Guidewire BillingCenter®

BillingCenter Best Practices Guide

RELEASE 8.0.4

Guidewire®
Deliver insurance your way™

Product Name: Guidewire BillingCenter
Product Release: 8.0.4
Document Name: BillingCenter Best Practices Guide
Document Revision: 02-July-2015

Contents

About BillingCenter Documentation

The following table lists the documents in BillingCenter 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 BillingCenter 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 BillingCenter.
Upgrade Guide	Describes how to upgrade BillingCenter from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing BillingCenter application extensions and integrations.
New and Changed Guide	Describes new features and changes from prior BillingCenter 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 BillingCenter. The intended readers are everyone who installs the application for development or for production.
System Administration Guide	Describes how to manage a BillingCenter 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 BillingCenter 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 BillingCenter.
Rules Guide	Describes business rule methodology and the rule sets in BillingCenter 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 BillingCenter 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 BillingCenter 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.

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, visit the Guidewire Resource Portal – `http://guidewire.custhelp.com`

chapter 1

Data Model Best Practices

The BillingCenter data model comprises metadata definitions of data entities that persist in the BillingCenter application database. Metadata definition files let you define the tables, columns, and indexes in the relational database that supports your application. Typelist definitions let you define sets of allowed codes for specific typekey fields on entities.

This topic includes:

Entity Best Practices

You can change the base data model of BillingCenter to accommodate your business needs with *data model extensions*. Extensions let you add fields to existing data entities and add entirely new data entities to the data model of your BillingCenter application. BillingCenter uses the data model and your extensions to create and manage the tables, columns, and indexes in the relational database that supports your application.

As a best practice, Guidewire recommends that you edit the metadata definition files of the data model by using the Data Model Extensions editor in Guidewire Studio.

See also

- "The BillingCenter Data Model" on page 147 in the *Configuration Guide*

Observe General Entity Naming Conventions

Generally, entity and field names begin with a capital letter. Medial capitals separate words in compound entity names. For example, `ContactAddress`.

Add a Prefix or Suffix to Entity Extensions

To avoid future naming conflicts when Guidewire adds or changes base entities, Guidewire recommends that you add `Ext` to your entity names and new field names on base entities.

- As a prefix – Add `Ext_` to beginnings of entity names so Studio and the *Data Dictionary* list them together in one group. For example, `Ext_ServiceArea`.
- As a suffix – Add `_Ext` to endings of entity names so Studio and the *Data Dictionary* list them next to any entities that they extend. For example, `CreditHistory_Ext`.

If you add a new entity, its field names do not need `Ext` in their names, because you have `Ext` in the name of the entity.

As an example of field extensions to a base entity, the following sample metadata file extends the base `Account` entity with an additional field (column) and an additional typekey.

```
<extension entityName="Account">
...
  <column desc="Description of the column"
    name="MyCustomColumn_Ext"
    nullok="true"
    default="abc"
    type="varchar">
    <columnParam
      name="size"
      value="60" />
  </column>

  <typekey desc="Description of the typekey"
        name="MyCustomTypekey_Ext"
        typelist="myCustomTypeList_Ext)"
        nullok="true" />
...
</extension>
```

Use Singular for Field Names Except for Arrays

Guidewire recommends that you name most fields with a singular word such as `Phone` or `Age`. However, because array fields reference a list of objects, Guidewire recommends that you name them with a plural word.

For example, `TroubleTicket.Escalated` and `TroubleTicket.EscalationDate` are single fields on a `TroubleTicket` entity, but `TroubleTicket.Activities` is an array of multiple activities. Also, for arrays fields that are extensions, make the primary name plural and not the `Ext` prefix or suffix. For example, use `Ext_MedTreatments` or `MedTreatments_Ext`, and not `MedTreatment_Exts`.

Add ID as a Suffix to Column Names for Foreign Keys

Guidewire recommends that you add `ID` as a suffix to the column names of foreign keys. By default, the column name of foreign keys have the same name as the foreign key names. Use the `columnName` attribute of `foreignkey` elements to override their default column names. For example:

```
<foreignkey
    columnName="AccountID"
    ...
    name="Account"/>
```

Adding the suffix `ID` to the column names of foreign keys helps database administrators identify columns in the database that Guidewire uses as foreign keys.

If you add a foreign key as an extension to a base entity, follow the best practice of adding a prefix or suffix to the name. For example:

```
<foreignkey
    columnName="AccountID"
    ...
    name="Account_Ext"/>
```

Typelist Best Practices

A *typelist* represents a set of allowed values for specific fields on entities in the data model. A *typecode* represents an individual value within a typelist. A typecode comprises:

- A code that the database stores as a column value
- A name that the user interface displays
- A priority setting that drop-down lists use to order they typecode names that they display

As a best practice, Guidewire recommends that you edit typelist definitions by using the Typelist editor in Guidewire Studio.

See also

- "Working with Typelists" on page 245 in the *Configuration Guide*

Observe Typelist Naming Conventions

The components of a typelist have these naming conventions:

Typelist Component	Naming Conventions	Example
Typelist	Names begin with a capital letter. Medial capitals separate words.	`ActivityCategory`
Code	Names contains only lower-case letters. Underscores (_) separate words.	`approval_pending`
Name	Each word in a name begins with a capital letter. Spaces separate words.	`Approval Pending`

Add a Suffix to New Typelists and Typecode Extensions

To avoid future naming conflicts when Guidewire adds or changes base typelists and typecodes, Guidewire recommends that you append the suffix `_Ext` to your typelist names and typecode codes.

As an example of new typelists, name one that represents types of medical procedures `MedicalProcedureType_Ext`. Name the typecodes in your new typelists without the suffix `_Ext`.

As an example of new typecodes in a base typelist, the following `AddressType` typelist has a new typecode for service entrances.

```
Code                 Name              Description       Priority   Retired
-------------------  ----------------  -----------       --------   -------
billing              Billing           Billing           -1         false
business             Business          Business          -1         false
home                 Home              Home              -1         false
other                Other             Other             -1         false
service_entrance_Ext Service Entrance  Service Entrance  -1         false
```

Data Model Best Practices Checklist

Use the following checklist before you complete your data model configuration tasks to ensure that your data model follows Guidewire best practices.

Best Practice to Follow	Best Practice Was Followed
"Observe General Entity Naming Conventions" on page 7	☐
"Add a Prefix or Suffix to Entity Extensions" on page 8	☐
"Use Singular for Field Names Except for Arrays" on page 8	☐

Best Practice to Follow	Best Practice Was Followed
“Add ID as a Suffix to Column Names for Foreign Keys” on page 8	☐
“Observe Typelist Naming Conventions” on page 9	☐
“Add a Suffix to New Typelists and Typecode Extensions” on page 9	☐

chapter 2

User Interface Best Practices

BillingCenter uses *page configuration format* (PCF) files to render the BillingCenter application. PCF files contain metadata definitions of the navigation, visual components, and data sources of the user interface. Display keys provide the static text that visual components of the user interface display.

This topic includes:

Page Configuration Best Practices

You can change the user interface of the BillingCenter application by adding, changing, and removing PCF files from the configuration of your BillingCenter instance. As a best practice, Guidewire recommends that you edit PCF files by using the Page Configuration (PCF) editor in Guidewire Studio.

See also

- "Using the PCF Editor" on page 269 in the *Configuration Guide*

Modify Base PCF Files Whenever Possible

As a best practice, Guidewire recommends that you modify the base configuration files wherever they can be modified. Create new files only when absolutely necessary.

Add a Suffix to New PCF Files to Avoid Name Conflicts

As a best practice, Guidewire recommends that every page configuration file in your BillingCenter instance must have a unique file name. The location of page configuration files within the folder structure of page configuration resources does *not* ensure uniqueness. To avoid future naming conflicts when Guidewire adds or changes base page configurations, Guidewire recommends that you append the suffix `_Ext` to the names of your page configuration files.

For example, name a new list view for contacts on an account `AccountContacts_ExtLV`.

Display Keys Best Practices

A *display key* represents a single piece of user-viewable text. Display keys in BillingCenter are equivalent to defined constants in programming languages. As a best practice, Guidewire recommends that you edit your display key definitions by using the Display Keys editor in Guidewire Studio.

This topic contains:

See also

- "Using the Display Keys Editor" on page 135 in the *Configuration Guide*

Use Display Keys to Display Text

As a best practice, Guidewire recommends that you define a display key for any string or text value that you use anywhere in BillingCenter. This includes, but is not limited to, the following:

- Field labels
- String values in Gosu code
- Error and validation messages

Create Unique Display Keys

As a best practice, Guidewire recommends that you define a unique display key for each string that you need to display.

Do not re-use existing display keys in new contexts, because the meaning of the English word can differ between old and contexts. For example, even though the English word is the same, a *check* issued by a financial institution is not the same as an instruction to *check* (select) a checkbox. It is impossible to translate the single word *check* to cover both English meanings.

