Customizing ReThink

User's Guide

Version 5.1 Rev. 1

Customizing ReThink User's Guide, Version 5.1 Rev. 1
February 2015

The information in this publication is subject to change without notice and does not represent a commitment by Gensym Corporation.

Although this software has been extensively tested, Gensym cannot guarantee error-free performance in all applications. Accordingly, use of the software is at the customer's sole risk.

Gensym Corporation
52 Second Avenue
Burlington, MA 01803 USA
Telephone: (781) 265-7100
Fax: (781) 265-7101

Part Number: DOC076-510

Contents Summary

Contents

Preface

Describes this guide and the conventions that it uses.

About this Guide

This guide describes the internal operations of ReThink so developers can customize its behavior. It describes:

- What you can customize, which includes blocks, instruments, resources, work objects, resource managers, and scenarios.
- How to customize ReThink, which includes subclassing objects, adding attributes, editing subobjects, and customizing procedures.
- How to customize blocks, including block behavior, animation, duration, and paths.
- How to customize instruments, including behavior and animation.
- How to customize resources and work objects, including animation and resource allocation.
- How to customize the user interface, including popup menus, dialogs, and the ReThink toolbar.

This guide also provides a reference to the internal ReThink operations, which consists of:

- Block processing, which determines the order in which events occur.
- Application programmer's interface, which provides procedures that you can call to perform ReThink operations.
- Relations, which ReThink creates as part of block processing.

In addition, it provides a glossary of key terms.

Audience

This guide is written for ReThink developers to teach them how to customize ReThink objects to obtain specific behavior. This guide explains how to customize each type of ReThink object and shows examples of typical customizations. This guide also contains a reference section, which provides reference documentation on the application programmer's interface (API) and related topics.

This guide assumes that you have a basic understanding of how to program in the G2 environment in which ReThink runs. In particular, ReThink developers write custom procedures, which use the API and refer to internal relations. ReThink developers also create object class definitions that contain attributes that are subobject.

If you are a ReThink modeler who wants to build and run models in ReThink, see *Getting Started with ReThink* and the *ReThink User's Guide*.

A Note About the API

The ReThink API, as described in this guide, is not expected to change significantly in future releases, but exceptions may occur. A detailed description of any changes will accompany the ReThink release that includes them.

The techniques by which ReThink implements its capabilities, however, are subject to change at any time without notice or explanation, and are expected to change as the product evolves. These techniques will not be described in any ReThink documentation.

Therefore, it is essential that you use ReThink exclusively through its API, as described in this guide. If you bypass the API, you cannot rely on your code to work in the future, since ReThink may change, or in the present, because the code may not correctly manage the internal operations of ReThink.

Conversely, if you use the ReThink API exclusively, you can rely on Gensym to notify you of any ReThink changes that might affect your code, and you can rely on ReThink to manage all internal operations correctly.

If ReThink does not seem to provide the capabilities that you need, please contact Gensym Customer Support. For details, see Customer Support Services.

Conventions

This tutorial uses the following typographic conventions:

Example	Description
true	Parameter and metric values
task	Glossary terms
c:\Program Files\Gensym\g2-2011\rethink\kbs\rethink.kb	Pathnames and filenames

Related Documentation

ReThink

- *Getting Started with ReThink*
- *ReThink User's Guide*
- *Customizing ReThink User's Guide*

G2 Core Technology

- *G2 Bundle Release Notes*
- *Getting Started with G2 Tutorials*
- *G2 Reference Manual*
- *G2 Language Reference Card*
- *G2 Developer's Guide*
- *G2 System Procedures Reference Manual*
- *G2 System Procedures Reference Card*
- *G2 Class Reference Manual*
- *Telewindows User's Guide*
- *G2 Gateway Bridge Developer's Guide*

G2 Utilities

- *G2 ProTools User's Guide*
- *G2 Foundation Resources User's Guide*
- *G2 Menu System User's Guide*
- *G2 XL Spreadsheet User's Guide*
- *G2 Dynamic Displays User's Guide*
- *G2 Developer's Interface User's Guide*
- *G2 OnLine Documentation Developer's Guide*
- *G2 OnLine Documentation User's Guide*
- *G2 GUIDE User's Guide*
- *G2 GUIDE/UIL Procedures Reference Manual*

G2 Developers' Utilities

- *Business Process Management System Users' Guide*
- *Business Rules Management System User's Guide*
- *G2 Reporting Engine User's Guide*
- *G2 Web User's Guide*
- *G2 Event and Data Processing User's Guide*
- *G2 Run-Time Library User's Guide*
- *G2 Event Manager User's Guide*
- *G2 Dialog Utility User's Guide*
- *G2 Data Source Manager User's Guide*
- *G2 Data Point Manager User's Guide*
- *G2 Engineering Unit Conversion User's Guide*
- *G2 Error Handling Foundation User's Guide*
- *G2 Relation Browser User's Guide*

Bridges and External Systems

- *G2 ActiveXLink User's Guide*
- *G2 CORBALink User's Guide*
- *G2 Database Bridge User's Guide*
- *G2-ODBC Bridge Release Notes*

- *G2-Oracle Bridge Release Notes*
- *G2-Sybase Bridge Release Notes*
- *G2 JMail Bridge User's Guide*
- *G2 Java Socket Manager User's Guide*
- *G2 JMSLink User's Guide*
- *G2 OPCLink User's Guide*
- *G2-PI Bridge User's Guide*
- *G2-SNMP Bridge User's Guide*
- *G2-HLA Bridge User's Guide*
- *G2 WebLink User's Guide*

G2 JavaLink

- *G2 JavaLink User's Guide*
- *G2 DownloadInterfaces User's Guide*
- *G2 Bean Builder User's Guide*

G2 Diagnostic Assistant

- *GDA User's Guide*
- *GDA Reference Manual*
- *GDA API Reference*

Customer Support Services

You can obtain help with this or any Gensym product from Gensym Customer Support. Help is available online, by telephone, by fax, and by email.

To obtain customer support online:

➔ Access G2 HelpLink at `www.gensym-support.com`.

You will be asked to log in to an existing account or create a new account if necessary. G2 HelpLink allows you to:

- Register your question with Customer Support by creating an Issue.
- Query, link to, and review existing issues.
- Share issues with other users in your group.
- Query for Bugs, Suggestions, and Resolutions.

To obtain customer support by telephone, fax, or email:

➔ Use the following numbers and addresses:

	Americas	Europe, Middle-East, Africa (EMEA)
Phone	(781) 265-7301	+31-71-5682622
Fax	(781) 265-7255	+31-71-5682621
Email	service@gensym.com	service-ema@gensym.com

Part I

Introduction

Chapter 1: Introduction to Customization

Provides an overview of what you can customize in ReThink.

Chapter 2: How to Customize ReThink

Provides an overview of how to customize ReThink objects, and how to make your customizations permanent.

Introduction to Customization

Provides an overview of what you can customize in ReThink.

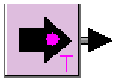
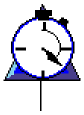
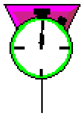
Introduction

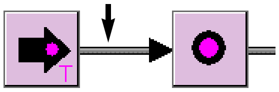
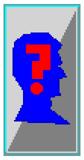
You can tailor the behavior of ReThink objects to reflect the processes of the business you are modeling. For example, you can change the behavior of ReThink blocks and instruments, create new types of resources and work objects, and customize the way ReThink computes cost and duration. These are called **customizations**.

To customize ReThink, you must be a **developer**, which means you need to have a detailed understanding of the structure of ReThink objects, including the customizable subobjects, methods, and procedures. You also need to know something about the internals of ReThink, including how blocks execute and how ReThink establishes relations. Finally, when you customize ReThink procedures, you need to understand how to use the ReThink internal procedures or application programmer's interface (API).

What You Can Customize

You can customize the following ReThink objects, using the techniques described in How to Customize ReThink:

Sample Icon	Class Name	Description
	bpr-block	User-level blocks that process work objects.
	bpr-probe	Instruments that obtain values from the model.
	bpr-feed	Instruments that supply values to the model.
●	bpr-object	Work objects that blocks create and process.
■	bpr-resource	Resources that blocks require to process work.
	bpr-resource-manager	Resource Managers that allocate and deallocate resources for an activity.
☒	bpr-surrogate	Copies of resources that appear in another pool.

Sample Icon	Class Name	Description
	bpr-path	Paths between blocks that carry work objects.
	bpr-scenario	Tools that controls the simulation clock for a model.

Customizing the Behavior of Objects

You can customize the behavior of objects in your model, including:

- The class-specific attributes of the object.
- The default behavior of a block or instrument.
- The way ReThink calculates duration and cost for a block, work object, or resource.
- The way a Resource Manager allocates and deallocates resources.
- The reset behavior of a Scenario tool.

You customize the default behavior of blocks and instruments by editing the default **stop method** for the object. The stop method is a method that a block executes after it computes duration and cost, and that an instrument executes before it probes or feeds a value. For example, you might create a custom instrument that generates an alarm when an attribute exceeds a threshold; you might create a custom Branch block that schedules work in progress before new work; or, you might create a custom Store block that stores objects in a pool to produce a scatter chart.

You can also create a **start method** for a block, which ReThink executes before it computes duration and before it executes the stop method. For example, you might want a block to perform a remote procedure call to an external program before it executes its default behavior.

ReThink provides several default procedures that compute the duration of blocks, including a random normal delay, a random exponential delay, and a duration based on an eight hour a day work week. You can customize the duration procedures for blocks, work objects, and resources. You can also customize the procedure that computes total cost for blocks, work objects, and resources.

You can customize the way Resource Managers allocate and deallocate resources from a pool. For example, you might want to allocate the least expensive available resource for an activity.

You can customize the behavior of a Scenario tool when you reset a model. For example, you might want to start a particular Source block each time you reset.

Finally, you can customize the behavior of a remote, which is the intermediate object that ReThink creates when you create a chart from a probe. To customize the computed values of the remote, you edit attributes in its table.

Required G2 Knowledge

To customize ReThink, you need to know more of the fundamentals of G2 than you do if you are using ReThink “off-the-shelf.” Specifically, you need to know how to:

- Define classes of objects.
- Write and edit methods and procedures.
- Work with G2 relations and lists.
- Create and edit parameters, using subtables.

How to Customize ReThink

Provides an overview of how to customize ReThink objects, and how to make your customizations permanent.

Introduction

The ReThink objects that appear on the palettes are instances of G2 classes, whose definitions you can customize. To customize objects, you must be in Developer mode.

When you customize ReThink objects, the first step is always to create a subclass of an existing ReThink class definition. You place your custom class definitions on a customization workspace, which you save in a separate module of ReThink.

Once you have created a subclass, you can customize any of the following properties of the subclass:

- The attributes that define its characteristics.
- The method that defines the default behavior of the block or instrument.
- The animation, duration, and cost procedures of the subobjects that define the default behaviors.
- The attributes of the subobjects.

Note ReThink implements some default behaviors as methods and other default behaviors as procedures. In a future release, all default behavior will be implemented as methods.

Switching User Modes

To customize ReThink, you must be in **Developer mode**. In Developer mode, the tables for ReThink objects include customization attributes and the menus include customization menu choices. These attributes and menu choices are hidden in Modeler mode.

To switch to Developer mode:

➔ Choose Tools > User Mode > Developer.

When you are in Developer mode, all user interactions are the same as they are in Modeler mode. The primary difference between Modeler mode and Developer mode is the number of menu choices and attributes that are visible. In Modeler mode, you see only the menu choices and attributes necessary for building ReThink models, whereas in Developer mode, you see a number of additional menu choices and attributes necessary for customizing ReThink models.

When you customize ReThink, you will always be in Developer mode. Furthermore, this manual assumes you are in Developer mode, unless otherwise stated.

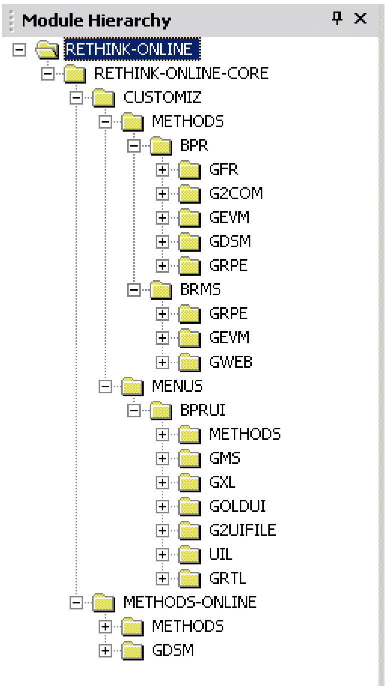
Making Customizations in the Module Hierarchy

When you customize ReThink, you save your customizations in a separate module, which ReThink loads automatically with your application. Because the customizations are located in a separate module, you can share customization modules across ReThink applications.

For more information, see Working with Modules on page 16.

Understanding the Module Hierarchy

To understand where to store your customizations, you need to understand ReThink's module hierarchy:



```
⊟ RETHINK-ONLINE
  ⊟ RETHINK-ONLINE-CORE
    ⊟ CUSTOMIZ
      ⊟ METHODS
        ⊟ BPR
          ⊞ GFR
          ⊞ G2COM
          ⊞ GEVM
          ⊞ GDSM
          ⊞ GRPE
        ⊟ BRMS
          ⊞ GRPE
          ⊞ GEVM
          ⊞ GWEB
      ⊟ MENUS
        ⊟ BPRUI
          ⊞ METHODS
          ⊞ GMS
          ⊞ GXL
          ⊞ GOLDUI
          ⊞ G2UIFILE
          ⊞ UIL
          ⊞ GRTL
    ⊟ METHODS-ONLINE
      ⊞ METHODS
      ⊞ GDSM
```

Module	Description
rethink-online	Default top-level module for ReThink.
rethink-core-online	The directly required module of the top-level rethink-online module, which every new application requires.

Module	Description
customiz	User-level module in which you save customizations to ReThink objects.
methods	Definitions and procedures for ReThink blocks, instruments, and resources.
methods-online	Definitions and procedures for the ReThink blocks found on the Online Activities palette.
menus	Specifies the top-level menu bar, which you can customize.
bpr	Internal, proprietary module containing ReThink's core technology.
brms	Business Rules Management System (BRMS)
gfr	G2 Foundation Resources (GFR).
g2com	G2 ActiveX Link for connectivity to Microsoft Excel, Microsoft Access, and other COM-compliant applications.
gevm	G2 Event Manager.
gdsm	G2 Data Source Manager.
grpe	G2 Reporting Engine
gweb	G2 Web
bprui	ReThink's proprietary user interface module.

Following is a more detailed description of the core ReThink modules.

The rethink-online and rethink-core-online Modules

The rethink-online module is the default, top-level module, which requires the rethink-core-online module. Typically, you create a new project, which creates a new top-level module with your project name, which requires the rethink-core-online module.

If you do not want to share customization across applications, you can save customizations and G2 system tables in the top-level module. For more information, see Customizing System Tables on page 15.

Caution In general, you should not create customizations in the rethink-online module, unless you do not care about sharing your customizations across different ReThink applications.

The customiz Module

The customiz module is where you can create and save customizations to ReThink objects, when you plan to share these customizations across different ReThink applications.

Once you have created your custom class definitions, methods, and procedures on a customization workspace, you assign this workspace to the customiz module, as described in Creating a Customization Workspace on page 14. The next time you load ReThink, the customiz module is automatically loaded.

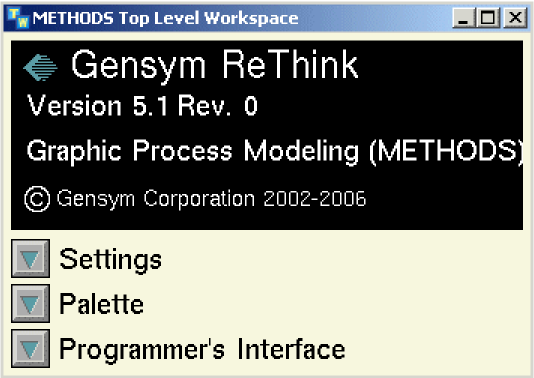
The methods and methods-online Modules

The methods module contains the class definitions for the basic ReThink objects and subobjects, and the methods and procedures that control their behavior. When you customize ReThink, you create subclasses of these class definitions and copy these methods and procedures.

The methods-online module contains definitions, methods, and procedures for the blocks on the Online Activities palette.

To display the methods module workspace:

1 Choose Workspace > Get Workspace and choose the methods-top-level workspace to display this top-level workspace:



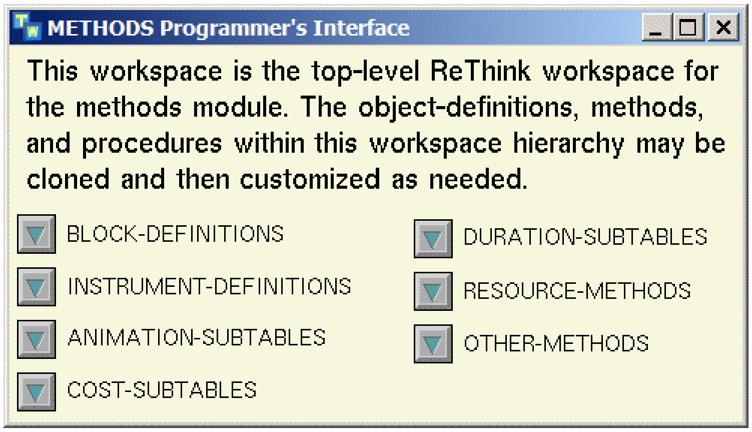
Gensym ReThink

Version 5.1 Rev. 0

Graphic Process Modeling (METHODS)

© Gensym Corporation 2002-2006

2 Click the Programmer's Interface button to display this workspace:



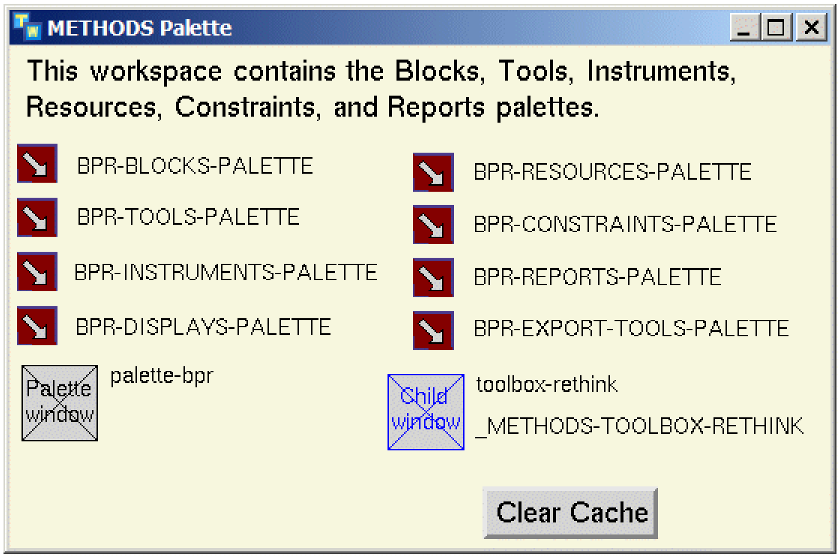
METHODS Programmer's Interface

This workspace is the top-level ReThink workspace for the methods module. The object-definitions, methods, and procedures within this workspace hierarchy may be cloned and then customized as needed.

Each button contains the class definition, methods, procedures, and rules that define the ReThink objects:

This workspace...	**Contains...**
Block Definitions	Class definitions, methods, and procedures that define the default ReThink blocks.
Instruments Definitions	Class definitions, methods, and procedures that define the default feeds and probes.
Animation Subtables, Cost Subtables, and Duration Subtables	Class definitions of subobjects and associated procedures that control the cost and duration computations for blocks, resources, and work objects, and the animation of blocks, resources, work objects, Resource Managers, surrogates, and paths.
Resource Methods	Procedures that define how Resource Managers allocate and deallocate resources.
Other Methods	Methods for adding work objects to the path queue and resetting the scenario.

3 Click the Palettes button to display this workspace:



METHODS Palette

This workspace contains the Blocks, Tools, Instruments, Resources, Constraints, and Reports palettes.

BPR-BLOCKS-PALETTE

BPR-TOOLS-PALETTE

BPR-INSTRUMENTS-PALETTE

BPR-DISPLAYS-PALETTE

BPR-RESOURCES-PALETTE

BPR-CONSTRAINTS-PALETTE

BPR-REPORTS-PALETTE

BPR-EXPORT-TOOLS-PALETTE

Palette window palette-bpr

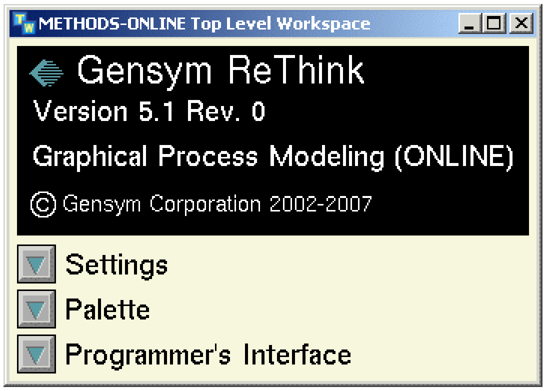
Child window toolbox-rethink _METHODS-TOOLBOX-RETHINK

Clear Cache

This workspace contains all the palettes that appear in the ReThink toolbox except the Online Activities palette, which is contained in the methods-online module.

To display the methods-online module workspace:

1 Choose Workspace > Get Workspace and choose the methods-online-top-level workspace to display this top-level workspace:



METHODS-ONLINE Top Level Workspace

Gensym ReThink

Version 5.1 Rev. 0

Graphical Process Modeling (ONLINE)

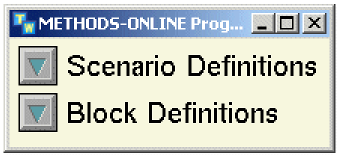
© Gensym Corporation 2002-2007

Settings

Palette

Programmer's Interface

2 Click the Programmer’s Interface button to display this workspace:

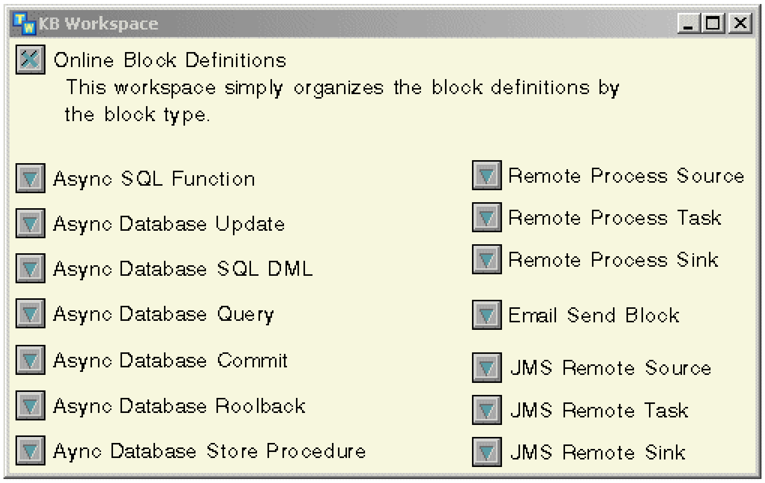


METHODS-ONLINE Prog...

Scenario Definitions

Block Definitions

Here are the Online Block Definitions and Scenario Definitions workspaces:

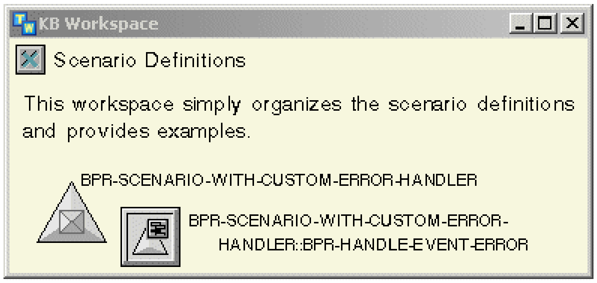


KB Workspace

Online Block Definitions

This workspace simply organizes the block definitions by the block type.

- Async SQL Function
- Async Database Update
- Async Database SQL DML
- Async Database Query
- Async Database Commit
- Async Database Roolback
- Aync Database Store Procedure
- Remote Process Source
- Remote Process Task
- Remote Process Sink
- Email Send Block
- JMS Remote Source
- JMS Remote Task
- JMS Remote Sink



KB Workspace

Scenario Definitions

This workspace simply organizes the scenario definitions and provides examples.

BPR-SCENARIO-WITH-CUSTOM-ERROR-HANDLER
BPR-SCENARIO-WITH-CUSTOM-ERROR-HANDLER::BPR-HANDLE-EVENT-ERROR

The bpr Modules

The bpr module is the proprietary core of ReThink. It contains the discrete event simulation engine and other internal mechanisms that are fundamental to the ReThink environment. It also contains the default layout of the ReThink menus.

The statements in this core part of ReThink are text-stripped so that you cannot access or delete any of the items.

Creating a Customization Workspace

When you customize ReThink, you typically create a separate workspace on which you place your custom class definitions, methods, and procedures. You assign this workspace to the customiz module and save it in the *customiz.kb* file.

When you load ReThink, the customizations module is automatically loaded, and the customizations automatically take effect.

When you install a new version of ReThink, you replace the default customiz module with your customiz module to maintain the customizations. In addition, you can share your customizations across multiple ReThink applications by using this module in another application.

Note If you do not care about sharing your customizations across multiple applications, you can create your customizations in the top-level module. However, we recommend always saving your customizations in the customiz modules to make it easy to share customizations in the future.

To create a customizations workspace:

1 Choose Workspace > New to create a new named workspace.

You create your customizations on this workspace.

2 Choose Table from the popup menu for the workspace.

3 Edit the module-assignment attribute to be the customiz module.

The workspace is now assigned to the customiz module, which means it will be saved in the *customiz.kb* file.

4 Choose File > KB Modules > Save and save the customiz module in the *customiz.kb* file.

You can also save the rethink-online module, using the Including All Required Modules option, which saves all the ReThink modules.

After you save your customizations, open the top-level module of your current ReThink application to load the customiz module automatically.

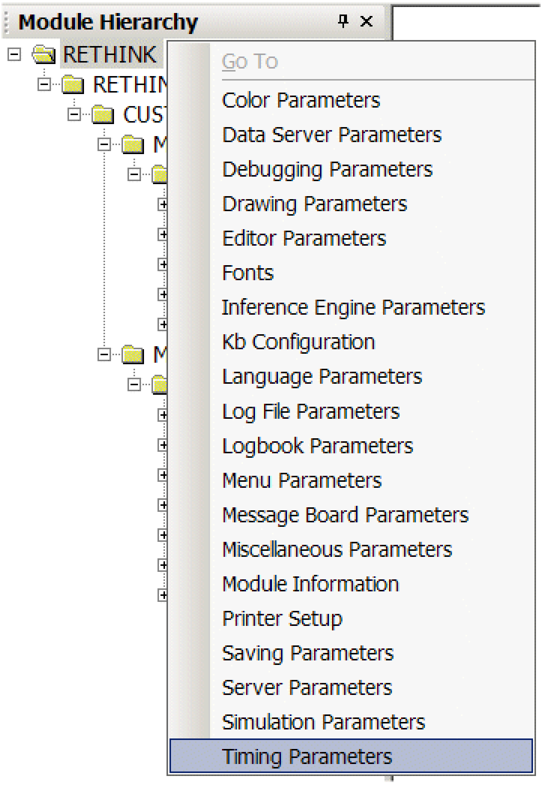
Customizing System Tables

G2 provides a number of system tables, which you can customize to suit your needs. For example, you can edit the default fonts and colors that G2 uses by editing the Fonts system table and the Color Parameters system table. G2 uses the system table definitions in the top-level module; therefore, you should customize system tables in the top-level module.

To customize the system tables for a ReThink application:

➔ Display the module hierarchy, then select the top-level module and choose any system table in the module.

For example, here is how you would edit the Timing Parameters system table for the top-level module:



For more information on editing system tables, see the chapter “System Tables” in the *G2 Reference Manual*.

Working with Modules

You might need to merge a module into an existing model and save it as part of your model. You might also want to rename the top-level module to reflect the name of your application, create a new module and include it in your model, or delete a module you have created or merged that you do not want to save with the current model.

You work with modules as a way of organizing your application, for example, when:

- Multiple individuals are building different parts of the same model.
- You create objects that you will share between multiple models.
- You customize the default menus that ReThink provides to add custom menu choices and custom palettes.

When you are working with modules, the list of available modules reflects the current user mode. To see all ReThink modules, you must be in Developer mode.

Merging a module into an existing model loads the merged module and any directly required modules below it in the module hierarchy.

This topic describes how to:

- Showing the module hierarchy.
- Understand the rules of consistent modularization.
- Check for consistent modularization.
- Merge existing modules into the current model.
- Rename modules.
- Create new modules.
- Create top-level workspaces assigned to modules.
- Save individual modules.
- Delete modules.
- Assign top-level workspaces to different modules.
- Merge multiple modules into a single model.
- Save ReThink definitions in the appropriate module.

Showing the Module Hierarchy

To begin working with modules, you should become familiar with the existing module hierarchy. When showing the module hierarchy, you can see which modules require which other modules in a tree view. You can also see the contents of each module, including all the top-level workspaces assigned to the module and all the system tables associated with the module.

To show the module hierarchy:

➔ Choose View > Module Hierarchy.

By default, the top-level module is rethink or rethink-online, depending on your license. When you create a new project, the top-level module is the same as the project name. The directly required module of the top-level module is rethink-online-core, which, in turn, requires the customiz module.

For a description of the modules in the hierarchy, see Making Customizations in the Module Hierarchy on page 9.

Rules of Consistent Modularization

When working with modules, it is important to understand two important rules for consistent modularization:

- Top-level workspaces must be assigned to modules that are part of the module hierarchy.
- Any work object, block, or resource must exist in either the same module as its class definition or in a module that is above it in the module hierarchy.

If a top-level workspace is assigned to a module that is not part of the module hierarchy, when you attempt to save the model, you will receive an error that states that the modules are inconsistently modularized. Similarly, if an instance of a class is in a module below its definition in the hierarchy, you will be unable to save the model as a modularized KB.

Checking for Consistent Modularization

When working with modules, you might encounter a situation in which your model is not consistently modularized when you attempt to save it. This can occur when:

- You delete a module without deleting its associated workspaces.
- You merge a module that has conflicts without automatically resolving those conflicts.
- Modules exist but are not required by the KB.
- Class definitions exist in a module, but instances of those classes are located in a directly required module.

To avoid module inconsistencies:

- When you delete a module, be sure to delete all its associated workspaces or reassign the workspaces to a new module.
- When merging modules, be sure to resolve all conflicts or assign items to a new module.
- When merging modules or creating new modules, be sure that the modules are required by the KB.
- When creating class definitions and instances, be sure the definitions are located in the same module as the instances or in a directly required (lower level) module, not vice versa.

When module inconsistencies occur and you attempt to save a model, ReThink displays a message at the bottom of the progress dialog indicating that the KB is not consistently modularized.

To find module inconsistencies:

➔ Choose Tools > Inspect and enter this command:

```
check for consistent modularization
```

ReThink displays a message indicating why the model is inconsistently modularized. If a module contains instances or class definitions with conflicts, ReThink displays those instances and definitions in the dialog. You can choose Go To and Properties on any item in the list.

Merging Modules

When you merge a module into your ReThink model, ReThink loads the specified module and any directly required modules that are not already open. ReThink automatically adds the merged module to the module hierarchy as a directly required module of the top-level module.

When saving the merged module with the model, you must click the Including Required Modules option on; otherwise, the merged modules will not be saved as part of the model.

When merging modules, you must consider how ReThink should handle conflicts if they arise. For example, if the module you are merging contains class definitions with the same name as those in the existing model, you typically want the definitions in the existing model to take precedence. By default, ReThink automatically resolves conflicts when merging modules by using definitions in the existing model.

When merging a module that was developed in an earlier version of ReThink, you might want the merged module to use the formats and system tables of the current model and ReThink version, thereby bringing the module up to date with the current version. By default, ReThink does not bring formats up to date or install system tables when merging modules.

To merge a module into an existing model:

1 Choose File > KB Modules > Merge.

2 Navigate to the file to merge.

3 Configure the options, as follows:

- **Resolve Conflicts Automatically**, which has this behavior when conflicts exist:
  - When selected, the model replaces definitions of the same name in the merged module with existing definitions. To ensure that all definitions are consistent, we recommend that you select this option whenever you merge a module. This option is the default.
  - When not selected, the model renames conflicting definitions in the merged module to include the module name, thereby retaining the merged module's definitions. If you believe the merged module contains newer definitions than the existing model, use this option, then integrate the merged module's definitions into the existing model's definitions and delete the merged definitions.
- **Bring Formats Up To Date**, which, when selected, replaces system-defined formats in the merged KB, such as the width of text boxes, with those defined in the current KB. We recommend that you not select this option when merging models developed in an earlier version of ReThink, unless you want to use the older formats. This option is not selected, by default.
- **Install System Tables of Merged KB**, which, when selected, replaces existing system table settings with the system tables in the merged KB. In general, we do not recommend that you select this option when merging models developed in earlier versions. This option is not selected, by default.

4 Click OK to merge the module into the current KB.

The module hierarchy now includes the merged module below the top-level module in the hierarchy.

Renaming Modules

If you create an application without first creating a project, you might need to rename the top-level module to reflect the name of your application. Each new workspace that you create is automatically assigned to the top-level module. Any existing workspaces are also automatically reassigned to the renamed module.

For information on configuring the Module Assignment of a workspace, see Assigning Top-Level Workspaces to Different Modules on page 24.

Caution You should only rename the top-level module or user-defined modules. Do not rename any of the built-in modules; otherwise, your model will no longer run.

To rename a module:

1 Choose File > KB Modules > Rename.

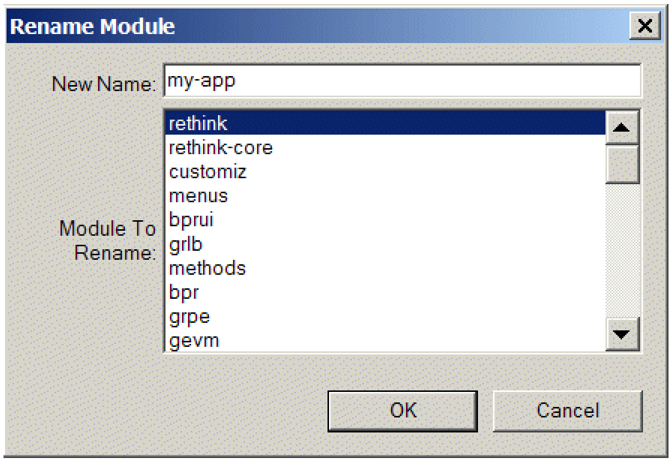
ReThink show an alphabetical listing of all modules.

2 Select the module to rename, for example, the top-level module named rethink-online.

3 Enter the new name for the module in the New Name field and click OK.

The module name must be a symbol.

For example, this dialog shows how to rename the rethink module to my-app:



Rename Module

New Name: my-app

Module To Rename: rethink

Creating Modules

When multiple individuals are building different parts of the same model, it is often useful to break the model up into separate modules. That way, individual modelers can build and save their modules separately. One way to do this is to create new modules below the top-level module, which each modeler works on separately. For example, you might create an as-is module and a vision module, and use the top-level module as a way of loading both lower-level modules.

ReThink automatically adds new modules to the module hierarchy as a directly required module of the top-level module. When you show the module hierarchy, the new module appears below the top-level module in the hierarchy.

When you create a new module, you typically edit the Directly Required Modules of the module, that is, the module that appears *below* the new module in the hierarchy. Typically, you make new modules directly require the rethink-core module so they have access to all built-in definitions.

You should always create a top-level workspace for newly created modules.

When you create a new module, you must save the model with the Including Required Modules option checked; otherwise, the new modules will not be saved as part of the model.

To create a new module:

1 Choose File > KB Modules > New.

2 Enter a name for the new module in the Module Name field.

The module name must be a symbol.

ReThink adds the new module to the module hierarchy just below the top-level module in the hierarchy.

Creating Top-Level Workspaces for Modules

Whenever you create a new module, you should create a top-level workspace associated with the module. You create your model on this top-level workspace.

To create a top-level workspace assigned to a new module:

1 Choose Workspace > New Workspace.

2 Display the properties dialog for the workspace and configure the Names attribute to be a unique name.

ReThink creates a new workspace that is assigned to that module.

Saving Individual Modules

When you merge a module or create a new module and add it to the module hierarchy, you must save the top-level module, including all required modules.

If you have made changes to an individual module that is lower down in the hierarchy, you can save just that module, rather than saving the entire model, including required modules. For example, when customizing ReThink, you might make changes to the customiz module, which you can save as an individual module.

To save an individual module:

1 Choose File > KB Modules > Save.

ReThink displays the Save Module dialog.

2 Navigate to the file you want to save.

For example, if you are saving the individual module named customiz, you would navigate to the *customiz.kb* file in the *rethink\kbs* directory.

3 Choose the module to save from the Module dropdown list.

By default, saving an individual module shows a progress dialog, saves the layout of workspaces, and does not save directly required modules.

4 Configure the saving options, as needed:

- **Including all required modules**, which saves the specified module and all modules below it in the module hierarchy. In general, you should only save the specified module.
- **Save all modules to one file**, which saves the entire project to a single file, which is no longer modularized. In general, you should not save all modules to a single file, unless it cannot be modularized.

5 Click the Save button to save the specified module.

Deleting Modules

You might have merged a module into your model or created a new module that you do not actually want to save with your model. Before you save your model, you must delete these modules.

Caution You should only delete merged modules or user-defined modules. Do no delete ReThink required modules of the top-level module; otherwise, your model will not work correctly.

When you delete a module, ReThink automatically removes all references to the module in the module hierarchy. By default, ReThink deletes all associated workspaces. When deleting modules, we recommend that you use this default.

You can also delete a module without deleting associated workspaces. However, if you use this option, before you save your model, you must ensure that the workspaces have an associated module. To do this, you must manually edit the Module Assignments of each workspace, which is only available in Developer mode.

To delete a module:

1 Choose File > KB Modules > Delete.

2 Select the module to delete.

By default, deleting the selected module deletes all workspaces associated with the module and removes references to it in the directly-required-modules from other modules in the hierarchy.

3 To delete the module without deleting its associated workspaces, disable the Delete Associated Workspaces option.

4 To delete the module without removing references to it in the module hierarchy, disable the Remove References to Module in Hierarchy option.

5 Click OK to delete the module.

The module hierarchy no longer includes the deleted module. If the module hierarchy is currently showing, you must refresh the view to see the updated module hierarchy.

Assigning Top-Level Workspaces to Different Modules

Once you create a new module, you can explicitly assign top-level workspaces to the module to save the workspaces in that module. For example, you can replace the default details of two different Model tools with top-level workspaces, then assign those workspaces to different modules.

You can only assign top-level workspaces to a particular module; details are automatically assigned to the module associated with their superior objects. Thus, if you have already created a model or organizer detail and you want to assign this workspace to a user-defined module, you must follow these steps.

Tip We recommend that you edit the background color of each top-level workspace in a model to distinguish between workspaces assigned to different modules. That way, when you make changes, you will remember to save both modules.

To assign a top-level workspace to a module and use it as a model detail:

1 Create a new module for each detail you want to assign to its own module.

For example, you might create two new modules named as-is and vision, which are directly required modules of the top-level module, whose name might be aero.

For details, see Creating Modules on page 21.

2 Create a new top-level workspace for each module, whose name corresponds to the name of the model or organizer whose detail you want to assign to its own module.

For example, you would create two new top-level workspaces named as-is-model-detail and vision-model-detail.

3 Assign each top-level workspace to its own module:

a Display the table for the workspace that you want to assign to a new module.

b Configure the module-assignment to refer to the new module.

For example, you would assign the as-is-model-detail workspace to the as-is module, and the vision-model-detail workspace to the vision model.

4 If you have already created detail for the model, transfer the contents of the existing detail to the new top-level workspace.

For example, if you had already created a model named as-is, you would transfer the contents of the existing model detail to the workspace named as-is-model-detail, which is assigned to the as-is module.

5 Delete the existing model detail.

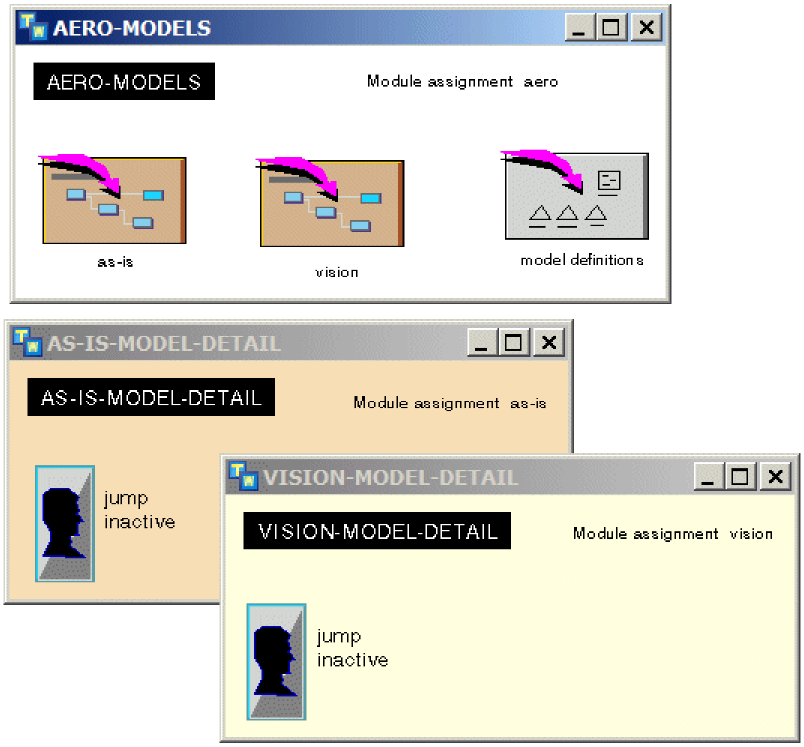
6 Display the properties dialog for the model, click the Customize tab, and configure the Name to be a unique name.

For example, as-is-model and vision-model.

7 Choose the Choose Detail menu choice on the existing Model tool, then choose Select on a top-level workspace that has been assigned to its own module.

For example, you would assign the as-is-model-detail workspace as the detail of the as-is-model model.

The following figure shows the top-level workspace of the Aero model, and the details of the As-Is and Vision models. The module assignments of each workspace appear in the upper-right corner of each workspace. The model details are top-level workspaces whose name boxes have been hidden. The background colors have been changed to indicate that the workspaces are assigned to different modules.



Notice that the Model Definitions organizer is assigned to the top-level aero module because it contains definitions that both the As-Is and Vision models share. However, this might not always be the case as the following section explains.

Merging Multiple Models into a Single Model

Another common technique for creating a model is to build separate models, which you later merge together to create a single model. This technique works well when multiple individuals are working on the same model.

To merge multiple models into a single model:

1 Create a new project.

2 Rename the top-level module to reflect the name of your combined model.

For details, see Renaming Modules on page 20.

3 Merge into the empty model each individual module that you want to include in the combined model.

For details, see Merging Modules on page 19.

The merged modules are automatically required modules of the top-level module.

4 Create a new workspace, assign it to the customiz module, and transfer all common definitions to this workspace.

For details, see Assigning Top-Level Workspaces to Different Modules on page 24.

5 Save the top-level module of the new model with its required modules.

For details, see Saving Individual Modules on page 22.

Saving ReThink Definitions in the Appropriate Module

When you create class definitions for work objects, blocks, instruments, or resources, you place them on one of several different workspaces, and you save them in one of several different modules, depending on how you plan to use them:

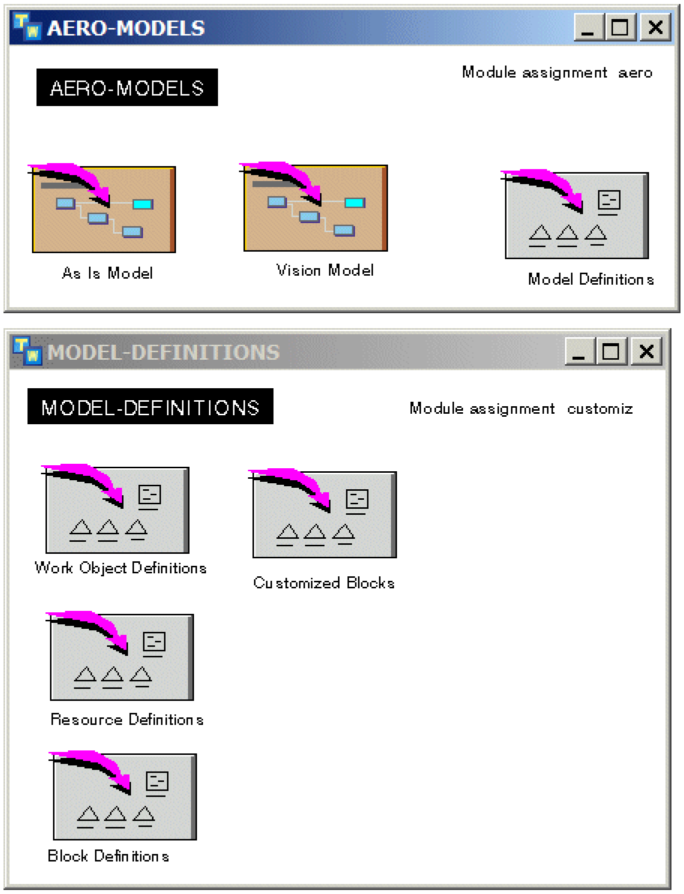
If you use the definitions in...	Then you place them on this workspace...	Which you assign to this module...
All modules of a single model	The top-level workspace of the model	The top-level module.
Individual modules of a single model	A detail of the top-level model	A user-defined module that is directly required by the top-level module, or the top-level module if you have not created user-defined modules.
Multiple models	A unique top-level workspace that is separate from the top-level model workspace	The customiz module.

