Cincom®

Cincom Smalltalk™

VisualWorks™ 7.6

Release Notes

P46-0106-14

SIMPLIFICATION THROUGH INNOVATION®

Part Number: P46-0106-14

Software Release 7.6

This document is subject to change without notice.

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

1

Introduction to VisualWorks 7.6

These release notes outline the changes made in the version 7.6 release of VisualWorks. Both Commercial and Non-Commercial releases are covered. These notes are not intended to be a comprehensive explanation of new features and functionality nor are they intended to be used in lieu of the product documentation. Refer to the VisualWorks documentation set for more information.

Release notes for 7.0 and later releases are included in the `doc/` directory (7.2.1 release notes cover 7.2 as well).

For late-breaking information on VisualWorks, check the Cincom Smalltalk website at http://www.cincom.com/smalltalk. For a growing collection of recent, trouble-shooting tips, visit http://www.cincomsmalltalk.com:8080/CincomSmalltalkWiki/Trouble+Shooter.

Product Support

Support Status

Basic support policies for the current release are described in the licensing agreement. As a product ages, its support status changes. To find the support status for any version of VisualWorks and Object Studio, refer to this web page:

> http://www.cincomsmalltalk.com:8080/CincomSmalltalkWiki/Cincom+Smalltalk+Platform+Support+Guide

Product Patches

Fixes to known problems may become available for this release, and will be posted at this web site:

http://www.cincomsmalltalk.com/CincomSmalltalkWiki/VW+Patches

ARs Resolved in this Release

The Action Requests (ARs) resolved in this release are listed in doc/fixed_ars.txt.

Additional ARs may be discussed in individual sections of these release notes.

Outstanding ARs and limitations are noted throughout these release notes, as appropriate.

Items Of Special Note

Installing VisualWorks on Vista

Microsoft Vista operating system imposes restrictions on file permissions that are not accommodated by the installer in this release. Accordingly, there are a few special considerations when installing VisualWorks on Vista. We recommend the following practices.

- We recommend that you turn UAC (User Account Control) **off** (in **Settings → User Accounts**), if you can, for the installation procedure. Note that switching UAC on / off requires a reboot. This will give you the equivalent of admin rights, allowing you to do the install directly.

  a Run the installation program.

  b When installation is complete, create a file association for Smalltalk image files to visual.exe

  c Start VisualWorks and set the VisualWorks home directory (**File →Set VisualWorks Home…**).

  d Re-enabling UAC. With UAC in place, VisualWorks Home cannot be changed.

- If UAC cannot be switched off by policy, you need to install as a user with admin rights. To run applications as administrator, right-click on the program and select **Run as administrator**.” Unfortunatly this does not work for the installWin.bat-Script on the distribution CD because it uses relative paths, and running it as administrator changes these paths. So you need to edit the script before running installation.

a Copy **`installWin.bat`** to the desktop or other location.

b Edit the path in this script to

```
@Start X:\vw7.6\bin\win\visual.exe
   X:\vw7.6\image\install.im -installMap X:\install.map
```

where the three occurrences of X are is the drive letter of the installation CD.

c Run the edited script by as administrator by right-clicking and selecting **Run as administrator.**

d Select options and install as usual.

e When installation is complete, create a file association for Smalltalk image files to visual.exe.

f Because all files are write-protected, reset the permissions as appropriate to allow saving the image file, etc. Also create program menu items and shortcuts as desired.

visual.im is ReadOnly

Over the years we have implemented various mechanisms to help users restore a clean image. These involved restoring a “clean” visual.im, either from install media or an installed backup. We have also recommended never saving work to visual.im, but also to a newly named image file.

While these facilities remain available and the recommendation holds, with this release we are making visual.im read-only, making the recommendation more forceful. The save dialog will be more helpful in encouraging you to create a new image file.

Widgetry Discontinued

Development of the Widgetry GUI framework has been discontinued, and it has been removed from preview/beta. It is, however, available under Contributed, and is available on an unsupported basis for user use and development. Widgetry is available on the CD and from the Cincom public repository.

Note that the `Announcement` event management system, though introduced in connection with Widgetry, remains supported and used in the system. It is documented in the *Application Developer’s Guide*.

Known Limitations

While a large number of ARs (Action Requests) have been addressed in this release, a number remain outstanding.

Known Limitations sections are provided throughout this document, pertaining to specific product areas.

Delay and Time Change Interaction

It has been noted, particularly on Windows systems, that changing the time clock adversely affects applications that are in a Delay. The results vary, but can be as severe as an image hang or crash.

The problem occurs if the system gets out of synchronization with network time, so that a large correction is necessary. The problem can be minimized by configuring windows to run a full NTP server, which changes time gradually, rather than the default SNTP server that corrects the time all at once.

Arbitrary changes to the clock will continue to cause problems with running applications in a Delay.

Issue loading Packages

Loading a package with an error in an “initialize” method pops up the debugger. After repairing the method, accepting in the debugger and proceeding, everything seems to be fine. But the package is not marked dirty and “published items/examine/Compare with parent” on the image package does not show the changed method.

The problem arises from the situation of the package load process when opening the debugger: the new package is not hooked up in the system yet and the change doesn't apply to the new loaded package.

This problem is noted and is slated to be addressed in the next release cycle.

HPUX11 User Primitive Engine

The HPUX11 User Primitive engine does not run for as yet not understood reasons. This is covered by AR 49661. The engine appears to compile and link correctly but then exits prematurely after executing a few Smalltalk expressions, apparently without error.

Warning Message Installing MacOS X

MacOS X seems to be over-sensitive to ISO formats, and so warns that the ISO downloads may be corrupt. We are aware of this issue and will resolve it once we identify a good solution. In the mean time, in general, the ISO is not corrupt and installation can safely proceed.

Limitations listed in other sections

- SNMP preview Limitations
- IDNA preview Limitations
- WinCE Known limitations

2

VW 7.6 New and Enhanced Features

This section describes the major changes in this release.

