required.) Add a coverage term for each type that can be used in this coverage. Add the maximum number coverage terms for each coverage term type. The `name` for each coverage term must be unique among coverage terms. For example, you can set the `name` of the first direct coverage term to `DirectTerm1`, and the `name` of the second to `DirectTerm2`.

For the homeowners line, add the following to the coverage term table:

- Two direct coverage terms
- One choice coverage term
- One Boolean coverage term

The following XML code represents a typical, complete coverage entity definition for `HomeownersLineCov`:

```
<?xml version="1.0"?>
<!--Homeowners Line Coverage-->
<entity
  xmlns="http://guidewire.com/datamodel"
  desc="A line-level coverage for Homeowners Line"
  effDatedBranchType="PolicyPeriod"
  entity="HomeownersLineCov"
  exportable="true"
  final="false"
  platform="false"
  table="homeownerslinecov"
  type="effdated">
  <foreignkey fkentity="HomeownersLine" name="HOLine" nullok="false"/>
  <column name="DirectTerm1" type="decimal" desc="direct covterm field"
      getterScriptability="doesNotExist" setterScriptability="doesNotExist">
    <columnParam name="scale" value="4"/>
    <columnParam name="precision" value="20"/>
  </column>
  <column name="DirectTerm1Avl"  type="bit" getterScriptability="doesNotExist"
      setterScriptability="doesNotExist"
      desc="whether or not the DirectTerm1 field was available the last time
      availability was checked"/>
  <column name="DirectTerm2" type="decimal" desc="direct covterm field"
      getterScriptability="doesNotExist" setterScriptability="doesNotExist">
    <columnParam name="scale" value="4"/>
    <columnParam name="precision" value="20"/>
  </column>
  <column name="DirectTerm1Av2"  type="bit" getterScriptability="doesNotExist"
      setterScriptability="doesNotExist"
      desc="whether or not the DirectTerm1 field was available the last time
      availability was checked"/>
  <column name="ChoiceTerm1" type="patterncode" desc="choice covterm field"
      getterScriptability="doesNotExist"
      setterScriptability="doesNotExist"/>
  <column name="ChoiceTerm1Avl" type="bit"
      desc="whether or not the ChoiceTerm1 field was available the last time
      availability was checked"
      getterScriptability="doesNotExist" setterScriptability="doesNotExist"/>
  <column name="BooleanTerm1" type="bit" desc="boolean covterm field"
      getterScriptability="doesNotExist" setterScriptability="doesNotExist" />
  <column name="BooleanTerm1Avl" type="bit" getterScriptability="doesNotExist"
      setterScriptability="doesNotExist"
      desc="whether or not the BooleanTerm1 field was available the last time
      availability was checked"/>
</entity>
```

To link the coverable to the coverage

In this example, the coverable is the policy line (`HomeownersLine`), and the coverage is `HomeownersLineCov`.

1. Use Studio to open the coverable entity. For the homeowners line, open `HomeownersLine.eti` in **configuration** → **config** → **extensions** → **entity.**

2. Add the `Coverage` as an array on the `Coverable`. For the homeowners line, add an `array` element to the `HomeownersLine` entity with the following properties:

Property	Value
`name`	`HOLineCoverages`
`arrayentity`	`HomeownersLineCov`
`cascadeDelete`	`true`
`desc`	`Line-level coverages for HomeownersLine`
`setterScriptability`	`external`

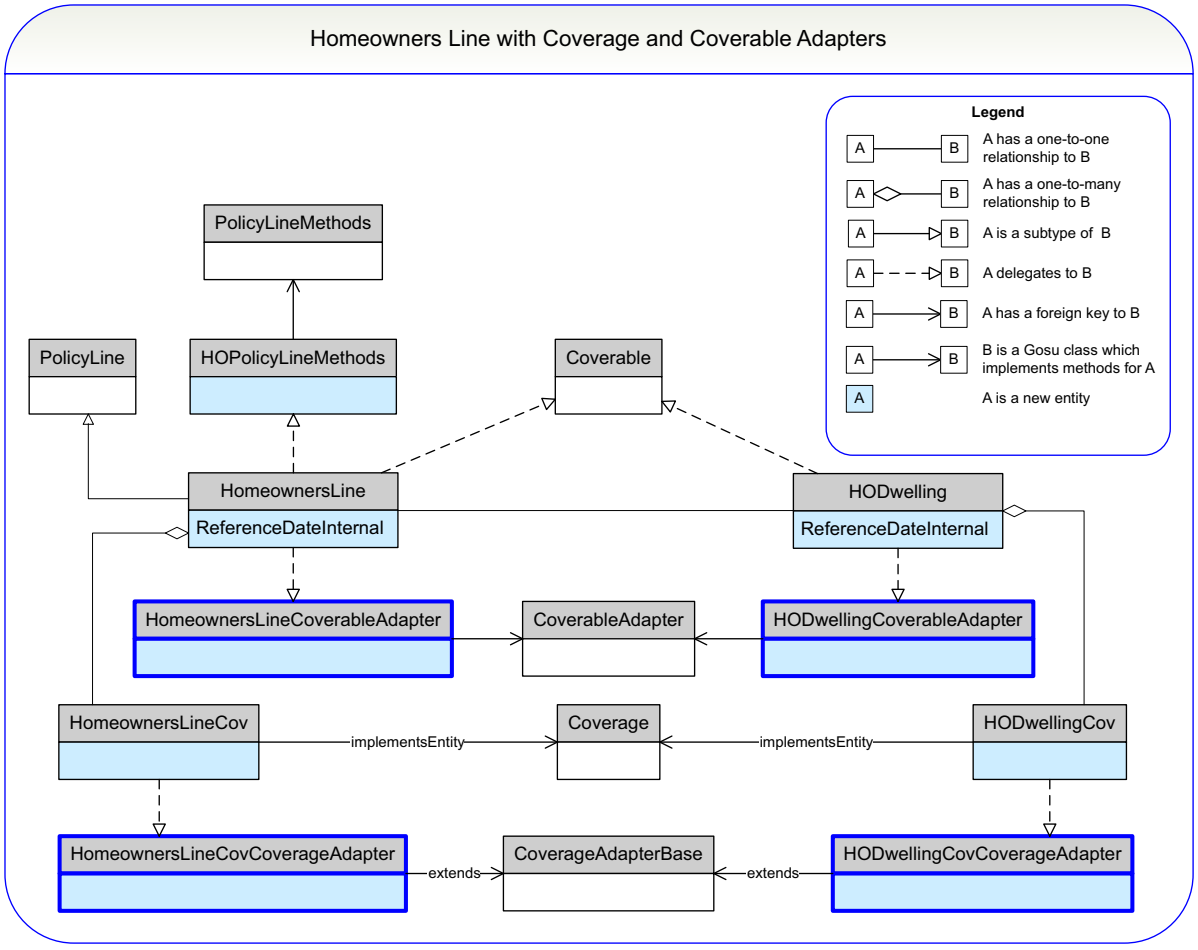
Note: At this point, you can verify your work. For instructions, see "Verifying Your Work" on page 136.

Understanding the Coverable and Coverage Adapters

In previous topics, you created coverable and coverage entities. However, other than their names, nothing in the definitions of these entities defines them as either coverable or coverage. Delegates (or adapters) provide the methods that determine whether these entities are coverables or coverages.

- The coverable adapter provides methods that a coverable needs, such as methods to link the coverable to the coverage. For example, the coverable adapter provides methods to add a coverage, remove a coverage, or get all coverages.
- The coverage adapter provides methods that a coverage needs, such as methods to link the coverage back to the coverable. For example, the coverage adapter provides methods to get the owning coverable and to get the policy line.

The following illustration shows the coverable and coverage adapters for the homeowners line.

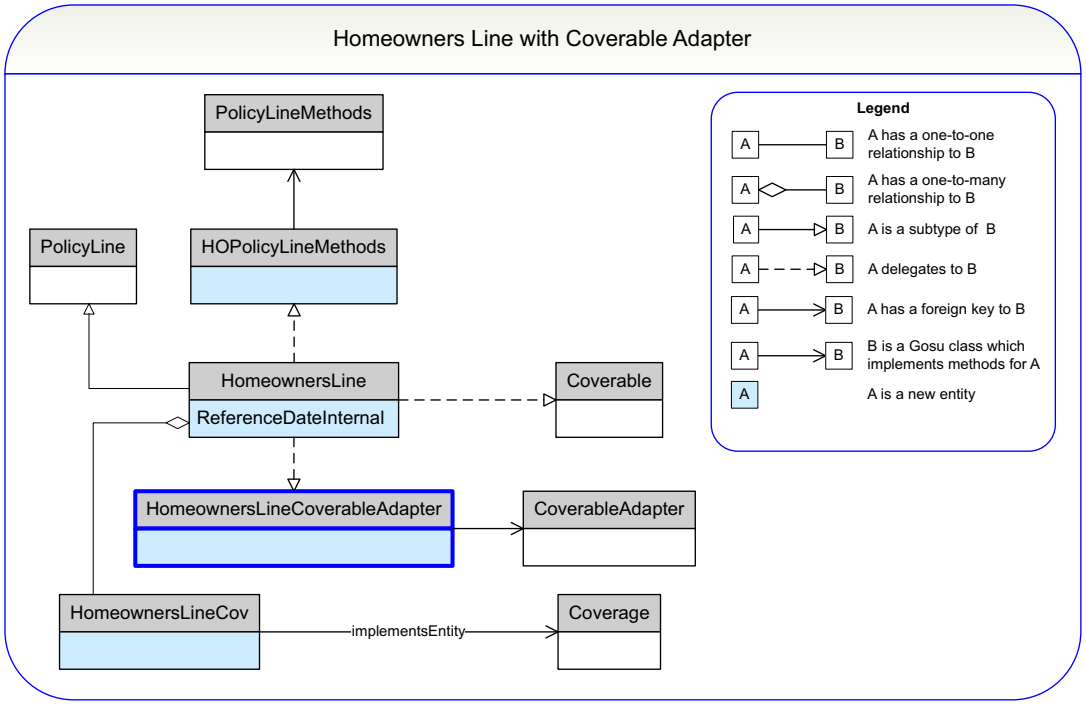


Creating the Coverable Adapter

In previous topics, you defined the `Coverable` entity. In this topic, you declare and define the coverable adapter for this entity. The coverable adapter contains methods of use for a coverable such as methods to add and remove coverages.

Note: To create the coverage adapter that connects the coverage to the coverable, see “Creating the Coverage Adapter” on page 146.

The following illustration shows a coverable adapter in the homeowners line. The `HomeownersLineCoverableAdapter` adapter establishes the interface between the `HomeownersLine` coverable and `HomeownersLineCov` coverage.



To declare the coverable adapter in the coverable entity

- Add an `implementsEntity` element to the coverable entity.

  For the homeowners line, use the Studio Entity editor to add an `implementsEntity` element to the entity definition in `HomeownersLine.eti` in **configuration** → **config** → **extensions** → **entity:**

Property	Value
name	`Coverable`
adapter	`gw.lob.ho.HomeownersLineCoverableAdapter`

  This statement references the Gosu class that defines the adapter methods and properties.

To create the coverable adapter

After you declare the coverable adapter, you must write the code for the adapter. Auto-complete in Studio helps you write the code. Create coverable adapters for each coverable object. For homeowners, create coverable adapters for `HomeownersLine` and `HODwelling`.

1. In the **Project** window in Studio, navigate in **configuration** → **gsrc** to the `gw.lob.`*`line`* package. For homeowners, navigate to `gw.lob.ho`.
2. Right-click the *`line`* node, and then select **New** → **Gosu Class**.
3. Enter the name of the adapter. You specified the name of the adapter in the `implementsEntity` element. For the `HomeownersLine` entity, the name of the coverable adapter is `HomeownersLineCoverableAdapter`.
4. In the new Gosu class, write a constructor similar to the following for the `HomeownersLineCoverableAdapter`:

```
package gw.lob.ho
uses gw.api.domain.CoverableAdapter
```

```
uses entity.HomeownersLine
class HomeownersLineCoverableAdapter implements CoverableAdapter
{
  var _owner : entity.HomeownersLine
  construct(owner : entity.HomeownersLine)
  {
    _owner = owner
  }
}
```

This code produces a Gosu error in the class statement that you fix in the next step.

5. Place the insertion point at the error (after `CoverableAdapter`) and press `Alt-Enter`. Click the **Implement-Methods** command that pops up to open the **Select Methods to Implement** dialog box. With all listed methods selected, click **OK** to insert the empty methods into your code. For homeowners, `HomeownersLineCoverableAdapter` looks similar to the following:

```
package gw.lob.ho
uses gw.api.domain.CoverableAdapter
uses entity.HomeownersLine
uses java.util.Date

class HomeownersLineCoverableAdapter implements CoverableAdapter
{
  var _owner : entity.HomeownersLine
  construct(owner: entity.HomeownersLine)
  {
    _owner = owner
  }

  override property get PolicyLine(): gw.pc.policy.lines.entity.PolicyLine {
    return null
  }
  override property get PolicyLocations(): gw.pc.policy.period.entity.PolicyLocation[] {
    return null
  }
  override property get State(): gw.pl.geodata.zone.typekey.Jurisdiction {
    return null
  }
  override property get AllCoverages(): gw.pc.coverage.entity.Coverage[] {
    return null
  }
  override function addCoverage(p0: gw.pc.coverage.entity.Coverage) {
  }
  override function removeCoverage(p0: gw.pc.coverage.entity.Coverage) {
  }
  override property get AllExclusions(): gw.pc.coverage.entity.Exclusion[] {
    return null
  }
  override function addExclusion(p0: gw.pc.coverage.entity.Exclusion) {
  }
  override function removeExclusion(p0: gw.pc.coverage.entity.Exclusion) {
  }
  override property get AllConditions(): gw.pc.coverage.entity.PolicyCondition[] {
    return null
  }
  override function addCondition(p0: gw.pc.coverage.entity.PolicyCondition) {
  }
  override function removeCondition(p0: gw.pc.coverage.entity.PolicyCondition) {
  }
  override property get ReferenceDateInternal(): Date {
    return null
  }
  override property set ReferenceDateInternal(p0: Date) {
  }
  override property get DefaultCurrency(): gw.pl.currency.typekey.Currency {
    return null
  }
  override property get AllowedCurrencies(): List <gw.pl.currency.typekey.Currency> {
    return null
  }
}
```

6. Write code for the new methods and properties. In most cases, the code for each method is a single line. Examine the coverable adapters for other lines of business as an example.

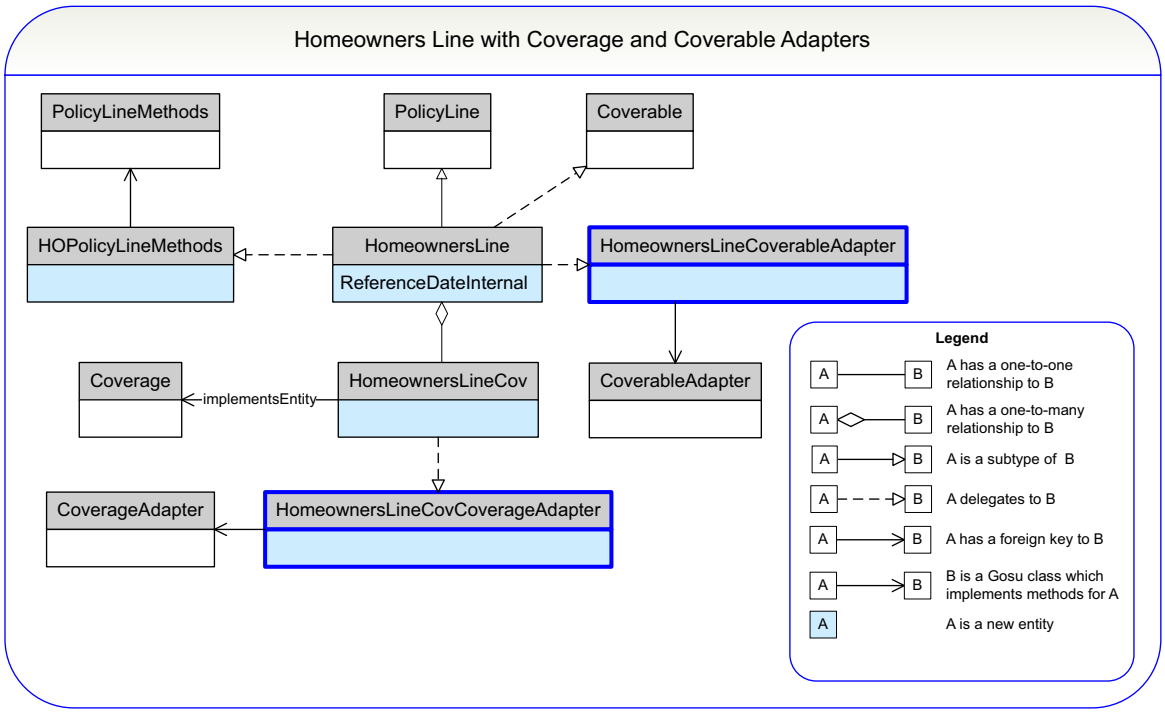
Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Creating the Coverage Adapter

The `Coverage` and the `Coverable` are closely connected logically, but the default coverage definition does not connect it to the coverable. The `CoverageAdapter` delegate actively connects the `Coverage` to the `Coverable`.

Note: To create the coverable adapter that connects the coverable to the coverage, see "Creating the Coverable Adapter" on page 143.

The following illustration shows a simple homeowners policy line in which the line is a coverable. There is a coverable adapter for the line. The coverage on the line has a coverage adapter. These two adapters establish the interface between the coverable and its coverages.



To declare the coverage adapter in the coverage entity

- Add an `implementsEntity` element to the coverage entity.

  For the homeowners line, use the Studio Entity editor to add an `implementsEntity` element to the entity definition in `HomeownersLineCov.eti` in **configuration** → **config** → **extensions** → **entity**:

Property	Value
`name`	`Coverage`
`adapter`	`gw.lob.ho.HomeownersLineCovCoverageAdapter`

  This statement references the Gosu class that defines the adapter methods and properties.

To create the coverage adapter

After you declare the coverage adapter, you must write the code for the adapter. Auto-complete in Studio helps you write the code. Create coverage adapters for each coverage. The process is similar to creating the `Coverable` adapter.

1. In the **Project** window in Studio, navigate in **configuration** → **gsrc** to the `gw.lob.`*`line`* package. For homeowners, navigate to `gw.lob.ho`.
2. Right-click the *`line`* node, and then select **New** → **Gosu Class**.

3. Enter the name of the adapter. You specified the name of the adapter in the `implementsEntity` element. For the `HomeownersLineCov` entity, the name of the coverable adapter is `HomeownersLineCovCoverageAdapter`.

4. In the new Gosu class, write a constructor similar to the following for the `HomeownersLineCovCoverageAdapter`. The constructor extends rather than implements the class because it is a subclass of `CoverageAdapterBase`, which add methods to `CoverageAdapter`.

```
package gw.lob.ho
uses gw.coverage.CoverageAdapterBase

class HomeownersLineCovCoverageAdapter extends CoverageAdapterBase
{
  var _owner : entity.HomeownersLineCov

  construct(owner : entity.HomeownersLineCov)
  {
    super(owner)
    _owner = owner
  }
}
```

   This code produces a Gosu error in the class statement that you fix in the next step.