Thus, if your model consists of a single top-level module with no user-defined modules, and if you do not plan to share definitions among multiple models, then you can save definitions in the top-level module. If, on the other hand, you create a user-defined module and have definitions that are unique to a particular module, you typically save those definitions in the user-defined module.

However, if you create definitions that you plan to share between multiple models, then you should save them in the customiz module.

To save definitions in the customiz module, create a new top-level workspace, assign it to the customiz module, then save the customiz module.

For example, suppose you wanted to share the definitions in the top-level Model Definitions organizer with other models. To do this, replace the default detail of the Model Definitions organizer with a new top-level workspace, then assign the workspace to the customiz module, as the following figure shows:



Module assignment aero

Module assignment customiz

To share these definitions between different models, replace the default customiz module with the customiz module that contains shared definitions, then open the model.

Working with Snapshot Files

You can save the contents of a running model in a snapshot file at any time during the simulation. When you save a model in a snapshot file, ReThink pauses all running models, saves the entire model in its current running state, then starts all the models running again.

The snapshot file is a single, unmodularized KB file with a *.snp* extension.

The advantage to saving a snapshot is that when you load the snapshot, the simulation can continue running at exactly the point at which you saved it.

You load a snapshot file to continue running a model from the point at which you saved the snapshot. This process is known as "warmbooting." A snapshot file is a single, unmodularized KB file with a *.snp* extension.

Warmbooting from a snapshot file replaces the existing KB with the snapshot, which includes the complete application, and automatically starts the server. The model continues running from exactly the point at which it was saved.

By default, when warmbooting from a snapshot file, ReThink uses catch-up mode, which means it sets the internal current time to the current time saved in the snapshot file. For simulations that run continuously, you should always warmboot from a snapshot file with catch-up mode enabled.

Saving a Model in a Snapshot File

To save a running model in a snapshot file:

1 While a model is running, choose File > Save Snapshot.

2 Enter the name of the snapshot file to save and click Save.

Note The filename of the snapshot file must end in the *.snp* file extension.

Warmbooting from a Snapshot

To warmboot from a snapshot file:

1 Choose File > Warmboot from Snapshot.

2 Navigate to the desired snapshot file and click Open.

3 Click Yes in the confirmation dialog.

Common Customization Features

This section provides summary information on the common customization methods, attributes, procedures, and subobjects available for each type of ReThink object. Depending on the type of object, you can customize a variety of features, including:

- Start and stop methods.
- Reset, delete, and update procedures.
- Duration, cost, and animation procedures.
- Duration, cost, and animation subobjects.

To customize the start and stop methods, you define new start and stop methods for your custom objects, based on the default methods. For details, see Customizing the Stop Method on page 43 and Customizing the Start Method on page 46.

To customize reset, delete, and update procedures, and duration, cost, and animation procedures, you define new procedures, based on the default procedures, then you edit the procedure name through the table. For details, see Customizing Animation on page 68.

To customize the duration, cost, and animation subobjects, you create new subobjects, based on the default subobjects, then you configure your custom class definition to use the custom subobject.

In addition to the common customization features, many objects provide specific customization attributes, such as the Branch Procedure Name of a Branch block.

The following tables describe the customization attributes that are common to all objects of a particular type. The tables refer to the internal G2 attribute names, which you need when referring to them in procedures. The workspaces on which the default methods, procedures, and subobjects appear are subworkspaces of the Methods workspace.

Customizing Blocks

You can customize these features of blocks:

- Start and stop methods.
- Reset, delete, and update procedure names.
- Animation, duration, and cost procedure names.
- Animation, duration, and cost subobjects.

The default start and stop methods are located on the individual block workspaces on the Block Definitions workspace. The default procedures are located on the Block Animation Subtable, Block Cost Subtable, and Block Duration Subtable workspaces.

Bpr-block Methods and Customization Attributes

Method/Attribute	Default Value
bpr-stop-method	N/A
bpr-start-method	N/A
names	none
reset-procedure-name	bpr-reset-block
delete-procedure-name	bpr-delete-block
update-procedure-name	bpr-update-block
needs-all-inputs	false

Bpr-block Subobjects

Subobject	Default Value	Attribute	Default Value
animation-subtable	bpr-block-animation-subtable		
		reset-procedure-name	bpr-animate-block
		procedure-name	bpr-animate-block
		active-color	aquamarine
		inactive-color	thistle
		error-color	yellow
		detail-color	salmon
duration-subtable	bpr-default-block-duration-subtable		
		reset-procedure-name	bpr-block-duration-subtable-reset
		procedure-name	bpr-random-normal-duration

Bpr-block Subobjects

Subobject	Default Value	Attribute	Default Value
cost-subtable	bpr-block-cost-subtable		
		reset-procedure-name	bpr-reset-cost-subtable
		procedure-name	bpr-block-cost

Customizing Paths

You can customize these features of paths:

- Reset, delete, and update procedure names.
- Queue procedure name.
- Animation procedure name.
- Animation subobject.

The default procedures and subobjects are located on the Path Animation Subtable and Other Methods workspaces.

Bpr-path Customization Attributes

Method/Attribute	Default Value
names	none
maximum-waiting	none
reset-procedure-name	bpr-reset-path
delete-procedure-name	bpr-delete-path
update-procedure-name	bpr-update-path
enqueue-procedure-name	none

Bpr-path Subobjects

Subobject	Default Value	Attribute	Default Value
animation-subtable	bpr-path-animation-subtable		
		procedure-name	bpr-animate-path
		waiting-color	green
		empty-color	grey
		error-color	yellow
		selected-color	magenta

Customizing Instruments

You can customize these features of instruments:

- Stop methods.
- Reset procedure name.
- Animation procedure name.
- Animation subobject.

The default stop methods are located on the individual instrument workspaces on the Instrument Definitions workspace. The default animation procedures and subobjects are located on the Instrument Animation Subtables workspace.

Bpr-instrument Methods and Customization Attributes

Method/Attribute	Default Value
bpr-stop-method	N/A
names	none
reset-procedure-name	bpr-reset-instrument

Bpr-probe Subobjects

Subobject	Default Value	Attribute	Default Value
animation-subtable	bpr-probe-animation-subtable		
		procedure-name	bpr-animate-instrument
		active-color	sky-blue
		inactive-color	blue
		error-color	yellow

Bpr-feed Subobjects

Subobject	Default Value	Attribute	Default Value
animation-subtable	bpr-feed-animation-subtable		
		procedure-name	bpr-animate-instrument
		active-color	magenta
		inactive-color	violet-red
		error-color	yellow

Customizing Work Objects and Resources

You can customize these features of work objects and resources:

- Start and stop procedure names.
- Reset, delete, and update procedure names.
- Animation, duration, and cost procedure names.
- Duration and cost reset procedure names.
- Animation, duration, and cost subobjects.

The default procedures and subobjects are located on the Object Animation Subtable, Object Duration Subtable, and Object Cost Subtable workspaces.

Bpr-object Methods and Customization Attributes

Method/Attribute	Default Value
names	none
reset-procedure-name	bpr-reset-object
delete-procedure-name	bpr-delete-object
update-procedure-name	bpr-update-object-metrics

Bpr-object Subobjects

Subobject	Default Value	Attribute	Default Value
animation-subtable	bpr-object-animation-subtable		
		procedure-name	bpr-animate-object
		active-color	red
		inactive-color	black
		error-color	yellow
duration-subtable	bpr-object-duration-subtable		
		procedure-name	bpr-object-duration
		reset-procedure-name	bpr-reset-object-duration-subtable
cost-subtable	bpr-object-cost-subtable		
		procedure-name	bpr-object-cost
		reset-procedure-name	bpr-reset-cost-subtable

Customizing Resource Managers

You can customize these features of resource managers:

- Procedure name for choosing resources from a pool.
- Procedure name for sequencing resources when multiple blocks are waiting.
- Procedure name for updating the utilization of the Resource Manager.
- Reset procedure name.
- Animation, duration, and cost procedure names.
- Animation, duration, and cost subobjects.

The default procedures are located on the Resource Methods, Resource Manager Animation Subtable, Resource Manager Duration Subtable, and Resource Manager Cost Subtable workspaces.

Bpr-resource-manager Customization Attributes

Attribute	Default Value
choose-resource-procedure-name	bpr-random-available-resource
sequence-block-procedure-name	bpr-random-waiting-block
update-utilization-procedure-name	none
reset-procedure-name	bpr-reset-resource-manager

Bpr-resource-manager Subobjects

Subobject	Default Value	Attribute	Default Value
animation-subtable	bpr-resource-manager-animation-subtable		
		procedure-name	bpr-animate-resource-manager
		active-color	red
		inactive-color	black
		error-color	yellow

Bpr-resource-manager Subobjects

Subobject	Default Value	Attribute	Default Value
duration-subtable	bpr-resource-manager-duration-subtable		
		reset-procedure-name	bpr-reset-resource-manager-duration-subtable
cost-subtable	bpr-resource-manager-cost-subtable		

Customizing Surrogates

☒ You can customize these features of a surrogate:

- Animation procedure name.
- Animation subobject.

The default animation procedure and subobject are located on the Surrogate Animation Subtable workspace.

Bpr-surrogate Subobjects

Subobject	Default Value	Attribute	Default Value
animation-subtable	bpr-surrogate-animation-subtable		
		procedure-name	bpr-animate-surrogate
		active-color	red
		inactive-color	black
		error-color	yellow

Customizing Scenarios

You can customize the reset procedure name of a scenario. You can find a sample scenario reset procedure on the Other Methods workspace.

Bpr-scenario Customization Attribute

Attribute	Default Value
names	none
reset-procedure-name	none

Creating Subclasses of ReThink Objects

The first step in customizing any ReThink object is to create a **subclass** of an existing ReThink object.

Creating a Subclass

You can create subclasses of ReThink objects in one of two ways, depending on the type of object. In either case, you use a class definition to define the subclass.

For blocks, instruments, and resources, you can create the subclass directly from an existing object. For work objects, you must explicitly create a class definition.

To create a subclass of an existing ReThink block, instrument, or resource:

➔ Create a block, instrument, or resource from the ReThink toolbox and choose Make Subclass.

ReThink automatically creates a class definition, whose superior class is the same as the object you created. ReThink appends a numeric suffix to the name of the class to make it unique, for example, bpr-task-1. You can edit the name of the custom class as desired.

You must use the following technique to create a custom work object; however, you can also use this technique to create a custom block, instrument, or resource.

To create a custom subclass explicitly:

1 Create a Class Definition from the Tools tab of the ReThink toolbar and place it on your customization workspace.

2 Configure the direct-superior-classes to be any available ReThink classes that supports customization.

For details, see Configuring the Superior Class on page 39.

Once you have created the class definition, you create an instance of the class and edit the instance. You can either use the instance directly in a model, or you can place the instance on a custom palette.

To create an instance of the custom class:

➔ Choose create instance on the class definition.

Configuring the Superior Class

Depending on what you are customizing, you can provide any number of built-in ReThink classes as the **superior class** of the custom object:

- To create a custom version of one of the built-in ReThink blocks or instruments, use one of the specific ReThink class names as the superior class. For example, to create a custom version of the Change feed, specify the superior class as bpr-change-feed.
- To create a completely new type of object, use one of the high-level ReThink classes as the superior class. For example, to create a completely new type of block, specify the superior class as bpr-block.

The following tables list all the built-in ReThink classes, organized by object category, upon which you can build custom definitions. The indentation of the class names reflects the class hierarchy of the objects. For example, bpr-source is a subclass of bpr-block.

Block Classes

Class Name	Description
bpr-block	Superior class for all blocks
bpr-source	Source block
bpr-sink	Sink block
bpr-task	Task block
bpr-copy	Copy block
bpr-merge	Merge block
bpr-branch	Branch block
bpr-batch	Batch block
bpr-associate	Associate block
bpr-reconcile	Reconcile block

Block Classes

Class Name	Description
bpr-store	Store block
bpr-retrieve	Retrieve block
bpr-insert	Insert block
bpr-remove	Remove block
bpr-copy-attributes	Copy Attributes block
bpr-yield	Yield block

Instrument Classes

Class Name	Description
bpr-instrument	Superior class for all instruments
bpr-feed	Superior class for all feeds
bpr-timestamp-feed	Timestamp feed
bpr-accumulate-feed	Accumulate feed
bpr-increment-feed	Increment feed
bpr-change-feed	Change feed
bpr-parameter-feed	Parameter feed
bpr-attribute-feed	Attribute feed
bpr-copy-attribute-feed	Copy Attributes feed
bpr-probe	Superior class for all probes
bpr-delta-time-probe	Delta Time probe
bpr-sample-probe	Sample Value probe
bpr-average-probe	Average probe
bpr-moving-average-probe	Moving Average probe
interval-sample-probe	Interval Sample probe

Instrument Classes

Class Name	Description
bpr-parameter-probe	Parameter probe
bpr-copy-attribute-probe	Copy Attributes probe
bpr-statistic-probe	Metric probe
bpr-criteria-probe	Criteria probe
bpr-update-trigger-probe	Update Trigger probe
bpr-n-dim-sample-probe	N-Dimensional Sample probe
bpr-message-probe	Message probe

Work Object and Resource Classes

Class Name	Description
bpr-object	Superior class for all work objects
bpr-resource	Superior class for all resources
person	Person resource
truck	Truck resource
computer	Computer resource
machine	Machine resource
bpr-pool-resource	Pool resource
bpr-container	Container work object that defines a container list attribute that is an item-list
bpr-surrogate	A copy of a resource

Adding Attributes to the Class

You can add class-specific attributes to the definition of any ReThink object to track specific information that is relevant to your business. Typically, you refer to these unique attributes in a custom method or procedure to change the default behavior of the object.

For example, you could associate resources with work objects by adding an attribute to each work object that refers to its associated resource; or, you could create a custom instrument that feeds a random number into the model, where the minimum and maximum values are attributes of the feed that you supply when you configure the instrument.

By default, user-defined attributes appear on the User tab of the properties dialog. You can configure the properties dialogs of any object to include user-defined attributes on custom tabs, as described in Customizing Properties Dialogs.

To add attributes to a class:

➔ Display the properties dialog for the custom class definition and configure the class-specific-attributes.

To display user-defined attributes in dialogs and to prevent run-time errors, specify the type and default values for these attributes, for example:

```
cost-of-order is a quantity, initially is 0.0;
commission is a quantity, initially is 0.0
```

When you run the simulation, you supply a value to the custom attribute through the properties dialog or by using a feed. Alternatively, you might create a custom method or procedure that computes a value for the custom attribute.

For an example of adding attributes to a custom class, see Adding Attributes to a Custom Block.

Customizing the Default Behavior of Blocks or Instruments

You can customize the basic functionality of a block or an instrument by editing the following methods:

- Bpr-stop-method, which specifies the default behavior of a block or instrument. For example, the method that defines the behavior of the Sink block is bpr-sink::bpr-stop-method. This particular method calls a ReThink API named bpr-delete-object, which deletes the objects on each input path.
- Bpr-start-method, which allows you to create an additional method that a block executes before it’s duration; this method is only available for blocks.

ReThink calls the stop method at the end of processing the block or instrument, and it calls the start method at the beginning of block processing, as described in Chapter 8, Block Processing.

ReThink does not call the start and stop methods for a Task block with details; it only calls the start and stop methods for the blocks on the detail subworkspace.

ReThink uses both methods and procedures to customize the behavior of objects. For example, you customize the basic behavior of blocks and instruments with methods, and you customize the additional aspects of the behavior of particular blocks with procedures. You cannot customize the default behavior of resources and work objects other than to customize the duration, animation, and cost subobjects, as described in Editing Subobjects on page 56.

Customizing the Stop Method

To customize the stop method of a block or instrument, create a subclass and edit the default stop method for the class, which is called bpr-stop-method.

Note Instruments execute the stop method either before or after the attached block applies its duration to the simulation, according to the instrument's Phase attribute. For more information, see the *ReThink User's Guide*.

To customize the stop method for a block or instrument:

1 Create a subclass of the block or instrument you want to customize.

2 Display the Block Definitions or Instrument Definitions workspace from the Methods workspace.

3 Click the button associated with the class whose stop method you want to customize.

ReThink displays the class definitions and associated stop methods that define the behavior of all the blocks and instruments.

4 Copy the default bpr-stop-method and place it on your customizations workspace.

5 Choose edit on the stop method and configure its text, as follows:

a Edit the first argument to the stop method to refer to the custom subclass you just created.

Editing the method's first argument changes the qualified name of the method so that it corresponds to the custom subclass.

b Delete the local name initializations and the body of the default method.

c Add a call next method statement to the procedure in the desired location.

To cause the block or instrument to perform the custom portion of the method before it performs the default behavior, place the call next method statement at the end of the method. Otherwise, place the call next method statement at the beginning of the method.

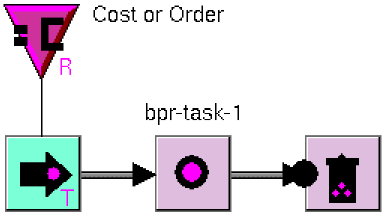
d Specify the custom portion of the procedure in the desired location.

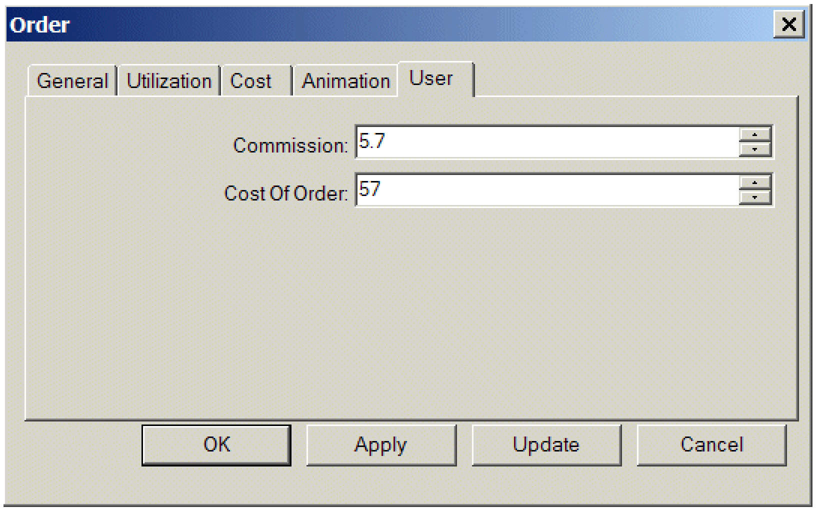
6 Create an instance of your custom class.

The custom class now obtains its default behavior from the custom stop method.

For example, suppose you wanted to compute the commission for a sale, based on the purchase price of an order. To do this, first create a subclass of a work object with two user-defined attributes: cost-of-order and commission. Next, create a subclass of a Task block and edit the default stop method to compute the commission, based on the order cost.

This simple model uses a custom Task block to compute the commission of an order, whose cost is a random number. The User Tab of the properties dialog for the order on the output path of the custom task shows the Cost of Order as 57 and the Commission as 5.7, which is 10% of the order price.

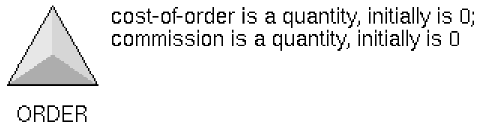




Commission: 5.7

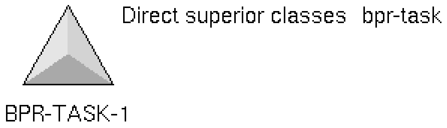
Cost Of Order: 57

Here is the class definition of the order class, which defines the two class-specific attributes:

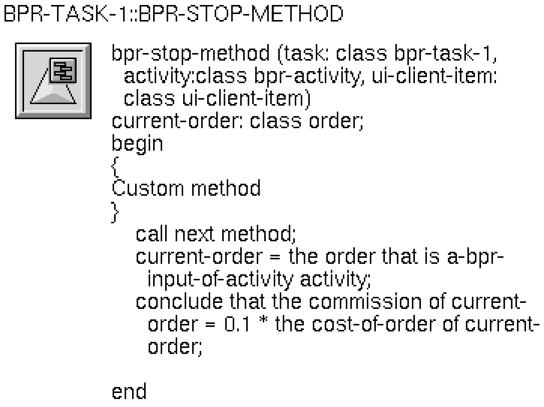


```
cost-of-order is a quantity, initially is 0;
commission is a quantity, initially is 0
```

Here is the custom stop method for the custom Task block and its class definition, which is a subclass of bpr-task. The first argument to the stop method is the custom class, bpr-task-1. The stop method obtains the current order from the activity and concludes that the commission of the order is 0.1 times the cost-of-order. The method includes a call next method statement before the custom portion of the method so that the custom portion of the method executes after the default stop method.



Direct superior classes   bpr-task



BPR-TASK-1::BPR-STOP-METHOD

```
bpr-stop-method (task: class bpr-task-1,
  activity:class bpr-activity, ui-client-item:
  class ui-client-item)
current-order: class order;
begin
{
Custom method
}
   call next method;
   current-order = the order that is a-bpr-
     input-of-activity activity;
   conclude that the commission of current-
     order = 0.1 * the cost-of-order of current-
     order;

end
```

Customizing the Start Method

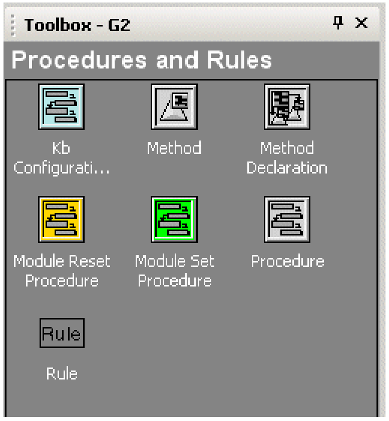
In addition to customizing the custom stop method for a block, you can customize the start method; you can only customize the start method of a block, not an instrument. The start method executes at the beginning of block processing, before the block executes the stop method and before the block computes its duration.

To customize the start method of a block:

1 Create a subclass of the block you want to customize.

2 Create a new method named bpr-start-method that has the desired behavior, as follows:

a Choose View > Toolbox - G2 and display the Procedures and Rules palette:



b Create a Method and place it on your customization workspace.

c Choose edit on the stop method.

d Edit the first argument to the start method to refer to the custom subclass you just created.

Editing the method's first argument changes the qualified name of the method so that it corresponds to the custom subclass.

e Edit the second argument to the method to refer to the bpr-activity class.

f Edit the third argument to the method to refer to the ui-client-item class.

g Specify the custom portion of the procedure, as desired.

h Add a call next method statement to the procedure in the desired location.

To cause the block to perform the custom portion of the method before it performs the default behavior, place the call next method statement at the end of the method.

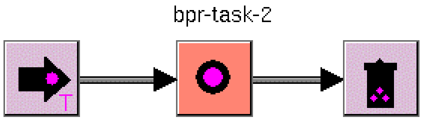
Note Currently, blocks do not define a default start method; however, this might change in the future. Therefore, we recommend you always include a call next method statement in your custom start method for future compatibility.

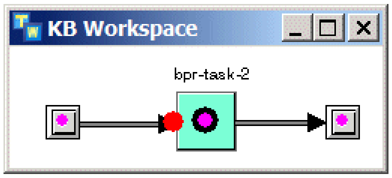
3 Create an instance of your custom class.

The custom class now obtains its default behavior from the custom stop method.

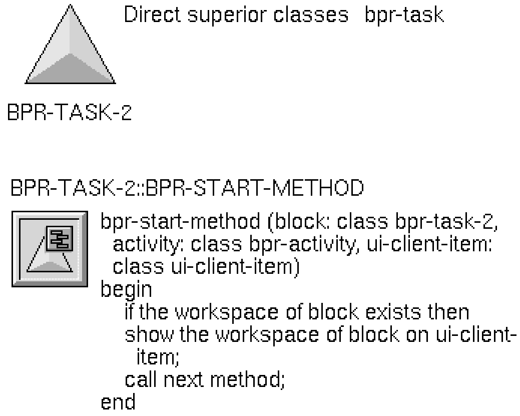
For example, suppose you wanted to display the detail of a Task block whenever a work object arrives at a task with detail. To do this, first create a custom subclass of a Task block, then create a custom start method that shows the workspace of the custom task.

This simple model shows a Task block with detail, which contain the custom Task block. If you step through the model, the detail automatically appears when the work object arrives at the Custom Task block on the detail.





Here is the custom start method for the Custom Task block and the bpr-task-2 class definition. The custom task is an instance of the bpr-task-2 custom class, thus the first argument to the custom start method is the custom class, bpr-task-2. The method tests to see if the workspace of the custom task exists, and then shows the workspace on the current window. The start method includes a call next method statement after the custom portion of the method so the subworkspace displays before the block does any processing.



```
Direct superior classes   bpr-task

BPR-TASK-2

BPR-TASK-2::BPR-START-METHOD
bpr-start-method (block: class bpr-task-2,
  activity: class bpr-activity, ui-client-item:
  class ui-client-item)
begin
    if the workspace of block exists then
    show the workspace of block on ui-client-
      item;
    call next method;
end
```

Customizing Procedures

You can customize various procedures for blocks, paths, instruments, work objects, resources, resource managers, surrogates, and scenarios. To do this, you define a custom procedure, then refer to this custom procedure name in the properties dialog of the object.

Certain default procedures exist on the various subworkspaces of the Methods workspace, while others do not.

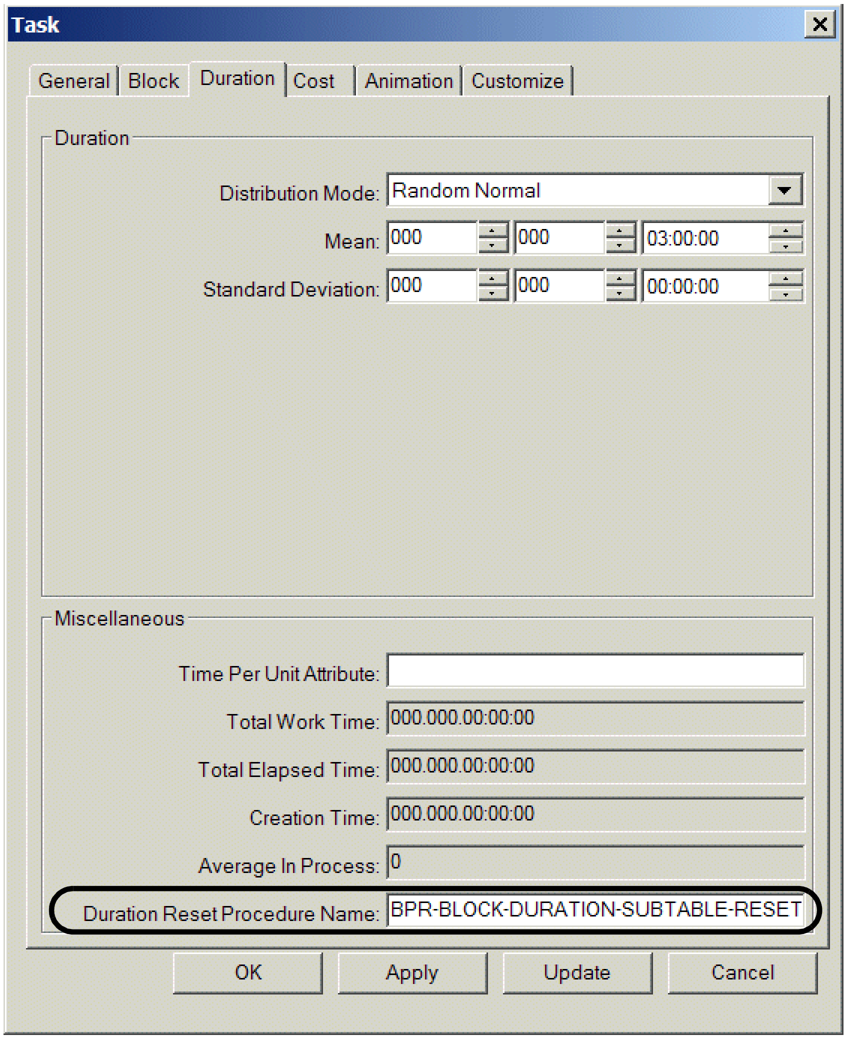
For information on the procedures that you can customize, see Common Customization Features on page 30.

Editing Procedure Names

To edit procedure names:

➔ In Developer mode, display the properties dialog for the object, click the various tabs, and edit the procedure name attributes, as needed.

For example, the following figures show the procedure name attributes that you can edit in Developer mode on the Duration, Cost, Animation, and Customize tab pages of the properties dialog for a Task block. The other objects provide similar procedure name attributes.



Duration

Distribution Mode: Random Normal

Mean: 000 000 03:00:00

Standard Deviation: 000 000 00:00:00

Miscellaneous

Time Per Unit Attribute:

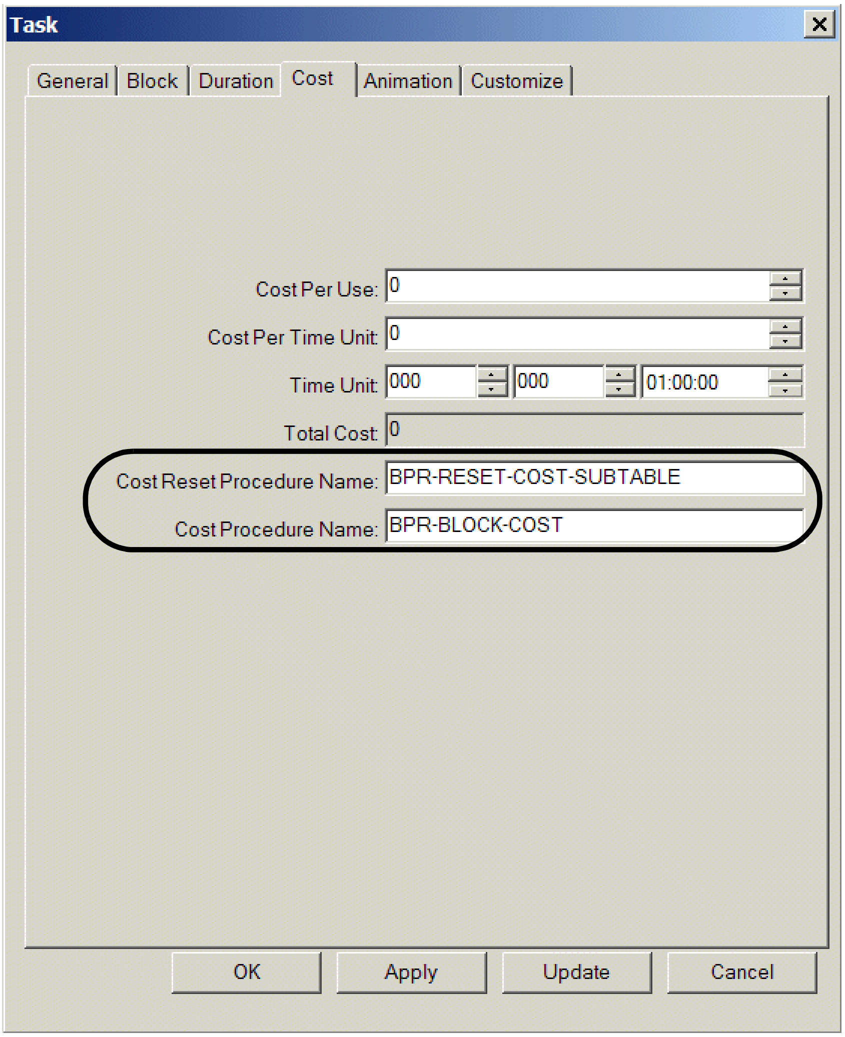
Total Work Time: 000.000.00:00:00

Total Elapsed Time: 000.000.00:00:00

Creation Time: 000.000.00:00:00

Average In Process: 0

Duration Reset Procedure Name: BPR-BLOCK-DURATION-SUBTABLE-RESET



Cost Per Use: 0

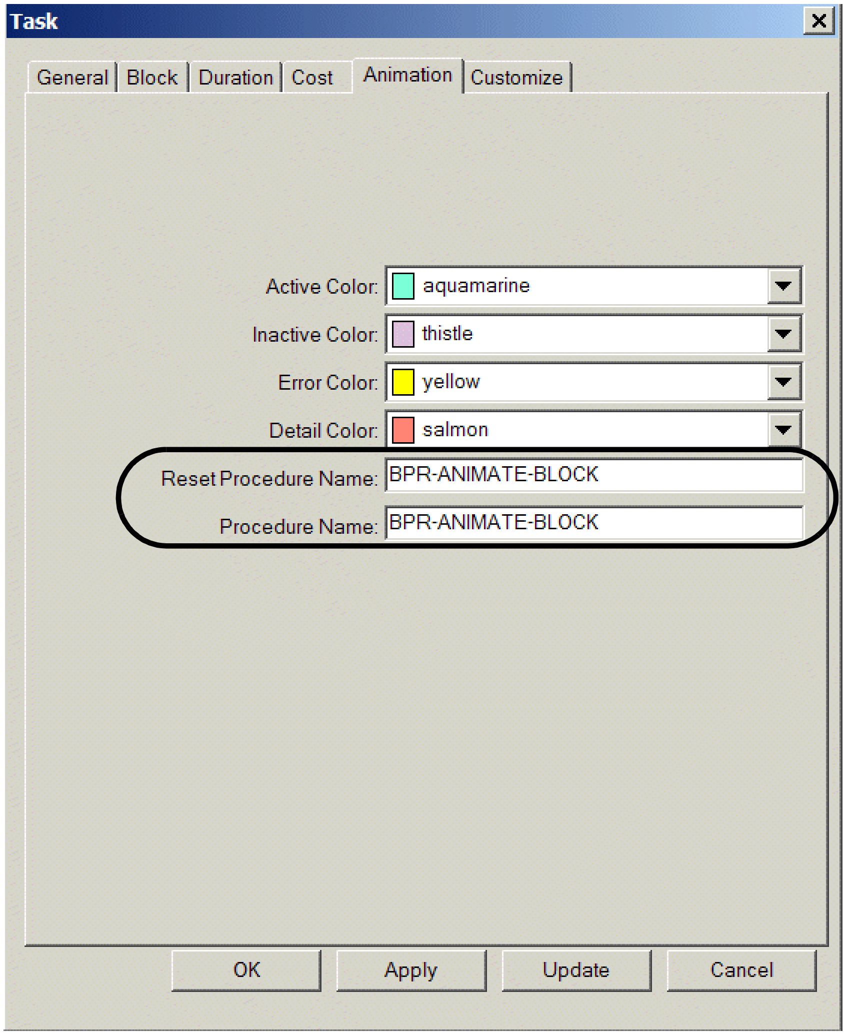
Cost Per Time Unit: 0

Time Unit: 000 000 01:00:00

Total Cost: 0

Cost Reset Procedure Name: BPR-RESET-COST-SUBTABLE

Cost Procedure Name: BPR-BLOCK-COST



Active Color: aquamarine

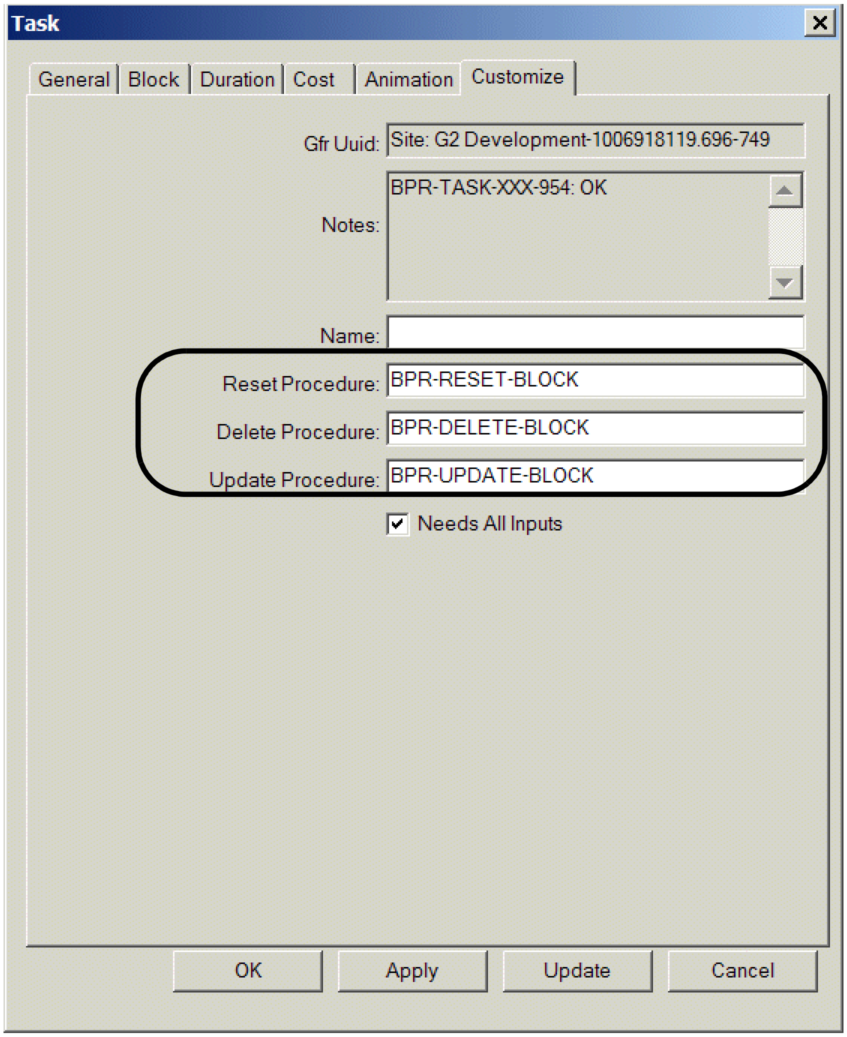
Inactive Color: thistle

Error Color: yellow

Detail Color: salmon

Reset Procedure Name: BPR-ANIMATE-BLOCK

Procedure Name: BPR-ANIMATE-BLOCK



Gfr Uuid: Site: G2 Development-1006918119.696-749

Notes: BPR-TASK-XXX-954: OK

Name:

Reset Procedure: BPR-RESET-BLOCK

Delete Procedure: BPR-DELETE-BLOCK

Update Procedure: BPR-UPDATE-BLOCK

☑ Needs All Inputs

Customizing Reset, Delete, and Update Procedures

The reset, delete, and update procedures are proprietary and, therefore, do not exist on any workspace. To create custom reset, delete, and update procedures for the various types of objects, create a new procedure with these signatures:

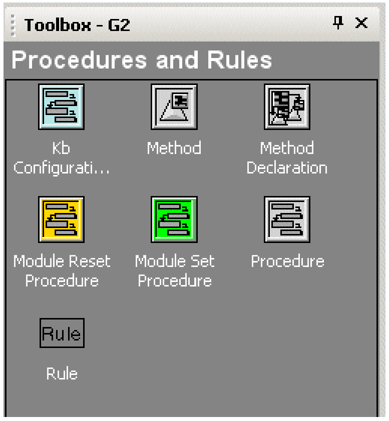
Class	Procedures
bpr-block	bpr-reset-block (*block*: class bpr-block) bpr-delete-block (*block*: class bpr-block) bpr-update-block (*block*: class bpr-block)
bpr-path	bpr-reset-path (*path*: class bpr-path) bpr-delete-path (*path*: class bpr-path) bpr-update-path (*path*: class bpr-path)
bpr-instrument	bpr-reset-instrument (*instrument*: class bpr-instrument)
bpr-object	bpr-reset-object (*object*: class bpr-object) bpr-delete-object (*object*: class bpr-object) bpr-update-object-metrics (*object*: class bpr-object)
bpr-resource-manager	bpr-reset-resource-manager (*manager*: class bpr-resource-manager)

To customize reset, delete, and update procedures:

1 Create a subclass of the block you want to customize.

2 Create a new procedure with the desired behavior, as follows:

   a Choose View > Toolbox - G2 and display the Procedures and Rules palette:



   b Create a Procedure and place it on your customization workspace.

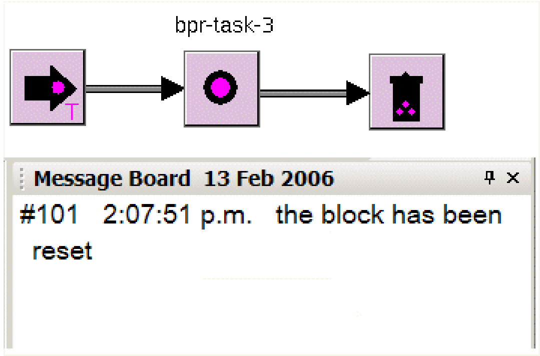
   c Choose edit on the procedure and edit the text of the procedure, using the signatures in the table above, as desired.

   The procedure name should be unique, for example, bpr-reset-task-3.

3 Create an instance of your custom class.

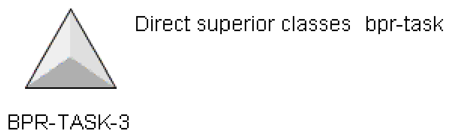
4 Edit the appropriate procedure name attribute in the properties dialog for the custom block.

For example, you might create a custom reset procedure for a custom Task block that posts a message to the Message Board when the block resets, as this example shows:



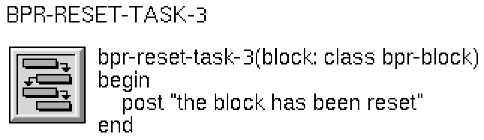
#101 2:07:51 p.m. the block has been reset

Here is the custom procedure named bpr-reset-task-3 and the custom block class definition:



Direct superior classes bpr-task

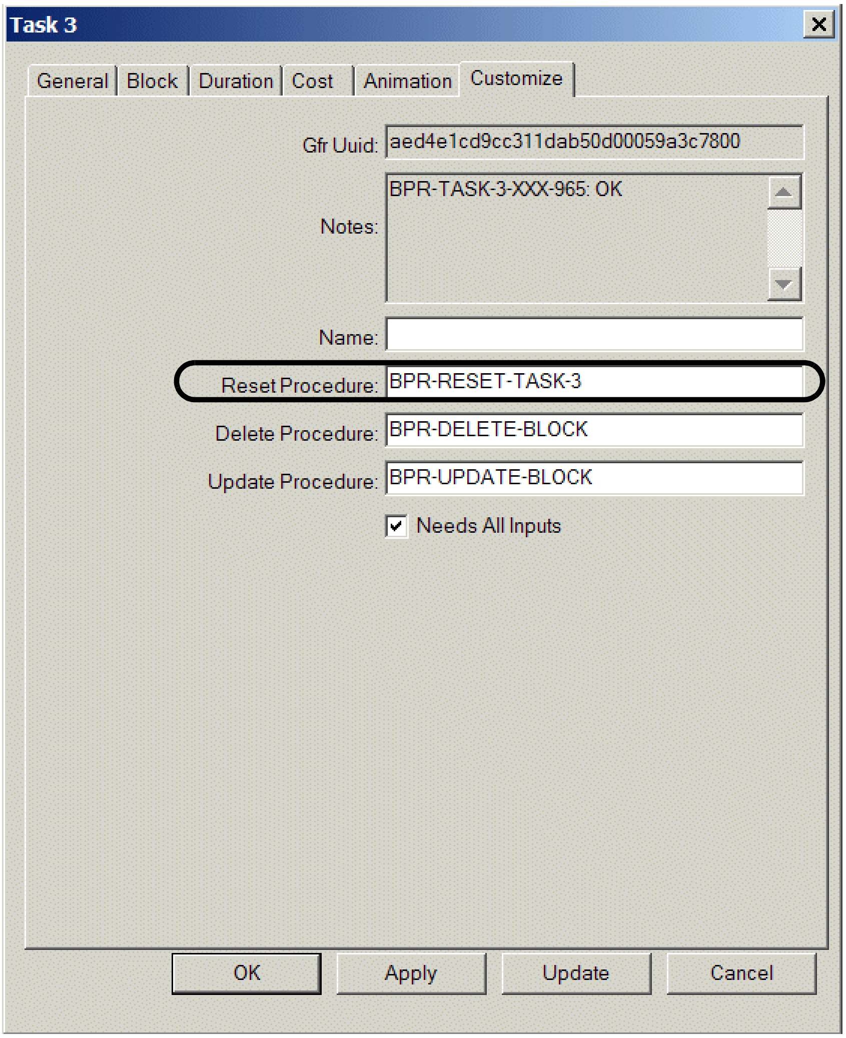
BPR-TASK-3



BPR-RESET-TASK-3

```
bpr-reset-task-3(block: class bpr-block)
begin
   post "the block has been reset"
end
```

You edit the Reset Procedure Name attribute on the Customize tab of the custom block to refer to the custom procedure:



Gfr Uuid: aed4e1cd9cc311dab50d00059a3c7800

Notes: BPR-TASK-3-XXX-965: OK

Name:

Reset Procedure: BPR-RESET-TASK-3

Delete Procedure: BPR-DELETE-BLOCK

Update Procedure: BPR-UPDATE-BLOCK

☑ Needs All Inputs

Customizing Animation, Duration, and Cost Procedures

To customize the animation, duration, and cost procedures of objects, you follow the steps described in Customizing Reset, Delete, and Update Procedures on page 53, except that:

- You can copy the default procedure and edit it, as needed.
- You edit the procedure names on the Animation, Duration, and Cost tabs of the properties dialog.