Use Placeholder Variables in Display Keys for Variable Data Only

As a best practice, Guidewire recommends that you use placeholder variables in display keys only for truly variable data. For example, do not do the following.

```
//Bad example
Java.Web.New = New {0}
```

In this case, the variable `{0}` can take on a value of Account, Trouble Ticket, User, or Contact.

Instead, do the following.

```
//Good examples
Java.Web.New.Account = New Account
Java.Web.New.TroubleTicket= New Trouble Ticket
Java.Web.New.Contact = New Contact
...
```

In many languages, nouns have a gender. Modifiers (adjectives) that modify that noun also have a gender that matches the gender of the noun. Thus, it is not possible to translate `New {0}` accurately for all possible substitutions.

Avoid Insertion of One Display Key into Another Display Key

As a best practice, Guidewire recommends that you not insert, or substitute, one display key into another key. For example, do not do the following.

```
//Bad examples
Java.Web.Account = {Term.Account.Proper}
Java.Web.New.TroubleTicket = New {Term.TroubleTicket.Proper}
Java.Web.Error.Message = {0} - {1}
```

Notice that each display key definition uses a variable that BillingCenter replaces at run-time with one or more additional display keys. As the word substitution is not known in advance, it is impossible to provide an accurate translation of the expanded phrase.

Instead, do the following.

```
//Good examples
Java.Web.Account = Account
Java.Web.New.TroubleTicket = New Trouble Ticket
Java.Web.Error.Contact = An error occurred - First name cannot be empty for Contact.
```

Observe Display Key Naming Conventions

As a best practice, Guidewire recommends that display key names begin with a capital letter. Medial capitals separate words in compound display key names. For example:

```
ContactDetail
```

BillingCenter represents display keys within a hierarchical name space. A period (.) separates display key names in the paths of the display key hierarchy. For example:

```
Validation.Contact.ContactDetail
```

Generally, you specify text values for display key names that are leaves on the display keys resource tree. Generally, you do *not* specify text values for display key names that are parts of the path to a leaf display key. In the preceding example, the display keys `Validation` and `Validation.Contact` have no text values, because they are parts of a display key path. The display key `ContactDetail` has a text value, “Contact Detail,” because it is a leaf display key with no child display keys beyond it.

Add a Suffix to New Display Keys to Avoid Name Conflicts

As a best practice, Guidewire recommends that you append the suffix `_Ext` to your new display key names. Including the suffix helps avoid future naming conflicts when Guidewire adds or changes base display keys.

For example, your BillingCenter instance has a branch of the display key hierarchy for text that relates to contact validation.

```
Validation.Contact.ContactDetail
Validation.Contact.NewContact
```

You want to add a display key for the text “Delete Contact.” Add a new display key named `DeleteContact_Ext`.

```
Validation.Contact.ContactDetail
Validation.Contact.DeleteContact_Ext
Validation.Contact.NewContact
```

You can change the text for base display keys to change the text the base application displays. Guidewire recommends that you reuse base display in this way. Do *not* add a new display key with the suffix `_Ext`, which then requires you to modify base PCF files to make use of the new value.

Organize Display Keys by Page Configuration Component

As a best practice, Guidewire recommends that you organize display keys under paths specific to the page configuration component types and PCF file names where display keys appear. For example,

```
LV.Activity.Activities.DueDate
```

User Interface Performance Best Practices

The ways in which you configure the user interface of your BillingCenter instance affects its performance. As performance best practices for user interface configuration, Guidewire recommends that you always do the following:

- "Avoid Post on Change and Client Reflection for Page Refreshes" on page 14
- "Avoid Repeated Calculations of Expensive Widget Values" on page 14
- "Avoid Expensive Calculations of Widget Properties" on page 15
- "Use Application Permission Keys for Visibility and Editability" on page 16

Avoid Post on Change and Client Reflection for Page Refreshes

As a performance best practice, Guidewire recommends that you not use `postOnChange` or client reflection unless you absolutely must refresh your page for users in response to field changes.

Use Post On Change with Extreme Caution

As a performance best practice, Guidewire recommends that you use `postOnChange` fields with extreme caution and only when client reflection cannot be used. Each edit that a user makes to a `postOnChange` field requires a round trip to the server to refresh the page. So, an edit to a `postOnChange` field may create a latency in response for the user. For pages with many queries, a single `postOnChange` can be slow and performance intensive as the server executes each query again. Be sure to remove `postOnChange` whenever your need for immediate page refresh goes away.

Use Client Reflection Instead of Post on Change If Possible

As a performance best practice, Guidewire recommends that you use client reflection instead of `postOnChange` if client reflection satisfies you requirements for page refresh. For example, your page has two range widgets. The choice a user makes in the first range widget determines the choices available in the second range widget. If you make the first range widget a `postOnChange` field, the server incidentally executes every non-cached query on the page again at least once. The user may have to wait a long time before the page lets the user choose from the second range widget. If you use client reflection to update the second widget instead, you avoid a server round trip and performance degradation to execute non-cached queries in the server.

Avoid Repeated Calculations of Expensive Widget Values

As a performance best practice, Guidewire recommends that you avoid repeated evaluation of performance intensive expressions for widget values. Depending on the needs of your application, performance intensive expressions may be unavoidable. However, you can improve overall performance of a page by choosing carefully where to specify the expression within the page configuration.

For example, the following value range expression has a performance problem. Evaluation of the expression requires three foreign key traversals and one array lookup. If the `InvoiceItem` instance is not cached, BillingCenter executes four relational database queries, which makes the evaluation even more expensive.

```
RangeInput
...
valueRange      |invoiceItem.AccountPayer.AccountInfo.Account.AllInvoices
```

If a page with this range input widget has any `postOnChange` fields, BillingCenter potentially evaluates the expression multiple times for each `postOnChange` edit that a user makes.

Use Recalculate on Refresh with Expensive Page Variables Cautiously

BillingCenter evaluates page variables only during the construction of the page, but sometimes you want BillingCenter to evaluate a page variable in response to `postOnChange` edits. If so, you set the `recalculateOnRefresh` property of the page variable to `true`. If a page variable specifies an expensive expression for its `initialValue`, carefully consider whether your page really must recalculate the variable. If you set the `recalculateOnRefresh` property to `true`, BillingCenter evaluates the expression at least once for every `postOnChange` edit to the page.

Although BillingCenter evaluates page variable with `recalculateOnRefresh` set to `true` for each `postOnChange` edit, page variables can yield performance improvements compared to widget values. If several widgets use the same expression for their values, using a page variable reduces the number of evaluations by a factor equal to the number widgets that use it. For example, the `valueRange` of a range input used for drop-down lists in a list view column are evaluated at least once for each row.

Do Not Use Post on Change with List Views

Do *not* use `postOnChange` with list views. Each row that a user inserts or removes causes a server round trip. To update list totals, use client reflection instead.

Avoid Expensive Calculations of Widget Properties

As a performance best practice, Guidewire recommends you place complex or expensive expressions in page variables instead of directly in widget properties, especially the `editable`, `visible`, `available`, and `required` properties. BillingCenter may need to evaluate expressions in widget properties in multiple contexts before it displays the page. BillingCenter evaluates page variables only during the construction of the page, not during subsequent `postOnChange` edits.

The following example suffers a performance problem. It assigns a performance intensive expression to the `visible` property of a widget.

```
Input: myInput
  ...
  id              |myInput
  ...
  visible         |activity.someExpensiveMethod()
```

The following modified sample code improves performance. It assigns a performance intensive expression to a page variable. BillingCenter evaluates page variables only once before it displays a page, regardless how many contexts under which it evaluates widget properties on the page.

```
Variables
  ...
  initialValue  |activity.someExpensiveMethod()
  name          |expensiveResult

--------------------------------
Input: myInput
  ...
  id            |myInput
  ...
  visible       |expensiveResult
```

Use Application Permission Keys for Visibility and Editability

As a performance best practice, Guidewire recommends that you structure the page configuration of your user interface so application permission keys determine visibility and editability. The `visible` and `editable` properties are evaluated many times during the lifetimes of locations and widgets. This is the most common pattern for user access in business applications.

For example, use the following Gosu expression in the `visible` property of an **Edit** button on a panel that displays information about an account.

```
perm.Account.edit(anAccount)
```

Application permission keys evaluate the current user against general system permissions and the access control lists of specific entity instances.

User Interface Best Practices Checklist

Use the following checklist before you complete your user interface configuration tasks to ensure that your user interface configuration follows Guidewire best practices.

