Cincom

Cincom Smalltalk™

VisualWorks®

# Source Code Management Guide

VisualWorks 8.1

P46-0138-10

SIMPLIFICATION THROUGH INNOVATION

**Part Number: P46-0138-10**

**Software Release 8.1**



**Cincom Systems, Inc.**

**55 Merchant Street**

**Cincinnati, Ohio 45246**

**Phone: (513) 612-2300**

**Fax: (513) 612-2000**

**World Wide Web: http://www.cincom.com**

# Contents

# About This Book

This document is describes how to configure and use Store, the VisualWorks source control management (SCM) environment. Store is an add-in to VisualWorks that enhances the development tools with facilities for partitioning and versioning code components, and storing them in a database.

This document is still under development. Nonetheless, it is more comprehensive than the documentation formerly provided in the VisualWorks *Application Developer's Guide*.

## Audience

This guide is written for VisualWorks users of any skill level. Since the interface is primarily tools, little specific knowledge of object oriented programming is required. Some parts, specifically the installation section, assume some knowledge of database administration for specific databases.

## Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

### Typographic Conventions

The following fonts are used to indicate special terms:

| Example | Description |
|---|---|
| *template* | Indicates new terms where they are defined, emphasized words, book titles, and words as words. |
| `cover.doc` | Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line). |

| Example | Description |
|---|---|
| ***filename.xwd*** | Indicates a variable element for which you must substitute a value. |
| windowSpec | Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface. |
| **Edit** menu | Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples. |

## Special Symbols

This book uses the following symbols to designate certain items or relationships:

| Examples | Description |
|---|---|
| **File > New** | Indicates the name of an item (**New**) on a menu (**File**). |
| <Return> key<br><Select> button<br><Operate> menu | Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name. |
| <Control>-<g> | Indicates two keys that must be pressed simultaneously. |
| <Escape> <c> | Indicates two keys that must be pressed sequentially. |
| Integer>>asCharacter | Indicates an instance method defined in a class. |
| Float class>>pi | Indicates a class method defined in a class. |

### Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

| | |
|---|---|
| <Select> button | *Select* (or choose) a window location or a menu item, position the text cursor, or highlight text. |
| <Operate> button | Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*. |
| <Window> button | Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*. |

These buttons correspond to the following mouse buttons or combinations:

| | 3-Button | 2-Button | 1-Button |
|---|---|---|---|
| <Select> | Left button | Left button | Button |
| <Operate> | Right button | Right button | <Option>+<Select> |
| <Window> | Middle button | <Ctrl> + <Select> | <Command>+<Select> |

## Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

### Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available.

#### Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help > About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help > About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **Copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

### Contacting Technical Support

Cincom Technical Support provides assistance by:

**Electronic Mail**

To get technical assistance on VisualWorks products, send email to:

helpna@cincom.com.

**Web**

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

**Telephone**

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

## Non-Commercial Licensees

VisualWorks Personal Use License (PUL) is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides an important public resource for VisualWorks developers:

- A mailing list for users of VisualWorks PUL, which serves a growing community of users. To subscribe or unsubscribe, send a message to: vwnc-request@cs.uiuc.edu.

  with the SUBJECT of **subscribe** or **unsubscribe**. You can then address emails to: vwnc@cs.uiuc.edu.

The Smalltalk news group, comp.lang.smalltalk, carries on general discussions about different dialects of Smalltalk, including VisualWorks, and is a good source for advice.

## Additional Sources of Information

Cincom provides a variety of tutorials, specifically for VisualWorks:

http://www.cincomsmalltalk.com/main/products/visualworks/visualworks-tutorials/

The guide you are reading is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincomsmalltalk.com/main/products/documentation/

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

# 1

# Introduction

## What is Store?

Large scale development projects are typically divided among and developed in parallel by a team of programmers. Individual team members work on their own part of the project, and periodically publish their work, making it available to other team members for integration with their code. At this scale of development, source code management and control is essential.

Store is an add-in component to VisualWorks that provides source code version management and team development facilities for the base development environment. Store provides:

- Source code repository support, retargetable to several common database backends.
- Tools for versioning units of code, including branching versions and browsing version histories.
- Tools for merging, comparing changes, and reconciling divergent lines of development.
- A simple and extensible team development methodology.

## Store Features

Store is a source code management and versioning system that is integrated into the VisualWorks environment. In VisualWorks, class, method and other definitions are organized into packages and bundles. When Store is added to the system, these same packages and bundles become versionable storage units in a database. Changes to the code in a package can then be published with incremental version identifiers.

The standard VisualWorks browser is extended by Store to simplify publishing and loading packaged code. Additional tools are provided for managing the packages in the repository, and performing operations such as comparing versions in the repository with the image, browsing version histories, and so on.

Teams coordinate their work by sharing packages via the server-based code repository. The tool set provides each developer with a client view of the repository. Unlike repository systems that use a check-out/check-in mechanism to ensure that only one developer is modifying code at a time, Store employs a merge mechanism. This approach allows several developers to “own” their own versions developed from a common version of the code, and publish their work simultaneously. At appropriate points in the development process, any versions that have diverged are merged and published as a unified version.

In a common process, team members begin by loading code from the shared repository into their local development image. As the developer modifies code, Store records a fine-grained version history in a repository-specific changes log, and marks the packages as candidates for publishing. The developer periodically publishes the modified packages, to record the changes and make them available to other members of the team. As needed, a code “integrator” reviews the changes, merges divergent versions, and republishes a new common version.

## Code Storage in Store

The Store code repository is implemented using an open, retargetable, database access strategy, enabling its use with a variety of popular, commercially available databases systems (e.g., Oracle, SQLServer, PostgreSQL, SQLite3, and DB2). Store interfaces with standard, well-understood, robust, transactional systems for which

administrative expertise and tools already exist in most companies, rather than a proprietary repository or flat-file system. Being retargetable, the tools allow access to several different repositories, with different database back-ends, during the course of a single project, from a single image.

Store supports geographically distributed and mobile teams. The versioning strategy does not demand a continuous connection to the code repository. Regardless of whether developers have high-speed LAN access on site, or slow modem access while travelling, Store's architecture enables the entire group to work together smoothly in a wide-area network environment.

## Concurrent Development

Store uses a "publish and merge" model for version control. Under this model, Store creates a local copy of the code under development when it is loaded into the developer's image. This copy is known as a working or child version, as opposed to the parent version in the repository.

Changes to a working version do not affect the parent version, but are tracked as "deltas" or "branches" from the parent version. For increased performance, only these deltas are saved when a component is published in the repository (unless published "binary," as described later). The parent-child relationship between versions helps to simplify the task of merging multiple lines of development into a single, consistent version. After publishing a branch, the new version can be merged at any later time.

This architecture provides two important benefits. First, the model is well-suited to a transient network environment. Once a version of a code unit has been loaded, no further network access is needed for the ordinary activities of development. Under Store, developers only need to be connected to the repository to load or publish updates.

Second, because there is no need for locking, Store promotes a more parallel workflow within the development group. The logical organization of a project can be preserved, allowing developers to work together closely. In short, the publish and merge design is more suitable for a high productivity environment like Store.

# A Development Methodology

One of the key benefits of a source control management system is that it brings a formal methodology to the often confused process of software development. Store provides a simple but extensible framework for defining a group development process.

Development is organized around the traditional idea of milestones, such as base-lining, coding, integrating, and releasing. As a project progresses from one milestone to the next, Store tracks the version history of its parts. Development groups can either use a set of pre-defined milestones or define their own.

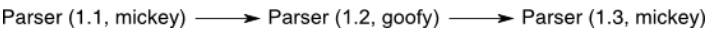
To simplify the task of managing parallel lines of development, a fine-granularity versioning technique is employed. During normal development, all changes are logged locally, and Store records how they impact the working versions of packages in the local image.

## Versioning

Versioning is not merely a way to track the change history of a unit of code, but a way to provide the needed insulation between different lines of development. Store provides developers with a versioning strategy and several powerful tools for managing both simple and complex version graphs.

When a package or bundle is first published in the repository, a new thread or line of development is established. A version string is created (e.g., “1.1“) at this time. The version stored in the repository is known as the parent version, while the copies created in the local image during loading are known as the child or working versions.

As work proceeds, the line of development is extended. For example, the following simple graph shows three successive versions of the “Parser” component:

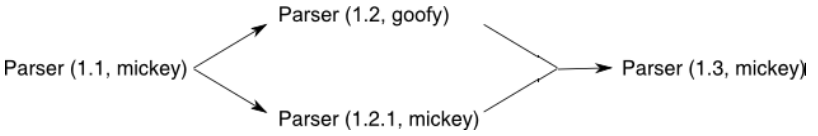



When an updated version is published, a new version string is assigned (e.g., “1.2”), and this version becomes parent of the working version in the image. The working versions in each developer’s image only have version numbers assigned when they are published. They can be tracked and merged because they are all descended from the same parent.

With Store, complex applications can be versioned easily. When a new version of a bundle is published in the repository, any changed sub-bundles or packages contained within the bundle are also published, creating new versions.

## Parallel Development

If the line of development is completely linear, it is not necessary to merge different published versions of the same unit of code. Development in a team setting is seldom linear, however, and a more complex version graph may have several branches for a single unit, each representing a parallel thread of development.

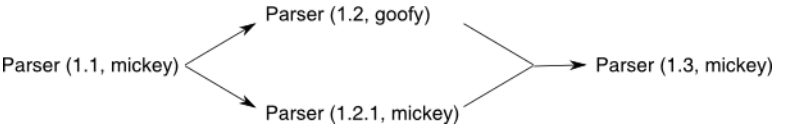



In this case, two developers are working simultaneously on the same package. After they publish their modifications with successive version numbers (“1.2” and “1.2.1”), it will be necessary to perform a step of integration.

It is often useful to create a version based upon a change set. Store supports these “code fragments” as a way to quickly produce small fixes without republishing an entire application. These fragments are full-fledged branches in the version graph.

With two or possibly more developers working on the same group of packages, the task of integration can rapidly become complex. Store simplifies this step in several ways. First, by representing the version graph as a series of deltas in a parent-child relationship, it is easier to pinpoint conflicts between the parallel threads. Second, Store provides a Merge Tool that largely automates the task of package integration. The tool can identify and semi-automatically resolve all points of conflict in an arbitrary n-way merge.

The following version graph illustrates the effect of merging a branch:



When merging two versions of the same package, Store first identifies any conflicts. If the same definition has been changed in different ways (e.g., a method has been added or removed from the same class), a potential conflict is identified. If a change were made to only one version, or if the changes in both versions are the same, Store considers the modifications to be nonconflicting.

The Merge Tool automates the task of integration by identifying conflicts and providing a simple mechanism for resolving them in a new composite version. A list of conflicting definitions is provided, and the option is given either to select one of the definitions or else create a new one. By default, nonconflicting modifications are excluded, but the Merge Tool can audit these as well. Once all conflicts are resolved, a new version is published.

## Blessing levels

As individual parts of a project reach each milestone, they are approved by the appropriate team members before proceeding. This practice—often referred to as “promotion management”—has the virtue of making it easier to coordinate a team around common development objectives.

Under Store, promotion is structured via a series of blessing levels that represent the following steps (in fact, Store includes a few more, but we need only consider the six main ones here):

1 in development

2 published

3 integrated

4 merged

5 tested

6 released

Blessing levels may be thought of as special annotations to component versions. They provide notations of quality in an otherwise unstructured version graph. These notations coordinate work on a component through all stages of development.

Consider, for example, a conventional versioning scheme without notations of quality. When version numbers alone are used to indicate quality, later versions are generally assumed to be better

than earlier ones. Of course, in practice this is often not the case, especially during early stages of development when functionality is incomplete.

By separating the number of a version from the notation of its quality, Store helps to eliminate this confusion. Parallel development can proceed apace, code can be extended through broken or incomplete phases, and branches can be merged as necessary.

### Publishing policies

Blessing levels can also be used by the Store tools to enforce certain rules of process. For example, a development group may decide that only those packages that have reached a certain level may be integrated using the Merge Tool.

Publishing in the repository may also be controlled using blessings. When the repository is configured for user/group management, only the owner of a package or the repository administrator is allowed to publish above the normal development levels. Similar rules restrict the “tested” level to members of the QA group.

Thus, blessings provide several important benefits: first, they facilitate tighter coordination between team members by indicating when code is ready to be shared, integrated, tested, etc. They encourage developers to publish and share intermediate stages of a package.

Second, they enforce rules of process without placing unnecessary constraints on publishing, and finally, they help shield each member of the development team from untested packages by providing “insulation” between parallel lines of work.

For organizations that wish to design their own methodology, Store provides a simple means for customizing the set of blessing levels. The name, number, and semantics of the blessing structure may all be changed by creating new blessing policy classes.

## Database limitations

Because Store depends on third-party databases for data storage, the limitations of those databases apply, and may appear to be limitations of Store when they are not.

For example, database platforms commonly limit field names to 255 characters. This limit applies to Store as a limit on the sizes of method, class, name space, and shared variable names. The

limitation applies only to simple names, not including the (name space) environment, so it is seldom a problem. Some users have experienced trouble, however, with long message selectors.

# 2

# Beginning to Use Store

## A Simple Approach

Store is chock full of options and alternatives, with flexibility to put a burlesque show contortionist to shame.

What we're going to do in this section is walk through a simplified scenario of setting up a base image, importing and packaging code, publishing a couple of versions, and a quick integration. This will not illustrate all the configuration options of Store, and it won't answer specific questions about how to package your code. But, it will show a way of going through a simple development and release process with Store.

Full, vendor-specific instructions for setting up a database and the requirements for users in enterprise environments are given in Configuring Store.

### Assumptions

For this extended example, we are assuming that you have access to a database into which you have rights install Store tables. What this assumption amounts to is that you have:

- A Store-compatible database installed
- Database rights to create three database users:
  - **BERN**, as the Store table owner
  - **BaseSystem**, a regular user that will be used only for publishing VisualWorks base packages
  - **YourID** (whatever your database login ID is), a regular user that you will use for logging in to load and publish your

application code

- You know the access string (sometimes called environment) or directory path for the database, which is assigned when creating the database

Only the database-level instructions are necessary, because we will install the tables for Store here. If the tables are already set up, then you can skip that part of this walk through.

> **Note:** There is nothing special or magical about the names BERN or BaseSystem — any name will do. These specific names are used throughout the following chapters for example purposes.

## Install Store into VisualWorks

For this section, we'll assume that you do not have Store already loaded into a VisualWorks image. If you already have a Store image, you can skip to the next section. Or you might want to read through quickly to see what was done.

Store is an add-on to VisualWorks that is installed from parcels. There are several parcels for Store, but you only need to pick one to load; the rest are installed automatically. Once the Store add-on has been installed, we will cover its configuration in a subsequent topic.

The parcel you will choose to install is the Store component for the kind of database you have available. At present, the options are:

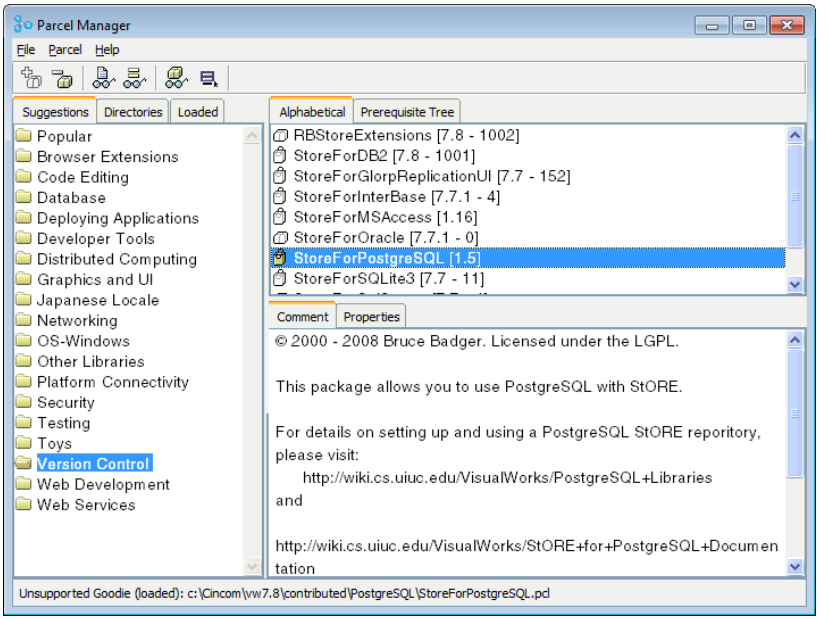
- StoreForOracle
- StoreForSqlServer
- StoreForPostgreSQL
- StoreForDB2
- StoreForInterbase
- StoreForSQLite3

Oracle, SQLIte3, DB2, and SQLServer are the only databases officially supported by Cincom. PostgreSQL and Interbase support parcels are included with VisualWorks as "contributed," provided and supported by third-party developers. For more information about these, browse the /contributed directories for documentation files.

To install Store into VisualWorks:

1 Launch a clean VisualWorks image (visual.im or visualnc.im).

2 In the Visual Launcher, select **System > Parcel Manager**.




3 In the **Suggestions** list, select **Version Control**.

4 In the list of parcels, select the **StoreFor...** parcel for your database, and select the **Parcel > Load** command.

5 Wait while Store loads.

6 Save the image to a new name, such as: **storeOnly** (in the Launcher window, select **File > Save Image As...**). The image file will be saved as storeOnly.im.

## Install the Store Database Tables

This discussion assumes you have a database installed, but no Store tables have yet been created in it. If you are accessing a database that already has Store tables installed, skip this section.

The installation here is simple, and suitable for a single-user database, such as one that you would use as a local repository, because it does not install the user/group management facility. This is also suitable for larger groups who are trusting, that is, that process is not tightly controlled.

The following procedure is general, but slanted towards Oracle. For vendor-specific instructions (e.g., PostgreSQL, SQLite, etc.) and details on configuring a controlled, multi-user workgroup or enterprise environment, refer to Configuring Store.

To install the Store tables:

1 Launch your Store image (storeOnly.im from the previous section).

2 In a workspace, enter and evaluate (Do It) this expression:

```
Store.DbRegistry installDatabaseTables
```

3 When the Store connection dialog opens, log in as **BERN**, the Store table owner (the account name assigned by your database administrator might be different). You also need to:

- select the connection type and
- enter the database **Environment** string that you got from your database administrator (for a local database, you can usually leave this empty).
- enter the database table owner ID in the **Table owner** field (for databases that have table owners, such as Oracle and SQL Server). This ID will then become the table owner.

4 If you are using Oracle or SQLServer, when prompted **Create tablespace?**, click **Yes**.

5 If you are using Oracle or SQLServer, when prompted for the database directory, enter the directory path name created for Store by the database administrator.

6 When you are prompted for a name for the store database, enter a name that will uniquely identify this Store database within your organization, and click **OK**.

This identifier is used for identifying this database to Store. If you have only one database to access, you may call it simply “store”. If you access two more Store databases in your organization, they must have different names. We suggest embedding the respective server or domain names in the database identifier.

7 When you are prompted whether to **Install management policies?**, click **No**. Click **OK** to dismiss the next notifier.

You can always install these later (see Setting up development behavior of users and groups.

8 When finished, disconnect from Store (**Store > Disconnect from Repository**). You don't want to work while logged in as the Store table owner.

The Store database tables are now installed, and you are ready to begin publishing.

## Publishing the Base

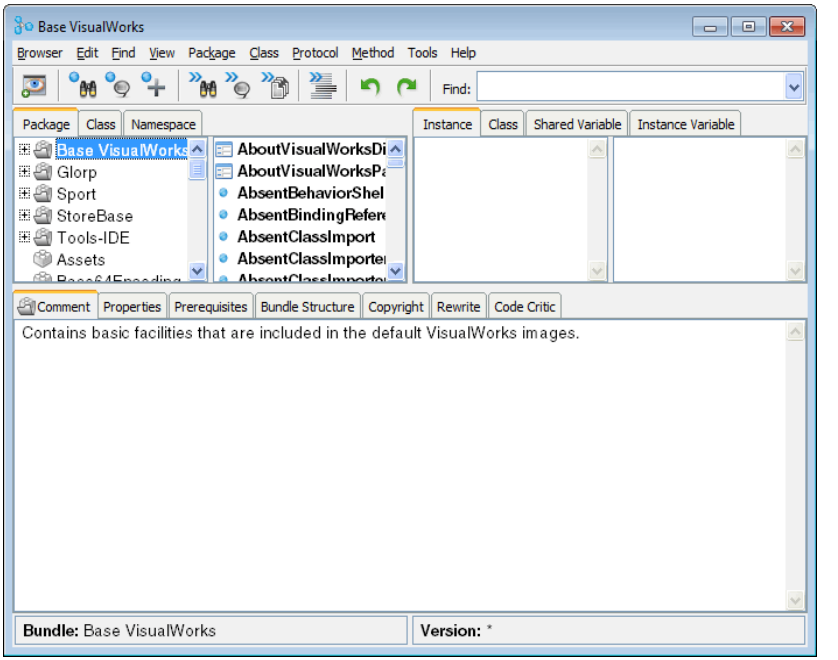
The first thing many Store users do is publish the VisualWorks base into their Store repository. It isn't necessary, but does provide some help in discovering if you have accidentally overwritten a method in the base, which most application programmers do not need to do.

We'll skip doing this at this point, but refer to Publishing the VisualWorks Base.

If you want to publish the base, this is the time to do it, though you can also do it later. It takes a while, and can make your database files pretty big, though, so be prepared.

# Explore the System Contents

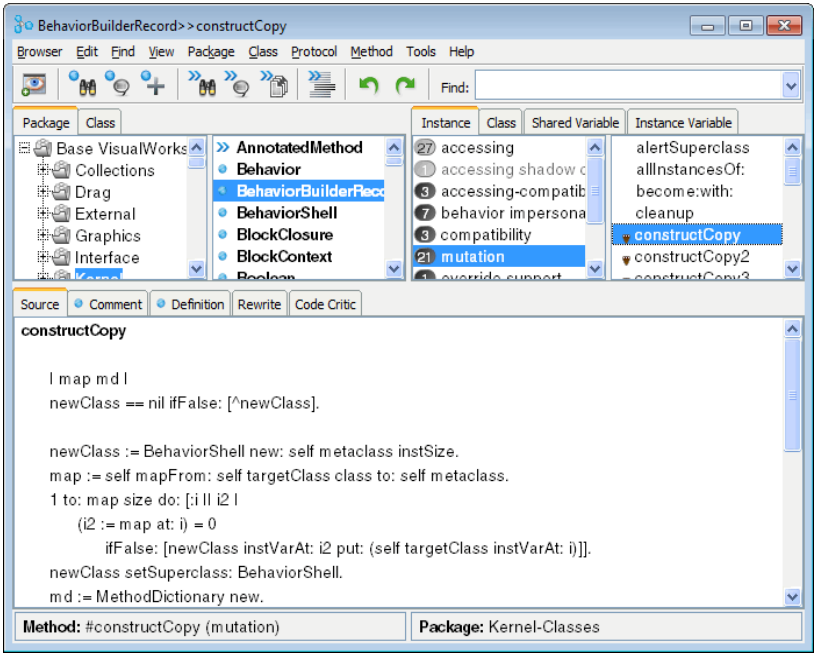
Before going further, let's take a quick look at the system as it stands now. Open the System Browser.




This browser is described in the VisualWorks Tools Guide and in the online Help. So, here we only point out features we're interested in for Store.

The top-left pane I'll call the "package list," even though it shows both packages and bundles. Bundles are the expandable ones. Click on an expansion button ( **[+]** ) to expand a bundle, such as Base VisualWorks, and continue expanding until there are no more expansions. At the end of the trail are the packages, which contain the actual code definitions.

If you select any of the bundles, the code definitions in the packages contained in that bundle, no matter how deep, are shown in the remaining panes: class, method category, message selector, and code definition at the bottom.




By selecting either a package or a bundle containing the package, you can narrow and widen the scope of classes displayed in the browser.

By providing for nesting of packages within bundles, and possibly bundles within bundles in this way, packages can be kept quite small and tightly focused, while allowing an easy way to group chunks of code for viewing and maintenance. Select packages and bundles up and down the bundle hierarchy to see how the classes are available for view.

Selecting the **Base VisualWorks** bundle, you can browse all of the familiar base classes. The sub-bundles are named similarly to the traditional class categories, so this should look familiar as well. An additional top-level bundle contains Store support, and then there are several loose packages containing other features.

Note that the packaging of the VisualWorks system classes was done automatically when you loaded Store. Similar automatic packaging is done when you load your own code, which we'll do next.

## Load Application Code

In Importing Code into Store we describe several ways for packaging existing code.

In this section we will load some existing code from parcels. Goodies are good for this kind of thing because they are available, and can be freely edited. Let's use the HotDraw goodie from John Brant.

### Loading Parceled Code into Store

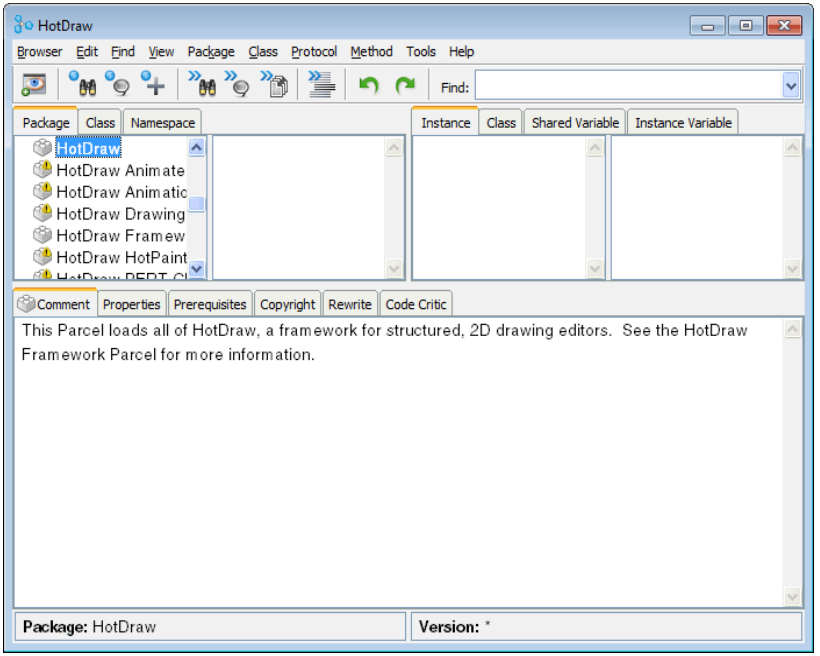
The HotDraw parcels can be loaded using the Parcel Manager. Your own parceled application code can also be loaded using the Parcel manager as long as it is in a directory on the VisualWorks parcel path, though you may have to use the **Directories** list. Otherwise, you will have to use alternative methods of loading parcels, as described in the Application Developer's Guide.