For details, see Displaying Default Subobject Procedures on page 60.

Customizing Specific Procedures

You can customize procedures for specific blocks, instruments, and resource managers. To do this, you follow the steps described in Customizing Reset, Delete, and Update Procedures on page 53, except that:

- You can copy the default procedure and edit it, as needed.
- You edit the procedure names on the Block tab, the Instrument tab, and the Allocate and Deallocate tabs, respectively.

For details, see:

- Customizing Specific Blocks.
- Customizing Specific Instruments.
- Customizing How Resource Managers Allocate Resources.

Editing Subobjects

ReThink objects define certain attributes, whose values are **subobjects** that you can customize. These attributes are:

- Animation-subtable, which defines the default colors and animation procedure.
- Duration-subtable, which defines the default procedure that computes duration and utilization statistics.
- Cost-subtable, which defines the default procedure that computes total cost.

Blocks, work objects, resources, and resource managers allow you to customize the animation, duration, and cost subtables. Paths, instruments, and surrogates allow you to customize the animation subtable, only.

Techniques for Editing Subobjects

You customize subobjects differently, depending on the type of customization:

To customize...	Do this...
The default procedure of a subobject	Edit the procedure in an instance of the custom class.
The default attributes of a subobject	Edit the subobject in an instance of the custom class.
New attributes for a subobject	Replace the subobject in the custom class definition.
The subobjects of a work object	Replace the subobject in the custom class definition, because you cannot edit an instance of a work object.

The following headings explain how to perform each of these types of customizations in a generic way, as well as how to customize particular types of subobjects.

Displaying Default Subobject Classes

The default values of the attributes that define subobjects depend on the type of object, as this table shows:

Attribute	Default Value	Example	Description
animation-subtable	a bpr-*object-type*-animation-subtable	a bpr-instrument-animation-subtable	Specifies the procedure the object uses to animate, and specifies the default colors the object uses when it is in an active, inactive, or error state.

Attribute	Default Value	Example	Description
duration-subtable	a bpr-*object-type*-duration-subtable	a bpr-object-duration-subtable	Specifies the procedure the object uses to compute duration, specifies the timing parameters you configure during modeling, and computes summary timing statistics for the object.
cost-subtable	a bpr-*object-type*-cost-subtable	a bpr-block-cost-subtable	Specifies the procedure the object uses to compute total cost, specifies the fixed and variable costs you configure during modeling, and computes summary cost statistics for the object.

For the specific default values of each ReThink class, see Common Customization Features on page 30.

To display the default subobject class:

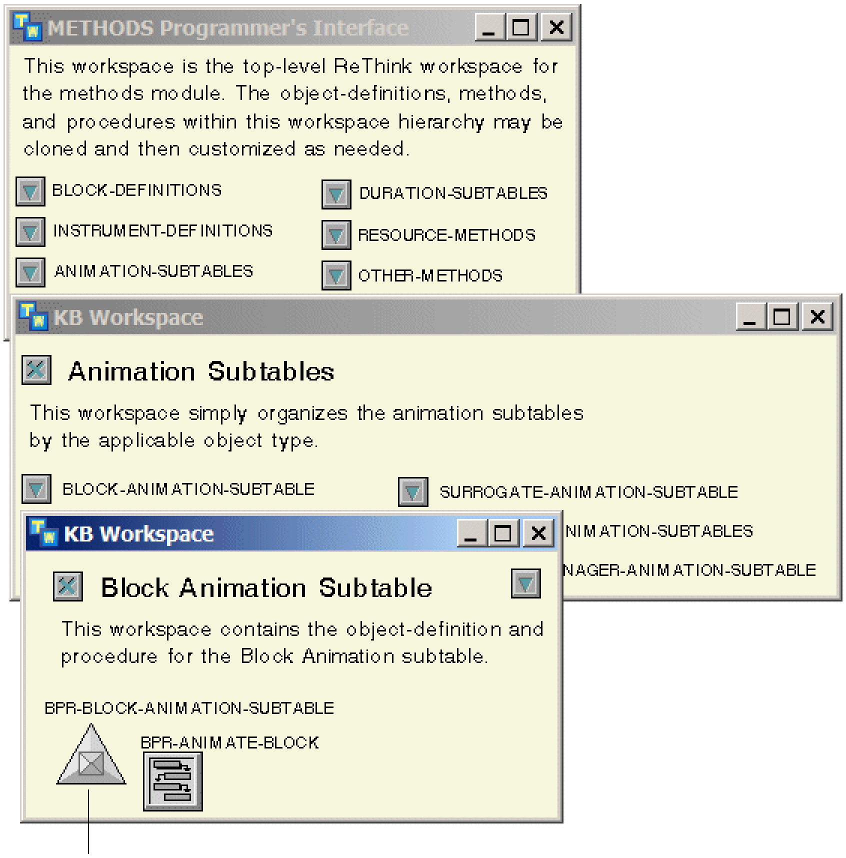
1. Display the Methods workspace.

   For details, see The methods and methods-online Modules on page 11.

2. Click the button associated with Animation Subtables, Duration Subtables, or Cost Subtables workspace, as needed.

3. Click the button associated with the type of object whose subobject you want to customize.

For example, here is the bpr-block-animation-subtable class definition on the Block Animation Subtable workspace:



This workspace is the top-level ReThink workspace for the methods module. The object-definitions, methods, and procedures within this workspace hierarchy may be cloned and then customized as needed.

Animation Subtables

This workspace simply organizes the animation subtables by the applicable object type.

Block Animation Subtable

This workspace contains the object-definition and procedure for the Block Animation subtable.

bpr-block-animation-subtable class

Displaying Default Subobject Procedures

Each subobject has an attribute named procedure-name, which determines the default procedure for the subobject. The default procedure name varies, depending on the type of object and the type of subobject. This table summarizes the default procedures for each type of subobject and for each type of object:

	Object Type					
Subobject	**Block**	**Instrument**	**Resource/ Work Object**	**Resource Manager**	**Surrogate**	**Path**
animation-subtable	bpr-animate-block	bpr-animate-instrument	bpr-animate-object	bpr-animate-resource-manager	bpr-animate-surrogate	bpr-animate-path
duration-subtable	bpr-random-normal-duration	N/A	bpr-object-duration	N/A	N/A	N/A
cost-subtable	bpr-block-cost	N/A	bpr-object-cost	N/A	N/A	N/A

As the table shows, not all types of ReThink objects define procedures for every subobject. Specifically, instruments, resource managers, surrogates, and paths only define an animation procedure.

Also, the default procedure-name of the duration-subtable object for all blocks except the Source block is bpr-random-normal-duration. The default procedure-name for the Source block is bpr-random-exponential-duration.

To edit the default procedure of any type of object except a work object, create an instance of a custom class and edit the Procedure Name on the appropriate tab page of the properties dialog. For details, see Customizing Procedures on page 48.

To edit the default procedure of a subobject for a work object, you must edit the default procedure in the definition of the class. For details, see Customizing Subobjects of Work Objects on page 66.

To display the default subobject procedures:

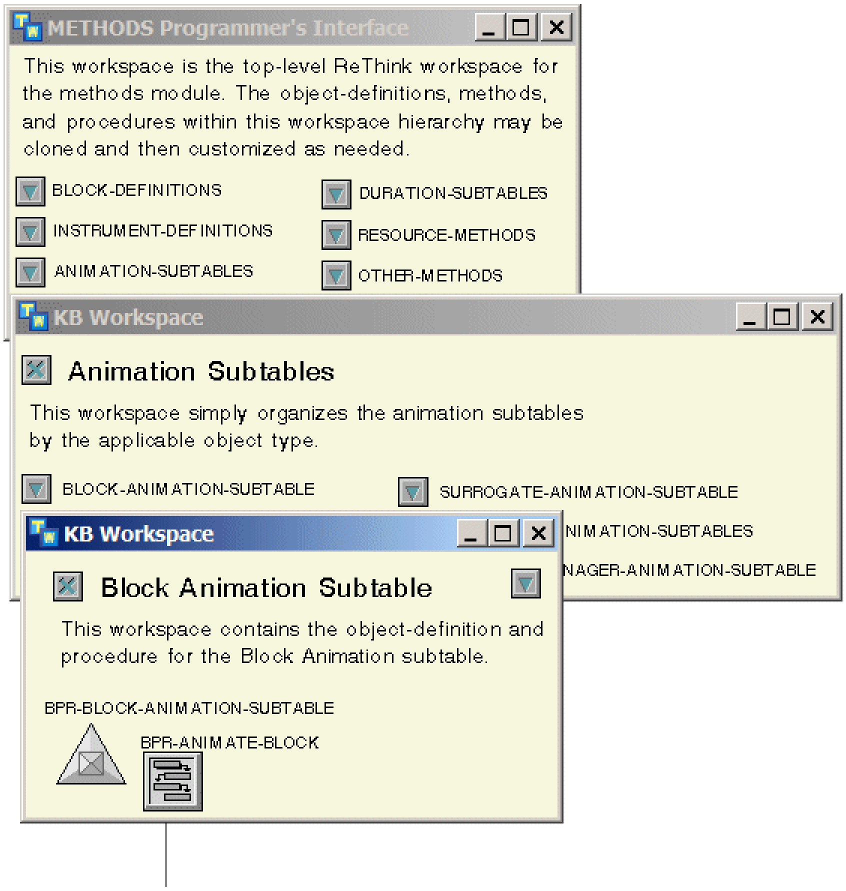
1. Display the Methods workspace.

   For details, see The methods and methods-online Modules on page 11.

2. Click the button associated with Animation Subtables, Duration Subtables, or Cost Subtables workspace, as needed.

3. Click the button associated with the type of object whose subobject procedure you want to customize.

For example, here is the bpr-block-animation procedure on the Block Animation Subtable workspace:



METHODS Programmer's Interface

This workspace is the top-level ReThink workspace for the methods module. The object-definitions, methods, and procedures within this workspace hierarchy may be cloned and then customized as needed.

KB Workspace

Animation Subtables

This workspace simply organizes the animation subtables by the applicable object type.

KB Workspace

Block Animation Subtable

This workspace contains the object-definition and procedure for the Block Animation subtable.

bpr-animate-block procedure

Editing Color Attributes of Animation Subobjects

You can edit the default colors of subobjects by editing the color attributes on the Animation tab of an instance of a custom class. These attributes are available in Modeler mode.

You can edit these attribute attributes of animation subobjects:

Attribute Name	Description
active-color	The color blocks flash when they evaluate, the color work objects flash when they are active, the color resources flash when they are allocated, and the color instruments flash when they evaluate.
inactive-color	The idle color for blocks, work objects, resources, and instruments.
error-color	The color all objects flash when they are in an error state.
detail-color	The color a Task block uses when it has detail.
waiting-color	The color a path uses when it is waiting for the attached block to start processing. A block might be waiting for resources, for other work objects, or because the number of current activities is equal to the Maximum Activities of the block.
empty-color	The color a path uses when it is not waiting for the attached block.
selected-color	The color a path uses when the user selects it for some operation, such as choosing the empty container output path of a Remove block.

To edit the color attribute of an animation subobject:

1 Create a custom class of ReThink object.

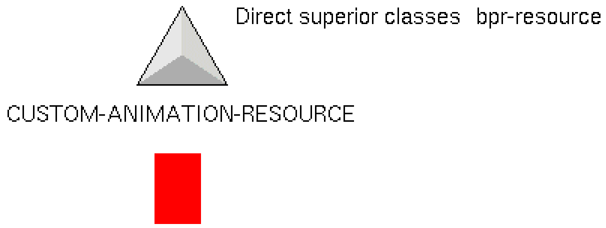
For example, you might create a custom resource class named custom-animation-resource that animates, using different colors.

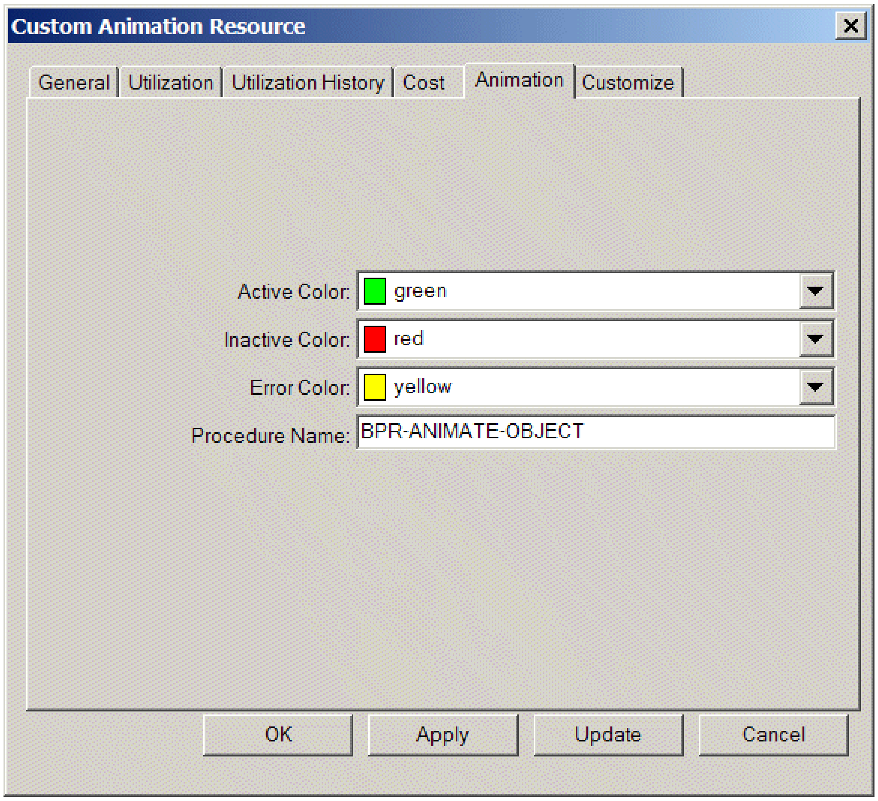
2 Create an instance of the custom class.

3 Display the properties dialog for the object and click the Animation tab.

4 Edit the color attributes, as needed.

For example, you might edit the Active Color and Inactive Color of a custom resource, as follows:


Direct superior classes bpr-resource
CUSTOM-ANIMATION-RESOURCE



Custom Animation Resource

Active Color: 🟩 green

Inactive Color: 🟥 red

Error Color: 🟨 yellow

Procedure Name: BPR-ANIMATE-OBJECT

Creating New Attributes for Subobjects

Suppose you are adding a new attribute to a subobject, for example, a new color for the animation procedure of a resource. In this case, you must replace the default subobject with a new subobject in the definition of the class. That way, each new instance has the custom attributes in its subtable.

To create new attributes for a subobject:

1 Create a subclass of an existing subobject, whose attributes you want to customize.

For example, you might create a new animation subobject for a custom resource class, as follows:

Attribute	Value
class-name	custom-person-animation-subtable
direct-superior-classes	bpr-object-animation-subtable

2 Configure new attributes with default values for the custom subobject in the class-specific-attributes of the custom subobject's class definition.

For example, you might define a new color attribute named custom-color, which a custom animation procedure named custom-animate-person uses:

Attribute	Value
class-specific-attributes	custom-color is a symbol, initially is green

3 Edit the existing attributes of the subobject, as needed, by configuring attribute-initializations for the class.

For example, you might define a default animation procedure by using a custom procedure named custom-animate-resource:

Attribute	Value
attribute-initializations	procedure-name initially is custom-animate-person

Note If you define the existing attributes in the class-specific-attributes of the class, rather than in the attribute-initializations, the instance creates a class-qualified attribute name for the original attribute, according to G2's multiple inheritance mechanism. However, the internal ReThink procedures use the unqualified attribute name. For more information, see *G2 Reference Manual*.

4 Create the custom procedure by creating and editing an existing procedure, for example, bpr-animate-object.

The new procedure might use the new attributes that the custom subobject defines, in this example, custom-color.

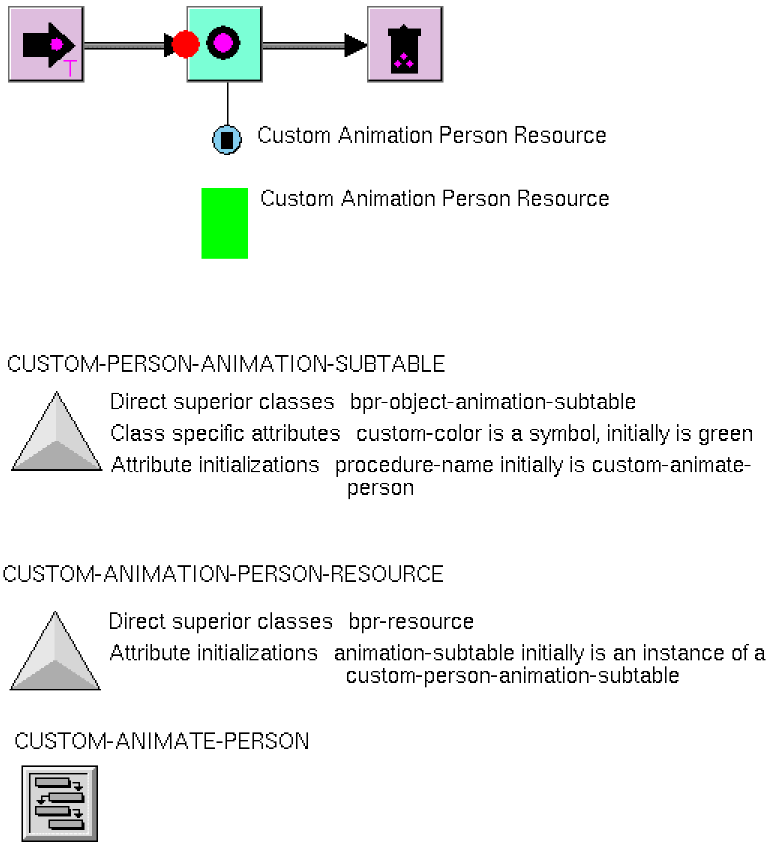
5 Create a custom subclass of ReThink object, whose attribute-initializations refers to the custom subobject class.

For example, your class definition might define these attributes:

Attribute	Value
class-name	custom-animation-person-resource
direct-superior-classes	bpr-resource
attribute-initializations	animation-subtable initially is an instance of a custom-person-animation-subtable

6 Create an instance of the custom class.

This figure shows this example, where the custom-animate-person procedure uses the custom-color of the custom-animation-person-resource as the active color:


T
Custom Animation Person Resource
Custom Animation Person Resource

CUSTOM-PERSON-ANIMATION-SUBTABLE

Direct superior classes	bpr-object-animation-subtable
Class specific attributes	custom-color is a symbol, initially is green
Attribute initializations	procedure-name initially is custom-animate-person

CUSTOM-ANIMATION-PERSON-RESOURCE

Direct superior classes	bpr-resource
Attribute initializations	animation-subtable initially is an instance of a custom-person-animation-subtable

CUSTOM-ANIMATE-PERSON

Here is the text of the custom-animate-person procedure:

```
custom-animate-person(m: class bpr-object-animation-subtable)
object: class bpr-object;
ws: class kb-workspace;
begin
    if the item superior to m does not exist then
        return;
    object = the item superior to m;
    if (the error of object exists and length-of-text(the error of object) > 0) then
        change the flashing icon-color of object to the error-color of m
    else
        if the total-starts of object > the total-stops of object then
            change the flashing icon-color of object to the custom-color of m
        else
            change the flashing icon-color of object to the inactive-color of m;
    if the workspace ws of object exists then
        call g2-work-on-drawing(ws);
end
```

For information about adding custom attributes to tab pages of the properties dialog, see Customizing Properties Dialogs.

Customizing Subobjects of Work Objects

Unlike other types of objects, work objects require that you always replace the existing subobject in the definition of the class, whether you are editing existing attributes or adding new ones; you cannot edit an instance of a work object because work objects are transient.

To customize the subobject of a work object:

1 Create a subclass of an existing ReThink subobject, which is the subobject you are going to replace.

For example, you might create a new cost subobject for a custom work object class, as follows:

Attribute	Value
class-name	custom-object-cost-subtable
direct superior classes	bpr-object-cost-subtable

2 Configure new attributes with default values, as needed, for the custom subobject in the class-specific-attributes of the custom subobject's class definition.

For example, you might add a new attribute named conversion-factor that the work object uses to compute total cost, using a different currency:

Attribute	Value
class-specific-attributes	conversion-factor initially is 1.5

3 Edit the existing attributes of the subobject, as needed, by configuring attribute-initializations for the class.

For example, you might define the default cost procedure by using a custom procedure named custom-object-cost:

Attribute	Value
attribute-initializations	procedure-name initially is custom-object-cost

For information on why you configure attribute-initializations, see the note in Creating New Attributes for Subobjects on page 63.

4 Create the custom cost procedure by copying an existing procedure, for example, bpr-object-cost.

The new procedure might use the new attributes the custom subobject defines, for example, conversion-factor.

5 Create a custom subclass of ReThink object, whose attribute-initializations refers to the custom subobject class.

For example, your class definition might define these attributes:

Attribute	Value
class-name	custom-cost-object
direct-superior-classes	bpr-object
attribute-initializations	cost-subtable initially is an instance of a custom-object-cost-subtable

6 Configure the path type of a block to be the custom subclass.

Customizing Animation

You can customize the animation of objects by:

- Customizing the default colors.

  Follow the steps described in Editing Color Attributes of Animation Subobjects on page 62.

- Customizing the default animation procedure.

  Follow the steps described in Displaying Default Subobject Procedures on page 60.

- Animating custom icon regions.

  Follow the steps described in Creating New Attributes for Subobjects on page 63.

The default animation procedures for ReThink objects cause an icon region named flashing to animate. You can edit the default colors ReThink uses when it animates the flashing icon region, or you can create a new animation procedure that animates a custom icon region. You can also add different types of animation to the procedure, such as moving or rotating the object.

For an example of customizing animation, see Customizing Resource Animation.

Note ReThink generates an error when you run the model if the object you are customizing does not have an icon region named flashing.

In addition to customizing the animation of blocks, instruments, work objects, and resources, you can customize the animation of:

- Resource Managers
- Surrogates
- Paths

To customize the animation of a resource manager:

1. Copy and edit the bpr-animate-resource-manager procedure on the Resource Manager Animation Subtable workspace.
2. Edit the Procedure Name on the Animation tab of a particular Resource Manager to use the new procedure.

To customize the animation of a path:

1 Copy and edit the bpr-animate-path procedure on the Path Animation Subtable workspace.

2 Edit the Procedure name on the Animation tab of a particular path to use the new procedure.

For more information, see Customizing the Paths of a Block.

To customize the animation of a surrogate:

1 Copy and edit the bpr-animate-surrogate procedure on the Surrogate Animation Subtable workspace.

2 Edit the Procedure name on the Animation tab of a particular surrogate to use the new procedure.

Configuring User Preferences

In System-Administrator and Administrator modes, you can configure additional attributes for each user preference. For information about basic user preferences, see “Configuring User Preferences” in Chapter 3 “Working with Models” in the *ReThink User’s Guide*.

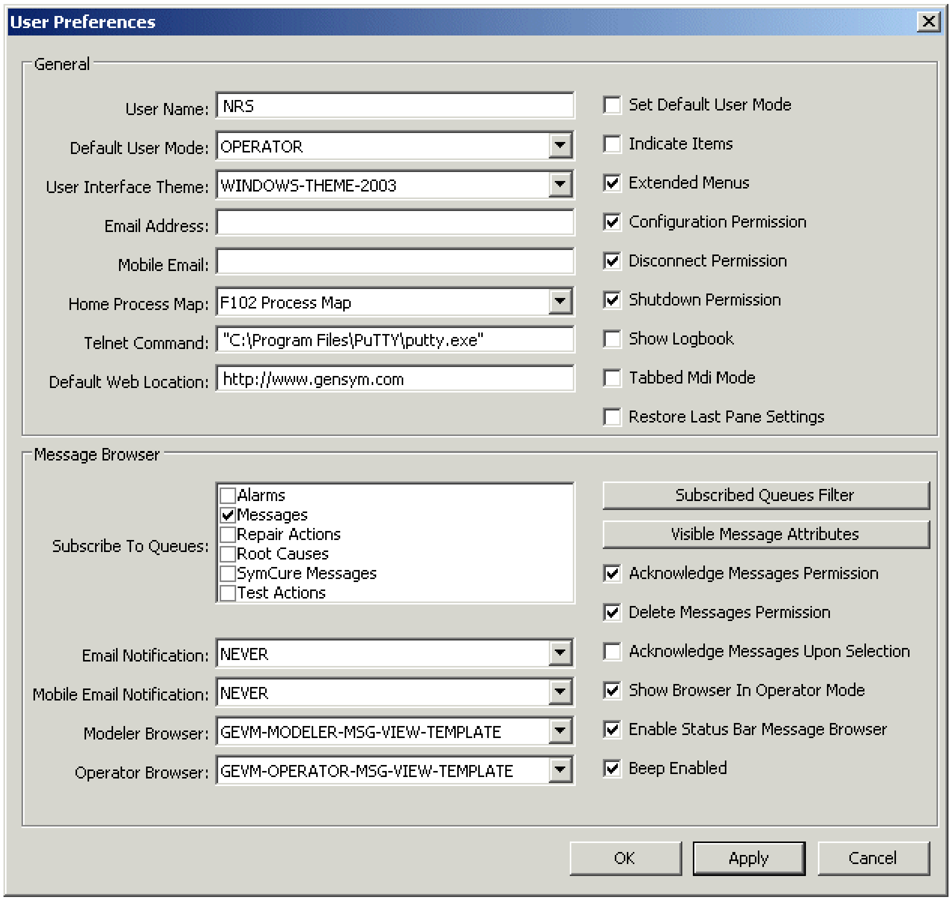
In addition to configuring user preferences that ReThink creates automatically when you start the server and client, you can also create new user preferences for specific clients, based on their user name.

To configure user preferences:

1 Switch to System-Administrator or Administrator mode.

2 Choose Project > System Settings > Users and choose the user preference to configure or create a new user preference, using the Manage dialog.

The User Preferences dialog appears:



User Preferences

General

User Name: NRS

Default User Mode: OPERATOR

User Interface Theme: WINDOWS-THEME-2003

Email Address:

Mobile Email:

Home Process Map: F102 Process Map

Telnet Command: "C:\Program Files\PuTTY\putty.exe"

Default Web Location: http://www.gensym.com

☐ Set Default User Mode
☐ Indicate Items
☑ Extended Menus
☑ Configuration Permission
☑ Disconnect Permission
☑ Shutdown Permission
☐ Show Logbook
☐ Tabbed Mdi Mode
☐ Restore Last Pane Settings

Message Browser

Subscribe To Queues:
☐ Alarms
☑ Messages
☐ Repair Actions
☐ Root Causes
☐ SymCure Messages
☐ Test Actions

Email Notification: NEVER

Mobile Email Notification: NEVER

Modeler Browser: GEVM-MODELER-MSG-VIEW-TEMPLATE

Operator Browser: GEVM-OPERATOR-MSG-VIEW-TEMPLATE

☑ Acknowledge Messages Permission
☑ Delete Messages Permission
☐ Acknowledge Messages Upon Selection
☑ Show Browser In Operator Mode
☑ Enable Status Bar Message Browser
☑ Beep Enabled

3 Configure the customization attributes, as follows:

Attribute	Description
General	
User Name	The user name associated with the user preference. The default User Name is the user name for the current user. To create a user preference for a new user, enter the user name of a user in the `g2.ok` file, which must be a symbol. For details, see Chapter 62 “Licensing and Authorization” in the *G2 Reference Manual*.
Configuration Permission	Whether to allow the user to switch to configure the application in Modeler mode. By default, Configuration Permission is enabled, which means when the operator clicks the close button in the operator interface, ReThink switches to Modeler mode. In Modeler mode, you can create and configure applications, using the top menu bar. When Configuration Permission is disabled, ReThink closes the client when the operator clicks the close button in the operator interface. We recommend that you disable this option for operators.
Disconnect Permission	Whether to allow the user to disconnect the client from the server, using the File > Close menu choice. By default, all users can disconnect the client from the server.
Shutdown Permission	Whether to allow the user to shut down the server, using the File > Exit menu choice. By default, modelers and operators cannot shut down the server.
Show Logbook	Whether to show the G2 Logbook when an error occurs.

Attribute	Description
Message Browser	
Subscribe to Queues	The message queues to which the specified user subscribes. By subscribing to a queue, the user sees messages associated with that queue in the Message Browser view of the operator interface. Messages for the Messages queue appear in the Message Browser.
Subscribed Queues Filter	The default filter to apply for filtering messages in the subscribed queues. For details, see Configuring Filters on page 73.
Visible Message Attributes	The properties to show in the message details. By default, all properties are showing. For details, see Configuring Message Details on page 75.
Acknowledge Messages Permission	Whether to allow the user to acknowledge messages in the Message Browser view of the operator interface. By default, operators can acknowledge messages.
Delete Messages Permission	Whether to allow operators to delete messages in the Message Browser view of the operator interface. By default, users can delete messages.

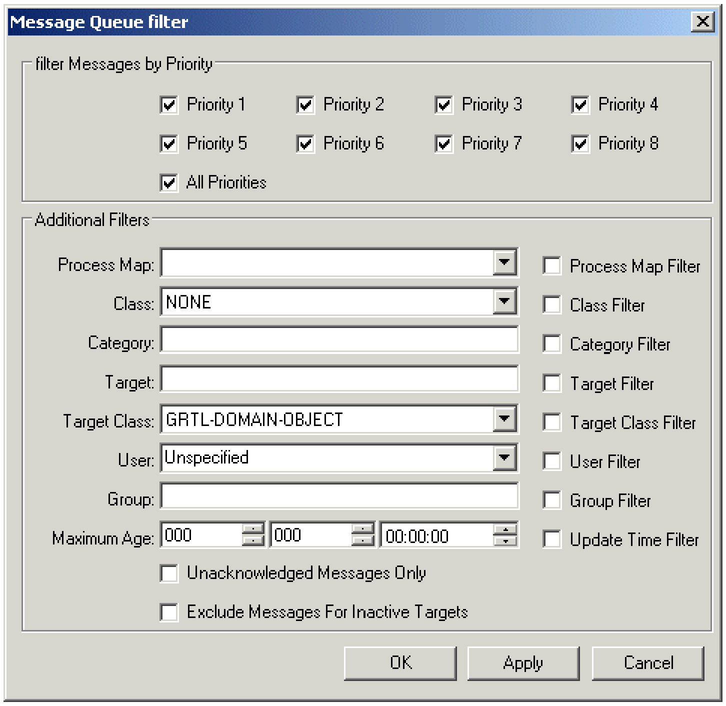
Configuring Filters

By default, the Message Browser shows all messages. You might want to restrict the messages that appear for a particular user in a given user mode. You can filter messages, based on a variety of criteria, including priority, object type, category, and age.

To configuring filters:

➔ In the user preferences dialog, click the Subscribed Queues Filter button and configure the filter criteria.

Here is the default filter dialog:



Message Queue filter

filter Messages by Priority

☑ Priority 1 ☑ Priority 2 ☑ Priority 3 ☑ Priority 4

☑ Priority 5 ☑ Priority 6 ☑ Priority 7 ☑ Priority 8

☑ All Priorities

Additional Filters

Process Map: ☐ Process Map Filter

Class: NONE ☐ Class Filter

Category: ☐ Category Filter

Target: ☐ Target Filter

Target Class: GRTL-DOMAIN-OBJECT ☐ Target Class Filter

User: Unspecified ☐ User Filter

Group: ☐ Group Filter

Maximum Age: 000 000 00:00:00 ☐ Update Time Filter

☐ Unacknowledged Messages Only

☐ Exclude Messages For Inactive Targets

Attribute	Description
Filter Messages by Priority	The priority of the messages to show.
Process Map Process Map Filter	The process map for which to show messages when Process Map Filter is enabled.
Class Class Filter	The classes for which to show messages when Class Filter is enabled.

Attribute	Description
Category Category Filter	The category of messages to show when Category Filter is enabled.
Target Target Filter	The target object for which to show messages when Target Filter is enabled.
Target Class Target Class Filter	The target class for which to show messages when Target Class Filter is enabled.
User User Filter	The user for which to show messages when User Filter is enabled.
Group Group Filter	The group for which to show messages when Group Filter is enabled.
Maximum Age Update Time Filter	The maximum age of messages to show when Update Time Filter is enabled.
Unacknowledged Messages Only	Whether to show unacknowledged messages only. By default, acknowledged messages are visible.
Exclude Messages For Inactive Targets	Whether to exclude messages if the target object status is inactive.

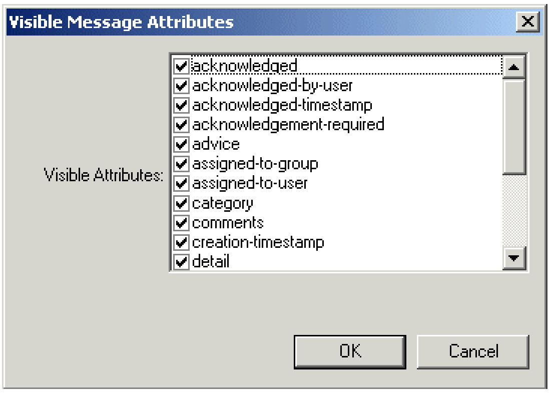
Configuring Message Details

By default, when you click the Properties button for a message in the Message Browser, all message details appear. You can restrict the contents of the message details dialog.

To configure message details:

➔ In the user preferences dialog, click the Message Details button and configure the attributes to appear in the Message Detail Selection dialog by removing attributes from the Selected Attributes column, as needed.

By default, all message details appear:

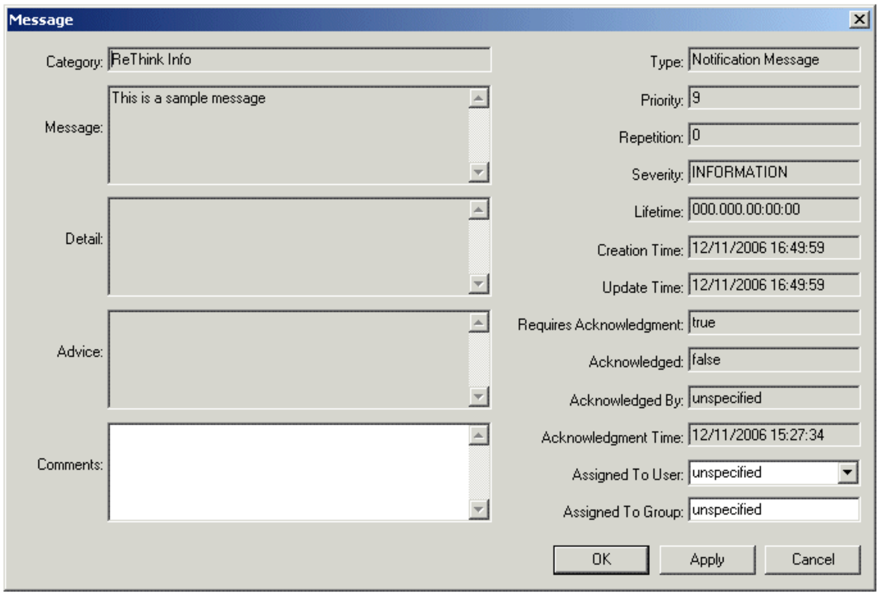


Visible Message Attributes

Visible Attributes:

- ☑ acknowledged
- ☑ acknowledged-by-user
- ☑ acknowledged-timestamp
- ☑ acknowledgement-required
- ☑ advice
- ☑ assigned-to-group
- ☑ assigned-to-user
- ☑ category
- ☑ comments
- ☑ creation-timestamp
- ☑ detail

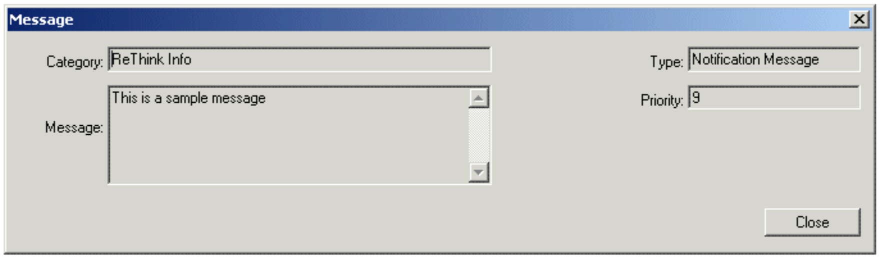
Here is the default message details for a message with all attributes showing:



Message

Category:	ReThink Info
Message:	This is a sample message
Detail:	
Advice:	
Comments:	
Type:	Notification Message
Priority:	9
Repetition:	0
Severity:	INFORMATION
Lifetime:	000.000.00:00:00
Creation Time:	12/11/2006 16:49:59
Update Time:	12/11/2006 16:49:59
Requires Acknowledgment:	true
Acknowledged:	false
Acknowledged By:	unspecified
Acknowledgment Time:	12/11/2006 15:27:34
Assigned To User:	unspecified
Assigned To Group:	unspecified

Here is the message details for a message with just four attributes visible:


Message

Category: ReThink Info

Message: This is a sample message

Type: Notification Message

Priority: 9

Configuring the Excel Macros for Formatting the Report

The default Excel report defines two macros, which ReThink uses to format the report:

- *`DefaultFormatSheet`*, which is the default macro for formatting the report template when you choose Create Report on a report object.
- *`DefaultFormatData`*, which you can use to format the report data when the report updates.

The *`DefaultFormatSheet`* macro creates the column headers and other header text.

To save computational resources, the default Excel report does not use the *`DefaultFormatData`* macro to format the report data. The macro exists, however, in the default report, as an empty macro.

You use Visual Basic to customize the macros, which have these signatures:

```
DefaultFormatSheet (SheetName As String, Activate As Boolean,
                    InputReport As Boolean)

DefaultFormatData (SheetName As String, Activate As Boolean,
                   InputReport As Boolean)
```

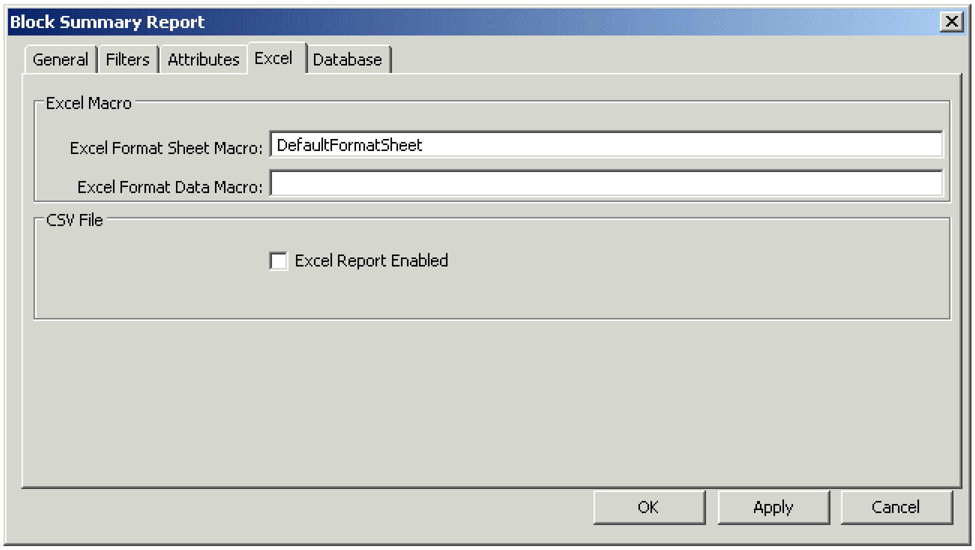
If you edit the macro, you can either create the report again to format it, using the new macro, or you can explicitly format the report in Excel.

To configure the Excel macros for formatting the report:

➔ In System-Administrator or Administrator mode, display the properties dialog for a report, click the Excel tab, and configure the Excel Format Sheet Macro and Excel Format Data Macro parameters.

This figure shows the default dialog for configuring the Excel macros of a Block Summary Report:

By default, the report uses the default macro for formatting the report template, and it uses no macro to format the data.



Excel Macro

Excel Format Sheet Macro: DefaultFormatSheet

Excel Format Data Macro:

CSV File

□ Excel Report Enabled

To format an existing report, using the new macro:

➔ Choose Report > Format from the floating toolbar in Excel.

For details, see “Creating Reports in Excel” in Chapter 8 “Using Reports” in the *ReThink User’s Guide*.

To display the VB code that calls the default macros:

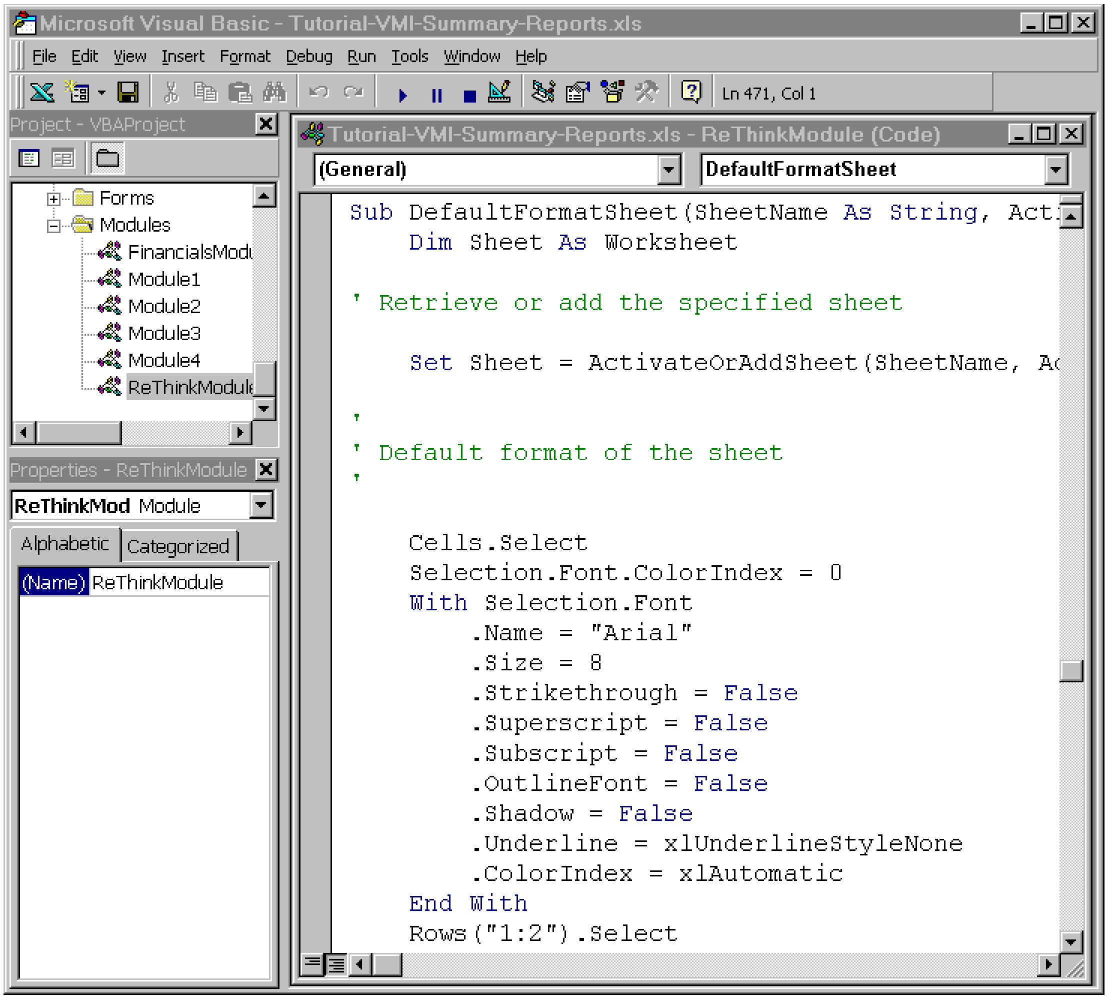
1 In Excel, choose Tools > Macro > Visual Basic Editor.

2 In the Project tree view in the upper-left of the editor, expand the Modules node under the `VBAProject` for the `Default-Summary-Reports.xlsReThink-Summary-Reports.xls`, then double-click the `ReThinkModule`.

   Visual Basic displays an editor that shows the VB code associated with the default spreadsheet.

3 Choose the `DefaultFormatSheet` macro from the scroll list at the upper-right of the code window to view the default macro.