Virtual Machine

Multi-core/CPU Issues

We have found that when Windows runs single threaded programs in multicore/multi-cpu computers, the performance of such programs can be severely degraded. At times, the impact is so high that programs run at less than half their potential speed.

The VisualWorks VM, like any other single threaded program, is also affected by this problem. Note this issue is caused by the Windows process scheduler, which frequently swaps the core/cpu in which the single threaded program is running. This leads to CPU cache thrashing, and thus the slowdown.

Something that can be done to avoid this performance loss under Windows is to use the tool **`imagecfg.exe`** to add information to the virtual machine executable so that Windows allocates only one core or CPU to it at the time the program is loaded. Once the .exe file has this additional information, Windows will cycle through the available cores/CPUs automatically each time the VM is loaded, so you only need to do this once. This behavior has been preliminary confirmed under Windows XP.

To do so for the VisualWorks VM, simply execute:

```
imagecfg.exe -u vwnt.exe
```

Make sure the VM executable file is not read only, because otherwise the operation will fail.

The **`imagecfg.exe`** tool comes with various Windows resources such as the Resource Kit, and is also referenced to be in the folder i386/support/debug. The following page has a link for imagecfg.exe:

http://www.robpol86.com/pages/imagecfg.php

We are investigating the issue under Linux and other Unix operating systems as well.

Digital Signing of Executables

The VMs for Windows platforms now include a manifest, and the VMs plus vwntoe.dll are digitally signed, as required for Vista deployment.

Note that if you use a a tool like ResHacker to add resources to one of these executables, the digital signature for the VM will no longer apply, and must be re-signed. See Microsoft resources for information on Authenticode signing, such as

http://msdn2.microsoft.com/en-us/library/ms537359%28VS.85%29.aspx

Microsoft Vista Support

With this release we add Microsoft Vista to our list of supported platforms.

At this time, our experience with Vista is limited. You should be aware, however, that user privileges have changed significantly from earlier versions of Windows, and could cause issues. Many resources cannot be updated without administrator privileges, so your application may need to be changed to avoid violating Vista's restrictions.

Refer to Microsoft documentation, such as this security note, for more information. Also refer to "Installing VisualWorks on Vista" on page 10 for installation recommendations.

New MacOS X VM

Host Printing

The default printing API uses the Postscript printer. A preview parcel, when loaded, provides an API to use the native host printing primitives in the VM instead. This code only works in VisualWorks 7.6, where printing API changes have been made to accomodate the new printing primitives for MacOSX.

Headless Support

There is no headless VM for MacOS X. To run headless on MacOS X, use the MacOS X X11 headless VM, vwmacxx11. X11 need not be installed.

VM Compilation Environments

The following table lists the compilers and versions used to build the VisualWorks 7.6 object engines.

Product Build Compilers (as of the 7.6 release)

Platform / OS Version	Compiler
HP-UX 11.11	HP92453-01 B.11.11.10 HP C Compiler
IBM AIX 5..2	C for AIX version 6
Intel Linux 32-bit	RedHat 7.2 with local gcc 3.2.3
Intel Linux 64-bit	RedHat Enterprise Linux ES 3 with stock gcc 3.2.3
MacOS 9	MPW Toolset, version 3.5x
MacOS X	GCC 4.0.1 build 5250
Windows Server 2003 SP2	MS Visual C/C++ 6.0 (MS C/C++ Optimizing Compiler Version 13)
SGI IRIX 6.5	MIPSpro Compiler version 7.41
Solaris 8	Sun WorkShop 6 update 2 C 5.3 2001/05/15

IEEE floating point

As of release 7.5, the engine supports IEEE floating-point primitives. The old system used IEEE floats, but would fail primitives that would have answered an IEEE **`Inf`** or **`NaN`** value. The new engine does likewise but can run in a mode where the primitives return **`Infs`** and **`NaNs`** rather than fail.

Image-level support for printing and creating NaNs and Infs has been contributed by Mark Ballard and is loadable from **`parcels/IEEEMath.pcl`**. To use this facility load the IEEE Math parcel and start the engine with the **`-ieeefp`** command-line option.

Base system

Process Termination

Formerly, the Process >> terminate method could interrupt or prevent an unwind block from being run. In general, this is undesirable behavior. In 7.6, this behavior has been changed to ensure that unwind blocks are executed to completion by terminate.

In addition, two new methods have been introduced, terminateUnsafely and terminateUnsafelyNow, that are more forceful in their termination behavior, interrupting the unwind blocks. These are rarely needed, except by the Debugger.

Typically, applications that sent terminate assumed the better behavior now provided. If your application depended on the unsafe behavior, use one of the new methods.

The method comments are repeated here for reference.

terminate
: Terminate the receiver process, by sending the Process terminateSignal. Allow all unwind blocks to run, even if they are currently in progress.

terminateUnsafely
: Terminate the receiver process, by sending the Process terminateSignal. Unwind blocks will usually be run, but if the process is in the middle of an unwind block when the terminate signal is received, then that unwind block will not be completed, but subsequent unwind blocks will be run. This is the semantics used when you close the debugger while debugging a process. In this circumstance it is often appropriate, because the unwind block may not be able to complete, either because of an error, or because it is hung, and this is the reason you are in the debugger. In most other circumstances, you would want to use #terminate, which allows unwind blocks in progress to complete. In very rare circumstances you might want #terminateUnsafelyNow, which terminates the process without attempting to run any of its unwind blocks.

terminateUnsafelyNow
: This is an even more unsafe variation of terminateUnsafely. It will immediately stop the process, making no attempt to run any unwind blocks. This should be used with great caution.

Command Line Option Comparisons

In previous versions, the command line interests defined by CommandLineInterest classes were matched in a case-insensitive way. This primarily affects options defined by e.g., the -option or -option:sequence: pragmas in Subsystem classes. This did not match the normal expectations of users, who often expect to be able to use e.g. -d and -D to indicate separate command line options.