To load HotDraw, open the Parcel Manager (**System > Parcel Manager**), select **Graphics** on the **Suggestions** page, select **HotDraw**, and then pick **Parcel > Load**. This one parcel loads all of the other HotDraw parcels, which it specifies as prerequisites.

When HotDraw has finished loading, it opens up an information workspace. Go ahead and close that; we won't be needing it.

Now open a System Browser and browse the results.

In the browser **Package** list, scroll down to find the HotDraw packages. They are all packages at this point.




The first thing to notice is that the HotDraw package names are exactly the same as the names of the parcels that we loaded. This is Store's default way of loading parcels; to create packages exactly corresponding to the parcels.

Select the top HotDraw package, and notice that there are no classes in it. It does, however, specify prerequisites, which identify the other packages. These are directly inherited from the parcel. Click on the **Prerequisites** tab to see these.

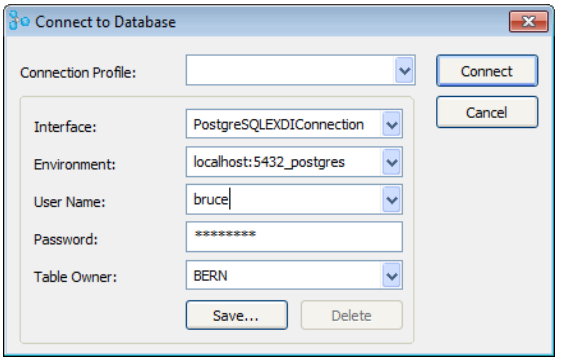
Since we loaded HotDraw as a parcel, the prerequisites are parcels, which in Store can be thought of (approximately) as the deployment counterparts of packages. When you deploy a package, you publish it as a parcel. As for the original HotDraw parcel, the HotDraw package, when deployed as a parcel, will make sure all of its prerequisite parcels are loaded before loading any code it contains (of which there isn't any).

Before any further work, we should publish the packages we have now.

## Publishing Packages

The HotDraw packages have never been published, so they are not yet under version control.

To publish, first we must connect to the Store repository. To connect, select **Store > Connect to Repository...** in the Visual Launcher.



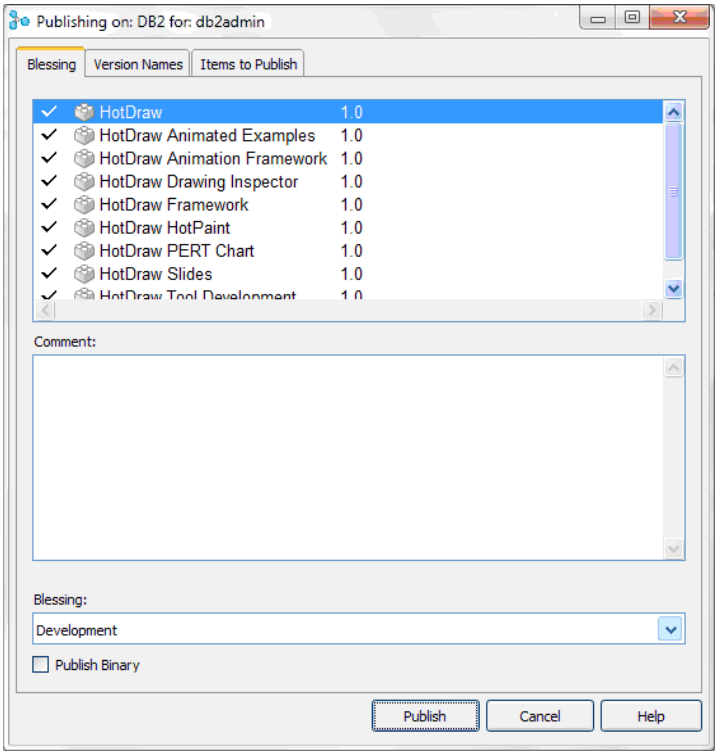

Previously, you have only logged in as the table owner (and possibly as the BaseSystem, if you published the base). Now log on using the same environment string, but use your own user name and password.

The **Table Owner** field may show the table owner, for databases that define owners (e.g., Oracle, DB2, and SQL Server). For installations with more than one Store repository in a single database, the table owner identifies the specific repository. Enter or select the table owner for your repository.

With the login information entered, click **Connect**.

You can save these, and alternative settings, as a **Connection Profile**. Click **Save** and enter a profile name. This is particularly useful if you frequently connect to alternate databases; you only need to select the profile next time you want to connect.

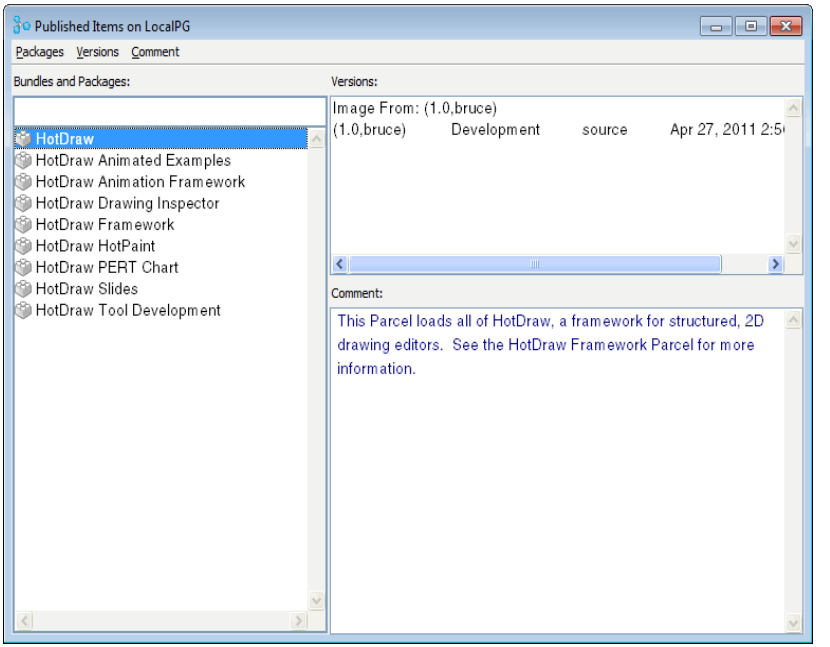
Now that you are connected to the repository, return to the system browser. Select all nine of the HotDraw packages in the package list (Shift-click to select a range of packages, Ctrl-click to add one more to the selection). Then select **Package > Publish...** This opens a Publishing Package dialog, showing all of the selected packages:




You can now publish all packages at once, but they will each be saved separately, since they are not grouped in a bundle. We'll do that later. For now, publish the packages. The version number of **1.0** is fine for now, and leave the **Blessing level** at **Development**.

Change the **Comment** to indicate that these are published from the original parcel. Then, click **Publish**. A progress dialog is displayed while publishing that displays a notification for each package as it is individually published.

Now we have versions of HotDraw packages in our repository, and we can begin working with them. To see them in the repository, select **Store > Published Items** in the Visual Launcher to display the published items list.




Exit VisualWorks without saving the image. We will load the packages into our image from the database.

## Loading and Reorganizing HotDraw

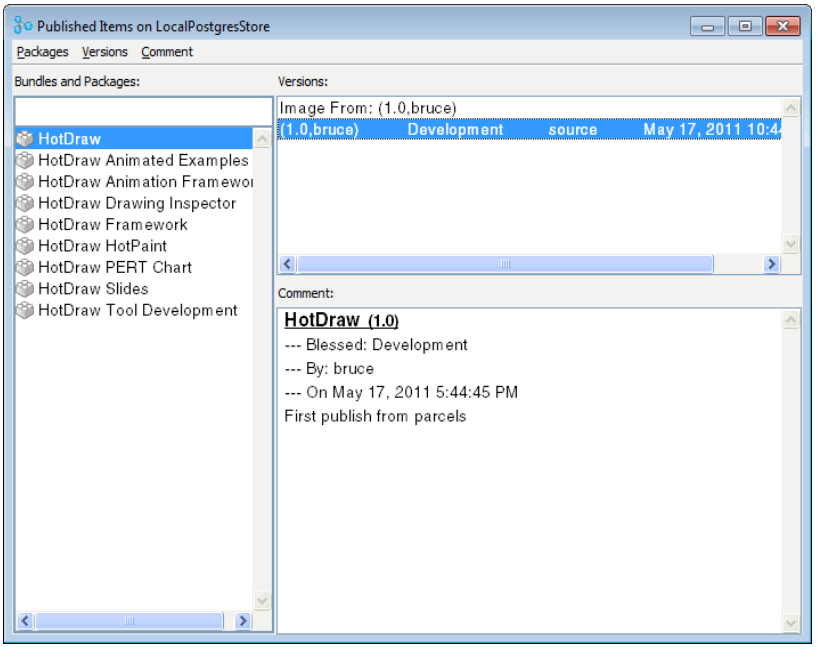
We left off after you published the HotDraw packages and exited your image. Now we need to relaunch the image and load the packages.

1 Launch your **`storeOnly`** image.

2 Connect to your Store repository using your normal user ID.

3 Select **Store > Published items**.

4 Select the HotDraw package, and then the only version of the package published so far.




5 Select **File > Load** (or **Load** on the <Operate> menu).

After a few moments, the package is loaded. Open a system browser and take a look.

Notice that only the HotDraw package is displayed. Select it, and it has no contents. But, the original HotDraw parcel loaded everything! What happened? Why didn't it load all the other packages like the parcel did?

Click on the **Prerequisites** tab for HotDraw. The prerequisites are all there, but they are specified as being **Applicable for Parcel Only**.

What happens is this. If you were to publish this package as a parcel (which we'll get to later), you would create a perfect duplicate of the original, and it would load the other parcels as before. In the case of HotDraw, though, its prerequisites are set to be applicable only when loading as a parcel. (This corresponds to the difference between *development* and *deployment* that existed in VisualWorks prior to version 7.7.) We'll explore how to change all of this later.

We could change the **Applicability** of these prerequisites, but it is better to organize these packages using a bundle.

Either way, we need to load the other HotDraw packages first.

So, go back to the Published Items list and select the next package, HotDraw Animated Examples, and load it. Click **OK** on the dialogs and close the workspace that opens. Now, look at the packages in the browser.

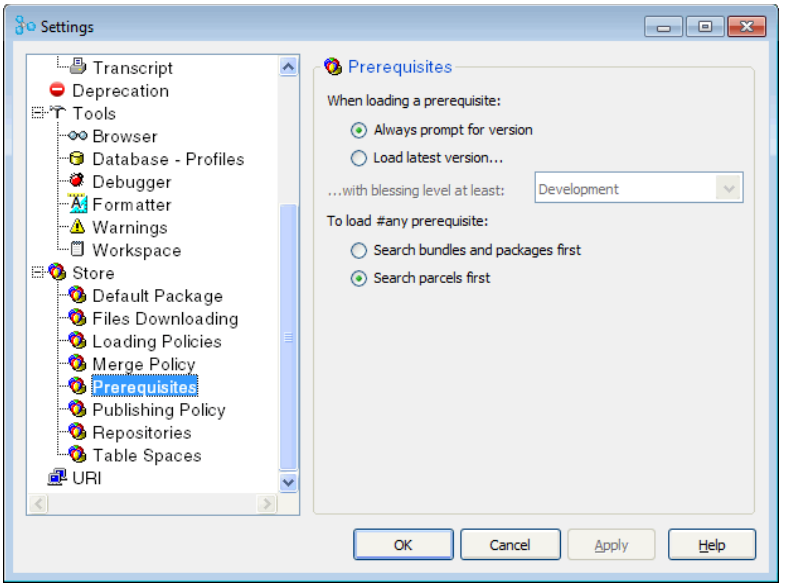
We still don't have all of HotDraw, but three packages are loaded. However, those dialogs popping up didn't look right. Select the first of the newly loaded packages in the browser, HotDraw Animated Examples, and notice the status bar, and/or look at the Information property page. It should say something like **(1.0,yourname)** for the version. Now click on the next package, HotDraw Amination Framework. It probably attributes publication to someone else. The same goes for HotDrawFramework.

Those last two are not your packages, which should also say **(1.0,yourname)**. Instead, loading the HotDraw Animated Examples package loaded two parcels to satisfy its prerequisites!

To see how this happened, first look at HotDraw Animated Examples in a System Browser. Select the **Prerequisites** tab, place the cursor over the **Current** prerequisite, and examine the **Applicability** setting using the <Operate> menu. It should be set to **Either**.

Each prerequisite can specify either **Parcel Only** or **Store Only** to satisfy the prerequisite, or **Either** indicating that the prerequisite can be met by either a parcel or a package. In our case, there are both a parcel (the original) and a package (which we published) available, so how does Store choose between them?

In the Launcher, select **System > Settings**, and go to the **Prerequisites** page in the **Store** section.




The top group controls the how packages are selected to satisfy prerequisites, if packages are used. The default is to ask for the version of the package. Remember it's there; you may want to change the current setting sometime.

The lower group is the one we need right now. By default, Store will load prerequisites from parcels when the prerequisite says **Either**, and often that's a good choice. However, for work in progress like we're pursuing, we want it to satisfy the prerequisites from packages, if there are any.

But, we're going to reload the image, so don't bother checking it now.

Exit the image without saving (we don't want to save with the parcels loaded), then restart the **`storeOnly`** image. Now, go to **System > Settings** and select **Search bundles and packages first** on the **Prerequisites** page. Click **OK** and close the tool.

Now, go back and connect to your Store repository (**Store > Connect to Repository...**), and open the Published Items list. Load the HotDraw package, as before. Then load the HotDraw Animation Examples package and load it.

This time, we get a **Prerequisite Selection** dialog for each of the prerequisites. There's only one version available for each, so select it and click **Use Selection**. Check the information property for each to see that they're the published versions and not the parcels.

Still, that only loaded three more packages. The rest are not prerequisites, so need to be loaded.

Fortunately, the Published Items list allows you to select and load several packages at a time. To load the rest of the packages, use multiple select to highlight these packages in the left pane:

HotDraw Drawing Inspector

HotDraw HotPaint

HotDraw PERT Chart

HotDraw Slides

HotDraw Tool Development

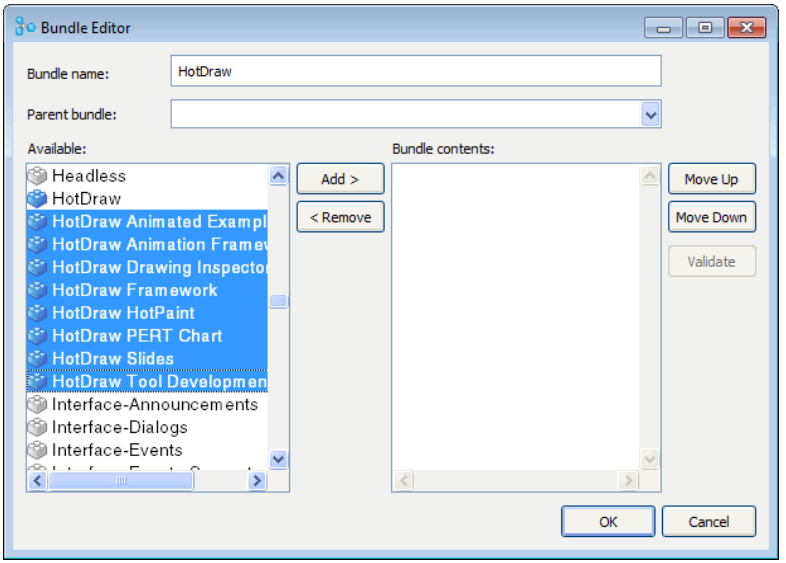
In the right pane, load a version of each (there's only one version of each so far, so just pick everything).

## Build a Bundle

With all the packages loaded, we can now build a bundle for loading all of the packages at once. This is a convenient way for loading sets of packages in the development environment.

1 Open a System Browser.

2 Select **Package > New > Bundle...**, which opens a Bundle Specification Editor.




3 For the **Bundle name**, enter **HotDraw**.

4 Search down the **Available bundles and packages** list to find the HotDraw packages.

5 Click on each HotDraw package, *except the one named* "HotDraw," so each is highlighted.

The HotDraw package itself will do no good in our bundle, but we will keep it for deployment purposes.

6 Click **Add >**.

The packages are added to the bundle, and are listed in the order in which they will load; their "load order." But, notice that they are listed in the same order that they were in the Published Items list, and we know that order caused problems. So, we need to change the load order.

7 Select the HotDraw Drawing Inspector package in the **Bundle contents** list, and click the down arrow to move it to the bottom of the list. Similarly, move HotDraw Animation Framework and HotDraw Animation Examples to the bottom of the list. Your Bundle contents list should now look like this:

HotDraw Framework

HotDraw HotPaint

HotDraw PERT Chart

HotDraw Tool Development

HotDraw Slides

HotDraw Drawing Inspector

HotDraw Animation Framework

HotDraw Animation Examples

8 Click **OK**.

9 In the system browser, select **Browser > Refresh** to update the view and show our new bundle.

The bundle is near the top of the package list, and is expandable. Expand it to see that the packages have been moved into the bundle. Look down the package list further to see that only the HotDraw package is left listed outside of the bundle. We now have both a bundle and a package named "HotDraw." That's no problem.

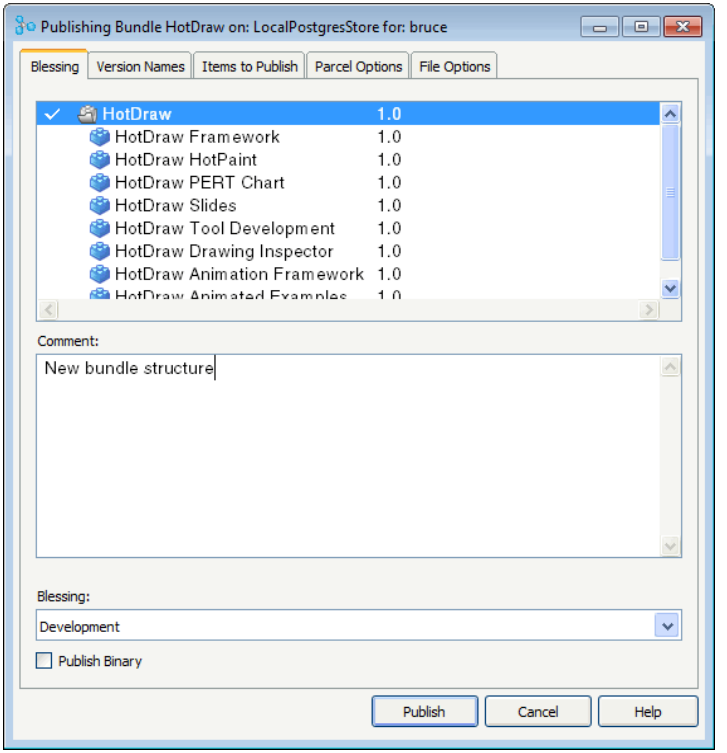
## Publish the Bundle

We now want to save the bundle and check the results.

1 In the system browser, select the HotDraw bundle (not the package), and select **Package > Publish...**

Notice that the Publish Bundle dialog shows the bundle and its contents, but only the bundle is checked. The packages contained in the HotDraw bundle haven't changed; adding them to the bundle makes no change to the package. There is no need to republish the packages, so they are not checked. If you have a special reason to publish the packages, such as to keep version numbers and comments in sync, you can click on them to set their check marks.

Unlike when we published all of the original packages, this view shows the HotDraw bundle offset from the contained packages.




2 Leave the blessing level at **Development**, and enter a blessing comment, like "**Development load bundle**."

3 Click **Publish**, and wait while the bundle is published.

Now, let's check and make sure it loads the whole application as we want.

1 Exit VisualWorks without saving the image.

2 Relaunch your **`storeOnly`** image.

3 Connect to your Store repository.

4 Open the Published Items list.

5 Select and load the HotDraw bundle (not the package).

This time the whole set of HotDraw packages should load correctly.

# Comparing with Another Repository

This is an optional exercise, but one that would be useful and informative.

If you don't do this, you will need to make a small code change manually in your image to keep synchronized.

## Get Open Repository Access

Cincom maintains a open repository for sharing code. If you are using the VisualWorks Personal Use version, your image has already been configured for guest access to the repository, allowing you to browse and compare already-published code.

To publish code to the repository, you need to obtain an account. For details, visit the Cincom Smalltalk web site:

http://www.cincomsmalltalk.com/main/developer-community/store-repository/

Follow the instructions to apply for acess to the open repository.

## Reconciling to the Repository

Assuming you have it, let's try reconciling our published version of HotDraw (the version that is loaded into our image) with the version published in the open repository.

1 If it isn't already, load your published version of HotDraw from your repository, and disconnect from your repository (**Store > Disconnect from <repository>** in the Visual Launcher).

2 Connect to the open repository using your assigned ID.

The instructions for connecting should have been included with your verification notification. The environment string is:

store.cincomsmalltalk.com:5432_store_public

3 Open the Published Items list (**Store > Published Items**).

4 Select some version of the HotDraw bundle (pick **1.7.phatch**, since a later version might not show what we want). Then select **Reconcile Image with Selection** in the <Operate> menu.

5 Wait while Store figures out the differences between the image and the published version.

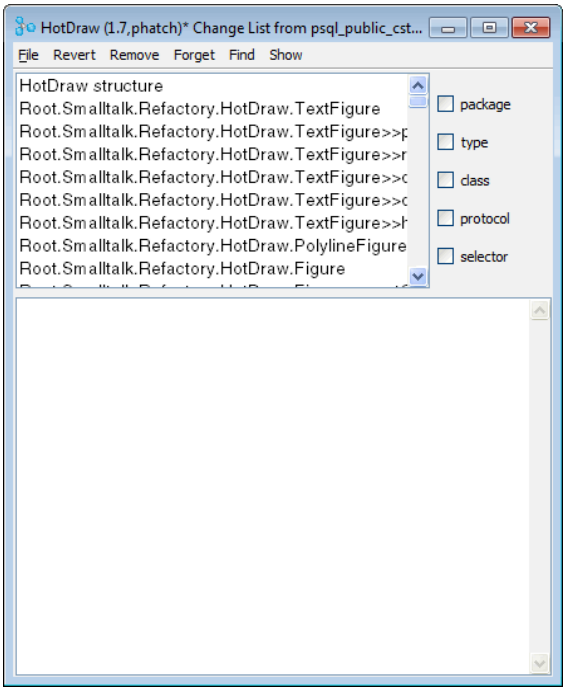
What happens in a reconcile is that Store assigns the selected version as the “parent” version of the code in your image *for this repository*. (Refer to Reconciling to a Database for more explanation.)

## Browsing Differences

Reconciling also creates a change set for this repository for each package and bundle, and records in it any differences between the version in the image and the version in the repository. Let’s explore this a bit right now.

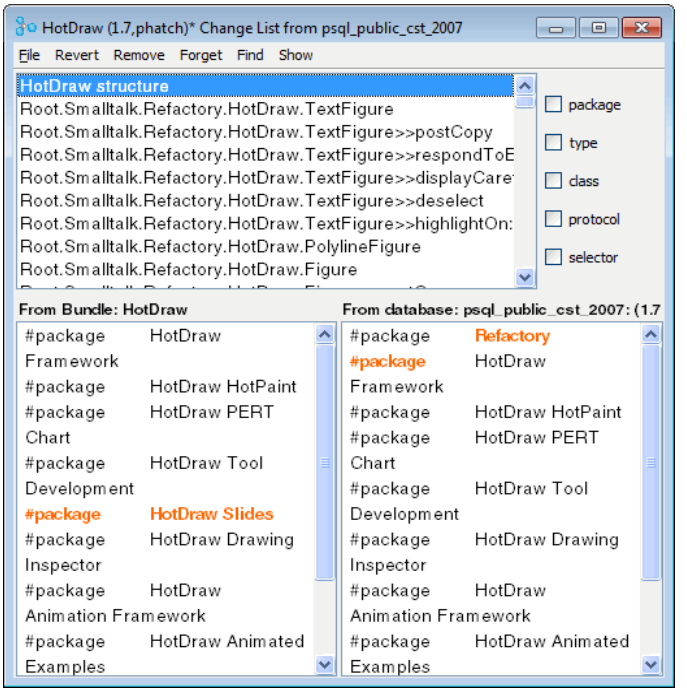
In the system browser, select the HotDraw bundle. First, select **Package > Browse > Change List on Changes**, and pick your repository. It opens on an empty change list, because we have made no changes to HotDraw since loading it. If you have made any changes, they will show up here. Close that Change List.

Again, select **Package > Browse > Change List on Changes**, but this time pick the open repository (**psql_public_cst**). This Change List shows quite a number of differences.



The change list shows definitions that are changed in the image from those in the repository, as if the repository version were the parent of the version in our image. That's not the actual history, but how it is represented.

The first item in the list is the bundle structure itself, and it shows the structure of our bundle, the bundle in the image. To compare it to the structure of the bundle in the repository, select **Show > Show Conflicts**. The definitions are shown side-by-side, with the differences highlighted in red.