5. Place the insertion point at the error (after `CoverageAdapterBase`) and press `Alt-Enter`. Click the **Implement-Methods** command that pops up to open the **Select Methods to Implement** dialog box. With all listed methods selected, click **OK** to insert the empty methods into your code. For homeowners, `HomeownersLineCovCoverageAdapter` looks similar to the following:

```
package gw.lob.ho
uses gw.coverage.CoverageAdapterBase

class HomeownersLineCovCoverageAdapter extends CoverageAdapterBase
{
  var _owner : entity.HomeownersLineCov
  construct(owner: entity.HomeownersLineCov)
  {
    super(owner)
    _owner = owner
  }

  override function addToCoverageArray(p0: gw.pc.coverage.entity.Coverage) {
  }
  override property get OwningCoverable(): gw.pc.coverage.entity.Coverable {
    return null
  }
  override function removeFromParent() {
  }
  override property get PolicyLine(): gw.pc.policy.lines.entity.PolicyLine {
     return null
  }
  override property get CoverageState(): gw.pl.geodata.zone.typekey.Jurisdiction {
    return null
  }
}
```

6. Write code for the new methods and properties. In most cases, the code for each method is a single line. Examine the coverage adapters for other lines of business, such as businessowners, as an example.

   Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Updating Policy Line Methods for the New Coverages

Next, make a change in `PolicyLineMethods` to support the new policy line coverages. Add the following:

- The `AllCoverage` property
- Other properties and methods as needed

To add the property for all coverages

1. In Studio, open the `gw.lob.`*`package`*`.`*`line`*`PolicyLineMethods` class that you created in **configuration** → **gsrc**. For homeowners, open `gw.lob.ho.HOPolicyLineMethods.gs`.

2. Define the `get AllCoverages` property.

   Using homeowners as an example, if the policy line coverages are the only coverages within the policy line, then the following code is sufficient:

   ```
   override property get AllCoverages() : Coverage[] {
     return _line.HOLineCoverages
   }
   ```

   If you added coverages to other objects, you can use code similar to the following example. This example for the homeowners line adds coverages from the line and coverages from the dwelling:

   ```
   uses java.util.ArrayList
   ...
   override property get AllCoverages() : Coverage[] {
     var coverages = new ArrayList<Coverage>()
     coverages.addAll( _line.CoveragesFromCoverable.toList())
     // add for each dwelling
     coverages.addAll( _line.Dwelling.Coverages.toList())
     return coverages as Coverage[]
   }
   ```

   Note: This code adds the line and dwelling coverages together by converting them to a linked list, because it is easier to add linked lists than to add arrays.

3. Define other properties and methods as needed.

   Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Adding Availability Lookup Tables for Coverages

Lookup tables store information about coverage and coverage term availability. The final step in enabling a new coverage is to add information about the new coverage to the lookup table. The lookup table has separate definitions for:

- Coverages
- Coverage Terms
- Coverage Term Options
- Coverage Term Packages

The standard lookup tables use effective and expiration dates (by default), jurisdictions, and underwriting companies to control availability. If you need availability to be based upon other criteria, you can extend the lookup table. For information on extending the lookup table, see “Extending an Availability Lookup Table” on page 92 and “Step 3: Using the Updated Availability Column” on page 94.

The lookup table defines the precedence rules for availability. The lookup table also specifies how the lookup values can be linked from the item using the value path.

To add lookup table information for new coverages

1. In Studio, navigate to **configuration** → **config** → **lookuptables** and open `lookuptables.xml`.

2. Add lookup table information for the new coverage. The following information is for the homeowners line.

   ```
   <!-- ============================================================================== -->
   <!-- Homeowners Lookup Tables                                                       -->
   <!-- ============================================================================== -->

   <LookupTable code="HomeownersLineCov" entityName="CoverageLookup" root="HomeownersLine">
     <Filter field="PolicyLinePatternCode" valuePath="HomeownersLine.PatternCode"/>
     <Dimension field="State" valuePath="HomeownersLine.BaseState" precedence="0"/>
     <Dimension field="UWCompanyCode" valuePath="HomeownersLine.Branch.UWCompany.Code" precedence="1"/>
     <DistinguishingField field="CoveragePatternCode"/>
   </LookupTable>
   ```

   Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Adding a Coverage Pattern in the Policy Line

Now that you have defined a coverage, you add a coverage pattern to the policy line.

To add a coverage pattern to the policy line

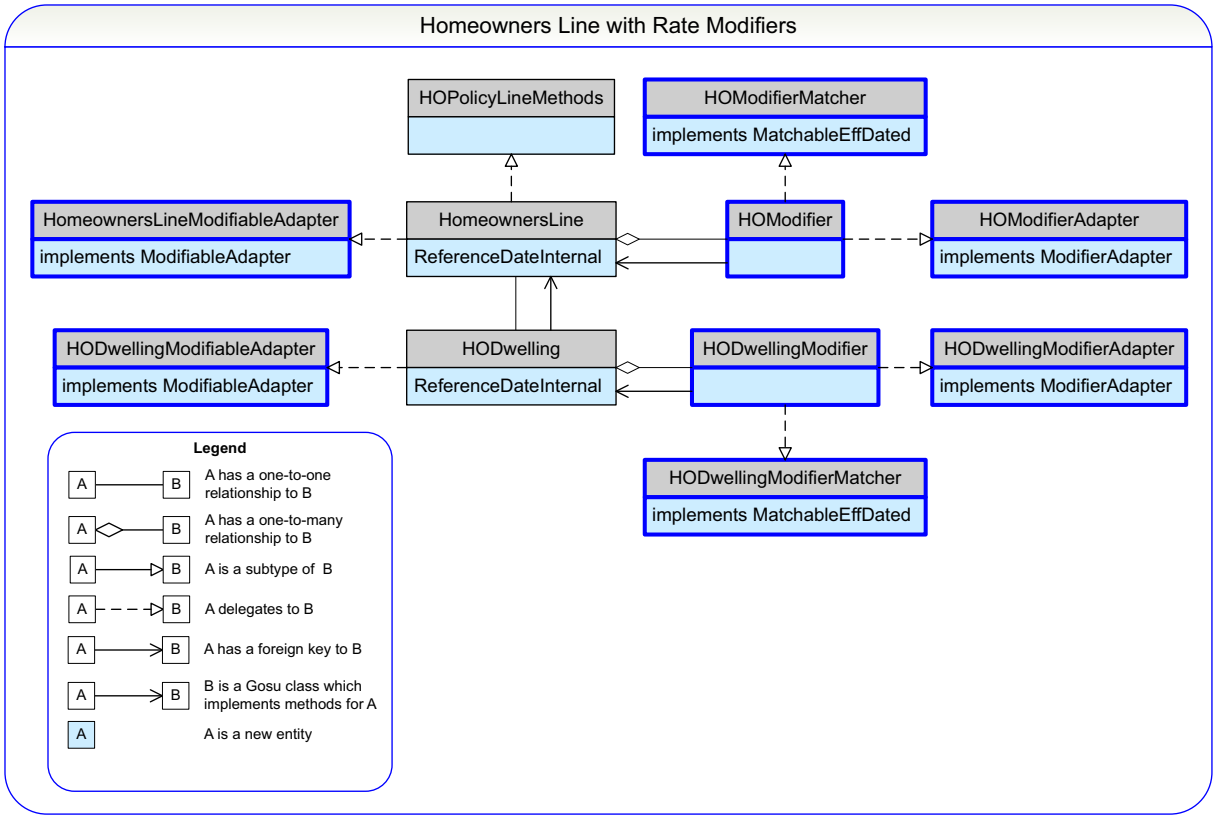
1. Use Product Designer to add a policy line as described in “Creating the Policy Line” on page 159. Log out and log in to Product Designer to reload the latest product model configuration from the specified workspace.
2. In the Policy Line home page, under **Go to**, click **Categories**.
3. Click **Add** to add a new category. In the **Add Category** dialog box, enter a code and name, and then click **OK**.
4. In the Policy Line home page, under **Go to**, click **Coverages**.
5. Click **Add** to add a new coverage. Fill in the **Add Coverage** dialog box with appropriate values, and then click **OK**.
6. Expand the **Changes** page, and the click **Commit All** to save your changes to the PolicyCenter configuration.

Step 5: Add Rate Modifiers to the New Line of Business

You can define rating modifiers on custom lines of business. The homeowners line has rating modifiers for both the policy line and the dwelling. For example, rating modifiers can modify the premium rate on the line with a multi-policy discount. Rating modifiers can modify the premium rate on the dwelling with a discount for protection against lightning. You can control the availability of rating modifiers by using availability lookup tables.

Delegates actively connect the modifiable to its modifier. A delegate establishes the interface from one object to another. The `ModifiableAdapter` links the `Modifiable` to the `Modifier`.

The following illustration shows the homeowners line with two modifiable objects: `HomeownersLine` and `HODwelling`. Each modifiable object has a modifier, `HOModifier` and `HODwellingModifier`, respectively. Each modifier has modifier methods. Each modifiable object has a modifiable adapter that connects the modifiable object to its modifier.



Defining modifiers is similar to defining coverages, with a few differences. Instead of defining a `CoverableAdapter`, you define a `ModifiableAdapter` for each modifiable object. Instead of defining a `CoverageAdapter`, you define a `ModifierAdapter` for each modifier. In addition, you must define a `MatchableEffDated` delegate on the modifier. This delegate handles out-of-sequence events and preemption.

The steps to add modifiers to a new line of business are:

- “Creating the Modifier Entity” on page 151
- “Creating the Modifiable Adapter” on page 151
- “Creating the Modifier Adapter” on page 152
- “Creating the Modifier Matcher” on page 153
- “Adding an Availability Lookup Table for Modifiers” on page 154
- “Adding a Modifier Pattern to the Policy Line” on page 154
- “Creating Rate Factors” on page 155
- “Creating the Rate Factor Delegate” on page 156
- “Creating the Rate Factor Matcher” on page 157
- “Adding Availability Lookup for Rating Factors” on page 157
- “Adding a Modifier Pattern with Rate Factors in the Policy Line” on page 157

Creating the Modifier Entity

This section describes how to create the modifier entity. The homeowners line needs two modifiers, `HOModifier` and `HODwellingModifier`. The instructions describe how to create the `HOModifier` entity.

To create the modifier entity

1. Use Studio to create the new modifier entity for the modifiable object. In the `entity` element:
   - Set the `table` attribute to the name of the modifier in lower case letters.
   - Set the `type` attribute to `effdated`.
   - Set the `effDatedBranchType` attribute to `PolicyPeriod`.

   For homeowners, create the `HOModifier` entity. In the Entity editor window, with the `entity` element selected, set the following properties:

Property	Value
`table`	`homodifier`
`type`	`effdated`
`effDatedBranchType`	`PolicyPeriod`
`desc`	`A line-level modifier for homeowners`

2. Add a foreign key to the modifiable object. For homeowners, add a `foreignkey` element to `HomeownersLine`.

Property	Value
`name`	`HOLine`
`fkentity`	`HomeownersLine`
`nullok`	`false`

Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Creating the Modifiable Adapter

Next you must declare and define the modifiable adapter for the modifiable entity. The modifiable adapter contains methods for a modifiable object, such as methods to add and remove modifiers. The homeowners line needs two modifiable entities, `HomeownersLine` and `HODwelling`. The following steps explain how to define the `HOModifiers` entity that provides coverage for the `HomeownersLine`.

IMPORTANT To create the modifier adapter that connects the modifier to the modifiable object, see “Creating the Modifier Adapter” on page 152.

To declare the modifiable adapter on the modifiable entity

1. In Studio, open the modifiable object. For homeowners, open `HomownersLine.eti`.
2. Define the modifier subtype as an `array` from the modified object with the following properties:

Property	Value
`name`	`HOModifiers`
`arrayEntity`	`HOModifier`
`desc`	`Rating information for the line`

3. Declare a modifiable delegate. The modifiable delegate connects the modifiable object with the modifier.

   For the homeowners line, use the Studio Entity editor to add an `implementsEntity` element to the entity definition in `HomeownersLine.eti` in **configuration** → **config** → **extensions** → **entity**:

Property	Value
`name`	`Modifiable`
`adapter`	`gw.lob.ho.HOLineModifierAdapter`

To create the modifiable adapter

After you declare the modifiable adapter, you must write the code for the adapter. Auto-complete in Studio helps you write the code. Create modifiable adapters for each modifiable object. For homeowners, create modifiable adapters for `HomeownersLine` and `HODwelling`.

1. In the **Project** window in Studio, navigate in **configuration** → **gsrc** to the `gw.lob.`*`package`* package. For homeowners, navigate to `gw.lob.ho`.
2. Right click the *package* node, and then select **New** → **Gosu Class**.
3. Enter the name of the adapter. You specified the name of the adapter in the `implementsEntity` element in the preceding procedure. For the `HomeownersLine` entity, the name of the modifiable adapter is `HOLineModifiableAdapter`.

   Note: Modifier instantiation is usually handled in the same manner as coverages using the `gw.web.productmodel.ProductModelSyncIssuesHandler` methods.
4. In the new Gosu class, write a constructor similar to the following for `HOLineModifiableAdapter`.

```
package gw.lob.ho
uses gw.api.domain.ModifiableAdapter
uses java.util.Date

class HOLineModifiableAdapter implements ModifiableAdapter {
  var _owner : HomeownersLine

  construct(owner : HomeownersLine) {
    _owner = owner
  }
}
```

   This code produces a Gosu error in the class statement that you fix in the next step.
5. Place the insertion point at the error (after `ModifiableAdapter`) and press `Alt-Enter`. Click the **Implement-Methods** command that pops up to open the **Select Methods to Implement** dialog box. With all listed methods selected, click **OK** to insert the empty methods into your code.
6. Write code for the new methods and properties. In most cases, the code for each method is a single line. Examine the modifier adapters for other lines of business as an example.

   Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Creating the Modifier Adapter

The `Modifier` and the `Modifiable` entities are closely connected logically, but the default modifier definition does not connect the modifier to the modifiable object. The `ModifierAdapter` delegate actively connects the `Modifier` entity to the `Modifiable` entity.

IMPORTANT To create the modifiable adapter that connects the modifiable object to the modifier, see “Creating the Modifiable Adapter” on page 151.

To declare the modifier adapter on the modifier entity

- Add an `implementsEntity` element to the modifier entity.

  For the homeowners line, use the Studio Entity editor to add an `implementsEntity` element to the entity definition in `HOModifier.eti` in **configuration** → **config** → **extensions** → **entity**:

Property	Value
`name`	`Modifier`
`adapter`	`gw.lob.ho.HOModifierAdapter`

  This statement references the Gosu class that defines the adapter methods and properties.

To create the modifier adapter

After you declare the modifier adapter, you must write the code for the adapter. Auto-complete in Studio helps you write the code. Create modifier adapters for each modifier. The process is similar to creating the `Coverable` adapter.

1. In the **Project** window in Studio, navigate in **configuration** → **gsrc** to `gw.lob.`*`line`*. For homeowners, navigate to `gw.lob.ho`.
2. Right-click the *`line`*, and select **New** → **Gosu Class**.
3. Enter the name of the adapter. You specified the name of the adapter in the `implementsEntity` element of the modifier. For homeowners, enter `HOModifier`.
4. In the new Gosu class, write a constructor similar to the following for the `HOModifierAdapter`. The class implements `ModifierAdapter`:

```
package gw.lob.ho
uses gw.api.domain.ModifierAdapter

class HOModifierAdapter implements ModifierAdapter {

  var _owner : entity.HOModifier

  construct(modifier : HOModifier) {
    _owner = modifier
  }
}
```

   This code produces a Gosu error in the class statement that you fix in the next step.
5. Place the insertion point at the error (after `ModifierAdapter`) and press `Alt-Enter`. Click the **ImplementMethods** command that pops up to open the **Select Methods to Implement** dialog box. With all listed methods selected, click **OK** to insert the empty methods into your code.
6. Write code for the new methods and properties. In most cases, the code for each method is a single line. Examine the modifier adapters for other lines of business as an example.

   Note: Some of these methods apply to rate factors. Add the code for these methods after you define rate factors in “Creating Rate Factors” on page 155.

   Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Creating the Modifier Matcher

Each modifier must have a modifier matcher delegate that handles out-of-sequence events and preemption. The modifier matcher extends `AbstractModifierMatcher`.

To declare the modifier matcher in the modifier entity

1. In Studio, open the modifier object. For homeowners, open `HOModifier.eti` entity in **configuration** → **config** → **extensions** → **entity**.

2. Declare the modifier matcher delegate. For homeowners, add the following `implementsInterface` element to the entity:

Property	Value
`iface`	`gw.api.logicalmatch.EffDatedLogicalMatcher`
`impl`	`gw.lob.ho.HOModifierMatcher`

To create the modifier matcher

Examine other lines of business, such as businessowners, for examples of modifier matcher code. The businessowners class is `BOPModifierMatcher` in **configuration** → **config** → **metadata** → **entity**. For homeowners, create `HOModifierMatcher.gs` for the `HOModifier`, and `HODwellingModifierMatcher.gs` for the `HODwellingModifier`.

Note: At this point, you can verify your work. For instructions, see "Verifying Your Work" on page 136.

Adding an Availability Lookup Table for Modifiers

Add availability lookup table information for the new modifier entities.

To add availability lookup table information for modifiers

1. Use Studio to navigate to **configuration** → **config** → **lookuptables** and open `lookuptables.xml`.
2. Add a `LookupTable` element for each modifier. Following is an example of code for the `HOModifier` entity in the homeowners line:

```
<!-- Modifiers -->
<LookupTable code="HOModifier" entityName="ModifierLookup" root="HomeownersLine">
  <Filter field="PolicyLinePatternCode" valuePath="HomeownersLine.PatternCode"/>
  <Dimension field="State" valuePath="HomeownersLine.BaseState" precedence="0"/>
  <Dimension field="UWCompanyCode" valuePath="HomeownersLine.Branch.UWCompany.Code" precedence="1"/>
  <DistinguishingField field="ModifierPatternCode"/>
</LookupTable>
```

For more information about availability lookup tables, see "Configuring Availability" on page 83.

Adding a Modifier Pattern to the Policy Line

Now that you have defined a modifier, you can use Product Designer to add a modifier pattern to the policy line.

To add a modifier pattern to the policy line

1. If you have not already done so, use Product Designer to add a policy line as described in "Creating the Policy Line" on page 159. Log out and log in to Product Designer to reload the latest product model configuration from the specified workspace.
2. In the Policy Line home page, under **Go to**, click **Modifiers**.
3. Click **Add** to add a new modifier. In the **Add Modifier** dialog box, enter a code and name, select a modifier type and data type, and then click **OK**.
4. Expand the **Changes** page, and the click **Commit All** to save your changes to the PolicyCenter configuration.