The behaviour was changed to use case-sensitive matching on all platforms. Note that a few options, where the case might not be completely obvious, have been provided with two option pragmas, one for each of the common cases, e.g., -fileIn and -filein.

New Hash Algorithms

For 7.6, the implementation of several hash methods has been improved.

The most important change made to hashing is that the hash values of strings, symbols, large integers and byte arrays are now calculated using all available bytes and/or character values. Furthermore, since doing this in Smalltalk would be onerous without adaptive optimization, the hash values are calculated by new primitives. Using primitives allows speedups such as cutting the time needed to hash all symbols by a factor of 7, while at the same time producing much higher quality hash values which further increases hashed collection efficiency.

Primitive 1700 calculates hash values for objects that look like byte arrays (byte strings, byte symbols, byte arrays, large integers).

Primitive 1701 calculates hash values for byte like strings for which the characters are mapped. It takes the decoding table for the string in question as an argument.

Primitive 1702 calculates hash values for two byte strings and two byte symbols.

The hash function used is basically as follows (e.g., for LargeInteger):

```
hash

| answer |
 <primitive: 1700>
 answer := self basicSize.
 1 to: self basicSize do:
     [:eachIndex |
         answer := (answer + (self basicAt: eachIndex)) hashMultiply
      ].
 ^answer
```

The operation hashMultiply, implemented in SmallInteger via primitive 1747, is a multiplication by 1664525 modulo 2^28. The constant is taken from a table of good quality factors for linear congruency random number generators in *The Art of Computer Programming*, volume 2, by Donald E. Knuth. This hash function has been measured to have good hash value distribution properties when hashing the objects referenced above.

Since the hash of symbols is now calculated via a primitive (by means of the message stringhash), the symbol table is now more efficient than before. In addition, the symbol table itself has been moved from class Symbol, and it is now an instance of SymbolTable or one of its subclasses. The system is shipped with a symbol table implemented by FastSymbolTable. Other symbol tables are available, and switching between different storage vs. speed strategies can be done by simply evaluating expressions such as:

```
SymbolTable new rehashIntoFastTable
```

This has allowed refactoring to occur, the result of which is an even more efficient symbol table. For instance, the previous symbol table could send the message stringhash to a string or symbol up to 3 times while servicing a single request. This issue has been rectified. To illustrate the performance gains, before the new hash values were calculated with primitives, the new symbol table was faster than the old one even though the Smalltalk implementation of hashMultiply is quite expensive.

Note that SymbolTable new will answer the symbol table singleton. The presence of the new symbol table also means that Symbol class>>table will no longer answer the array of hash buckets with which the old symbol table was implemented. To enumerate all symbols in the system, simply use Symbol class>>allSymbolsDo:. Direct access to the internal storage of the symbol table is not recommended.

Finally, symbol tables understand the message normalizedHashBucketSizeChiSquared. This message will answer a number representing the effectiveness of the hash buckets in terms of deviation from the expected average hash bucket size. The ideal value is zero. The greater the value, the higher the benefit to be obtained from rehashing the symbol table. Note that, for this to be meaningful, the hash of strings and symbols should be of high quality. If rehashing the symbol table does not reduce this value to almost zero, then it could be an indication of low quality character array hashing. It is suggested that symbol tables be rehashed if the answer to this message becomes larger than about 1.

Note that SequenceableCollection>>hash has also been changed to use a hash function dependent on hashMultiply, and that the whole contents of the collection will be considered to calculate the hash value. While this may be seen as a potential performance problem, the issue is that having a default hash function which ignores data runs the unavoidable risk of being completely inadequate for easily described data sets. This is acknowledged to be a flaw of hash functions because it causes the performance of hashed collections to become that of linear search. Therefore, if an application typically hashes large sequenceable collections, it is suggested that what should be hashed is the meta data describing what the large sequenceable collection is. This will most likely be much smaller and also more appropriate to hash, because the meta data will include some form of tag which uniquely identifies what the objects contained in the large collection represent.

A number of other implementors of hash have been improved as well, particularly in terms of hash value distribution. Some example classes which now have updated instance side hash methods are: Point, AnnotatedMethod, GeneralBindingReference, Time, and Timestamp. Also, note that the hash functions for Float and Double have been enhanced, and that they are now platform independent.

Finally, note that Set class>>largePrimes has been updated so that better prime numbers are used as sizes of hashed collections. The method comments include the means to generate the list of prime numbers chosen.

For interest sake, a tool, called Hash Analysis Tool, is available from the public Store repository. To use it, load the bundle called Hash Analysis Tool and then evaluate:

```
HashAnalysisToolUI open
```

Default Random Class

The default random number generator has changed. It is now a new generator implemented by the class LaggedFibonacciRandom.

The previous MinimumStandardRandom generator, now renamed to ParkMillerRandom, is not recommended when a quality random is required. Applications should use the LaggedFibonacciRandom instead. This is because the random values provided by the Park-Miller generator are double precision floating point numbers which have up to 53 significant bits. Since only the first 31 bits of their mantissa are known to have good random properties, the behavior of the remaining 22 bits is undefined. In particular, bit aliasing occurs during the calculation of the next random value, and bit 22 of the mantissa is always 1.

The LaggedFibonacciRandom generator is superior to the Park-Miller generator because, as configured by default, it has a period of the order of about 2^258, it produces good quality values across all of the first 53 significant bits starting at the binary point, and it is also about 35% faster.

Collection sort methods

In former versions of VisualWorks, some methods which made use of SequenceableCollectionSorter (methods sort, sort:, sorted, sorted: and sortWith:), would return the sorter, rather than the collection. These have been changed to return the collection instead.

aDictionary collect:

Dictionary protocols such as reject:, select:, etc, usually return another Dictionary of the same sort and, according to the ANSI spec, collect: should too. But for a long time it has returned an arbitrarily ordered OrderedCollection of the values with the transform applied. The correct behavior has been available in the ANSICompatibility parcel as an optional load.

