Cincom®

Cincom Smalltalk™

Source Code Management Guide

P46-0138-03

Part Number: P46-0138-03

Software Release 7.5

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 ix

Audience ix
Conventions ix
Typographic Conventions ix
Special Symbols x
Mouse Buttons and Menus xi
Getting Help xi
Commercial Licensees xi
Non-Commercial Licensees xii
Additional Sources of Information xiii

Chapter 1 Introduction

Store Features 1-1
An Open and Distributed Technology 1-2
Concurrent Development 1-3
A Development Methodology 1-3
Versioning 1-4
Parallelism 1-4
Blessing levels 1-5
Publishing policies 1-6
Database limitations 1-7

Chapter 2 Beginning to Use Store

A simple approach 2-1
Assumptions 2-1
Install Store into VisualWorks 2-2
Install the Store database tables 2-4
Publishing the base 2-5
Explore the system contents 2-6
Load application code 2-8
Loading parceled code into Store 2-8
Publishing packages 2-10

Loading and reorganizing HotDraw 2-13
Build a bundle 2-17
Publish the bundle 2-19
Comparing with another repository 2-21
Get Open Repository access 2-21
Reconciling to the repository 2-21
Browsing differences 2-22
Adopting a difference 2-25
What we changed in this section 2-25

Chapter 3 Configuring Store

Loading Store into VisualWorks 3-1
Configuring the Store database 3-2
Oracle setup 3-2
SQL Server setup 3-4
PostgreSQL setup 3-5
Publishing the VisualWorks base 3-7
Making changes to the base 3-9
Updating to a new base 3-9
Team working environments 3-9
Local and shared repositories 3-9
Configuring Store policies 3-10
Installing a policy 3-11
Blessing Policy 3-11
Merge Policy 3-13
Ownership Policy 3-13
Package Policy 3-14
Prerequisite Policy 3-14
Publish Policy 3-14
Version Policy 3-15

Chapter 4 Organizing Code in Store

Patterns for organizing code 4-1
Guidelines for defining packages 4-1
Guidelines for defining bundles 4-2
Using bundles to organize projects 4-3
Prerequisites and load orders 4-4
Suggestions for development dependencies 4-5
Suggestions for deployment dependencies 4-6
A simplified approach 4-6
Importing code into Store 4-7
Packaging source in the image 4-7

Packaging source from file-outs 4-7
Packaging code from parcels 4-8
Managing package content 4-8
Creating packages 4-8
Assigning new definitions to packages 4-9
Moving definitions to packages 4-9
Specifying prerequisites 4-10
Suppress warnings 4-11
Specify prerequisite version 4-11
Package load and unload actions 4-12
Initializing and finalizing packages 4-12
How a package is loaded 4-12
Managing bundle content 4-13
Creating and arranging bundles 4-13
Editing a bundle specification 4-14
Specify prerequisites 4-14
Suppress warnings 4-15
Specify prerequisite version 4-15
Bundle load and unload actions 4-15
Including external files 4-15
Publishing packages and bundles 4-16
Basic publishing 4-16
Publish binary 4-18
Publish parcel 4-19
Overriding definitions 4-21
Reorganizing packages 4-22
Renaming a package or bundle 4-22
Reorganizing namespaces 4-22

Chapter 5 Maintaining Your Store Environment

Overview 5-1
Working connected and disconnected 5-2
Connecting to the database 5-2
Detaching from the database 5-2
Saving connection profiles 5-3
Working off-line 5-3
Preparing to work off-line 5-4
Resuming work with the database 5-4
Working with multiple databases 5-5
Reconciling to a database 5-6
Switching databases 5-7
Removing database links 5-8

Using a local database 5-8
Publishing back to the team database 5-9
Maintaining your working image 5-9
Browsing loaded packages and bundles 5-9
Examining the contents of a bundle 5-9
Loading published code 5-10
Loading a bundle 5-10
Loading a package 5-10
Updating to new versions 5-11
Browsing published bundles and packages 5-11
Browsing the Definitions in a Published Version 5-12
Browsing packages and definitions 5-12
Browsing loaded code 5-12
Browsing unloaded code 5-12
Browsing shared variable definitions 5-13
Browsing with package changes and overrides 5-13
Updating published source code 5-14
Updating from a build 5-14
Publishing packages 5-14
Pre-publication checks 5-15
Comparing to the parent version 5-15
Inspecting changes 5-15
Merging with another version 5-15
Publishing a bundle 5-15
Publishing an individual package 5-17
Publishing changes 5-19
Exporting code 5-20

Chapter 6 Version Control

Versions 6-1
Package and bundle version strings 6-2
Blessing levels 6-2
Working with versions and blessings 6-3
Browsing a version history 6-3
Comparing versions 6-4
Package views 6-4
Class views 6-4
Protocol and method views 6-5
Text views 6-5
Changing a version's blessing level 6-5

Chapter 7 Integrating Versions

Comparing versions 7-1
- Package views 7-2
- Class views 7-2
- Protocol and method views 7-2
- Text views 7-2

Changing a version's blessing level 7-3
Integrating code versions 7-3
- Relationships among versions 7-4
- Conflicting and nonconflicting modifications 7-5
- Merging two versions of a package 7-5
- Integrating a set of packages 7-6
- Resolving conflicts 7-7
- Excluding nonconflicting modifications 7-8
- Creating the merged version 7-8

Chapter 8 Administering Store

User Administration 8-1
- Adding Store users 8-1
  - Table owner accounts 8-1
  - Normal user accounts 8-2

Setting up users and groups 8-3
- Installing user/group management 8-3
- Configuring user groups 8-4
  - Add a group 8-5
  - Add a user 8-5
  - Change group membership 8-6
  - Delete a user 8-6
- Assigning privileges 8-6

Garbage collecting the database 8-7
Checking consistency 8-9

Appendix A Porting from ENVY/Developer A-1

Conceptual porting A-2
Using file-outs and parcels A-3
Store Bridge A-4
- Compatibility A-5

Installing the bridge A-5
- Installing the Bridge in the ENVY environment A-5
- Installing the Bridge in the Store environment A-6

Exporting an ENVY configuration map A-7

Importing and publishing an ENVY configuration map A-8
Publishing a converted bundle to the Store repository A-9
Migrating complete version history A-10
Known limitations A-11

Appendix B Store Setup for DBAs B-1

Set Up Oracle B-2
Set Up SQL Server B-3
Set Up PostgreSQL B-4
Set Up DB2 ..... B-5
Set Up Interbase B-6

Appendix C Creating a Custom Install Script C-1

Index Index-1

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.

For VisualWorks 7.5 this document is still under development, and is incomplete in several areas. Nonetheless, it is more comprehensive than the documentation formerly provided in the *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.

Example	Description
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`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 <Select> button <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. For more information, send email to supportweb@cincom.com.

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:

supportweb@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 Non-Commercial 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 several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe".

- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:

  http://st-www.cs.uiuc.edu/

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://wiki.cs.uiuc.edu/VisualWorks

- A variety of tutorials and other materials specifically on VisualWorks at:

  http://wiki.cs.uiuc.edu/VisualWorks/Tutorials+and+courses

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincom.com/smalltalk/documentation

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

1

Introduction

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 to 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.

An Open and Distributed Technology

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, ODBC, PostgreSQL, 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 VisualWorks.

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.

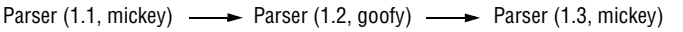
To simplify the task of managing parallel lines of development, a fine-granularity versioning technique is employed. During normal development, all changes are logged locally, and Store records how they impact the working versions of packages in the local image.

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:

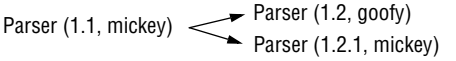


When an updated version is published, a new version string is assigned (e.g., “1.2”), and this version becomes parent of the working version in the image. The working versions in each developer’s image only have version numbers assigned when they are published. They can be tracked and merged because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

Parallelism

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.

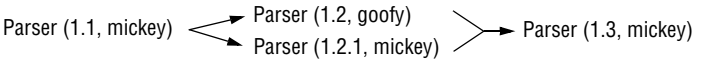


In this case, two developers are working simultaneously on the same package. After they publish their modifications with successive version numbers (“1.2” and “1.2.1”), it will be necessary to perform a step of integration.

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, Oracle limits 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 the simple names, not including the (name space) environment, so 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; 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. The details are up to you.

Assumptions

We need to make some assumptions here, but they're small:

1. You have access to a database into which you can install Store. What this assumption amounts to is that you have:
    - a Store-compatible database installed
    - three database users are created for store:
        - 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 database access string for the database, which is assigned when creating the database

- You know the directory path created for the Store database files, which was created by your database administrator

This is really the only assumption, but it can be a pretty hard one. Instructions for setting up the databases and the requirements for users are given in Chapter 3, "Configuring Store." Only the database level instructions are necessary, because we will install the tables here. If the tables are already set up, then you will be able to skip that part of this walk through.

You also need VisualWorks installed, of course. That's it for assumptions.

Install Store into VisualWorks

I'm assuming for this section that you do not have Store already loaded into a VisualWorks image. If you already have a Store image, you can skip this section. Or you might want to read through quickly to see what was done.

Store is an add-in to VisualWorks that is installed from parcels. There are several parcels, but you only need to pick one to load; the rest are installed by that one.

The parcel you will choose to install is the Store parcel for the kind of database you have available. The options at this writing are:

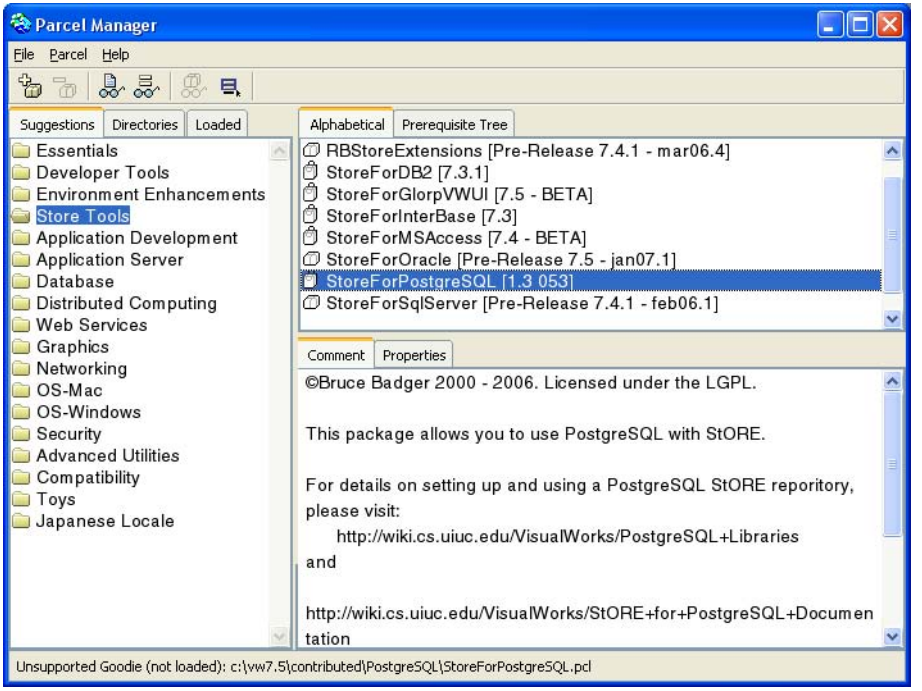
- StoreForOracle
- StoreForSqlServer
- StoreForPostgreSQL
- StoreForDB2
- StoreForInterbase
- StoreForMSAccess
- StoreForGlorpVWUI

Oracle and SQLServer are the only two databases officially supported by Cincom. StoreForMSAccess is provided as a preview. PostgreSQL, DB2, and Interbase support parcels are included with VisualWorks as "goodies," provided by third-party VisualWorks developers, and supported by them. For more information about these, browse the **`contibuted/`** 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 of, select **Store Tools**.

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”. (The image will be saved as **`storeOnly.im`**.)