The Visual Basic Editor looks like this:



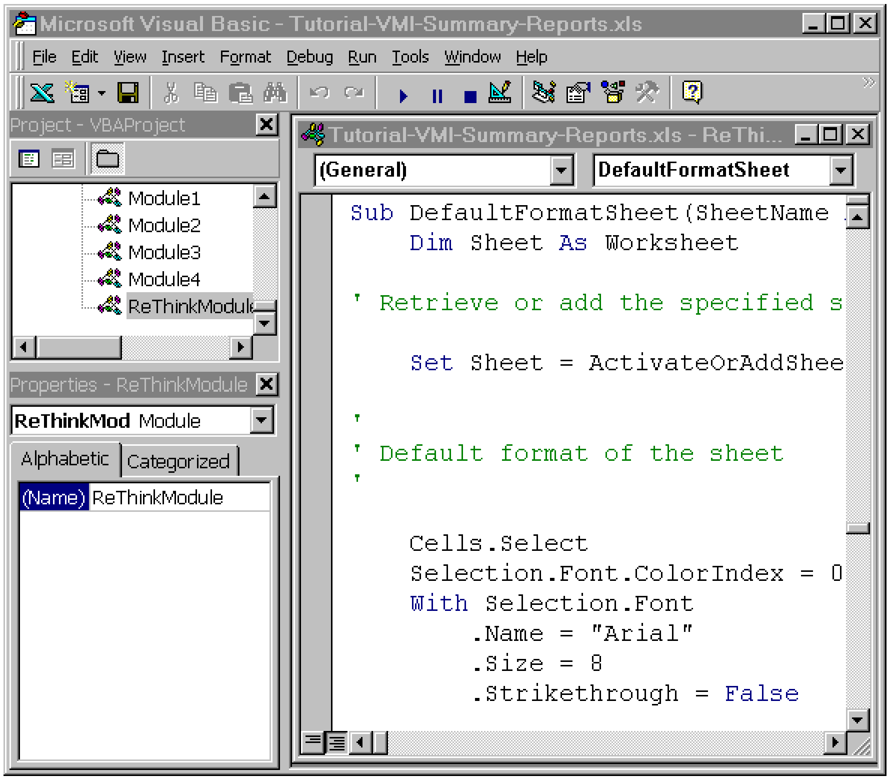
```
Sub DefaultFormatSheet(SheetName As String, Act
    Dim Sheet As Worksheet

' Retrieve or add the specified sheet

    Set Sheet = ActivateOrAddSheet(SheetName, Ac

'
' Default format of the sheet
'

    Cells.Select
    Selection.Font.ColorIndex = 0
    With Selection.Font
        .Name = "Arial"
        .Size = 8
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = xlUnderlineStyleNone
        .ColorIndex = xlAutomatic
    End With
    Rows("1:2").Select
```


Microsoft Visual Basic - Tutorial-VMI-Summary-Reports.xls
File Edit View Insert Format Debug Run Tools Window Help
Project - VBAProject
Module1
Module2
Module3
Module4
ReThinkModule
Properties - ReThinkModule
ReThinkMod Module
Alphabetic
Categorized
(Name) ReThinkModule
Tutorial-VMI-Summary-Reports.xls - ReThi...
(General)
DefaultFormatSheet

```
Sub DefaultFormatSheet(SheetName
    Dim Sheet As Worksheet

' Retrieve or add the specified s

    Set Sheet = ActivateOrAddShee

'
' Default format of the sheet
'

    Cells.Select
    Selection.Font.ColorIndex = 0
    With Selection.Font
        .Name = "Arial"
        .Size = 8
        .Strikethrough = False
```

4 Choose the `DefaultFormatData` macro from the scroll list to view the arguments to the macro that formats data when the report updates.

Using Visual Basic, you would record a new macro for formatting the report template and/or report data, which would appear in this code file. See the Visual Basic documentation for details.

Part II

Customizing ReThink Objects

Customizing Blocks

Provides specific descriptions and examples of how to customize blocks in general, as well as how to customize specific blocks.

Introduction

You can customize the following aspects of any block:

- Its characteristics by adding new attributes.
- Its behavior by modifying its default methods.
- Its appearance when it changes from active to inactive by modifying its default colors and animation procedure.
- The way it computes duration statistics.

- The way it computes cost statistics.
- The behavior of its input paths.

To perform any of these customizations, first create a subclass of the block you are customizing, then make the customizations to the subclass of the block.

To customize the default behavior of a block, you edit the stop method for the block. To customize the default behavior of particular blocks, you customize specific block procedures. For example, you might want to create a different version of the Retrieve block that retrieves objects from a pool based on some criteria other than an association name or an attribute value.

For general information on how to customize, see How to Customize ReThink.

Adding Attributes to a Custom Block

You can add attributes to the definition of a block to:

- Provide information to the block, such as a priority, which the block refers to in its custom procedures and methods.
- Compute information about the block in its custom procedures and methods, and store the value in the custom attribute.

For general information about adding attributes to block definitions, see Adding Attributes to the Class.

For example, a custom Task block that notifies people about a meeting might require a new attribute called meeting-time, which you configure during modeling by specifying the time of the meeting. The block notifies the user of the meeting time when it executes.

When you add attributes to a custom block, you typically edit the block's icon, as described in the *ReThink User's Guide*.

To add attributes to a custom block:

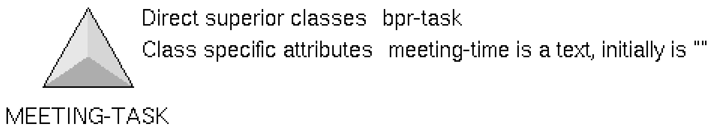
1 Create a subclass of bpr-task by creating a class definition as follows:

Attribute	Value
class-name	meeting-task
direct-superior-classes	bpr-task

2 Edit the class-specific-attributes of the meeting-task to create an attribute named meeting-time:

Attribute	Value
class-specific-attributes	meeting-time is a text, initially is ""

For example, here is custom Task block class definition:



Direct superior classes bpr-task
Class specific attributes meeting-time is a text, initially is ""

MEETING-TASK

In the next section, you will use this attribute to announce the time of the meeting.

Customizing the Default Behavior of a Block

To customize the default behavior of a block, you customize the bpr-stop-method or the bpr-start-method.

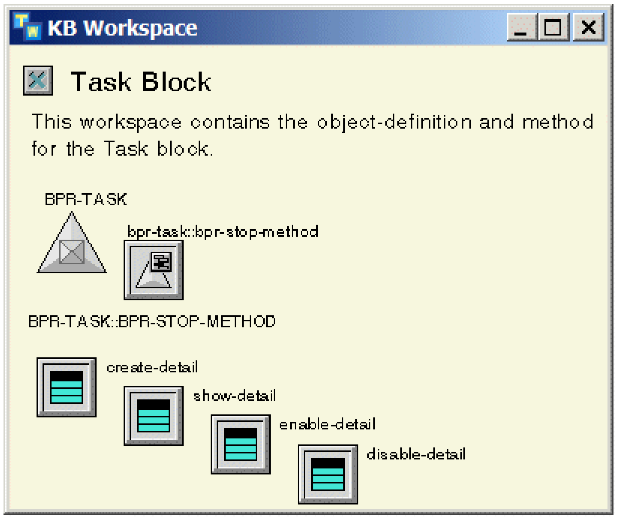
For general information about how to customize the default behavior of blocks, see Customizing the Default Behavior of Blocks or Instruments.

Building on the Meeting task example, suppose you want to customize the default operations of the block to inform the operator of the meeting time.

To customize the default behavior of a custom block:

1 Display the Methods workspace, choose Block Definitions, then click the button next to the block whose behavior you want to customize.

For example, here is the Task Block workspace:



KB Workspace

Task Block

This workspace contains the object-definition and method for the Task block.

2 Copy the bpr-stop-method method for the block and place it on your customization workspace.

In this example, you would copy bpr-task::bpr-stop-method.

3 Edit the method by changing its first argument to be the name of your custom class, for example, meeting-task.

4 Customize the method, as needed.

For example, you might add a post statement to notify the operator of the meeting-time of the task, as follows:

```
post "There is a meeting at [the meeting-time of task]"
```

Meeting-time is a custom attribute of the block, and meeting-task is the argument to the method that refers to the custom Task block class.

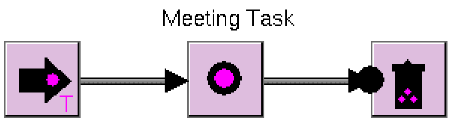
5 Delete the default statements in the method.

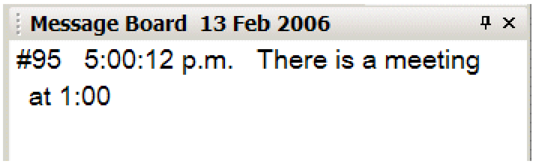
6 Add a call next method statement at the end of the method.

7 Create an instance of the custom block.

The custom block uses the custom stop method.

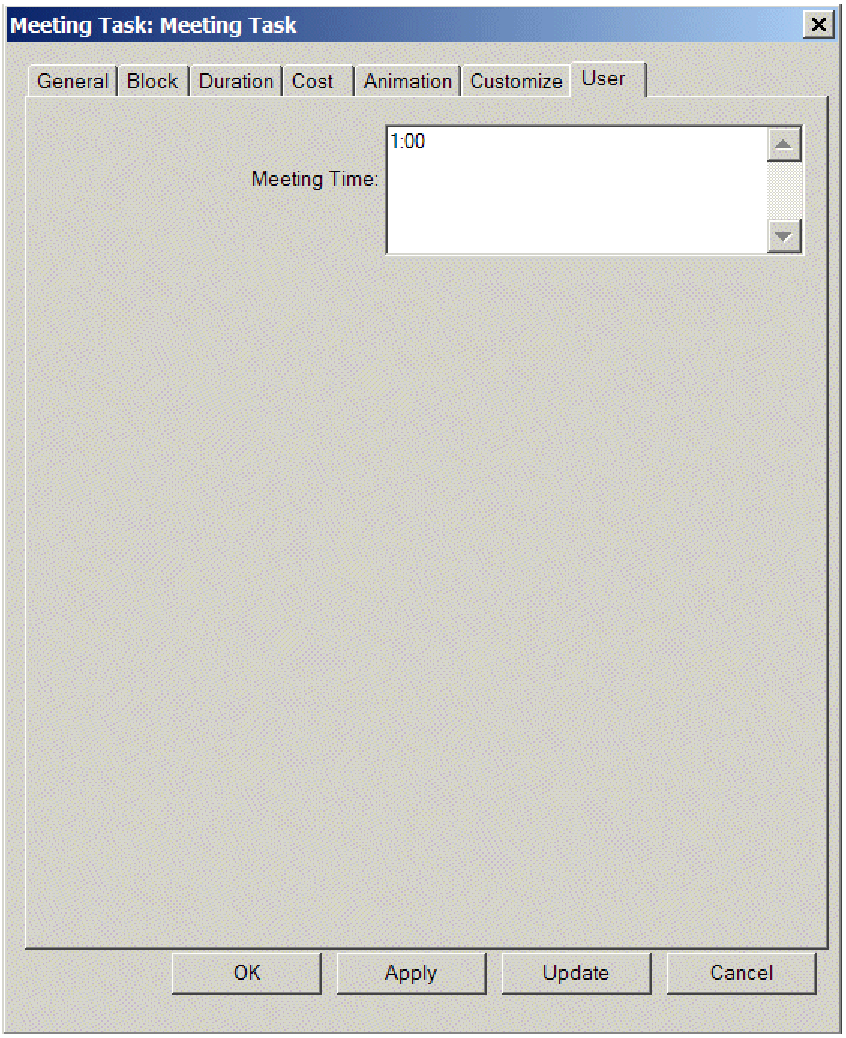
This figure shows an example that uses the custom task block. When the work object passes to the block downstream of the Meeting Task, the following message appears:




Message Board 13 Feb 2006

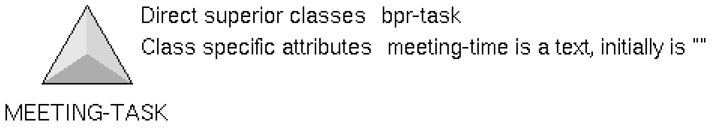
#95 5:00:12 p.m. There is a meeting at 1:00

Here is the User Tab of the properties dialog for the custom task block:


Meeting Task: Meeting Task

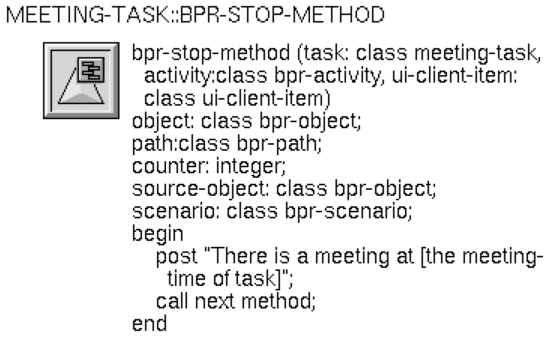
Meeting Time: 1:00

Here is the class definition for the meeting-task and its custom stop procedure:



```
Direct superior classes   bpr-task
Class specific attributes   meeting-time is a text, initially is ""

MEETING-TASK
```



```
MEETING-TASK::BPR-STOP-METHOD

bpr-stop-method (task: class meeting-task,
  activity:class bpr-activity, ui-client-item:
  class ui-client-item)
object: class bpr-object;
path:class bpr-path;
counter: integer;
source-object: class bpr-object;
scenario: class bpr-scenario;
begin
   post "There is a meeting at [the meeting-
     time of task]";
   call next method;
end
```

Customizing How Blocks Animate

You can customize the default colors a block uses to animate and the default procedure that defines how the block animates.

You customize the colors in the animation subobject when you want to create a class of blocks that uses custom colors. Otherwise, you can customize the colors of individual blocks on the Animation tab of the Set Block dialog, as described in the *ReThink User's Guide*.

When you customize block animation, you can either:

- Use the default animation procedure, bpr-animate-block, to animate the flashing icon region and edit the existing colors in the animation subtable of an instance of a custom class.
- Add new regions to your icon, add new attributes to a subclass of bpr-block-animation-subtable that animate the new regions, and create a custom animation procedure that animates the new regions.

Note ReThink generates an error when you run the model if the block you are customizing does not have an icon region named flashing.

For information on...	See...
Editing the default animation procedure for a block	Displaying Default Subobject Procedures.
Editing the default colors for animating blocks	Editing Color Attributes of Animation Subobjects.
Adding new color attributes to the animation subtable	Creating New Attributes for Subobjects.

Editing the Default Colors of a Block

Suppose you want all instances of a class to turn to magenta when they are active, blue when they are inactive, and red when an error condition occurs. To do this, edit the color attributes of the animation-subtable in an instance of the class.

To edit the default colors of a block:

1. Create a subclass of block whose default colors you want to edit, or use an existing ReThink class, such as bpr-source.

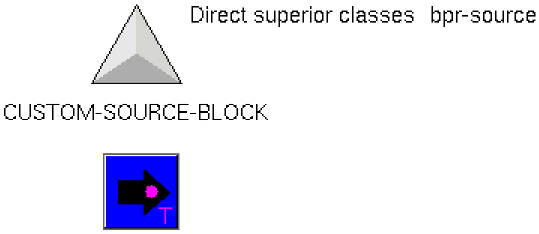
   For example:

Attribute	Value
class-name	custom-source-block
direct-superior-classes	bpr-source

2. Create a new instance of the custom block.
3. Display properties dialog for the custom block and configure the Active Color, Inactive Color, and Error Color attributes.

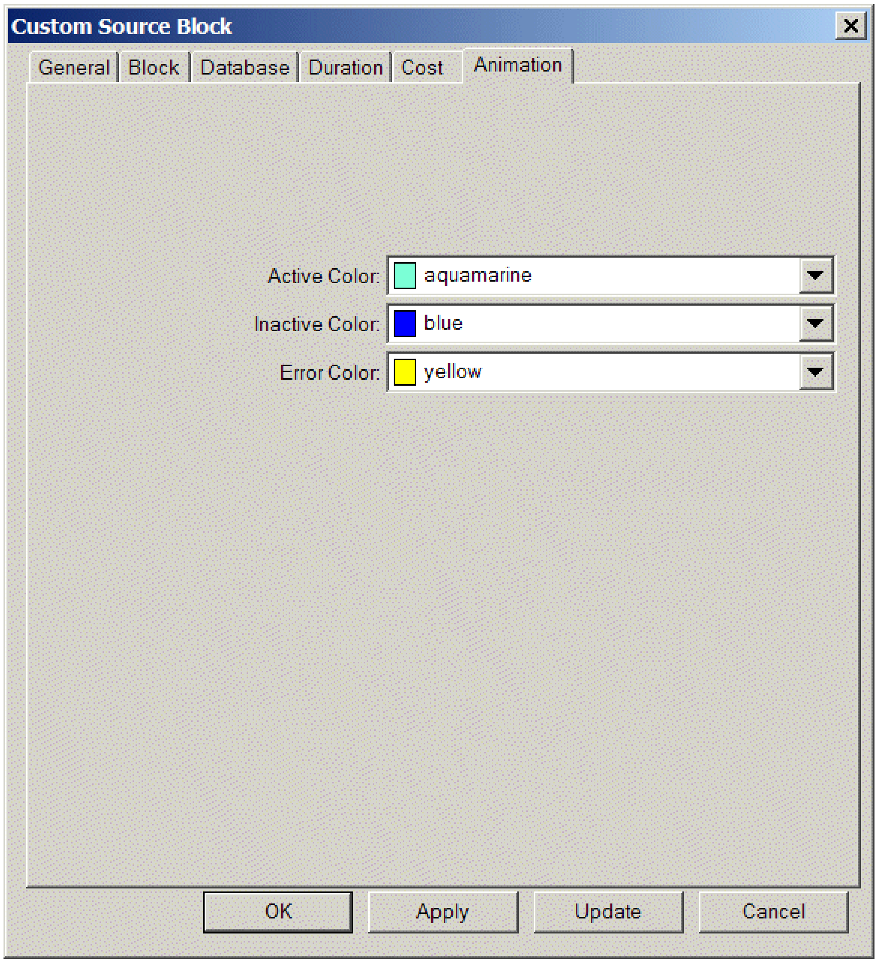
Note The colors you specify animate the icon region named flashing.

This example shows how you would edit the default color of a custom block:



Direct superior classes bpr-source

CUSTOM-SOURCE-BLOCK



Custom Source Block

Active Color: aquamarine

Inactive Color: blue

Error Color: yellow

Creating Custom Icon Regions

You might want to create and animate custom icon regions for your icon. To do this, edit the icon to add new regions, add new colors to a custom animation subtable for the block, and create a custom animation procedure that references these custom colors.

To animate custom icon regions:

1 Create a subclass of block and place it on your customization workspace.

For example:

Attribute	Value
class-name	custom-task-with-new-region
direct-superior-classes	bpr-task

2 Choose Edit Icon on the class definition and add new named regions, which you will animate.

For details, see the *ReThink User's Guide*.

3 Create a subclass of bpr-block-animation-subtable that specifies new color attributes of the custom icon regions.

For example, if you add a region called new-region, the class definition might look like this:

Attribute	Value
class-name	custom-task-animation-subtable
direct-superior-classes	bpr-block-animation-subtable
class-specific-attributes	new-region-active-color initially is red; new-region-inactive-color initially is black

4 Edit the class definition of the custom block to refer to the subclass of bpr-block-animation-subtable by specifying attribute-initializations for the class.

For example, the following specification overrides the default animation-subtable in the definition of the custom task:

Attribute	Value
class-name	custom-task-with-new-region
attribute-initializations	animation-subtable initially is an instance of a custom-task-animation-subtable

For information on why you use Attribute Initializations, see the note in Creating New Attributes for Subobjects.

5 Copy the bpr-animate-block procedure from the Block Animation Subtables workspace and place it on your customizations workspace.

This is the default procedure-name for the bpr-block-animation-subtable.

6 Change the name of the procedure and edit the procedure to animate the custom icon regions by using the custom color attributes.

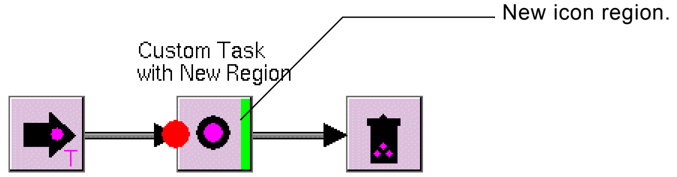
For example, the procedure might be named animate-task-block, and you might change the procedure to animate the new-region icon region rather than the flashing icon region by referring to the custom color attribute of the custom block class.

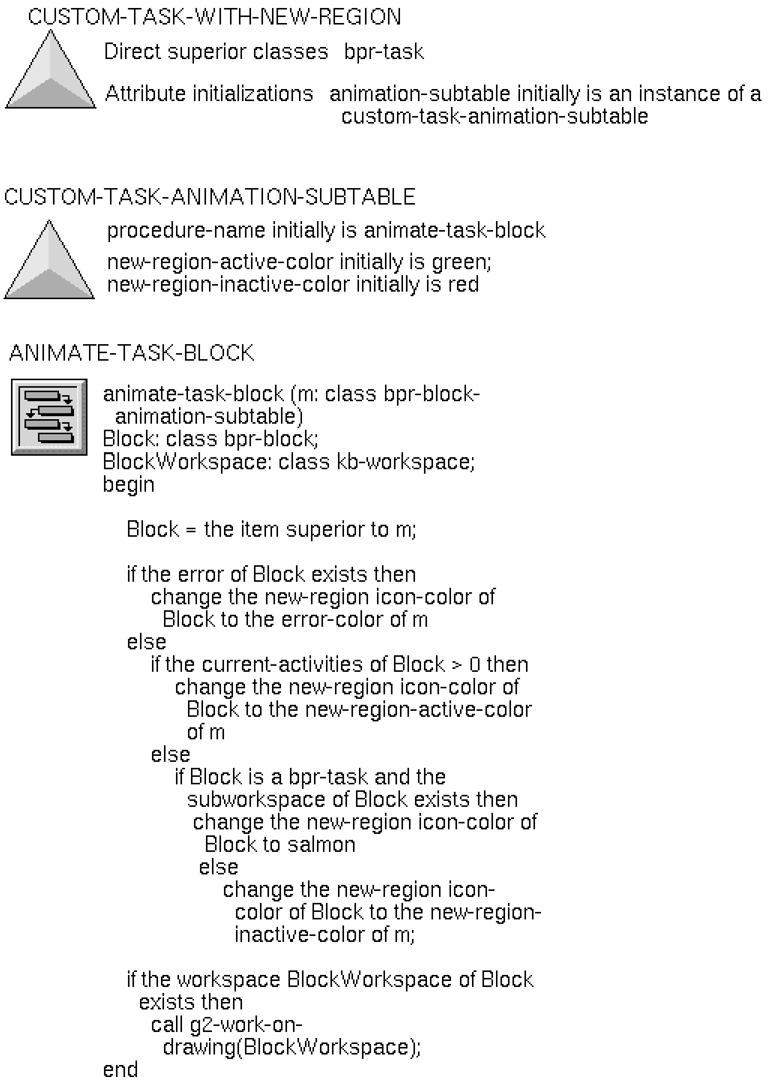
7 In the class definition of the subclass of bpr-block-animation-subtable, initialize the procedure-name to refer to the custom procedure by specifying attribute-initializations for the class.

For example, if the custom procedure is named animate-task-block, the specification would be:

Attribute	Value
class-name	custom-task-animation-subtable
attribute-initializations	procedure-name initially is animate-task-block

The following simple example shows a custom task block with a new region. When the work object arrives at the block, the region at the right of the icon turns green. The custom block and animation subtable class definitions show the attributes that you must configure.





CUSTOM-TASK-WITH-NEW-REGION

Direct superior classes bpr-task

Attribute initializations animation-subtable initially is an instance of a custom-task-animation-subtable

CUSTOM-TASK-ANIMATION-SUBTABLE

procedure-name initially is animate-task-block

new-region-active-color initially is green;
new-region-inactive-color initially is red

ANIMATE-TASK-BLOCK

```
animate-task-block (m: class bpr-block-
  animation-subtable)
Block: class bpr-block;
BlockWorkspace: class kb-workspace;
begin

    Block = the item superior to m;

    if the error of Block exists then
        change the new-region icon-color of
          Block to the error-color of m
    else
        if the current-activities of Block > 0 then
            change the new-region icon-color of
              Block to the new-region-active-color
              of m
        else
            if Block is a bpr-task and the
              subworkspace of Block exists then
               change the new-region icon-color of
                 Block to salmon
                else
                    change the new-region icon-
                      color of Block to the new-region-
                      inactive-color of m;

    if the workspace BlockWorkspace of Block
      exists then
        call g2-work-on-
          drawing(BlockWorkspace);
end
```

Customizing the Duration of a Block

ReThink provides numerous default procedures for the duration-subtable of blocks:

This procedure...	Computes duration based on...
bpr-fixed-duration	A fixed value with no variation.
bpr-random-normal-duration	A random normal distribution, which takes a mean and a standard deviation. This is the default procedure for all blocks except the Source block.
bpr-random-exponential-duration	A random exponential distribution, which takes a mean. This is the default procedure for a Source block.
bpr-random-uniform-duration	A random uniform distribution, which takes a minimum and a maximum.
bpr-random-triangular-duration	A random triangular distribution, which takes a minimum, a maximum, and a mode.
bpr-random-erlang-duration	A random Erlang distribution, which takes a mean and a number of samples.
bpr-random-weibull-duration	A random Weibull distribution, which takes a shape and scale.
bpr-random-lognormal-duration	A random lognormal distribution, which takes a mean and a standard deviation.
bpr-random-gamma-duration	A random gamma distribution, which takes an alpha and beta.
bpr-random-beta-duration	A random beta distribution, which takes a minimum, a maximum, and an alpha and beta.
bpr-file-duration	An input duration file.
bpr-work-object-duration	An attribute of a work object.
bpr-report-indexed-lookup-duration	An indexed report lookup.

This procedure...	Computes duration based on...
bpr-report-key-lookup-duration	A report lookup that uses attributes of a work object as search criteria for looking up durations in a report.
bpr-arrival-rate-input-duration	Uses an Arrival Rate Input Graph to determine duration.

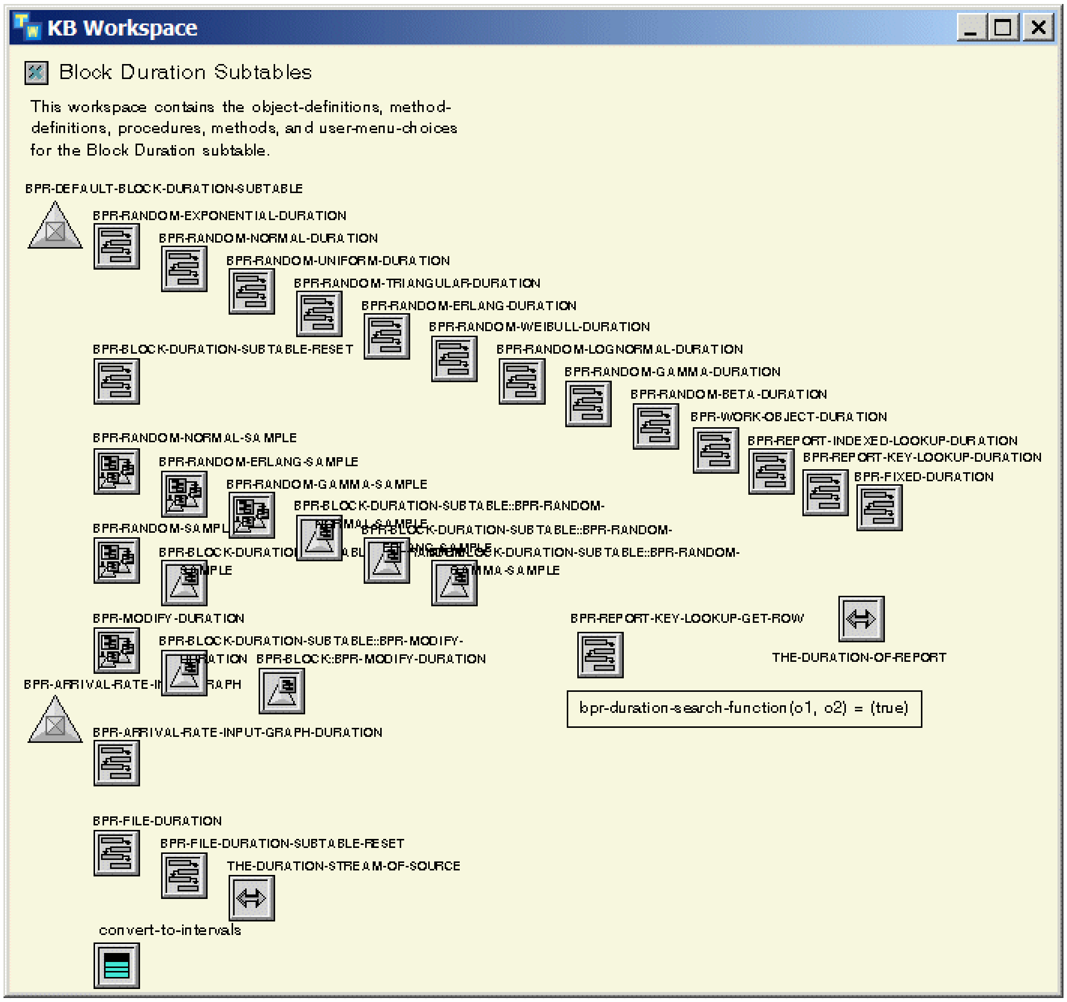
Displaying the Block Duration Subtables Workspace

The default duration procedures are located on the Block Duration Subtables workspace.

To display the default duration procedures for blocks:

➔ Display the Methods workspace, then choose Duration Subtables > Block Duration Subtables.

Here is the Block Duration Subtables workspace:



KB Workspace

Block Duration Subtables

This workspace contains the object-definitions, method-definitions, procedures, methods, and user-menu-choices for the Block Duration subtable.

BPR-DEFAULT-BLOCK-DURATION-SUBTABLE
BPR-RANDOM-EXPONENTIAL-DURATION
BPR-RANDOM-NORMAL-DURATION
BPR-RANDOM-UNIFORM-DURATION
BPR-RANDOM-TRIANGULAR-DURATION
BPR-RANDOM-ERLANG-DURATION
BPR-RANDOM-WEIBULL-DURATION
BPR-BLOCK-DURATION-SUBTABLE-RESET
BPR-RANDOM-LOGNORMAL-DURATION
BPR-RANDOM-GAMMA-DURATION
BPR-RANDOM-BETA-DURATION
BPR-WORK-OBJECT-DURATION
BPR-REPORT-INDEXED-LOOKUP-DURATION
BPR-REPORT-KEY-LOOKUP-DURATION
BPR-FIXED-DURATION
BPR-RANDOM-NORMAL-SAMPLE
BPR-RANDOM-ERLANG-SAMPLE
BPR-RANDOM-GAMMA-SAMPLE
BPR-BLOCK-DURATION-SUBTABLE::BPR-RANDOM-NORMAL-SAMPLE
BPR-BLOCK-DURATION-SUBTABLE::BPR-RANDOM-ERLANG-SAMPLE
BPR-BLOCK-DURATION-SUBTABLE::BPR-RANDOM-GAMMA-SAMPLE
BPR-RANDOM-SAMPL
BPR-BLOCK-DURATIO
SAMPLE
BPR-MODIFY-DURATION
BPR-BLOCK-DURATION-SUBTABLE::BPR-MODIFY-DURATION
BPR-BLOCK::BPR-MODIFY-DURATION
BPR-ARRIVAL-RATE-I
RAPH
BPR-ARRIVAL-RATE-INPUT-GRAPH-DURATION
BPR-REPORT-KEY-LOOKUP-GET-ROW
THE-DURATION-OF-REPORT
bpr-duration-search-function(o1, o2) = (true)
BPR-FILE-DURATION
BPR-FILE-DURATION-SUBTABLE-RESET
THE-DURATION-STREAM-OF-SOURCE
convert-to-intervals

Creating a Custom Block Duration Procedure

You might want to create a custom duration procedure for a block. For example, you might create a block that dynamically sets its duration based on an attribute of the work object, so that certain types of work objects take longer to process than others. For example, you might create a custom duration procedure that multiplies the duration by the number of line items in an order.

Note This capability is available in Modeler mode by using the Time per Unit Attribute attribute on the Duration tab of the properties dialog.

To create a custom block duration procedure:

1 Create a custom procedure, based on one of the existing duration procedures, which obtains an attribute from a work object and multiplies the duration calculations by the value of the attribute.

For example, you might create an attribute of a work object named number-of-line-items, which will impact the duration of the custom task.

2 Create a custom procedure that implements the custom duration, based on one of the existing duration procedures.

For example, you might create a procedure named custom-per-line-item-duration, which is based on the bpr-fixed-duration procedure and which obtains the value of number-of-line-items of each work object as it arrives at the block and multiplies the duration by the value of this attribute.

To do this, you must also configure the Mean of the block, which you do through the dialog.

3 Create a custom subclass of a block and create an instance of the class, or edit any existing ReThink block.

4 Display the properties dialog for the custom task block and click the Duration tab.

5 First, configure the Distribution Mode to be Fixed Distribution, configure the Mean, and click Apply.

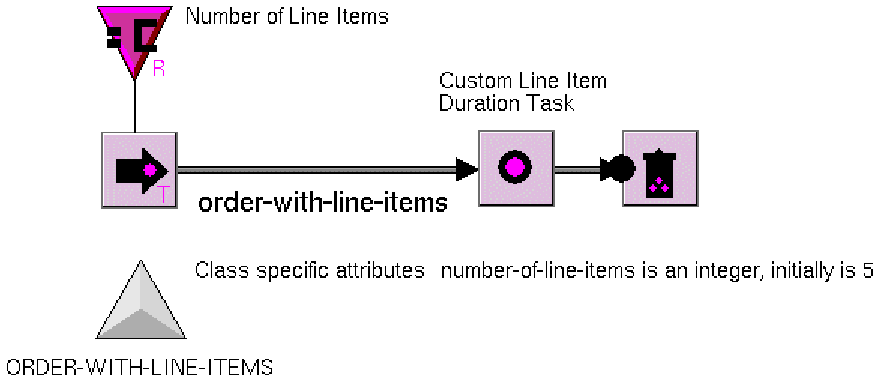
The block retains this value, which it will use in the custom duration procedure.

6 Configure the Distribution Mode to be Custom.

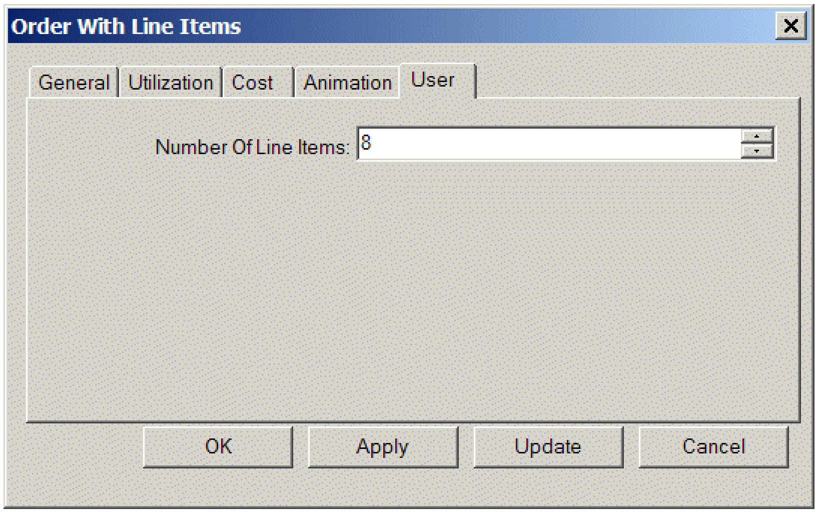
7 Configure the Procedure Name to be the custom procedure and click Apply.

For example, you would specify custom-per-line-item-duration as the procedure name.

This model uses a Change feed to supply random value into the number-of-line-items attribute of each order as it arrives at the Custom Task block. The custom block then computes the duration based on the value of the random attribute. The class definition declares number-of-line-items as an attribute of the order.

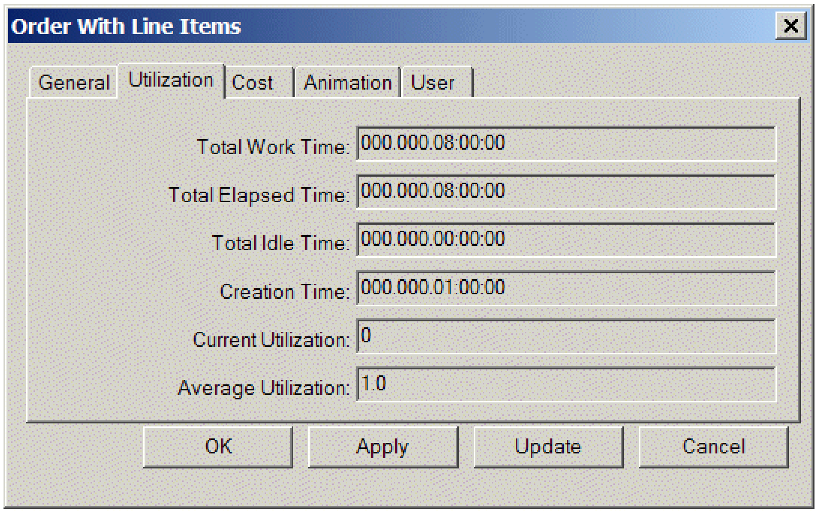


Here is the User tab for a work object where the number of line items is 8:



Number Of Line Items: 8

Here is the Utilization tab for the work object, which shows that the Total Work Time is 8 hours:



Total Work Time: 000.000.08:00:00

Total Elapsed Time: 000.000.08:00:00

Total Idle Time: 000.000.00:00:00

Creation Time: 000.000.01:00:00

Current Utilization: 0

Average Utilization: 1.0

Here is the procedure that implements the customization of the task:

CUSTOM-PER-LINE-ITEM-DURATION

```
custom-line-item-duration (Subtable: class bpr-block-duration-subtable, Activity: class
bpr-activity)
number-of-line-items: quantity;
WorkTime: quantity;
begin
{
This procedure implements the fixed distribution for block duration subtables. It
generates a sample based on the mean and then allows customization of the duration
based upon the input objects and/or the resources associated with the activity.
}
   WorkTime = round(the mean of Subtable);

   if WorkTime < 0 then
      WorkTime = 0; { make any negative values be zero }
{
Set the work and elapsed times.
}
   conclude that the work-time of Activity = WorkTime;
   conclude that the elapsed-time of Activity = WorkTime;
```

```
{Get the number of line items of the input work object}

   if there exists a order-with-line-items obj that is a-bpr-input-of-activity activity
     such that (the number-of-line-items of obj exists and the number-of-line-items of
     obj is a quantity) then
   begin
      number-of-line-items = the number-of-line-items of obj;
   end
   else
      post "Error";

   WorkTime = round((number-of-line-items * the mean of SubTable));

   call bpr-modify-duration(Subtable, Activity);
end
```

Customizing the Paths of a Block

By default, when work backs up in a process, ReThink places work objects in the path queue with the newest work objects at the end of the queue. When ReThink processes work objects from the queue, it processes the oldest work objects first, using a first-in first-out ordering (FIFO).

You can customize the procedure that ReThink uses to store work objects in the path queue when there are work backups. You customize this procedure for a particular path on a block. You can then copy the block to use it in your model, or you can place the block on a custom palette for general use.

When you customize the procedure for a particular path, you typically also edit the active and inactive colors of the path to distinguish it from the default paths.

To customize the queue behavior of a path:

1 Display the properties dialog for the path whose procedure you want to edit and click the Customize tab.

   The dialog has paths has an attribute named Enqueue Procedure Name.

   Internally, ReThink calls the procedure named bpr-enqueue-fifo, which is available for you to customize.

2 Display the Methods workspace and display the Other Methods workspace.

3 Copy the bpr-enqueue-fifo procedure and place it on your customization workspace.

4 Edit the procedure to obtain the desired behavior.

   For example, you might edit the procedure to place work objects at the beginning of the path queue, rather than at the end, to create a last-in first-out (LIFO) ordering.

5 In the properties dialog for the path, configure the Enqueue Procedure Name to refer to the custom procedure.

ReThink now removes the newest work objects from the path queue first.

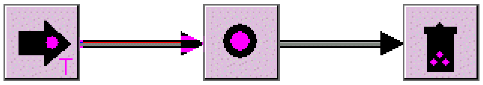
To customize the default colors of a path:

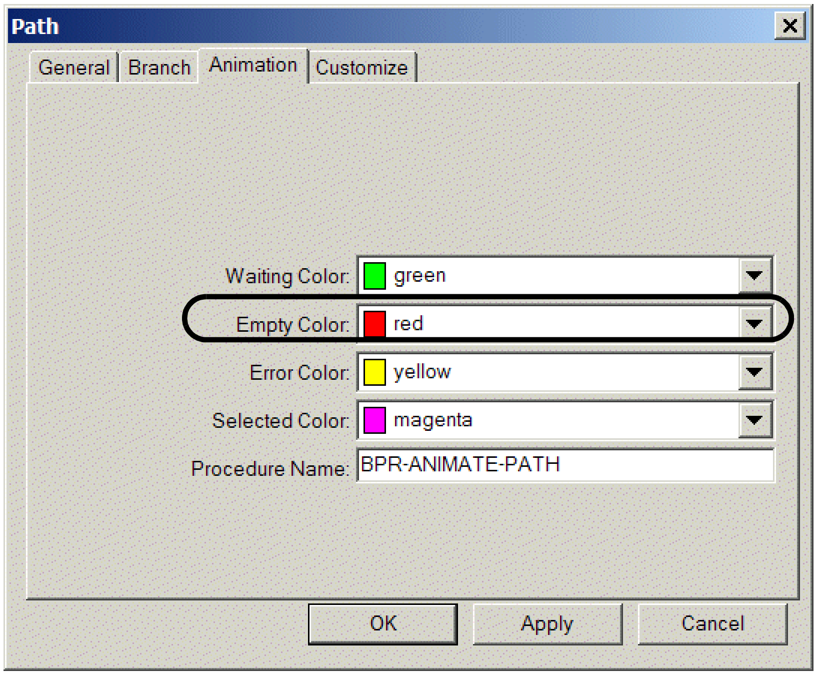
1 Display properties dialog for a path and click the Animation tab.

2 Edit one or more of the following path colors:

Path Color	Description
waiting-color	The path color when work objects back up in the queue or when the path is waiting for an input on another path, for example, when a block synchronizes its inputs.
empty-color	The path color when no work objects are traveling on the path.
error-color	The path color when the path is in an error state.
selected-color	The path color when you select the path for particular block operations, such as selecting the container input path of an Insert block.

This example shows a custom input path to a Task block that specifies a custom Empty Color:





Waiting Color: 🟩 green

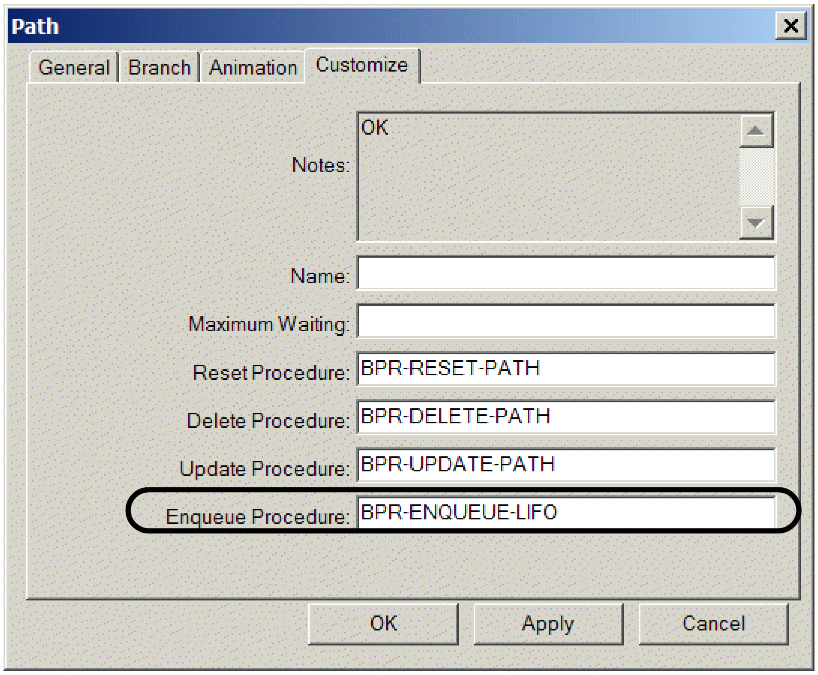
Empty Color: 🟥 red

Error Color: 🟨 yellow

Selected Color: 🟪 magenta

Procedure Name: BPR-ANIMATE-PATH

On the Customize tab, the path specifies a custom Enqueue Procedure:



Notes: OK

Name:

Maximum Waiting:

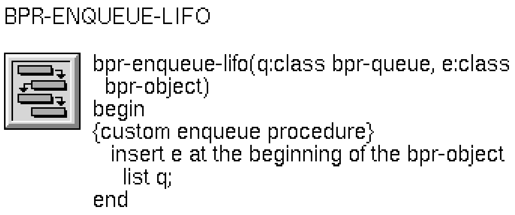
Reset Procedure: BPR-RESET-PATH

Delete Procedure: BPR-DELETE-PATH

Update Procedure: BPR-UPDATE-PATH

Enqueue Procedure: BPR-ENQUEUE-LIFO

The bpr-enqueue-lifo procedure inserts waiting work objects at the beginning of the path queue, rather than at the end:



```
BPR-ENQUEUE-LIFO

bpr-enqueue-lifo(q:class bpr-queue, e:class
  bpr-object)
begin
{custom enqueue procedure}
   insert e at the beginning of the bpr-object
     list q;
end
```

Customizing Specific Blocks

You can customize specific block procedures to change the behavior of a custom block. ReThink provides these specific default procedures on the various subworkspaces of the Block Definitions workspace.

Caution The subworkspaces of the Block Definitions workspace provide a number of additional procedures, user menu choices, rules, and relations. These items are available so you can understand how particular blocks work; we do not recommend that you customize them.