In this release, the default behavior is the ANSI specified behavior: collect: when sent to a Dictionary returns the same kind of Dictionary. The old method can be loaded from the ANSIUnCompatibility parcel, if you need the old behavior to be restored.

Float timesTwoPower:

(AR 53035) The implementation of Float>>timesTwoPower: as a primitive was capable of answering wrong results if the receiver of the message was denormalized. For example, evaluating the expression

```
(2.0 timesTwoPower: -150) timesTwoPower: 150
```

would result in 2^23 expressed as a float, 8.38861e6, instead of answering the correct result, 2.0. As it turned out, implementing the correct functionality in the image via a lookup table of factors was just as fast as the existing primitive, and so this operation has been moved into the image.

While the implementation of the primitive for Double>>timesTwoPower: was correct, it was quite inefficient. The image based implementation is just as correct, and in addition it is significantly faster.

The new class instance variable powersOfTwo, defined in LimitedPrecisionReal, holds on to the lookup table.

RTP ErrorNotifier

The Runtime Packager previously installed its own Error notifier and had a hierarchy of RuntimeImageDumper classes for dumping image state in packaged/stripped images. This has now been refactored to move the Runtime Packager's error notifier and system dumper into a package called ErrorNotifier, included in the base system.

Previously, the base runtime image (base.im) included the original debugger in order to be able to respond to errors. This debugger has been superseded by the PDP debugger in development images for some time, but needed to remain in the system in order to have some response to error conditions when the PDP debugger was not present. This change allows us to remove all debuggers from base.im and have it use the ErrorNotifier to generate stack dumps on exceptions.

Point Positions

The methods center, topCenter, rightCenter, bottomCenter, leftCenter are no longer guaranteed to return integer values. They now return the true midpoint between the appropriate points, whether it be in integers, fractions, floats, or any other element of the Smalltalk transcendental number types.

Two APIs of interest have been added to the ArithmeticValue. One is the unary method half, which returns the equivalent of

```
(self / (self unity + self unity)).
```

Various subclasses have optimal implementations.

The other is midpoint: which returns the a value that is midway between the receiver and the argument.

ColorPreferenceCollection

It has been observed that if ColorPreferenceDictionaries are created that reference a SymbolicPaint rather than a concrete color (i.e. ColorValue, Pattern, or DevicePaint), there is a likelihood that a color look-up recursion fault will occur.

To assist in avoiding this situation, a new protocol named "testing" has been added to ColorPreferenceCollection class that includes isCyclic and helper method cyclicEntry:. Sending isCyclic to a ColorPreferencesCollection sub-instance answers true if the receiver has a cyclic entry that would lead to a runaway cyclic lookup.

To test the isCyclic method try the following in a workspace, which introduces two cyclic entries:

```
preferences := Window currentWindow paintPreferences copy.
preferences  matchAt: SymbolicPaint shadow put: SymbolicPaint background.
preferences matchAt: SymbolicPaint background put: SymbolicPaint border.
preferences matchAt: SymbolicPaint border put: SymbolicPaint shadow.
preferences matchAt: SymbolicPaint foreground put:
    SymbolicPaint foreground.
preferences isCyclic. "true"

Window currentWindow paintPreferences isCyclic. "should be false"
```

Package Scope for Extension Methods

A common problem with writing packages that extend the behavior of objects found in other packages is that the extension methods should be scoped to the code the package represents rather than that of the class.

Consider the following scenario. You want to add your own inspector to the system. You create a package. You place a namespace called MyOwnInspector in your package, and place the classes for your code in that namespace; doing so keeps them from conflicting with other inspector classes in the system. Then to make your inspectors easier to get at, you extend some of the base classes of the system with messages like:

```
Object>>inspectMyWay
    MyOwnInspector.StandardInspector openOn: self

Number>>inspectMyWay
    MyOwnInspector.NumericInspector openOn: self
```

Nominally, every extending method has to specify the complete path to your objects. New in 7.6, is the ability to specify a “default name space” for a package. Set the default name space using the **Default Namespace** menu item in the packages lilst. This is stored as a String in the namespace property, in the **Inspect All** properties item. When you set the default name space, extension methods in your package use this name space of the extending package, rather than the classes name spaces. The above methods can now read:

```
Object>>inspectMyWay
    StandardInspector openOn: self

Number>>inspectMyWay
    NumericInspector openOn: self
```

This affects the search order as follows. For a non-extension method, or if the package does not define a default namespace, the search order is:

1. Temporary variables in the method
2. Instance variables
3. Shared variables defined by the class, and variables imported by the class
4. Shared variables defined in superclasses, and variables imported by superclasses
5. The namespace that contains the class

If a default namespace is defined, the default name space changes the last item in the search order. The order changes to:

1. Temporary variables in the method
2. Instance variables
3. Shared variables defined by the class, and variables imported by the class
4. Shared variables defined in superclasses, and variables imported by superclasses
5. The package's default namespace

Note also that this feature applies only to packages, and not to bundles.

InteractiveCompilerErrorHandler now declares imports as private

In former versions of VisualWorks, when one used the interactive compiler warning handler, if one chose to import a variable name (or it's containing namespace) into a class, the import was public. These imports are now marked as private by default.