See also

- "Rate Modifiers" on page 52

Creating Rate Factors

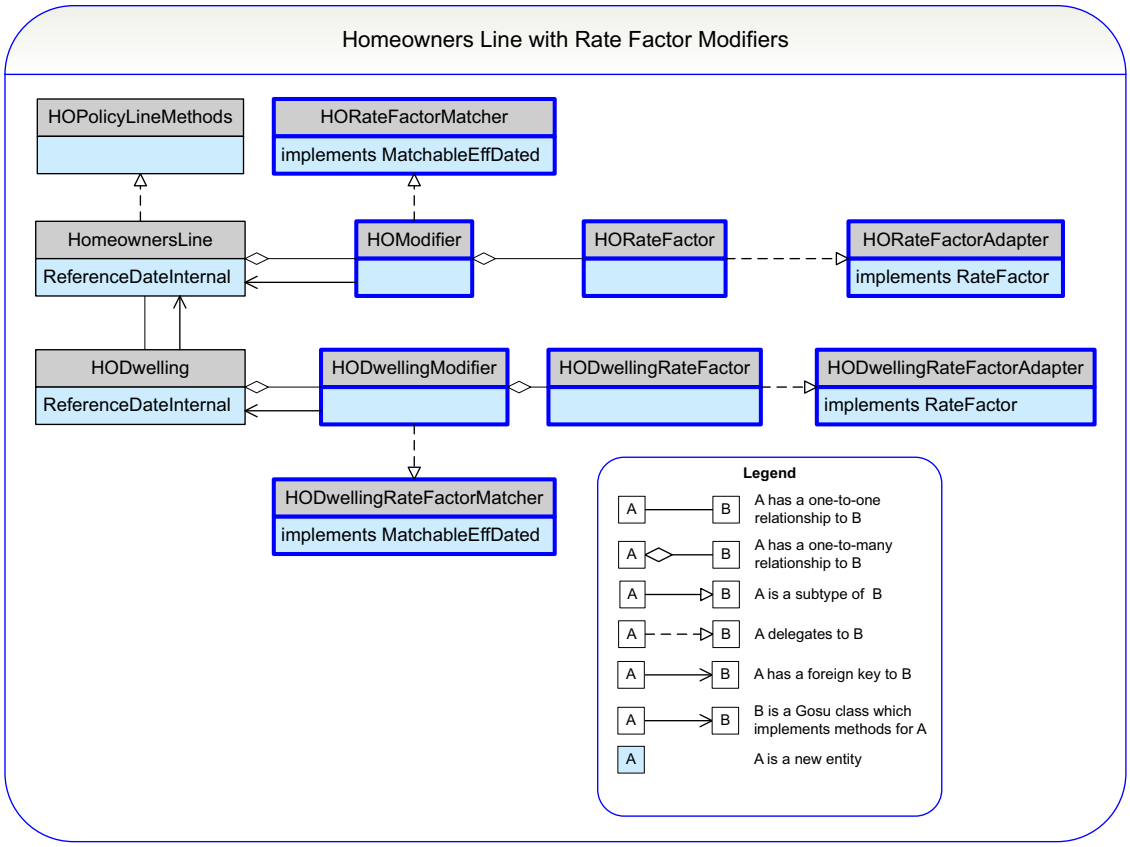
A rating input modifier consists of one or more rate factors. Rate factors are generally used in commercial lines of business. For demonstration purposes, you can add them to the homeowners line, which is a personal line of business.

Rate factors must be listed in the `RateFactorType` typelist before they can be added to a schedule rate. Therefore, the first step in creating a schedule rate modifier is to create the necessary rate factors in `RateFactorType`. Studio automatically handles new typecodes, so you can create a schedule rate modifier without restarting the server. However, because changes to typelists are considered data model changes, you must restart the PolicyCenter server before you can select the rate factor type in Product Designer.

You add modifiers in Product Designer by using the **Modifiers** page for products and policy lines. Product Designer adds a **Rate Factors** link under **Go to** on the **Modifiers** page if you do both of the following when adding a new modifier:

- Set the **Data Type** to **rate**.
- Select the **Schedule Rate** check box.

The following illustration shows the homeowners line with rate factor modifiers on `HomeownersLine` and `HODwelling`. You must define a `MatchableEffDated` delegate on the modifier for rate factors. This delegate handles out-of-sequence events and preemption. (You previously defined `MatchableEffDated` delegates for the modifiers.)



For an example of rate factors in the base configuration, see `BOPRateFactor.eti` in **configuration** → **config** → **metadata** → **entity**.

To define entities for rate factors

1. Use Studio to define a new entity. For homeowners, define `HORateFactor.eti` in **configuration** → **config** → **metadata** → **entity**. Configure the entity as follows:

Property	Value
entity	HORateFactor
table	horatefactor
type	effdated
desc	A rate factor is a risk characteristic and its associated numeric value.
autoSplit	false
effDatedBranchType	PolicyPeriod
exportable	true
extendable	true

2. Add a required `foreignkey` to the owning modifier. The owning modifier must implement the `Modifiable` delegate. For homeowners, the owning modifiable is `HOModifier.eti`. Configure the foreign key as follows:

Property	Value
name	HOModifier
fkentity	HOModifier
nullok	false

To add rate factors as an array on the owning modifiable

1. Use Studio to open the owning modifiable. For homeowners, open `HOModifier.eti`.
2. Add an `array` of rate factors. For homeowners, configure the array as follows:

Property	Value
name	HORateFactors
arrayentity	HORateFactor
desc	Individual components of the rate factor.
cascadeDelete	true
owner	false

Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Creating the Rate Factor Delegate

The rate factor delegate implements the methods for the rate factor.

To update the entity

Add the rate factor delegate to the rate factor entity.

- For homeowners, use Studio to add an `implementsEntity` element to the `HORateFactor` entity as follows:

Property	Value
name	RateFactor
adapter	gw.lob.ho.HORateFactorAdapter

To implement the rate factor delegate

- Use Studio to create the rate factor delegate `gw.lob.ho.HORateFactorAdapter.gs` in **configuration** → **gsrc**. Use `gw.lob.bop.BOPRateFactorAdapter.gs` as a model.

  Note: At this point, you can verify your work. For instructions, see "Verifying Your Work" on page 136.

Creating the Rate Factor Matcher

The rate factor matcher implements the methods for the `MatchableEffDated` interface.

To update the entity

- Use Studio to add the `MatchableEffDated` delegate to the rate factor entity. For homeowners, add an `implementsInterface` element to the `HORateFactor` entity as follows:

Property	Value
iface	`gw.api.logicalmatch.EffDatedLogicalMatcher`
impl	`gw.lob.ho.HORateFactorMatcher`

To implement the rate factor matcher

- Use Studio to create the code for the rate factor matcher `gw.lob.ho.HORateFactorAdapter` in **configuration** → **gsrc**. Use `gw.lob.bop.BOPRateFactorMatcher` as a model.

  Note: At this point, you can verify your work. For instructions, see "Verifying Your Work" on page 136.

Adding Availability Lookup for Rating Factors

Add availability lookup table information for the new rate factor entities.

To add availability lookup table information for new rate factors

1. Use Studio to navigate to **configuration** → **config** → **lookuptables** and open `lookuptables.xml`.
2. Add a `LookupTable` element for each rate factor. Following is an example of code for the `HORateFactor` entity in the homeowners line:

```
<!-- Rate Factors -->
<LookupTable code="HORateFactor" entityName="RateFactorLookup" root="HomeownersLine">
    <Filter field="PolicyLinePatternCode" valuePath="HomeownersLine.PatternCode"/>
    <Dimension field="State" valuePath="HomeownersLine.BaseState" precedence="0"/>
    <Dimension field="UWCompanyCode" valuePath="HomeownersLine.Branch.UWCompany.Code" precedence="1"/>
    <Dimension field="JobType" valuePath="HomeownersLine.Branch.Job.Subtype" precedence="2"/>
    <DistinguishingField field="RateFactorPatternCode"/>
</LookupTable>
```

For more information about availability lookup tables, see "Configuring Availability" on page 83.

Adding a Modifier Pattern with Rate Factors in the Policy Line

Now that you have defined rate factors, you can use Product Designer to add a modifier pattern with rate factors to the policy line.

To add a modifier pattern with rate factors to the policy line

1. If you have not already done so, use Product Designer to add a policy line as described in "Creating the Policy Line" on page 159. Log out and log in to Product Designer to reload the latest product model configuration from the specified workspace.
2. In the Policy Line home page, under **Go to**, click **Modifiers**.

3. Click **Add** to add a new modifier. In the **Add Modifier** dialog box, enter a code and name, and select a modifier type. Set the **Data Type** to **rate** and select the **Schedule Rate** check box, and then click **OK**.
4. On the new modifier's home page, under **Go to**, click the **Rate Factors** link.
5. Click **Add** and add a rate factor.

Step 6: Add Optional Features to a Policy Line

Certain features do not apply to all policy lines. For each policy line pattern, you can define whether the line includes or excludes the following optional features:

- Blankets
- Coverage Symbol Groups
- Official IDs
- Split Rating Period options, also known as anniversary rating date (ARD) options.

To configure coverage symbol groups, official IDs, and blankets

1. Use the **Project** window in Studio to navigate to **configuration** → **config** → **resources** → **productmodel** → **policylinepatterns** → ***XX*Line** where *XX* is the line abbreviation of an existing line. This node contains properties files for each policy line pattern.
2. Copy and paste one of the properties files. Studio opens the **Copy** dialog box, where you can provide a new name for the copied item.
3. In the **New name** field, enter the policy line code you specified in "Creating the Policy Line" on page 159, followed by `Line.properties`. For the homeowners example as configured in this tutorial, name the file `HomeownersLine.properties`.
4. Edit the properties file for the new line of business. Set the values for each optional item as needed. For example, to display the **Official IDs** tab in PolicyCenter for this product line, set:

```
# Indicates whether this policy line uses Coverage Symbol Groups
line.usesCoverageSymbolGroups=false

# Indicates whether this policy line uses Official IDs
line.usesOfficialIDs=false

# Indicates whether this policy line uses Blankets
line.usesBlankets=false

# Indicates whether this policy line's modifiers use the Split Rating Period field
line.usesSplitRatingPeriod=false
```

Note: PolicyCenter includes any omitted properties by default. Likewise, if you do not create a properties file for the line, PolicyCenter includes all properties.

Step 7: Build the Product Model for the New Line of Business

After you have completed the data model, you can build the product model in Product Designer. The product model contains the policy line. The product model also defines the pattern that you use to create policies for the new line. In this step, you create patterns for the policy line and the product.

The steps to build the product model are:

- "Creating the Policy Line" on page 159
- "Creating the Product" on page 159
- "Adding Icons for the Product and Policy Line" on page 161
- "Adding Product and Policy Line Icons to Product Designer" on page 161

Creating the Policy Line

In this topic, you create the pattern for the policy line. A policy line pattern is a collection of patterns relevant to a specific line of business (such as businessowners or personal auto). PolicyCenter uses policy line patterns to create policy line instances.

To create the policy line

1. Log in to Product Designer. If you are already logged in, log out and log in again to reload the latest product model configuration from the specified workspace.
2. Select or create a change list.

   Note: Be sure to create or select a change list related to a workspace that references the PolicyCenter instance where you have defined your new line of business.

   You specify a workspace when you create a change list in Product Designer. A *workspace* is a named reference to a specific PolicyCenter root folder. Workspaces are defined by Product Designer administrators. A *change list* is a named set of changes related to a workspace. Change lists are defined by Product Designer users. All work you do in Product Designer takes place in change lists of your choice.
3. Navigate to **Policy Lines**. The **Policy Lines** page lists all of the lines that are currently defined in your PolicyCenter configuration. Click **Add** to open the **Add Policy Line** dialog box.
4. In the **Add Policy Line** dialog box, supply the following values:

Property	Value
Code	Enter the public ID you specified as the **name** parameter in "Step 2: Register the New Line of Business" on page 132 when you added `InstalledPolicyLine.HO.ttx`. For the homeowners example, enter `Homeowners`.
Name	The name that appears on the policy line tab. For homeowners, enter `Homeowners Line`.
Policy Line Type	Select the policy line subtype that you created in the data model. For homeowners, select **HomeownersLine**.
Available Coverage Currency	Every policy line must specify at least one coverage currency. Select one coverage currency from this list. For example, select **USD** to specify United States Dollars. Coverage currencies specify the currencies users can select when entering monetary values in PolicyCenter. You can add more coverage currencies after defining the line.

5. Click **OK** to create the Homeowners line and open its home page.

   On this page, you can change the name or description, and you can supply translated names for these fields. You also can add coverage currencies, specify an integration reference code, and configure advanced properties. Using the links under **Go to**, you can navigate to pages that enable you to configure all other aspects of the policy line.

Creating the Product

In this topic, you create the pattern for the product. A *product* is the policy type that appears on the **New Submissions** screen in PolicyCenter. A product is a pattern used to create policy instances. PolicyCenter lists each product on a separate row of the **New Submission** screen.

To create a product

1. In Product Designer, navigate to the **Products** page. Click **Add** to add a new product.

2. In the **Add Product** dialog box, supply the following values:

Property	Value
Code	`Homeowners`
Name	`Homeowners`
Default Policy Term	Select a default policy term.
Policy Line	`Homeowners Line`
Product Type	`Personal`. This value enables PolicyCenter to show or hide portions of certain built-in user interface objects that are designed either for personal or commercial contexts.
Account Type	Select the types of accounts that can have a product of this type: **Person**, **Company**, or **Any**. For homeowners, select **Any**.

3. Click **OK** to open the Product home page for the Homeowners product.

   On this page, you can change the name or description, and you can supply translated names for these fields. You also can change the settings you specified in the **Add Product** dialog box, and specify whether or not the product requires an offering.

4. In the **Abbreviation** field, enter an abbreviation for the product. For homeowners, enter `HO`.

5. Examine the other sections on the Product home page:
   - Under **Policy Lines**, you can add more policy lines to create a package product.
   - Under **Policy Terms**, you can add and remove terms.
   - Under **Integration**, you can specify the public ID of the product pattern as used in a legacy policy system of record.
   - Under **Advanced**, you can configure the quote rounding level, specify the number of days until a quote is needed, enter the code for an initialization script, and manage document templates.
   - Using the links under **Go to**, you can navigate to pages that enable you to configure all other aspects of the product.

6. Under **Go to**, select **Availability**. Make any needed modifications. You can set product availability based on:
   - Start and end effective dates
   - Jurisdiction
   - Job type
   - Industry code

   For more information, see “Configuring Availability” on page 83.

7. Make other changes to the product as needed.

8. Navigate to the **Changes** page and click **Commit** to commit your changes to the PolicyCenter configuration.

You have completed the product model definition. If you start PolicyCenter and create a new submission, your new line appears under **Product Name** on the **New Submissions** screen.

To verify your work

1. Start the server. Check for errors in the application console. To avoid database conflicts, drop the database before starting the server:

   ```
   gwpc dev-dropdb dev-start
   ```

2. Load the sample data, and then return to PolicyCenter. For more information, see “Installing Sample Data” on page 55 in the *Installation Guide*.

3. Create a new submission. PolicyCenter displays your new line of business as a choice on the **New Submissions** screen.

To create a new submission:

a. Click the **Search** → **Accounts**.

b. In the **Company Name** field, type `Wright Construction`, and then click **Search**.

c. Select the Wright Construction account by clicking the **Account Number** link. PolicyCenter opens the **Account File Summary**.

d. Select **Actions** → **New Submission**.

4. Click the **Select** button for **Homeowners** to begin a submission.

Adding Icons for the Product and Policy Line

You can define the icons that PolicyCenter uses for your new product and policy line. The image can be a GIF or PNG image.

To add a product icon

1. In Product Designer, navigate to the Product home page for your new product definition. For homeowners, open the **Homeowners** product. Note the **Abbreviation** for the product. For homeowners, the abbreviation is `HO`.
2. In Studio, navigate to **configuration** → **webresources** → **themes** → **Titanium** → **resources** → **images** → **app**.
3. Right-click the **app** node, and select **Show in Explorer**. Windows Explorer opens to the directory that contains the icons: `PolicyCenter_installation/modules/configuration/webresources/themes/Titanium/resources/images/app`
4. Create a GIF or PNG icon. The recommended maximum size is 32 pixels wide by 24 pixels wide. Most icons in the base configuration are 24 x 24 pixels.
5. Save your product icon to the location in Step 3 and name it `infobar_<abbreviation>.gif` or `infobar_<abbreviation>.png`. For homeowners, save the icon as `infobar_HO.png`.

   Note: PolicyCenter is supplied with a default `infobar_HO.png` icon. If you prefer, use the supplied icon. Otherwise, you can overwrite the existing icon with your new icon.

To add a policy line icon

1. In Product Designer, navigate to the Policy Line home page for your new policy line definition. For homeowners, open the **Homeowners Line** policy line. Note the code for the policy line. The code appears in square brackets following the policy name line in the page title. For homeowners, the code is `HomeownersLine`.
2. Follow the previous steps for adding the product icon. The recommended icon size is 32 pixels wide by 24 pixels high. Name the icon `infobar_<code>.gif` or `infobar_<code>.png`. For homeowners, the icon is `infobar_HomeownersLine.png`.

   Note: PolicyCenter is supplied with a default `infobar_HomeownersLine.png` icon. If you prefer, use the supplied icon. Otherwise, you can overwrite the existing icon with your new icon. Also note that in most cases, the product and policy line icons are identical, except in package products.

Adding Product and Policy Line Icons to Product Designer

You can define the icons that Product Designer displays for your new product and policy line. The image can be in PNG or GIF format. Use `.png` or `.gif` as the file extension.

Product Designer uses two icon sizes in various places. Recommended sizes for large icons is 32 pixels wide by 24 pixels high. Recommended sizes for small icons is 20 pixels wide by 16 pixels high.

To add product icons

1. Navigate to the following directory in the PolicyCenter installation:
   `modules/configuration/config/resources/productmodel/products/`*`product_name`*`/images`
2. Copy the icons to this folder using the following naming convention:

Icon size	File name	Example
Large	*`Product_Code`*`_big.png`	`Homeowners_big.png`
Small	*`Product_Code`*`.png`	`Homeowners.png`

To add policy line icons

1. Navigate to the following directory in the PolicyCenter installation:
   `modules/configuration/config/resources/productmodel/policylinepatterns/`*`product_name`*`/images`
2. Copy the icons to this folder using the following naming convention:

Icon size	File name	Example
Large	*`PolicyLine_Code`*`_big.png`	`HomeownersLine_big.png`
Small	*`PolicyLine_Code`*`.png`	`HomeownersLine.png`

Step 8: Define the Data Model for Rating in the New Line of Business

In previous steps, you designed and extended the basic data model and created the product model for your new line of business. In this step, you define and extend the cost and transaction objects for rating. To extend the data model, you must perform the following steps:

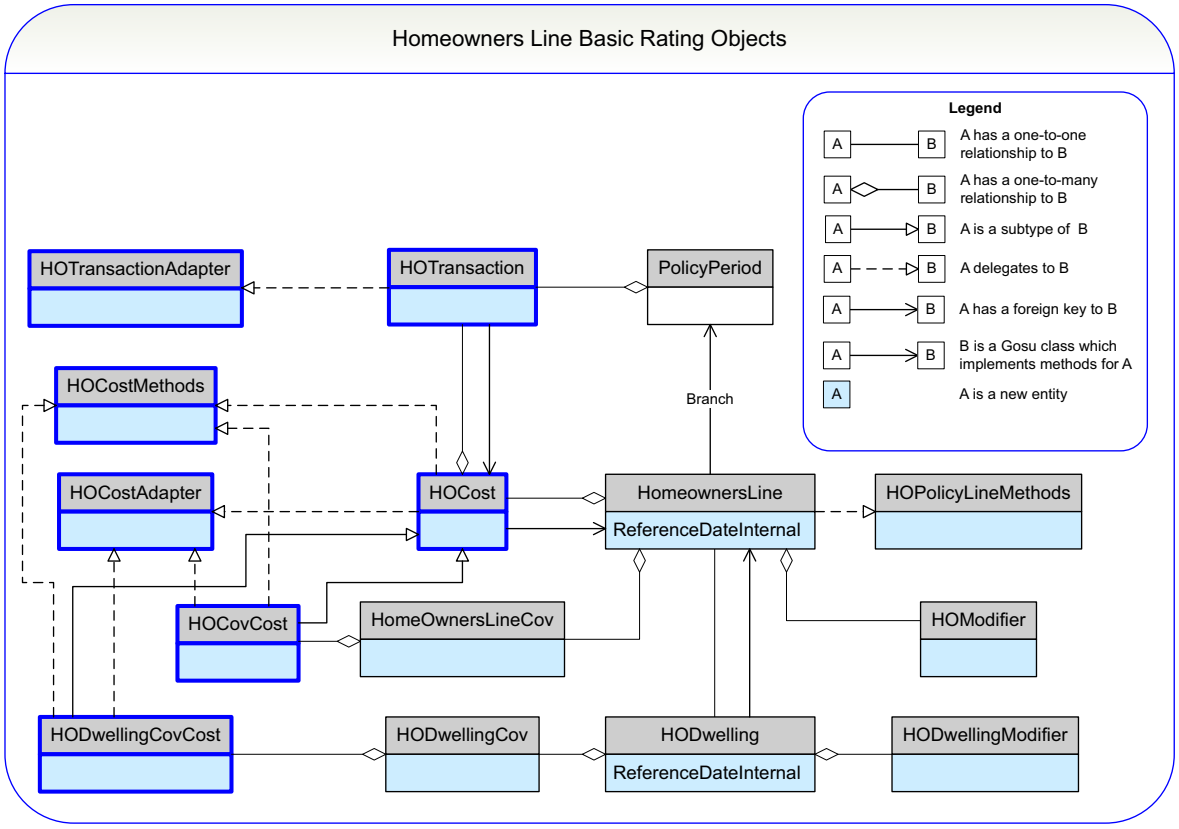
- “Defining the Data Model for Rating” on page 163
- “Creating the Abstract Cost Entity” on page 164
- “Creating the Transaction Entity” on page 165
- “Creating Cost and Transaction Adapters” on page 166
- “Creating Cost Subtypes” on page 168
- “Creating Cost Methods” on page 170
- “Reflection in the Policy Period Plugin” on page 172

See also

- “Quoting and Rating” on page 423 in the *Application Guide*
- “Rating Integration” on page 351 in the *Integration Guide*

Defining the Data Model for Rating

Define cost and transaction entities for your line of business. The following illustration shows the cost and transaction entities for the homeowners line of business. The abstract cost entity is `HOCost`. The cost subtypes are `HOCovCost` and `HODwellingCost`. The transaction entity is `HOTransaction`.