When you work with blocks in Developer mode, you can also customize certain attributes to change the default behavior of specific blocks.

The following sections describe the:

- General technique for customizing specific block procedures.
- Specific block behavior you can customize for each relevant block.

Only certain blocks have specific behaviors you can customize.

Customizing Specific Block Procedures

To customize a specific block procedure, create a custom block procedure, based on the default procedure and use this custom procedure in a custom subclass of the particular block.

For specific information about the procedures you can customize for specific blocks, see the following sections.

To customize specific block procedures:

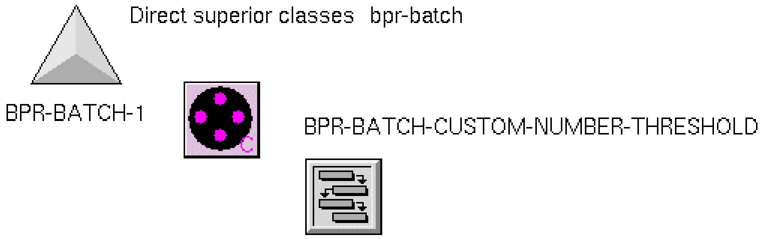
1. Create a block whose procedure you want to customize or create a custom subclass of the block you want to customize and create an instance of the class.
2. Display the Block Definitions workspace and choose the subworkspace of the block whose specific procedure you want to customize.

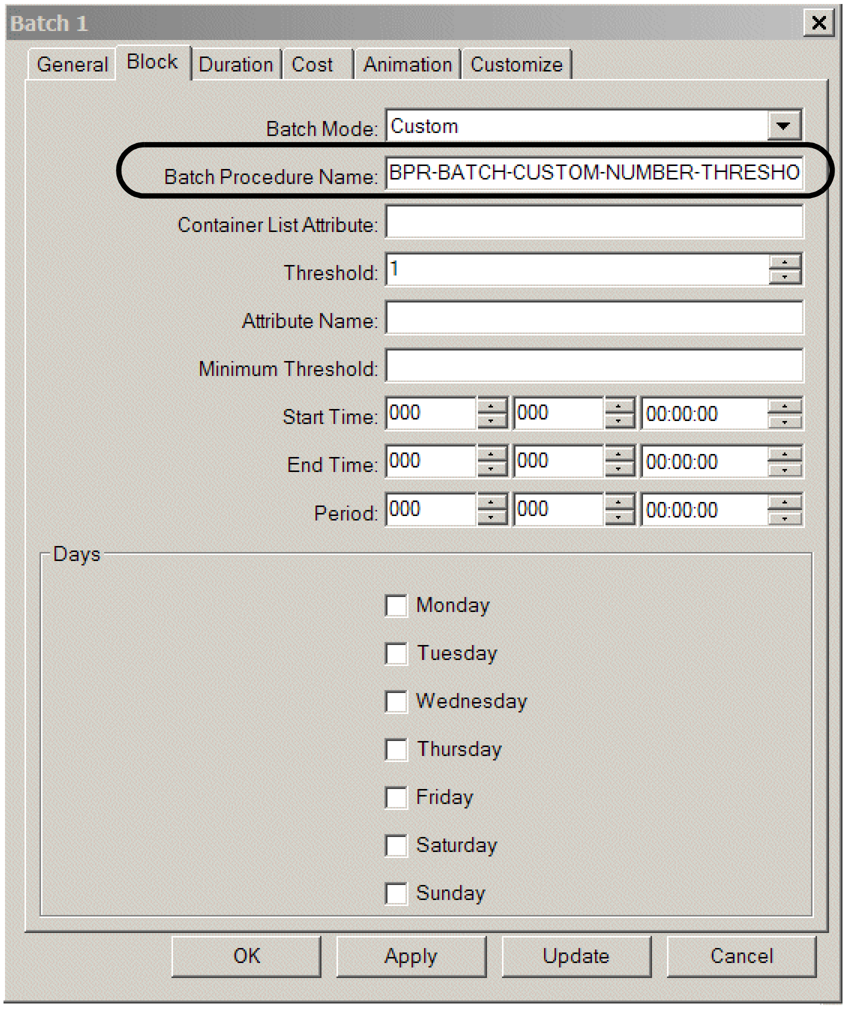
Note Only certain blocks have customizable procedures; the other blocks define only default stop methods.

3. Copy the specific block procedure you want to customize and place it on your customization workspace.
4. Click the Block tab and configure the mode attribute to be Custom.

   For example, for a Batch block, you would configure the Batch Mode to be Custom.
5. Edit the procedure name to create a new custom block procedure and edit the text of the procedure to obtain the desired behavior.
6. Edit the attribute that defines the procedure name in the properties dialog of the block.

For example, here is a custom Batch block that specifies a custom Batch Procedure Name:


Direct superior classes bpr-batch
BPR-BATCH-1
BPR-BATCH-CUSTOM-NUMBER-THRESHOLD



Batch Mode: Custom

Batch Procedure Name: BPR-BATCH-CUSTOM-NUMBER-THRESHO

Container List Attribute:

Threshold: 1

Attribute Name:

Minimum Threshold:

Start Time: 000 000 00:00:00

End Time: 000 000 00:00:00

Period: 000 000 00:00:00

Days

□ Monday
□ Tuesday
□ Wednesday
□ Thursday
□ Friday
□ Saturday
□ Sunday

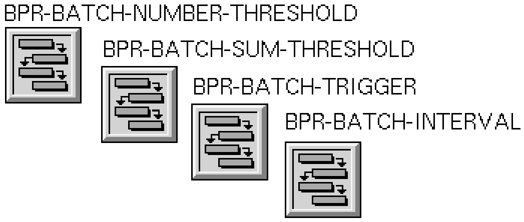
Customizing the Batch Block

Customizing the Batch Procedure

You can customize the procedure the Batch block uses to pass the batch or container by editing this attribute on the Block tab of the properties dialog when the Batch Mode is Custom:

Attribute	Description
Threshold Procedure Name	Specifies the procedure name that determines when the Batch block passes the batch. The default value depends on the Batch Mode.

The procedures you can customize are:



Determining Whether the Block Needs All Inputs

By default, if the Batch block has multiple input paths, it requires all inputs to arrive at the block before processing. You can configure this attribute on the Customize tab:

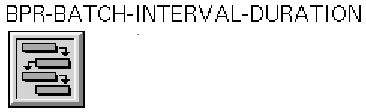
Attribute	Description
Needs All Inputs	Specifies whether the block requires inputs on all input paths before processing. By default, the block does not require all inputs.

Customizing the Batch Reset Procedure

The default Reset Procedure of the Batch block is bpr-reset-batch, which you can customize on the Customize tab of the properties dialog:

Customizing How the Block Computes Duration in Interval Mode

When Batch Mode is Interval, the Batch block computes its duration based on the mode-specific attributes, Start Time, End Time, Period, and Days. You can customize the bpr-batch-interval-duration procedure, which the Batch block uses to compute its duration when the Batch Mode is Interval:



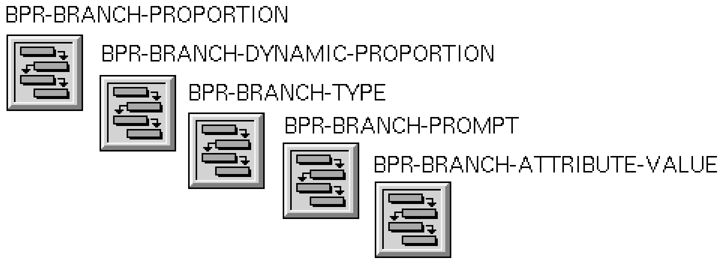
Customizing the Branch Block

Customizing the Branch Procedure

You can customize the procedure the Branch block uses to branch work objects by editing this attribute on the Block tab of the properties dialog when the Branch Mode is Custom:

Attribute	Description
Branch procedure Name	Specifies the procedure name that determines how the block branches work objects. The default value depends on the Branch Mode.

The procedures you can customize are:



Customizing the Copy Block

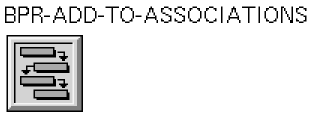
Copying Item-List Attributes and Their Items

By default, the Copy block copies the contents of all attributes that contain item lists and the items within those lists. You can customize this behavior by editing these attributes on the Block tab of the properties dialog:

Attribute	Description
Copy Item Lists	A truth-value that specifies whether the block should copy the contents of attributes of work objects that contain item-lists. The default value is true.
Copy Item List Items	A truth-value that specifies whether the block should copy the items within item-list attributes of work objects. The default value is true.

Adding to Associations

You can customize the procedure the Copy block uses when it adds a copy to an existing association by editing this procedure:



BPR-ADD-TO-ASSOCIATIONS

The block calls this procedure when the Add to Associations option is on.

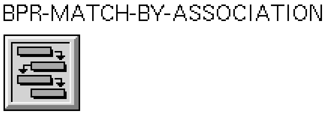
Customizing the Reconcile Block

Customizing the Match Procedure

You can customize the procedure the Reconcile block uses to determine how it reconciles objects by editing this attribute on the Block tab of the properties dialog when the Reconcile Mode is Custom:

Attribute	Description
Match Procedure Name	Determines the criteria the block uses to reconcile objects. The default value is bpr-match-by-association, which reconciles objects, based on an association name.

You can edit the following procedure to customize how the block reconciles work:



BPR-MATCH-BY-ASSOCIATION

Customizing the Reconcile Reset Procedure

The default Reset Procedure Name of the Reconcile block is bpr-reset-reconcile, which you can customize on the Customize tab of the properties dialog:

BPR-RESET-RECONCILE

Customizing the Remove Block

Customizing What the Remove Block Considers “Empty”

You can customize what the Remove blocks considers “empty” by editing the following attribute on the Block tab of the properties dialog:

Attribute	Description
Empty Breakpoint	Determines when the container object is considered empty and, thus, when the Remove block passes the container to the empty output path. The block looks at the contents of the item-list of the container when the container first arrives at the block. By default, if the item-list contains a single object upon arrival, it passes the input object to the empty container output path. The default is 1.

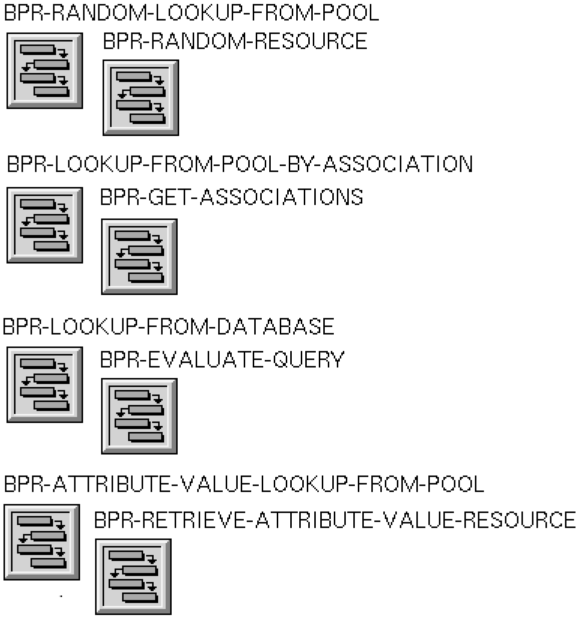
Customizing the Retrieve Block

Customizing the Retrieve Procedure

You can customize the procedure the Retrieve block uses to retrieve work objects from a pool or database by editing this attribute on the Block tab of the properties dialog when the Retrieve Mode is Custom:

Attribute	Description
Lookup Procedure Name	A procedure that determines how the block retrieves objects from a pool. The default value depends on the value of the Retrieve Mode.

The procedures you can customize are:



BPR-RANDOM-LOOKUP-FROM-POOL

BPR-RANDOM-RESOURCE

BPR-LOOKUP-FROM-POOL-BY-ASSOCIATION

BPR-GET-ASSOCIATIONS

BPR-LOOKUP-FROM-DATABASE

BPR-EVALUATE-QUERY

BPR-ATTRIBUTE-VALUE-LOOKUP-FROM-POOL

BPR-RETRIEVE-ATTRIBUTE-VALUE-RESOURCE

Copying Item-List Attributes and Their Items

By default, when the Retrieve Copy option is on, the Retrieve block copies the contents of all attributes that contain item lists and the items within those lists. You can customize this behavior by editing these attributes on the Block tab of the properties dialog:

Attribute	Description
Copy Item Lists	A truth-value that specifies whether the block should copy the contents of attributes of work objects that contain item-lists. The default value is true.
Copy Item List Items	A truth-value that specifies whether the block should copy the items within item-list attributes of work objects. The default value is true.

Customizing the Retrieve Reset Procedure

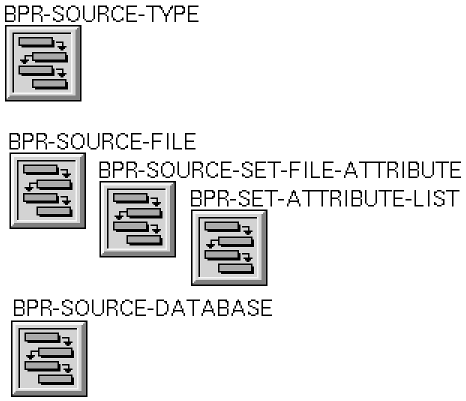
The default reset-procedure-name of the Reconcile block is bpr-retrieve-reset, which you can customize:

Customizing the Source Block

You can customize the procedure the Source block uses to generate work objects by editing this attribute on the Block tab of the properties dialog when the Source Mode is Custom:

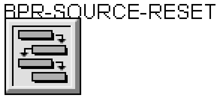
Attribute	Description
Source Procedure Name	Specifies the procedure the block uses to determine how it generates work objects. The default value depends on the Source Mode.

The procedures you can customize are:



Customizing the Source Reset Procedure

The default Reset Procedure Name of the Reconcile block is bpr-source-reset, which you can customize:



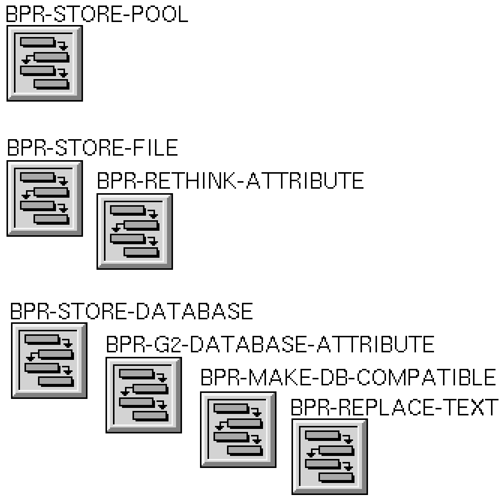
Customizing the Store Block

Customizing the Store Procedure

You can customize the procedure the Store Block uses to store objects in a pool by editing this attribute on the Block tab of the properties dialog when the Store Mode is Custom:

Attribute	Description
Store Procedure Name	Specifies the procedure that determines how and where the block stores objects. The default value depends on the Store Mode.

The procedures you can customize are:



For example, you could use the subworkspace of the pool to create a scatter plot of the objects, whose position in the chart is a function of its cycle time. Alternatively, you could use the subworkspace of the pool as an inventory by positioning each item vertically according to its type.

Customizing Database Mode

You can customize various aspects of database store mode by editing these attributes on the Database tab of the properties dialog when the Store Mode is Database:

Attribute	Description
Database Input Object Name	When you have a SQL query that include an expression in square brackets to be evaluated, specifies the name by which you reference the input work object. The default value is the symbol InputObject, which is case insensitive.

Attribute	Description
Database Quote String	Specifies the character that ReThink uses to specify a text string. The default value is ‘, which is the default character that the Microsoft Access database uses to specify a text string. You can change this default character depending on the database you are accessing.
Database Quote In Text String	Specifies the character that ReThink uses to specify an embedded quote character within a text string. The default value is ", which is the default character that the Microsoft Access database uses to specify an embedded quote character in a text string. You can change this default character depending on the database you are accessing.

Customizing the Store Reset Procedure

The default Reset Procedure Name of the Reconcile block is bpr-store-reset, which you can customize on the Customize tab of the properties dialog:

Customizing the Yield Block

Customizing the Yield Procedure

You can customize the procedure the Yield Block uses to compute the yield by editing this attribute on the Block tab of the properties dialog when the Yield Mode is Custom:

Attribute	Description
Yield Procedure Name	Specifies the procedure that determines how the block computes the yield when Yield Mode is Custom.

The procedure you can customize is:

BPR-YIELD-CUSTOM-PROCEDURE-NAME

The bpr-yield::bpr-stop-method describes the default behavior of the Yield block for each of the yield modes.

Common Customization Attributes of Blocks

The common customization attributes of blocks, which are visible on the Customize tab, are:

Attribute	Description
GFR UUID	The internal identification number of the object.
Notes	The current status of the block.
Name	A symbol that represents the name of the block. ReThink uses the Label attribute rather than the names attribute to identify blocks, to avoid naming conflicts.
Reset Procedure	The procedure name the block uses when the simulation is reset. The default value for most blocks is bpr-reset-block. See also Customizing Specific Blocks.
Delete Procedure	The procedure name the block uses when it is deleted. The default value is bpr-delete-block.
Update Procedure	The procedure name the block uses when the Update button is clicked. The default value is bpr-update-block.
Animation Subtable	Subobject that specifies the default colors the block uses when it is in an active, inactive, or error state. The default value is an instance of a bpr-block-animation-subtable. This attribute is available through the table only.

Attribute	Description
Duration Subtable	Subobject that specifies the timing parameters of the block, and computes summary timing statistics. The default value is an instance of a bpr-default-block-duration-subtable. This attribute is available through the table only.
Cost Subtable	Subobject that specifies how the block computes cost statistics. The default value is an instance of a bpr-block-cost-subtable. This attribute is available through the table only.

For additional information, see:

- *G2 Reference Manual*
- Customizing the Default Behavior of Blocks or Instruments.
- Editing Subobjects.

Common Attributes of Animation Subtable

The customization attributes of the animation-subtable of a block, which are visible on the Animation tab, are:

Attribute	Description
Reset Procedure Name	The name of the procedure the block uses to reset the animation subtable when the simulation resets. The default value is bpr-animate-block.
Procedure Name	The default animation procedure for the block. The default value is bpr-animate-block.

For additional information, see the references in this table:

For information on...	See...
Customizing the default colors of a block	Editing the Default Colors of a Block.
Customizing the default animation procedure of a block	Creating Custom Icon Regions.

Common Attributes of Duration Subtable

The customization attributes of the duration-subtable of a block, which are visible on the Duration tab, are:

Attribute	Description
Distribution Mode	Specifies how the block computes duration. When the value is Custom, you configure the Procedure Name to be the name of a custom procedure for computing block duration. For details, see Customizing the Duration of a Block.
Procedure Name	The name of the procedure the block uses to compute duration. The default value for all blocks except the Source block is bpr-random-normal-duration. The default value for the Source block is bpr-random-exponential-duration. In the duration-subtable, this attribute is called procedure-name.
Duration Reset Procedure Name	The name of the procedure the block uses to reset the duration subtable when the simulation resets. The default value is bpr-block-duration-subtable-reset.

For an example of how to customize the duration of a block, see Customizing the Duration of a Block.

Common Attributes of Cost Subtable

The customization attributes of the cost-subtable of a block, which are visible on the Cost tab, are:

Attribute	Description
Cost Reset Procedure Name	The name of the procedure the block uses to reset the cost subtable when the simulation resets. The default value is bpr-reset-cost-subtable.
Cost Procedure Name	The name of the procedure the block uses to compute Total Cost. The default value is bpr-block-cost.

Customizing Instruments

Provides specific descriptions and examples of how to customize instruments.

Introduction

You can create custom feeds and probes that have the behavior you want. For example, you might want a custom feed that supplies a random value to an attribute, or you might want a custom probe that signals an alarm when the value of an attribute exceeds a threshold.

You base the definition of the custom instrument on the definition of one of the existing instruments.

You can customize the following aspects of any instrument:

- Its characteristics by adding new attributes.
- Its behavior by modifying its default stop method.
- Its appearance when it changes from active to inactive by modifying its default colors and animation procedure.
- The behavior of the connection when you move the attached instrument.

To perform any of these customizations, first create a subclass of the instrument you want to customize, then customize the subclass.

For general information on how to customize, see How to Customize ReThink.

Creating a Custom Feed

Suppose you want to create a custom feed that supplies a random value to the attribute of a work object.

Note This behavior is available in Modeler mode by using a Change feed with the Change Mode attribute set to Random.

For general information on how to customize the behavior of instruments, see Customizing the Default Behavior of Blocks or Instruments.

To create a custom feed that supplies a random value to a work object:

1 Create a subclass of bpr-change-feed, the feed subclass most similar to that of the custom feed.

2 Edit the class-name to be a name that reflects the behavior of the new custom feed, such as randomize-feed.

3 Because the feed will store a random value between a given minimum value and a given maximum value, define two class-specific attributes to store these values.

The class definition for the custom feed looks like this:

Attribute	Value
class-name	randomize-feed
direct-superior-classes	bpr-change-feed
class-specific-attributes	minimum-allowable; maximum-allowable

The class-specific attributes appear on the User Tab of the custom feed.

4 To define the behavior of the custom feed, copy the bpr-change-feed:: bpr-stop-method method, the default stop method of the Change feed, and modify the method, as follows:

a Edit the first argument to the method to refer to the custom subclass.

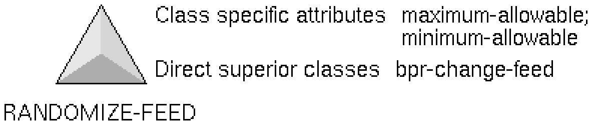
b Delete the existing body and local names of the method.

c Add a local name for the random value.

d Edit the custom portion of the method to compute a random value based on the maximum-allowable and minimum-allowable of the custom feed.

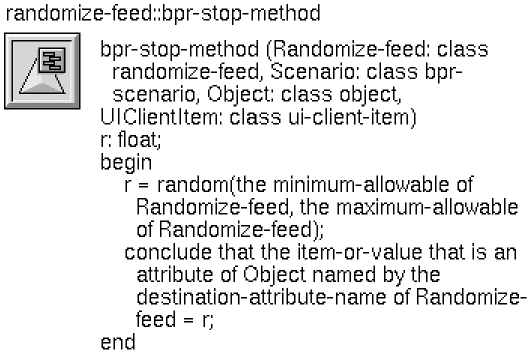
e Conclude that the destination-attribute-name of the Randomize feed is the random value the method computes.

Here is the class definition and stop method for the randomize-feed, which computes a random value and then concludes a value for the destination-attribute-name of the feed:



Class specific attributes	maximum-allowable; minimum-allowable
Direct superior classes	bpr-change-feed

RANDOMIZE-FEED



randomize-feed::bpr-stop-method

```
bpr-stop-method (Randomize-feed: class
  randomize-feed, Scenario: class bpr-
  scenario, Object: class object,
UIClientItem: class ui-client-item)
r: float;
begin
    r = random(the minimum-allowable of
      Randomize-feed, the maximum-allowable
      of Randomize-feed);
    conclude that the item-or-value that is an
      attribute of Object named by the
      destination-attribute-name of Randomize-
      feed = r;
end
```

To test the custom feed:

1 Create a model with a Source block, Task block, and Sink block.

2 Create a new class of work object with an attribute, such as weight, which the feed updates.

The class definition looks like this:

Attribute	Value
class-name	box
direct-superior-classes	bpr-object
class-specific-attributes	weight

3 Configure the path type of the output path of the Source block to be box.

4 Attach an instance of randomize-feed to the Task block.

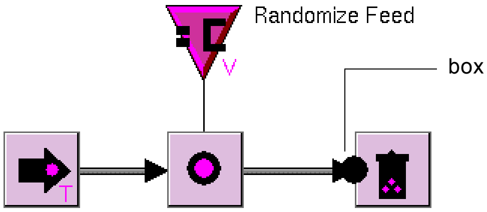
5 On the User tab of the custom feed, configure the Minimum Allowable and Maximum Allowable attributes.

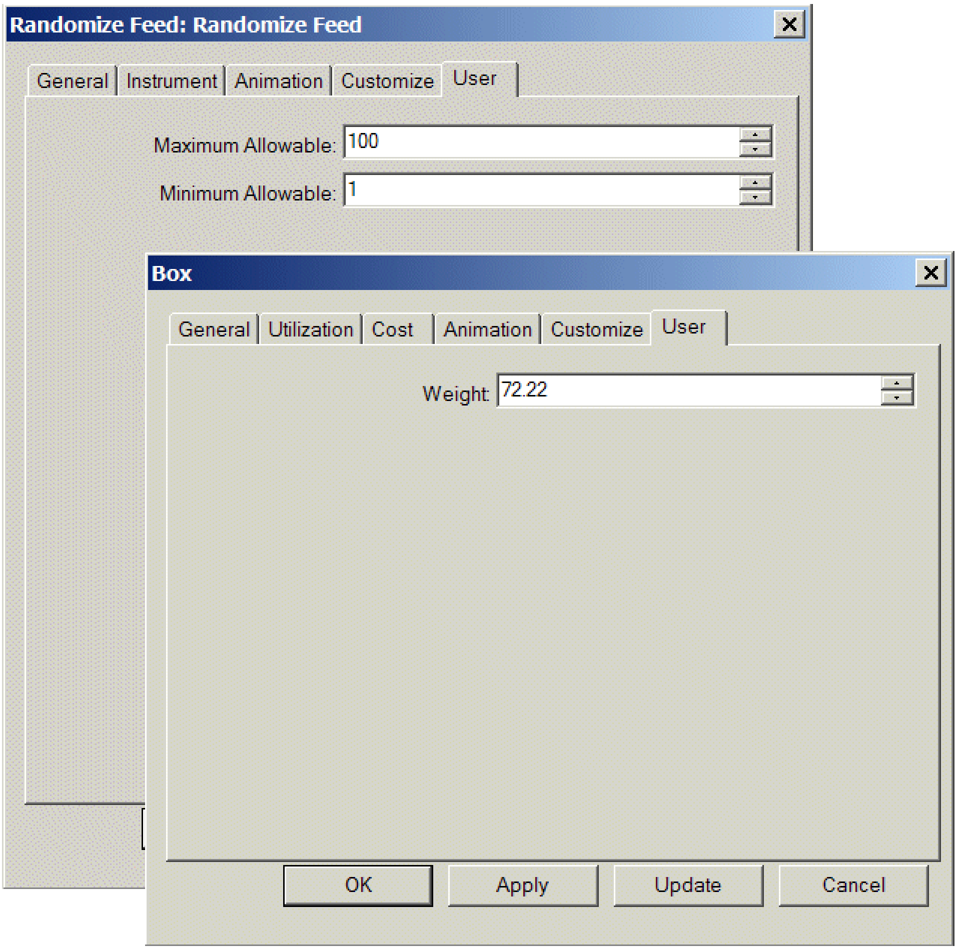
6 Assign weight as the value of the Destination Attribute Name of the feed.

7 Configure the Apply to Class Name attribute of the feed to be box.

When the simulation runs, the feed updates the weight attribute of the work object by using a random value between the maximum-value and minimum-value of the feed.

Here is a running model that shows how the Randomize Feed updates the weight attribute of the work object:




Randomize Feed: Randomize Feed

Maximum Allowable: 100

Minimum Allowable: 1

Box

Weight: 72.22

Creating a Custom Probe

Suppose you want to create a custom probe that signals an alarm when a value exceeds a threshold. When the value exceeds a threshold, the probe pauses the simulation, displays a message to the user, and displays an arrow next to the probe. This probe is similar to a Sample Value probe because it must obtain a value from the model.

For general information on how to customize the behavior of instruments, see Customizing the Default Behavior of Blocks or Instruments.

To create a probe that signals an alarm when a value exceeds a threshold:

1 Create a subclass of bpr-sample-probe, the probe subclass most similar to the custom probe.

2 Edit the class-name to be a name that reflects the behavior of the new probe, such as alarm-probe.

3 Assign the probe an attribute to store a threshold value.

The class definition looks like this:

Attribute	Value
class-name	alarm-probe
direct-superior-classes	bpr-sample-probe
class-specific-attributes	threshold initially is 1.0

The class-specific attribute appears on the User tab of the properties dialog for the custom probe.

4 To define the behavior of the custom probe, copy the bpr-sample-probe:: bpr-stop-method, the default stop method for a Sample Value probe, and modify the method:

a Edit the first argument to the method to refer to the custom subclass.

b Delete the existing body of the method.

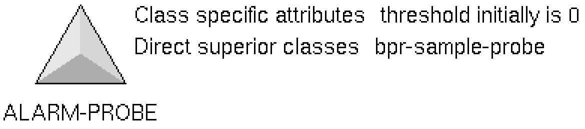
c Add a call next method statement to call the default stop method for the Sample Value probe at the beginning of the custom method.

d Edit the procedure to check if the probed value exceeds the threshold value, and, if so, pause the simulation, display a message to the user, and display an arrow next to the probe.

Tip To pause the simulation and display an arrow next to the probe, use the ReThink API procedures bpr-pause and bpr-indicate.

For information on these procedures, see Application Programmer's Interface.

Here is the class definitions and stop method for the alarm-probe:



Class specific attributes threshold initially is 0
Direct superior classes bpr-sample-probe

ALARM-PROBE

alarm-probe::bpr-stop-method

Here is the text of the bpr-stop-method:

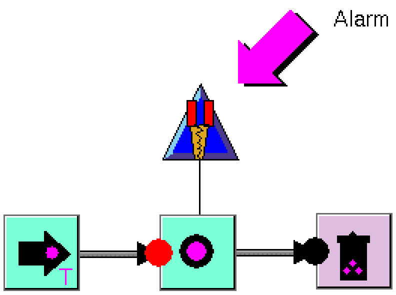
```
bpr-stop-method (alarm-probe: class alarm-probe, scenario: class bpr-scenario,
object: class object, ui-client-item: class ui-client-item)
begin
   call next method;
   if the sample-value of alarm-probe > the threshold of alarm-probe then
      begin
         call bpr-pause(scenario);
         inform the operator for the next 10 seconds that "Alarm: [the sample-value of
               alarm-probe] > [the threshold of alarm-probe]";
         start bpr-indicate(alarm-probe, "Alarm", ui-client-item);
      end;
end
```

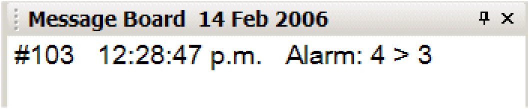
To test the custom probe:

1. Create a model with a Source block, Task block, and Sink block.
2. Attach the alarm probe to the Task block.

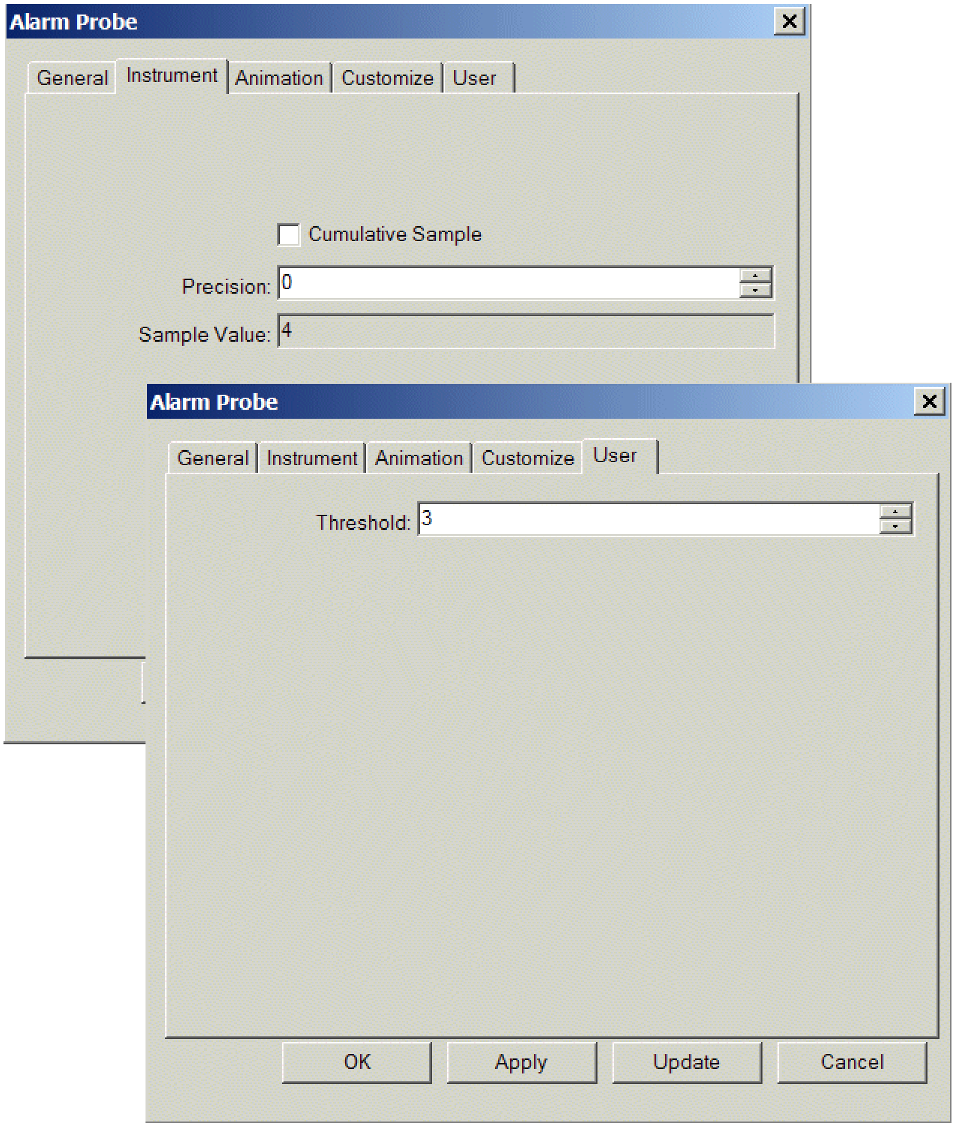
   You can test the alarm by probing the current-activities of the Task block. The alarm signals when the number of concurrent activities of the block exceeds the threshold.
3. On the User tab of the custom probe, configure the Threshold, for example, 3.
4. Configure the Source Attribute Name of the probe to be current-activities.
5. Configure the Apply to Class Name attribute of the probe to be bpr-task.
6. Configure the duration of the Source block to be 1 hour.
7. Configure the duration of the Task block to be 3 hours.
8. Run the simulation.

When the value the probe receives exceeds the threshold, the simulation pauses, and ReThink notifies the operator and displays an arrow next to the probe:




Message Board 14 Feb 2006

#103 12:28:47 p.m. Alarm: 4 > 3


Alarm Probe

☐ Cumulative Sample

Precision: 0

Sample Value: 4

Alarm Probe

Threshold: 3

Common Customization Attributes of Instruments

The common customization attributes of instruments, which are visible on the Customize tab, are:

Attribute	Description
GFR UUID	The internal identification number of the object.
Notes	The current status of the instrument.
Name	A symbol that represents the internal name of the instrument. ReThink uses the label attribute rather than the names attribute to identify instruments and avoid naming conflicts between ReThink models.
Reset Procedure	The procedure name the instrument uses when the simulation is reset. The default value is bpr-reset-instrument.
Animation Subtable	Subobject that specifies the default colors the instrument uses when it is in an active, inactive, or error state. The default value is an instance of a bpr-probe-animation-subtable or a bpr-feed-animation-subtable, depending on the type of instrument. This attribute is available through the table only.

For additional information, see:

- *G2 Reference Manual*.
- Customizing the Default Behavior of Blocks or Instruments.
- Editing Subobjects

Common Attributes of Animation Subtable

The customization attribute of the animation-subtable of an instrument, which is visible on the Animation tab, is:

Attribute	Description
Procedure Name	The default animation procedure for the instrument. The default value is bpr-animate-instrument.

Customizing the default colors of instruments is similar to customizing the default colors of blocks. Similarly, customizing the animation procedure of instruments is similar to customizing the animation procedure of blocks.

For additional information, see:

- Editing the Default Colors of a Block.
- Creating Custom Icon Regions.

Common Attributes of Instrument Paths

By default, the paths connected to instruments automatically reconfigure themselves whenever you move the connected instrument. You can customize this behavior by editing this attribute:

Attribute	Description
Redraw Connection	Set to false to cause instrument paths not to reconfigure themselves along the object when you move the connected instrument. The default value is true.

Customizing Specific Instruments

You can customize specific instrument procedures to change the behavior of a custom instrument. ReThink provides these specific default procedures on the various subworkspaces of the Instrument Definitions workspace.

When you work with instruments in Developer mode, you can also customize certain attributes to change the default behavior of specific instruments.

The following sections describe the specific behavior you can customize for each relevant instrument. Only certain instruments have specific behaviors you can customize.

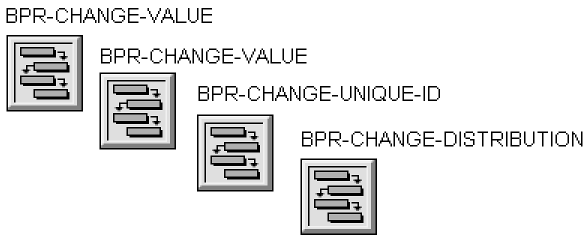
For information about customizing specific instrument procedures, see Customizing the Default Behavior of Blocks or Instruments.

Customizing the Change Feed

You can customize the procedure the Change feed uses to generate new values by configuring this attribute on the Instrument tab of the properties dialog when the Change Mode is Custom:

Attribute	Description
Change Procedure Name	Specifies the procedure the instrument uses to generate values. The default value depends on the Source Mode.

The procedures you can customize are:



Customizing the Copy Attributes Feed and Probe

The Copy Attributes feed and Copy Attributes probe define the list-of-operations attribute, which specifies a sequence of operations to perform between the source and target objects. The bpr-copy-attribute-feed::bpr-stop-method and bpr-copy-attribute-probe::bpr-stop-method refer to this attribute when they call the bpr-copy-list-of-attributes API to copy the attributes. For details, see bpr-copy-list-of-attributes.

Customizing Resources and Work Objects

Provides specific descriptions and examples of how to customize resources, work objects, Resource Managers, and surrogates.

Introduction

You can customize resources, work objects, resource managers, and surrogates by:

- Customizing the colors and animation behavior of resources when they are allocated and deallocated.
- Customizing the colors and animation behavior of work objects when they are active and inactive.
- Customizing how a resource or work object computes duration and utilization statistics.
- Customizing how a resource or work object computes total cost.
- Customizing how resource managers choose which resources to allocate to an activity.

- Customizing how resource managers determine which block allocates a resource first when multiple blocks are waiting for the same resource.
- Animating the colors and animation behaviors of a surrogate.

To customize a resource or work object, create a subclass of the object you are customizing and customize the subclass. To customize a resource manager or surrogate, customize a particular instance in the model.

For information on...	See...
General information on how to customize	How to Customize ReThink.
Customizing the duration and cost of resources	Editing Subobjects.
Customizing the duration, animation, and cost of work objects	Customizing Subobjects of Work Objects.
Summary information on how to customize Resource Managers	Customizing Resource Managers.
Summary information on how to customize surrogates	Customizing Surrogates.

Customizing Resource Animation

You can change the colors, appearance, and behavior of resources when the model allocates and deallocates them by customizing resource animation. To do this, you can either:

- Animate the flashing region of the icon.
- Add new regions to the icon and edit the animation procedure to animate the new icon regions.

Note ReThink generates an error when you run the model if the resource you are customizing does not have an icon region named flashing.

You use the same techniques to customize the animation of work objects and surrogates.

For example, you might want to move truck resources around a map when the model allocates them, or you might want to animate a person resource or a machine resource when it is active.

To animate a custom icon region of a resource:

- Define appropriate icon regions.
- Copy bpr-animate-object, the default procedure that controls resource animation.
- Edit the procedure to change the region colors as desired.
- Specify the custom animation procedure in the resource's animation subtable.

For additional information, see the references in this table:

For information on...	See...
How to edit the default animation procedure of a resource	Displaying Default Subobject Procedures.
How to edit the default colors of a resource	Editing Color Attributes of Animation Subobjects.
How to add new color attributes to the animation subtable of a resource	Creating New Attributes for Subobjects
Customizing the animation of work objects	Customizing Subobjects of Work Objects

For example, suppose you have a worker resource whose icon is a face. The face has a smile region and a frown region. The worker is happy when it is working and unhappy when it is idle. Therefore, you want the smile region to become visible when the resource is allocated, and you want the frown region to become visible when the resource is deallocated.

To animate a resource when it is allocated:

1 Define a subclass of bpr-resource named worker with class-specific attributes, as follows:

```
mouth-color initially is black
```

2 Edit the icon to create the following icon regions:

a A region named smile that contains a representation of a smile.

Its color should be the inactive-color of your resource, for example, gold.

b A region named frown that contains a representation of a frown.

  Its color should be any color, such as black, that is visible when the region sits on top of a region that contains the active-color of the resource.

c A region named flashing that contains a solid, round face.

  Its color should be the inactive-color of your resource.

For details, see the *ReThink User's Guide*.

3 Copy the bpr-animate-object procedure from the Object Animation Subtable workspace, which is the default procedure-name of the bpr-object-animation-subtable.

4 Edit the name of the procedure to be animate-worker.

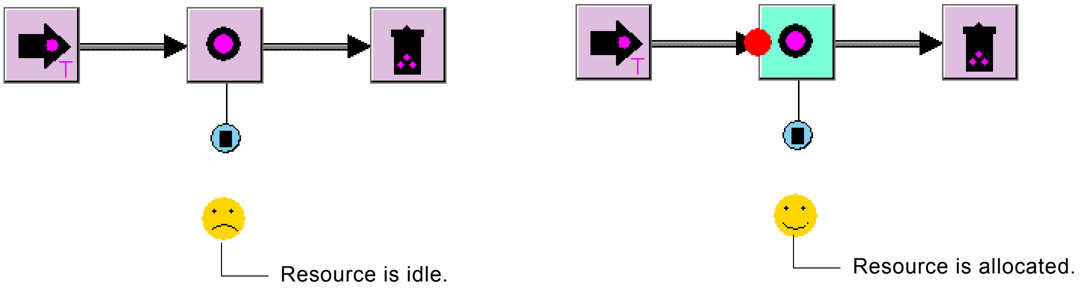
5 Edit the procedure to animate the frown and smile regions, using the appropriate colors.

6 Edit the procedure-name in the animation-subtable of the worker resource to refer to the custom animation procedure, animate-worker.

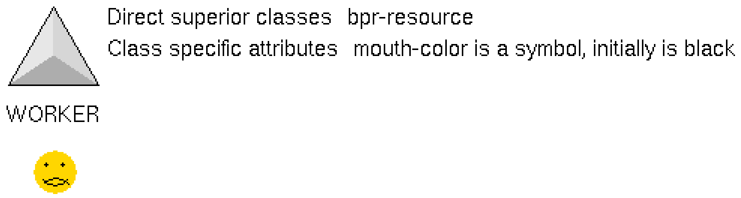
7 Edit the active-color and inactive-color in the animation-subtable of the worker resource to be the default color of the smile icon region, for example, gold.

8 Edit the label attribute of the resource to be worker.

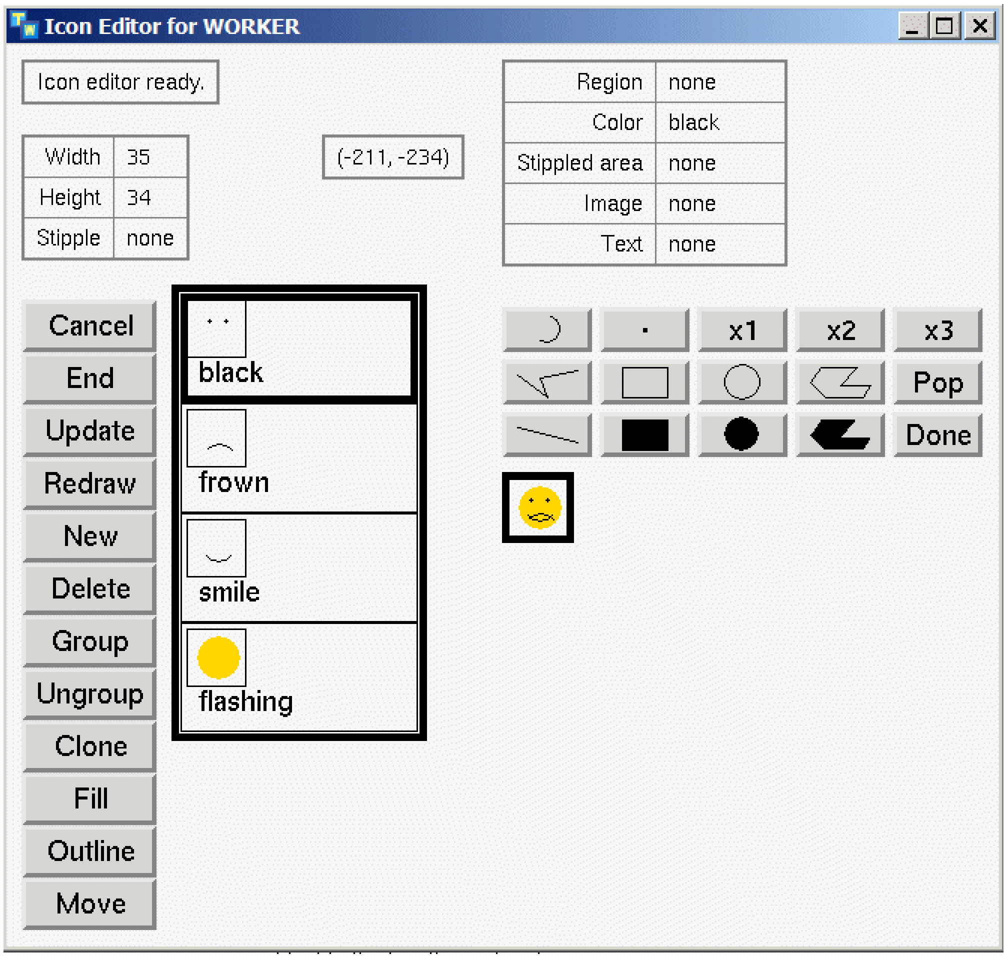
Here is a simple model that shows how the worker resource looks when it is idle and when it is allocated:



Here is the class definition and icon of the worker resource:



Direct superior classes	bpr-resource
Class specific attributes	mouth-color is a symbol, initially is black



Icon Editor for WORKER

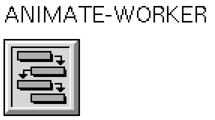
Icon editor ready.