New BlockClosure cull methods

Three new interesting methods have been added to BlockClosure:

- cull:
- cull:cull:
- cull:cull:cull:

These methods are similar to value: counterparts. The difference is that when a BlockClosure receives the value: message, it will raise an error if the receiving blocks argument count does not match the argument count of the value: message. The cull: methods are tolerant of receivers that may have fewer arguments than the calling method. For example:

```
[:a :b | a + b] value: 3 value: 4 --> 7
[:a :b | a + b] value: 3 value: 4 value: 5 --> error
```

whereas

```
[:a :b | a + b] cull: 3 cull: 4 --> 7
[:a :b | a + b] cull: 3 cull: 4 cull: 5 --> 7
```

This allows methods which take blocks as arguments to support optional arguments. An example of this in pre 7.6 releases is the ifNotNil: method, which when sent to an object, may take a block with 0 or 1 arguments. It now uses cull:. Additionally, the BlockClosure on:do: message now uses cull: to allow the argument to the handler to be additional. For example:

Pre 7.6

```
[3 / 0] on: ZeroDivide do: [:ex | 17] --> 17
[3 / 0] on: ZeroDivide do: [ 17] --> error
```

7.6

```
[3 / 0] on: ZeroDivide do: [:ex | 17] --> 17
[3 / 0] on: ZeroDivide do: [ 17] --> 17
```

Moved Timed Block from Trippy to Base

A facility originally put in place to protect against infinite print strings has been refactored and made generally available. Blocks of code can now be given a timeout (in seconds) and an alternative action to take if they do not finish in time. A simple example:

```
[10000 factorial] valueWithinSeconds: 2 orDo: [-1]
```

Care should be taken when using this facility. Consider the following example:

```
| set random |
set := Set new.
random := Random new.
[1000000 timesRepeat: [set add: (random next roundedTo: 0.001)]
   valueWithinSecond: 0.5 orDo: [ ].
set do: [...]
```

This would be a bad idea. We can't determine exactly where the code was at when it timed out and interrupted. So the set object may be in an invariant state that makes the object undesirable for further use. Care should be taken using a timeout facility to verify the integrity of any data that may be under alteration in the block, if the same data is to be used in later computations.

BufferedExternalStream contentsLineEndConvention removed

In an effort to improve support for automatic line end convention detection for wider range of stream types we decided that following the contentsLineEndConvention pattern would be a poor choice. This method leaves the receiver in transparent line-end mode which is usually neither what the mode was before the method was called nor is it what it should be.

Instead, the way to inquire about the line end convention of the contents of an external or an encoded stream is to set its line end convetion to LineEndAuto and then inquire about the detected mode using the lineEndConvention accessor.

UTF-8 File and Directory name encoding

It is increasingly common that operating systems use UTF-8 for the encoding of file and directory names in the file system. Previous releases would not handle this correctly. In the case of Unix and Linux systems, this had to do with the presence of Locales. The current Locale class combines a country/language and an encoding. e.g., en_US.iso-8859-1. There were no encodings using UTF-8, and if no matching encoding was found, it would fall back to the default "C" encoding, which used ISO-8859-1. If this didn't match the filesystem, then using characters outside the basic ASCII range could fail.

This issue has been resolved by adding .UTF-8 locales for all existing locales in the system. For operating systems where we haven't defined a locale, adding an appropriate one will make it use the filesystem correctly. A longer-term solution is to separate the encoding from the country/language.

For MacOSX, the file system is always in a particular form of UTF-8, in which accented characters are decomposed. In this case we simply need to hard-code the MacOSXFilename to use this encoding consistently.

In both of these cases, the fix can be made entirely at the image level. For MS-Windows, the fix to support "unicode" characters in file and directory names will require VM changes, and is not yet available.

Snapshot and leave running

It can be useful to make a snapshot of the current image, but leave the image running, and not change the image file name it thinks it has, the change file name, and so forth. Saving an image as headless did this previously. This capability is now generalized so that you can call Snapshot>>saveDetachedTo:thenQuit:.

Partial URL Support

The Seaside efforts in this release cycle required some improvements in handling of partial URLs. Previously there was some basic functionality available in the form of the resolvePath: message, but it was rather limited because the argument was only a String, which made handling of more complex cases difficult. In this release we have introduced a new PartialURL class, modelling a URL that is just a path, query and fragment (any of these elements can be omitted).

For example, the following expression is now valid

```
'/images/test.png' asURI
```

Partial URLs are primarily used to augment other URLs, i.e., to modify the other URL using elements of the partial one. For example, often when dealing with XML resources of one form or another, you'll be handed a relative URL that is meant to be relative to the xml:base URL.

As part of the PartialURL changes, the capability of the resolvePath: API was greately enhanced. It can now resolve any combination of full and partial URLs. We've also made few changes for convenience. First, the argument can be provided either as a string or as a URL object. Second, we've aliased the resolvePath: message with #, (comma, as a message name). With these changes we can present few example resolutions:

```
'http://localhost:7777/' asURI, '/images/test.png'
```

returns

```
<URL:http://localhost:7777/images/test.png>
```

```
'http://localhost:7777/' asURI, 'https://secret:1234/files/things.zip'
```

returns

```
<URL:https://secret:1234/files/things.zip>
```

The established combination rules are somewhat complicated, however one thing to keep in mind when using partial URLs is that they can be absolute or relative. Path of an absolute URL entirely replaces the path of the receiver whereas path of a relative URL replaces only the last component of the receiver's path. For example:

```
'http://localhost:5432/path/file.txt?query#fragment' asURI, 'absolute.jpg'
```

returns

```
<URL:http://localhost:5432/path/absolute.jpg>
```

```
'http://localhost:5432/path/file.txt?query#fragment' asURI, '/absolute.jpg'
```

returns

```
<URL:http://localhost:5432/absolute.jpg>
```

Note that resolveRelativePath: is now deprecated. Use resolvePath: instead, the right thing happens depending on what type of URL the argument is.

Invoking External Process

There has long been a mechanism for invoking external processes from Smalltalk, but it has lacked some important features and has not been compatible between different operating systems. This release contains a significant overhaul of the functionality. This is based on work that was in the package ExternalProcessStreams in the public repository, but is significantly changed from that.

Most of the protocol is now instance-based rather than class-based, so you work by creating an instance of ExternalProcess or one of its subclasses. Sending new to ExternalProcess will automatically create an instance of the appropriate subclass.