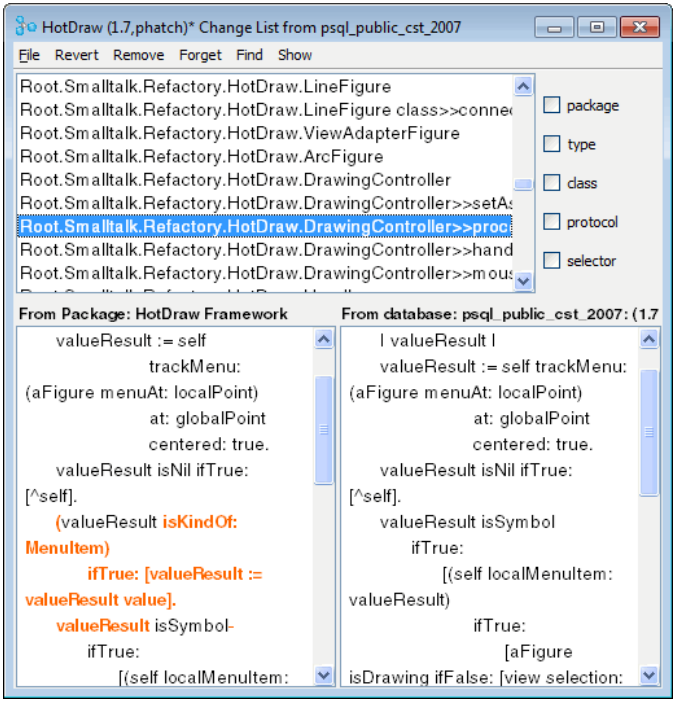



What this comparison view shows is the conflicts, or differences, between the image and its "parent" in the repository. The difference is that the repository bundle includes the Refactory package while ours does not. As mentioned before, it is now part of the base system, so has been removed from HotDraw.

In the same way, you can examine other differences between the version of HotDraw in your image with that in a repository. Since we're browsing the change list for the bundle, the changes for all of the

contained packages are listed. You can browse the changes for a single package by selecting that package in the System Browser, and browsing the change list for it.

Looking at changes down the list, many of them are trivial, being only a change of category in the class definitions. But, there are a few substantive changes as well. For example, look at the change labeled **Root.Smalltalk.Refactory.HotDraw.DrawingController>>processMenuAt:local:for:**.




There change here is a more substantive difference. Looking closely, the change is that the code in our image inserts (or the code in the repository removes):

```
(valueResult isKindOf: MenuItem)
    ifTrue: [valueResult := valueResult value].
```

You can do this comparison with any connected repository, as long as there is a package or bundle with the same name as the one in your image. Some of the changes will be important, others trivial.

Store maintains change sets for each repository with which a package is reconciled, so it is reasonably easy to work with multiple repositories, even repositories containing different versions of the

same application code. This is important, for example, if you work with a local database for versioning your work before publishing a version to the shared database for use by others on your team.

## Adopting a Difference

You can edit the code in the image right here, by editing in the left-hand pane, and then **Accept** the changes as usual with editing code.

Another option, however, is to adopt the code in the repository, because it is regarded as the “parent.” To do this, simply select the definition in the change list, then pick **Revert > Selection**. Once that’s done, there is no longer a conflict. This is one way of picking changes in a published version and applying them in your own version.

Do this for the processMenuAt:local:for: method we were looking at above. Upon completion, the code in the local image pane is updated and the repository pane shows no conflict. We have changed the code in our image to match the repository.

Look over the various other options, and try them out if you want to. You cannot publish from here, so you can’t do any permanent damage, though you can make unwanted changes to your image. If you do, just exit the image and restart, repeat the steps above.

## What we Changed in this Section

Whether you followed the steps in this section or not, there should be one method changed in your image at this point. The processMenuAt:local:for: method in class DrawingController should now be:

```
processMenuAt: globalPoint local: localPoint for: aFigure
    | valueResult |
    valueResult := self trackMenu: (aFigure menuAt: localPoint)
        at: globalPoint
        centered: true.
    valueResult isNil ifTrue: [^self].
    valueResult isSymbol
        ifTrue:
            [(self localMenuItem: valueResult)
                ifTrue:
                    [aFigure isDrawing ifFalse: [view selection: aFigure].
                    view perform: valueResult]
                ifFalse:
                    [(aFigure model notNil and:
                            [aFigure model respondsTo: valueResult])
                        ifTrue: [aFigure model perform: valueResult]
                        ifFalse: [aFigure perform: valueResult]]]
        ifFalse: [valueResult value]
```

The only difference is that the two lines shown in the previous section have been removed.

# 3

# Configuring Store

Store configuration involves, initially, loading Store support into VisualWorks, and building the Store tables in a database. Building the tables requires that you have appropriate access to a database supported by Store, which may require the services of a database administrator.

In this chapter we assume the database has been installed and that you know the Store administrator ID and password. Depending upon your working environment, you may set up the database yourself, or another member of your team or organization may be responsible.

For an overview and guide to the most common development scenarios, see: Setting up Store for your Organization. This discussion also includes specific directions for a DBA or IT department in enterprise environments.

# Setting up Store for your Organization

How you set up Store in your organization depends on the type of organization you have, on the existing infrastructure, and on how strictly it is managed.

To give you some guidance, we'll propose three development scenarios, and give you some hints on how to setup Store for each: stand-alone, workgroup, and enterprise.

## Stand-alone

In this scenario, you are not developing code with others, but you want to be able to take advantage of the sophisticated version control that Store supplies, and you might even like to share some code with the Smalltalk community in the Cincom Public Repository.

For this type of environment, you might consider using one of the free databases that Store supports: SQLite3 or PostgreSQL.

SQLite3 is the simplest in that users and table owners are simply ignored by the database. You are in effect a super-user when you connect to your SQLite3-based repository, and there aren't any rights or restrictions that the database imposes on what you can do with the code you publish. SQLite3 doesn't even have mechanisms to create users or groups/roles.

If you would like to use a more sophisticated database, PostGreSQL is an easy-to-install choice. With this you can choose to have the BERN, table owner, as well as your own username. You can choose to have a BaseSystem user if you wish.

With PostgreSQL, you do have to manage rights to the tables. The simplest way is to grant select, insert, update and delete to all of the Store tables, and select to the Store sequences, to the pre-installed role **public**, which all users are automatically assigned to. Or you could create your own set of roles, as detailed in the topic Using Groups and Table Rights.

## Workgroup

A workgroup configuration is characterized by a group of developers, typically co-located, who have full authority to manage their own environment. They may have a peer-to-peer network, or a server that is under their full control.

For this group, SQLite3 is not an option, since it is a single-user database. They need a database server, either on one of their workstations or on a shared server. However, they may not have DBA (Database Administrator) training, nor want to become DBAs just for using Store.

Oracle and PostgreSQL fit the bill for this kind of environment, being robust and fairly easy to maintain.

With Oracle, the owner/creator of the Store tables (by default, BERN) automatically has all rights to the Store table and sequences. Oracle, like PostgreSQL has a **public** role which all users are automatically assigned to. Also like PostgreSQL, granting select, update, insert and delete to **public** for each Store table, and granting select for each Store sequence, all users then have full access to all Store data.

Another way of configuring users with Oracle, is to give them global user rights by granting them SELECT ANY TABLE, UPDATE ANY TABLE, INSERT ANY TABLE and DELETE ANY TABLE. Again, you could your own set of roles and give out rights individually as detailed in the topic Using Groups and Table Rights.

### Enterprise

A development group that exists in an environment where a Database Administrator or IT department is in place that manages all enterprise database use characterizes an enterprise configuration. The choice of database is either only able to be influenced by the developers, or is out of their hands altogether. In this environment, the developers are given their login and the Database Administrator or IT department manages all rights and access.

In this scenario, the developers use the #createInstallScript mechanism, and provide the DBA/IT department with the resulting script to install the Store tables and sequences. The discussions of Store Setup for DBAs in the documentation, as well as the sections Using Groups and Table Rights and Administering Store provide guidance to the DBA/IT department on what choices they may make to ensure that the all developers are granted the correct rights to the various Store database objects.

## Loading Store into VisualWorks

Store is provided as an add-in to VisualWorks, and must be loaded in a VisualWorks image.

For building a baseline image, you may load Store either into a clean release image, or into an image in which you have code. Usually, it is better to add Store to a clean image, and then load your code, since this gives you better control of the package locations of your code. If your code is simply in an image, then you can load Store into that image, and Store will automatically package your code.

To install Store, launch a clean image (visual.im) or the image containing your code. Then, in the Parcel Manager **Version Control** section, select and load the Store support parcel which matches your database (for example, **StoreForOracle** or **StoreForSQLServer**).

Loading Store adds a menu and toolbar buttons to the Launcher.

# Configuring the Store Database

Store is retargetable to use a variety database back-ends for code storage. Currently, VisualWorks development supports:

- Oracle 8 or later, except Oracle Lite which is not supported.
- SQL Server 2000 or later is supported on Windows platforms.
- SQLite3
- DB2

Other back-ends are supported by a variety of third-parties. Back-ends provided with VisualWorks as goodies include:

- PostgreSQL
- Interbase

Support for the PostgreSQL implementation is provided by the developer at http://sourceforge.net/projects/st-postgresql/ and is available in the Cincom Open Repository.

The following instructions use standard installation scripts, using the standard file directory paths and table names. If you need to use custom parameters, you can create a custom installation script. Refer to Creating a Custom Install Script for instructions.

## Oracle Setup

Steps 1 and 2, and the steps to add users, may need to be performed by a database administrator.

All users must have the Oracle UPDATE right (privilege).

1 Using the database administration tools, create a database administrator account, with the roles CONNECT and DBA.

We recommend using the default DBA account name, **BERN**. This account will be the table owner. If you use another name, you will have to specify the **Table owner** in the connection dialog.

Also, you can create multiple Store repositories in the same physical database, but each must have a different table owner.

2 Create a directory to hold the Store data files.

During installation, Store creates two new table space files, newbern1 and newbern2, for the Store databases. The files should be in one of your database data directories, usually where Oracle data files are stored.

Because these files will need to be accessed by later VisualWorks installations as well, *do not* create them in your VisualWorks installation directories.

3 In VisualWorks, create the Oracle table spaces, by evaluating (Do It) this expression in a workspace:

```
Store.DbRegistry installDatabaseTables
```

4 You will be prompted to connect to the Store database using the table owner (database administrator) account you created in step 1 (default **BERN**).

You also need to enter the database **Environment** string, or database alias, which you may need to get from your database administrator. This is the identifier defined in the tnsnames.ora file.

Also enter the ID in the **Table owner** field, which is the ID you are logging in with. This sets the table owner ID in the Oracle database.

5 When you are prompted for the database directory, enter the directory path name created in step 2.

6 When you are prompted for a database identifier, enter a string that will uniquely identify this Store database within your organization.

This identifier is used for identifying this database to Store. If you have only one database to access, you may call it simply **store**. If you access two more Store databases in your organization, they must have different names. We suggest embedding the respective server or domain names in the database identifier.

7 You will be prompted whether to install management policies.

User/group management support allows assigning users to groups and restricting certain publishing activities to members of specific groups. See Setting up development behavior of users and groups for details.

Answer **Yes** to install support for user/group level access management, or **No** not to install this support. If you are unsure, select **No**, because you can add this later.

8 If you selected to install user/group maintenance, you will be prompted for the **Image Administrator Name**.

Only the image administrator is allowed to publish at blessing levels above normal development levels (i.e., Released), when user/group maintenance features are installed. Enter the user ID, which should not be the table owner (and must be pre-defined in the database).

The Store database is now ready to use. You will need to publish packages for use by your team.

## SQL Server Setup

SQL Server can be configured to use either Windows NT Autthentication or SQL Server Authentication. Store for SQL Server installs connections for both. You must use the connection that matches the server installation. For local Store installations, it is suggested that you use NTAuthenticatedConnection.

When installing SQL Server, you have a choice of making it case sensitive or case insensitive. It is important, for the proper operation of Store, that it be installed *case sensitive*.

Steps 1 and 2, and the steps to add users, may need to be performed by a database administrator.

1 Using the SQL Server Manager, create a database owner account (default: **BERN**).

2 Create a directory (for example, \visualworks\packages) to hold the Store data files.

3 Create the SQL Server datasets, the database account and tables, by evaluating (Do It) this expression in a workspace:

```
Store.DbRegistry installDatabaseTables.
```

4 When you are prompted to connect to the Store database, connect as the table owner (database owner) created in step 1.

  Also enter the table owner ID in the **Table owner** field.

5 When you are prompted for the database directory, enter the directory path name created in step 2.

6 When you are prompted for a database identifier, enter a string that will uniquely identify this Store database within your organization.

  This identifier is used for identifying this database to Store. If you have only one database to access, you may call it simply **Store**. If you access two more Store databases in your organization, they must have different names. We suggest embedding the respective server or domain names in the database identifier.

7 You will be prompted whether to create user management tables.

  Answer **Yes** to install support for user/group level access management, or **No** not to install this support.

  If you install user/group management support, you need to set ownership policies in each image. See Setting up development behavior of users and groups for details.

The Store database is now ready to use. You will need to publish packages for use by your team.

## SQLite Setup

SQLite is the simplest option for a stand-alone Store repository. If you are using MS-Windows, it may be necessary to install the SQLite client libraries.

To configure a SQLite repository using VisualWorks, load the StoreForSQLite3 parcel, and then follow these steps:

1 In your VisualWorks image, evaluate (**Do It**) this expression in a workspace:

```
Store.DbRegistry installDatabaseTables
```

  This immediately opens a connection dialog. Use it to specify that database you wish to configure for Store.

2 In this dialog, ensure that **SQLite3Connection** is selected as the **Interface**.

3 For the database **Environment** string, enter the absolute or relative (to the current VisualWorks image directory) path, and full file name of the SQLite database file.

4 Provide a **User Name**.

While users and table owners are simply ignored by SQLite, you do need to provide a user name (any string) in your connection profile. This string is used as the author name of any components published while you are connected using that profile.

**Note:** You can connect to a SQLite database using any arbitrary string for a user name, but you must connect using **BERN** as a user name to perform administrative operations.

5 For SQLite, no **Password** is required.

6 Similarly, SQLite ignores the **Table Owner**. The default is okay.

7 When the connection dialog is complete, you may wish to **Save…** the profile.

8 Finally, click on the **Connect** button.

9 When you are prompted for a database identifier, enter a string that will uniquely identify this Store database within your organization. (Generally, the name of the database file is used for this.)

10 You will be prompted whether to install management policies.

This feature is not supported by SQLite. Select **No**.

The Store database is now ready to use.

## DB2 Setup

Store for DB2 supports DB2/UDB version 7.2 and higher.

DB2 uses operating system accounts for user access. To use the default **BERN** table owner in Store, create an operating system administrator account and log on to that account before creating the Store tables. This account will be the table owner (default for Store is **BERN**). You will need to specify the correct table owner when connecting to Store, which will be the user who created the tables.

Assuming you already have a DB2 database installed and configured for normal access, use the following steps to set it up for use with Store:

1 Log on with SYSDBA authority (for Windows, by default all members of the Administrators group have this).

All normal Store users require accounts with authority CONNECT (after installation of Store, the needed rights on created tables and other database objects will be granted to PUBLIC).

2 Create a new database instance.

On Windows, use the administration tools to create the database, or execute:

```
db2 create database <myStore> on C
```

On Linux, execute the following command:

```
create database <myStore> on /usr/mystore
```

The parameter **<myStore>** is the database name, and **/usr/mystore** is the path to the directory containing the database files.

3 Change some database parameters:

```
update db cfg for mystore using APP_CTL_HEAP_SZ 512 LOGSECOND 50
```

4 Launch VisualWorks, and use the Parcel Manager to load the StoreForDB2 parcel.

The Store and DB2EXDI parcels will be automatically loaded as prerequisites.

5 In the VisualWorks image, evaluate (**Do It**) this expression in a workspace to create the database tables:

```
Store.DbRegistry installDatabaseTables
```

6 When you are prompted for a database identifier, enter a string that will uniquely identify this Store database within your organization.

The identifier is used for identifying this database to Store. If you have only one database to access, you may call it simply **Store**. If you access two more Store databases in your organization, they must have different names. We suggest embedding the respective server or domain names in the database identifier.

7 Choose appropriate database connection class (i.e., DB2Connection or DB2Connection72) and the installation will proceed.

8 Save and restart the image.

The Store database is now ready to use. You will need to publish packages for use by your team.

## PostgreSQL Setup

PostgreSQL support for Store is provided as a goodie and is supported by its developer. For updated and more complete information, refer to SourceForge.net.

General documentation for setting up a PostgreSQL database is available on the project site: http://www.postgresql.org/

Assuming you already have a PostgreSQL database installed and configured for normal access, use the following steps to set it up for use with Store:

1 Make sure the PostgreSQL postmaster is running with the TCP/IP option (-i) set.

StoreForPostgreSQL uses TCP/IP as its connection. If you use pg_ctl to start the postmaster from a shell (as is generally recommended), the startup command would be:

```
#> pg_ctl start -o "-i"
```

**Note:** All of the following tasks to create users and databases can be performed using the open source PGAdminIII GUI tool available at http://www.pgadmin.org/.

2 Log on as the PostgreSQL owner (typically user postgres). For example, on Linux or OS X:

```
#> sudo -u postgres psql
```

While it is recommended that you use the system user postgres when creating a local repository, note that this does not imply that you must login as postgres to actually use the database. Typically, the person using the database is not postgres. VisualWorks does not expect the login user to be the same as the image owner.

3 Create a database table owner account for Store, by executing the following in a shell:

```
#> createuser -U postgres -d -a -P <username>
```

You are now prompted twice for the new password, and then once for the superuser password for postgres.

The default Store table owner account name is **BERN**. If you use another name, set the **Table Owner** in the **Store** > **Settings** before

building the tables. The -d and -a switches allow this user to create databases and to add users.

You can create additional users at this time as well. In particular, you will want to add at least one “normal” store user account. If you plan to install user/group maintenance (not recommended), you must create an administrator account.

To exclude normal users from adding databases and users, use this command line:

```
#> createuser -U postgres -D -A -P <username>
```

4 Create the database in PostgreSQL, by executing at the command prompt:

```
#> createdb -U postgres <dbName>
```

The parameter **<dbName>** needs to be the full path to the database directory.

If the shell/environment setting $PGDATA is defined, you can omit <dbName>, and the path defaults to the value of $PGDATA. Refer to the createdb manpage for command details.

5 In your Store VisualWorks image, evaluate (**Do It**) this expression in a workspace to create the database tables:

```
Store.DbRegistry installDatabaseTables
```

6 When you are prompted to connect to the Store database, log on as the database owner.

To connect, you must specify an **Environment** string. This is the machine identifier and database name, in the format myHost:port_dbName (e.g. **192.168.10.3:5432_storedb**). The machine identifier may also be its network name. The default port number is 5432.

For example, the environment string for the Cincom Open Repository is:

```
www.cincomsmalltalk.com:5432_store_public
```

When you are prompted to confirm installing the tables, click **OK**.

7 When you are prompted for a database identifier, enter a string that will uniquely identify this Store database within your organization.

The identifier is used for identifying this database to Store. If you have only one database to access, you may call it simply **Store**. If

you access two more Store databases in your organization, they must have different names. We suggest embedding the respective server or domain names in the database identifier.

8 You will be prompted whether to create user management tables.

Generally, this option is discouraged.

You can answer **No** here, and install user/group support later if you wish.

Answer **Yes** to install support for user/group level access management. If you choose this option, you will be prompted for the name of an administrative user to manage user/group support. Refer to Setting up development behavior of users and groups for additional information and instructions.

9 Click **OK** at the last prompt.

The Store database is now ready to use. You will need to publish packages for use by your team.

## Backward Compatibility

Beginning with VisualWorks 7.7, Store uses the Glorp object-relational database framework. The old Store code used hand-coded access and update SQL code, which relied heavily on database views to facilitate gathering data into a form that can could be translated into Smalltalk objects.

Glorp does not require these database views, and instead uses dynamically created and cached database joins. Beginning in version 7.8, when you create a Store repository with these mechanisms, the database views are no longer created.

When you create a new Store repository, you cannot access it from an image that used the old Store code that was in VisualWorks 7.6 or earlier, because that new repository will not have the database views that older images rely on.

If you find that you need to have your new repository accessible from a 7.6 or earlier image, you can update it using this procedure:

1 Load the parcel OldStoreDatabase.

Using the Parcel Manager tool, this is located in the **obsolete** directory (under the **Directories** tab).

2 While connected to the target repository, execute the following in a Workspace:

```
Store.Package createPkgViews.
```

Once this is done, both old and new Store images will be able to access the new repository.

## Publishing the VisualWorks Base

We recommend that you publish the VisualWorks base in your Store repository.

While developing an application, it is easy to modify or add methods that belong to a base class and have them inadvertently associated with a base package. If you have published the base packages, you can clearly see if the base has been modified, because it will be marked as “dirty.” Noticing that, it is easy to find what was changed and move the changes to a more appropriate package.

If you do not publish the base, it is easy to overlook such changes, and you probably won't notice them until you build your application, and find that it does not work.

Also, when you load a new VisualWorks release and reconcile to the previously published version, you can easily browse changes in the areas that interest you. This aids in discovering base definitions you may have overridden, but no longer need, for example due to bug fixes, or other system changes that you may need to adapt to.

Starting with a clean image with only the Store parcels for your database loaded, do the following:

1 Connect to Store as a special user, such as **BaseSystem**.

This user ID needs to be defined for the database, with “normal user” privileges and roles (see Adding Store users). If your installation uses user/group management, additional privileges may need to be assigned at that level.

2 Load the **DLLCC** and **LensRuntime** parcels.

These are necessary to successfully publish BOSS, which is a prerequisite for Store.

3 If this is not the first time of publishing the base to this repository, then reconcile the most recent version in the repository with your image. In the Published Items dialog, select the **Base Image**

package and version, then select **File > Reconcile Image with Selection**. (See Reconciling to a Database for more information.)

4 In a browser, select the Base VisualWorks bundle, and select **Package > Publish**.

5 In the Publish Bundle dialog,

- leave all packages and bundles selected (checked),
- set the **Blessing Level** to **Released**,
- set the version string to indicate the base version (e.g., **7.0** for **VisualWorks 7**),
- do not check **Publish Binary** (you will not be loading these packages, so there's no need to publish binary),
- optionally, add a **Blessing Comment**,
- click **Set Global Blessing Level and Comment**, to set the above to all base packages.

Then click **Publish**.

---

**Note:** These steps assume that you will not be loading the **Base VisualWorks** packages from the database, and so we recommend that you *do not* publish them as binary. If for some reason you do need to load them from the database, the AT Parser Compiler must be published as binary; otherwise, DLLCC cannot be loaded from the database.

---

6 (Optional) Repeat step 3 for the StoreBase bundle, the BOSS package, and all other base bundles and packages.

Very few developers will need to publish StoreBase, since few extend it. Nonetheless, publishing Store and BOSS does, as for the rest of the base, provide a mechanism for seeing what has changed between releases.

7 Load all parcels that you will need for the base, and publish them.

For example, you probably want the UIPainter, and possibly Advanced Tools.

For some of these packages or bundles, you might also check **Publish Binary**. Do so, however, only for packages that you think you will want to load from the repository rather than from their distribution parcels. Loading a package that has been published

binary is faster than loading source code, but not faster than loading its parcel.

8 Disconnect from the Store database, and save the image under a new name, such as **`baseImage`**.

Use this image as your base for all further development.

Having logged in as the special BaseSystem user includes that ID in the version string for each package and bundle, making it clear that this is part of the base, and should not be overwritten. Do not use the special user for anything but publishing updates to the base.

## Making Changes to the Base

We strongly recommend that you *not* modify base code and publish new versions of base packages, except when you receive a new version of the base.

Instead, use the Store override capabilities, and version your modifications in your own packages. Doing so makes it much easier to preserve your overrides when migrating to a new release.

For instructions on overriding code, whether in the base or other packages, refer to Overriding Definitions.

## Updating to a New Base

When you receive a new VisualWorks distribution, you do not need to publish the whole base again. Instead, you should reconcile the new version to your database and publish, which will then only publish the changes.

1 Start the new VisualWorks image, and load Store and other distribution parcels that you have published.

2 Connect to your Store repository.

3 In the Visual Launcher, select **Store** > **Switch Databases**. Respond to prompts as presented.

Switching databases does a bulk reconcile. You can also reconcile individual packages or bundles if you prefer, by selecting **Reconcile Image with Selection** in the Published Items list for your repository.

# Team Working Environments

## Local and Shared Repositories

In addition to the team's shared repository, many teams also allow or encourage individual team members to use their own local repositories. This is particularly valuable for teams that are distributed, with several team members working from remote locations, which can make connecting to the shared repository slow.

Local, private databases are useful because:

- Access is fast.
- The developer can publish locally several intermediate versions before committing a version to the shared repository.

However, using local databases adds a level of complexity that needs to be controlled.

- Merge packages in the shared repository regularly; long delays make merging very difficult.
- Avoid multiple developers developing and versioning the same package locally; the complexity of merging quickly becomes very great.

Further, certain critical operations should be done only on the shared database:

- Renaming of name spaces and superclasses should only be done in the shared repository, and only after an integration. Then, the whole team must update to the new integrated version and resume working.

# Configuring Store Policies

Store allows you to customize several usage policies:

- Blessing (BasicBlessingPolicy)
- Merge (BasicMergePolicy)
- Ownership (BasicOwnershipPolicy)
- Package (BasicPackagePolicy)
- Prerequisite (BasicPrerequisitePolicy)

- Publish (BasicPublishPolicy)
- Version (BranchingVersionPolicy)

A policy is defined by a class, with the default policy classes as shown above. Custom policies are typically subclasses of the basic policies. A policy is installed as an instance of its defining class, and held in the Policies shared variable, a singleton of Store.Access.

## Installing a Policy

To install a policy, send the appropriate message (blessingPolicy:, mergePolicy:, etc.,) to Policies. For example, to install the ENVY blessing policy, EnvyStyleBlessingPolicy, send the message:

```
Store.Policies blessingPolicy: EnvyStyleBlessingPolicy new.
```

Note that policies are stored in the image, not in the database, so need be included in development image setup.

## Blessing Policy

A blessing policy specifies blessing levels and any restrictions on who can publish at specific blessing levels.