Width	35
Height	34
Stipple	none

(-211, -234)

Region	none
Color	black
Stippled area	none
Image	none
Text	none

Here is the animate-worker procedure:

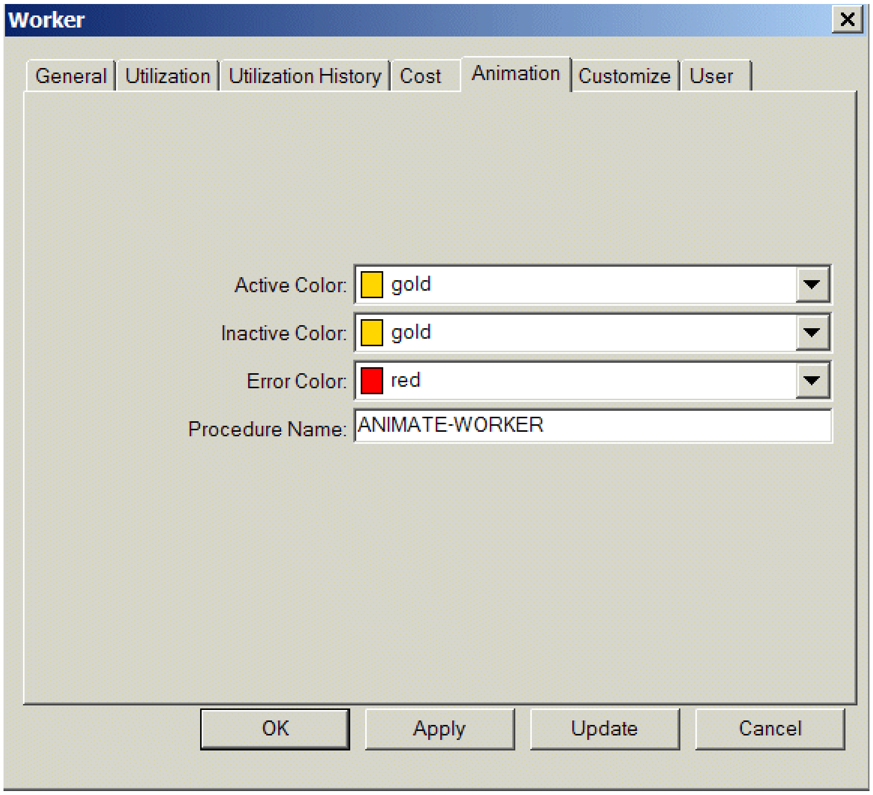


```
animate-worker(m: class bpr-object-animation-subtable)
object: class bpr-object;
ws: class kb-workspace;
begin
   object = the item superior to m;

   if the error of object exists then
      change the flashing icon-color of object to the error-color of m
   else
      if the total-starts of object > the total-stops of object then
         begin
            change the smile icon-color of object to the mouth-color of object;
            change the frown icon-color of object to the inactive-color of m;
         end
      else
         begin
            change the flashing icon-color of object to the inactive-color of m;
            change the smile icon-color of object to the inactive-color of m;
            change the frown icon-color of object to the mouth-color of object;
         end;

   if the workspace ws of object exists then
      call g2-work-on-drawing(ws);
end
```

Here is the Animation tab of the properties dialog of the worker resource:



Active Color: 🟨 gold

Inactive Color: 🟨 gold

Error Color: 🟥 red

Procedure Name: ANIMATE-WORKER

Customizing How Resource Managers Allocate Resources

A Resource Manager allocates and deallocates resources to and from an activity. Each time a block processes a work object, the attached manager:

- Checks to see if a resource is available; if so, it allocates a resource for the duration of the activity, according to the value of the Choose Resource attribute on the Allocate tab of the Resource Manager dialog.
- If no resources are available, queues the block for processing when a resource becomes available.
- If more than one block requires the same resource, keeps track of the blocks that are waiting for the resources and allocates the resource to each block that is waiting, according to the value of the Choose Manager attribute on the Deallocate tab of the Resource Manager dialog.

Resource Managers specify the following two attributes and default values to determine how it allocates and deallocates resources:

Attribute	Default Value	Description
Choose Resource	bpr-random-available-resource	Checks to see if there are any idle resources: • If so, it allocates the first available resource to the activity and randomly reorders the resources. • If not, it returns the symbol none. In the table, this attribute is called choose-resource-procedure-name.
Choose Manager	bpr-random-waiting-block	Queues the activities that are waiting for resources by inserting the block in the blocks-waiting list of the pool. The procedure then allocates a resource to the first block in the list and randomly reorders the waiting blocks. In the table, this attribute is called sequence-block-procedure-name.

These two attributes have different values, depending on the values of the Choose Resource and Choose Manager attributes of the Resource Manager:

Attribute	Possible Values	Descriptions
Choose Resource	bpr-lowest-cost-available-resource	Chooses the resource with the lowest total-cost.
	bpr-lowest-utilization-available-resource	Chooses the resource with the lowest average-utilization.
	bpr-highest-priority-available-resource	Chooses the resource whose resource-priority is the smallest number.
	bpr-lowest-priority-available-resource	Chooses the resource whose resource-priority is the largest number.

Attribute	Possible Values	Descriptions
Choose Manager	bpr-highest-priority-waiting-block	Allocates a resource to the block in the blocks-waiting list whose manager-priority is the largest number.
	bpr-lowest-priority-waiting-block	Allocates a resource to the block in the blocks-waiting list whose manager-priority is the smallest number.

For a summary of how to customize Resource Managers, see Customizing Resource Managers.

Choosing the Resource that has Worked the Shortest Amount of Time

You might want to choose the resource that has worked the shortest amount of time or the resource that has been allocated the fewest number times.

To customize which resource the model allocates to an activity, you customize the choose-resource-procedure-name of the Resource Manager. The following example shows how to customize the procedure that chooses resources from a pool to choose the resource whose total-work-time is the smallest number.

To choose the resource that has been worked the shortest amount of time:

1 Copy the bpr-lowest-cost-available-resource procedure from the Resource Methods workspace.

This procedure is the procedure that is most similar to the one you will create.

2 Edit the procedure name to be bpr-most-idle-resource.

The procedure takes three arguments: the activity that requires the resource, a usage object, and a resource.

A **usage object** is an interim object that computes how much of a resource is used by an activity. The usage object is related to the Resource Manager in the same way that an activity is related to a block.

3 Edit the procedure to choose the resource whose total-work-time is the smallest number.

Here is the custom procedure:

```
bpr-most-idle-resource(activity:class bpr-activity, usage:class bpr-usage,
resource:class bpr-object) = (item-or-value)

{
 This procedure selects the available resource with the smallest
 total-work-time.  It recursively searches of of the resources in the
 resource pool, returning the available resource in the resource pool
 with the smallest work time.
}

current-resource:class bpr-object;
temp-resource:item-or-value;
temp-work-time:quantity;
most-idle-resource:item-or-value = the symbol none;
smallest-work-time:quantity;

begin
 {
  If the resource is a resource pool, the search all of the members in the
  resource pool.
 }
 if the subworkspace of resource exists then begin
  for current-resource = each bpr-object in the members of resource do
   temp-resource = call bpr-most-idle-resource(activity, usage, current-resource);
   {
    If the resource pool has an available resource, compare its
    total-work-time with the smallest work time so far.
   }
   if temp-resource is a bpr-object then begin
    temp-work-time = the total-work-time of the duration-subtable of temp-resource;
    {
     If this is the first available resource or if the total-work-time is smaller,
     make this the most-idle-resource.
    }
    if most-idle-resource is not a bpr-object or temp-work-time < smallest-work-time
then begin
     most-idle-resource = temp-resource;
     smallest-work-time = temp-work-time;
    end;
   end;
  end
 end else begin
 {
  If the resource is not a resource pool, simply return the resource if it
  is available.
 }
  if the maximum-utilization of the duration-subtable of resource - the current-utilization
of the duration-subtable of resource >= the utilization of usage then
   most-idle-resource = resource;
 end;

 return (most-idle-resource);
end
```

To test the model:

1 Create a simple model with a Source block, a Task block, and a Sink block.

2 Create a resource pool with three resources.

3 Create a Resource Manager from the resource pool and attach it to the Task block.

4 Display the properties dialog for the Resource Manager, click the Allocate tab, and configure Choose Resource to be bpr-most-idle-resource.

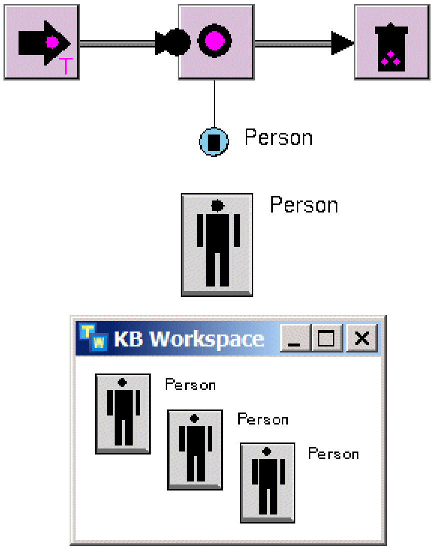
5 Configure a duration of 1 hour for the Source block.

6 Configure a duration of 1 hour with a deviation of 15 minutes for the Task block.

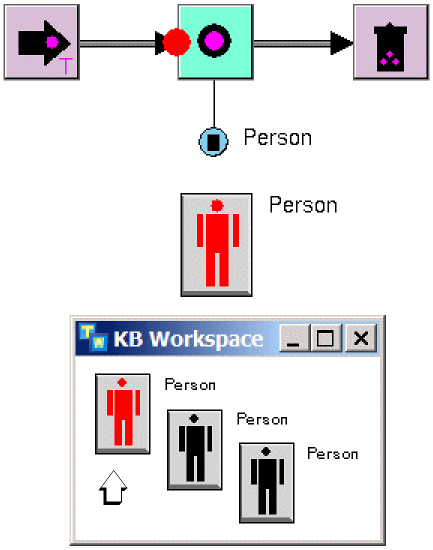
7 Run the model in Step mode and observe how the Resource Manager allocates resources.

The manager chooses the available resource that has the shortest total work time for each activity.

This figure shows a running model, where the Task block allocates resources from a pool of three resources:

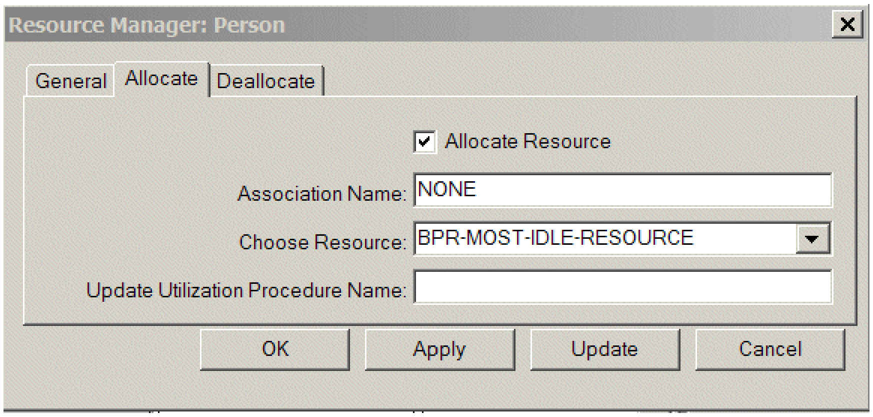


This figure shows the result of taking one more step in the simulation. ReThink allocates the resources with the smallest total-work-time, which is the last resource in the pool.



ReThink allocates the resources with the smallest total-work-time.

Here is the Allocate tab of the properties dialog for the Resource Manager that chooses resources from the pool, using the custom procedure:



☑ Allocate Resource

Association Name: NONE

Choose Resource: BPR-MOST-IDLE-RESOURCE

Update Utilization Procedure Name:

Common Customization Attributes of Resources and Work Objects

A resource is a subclass of a work object, which means they define some of the same attributes. The common G2 and ReThink customization attributes of resources and work objects, which are visible on the Customize tab, are:

Attribute	Description
GFR UUID	The internal identification number of the object.
Notes	The current status of the resource or work object.
Names	The internal name of the resource or work object. ReThink uses the label attribute rather than the names attribute to identify objects and avoid naming conflicts.
Reset Procedure	The procedure name the object uses when the simulation is reset. The default value is bpr-reset-object.
Delete Procedure	The procedure name the object uses when it is deleted. The default value is bpr-delete-object.
Update Procedure	The procedure name the object uses when the Update button on the object is clicked. The default value is bpr-update-object.
Animation Subtable	Subobject that specifies the default colors the resource or work object uses when it is in an active, inactive, or error state. The default value is an instance of a bpr-object-animation-subtable. This attribute is only visible through the table.

Attribute	Description
Duration Subtable	Subobject that specifies the timing and utilization parameters of the resource or work object, and computes summary timing and utilization statistics. The default value is an instance of a bpr-object-duration-subtable. This attribute is only visible through the table.
Cost Subtable	Subobject that specifies how the resource or work object computes cost statistics. The default value is an instance of a bpr-object-cost-subtable. This attribute is only visible through the table.

For additional information, see:

- *G2 Reference Manual*.
- Editing Subobjects.

Attributes of Animation Subtable

The customization attributes of the animation-subtable of a resource or work object, which are visible on the Animation tab, are:

Attribute	Description
Procedure Name	The default animation procedure for the resource or work object. The default value is bpr-animate-object.

Customizing the colors of a resource is similar to customizing the colors of a block. To customize the colors or animation of a work object, you must create a custom subclass of the animation subobject and edit the subobject.

For additional information, see the references in this table:

For information on...	See...
How to customize the default colors of a block	Editing the Default Colors of a Block.

For information on...	See...
How to customize the default animation procedure of a resource	Customizing Resource Animation.
Customizing sub-objects of work objects	Customizing Subobjects of Work Objects.

Attributes of Duration Subtable

The customization attributes of the duration-subtable of a resource or work object, which are visible on the Utilization tab, are:

Attribute	Description
Reset Procedure	The name of the procedure the resource or work object uses to reset the duration subtable when the simulation resets. The default value is bpr-reset-object-duration-subtable.
Duration Procedure	The name of the procedure the resource or work object uses to compute duration and utilization. The default value is bpr-object-duration.

For general information on how to customize subobjects, see Editing Subobjects.

Attributes of Cost Subtable

The customization attributes of the cost-subtable of a resource or work object, which are visible on the Cost tab, are:

Attribute	Description
Cost Reset Procedure Name	The name of the procedure the resource or work object uses to reset the cost subtable when the simulation resets. The default value is bpr-reset-object-cost-subtable.
Cost Procedure Name	The name of the procedure the resource or work object uses to compute total-cost. The default value is bpr-object-cost.

Customization Attributes of Resource Managers

The customization attributes of Resource Managers, which are visible in the dialog, are:

Attribute	Description
Choose Resource (Allocate tab)	Attribute that defines the procedure that determines how the Resource Manager allocates and deallocates resources. The default value is bpr-random-available-resource. The attribute is called choose-resource-procedure-name in the table.
Update Utilization Procedure Name (Allocate tab)	Attribute that defines the procedure that determines how the Resource Manager configures utilization. ReThink automatically calls this procedure each time a block is evaluated and before resources are allocated. Therefore, this procedure may update the Utilization attribute of the Resource Manager, based on the context of the model or based on information from the work object. For example, the work object may specify the number of resources required to perform a task. This procedure would copy the utilization from an attribute of the work object to the Utilization attribute of the Resource Manager. The signature of this procedure is: my-custom-update-resource-utilization (*block*: class bpr-task, *manager*: class bpr-resource-manager, *activity*: class bpr-activity)

Attribute	Description
Choose Manager (Deallocate tab)	Attribute that defines the procedure that determines how the Resource Manager allocates blocks waiting for resources when multiple blocks allocate resources from the same pool. The Resource Manager update the Blocks-waiting attribute of the resource, according to the procedure. The default value is bpr-random-waiting-block. The attribute is called sequence-block-procedure-name in the table.
Animation Subtable	Subobject that specifies the default colors the resource or manager uses when it is in an active, inactive, or error state. The default value is an instance of a bpr-resource-manager-animation-subtable. This attribute is only available in the table.
Duration Subtable	Subobject that specifies the timing and utilization parameters of the resource or manager, and computes summary timing and utilization statistics. The default value is an instance of a bpr-resource-manager-duration-subtable. This attribute is only available in the table.
Cost Subtable	Subobject that specifies how the resource or work object computes cost statistics. The default value is an instance of a bpr-resource-manager-cost-subtable. This attribute is only available in the table.

Attributes of Resource Manager Paths

By default, the paths connected to Resource Managers automatically reconfigure themselves whenever you move the connected manager along the block. You can customize this behavior by editing this attribute:

Attribute	Description
Redraw Connection	Set to false to cause Resource Manager paths not to reconfigure themselves along the block when you move the connected manager. The default value is true.

6

Customizing the User Interface

Describes how to customize various aspects of the user interface.

Introduction

ReThink allows you to customize these aspects of the user interface:

- Properties dialogs for blocks, instruments, resources, and work objects.
- ReThink toolbox to add custom tabs with custom objects.

ReThink does not currently support customization of these aspects of the user interface:

- Top-level menu bar.
- Toolbars other than the ReThink toolbox.

For more elaborate customizations or to build a completely new user interface to provide a custom product identification ReThink includes extensive libraries to build G2-based user interfaces. For details, see the G2 Developers' Utilities, as well as G2 custom dialogs.

Customizing Properties Dialogs

When subclassing ReThink blocks, instruments, resources, and work objects, you often define class-specific attributes. By default, ReThink creates a new tab on the properties dialog named User tab on which it places all the class-specific attributes of the object.

ReThink determines the appropriate user interface control for displaying and editing the attribute value, based on the attribute type. For example, if the attribute is a numeric value it will display an editable field with spin buttons to increment or decrement the value. If the attribute is a color, ReThink creates a dropdown list of all available colors.

When subclassing ReThink blocks, instruments, resources, and work objects, you can control the organization of the information to place the attributes on different tabs.

You can only add and modify attributes on user-defined tabs; you cannot add attributes to the default tabs or modify the controls used for the default attributes.

To create custom tabs and to control the order of the attributes that appear on each tab, implement the following method:

```
bpr-dialog-add-user-tabs
   (o: class subclass, dlg: class gdu-dialog-definition, tab-control-id: symbol,
   tab-names: sequence, win: class g2-window)
   -> tabs: sequence
```

where:

- *subclass* is the class of the object for which you're redefining the user tabs.
- *dlg* is the associated gdu-dialog-definition.
- *tab-names* is a sequence of texts of tabs added so far for the object.
- *tabs* is the sequence of texts of tabs to be added.

For example, if you have a custom block with a custom attribute, and you don't want any user tabs to appear, create a method with the signature above that simply returns *tab-names*.

If you have a custom block with two custom attributes, and you want to show each on a separate tab, create a method such as this:

```
bpr-dialog-add-user-tabs (o: class custom-task, Dlg: class gdu-dialog-definition,
    tab-control-id:symbol, tab-names:sequence, Win: class g2-window) = (sequence)
x:integer = 0;
y:integer = 1;
w:integer = 1;
h:integer = 1;
tab-label:text;
begin
    tab-label = call gdu-localize-text(dlg, the symbol bpr-custom-user-tab,
        "Custom User", win);
    tab-names = insert-at-end(tab-names, tab-label);
    call gdu-add-text-box-control(dlg, the symbol custom-counter, the symbol
        custom-counter, true, true, false, x, y, w, h, tab-control-id, tab-label);
    tab-label = call gdu-localize-text(dlg, the symbol bpr-custom-user-tab2,
        "Custom User 2", win);
    tab-names = insert-at-end(tab-names, tab-label);
    call gdu-add-text-box-control(dlg, the symbol custom-counter2, the symbol
        custom-counter2, true, true, false, x, y, w, h, tab-control-id, tab-label);
    return tab-names;
end
```

Note that you could split up individual tabs in separate methods or procedures if you desire. For more examples, see the `gdu-demo.kb` located in the `g2i\examples` directory of your ReThink installation.

Customizing the ReThink Toolbox

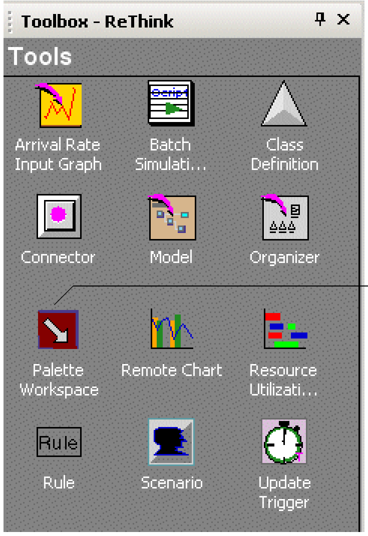
When subclassing ReThink blocks, instruments, and resources, you might want to make the custom objects available on the ReThink toolbox. You can add any number of user tabs with custom objects.

To customize the ReThink toolbox, you create a Palette Workspace object, place custom objects on its detail, and configure the objects on the detail for palette behavior.

Note You must configure the Palette Workspace detail label and objects in the server, using Administrator mode.

To customize the ReThink toolbox:

1 Choose View > Toolbox - ReThink and display the Tools palette:



Palette Workspace

2 Create a Palette Workspace and place it on your customization workspace.

3 Configure the names attribute to be a unique name.

4 Configure the palette-name to be the name of the palette to appear in the ReThink toolbox.

5 Configure the palette-group to be "ReThink".

ReThink uses a default icon for the palette button at the bottom of the toolbox, which looks like this: ⚒

You can specify the name of any class or any of the GMS built-in icons. For details, see *G2 Menu System User's Guide*.

6 Configure the icon-name, as desired.

7 Choose show detail on the palette tab.

8 Click the label of the detail and edit it to be the palette label.

9 Place objects on the detail and enable GFR palette behavior for each object, as follows:

a Enable GFR palette behavior by choosing Tools > GFR Palette Behavior.

a Place custom objects on the detail to make them appear on the custom tab.

b For each object, choose add palette behavior.

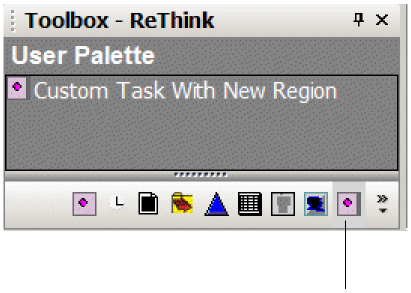
c Once you have added all the objects you want, disable GFR Palette Behavior.

10 Switch back to Developer mode.

11 Restart G2 and display the ReThink toolbox.

The custom tab appears in the toolbox with the custom objects. By default, the label for the custom object is the class name. To customize the label, configure the GRF text resource. For details, see the *G2 Foundation Resources User's Guide*.

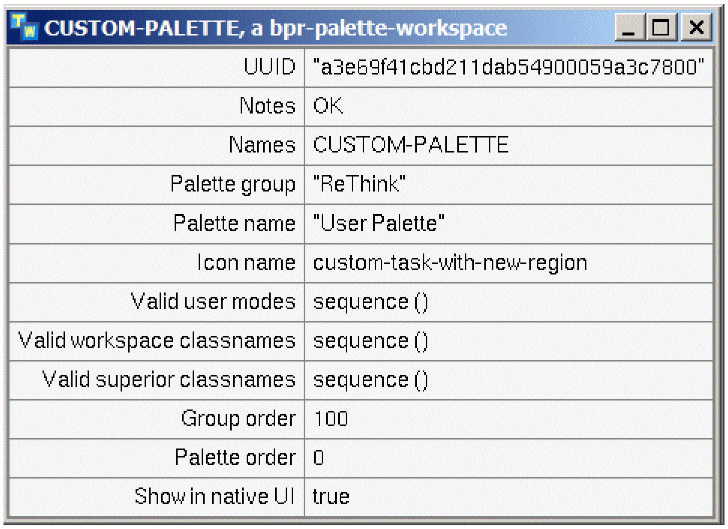
Here is an example that shows a custom tab on the ReThink toolbar:



Custom palette button

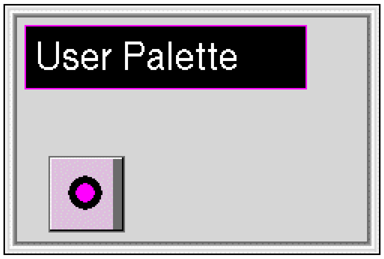
Here is the palette workspace object and its properties dialog:

CUSTOM-PALETTE



UUID	"a3e69f41cbd211dab54900059a3c7800"
Notes	OK
Names	CUSTOM-PALETTE
Palette group	"ReThink"
Palette name	"User Palette"
Icon name	custom-task-with-new-region
Valid user modes	sequence ()
Valid workspace classnames	sequence ()
Valid superior classnames	sequence ()
Group order	100
Palette order	0
Show in native UI	true

Here is the detail of the palette workspace with the custom object:



7

Customizing Menus

Provides specific descriptions and examples of how to customize ReThink menus and palettes.

Introduction

You can customize the ReThink menus to provide menu choices relevant to your ReThink application. For example, you can:

- Add menu choices to the Application menu that display the workspaces of your ReThink models.
- Extend the Palettes menu to include custom palettes of ReThink objects.

You use the G2 Menu System (GMS) module and the G2 Foundation Resources (GFR) module to customize the ReThink menus.

Displaying the Menu Bars Layout Workspace

You customize ReThink's menus by configuring objects on the Menu Bars Layout workspace.

To display the Menu Bars Layout workspace:

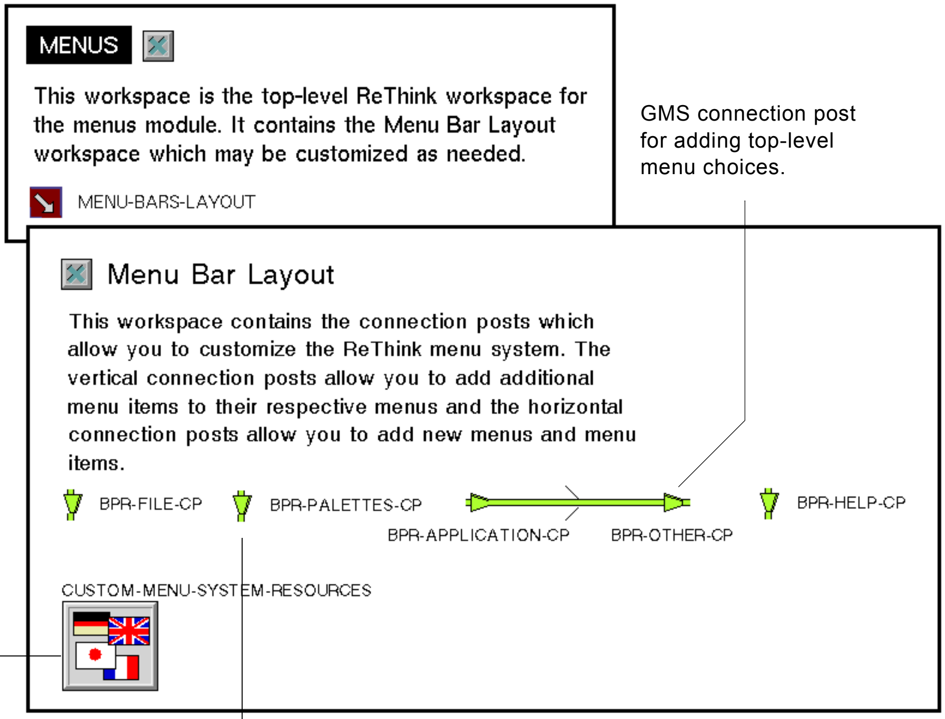
1 Choose Workspace > Get Workspace and choose MENUS.

2 Click the menu-bars-layout icon to display its subworkspace.

The two vertical connection posts represent extensions to ReThink's proprietary top-level menu choices: File and Palettes. The horizontal connection posts allow you to add menu choices to the top-level menu. The objects on this workspace are part of the G2 Menu System (GMS), a module of G2.

The Menu Bar Layout workspace also contains a resource icon with flags, which you use to extend the menus. This object, and the object on its subworkspace, are part of the G2 Foundation Resources (GFR), a module of G2.

Here are the Menus and Menu Bar Layout workspaces:



MENUS

This workspace is the top-level ReThink workspace for the menus module. It contains the Menu Bar Layout workspace which may be customized as needed.

Menu Bar Layout

This workspace contains the connection posts which allow you to customize the ReThink menu system. The vertical connection posts allow you to add additional menu items to their respective menus and the horizontal connection posts allow you to add new menus and menu items.

GMS connection post for adding top-level menu choices.

GFR text resource

GMS menu connection posts for extending the Palettes menu.

The Menus and the Menu Bar Layout workspaces are assigned to the menus module. You make all customizations to the ReThink menu structure by editing the objects on the Menu Bar Layout workspace and saving these edits to the menus module.

The proprietary ReThink menus are defined on a different workspace, named the Menu System Layout workspace, which is assigned to the bpr proprietary module.

Note Do not customize the menu layout on the Menu System Layout workspace; otherwise, your customizations will be overwritten the next time you install a new version of ReThink. You can extend the existing menus by customizing the Menu Bar Layout workspace in the menus module.

Creating an Applications Menu

You can create an Applications menu choice to the top-level menu, with menu choices that display the workspaces of your ReThink models. To do this:

- Create an Applications menu choice.
- Add menu templates to the diagram for each workspace you wish to display.
- Create named workspaces for each workspace you wish to display.
- Edit the local text resource object to include symbol definitions for the top-level menu choice and each new menu choice.
- Edit the definition of the menu templates to display the desired workspace.

To customize the menu, you use the GMS and GFR modules of G2. These modules are included in the ReThink module hierarchy.

For more information on these modules, see the *G2 Menu System User's Guide* and the *G2 Foundation Resources User's Guide*.

Creating an Applications Menu

You can create a top-level menu choice called the Applications menu, which allows you to display top-level workspaces of your ReThink models. To do this, you add a **Cascade Menu Template** to the Menu Bar Layout workspace.

To add an Applications menu choice to the top menu bar:

1. In Developer mode, click the connection between the bpr-application-cp and bpr-other-cp connection posts to display the path menu and choose delete to delete the connection.

2. Choose Workspace > Get Workspace and choose GMS-TOP-LEVEL to display the GMS top level workspace.

   This workspace contains a number of GMS items for customizing menus.

3. Clone a Cascade Menu Template from the palette and place it between the two unconnected connection posts.

   The Cascade Menu Template is the top-left item under Cascading Entries:

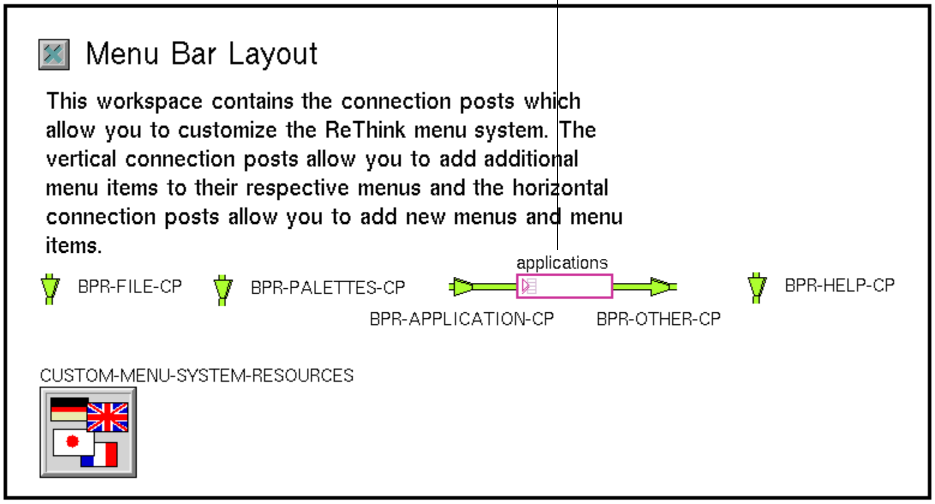
4. Drag the stubs from each connection posts into the Cascade Menu Template.

5. Click the none label on the menu template to display the G2 Text Editor.

6. Edit the label of the menu template to be applications and drag the label above the menu template box so it is visible.

You have now added the top-level Applications menu choice to the menu bar. The Menu Bar Layout workspace looks like this:

Cascade Menu Template



Menu Bar Layout

This workspace contains the connection posts which allow you to customize the ReThink menu system. The vertical connection posts allow you to add additional menu items to their respective menus and the horizontal connection posts allow you to add new menus and menu items.

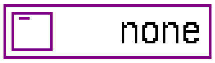
Adding a Menu Choice that Displays Your Model

Now you will add a menu choice that displays the top-level workspace of your ReThink application. GMS supplies a special purpose menu template item, called a **Show Workspace Template**, which automatically displays a named workspace. You will use a Show Workspace Template to add a custom menu choice to the Applications menu.

To add a menu choice that displays your model workspace:

1 Display the GMS top-level workspace and clone a Show Workspace Template and position it under the Applications menu template item.

The Show Workspace Template is the middle item under Leaf Entries:



Now you need to add stubs to the Show Menu Template so you can connect it to the Cascade Menu Template above.

2 Click the border of the Show Menu Template that you just cloned to display its menu and choose add submenu stubs.

GMS creates yellow stubs on all four sides of the menu template.

3 Connect the stub leading out of the top of the Show Menu Template to the bottom of the Cascade Menu Template.

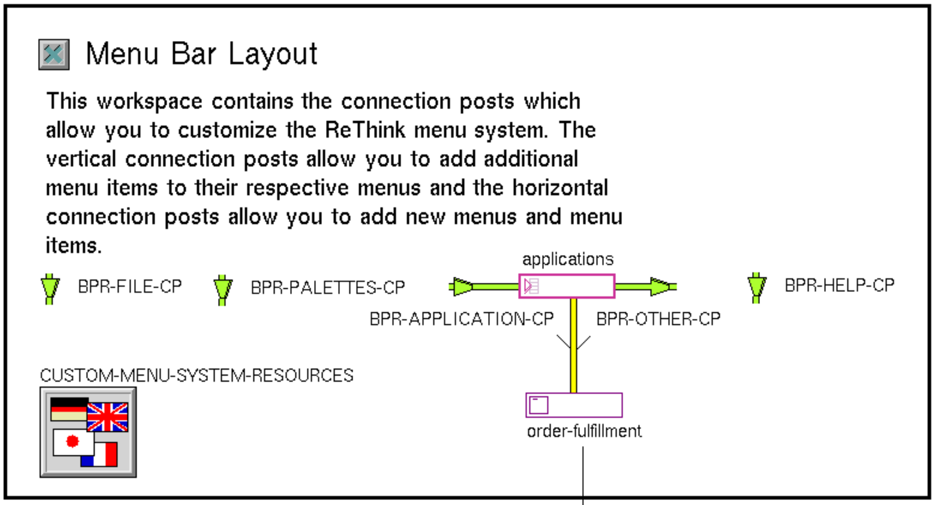
4 Display the Show Menu Template menu again and choose remove stubs to remove the unconnected stubs.

5 Click the none label, enter the text label for the menu template, and drag the label to below the template item.

For example, if you are going to use this menu choice to display an order fulfillment model, the label might be order-fulfillment.

6 Hide the GMS top-level workspace.

The Menu Bar Workspace looks like this:



Menu Bar Layout

This workspace contains the connection posts which allow you to customize the ReThink menu system. The vertical connection posts allow you to add additional menu items to their respective menus and the horizontal connection posts allow you to add new menus and menu items.

Show Workspace Template

Creating a Named Workspace to Display

Once you have added the menu template item to the diagram, the next step is to create the named workspace to display.

Note If you already have a model workspace to display, skip this step.

To create a named workspace to display:

1. Choose Workspace > New Workspace.
2. Choose KB Workspace > Name to specify the name of the workspace to display.

For example, if your ReThink application is a model of an order fulfillment process, you might name the workspace order-fulfillment-top-level. This workspace is assigned to the models module.

For demonstration purposes, you might want to display the top-level workspace of the Order Fulfillment model that ships with ReThink. To do this, merge the *orderful.kb* file from the ReThink directory into the current model. The name of the top-level Order Fulfillment model workspace is order-fulfillment-tutorial.

Tip Named workspaces appear in the list of available workspaces when you choose Workspace > Get Workspace.

Using a Local Text Resource to Create Your Custom Menu Choice

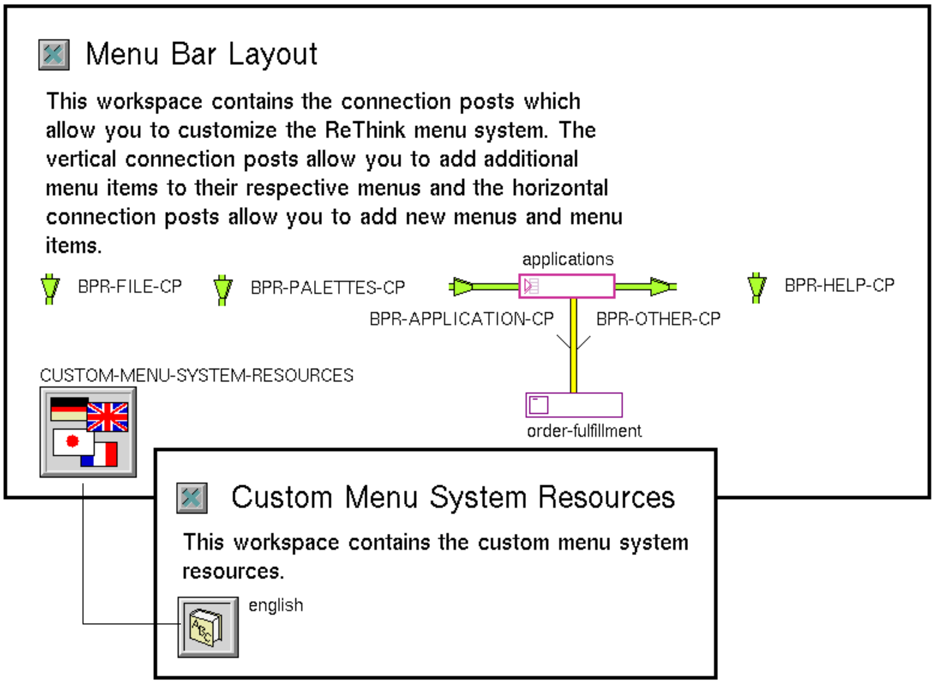
Now you must edit the GFR local text resource on the subworkspace of the custom-menu-system-resources **text resource group** on the Menu Bar Layout workspace. To identify each new menu choice, you create a unique symbol and associated text string that identifies the menu choice and specifies the menu choice text. You edit the **local text resource** on the subworkspace of the text resource group by using an external editor.

To display the local text resource for the custom menus:

➔ Click the icon labeled CUSTOM-MENU-SYSTEM-RESOURCES and choose go to subworkspace.

ReThink displays the Custom Menu System Resources workspace, which contains a GFR local text resource named english. This is the local text resource in which you add symbols and associated strings for each custom menu choice you create.

Here is the custom-menu-system-resources object and its subworkspace, which contains a local text resource named english:



Menu Bar Layout

This workspace contains the connection posts which allow you to customize the ReThink menu system. The vertical connection posts allow you to add additional menu items to their respective menus and the horizontal connection posts allow you to add new menus and menu items.

Custom Menu System Resources

This workspace contains the custom menu system resources.

To edit this local text resource, you use any text editor.

To edit the GFR local text resource:

1 Click the local text resource labeled english on the Custom Menu System Resources workspace and choose table.

2 Edit the Gfr-file-location attribute to specify the complete pathname of a new text file, including double quotes.

3 Choose write resource to file on the local text resource to write the file.

4 Go to any editor and edit the file by adding a line of text for each new menu choice that you have created.

   The line of text contains a symbol name, followed by a comma, followed by a string that is the menu choice to display.

   In the previous example, you added the Applications menu choice to the top menu bar, and you added a submenu choice that displays the order fulfillment model workspace. The text lines you would enter would look as follows, where applications is the label for the Applications menu, and order-fulfillment is the label for the submenu choice.

   ```
   applications, "Applications"
   order-fulfillment, "Order Fulfillment Model"
   ```

5 Save the text file.

6 Choose load text resource on the local text resource to use the edited text resource file.

7 Choose make resource permanent to make the edits permanent.

Here is a external text resource file with the Applications menu and the custom menu choice added. Application and order-fulfillment are both unique symbols, and "Applications" and "Order Fulfillment Model" are the text strings to display in the menu:

```
CUSTOM-MENU-SYSTEM-RESOURCES
4.1 Rev. 0
ENGLISH
applications, "Applications"
order-fulfillment, "Order Fulfillment Model"
```

Editing the Workspace Templates

Now that you have created a local text resource for the custom menu choices, you can edit the definition of the Cascade Workspace Template and the Show Workspace Template on the Menu Bar Layout workspace to refer to the text resource group of which they are a part.

To edit the cascade workspace template to refer to the text resource group:

1 Click the Cascade Workspace Template item on the Menu Bar Layout workspace and choose table.

2 Edit the Gms-text-resource-group attribute to be the name of the GFR text resource group on the Menu Bar Layout workspace.

In this example, the name of this resource group is custom-menu-system-resources.

Notice that the Gms-label attribute is the label you entered in a previous step, applications.

For the Show Workspace Template, you must also specify the name of the workspace to display.

To edit the show workspace template to show your model workspace:

1 Click the Show Workspace Template item on the Menu Bar Layout workspace and choose table.

2 Edit the Gms-text-resource-group attribute to be the name of the GFR text resource group on the Menu Bar Layout workspace.

In this example, the name of this resource group is custom-menu-system-resources.

Notice that the Gms-label attribute is the label you entered in a previous step, order-fulfillment.

3 Edit the Gms-display-target attribute to be the name of the workspace to display.

In this example, the name of the workspace is order-fulfillment-top-level. Alternatively, to display the top-level workspace of the Order Fulfillment model that ships with ReThink, specify the workspace named order-fulfillment-tutorial. Be sure that you have merged the file `orderful.kb` into the current model first.

To see the effects of the changes to the menus:

➔ Restart G2.

The Application menu choice is now available on the top menu bar, and the Order Fulfillment Model menu choice is available as a submenu.

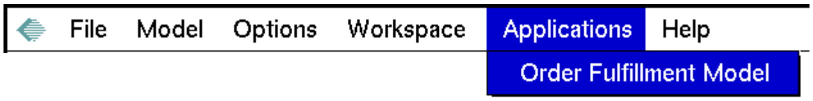
To test the custom menu choices:

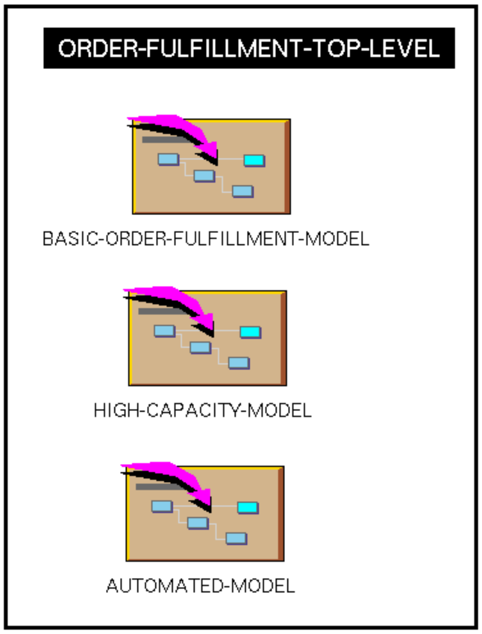
1 Click the Application menu.

The Order Fulfillment Model menu choice is available. This menu choice is the text string you entered in the text file associated with the GFR local text resource, which corresponds to the GMS label you entered in the Show Workspace Template item.

2 Click the Order Fulfillment Model menu choice.

ReThink displays the Order Fulfillment Top Level workspace:





Note By default, GMS displays the workspace in its previous location on the G2 window. To specify the default location of the workspace, edit the attributes Gms-window-symbolic-location and Gms-workspace-symbolic-location in the Show Workspace Template item.

Adding a Custom Palette to the Palettes Menu

In the *ReThink User's Guide*, you learned how to create a named workspace of commonly used icons from the ReThink icon library, which you can display by using Workspace > Get Workspace. You might want to create a palette of custom icons and objects, which you access from a menu choice on the Palettes menu.

Caution Do not add custom objects to the default ReThink palettes; otherwise, your KB will be inconsistently modularized, and your customizations will be deleted the next time you install a new version of ReThink.