Best Practice to Follow	Best Practice Was Followed
“Modify Base PCF Files Whenever Possible” on page 11	☐
“Add a Suffix to New PCF Files to Avoid Name Conflicts” on page 12	☐
“Use Display Keys to Display Text” on page 12	☐
“Create Unique Display Keys” on page 12	☐
“Use Placeholder Variables in Display Keys for Variable Data Only” on page 12	☐
“Avoid Insertion of One Display Key into Another Display Key” on page 13	☐
“Observe Display Key Naming Conventions” on page 13	☐
“Add a Suffix to New Display Keys to Avoid Name Conflicts” on page 13	☐
“Organize Display Keys by Page Configuration Component” on page 14	☐
“Avoid Post on Change and Client Reflection for Page Refreshes” on page 14	☐
“Avoid Repeated Calculations of Expensive Widget Values” on page 14	☐
“Avoid Expensive Calculations of Widget Properties” on page 15	☐
“Use Application Permission Keys for Visibility and Editability” on page 16	☐

chapter 3

Rules Best Practices

BillingCenter rules comprise hierarchies of conditions and actions that implement complex business logic. As a best practice, Guidewire recommends that you edit rules by using the Rules editor in Guidewire Studio.

This topic includes:

See also

- "Rules: A Background" on page 11 in the *Rules Guide*

Rules Naming Best Practices

Guidewire recommends a number of rule naming best practices to help you identify and locate specific rules during configuration, testing, and production.

Observe Rule Naming Conventions

Each rule name within a rule set must be unique. To help ensure uniqueness, Guidewire recommends that you follow the best practices naming conventions for rules described in this topic. In addition, these naming conventions help you quickly identify each rule within the complex hierarchy of rules in your Guidewire instance during testing and in production.

The basic format for a rule name has two parts:

```
Identifier - Description
```

Follow these conventions for *Identifier* and *Description*:

- Separate *Identifier* from *Description* with a space, followed by hyphen, followed by a space.
- Limit *Identifier* to eight alphanumeric characters.

IMPORTANT Guidewire truncates Identifier values that exceed eight characters if you include the actions.ShortRuleName property in rule actions to display rule names in messages that you log or display. Guidewire also truncates Identifier values that exceed eight characters in automatic log messages if you enable the RuleExecution logging category and set the server run mode to Debug.

- Begin *Identifier* with up to four capital letters to identify the rule set or parent rule of which the rule is a member.
- End *Identifier* with at least four numerals to identify the ordinal position of the rule within the hierarchy of rules in the set.
- For *Description* values, keep them simple, short, and consistent in their conventions.
- Limit the total length of rule names to 60 characters.

For example:

```
CV000100 - Future loss date
```

Rule Naming Summary Principles

Remember these principles for rule names:

- Rule names are unique within a rule set.
- Rules numbers are sequential to mimic the order of rules in the fully expanded set.

The following example demonstrates these principals.

```
Claim Validation Rules
  CV001000 - Future loss date
  CV002000 - Policy expiration date after effective date
  CV002500 - Not Set: Coverage in question
  CV003000 - Injury
    CVI03100 - Workmen's Compensation
      CVIW3110 - Claimant exists
      CVIW3120 - Not Set: Injury description
    CV103900 - Default
  CV004000 - Expected recovery exceeds 100
```

Root Rules Naming Conventions

Consider the following example rule set, Claim Validation Rules. The identifiers of rules in this set all begin with CV, a code to identify "Claim Validation" rules.

```
Claim Validation Rules
  CV001000 - Future loss date
  CV002000 - Policy expiration date after effective date
  CV003000 - Injury
    CVI03100 - Workmen's Compensation
    CV103900 - Default
  CV004000 - Expected recovery exceeds 100
```

The rule set contains four root rules, with identifiers CV001000, CV002000, CV003000, and CV004000. The numbers at the end of the identifiers, 1000, 2000, 3000 and 4000, are units of one thousand. This spread of numbers lets you add new root rules between existing ones without renumbering. You want identifier numbers for rules in a set to remain in sequential order to mimic the order of rules within the fully expanded set.

For example, you want to add a rule between CV002000 and CV003000. Assign the new rule the identifier CV002500.

```
Claim Validation Rules
  CV001000 - Future loss date
  CV002000 - Policy expiration date after effective date
  CV002500 - Not Set: Coverage in question
  CV003000 - Injury diagnosis validity dates
```

```
    CVI03100 - Workmen's Compensation
    CV103900 - Default
  CV004000 - Expected recovery exceeds 100
```

Parent and Child Rules Naming Conventions

Many rule sets achieve their business function with simple rules in the root of the set. In preceding example, rules CV001000, CV002000, CV002500, and CV004000 are simple root rules. Frequently however, rule sets achieve their business function *only* with a hierarchy of parent and child rules. In the example, rule CV003000 is a parent rule with two child rules.

When you add child rules to a parent, follow these conventions:

- Expand the beginning code for the child rules with an additional letter to identify their parent.
- Assign each child rule an ending number that falls between the number of the parent and the sibling rule that follows the parent.
- Assign the children a spread of numbers so you can add more children later without renumbering.

In the preceding example, the identifiers for the child rules of CV003000 all begin with CVI, a code to identify "Claim Validation Injury" rules.

```
Claim Validation Rules
...
  CV003000 - Injury
    CVI03100 - Workmen's Compensation
    CV103900 - Default
  CV004000 - Expected recovery exceeds 100
```

The spread of numbers for child rules of a root parent rule generally are units of one hundred. This spread of numbers lets you add new child rules between existing ones without renumbering. Most importantly, the numbers of child rules must fall between the numbers of their parent rule and the sibling rule that follows their parent. In this example, the numbers for child rules satisfy both conventions.

The parent and child naming convention applies to another, third level of children. For example, you want to add two new child rules to the rule CVI03100 - Workmens's Compensation. Begin the child identifiers with CVIW, a code to identify "Claim Validation Injury Workmen's Compensation" rules. At the third level of a rule set hierarchy, the spread of numbers for the child rules generally are units of ten.

```
Claim Validation Rules
...
  CV003000 - Injury
    CVI03100 - Workmen's Compensation
      CVIW3110 - Claimant exists
      CVIW3120 - Not Set: Injury description
    CV103900 - Default
  CV004000 - Expected recovery exceeds 100
```

Observe Operating System Length Restrictions on Rule Names

Guidewire stores rules in files within a directory structure that mimics the rule set category structure in Studio. The fully qualified path for a rule file can be quite long, depending on:

- The length of the path to your Guidewire installation directory
- The length of the path from your installation directory to the root of the rules directory, which is 34 characters:

  ```
  modules/configuration/config/rules
  ```

- The depth of the hierarchy of rule categories, rule sets, rules, and parent/child rules
- The length of individual rule names

In addition, you must add four characters to the path for the name of each rule category, rule set, and parent rule.

You must understand the file system implications of rule names. Windows file systems have a file path limit of 255 characters. To avoid exceeding the limit, Guidewire recommends that you:

- Avoid a long path to your Guidewire installation directory.

- Avoid deep hierarchies of rule set categories.
- Avoid hierarchies of parent and child rules within a rule set deeper than three levels.
- Avoid long names for rule set categories, rule sets, and rules.

Get and Display Rule Names in Messages

As a best practice, Guidewire recommends that you get and display rule names in messages. So, following the Guidewire best practices for rule names helps you identify specific rules in messages to users, in print statements for testing, and in log messages. The alphabetic beginning of a rule identifier helps you find the rule set or parent rule that contains the rule. The numeric ending helps you determine the order of a rule in a rule set or parent rule.

For example, you want to test a regional validation rule, `RGV01000 - No zones in lookup`. The identifier portion of the rule name begins with `RGV`. That identifies the rule as a member of the `RegionValidationRules` rule set. The identifier ends with `01000`. That indicates that rule is near the beginning of the rule set. The rule has the following definition.

```
RGV01000 - No zones in lookup
Rule Conditions:
Region.getOrganization() = null

Rule Actions:
producerCode.reject(null, null, "loadsave",
        "There are zones that do not appear in the zone lookup dataset."
)
```

As written, the rejection message that the rule action displays makes it difficult to determine exactly which rule caused an update to fail. To help identify the specific rule in the rejection message, use the `actions.getRule().DisplayName` property to include the identifier portion of the rule name in the message.

```
Rule Actions:
Region.reject( null, null, "loadsave"
        "There are zones that do not appear in the zone lookup dataset. Rule:" +
        actions.getRule().DisplayName.substring(8)
)
```

By including the rule display name in the rule action, users see the following statement in the **Validation** window when the rule action executes.

```
There are zones that do not appear in the zone lookup dataset. Rule: RGV01000
```

Note: In actual practice, Guidewire recommends that you make all `String` values into display keys.

Assign a Dedicated Rules Librarian to Manage Rule Names

As a best practice, Guidewire recommends that you appoint someone in your organization to develop and enforce simple and consistent naming conventions for rule categories, rule sets, and rule names. This helps ensure that naming standards are followed to make sure that rule identifiers readily identify specific rules within the total catalog of rules in your Guidewire instance.

Rules Performance Best Practices

Guidewire recommends performance best practices for rules to help you avoid known performance issues.