The underlying method which all of these facilities call is now execute:arguments:do:errorStreamDo:. This allows you to pass a command, arguments, and blocks which are given the standard in, out, and error streams from the resulting process. This gives the ability to stream in and out to the process, and because separate processes are spawned for the two blocks, to respond to error and input information separately. Where this generality is not required, convenience methods like fork:arguments: can be used. The existing protocol like shOne: will also work, and will also work on Windows systems, where it will invoke the Windows command interpreter. However, the actual command string sent will usually need to be different between Windows and Unix systems.

Because most of the work is now done by instances of ExternalProcess, it is possible to configure those instances before the process is invoked, and it is also possible to hold onto the instance and get access to its various attributes. Two common areas for configuration are the expected line end convention and the encoding for the input/output streams. These are set with the messages lineEndConvention: and encoding:. e.g.

```
anExternalProcess lineEndConvention: IOConstants.LineEndTransparent.
anExternalProcess encoding: #'utf-8'.
```

Note that the underlying implementation of WinProcess has changed significantly. The older version always forked off a Windows shell as a long-running process and issued commands to it with a special delimiter to try and detect the end of a command output, and then explicitly exited the shell when finished. The newer version is much more compatible with the UnixProcess implementation, and, for example, the shOne: command will fork a Windows shell which executes the command and then exits. If you need the functionality of a long-running Windows shell, this can be built using the execute:arguments:do:errorStreamDo: functionality. Note that it is also possible to invoke Windows programs directly, with no shell, and this is what fork: and fork:arguments: do.

For examples of how to use these facilities, see the package comment in OS-ExternalProcess, or the class and method comments on ExternalProcess and its subclasses.

Parcel/Package Loading Discrepancies

(AR 53000) Discrepancies were discovered in the state of the system at postLoadBlock-time, depending on whether the code was being loaded from a parcel or a package. In the case of a parcel, the parcel would not yet be registered with the system. This has been changed to occur later, so that there is consistency between parcels and packages.

HandleRegistry Change

Prior to this release, HandleRegistry was not only a WeakDictionary, but one whose keys were accessed via identity (e.g. IdentityDictionary). This was found to be problematic in light of platforms that key things by integer handles when those handles become large integers and later access is no longer identical, but still equal.

HandleRegistry now uses equality as its API for storing and indexing. This was the behavior of a subclass of HandleRegistry called ExternalRegistry. ExternalRegistry remains in the image for backwards compatibility and will be be removed in the next release.

GUI

Hover Help

The old FlyByHelp has been replaced by Tools-HoverHelp which uses independent state machine implementations based on Announcements.

The programming interface is exactly the same, so no application code should require changing.

The old FlyByHelp is available in the **/obsolete** directory. To revert to the old help system first remove the Tools-HoverHelp package and then load the obsolete Tools-FlyByHelp package. The two should not be used together as they will produce double help windows.

While the new version is backwards compatible, it does so through a more powerful API than provided before. In addition to the original helpText: API, one may also set the tooltip: of a widget. This maybe any object which responds to asTooltipGraphicFor:. In this way, any VisualComponent can be used for a tooltip. Furthermore, it may be a block, allowing the tooltip to be dynamic and computed at open time.

Tree View

Release 7.5 included a change to TreeModel that prevented nil being the value of a node. While this generally makes sense, there may be circumstances in which nil may be a legitimate value. In 7.6, an instance variable has been added, and behavior slightly changed, to accommodate this possibility.

The new instance variable is mayHaveNilValue, with the usual setter and getter methods. By default the value is false, supporting the 7.5 behavior. If set to true, a node may have a nil value, and the behavior is more like the pre-7.5 behavior. The method childrenFor: is changed so the behavior depends on the value of the variable.

Similarly, release 7.3.1 changed the behavior of expandFound: in TreeModel. Currently, the method looks for object as its argument in the tree and if found, expands the tree to the object and answers true. Prior release 7.3.1 the method would also expand the node found to display its children, if any. In 7.6 the old functionality of expanding a node and its children when the argument is found in the tree is offered by a new method, named findAndExpandChildren:.

PragmaticMenuItems Package

One more (and hopefully the last) menu pragma (or method tag) has been added.

```
<itemInMenu: anArrayOfMenuIDs position: aNumber>
```

It takes only two arguments in the tag body which have the common interpretation. It differs from the other menu tag types in that rather than having the code of the method represent action to preform the indicated item, the method actually returns a menu item. So besides which menu and position, the rest of the behavior is configured by sending messages to the MenuItem object to be returned. For example:

```
<itemInMenu: #(#selectorMenu) position: 300.1>

^(MenuItem labeled: #SyncAssets >> 'Sync Assets' << #assets)
    hidden: [self isAnyAssetClassSelected not];
    enablement: #areAssetMethodsSelected;
    value: #syncSelectedAssets
```

and

```
<itemInMenu: #(#selectorMenu) position: 300.2>

^(MenuItem labeled: #AddAssetsC >> 'Add Assets:' << #assets)
    hidden: [self isSingleAssetClassSelected not];
    submenu: [self computeAssetImportMenu]
```

This new tag type allows more of the menu code to be expressed as Smalltalk code and exposes the full APIs (including those one may add) of the MenuItem object.

VisualRow and VisualStack

VisualWorks 7.6 introduces VisualStack, a handy VisualComponent for stacking arbitrary VisualComponents on atop the other, and VisualRow which is a handy way to create a nice horizontal row of VisualComponents. The system itself uses these two "layer" icons as well as show multiple arbitrary icons side by side.

Expose VisualPart Properties API

In order to support the desire to begin evolving the VisualPart framework, we have added a properties dictionary to VisualPart. This allows us to begin experimenting with adding state to VisualPart which at some point may become a real instance variable. It also allows VisualParts to have "optional" instance variables. For example, the new tooltips code registers a widget's TooltipAssistant in it's properties.