This example shows how you add a custom palette named Custom Blocks to the Palettes menu.

For the detailed steps of each major step, see Creating an Applications Menu on page 155.

To add a menu choice that displays a custom palette:

1. Clone a Show Workspace Template item from the GMS Top Level workspace and connect it below the bpr-palettes-cp connection post on the Menu Bar Layout workspace.

   For the steps of how to do this, see Adding a Menu Choice that Displays Your Model on page 157.

2. Create a named workspace that contains your custom blocks.

   For the steps of how to do this, see Creating a Named Workspace to Display on page 158.

3. Replace the workspace name with a workspace label that hides the workspace when you select it in Modeler mode.

   To replace the label:

   a. Choose hide name on the title.

   b. Go into Administrator mode.

   c. Clone a workspace label from any ReThink subworkspace, for example, the Scenario Control Panel.

   d. Edit the title.

   e. Choose change min size on the label to adjust its width, as needed.

4. Assign the workspace to the customiz module by editing the Module-assignments attribute in the table for the workspace.

5. Add a unique symbol and text string that identifies the Custom Blocks menu choice to the text file associated with the GFR local text resource named english, then load the text resource and make the resource permanent.

   For example, you might add the following line to the text file:

   ```
   custom-blocks-palette, Custom Blocks
   ```

   For more information, see Using a Local Text Resource to Create Your Custom Menu Choice on page 159.

6. Edit the Show Workspace Template item to display the local text resource you just added.

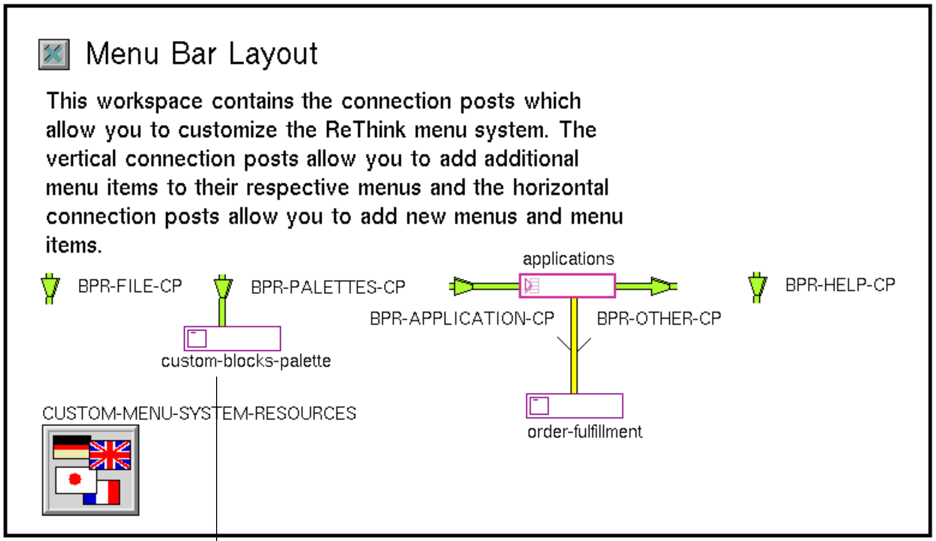
   For the steps of how to do this, see Editing the Workspace Templates on page 160.

7 Specify the default location of the workspace to be the top-left corner of the G2 window:

- **a** Edit the Gms-window-symbolic-location attribute of the Show Workspace Template item to be top-left-corner.
- **b** Edit the Gms-workspace-symbolic-location attribute of the Show Workspace Template item to be top-left-corner.

8 Restart G2.

Here is the Menu Bar Layout workspace with the custom blocks menu choice added to the Palettes menu, as well as the order fulfillment menu choice added to the Application menu:

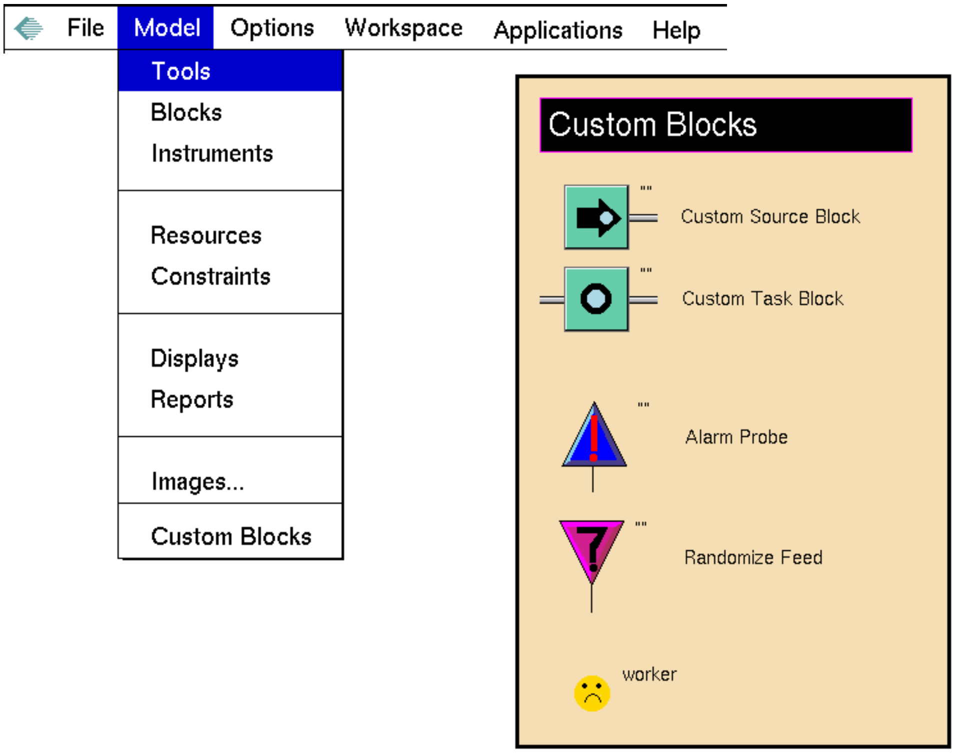


Menu Bar Layout

This workspace contains the connection posts which allow you to customize the ReThink menu system. The vertical connection posts allow you to add additional menu items to their respective menus and the horizontal connection posts allow you to add new menus and menu items.

Custom palette menu choice.

Here is the top-level menu bar with the custom menu choice added and a sample Custom Blocks palette with some custom ReThink objects:



Saving the Menus Module

Once you have finished customizing the ReThink menus, you save the menus module in the *menus.kb* modularized KB file. The next time you load ReThink, ReThink automatically loads the menu customizations because the menus module is a required module in the module hierarchy.

If you install a new version of ReThink, you need to copy your customized menus module to the new ReThink directory, overwriting the default module in the new version of the software. For information on how to do this, see the *ReThink Installation Guide*.

To save your customizations to the menus module:

1. Choose File > Save As.
2. To save the menus module, save the models module and click the Save Module, Including All Required Modules option on, or save just the menus module to the file *menus.kb*.

Part III

ReThink Internals

Block Processing

Describes the internal processing that ReThink performs when a work object arrives at a block.

Introduction

When any block evaluates within a model, ReThink executes a number of procedures. These procedures establish relations between objects, compute cost and duration statistics for objects, animate objects, allocate and deallocate resources, and activate instruments. When you customize ReThink, it is essential to understand the order in which these events take place so you can write your methods and procedures accordingly.

The High-Level View

Block processing consists of three basic phases:

- The planning phase
- The start activity phase
- The stop activity phase

Whenever a block receives a work object, ReThink must first determine whether the block is ready to run by checking to see if the current number of activities is below the maximum that the block specifies.

If the block is ready, it enters a planning phase for the activity. It checks to see if all the required inputs are available for the block. If the block has an attached Resource Manager, it checks to see if all the required resources are available for the activity. If the resources are not available, the block waits for the resources to become available.

When all the required resources are available, ReThink creates an activity object for the block and schedules the start event for the activity. Every activity has a start event, which invokes the procedures that process the block. The block executes the start event at the same simulation time at which it schedules the event.

When the activity's start event completes, it schedules the activity's stop event to occur after the duration. The stop event invokes the actions that conclude the processing of the block. After the stop event concludes, the work object moves to the next block in the simulation, which activates the processing of that block.

ReThink sorts and processes the start and stop events, based on their timestamps and their priorities. Within each second of simulation time, ReThink sorts the events by priority, then processes events with the lowest priority first. Note that the scheduling of events is not limited to the start and stop activities of blocks; it is also used to update reports, calculate metrics, and schedule parameter changes.

The internal priorities that ReThink uses for various events are:

Priority	Event
5	The end of the statistic period for a Statistic probe.
10	Attribute changes scheduled from a Scenario Manager or an Attribute Change Event Report.
100	Start and stop phase events of blocks.

Priority	Event
500	Update events associated with Update Trigger tools and probes.
9999	Update events that automatically update reports when the Update Mode of the report is set to Simulation Time.

If the scenario is online mode and your model uses the online blocks, the processing order of work objects cannot be deterministic. ReThink always processes work objects, using the event timestamp and priority. However, the online blocks use asynchronous calls to remote procedures or execute SQL statements. ReThink does not wait until the remote process or database has completed the task to process other work objects. Instead, it spawns a new processing thread in the remote program, remote ReThink model, or database, and continues processing other work objects. Once the processing thread completes, the work object is propagated through the ReThink diagram. Thus, because the execution time of the processing thread or database SQL statement may vary, the order in which work objects are processed may vary as well.

Note that if the scenario is in online mode, the duration specified in a block is ignored, that is, the work-time and elapsed-time of the activity are set to 0. The exception is the Delay block and the Source block, including any subclass. When using ReThink as a workflow engine, block durations are not relevant because the timing of events is driven by other constraints. You use the Delay block to implement specific operational process delays. The Time Period of a Statistics probe and the Update Interval of a report are still based on scenario time in online mode.

The Planning Phase

When a block receives an input, ReThink calls an internal API procedure that causes the block to evaluate.

Check to See If the Block Is Ready

First, the block evaluator checks to see if the block is ready to create an activity. It does this by checking to see if the maximum-activities attribute of the block has a value. If it has a value, the block checks to see whether the value exceeds the current-activities for the block. If so, the block waits; otherwise, it proceeds to the next step.

Create An Activity

If the block is ready, the block evaluator creates an activity. The block then relates the activity to the work object and the work object to the path by creating these relations:

- A a-bpr-input-of-activity relation between the work object and the activity.
- A a-bpr-object-of-input-path relation between the work object and the path leading out of the block.

For information on relations, see Relations.

Synchronize Inputs

Every block has a needs-all-input attribute, which is visible in Developer mode on the Customization tab. If needs-all inputs is true, the block requires inputs on all of its input paths before it can execute. For example, a Task block synchronizes its inputs, which means it must wait until all of its inputs have arrived before it processes them.

If needs-all-inputs is false, the block can evaluate whenever it receives a value on any of its input paths. For example, a Merge block, which merges separate streams of work, can process its inputs whenever it receives them.

Once an activity is created, ReThink checks to see if it has all of its required inputs. If so, the block proceeds to the next step; otherwise, it waits to receive all of its inputs.

Establish a Relation Between the Activity and the Block

Once the required inputs to the block are available, ReThink establishes a a-bpr-activity-of-block relation between the activity and the block.

Update Block Statistics

ReThink sets the time of the start event of the activity to the current time of the scenario.

ReThink then updates the following summary statistics related to the duration, cost, and status of the block:

- Duration statistics: average-in-process, last-update-time, total-work-time, total-elapsed-time
- Cost statistics: total-cost
- Status statistics: total-starts, total-stops, current-activities

Pause the Simulation

If there is a break point set for the block, that is, if the user has chosen the set break menu choice, ReThink pauses the simulation at this stage.

Request Resources

ReThink is ready to request resources for the block. It does this by following these steps:

- The block checks to see if there is an attached Resource Manager.
- If so, it creates a usage object and establishes the following relations:
  - A a-bpr-usage-of-activity relation between the usage object and the activity.
  - A a-bpr-usage-of-resource-manager relation between the usage object and the Resource Manager.
  - A a-bpr-usage-of-object relation between the usage object and the resource.
- The block then finds the first resource in the pool with sufficient availability, randomly reordering the resources as it chooses.
- When the block finds a resource or set of resources with sufficient availability for each resource manager attached to the block, it proceeds to the next step.

Schedule the Start Activity

ReThink now schedules the start activity phase.

Dequeue the Work Objects

At the end of the planning phase, ReThink removes the input work objects from the path queue.

The Start Activity Phase

Each activity has a start-event, which is a subobject of bpr-event, which defines the following attributes:

Attribute	Description
time	The start time of the event.
priority	The priority of the event.

To implement the behavior of any event, you implement the following method:

```
bpr-event-method
    (event: class bpr- event, scenario: class bpr-scenario,
    ui-client-item: class ui-client-item)
```

For details, see bpr-schedule-an-event.

Compute Block Duration and Costs

The block calls the procedure defined by the procedure-name of the block's duration-subtable attribute, which calculates duration statistics.

The block then calls the procedure-name of the block's cost-subtable attribute, which calculates cost statistics.

Start Input Work Objects

The block now "starts" each input work object, which does the following:

- Calls the procedure-name of the cost-subtable attribute of each work object, which calculates cost statistics.
- Calls the procedure-name of the duration-subtable of each work object, which computes all the summary duration statistics.
- Increments the total-starts and current-activities attributes of each work object.
- Calls the procedure-name of the animation-subtable of each work object, which animates the input work objects.

Start Allocated Resources

The block also "starts" each allocated resource, which performs the same actions as starting the input work objects, as described in Start Input Work Objects.

Evaluate Instruments When Phase is Start

If the Phase attribute of any attached instrument is start, the block evaluates the bpr-stop-method of the instrument, which happens before the attached block applies its duration to the simulation.

Update Block Statistics

The block again updates its activity, duration, and cost statistics, as described in Update Block Statistics.

Animate the Block

ReThink animates the block by calling the procedure-name of the block's animation-subtable.

Execute the Block's Start Method

Now, the block calls the bpr-start-method. This is a method you can customize for a block, as described in Customizing the Start Method.

Schedule the Stop Activity

ReThink now sets the time of the stop event of the activity to the current simulation time plus the elapsed time of the activity. It then schedules the stop-event of the activity.

The Stop Activity Phase

Each activity has a bpr-stop-event, which is a subobject of bpr-event, which defines the following attributes:

Attribute	Description
time	The start time of the event.
priority	The priority of the event.

To implement the behavior of any event, you implement the following method:

```
bpr-event-method
    (event: class bpr-event, scenario: class bpr-scenario,
    ui-client-item: class ui-client-item)
```

For details, see bpr-schedule-an-event.

Execute the Block's Stop Method

The block executes the bpr-stop-method, which typically defines its specific behavior. You can customize this method to refer to any of the attributes or relations computed up to this point.

Update Block Statistics

The block again updates various statistics, as described in Update Block Statistics.

Stop Allocated Resources

Now that the block is finished processing, it “stops” the resources. Stopping a resource does the following:

- Deallocates the resource and signals the availability of the resource for use elsewhere in the model.
- Recomputes costs, utilization, and duration of the resource.
- Increments total-stops and decrements current-activities of the resource.
- Animates the resource to its idle state.

Stop the Work Objects

Similar to stopping the resources, the block now stops the work objects. If there is no output path whose type matches the class of the work object, the block deletes the work object.

Update Total Cost of Work Objects

The block computes the new total cost of the work object by adding the cost of the activity.

Evaluate Instruments When Phase is Stop

If the Phase attribute of any attached instrument is Stop, the block evaluates the bpr-stop-method of the instrument, which happens after the attached block applies its duration to the simulation.

Send the Work Object Downstream

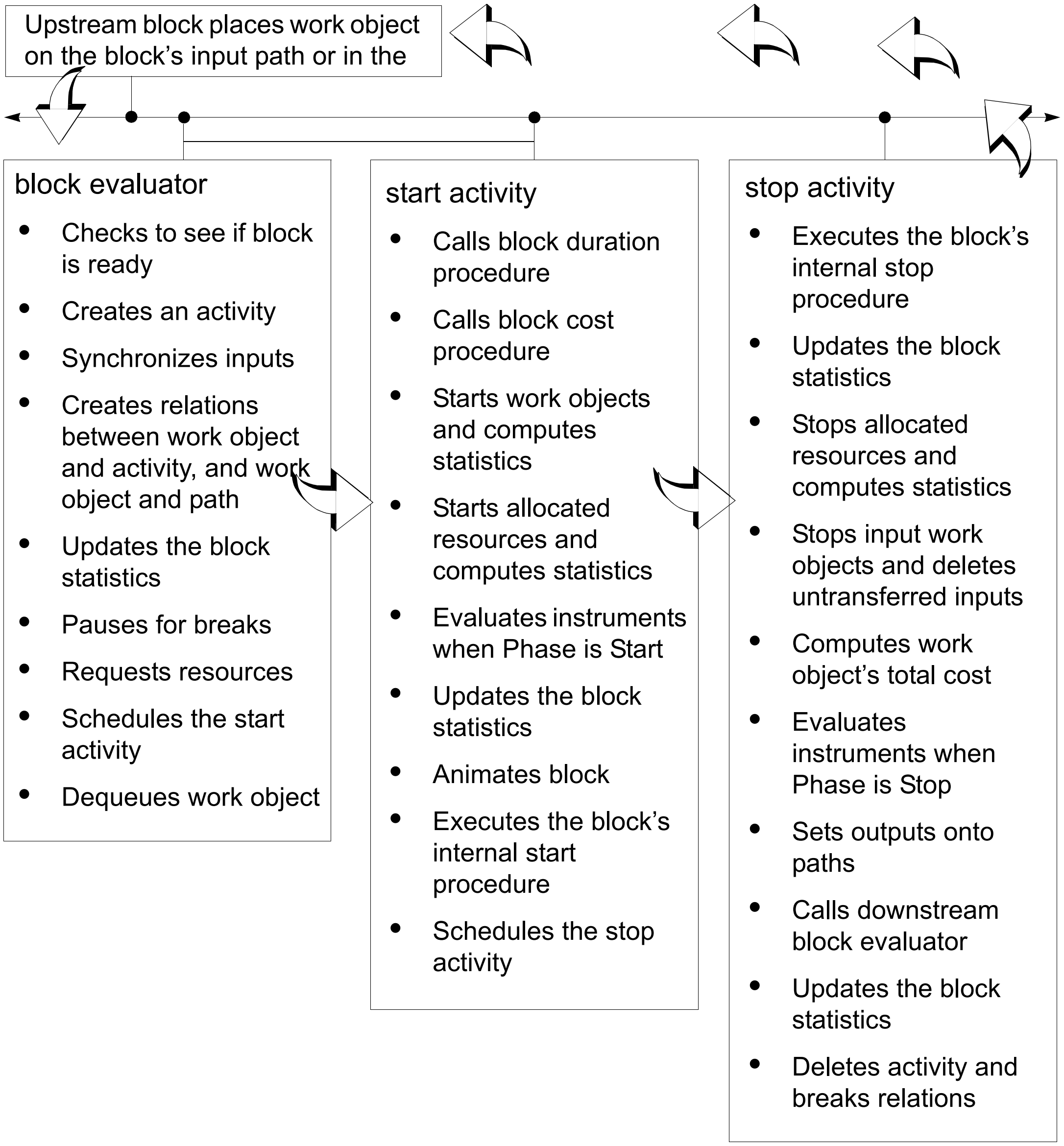
The block now sends the work object to the downstream block for processing. This triggers the planning phase of the downstream block, which repeats this entire process.

Clean Up

The block updates its statistics once more, and it deletes the activity and usage objects.

Block Processing Summary

The following time-line summarizes the steps in each internal procedure. The arrows represent the direction of execution and the looping that occurs.



Upstream block places work object on the block’s input path or in the

block evaluator

- Checks to see if block is ready
- Creates an activity
- Synchronizes inputs
- Creates relations between work object and activity, and work object and path
- Updates the block statistics
- Pauses for breaks
- Requests resources
- Schedules the start activity
- Dequeues work object

start activity

- Calls block duration procedure
- Calls block cost procedure
- Starts work objects and computes statistics
- Starts allocated resources and computes statistics
- Evaluates instruments when Phase is Start
- Updates the block statistics
- Animates block
- Executes the block’s internal start procedure
- Schedules the stop activity

stop activity

- Executes the block’s internal stop procedure
- Updates the block statistics
- Stops allocated resources and computes statistics
- Stops input work objects and deletes untransferred inputs
- Computes work object’s total cost
- Evaluates instruments when Phase is Stop
- Sets outputs onto paths
- Calls downstream block evaluator
- Updates the block statistics
- Deletes activity and breaks relations

Working with Time

ReThink allows you simulate multiple independent models simultaneously. To support this capability, each model must have its own Scenario tool, which in turn has its own simulation clock. Typically, these simulation clocks are separate from and independent of G2's real-time clock. This section briefly describes G2's real-time clock, ReThink's simulation clocks, and the relationship between them.

G2's Real-Time Clock

G2's real-time clock represents a number of seconds since a fixed time, typically the time at which the G2 session was started. So, the G2 expression the current time returns the number of seconds since the fixed reference time. For example, evaluating the expression the current time one and a half minutes after starting the G2 session returns the value 90. G2 keeps its clock synchronized with time in the real world, so 10 seconds later the expression returns 100. G2 provides numerous expressions and functions for displaying and manipulating time. For example:

- The G2 expression the current time as an interval displays the time in the more readable format of 1 minute and 30 seconds.
- The G2 expression the current time as a time stamp displays the time as a fixed time, such as, 1 Jan 2006 12:01:30 a.m., assuming the G2 session was started on January 1, 2006 at 12 AM.

See Task Scheduling in the *G2 Reference Manual* for a more detailed explanation of the G2 real-time clock.

ReThink's Simulation Clocks

ReThink simulation clocks work in a similar but independent way to G2's real-time clock by maintaining a relative time from a fixed reference time. In ReThink, the fixed reference time is the start time of the scenario, which is January 1, 2006 at 12:00 AM, by default. The scenario time attribute represents the number of seconds of simulation time that has elapsed since the start time. So after 1 hour of simulation time, on January 1, 2006 at 1 AM, the time of the scenario will be 3600.

It is important to note that this is *simulated* time, which has no relation to time in the real world. Running the simulation for that hour might have taken only 10 seconds in the real world. Thus, a discrete-event simulation models only significant changes that take place in the process being modeled. For example, the simulation clock advances when blocks start and stop, and then jumps forward in time between those events.

Referencing Real Time in ReThink

Because G2 provides very useful expressions and functions for displaying and manipulating time, ReThink scenarios maintain an additional attribute that you can use with these expressions and functions. This attribute is called g2-time and represents the number of simulated seconds since the G2 fixed reference time. This allows you to evaluate expressions such as the g2-time of scenario as a time stamp, which is the definition of the clock-time attribute of a scenario that appears as an attribute display showing the current simulation time.

Thus, the g2-time attribute is an alternative representation of the simulation time, or time of the scenario, which is convenient for use with G2's time expressions and functions.

9

Application Programmer’s Interface

Defines the ReThink API, which are the internal procedures you can call when you customize the behavior of ReThink objects.

Introduction

ReThink calls a number of internal procedures to define the behavior of objects. These procedures define the ReThink **application programmer's interface**, or API. When you customize ReThink, you can call any of these internal procedures to define custom behaviors for blocks and other ReThink classes. This chapter describes these procedures and provides a signature, description, and example of each.

Note ReThink supports only the API procedures that appear in this chapter; it does not support any other internal procedures the ReThink methods and procedures might call.

The categories of APIs are:

Scenarios

bpr-activate-scenario
bpr-continue
bpr-pause
bpr-reset

Blocks

bpr-block-evaluator
bpr-detach-input

Work Objects

bpr-clone-object
bpr-create-object
bpr-delete-object

Paths

bpr-dequeue-object
bpr-detach-input
bpr-post
bpr-post-path

Resources

bpr-remove-from-pool
bpr-update-pool

Miscellaneous

bpr-copy-list-of-attributes
bpr-get-item-for-label
bpr-get-item-for-label-class-scenario
bpr-handle-event-error
bpr-indicate
bpr-indicate-connection
bpr-lookup-by-id
bpr-message-to-all-users
bpr-schedule-an-event
bpr-updated-attributes

bpr-activate-scenario

Activates a scenario.

Synopsis

bpr-activate-scenario
    (*scenario*: class bpr-scenario, *workspace:* class kb-workspace,
     ui-client-item: class ui-client-item)

Argument	Description
scenario	The scenario associated with the model.
workspace	The top-level workspace associated with *scenario*.
ui-client-item	A ui-client-item, which is the superior class of: • A g2-window, which is a window in the current G2 session or a remote connection to G2 through a Telewindows client. • A ui-client-session, which is a remote connection to G2.

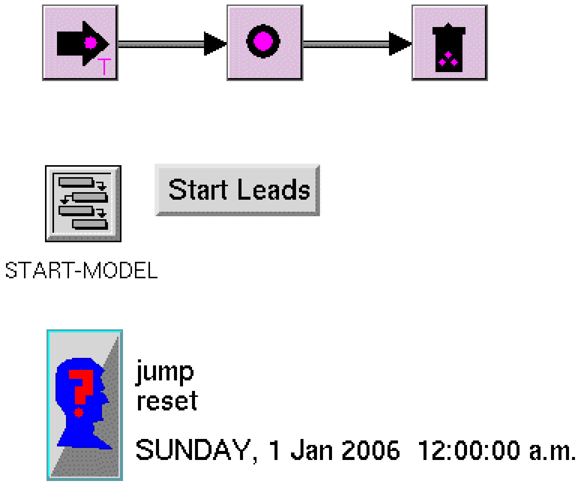
Description

This procedure activates *scenario* and changes its color to indicate an active state. Calling this procedure is equivalent to activating a scenario manually by choosing the activate scenario menu choice, except that it does not display the subworkspace of the scenario. A scenario must be active to run its associated model.

Example

The following custom procedure activates the scenario upon the current workspace, resets the scenario, evaluates the Source block on the current workspace, and continues running the scenario. You can activate the procedure with an action button.

For information about the other internal procedures in this example, see bpr-reset, bpr-continue, and bpr-block-evaluator.



Here is the text of the start-model procedure:

```
start-model (scenario: class bpr-scenario, client: class ui-client-item)
begin
    call bpr-activate-scenario(scenario, the workspace of scenario, client);
    call bpr-reset (scenario);
    allow other processing;
    call bpr-block-evaluator (the bpr-source upon this workspace);
    call bpr-continue(scenario);
end
```

bpr-block-evaluator

Starts a block by evaluating it.

Synopsis

bpr-block-evaluator
    (*block*: class bpr-block)

Argument	Description
block	The block to evaluate.

Description

This procedure initiates the execution of a block. Evaluating a block is the first step in the block processing described in Block Processing.

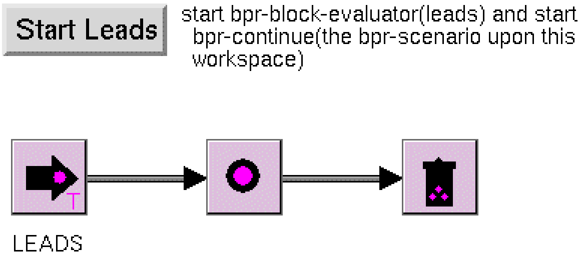
Calling this procedure for a Source block is similar to choosing the start menu choice from a Source block's menu, except that the start menu choice also calls bpr-continue.

For a detailed description of all of the actions this procedure performs, see The Planning Phase.

Example

You can create an action button that calls this procedure. The action button starts the blocks named leads and continues running the current scenario. You configure the name of the block on the Customize tab.

For information about the other internal procedure in this example, see bpr-continue.



```
start bpr-block-evaluator(leads) and start
 bpr-continue(the bpr-scenario upon this
 workspace)
```

bpr-clone-object

Creates a copy of a bpr-object.

Synopsis

bpr-clone-object
    (*original:* class bpr-object)
    -> *copy*: class bpr-object

bpr-clone-object
    (*original:* class bpr-object, *copy-item-lists*: truth-value,
      copy-item-list-items: truth-value)
    -> *copy*: class bpr-object

Argument	Description
original	The bpr-object to be cloned.
copy-item-lists	True if you want to clone the contents of attributes that are item-lists, false otherwise.
copy-item-list-items	True if you want to clone the items within item-lists, false otherwise.

Return Value	Description
copy	The bpr-object that is the copy.

Description

This procedure creates a copy of *original*, including its attribute values, by cloning *original*. When the Copy block receives an input, bpr-copy::bpr-stop-method, the method that defines the behavior of the Copy block, calls this API to output as many copies as it has output paths, as the example below shows.

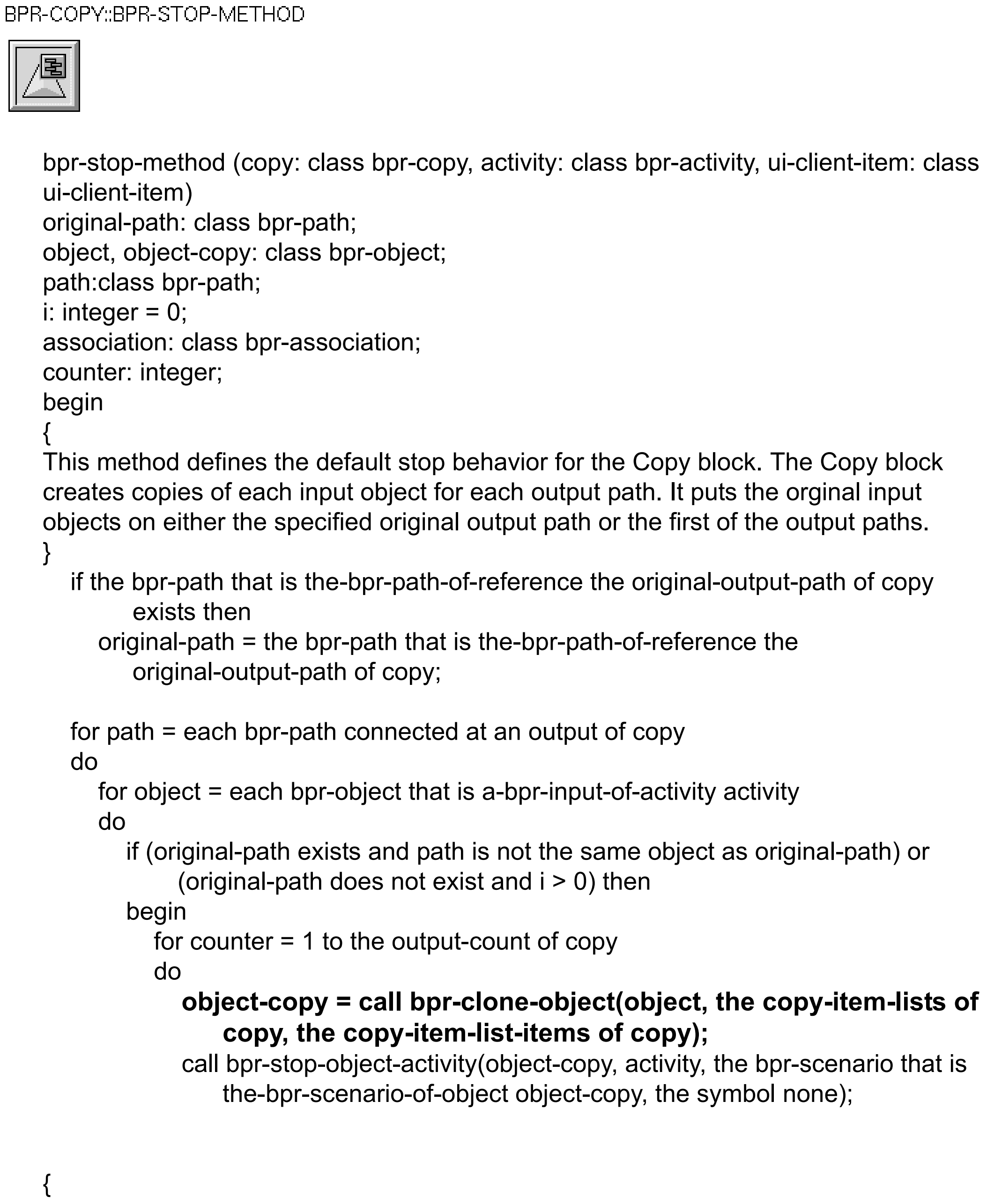
The first version of the procedure clones the contents of attributes that are item-lists and the items within those lists. It also clones associations of the object.

If you do not want the procedure to clone the contents of attributes that are item-lists or the items within those lists, use the second version of the procedure.

Example

The following method is bpr-copy::bpr-stop-method, the method that defines the behavior of the Copy block. The method loops through the block's output paths. For every bpr-object input to the block, the method calls bpr-clone-object to create a copy of the bpr-object for each output path. The method then posts the original and the copied objects to the output paths.

For information about the other API in this example, see bpr-post-path.



```
bpr-stop-method (copy: class bpr-copy, activity: class bpr-activity, ui-client-item: class
ui-client-item)
original-path: class bpr-path;
object, object-copy: class bpr-object;
path:class bpr-path;
i: integer = 0;
association: class bpr-association;
counter: integer;
begin
{
This method defines the default stop behavior for the Copy block. The Copy block
creates copies of each input object for each output path. It puts the orginal input
objects on either the specified original output path or the first of the output paths.
}
   if the bpr-path that is the-bpr-path-of-reference the original-output-path of copy
          exists then
      original-path = the bpr-path that is the-bpr-path-of-reference the
          original-output-path of copy;

   for path = each bpr-path connected at an output of copy
   do
      for object = each bpr-object that is a-bpr-input-of-activity activity
      do
         if (original-path exists and path is not the same object as original-path) or
               (original-path does not exist and i > 0) then
         begin
            for counter = 1 to the output-count of copy
            do
               object-copy = call bpr-clone-object(object, the copy-item-lists of
                    copy, the copy-item-list-items of copy);
               call bpr-stop-object-activity(object-copy, activity, the bpr-scenario that is
                    the-bpr-scenario-of-object object-copy, the symbol none);


{
```

```
If the add to associations attribute of the copy block is true, then add the object copy
to the associations of the object.
}
                    if the add-to-associations of copy then
                        call bpr-add-to-associations(object, object-copy);

                    call bpr-post-path(activity, object-copy, path);
                end;
            end
            else
                call bpr-post-path(activity, object, path);
        end;

        i = i + 1;
    end;
end
```

bpr-continue

Continues a simulation after it has been paused or initially starts a simulation running.

Synopsis

bpr-continue
    (*scenario*: class bpr-scenario)

Argument	Description
scenario	The scenario associated with the model.

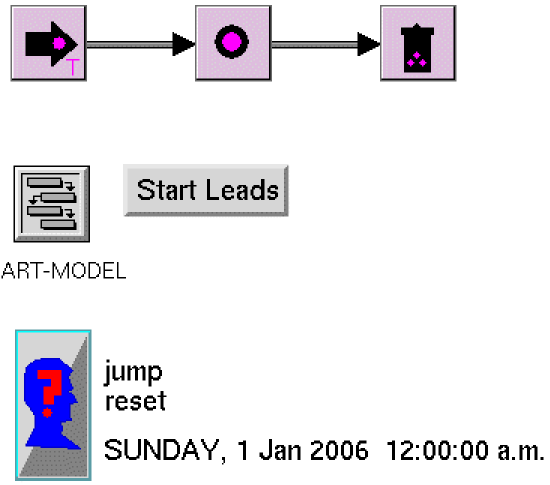
Description

This procedure sets the state of *scenario* to running and changes its color to indicate a running state. Calling this procedure is equivalent to continuing a simulation manually by clicking the Continue button on the subworkspace of *scenario*.

Example

The following custom procedure activates the scenario upon the current workspace, resets the scenario, evaluates the Source block on the current workspace, and continues running the scenario. You activate the procedure with an action button.

For information about the other internal procedures in this example, see bpr-reset, bpr-activate-scenario, and bpr-block-evaluator.



Here is the text of the start-model procedure:

```
start-model (scenario: class bpr-scenario, client: class ui-client-item)
begin
    call bpr-activate-scenario(scenario, the workspace of scenario, client);
    call bpr-reset (scenario);
    allow other processing;
    call bpr-block-evaluator (the bpr-source upon this workspace);
    call bpr-continue(scenario);
end
```

bpr-copy-list-of-attributes

This procedure copies attributes from a source object to a destination object.

Synopsis

bpr-copy-list-of-attributes
(*source-item*: class item, *destination-item*: class item,
operations: sequence, *copy-all-attributes*: truth-value,
add-to-associations: truth-value, *scenario*: item-or-value,
ui-client-item: class ui-client-item)

Argument	Description
source-item	The class name of the source item.
destination-item	The class name of the destination item.
operations	A sequence describing the list of operations to perform. See Description.
copy-all-attributes	Whether to copy all common user-defined attributes from the source to the destination object.
add-to-associations	Whether to add the source object to the list of associations of the destination object.
scenario	The scenario associated with the item.
ui-client-item	A ui-client-item, which is the superior class of: • A g2-window, which is a window in the current G2 session or a remote connection to G2 through a Telewindows client. • A ui-client-session, which is a remote connection to G2.

Description

This procedure is a general-purpose routine for copying attributes from a source item to a destination item. The "copy" operation can be a simple copy of values or a more complex operation that adds, subtracts, multiplies, divides, averages, or applies any user-defined function. Valid operations are: =, +, -, *, *, AVG, and FCT, as a text string.

Optionally, this procedure can copy all user-defined attributes from the source to the destination object before doing any operations, and it can copy the source object to the associations of the destination object.

These operations always store the result in the destination attribute of the *destination-item*. FCT allows you to specify a more complex mathematical operation, such as adding several attributes from the source and destination items and storing the result in the destination object by specifying a user-defined G2 function to call.

The user-defined function can refer to attributes of the *source-item*, *destination-item*, or *scenario* by using this syntax:

the *attribute* of *source-item*

the *attribute* of *destination-item*

the *attribute* of *scenario*

The format of the sequence describing the *operations* is:

```
sequence (
    structure
        (SOURCE-ATTRIBUTE-NAME: the symbol attribute-name,
         SOURCE-SUBTABLE-NAME: the symbol subtable-name,
         DESTINATION-ATTRIBUTE-NAME: the symbol attribute-name,
         DESTINATION-SUBTABLE-NAME: the symbol subtable-name,
         OPERATION: "operator",
         FCT: "function”)
```

Example

Here is the bpr-stop-method of the bpr-task block, which calls bpr-copy-list-of-attributes to copy attributes from the source to the destination work object:

```
bpr-stop-method (task: class bpr-task, activity:class bpr-activity, ui-client-item: class
ui-client-item)
object: class bpr-object;
path:class bpr-path;
counter: integer;
source-object: class bpr-object;
scenario: class bpr-scenario;
begin
{
This method defines the default stop behavior for the Task block. The Task block first
tries to post all of its input objects on the output paths of the block. It then creates and
posts objects for all of the output paths which do not already have an input object
posted. Optionally it may also copy attributes from the input object to newly created
output objects.
}
   for object = each bpr-object that is a-bpr-input-of-activity activity
   do
      source-object = object;
      call bpr-post(activity, object);
   end;

   for path = each bpr-path connected at an output of task
   do
      if not(there exists a bpr-object that is a-bpr-object-of-output-path path) then
      begin
         for counter = 1 to the output-count of task
         do
            object = call bpr-create-object(activity, task, the type of path);

            { Copy attributes ito new type if applicable }
            if the copy-attributes of task and source-object exists and the bpr-scenario
               scenario that is the-bpr-scenario-of-activity activity exists then begin

               call bpr-copy-list-of-attributes (source-object, object,
                   the list-of-operations of task, the copy-all-attributes of task,
                   true, scenario, ui-client-item);

            end;

            call bpr-post-path(activity, object, path);
         end;
      end;
   end;
end
```

bpr-create-object

Creates a new work object.

Synopsis

bpr-create-object
(*activity*: class bpr-activity, *creator*: class object, *object-type*: symbol)
-> *object*: class bpr-object

Argument	Description
activity	The activity related to the scenario that controls the model.
creator	The object from which the new bpr-object is created.
object-type	The type of the output to be created.

Return Value	Description
object	The new object of type *object-type*.

Description

This procedure creates a new bpr-object, whose class-name matches *object-type.* It creates a new class definition for the bpr-object, if none exists, and it places the new definition on the workspace near *creator*. The class-name for the new class definition is *object-type*, and the direct-superior-classes is bpr-object.

The default bpr-stop-method for the Task block and the Batch block call bpr-create-object to generate objects on the output paths of the block. The various procedures that the Source block uses as its source-procedure-name also call this procedure.

Example

Here is the bpr-source-type procedure, which is the default procedure that the Source block calls as its source-procedure-name.

For information about the other API in this example, see bpr-post-path.

```
bpr-source-type (activity: class bpr-activity)
block: class bpr-source;
path: class bpr-path;
counter: integer;
object: class bpr-object;
begin
{
This procedure implements the type operating mode for the Source block. It bases the
output object type on the class specified in the type attribute of the paths.
}

   block = the bpr-source that is the-bpr-block-of-activity activity;

   for path = each bpr-path connected at the output of block
   do
      for counter = 1 to the output-count of block
      do
         object = call bpr-create-object(activity, block, the type of path);
         call bpr-post-path(activity, object, path);
      end;
   end;
end
```

bpr-delete-object

Deletes a work object.

Synopsis

bpr-delete-object
    (*object*: class bpr-object)

Argument	Description
object	The bpr-object to delete.

Description

This procedure deletes a work object and its associated subobjects. It also deletes the association objects related to the work object, if any exist. ReThink uses this procedure internally to delete any work objects that are not posted on output paths. For example, the Sink block calls bpr-delete-object to delete its inputs.

Example

This method is bpr-sink::bpr-stop-method, the method that defines the default behavior of the Sink block. It calls bpr-delete-object to delete the input work objects.

```
bpr-stop-method (sink: class bpr-sink, activity:class bpr-activity, ui-client-item: class
ui-client-item)
object: class bpr-object;
begin
{
This method defines the default stop behavior for the Sink block. The Sink block
simply deletes all of its input objects.
}
  for object = each bpr-object that is a-bpr-input-of-activity activity
  do
    call bpr-delete-object(object);
  end;
end
```

bpr-dequeue-object

Removes a work object from the input path queue of a block.

Synopsis

bpr-dequeue-object
(*path*: class bpr-path, *object*: class bpr-object)

Argument	Description
path	The bpr-path whose path queue contains the object.
object	The bpr-object to be removed from the path queue.

Description

This procedure deletes the specified work object from the path queue of a block. The procedure updates the path statistics and animates the path, if animation is specified.

Example

This partial method is bpr-insert::bpr-stop-method, the method that defines the default behavior of the Insert block. It calls bpr-dequeue-object to remove the input work objects from the path queue when the container object arrives at the block.

```
bpr-stop-method (insert: class bpr-insert, activity:class bpr-activity, ui-client-item: class
ui-client-item)
r: class bpr-path-reference;
path: class bpr-path;
container, object: class bpr-object;
list: class item-list;
queue-object: class bpr-object;
usage: class bpr-usage;
begin
{
This method defines the default stop behavior for the Insert block. The Insert block
inserts input objects received by the block into an item-list of the container input
object. The first input objects in the path queues can be inserted the beginning or end
of the container item-list or all of the waiting input objects can be inserted at the
beginning of the container item-list.
}
```

```
{
Get the container input path and the container input object.
}
. . .
{
For each input object, insert it if it is not the container input object.
}
. . .
{
For each object in the queue of the path; remove it from the queue, insert it into the
container item-list, update its statistics, and transfer the object off of the workspace.
}
              for queue-object = each bpr-object in the queue of path
              do
                 call bpr-dequeue-object(path, queue-object);

                 insert queue-object at the end of the item list list;
{
This simply updates the statistics of the objects in the path queue.
}
. . .
{
If there is a resource that should remain allocated while the object is inserted into the
container, then relate the usage back to the object.
}
. . .
{
Post the container input object on the output paths.
}
   call bpr-post(activity, container);
end
```

bpr-detach-input

Breaks relations between a work object and a path and between a work object and its activity.

Synopsis

bpr-detach-input
    (*object*: class bpr-object)

Argument	Description
object	The bpr-object whose relations are broken.

Description

This procedure breaks the relations named a-bpr-object-of-input-path and the-bpr-input-path-of-object between *object* and its related bpr-path. The procedure also breaks the relations named a-bpr-input-of-activity and the-bpr-activity-of-input between *object* and its related bpr-activity.

The method that defines the default behavior of a Batch block, bpr-batch:: bpr-stop-method, calls this internal procedure prior to inserting a bpr-object into a batch.

Example

This partial method is from bpr-batch::bpr-stop-method, the method that defines the behavior of the Batch block. It calls bpr-detach-input before it inserts a bpr-object into a batch.

```
bpr-stop-method (batch: class bpr-batch, activity:class bpr-activity, ui-client-item: class
ui-client-item)
e, x, g: class bpr-object;
full: truth-value;
usage: class bpr-usage;
TriggerPath: class bpr-path;
BatchPath: class bpr-path;
begin
{
This method defines the default stop behavior for the Batch block. The Batch collects
input objects until a threshold criteria is met. When the threshold criteria is met, it
either posts all of the input objects on the output path or it inserts the input objects into
a container output object which then gets posted on the output path.
}
   for e = each bpr-object that is a-bpr-input-of-activity activity
   do
{
Remove the input object from the path and put it into the batch item-list.
}
      call bpr-detach-input(e);
      insert e at the end of the item-list list the batch of batch;
. . .
end
```

bpr-get-item-for-label

Returns the item with a particular label.