Cost Entities

The `Cost` entities contain premium values. Guidewire recommends that you record each premium entry at the smallest available level. For each cost subtype, attach each cost to the object that generates the cost and attach each cost for the shortest-priced period of time.

Create an abstract Cost delegate for the new line of business. In the preceding illustration, the `HOCost` entity represents the Cost delegate. For all objects that have costs, create `Cost` subtypes of this delegate. The cost subtypes are arrays off of coverages on the policy line and other coverages and objects that affect the premium value. The preceding illustration shows cost subtypes as `HOLineCovCost` and `HODwellingCovCost`. In most cases, `Cost` subtype entities attach to `Coverage` entities, because coverages have associated premiums. Always attach `Cost` subtypes to the most discrete items to be rated. For example, because auto liability coverage has rates for each vehicle, personal and business auto lines have `Cost` subtypes as arrays both on the coverages and on the rated vehicles.

Transaction Entities

The `Transaction` entities record the individual premium debits and credits for the current policy period. Transactions relate to costs in the same manner that journal entries relate to the general ledger. Model transactions in your line of business as an array on the `PolicyPeriod`.

Costs are subtyped and associated with a particular entity, however transactions are not subtyped. Each policy line has a single transaction type. Each transaction is associated with a particular cost item.

This topic explains how to perform the following steps:

Creating the Abstract Cost Entity

Use Studio to create an abstract cost entity for the new line of business. For homeowners, `HOCost` is the abstract entity. The cost entity implements the `Cost` delegate. Because the cost entity is a delegate, it automatically includes required columns such as the `Amount` and `Basis` fields. The cost entity also has an interface to cost methods for the line. In later steps, you create subtypes of this abstract cost class.

To create the abstract cost entity

1. Create the `HOCost` entity by using the Studio **Projects** window to navigate to **configuration** → **config** → **extensions**. Then right-click **Entity** node and select **New** → **Entity.** In the **Entity** dialog box, enter the following properties:

Property	Value
Entity	`HOCost`
Entity Type	`entity`
Desc	`A Homeowners unit of cost for a period of time, not to be further broken up.`
Table	`hocost`
Extendable	`true`
Exportable	`true`
Final	`false`

2. After you add the `HOCost` entity, add the following parameters to the entity to make it effective-dated and an abstract cost:

Property	Value
`type`	`effdated`
`effDatedBranchType`	`PolicyPeriod`
`abstract`	`true`

Note: The cost is an *abstract entity*, which means that it is accessed within the application only through its subtypes.

3. Add a `foreignkey` element to the policy line. The policy line for homeowners is `HomeownersLine`. Set the foreign key properties as follows:

Property	Value
`name`	`HomeownersLine`
`fkentity`	`HomeownersLine`
`nullok`	`false`

4. Add an `array` element to the transaction entity. For homeowners, create an array to `HOTransaction` with the following properties:

Property	Value
`name`	`Transactions`
`arrayentity`	`HOTransaction`
`cascadeDelete`	`true`

To create an array of costs on the line

1. Use Studio to open the policy line entity, and add an array to access the costs. For homeowners, navigate to **configuration** → **config** → **extensions** → **entity** and open `HomeownersLine.eti`.
2. Add an array to the line and specify the following properties:

Property	Value
`name`	`HOCosts`
`arrayentity`	`HOCost`
`cascadeDelete`	`true`

Note: At this point, you can verify your work. For instructions, see "Verifying Your Work" on page 136.

Creating the Transaction Entity

Use Studio to create a transaction entity for the new line of business. The homeowners line has a single transaction, `HOTransaction`. The transaction entity implements the `Transaction` delegate. Because the transaction entity is a delegate, it automatically includes required columns such as the `Amount` field. The transaction entity also has an interface to cost methods for the line.

To create the transaction entity

1. Create the `HOTransaction` entity by using the Studio **Projects** window to navigate to **configuration** → **config** → **extensions**. Then right-click **Entity** node and select **New** → **Entity**. In the **Entity** dialog box, enter the following properties:

Property	Value
Entity	`HOTransaction`
Entity Type	`entity`
Desc	`A transaction for the Homeowners line`
Table	`hotransaction`
Extendable	`true`
Exportable	`true`

2. After you add the `HOTransaction` entity, add the following parameters to the entity to make it effective-dated off the policy period:

Property	Value
`type`	`effdated`
`effDatedBranchType`	`PolicyPeriod`

3. Add a `foreignkey` element to the abstract cost entity. The abstract cost entity for homeowners is `HOCost`. Set the foreign key properties as follows:

Property	Value
`name`	`HOCost`
`fkentity`	`HOCost`
`desc`	`The cost this transaction modifies`
`nonEffDated`	`true`
`nullok`	`false`

To create an array of transaction on the policy period

Add the transactions as an array on the policy period entity. In the default implementation, an extension to the policy period entity already contains arrays to transactions for existing lines.

1. Use Studio to navigate to **configuration** → **config** → **extensions** → **entity** and open `PolicyPeriod.etx`.
2. Add an `array` of the new transaction entity to this extension. For homeowners, add `HOTransaction` with the following properties:

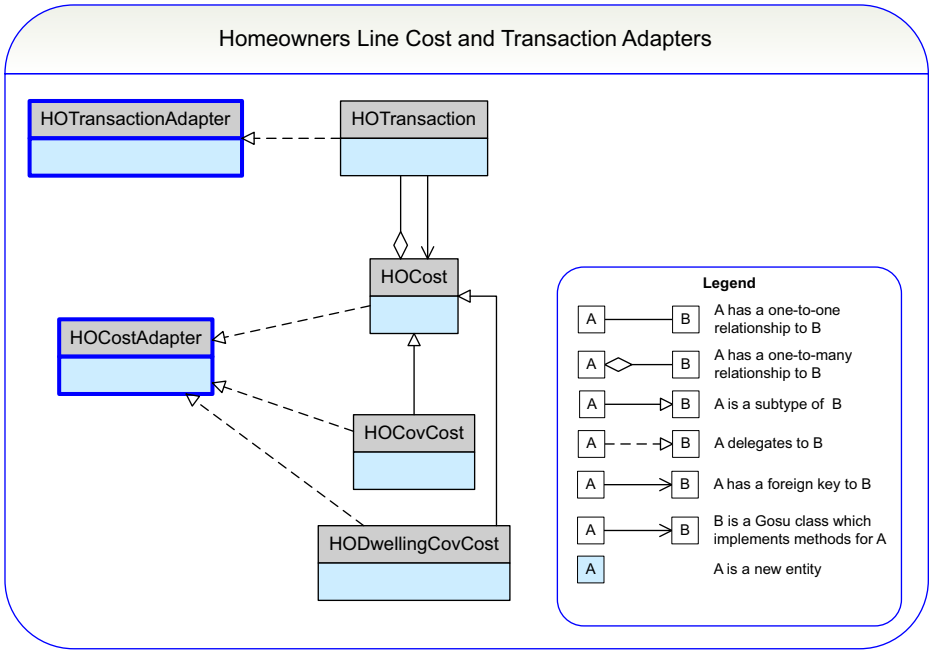
Property	Value
`name`	`HOTransactions`
`arrayentity`	`HOTransaction`
`cascadeDelete`	`true`

Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Creating Cost and Transaction Adapters

Use Studio to create cost and transaction adapters for the new line of business. The cost adapter defines how to create a transaction from the cost. The same cost adapter is used for all costs regardless of the subtype. So only one cost adapter is required for a policy line. The abstract cost entity and the cost subtypes delegate to the cost adapter class.

The transaction adapter defines how to access the cost from the transaction.



To declare the cost adapter in the abstract cost

Use Studio to add an `implementsEntity` element to the abstract cost entity.

1. For the homeowners line, navigate to **configuration** → **config** → **extensions** → **entity** and open `HOCost.eti`.
2. Add an `implementsEntity` element with the following properties:

Property	Value
`name`	`Cost`
`adapter`	`gw.log.ho.financials.HOCostAdapter`

This element references the Gosu class that defines the adapter methods and properties.

To add a cost adapter

1. In Studio, navigate in **configuration** → **gsrc** to the `gw.lob.`*`line`* package. For homeowners, navigate to `gw.lob.ho`.
2. Right-click the *`line`* node, and then select **New** → **Package**. In the **New Package** dialog box, enter a new package name of `financials`.
3. Right-click the new `financials` package, and then select **New** → **Gosu Class**. In the **New Gosu Class** dialog box, enter the name of the adapter. You specified the name of the adapter in the `implementsEntity` element of the abstract cost entity. For homeowners, enter `HOCostAdapter`.
4. In the new Gosu class, write a constructor similar to the following for the `gw.lob.ho.HOCostAdapter`:

```
package gw.lob.ho.financials
uses gw.api.domain.financials.CostAdapter

class HOCostAdapter implements CostAdapter {
  var _owner : entity.HOCost
  construct(owner : entity.HOCost)
  {
    _owner = owner
  }
}
```

This code produces a Gosu error in the class statement that you fix in the next step.

5. Place the insertion point at the error (after `CostAdapter`) and press `Alt-Enter`. Click the **ImplementMethods** command that pops up to open the **Select Methods to Implement** dialog box. With all listed methods selected, click **OK** to insert the empty methods into your code.
6. Write code for the new methods and properties. In most cases, the code for each method is a single line. Examine the cost adapters for other lines of business as an example.

To declare the transaction adapter in the transaction entity

Use Studio to add an `implementsEntity` element to the transaction entity.

1. For the homeowners line, navigate to **configuration** → **config** → **extensions** → **entity** and open `HOTransaction.eti`.
2. Add an `implementsEntity` element with the following properties:

Property	Value
`name`	`Transaction`
`adapter`	`gw.log.ho.financials.HOTransactionAdapter`

This element references the Gosu class that defines the adapter methods and properties.

To add a transaction adapter

1. In Studio, navigate in **configuration** → **gsrc** to the `gw.lob.`*`line`*`.financials` package. For homeowners, navigate to `gw.lob.ho.financials`.
2. Right-click the `financials` node, and then select **New** → **Gosu Class**. In the **New Gosu Class** dialog box, enter the name of the adapter. You specified the name of the adapter in the `implementsEntity` element of the transaction entity. For homeowners, enter `HOTransactionAdapter`.
3. In the new Gosu class, write a constructor similar to the following for the `gw.lob.ho.HOTransactionAdapter`:

```
package gw.lob.ho.financials
uses gw.api.domain.financials.TransactionAdapter
class HOTransactionAdapter implements TransactionAdapter {

  var _owner : entity.HOTransaction

  construct(owner : HOTransaction) {
    _owner = owner
  }

}
```

This code produces a Gosu error in the class statement that you fix in the next step.

4. Place the insertion point at the error (after `TransactionAdapter`) and press `Alt-Enter`. Click the **Implement-Methods** command that pops up to open the **Select Methods to Implement** dialog box. With all listed methods selected, click **OK** to insert the empty methods into your code.
5. Write code for the new methods and properties. In most cases, the code for each method is a single line. Examine the transaction adapters for other lines of business as an example.

   Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

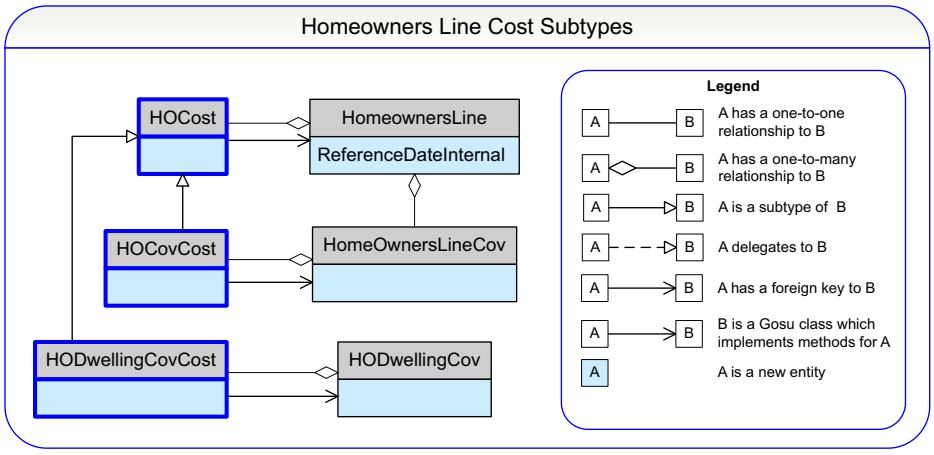
Creating Cost Subtypes

Use Studio to create `Cost` subtype entities for the new line of business. The cost entities are subtypes of the abstract `Cost` class. Homeowners has two cost entities that are subtypes of `HOCost`:

- `HOCovCost` – Cost information for the homeowners line.

- `HODwellingCovCost` – Cost information on the dwelling.

In homeowners, all cost subtypes are for coverages. In other lines of business, cost subtypes can be on other types of objects.



To define a cost subtype

1. Create the `HOCovCost` entity by using the Studio **Projects** window to navigate to **configuration** → **config** → **extensions**. Then right-click **Entity** node and select **New** → **Entity.** In the **Entity** dialog box, enter the following properties:

Property	Value
Entity	`HOCovCost`
Entity Type	`subtype`
Desc	`A taxable unit of cost for a period of time, for a Homeowners coverage`
Supertype	`HOCost`

2. Add a `foreignkey` element to the coverage entity. The coverage entity for homeowners is `HomeownersLineCov`. Set the foreign key properties as follows:

Property	Value
`name`	`HomeownersLineCov`
`fkentity`	`HomeownersLineCov`
`nullok`	`false`

To add the cost subtype as an array on the coverage

Open the coverage entity to which the costs apply. Add the cost subtype as an array on this entity.

1. Use Studio to navigate to **configuration** → **config** → **extensions** → **entity** and open `HomeownersLineCov.eti`.

2. Add an `array` element to the entity with the following properties:

Property	Value
`name`	`Costs`
`arrayentity`	`HOCovCost`
`cascadeDelete`	`true`

Note: At this point, you can verify your work. For instructions, see "Verifying Your Work" on page 136.

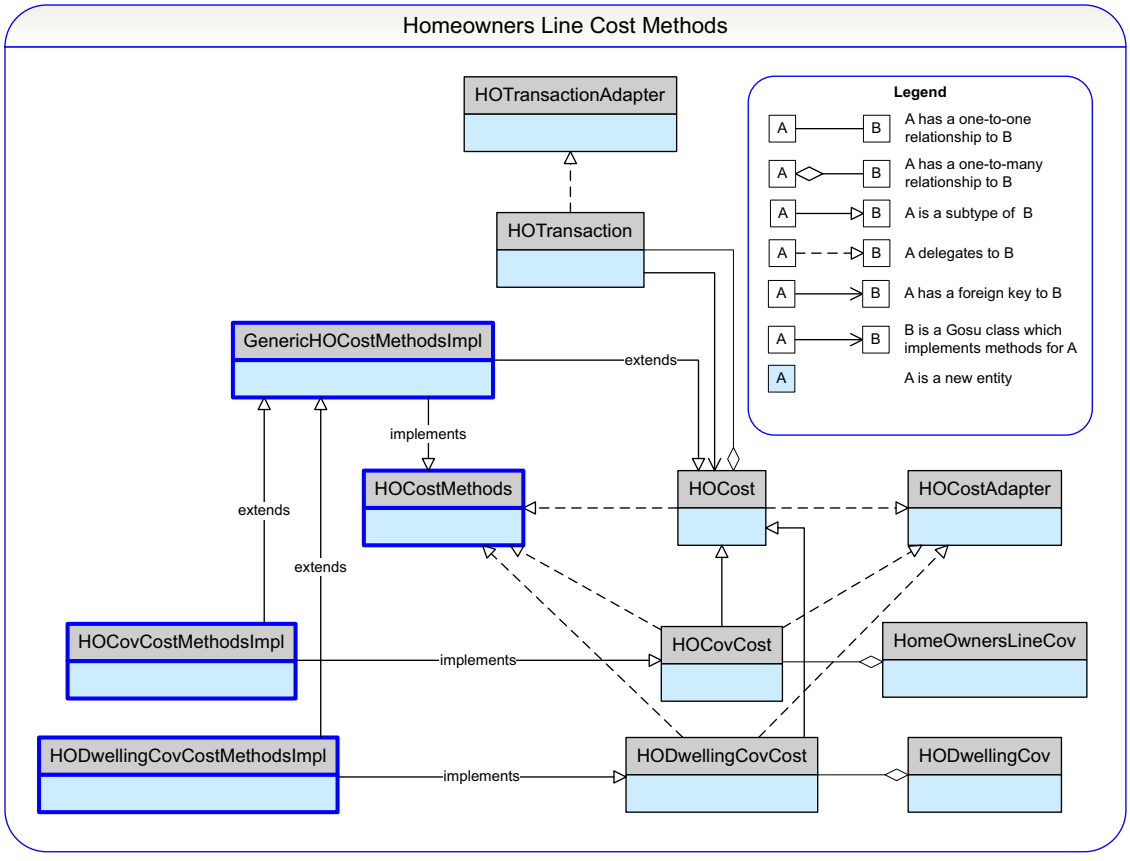
Creating Cost Methods

Cost methods implement the methods and properties for the cost. Often, cost methods are used to get properties on the cost. The abstract cost and cost subtypes delegate to the cost methods for the line.

The base configuration provides one interface for the cost methods in each policy line. In homeowners, you define the interface in the `HOCostMethods` class. The abstract cost and cost subtypes delegate to this interface. The line level interface defines cost properties that are required for any of the specific cost types. Typical required cost properties include the coverage, any coverables other than the policy line, and other properties required in the user interface, such as properties for the quote page. Properties can include the policy location or jurisdiction.

Cost method implementation classes implement the cost method interface. A generic cost method implementation implements the cost interface and implements line level cost methods. For homeowners, you define the `GenericHOCostMethodsImpl` class to implement the cost interface, `HOCostMethods`.

Each cost subtype has a cost method implementation class that extends the generic class. For homeowners, you define the cost method implementation classes as `HoCovCostMethodsImpl` and `HODwellingCovCostMethodImpl`.



Create the line level interface first, then create the generic cost method implementation. Finally, create the cost specific extensions.