Install the Store database tables

This section assumes your database does not have Store tables installed already. 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 slanted towards Oracle. For installing and configuring a controlled, multi-user environment, refer to Chapter 8, "Administering 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. When prompted **Create tablespace?**, click **Yes**.
5. 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 Chapter 8, "Administering Store", "Setting up 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

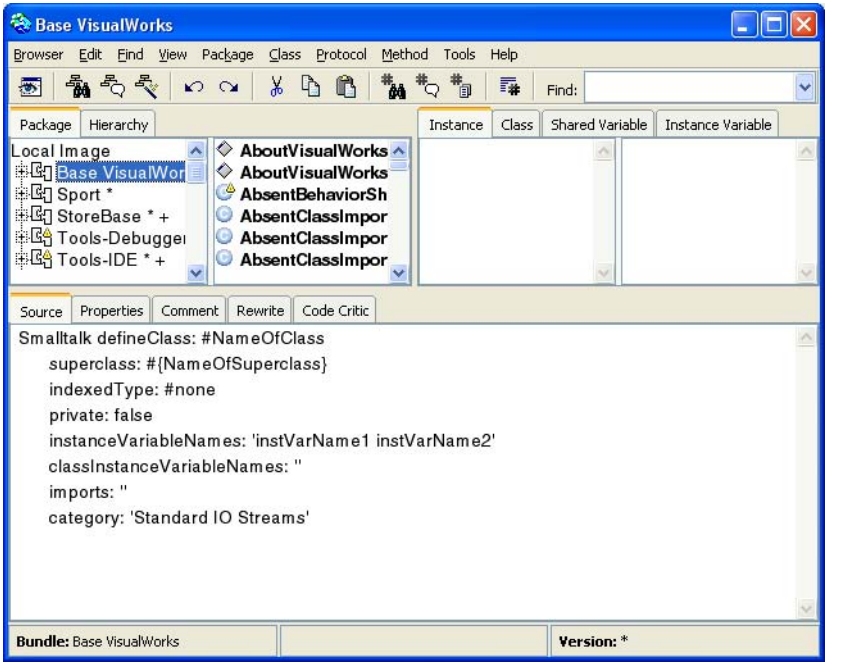
The first thing many Store users do is publish the VisualWorks base into their Store repository. It isn't necessary, but does provide some help in discovering if you have accidentally overwritten a method in the base, which most application programmers do not need to do.

We'll skip doing this at this point, but refer to "Publishing the VisualWorks base" in Chapter 3, "Configuring Store" for more information.

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

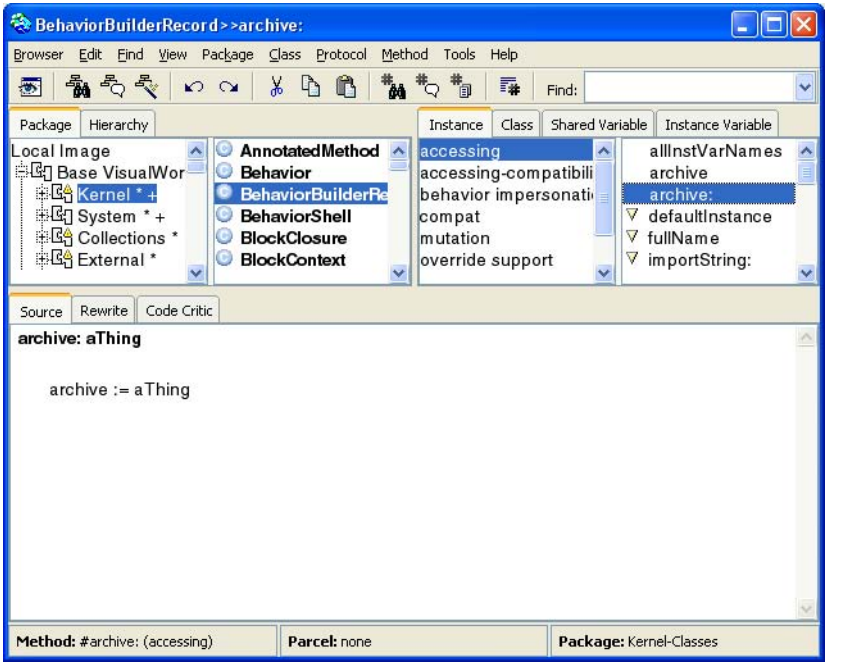
Before going further, let's take a quick look at the system as it stands now. Open the System Browser.



This browser is described in the *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 the section "Importing code into Store" in Chapter 4, "Organizing Code in Store" we describe several ways for packaging existing code. And in Appendix A, "Porting from ENVY/Developer" we cover techniques specific to bringing code from ENVY.

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

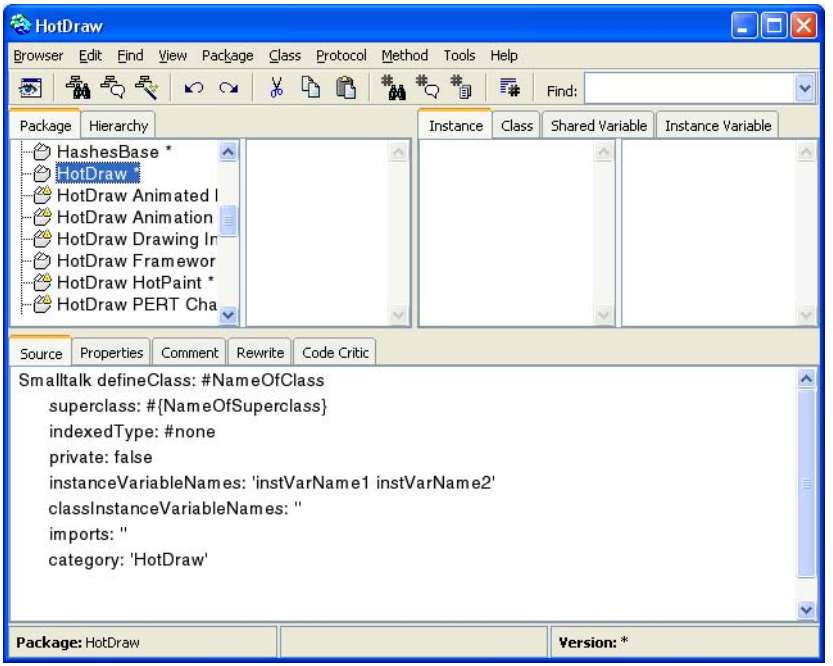
The HotDraw parcels can be loaded using the Parcel Manager. Your own parceled application code can also be loaded using the Parcel manager as long as it is in a directory on the VisualWorks parcel path, though you may have to use the **Directories** list. Otherwise, you will have to use alternative methods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

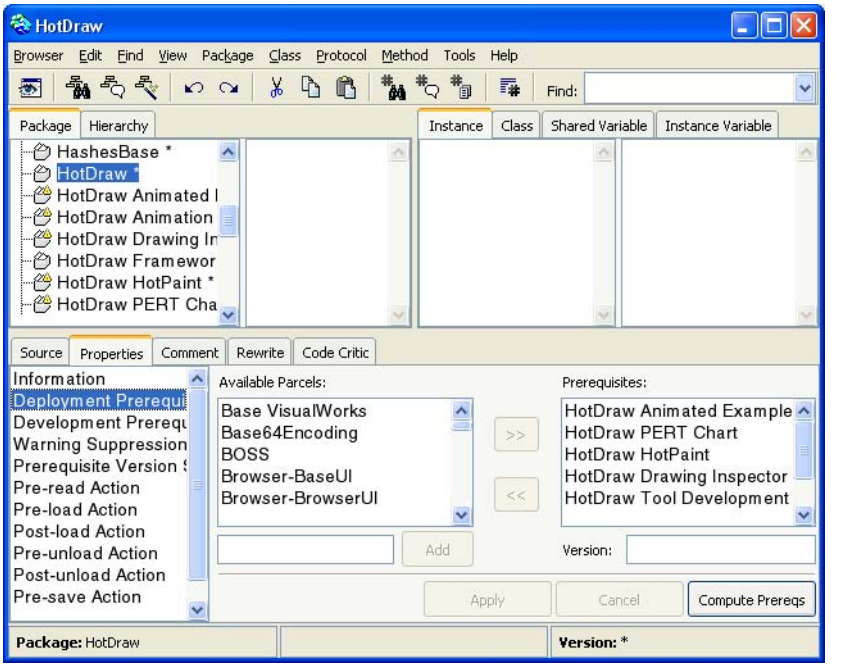
Now open a system browser and browse the results.

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 **Properties** tab and select **Deployment Prerequisites** to see these.



Since these are deployment prerequisites, 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).

But, what about during development? There is also a **Development Prerequisites** property. You can look at it, and see that it is empty; there are no development prerequisites yet. But, before we work with those, we should publish the packages we have now.

Publishing packages

The HotDraw packages have never been published, so they are marked as "dirty," or changed since they were last published. That's the asterisk (*) after the package name, which you can probably see on the HotDraw package, but not on the others unless you resize the package list pane.

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

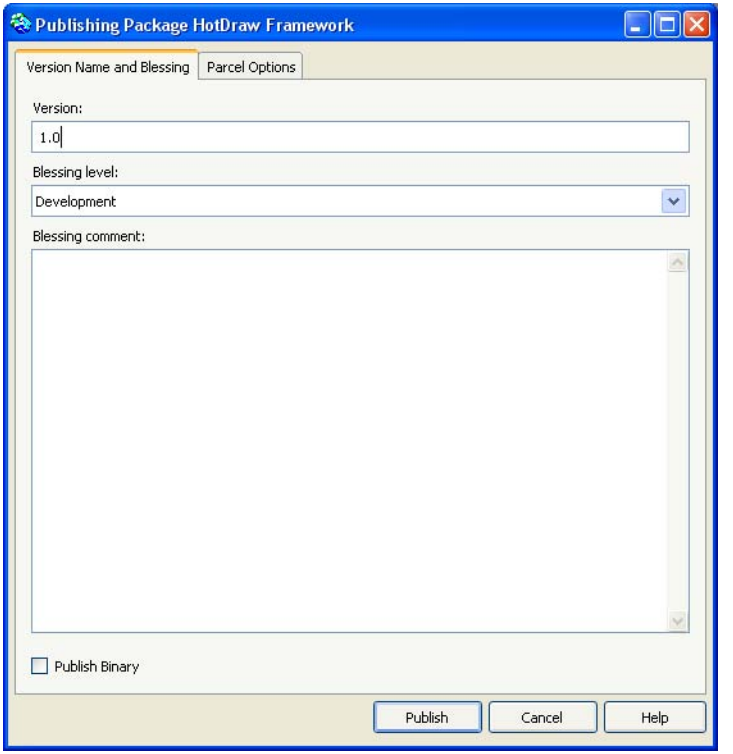
Previously, you have only logged in as the table owner (and possibly as the BaseSystem, if you published the base). Now log on using the same environment string, but use your own user name and password.

The **Table Owner** field may show the table owner, for databases that define owners (e.g., Oracle 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 eight 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 for each of the packages, such as this one for the HotDraw Framework package:

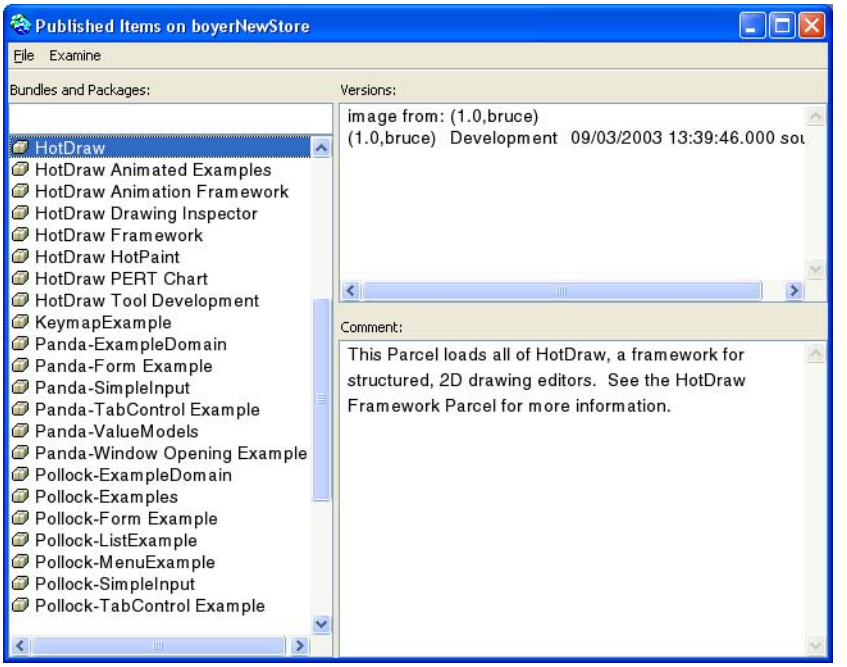


(You might get one or more confirmation dialogs stating that a parent version was expected but not found. This occurs if the package carries a version ID from its source in another database. If these dialogs display, click **Yes** to continue.)