Purge Unused and Obsolete Rules Before Upgrading

As a best practice, Guidewire recommends that you purge unused and obsolete rules from your BillingCenter configuration. This improves the upgrade process because BillingCenter does not spend time to evaluate inactive rules that are unused or obsolete.

Rules Best Practices Checklist

Use the following checklist before you complete your rule configuration tasks to ensure that your rules follow Guidewire best practices.

Best Practice to Follow	Best Practice Was Followed
“Observe Rule Naming Conventions” on page 17	☐
“Rule Naming Summary Principles” on page 18	☐
“Root Rules Naming Conventions” on page 18	☐
“Parent and Child Rules Naming Conventions” on page 19	☐
“Observe Operating System Length Restrictions on Rule Names” on page 19	☐
“Get and Display Rule Names in Messages” on page 20	☐
“Assign a Dedicated Rules Librarian to Manage Rule Names” on page 20	☐
“Purge Unused and Obsolete Rules Before Upgrading” on page 20	☐

chapter 4

Gosu Language Best Practices

Gosu is a general-purpose programming language built on top of the Java Virtual Machine. BillingCenter uses Gosu as its common programming language.

This topic includes:

See also

- "Gosu Introduction" on page 15 in the *Gosu Reference Guide*

Gosu Naming and Declaration Best Practices

Guidewire recommends a number of best practices for naming and declaring Gosu variables, functions, and classes.

Observe General Gosu Naming Conventions

As a best practice, Guidewire recommends the following general naming conventions.

Language Element	Naming Conventions	Examples

Variable names	Name variables in mixed case, with an initial lower case letter and a medial capital for each internal word. Name variables mnemonically so that someone reading your code can understand and easily remember what your variables represent. Do *not* use single letters such as "x" for variable names, except for short-lived variables, such as loop counts.	`nextPolicyNumber`
		`firstName`
		`recordFound`

Function names	Compose function names in verb form. Name functions in mixed case, with an initial lower case letter and medial capitals for each internal word. Add the suffix `_Ext` to the ends of function names to avoid future naming conflicts if Guidewire adds or changes base functions.	`getClaim_Ext()` `getWageLossExposure_Ext()`
Class name	Compose class names in noun form. Name classes in mixed case, with an initial upper case and medial capitals for each internal word. Add the suffix `_Ext` to the ends of class names to avoid future naming conflicts if Guidewire adds or changes base classes.	`StringUtility_Ext` `MathUtility_Ext`

Omit Type Specifications with Variable Initialization

Type specifications in variable declarations are optional in Gosu if you initialize variables with value assignments. Whenever you initialize a variable, Gosu sets the type of the variable to the type of the value. As a best practice, Guidewire recommends that you always initialize variables and omit the type specification.

```
var amount = 125.00  // use an intialization value to set the type for a variable
var string = new java.lang.String("") // initialize to the empty string instead of null
```

Add a Suffix to Functions and Classes to Avoid Name Conflicts

To avoid future naming conflicts if Guidewire adds or changes base functions and classes, Guidewire recommends that you append the suffix `_Ext` to your new functions and classes.

For example, name a new function that calculates the days between two dates `calculateDaysApart_Ext(...)`.

Declare Functions Private Unless Absolutely Necessary

As a best practice, Guidewire recommends that you declare functions as public only with good reason. The default access in Gosu is public. So, declare functions as private if you intend them only for use internally within a class or class extension. Always prefix private and protected class variables with an underscore character (_).

Use Public Properties Instead of Public Variables

As a best practice, Guidewire recommends that you convert public variables to properties. Properties separate the interface of an object from the implementation of its storage and retrieval. Although Gosu supports public variables for compatibility with other languages, Guidewire strongly recommends public properties backed by private variables instead of public variables.

The following sample Gosu code declares a private variable within a class and exposes it as a public property by using the `as` keyword. This syntax makes automatic getter and setter property methods that the class instance variable backs.

```
private var _firstName : String as FirstName  // Delcare a public property as a private variable.
```

Avoid declaring public variables, as the following sample Gosu code does.

```
public var FirstName : String                 // Do not declare a public variable.
```

For general information, see “Properties” on page 201 in the *Gosu Reference Guide*.

Do Not Declare Static Scope for Mutable Variables

As a best practice, Guidewire recommends that you do *not* use `static` scope declaration for fields that an object modifies during its lifetime. Static fields have application scope, so all sessions in the Java Virtual Machine (JVM) share them. All user sessions see the modifications that made any user session makes to static properties.

For example, the following sample Gosu code is a bad example.

```
class VinIdentifier {
  static var myVector = new Vector()  // All sessions share this static variable.

  static function myFunction(){
    myVector.add("new data")     // Add data for the entire JVM, not just this session.
  }

}
```

Use Extensions to Add Functions to Entities

As a best practice, Guidewire recommends that you add functions that operate on entities as extensions to the existing entity type instead of as static functions on separate utility classes.

Implement Functions that Operate on Single Entity Instances as Extensions

If you want a new function that operates on single instances of an entity type, declare the new function in a separate class extension to that entity type.

For example, you want a new function to suspend an account. Name your new function `suspend`. Do not declare the function as static with a `Account` instance as its parameter. Instead, declare the function as an extension of the `Account` class, so callers can invoke the method directly on a `Account` instance. For example:

```
if account.suspend() {
  // Do something to suspend the account.
}
```

Package Entity Extensions for an Entity Type in a Single Package

Package all of your extensions for an entity type together in a package with the same name as the entity they extend. Do not place all of your entity extensions in a single package. Place all of your extension packages in a package folder that identifies your organization.

For example, place all of your extensions to the `Activity` entity type in a package named `com.`*`CustomerName`*`.activity`.

Match Capitalization of Types, Keywords, and Symbols

The Gosu language is case sensitive.

Guidewire entity types are case insensitive, but it is best to write your code as if they are case sensitive.

Use standard capitalization of program symbols and language elements:

- **Package and subpackage names** – All lower case
- **Type names** – Initial upper case
- **Method names** – Initial lower case
- **Property names** – Initial upper case
- **Variable names** – Initial lower case
- **Keywords, such as ‘var’ and ‘if’** – All lower case
- **All other language elements** – As defined

See also

- "Case Sensitivity" on page 25 in the *Gosu Reference Guide*

Gosu Commenting Best Practices

As a best practice, Guidewire recommends these styles and usages of comments:

- Block Comments
- Javadoc Comments
- Single-line Comments
- Trailing Comments
- Using Comment Delimiters to Disable Code

As a commenting best practice, always place block comments before every class and method that you write. Briefly describe the class or method in the block comment. For comments that you place within methods, use any of the commenting styles to help clarify the logic of your code.

Block Comments

Block comments provide descriptions of libraries and functions. A block comment begins with a slash followed by an asterisk (/*). A block comment ends with an asterisk followed by a slash (*/). To improve readability, place an asterisk (*) at the beginnings of new lines within a block comment.

```
/*
 * This is a block comment.
 * Use block comments at the beginnings of files and before
 * classes and functions.
 */
```

Place block comments at the beginnings of files, classes, and functions. Optionally, place block comments within methods. Indent block comments within functions to the same level as the code that they describe.

Javadoc Comments

Javadoc comments provide descriptions of classes and methods. A Javadoc comment begins with a slash followed by two asterisks (/**). A Javadoc comment ends with a single asterisk followed by a slash (*/).

```
/**
 * Describe method here--what it does and who calls it.
 * How to Call: provide an example of how to call this method
 * @param parameter description (for methods only)
 * @return return description (for methods only)
 * @see reference to any other methods,
 * the convention is
 * <class-name>#<method-name>
 */
```

Block comments that you format using Javadoc conventions allow automated Javadoc generation.

Single-line Comments

Single-line comments provide descriptions of one or more statements within a function or method. A single-line comment begins with double slashes (//) as the first two characters two non-whitepsace characters on a line.

```
//  Handle the condition
if (condition) {
  ...
}
```

If you cannot fit your comment on a single line, use a block comment, instead.

Trailing Comments

Trailing comments provide very brief descriptions about specific lines of code. A trailing comment begins with double slashes (//) following the code that the comment describes. Separate the double slashes from the code with at least two spaces.

```
if (a == 2) {
  return true  // Desired value of 'a'
}
else {
  return false // Unwanted value of 'a'
}
```

If you place two or more trailing comments on lines in a block of code, indent them all to a common alignment.

Using Comment Delimiters to Disable Code

Use a single-line comment delimiter (//) to comment out a complete or partial line of code. Use a pair of block comment delimiters (/*, */) to comment out a block of code, even if the block you want to comment out contains block comments.

Note: Do not use single-line comment delimiters (//) on consecutive lines to comment out multiple lines of code. Use block comment delimiters (/*, */), instead.