BasicBlessingPolicy defines the default blessing policy, and is appropriate for Store installations that do not use user/group management (refer to Setting up development behavior of users and groups). It also contains the mechanism for displaying available blessing levels in the publishing dialogs.

The blessing policy specifies the set of blessing levels as an IdentityDictionary with level names (as symbols) as the keys and instances of BlessingLevel as values. These are defined in BasicBlessingPolicy in the initializeBlessings instance method as:

```
initializeBlessings
    blessings := IdentityDictionary new
        at: #MarkedForDeletion put:
            (BlessingLevel name: 'Marked For Deletion' level: -54);
        at: #Obsolete put:
            (BlessingLevel name: 'Obsolete' level: -10);
        at: #ReplicationNotice put:
            (BlessingLevel name: 'Replication Notice' level: -1);
        at: #Broken put:
            (BlessingLevel name: 'Broken' level: 10);
        at: #WorkInProgress put:
            (BlessingLevel name: 'Work In Progress' level: 15);
        at: #Development put:
```

```
    (BlessingLevel name: 'Development' level: 20);
at: #ToReview put:
    (BlessingLevel name: 'To Review' level: 25);
at: #Patch put:
    (BlessingLevel name: 'Patch' level: 30);
at: #IntegrationReady put:
    (BlessingLevel name: 'Integration-Ready' level: 40);
at: #Integrated put:
    (BlessingLevel name: 'Integrated' level: 50);
at: #ReadyToMerge put:
    (BlessingLevel name: 'Ready to Merge' level: 55);
at: #Merged put:
    (BlessingLevel name: 'Merged' level: 60);
at: #Tested put:
    (BlessingLevel name: 'Tested' level: 70);
at: #InternalRelease put:
    (BlessingLevel name: 'Internal Release' level: 80);
at: #Release put:
    (BlessingLevel name: 'Released' level: 99);
yourself.
```

As shown above, each BlessingLevel is created with a name and a level number, which is an integer. The level number gives a ranking to each blessing, allowing limiting some actions to versions with a certain blessing level or above.

You can easily change the set of blessing levels, either reducing the number of adding others, by redefining initializeBlessings in a subclass of BasicBlessingPolicy, and then installing the new policy.

Note that if you create a custom blessing policy, you may have to define other custom policies as well, to ensure consistency. Look in particular at the merge policy for necessary changes.

The keys used in BasicBlessingPolicy are referenced at several points in the Store framework, and so should be used to set blessing levels, even if the BlessingLevel name is different. See EnvyStyleBlessingPolicy for an example. Also, accessor methods are provided in BasicBlessingPolicy for retrieving the level at these keys, which should not be overridden.

In particular, a BlessingLevel should be assigned to the #Development key, which the framework specifies as the default blessing level (in the BasicBlessingPolicy initialize method). Alternatively, or if you want some other level to be the default, override the initialize method and specify another level.

The keys #Merged, #IntegrationReady, and #Integrated are also relied upon by BasicMergePolicy, and so should also be represented in a custom blessing policy.

OwnerBlessingPolicy is the basic policy class for a user/group managed system (refer to Setting up development behavior of users and groups). It specifies several blessing levels as for use by the owner, administrator, or QA only, restricting publishing to the package owner or to members of the administrator or QA groups. These are assigned in the initialize method.

OwnerBlessingPolicy also overrides basicCanPublish:atBlessing:, replacing the general, open publishing policy with one that recognizes the restrictions, and the objectionsTo* messages, to include user/group objections.

To customize blessing, subclass either BasicPublishPolicy or OwnerPublishPolicy as appropriate, overriding methods as required to provide the desired behavior.

## Merge Policy

The merge policy primarily specifies the minimum blessing level required for a package to integrated, and the blessing levels to assign to a package is merged or integrated. These levels are referenced by the blessing policies at keys #Merged, #IntegrationReady, and #Integrated via accessor methods.

## Ownership Policy

The ownership policy identifies whether the current user has the publishing rights of the package/bundle owner. Being the owner or not affects publishing privileges in some cases.

The default ownership policy without user/group management is BasicOwnershipPolicy, which doesn't check, but simply grants ownership privileges to all users.

Part of installing user/group management is to set ownership policy to OwnerOwnershipPolicy, which checks for the package/bundle to have been assigned to the current user as its owner, and answers accordingly.

## Package Policy

The package policy primarily specifies what package a new definition will be placed in.

A default package can be assigned, and set into the alwaysUse instance variable by sending a forcePackage:while: message to the policy. This is done by the **Package > Make Current** menu command in the system browser.

In the absence of an "alwaysUse" package, several messages specify policies for the package to use in a variety of contexts (e.g., packageForClassSymbol: and packageForNewClassSymbol:). Browse BasicPackagePolicy, in the package assignment message category, for additional methods. These methods are sent by PundleAccess.

To create a custom package policy, subclass BasicPackagePolicy and override the packageFor* methods to specify your new packaging requirements. Then install the new policy into Policies.

## Prerequisite Policy

A prerequisite policy specifies how to select and load development prerequisites. The policy is controlled by three instance variables:

**blessingLevel**

The blessing level (an integer) used if #latest is the versionSelection criteria

**searchOrder**

Either #parcelsFirst or #pundlesFirst, indicating whether parcels or bundles/packages are searched first to fulfill prerequisites.

**versionSelection**

Either #ask or #latest, indicating whether, in the presence of multiple components satisfying a prerequisite, whether to prompt for the specific version or to automatically use the latest.

The search order is set in the Store Settings, on the Prerequisite Loading page, but can be set programmatically by sending a versionSelection: message to the policy. The actual selection is done by the getPrereq:from:version:for: message, which is sent by Pundle. Override this method in a subclass to customize package selection criteria.

## Publish Policy

The publish policy governs whether binary packages can be loaded, and is a central policy for objections to be raised to the publishing of packages, bundles, and parcels.

By default, binary packages can be loaded from the repository. To change the setting, send an allowBinaryLoading: message to the policy.

Objections to publishing defer to the blessing policy controls. For Store without user/group management, the default is no objections. For Store with user/group management and OwnerBlessingPolicy (or a subclass) installed, objections may be raised due to ownership restrictions. See the implementation of objectionsToPublishingPundle:atBlessingLevel: in OwnerBlessingPolicy. In general, to customize publish restrictions you would override this method in your subclass of OwnerBlessingPolicy.

## Version Policy

A version policy specifies how to increment a version number.

BasicVersionPolicy defines a simple policy without branching versions. It provides the initial version number for a package/bundle, and a method for incrementing, prompting the user if the incremented version already is in use.

BranchingVersionPolicy is the default policy. If incrementing a package/bundle version generates a version that already exists for the package/bundle, then it creates a branch instead, by appending '.1' to the current version number, and continues creating a branch in this way until a new version number is attained.

The work is done in the versionStringForPundle:initialVersion: method, which you may override in your own policy subclass to customize versioning behavior.

# 4

# Organizing Code in Store

One of the most important, and most difficult, aspects of a version control system has to do with how to organize the units that are versioned. In the case of Store, the units are packages and bundles, and the mechanisms for using them are highly flexible.

## Patterns for Organizing Code

Store is very flexible, and teams need to decide how they are going to organize code for their project. Some teams organize everything by bundle, while others use packages exclusively and use prerequisites to ensure proper loading. The important thing is to select a development pattern and stick with it.

We prefer organizing with bundles, because it is an efficient way to organize related functionality. We keep packages relatively small, as the smallest units that make for a completely functional element, and assembled them into larger units (components) using bundles.

### Guidelines for Defining Packages

A good general guideline for package size is, the smallest possible unit of code that can stand alone. For purposes of team development, this guideline suggests that packages should represent units that can be reasonably worked on independently of others. Looking ahead to deployment, however, this guideline suggests a fully functional component. These are related goals, but are not always in agreement.

Practical considerations suggest that packages should be:

- Small enough to be easily comprehensible

- Small enough to be maintained by a single developer
- Large enough to contain a complete piece of functionality
- Not necessarily as large as a complete component, which may be represented by a bundle of packages

There is a lot of room for judgment, and each team needs to decide how best to divide the work into packages.

Note, though, that decisions you make now can be changed later. Code can be moved between packages as needed; large packages divided into smaller packages; small packages combined into bundles. Just as refactoring your code is a iterative process, so is refactoring your storage structures.

When defining packages, consider:

- Which classes belong together in the same package, and which should be packaged separately
- Which methods belong with the classes that define them (most methods do), and which should be packaged as class extensions (special purpose additions to a class)

These guidelines suggest, for example, the following:

- Package code that is needed only for development (such as testing and development tools) separately from code that is needed by the deployed application. You may, however, bundle these in some high-level development bundle for convenience.
- Package client code, server code, and code that is shared by the client and server separately. This structure suggests at least three packages for client-server applications. Distinct name spaces can also be helpful to ensure this separation.
- Don't mix inessential (like examples and tests) with essential code in a single package.

You don’t have to “get it right” the first time. There is plenty of room for rearranging and refactoring your packages.

## Guidelines for Defining Bundles

Bundles are a flexible mechanism for grouping packages into larger units. They are useful for assembling a component, a complete unit of functionality, out of sub-components.

A bundle guarantees the consistency of the set of packages it contains, because any dirty packages in the bundle are also marked as dirty, needing to be published. Bundles also provide a fixed structure of their contents, by containing only specific versions of contained packages and bundles.

While a bundle contains smaller packages and bundles, it is itself to be understood as “atomic,” in the sense that the code contained in it is all intended to be loaded together in the specified order. While Store allows you to unload individual packages in a loaded bundle, this is not intended, and can compromise the functionality of the bundle.

### Using Bundles to Organize Projects

It is often useful to build several different bundles that load different deployment and development configurations.

With this limitation in view, you may, for a client-server application for example, make the following bundles:

- A complete deployment server bundle consisting of:
  - server-specific packages
  - shared application packages
- One or more development server bundles consisting of:
  - server-specific packages
  - shared application packages
  - development packages
- A complete deployment client bundle consisting of:
  - client-specific packages
  - shared application packages
- One ore more development client bundles consisting of:
  - server-specific packages
  - shared application packages
  - development packages
- A complete bundle that loads everything

Between packages and top-level development and deployment bundles, there may be several levels of intermediate bundles, representing increasingly large assemblies.

## Prerequisites and Load Order

Loading a code component is frequently dependent upon the presence of other code. A class extension in one component, for example, is dependent upon the presence of the class it extends, and a method that invokes another method requires the presence of that method. Without the presence of the required code, the component will either fail to load or fail during execution.

You could, by remembering for each package what other package needs to be loaded first, carefully load packages in the required order, and make sure they are all loaded. This is inconvenient, to say the least.

Instead, Store provides two mechanisms for controlling how packages are loaded to ensure that dependency conditions are satisfied: *prerequisites* and *load order*.

Prerequisites specify parcels and/or packages that must be present before the current package (or bundle) is loaded, and loads them if necessary. As such, a prerequisite is a special kind of pre-load action; to load a parcel or package. Both packages and bundles can specify prerequisites.

Bundles can also specify the load order of their contents. The load order is set in the bundle specification, by arranging the bundles in the order in which they should be loaded. In this way you can make sure that component packages or bundles that are required by others are loaded first.

There are no rules for when to use one mechanism rather than the other, except, of course, that a package can only specify prerequisites. And, there are some differences depending on whether you are setting up dependencies for development or for deployment. But, here are some suggestions.

Note that the following are only recommendations, particularly applicable while you are developing your package and bundle structure. Your team's development processes may require disregarding any of them.

## Loaders and Dependencies

For loading packages and bundles, Store employs two advanced loaders: AtomicLoader and AtomicAnalysisLoader. These provide great improvements over the original Store loader, which is still available.

The original Store loader read each package, gathered the name spaces, then compiled and installed each in order. It then did the same for classes, then for shared variables, and finally for methods. The defined load order was critical and could be sensitive. If a definition could not be resolved, the load was finished.

The Atomic Loader separates the compile and install phases for entire packages or bundles, allowing it to resolve issues that the original loader could not, or to abort without destabilizing the image.

During the compilation phase, the loader performs the same process as before, but compiles into a “shadow environment,” a staging area that prevents any meaningless or uncompilable code from becoming the current definition of any class, shared variable, or method.

Once the component has been compiled into the shadow environment, it can be installed into the environment in an orderly manner. If the loader determines that it cannot cleanly install all of the code in the component, it can (subject to settings) prompt you to abort the load before any code is installed, thus avoiding destabilizing the environment, or to proceed with the installation phase.

In some instances, the Atomic Loader goes further, and performs an “early install” of a critical component. For instance, if you created a new parser or scanner using the AT Parser Scanner utility, the package in which your new scanner or parser is defined must be separate from any package that has code that uses it. This remains true for the Atomic Loader. However, when the Atomic Loader sees such code, it compiles it and then does an early install. Once installed, the loader finishes compiling the package and installs everything that has been compiled but not installed up to that point, including the package that triggered the early install. It then returns to Atomic behavior for any remaining packages or bundles.

Other cases that cause an early install are:

- Loading a package with a postLoadBlock.
- Loading a parcel which might be a prerequisite for a parcel or bundle.

- Loading a package that was published as binary, when it is part of a bundle or is a prerequisite.
- Loading a package that has an extension method to ExternalInterface. Note that the extension method must not be in a package that contains code that uses that extenension method.

The Analysis Loader (AnalysisAtomicLoader) advances the above a bit further, performing analysis on each package before compiling any of them. If an early install situation is discovered, it compiles and installs all packages analyzed up until that point before proceeding to the others.

In addition, the Analysis Loader analyzes dependencies between packages, beyond those indicated by the defined load order, and adjusts the load order accordingly. This allows the Analysis Loader to install bundles that the Atomic Loader (or original Loader) cannot.

If for any reason you need to disable loaders, you can disable the Analysis Loader alone, but if you disable the Atomic Loader, the Analysis Loader is also disabled. With the Atomic Loader disabled, the system falls back to the original loader. There is generally no reason to disable these loaders in favor of the original, but the option is available in case a situation does arise. This is generally recommended only if you are working with Technical Support.

## Suggestions for Setting up Dependencies

In your development environment:

- Specify a prerequisite for any unit (package or bundle) that can reasonably be loaded individually. (Often that is the package level, but sometimes the package is too small a unit to be maintainable.)
- Specify as prerequisites any required parcels or packages that are not part of your application. (For example, VisualWorks add-in components, components from other vendors, or your own additional components that are not strictly part of this application, but required by it.)
- Specify dependency between application packages and bundles by adding them to a bundle. Set the load order as necessary, to make sure packages required by other packages are loaded first.

Note that this applies only to the development environment. When it comes to deploying your application, you may need to review the prerequisite settings, or move prerequisites defined for constituent packages and bundles to the bundles that will be used to generate parcels.

For example, assume your application is decomposed into two packages, say a package containing client specific code (ClientCodePkg) and a package containing shared code (SharedCodePkg, shared with a server component, perhaps). And, assume that the client code requires the shared code to be loaded. To load these, use the System Browser to create a bundle (MyClientApp), and add the two packages to it. Then set the load order as:

1 SharedCodePkg

2 ClientCodePkg

But, suppose the shared code, which establishes communications protocols, is dependent upon the Net Clients HTTP support code. Select the **Prerequisites** tab for SharedCodePkg, add the HTTP parcel to the **Current** list. To specify loading from a parcel, right-click and select **Parcel Only**. The HTTP parcel has its own prerequisites, which you don't need to deal with.

The rationale for this approach is that for code units that are in your application, you have full control over their presence, and the order in which they are loaded. Bundles also ensure the set of packages is consistent, while prerequisites do not. But, for components that are outside your application, you are really only requesting a service from those components, even though their presence is a prerequisite for the functionality of your application. So, this approach respects encapsulation.

## Suggestions for Setting Dependencies for Deployment

When it comes to configuring packages and/or bundles in preparation for deploying parcels, the above scheme needs to change a little.

You create parcels by publishing either packages or bundles as parcels. Which units you publish depends on which units form useful components. Prerequisites must be specified for the package or bundle that you will publish as a parcel. If the unit is a bundle, the bundle does not know about the prerequisites of the packages it contains, and so package-level prerequisites do not become prerequisites of that parcel.

This means that you must be more careful in building your deployment bundles. When you build the bundle, make sure that you add the prerequisites of the constituent packages to the bundle's list of deployment prerequisites. Then, when you publish that bundle as a parcel, the prerequisites will become the parcel's prerequisites.

Note that the prerequisites will only be parcels, which may be other components belonging to the application, VisualWorks add-in parcels, or third-party add-in parcels.

### A Simplified Approach

There are many alternatives to the scheme employed above. One that simplifies the package/bundle/parcel relationship, but at certain other costs, is the following:

- Create packages that will map one-to-one to your deployment parcels.
- Use bundles only for convenience of loading packages in the development environment (but beware of the restriction on overrides).
- For deployment, publish each package separately as a parcel.

In this scheme, the prerequisites are relevant for deployment, and the package deployment prerequisites become the parcel prerequisites.

The cost is that your packages will be larger, and you have to design your deployment parcels by moving code in and out of packages, rather than by simply arranging packages in bundles.

There are trade-offs. Your team needs to work out a scheme that works for your development process.

## Importing Code into Store

### Packaging Source in the Image

Through VisualWorks 7.2, basic class organization was provided by categories. Loading Store into an image converted categories to packages. Starting in 7.3, categories were replaced with packages as the default class organization. Accordingly, loading Store into a base image no longer changes the class organization scheme.

There are a few issues about how code loaded from parcels and file-ins is organized into packages.

## Packaging Source from File-outs

Traditionally for Smalltalk, source code was saved into external files by “filing out” the code, and this remains a popular method for saving code external to the image.

If your code is saved this way, the natural way to import it into Store would be to:

1 Load Store into a fresh image.

2 Select your file-out in the File List tool, and pick **File in** on the <Operate> menu.

This is not an optimal choice, because the default behavior is to load that code into a special pseudo-package, listed in the browser as **(none)**. You will then need to define your own package and move this code to it.

Instead, it is generally better to create a package first for the code and then file in to that package:

1 In a browser with a package view, select **Package > New Package…** .

2 Enter a name for the new package and click **OK**.

3 Select the new package in the package view.

4 Pick **Package > File into…** and select the file-out to file in.

The entire contents of the file-out is loaded into the selected package. You might not want it there, but this is a good place to start, and you can move code to other packages later.

For another file-out, you can create another package or load the code into the package you have already created. If the separation of code already present in the separate file-outs represents intentional modularization, then create another package.

## Packaging Code from Parcels

If you store code in parcels, then moving code to Store is very simple. It really does not matter whether you

- load Store into an image that has your parcels already loaded, or
- load your parcels into an image that already has Store loaded.

In either case, Store creates a package for each parcel, with the same name as the parcel, and adds the parcel’s source code to the package.

In addition to moving code into the package, all parcel properties are added to the package as its properties, including prerequisites.

There is an exception to the package name being the same as the parcel name. If you are loading a parcel that was created by a system with Store installed, but the parcel was generated with a different name than the package, and Store structure was saved with the parcel, the package name will be the same as it was in the repository.

Note that once the parcel code has been packaged, the parcel remains in the system, and is listed in the System Browser parcel view. To unload the parcel, unload the package, instead. As long as the resulting parcel is empty (nothing has been added to it) and the parcel unloads cleanly), the parcel is also removed. If the parcel doesn't unload, try unloading the parcel first and then unloading the package.

# Working with Packages

## Creating Packages

A package is first created in a VisualWorks image, and then created in the database when it is published. You can create a new package as follows: in the System Browser, choose **Package** > **New Package…**, and specify a name for the new package.

The new package is represented in the image, and so is saved with the image. A package is added to the database only when it is first published.

## Assigning New Definitions to Packages

In general, all new definitions should be assigned to a package. You can, however, for temporary code, assign it to the **(none)** pseudo-package rather than to a named package. (You can browse the contents of **(none)** by selecting **Store > Browse Unpackaged** in the Launcher window.) Except for assigning a package, you create definitions in the same way as in VisualWorks without Store.

Store provides a flexible mechanism for assigning new definitions to packages. The mechanism uses two tools:

- The "current" package, set in a list dialog opened by selecting **Store > Current Package** in the Launcher window.

- Settings specified in the **New Classes**, **New Methods**, and **New Shared** pages in the Store Settings tool (**Store > Settings**).

The Settings tool determines what action to take when you create a new definition. For example, you can set options to place all new definitions in the current package or to always prompt for the package.

Look at these pages in the Settings tool, and set your system to suit your current needs. While you are learning to work in the Store environment, it may be a good idea to set all three pages to **Always prompt**.

### Moving Definitions to Packages

You reorganize the contents of packages by moving individual definitions from one package to another. You can create a class extension by moving a method definition out of the package that contains its defining class.

To reassign a definition to another package:

1 In a Open the System Browser, locate and select the definition you want to move.

2 Choose M**ove > to Package…** from the <Operate> menu. This prompts you with a list of packages.

3 Select the name of the destination package from the list.

### Package Load and Unload Actions

Action blocks can be set to be evaluated at several stages of loading and unloading a package: pre-read, pre-load, post-load, pre-unload, post-load, and pre-save. These are all listed as properties of the package. For more information, view the **Help** for each action and browse the Store bundles for examples.

## Working with Bundles

Bundles are used to collect and organize packages and other bundles. Bundles are used to make loading packages more convenient, allowing for flexible configurations, and also for assembling the contents of deployment parcels out of smaller packages.

## Creating and Arranging Bundles

A bundle provides a convenient way for you and your team to publish, load, and merge the project packages as a set.

To create a bundle:

1 In the Refactoring Browser package list, select Local Image for a top-level bundle. For a new sub-bundle, select the parent bundle.

2 Select **Package > New Bundle...** to open the Bundle Editor.

3 In the editor, enter the name for the new bundle.

4 Select packages and/or bundles to include in the new bundle, and click the **Add >>** button.

5 Arrange the load order of packages.

The Bundle Editor lists bundles and packages in their load order. If any definition in one package refers to a definition in another package, then the referring package should be listed first.

To change the load order for an item, select it and move it using the up and down buttons.

6 Click the **Validate** button to verify that the specified order will load.

Validating creates a list of packages that the bundle will load, and verifies that, in the resulting load order, that each name space and class required by each package is either:

- loaded by the package or a package earlier in the ordering, or
- not loaded by any package later in the ordering.

If so, then the package is valid. It makes no attempt to validate definitions that are not loaded by any of the packages, since they are outside of the bundle's control.

Make further adjustments as necessary.

7 When the bundle is complete, click **Apply**.

This creates the bundle in your image. It will be created in the database when you publish it.

## Editing a Bundle Specification

To modify the contents of a bundle, use the Bundle Editor, just as you did for creating the bundle. To view the editor:

1 Select the bundle in the System Browser package list

2 Select the **Bundle Structure** tab.

3 Move packages and bundles into or out of the **Bundle contents** list.

4 Arrange the load order by selecting a package or bundle and clicking the **Move Up** or **Move Down** button.

5 Click the **Validate** button to verify that the specified order will load, to check for conflicts.

6 When the bundle is complete, click **Apply**.

## Bundle Load and Unload Actions

Action blocks can be set to be evaluated at several stages of loading and unloading parcels or packages by the bundle: preread, preload, postload, preunload, postload, and presave. These are all listed as properties of the bundle. View the help for each action for more information, and browse the Store bundles for examples.

## Including External Files

Store has the capability of including arbitrary files in a bundle, allowing non-code to be included in a bundled project. This is useful, for example, if a release of a project includes documentation, HTML, or graphics files.

The publish dialog for bundles includes a **Files** page on which you select the files in the bundle to publish with the new version.

Use the <Operate> menu in the publish dialog to add or remove files that are to be published with the component. **Add File…** opens a separate dialog to select a file. **Remove File**, which is only enabled if a file is selected, removes the specified file from the list of items to publish. Adding or removing files does not affect the component if the dialog is cancelled. Adding a new file automatically marks it for publishing with the existing check-mark behavior.

Later, when you load a bundle with a file attached, you are prompted whether to download the file.

## Specifying Prerequisites

Prerequisites are parcels, packages, or bundles that must be in the system before the code unit is loaded. Before loading, a package or bundle verifies that its prerequisites are loaded and, if not, loads them.

Package and bundle prerequisites can be specified either from a Store repository (for development) or from parcels on the local disk (for deployment), or both.

VisualWoks includes a special mechanism to analyze prerequisite relations, which you can use to specify them semi-automatically. It is useful to understand its operation.

To specify prerequisites for a component:

1 Load any components that will be required as prerequisites.

2 Select the package or bundle in a System Browser and click the **Prerequisites** tab.

Prerequisites are listed in three groups:

- **Current** lists components that have already been specified as prerequisites.

  **Missing** lists components that the prerequisite engine recognizes as defining required functionality, but are not listed under **Current**.

  **Disregard** lists components which, though they provide required functionality, can be assumed to be present, and so disregarded by the prerequisite engine. For example, Base VisualWorks is a prerequisite of everythng, but can be disregarded.

3 Add or move any components that should be listed as prerequisites to the **Current** list. Remove any that are listed as **Current**, but are not prerequisites, to the **Disregard** or **Missing** list.

4 In the **Current** list, you can change the load order using drag-and-drop. For the other lists, order is not important.

5 To specify that a prerequisite applies only when loading from Store or from a parcel, right-click and select either **Store Only** or **Parcel Only**. (This corresponds to the former distinction between deployment and development prerequisites).

Edits are saved when you leave the **Prerequisites** page.

There are several options for moving components between lists:

- Drag-and-drop between lists.
- Click the **+** icon to add an item to the list (**Current** or **Disregard**). A list of components is displayed to choose from.
- Click the **+** icon on an item in the **Missing** list to add the component to **Current**.
- Click the **x** icon to move a component in the **Current** or **Disregard** lists to **Missing**.
- Right-click and select **Add to Current**, **Remove**, or **Disregard**.

As you mouse over an item, a brief listing of definitions is shown. These are definitions that the prerequisites engine believes are required by the component whose prerequisites you are specifying. For a longer listing, click the expansion icon.