Synopsis

bpr-get-item-for-label
  (*label*: text)
  -> *item-or-value*

Argument	Description
label	A text value that identifies the item.

Return Value	Description
item-or-value	Returns the item associated with the label.

Description

This procedure lets you search for items in a model, based on a label. The *label* argument can be the:

- Label attribute of an object, such as a block, instrument, or model.
- Report-title of a report object.
- Gfr-uuid of any object.
- Names attribute of any object.

Example

bpr-get-item-for-label (“Generate Order”)

bpr-get-item-for-label (“Block Summary Report”)

bpr-get-item-for-label (“custom-scenario”)

bpr-get-item-for-label-class-scenario

Returns the item with a particular label, of a particular class, and associated with a particular scenario.

Synopsis

bpr-get-item-for-label-class-scenario
  (*label*: text, *class-name*: symbol, *scenario*: item-or-value)
  -> *item-or-value*

Argument	Description
label	A text value that identifies the item.
class-name	A symbol that names the class of item to find.
scenario	The scenario associated with the item.

Return Value	Description
item-or-value	Returns the item associated with the label and scenario.

Description

This procedure is similar to bpr-get-item-for-label except that it allows you to search for items of a specified class name that are also associated with a particular scenario. If the scenario is specified, the procedure returns only those items that are related to the scenario, that is, on the same workspace or any subworkspace.

For a description of the *label* argument, see bpr-get-item-for-label.

Example

```
bpr-get-item-for-label-class-scenario (“Generate Order”, my-custom-task,
  “scenario-1”
```

bpr-handle-event-error

Provides a default error handler for scenarios.

Synopsis

bpr-handle-event-error
(*scenario*: class bpr-scenario, *event*: class bpr-event, *error-symbol*: symbol, *error-text*: text)

Argument	Description
scenario	The scenario whose errors the method handles.
event	The bpr-event that causes the error.
error-symbol	The error symbol.
error-text	The error text.

Description

This method is defined on bpr-scenario and is called whenever a run-time error occurs when the model is running. To catch the error and perform custom operations, you can subclass bpr-scenario and provide your own error handler by implementing this method.

Example

For an example, see *`methods-online.kb`* included with ReThink Online.

bpr-indicate

Places an indicator arrow and text next to an item on a workspace to draw attention to it.

Synopsis

bpr-indicate
(*item*: class item, *note*: text, *ui-client-item:* class ui-client-item)

Argument	Description
item	The item to which the arrow points.
note	The text string to display next to the arrow.
ui-client-item	The ui-client-item on which the indicator is displayed, which is the superior class of: • A g2-window, which is a window in the current G2 session or a remote connection to G2 through a Telewindows client. • A ui-client-session, which is a remote connection to G2.

Description

This procedure displays a magenta arrow and text next to an item until you select it, or for ten seconds, depending on the value of the Indicate Mode on the Indicate tab of the Set Scenario dialog. You use the procedure to draw attention to a piece of information, such as the attribute value of an object.

For example, the method that defines the default behavior of an Insert block calls bpr-indicate if you have not identified a container path by choosing the Choose Container Input Path menu choice.

Note You cannot use bpr-indicate to highlight a path.

Example

This customized procedure for an Alarm probe first calls the default stop method for a Sample Value probe. The method then checks the sample value against a threshold, which is an attribute of the custom probe. If the sample value is greater than the threshold, the method pauses the simulation, sends a message to the message board, and calls bpr-indicate to display an arrow and the text “Alarm” next to the probe.

For a description of bpr-pause, see bpr-pause.

For the complete example of creating an Alarm probe, see Creating a Custom Probe.

```
bpr-stop-method (alarm-probe: class alarm-probe, scenario: class bpr-scenario,
object: class object, ui-client-item: class ui-client-item)
begin
   call next method;
   if the sample-value of alarm-probe > the threshold of alarm-probe then
      begin
         call bpr-pause(scenario);
         inform the operator for the next 10 seconds that "Alarm: [the sample-value of
               alarm-probe] > [the threshold of alarm-probe]";
         start bpr-indicate(alarm-probe, "Alarm", ui-client-item);
      end;
end
```

bpr-indicate-connection

Places an arrow and text next to a connection on a workspace to draw attention to it.

Synopsis

bpr-indicate-connection
  (*object*: class object, *connection*: class connection,
   ui-client-item: class ui-client-item)

Argument	Description
object	The object to which the connection is connected.
connection	The connection to which the arrow points.
ui-client-item	The ui-client-item on which the indicator is displayed, which is the superior class of: • A g2-window, which is a window in the current G2 session or a remote connection to G2 through a Telewindows client. • A ui-client-session, which is a remote connection to G2.

Description

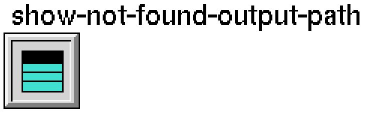
This procedure displays a magenta arrow and text next to a connection until you select it, or for ten seconds, depending on the value of the Indicate Mode on the Indicate tab of the Set Scenario dialog. You use the procedure to draw attention to a connection, such as the not found path of a Retrieve block.

For example, the show not found output path menu choice for a Retrieve block uses bpr-indicate-connection to identify the path to be used when no work object is found.

The text for the magenta arrow indicates if the connection is an input or output connection of the object.

Example

Here is the expression attribute of the user menu choice named show-not-found-output-path for the Retrieve block:



show-not-found-output-path

```
start bpr-indicate-connection
    (the item, the bpr-path that is the-bpr-path-of-reference the not-found-output-path
    of the item, this window)
```

bpr-lookup-by-id

Returns an object of a specific class with the specified ID for the model.

Synopsis

```
bpr-lookup-by-id
    ( class: symbol, id: value, scenario: class bpr-scenario )
    -> object: item-or-value
```

Argument	Description
class	The class of the object to be identified.
id	The id to be matched.
scenario	The currently active scenario of the model.

Return Value	Description
object	The object with the specified id, or the symbol none, if no such object exists.

Description

This procedure finds an object of a specific class with the specified id associated with the current active scenario.

Example

This procedure is bpr-lookup-from-pool-by-association, one of the available default procedures for the procedure-name of the bpr-lookup-subtable of a Retrieve block. This procedure implements the association lookup mode of a Retrieve block.

```
bpr-lookup-from-pool-by-association(subtable: class bpr-lookup-subtable, object:
class bpr-object) = (item-or-value)
block: class bpr-block;
pool: item-or-value;
resource: item-or-value = the symbol none;
associated-objects: class item-list;
found-one: truth-value = false;
begin
{
This procedure implements the association lookup operating mode for the Retrieve
block. It selects an associated object in the resource pool by calling the bpr-get-
association procedure.
}
   block = the bpr-block superior to subtable;

   pool = call bpr-lookup-by-id(the symbol bpr-object, the pool-id of
     subtable, the bpr-scenario that is the-bpr-scenario-of-workspace the
     workspace of block);

   if pool is a bpr-object then
      associated-objects = call bpr-get-associations(object, the lookup-argument of
subtable);

   for resource = each bpr-object in associated-objects
   do
      if resource is a-bpr-member-of-object pool then
      begin
         found-one = true;
      end
      else
         remove resource from associated-objects;
   end;

   if found-one = false then
   begin
      resource = the symbol none;

      delete associated-objects;
   end
   else
      if the retrieve-all of block = true then
         resource = associated-objects
      else
      begin
         resource = the first bpr-object in associated-objects;

         delete associated-objects;
      end;

   return resource;
end
```

bpr-message-to-all-users

Sends a message to connected clients.

Synopsis

bpr-message-to-all-users
    (*msg*: text, *type*: text)

Argument	Description
msg	The text of the message to send.
type	The type of message to send as a text. The options are "WARNING" and "ERROR". A warning sends the message to the Message Board, and an error sends the message to the Logbook.

Description

This procedure sends *msg* to all connected clients of the server, which includes the client application and connected Excel workbooks. The procedure can send the message to the Message Board or Logbook.

Example

```
bpr-message-to-all-users("You should not be working on Saturday!",
"WARNING")
```

bpr-pause

Pauses a simulation.

Synopsis

bpr-pause
    (*scenario*: class bpr-scenario)

Argument	Description
scenario	The scenario associated with the model.

Description

This procedure pauses *scenario* and changes the color of *scenario* to indicate a paused state. Calling this procedure is equivalent to pausing a simulation manually by clicking the Pause button on the subworkspace of *scenario*. Pausing ReThink is not the same as pausing G2.

Example

This customized procedure for an Alarm probe first calls the default stop method for a Sample Value probe. The method then checks the sample value against a threshold, which is an attribute of the custom probe. If the sample value is greater than the threshold, the method calls bpr-pause to pause the simulation, sends a message to the message board, and displays an arrow and the text “Alarm” next to the probe.

For a description of bpr-indicate, see bpr-indicate.

For the complete example of creating an Alarm probe, see Creating a Custom Probe.

```
bpr-stop-method (alarm-probe: class alarm-probe, scenario: class bpr-scenario,
object: class object, ui-client-item: class ui-client-item)
begin
   call next method;
   if the sample-value of alarm-probe > the threshold of alarm-probe then
      begin
         call bpr-pause(scenario);
         inform the operator for the next 10 seconds that "Alarm: [the sample-value of
               alarm-probe] > [the threshold of alarm-probe]";
         start bpr-indicate(alarm-probe, "Alarm", ui-client-item);
      end;
end
```

bpr-post

Locates an output path of the correct type for a work object and sets the work object onto the path. This is called **posting** a work object to a path.

Synopsis

bpr-post
    (*activity*: class bpr-activity, *object*: class bpr-object)

Argument	Description
activity	The bpr-activity related to *object*.
object	The bpr-object posted to the path.

Description

This procedure loops through all the output paths of a block to locate a path whose type matches the *object*, or whose type is a superior class of the Class-name of *object*. When it locates such a path, bpr-post calls bpr-post-path to set *object* onto that path. If bpr-post fails to locate a path with a matching type, it returns without calling bpr-post-path.

For information on bpr-post-path, see bpr-post-path.

The methods that define the default behavior for many of the blocks call bpr-post when the intended path has not yet been identified.

Example

This method is bpr-merge::bpr-stop-method, the default stop method of a Merge block. The method loops through each input work object to the block and calls bpr-post to locate a path of the type that matches the class of the work object.

```
bpr-stop-method (merge: class bpr-merge, activity:class bpr-activity, ui-client-item:
class ui-client-item)
object: class bpr-object;
begin
{
This method defines the default stop behavior for the Merge block. The Merge block
simply tries to post all of its input objects on the output paths of the block.
}
   for object = each bpr-object that is a-bpr-input-of-activity activity
   do
      call bpr-post(activity, object);
   end;
end
```

bpr-post-path

Transfers a work object to an output path of a block when the output path for the object has already been determined. This is called **posting** an object onto a path.

Synopsis

bpr-post-path
  (*activity:* class bpr-activity, *object*: class bpr-object,
   output-path: class bpr-path)

Argument	Description
activity	The bpr-activity related to *object.*
object	The bpr-object posted to *output-path*.
output-path	The path onto which *object* is posted.

Description

This procedure establishes a number of internal relations between *activity* and *object*, and between *object* and *output-path*. Various internal ReThink procedures use these relations to transfer objects between blocks.

After the bpr-post procedure identifies a path of the correct type for *object*, it calls bpr-post-path to transfer the objects between blocks.

If the output path has not yet been determined, use bpr-post to identify the output path whose type matches the Class-name (or a superior class) of the bpr-object. Once the output path has been determined, use bpr-post-path to post the bpr-object to that specific path.

The procedures that define the default procedures for many of the blocks call bpr-post-path when the intended path has already been identified.

Example

This method is bpr-task::bpr-stop-method, the method that defines the default behavior of the Task block. The block first posts the inputs to the output paths for which there are corresponding path types, using bpr-post. If output paths with no corresponding output work objects type exist, the method creates a new bpr-object of the output path type, and then calls bpr-post-path to post the bpr-object onto the output path.

Notice the difference between the use of bpr-post and bpr-post-path; the method uses bpr-post to both locate and post the object to the output path, whereas the method uses bpr-post-path to post the object to the output path when the intended path has already been determined.

```
bpr-stop-method (task: class bpr-task, activity:class bpr-activity, ui-client-item: class
ui-client-item)
object: class bpr-object;
path:class bpr-path;
counter: integer;
source-object: class bpr-object;
scenario: class bpr-scenario;
begin
{
This method defines the default stop behavior for the Task block. The Task block first
tries to post all of its input objects on the output paths of the block. It then creates and
posts objects for all of the output paths which do not already have an input object
posted. Optionally it may also copy attributes from the input object to newly created
output objects.
}
   for object = each bpr-object that is a-bpr-input-of-activity activity
   do
      source-object = object;
      call bpr-post(activity, object);
   end;

   for path = each bpr-path connected at an output of task
   do
      if not(there exists a bpr-object that is a-bpr-object-of-output-path path) then
      begin
         for counter = 1 to the output-count of task
         do
            object = call bpr-create-object(activity, task, the type of path);

            { Copy attributes ito new type if applicable }
            if the copy-attributes of task and source-object exists and the bpr-scenario
               scenario that is the-bpr-scenario-of-activity activity exists then begin

               call bpr-copy-list-of-attributes (source-object, object,
                    the list-of-operations of task, the copy-all-attributes of task,
                    true, scenario, ui-client-item);

            end;

             call bpr-post-path(activity, object, path);
         end;
      end;
   end;
end
```

bpr-remove-from-pool

Removes an object from a pool.

Synopsis

bpr-remove-from-pool
(*pool:* class bpr-object, *object*: class bpr-object)

Argument	Description
pool	A bpr-object with a subworkspace that contains *object*.
object	The bpr-object that is removed from *pool*.

Description

Removes *object* from *pool*, by removing it from the list of items in the pool and from the subworkspace of the pool. The method that defines the default behavior of the Retrieve block calls bpr-remove-from-pool to retrieve objects from a pool.

Example

Here is the partial bpr-retrieve::bpr-stop-method method, which defines the default behavior of the Retrieve block. The method retrieves objects from the pool according to the procedure-name of the lookup-subtable of the block. The lookup subtable determines how the block retrieves objects from the pool.

```
bpr-stop-method (retrieve: class bpr-retrieve, activity:class bpr-activity, ui-client-item:
class ui-client-item)
s: class bpr-scenario;
e: class bpr-object;
p: item-or-value = the symbol none;
r: item-or-value;
o: class bpr-object;
nr: class bpr-path-reference;
original: class bpr-object;
begin
{
This method defines the default stop behavior for the Retrieve block. The Retrieve
block operates in different modes depending upon the procedure named in the
procedure-name attribute of the lookup-subtable. This method invokes the named
procedure and then post the current input object and the retrieved object on the output
paths.
}
. . .
```

```
{
If the add to associations attribute of the retrieve block is true, then add the object
copy to the associations of the object.
}
                  if the add-to-associations of retrieve then
                     call bpr-add-to-associations(original, r);
               end
               else
                  call bpr-remove-from-pool(p, r);
            end;

            if r is not a member of the objects of s then
               insert r at the end of the objects of s;

            call bpr-post-except-not-found-output-path(activity, r);

            call bpr-post-except-not-found-output-path(activity, e);
         end
         else
. . .
end
```

bpr-reset

Resets a scenario.

Synopsis

bpr-reset
    (*scenario*: class bpr-scenario)

bpr-reset
    (*scenario*: class bpr-scenario, ui-client-item: class ui-client-item)

Argument	Description
scenario	The scenario associated with the model.
ui-client-item	The ui-client-item whose model is to be reset, which is the superior class of: • A g2-window, which is a window in the current G2 session or a remote connection to G2 through a Telewindows client. • A ui-client-session, which is a remote connection to G2.

Description

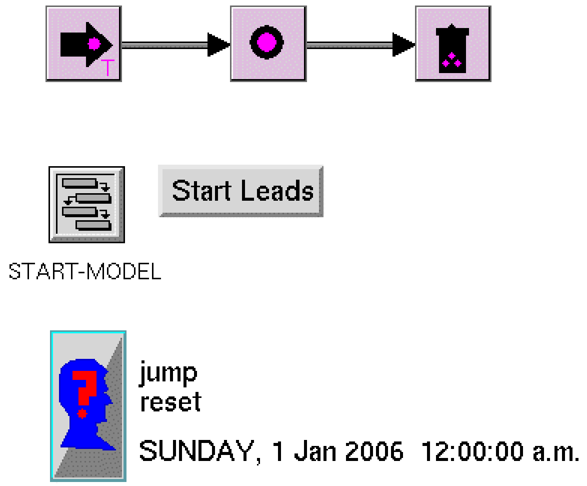
The first version of this procedure resets all of the objects in the model, sets the state of *scenario* to reset and changes its color to indicate a reset state. Calling this procedure is equivalent to resetting a simulation manually by clicking the Reset button on the subworkspace of *scenario*. Resetting ReThink is not the same as resetting G2.

The second version of this procedure activates the scenario, if necessary, and then calls the first version of the procedure.

Example

The following custom procedure activates the scenario upon the current workspace, resets the scenario, evaluates the Source block on the current workspace, and continues running the scenario. You can activate the procedure with an action button.

For information about the other internal procedures in this example, see bpr-activate-scenario, bpr-continue, and bpr-block-evaluator.



Here is the text of the start-model procedure:

```
start-model (scenario: class bpr-scenario, client: class ui-client-item)
begin
    call bpr-activate-scenario(scenario, the workspace of scenario, client);
    call bpr-reset (scenario);
    allow other processing;
    call bpr-block-evaluator (the bpr-source upon this workspace);
    call bpr-continue(scenario);
end
```

bpr-schedule-an-event

Schedules a custom event.

Synopsis

bpr-schedule-an-event
(*scenario*: class bpr-scenario, *event*: class bpr-event)

Argument	Description
scenario	The scenario associated with the model.
event	The bpr-event to schedule.

Description

To schedule a custom event to occur in your model at a specified time, based on the simulation or online clock, you create a subclass of bpr-event and configure these attributes:

Attribute	Description
time	The simulation time at which the event should occur, relative to the start time of the simulation in units of seconds.
priority	The priority of the event.

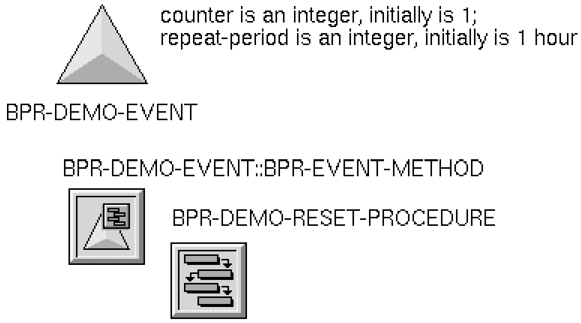
In addition, you must implement the following method for the subclass:

bpr-event-method
(*event*: class bpr-event, *scenario*: class bpr-scenario,
ui-client-item: class ui-client-item)

To schedule the event, call the method bpr-schedule-an-event for the scenario. The bpr-event-method is called when the simulation time reaches the time of the event.

Example

This example schedules an event for a scenario:



counter is an integer, initially is 1;
repeat-period is an integer, initially is 1 hour

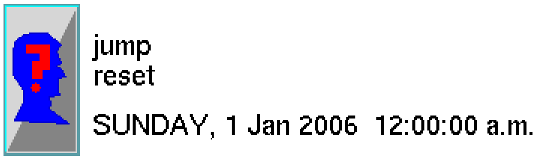
Here is the bpr-event-method for the bpr-demo-event:

```
bpr-event-method(event: class bpr-demo-event, scenario: class bpr-scenario, client:
class ui-client-item)
begin
    if the counter of event = 0 then begin
        post "End of demo";
        delete event without permanence checks;
    end else begin
        conclude that the time of event = the time of scenario + the repeat-period of
            event;
        call bpr-schedule-an-event(scenario, event);
        conclude that the counter of event = the counter of event - 1;
        post "Event counter = [the counter of event].  Current time is [the time of
            sceario as an interval]";
    end
end
```

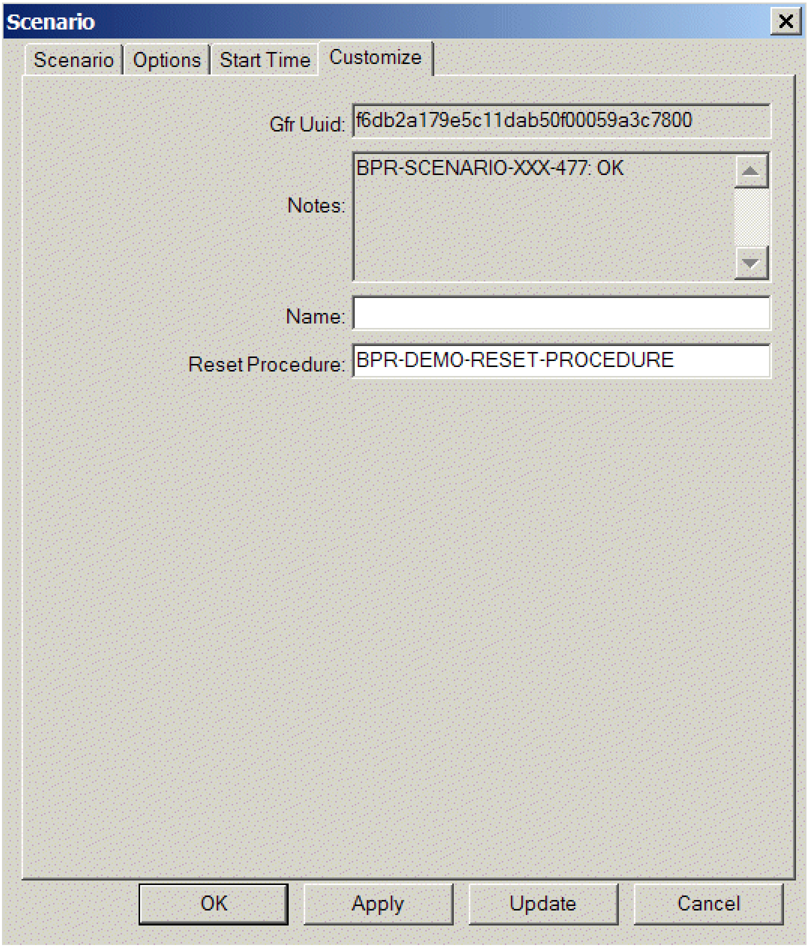
Here is the bpr-demo-reset-procedure, which is the reset-procedure-name of a scenario:

```
bpr-demo-reset-procedure(scenario: class bpr-scenario)
event: class bpr-demo-event;
begin
    create a bpr-demo-event event;
    insert event at the end of the objects of scenario;
    conclude that the priority of event = 8;
    conclude that the time of event = 0.0;
    call bpr-schedule-an-event(scenario, event);
    post "Reset: schedule demo event with period of [the repeat-period of event as an
interval] "
end
```

Here is the scenario that uses the custom reset procedure:



jump
reset
SUNDAY, 1 Jan 2006 12:00:00 a.m.



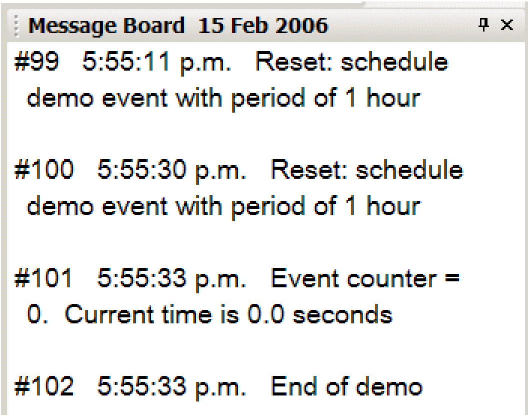
Gfr Uuid: f6db2a179e5c11dab50f00059a3c7800

Notes: BPR-SCENARIO-XXX-477: OK

Name:

Reset Procedure: BPR-DEMO-RESET-PROCEDURE

Here is the Message Board after resetting the scenario and starting the simulation running:



```
#99   5:55:11 p.m.   Reset: schedule
 demo event with period of 1 hour

#100   5:55:30 p.m.   Reset: schedule
 demo event with period of 1 hour

#101   5:55:33 p.m.   Event counter =
 0.  Current time is 0.0 seconds

#102   5:55:33 p.m.   End of demo
```

bpr-update-pool

Adds an object or surrogate to a pool.

Synopsis

bpr-update-pool
    (*object*: class object)

Argument	Description
object	The object that is applied to the pool.

Description

This procedure adds an object to a pool. The resource is either an object of type bpr-resource or a subclass, or it is a surrogate object that you create by choosing create surrogate on a resource.

The default procedure of the store-procedure-name attribute of a Store block calls bpr-update-pool.

Example

This partial procedure is from bpr-store-pool, the default procedure of the store-procedure-name attribute of a Store block. The procedure stops the activity associated with the work object it stores in the pool, then it updates the pool.

The body of the procedure identifies the pool to use and transfers the work object from the workspace to the subworkspace of the pool.

```
bpr-store-pool (activity: class bpr-activity, object: class bpr-object)
block: class bpr-store;
scenario: class bpr-scenario;
ui-client-item: class ui-client-item;
pool: item-or-value;
x: integer;
y: integer;
usage: class bpr-usage;
begin
{
This procedure implements the pool operating mode for the Store block. It stores the
input object in the resource pool specified by the pool-id attribute of the block.
}
   block = the bpr-store that is the-bpr-block-of-activity activity;

   make object transient;
   transfer object off;
```

```
{
Find the resource pool specified in the pool-id attribute of the block. The resource pool is uniquely identified by the id of the resource pool and its associated scenario.
}
    if the pool-id of block does not exist then
    begin
        if the workspace of block exists and the bpr-scenario scenario that is the-bpr-scenario-of-workspace the workspace of block exists and the ui-client-item ui-client-item that is the-ui-client-item-of-scenario scenario exists then
            start bpr-indicate(block, "This store block needs to have a resource pool specified", ui-client-item);

        return;
    end;

    pool = call bpr-lookup-by-id(the symbol bpr-object, the pool-id of block, the bpr-scenario that is the-bpr-scenario-of-workspace the workspace of block);
{
Add the input object to the resource pool by transferring it to the subworkspace of the resource pool and then updating the resource pool.
}
    if pool is a bpr-object then
    begin
        if the subworkspace of pool exists then
        begin
            x = 0;
            y = - the number of elements in the members of pool * (2 * the item-height of object);
            transfer object to the subworkspace of pool at (x, y);
        end;

        call bpr-update-pool(object);
    end;
{
If there is a resource that should remain allocated while the object is stored in the pool, then relate the usage back to the object.
}
    for usage = each bpr-usage that is a-bpr-usage-of-activity activity
    do
        if the deallocate-resource of the bpr-resource-manager that is the-bpr-resource-manager-of-usage usage is false then
        begin
            conclude that usage is not a-bpr-usage-of-activity activity;

            conclude that usage is the-bpr-usage-of-output-object object;
        end;
    end;
end
```

bpr-updated-attributes

Updates attribute values after they have been set.

Synopsis

bpr-updated-attributes
    (*item*: class object)

Argument	Description
item	The class name of the object whose attributes are to be updated.

Description

This method is called whenever attribute values are set through a properties dialog, a report, or a call to bpr-set-attribute-value. You implement this method to perform additional validation or to update additional attributes or objects.

Relations

Describes the relations that ReThink creates and deletes during processing.

Introduction

ReThink creates a number of relations at various stages in its evaluation process, as well as when you create various types of ReThink objects. The methods and procedures that you customize make frequent use of these relations. Therefore, it is critical that you understand what these relations are and when ReThink creates and deletes them.

ReThink creates a number of categories of relations when various events occur during modeling. ReThink creates relations when:

- Blocks evaluate.
- You create a Resource Manager from a resource.
- You add a resource to a resource pool.
- You create or activate a Scenario tool.
- An Associate block evaluates.

For examples of procedures and methods that refer to these relations, see Application Programmer's Interface.

This chapter describes the relations that you are most likely to see during modeling; ReThink creates other relations as well.

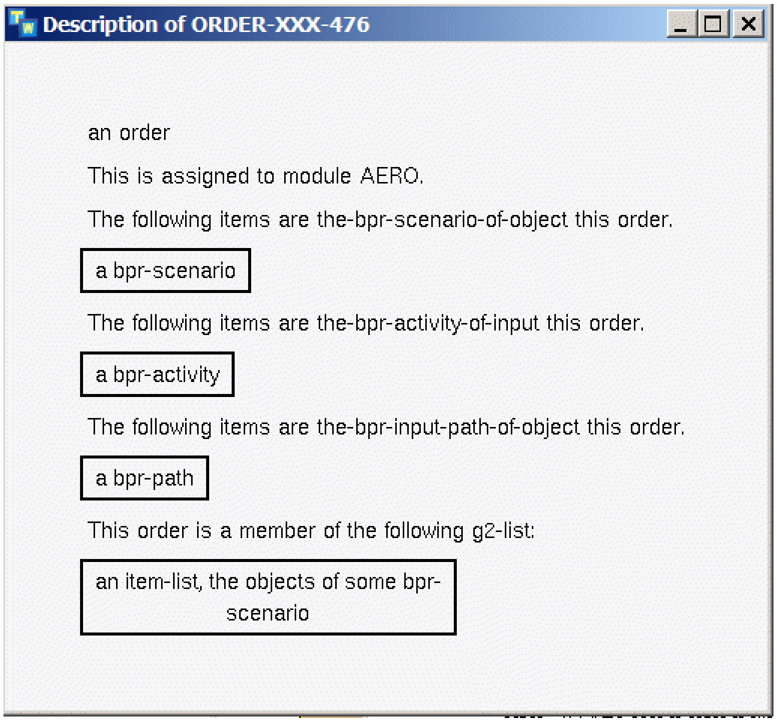
Working With Relations During Modeling

You can see the relations that ReThink creates while a model is running by using the describe facility.

To see the relations that ReThink creates:

➔ In Developer mode, choose describe on an object.

The following figure shows the workspace that appears when you choose Describe on a work object that is currently active. You can mouse-right on the representation of the related object on the workspace to get a menu, as shown. You can then describe the related object, display its table, or go to the object.



an order

This is assigned to module AERO.

The following items are the-bpr-scenario-of-object this order.

a bpr-scenario

The following items are the-bpr-activity-of-input this order.

a bpr-activity

The following items are the-bpr-input-path-of-object this order.

a bpr-path

This order is a member of the following g2-list:

an item-list, the objects of some bpr-scenario

ReThink creates additional internal relations that are not visible because ReThink establishes, uses, and breaks these relations within a single step of block processing, rather than between steps.

Understanding Naming Conventions

ReThink uses standard conventions for relation names:

If the relation starts with...	**The type of relation is...**	**For example...**
a	Many-to-one or Many-to-many	A-bpr-activity-of-block creates a relation between one of many activities and a single block.
the	One-to-many or One-to-one	The-bpr-block-of-activity creates a relation between a single block and one of many activities.

A many-to-one relation means many instances of the relation source can be related to only one instance of the relation target. A one-to-many relation means only one instance of the relation source can be related to many instances of the relation target. For more information on relations, see the *G2 Reference Manual*.

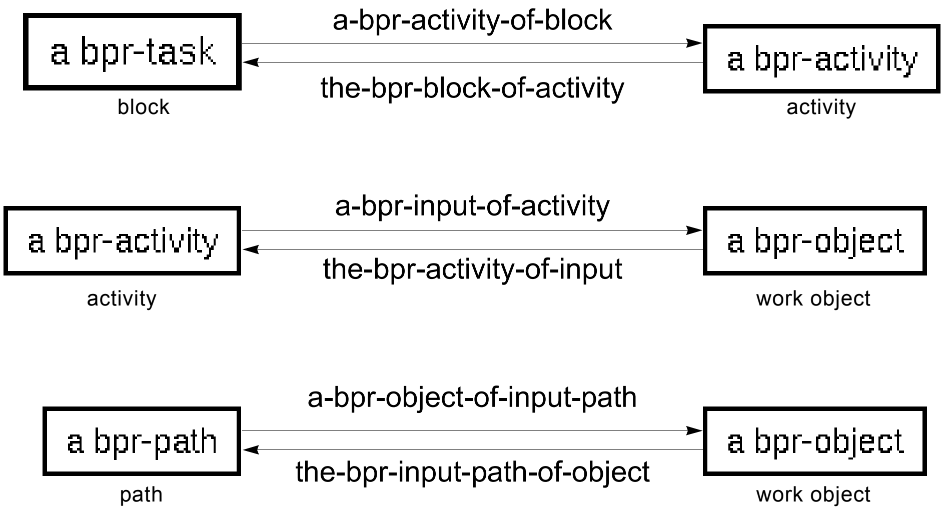
Creating Relations When a Block Evaluates

ReThink creates a number of relations whenever a block evaluates. The relations are between these types of objects:

Blocks
Activities
Work objects
Paths

ReThink creates all of the following relations in the planning phase of block evaluation, as described in The Planning Phase.

ReThink deletes these relations when the block deletes the activity.



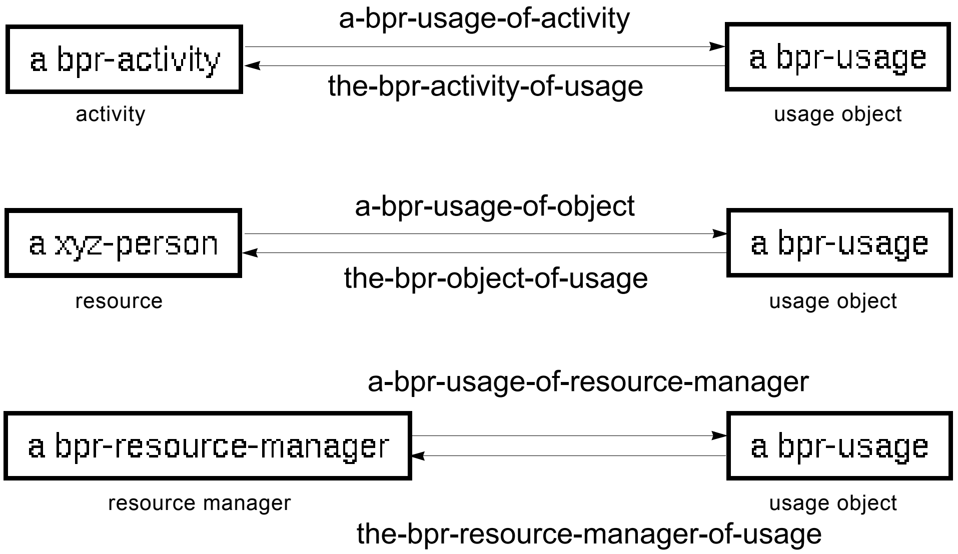
Creating Relations When a Resource Evaluates

ReThink creates a number of relations whenever a block with an attached Resource Manager evaluates. The relations are between these types of objects:

Resource
Resource Manager
Usage object
Activity

A **usage object** is an internal object that ReThink creates to track the utilization of current resources. A usage object is related to a Resource Manager in the same way that an activity is related to a block.

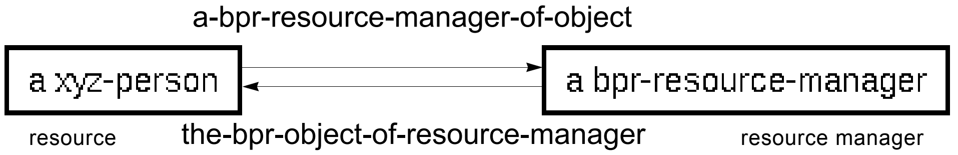
ReThink creates the following relations in the planning phase of evaluation, as described in The Planning Phase. ReThink deletes these relations when the block deletes the activity.



Creating Relations When You Create a Resource Manager

ReThink creates relations when you create a Resource Manager from a resource by using the **create manager** menu choice. The relations are between these two types of objects:

Resource
Resource Manager

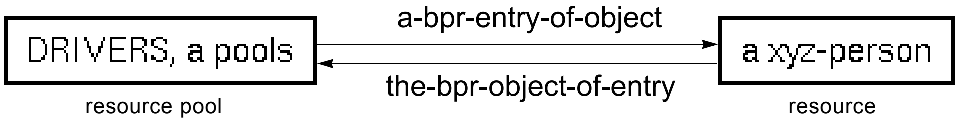

a-bpr-resource-manager-of-object
a xyz-person
a bpr-resource-manager
resource
the-bpr-object-of-resource-manager
resource manager

ReThink deletes these relations when you delete the Resource Manager.

Creating Relations When You Add Objects to a Pool

ReThink creates relations between a resource pool and objects stored in the pool. A pool is a resource with a subworkspace. ReThink creates the relation when the object is transferred to the subworkspace of the pool. The relations are between these types of objects:

Resource pool
Resource



ReThink deletes these relations when you delete a resource from a pool.

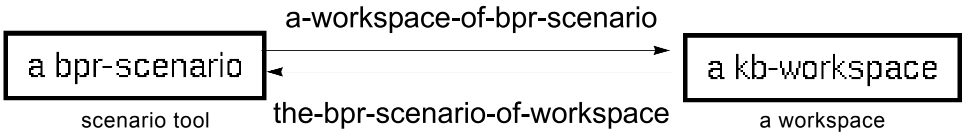
Creating Relations When You Activate Scenarios

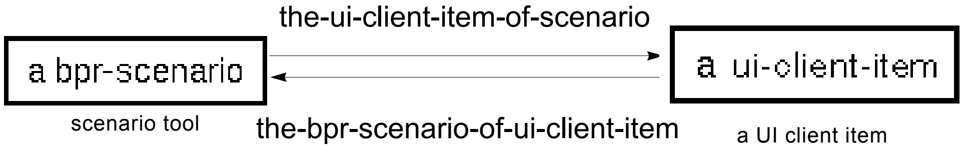
ReThink creates relations when you activate a Scenario tool. If the scenario has a subworkspace, it creates a relation of the same type between the scenario and the subworkspaces of the scenario as well. The relations are between these types of objects:

Scenario
Workspace
UI client item

A ui-client-item is the superior class of:

- A g2-window, which is a window in the current G2 session or a remote connection to G2 through a Telewindows client.
- A ui-client-session, which is a remote connection to G2.



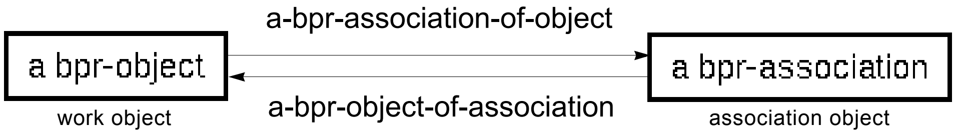

the-ui-client-item-of-scenario
a bpr-scenario
a ui-client-item
scenario tool
the-bpr-scenario-of-ui-client-item
a UI client item

ReThink deletes these relations when you deactivate the scenario.

Creating Relations When You Use the Associate Block

ReThink creates the following relations when the Associate block evaluates. The relations are between these types of objects:

Work object
Association object



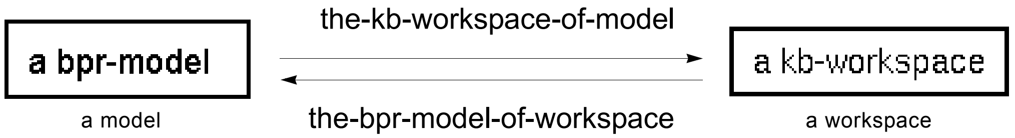
ReThink deletes these relations when a block deletes the associated work object.

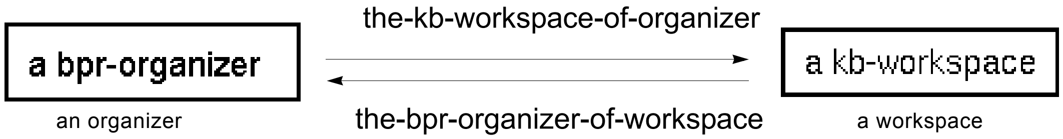
Creating Relations When Replacing Details

You can replace the default detail of a Model tool and Organizer tool with a top-level workspace in a different module, using the Choose Detail menu choice. When you replace the detail, ReThink establishes relations between these types of objects:

Model tool
Workspace

Organizer tool
Workspace

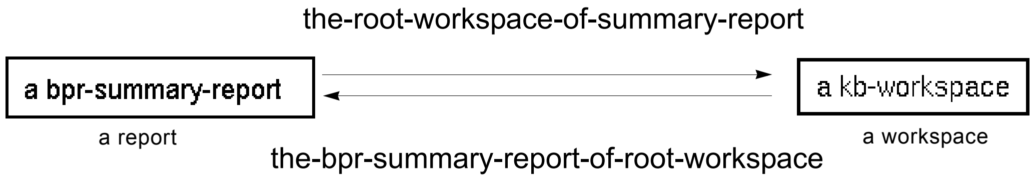




Creating Relations When Choosing the Root Workspace of a Report

You can assign the root workspace of a report by using Choose Root Workspace. When you choose the root workspace, ReThink establishes relations between these types of objects:

Report
Workspace


the-root-workspace-of-summary-report
a bpr-summary-report
a kb-workspace
a report
a workspace
the-bpr-summary-report-of-root-workspace

Glossary

A

allocate: To assign a resource to an activity. By default, Resource Managers allocate resources to activities at random. You can customize how a particular Resource Manager allocates resources to an activity.

animation subtable: A subobject that defines the default colors and animation procedure for a block, work object, resource, instrument, Resource Manager, surrogate, or path.

application programmers' interface (API): Internal ReThink procedures that you use to customize objects. You use these procedures to perform actions on ReThink objects programmatically.

association object: An internal object that ReThink creates when an Associate block evaluates. ReThink creates relations between the association object and the associated objects.

B

block processing: The actions that ReThink performs when a block evaluates. During block processing, ReThink establishes relations between objects, computes cost and duration statistics for objects, animates objects, allocates and deallocates resources, and activates instruments. When you customize ReThink, it is essential to understand the order in which these events take place so that you can write your methods and procedures accordingly.

bpr module: The proprietary core of ReThink. This module contains the discrete event simulation engine and other internal mechanisms that are fundamental to the ReThink environment.

C

class-specific attributes: The attributes of a class that are specific to that class. When you customize ReThink objects, you can add class-specific attributes to custom classes. Typically, you refer to these attributes in the custom methods and procedures that determine the behavior of the object.

cost subtable: Subobject that determines the default procedure that computes total cost for blocks, resources, and work objects.

customiz module: The module in which you create and save customizations to ReThink objects, when you plan on sharing these customizations across different applications of ReThink.

customization: The ability to create ReThink objects that are specific to your particular business process. As a ReThink developer, you can customize such things as the way a ReThink object behaves, the way it looks or animates, the way it computes duration, and the way it computes cost. To customize ReThink, you must be in Developer mode.

D

deallocate: To release a resource assigned to an activity. You can customize how a particular Resource Manager deallocates resources from an activity.

developer: A ReThink user who uses G2, Gensym's core technology upon which ReThink is built, to create custom ReThink objects. Developers create new ReThink objects that are specific to a particular application and are based on existing ReThink objects. Developers work in Developer mode.

Developer mode: The user mode in which you customize ReThink. In Developer mode, the tables for ReThink objects include customization attributes, and the menus include customization menu choices.

duration subtable: Subobject that defines the default procedure that computes duration and utilization statistics for blocks, resources, and work objects. The default duration of most blocks is computed based on a random normal function and represents the amount of simulation time the block has been processing work objects. The duration of a resource is the amount of simulation time the resource has been allocated to activities in the model. The duration of a work object is the amount of simulation time the blocks in the model have spent processing the work object.

G

G2 Foundation Resources (GFR) module: A G2 module that provides the text resource group and local text resources that you edit when you customize the ReThink menus.

M

methods module: The module that contains the class definitions for ReThink objects and subobjects, and the methods and procedures that control their behavior. When you customize ReThink, you create subclasses of these class definitions and copy these methods and procedures. You should not edit the definitions in the methods module, otherwise your edits will be overwritten when you upgrade to a new version of ReThink.

P

posting: Locating an output path of the correct type for a work object and setting the work object onto the path.

R

relation: A type of association between two objects that is not visible. ReThink creates a number of relations at various stages in block processing, as well as when you create various types of ReThink objects. The methods and procedures you customize make frequent use of these relations.

rethink-online module: The top-level ReThink module in which you create your ReThink applications.

S

start method: A method that a block executes before its duration and before it executes the stop method. You can customize the start method of a subclass of ReThink block to provide custom behavior.

stop method: A method that defines the default behavior of a block or instrument. You can customize the stop method of a subclass of a ReThink block or instrument to provide custom behavior.

subclass: A class of objects that inherits part of its definition from another class of object. When a ReThink model processes work objects, ReThink automatically creates subclasses of the built-in bpr-object class when you specify the output path type to be a user-defined object. In addition, when you customize ReThink, you define subclasses of ReThink classes, whose icon, attributes, methods, procedures, and subobjects you customize.

subobject: The value of an attribute that contains an object. ReThink objects define three types of subobjects: animation subobjects, duration subobjects, and cost subobjects, all of which you can customize. The type of subobjects that an object defines depends on the type of object.

superior class: The class from which a subclass inherits. You can specify any ReThink block, instrument, work object, or resource class as the superior class of a custom class.

U

usage object: An internal object that ReThink creates to track the utilization of current resources. ReThink creates relations between usage objects and resources, Resource Managers, and activities. A usage object is related to a Resource Manager in the same way that an activity is related to a block.

Index