Use caution if you include slashes and asterisks within block comments to set off or divide parts of the comment. For example, do not use a slash followed by a line of asterisks as a dividing line within a block comment. In the following example, the compiler interprets the dividing line as the beginning of a nested comment (/*) but finds no corresponding closing block comment delimiter (*/).

```
/*
    The dividing line starts a nested block comment and causes an unclosed comment compiler error.
    //*******************************************************************************************
*/
```

In the preceding example, compilation fails due to an unclosed comment. The following example avoids compilation errors by inserting a space between the slash and the first asterisk.

```
/*
    The dividing line does not start a nested block comment and causes no compiler error.
    // ******************************************************************************
*/
```

Single-line comment delimiters (//) are ignored in block comments.

Gosu Coding Best Practices

Guidewire recommends a number of best practices for Gosu code.

Use Whitespace Effectively

Guidewire recommends the following best practices for effective use of whitespace:

- Add spaces around operators.

```
premium = Rate + (minLimit - reductionFactor)  // proper form
premium=Rate+(minLimit-reductionFactor)        // improper form
```

- Add no spaces between parentheses and operands.

```
((a + b) / (c - d))        // proper form
( ( a + b ) / ( c - d ) )  // improper form
```

- Indent logical blocks of code by two spaces only.
- Add a blank line after code blocks.
- Add two blank lines after methods, even the last method in a class.

Use Parentheses Effectively

As a best practice, Guidewire recommends that you always use parentheses to make explicit the operator order of precedence in computational expressions. Otherwise, your computations can easily produce inappropriate results. In the following sample Gosu code, the computation expression without parenthesis produces incorrect results.

```
premium = (rate + limit) * (10.5 + deductible) / (autoGrade - 15)  // proper form
premium = rate + limit * 10.5 + deductible / autoGrade - 15        // improper form, bad results
```

Math operators have a natural order of precedence, which controls the order of incremental computations in compound computations if parentheses are absent. The following sample Gosu code makes explicit the operator order of precedence in the computation without parentheses in the preceding example.

```
premium = rate + (limit * 10.5) + (deductible / autoGrade) - 15    // natural order of precedence
```

Use Curly Braces Effectively

Guidewire recommends the following best practices for effective use of curly braces:

- Surround every logical block, even single-statement blocks, with curly braces ({}).
- Put the opening curly braces ({) on the same line that starts the block.
- Put the closing curly brace (}) on the line after the last statement in the block, aligned with the starting column of the block.

```
if(premium <= 1000) {                  // Put opening curly brace on line that starts the block.
  print("The premium is " + premium)   // Surround even single-line blocks with curly braces.
}                                      // Put closing curly brace on line after last statement.
```

Program Defensively Against Conditions that Can Fail

As a best practice, Guidewire recommends that you program defensively. Always assume that conditions might fail. Follow these recommendations to avoid common but easily missed programming flaws, such as potential null pointer exceptions and out-of-bounds exceptions.

Use case-insensitive comparisons

Use the `equalIgnoreCase` method on the string literal to compare it with the value of a variable. Case mismatch can cause comparisons to fail unintentionally.

```
if("Excluded".equalsIgnorecase(str)) {  // proper comparison method
  print("They match.")
}

if(string.equals("Excluded")) {  // improper comparison method
  print("They do NOT match.")
}
```

Check for null values

If your code cannot handle a variable that contains non-null values, check the variable for `null` before you access properties on the variable.

```
function formatName (aUser : User) {
  if (aUser != null) {  // Check for null to avoid a null pointer exception later in the code.
    print(aUser.DisplayName)
   }
   else {
     print("No user")
   }
}
```

Also, check variables for `null` before you call methods on variables:

```
function addGroup (aUser : User, : aGroupUser : GroupUser) {
  if (aUser != null) {  // Check for null to avoid a null pointer if you call methods on the variable.
     aUser.addToGroup(aGroupUser)
  }
}
```

Consider the following issues If you do not check variables for `null`:

- Accessing a property on a variable may use null-safe property access, which can cause null pointer exceptions in later code that handles only non-null values.
- Calling a method on a variable risks a null pointer exception.

Allow default null-safe property access

If your code can handle a variable that contains null values, you can rely on null-safe access of simple properties in most cases. With null-safe property access, the entire expression evaluates to `null` if the left-side of a period operator is `null`.

For example, the following sample Gosu code returns `null` if the `user` object passed in by the caller is `null`. Default null-safe property access prevents a null pointer exception in the `return` statement.

```
function formatName (aUser : User) : String {
   return aUser.DisplayName  // Default null-safe access returns null if the passed-in user is null.
}
```

Use explicit null-safe operators

If your code can handle an object path expression that evaluates to `null`, use the explicit Gosu null safe operators:

- `?.` – Null-safe access to properties and methods
- `?[]` – Null-safe access to arrays

The following sample Gosu code uses the explicit null-safe operator `?.` to check for `null` to avoid a null pointer exception while accessing a property. If `aUser` is `null`, the entire object path expression `aUser?.DisplayName` evaluates to `null`.

```
function formatName (aUser : User) : String {
   return aUser?.DisplayName  // An explicit null-safe operator returns null if aUser is null.
}
```

The following sample Gosu code uses the explicit null safe operator `?.` to check for `null` to avoid a null pointer exception while calling a method. If `aUser` is `null`, Gosu does not call the method `addToGroup`, and the entire expression `aUser?.addToGroup(aGroupUser)` evaluates to null.

```
function addGroup (aUser : User, aGroupUser: GroupUser) {
  aUser?.addToGroup(aGroupUser)  // An explicit null-safe operator returns null if aUser is null.
}
```

The following sample Gosu code uses the explicit null-safe operator `?[]` to check for `null` to avoid a null pointer exception while accessing an array. If `strings[]` is `null`, the entire expression `strings?[index]` evaluates to `null`. However, the null-safe operator does not avoid out-of-bounds exceptions if `strings[]` is non-null and the value of `index` exceeds the array length.

```
function getOneString (strings  : String[], index : int) : String {
  return = strings?[index]  // An explicit null-safe operator evaluates to null if an array is null.
}
```

See also

- For more information about null-safe operators, see "Handling Null Values In Expressions" on page 87 in the *Gosu Reference Guide*.

Check boundary conditions

In `for` loops, check for boundary conditions ahead of the loop. Entering a loop if the boundary condition is satisfied already can cause null pointer exceptions and out-of-bounds exceptions.

Use structured exception handling

Use `try`/`catch` blocks wherever required and possible. If you want to catch multiple exceptions, handle them in a hierarchy from low-level, specific exceptions to high-level, generic exceptions.

Omit Semicolons as Statement Delimiters

Semicolons as statement delimiters are optional in Gosu. As a best practice, Guidewire recommends that you omit semicolons. They are unnecessary in almost all cases, and your Gosu code looks cleaner and easier to read without them.

```
// Omit semicolons with statements on separate lines.
print(x)
print(y)
```

Gosu requires semicolons only if you place multiple statements on a single line. As a best practice, Guidewire generally recommends against placing multiple statements on a single line. Exceptions include simple statement lists declared in-line within Gosu blocks.

```
// Include semicolons with multiple statements on a single line.
var adder = \ x : Number, y : Number -> { print("I added!"); return x + y; }
```

Use != Instead of <> as the Inequality Operator

Use the `!=` form instead of the `<>` form of the inequality operator. The `<>` form in earlier versions of Gosu is deprecated. Studio provides a code inspection to flag this issue.

```
if (activitySubject != row.Name.text) {  // The <> form of inequality operator is deprecated.
  ...
}
```

Observe Null Safety with Equality Operators

The equality and inequality comparison operators `==` and `!=` are null safe if one side or the other evaluates to `null`. Operators that are null safe do not throw null pointer exceptions. As a best practice, Guidewire recommends that you use these comparison operators instead of the `equals` method on objects.

```
if (variable1 == variable2) {        // Comparison operators are null safe.
  print("The variables are equal.")
} else {
  print("The variables are NOT equal.")
}
```

Rewrite Comparison Operators to Avoid the Equals Method

Do *not* write Gosu code that uses the `equals` method, because it is not type safe.

```
if (activitySubject.equals(row.Name.text) {  // This expression is not null safe.
  ...
}
```

As a best practice, Guidewire recommends that you rewrite your Gosu code with comparison operators instead of `equals` methods to make your code type safe and easier to read.

```
if (activitySubject == row.Name.text) {  // This is expression is null safe and easier to read.
  ...
}
```

Avoid Implicit Type Coercion with Comparison Operators

Be aware of type coercion that occurs if you use the comparison operators `==` and `!=` with operands of different types. For example, the following expression evaluates to `true`.

```
1 == "1"
```

Gosu implicitly coerces `"1"` to an integer value of `1`, without explicit casting. Implicit coercion is convenient and powerful, but it can cause unexpected results if used without caution.

Gosu produces compiler warnings for implicit coercions. As a best practice, Guidewire recommends that you take these warnings seriously. If you want the coercion, explicitly cast the operand on the right by using an `as` *Type* expression. If you do *not* want the coercion, rewrite your code to avoid the implicit coercion.