Other indicators, such as a red circle indicating a cyclical reference, also help you properly organize prerequisites or trace potential problems.

After you've made changes, click the **Recompute Relationships** button to make sure changes have not added further prerequisites.

## Specifying a Prerequisite Version

You can specify simple or complex version requirements for a prerequisite using the **Prerequisite Version Selection Action** property on the **Properties** page. The value of the property is a three-argument block in the form:

```
[:parcelName :versionString :requiredVersionString |
    booleanExpression]
```

The block arguments are the name of a prerequisite parcel being loaded, its version string, and the version string specified in the prerequisite property.

The block should answer true if the version is acceptable, and loading continues. Otherwise the loader will continue to search for another parcel of the same name with a different version. For example, this will load versions greater than the required version:

```
[:parcelName :versionString :requiredVersionString |
    versionString >= requiredVersionString]
```

### Suppress Warnings

A warning suppression action is a one-argument block, where the argument is the name of a prerequisite. The block suppresses the absent class warnings, that is, the a warning about an attempt to add code to a non-existent class. It does so on a per prerequisite basis, so you can suppress warnings for selected prerequisites.

The block must return true for any prerequisite for which warnings should be suppressed. For example, to suppress only warnings for MyPrereq, you could enter:

```
[ :prerequisiteName |
    prerequisiteName = 'MyPrereq' ifTrue: [ true ] ]
```

To suppress warnings for additional prerequisites, simply add them to the test.

The warning suppression block is run before any of the package code is loaded. Consequently it must not mention any code in the package.

The mechanism is limited. For example, if a prerequisite loads another prerequisite that raises warnings, the block will not suppress those.

## Publishing Packages and Bundles

Publishing a package or bundle is the mechanism for committing code in a working image to the repository. Until code is published, it is not available to other developers who access the repository.

Normal publishing stores source code only in the database. Initially, the entire source is published. Subsequent publishing writes only the differences, or deltas, between a parent version and the new version.

The package and bundle publishing dialogs provide two related publication options: **Publish Binary** and **Publish Parcel**, as described below.

### Basic Publishing

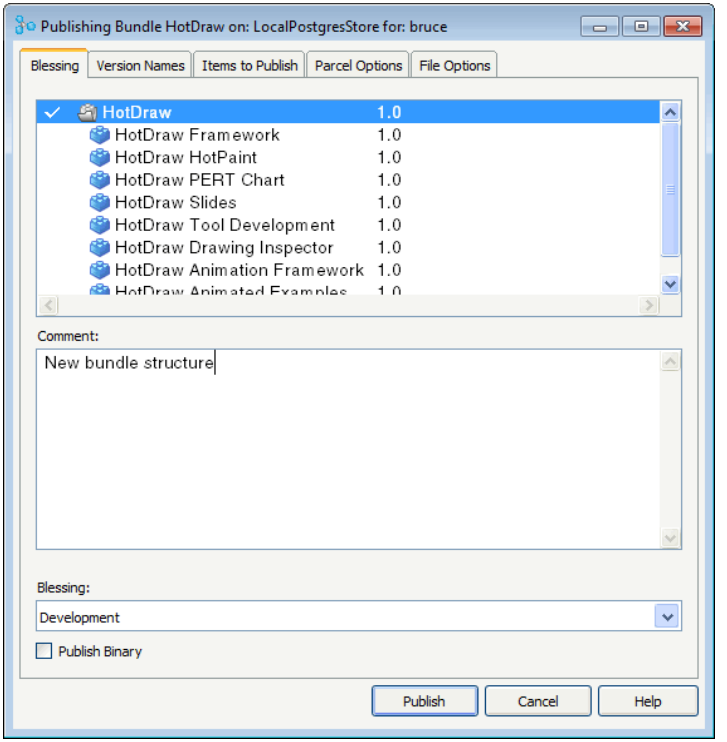
Publishing is a common, daily activity for team members, and so is described in greater detail later (see Maintaining Your Store Environment). Here we give a brief account of publishing a bundle.

To publish a bundle, you:

1 Select the bundle in the Refactoring Browser.

2 Choose **Package > Publish...**

A multi-page dialog lists the bundle and its component packages.




Initially, any package or bundle that has been changed since it was last published is marked with a check mark, indicating that it is selected to be published. On the **Items to Publish** page you can check other items for publishing, which is sometimes useful, for example to set consistent version numbers.

3 On the **Blessing** page specify:

- A blessing **Comment**,
- A **Blessing** level for each package or bundle,
- And whether to publish in fast-loading binary format (see Publish Binary).

4 On the **Version Names** page, specify the version number for each package or bundle.

Version numbers are arbitrary strings, but Store automatically increments a string that ends with a number. See Package and bundle version strings for more information.

5 On the **Items to Publish** page, select items to publish in addition to those already chosen.

6 On the **Parcel Options** page, set parcel options, if you are publishing as a parcel.

7 On the **File Options** page, select any external files that have been added to the bundle to be published. No change tracking is available in Store for external files, so you must select these.

8 Click **Publish** to publish the selected bundles and packages.

## Publish Binary

The **Publish Binary** option, on the **Blessing** page, includes a parcel-format binary representation of the package in the database. The advantage is that loading the package can use the fast loading features of the parcel technology.

Due to enhancements in the parcel loader, you can both load binary code initially, and load it for updates. This greatly speeds up the load process.

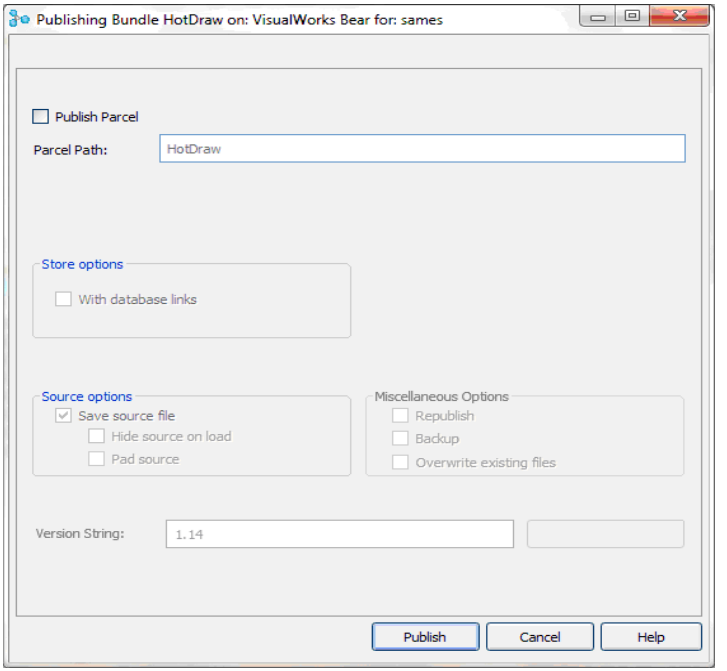
However, publishing binary uses a lot of disk space, because each publish is the whole package as a parcel in the repository as well as, for reference, all of the code changes.

For some packages, it may be necessary to publish binary, such as ExternalInterface subclasses, but this is unusual.

## Publish Parcel

This option, on the **Parcel Options** page, writes the package or bundle out as a parcel (including both .pcl and .pst files), in addition to publishing to the database. The pages differ slightly in the Package and Bundle Publishers.

The Bundle Publisher looks like this:




To publish as a parcel:

1 Check the **Publish Parcel** checkbox. This enables the other fields.

2 Enter the parcel path and name in the **Parcel Path** field.

Without path information, the parcel will be written to the current working directory.

3 In the **Store options** section:

- Select **With database links** to restore the links upon loading the parcel. This restores the code's reconciliation with the database upon load, and so is only useful with databases with which it has been reconciled.

4 In the **Source options** section:

- Check **Save source file** to write the source code into the parcel source file (.pst)
- Check **Hide source on load** to hide the source code once the code is loaded.

- Uncheck **Pad source** unless the parcel is huge. (Refer to the Application Developer's Guide for more information on parcels.

5 In the **Miscellaneous options** section:

- Check **Republish** if you are publishing a parcel that is already in the system
- Check **Backup** to make a backup copy of an existing parcel, if it is going to be overwritten
- Check **Overwrite existing files** if the parcel files already exist and are being updated.

6 When these and the other publishing options are set correctly, click **Publish**.

When publishing a package as a parcel, the package load actions get translated to parcel load actions.

If you save a bundle as a parcel, all the sub-component actions are saved. However, only the outer-most bundle's load actions are performed.

When publishing a bundle in binary form, the bundle and each contained bundle or package is published, each with its own load actions. So, when reloading, all load actions are performed.

Saving the bundle structure in the parcel increases the size of the parcel slightly, but restores the bundle structure when it is loaded into an image with Store installed. If you save the bundle structure in the parcel, you may also select to save database links. This may be useful for using parcels to distribute internal releases. When loading, Store attempts to match the links to the database. If they don't match, you will be asked whether to keep the links.

## Publish as Smalltalk Archive

Smalltalk Archives provide a way to publish packages and bundles as a single file, preserving bundle structures.

Smalltalk Archives is an optional component. To access this feature, load the **Smalltalk Archive** parcel (in the **Deploying Applications** group in the Parcel Manager). A subset of Store functionality is loaded as well if Store is not already loaded.

When Smalltalk Archive is loaded into the system, one option, **Save as Smalltalk Archive**, is added to the **Store** section of the **Publish as Parcel** dialog. With this option selected, a file is written upon publishing. The filename extension is either:

- **`.store`** if the **Save source file** option is checked
- **`.star`** if the **Save source file** option is not checked.

Technically, a SmalltalkArchive is a tar file with all bundles and packages represented as parcel files (both **`.pcl`** and **`.pst`** files). The File Manager can handle the files inside the archive, to which the archive looks like a directory. Use the TarredFilename class to access these files.

## Overriding Definitions

The literature on object-oriented language often speaks of a situation in which a subclass that re-implements a method already defined in a superclass is said to *override* that definition. In this sense, overriding is just polymorphism. In the context of packages, bundles, and parcels, “override” is used in the sense of a temporary replacement of a definition, while the defining code unit is loaded.

The ability to override definitions already in the image is a necessary feature for building components. This permits a component to provide specific behavior that it requires in place of general behavior, but also to restore previous behavior upon removal of the component.

When you unload a package (or parcel) with overrides, the original, overridden definition is restored. In this way, the overriding component can also be unloaded without compromising the system’s integrity.

It is most common to override individual methods, though class and name space definitions can also be overridden.

Note that bundles do not recognize or preserve overrides between their constituent packages and bundles. Overrides are preserved, however, if a bundle overrides a definition of one of its prerequisites.

To create an override:

1 In a System Browser, select the method, class, or name space that you want to override.

2 On the item's <Operate> menu, select **Override > in Package…** and select a package to contain the override.

3 Edit the definition as required for your application, an publish the package.

It is not permitted to create an override in either:

- a package containing the original definition, or
- a package already containing an override of the same definition.

In case you accidentally modify a definition that should be overridden, such as a base definition, and want to make it an override, you can recover as follows:

1 Move the overriding definition into your own package.

2 Publish your package and unload it.

3 Reload the overridden package.

4 Reload your package containing the override.

Now your package is recognized as overriding the base definition.

# Reorganizing Packages

Reorganizing the code in packages is essential to refactoring a system, as you search for the optimal distribution between shared and exclusive code. However, some rearranging can cause serious problems for a development team if it is not done carefully. Special issues arise in a Store environment. This section identifies some of those issues, and how to deal with them.

## Renaming a Package or Bundle

Renaming a package or bundle can have far-reaching implications for bundles and the teams that use them.

In releases prior to VisualWorks 7.6, renaming a component would lose version history information, unless the change were made by the Store administrator in the database. Following 7.6, the Store tools display both the newer and the older component, with the older component listed as a parent of the newer one.

To avoid confusion, the recommended practice is to either rename the component in the database, as described below, or to publish a version of the component under the old name with a blessing level of **Obsolete** and a comment that refers to the new name.

To rename components in the database, there is an administrative utility: in the Visual Launcher, select **Store > Administration > Rename Package in Database**, or **Rename Bundle in Database**.

This utility prompts to ask whether you want to update all loaded packages/bundles that have a prerequisite pointing to the component you are renaming. If you choose this option, all prerequisites pointing to the “old” name will be updated to the new name.

When a component is renamed, note that all versions in the database are affected, not just the one being edited.

Even if you are not preserving version histories, you do need to coordinate this change with all members of your team. If the package or bundle is a prerequisite for any others, that too must be coordinated. Make sure all users have published their latest work, then make the change, and notify team members to load the newly renamed bundle from the repository.

## Reorganizing Name Spaces

Moving definitions to a new name space is sometimes necessary when refactoring code. However, when the move is made in a package that is shared by several developers, serious problems can occur if it is not done carefully.

For example, suppose a framework package Framework 1.0 defines a class, FWClass, and an application package App 1.0 extends the class by adding a method. Lara, who is developing the application, has both Framework 1.0 and App 1.0 loaded. But then Alfred modifies the Framework package by moving FWClass to another name space, NewNS, and publishes it as Framework 1.1. Lara naturally wants the update and loads Framework 1.1. But, now FWClass has moved to the new name space, and her extension methods in App are unloaded.

To avoid this situation, Alfred should make his changes only when no one else is depending on the current name space location of FWClass. As a recommended procedure for this kind of change, do the following:

1 Instruct developers to stop work on code that has dependencies on the framework code, publish their code, and wait till further notice.

2 Load *all* packages that will be affected by the name space changes.

3 Move the classes in the framework code into their new name spaces.

4 Publish all packages that were marked dirty during the change.

5 Instruct developers to start with a new image, load the new versions of the packages, and continue working.

If the changes are made without these precautions, there are two problem situations that could arise:

- If a developer's current working image contains Framework 1.0 and App 1.0, and updates to Framework 1.1, the update will remove any methods in App that extended classes moved to new name spaces.

  There is no work around for this situation. Instead, reload as in the next situation.

- If the developer starts with a new image, loads Framework 1.1 and tries to load App 1.0, an Unloadable Definitions browser opens containing all extension methods of classes currently not in the system (due to being moved to the new name space).

  In this situation, you can copy and paste all methods from the Unloadable Definitions list into the right classes. There is no easy way to restore any lost class definitions.

Alternatively, you can file out App from the Published Items browser. Then either:

- edit the file-in to renaming the relevant classes, and file it in, or
- load the file into the GHChangeList goodie, set the Target Parcel to the desired package (create that package, if not present), and add any substitutions for all class names that have been moved to another name space. The use **Replay All** to load the code.

Then reconcile this package with the latest version in the repository and publish.

# How Code is Loaded

Packages, like parcels, provide a mechanism for initializing code after they are loaded, and for cleaning up code before they are unloaded.

The load sequence of a package is as follows:

1 The package's pre-read action is performed, if defined.

2 If the package defines the #installBeforeContinuing property, any pending components are installed.

3 The package's pre-load action is performed, if defined.

4 The objects in the package are installed into the system.

5 Every class defined in the package is sent the postLoad: message with the package as argument.

6 The package's post-load action, if defined, is executed.

A pre-read action determines whether the package contents should be parsed and loaded, i.e., before parsing. If this action returns false, the load is aborted.

A pre-load action is used to make any preparations for the code about to be loaded, such as to initialize any variables required, prior to its initialization. If the pre-load action returns false, the load is aborted.

The default behavior of the post-load action is to run the class's initialize method, if it has one. The pre-load action block can specify additional actions.

Package prerequisites, pre-load and post-load actions, and pre- and post unload actions are defined using the **Properties** page in the System Browser. Help text (**Help** > **Help**) is linked to each property.

When a package is updated, loading a newer version of a package that is already in the system, only the pre-unload and post-load actions are executed. Note that the postLoad: message is not sent to each class in the package in this case, only those classes that are newly defined or which have their definitions changed.

## Atomic Loading

Components (bundles and packages) are loaded *atomically*, in the sense of "indivisibly," by default, rather than incrementally.

When a load request is issued for a component, it, along with any prerequisites, parcels and packages is compiled into a “shadow” environment. If the entire component is successfully compiled, it is then loaded into the working image.

If, on the other hand, the loader determines that it cannot cleanly load all of the code in the requested component, then, depending on the load option, either the user is asked whether to load the incomplete code anyway, or to abandon the load, leaving the image unchanged.

To configure how the system responds to unloadable code, use the Settings Tool, **Settings > Store > Loading Policies**. Select the desired policy under **How should load errors be handled?** Refer to Handling Unloadable Code for a full explanation of the options.

## Early Loading

While the atomic loader often can load a whole bundle atomically, there are situations where a part of a component must be loaded early, before the entire component can be loaded.

When an early install is required, the loader finishes compiling the package, then goes into an install phase for everything that has been compile but not installed up to that point, including the package that triggered the early install. It then returns to atomic load behavior for any remaining packages or bundles.

The following are the cases where an early install will be invoked.

- If the component includes a parser or scanner subclass, the package in which the scanner or parser is defined must be separate from any package that uses it. When the loader encounters such code, it will compile it and do an early install.
- Parcels are loaded in their own separate atomic fashion. Any parcel that might be a prerequisite of a bundle or package will cause an early install of any package or bundle prerequisite that has up to that point been compiled but not yet installed.
- A package that is saved binary will cause an early install when it is found as part of a bundle or as a prerequisite package.
- If a package contains an extension method to ExternalInterface, the loader detect that and does an early install.

**Note:** If you need to add a method directly to the ExternalInterface class instead of one of your own subclasses, it needs to be in a package that does not have code that uses the extension method.

- If a bundle contains a package that has a postLoadBlock, that package will require an early install.
- If the loader detects any method requiring DLLCC, but DLLCC is not yet installed, it will load DLLCC and to an early install at that point.

You can force any package or bundle to install early by adding an #installBeforeContinuing property to that bundle or package. The value for that property is ignored by the loader; the presence of the property is all that matters. However, we suggest that you make it a string with a brief description of why it is needed.

The Atomic Analysis loader analyzes the component(s) being loaded, and attempts to load the component(s) in as few passes as possible. This analysis makes it possible for the loader to load definitions in packages that are out of order. For example, consider a bundle with:

- Package1 defines a subclass of ClassX
- Package2 defines ClassX

The Atomic Analysis loader will cause ClassX in Package2 to be loaded before the subclass in Package1. However, if Package1 contains a postLoadBlock, an early install would be executed. Accordingly, Package1 cannot have a postLoadBlock of any kind. The solution is to either move the definition of ClassX to Package1, or move the postLoadBlock to Package2.

# 5

# Maintaining Your Store Environment

This chapter addresses the bulk of the daily usage issues for individual developers working in the Store environment. Accordingly, this chapter covers common procedures such as publishing and loading bundles and packages.

Since development teams are increasingly becoming distributed, commonly working from remote offices and their homes, and since Store is particularly well suited to this working environment, we also cover many of the operations entailed by such distributed environments, such as switching between a private and public databases.

In some environments, you may be given a base image configured by an image administrator and imposing certain process structures on how you work. This chapter, obviously, cannot describe these processes.

Instead, for purposes of this chapter, we assume that you, the developer, have responsibility for assembling your own working image. Individuals and small teams typically work this way, and even quite large teams can and do.

This presupposes that your working environment does *not* have the user/group management feature installed. If it is installed, that image will impose limits on what you, as a developer, can do. It is a responsibility of your image administrator to explain any such restrictions.

Beginning to Use Store runs through installing Store in this sort of environment, and demonstrates working in it. This chapter provides a more comprehensive view of these work practices.

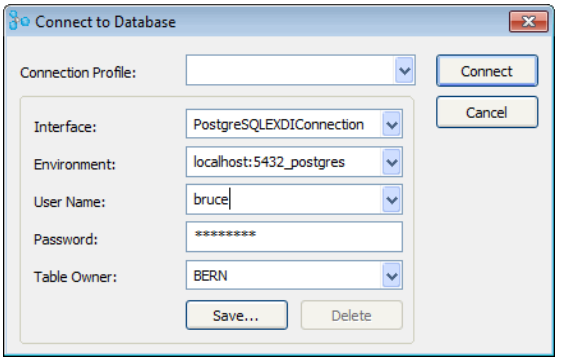
# Working Connected and Disconnected

Unlike some source control systems, Store does not require that you be connected to the database in order to work on project code. In general, you can do most of your work disconnected from the database, because the code you are working on is in your current Smalltalk image, on your disk drive.

## Connecting to the Database

It is only necessary to be connected to the database when you are performing database functions, such as publishing and loading packages. Otherwise, you can work detached from the database.

To connect to the database, select **Store > Connect to Repository...** in the VisualWorks Launcher.




Select the database type in the **Interface** box. Enter the **Environment** string, your **User Name**, and **Password**, as assigned by the database administrator (which might be you, if you installed a local database). To connect to your own local database, rather than a remote database, you can leave the **Environment** field blank. If there are multiple Store repositories in the database, select the **Table owner** for the repository you want. Then click **Connect**.

## Detaching from the Database

When you have working versions of the packages you need loaded, you can detach from the database and work strictly within your image. You can still perform all programming tasks, including defining and rearranging packages and bundles, but you cannot perform database tasks, such as publishing or loading.

To detach from the database, select **Store > Disconnected from DB** in the VisualWorks main window.

### Saving Connection Profiles

You can save your connection settings, and alternative settings for other Store repositories, as a **Connection Profile**. In the **Connect to Database** dialog, enter the connection parameters and click **Save** and enter a profile name. This is particularly useful if you frequently connect to alternate databases; you only need to select the profile next time you want to connect.

Store can hold multiple database connection profiles in the image, and you can export and import these as a XML file. If you update to a new version of VisualWorks, or start over from a clean image, you can import the saved settings from a file.

To save all database connection profiles in your image, use the <Operate> menu in the lower status bar of the VisualWorks Launcher window, and select **Export Repositories…**. Enter the name of an XML file to contain the profiles. To load the connection profiles from this file, use **Import Repositories…**.

You can also save and load connection profiles using the **Save Page…** and **Load Page…** menu items in the Settings tool (select **Settings** from the **System** menu in the Launcher window). Note, however, that these employ a different XML file format.

## Working Off-line

Because your image contains working versions of the packages you are developing, most of your work can be done while disconnected from any database. Working off-line allows you to work at home or another remote site, continue working when your data connection is down, or any other time when it is not possible or convenient to be connected.

In a detached image, you can do anything that does not require database access. That is, you can:

- Browse, modify, and test the working versions of the packages and bundles in your detached image.
- Create working versions of new packages and bundles.

However, without a database connection, you cannot:

- Open the Published Items list or a Versions list.
- Publish your working versions.
- Load new versions into your image.
- Merge versions.

Of course, if you work on a notebook or other portable computer, there is nothing to do. Just take the computer along and work as usual. The following comments only apply if you are actually moving your work to another computer.

## Preparing to Work Off-line

If you are working in an image that is connected to the Store database and you decide to continue your work off-line, do the following:

1 Verify that your image contains the correct working versions of all packages and bundles you want to work on. If necessary, load the desired versions from the database.

2 Save and exit your image.

3 Take copies of your work to the remote workstation.

If you transport your image using removable media, be sure to take the .im file and the .cha file associated with your image.

4 Also, copy the directory containing the source files for any packages you have loaded binary. It is named after your repository name, in your image/ directory.

## Resuming Work with the Database

When you have finished your off-line work, you:

1 Save and exit your detached image.

2 Copy your work back to your normal workstation.

This may be the working image and associated files, or copies of filed-out or parceled-out code.

3 Start your VisualWorks image.

4 If necessary, file-in or parcel-in your changes.

5 Connect to the Store database.

You can now resume your normal Store work, with full access to the Store database.

## Working with Multiple Databases

In many working environments you will need to publish and load code from multiple databases. Many developers, especially in geographically distributed teams, connect and publish most frequently to their personal, local database, and less frequently to a remote, shared database. Then, there is also the Cincom public repository, which is available for code updates and user contributions. Visit the Cincom Smalltalk web site for more information.

There are also utilities available for replicating a repository. StoreForGlorpReplicationUI is included as a convenience in the **Version Control** page of the Parcel Manager, and is provided as contributed.

Store makes working with multiple databases easy, by remembering relationships between the code in your image and the code in each of the databases, on a package-by-package basis. For each package, Store maintains a change set of differences between the package in your image and a "parent" version in a database, one change set for each database that has been linked to that package in your image.

For example, say you load package Foo from Store repository A. The version in the repository is the parent of the version loaded into your image, and Store maintains a change set between the two. Initially, the change set is empty, because there are no differences. As you make changes to the package in your image, those changes are written to the change set.

Now, suppose you connect to another repository, B, that also has Package Foo. (Let's assume that the name indicates that the packages really are the same, except for versions, and not completely different code that happen to be named the same. That presents different problems.) You cannot publish or load a version of this package in repository B, because Store doesn't know the relationship between them. To establish the relationship, you reconcile your image to the repository, as described in Reconciling to a Database. Once reconciled, Store also has a change set of differences between Foo in your image and Foo in repository B.

The parent remains the version loaded from A, until you publish or load another version from either repository, at which point the parent changes and the change sets are updated. With these records of relationships and differences, it is easy to switch back and forth between the repositories, and to do code comparisons.

In general, you only have to reconcile a package once to each database, though occasions arise when you may have to reconcile again. But, Store will notify you if reconciling is necessary.

This facility makes it easy to use a private, local database for frequently publishing work in progress, and then to switch to a shared remote database to publish your stable code for access by the team. This is a common practice remote developers. Once reconciled to each of the alternate databases, you simply connect to one of them and continue working; there is no need to re-reconcile each time.

## Reconciling to a Database

To coordinate code in an image with a code base in a Store repository, the image and the database must be reconciled. Reconciling compares the sources for packages in your image to the sources for the same packages in the database, and creates a change set for each package. The change sets represent the differences, or “deltas,” between your image and the database. Then, when you publish to the database, only the deltas need to be published. Reconciling also sets the “parent” version, so your next published version will have a history.

Usually you would reconcile to the most recent published version in the repository, but you can reconcile to any version. You might reconcile to an older version, for example, if you are developing a branch or need to create a new branch, possibly to do maintenance development for a previous release.