Cost Methods

To declare the interface for the cost methods

1. In Studio, open the cost entity. For homeowners, open `HOCost.eti` entity in **configuration** → **config** → **extensions** → **entity.**
2. Declare the interface. For homeowners, add the `HOCostMethods` interface by adding an `implementsInterface` element to `HOCost` with the following properties:

Property	Value
iface	`gw.lob.ho.financials.HOCostMethods`
impl	`gw.lob.ho.financials.GenericHOCostMethodsImpl`

To create the line level interface

1. In Studio, navigate in **configuration** → **gsrc** to the `gw.lob.`*`line`*`.financials` package. For homeowners, navigate to `gw.lob.ho.financials`.
2. Right-click the `financials` node, and then select **New** → **Gosu Class**. In the **New Gosu Class** dialog box, enter the name of the interface. For homeowners, enter `HOCostMethods`.
3. Add code for the line level interface. For homeowners, add the following code for the interface for `HOCostMethods`:

```
package gw.lob.ho.financials

interface HOCostMethods {
  property get Coverage() : Coverage
property get State() : Jurisdiction
}
```

To create the generic cost method implementation

1. In Studio, navigate in **configuration** → **gsrc** to the `gw.lob.`*`line`*`.financials` package. For homeowners, navigate to `gw.lob.ho.financials`.
2. Right-click the `financials` node, and then select **New** → **Gosu Class**. In the **New Gosu Class** dialog box, enter the name of the interface. You For homeowners, enter `GenericHOCostMethodsImpl`.
3. In the new Gosu class, write a constructor similar to the following for the `GenericHOCostMethodsImpl`:

```
package gw.lob.ho.financials

class GenericHOCostMethodsImpl<T extends HOCost> implements HOCostMethods  {
  protected var _owner : T as readonly Cost
  construct(owner : T) {
     _owner = owner
  }
}
```

   This code produces a Gosu error in the class statement that you fix in the next step.
4. Place the insertion point at the error (after `HOCostMethods`) and press `Alt-Enter`. Click the **ImplementMethods** command that pops up to open the **Select Methods to Implement** dialog box. With all listed methods selected, click **OK** to insert the empty methods into your code.
5. Write code for the new methods and properties. In most cases, the code for each method is a single line. Examine the transaction adapters for other lines of business as an example.

   Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Coverage Cost Methods

This topic describes how to create the coverage cost methods for the line of business. In homeowners, the coverage cost methods are `HOCovCostMethodsImpl` and `HODwellingCovCostMethodsImpl`. Instructions are provide for `HOCovCostMethodsImpl`.

To declare the coverage cost method implementation

1. In Studio, open the coverage cost entity. For homeowners, open `HOCovCost.eti` entity in **configuration** → **config** → **extensions** → **entity.**
2. Declare the interface for the cost method implementation. For homeowners, add the `HOCostMethods` implementation by adding an `implementsInterface` element to `HOCovCost` with the following properties:

Property	Value
`iface`	`gw.lob.ho.financials.HOCostMethods`
`impl`	`gw.lob.ho.financials.HOCovCostMethodsImpl`

To create the coverage cost method implementation for the line

1. In Studio, navigate in **configuration** → **gsrc** to the `gw.lob.`*`line`*`.financials` package. For homeowners, navigate to `gw.lob.ho.financials`.
2. Right-click the `financials` node, and then select **New** → **Gosu Class**. In the **New Gosu Class** dialog box, enter the name of the interface. For homeowners, enter `HOCovCostMethodsImpl`.
3. In the new Gosu class, write a constructor similar to the following for the `gw.lob.ho.HOCovCostMethodsImpl`:

```
package gw.lob.ho.financials

class HOCovCostMethodsImpl extends GenericHOCostMethodsImpl<HOCovCost> {

construct(owner : HOCovCost) {
    super( owner )
  }
...
  }
```

4. Override property values as necessary. For homeowners, the code is:

```
package gw.lob.ho.financials

class HOCovCostMethodsImpl extends GenericHOCostMethodsImpl<HOCovCost> {

  construct(owner : HOCovCost) {
    super( owner )
  }

  override property get Coverage() : Coverage  {
    return Cost.HomeownersLineCov
  }

  override property get State() : Jurisdiction {
    return _owner.Jurisdiction.State
  }
}
```

Note: At this point, you can verify your work. For instructions, see “Verifying Your Work” on page 136.

Reflection in the Policy Period Plugin

When PolicyCenter creates a new policy branch, it does not copy all costs and transactions from the original policy branch. Instead, PolicyCenter uses reflection to determine which transactions are copied. The code that performs this reflection is located in the `PolicyPeriodPlugin` class. Typically, adding a new line of business requires no changes to this code.

Step 9: Design the User Interface for the New Line of Business

In previous topics, you designed the data model, product model, and rating for a new line of business. The remaining step is to use Studio to design the user interface for your new line. As you design the user interface, use the lines of business provided in the PolicyCenter base configuration as a guide.

See also

- "Using the PCF Editor" on page 289 in the *Configuration Guide*
- "Introduction to Page Configuration" on page 303 in the *Configuration Guide*

PCF Files and Folders for a Line of Business

By convention, the PCF files for lines of business in the base configuration have a similar file structure and similar organization within each file. Guidewire suggests that you develop your line of business using this file structure and organization.

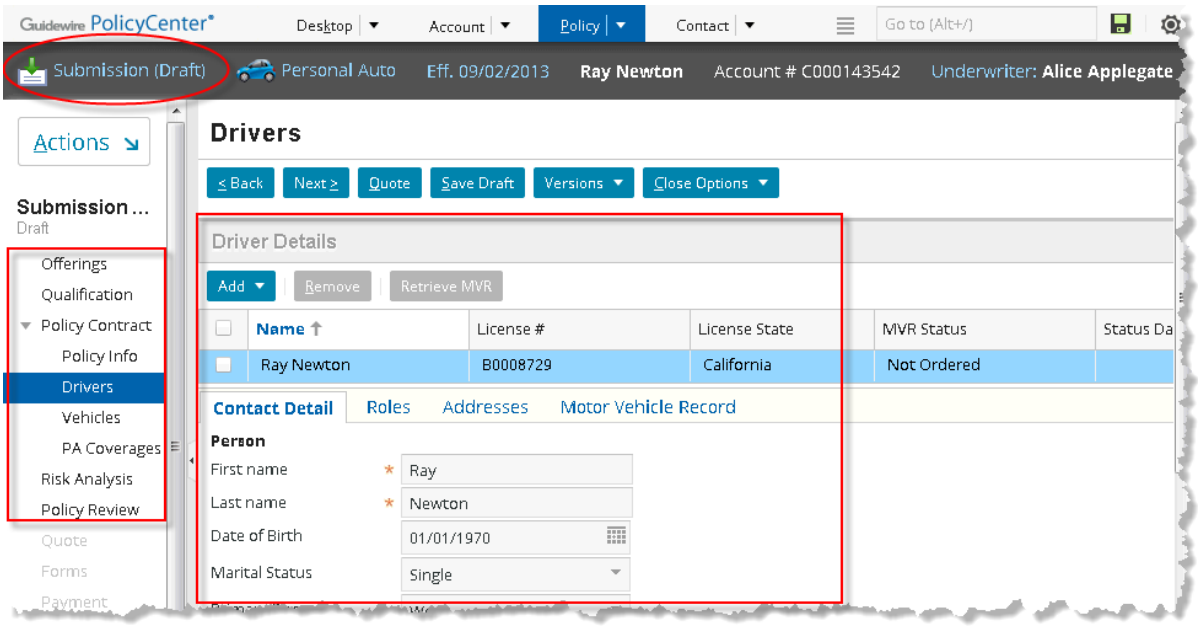
> **Note:** Multi-line products, such as commercial package policy, have a somewhat different structure. For more information, see "Step 4: Create the Policy File Screens" on page 187.

Each line of business in **Page Configuration** → **pcf** → **line** has the following folders:

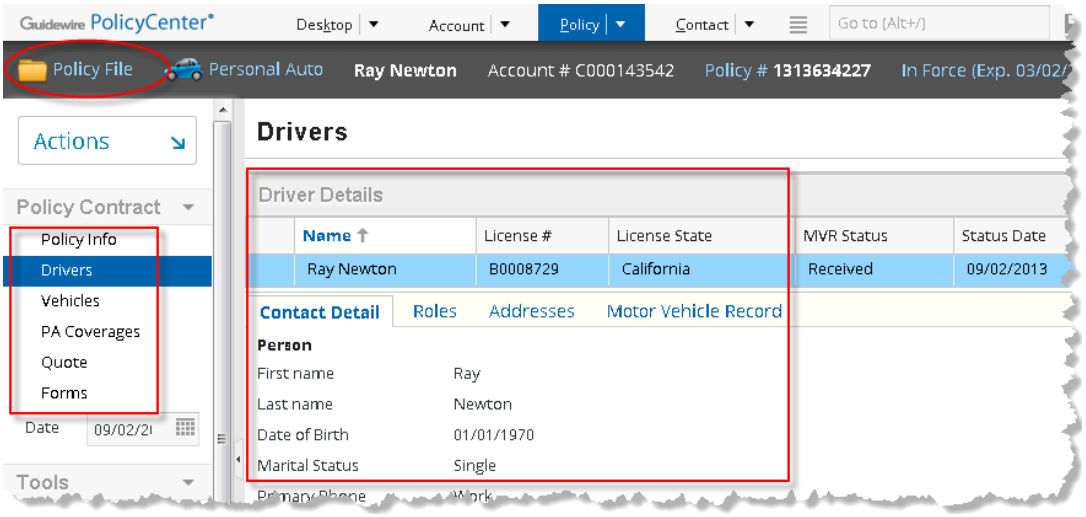
- **job** – Single PCF file for steps used by all wizards—submission, policy change, and others. For example, the wizard for personal auto is `LineWizardStepSet.PersonalAuto`. Wizard steps appear along the left side of the PolicyCenter window when a job is in process.
- **policy** – PCF files for the line of business. Used in both jobs and in the policy file.
- **policyfile** – PCF files for interacting with the policy file within the wizard. The main PCF file is the menu item set. For example, the main PCF file for personal auto is `PolicyMenuItemSet.PersonalAuto`.

  For more information about the policy file, see "Policy File" on page 305 in the *Application Guide*.

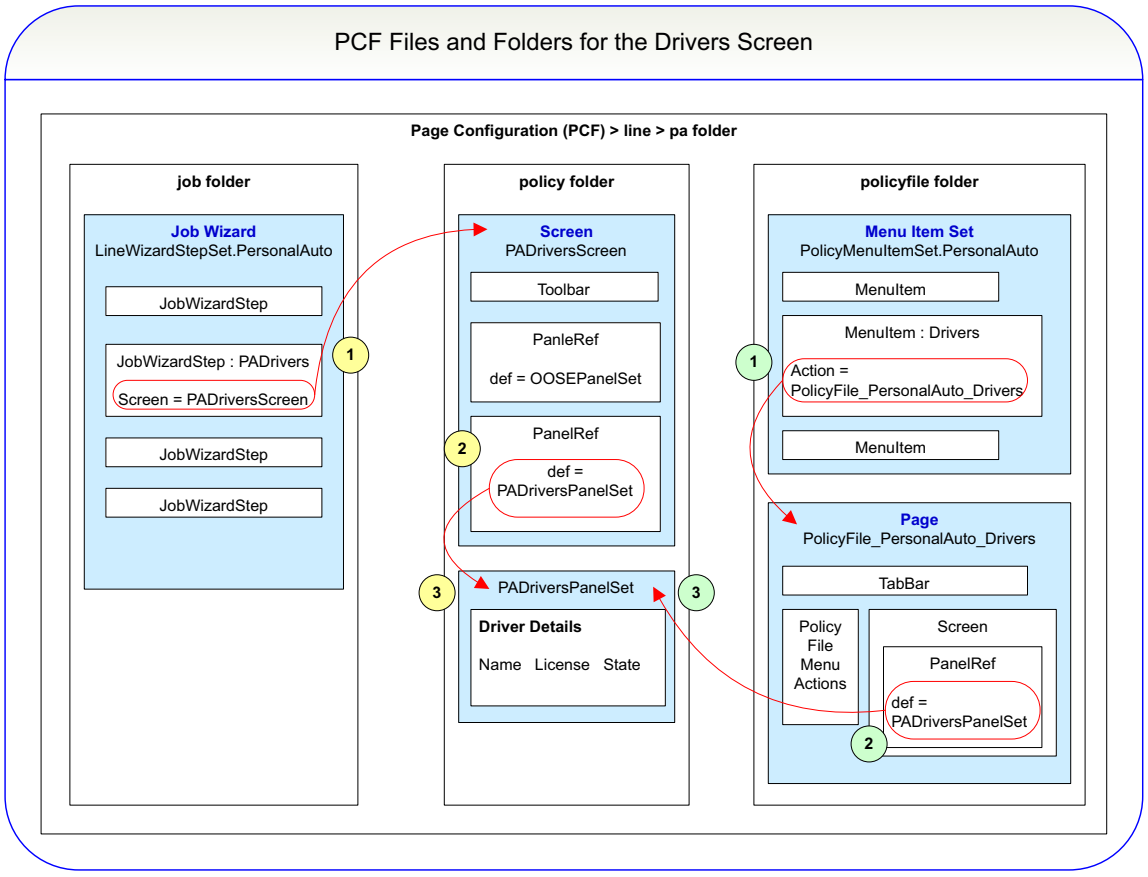
The following illustration shows the **Drivers** screen in a submission job. Jobs use the PCF file for the Wizard Step Set in the job folder. The wizard steps appear in the red box along the left side of the screen. The **Drivers** screen displays the **Driver Details** panel set.



The following illustration shows the **Drivers** screen in the policy file. The policy file uses the menu item set in the **policyfile** folder. The menu items appear in the red box along the left side of the screen. The **Drivers** screen for both the policy file and the job display the same **Driver Details** panel set.



The following illustration shows the similar file organization for both the wizard and the policy file. The **Driver Details** panel set is shared by both the wizard and the policy file.



In this illustration, the numbered circles correspond to the following items in the job wizard and policy file:

Ref. No.	Job wizard (yellow reference numbers)	Policy file (green reference numbers)
1	In the job wizard, `JobWizardStep` points to a screen, `PADriversScreen`.	In the menu item set, a `MenuItem` points to a page, `PolicyFile_PersonalAuto_Drivers`.
2	In the `PADrivers` screen, a `PanelRef` displays `PADriversPanelSet`.	In the `PolicyFile_PersonalAuto_Drivers` page, a `PanelRef` displays `PADriversPanelSet`.
3	`PADriversPanelSet` displays **Driver Details** such as **Name**, **License**, and **State**.	`PADriversPanelSet` displays **Driver Details** such as **Name**, **License**, and **State**.

Creating the Wizard for Your Line of Business

Use personal auto or another line of business as a model to create the PCF files for your wizard and wizard steps.

The submission, issuance, policy change, renewal, and rewrite jobs have a `WizardStepSetRef` that displays the modal container for each product. You must define the modal container for your new product. For example, the modal container for personal auto is `LineWizardStepSet.PersonalAuto`. Each `LineWizardStepSet` contains wizard steps specific to the product line.

To create the line wizard step set for your new product

1. In Studio, navigate to **configuration** → **config** → **Page Configuration** → **pcf**. Right-click the **line** node, and then select **New** → **PCF folder**. In the **New Package** dialog box, enter the abbreviation for the line. For homeowners, enter `ho`.
2. Right-click the new **ho** folder, and then select **New** → **PCF folder**. In the **New Package** dialog box, enter `job`.
3. Right-click the new **job** folder and select **New** → **PCF file**. In the **PCF File** dialog box, specify the following properties:

Property	Value
File name	`LineWizardStepSet`
File type	`Wizard Steps`
Mode	`Homeowners`

   Studio creates the line wizard step set. For homeowners, it creates `LineWizardStepSet.Homeowners`. Because the PCF file has missing information, it appears red in Studio.
4. In the Studio PCF editor, select the line wizard step set.
5. In the **Required Variables** tab, add the required variables. Examine another product, such as personal auto, as an example.

   After you add the required variables, the line wizard step set changes from red to white indicating that there are no errors.

To verify your work

1. In Studio, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **job** → **submission** and open `SubmissionWizard`.
2. In the `WizardStepSetRef` next to `JobWizardStep: PolicyInfo`, view the **Shared section mode** drop-down list. Verify that your new product appears in this list. For homeowners, verify that **Homeowners** appears in the list.

Completing the Line Wizard Step Set for Your Line of Business

Complete the line wizard step set by adding wizard steps for the line of business. Use personal auto as an example. Notice that the wizard has `JobWizardStep` elements for common steps such as the **Risk Analysis** and **Policy Review** screens. These steps are common to all lines of business. The line wizard step set includes only those wizard steps specific to the line of business.

To complete the line wizard step set

1. In the line wizard step set for your new line of business, drag a **JobWizardStep** onto the **LineWizardStepSet**.

   The **WizardStepGroup** turns red because there is missing information.

2. On the **Properties** tab, enter values for properties using personal auto as an example. For the screen property, enter the name of the screen to display. The name you enter can be a screen that already exists. For example, the `LocationsScreen` is used by several lines of business. If the screen does not exist, create the screen as explained in "Creating the Policy Screens" on page 176.

3. Add other **JobWizardStep** elements as needed by your line of business.

Creating the Policy Screens

Create the PCF files for the **policy** folder. This folder contains PCF files for the line of business. Some of these PCF files are used in both jobs and the policy file. Create the following types of PCF files:

- Wizard step screens referenced by the line wizard step set that you created in "Completing the Line Wizard Step Set for Your Line of Business" on page 176.
- Line-specific PCF files for **Shared section mode** display.
  - If you reused an existing screen in a wizard step set, add any needed line-specific information to that screen. For example, the `LocationsScreen` has a **Shared section mode** that displays specific information for businessowners and different specific information workers' compensation.
  - Common screens, such as **Risk Analysis** and **Policy Review.** Common screens often have a **Shared section mode** that displays line-specific information.

Create these files in the **policy** folder located in **configuration** → **config** → **Page Configuration** → **pcf** → **line** → *`line`*. For example, for homeowners, create the files in **configuration** → **config** → **Page Configuration** → **pcf** → **line** → **ho** → **policy.** Examine the PCF files from other lines of business as an example.

Creating the Policy File Screens

Create the PCF files for the policy file in the **policyfile** folder located in **configuration** → **config** → **Page Configuration** → **pcf** → **line** → *`line`*. For example, for homeowners, create the files in **configuration** → **config** → **Page Configuration** → **pcf** → **line** → **ho** → **policyfile**. This folder contains PCF files for viewing the policy file. The main PCF file is the menu item set. For example, the main PCF file for personal auto is `PolicyMenuItemSet.PersonalAuto`. The menu item set is similar to the wizard step set, and often contains similar menu items.

Create the following types of PCF files:

- A menu item set for the product such as `PolicyMenuItemSet.PersonalAuto`
- Pages for displaying menu items.

  Note: Some menu items reference pages that are used by multiple lines of business. These pages are located in **configuration** → **config** → **Page Configuration** → **pcf** → **policyfile.** These common pages often have a **Shared section mode** for displaying line-specific information. Create any needed PCF files for the line-specific information. An example of a shared file is `PolicyFile_PolicyInfo`. Create the PCF files in the same directory as the other line-specific files.

Step 10: Set ClaimCenter Typelist Generator Options (Optional)

After you define a product in PolicyCenter, you can instruct ClaimCenter to use the relevant information from PolicyCenter for its line of business codes, policy types, and coverage codes. Guidewire provides the ClaimCenter Typelist Generator tool to manage this process.

For information on using this tool and setting its options, see “PolicyCenter Product Model Import into ClaimCenter” on page 510 in the *Integration Guide*.

Lines of Business – Advanced Topics

This topic describes and provides links to advanced topics related to configuring or adding new lines of business.