You publish each package separately, since they are not grouped in a bundle. We'll do that later. For now, publish each package individually. The version number of 1.0 is fine for now, and leave the **Blessing level** at Development.

Change the **Blessing comment** to indicate that this is published from the original parcel. We don't need to do anything with the **Parcel Options** page, and we don't need to bother with the **Publish Binary** option. So, click **Publish**. A progress dialog is displayed while publishing. When the first package finishes, repeat the process for each of the other HotDraw packages.

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.

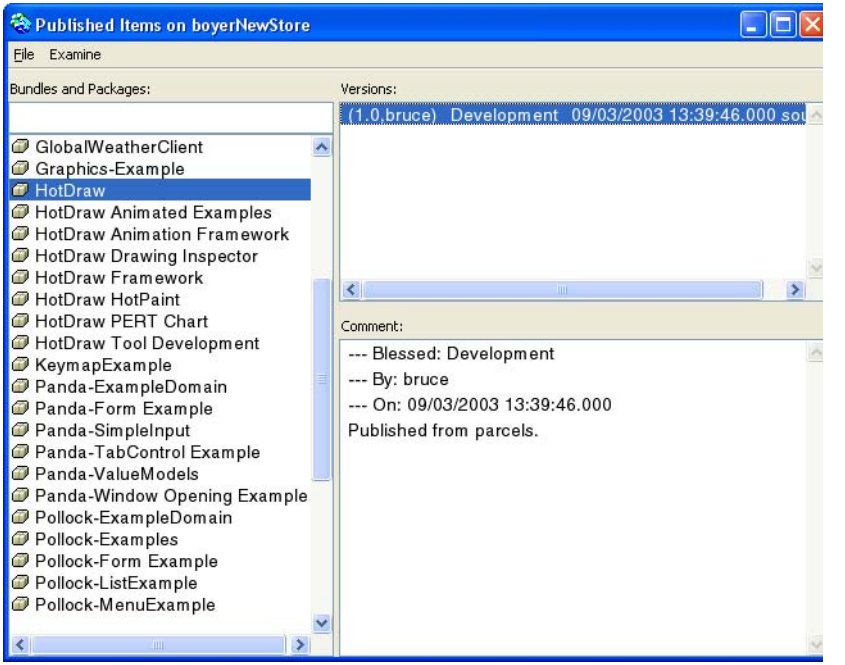


Exit VisualWorks without saving the image. We will load the packages into our image from the database.

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 **Properties** tab and select **Deployment Prerequisites** property. The prerequisites are all there in the **Prerequisites** list, but these are *deployment* prerequisites. Select the **Development Prerequisites** property and see that its **Prerequisites** list is empty.

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. **Deployment Prerequisites** specify what the prerequisites of the resulting parcel will be. But, for development, only **Development Prerequisites** are used.

We could add the equivalent prerequisites for the development prerequisites, but it is better to do this with 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. A few more “Parent expected” dialogs and the workspace open. Click OK on the dialogs and close the workspace. 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 says something like “(1.1,brant).” The same goes for HotDrawFramework.

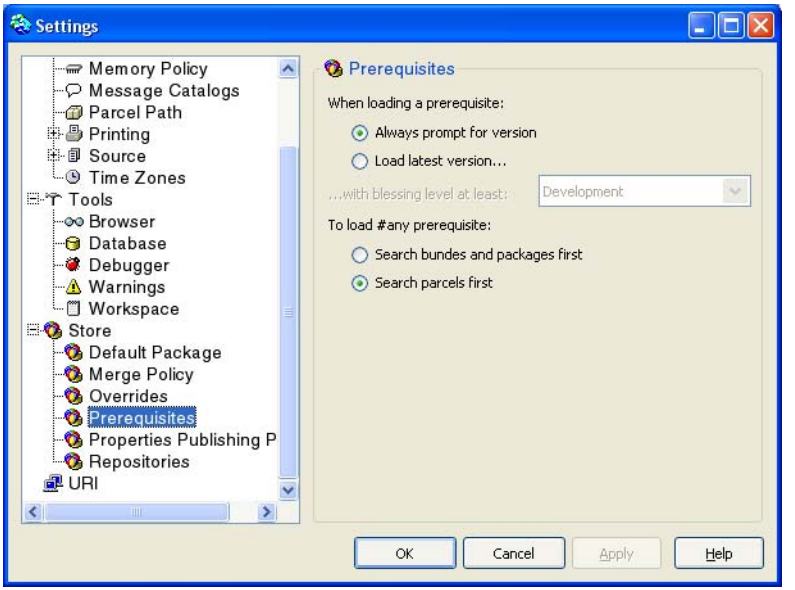
Those last two are not your packages, which should also say “(1.0,yourname).” Instead, loading the HotDraw Animated Examples package loaded two parcels to satisfy its prerequisites!

To see how this happened, first look at HotDraw Animated Examples in a system browser. Select the **Properties** tab, then **Deployment Prerequisites**. The prerequisite it:

Any: HotDraw Animation Framework

The prerequisite can specify either “Parcel” or “Package” to satisfy the prerequisite, or “Any” 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 "Any," 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 both 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 & 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, no “Parent expected” dialogs, and three progress bar dialogs are displayed, one for each of the three packages being loaded. 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 Tool Development

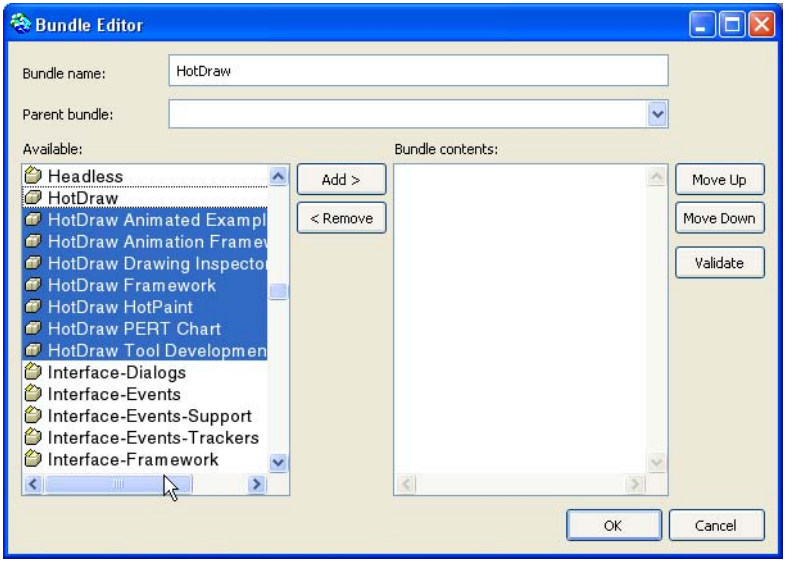
Now, in the right pane, multiple-select a version of each (there’s only one version of each so far, so just pick everything). Then, **File → Load** to load them.

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 there is a check mark next to each.

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 Drawing Inspector
HotDraw Animation Framework
HotDraw Animation Examples

8 Click **Accept**.

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.

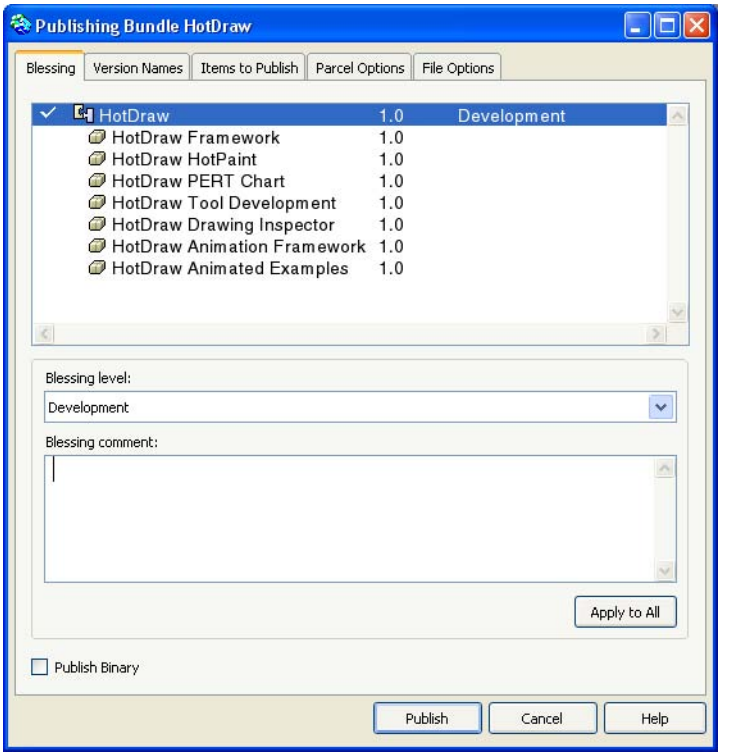
Publish the bundle

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.



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 which you can request access. Visit the Cincom Smalltalk Developer's Wiki:

http://www.cincomsmalltalk.com:8080/CincomSmalltalkWiki

and select "Register For Open Repository Access." Or, go directly to:

www.cincomsmalltalk.com:8008/launch/PostGresRequest

Follow the instructions to apply for an account ID on 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.

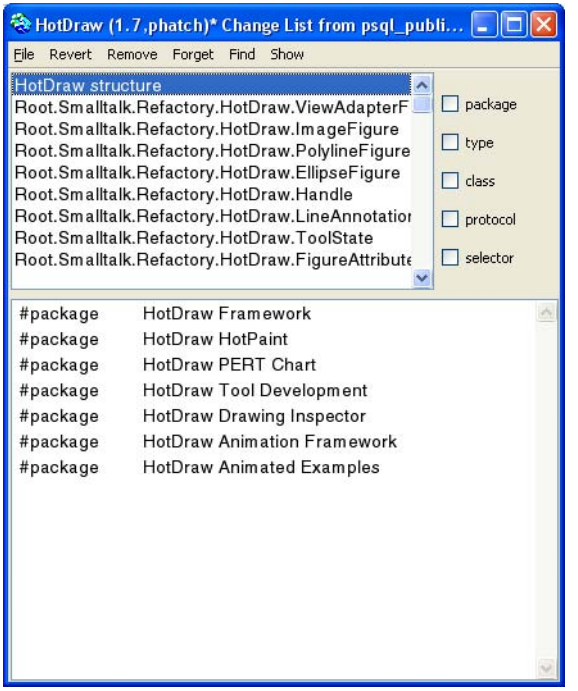
What happens in a reconcile is that Store assigns the selected version as the "parent" version of the code in your image *for this repository*. (Refer to "Reconciling to a database" for more explanation.)

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.