To reconcile to a package or bundle to a database:

1 Connect to the database.

2 In a Package Browser, select the package or bundle.

3 Select **Package > Reconcile with Database**.

4 If there are multiple candidates, a dialog lists them. Select the version to which to reconcile and click **OK**.

In the typical case you should select the most recent version. Select an older version only if you have a reason to modify or create a branch.

## Switching Databases

Switching databases is essentially one large reconcile. By switching databases, you are choosing to reconcile all of the components in your image to components stored in the database. Needless to say, this can take a long time for a large application. However, once done, you can freely switch back and forth between databases without having to re-reconcile.

To switch databases:

1 Connect to the new target database.

2 Select **Store > Switch Databases** in the Visual Launcher.

3 When prompted, select whether or not to **Maintain existing links to** the previous database.

For a database that you will connect to again, you want to maintain links. Choose to remove links only if you will not be using it again, or using it only rarely. Once you have removed links, you will need to reconcile the database again before you can use it.

You can choose now to retain links, and then delete them later using **Store > Remove Database Links…** command, if necessary.

4 When prompted **Which should be used to reconcile?**, click either:

- **Use most recently published** - to automatically reconcile the packages in your image to the most recently published versions in the new target database that match your code.
- **Select published versions** - to specify individually the versions in the new target database to reconcile.

5 If you chose to select versions to reconcile, you will be prompted with a list of applicable versions when there are more than one candidates. Select a version and click **OK**. This may occur several times, depending on the size of the database you are reconciling.

Once the database has been reconciled to your image you can begin publishing packages to the database.

From this point on, you seldom need to re-reconcile your image to the database. Simply connect and continue working.

## Removing Database Links

If you are never going to access a particular Store database again, you may want to remove the links to it. This also releases its change set. If you change your mind later and want to access this database again, you need to reconcile your image to it again.

To remove the links:

1 In the Visual Launcher, select **Store > Remove Database Links...** .

2 Pick the database to unlink from the displayed list.

3 Click **OK**.

## Using a Local Database

It is frequently useful, especially for remote developers, to be able to version changes they make locally as well as when publishing to the shared database. Doing so requires using a local database. Using a local database is just a special case of using multiple databases, except that a good deal of your local database will be a duplication of what is on the a remote database.

To connect to a local database, select **Store > Connected to DB** as usual, but specify the environment string for the local database. Often, leaving this field empty defaults to your local database, but depends on your environment configuration.

The primary issue in working between the local and team databases is keeping version numbers consistent. Because Store maintains links to multiple databases, this is not a problem. Once a database has been reconciled to your image, links and changes are tracked for each database. You can freely publish your changes to any of your databases.

To start using a new local database:

1 Load the current versions of your packages from the shared database.

2 Disconnect from the shared database, and connect to your local database.

3 Publish your packages.

The version numbers will be different than those in the shared database, but this is alright. Store maintains links to both, so when you reconnect to the shared database the versions will be correct.

### Publishing Back to the Team Database

To publish back to the shared team database:

1 If you have not published since last updating from the shared database, publish to your local database.

2 Disconnect from your local database, and connect to the shared database.

3 Publish your packages.

## Maintaining your Working Image

At the beginning of a project, your baseline image is probably configured by your project leader. Starting there, you modify the image by making changes to the code for which you are responsible, and by loading packages published by other developers on the team.

Store provides several browsers for determining what is loaded into your image, for comparing your image with published packages, and for updating your image configuration.

### Browsing Loaded Packages and Bundles

To browse all loaded packages, you can simply open the Package Browser by choosing **Store** > **Browse Packages** in the VisualWorks Launcher.

You can use the Loaded Items list to see which bundle and package versions your image contains. To do this, choose **Store** > **Loaded Items** in the launcher. The Loaded Items list shows the bundles and packages for which your image contains working versions, and indicates (in parentheses) the parent version of each working version.

### Examining the Contents of a Bundle

It is often convenient to have a top-level project bundle that loads all of the project packages. When this is the case, the Loaded Items list has entries for the project bundle and its contents, listed alphabetically among entries for the system packages. To see just the package versions that are contained in the project bundle:

1 Select the project bundle in the Loaded Items list. (Bundle entries are listed in alphabetical order preceding package entries).

2 Choose **Examine** > **List Contents** in the Loaded Items list.

The Bundle Contents list displays an entry for each component package or bundle that belongs to the bundle you selected. These entries are displayed in the order in which the components are loaded into an image.

## Browsing Packages and Definitions

### Browsing Loaded Code

Any packaged code that you have loaded into your image can be viewed using any of the standard browsers. The browser displays the current working version of the code, including any changes you have made, rather than the parent package's version.

Text formats and other indicators are used in the System Browser to indicate various states of code with respect to packages. For example, a bold type face indicates items (class, name space, or method) that are defined in the selected package. If a package has changes, a number indicates the number of changes not yet published. A modification to the icon attached to a package or bundle may also indicate a state needing attention. In general, the indicator is fairly self-explanatory, or clear with a little investigation.

When you select a package or bundle in the Package Browser, the text view shows all of the databases and versions to which it is linked.

### Browsing Unloaded Code

To browse the code in a package that is not loaded, or a version of code as it is in the database, you need to use the Package Browser.

1 Connect to the database.

2 Do either of the following:

- Open the Published Items browser, select the package and version you want to browse, and select **Examine > Browse**
- Select the item in the System Browser and select **Browse Versions** in the <Operate> menu for the item.

A Package Browser is opened on the selected version.

### Browsing Shared Variable Definitions

You can open a Definition browser on published Namespace and Class definitions. The browser lists all published versions of the specified definition, for easy comparison.

To open the Definition browser, select **Store > Browse Definitions**, and then either **NameSpace named…** or **Class named…** . A prompter asks for the name of the definition. Enter the definition name (case-sensitive), and click **OK**. The definition browser opens on the published versions of that definition, if any.

## Browsing with Package Changes and Overrides

Store maintains a change set for each package for each database, without you having to set it up.

Tools to browse these change sets are available on the **Package > Browse** menu in the Package Browser. Select the package to browse, then select **Package > Browse > <command>** (or **Browse > <command>** on the <Operate> menu). The options are:

### Changed methods

Opens a method browser on methods changed in this package since it's last publication.

### Change set

Displays a list of linked databases containing the definition, and opens the Change Set inspector on the change set for the database you select.

### Change list on changes

Opens a Change List on the changes to this package

### Overrides of others

Opens an Override Browser on definitions in this package that override definitions of others, showing the overridden definition.

### Overridden by others

Opens an Override Browser on definitions in this package that are overridden by definitions in others, showing the overridden definition.

Note that overrides are suppressed from these change sets, so loading a package B that overrides package A will not show up in package A's change set or the changes list. They will, of course, show up in the relevant overrides/overridden tools.

# Loading Published Code

You can load code from the database either from individual packages, or from bundles that specify their constituent packages and versions.

It is generally better to load a bundle than an individual package. You can still select and load individual packages in the bundle, and the packages you choose are automatically loaded in the correct order.

## Loading a Bundle

To load a particular version of a bundle:

1 Open a Versions list for the bundle and select the desired version.

2 Choose **Version > Load**, and confirm that you want to load the bundle version.

Store loads the bundle's component versions in order, prompting you for additional confirmation as needed.

After the operation is complete, your image contains a new working version of the bundle, whose parent is the bundle version you selected.

## Loading a Package

To load an individual version of a package:

1 Open a Versions List for the package, and select the desired version.

2 Select **Version > Load**.

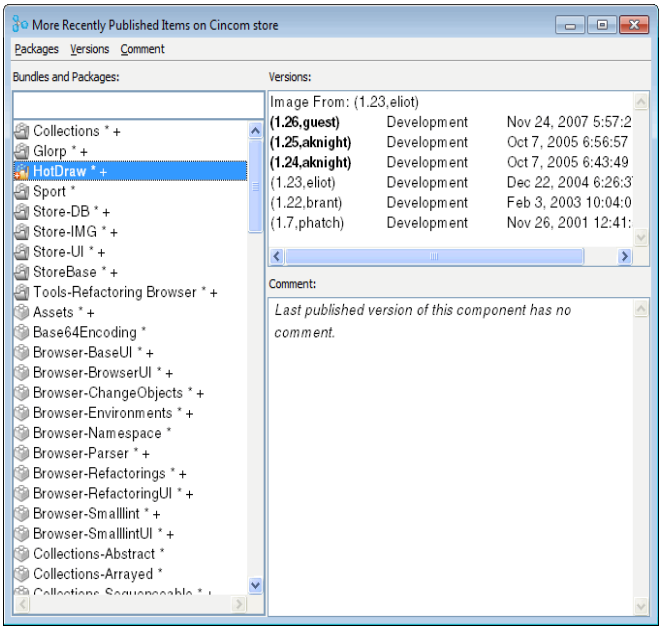
If your image already contains a working version of the package, you must confirm that it is to be replaced with the selected version from the database. Any unpublished changes in the current working version will be overwritten, and can only be retrieved from the change file.

After loading, your image contains a new working version of the package. If the package is a component of a bundle that is loaded in your image, your working version of the bundle is marked as modified.

## Updating to New Versions

Before you load a bundle or package, you need to browse enough of the database to find what items have been published.

To browse the published bundles and packages, choose **Store > Published Items** in the VisualWorks Launcher. To browse only bundles and packages that were published more recently than those you already have loaded, choose **Store > More Recent Published Items** instead. This opens the Published Items browser.




The Published Items browser displays the names of published bundles and packages. Bundles are listed first, followed by packages, each being distinguished by different icons. For long lists, you can type in part of the name you are searching for into the entry field, top left, to filter the list, and use * for pattern matching.

The **Versions** pane lists the versions of the selected package or bundle that are in the database. These are sorted by the date and time they were published, with the newest versions shown at the top.

To browse the code definitions for a version of a package, select the package version in a version list or graph, and select **Versions > Browse**. This opens a Package Browser on the selected package version.

### Updating Published Source Code

During development, members of the team will periodically publish their updates to the shared database. Some of these you will want to use to update your image, so you can take advantage of those changes. Which packages you update will depend on your team's development practices and policies, and the parts of the system you are yourself working on.

To update from a published version of a package:

1 Connect to the shared database and load the updated packages.

2 Disconnect from the shared database.

3 If necessary, connect to your local database and publish the updated packages.

### Updating from a Build

During an extended project, a number of "builds" might be created, each build consisting of a new image built from a set of packages in the shared database. Rather than update all the packages yourself, it is often convenient to pick up this new build image and make it your new baseline image. This is particularly true if areas of the system are updated that you do not normally work with yourself.

The build image already has links to the shared database. To begin using it with your local database, you need to reconcile it with your local database. The easiest way to do this is by using the Switch Database command, as follows:

1 Save a copy of the build image as your new network image.

2 Launch it and connect to your local database.

3 Select **Store > Switch Database** to reconcile the image with your local database.

4 Publish the packages locally.

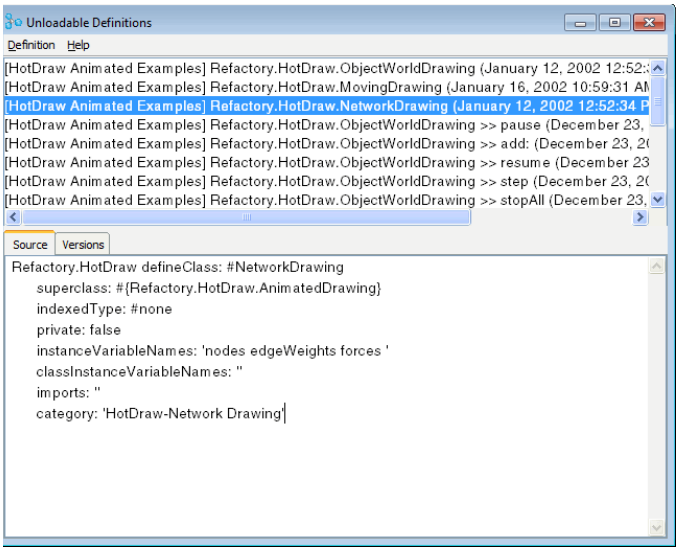
## Handling Unloadable Code

When loading a package from the repository, certain definitions may not be loadable. This can occur because there is a missing superclass, a missing namespace, or a missing class needed by an extension method or shared variable.

You can configure how the system responds to unloadable code, using the Settings Tool. Open the tool, **Settings > Store > Loading Policies**, and select the desired policy under **How should load errors be handled?** The four options are:

- **Complete the load but open a tool to manage error.** (Default) Definitions whose prerequisites are satisfied are loaded. The Unloadable Definitions browser is opened on definitions that could not be loaded, allowing you to resolve the issues and make the definitions loadable.
- **Ignore errors and load what can be loaded.** Definitions whose prerequisites are satisfied are loaded. Any unloadable definitions are ignored, and you are not notified.
- **Open a dialog to ask what to do.** For each definition that cannot be loaded, you are prompted for a resolution action.
- **Load only if there are no errors.** Only packages for which all definitions can be loaded are loaded. Otherwise, no definitions are loaded.

The first option above is the system default, and allows loading to complete without blocking, while providing tool support for resolving Store loading errors.




By examining the code, you may understand what is preventing some or all of these definitions from loading. You then have three options for handling the errors, using commands in the **Definition** menu:

- **Load** or **Load All**. If you correct the problem, for example by fulfilling a prerequisite, one or all of the definitions may now be loadable. Once the problem is fixed and the definitions loaded, republish the package. However,if the problem can be fixed without modifying this package, you may wish to simply fix and republish the containing bundle, quit your image and reload.
- **Remove from Package**. If you cannot fix the problem so that the definition can be loaded, it should be removed from the package and the package republished (though you may want to file out the code first to save it). Unless the definition is removed and the package republished, the problem will recur the next time you load the package.
- **File Out As** or **File Out All**. In some cases, you might want to file out a definition, edit it, and then file it back into the appropriate package. For example, if a class or namespace has been moved to a different namespace, extension methods be fixed in this way. Remember to remove these old definitions (**Remove from Package**) to prevent the load problem from recurring.

In some circumstances, when code is moved to new namespaces, you can copy and paste all methods from the Unloadable Definitions list into the right classes.

There is no easy way to restore lost class definitions. These classes must be redefined in the proper package.

Lastly, to clear the Change Set of those packages affected by these changes (to resolve the errors), reconcile these packages with the latest version in the repository and republish.

## Publishing a Component

When you have developed a package to a point where you are ready to make your work available to the team, you publish the package or a bundle containing it. This writes your new version to the Store database, and makes it publicly available.

All components are published using a UTC timestamp obtained from the database server. If your database does not directly support this feature (e.g., SQLite3), then Store uses the local image's current UTC time.

## Pre-publication Checks

To save the headaches of needlessly publishing bad versions, perform the following pre-publication checks.

### Comparing to the Parent Version

Before publishing, you may want to run a comparison check with the parent version, to evaluate the changes you are about to publish. To perform the comparison, either:

- Select the package in the Loaded Items list or Bundle Contents list, and choose **Versions > Compare with Parent**, or
- Select the package in a Versions List, and choose **Versions > Compare with Image**.

This opens a Difference Browser on your working version and its parent.

### Inspecting Changes

You can review the changes you have made to your working version of a package (changes from the working version's parent). To do this, select the package in the Package Browser, and choose:

- **Package** > **Browse** > **ChangeList on Changes**, to inspect all definition changes, or
- **Package** > **Browse > Changed methods…**, to examine only changed methods.

### Merging with Another Version

It is possible that while you were making changes to your working version, another developer has published a new version of the same package. If so, you may want to merge your working version and then publish the integrated version. Refer to Integrating code versions for more information.

## Publishing a Bundle

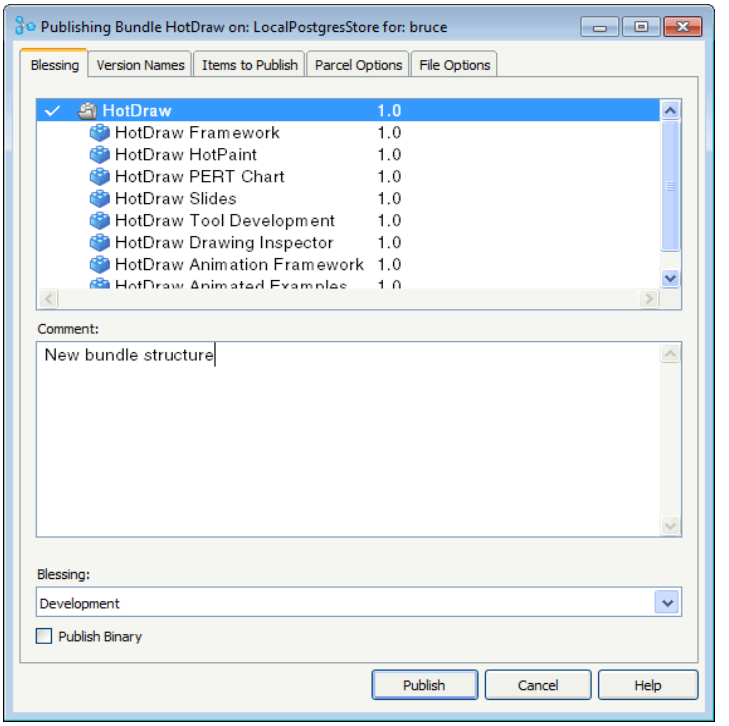
If your project uses bundles, you normally publish bundles rather than individual packages. Publishing a bundle automatically publishes all component packages whose working versions have been modified in your image.

To publish a bundle, you:

1 Select the bundle in the Package Browser or in the Loaded Items list.

2 Choose **Package > Publish...** .

A multi-page dialog lists the bundle and its component packages.




Initially, any package or bundle that has been changed since it was last published is marked with a check mark, indicating that it is selected to be published. On the **Publishing Options** page you can include or other items.

3 On the **Blessing** page specify:

- Whether to publish in fast-loading binary format (see Publish Binary). If checked, all packages will be published in binary format.
- A blessing version for each package or bundle.
- A blessing comment, giving additional information, for each package or bundle.

4 On the **Version Names** page, specify the version number for each package or bundle.

Version numbers are arbitrary strings, but Store automatically increments a string that ends with a number, based on the version currently in your image and other published versions in the database. See Package and bundle version strings for more information.

5 On the **Items to Publish** page, select items to publish in addition to those already chosen.

6 On the **Parcel Options** page, set parcel options, if you are publishing as a parcel.

7 On the **File Options** page, select any external files already added to the bundle to be published (see Including External Files). No change tracking is available in Store for external files, so you must select these.

8 Click **Publish** to publish the selected bundles and packages.

## Publishing an Individual Package

If your project does not use bundles, you must publish your packages individually.

Even if you do use bundles, sometimes you only want to update a single package. Note, however, that no bundle will contain that version of the package, so that it will not be loaded with the bundle.

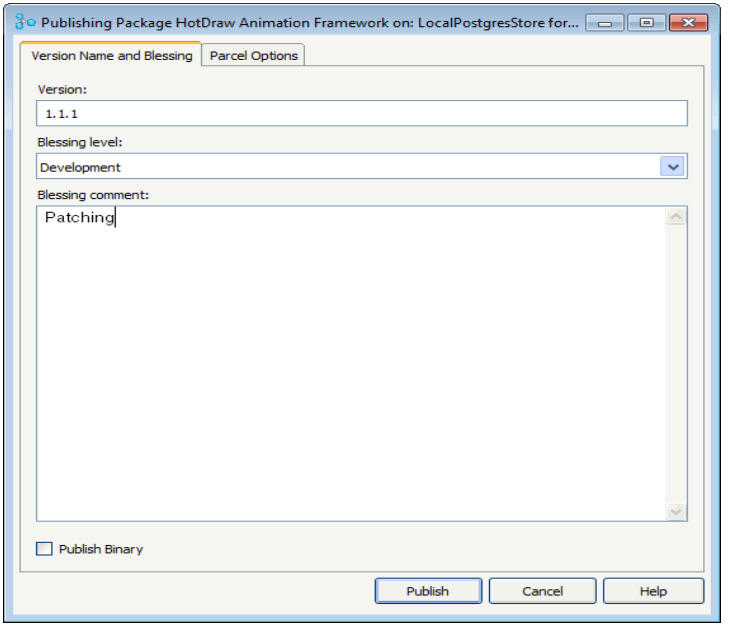
The Package Publisher dialog is an abbreviated version of the Bundle Publisher.

To publish a package, you:

1 Select the package in the Package Browser.

2 Choose **Package > Publish...**.

A multi-page notebook dialog lists the component name in its title:




3 On the **Version Name and Blessing** page specify:

- The version number for the package, in the **Version** field
- A blessing level for the package
- A blessing comment, giving additional information, for the package
- Whether to publish in fast-loading binary format (see Publish Binary). If checked, the package will be published in binary format.

4 On the **Parcel Options** page, set the parcel settings.

Refer to Publish Parcel below for more information. This page allows you to save your code in parcel files (.pcl and .pst) at the same time as you publish the bundle to the database. The options are the same as for saving parcels in general.

5 Click **Publish** to publish the selected bundles and packages.

# Exporting code

For various reasons, including code sharing with a developer who does not have access to the database, you may need to file out your code. File out options are provided for bundles and packages, on the <Operate> menu of the Package Browser package list, and in the **Packages** menu.

When filing out, no package or bundle structure is preserved, so Store is not needed in the image it is filed into.

If a package has overridden code, filing out the package includes the overridden code, not the overriding code. Note that in versions 5i.2 and earlier, overridden code was excluded from the file-out, but it is now included.

# 6

# Version Control

Under VisualWorks, individual team members develop and update components called packages. At integration time, appropriate versions are merged for each package in the project to produce a new project baseline.

## Versions

The first time a package is published to a database, VisualWorks creates an initial version string and stores the source code for definitions in the package.

When you load a package into your image, you load a copy, or working version, of the package. You can modify this copy in your image without affecting the parent version, the version in the database.

When you publish your working version, you create in the database a new version, with a new version string, that stores *only those definitions that have changed* (“deltas”) from the parent version.

Bundles are also versioned in this way. When a bundle is created, its specification identifies the current working versions of its component packages and bundles.

When you load a bundle, the specified version of each of its components is loaded. The operation is recursive on nested bundles, so that all contained packages are loaded.

When you publish a bundle, Store automatically publishes any component whose working version has been modified, and creates a new version of the bundle that specifies the current component

versions. A working version of a bundle is considered modified whenever you edit the bundle's specification or modify any of the working versions of its components.

## Package and bundle version strings

Whenever a package or bundle is published, it is assigned a new version string to identify it. A version string is any arbitrary string, such as "1.0" or "Experiment".

Although Store supplies simple version strings as defaults, your development group may need a more detailed version identification scheme. If the version string ends with a number, Store automatically increments it when you publish. You need to approve the increment. If the string does not end with a number, Store will append a number (.1), again subject to your approval.

Note that the publishing developer's name is automatically appended to the version string, so user names may be omitted from your naming convention.

## Blessing levels

Most development processes call for publishing components at various stages of completion, from early prototyping to final customer product. In VisualWorks, you specify a blessing level, plus comments, to indicate where a version is in the development cycle.The standard blessing levels and some suggested uses are:

| Blessing Level | Suggested Use |
|---|---|
| Broken | Version has known defects; should not be used until fixed. |
| Work in Progress | Code is unfinished and functionality is incomplete. |
| Development | Interim version; code may be unfinished and functionality is incomplete. |
| Patch | An update to a previous release, but on a separate development branch. |
| Integration-Ready | Version is ready for merging with versions developed by other team members. |
| Obsolete | A component that is no longer in use or have been renamed. |

| Blessing Level | Suggested Use |
|---|---|
| Replication Notice | A blessing added to versions that were replicated, noting when each was replicated, by whom, and the name of the source and target databases. |
| Integrated | Version has been successfully merged with other integration-ready versions. |
| Merged | Version is the result of merging multiple integration-ready versions. |
| Tested | Version has been tested and is ready for general release. |
| Internal Release | Released but only for internal deployment. |
| Released | Version is available for all users and customers. |

A version's initial blessing level is normally set by the developer who publishes the version. As the version progresses through the test and review cycle, various authorized team members change the blessing level as appropriate. The Merge Tool uses the Integration-Ready, Integrated, and Merged levels.

Your policy for blessing levels should determine the following:

- How many levels are relevant to your process?
- What should the levels be called?
- What does each level mean?
- What kind of information should appear in the associated comment?
- For each blessing level, who is authorized to set it?
- For each blessing level, who is authorized to load a version at that level?

The standard blessing levels provided with VisualWorks can be changed to fit your development process:

1 Subclass Store.BasicBlessingPolicy or Store.OwnerBlessingPolicy, and define the new blessing levels by overriding the initializeBlessings method.

2 Set the new blessing levels as the policy by sending:

```
Store.Policies blessingPolicy: MyBlessingPolicy new.
```

Using OwnerBlessingPolicy allows enforcing user/group rules for blessing policies. For example, you might allow only the owner to set Integrated and Ready to Merge blessings, or only QA to set the Tested blessing. Browse the default code to see how to set the restrictions.

# Working with versions and blessings

## Browsing a version history

When you load a version of a package into your image with new versions of packages, you may notice that individual definitions have been changed. To find out more about these changes, you can browse the definition's change history. To browse the history of a definition, you:

1. Select the definition in any VisualWorks browser or in a Package Browser.
2. Choose **Store** > **Browse Versions** from the <Operate> menu in the class or method view. This opens a Version Browser. Each listed entry contains the definition's selector followed by the version string of the containing package version.
3. In the Version Browser, select the entry of the version you want to examine. The definition version appears below it.

## Comparing versions

Comparing past versions of a definition shows what changes have been made to produce the final version. To compare versions of a class or method definition, you:

1. Select the definition from any source browser.
2. Choose **Store** > **Compare with...** from the <Operate> menu in the class or method view to launch the Version Browser on the selected definition.
3. Select the version to compare, and the Definition Differences Browser opens.

The Definition Differences Browser lists the class or method definitions side by side and displays any differences highlighted in bold and red.