Topic	See...
Extending the data model	• “The PolicyCenter Data Model” on page 149 in the *Configuration Guide* • “Modifying the Base Data Model” on page 211 in the *Configuration Guide*
Multi-line product	• “Creating a Multi-line Product” on page 179
Premium audit	• “Adding Premium Audit to a Line of Business” on page 189
Copying data	• “Configuring Copy Data in a Line of Business” on page 197
Locations	• “Adding Locations to a Line of Business” on page 207
Policy differences	“Customizing Differences for New Lines of Business” on page 447 in the *Integration Guide*

Writing Modular Code for Lines of Business

Guidewire recommends that you define line-of-business code in the policy-line-methods classes. Avoid putting line-of-business code in generic locations such as the rule sets, plugins, and non-line-of-business PCF files and Gosu classes. This recommendation is intended to make upgrade easier by grouping line-of-business changes in the `gw.lob` package.

For an example, see “Guidelines for Modularizing Line-of-business Code” on page 443 in the *Configuration Guide*.

chapter 14

Creating a Multi-line Product

This topic describes how to create a product that includes more than one line of business. In the base configuration, the commercial package policy is a multi-line product that includes commercial property, general liability, and inland marine lines. You can examine commercial package as an example when you develop your own multi-line product.

This topic provides step-by-step instructions to create a personal multi-line product that includes personal auto and general liability lines. The product is called Personal Package.

This topic includes:

Step 1: Define the Multi-line Product

The first step in creating a multi-line product is to define the product in Product Designer.

To define the product

1. In Product Designer, create a new change list. For example, create a change list named **Personal Package**. Be sure the change list is associated with the correct workspace—the one that represents the intended PolicyCenter instance.
2. Navigate to the **Products** page, and then click **Add**.

3. In the **Add Product** dialog box, supply the following details:

Property	Value
Code	`PersonalPackage`
Name	`Personal Package`
Default Policy Term	`6 months`
Policy Line	`Personal Auto Line [PersonalAutoLine]`
Product Type	`Personal`
Account Type	`Any`

Note: While you are adding a new product, you can select only one policy line and one default policy term. After you complete the initial product definition, you can add additional product lines and policy terms, along with other product details.

Product Designer displays the new product in the Product home page titled **Personal Package [PersonalPackage]**.

4. In the upper portion of the Product home page, enter the following details:

Property	Value
Description	`Personal Package Policy including Personal Auto and General Liability lines.`
Abbreviation	`ppp`

5. Under **Policy Lines**, click **Add**. In the pop-up list that appears, select **General Liability Line [GLLine]**. Your package now contains the two needed lines of business.
6. Under **Policy Terms**, add the other terms that the product allows. For personal package, add **Annual**. Your package now allows the two needed term types. In the upper portion of the page, be sure that the **Default Policy Term** is set to `6 months`.

Step 2: Design the Wizard for Your Multi-line Product

After you define the new product, you must design its user interface wizard. The wizard can reuse many of the policy line screens from the base configuration.

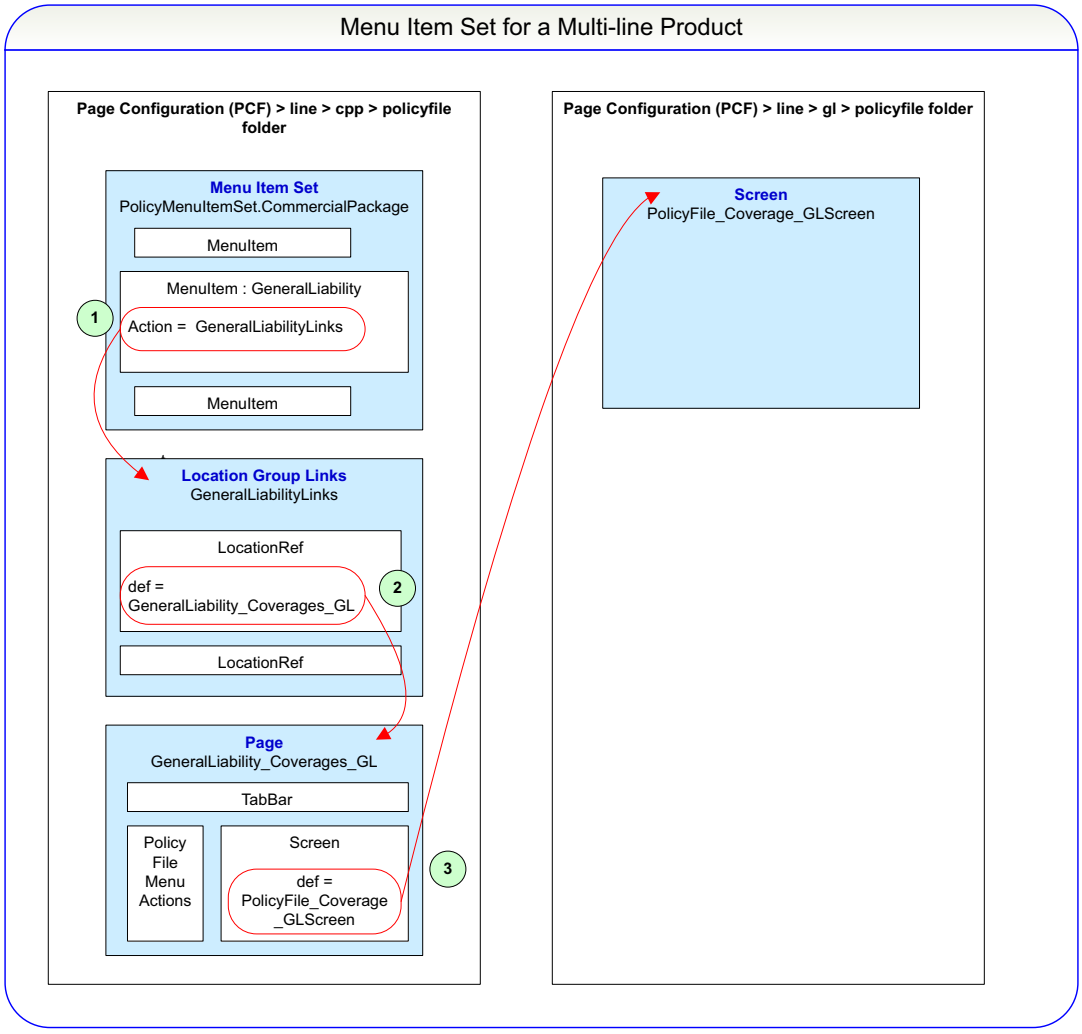
You can examine the PCF files for each product by navigating in Studio to **configuration** → **config** → **Page Configuration** → **pcf** → **line**. Within the **line** node is a node for each product. The nodes are named with the abbreviation of the product. Each line node contains three folders. For a multi-line product, these folders contain the following:

- **job** – Contains a PCF file that has the `LineWizardStepSet` for the multi-line product. In a multi-line product, the wizard steps typically include the following:
  - **Line selector** – A job wizard step that enables you to select the policy lines to include in a policy.
  - **Multi-line locations** – A job wizard step that enables you to define locations that apply at the package level.
  - **One or more lines** – One of more wizard step group widgets for each line in the product.
  - **Modifiers** – A job wizard step for modifiers that apply at the package level.

  You can add any other steps that your multi-line product needs.
- **policy** – Contains PCF files for the line of business. The PCF files can be used in both jobs and the policy file. In a multi-line product, this node contains:
  - **Policy review screens** – Enable you to review all policy lines in the package.
  - **Rating screens** – Display rating information for all policy lines in the package.

- **policyfile** – Contains PCF files for viewing the policy file.
  - **Menu item set** – The main PCF file, similar to the wizard for jobs. For example, the menu item set for commercial property is `PolicyMenuItemSet.CommercialProperty`. In a multi-line product, the menu item widgets in a menu item set point to location group links pages.
  - **Location group links and pages** – Each link points to a page that displays screens for the individual line of business. These pages display PCF files from the **policy** folder.

The menu item sets for single and multi-line product differ. In a multi-line product, each menu item points to a location group links file, as shown in the following illustration.



1. In the menu item set, a menu item points to a location group links file. The links file contains a series of `LocationRef` widgets.
2. A `LocationRef` points to a page in the policy file.
3. The page points to a screen in the *`line`* → **policyfile** folder.

See also

- "PCF Files and Folders for a Line of Business" on page 173

Adding the Line Wizard Step Set for Your Multi-line Product

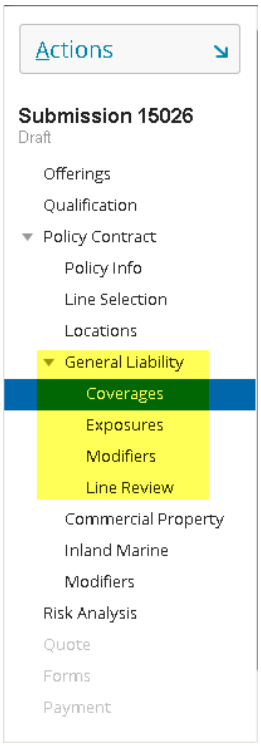
The submission, issuance, policy change, renewal, and rewrite jobs have a `WizardStepSetRef` that displays the modal container for each product. You must define the modal container for your new product. For example, modal container for personal auto is `LineWizardStepSet.PersonalAuto`. Each `LineWizardStepSet` contains wizard steps specific to the product line.

For instructions, see "Creating the Wizard for Your Line of Business" on page 175. For personal package, name the file `LineWizardStepSet.PersonalPackage`.

Completing the Line Wizard Step Set for Your Multi-line Product

For each line, add wizard steps to your wizard step set. Use commercial package as an example. You can use wizard step groups to group steps for each line of business.

In the following illustration of a commercial property submission, the area highlighted in yellow shows that **General Liability**, **Commercial Property**, and **Inland Marine** are wizard step group elements. Each wizard step group expands to reveal its wizard steps as you navigate into them. In the illustration, **General Liability** expands to reveal several indented wizard steps.



To complete the line wizard step set

1. In Studio, drag a **Wizard Step Group** widget onto the `LineWizardStepSet`.

   The `WizardStepGroup` turns red because there is missing information.

2. On the **Properties** tab, enter the following:

Property	Value
`id`	For example, `PAWizardStepGroup` for personal auto.
`label`	Name of a display key that contains the label. Create the display key by navigating to **configuration** → **config** → **Localizations** → **en_US** and opening `display.properties`. For personal auto, create the display key `Web.LineWizardMenu.PersonalPackage.PersonalAuto = Personal Auto`. After creating the new display key, enter `displaykey.Web.LineWizardMenu.PersonalPackage.PersonalAuto` in the `label` field.
`collapseIfNotCurrent`	`true`. The wizard step group appears collapsed in PolicyCenter when it is not the current group.
`visible`	Script that displays the object based on whether the line exists in the policy period. For example: `policyPeriod.PersonalAutoLineExists`

3. Add **JobWizardStep** widgets. In many cases, you can copy the widgets directly from the line wizard step set for the line. Using personal auto as an example:
   a. Navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **line** → **pa** → **job** and open `LineWizardStepSet.PersonalAuto`.
   b. Copy the `JobWizardStep` widgets from personal auto to `PAWizardStepGroup` in `LineWizardStepSet.PersonalPackage`.
   c. Make any needed changes.

   When you have correctly specified all required fields, the `WizardStepGroup` widget changes from red to white.

   Note: Later, after you define line review screens, you add job wizard step widgets for this screen.

Step 3: Create the Policy Screens

This topic provides step-by-step instructions to create the PCF files which allow the user to select policy lines, review each policy line, and review the quote. These files are located in Studio in **configuration** → **config** → **Page Configuration** → **pcf** → **line** → *`line`* → **policy.**

This topic provides instructions for the following tasks:

- Adding a Line Selection Screen for a Multi-line Product
- Adding Line Review Screens for a Multi-line Product
- Adding Quote Screens for a Multi-line Product

Adding a Line Selection Screen for a Multi-line Product

A **Line Selection** screen enables you to select the lines of business to include in an instance of a multi-line product. Examine the commercial package policy line selection screen, `CPPLineSelectionScreen.pcf`, as an example.

To add a line selection screen

1. In Studio, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **line**.

2. Right-click the folder for your new multi-line product and select **New** → **PCF folder.** For personal package, right-click the **ppp** folder, and then select **New** → **PCF folder.** In the **New Package** dialog box, enter `policy`.

   Note: The Studio **Project** window displays only the last folder in the path unless you disable the **Compact Empty Middle Packages** feature in the title bar context menu. This menu appears when you right click the **Project** window title bar.

3. Right-click the new **policy** folder, and then select **New** → **PCF File**. In the **PCF File** dialog box, specify the following properties:

Property	Value
File name	`PPPLineSelection`
File type	`Screen`
Mode	(blank)

   Studio creates `PPPLineSelectionScreen.pcf`.

4. On the **Required Variables** tab for the screen, add required variables. Examine `CPPLineSelectionScreen.pcf` as an example.

5. Create a toolbar or copy and paste the `Toolbar` from commercial package.

6. Copy and paste the `DetailViewPanel` from commercial package. After copying, change the `id` property under **Basic properties** on the **Properties** tab. Remove the widgets labeled **Coverage Part Selection** and **Package Risk Type**.

7. Customize this screen to meet your requirements.

Adding Line Review Screens for a Multi-line Product

Create a **Line Review** screen for each line of business. The line review screen appears last in the line wizard step set for each product line.

To add a line review summary detail view

1. In Studio, navigate to the new product line in **configuration** → **config** → **Page Configuration** → **pcf** → **line**. For personal package, navigate to **pcf** → **line** → **ppp**.

2. Within the **ppp** folder, create a new PCF file in the **policy** folder with the following properties:

Property	Value
File name	`PPPLineReviewSummary`
File type	`Detail View`
Mode	(blank)

   Studio creates `PPPLineReviewSummaryDV.pcf`.

3. On the **Required Variables** tab for the screen, add required variables. Examine `CPPLineReviewSummaryDV.pcf` as an example.

4. On the **Variables** tab for the screen, add variables. Examine `CPPLineReviewSummaryDV.pcf` as an example.

5. Copy and paste the two `InputColumn` widgets from `CPPLineReviewSummaryDV.pcf`.

   One input column displays information about the primary named insured. The other input column displays information about the product and policy line.

To add a line review screen that references the detail view

1. In Studio, navigate to your new product line in **configuration** → **config** → **Page Configuration** → **pcf** → **line**. For personal package, navigate to **pcf** → **line** → **ppp**.
2. Within the **ppp** folder, create a new PCF file in the **policy** folder with the following properties:

Property	Value
File name	`PPPLineReview`
File type	`Screen`
Mode	(blank)

   Studio creates `PPPLineReviewScreen.pcf`.
3. On the **Required Variables** tab for the screen, add required variables. Examine `CPPLineReviewScreen.pcf` as an example.
4. On the **Variables** tab for the screen, add variables. Examine `CPPLineReviewScreen.pcf` as an example.
5. Copy and paste the `Toolbar` from `CPPLineReviewScreen.pcf`.
6. Copy and paste the `CardViewPanel` from `CPPLineReviewScreen.pcf`.
7. Add a `PanelRef` widget.
8. Under **Basic properties** on the **Properties** tab, with the `PanelRef` widget selected, set the `def` field to the line review summary detail view that you created in the previous procedure. For personal package, set the `def` field to `PPPLineReviewSummaryDV(line)`.
9. Copy and paste the `DetailViewPanel` from `CPPLineReviewScreen.pcf`. Under **Basic Properties** on the **Properties** tab, change the `id`. After copying, change the `id` property under **Basic properties** on the **Properties** tab. Remove the widgets labeled **Coverage Part Selection** and **Package Risk Type**.
10. Customize this screen to meet your requirements.

To add the line review screen to the job wizard steps

1. Open the line wizard step set that you created in "Adding the Line Wizard Step Set for Your Multi-line Product" on page 182. For personal package, open `LineWizardStepSet.PersonalPackage` in **configuration** → **config** → **Page Configuration** → **pcf** → **line** → **ppp** → **job**.
2. Add a **JobWizardStep** widget as the last step in the `WizardStepGroup` for each policy line. Using personal package as an example, drag a **JobWizardStep** widget and drip it as the last step in `PAWizardStepGroup`.
3. Using the `JobWizardStep` widgets in `LineWizardStepSet.CommercialPackage` as an example, set the properties for the new `JobWizardStep`.

Adding Quote Screens for a Multi-line Product

Create a **Quote** screen that displays a summary for each line of business, as explained in the following procedures.

To create panel sets and popup for rating details

Using commercial package as an example, create the following PCF files in **configuration** → **config** → **Page Configuration** → **pcf** → **line** → *`line`* → **policy**:

Property	Value
File name	*`abbr`*`RatingCumulPopup`
File type	`Popup`

Property	Value
Mode	(blank)
File name	*line*RatingCumulPanelSet.drilldown
File type	Panel Set
Mode	drilldown
File name	*line*RatingCumulPanelSet.scroll
File type	Panel Set
Mode	scroll

To create methods for the quote screen

These methods are used by the rating panel set to set the page length for each line of business.

1. In Studio, navigate to the `gw.lob.multiline` package in **Classes**. Right-click `multiline` and select **New** → **Gosu Class**. In the **New Gosu Class** dialog box, enter *line*`QuotePage` for the class name. For personal package, enter `PPPQuotePage`.
2. Write a *line*`QuotePageLength` method. Examine `CPPQuotePage.gs` in `Classes.gw.lob.multiline` as an example.

To create a rating panel set for your product

1. In Studio, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **line** → *line* → **policy.**
2. Create a new PCF file in the **policy** folder with the following properties:

Property	Value
File name	Rating
File type	Panel Set
Mode	Personal Package

3. Using `RatingPanelSet.CommercialPackage` as an example, set the **Properties**, **Code**, **Variables**, and **Required Variables** for the new rating panel set.

To create a quote screen for each job wizard

The job wizards display a multi-line quote screen for multi-line products. Create a multi-line quote screen for the following job types:

- Cancellation
- Issuance
- Policy change
- Reinstatement
- Renewal
- Rewrite
- Submission

1. For each type of job, in Studio, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **job** → *jobname*.
2. Right-click *jobname***Wizard_MultiLine_QuoteScreen.CommercialPackage** and select **Copy.**
3. Right click in the same node and select **Paste.**

4. In the **Copy** dialog box, replace `CommercialPackage` in the **New name** field with the name of your package. For personal package in the submission wizard, the new name is `SubmissionWizard_MultiLine_QuoteScreen.PersonalPackage`.
5. Make any necessary modifications to the PCF file.
6. Repeat steps 1–5 for the remaining job types.

Step 4: Create the Policy File Screens

Create the PCF files for the policy file in the **policyfile** folder. This folder contains PCF files for viewing the policy file. The main PCF file is the menu item set. For example, the main PCF file for commercial package is `PolicyMenuItemSet.CommercialPackage`. The menu item set is similar to the wizard step set, and often contains similar menu items.

Create the following types of PCF files in the **policyfile** folder:

- A menu item set for the product such as `PolicyMenuItemSet.CommercialPackage`. The menu item set is similar to wizard steps in a job wizard except that the menu items point to location group links. The menu item set includes one menu item for each policy line in the package.
- Location group links files for each policy line in the package. Each link points to a page in the **policyfile** folder.
- Pages for displaying menu items in the multi-line product. These pages reference the PCF files in the **policy** folder.

chapter 15

Adding Premium Audit to a Line of Business

PolicyCenter provides premium audit for several lines of business in the base configuration. You can add the premium audit job to other lines of business. The premium audit job includes final audit and premium reports. In the base configuration, final audit is provided in workers' compensation and general liability. Workers' compensation includes configuration of premium reports. This topic describes how to add final audit to a line of business using general liability as an example.

See also

- "Premium Audit Policy Transaction" on page 143 in the *Application Guide*

Step 1: Add Audited Basis to the Data Model

The data model must be updated with fields to hold the audited basis. First, determine which entities are to be audited. Then create a field on the entity to store the audited basis amount. You enter the audited basis in PolicyCenter in the audit **Details** screen when completing an audit. For more information about this screen, see "Adding the Audit Details Panel Set" on page 190.