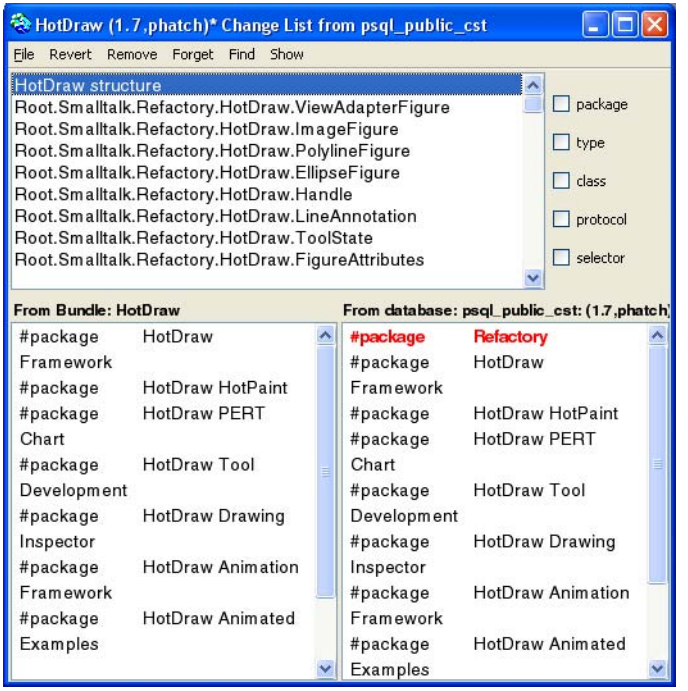
In the system browser, select the HotDraw bundle. First, select **Package → Browse → Change List on Changes**, and pick your repository. It opens on an empty change list, because we have made no changes to HotDraw since loading it. If you have made any changes, they will show up here. Close that Change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

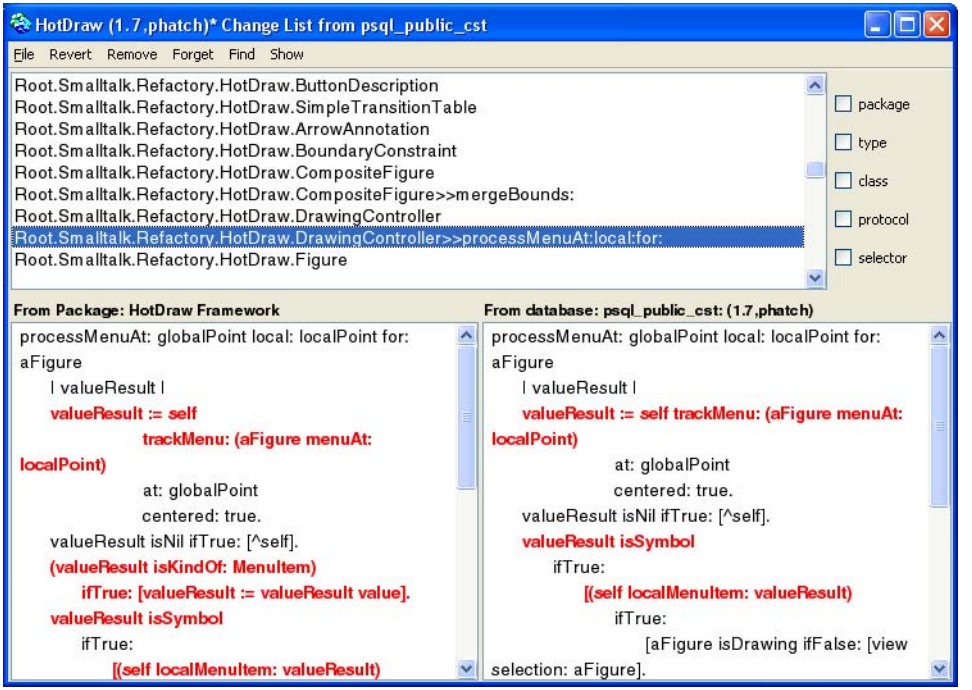
The first item in the list is the bundle structure itself, and it shows the structure of our bundle, the bundle in the image. To compare it to the structure of the bundle in the repository, select **Show → Show conflicts**. The definitions are shown side-by-side, with the differences highlighted in red.



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 are two changes shown here. The first one, if you look closely, is only a format change, without any real difference in the code, while the second has a more substantive difference. To confirm this, select **Show → Conflicts → Code differences**, which filters out mere formatting and comment differences. The second change remains highlighted, though a little differently. 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.

[To be continued ...]

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 is set up and that you know the Store administrator ID and password. Information that your DBA may need to set up the database and account is provided in Appendix B, "Store Setup for DBAs". You can print out those pages and give them to your database administrator. Those instructions are repeated in the individual installation sections here.

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 according to its own algorithms.

To install Store, launch a clean image (visual.im) or the image containing your code. Then, in the Parcel Manager **Store Tools** section, select and load the Store support parcel matching 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 version 7 is supported on Windows platforms.

A preview (beta) version Store support for MS Access is available as StoreForMSAccess. Other back-ends are supported by a variety of third-parties. Back-ends provided with VisualWorks as goodies include:

- PostgreSQL
- DB2
- Interbase

Support for the PostgreSQL implementation is provided by the developer at:

http://sourceforge.net/projects/st-postgresql/

More databases are added every release.

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 Appendix C, “Creating a Custom Install Script” for instructions.

Oracle setup

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

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 (step 4).

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.

The remaining steps are performed in VisualWorks.

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 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

Note: 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 addition of 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.

The remaining steps are performed in VisualWorks.

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 users and groups" for details.

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:

http://wiki.cs.uiuc.edu/VisualWorks/Store+for+PostgreSQL+Documentation

and

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 (typically user postgres).

2 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 (as is generally recommended), the startup command may be:

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

3 Create a database table owner account for Store, by executing at the command prompt:

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

The default Store table owner account name is BERN. If you use another name, set the **Database table owner** in the **Store → Settings** before building the tables in step 5. 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 and, if you will install user/group maintenance, an administrator account. To exclude normal users from adding databases and users, use this command line:

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

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

```
#> createdb -D <dbpath> <dbname>
```

If **$PGDATA** is set, you can omit **-D <dbpath>**, and the path defaults to the value of **$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 created in step 3.

The environment string is the machine identifier and database name, in the format **myComputer_dbName** (e.g. **192.168.10.3_storedb**). The machine identifier may also be its network name. The default port number is 5432, but the port may need to be specified, especially if PostgreSQL is running on another port. For example, the identification for the VisualWorks open repository, specify: **www.cincomsmalltalk.com:5432_bern**.

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

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

Answer **Yes** to install support for user/group level access management. You can answer No here, and install user/group support later if you wish. Refer to "Setting up users and groups" in Chapter 8, "Administering Store" for additional information and instructions.

8 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.

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 Store for your database installed, 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 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: We assume that you will not be loading these from the database, and so recommend against publishing binary. If for some reason you do need to load them from the database, the AT Parcer Compiler must be pubished 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 as in step 4-5.

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" in Chapter 4.

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 namespaces 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 users and groups" below). 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: #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 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 BasicOwnershpPolicy, 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 judgement, 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. Separate namespaces can also be helpful to ensure this separation.
- Don't mix in a package inessential (like examples and tests) with essential code.

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. Bundles guarantee 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. They also provide a fixed structure of their contents, by containing only specific versions of contained packages and bundles.

Bundles provide a mechanism for assembling smaller code units into any number of larger units. While the 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.

Bundles do not allow or retain overrides between their contained packages and bundles. This is in keeping with the view that they are atomic, specifying a single, coherent collection of code definitions. Accordingly, they are useful for assembling a component, a complete unit of functionality, out of sub-components.

Using bundles to organize projects

Despite the restriction that overrides are not allowing in a bundle, it often is useful to build several different bundles that load different deployment and development configurations. This is possible because projects should never need to override definitions within the same project.

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 orders

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 a bundle doesn't support overrides between its contents. 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.

Suggestions for development dependencies

In your development environment:

- Specify development prerequisites 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 setting prerequisites for deployment, you have to 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, create a bundle (MyClientApp), and add the two packages to it. Then set the load order as:

SharedCodePkg

ClientCodePkg

But, suppose the shared code, which establishes communications protocols, is dependent upon the Net Clients HTTP support code. In the development prerequisites property for SharedCodePkg, select the HTTP parcel and move it to the **Prerequisites** list. That 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 success of your application. So, this approach respects encapsulation.

Suggestions for deployment dependencies

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. Deployment 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 the 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 1-to-1 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, only 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.

Managing package content

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 in several ways, for example:

- In the Loaded Items list (**Store → Loaded Items**), choose **Change → Add Package…**, and specify a name for the new package.
- In the System Browser, choose **Package → New Package…**, and specify a name for the new package.

The new package is added to the Loaded Items list. A new package is represented in the image, and so is saved with the image, but it is not recorded in the Change List. 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 **(none)** rather than to a named package. 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 VisualWorks main 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.

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 it's prerequisites are loaded and, if not, loads them. Packages and bundle prerequisites can be specified for either development or deployment.

Deployment prerequisites are parcels, and are turned into parcel prerequisites when the package or bundle is published as a parcel.

Development prerequisites may be either parcels, packages or bundles, and are used when loading code from the Store repository. Often, development prerequisites are a superset of package/bundle correlate of the deployment prerequisites, including additional items such as development tools and tests.

To specify prerequisites:

1. Select the bundle in the System Browser package list and select either the **Development Prerequisites** or **Deployment Prerequisites** page.
2. (Optional) Click **Compute Prereqs** (in the **Deployment Prerequisites** property) to have the system make an initial suggestion of prerequisites.
3. (Optional) Click **Copy Deployment** (in the **Development Prerequisites** property) to copy deployment prerequisites into the development prerequisites list.
4. Select a parcel, package, or bundle (only parcels are listed for deployment prerequisites), and click >> to move it to the **Prerequisites** list. Code units must be loaded to appear in the list of available units.
5. To remove items from the prerequisite list, select the item and click <<.
6. When done adjusting prerequisites, click **Apply**.

For development prerequisites, you can specify that the prerequisite is a package, a bundle, a parcel, or any of those. You can, using this item, select a loaded package or bundle, select #parcel, and the item will be entered as a prerequisite parcel with the same name. Make sure that the parcel is created before you try to load the package or bundle again.

You can also specify a version string. Only the version number should be listed, excluding the user name. For example, to load only version 1.1 of a package, enter “1.1” in the **Version:** field, not “1.1,bruce”. A parcel also may have a version string, if the string was assigned to the parcel before publishing.

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 should 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.

Specify prerequisite version

A prerequisite version string can be specified in the prerequisite property, and is adequate if a specific version number is required. For more general version control, such as to allow a range of versions, create a three-argument block in the **Prerequisite Version Selection Action** property.

```
[: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 ]
```

Package load and unload actions

Action blocks can be set to be evaluated at several stages of loading and unloading a package: 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.

Initializing and finalizing packages

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

How a package is loaded

The load sequence of a package is as follows:

1. The package's pre-load action is performed, if defined.
2. The objects in the package are installed into the system.
3. Every class defined in the package is sent the postLoad: message with the package as argument.
4. The package's post-load action, if defined, is executed.

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.

Managing bundle content

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 namespace 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 open the editor:

1. Select the bundle in the System Browser package list
2. Select **Package → Edit Bundle Specifications…**
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**.

Specify prerequisites

Prerequisites are parcels or packages that must be loaded before the bundle is loaded. Prerequisites can be specified for development and for deployment. Often, development prerequisites are a superset of deployment prerequisites, including additional items such as development tools and tests.

Deployment prerequisites are converted to parcel prerequisites if a parcel is published from the bundle. Otherwise they are ignored.

To specify prerequisites:

1. Select the bundle in the System Browser package list and select the **Properties** tab.
2. Select either the **Development Prerequisites** or **Deployment Prerequisites** property.
3. (Optional) Click **Compute Prereqs** to have the system make an initial guess at the necessary prerequisites.
4. Select a parcel or package (packages are only available for development prerequisites) and click **>>** to move it to the **Prerequisites** list.

   The item must be loaded in the system to use this method, but it usually is if you have been doing development. To add a prerequisite that is not currently loaded, enter its name in the entry field below the

parcels/packages list and click **Add**. You may also specify a version number for the parcel/package.

5 To remove items from the prerequisite list, select the item and clicking <<.

6 When done adjusting prerequisites, click **Apply**.

Suppress warnings

A package's warning suppression action must be a block taking one argument, the name of a prerequisite parcel being loaded. The block should answer true if any warnings caused by loading the prerequisite should be suppressed.

The warning suppression block is run before any of the parcel's code is loaded. Consequently it cannot and should not mention any code in the parcel.

Specify prerequisite version

A bundle prerequisite version can be specified as a code block on the **Prerequisite Version Selection Value** page of the bundle's properties.

The block is a block taking three arguments, the name of a prerequisite being loaded, its version string, and the version string defined in the package's list of prerequisites.

The block should answer true if the prerequisite's version is acceptable, and should be loaded. Otherwise the loader will continue to search for another parcel of the same name with a different version.

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.

Adding files to a bundle is currently done by evaluating expressions such as:

```
bundle := Store.Registry bundleNamed: 'Foo'.
bundle addFile: 'foo.txt'
```

Similarly, send a removeFile: message to remove a file from the bundle.

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

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.