### Changing a version's blessing level

An initial blessing level for a version is set when the version is published. As the version progresses through a verification and approval cycle, its blessing level needs to be changed *without* changing the version string. For example, a version initially published with a "Development" blessing level may need to be advanced to "Integration-ready" or demoted to "Broken."

Usually, a team policy determines who can set specific blessing levels.

To change the blessing level for a published version of a package or bundle:

1 Select the package or bundle from any list (for example, the Loaded Items list or Bundle Contents list).

2 In the Versions list, select the version whose blessing level you want to change.

3 Choose **Versions > Set Blessing Level…**, to open the Blessing Level dialog.

4 Select the new blessing level, enter a comment, and click **Accept**.

## Integrating code versions

Application development is not typically linear. In the process of team development, several developers may make changes or additions to the same classes and the packages that contain them. Periodically during development, and especially near project completion, these different pieces of work must be integrated, or merged together and made consistent.

The VisualWorks Merge Tool assists in this integration process. The Merge Tool examines the parent-child relationships among published versions of a package, identifying the modifications that differentiate two or more related versions from their latest common ancestor. It then combines user-selected modifications into a new working version of a designated base version.

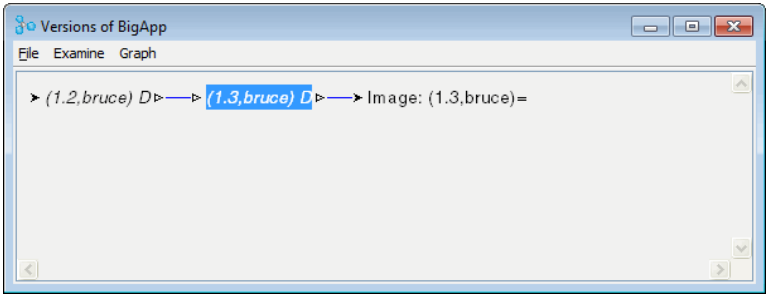
**Note:** The Merge Tool merges published versions of packages only, one of which is loaded as the base version. The merge results in an unpublished package version in the image. For successive merges, you *must publish* the resulting merged package before performing subsequent merges.

## Relationships among versions

A package's published versions bear parent-child relationships to each other in a family tree rooted in a common ancestor. In this tree, each branch represents a divergent line of development.

A line of development starts when you load a working version into your image from a published version of a package (say, version 1.1). You make modifications to your working version. Once changed, the image copy has no version number, but the System Browser (and version graph) show that the loaded version is "dirty."

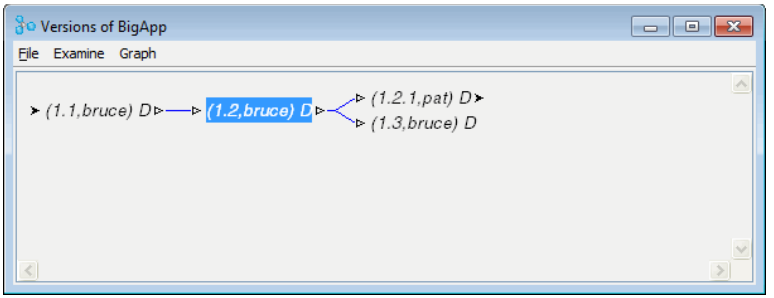
When you publish this package, a new version (1.2) is created in the database. This new version is now the parent of the working version in the image, and is also the child of the original version 1.1. This continues each time you publish. The version tree is completely linear, and may look like this:




There is no need for integration in a linear version tree, because the latest version has all of the changes already.

Another team member may also start a line of development based on any of these published versions, and may publish changes. Suppose this line is started from version 1.2, and is published as 1.2.1 (The

version numbering is determined by your team's publishing conventions.) The version tree is no longer linear, and might look like this:




This branching can become arbitrarily complex. At some point, these divergent lines of development need to be integrated.

## Conflicting and nonconflicting modifications

Each published version contains some modifications. These modifications may or may not cause conflicts when the versions are merged.

Conflicting modifications exist when the same definition has been changed in different ways in two or more of the versions being merged. For example, a conflict would exist if different expressions have been added to the same method in each of two versions being merged. A conflict would also occur if a method has been changed in one version and removed entirely in another.

In contrast, a nonconflicting modification exists in either of the following cases:

- A change has been made to a definition in only one of the versions being merged.
- A change has been made to a definition in more than one version being merged, but all of the changes are exactly the same.

## Integrating a set of packages

Integrating a set of package revisions is done by merging one or more versions from the repository into the version in the image (the "trunk" version). The merge process first finds the most recent version that is the common ancestor of all these versions. Next, changes are computed from this common (base) version to both the image version and the selected revisions.

For each component change, a modification set is created. Each modification set holds the changes made by the versions being merged.

> **Note:** The Merge Tool merges published versions of packages only, one of which is loaded as the base version. The merge results in an unpublished package version in the image. For successive merges, you *must publish* the resulting merged package before performing subsequent merges.

Multiple packages and revisions can be merged in a single session, though the complexity of the merge, and the reconciliation task, compounds quickly.

A merged package or bundle may receive a new blessing, but not if the new blessing would be lower than the previous blessing. If it would be lower, an “Informative Blessing” is applied, containing all the new information and attached to the package or bundle. **Package > Publish Informative Blessings Only** is available in the Merge Tool so blessing levels never change in a merge.

To integrate package versions:

1 Ensure that all versions to be merged have been published, including the image version.

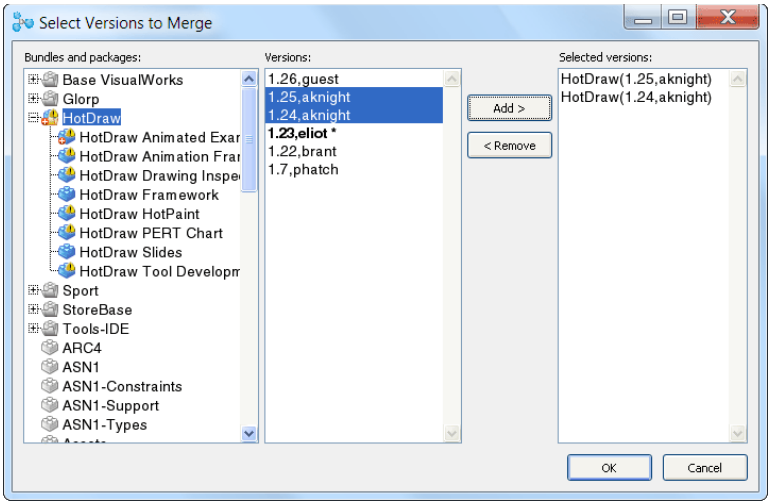
  The Merge Tool only examines published versions of packages. Unpublished modifications in the working version may be overwritten in the merge process.

2 Connect to the repository and load the “trunk” version of the package (or packages or bundles) to be integrated.

  The trunk version is the version into which changes will be integrated.

3 Open the Merge Tool (**Store > Merge Tool**) from the VisualWorks Launcher.

4 Click **Select Packages...**.

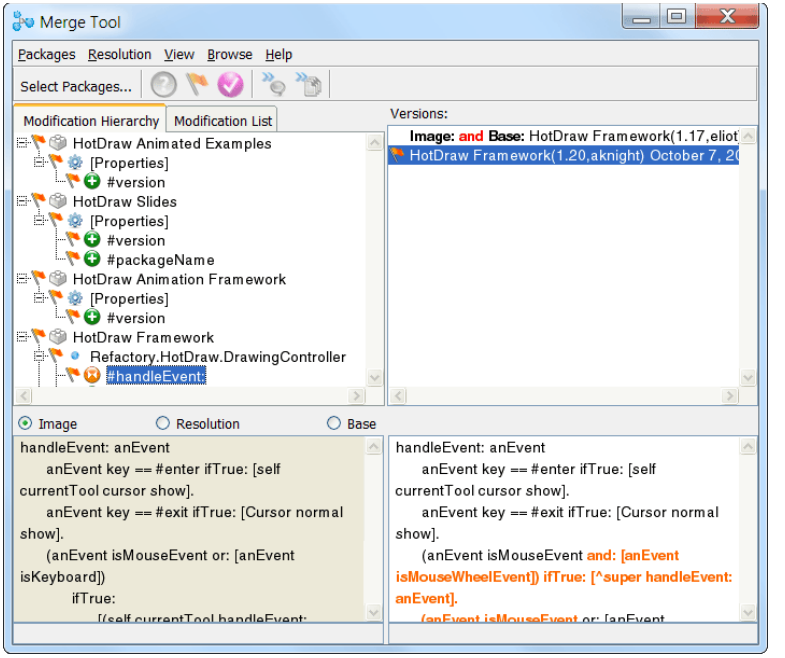



5 Select the package(or bundle) that you want to integrate.

The Merge Tool then displays the versions of the package, with the loaded version in bold.

6 Select the version(s) of the package to integrate to the trunk version. Repeat for each package to include in the integration session.

7 Click **OK** to begin the merge operation. When completed, the results are shown.




8 If there are conflicts indicated, you must select a resolution (see Resolving conflicts).

9 Apply the resolutions, either individually (**Apply Resolution**) or in bulk (**Apply All Resolutions**) to commit the resolutions to the image version.

The tool provides other operations that you can use as needed.

10 Click **Publish Packages...** to publish the merged version of the package.

## Resolving conflicts

When conflicting modifications exist among the versions you are merging, you must choose a resolution for each conflict. You may resolve the conflict by selecting any one of the existing modifications, or you may create a new modification in the Merge Tool. The resolution you choose will be included in the new composite version.

To choose a resolution from existing alternatives:

1 In the modification view, select the name of the definition or comment that has the conflict to be resolved.

2 In the version view, select the version that contains the alternative you want.

3 Choose **Resolution > Select as Resolution**. The square icon next to the definition or comment name is filled with an X to indicate that the conflict for this item has been resolved.

If none of the alternative modifications is appropriate as a resolution, you can create a new modification.

1 In the modification view, select the name of the definition or comment that has the conflict to be resolved.

2 In the version view, select the version whose modification is closest to the one you want.

3 Edit the contents of the modification in the code view.

4 Choose **Accept** from the code view's <Operate> menu.

This creates a new alternative modification, selects it as the resolution, and immediately applies it to the working version in the image.

As you resolve more and more conflicts, you may wish to eliminate them from the display. Choose **View > Show Unresolved** to filter out resolved conflicts.

## Excluding nonconflicting modifications

By default, the Merge Tool assumes that all non-conflicting modifications are to be included in the composite version, and automatically marks each non-conflicting modification as a resolution.

If, upon inspection, you decide that certain non-conflicting modifications are unwanted, you can exclude them. For example, one version may contain a new method called `cut`, while the other contains a method called `cutToClipboard`. These methods are reported as non-conflicting modifications, even though they do the same thing. You probably want to exclude one of these modifications from the merged version.

To exclude a non-conflicting modification, choose the base version (which does not include the modification) as the resolution:

1 Choose **View > Show All**, if necessary, to display non-conflicting modifications.

2 In the modification view, select the base version.

3 Choose **Resolution > Select as Resolution**.

# 7

# Programmatic Interface

Store provides a programmatic interface for standard load and publish operations. This API is useful for scripting loading and/or publishing of packages and bundles, either using workspace, script file, or application code.

## Component Model Classes

There are two fundamental sets of objects representing Store components:

- objects that represents packages and bundles in the image, and
- objects that represent packages and budles in a repository.

In effect, Store is all about taking the former, and publishing them to the latter, and loading the latter into the former.

Component objects in the image are represented by two subclasses of CodeComponent, classes PackageModel and BundleModel. These are “live” objects in the image.

The class hierarchy is:

```
CodeComponent
    Parcel
    PundleModel
        BundleModel
        PackageModel
```

Notice that parcels are also a kind of CodeComponent, and when you load them, they also “live” in the image.

When we publish or load these image objects, we create transient Store/Glorp objects representing all of the relevant information for the live image objects. These transient objects are represented by two sibling classes in the Store-Database Model package, classes StorePackage and StoreBundle.

The class hierarchy is:

```
StoreObject
    StoreSourceObject
        StorePundle
            StoreBundle
            StorePackage
```

In both of these hierarchies, the bulk of the implementation is in the pundle class. However, to perform operations, the specific package or bundle class must be used for the target object.

## Connecting

VisualWorks connects to a Store session using a connection profile that specifies a profile name, database connection environment, and a user name and password. The profile is then used to connect to the repository.

Each profile is held by the Store RepositoryManager. The easiest way to select a profile is to use the selection dialog:

```
profile := Store.ConnectionDialog chooseProfile.
```

The dialog is the familiar one for connecting to store, though clicking **Connect** does not actually make the connection in this case.

If you know the profile name, you can bypass the dialog:

```
profile := StoreLoginFactory currentStoreSessionFromProfile:
    (RepositoryManager repositories detect:
        [:each | each name = 'LocalPostgresStore'] ifNone: [^self])
```

Then, use the profile to create the repository connection and establish a GlorpSession:

```
session := StoreLoginFactory currentStoreSessionFromProfile: profile.
```

GlorpSession is the object used for repository access. Use the resulting session for reading package and bundle versions. You can also query the repository through the session, for example, for packages and bundles:

```
session read: StoreBundle.
```

which returns a Set of all bundles in the repository. For packages, send:

```
session read: StorePackage.
```

If you connect using the **Store > Connect to Repository...** command, you can access the session with:

```
session := StoreLoginFactory currentStoreSession
```

Note that the session created using the menu command and the session created by currentStoreSessionFromProfile: are different sessions.

Reusing a session is often a good thing, but because a session buffers database objects it will consume memory. Reuse is generally only advised for queries and loading, and use a separate session for publishing.

## Loading Packages and Bundles

Given a GlorpSession to a Store repository, you can make queries about the repository contents and load packages and bundles.

Most of the protocol for loading is defined in class StorePundle, but requests for operations on packages and bundles must be sent to the specific class, either StorePackage or StoreBundle.

To load a package or bundle, you need its name and the version number to load. Given that information, you can send, for example:

```
StoreBundle loadWithName: 'HotDraw' version: '1.0' in: session
```

If you connected using the usual Store menu command, you can use the shorter form which uses the default session:

```
StoreBundle loadWithName: 'HotDraw' version: '1.0'
```

To load a package, send loadWithName:... to StorePackage. If you know the object name and version number, this is all you need.

To get the names and versions available in the repository, you need to querie it. For this there are many messages that can help identify the package or bundle you want. For example, to get all package or bundle names in the repository, send the appropriate one or these messages:

```
StorePackage allNamesIn: session.
StoreBundle allNamesIn: session.
```

If you have a package or bundle name, you can get the available versions with:

```
StorePackage allVersionsWithName: 'HotDraw' in: session.
StoreBundle allVersionsWithName: 'HotDraw' in: session.
```

StorePundle implements a lot of special methods for more precise queries. Browse the **query utility** method category for these. A lot of examples are provided in the **Store Workbook**, available on the **Store** menus in the Launcher.

Once you have a package or bundle identified, you can load it by sending a loadSource message.

## Publishing Packages and Bundles

When you publish a package or bundle version, there are several information items that need to be set. These are familiar to you from the Publishing Dialog, and include:

- name
- version string
- comment
- as binary
- as parcel
- save bundle structure

To do this, create a PublishSpecification instance for the package, or a PublishSpecificationList instance for a bundle, and then set the relevant properties. For example, for a package:

```
publishSpecification := (Store.PublishSpecification pundle: aPackage)
    version: aVersionString;
    publish: true;
    modified: true;
    publishPundle.
```

and for a bundle:

```
| specs |
    specs := Store.PublishSpecificationList fromBundle: aBundle.
    specs do:
        [:each |
        each publish: true.
        each modified: true.
```

```
        each version: aVersionString].
    specs publishPundle.
```

Get the aPackage or aBundle objects from the Store.Registry:

```
Store.Registry bundleNamed: 'HotDraw'
Store.Registry packageNamed: 'HotDraw Slides'
```

A selection of options are as follows (browse AbstractPublishSpecifcation and its subclasses for others).

**binarySave:** *aBoolean*

Specify if the package is to be saved as binary.

**blessing:** *anInteger*

Set a blessing level.

**comment:** *aString*

Set a version comment.

**description:** *aString*

Set the displayed description for the component.

**modified:** *aBoolean*

Indicates whether the component has been modified (is “dirty”). By default, this is set to true if the component is “dirty.”

**parcelSave:** *aBoolean*

Specify whether the component should also be saved as a parcel. If true, other options become applicable, for example, bundleStructure:, databaseLinks:, parcleVersionString:. Browse PublishSpecification for these and additional option methods.

**publish:** *aBoolean*

Indicates whether a new version of the component is to be published. By default, this is set to true if modified is true. You can set this to true to force publishing of an unmodified component.

**version:** *aString*

Set the version string.

To publish the package or bundle, send the publishPundle message, as shown in the examples above, or publishSilentlyUsingSession:.

**publishPundle**

Publish. Answer false if aborted.

**publishSilentlyUsingSession:** *aSession*

Publish using a specified GlorpSession. Answer false if aborted.

Browse PublsihSpecification for specialised publishing methods.

# Notifications

Store raises notifications when errors or other exceptions occur during operations. By default, these open dialogs, which is appropriate for the Store tools. However, if you use workspace scripts or other code to load packages and bundles, you can trap these exceptions and resolve them programmatically.

This section lists several of the exceptions currently available, and how to use them in a script. Additional exceptions may have been implemented, and can be browsed. Yet other exceptions may be raised during the load or save of a component and are handled internally.

In the following, each "do:" is an example of how to respond to the exception being raised, along with a description of what happens. For example,

```
["Code to do something"]
    on: AnExceptionClass
    do: [:exception | "code options"] "explanation"
```

## Store.AlreadyConnected

Raised when an attempt is made to reconnect to a repository that is already connected.

```
["Code to connect to a repository"]
    on: AlreadyConnected
    do: [:exception | exception resume: true] "This will do the reconnect
and proceed"


["Code to connect to a repository"]
    on: AlreadyConnected
    do: [:exception | exception resume: false] "This will do cancel the
reconnect and not proceed"
```

### Store.AtomicLoadingError

Raised during loading if the Atomic loader is turned on, but the Analysis loader is turned off, and either

- a pre-load action block has failed and the LoadingActionError exception has answered false, or
- a pre-read action block has answered false

Returning from this exception will terminate the exception, and proceed without loading the package or bundle in question.

```
["Code that attempts to load a Package or Bundle"]
    on: AtomicLoadingError
    do: [:exception | exception return] "This will quietly skip loading the
package or bundle"
```

### Store.BundleHasUnpublisedChangesConfirmation

Raised when a bundle is being published and also being published as a parcel, if the bundle has been modified and the choice to save the bundle with database links is set to true.

It is also raised if a PublishSpecificationList has as its first item a bundle, and it is sent #publishParcel.

```
["Code that attempts to save a Bundle and publish as Parcel"]
    on: BundleHasUnpublisedChangesConfirmation
    do: [:exception | exception resume: true] "This will ignore the changes
and publish the bundle as a Parcel as is"
```

```
["Code that attempts to save a Bundle and publish as Parcel"]
    on: BundleHasUnpublisedChangesConfirmation
    do: [:exception | exception resume: false] "This will cancel the
publishing of the bundle as a Parcel"
```

### Store.ContainsUndeclaredError

Raised when a package is being published and there are undeclared variable or class references in the package.

When raised, this error will always cancel the publishing of the bundle or package.

```
["Code that attempts to save a Bundle or Package"]
    on: ContainsUndeclaredError
    do: [:exception | exception return] "This will prevent the error from
raising a warning dialog. The publish will not proceed"
```

### Store.CreateParcelDirectoryConfirmation

Raised when a package or bundle is published as a parcel and the directory for the parcel specified does not exist

```
["Code that attempts to save a Package or Bundle and publish as Parcel"]
    on: CreateParcelDirectoryConfirmation
    do: [:exception | exception resume: true] "This will create the needed
directory, and continue to publish the Package or Bundle"


["Code that attempts to save a Package or Bundle and publish as Parcel"]
    on: CreateParcelDirectoryConfirmation
    do: [:exception | exception resume: false] "This will cancel the
publishing of the Bundle or Package if the directory does not exist"
```

### Store.InvalidStorePundleError

Raised when publishing if the StorePackage or StoreBundle is not a valid object to be published.

```
["Code that attempts to save a Bundle or Package"]
    on: InvalidStorePundleError
    do: [:exception | exception return] "This will prevent the error from
raising a warning dialog. The publish will not proceed"
```

### Store.LoadingActionError

Raised during a component pre-load or post-load if an error occurs while trying to execute the pre/post-load action.

```
["Code that attempts to load a Bundle or Package"]
    on: LoadingActionError
    do: [:exception | exception resume: true] "This will cause the error to
be ignored and the load to continue"


["Code that attempts to load a Bundle or Package"]
    on: LoadingActionError
    do: [:exception | exception resume: false] "Load to terminate"
```

### Store.LoadOrSaveCompilationError

Raised when loading a bundle or package during a pre-read if the source in the pre-read cannot be compiled.

```
["Code that attempts to load a Bundle or Package"]
    on: LoadOrSaveCompilationError
    do: [:exception | exception return] "This will prevent the error from
raising a warning dialog. The publish will not proceed"
```

### Store.LoadOrSaveEvaluationError

Raised when loading a bundle or package during a pre-read if the compiled block from the source in the pre-read causes any error when executed.

```
["Code that attempts to load a Bundle or Package"]
    on: LoadOrSaveEvaluationError
    do: [:exception | exception return] "This will prevent the error from
raising a warning dialog. The publish will not proceed"
```

### Store.LoadOrSaveInvalidArgumentsError

Raised when loading a bundle or package during a pre-read if the compiled block from the source has the wrong number of arguments (0).

```
["Code that attempts to load a Bundle or Package"]
    on: LoadOrSaveInvalidArgumentsError
    do: [:exception | exception return] "This will prevent the error from
raising a warning dialog. The publish will not proceed"
```

### Store.NullPackageCanNotBeSavedError

Raised when attempting to publish the "(none)" package

```
["Code that attempts publish the (none) Package"]
    on: NullPackageCanNotBeSavedError
    do: [:exception | exception return] "This will prevent the error from
raising a warning dialog. The publish will not proceed"
```

### Store.PreReadActionConfirmation

Raised after executing the pre-read action if that action answers false.

```
["Code that attempts to load a Bundle or Package"]
    on: PreReadActionConfirmation
    do: [:exception | exception resume: true] "This will cause the load to
proceed"
```

```
["Code that attempts to load a Bundle or Package"]
    on: PreReadActionConfirmation
    do: [:exception | exception resume: false] "This will cause the load to
be canceled for the target Package or Bundle"
```

### Store.PrerequisiteUnableToLoadConfirmation

Raised when loading a package or bundle and a prerequisite component is not already successfully loaded.

```
["Code that attempts to load a Bundle or Package"]
    on: PrerequisiteUnableToLoadConfirmation
    do: [:exception | exception resume: true] "This will cause the load to
proceed without the prerequisite component"


["Code that attempts to load a Bundle or Package"]
    on: PrerequisiteUnableToLoadConfirmation
    do: [:exception | exception resume: false] "This will cause the load to
be canceled for the target package or bundle"
```

### Store.ReconcileWarning

Raised when loading a new version of a package from a database which has not previously been reconciled.

```
["Code that attempts to load a Bundle or Package"]
    on: ReconcileWarning
    do: [:exception | exception resume: true] "This will cause the reconcile
to be done and then the load to proceed"


["Code that attempts to load a Bundle or Package"]
    on: ReconcileWarning
    do: [:exception | exception resume: false] "This will cause the load to
be canceled"
```

### Store.ReplaceModifiedPackageNotice

Raised when a bundle or package is modified and a request is made to load a new version.

```
["Code that attempts to load a Bundle or Package"
    on: ReplaceModifiedPackageNotice
    do: [:exception | exception resume: true] "This will cause the load to
proceed"


["Code that attempts to load a Bundle or Package"]
    on: ReplaceModifiedPackageNotice
    do: [:exception | exception resume: false] "This will cause the load to
be canceled"
```

### Store.StoreNewVersionWarning

Raised during a save if the value of WarnIfNewerVersionPublished is true, and a newer version of the package or bundle already exists in the repository.

```
["Code that attempts to save a Bundle or Package"]
    on: StoreNewVersionWarning
    do: [:exception | exception resume: true] "This will cause the save to
proceed"


["Code that attempts to save a Bundle or Package"]
    on: StoreNewVersionWarning
    do: [:exception | exception resume: false] "This will cause the save to
be canceled"
```

### Store.StorePublishingError

Raised when saving a bundle or package and some otherwise unhandled publishing error occurs.

```
["Code that attempts to save a Bundle or Package"]
    on: StorePublishingError
    do: [:exception | exception return] "This will prevent the error from
raising a warning dialog. The publish will not proceed"
```

### Store.ViewUnloadableDefinitionsNotification

Raised after a load if an unloadable definition error viewer is asked to be opened. If this is not handled, that viewer is opened; otherwise, this object is raised.

If handled, the instance variable unloadableDefinitionErrors contains a collection of all UnloadableDefinitionError objects created during the load.

```
["Code that attempts to load a Bundle or Package"]
    on: ViewUnloadableDefinitionsNotification
    do: [:exception | "Code that may process exception
unloadableDefinitionErrors"
        exception return] "This will prevent the error from opening the
unloadable definitions browser. The load will proceed"
```

### Store.WasConvertedFromParcelWithUndeclaredError

If an image package or bundle is marked as read-only, this notification is raised when attempting to publish the package or bundle.

By default, Store does not mark bundles or packages read-only if the source parcel has undeclared objects.[

```
"Code that attempts to save a Bundle or Package"]
    on: WasConvertedFromParcelWithUndeclaredError
    do: [:exception | exception return] "This will prevent the error from
raising a warning dialog. The publish will not proceed"
```

# 8

# Administering Store

## User Administration

Users who will publish to and load code from the VisualWorks database, as well as users who might only have administration responsibilities, must be assigned a login account for the host database. These accounts are normally created by the database administrator, using database administration facilities. The Store user then enters the account name and password, if applicable, in the Store connection dialog to access the repository.