For example, the following expression compares a date value to a string representation of a date value.

```
(dateValue == "2011-11-15")
```

Because Gosu coerces of the string `"2011-11-15"` to a date, rewrite the code with an explicit type cast.

```
(dateValue == "2011-11-15" as DateTime)
```

Use typeis Expressions for Automatic Downcasting

As a best practice, Guidewire recommends that you use `typeis` expressions to compare the type of an object with a given subtype. After a `typeis` expression, Gosu automatically downcasts subsequent references to the object if the object is of the type or a subtype of the original `typeis` expression. Use `typeis` expressions for automatic downcasting to improve the readability of your code by avoiding redundant and unnecessary casts.

The automatic downcasting of `typeis` expressions is particularly valuable for `if` statements and similar Gosu flow of control structures. Within the code block of an `if` statement, you can omit explicitly casting the object as its subtype. Gosu confirms that the object is the more specific subtype and considers it be the subtype within the `if` code block.

The following sample Gosu code shows a pattern for how to use a `typeis` expression with an `if` statement.

```
var variableName : TypeName             // Declare a variable as a high-level type.

if (variableName typeis SubtypeName) {  // Downcast the variable to one of its subtypes.
    ...                                 // Use the variable as its subtype without explicit casting.
}
```

The following sample Gosu code follows the pattern and declares a variable as an `Object`. The `if` condition downcasts the variable to its more specific subtype, `String`.

```
var x : Object = "nice"    // Declare a String variable as type Object.
var strlen = 0

if (x typeis String) {     // Downcast the Object variable to its subtype String.
  strlen = x.length        // Use the variable as a String without explicit casting.
}
```

Because Gosu propagates the downcasting from the `if` condition into the `if` code block, the expression `x.length` is valid. The `length` property is on `String`, not `Object`.

The following sample Gosu code is equivalent to the preceding example, but it redundantly casts the variable as a `String` within the `if` code block.

```
var x : Object = "nice"          // Declare a String variable as type Object.
var strlen = 0

if (x typeis String) {           // Downcast the Object variable to its subtype String.
  strlen = (x as String).length  // Explicit casting as String is redundant and unnecessary.
}
```

For general information, see “Basic Type Checking” on page 369 in the *Gosu Reference Guide*.

Observe Loop Control Best Practices

Gosu provides `for()`, `while()`, and `do…while()` statements for loop control. Guidewire recommends a few best practices for your loop control logic.

Implement Conditional Operators in Loop Conditions Correctly

As a best practice, Guidewire recommends that you verify the conditional operators in your conditional expressions to be certain that you fully satisfy the requirements for your loop control logic. For example, <, >, or = might need to be <=, >=, or !=.

Interrupt Loop Execution as Early as Possible

As a best practice, Guidewire recommends that you interrupt loop execution as early as possible with `continue` or `break` commands.

Use ‘break’ to Break Out of Loop Iteration Altogether

The `break` command stops execution of the loop altogether, and program execution proceeds immediately to the code that follows the loop.

The following sample Gosu code breaks out of the loop altogether on the fourth iteration, when `i` equals `4`.

```
for (i in 1..10) {
  if (i == 4) {
    break         // Break out of the loop on the fourth iteration.
  }

  print("The number is " + i)
}

print("Stopped printing numbers")
```

The preceding sample Gosu code produces the following output.

```
The number is 1
The number is 2
The number is 3
Stopped printing numbers
```

Notice that the loop stops executing on the fourth iteration, when `i` equals `4`.

Use ‘continue’ to Continue Immediately with the Next Loop Iteration

The `continue` command stops execution of the current iteration, and the loop continues with its next iteration.

The following sample Gosu code interrupts the fourth iteration, when `i` equals `4`, but the loop continues executing through all remaining iterations.

```
for (i in 1..10) {
  if (i == 4) {
    continue      // End the fourth iteration here.
  }

  print("The number is " + i)
}
```

The preceding sample code produces the following output.

```
The number is 1
The number is 2
The number is 3
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
The number is 10
```

Notice that the loop continues through all nine iterations, but it interrupts the fourth iteration, when `i` equals `4`.

Return from Functions as Early as Possible

As a best practice, Guidewire recommends that functions return as early as possible to avoid unnecessary processing.

The following sample Gosu code is inefficient. The function uses a variable unnecessarily, and it does not return a result as soon as it detects the affirmative condition.

```
public function foundThree() : boolean {
  var threeFound = 0              // A numeric variable is unnecessary to return a boolean result.

  for (x in 5) {
    if (x == 3) {
      threeFound = threeFound + 1  // The loop keeps iterating after third one is found.
    }

  }

  return threeFound >= 1           // The function returns long after third one is found.
}
```

The following modified sample code is more efficient. The function returns a result as soon as it detects the affirmative condition.

```
public function foundThree() : boolean {
  for (x in 5) {
    if (x == 3) {
      return true                  // The function returns as soon as the third one is found.
    }

  }

  return false
}
```

Use Query Builder APIs instead of Find Expressions in New Code

As a best practice, Guidewire recommends that you query the database by using the query builder APIs instead of find expressions in all new code. Guidewire developed the query builder APIs as an alternative way to query the application database more fully than is possible with find expressions. For example, many requires joining tables to produce useful results. The query builder APIs provide a fuller set of features for joining tables than find expressions can specify. Furthermore, Gosu provides these enhanced capabilities using the structure of traditional API libraries rather than through a special language keyword.

See also

- “Query Builder APIs” on page 129 in the *Gosu Reference Guide*

Gosu Performance Best Practices

The ways in which you write your Gosu code affects compile-time and run-time performance. As best practices for improving the performance of your code, Guidewire recommends that you always do the following:

Use the Fastest Technique for String Concatenation

As a performance best practice, Guidewire recommends that your code perform string concatenation using the fastest technique for a given situation. If in doubt, concatenate strings by using a Java string builder.

Concatenation technique	Performance	Sample Gosu code
Concatenation (+) on literals	Fastest	`var aString = "Test" + " Test"`

Gosu string template	Faster	`var aString : String`
		`var anotherString : String = "Test"`
		`aString = "Test ${anotherString}"`

Java string builder	Intermediate	`var aString = new java.lang.StringBuilder()`
		`aString.append("Test")`
		`aString.append(" Test")`

Java string buffer	Moderate	`var aString = new java.lang.StringBuffer()`
		`aString.append("Test")`
		`aString.append(" Test")`

Concatenation (+) on a variable with a literal	Slower	`var aString : String = "Test"` `aString = aString + " Test"`

Concatenation (+) on a variable with a variable	Slowest	`var aString : String = "Test"`
		`var anotherString : String = " Test"`
		`aString = aString + anotherString`

Consider the Order of Terms in Compound Expressions

As a performance best practice, Guidewire recommends that you carefully consider the order of comparisons in compound expressions that use the `and` and or `operators`. Runtime evaluation of compound expressions that use `and` proceed from left to right until a condition fails. Runtime evaluation of compound expressions that use `or` proceed from left to right until a condition succeeds. The order in which you place individual conditions can improve or degrade evaluation performance of compound expressions.

With and Expressions, Place Terms Likely to Fail Earlier

Whenever you use the `and` operator, place the condition that is most likely to fail or the least performance intensive earliest in the compound expression. Use the following formula to help you determine which condition to place first, based on the condition with the lowest value.

```
(100 - failurePercentage) * (performanceCost)
```

For example, you have a condition that you expect to fail 99% of the time, with an estimated performance cost of 10,000 per evaluation. You have another condition that you expect to fail only 1% of the time, with an estimated performance cost of 100 per evaluation. According to the formula, place the second condition earliest because it has the lowest score.

```
(100 - 99) * 10,000 = 10,000
(100 - 1) * 100 = 9,999
```

You rarely have accurate figures for the failure percentages or performance costs of specific condition. Use the formula to develop an educated guess about which condition to place earliest. In general, give preference to less performance intensive condition. If the performance costs are roughly equal, give preference to condition with a higher percentage of likely failures.

With or Expressions, Place Terms Likely to Pass Earlier

Whenever you use the `or` operator, place the condition that is most likely to succeed earliest in compound expressions.

Avoid Repeated Method Calls Within an Algorithm

Calling a method repeatedly to obtain a value often results in poor performance. As a performance best practice, Guidewire recommends that you save the value from the first method call in a local variable. Then, use the local variable to repeatedly test the value.

The following sample Gosu code suffers a performance problem. It calls a performance intensive method twice to test which value the method returns.

```
if (policy.expensiveMethod() == "first possibility") {  // first expensive call
  // do something
}

else if (policy.expensiveMethod() == "second possibility") {  // second expensive call
  // do something else
}
```

The following modified sample code improves performance. It calls a performance intensive method once and saves the value in a local variable. It then uses the variable twice to test which value the method returns.

```
var expensiveValue = policy.expensiveMethod()  // Save the value of an expensive call.

if (expensiveValue == "first possibility") {  // first reference to expensive result
  // do something
}

else if (expensiveValue == "second possibility") {  // second reference to expensive result
  // do something else
}
```

Remove Constant Variables and Expressions from Loops

As a performance best practice, Guidewire recommends that you do not include unnecessary variables, especially ones that hold objects and entity instances, within loops.