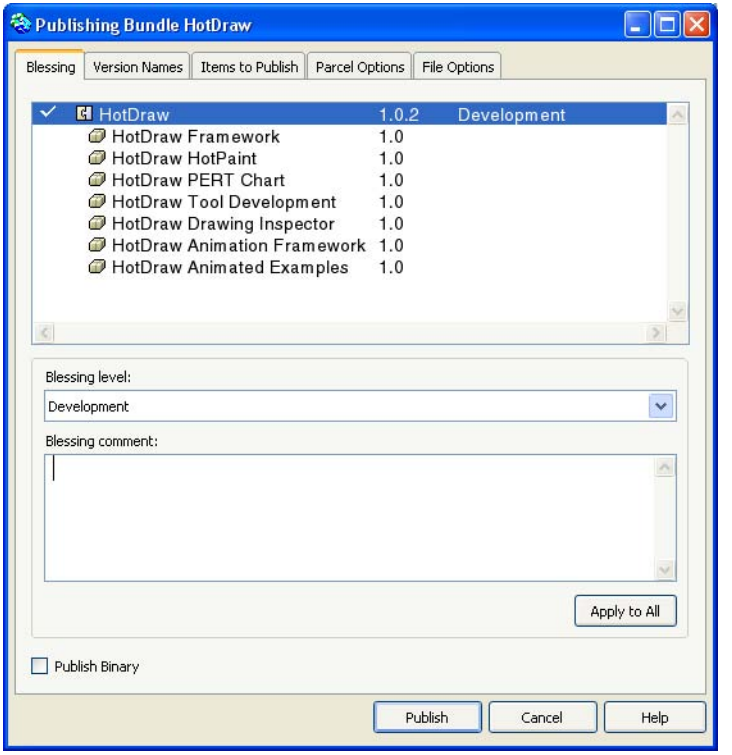
Basic publishing

Publishing is a common, daily activity for team members, and so is described in greater detail later (see Chapter 5, “Maintaining Your Store Environment”). Here we give a brief account of publishing 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 level for each package or bundle,
- A blessing comment.
- 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 (already added to the bundle, see Including external files) 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.

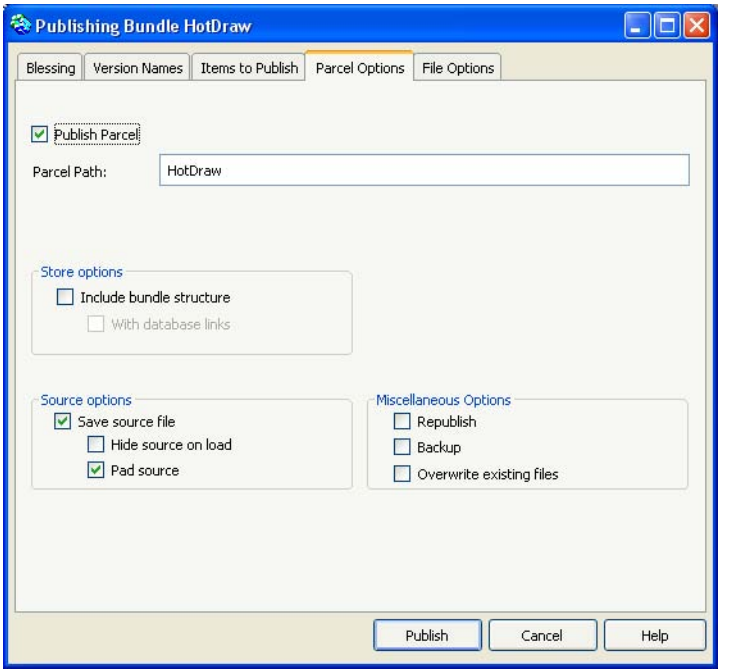
Due to enhancements in the parcel loader, you can both load binary code initially, and load it for updates. This greatly speeds up the load process.

However, publishing binary uses a lot of disk space, because each publish is the whole package rather than just the deltas.

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 version 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:
   - Check the **Include bundle structure** checkbox to save structure information in the parcel, so the structure can be recovered if the parcel is loaded into an image that has Store installed.
   - If you save the structure, you can also check **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.

Overriding definitions

There is a general sense of “override,” often used in OOP literature, in which a subclass that reimplements 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 namespace 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 namespace 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.

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.

Among other effects, you can lose version history information, unless the change is made by the Store administrator in the database. For this there is an administrative utility: in the Visual Launcher, select **Store → Administration → Rename Package in Database**, or **Rename Bundle in Database**.

Even if you are not preserving version histories, you do need to coordinate this change in all containing bundles. 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 latest bundles.

Reorganizing namespaces

Moving definitions to a new namespace 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 namespace, 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 namespace, 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 namespace 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 namespace changes.

3 Move the classes in the framework code into their new namespaces.

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:

1. 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 namespaces.
2. 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 namespace).

There is no work around for situation 1. Instead, reload as in situation 2. In situation 2, 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 namespace. The use Replay-All to load the code.

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

5

Maintaining Your Store Environment

Overview

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.

An implication of this assumption is 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.

Chapter 2, “Beginning to Use Store” runs through installing Store this sort of environment, and demonstrates working in it. This chapter provides a more comprehensive view of the same environment.

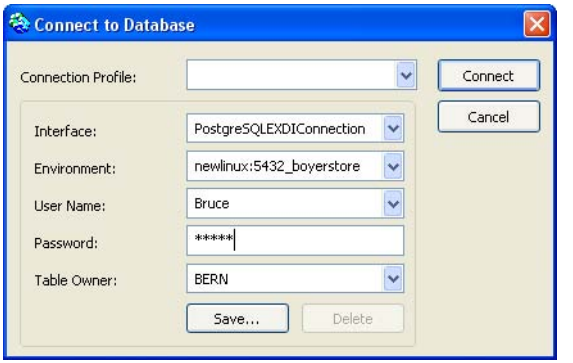
Working connected and disconnected

Unlike some source control systems, you do not have to 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 → Connected to DB** in the VisualWorks main window.



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**. 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 holds multiple database connection profiles in the image. Unfortunately, if you update to a new version of VisualWorks, or start over from a clean image, you have to re-enter your database connection information.

Store does not current provide a way to write your connection profiles out to a file and reload them, to simplify this procedure. Until such a facility is added, you can use workspace scripts. Here are two samples:

```
"Load profiles, after loading Store"
| f b p |
f := 'profiles.bos' asFilename readStream.
b := BinaryObjectStorage onOld: f.
p := b next.
b close.
p do: [:each | Store.ConnectionDialog new addOrReplaceProfile: each]

"Write out profiles"
| f b |
f := 'profiles.bos' asFilename writeStream.
b := BinaryObjectStorage onNew: f.
b nextPut: Store.ConnectionDialog profiles.
b close.
```

The write script simply writes out the Profile information to a BOSS file. The load script loads the Profile information back into the shared variable.

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 database, which is available for code updates and user contributions (visit the Cincom Smalltalk Developer's Wiki, http://www.cincomsmalltalk.com:8080/CincomSmalltalkWiki, for more information.)

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 below (see "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. (See “Using a local database”.)

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 packages in your image to packages 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.

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 **File → 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 **File → 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.

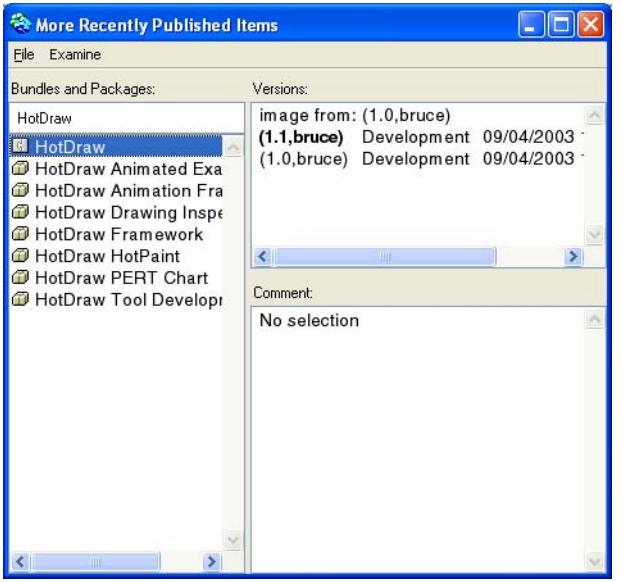
After loading, your image contains a new working version of the package. If the package is a component of a bundle that is loaded in your image, your working version of the bundle is marked as modified.

Updating to new versions

Browsing published bundles and packages

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 More Recently Published Items browser displays the names of published bundles and packages. Bundles are listed first, followed by packages, 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 * for pattern matching.

The Versions pane lists the versions of the selected package or bundle that are in the database. You can open a dedicated versions browser by selecting **Examine → List Versions**, or a graphical representation by selecting **Examine → Graph Versions**.

Browsing the Definitions in a Published Version

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

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 are used in the System Browser to indicate various states of code with respect to packages.

Bold	Marks any item (class, name space, or method) that is defined in the selected package.

In the package list, packages have characters following their name indicating the package's state:

*	The package has been modified.
+	The package extends classes in other packages, possibly overriding some definitions.
-	The package has definitions that are overridden by another package.

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.

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.

Publishing packages

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.

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 **Examine → Compare with Parent**, or
- Select the package in a Versions List, and choose **Examine → 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 → inspect 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" in Chapter 7, "Integrating Versions" for more information.

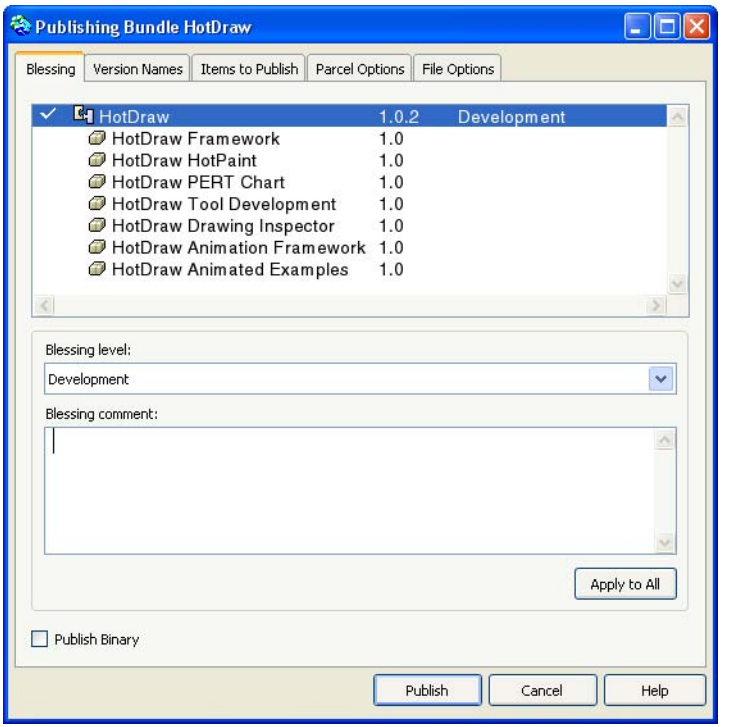
Publishing a bundle

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 and bundles will be published in binary format.
- A blessing version for each package or bundle (click the **Set Global Blessing Level and Comment** to give all the same blessing level.
- A blessing comment, giving additional information, for each package or bundle (click the **Set Global Blessing Level and Comment** to give all the same comment)

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

To set all packages to the same version number, select a package, set its version number, then click **Set Global Version**.

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, see "Including external files" on page 4-15) 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.

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.

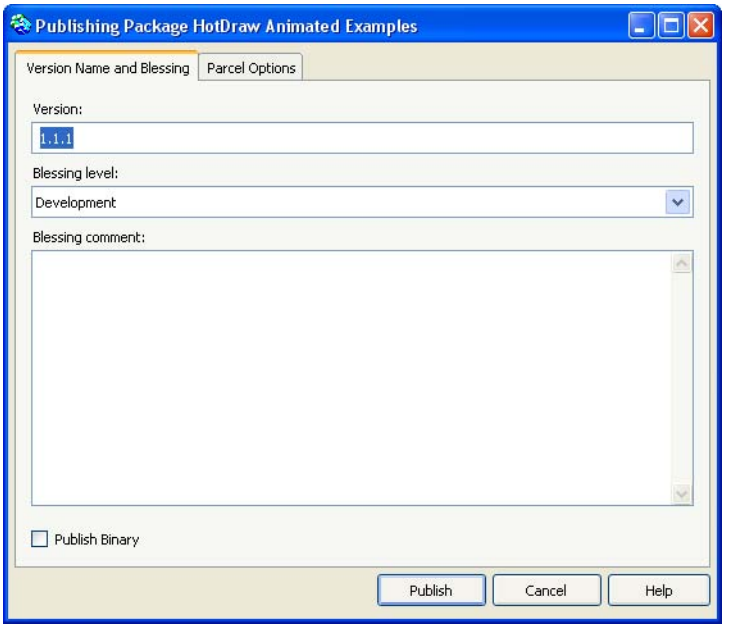
The Package Publisher dialog is an abbreviated version of the Bundle Publisher.

To publish a package, you:

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

2 Choose **Package → Publish…** (**File → Publish…** in the Loaded Items list).

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



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

- The version number for the package, in the **Version** field
- A blessing version 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.