For example, the general liability line of business supports final audit on exposures. The `GLExposure` entity has an `AuditedBasis` field. You can examine the definition of this field in Studio by opening `GLExposure.eti`. To open this file, press `CTRL+N` and type in the file name. The `AuditedBasis` field is defined as a column on the `GLExposure` entity.

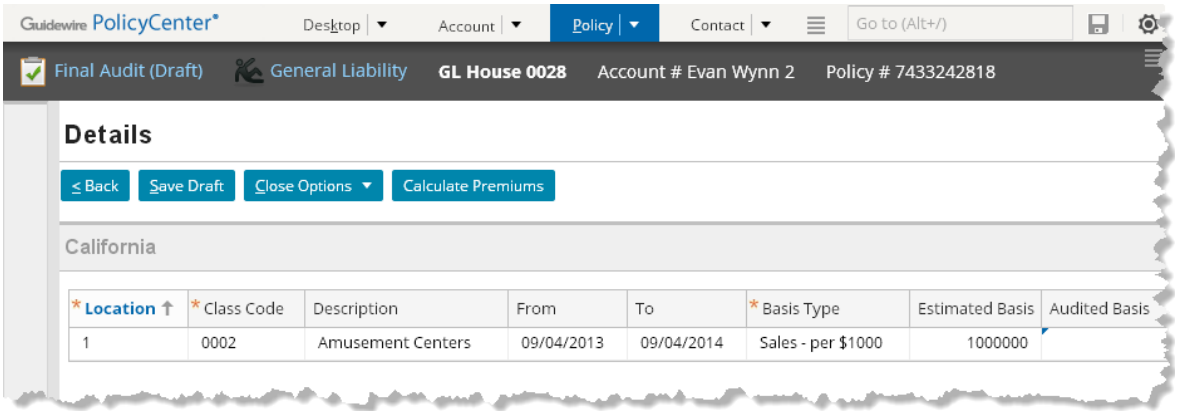
Add a similar field to any entities to be audited in your line of business.

Step 2: Add the Line of Business to the Audit Wizard

Update the audit wizard to support final audit in your line of business. This topic describes how to design panels for the audit **Details** and **Premium Details** panel sets in the audit wizard.

Adding the Audit Details Panel Set

When you start an audit, the audit **Details** screen enables you to enter audited amounts. Design and add an audit **Details** panel for your line of business. The **Details** panel for general liability lists exposures by jurisdiction. The **Audited Basis** text box enables you to enter the audited basis for each exposure. Each exposure is listed by class code. The following illustration shows the **Details** panel for general liability.



To add the Audit Details panel set

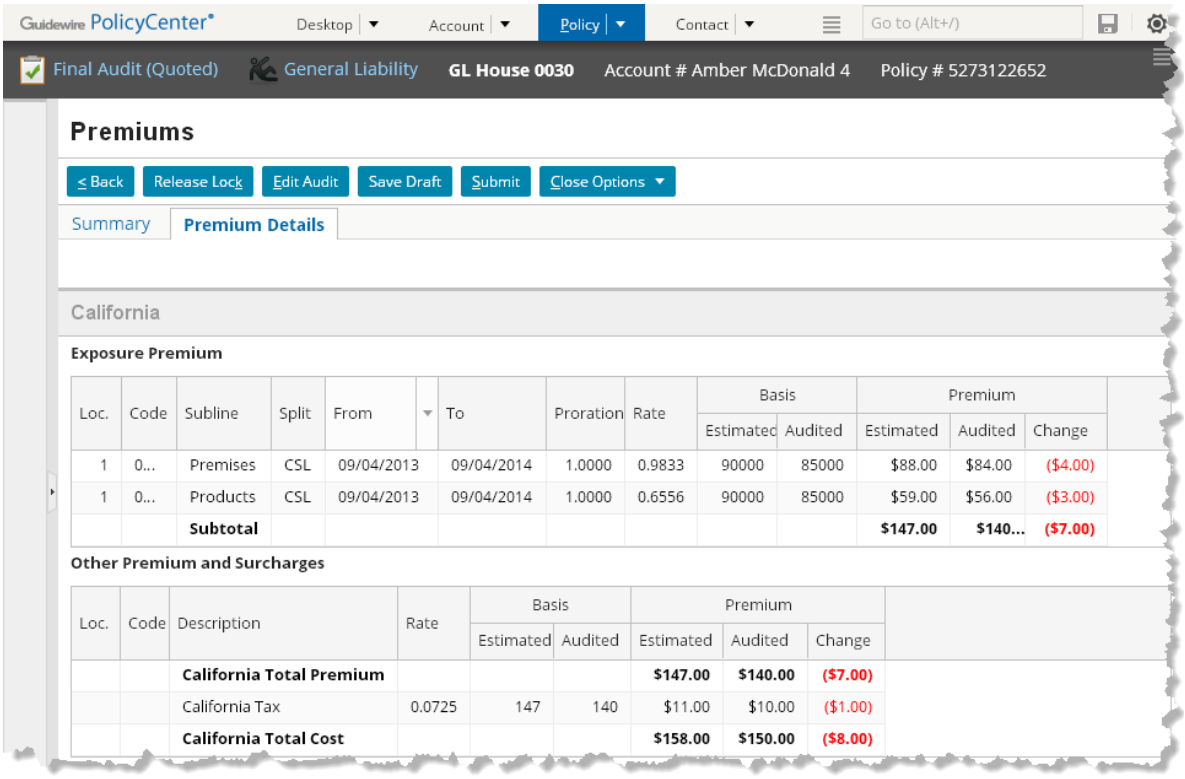
1. In the Studio **Project** window, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **job** → **audit.**
2. Right-click the **audit** folder, and then select **New** → **PCF File**. In the **PCF File** dialog box, specify the following properties:

Property	Value
File name	`AuditDetails`
File type	`Panel Set`
Mode	Line of business name. For example, `Homeowners`

   Studio creates and opens, for example, `AuditDetailsPanelSet.Homeowners.pcf` and automatically adds the panel set to the **Shared section mode** `PanelRef` in the `AuditWizard_DetailsScreen.pcf`.
3. Add one or more **Panel Iterator** widgets and other widgets as needed to display the auditable items.
4. Add one or more widgets to enable users to enter the audited amounts. Define the `value` of the widget as the additional basis field. You added the additional basis field to the data model in “Step 1: Add Audited Basis to the Data Model” on page 189.

Adding the Premium Details Panel Set

When you click **Calculate Premiums** on the audit **Details** screen, the **Premiums** screen appears. The **Premium Details** tab shows the details for the audited amounts in a format that is specific to the line of business. This topic describes how to add the premium details panel set for your line of business. The following illustration shows the **Premium Details** tab for general liability.



To add the Premium Details panel set

1. In the Studio **Project** window, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **job** → **audit.**
2. Create a new PCF file in the **audit** folder with the following properties:

Property	Value
File name	`AuditPremiumDetails`
File type	`Panel Set`
Mode	Line of business name. For example, `Homeowners`

   Studio creates and opens, for example, `AuditPremiumDetailsPanelSet.Homeowners.pcf` and automatically adds the panel set to the **Shared section mode** `PanelRef` in the `AuditWizard_PremiumsScreen.pcf`.
3. Add one or more **Panel Iterator** widgets and other widgets as needed to display the premium details.

Step 3: Add Gosu Code for Final Audit

You must add Gosu code that enables audit for your line of business. Add code to ensure that all audit amount fields on the audit **Details** screen have values. Finally, update the rating engine to calculate premiums based on the audited amounts.

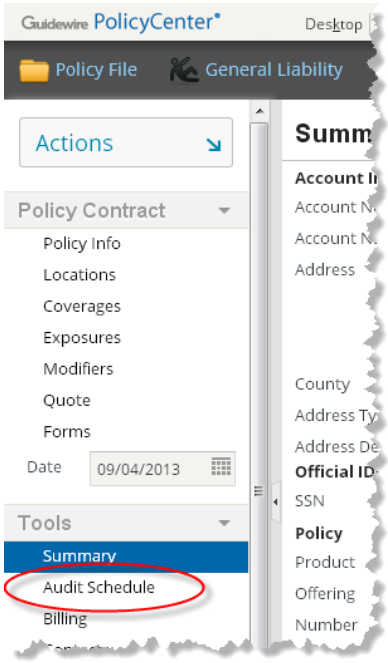
Enabling Audit for a Line of Business

In Studio, add code to enable audit for your line of business.

1. In Studio, navigate to **configuration** → **gsrc** and open `gw.lob.`*`line`*`.`*`line`*`PolicyLineMethods.gs`.
2. Add the following code to set the `Auditable` property to `true`:

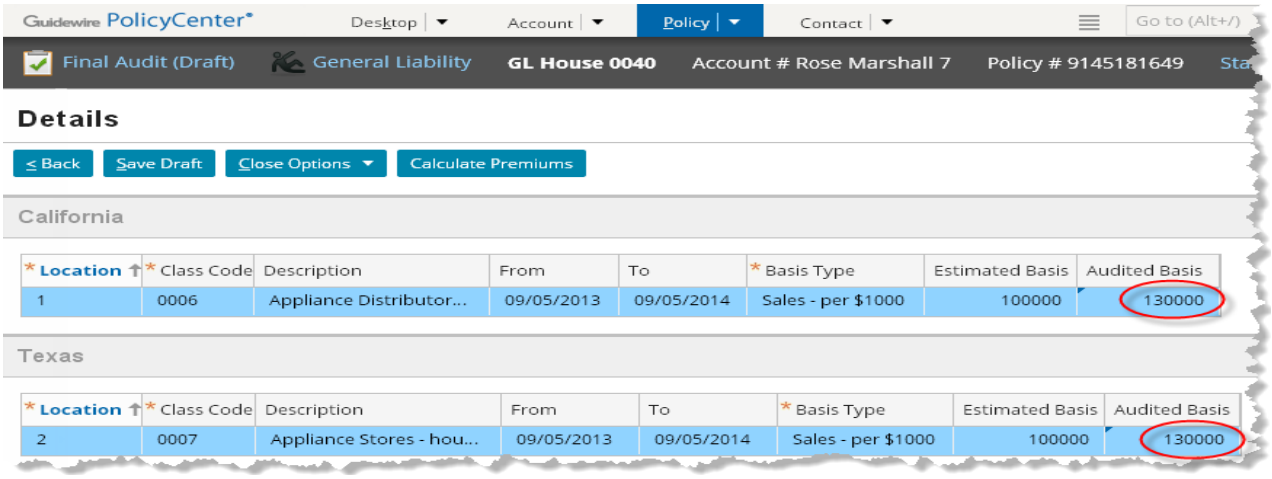
```
override property get Auditable() : boolean {
    return true
}
```

After you add this code in Studio and deploy your changes, PolicyCenter displays the **Audit Schedule** link under **Tools** in the left sidebar of a Policy File.



Validating the Line Before Calculating Premiums

Add code to ensure that all audit amount fields have values on the audit **Details** screen. For example, in the general liability line, the code checks that the auditor has entered a value in each **Audited Payroll** field. The value of the **Audited Payroll** field is the audited basis field defined in "Step 1: Add Audited Basis to the Data Model" on page 189.



1. In Studio, navigate to **configuration** → **gsrc**, and open `gw.lob.`*`line`*`.`*`line`*`LineValidation`. For homeowners, open `gw.log.ho.HOLineValidation`.
2. Add the `validateLineForAudit` method. This methods determines whether all audit amount fields have values prior to calculating premiums.

```
override function validateLineForAudit() {
  allAuditAmountsShouldBeFilledIn()
}
```

3. Define the `allAuditAmountsShouldBeFilledIn` method. For example, in the general liability line, this method checks that all `glExposure.AuditedBasis` fields have a value:

```
function allAuditAmountsShouldBeFilledIn() {
    if (glLine.Branch.Job typeis Audit) {
        glLine.VersionList.Exposures.flatMap(\ g -> g.AllVersions).each(\ glExposure -> {
            if (glExposure.AuditedBasis == null) {
                Result.addError(glLine,
                          "quotable",
                          displaykey.Web.AuditWizard.Details.NullAmountsError,
                          displaykey.Web.AuditWizardMenu.Details)
            }}
        )
    }
}
```

Updating the Rating Engine

Update the rating engine to calculate the premiums for final audit on your line of business. In an audit job, use the `AuditedBasis` field to calculate the premium in place of the `BasisAmount` field, which stores the estimated basis. You can use a general purpose property that returns either the `AuditedBasis` in an audit job or the `BasisAmount` in other jobs. This general purpose property keeps the rating code the same for audit and other jobs. For an example, examine the `BasisForRating` property in `gw.lob.gl.GLExposureEnhancement`.

PolicyCenter displays the premiums on the **Premiums** screen of the audit wizard. This screen appears after the auditor enters the audited amounts on the **Details** screen, and then clicks **Calculate Premiums**.

Step 4: Select the Audit Schedule for Final Audit

The audit schedule pattern selector plugin (`gw.plugin.job.IAuditSchedulePatternSelectorPlugin`) determines the default schedule for final audit. The audit schedule specifies the start date, due date, and audit method (physical, voluntary, or by phone). The default final audit schedules are:

- **Cancellation Audit by Phone** for all cancellation final audits
- **Expiration Audit by Physical** for all non-cancellation final audits

You can modify this plugin or define your own class to select audit schedules for final audit. For example, you might want to require physical audits for policies with total premiums that are greater than a certain threshold.

Step 5: Enable Premium Reports

After you configure a line of business for final audit, you can enable premium reporting. This topic describes how to enable premium reports for a line of business.

Filtering Reporting Plans

1. In Studio, navigate to **configuration** → **gsrc**, and then open `gw.plugin.policy.impl.PolicyPaymentPlugin`.

   The `filterReportingPlans` method returns an array of reporting plans based on the product code of the policy.

2. Modify the method to return reporting plans for the line of business. For example, the following code adds premium reporting for the general liability line.

```
override function filterReportingPlans( policy : Policy, plans : PaymentPlanSummary[] ) :
      PaymentPlanSummary[]
{
  if (policy.ProductCode.equals("WorkersComp") || policy.ProductCode.equals("GeneralLiability")) {
      return plans
  }
  else {
    return plans.where( \ p -> p.PaymentCode == null )
  }
}
```

Modifying the Audit Wizard to Support Premium Reports

You defined the audit details panel set in “Adding the Audit Details Panel Set” on page 190. To support premium reports, you must update this panel set as follows:

1. Navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **job** → **audit** and open `AuditDetailsPanelSet.line`.

2. In the `PanelSet`, configure the row iterator to display the auditable items that have an effective date that falls within the audit period. For example, the auditable items in general liability are exposures.

   Examine the `wcCovEmp` row iterator in `AuditDetailsPanelSet.WorkersComp`. The value of the iterator is set to `wcCoveredEmpInLocation(ratingPeriod)`. Ctrl-click `wcCoveredEmpInLocation` to view this method in the **Code** tab.

3. For premium reports, display a prorated amount for the value of the **Estimated Payroll** cell in the row iterator.

   For an example in workers’ compensation, examine the `EstPayroll` cell. The value is set to `basisForRating(wcCovEmp)`. Ctrl-click `basisForRating` to view this method in the **Code** tab. The prorated amount is calculated according to this formula:

```
BasisAmount * Days of Overlap with Audit Period / Total Days in Effective Period for the row
```

Modifying the Rating Code to Support Premium Reports

The rating code must rate only the audited amounts that overlap the audit period. Furthermore, rating a full policy (such as a submission or final audit) is different than rating a premium report.

You must determine how rating differs for a premium report, and then modify the rating code appropriately. For example, in premium reports for workers' compensation, the rating code does not calculate an Expense Constant. Also, for Premium Discount, the rating code uses a previously-determined discount factor rather than calculating a new one.

Step 6: Add Premium Audit to a Multi-line Product

When you enable premium audit in a multi-line product, the multi-line policy is auditable if one of the lines in the policy is auditable. For example, commercial package policy includes commercial property, general liability, and inland marine lines. During a submission, you can add any combination of these lines to a policy. In the base configuration, commercial package policy is not auditable even though it contains the auditable general liability line.

This topic uses commercial package policy as an example of how to enable audit in a multi-line product.

Enabling Audit in a Multi-Line Product

In `gw.policy.policyPolicyPeriodAuditEnhancement`, the `IsAuditable` property returns `false` if the product of the policy is multi-line. To enable auditing, change `IsAuditable` to return `true` by commenting out the `if` statement:

```
property get IsAuditable() : boolean {
  //CPP policy
  /*
      if (this.MultiLine) {
          return false
      }
  */
  return this.Lines.hasMatch(\ p -> p.Auditable)
}
```

After you make this change, the **Payment** screen in PolicyCenter displays the **Audits** section if the policy includes a line that is auditable.

Modifying the Audit Wizard to Support a Multi-Line Product

In the base configuration, the audit **Details** and **Premiums** screens do not handle multi-line products. You must configure these screens to find the auditable lines and display the auditable items for each line.

chapter 16

Configuring Copy Data in a Line of Business

This topic describes how to configure copy data in a new line of business. In the base configuration, copy data functionality is available only in the personal auto line of business.

This topic includes:

See also

- "Overview of Copying Data Between Policies" on page 305 in the *Application Guide*

Overview of Configuring Copy Data

By using copy data functionality, you can quickly and accurately copy information from one policy to another. You can also copy information from an existing transaction to the current transaction. Copy data functionality provides:

- A mechanism that searches for policies and transactions and selects the source period.
- A user interface that controls the items you can copy from the selected source period.
- `Copier` Gosu classes that copy information from the source period to the target period.

  Note: The ability to split and spin policies from an existing policy requires that copy data be configured for that line of business. *Splitting* a policy creates two new accounts and splits the coverables on the source policy into coverables on policies in the new accounts, creating two new submission transactions. *Spinning* a policy creates one new account and moves the coverables on the source policy into coverables on the target policy in the new account, creating one new submission transaction.

Copy Data Jobs

Copy data is available only on transactions that manipulate policy data. These transactions are:

- Submission
- Policy change
- Renewal
- Rewrite
- Rewrite new account

Searching for the Source Policy

In the base configuration, when you search for a policy from which to copy, you can search only within the product specified in the policy period for the target policy. For example, to copy data to a personal auto transaction, you can search only within other personal auto policies.

You can configure the search to find policies from different products. However, the different products must include the same line or lines of business. In the base configuration for example, the commercial package product includes the commercial property, general liability, and inland marine lines. The base configuration also has monoline products for commercial property, general liability, and inland marine. You can configure copy data to search within and copy from the monoline products provided those lines are included in the package product.

Copying Data to a Multi-line Product

In a multi-line product, copy data can only copy from the policy line of the source period into the same policy line on the target period.

When copying data to a multi-line product, it is possible for the copy to create a policy line. Using commercial package as an example, assume the target period has a commercial property line but not a general liability line. If you select a source period and a general liability exposure, copy data creates a general liability line and adds the exposure to the target.

Copy Data and Data Integrity

The information that copy data copies to the target period can be incomplete or inconsistent with the product model. The copier classes have no mechanism to verify that the copied information is available in the target period. For example, you can copy a coverage that is no longer available into a new period. If you do, quote level validation and product model consistency checks ensure the integrity of data. When information is initially copied, the policy is in draft mode. Quote and bind validation ensure the integrity of the data.

Copy Data Gosu Classes

When you implement copy data, you must write a Gosu class for each entity that can be copied. These classes extend an abstract `Copier` Gosu class that copies data from a source to a target. This target is a container for the copied object. For example, for a line-specific coverage, this target typically is the policy line. A typical implementation could have a `PersonalVehicleCopier` class for copying vehicles and a `DriverCopier` class for copying drivers.

In the base configuration, the personal auto line implements copy data functionality. Examine the copier classes and PCF files in personal auto when you add copy data to another line of business or add additional copiers to the personal auto line.

The design of the copy data functionality provides the following features:

- You define a `Copier` subclass for each copy data type.