The following sample Gosu code suffers a performance problem. The loop includes an object access expression, `period.Active`, which remains unchanged throughout all iterations of the loop.

```
var period : PolicyPeriod

for (x in 5) {
if (x == 3 and period.Active) {  // Evaulate a constant expression redunantly within a loop.
    print("x == 3 on active period")
  }

}
```

In the preceding example, Gosu evaluates the expression `period.Active` at least twice unnecessarily. The following modified sample code improves performance.

```
var period : PolicyPeriod

if (period.Active) {            // Evaluate a constant expression only once before a loop.
   for (x in 5) {
     if (x == 3) {
       print("x == 3 on active period")
     }
  }
}
```

Avoid Doubly Nested Loop Constructs

Nesting a loop construct inside another often produces inefficient code or code that does not produce correct results. As a performance best practice, Guidewire recommends that you avoid any doubly nested loop constructs.

The following sample Gosu code attempts to find duplicate values in an array by using a for loop nested inside another. The code is inefficient because it loops through the array five times, once for each member of the array. It produces inappropriate results because it reports the duplicate value twice.

```
var array = new int[]{1, 2, 3, 4, 3}  // An array with a duplicated value

for (y in array index m) {            // Loop through the array

  for (z in array index n) {          // Nested loop through the array or each
                                      // member in the outer loop

    if (m != n and y == z) {          // If current members in outer and inner
                                      // loops differ and member values are equal
      print("duplicate value: " + z)    // Print a duplicate value in the array

    }
  }
}
```

The preceding sample code produces the following output.

```
duplicate value: 3
duplicate value: 3
```

The following sample Gosu code is a better solution. The code is more efficient because it loops through the array once only. It produces appropriate results because it reports the duplicate value once only.

```
var array = new int[]{1, 2, 3, 4, 3}   // An array with a duplicated value
var hashSet = new java.util.HashSet()  // Declare an empty hash set, which prohibits
                                       // duplicate values

for (y in array)                       // Loop through the array
  if (!hashSet.add(y)) {               // If array value cannot be added
                                       // to the hash set
    print("duplicate value: " + y)       // Print a duplicate value in the array

  }
```

The preceding sample code produces the following output.

```
duplicate value: 3
```

Pull Up Multiple Performance Intensive Method Calls

As a performance best practice, Guidewire recommends a technique called *pulling up*. With the pulling up technique, you examine your existing code to uncover performance intensive method calls that occur in multiple, lower-level methods. If you identify such a method call, pull it up into the higher level-method, so you call it only once. Cache the result in a local variable. Then, call the lower-level methods, and pass the result that you cached down to the lower-level methods as a context variable.

The following sample Gosu code suffers a performance problem. It pushes an expensive method call down to the lower-level routines, so the code repeats the expensive call three times.

```
function computeSomething() {  // Performance suffers with an expensive call pushed down.

  computeA()
  computeB()
  computeC()
}

function computeA() {
  var expensiveResult = expensiveCall()  // Make the expensive call once.
  //do A stuff on expensiveResult
}

function computeB() {
  var expensiveResult = expensiveCall()  // Make the same expensive call twice.
  //do B stuff on expensiveResult
}

function computeC() {
  var expensiveResult = expensiveCall()  // Make the same expensive call three times.
  //do C stuff on expensiveResult
}
```

The following modified sample code improves performance. It pulls the expensive method call up to the main routine, which calls it once. Then, it passes the cached result down to the lower-level routines, as a context variable.

```
function computeSomething() {  // Performance improves with an expensive call pulled up.

  var expensiveResult = expensiveCall()  // Make the expensive call only once.

  computeA(expensiveResult)
  computeB(expensiveResult)
  computeC(expensiveResult)
}

function computeA(expensiveResult : ExpensiveResult) {  // Use the pushed down result.
  //do A stuff on expensiveResult
}

function computeB(expensiveResult : ExpensiveResult) {  // Use the pushed down result.
  //do B stuff on expensiveResult
}

function computeC(expensiveResult : ExpensiveResult) {  // Use the pushed down result.
  //do C stuff on expensiveResult
}
```

Be Wary of Dot Notation with Object Access Paths

As a performance best practice, Guidewire recommends that you be aware of the performance impact of dot notation to access instance arrays on object access paths. You can write an object access path quickly, but your code at runtime can run extremely slowly.

The following sample Gosu code suffers a performance problem. It acquires an array of addresses for all additional interests for all vehicles on the Personal Auto line of business.

```
var personalAutoLine = PersonalAutoLine
var personalAutoAddresses = personalAutoLine.Vehicles.AdditionalInterests.Addresses
```

Most likely this was not what the developer intended. Determine the most efficient means of acquiring just the data that you need. For example, rewrite the preceding example to use a query builder expression that fetches a more focused set of addresses from the application database.

For more general information, see “Query Builder APIs” on page 129 in the *Gosu Reference Guide*.

Avoid Code that Incidentally Queries the Database

As a performance best practice, Guidewire recommends that you avoid object property access or method calls that potentially query the relational database.

Accessing Entity Arrays Does Not Incidentally Query the Database

The following sample Gosu code accesses an array of `Vehicle` instances as entity array.

```
policyLine.Vehicles  // Accessing an entity array does not query the database.
```

Accessing the entity array does not incidentally query the relational database. The application database caches them whenever it loads a parent entity from the relational database.

Accessing Finder Arrays Incidentally Queries the Database

The following sample Gosu code accesses an array of `Policy` entities by using a `Finder` method on a `Policy` instance.

```
policy.Finder.findLocalPoliciesForAccount(account)  // Accessing a finder array queries the database.
```

Calling a `Finder` method does incidentally query the relational database. However, the application database does not cache finder arrays. Only your code keeps the array in memory.

To avoid repeated, redundant calls that incidentally query the database, Guidewire recommends as a best practice that you cache the results once in a local array variable. Then, pass the local array variable to lower level routines to operate on the same results. This design approach is an example of *pulling up*. For more information, see "Pull Up Multiple Performance Intensive Method Calls" on page 36.

Use Comparison Methods to Filter Queries

The relational database that supports BillingCenter filters the results of a query much more efficiently than your own Gosu code, because it avoids loading unnecessary data into the application server. As a performance best practice, Guidewire recommends that you filter your queries with comparisons methods rather than filter the results with your own code.

The following sample Gosu code suffers a performance problem. It inadvertently loads most of the claims, along with their policies, from the relational database. Then, the code iterates the loaded claims and their policies and process only those few that match a specific policy. In other words, the code loads an excessive amount of data unnecessarily and takes an excessive amount of time to search for a few instances to process.

```
var targetPolicy : Policy
var claimQuery = Query.make(Claim)  // Performance suffers because the query loads all claims.

for (claim in claimQuery.select()) {
  if (claim.Policy == targetPolicy) {                // Local Gosu code filters claims.
    // Do something on claims of targeted policies.
    ...
  }
}
```

The following modified sample Gosu code improves performance. It finds only relevant policies to process with the compare method:

```
var targetPolicy : Policy
var claimQuery = Query.make(Claim)  // Performance improves because the query loads few claims.

claimQuery.compare("policy", Equals, targetPolicy)  // Query comparison method filters claims.

for (claim.Policy in query.select()) {
  // Do something on claims of targeted policies.
}
```

Use Count Properties on Query Builder Results and Find Queries

As a performance best practice, Guidewire recommends that you obtain counts of items fetched from the application database by using the `Count` properties on query builder result objects. The same recommendation applies to find expression query objects. Do *not* iterate result or query objects to obtain a count.

IMPORTANT Guidewire recommends the query builder APIs instead of find expressions to fetch items from the application database whenever possible, especially for new code. For more information, see "Query Builder APIs" on page 129 in the *Gosu Reference Guide*.

Use Count Properties If You Want the Number Found

The following sample Gosu code uses the `Count` property on a query builder API result object.

```
uses gw.api.database.Query

var accountQuery = Query.make(Account) // Find all accounts.
var result = accountQuery.select()
print("Number of accounts: " + result.Count)
```

The following sample Gosu code uses the `Count` property on a find expression query object.

```
var accountQuery = find(p in PolicyPeriod)  // Find all accounts.
print("Number of accounts: " + accountQuery.Count)
```

Use Empty Properties If You Want to Know whether Anything Was Found

If you want to know only whether a result or query object fetched anything from the application database, use the `Empty` property instead of the `Count` property. The value of the `Empty` property returns to your Gosu code faster, because the evaluation stops as soon as it counts at least one fetched item. In contrast, the value of the `Count` property returns to your Gosu only after counting all fetched items.