Publishing changes

You can now publish code fragments, individual or sets of fixes, to a package, based on a named change set. The version can be published as a branch, which is then available for merging later.

To publish a fragment:

1. Open the Change Set list (**Changes → Change Sets** in the Visual Launcher).
2. Create and select a named change set to contain the fix.
3. Make your fix.
4. Return to the Change Set list and, on the **ChangeSet** or <Operate> menu, select **Publish as Fragment**.

   The usual publishing dialog opens, and you can specify the usual parameters.
5. Set the publishing parameters and click **OK**.

For example, suppose you are doing development on a large bundle with a number of sub-bundles and packages. You then discover a bug in the same bundle, which you decide to fix. You then create a new Change Set and fix the bug. Now those changes are isolated to that Change Set. You can now publish just those changes, leaving the rest of your work unpublished.

Without this option, you would have to go to another image, or get rid of your current changes, fix the bug, and publish the package or bundle with just those changes. Being able to isolate Change Sets and publish them separately allows you to make fixes like this without interrupting your normal work session.

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 Store, 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, Store 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 Store, 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.
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: NEWBlessingPolicy 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 **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 Differences Browser opens.

The Differences Browser lists the class and method definitions that differ between two versions. The definitions are grouped hierarchically by class and protocol.

The Differences Browser displays its information in pairs of vertically-stacked views, where the upper member of each pair displays information from one version, and the lower member displays information from the other.

You can switch the view between showing differences in code (**View → Show code differences**) and differences in source (**View → Show source differences**). Source differences show differences in the way the source is written, such as formatting differences, whereas code differences show actual differences in the code.

Package views

At the top of the Differences Browser, a pair of package views displays the names and version strings of the compared packages. The version string of the working version ends in a plus sign (+) if the version contains unpublished changes. The version string ends in an equals sign (=) if the working version is identical to its published parent version.

Class views

Below the package views, a pair of class views lists the differing classes in each of the compared versions:

- Classes listed in **bold** are classes whose class definitions or class comments differ. Each class name is followed by a string indicating when the definition was published and by whom.
- Classes listed in the normal font are classes whose methods differ.
- Classes listed in *italic* are classes whose extensions differ. That is, these classes contain method definitions that are part of the package, although the classes themselves are not; the differences exist in these methods.

Protocol and method views

The protocol views list the protocols in each version that contain differing methods. When you select a protocol for a version, the corresponding method view displays the names and versions strings of the method definitions that differ between the compared package versions.

Text views

At the bottom of the Differences Browser is a pair of text views, where you inspect the contents of a selected class or method definition. If the selected definition exists in both package versions, the text views provide a line-by-line comparison, emphasizing the lines that contain differences by displaying them in **bold**.

If nothing is selected in any class, protocol, or method views, then the text view displays the package comment, initialization string, and finalization string.

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. Choose **Examine → List Versions**. This brings up a Versions list that shows the item's versions.

3 In the Versions list, select the version whose blessing level you want to change.

4 Choose **File → Set Blessing Level**, to open the Blessing Level dialog.

5 Select the new blessing level, enter a comment, and click **Accept**.

7

Integrating Versions

Team members can synchronize their work by *merging* variant versions of a package into a single consistent version. At integration time, appropriate versions are merged for each package in the project to produce a new project baseline.

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 Differences Browser opens.

The Differences Browser lists the class and method definitions that differ between two versions. The definitions are grouped hierarchically by class and protocol.

The Differences Browser displays its information in pairs of vertically-stacked views, where the upper member of each pair displays information from one version, and the lower member displays information from the other.

Package views

At the top of the Differences Browser, a pair of package views displays the names and version strings of the compared packages. The version string of the working version ends in a plus sign (+) if the version contains unpublished changes. The version string ends in an equals sign (=) if the working version is identical to its published parent version.

Class views

Below the package views, a pair of class views lists the differing classes in each of the compared versions:

- Classes listed in **bold** are classes whose class definitions or class comments differ. Each class name is followed by a string indicating when the definition was published and by whom.
- Classes listed in the normal font are classes whose methods differ.
- Classes listed in *italic* are classes whose extensions differ. That is, these classes contain method definitions that are part of the package, although the classes themselves are not; the differences exist in these methods.

Protocol and method views

The protocol views list the protocols in each version that contain differing methods. When you select a protocol for a version, the corresponding method view displays the names and versions strings of the method definitions that differ between the compared package versions.

Text views

At the bottom of the Differences Browser is a pair of text views, where you inspect the contents of a selected class or method definition. If the selected definition exists in both package versions, the text views provide a line-by-line comparison, emphasizing the lines that contain differences by displaying them in **bold**.

If nothing is selected in any class, protocol, or method views, then the text view displays the package comment, initialization string, and finalization string.

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. Choose **Store → Browse Versions**. This brings up a Versions list that shows the item's versions.
3. In the Versions list, select the version whose blessing level you want to change.
4. Choose **Package → Reset Blessing Level**, to open the Blessing Level dialog.
5. Select the new blessing level, enter a comment, and click **OK**.

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.

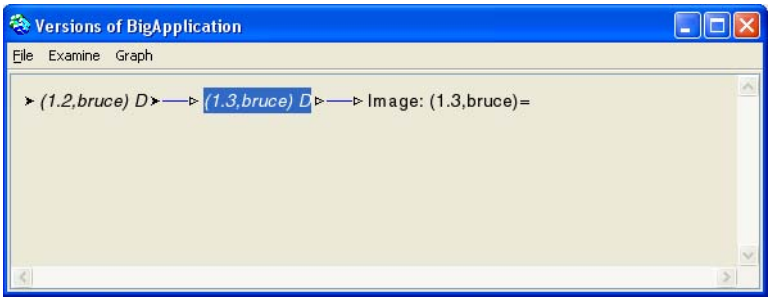
The Store Merge Tool assists in this integration process. The Merge Tool examines the parent-child relationships among published versions of a package, identifying the modifications that differentiate two or more related versions from their latest common ancestor. It then combines user-selected modifications into a new working version of a designated base version.

Relationships among versions

In general, 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.

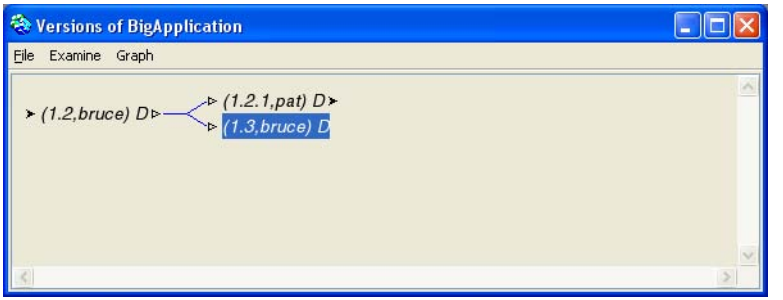
A line of development starts when you load a working version into your image from a published version of a package (say, version 1.1). You make modifications to your working version. It has no version number at this point, but the Version Browser will show that it is from a particular version.

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's no need for integration in a linear version tree.

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:



Clearly, this can become arbitrarily complex. At some point, these divergent lines of development will need to be brought back together, or integrated.

Conflicting and nonconflicting modifications

Each published version will contain 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.

Merging two versions of a package

You may often only need to merge two versions of a single package, for example, if you and another team member have each modified the same package. To do this:

1. Both team members must publish their working versions.

   The Merge Tool only examines published versions of packages. Unpublished modifications in your working version may be overwritten in the merge process.

2. Load the version of the package that is to become the main stream version.

   This procedure merges one version into the currently loaded version.

3. Open the Version List for the package whose versions are to be merged (**Store → Browse Versions** in the Loaded Items list).

4. In the Version List, select the version to be merged into your image and choose **Package → Merge Into Image**. This starts the Merge Tool.

5 In the Merge Tool, identify which versions of the modifications to include in the new version.

See "Resolving conflicts" and "Excluding nonconflicting modifications" below for details.

6 Choose **Resolution → Apply Resolution** to apply all resolved modifications to the current image. To apply all resolutions in bulk, select **Resolution → Apply All Resolutions**.

7 Choose **Packages → Publish Packages…** to publish the merged version of the package.

Integrating a set of packages

To integrate multiple versions:

1 Publish all versions to be included in the integration.

The Merge Tool only examines published versions of packages. Unpublished modifications in your working version may be overwritten in the merge process.

2 Set the blessing level of all versions to be merged to **Integration-Ready**.

The Merge Tool will display only packages marked Integration-Ready for merging.

3 Start the integration base image.

This may be a special integration image, such as the image created by the last integration, or an ordinary working image which contains base versions of each package to be merged.

4 Choose **Store → Merge Tool** from the VisualWorks Launcher window to start the Merge Tool.

5 In the Merge Tool, choose **Package → Select Packages…**.

This opens a dialog that displays all the packages in the database that have at least one Integration-Ready version.

6 Select all the packages that you want to integrate.

The Merge Tool then displays modifications for all the Integration-Ready versions of the packages you selected.

7 In the Merge Tool, identify which versions of the modifications to include in the new version.

See "Resolving conflicts" and "Excluding nonconflicting modifications" below for details.

8 Choose **Resolution → Apply Resolved** to apply all resolved modifications to the current image.

9 Choose **Packages → Publish Packages…** to publish the merged version of the package.

10 Save the image, if desired, for use as the next integration image.

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**.

Creating the merged version

After you have chosen the desired modifications, you apply them to the working version of the base version, and publish the results as the new merged version:

1. Choose **Resolution → Apply Resolution**.

   Every modification marked as resolved (both conflicting and non-conflicting) is applied to the working version in your image, adding, replacing, or removing existing definitions and comments as necessary.

   In the modification view, a solid square icon indicates the modifications that have been applied.

2. Choose **Packages → Publish Packages…** to publish the merged version of the package. In the resulting dialog, fill in the version string and the blessing level for the new version.

8

Administering Store

User Administration

Users who will publish to and load code from the Store 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, as described below in “Setting up users and groups”.

Adding Store users

There are two general classes of Store user: the Store table owner, that you created for installing the Store tables into the database, and “normal” Store users. Both kinds of users are created using host database administratiom facilities.

Table owner accounts

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
	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
	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.

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

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 be different, but equivalent.

Setting up 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.

Installing user/group management

Store 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:

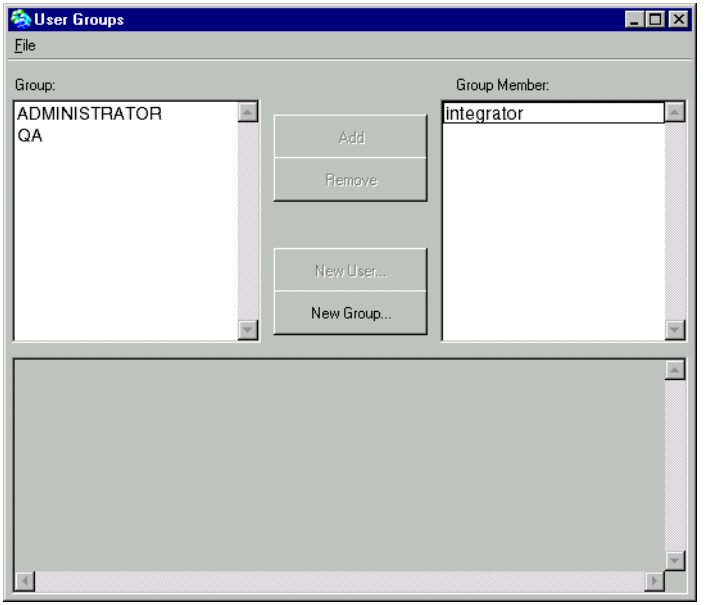
```
Store.Policies blessingPolicy: OwnerBlessingPolicy new.
Store.Policies ownershipPolicy: OwnerOwnershipPolicy new.
```

You may substitute your own policy methods, overriding these methods. See "Configuring Store policies" for information.

5 Create Store users (as described below in "Configuring user groups") 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**.