- Your code determines which child and containing entities to copy.
- You can use methods and properties from existing interfaces such as Matchers.
- Copier classes can make use of existing methods for creating data. For example, to copy a building, call the existing method for creating a building in the line of business. Then call the copier code to populate the properties of the building. By using existing methods, you can reuse existing code such as building auto-numbering.
- The `Copier` classes provide interfaces for presenting copy data in PolicyCenter through PCF file configuration.

Configuring Copy Data Screens

In PolicyCenter, the copy data feature has two screens:

- **Copy Policy Search Policies Screen** – Enables you to search for policies from which to copy data.
- **Select Data to Copy From Policy Screen** – After you select a source policy, enables you to select the data to copy.

Copy Policy Search Policies Screen

The **Copy Policy Search Policies** screen is similar to the **Search Policies** screen. It appears when you are in a transaction, such as a submission, for a line that supports copy data, and then select **Actions** → **Copy Data**. The main PCF file for this screen is the `CopyPolicyDataSearchPopup` in Studio in **configuration** → **config** → **Page Configuration** → **pcf** → **job** → **common** → **copydata**.

The `PolicySearchCriteria` entity and `gw.search.PolicySearchCriteriaEnhancement` Gosu file implement much of the search functionality. This entity and Gosu file are also used by the **Search Policies** screen. Because both search screens share the entity and Gosu code, a change to the entity or Gosu code affects both types of searches. Likewise, a change to the PCF file for one search type may require a corresponding change in the other.

On the **Copy Policy Search Policies** screen, you can search for policies with the same product as the target policy period. For example, if you select **Copy Data** from a personal auto transaction, the search returns only personal auto policies. If needed, you can configure the search to find products different from the target policy. For more information, see "Searching for the Source Policy" on page 198.

By default, the **Account Number** field is set to the account number of the target policy.

The `ExcludedPolicyPeriod` field on the `PolicySearchCriteria` entity contains the current policy period. Therefore, the current policy period does not appear in the search results.

Select Data to Copy From Policy Screen

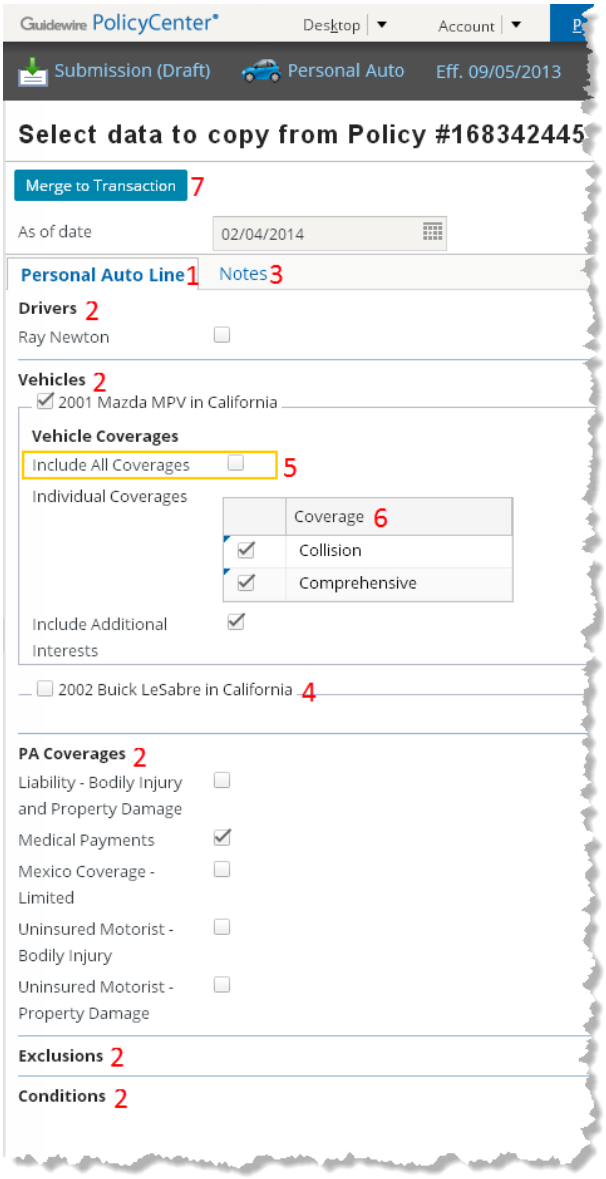
After you select a source policy from which to copy data, PolicyCenter displays the **Select data to copy from Policy** screen. The PCF file for this screen is `CopyPolicyDataDetailPopup`.

This screen has a `copier` variable that creates a `PolicyPeriodCopier` at the top level of the source period. The `PolicyPeriodCopier` class also creates line level copiers in its `initLines` method. For personal auto, the line level copier is `gw.lob.pa.PAPolicyLineCopier`.

The `PolicyPeriodCopier` class provides a method for the `AllNotesCopiers` for copying notes.

Besides notes, the other information relevant to a policy is almost always associated with a `PolicyLine`. The `PolicyPeriodCopier` initializes a copier for each policy line contained in the product. Each line initializes the copiers that it provides.

The copier for each policy line creates copiers for the top-level entity or types that can be copied.



1. The **Card Iterator** tab has a **Shared section mode** drop-down list for each policy line. Create a modal `DetailViewPanel` for each line of business in the product. The **Personal Auto Line** card corresponds to the `PAPolicyLineCopier`.

2. For personal auto, the `CopyPolicyDV.PersonalAutoLine` PCF file has sections to display copy data for **Drivers**, **Vehicles**, **PA Coverages**, **Exclusions**, and **Conditions**. This PCF file defines variables that create copiers for policy drivers, vehicles, personal auto coverages, exclusions, and conditions.

   The `Copier` class provides `ShouldCopy` and `ShouldCopyAll` Boolean properties. On the PCF file, check boxes and radio buttons set the values of these properties for coverages and other data that can be copied. For example, if you select the **Include All Coverages** check box, PolicyCenter sets the `ShouldCopyAll` property to `true`.

3. On the **Notes** tab, the `CopyNotesDV` PCF file has a variable that creates a notes copier.

4. The PCF file can include an input iterator that displays high level coverables such as vehicles or buildings on the source policy. The iterator enables you to select one or more of these coverables for copying. When you select a coverable, child elements such as the coverages and additional interests appear and can be selected for copying.

   For personal auto, the input iterator displays all vehicles on the source policy. In the illustration, the **2001 Mazda MPV** is selected and displays the child elements:

   - **Include All Coverages**
   - **Individual Coverages**
   - **Additional Interests**

   These child elements do not appear for the **2002 Buick LeSabre** because it is not selected.

5. The **Include All Coverages** option ties to a composite copier. The value of this check box is set to the `ShouldCopyAll` Boolean property.

6. The PCF file can include a list view with rows that represent the available copiers. Each row has a check box for selecting the entity or group of entities for copying.

7. When you click **Merge to Transaction**, PolicyCenter calls the `copyInto` method in all selected copiers.

   PolicyCenter displays any warning messages in **Validation Results** at the bottom of the screen. For example, PolicyCenter displays a warning if you try to copy John Smith, who already exists as a driver on the target policy.

   - If you ignore the warning, make no further changes, and click **Merge to Transaction** again, PolicyCenter overwrites the data.
   - If ignore the warning, make additional changes that do not fix the condition, and then click **Merge to Transaction**, PolicyCenter displays warning messages again.

Understanding Copiers

The `gw.api.copy.Copier` abstract class provides base functionality for copying data from a source to a target. In the same package, the `CompositeCopier` abstract class extends `Copier`. The `CompositeCopier` wraps a collection of copiers of one or more types, creating a tree of copiers that reflects the structure of the data to be copied.

You can create concrete implementations of these abstract copier classes. The concrete implementation of the `Copier` class is responsible for copying the data on an entity. The concrete implementation of the `CompositeCopier` class is responsible for copying data on an entity and its child entities.

In your concrete subclass of the `Copier` class, define the `copyInto` method to perform the following actions:

1. **Check for an existing matching entity** – Check for a matching entity in the destination period, such as a vehicle with the same VIN. If a matching entity already exists in the destination period, you can configure your code to throw an exception or you can copy source fields over the existing entity.

2. **Create a new entity** – Create a new entity if a matching entity does not exist. To create a new entity, examine existing domain methods, such as `PersonalAutoLine.createAndAddVehicle` to create personal vehicles. If possible, reuse these existing methods. Doing so ensures that the code performs additional logic, such as auto-numbering objects.

3. **Create additional entities** – When you create a new entity, also create any additional entities needed for the copy data operation. For example, if you copy a `PolicyDriver` who is not a contact on the target account, your code must create both an account `Driver` and an `AccountContact`.

4. **Copy entity fields** – Copy all relevant fields and child elements. The simplest implementation uses a sequence of assignment statements. To automate copying, use utilities such as the `copy` and `KeyableBean.shallowCopy` methods of `GWKeyableBeanEnhancement`, or use code from side-by-side quoting that copies entities. If you use one of these utilities, the `copyInto` method must remove the fields that are not part of the copied data.

5. **Copy child entities** – Copy child entities where the `ShouldCopy` and `ShouldCopyAll` Boolean properties are `true`. You can provide this functionality in your `Copier` subclass or in a helper class. Control over whether these children are copied can be delegated to child copier classes by overriding the `CollectCopiersWhere` method in `gw.api.copy.CompositeCopier`. For example, you can configure the `PersonalVehicle` copier to automatically copy all of its coverages or just selected coverages.

Copiers in the Base Configuration

The base configuration includes copiers for the personal auto policy line and for notes.

In the `gw.lob.pa` package, personal auto contains copiers for the following:

- Personal auto line
  - Personal vehicles
    – Vehicle coverages
    – Vehicle modifiers
    – Vehicle additional interests
  - Policy drivers
- Personal auto line-level coverages
- Personal auto line-level exclusions
- Personal auto line-level conditions

For personal auto, you can define additional copiers for other key data elements or for any extensions you have made to personal auto.

How to Create Copiers for a Policy Line

This topic provide step-by-step instructions on how to create copiers for a policy line. The steps create a copier for the commercial property line of business. The commercial property line needs copiers for locations and buildings.

Examine PCF files and Gosu classes for personal auto line as an example.

To enable copy data for a product

1. In Studio, navigate to the Gosu class `gw.job.CopyDataVisibilityByProduct`.
2. Add your product code to the `ENABLED_PRODUCTS` variable.

To create a copier for a policy line

1. Create a line-specific copier that extends the `gw.api.copy.CompositeCopier` class. In Studio, create a `gw.lob.cp.CPPolicyLineCopier` class.

   The signature for `CompositeCopier` is:

   ```
   CompositeCopier<T extends KeyableBean, S extends KeyableBean>
   ```

   `T` is the target and `S` is the source.

2. In your class, define the signature where the target is `PolicyPeriod`, and the source is the policy line.

   ```
   uses gw.api.copy.CompositeCopier

   class CPPolicyLineCopier extends CompositeCopier<PolicyPeriod, CommercialPropertyLine> {

     construct() {

     }

   }
   ```

3. In `gw.lob.cp.CPPolicyLineMethods`, override the `get Copier` property to return your new line copier class.

```
override property get Copier() : CompositeCopier<PolicyPeriod, CommercialPropertyLine> {
  return new CPPolicyLineCopier(_line)
}
```

4. In **configuration** → **config** → **Page Configuration** → **pcf** → **job** → **common** → **copydata**, create a `CopyPolicyDV.CPLine` PCF file. Using `CopyPolicyDV.PersonalAutoLine` as an example, design the user interface for copying data.

5. Create copiers and composite copiers for the entities and other data that you want to copy.

   Decide which coverables to copy, such as locations and buildings for commercial property. Create copiers for these coverables.

   For example, the personal auto line has the following copiers in the `gw.lob.pa` package:

   - `AddlInterestDetailsCopier extends Copier`
   - `AllAddlInterestDetailsCopier extends GroupingCompositeCopier`
   - `ModifierCopier extends Copier`
   - `PAPolicyLineCopier extends CompositeCopier`
   - `PersonalVehicleCopier extends CompositeCopier`
   - `PolicyDriverCopier extends Copier`

6. **Optional** – In the line `Copier` class, add methods to retrieve the child copiers of the line. In `gw.lob.pa.PAPolicyLineCopier`, personal auto has an example that gets the personal vehicle copier:

```
property get PersonalVehicleCopiers() {...}
```

Copier API Classes

This topic describes the Copier API abstract classes. The description of each abstract class includes a description of the concrete classes that extend that abstract class.

Copier API

The Copier abstract class provides the following functionality:

- Access to the source entity
- Control of whether to copy the source entity
- Optional matching method
- Base method copy that copies the source into the target.

The type of the target parameter varies by copier. The following table shows the types of the target parameter for certain copiers.

Copier	Target parameter type
`NoteCopier`	`PolicyPeriod`
`ModifierCopier`	`Modifiable`
`PersonalVehicleCopier`	`PersonalAutoLine`
`PolicyDriverCopier`	`PersonalAutoLine`

Note Copier: A Simple Copier

The note copier is an example of a simple copier that copies a `Note` entity. The note copier is a simple example that you can use for developing your own copiers.

The note copier:

- Copies a non-revisioned entity.
- Does not copy any child entities.
- Copies directly into `PolicyPeriod`
- Does not use matching. If a note is copied twice, the note appears twice in the target.
- Copies a single note from one policy period to another.

See also

- "All Note Copier: A Simple Grouping Composite Copier" on page 205

Policy Driver: A Copier for a Matching Entity

The copier for policy driver is a more complex example that checks for matching drivers.

The `PolicyDriverCopier` overrides the `findMatch` method on the abstract `gw.api.copy.Copier` class. The `findMatch` method returns a matching `PolicyDriver`, if any, preventing the copier from creating duplicate drivers.

Composite Copier API

The `CompositeCopier` is an abstract class that copies an entity together with its child entities. The `CompositeCopier` class extends `Copier`. For example, the `PersonalVehicle` composite copier copies the properties directly associated with the `PersonalVehicle` entity and also copies coverages and additional interests on the vehicle. The `PersonalVehicleCopier` delegates the work of copying child entities to other copiers.

The abstract `CompositeCopier` class provides the following methods for constructing the copier tree:

- `addCopier` – Adds a specific copier to the composite tree of copiers.
- `addAllCopiers` – Adds multiple copiers to the tree of copiers.

Personal Vehicle Copier: A Composite Copier

The `PersonalVehicleCopier` is a subclass of `CompositeCopier`. This class delegates the copying of coverages, additional interests, and modifiers to copiers for those entities. The `PersonalVehicleCopier` constructor initializes the other copiers. The `CompositeCopier` abstract class automatically calls each of the other copiers, which some or all of which can be marked as `ShouldCopy`.

The personal vehicle copier copies the following:

- **Modifiers** – `ModifierCopier` is always included. Therefore, modifiers do not appear as a selection on the **Select data to copy from Policy** screen.
- **All coverages** – If you select **Include All Coverages**, `vehicleCopier.AllCoverageCopier.ShouldCopyAll` is set to `true` in the PCF file, and all coverages are copied.
- **Individual coverages** – If you select coverages under **Individual Coverages**, each selected coverage has `ShouldCopy` set to `true` in the PCF file. PolicyCenter iterates through the individual copiers in the `AllCoveragesCopier` by executing the following code:

  ```
  SelectIndividualCoveragesLV(vehicleCopier.AllCoverageCopier.AllExistingCoverageCopier.Copiers)
  ```

- **Additional interests** – If you select **Include Additional Interests**, `vehicleCopier.AllAddlInterestDetailsCopier.ShouldCopyAll` is set to `true` in the PCF file. All additional interests are copied.

Grouping Composite Copier API

The `GroupingCompositeCopier` extends `CompositeCopier`. The `GroupingCompositeCopier` is a composite copier for a group of similar copiers. This copier provides determines whether to copy individual child entities. As with other copiers, use the `ShouldCopyAll` method to handle a **Copy All** request.

All Note Copier: A Simple Grouping Composite Copier

The `AllNoteCopier` extends `GroupingCompositeCopier`. Use the `AllNoteCopier` to enable a single method call to copy all notes.

All Coverage Copier: A Grouping Composite Copier

The `AllCoverageCopier` extends `GroupingCompositeCopier` and provides the following functionality:

- Adds new coverages on the target coverable.
- Overwrites existing coverages on the target coverable.
- Removes any coverages from the target coverable that does not exist on the source coverable.

The `AllCoveragesCopier` delegates to two other copiers:

- `AllExistingCoverageCopier` – Adds coverages.
- `AllRemovingCoverageCopier` – Removes coverages.

Using copy data in personal auto as an example, assume you have a matching vehicle on both the source policy and the target policy. On the source policy, only collision coverage is selected. On the destination policy, only comprehensive coverage is selected. When copying the data from the source to the target, the copier adds the collision coverage and removes the comprehensive coverage.

Generic Copier Templates

The base configuration provides generic copier classes that you can extend to simplify a copy data implementation. Many of the copiers in the base configuration extend these copier classes.

The generic copier classes are:

Copier	Description
`AddressCopier`	`extends Copier`
`AllConditionCopier`	`extends GroupingCompositeCopier`
`AllCoverageCopier`	`extends GroupingCompositeCopier`
`AllExclusionCopier`	`extends GroupingCompositeCopier`
`AllExistingCoverageCopier`	`extends GroupingCompositeCopier`
`AllExposureCopier`	`extends GroupingCompositeCopier`
`AllRemovingCoverageCopier`	`extends GroupingCompositeCopier`
`ClausePatternCopier`	`extends Copier`, and can be used on most policy lines for copying line level coverages
`RemovingClausePatternCopier`	`extends Copier`

chapter 17

Adding Locations to a Line of Business

If you are configuring or adding a new line of business with locations, the policy line methods file contains methods for handling locations. This topic describes some of those methods. For an example, see the `gw.lob.cp.CPPolicyLineMethods` class which contains the policy line methods for the commercial property line of business.

This topic includes:

See also

- "Location Plugin" on page 157 in the *Integration Guide*

Methods to Remove a Location from a Policy Line

The policy line methods include methods for removing a location from a policy line. These methods must account for earlier policy periods and future policy periods caused by out-of-sequence changes.

The policy line methods are defined in the gw.api.policy.AbstractPolicyLineMethodsImpl class. The methods related to removing a location include the following:

Method	Description
canSafelyDeleteLocation	Determines whether the location can be safely deleted. Use this method with line-specific location screens. For example, the location screen can call this method to determine whether to enable the **Remove** button. In the base configuration, this method applies to line-specific locations but not the corresponding PolicyLocation in a multi-line package product. The **Remove** button on the **Locations** screen deletes a line-specific location, such as BALocation or BOPLocation.
checkLocationInUse	Determines which locations can be removed during quote. Quoting removes any PolicyLocation objects that this method reports as not in use. In the base configuration, PolicyCenter removes PolicyLocation objects but not line-specific locations. The implementation assumes that the line-specific locations have been removed through the canSafelyDeleteLocation method on the **Locations** screen user interface. The checkLocationInUse method removes obsolete locations created: • In products without a **Locations** screen. • In package products with separate screens for policy-wide PolicyLocation objects and line-specific locations.
preLocationDelete	PolicyCenter calls this method before a location is deleted. Use this method to cleanup the policy location or update the data model.
canSafelyDeleteBuilding	Determines whether a building can be safely deleted.

Use the checkLocationInUse and canSafelyDeleteLocation methods to determine whether a location can be deleted from a policy line. In the PolicyLineMethodsDefaultImpl class, these methods check to make sure that the primary location is not deleted. Each line of business overrides these methods in its *line*PolicyLineMethods.gs implementation.

While the checkLocationInUse and canSafelyDeleteLocation methods are similar, there is an important difference in how they are intended to be used. In some lines of business, the canSafelyDeleteLocation and checkLocationInUse methods are the same. For example, the methods can check whether there are coverages attached or coverables located at the location. However, in some lines of business the methods must perform different operations. Therefore, two methods are provided.