Add VisualPart>>superpartsDo: and derived APIs

The terminology of a "superpart" has been introduced. It is synonymous with the "container" term, but is symmetric with the term subpart which will probably be added in a future release. Three methods of interest that have been added:

superpart
Return the VisualPart that contains the receiver

superpartsDo: *aBlock*
Starting with the receiver, enumerate it and all containing super parts up to, but not including the Window

findSuperpart: *aBlock*

Search the superparts tree (not including the receiver) for which *aBlock* answers true when evaluated with each superpart. Return nil if none.

FrameExited/Entered announcements

Views now use the Announcements framework to announce FrameEnter and FrameExited events. This facility is used by the new tooltips code. In a future release, the intent is to make the implementation not rely on a Controller, and thus be able to be issued from any VisualPart.

Graphics Clipping Outside OS Coordinate Bounds

Prior to this release an object engine primitive failure would occur if a GraphicsContext coordinate was outside a 16bit integer range (-32768 to 32767) on X11 platforms. In VW 7.6, graphics is now always clipped for coordinates outside this range instead of raising an exception.

Reduction of UI Flicker

A large portion of annoying UI "flickering" has occurred because invalidateNow or invalidateRectangle:repairNow:, when the argument to repairNow is true, would not use the DamageRepairPolicy. Instead of queuing a damage event it would unnecessariliy create a new GraphicsContext, draw to it, then display it. As of 7.6, all damage is queued with the DamageRepairPolicy and, if repairNow is true, pending damage events are handled immediately.

Note that as of this release the argument to repairNow: in invalidateRectangle:repairNow: or invalidateRectangle:repairNow:forComponent: may only be a Boolean. In recent VisualWorks releases this argument could also be the Symbol #repairNowNoFill, primarily to optimize tree view updates. It is no longer necessary to justify this inconsistency for the tree view.

Tools

Refactoring Browser

A number of changes have been made to the browser and the underlying engine. A few will be obvious, such as that the browser now opens with an Overview pane rather than a class definition template.

Less obvious are that there is no class definition template. You now define classes either using the class definition dialog, or by editing and saving an existing class definition. A partial list of the more notable changes is:

- Able to parse every method in the system.
- Able to properly model <tags> in methods. This means they are now included in rewrite rules. They are also formatted by the structured formatter.
- Able to properly model <C: > methods (since they appear like <tags>, but don't actually follow the same syntax).
- Better management of code model tabs, including a desire to make the comment tab more prevalent (but smart about it). The underlying engine below this has been rewritten and is hopefully much easier to maintain and add new code tools for.
- Richer tabGraphic management for the code tools.
- Add an "Overview" tab.
- StatusBar has been completely rewritten to be a pluggable entity.
- Default browser sizes have been increased.
- Move is a proper refactoring now. This is a substantial piece of work, and might not be entirely finished. When you move objects from namespace to namespace, the references are now rewritten.
- Just one formatter is used now, the RBConfigurableFormatter.
- Ability to "format on view" so that you can look at all of your code the way you like to format it.
- Ability to "format on save" so you don't have to format then save.
- Work was done on the formatter to better handle multikeyword messages, with simple (literal) arguments.

- Work was done to better format literal arrays that appear to be "structured", for example, literalArraySpecs. So you don't have to worry about formatting them when you change them.
- Use of the ExternalWebBrowser-Text APIs to make http links clickable in various views.
- Removal of categories code
- Removal of Parcel Status widget

Assets

Assets are libraries of resources for an application, things such as Images, Masked Image, (OpaqueImages), Strings, Bytes, etc. The libraries support a framework for deriving these objects from external files found in a directory associated with the object and keeping them up to date. They also provide caching of computed assets so that access is fast as possible.

The intent is that an application creates a subclass of Assets and then adds class side methods to retrieve different “assets.” Usually, these methods are annotated with method tags which indicate resource files which may be imported and integrated as Smalltalk code.

A separate loadable package is provided for developers to maintain these (Assets-IDE). The package extends the IDE to show one additional menu items:

Sync Assets
: This menu item is in the Class Menu. It will show up (i.e. it is completely hidden, not just disabled) only when Assets classes are selected.

To begin importing your assets, create a subclass of Assets, select it and invoke **Sync Assets** from the **Class** menu. If this is the first time you've synced it, it will prompt you for a directory to import from. After the first time, it will remember the directory for future syncing. Sync'ing involves checking the file's md5sum against what is stored in the method. If it does not match, the method is regenerated so that the once block again produces an object consistent with that of the file. In the event that a directory is synced against, there are asset methods for which no file was found, the user will be given the chance to remove these or let them stay.

Default import implementations are provided in the Assets class for files that have extension types of .bmp, .gif, .jpg, .jpeg, and .gif. You may add more of these to the Assets class or to your own specific subclasses. And if you don't like the default import type of one of the 5 predefined ones, you can of course override the method in your subclass to do something different. See the import-types and private-import-helpers method categories for examples.

Inspector Enhancements

The Inspector now shows icons next to each field. The icons are the toolListIcon of the class of the object. Icons which show the immutability (lock icon) as well as protected fields (shield icon) are added next to the receiver type icon where appropriate. Dynamic computed fields have a gear icon next to them.

The inspector also now presents the variables alphabetically by default. The list menu has an option to toggle to the defined order (for that inspector window).

The inspector had some work done so that it is safer in how it deals with proxy objects. It will correctly show proxies as a proxy object, rather than the proxied object. And takes care to not send methods that help it do its work unless it determines it can safely do so.

The inspector's ability to "undo" changes you make from it is generally a good idea. One problem it can create though is when you remove something via the inspector, and then can't figure out why it's not being garbage collected. The "undo" history holds on to the old value. So an option was added to "forget" undos.

Improve automatic method categorization

The browser integrates functionality similar to the goodie ConsistentProtocols. It is generally a good practice to have the method interface of an object appear in the same method categories as defined in super classes so that API information is preserved. The browser will attempt to keep subclass method implementations in the same as the super class. There is a setting in the Settings Tool to control how aggressive this behavior is. The default is to do so only for new methods. If a method has been re-categorized and then changed, it will remain in the newer category. The other setting will always move the method (old or new) to match that of a parent implementation.