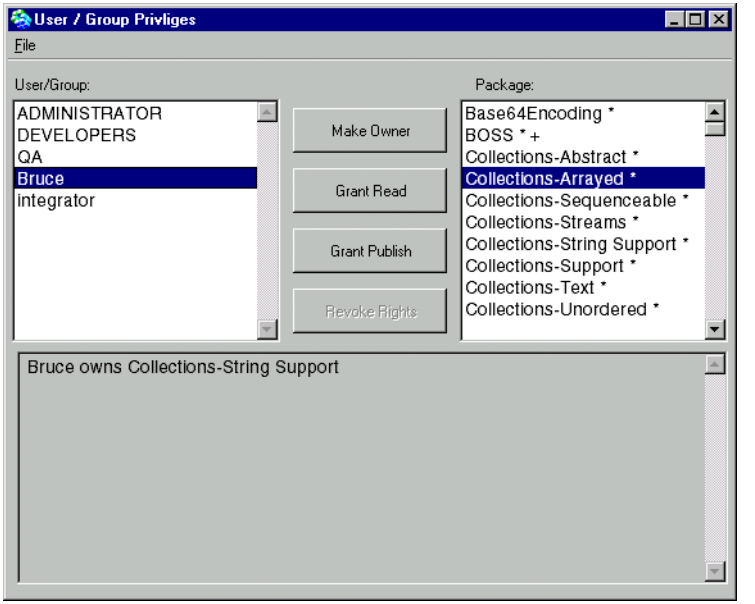
To remove a user from a group, select the user and the group and click **Remove**.

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.

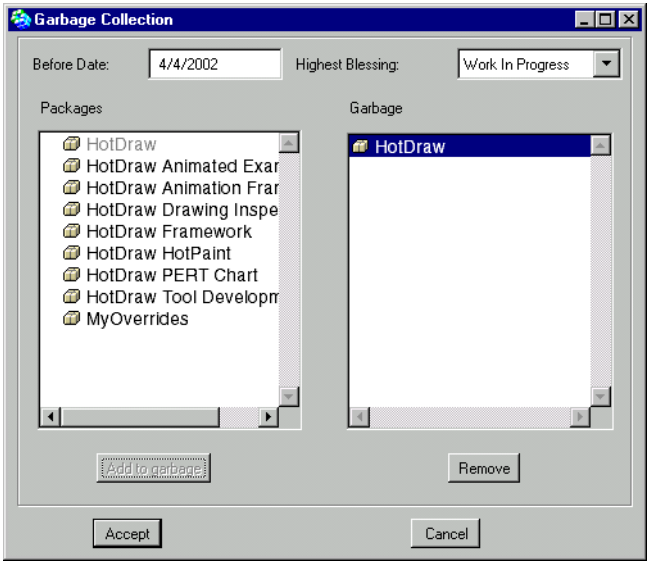
During garbage collection, Store identifies definitions required by versions that are not being removed, and assigns them to versions that are remaining in the database. In this way, although versions only store “deltas,” all required definitions are preserved.

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 cannot be garbage collected. Doing so would cause the image to be inconsistent.

Garbage collection is a very slow 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.

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 two conditions for garbage collection:

- **Before Date:** specifies that only package versions published before this date will be garbage collected.
- **Highest Blessing:** specifies that only package versions at this blessing level or lower will be garbage collected.

Once you have selected packages and collection criteria, click **Accept** to proceed.

The Garbage Collector then scans all the definitions in those packages for methods and other objects that are not referenced in any remaining packages, and removes them from the database. If a removal would invalidate a bundle, a prompter verifies the removal before proceeding.

To garbage college (remove) a bundle, garbage collect all of its contents. The bundle is removed when it has no contents.

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.

A

Porting from ENVY/Developer

A large number of VisualWorks users have used ENVY/Developer as their source code management system for many years. Since ENVY is no longer being developed and supported for VisualWorks, it is essential to move code from the old ENVY database into Store, to continue taking advantage of advancements in VisualWorks.

ENVY and Store are based on different component models. ENVY employs a hierarchical model of four different types of components: configuration maps, applications, subapplications, classes and class extensions. Store employs a hierarchical model consisting of only two types of components: bundles and packages.

There are essentially two approaches to moving code from and ENVY environment into a Store environment:

- use the Store Bridge
- file-out or parcel-out code, and load into a Store image

The Store Bridge provides the advantages that it:

- automatically transfers application structure
- is designed specifically for porting from ENVY 5i to Store
- supports porting ENVY 4.0 to VisualWorks 3.x

However, you cannot use the Store Bridge for:

- porting system changes
- porting from VisualWorks 2.5.x

Since the Bridge is an automated tool, you are should not use the bridge if you need to, or think you should observe methods and classes as they are being loaded.

Note that ENVY-based applications often use ENVY specific methods. You must treat these methods like system changes.

Conceptual porting

One major stumbling block for ENVY users moving to Store has been in mapping the ENVY conceptual model to the Store conceptual model.

ENVY structures applications as applications and subapplications, and imposes a number of restrictions on subapplications. A “snapshot” of an application consists of an archived set of specific versions of applications and subapplications.

It is tempting, at first look, to think of packages as analogous to ENVY applications, and bundles as analogous to configuration maps. There is enough to the analogies to lead to deeper confusion, but it is not quite right and will lead to frustration before long. These analogies are tempting because packages contain code while bundles contain only packages and other bundles. The analogy breaks down because thinking in terms of the entities that actually hold the code is not as important in Store as it is in ENVY, and bundles do not behave sufficiently like configuration maps.

It would be nearer to think of packages as analogous to applications, when there are no subapplications, and bundles analogous to applications when there are subapplications. In ENVY, when an application became too complex, the practice was to pull out parts into subapplications, but the application continues holding some of the code. In Store, the factoring goes differently. You might start developing in packages, as with applications. But then, when the package becomes more complex and you need to break it up, you divide all the code among simpler packages and unite them in a bundle; you add a higher-level structure rather than just split out smaller substructures.

While it is true that the code is actually in the package, rather than the bundle, bundles behave as if they contain the code. For example, if two packages in a bundle both define the same method, so that one would, if the packages were loaded individually, override the other, if they are bundled they do not behave this way. Instead, the code in the package loaded later would simply overwrite the earlier.

Bundles are partially analogous to ENVY configuration maps, but not near enough. A configuration should be a collection of specific versions of individual components. A bundle, however, itself behaves as one of those components. As described elsewhere in this document, there are

contexts in which it is safe to treat bundles as configurations of components (packages), but there are limitations. A true configuration entity will be introduced for Store in a future release.

Using file-outs and parcels

ENVY environments support both the file-out and parceling mechanisms in VisualWorks, so provide a straight-forward approach to porting code from an ENVY environment. You can use this approach either to porting your entire application, or for porting just that code, such as system overrides, that cannot be ported using the Store Bridge.

The basic procedure is to:

1 Either:

- file out code from your ENVY environment into **`.st`** files, or
- create parcels in your ENVY environment, move your code into them, and save the parcels.

2 Load the file-outs or parcels into a Store environment, as described in Chapter 10, "Maintaining a Store Code Base."

Because there is nothing in this approach that automatically preserves the structure of your ENVY application, you should create the file-outs or parcels in such a way that will simplify recovering that structure in Store.

For the fileout approach, you need to set Fileout Format to Standard, and evaluate:

```
System genericFormat: true.
```

A simple approach that will serve as a first approximation, at least, is:

1 In the ENVY environment, save each application and subapplication into its own file-out.

2 In the Store environment, create a new package for each file-out.

3 File-in each file-out into the package you created for it.

You can do the equivalent with parcels, though parcels can involve a little more work to create in the first place.

You cannot represent an ENVY Configuration Map with file-outs or parcels. The nearest correlation to a configuration map in Store is currently the bundle, which you can create later, after you have moved your code into Store packages.

Once your code has been moved into Store packages, you can begin building bundles, defining prerequisites, and imposing other structure on the components.

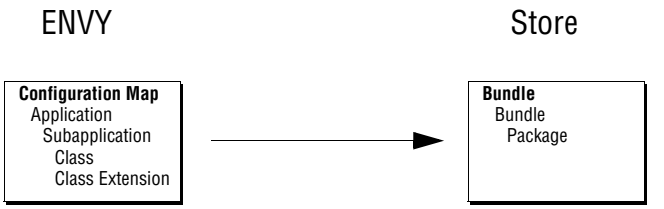
Initialization code also is not preserved, and must be recreated for Store packages.

Store Bridge

The Store Bridge greatly simplifies the task of migrating applications between ENVY/*Developer* and Store. The Bridge is a VisualWorks add-on component that assists in translating between the different component and versioning models used by ENVY and Store.

During the conversion, the Bridge provides two principle functions. First, it assists by converting the organization of a project, preserving the hierarchical composition of its components; and second, it provides precise control over how the version history of a project is migrated.

The Store Bridge manages the conversion between these two models while preserving the “shape” of a project as it would be represented in each environment.



Thus, when migrating from ENVY to Store, a configuration map is converted into a bundle, while applications and subapplications are converted into either bundles or packages, depending upon the nesting of subapplications. Since subapplications can be nested in ENVY, the Bridge uses bundles (which can also be nested) to represent the structure of a configuration map.

To simplify the conversion process, the Bridge only translates between the highest-level components in each environment, i.e., ENVY configuration maps and Store bundles. For example, when migrating a project from ENVY to Store, it must be exported as a configuration map in order to translate the entire structure of the project.

The Bridge makes use of VisualWorks parcels when transporting projects. The component structure of the project is preserved using special properties in the parcel. Parcels provide the flexibility of a shared medium for transporting components between the two environments.

The following pages describe the process of exporting a project from ENVY, importing it into Store, and finally, publishing the project in the Store repository.

Compatibility

The Store Bridge can be used to port project code from both ENVY/*Developer* R4.0 (the ENVY version for VisualWorks 3.0) and R5i.1 (the ENVY version for VisualWorks 5i.x) to VisualWorks 5i.x.

Environment	Migration Options
ENVY/*Developer* R4.0	Export configuration maps with version history
ENVY/*Developer* R5i.2	Export configuration maps with version history
VisualWorks 5i.x	Import configuration maps with version history

When moving projects between ENVY R4.0 and VisualWorks 3.x, there will be general porting issues relating to the use of the new language features added in VisualWorks 5i and subsequent releases. For example, in VisualWorks 5i, global variables, class variables, and pool variables are replaced by shared variables, and the monolithic Smalltalk pool is broken up into name spaces. For details on such changes, refer to the *Application Developer's Guide*.

Installing the bridge

The Store Bridge is delivered as two code components. The first is an ENVY DAT file (**`Bridge.dat`**), while the second is a VisualWorks 5i parcel (**`Store-Bridge.pcl`**). The Store Bridge is installed by loading the DAT file into the ENVY environment, and then loading the parcel into the Store environment. Both of these components are located in the **`store`** subdirectory of the standard release of VisualWorks.

Installing the Bridge in the ENVY environment

To install the Store Bridge, you must first import a DAT file into a running ENVY/Developer image:

1 Copy the **`Bridge.dat`** file to where your ENVY library resides (see your ENVY library supervisor if you need assistance).

2 Open a Configuration Maps browser (by selecting **ENVY → Browse Configuration Maps** in the Launcher window).

3 In the left-hand pane of the Configuration Maps browser, select **import…** from the **<Operate>** menu.

4 Enter **`Bridge.dat`** (provide the path, if necessary).

5 Select the following configuration map and version to import into the current library:

VisualWorks Store Bridge

To actually choose the configuration map, click on the **>>** button and then click on **OK**.

The VisualWorks Store Bridge should now be visible in the **Names** list of the Configuration Maps browser.

6 To load the Store Bridge, select the name of the configuration map, the version number in the **Editions and Versions** view (**1.0**), and the name of the application in the right-hand view (**StoreBridge 0.21**). Then, select **load** from the **<Operate>** menu in the **Applications** view.

The Store Bridge may now be used to export configuration maps.

Before proceeding, you should save your ENVY/*Developer* image.

Installing the Bridge in the Store environment

To complete the installation of the Store Bridge, launch a VisualWorks team development image (VisualWorks with Store loaded). Next, load the Store Bridge parcel into the running image:

- In the Launcher window, select **Tools → Load Parcel Named…** and enter **StoreBridge** at the prompt.

Once the parcel has loaded, the menu item **Convert Configuration Map…** should appear on the Launcher window's **Store** menu.

If you wish to use the Store Bridge a number of times, you may want to save the image now.