In addition to the database user accounts, if user/group management is installed, users also have to be defined in Store in order to take advantage of the privilege control features. Adding and specifying access rights at this level is all done within Store.

### Adding Store users

There are three general classes of Store user: (1) the Store table owner, that you created for installing the Store tables into the database, (2) “Administrator” users who can do Garbage Collection and other administrative actions, and (3) “normal” Store users. There are three ways of creating these kind of users, either assigning them global database-wide rights, by assigning rights for Store tables and sequences to a role or group and assigning users to those groups, or by assigning rights for the Store tables and sequences to each user depending on their general class.

Not all installations need Administrator users. Smaller installations with a small number of users can use the Store table owner, by default BERN, to do administration tasks.

### Using global database rights

The Store table owner, by default BERN, needs the fullest capabilities. This user needs sufficient privilege to physically modify the database structure.

For Oracle, the table owner needs to be created with these roles and privileges:

| | |
|---|---|
| Roles: | CONNECT |
| | RESOURCE |
| Privileges: | EXECUTE ANY PROCEDURE |
| | DELETE ANY TABLE |
| | INSERT ANY TABLE |
| | SELECT ANY SEQUENCE |
| | SELECT ANY TABLE |
| | UNLIMITED TABLESPACE |
| | UPDATE ANY TABLE |

For SQL Server, the table owner needs these permissions:

| | |
|---|---|
| Object Permissions | SELECT |
| | INSERT |
| | UPDATE |
| | DELETE |
| | EXECUTE |
| Statement Permissions | CREATE DATABASE |
| | CREATE TABLE |

For PostgreSQL, the table owner needs to be able to create the database, so the command line must include the -d switch. It is useful for the user to be able to add users, too, indicated by the -a switch, to the command to create this user is:

```
#> createuser -a -d -P <username>
```

For other databases, equivalent permission sets should be assigned to this user.

### Administrator user accounts

Administrator user accounts need almost the same rights as the table owners, but they do not need rights to create databases or tables.

The required permissions for Oracle are:

| | |
|---|---|
| Roles | CONNECT |
| | RESOURCE |
| Privileges | EXECUTE ANY PROCEDURE |
| | DELETE ANY TABLE |
| | INSERT ANY TABLE |
| | SELECT ANY SEQUENCE |
| | SELECT ANY TABLE |
| | UNLIMITED TABLESPACE |
| | UPDATE ANY TABLE |

For SQL Server, the administrator needs these permissions:

| | |
|---|---|
| Object Permissions | SELECT |
| | INSERT |
| | UPDATE |
| | DELETE |
| | EXECUTE |

For PostgreSQL, the user doesn't need to be able to create the database, and doesn't need to add users, so the command line can include the -D and -A switches. Accordingly, the command to create this user is:

```
#> createuser -A -D -P <username>
```

For other databases the specific permissions will be different, but equivalent.

### Normal user accounts

Normal user accounts need slightly fewer permissions, since their activities only involve reading and updating the database table records.

The required permissions for Oracle are:

| | |
|---|---|
| Roles | CONNECT |
| | RESOURCE |
| Privileges | SELECT ANY SEQUENCE |
| | SELECT ANY TABLE |
| | UNLIMITED TABLESPACE |
| | UPDATE ANY TABLE |
| | INSERT ANY TABLE |

For SQL Server, the user needs these permissions:

| | |
|---|---|
| Object Permissions | SELECT |
| | INSERT |
| | UPDATE |
| Statement Permissions | (none) |

For PostgreSQL, the user doesn't need to be able to create the database, and doesn't need to add users, so the command line can include the -D and -A switches. Accordingly, the command to create this user is:

```
#> createuser -A -D -P <username>
```

Again, for other databases the specific permissions will be different, but equivalent.

## Using Groups and Table Rights

The Database Administrator creates two Groups/Roles, assigns rights to those roles for the Store tables, and later assigns users to a Group or Role, allowing them to inherit their rights from that Group/Role. The three Groups/Roles are the Store table owner, the Store Administrator, and the Store User. The Store User only performs regular loading, publishing, and merging during everyday development. The Store Administrator has special rights allowing him/her to perform Garbage Collection, and rename code components.

Store uses 25 tables: TW_BinaryBlob, TW_Blessing, TW_Blob, TW_Bundle, TW_Bundles, TW_ClassRecord, TW_Data, TW_DatabaseIdentifier, TW_DataElement, TW_DBPundlePrivileges, TW_DBUserGroup, TW_FileRecord, TW_Files, TW_LoadRecord, TW_Method, TW_MethodDocs, TW_NameSpaceRecord, TW_Package, TW_Packages, TW_ParcelRecord, TW_PkgClasses, TW_PkgNameSpaces, TW_Properties, TW_PropertyRecord.

Store also uses 12 sequences: TW_BinaryBlob_Sequence, TW_Blessing_Sequence, TW_Blob_Sequence, TW_Bundle_Sequence, TW_ClassRecord_Sequence, TW_DataElement_Sequence, TW_FileRecord_Sequence, TW_Method_Sequence, TW_NameSpaceRecord_Sequence, TW_Package_Sequence, TW_ParcelRecord_Sequence, TW_PropertyRecord_Sequence.

The two Groups/Roles are Store Administrator and Store User.

The Store Administrator Group/Role needs Select, Insert, Update and Delete rights to all Store tables, as well as Select rights to the Sequences.

The Store User Group/Role needs Select and Insert on all Store tables, as well as Update rights on the TW_Bundle and TW_Package tables. It also needs Select rights on all Store Sequences.

Once these Group/Roles are created, individual Store users can be assigned to a specific Group/Role, and they will then have all the appropriate rights.

### Assigning rights directly

Following this method, as each user is created, the rights described in the previous section are assigned depending on whether the user is considered an administrator or normal user of Store.

## Setting up development behavior of users and groups

User groups provide Store with a mechanism for controlling which users can publish at various blessing levels, and for assigning package owners and access. Accordingly, it is a mechanism in which a team can enforce some level of its development processes.

This mechanism is distinct from rights in the database in that it restricts rights in the image who can load and publish bundles and packages, and at what level of blessing. It does not grant or revoke any database level rights.

### Installing user/group management

VisualWorks can optionally enforce user and group access restrictions. To configure this option:

1 If you did not install user management while setting up Store, evaluate this expression to add management support to the Oracle database:

```
Store.Privileges installUserManagement.
```

2 When prompted, log on as the table owner (such as BERN).

3 When prompted for an image administrator, enter a user name.

You must enter a name different than the table owner/database administrator you are logged on as. The user should have normal user privileges (not table owner), but will be assigned to the ADMINISTRATOR group.

The two additional tables, TW_DBUserGroup and TW_DBPundlePrivileges, are then created in the database.

4 In each image, or in the baseline image to be distributed to users, evaluate these expressions:

```
Store.Policies blessingPolicy: OwnerBlessingPolicy new.
Store.Policies ownershipPolicy: OwnerOwnershipPolicy new.
```

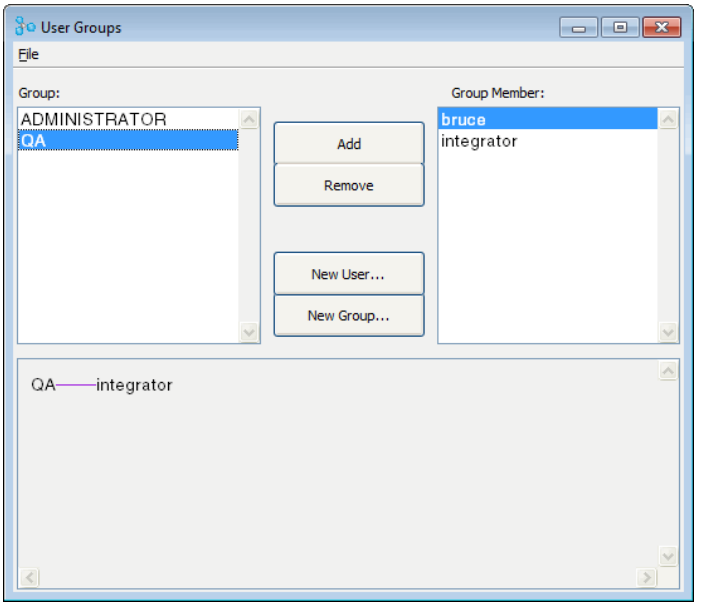
You may substitute your own policy methods, overriding these methods. See Configuring Store Policies for information.

5 Create Store users (see "Add a user") corresponding to the database users you have created (using database utilities), and assign them to groups.

## Configuring user groups

User group configuration is done using the User Groups tool. To open the tool, connect to the repository as the image administrator, then select **Store > Administration > User/Group Management**.

Initially there are two default groups, ADMINISTRATOR and QA, and one user, the image administrator (**integrator** in this example):




The ADMINISTRATOR group is special, in that members of this group have access to the administration utilities, including User/Group Management. The image administrator named when installing User/Group Management is in this group.

Group memberships are shown in a graph. Select one or more groups and/or users to see the graph.

### Add a group

To create a new group, click New Group... Enter a group name, such as **DEVELOPERS**, in the prompter, and click **OK**.

You cannot remove groups in this tool, but can do so in the database table itself using database administration tools.

### Add a user

To add a user, select the group or groups to which the user will belong, and click **New User...** Enter the user name in the prompter and click **OK**. The user is added to Store as belonging to the selected groups.

The user name should be the same as a defined user ID, so the two can be associated and controlled properly.

### Change group membership

To add a user to a group, select the user and the group and click **Add**.

To remove a user from a group, select the user and the group and click **Remove**.

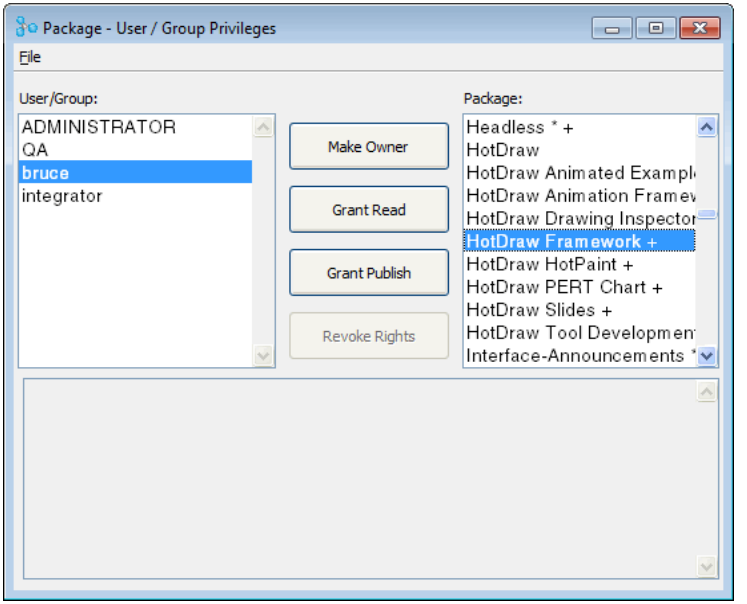
### Delete a user

When a user has been removed from all groups, the user is also removed from the user list when the tool is closed.

## Assigning privileges

Ownership, and read and publish privileges can be restricted by assigning these to users or to groups. Use the User/Group Privileges tool to assign these access rights.

To open the tool, connect to the repository as the image administrator, and select **Store > Administration > Package Ownership**:




To assign a privilege, select the package and the user or group. Then click **Make Owner**, **Grant Read**, or **Grant Publish** privilege.

## Garbage collecting the database

At the end of a project you will have accumulated a lot of versions of packages and bundles in your database that are not useful for continued development. Under normal conditions, since Store employs a versioning database, nothing is ever deleted. The Store garbage collector gives the database administrator a way to clear out versions of objects that are no longer needed.

During garbage collection, Store identifies definitions required by versions that are not being removed, and assigns them to versions that are remaining in the database. In this way, although versions only store “deltas,” all required definitions are preserved.

You must be connected as the database administrator to start this utility. Also, because this is an administrative utility, you need to inform users of what you are doing, and advise them to reconcile

their images with the database, if required. Reconciling will be necessary for any image that has a version loaded that has been garbage collected.

Note that if a version is currently loaded in the image from which garbage collection is being run, that version should not be garbage collected. Doing so will cause the image to be inconsistent.

Garbage collection used to be a very slow and memory-intensive process, and so would normally be done at the end of a project, and *after* the source has been archived for future retrieval. Unnecessary legacy versions can then be cleared out before further development is done.

Beginning in VisualWorks 7.10, the garbage collector has been made vastly more efficient, and the speed has been increased especially when a large number of components are being deleted. Nonetheless, it is a process that permanently deletes code from your repository, so care should be taken when deciding when or if it should be used.

## Loading and Configuration

To load the garbage collector, use the Parcel Manager to load Store-GarbageCollection (located in the **Version Control** category).

Under the hood, the new Store garbage collector employs complex SQL queries that use the IN clause, with a list of values. All databases have a limit on the number of values for an IN clause, but there are other factors that can limit the number of items, such as size of the SQL statement string, or real performance.

By default, the garbage collector uses a fairly safe value of 500, as defined by the Glorp framework. You can change this, as follows:

```
Store.GarbageCollector defaultINClauseLimit: <anIntegerOrNil>
```

Once set, this value is remembered for the lifetime of the image. Setting it to nil will reset the value to 500.

Once you begin to delete large numbers of packages, the possibility of running into this limit increases for the total number of methods and classes contained in the targeted packages and bundles. Therefore, the garbage collector will, by default, process components in groups of 25, which greatly reduces the possibility of running into these limits. This value, too, can be changed:

```
Store.GarbageCollector pundleLoopSize: <anIntegerOrNil>
```

As with the IN clause limit, once set, this value will be remembered for the lifetime of the image, and setting it to nil will restore the default of 25.

## Building Indexes

Because of the changes to the garbage collector in VisualWorks 7.10, new indexes are suggested for your store database. New repositories created by a 7.10 image will automatically have these indexes created for them.For repositories created using older versions of VisualWorks, the indexes can be created by evaluating:

```
DbRegistry update710.
```

While the garbage collector will work without these additional indexes, the performance could be extremely poor.

These indexes can be also be created manually with a suitable database administration tool using the following templates.

For DB2, Oracle, PostGreSQL and SQLite3:

```
CREATE UNIQUE INDEX NAMESPACES_GC ON TW_PKGNAMESPACES
(NAMESPACEREF, PACKAGEREF);

CREATE UNIQUE INDEX CLASSES_GC ON TW_PKGCLASSES
(CLASSREF, PACKAGEREF);

CREATE UNIQUE INDEX METACLASSES_GC ON TW_PKGCLASSES
(METACLASSREF, PACKAGEREF);

CREATE UNIQUE INDEX SHARED_VARIABLES_GC ON TW_DATA
(DATAREF, PACKAGEREF);

CREATE UNIQUE INDEX METHODS_GC ON TW_METHODS
(METHODREF, PACKAGEREF);

CREATE UNIQUE INDEX PROPERTIES_GC ON TW_PROPERTIES
(PROPERTYREF, PUNDLEREF);
```

For SQLServer, the following code is required:

```
CREATE UNIQUE INDEX NAMESPACES_GC ON
NEWBERN2.dbo.TW_PkgNameSpaces (nameSpaceRef, packageRef);

CREATE UNIQUE INDEX CLASSES_GC ON
NEWBERN2.dbo.TW_PkgClasses (classRef, packageRef);

CREATE UNIQUE INDEX METACLASSES_GC ON
NEWBERN2.dbo.TW_PkgClasses (metaclassRef, packageRef);
```

```
CREATE UNIQUE INDEX  SHARED_VARIABLES_GC ON
NEWBERN2.dbo.TW_Data (dataRef, packageRef);

CREATE UNIQUE INDEX METHODS_GC ON
NEWBERN2.dbo.TW_Methods (methodRef, packageRef);

CREATE UNIQUE INDEX PROPERTIES_GC ON
NEWBERN2.dbo.TW_Properties (propertyRef, pundleRef);
```

Depending upon the installation, dbo may need to be changed to BERN.

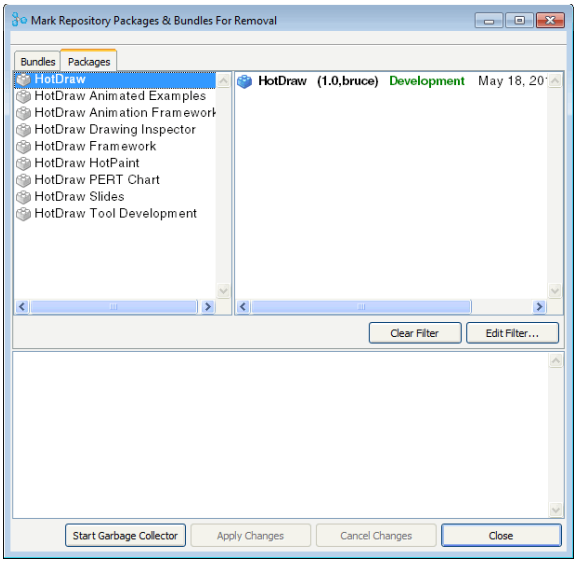
Furthermore, on DB2, the following REORG statements are often required to be performed using the DB2 command line tool:

```
REORG TABLE BERN.TW_METHOD

REORG TABLE BERN.TW_DATAELEMENT

REORG TABLE BERN.TW_PACKAGE

REORG TABLE BERN.TW_CLASSRECORD

REORG TABLE BERN.TW_NAMESPACERECORD

REORG TABLE BERN.TW_BUNDLE
```

Again, depending on your installation, you may need to replace BERN with the schema table owner name.

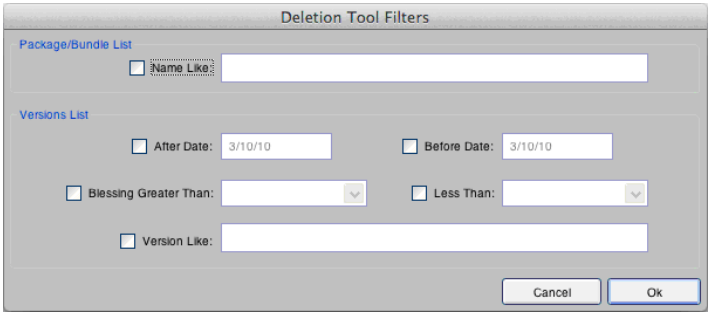
## Using the Garbage Collector

To open the Store Garbage Collector, choose **Store** > **Administration** > **Garbage Collection** in the Visual Launcher.




The packages published in the database are listed in the left pane. Select items to garbage collect, and click **Add to garbage** to move them to the **Garbage** list. Only packages in the **Garbage** list are checked and garbage collected.

You may also set several filters for limiting the view in the top panes by pressing the **Edit Filter...** button. This opens a dialog which allows you to set the following filters applied to the displayed components and versions:



**Name Like**

Filter based on the name of the component using the common '*match*' format.

**After Date, Before Date**

Filter based on a range of the versioned published dates of the components.

**Blessing Greater Than, Blessing Less Than**

Filter based on a range of version blessing levels.

**Version Like**

Filter based on the string in the version for the component, using the common '*match*' format.

You can also clear all set filters by pressing the **Clear Filter** button.

The bottom pane shows all packages and bundles you have added. If they have a black **X** on their icons, they are not yet marked for deletion. To move all items in the bottom pane from in queue state to Marked For Deletion state, press the **Apply Changes** button. This will mark the selected components for deletion and change their icon to have a red **X**.

Once you have components for deletion, click the **Start Garbage Collector** button. If the Store-GarbageCollection parcel is not loaded, nothing will happen, but the window will close.

Otherwise, the window will close while the garbage collector scans all the definitions in those components for methods and other objects that are not referenced in any remaining packages, and removes them from the database.Deleting a bundle only deletes the container that represents that bundle, not any of its contents. Package versions, or other bundle versions which are contained within a bundle targeted for deletion, will become orphaned.

To completely remove a bundle and all of its contents, you must not only select that bundle but also all of its contained packages and bundles.

# Checking consistency

It is occasionally valuable to verify the consistency between package contents and the image. This is done by selecting **Store > Check Consistency** in the Visual Launcher. The command will either inform you that the image is consistent, or issue a warning of errors.

The check is an internal model consistency check, verifying, for example, that all classes and methods in packages actually exist in the image, and that there is no confusion about which package owns a class definition.

If errors are discovered, you may need to execute:

```
Store.Registry makeConsistent
```

which attempts to correct several errors.

# 9

# Store Setup for DBAs

With Store, there is generally little setup required for the database backend, beyond having the database itself installed and a user account defined to be used by the Store administrator.

This appendix provides instructions that summarize only the steps to be performed by a Database Administrator (DBA), prior to the Store administrator's task of installing the database tables. These steps may be useful for enterprise organizations with a dedicated DBA.

## Supported Database Platforms

VisualWorks can use several database back-ends for code storage.

Currently, VisualWorks supports:

- Any Oracle 7 or later database, except Oracle Lite which is not supported.
- SQL Server version 2000 or later is supported on Windows platforms.
- DB2
- SQLite3

In addition, third-party supported backends are available for

- PostgreSQL
- InterBase

For full Store installation instructions, see Configuring Store.

## Set Up Oracle

1 Using the database administration tools, create a database administrator account, with the roles CONNECT and DBA.

The default DBA account name is BERN. If you use another name, set the **Database table owner** in **Store** > **Settings** to that name before building the tables in step 3.

2 Create a directory to hold the VisualWorks data files.

## Set Up SQL Server

When installing SQL Server, you have a choice of making it case sensitive or case insensitive. It is important, for the proper operation of Store, that it be installed *case sensitive*.

1 (Optional) Using the SQL Server Enterprise Manager, create a database owner account.

The default database owner account name is BERN. To use another name, set the **Database table owner** in the **Store** > **Settings** to that name before building the tables in step 3.

2 Create a directory (for example, \visualworks\packages) to hold the Store data files.

## Set Up PostgreSQL

PostgreSQL support for Store is provided as a goodie and is supported by its developer. For updated and more complete information, refer to http://sourceforge.net/projects/st-postgresql/.

Assuming you already have a PostgreSQL database installed and configured for normal access, do the following to set up Store:

1 Log on as the PostgreSQL owner.

2 Create a database owner account for Store, by executing at the command prompt:

```
#> createuser -d -a -P <username>
```

The default Store database owner account name is BERN. To use another name, the table owner will have to set the **Database table owner** in the **Store** > **Settings** in Store before building the tables.

3 Create the database in PostgreSQL, by executing at the command prompt:

```
#> createdb <dbname>
```

This creates the database in the directory set in $PGDATA, usually /var/lib/pgsql, but may differ for your installation. To create it in a different directory, use the -D switch:

```
#> createdb -D <dbpath> <dbname>
```

Refer to the createdb manpage for command details.

## Set Up DB2

These instructions are extracted from the instructions provided by the developer (goodies/other/db2/doc/db2connect.pdf).

1 Create new DB2 database

Example below: On Windows run “DB2 Command Window” and then execute:

```
db2 create database myStore on D
```

where myStore is the database name, and D is the location (drive D:).

On Linux execute:

```
db2 create database myStore on /usr/mystore
```

where /usr/mystore path to database files.

2 Change some database parameters:

Execute (it’s single command):

```
db2 update db cfg for mystore using
APP_CTL_HEAP_SZ 512 LOGSECOND 50
```

## Set Up Interbase

These instructions are extracted from the instructions provided by the developer (goodies/other/InterBase/doc/ibusing.html).

Interbase and Firebird databases and instructions for their installation are available at http://ibphoenix.com/.

Now you can:

- create database

- add new user account;
- test of connection;

# 10

# Creating a Custom Install Script

You can create a custom installation script for special purposes. A script allows you to change the directory path and the tablespace names, and to customize table access rights. The script can be executed from within VisualWorks, or saved as a file and executed as a SQL script by a database administrator.

This script option is particularly important for tightly-controlled database environments, in which the database administrator carefully controls how tables are created and access is granted. The script, which contains SQL, can be submitted for review, modification, and execution.

## Creating an Installation Script

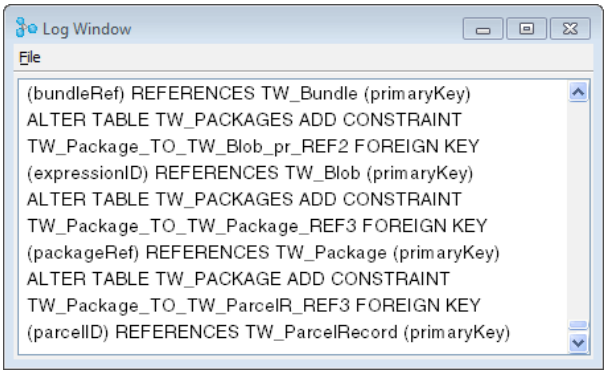
To create a script, evaluate this expression in a workspace:

```
Store.DbRegistry createInstallScript
```

This opens a connection dialog, in which the only desired value is the interface class. Select the appropriate class for your database (e.g., OracleConnection, PostgresSocketConnection, MS_SQLServerConnection, etc.) and click the **Connect** button. In this specialised use of the dialog, this does not cause any actual connection. Instead, it provides the class to the script, which now opens the Log window.

As the script appears in the Log window, you will be prompted with the same questions you would see if you were in fact connected to a database of that type and creating the Store schema: i.e., what is the name of the database, (if appropriate) where is its table space, whether the image-side management policies should be supported, whether the database should support connections from VisualWorks 7.6 and older images.

The script to create the files and build the tables is created in the Log Window.




Edit this script as needed.

The script can then be saved to a file (using **File > Save As…**), or simply copied and pasted. Probably, you are creating a script because you do not wish to, or do not have permissions to, create the schema in situ directly from VisualWorks; supply the script to your DBA or other authorised person to run.

If you wish to run the script from VisualWorks, you can open the Ad-Hoc SQL tool from the **Tools > Database** menu in the Launcher window, connect to the database and execute the script's statements. Additional details on this tool may be found in the *Database Application Developer's Guide*.

# Index