The following sample Gosu code uses the `Count` property on a query builder API result object.

```
uses gw.api.database.Query

var accountQuery = Query.make(Account) // Find all accounts.
var result = accountQuery.select()

if (result.Empty) {  // Does the result contain anything?
  print ("Nothing found.")
}
else {
  print ("Got some!")
}
```

The following sample Gosu code uses the `Count` property on a find expression query object.

```
var accountQuery = find(a in Account)  // Find all accounts.

if (accountQuery.Empty) {  // Did the query fetch anything?
  print ("Nothing found.")
}
else {
  print ("Got some!")
}
```

Use Activity Pattern Codes Instead of Public IDs in Comparisons

As a performance best practice, Guidewire recommends that you always use activity pattern codes (`Activity.Pattern.Code`) in comparison expressions for conditional processing and in queries. Comparisons of activity patterns by codes frequently avoid database reads to evaluate the expression.

The following sample Gosu code suffers a performance problem. The comparison with `Activity.Pattern` by public ID `cc:12345` always requires a database read to evaluate the expression.

```
if (Activity.ActivityPattern == ActivityPattern("cc:12345")) {  // Comparisons of activity pattern
                                                                // public IDs always require a
}                                                               // database read.
```

The following sample Gosu code improves performance by comparing `Activity.Pattern.Code` with an activity pattern code in the conditional expression.

```
if (Activity.ActivityPattern.Code == "MyActivityPatternCode") {  // Comparisons of activity pattern
  ...                                                            // codes generally avoid a
}                                                                // database read.
```

Never localize activity pattern codes. These codes are intended for use in Gosu expressions, not for display in error messages of the application user interface.

See also

- To learn how to add a column of localized pattern codes to the `ActivityPattern` entity type, see "Localized Columns in Entities" on page 67 in the *Globalization Guide*.

Do Not Instantiate Plugins for Every Execution

As a performance best practice, Guidewire recommends that you do *not* instantiate plugins for every execution of code that needs to call them. Plugins are reusable classes that you can instantiate once within application scope and reuse if needed.

The following sample Gosu code suffers a potential performance problem. It instantiates a plugin within local scope, so the plugin must initialize itself for every execution of the code that calls it.

```
var XPRPlugin = new com.acme.pc.webservices.plugin.VehiclePlugin()  // low performance
```

The following modified sample code improves performance. It instantiates the plugin within application scope, so the plugin must initialize itself only once, regardless of the number of executions of code that call it.

```
static var XPRPlugin = new com.acme.pc.webservices.plugin.VehiclePlugin()  // higher performance
```

The following modified sample code improves performance even more.

```
var XPRPlugin = new com.acme.pc.webservices.plugin.VehiclePlugin()
gw.api.web.Scopes.getApplication().put("XPRPlugin", XPRPlugin)      // highest performance
```

Gosu Best Practices Checklist

Use the following checklist before you complete your Gosu coding tasks to assure your Gosu code follows Guidewire best practices.

Best Practice to Follow	Best Practice Followed
“Observe General Gosu Naming Conventions” on page 24	☐
“Omit Type Specifications with Variable Initialization” on page 24	☐
“Add a Suffix to Functions and Classes to Avoid Name Conflicts” on page 24	☐
“Declare Functions Private Unless Absolutely Necessary” on page 24	☐
“Use Public Properties Instead of Public Variables” on page 24	☐
“Do Not Declare Static Scope for Mutable Variables” on page 25	☐
“Use Extensions to Add Functions to Entities” on page 25	☐
“Gosu Commenting Best Practices” on page 26	☐
“Use Whitespace Effectively” on page 28	☐
“Use Parentheses Effectively” on page 28	☐
“Use Curly Braces Effectively” on page 28	☐
“Program Defensively Against Conditions that Can Fail” on page 28	☐
“Omit Semicolons as Statement Delimiters” on page 30	☐
“Use != Instead of <> as the Inequality Operator” on page 30	☐
“Observe Null Safety with Equality Operators” on page 30	☐
“Use typeis Expressions for Automatic Downcasting” on page 31	☐
“Rewrite Comparison Operators to Avoid the Equals Method” on page 30	☐
“Avoid Implicit Type Coercion with Comparison Operators” on page 31	☐
“Implement Conditional Operators in Loop Conditions Correctly” on page 32	☐
“Interrupt Loop Execution as Early as Possible” on page 32	☐
“Return from Functions as Early as Possible” on page 33	☐
“Match Capitalization of Types, Keywords, and Symbols” on page 25	☐
“Use Query Builder APIs instead of Find Expressions in New Code” on page 33	☐
“Use the Fastest Technique for String Concatenation” on page 34	☐
“Consider the Order of Terms in Compound Expressions” on page 34	☐
“Avoid Repeated Method Calls Within an Algorithm” on page 35	☐
“Remove Constant Variables and Expressions from Loops” on page 35	☐
“Pull Up Multiple Performance Intensive Method Calls” on page 36	☐
“Be Wary of Dot Notation with Object Access Paths” on page 37	☐

Best Practice to Follow	Best Practice Followed
“Avoid Code that Incidentally Queries the Database” on page 37	☐
“Use Comparison Methods to Filter Queries” on page 38	☐
“Use Count Properties on Query Builder Results and Find Queries” on page 38	☐
“Use Activity Pattern Codes Instead of Public IDs in Comparisons” on page 39	☐
“Do Not Instantiate Plugins for Every Execution” on page 39	☐

chapter 5

Upgrade Best Practices

An upgrade of your BillingCenter installation comprises automated and manual procedures. The ways in which your configure your BillingCenter installation help determine the ease or difficulty of the procedures for future upgrades.

This topic includes:

See also

- "Planning Your BillingCenter Upgrade" on page 15 in the *Upgrade Guide*

Upgradability Best Practices

Guidewire recommends a number of best practices to help prepare your BillingCenter installation for future upgrades.

Add Minor Changes Directly to Base Files

Whenever you make only minor changes to a file, make them directly within the base file. If the file changes in a future release, you can accept or reject the changes during the upgrade. Your changes and the changes in the new release are visible side-by-side within your three-way merge tool while you merge the upgrade code manually.

Copy Base Files to Add Major Changes

Whenever you make major changes to a file, make a copy of the file. Name the copy of file the same as the original, with the customer identifier inserted. For example, make a copy of `SomeBaseScreenDV.pcf` and give it the name `SomeBaseScreen_ExtDV.pcf`.

In the original file, add a comment at the top that states you copied the file to make major changes, and include the filename of the copy. For example:

```
<!-- This file was copied to SomeBaseScreen_ExtDV.pcf -->
```

If the file changes in a future release, you will notice that the file was copied. You then can decide whether to replicate the changes in the new file in your copy of the base version of the file. Especially if the change enhances the base file or fixes a defect, you may want to apply the same changes to your copy of the file.

Copy Base Functions to Make Major Changes

Whenever you make major changes to a function, or method, defined in a Gosu class, make a copy of the function and place it in a customer class. Give the copy of the function the same name as the original, with the customer identifier as a suffix. For example, make a copy of `SomeBaseFunction` in a folder within your customer package, such as `com.`*`Customername`*. Name the copied function `SomeBaseFunction_Ext`.

In the original function, add a comment at the top that states you copied the function to make major changes. For example:

```
// -- This function was copied to SomeBaseFunction_Ext --
```

To confirm that you changed all existing code to use your copied function, temporarily rename the original function and then compile your entire project to check for compilation errors. After you remove all calls to the original function, consider commenting out the original function to prevent developers in the future from using it accidentally.

If the function changes in a future release, you will notice that the function was copied. You then can decide whether to replicate the changes in your copy of the base function. Especially if the change enhances the function or fixes a defect, you may want to apply the same changes to your copy of the function.

Switching From Minor to Major Changes

After you make minor changes to a file or function, you might decide to make additional major changes to the same file or function. If you switch from making minor changes to making major changes, switch from the recommendations for minor changes to the recommendations for major changes.

1. Rename the base file or function by including the customer identifier.

   For example, make a copy of `SomeBaseScreenDV.pcf` and give it the name `SomeBaseScreen_ExtDV.pcf`.

2. Restore the original base file from `base.zip` or from your source code repository.

3. Add a comment to the top of the restored base file or function to state that the file or function was copied.

   For example:

   ```
   <!-- This file was copied to SomeBaseScreen_ExtDV.pcf -->
   ```

4. Make your additional major changes to the copy of the file or function.

Upgrade Best Practices Checklist

Use the following checklist before you complete your configuration of the base BillingCenter installation to help ease future upgrades.

Best Practice to Follow	Best Practice Was Followed
“Add Minor Changes Directly to Base Files” on page 43	☐
“Copy Base Files to Add Major Changes” on page 44	☐
“Copy Base Functions to Make Major Changes” on page 44	☐
“Switching From Minor to Major Changes” on page 44	☐