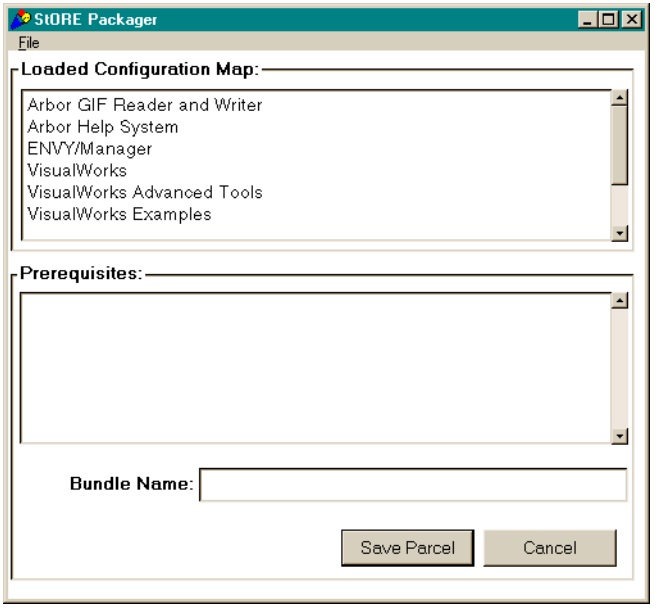
Exporting an ENVY configuration map

To export a Configuration map from an ENVY/*Developer* image, perform the following steps:

1 With Store Bridge loaded, use the ENVY Configuration Maps browser to load the version of the configuration map that you want to export.

2 Open the Store Packager (select **Tools → Open StorePackager...** in the Launcher window).

The Store Packager Tool opens as shown below:



3 A list of loaded configuration maps is displayed in the upper part of the tool. Pick the one that you want to export. A list of prerequisites for the selected configuration map will appear.

4 To specify a different name for the exported bundle, change **Bundle Name**.

5 To export the configuration map, click on **Save Parcel**. A parcel save dialog will appear, prompting you for various options (for details on these options, see the *VisualWorks Application Developer's Guide*).

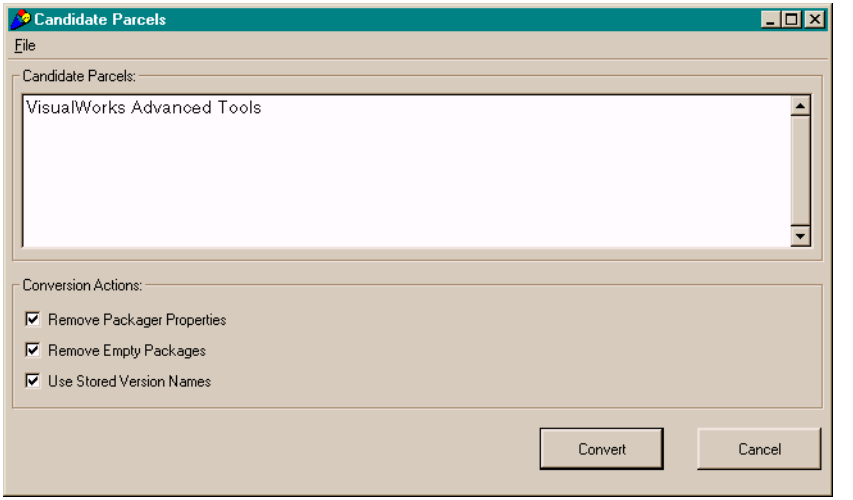
The configuration map is saved as a parcel file.

Importing and publishing an ENVY configuration map

To import an ENVY configuration map into a running Store image:

1. In the Launcher window, select **Tools → Load Parcel Named…** and enter the name of the parcel that contains the exported configuration map.
2. Once the parcel has loaded, select **Store → Convert Configuration Map…** in the Launcher window.

   The Conversion Tool opens as shown below:



   The list of candidate parcels should display the one that you just loaded in step 1.

3. Select the parcel that you want to import.
4. Select the appropriate conversion options.

   To remove the special package properties added by the Store packager, select **Remove Packager Properties** when converting to a bundle. By default, this should be selected.

   To remove the empty package that remains after the parcel has been converted to a bundle, select **Remove Empty Packages**. By default, this should be selected.

   To preserve the same version identifiers used in the ENVY environment, select **Use Store Version Names**. This must be selected if

you plan to use **Publish Former Configuration Map** in the Package Browser.

When the conversion is complete, the configuration map appears as a new bundle. The original structure of the ENVY configuration map is accurately represented by the shape of the Store bundle.

You may use the Store Loaded Items List to examine the new bundle (select **Store → Loaded Items** in the Launcher window).

Publishing a converted bundle to the Store repository

Before publishing the converted bundle to the Store repository, you should first reconcile it to check if a different version has already been published.

Reconciling a bundle with the repository assigns the version already published as the parent of the version in the local image. This allows Store to establish the appropriate relations and version numbers when you publish.

If another version has previously been published, the reconcile function will allow you to choose the version that you want to keep in the repository, and will establish a proper relation with the parent version.

1. To reconcile a package, first open a Package Browser (select **Store → Browse Packages** in the Launcher window).
2. Highlight the package in the upper-left-hand view of the Package Browser, and select **Reconcile with Database** from the **<Operate>** menu.

   Note: You may skip this step if you know the bundle and all of its elements have never been previously published; however, to establish proper version history it is best to check by reconciling first.

3. To publish the package in the repository, highlight the package in the upper-left-hand view of the Package Browser, and select **Publish…** from the **<Operate>** menu.

   The Store Publishing dialog window will open, allowing you to set the version identifier, blessing level, and blessing comment.

   You may also preserve the ENVY version identifier when publishing the converted project by selecting **Publish Former Configuration Map**... from the **<Operate>** menu.

Note: To preserve the ENVY version identifiers, you must have selected **Use Store Version Names** in Conversion Tool. See: "Importing and publishing an ENVY configuration map", above.

Migrating complete version history

The Store Bridge enables the migration of the complete version history of a project. A series of project versions may be imported and published in the code repository of the target environment.

This version history can only be exported from the ENVY environment and imported into Store. Each version of the project must be transferred as a separate parcel.

To migrate a series of versions from ENVY to Store:

1. Launch the ENVY image and ensure that the Store packager has been loaded.
2. Beginning with the oldest version you wish to migrate, follow the steps described above for "Exporting an ENVY configuration map."
3. Repeat step 2 for each version that you wish to migrate, creating a different parcel for each version.
4. Exit the ENVY environment and start the Store image with the Bridge parcel loaded.
5. Beginning with the oldest version you wish to migrate, follow the steps described above for "Importing and publishing an ENVY configuration map."

   Before loading each subsequent version of the project, it is necessary to unload the currently loaded version. This can be accomplished either by unloading the bundle (select the bundle in the Package Browser, and then choose **Unload...** from the **<Operate>** menu in the upper-left-hand view), or by simply quitting and restarting the image.
6. Import the next version of the project, but when publishing it make sure to reconcile it, selecting the previously published version as the parent (see "Publishing a converted bundle to the Store repository,"above, for details).

When performing a reconcile, Store will establish a proper relation with the parent version, only publishing the deltas between the two subsequent versions.

7 Repeat step 6 for each version that you wish to migrate, publishing each with a distinct version number.

Known limitations

The Store Bridge works with ENVY configuration maps and the applications and subapplications contained within. ENVY provides native support for exporting individual applications as parcels, but any subapplications contained within the applications will not be exported.

In porting from ENVY 4.0, some information that is tagged on methods in the ENVY environment is not exported, in particular “user fields.”

B

Store Setup for DBAs

Store can use several database back-ends for code storage.

Currently, we support:

- Any Oracle 7 or later database, except Oracle Lite which is not supported.
- SQL Server version 7 is supported on Windows platforms.

In addition, third-party supported backends are available for

- PostgreSQL
- DB2
- InterBase

There is generally little setup required for the database backend itself, beyond having the database itself installed and a user account defined to be used by the Store administrator. The instructions provided in this appendix summarize only the steps that may need to be performed by the database administrator, prior to the Store administrator's installing the database tables.

For full Store installation instructions, see Chapter 3, "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 Store data files.

Set Up SQL Server

Note: 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://wiki.cs.uiuc.edu/VisualWorks/Store+for+PostgreSQL+Documentation

and

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 VisualWorks 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://www.ibphoenix.com

Now you can:

- create database
- add new user account;
- test of connection;

C

Creating a Custom Install Script

You can create a custom installation script for special purposes. The script allows you to change the directory path and the tablespace names, and customize table access rights. The resulting script can be executed from within VisualWorks, or saved as a file and executed as an SQL script by a database administrator.

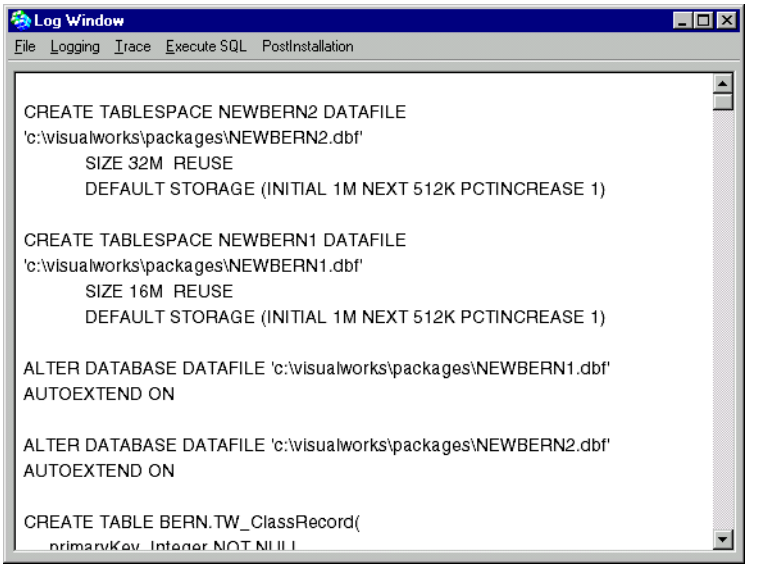
This script option is particularly important for tightly controlled database environments, in which the database administrator carefully controls how tables are created and access is granted. The script, which contains SQL, can be submitted for review, modification, and execution.

To create the script, evaluate in a workspace:

```
Store.DbRegistry createInstallScript
```

This opens Log Window. You will be prompted to enter the table space directory path, which is the directory where the Store database files will be created. Enter the directory path and click **OK**.

The script to create the files and build the tables is created in the Log Window.



Edit this script as needed.

To execute the script within VisualWorks, select **Execute SQL → Execute all**. After the tables have been installed, select **PostInstallation → Run**, which will prompt you for a database name. Enter a name that will uniquely identify this Store repository and click **OK**.

If the script must be run outside of VisualWorks by a DBA, save the script using **File → Save As**. When the script is executed, no database name is specified. The first person to attempt to connect to the repository will be prompted for the database name. Enter the name and click **OK**.

Index

A

addFile: 4-16
adding
    user accounts 8-1

B

blessing levels
    changing 6-5, 7-3
bundles
    browsing with StORE 5-9
    loading from the StORE repository 5-10
    publishing 5-15, 5-17

C

change set
    StORE 5-13
code repository
    local 5-8
    setting owner 3-2, 3-4, 3-5, B-2, B-3, B-4
    setting up for StORE 3-2

E

external files 4-15

L

limitations
    database 1-7
    StORE 1-7

M

maintenance development 5-6

O

override
    defined 4-21

P

packages
    browsing with StORE 5-9
    creating in StORE 4-8
    guidelines for defining 4-1
    initialization and finalization 4-12
    loading from the StORE repository 5-10
    loading sequence 4-12
    publishing 5-14
    setting properties 4-12
    state indicators in browsers 5-12
pad source 4-20

R

reconcile 2-21, 5-6
removeFile: 4-16
repository
    local 5-8
    See also StORE
    setting owner 3-2, 3-4, 3-5, B-2, B-3, B-4
    setting up for StORE 3-2
    StORE, garbage collecting 8-7

S

Sore
    adding users 8-1
StORE
    bundles, publishing 5-15, 5-17
    garbage collecting the repository 8-7
    Merge Tool 7-6
    packages, creating 4-8
    packages, initialization and finalization 4-12
    packages, publishing 5-14
    packages, selecting current 4-9
    packages, setting properties of 4-12
    packages, state indicators 5-12
    Published Items browser 5-11
    publishing code fragments 5-19
    setting repository owner 3-2, 3-4, 3-5, B-2, B-3, B-4
    setting up the repository 3-2
    switching repositories 5-7
    user/group management, setup 8-3
    using a local repository 5-8
    versions, browsing and comparing 6-3
    versions, integrating 7-3
    working off-line 5-3

U

users
    adding 8-1

V

versions
    browsing and comparing 6-3
    changing blessing levels 6-5, 7-3

development or working 6-1
in StORE, defined 6-1
integrating 7-3
parent 6-1