Also, the browser now always provides a method category to add methods in, even if there isn't one. It shows up as an empty protocol and may be added to immediately. It will only show up when there is no method categories yet. The default for instance side programming is 'accessing' and for the class side is 'instance creation'. However, a given class heirarchy may choose to adjust this by implementing the initialMethodCategory message to return a default category. For example, the TestCase implementation in the SUnitToo package implements this to return 'tests'. In this way as soon as the class is created, one can simply start writing tests. The change is intended to reduce the initial experience where one creates a class and then wonders “how do I add methods now?”

Improvements to the Debugger’s "browse" feature

The **browse** option found in the Debugger's stack list menu now usually browses the method shown. If the receiver class and method implementing class do not match, the browser is opened with the hierarchy expanded to the receiver class, but the implementing class/method. This is true for most messsages in the system. There is a small set of methods which are commonly encountered in the debugger for which this would not make sense to do. An example is the doesNotUnderstand: selector. For methods found in Object and implemented in the Kernel-Objects package, the debugger will open the browser on the receiver's class, with no method selected.

Database

MySQL EXDI

The MySQLEXDI package works directly with the MySQL api library (a DLL) to perform MySQL database queries. It works just like our other database packages, with a few quirks based on the MySQL system. The most pronounced difference is that with MySQL, prepared statements (e.g., queries with bound values that can be prepared once and sent many times) are only allowed with limited SQL commands, like: CREATE TABLE, DELETE, DO, INSERT, REPLACE, SELECT, SET, UPDATE, and most SHOW statements. Notably absent are queries like: CREATE DATABASE, CREATE PROCEDURE, CREATE FUNCTION, CALL, USE. These latter are still available, but using an entirely different api.

This division of capabilities challenges the VisualWorks convention which assumes that a single “session” type can do it all. The solution adopted with MySQLEXDI is to use two distinct session types, MySQLSession and MySQLAdminSession. The former works just like the usual EXDI session, with prepared statements and bound values. But only the subset of queries mentioned above will work. The other, administrative, queries can be executed using MySQLAdminSession, which disallows bound values, and returns only string-like objects.

Since most EXDI-using applications, including Ad Hoc SQL and Store, expect to perform both kinds of queries using a single session object, a third session type is available, called MySQLHybridSession. This session is little more than a container of one each of the other specialized sessions. It knows which child session to invoke based on a quick parse of the query at hand. We haven't done extensive research into the limits of this hybrid scheme, but it appears to work reasonably well. One might imagine a complex query that involves both types (prepared and administartive), but such a query probably wouldn't work with a MySQL database anyway.

So, for generic database apps, it is recommended to use the hybrid session, unless the query command set is covered entirely by one or the other session types. Check the comments for these session classes for a complete list of their allowed commands, as dictated by the MySQL vendor.

Here's a simple example, borrowed from the class side examples protocol of MySQLConnection.

```
| conn sess ans data |
conn := MySQLConnection new.
conn
    username: 'root';
    password: 'user';
    environment: 'localhost'.

[conn connect.
sess := conn getSession.
sess prepare:  'SELECT description, example FROM help_topic WHERE
    name=?'.
sess bindInput: (Array with: 'SELECT').
sess execute.
ans := sess answer.
[ans = #noMoreAnswers] whileFalse:
    [ans = #noAnswerStream ifFalse: [data := ans upToEnd].
    ans := sess answer]]
    ensure: [conn disconnect].
data inspect
```

One final note about calling stored procedures. These are not offered as prepared statements (as of MySQL version 5), and that means the no bound parameter passing or retrieving is possible. However, MySQL does offer 'server side variables'. Here's an example borrowed from the MySQL documentation, using server sice variables as parameters to a stored procedure call. Note that the answer stream should ideally be fully flushed after every query. This example uses the hybrid connection to accomodate CREATE and SELECT queries.

```
| conn sess ans data|
conn := MySQLHybridConnection new.
conn
    username: 'root';
    password: 'user';
    environment: 'localhost'.

[conn connect.
sess := conn getSession.
sess prepare:
    'CREATE PROCEDURE p (OUT ver_param VARCHAR(25), INOUT
    incr_param INT)
        BEGIN
            SELECT VERSION() INTO ver_param;
            SET incr_param = incr_param + 1;
        END;'.
sess execute.
sess answer.
sess prepare: 'SET @version = 0'.
sess execute.
sess answer.
sess prepare: 'SET @x = 10'.
sess execute.
sess answer.
sess prepare: 'CALL  p( @version, @x)'.
sess execute.
sess answer.
ans := sess answer.
sess prepare: 'SELECT @version, @x'.
sess execute.
ans := sess answer.
[ans = #noMoreAnswers] whileFalse:
    [ans = #noAnswerStream ifFalse: [data := ans upToEnd].
    ans := sess answer]]
    ensure: [conn disconnect].
(Array with: (data first first) with: ((data first at: 2) asString)) inspect.
```

Client Setup Notes

To obtain the MySQL client software for your platform, see http://www.mysql.com.

Many Linux distributions have the MySQL library already installed. However, you need to ensure that your MySQL version is at least 5.0, which is the version we have tested against. An easy way to check the version is to run the following at the command prompt:

```
% mysqladmin version
```

When you have the correct version installed, there is a chance that the client shared library is not named **`libmysqlclient.so`**, in which case there is typically a symbolic link file with this name. If not, you may want to edit the `libraryFiles:` and `libraryDirectories:` attributes of `MySQLLinuxInterface` to reflect your installation. For example, you may need to change `'libmysqlclient.so'` to `'libmysqlclient.so.0.0.15'`. More information about changing these parameters can be found in the *DLL and C Connect User's Guide* (p.51).

Scrollable Cursors

The VisualWorks EXDIs for Oracle, Sybase, and DB2 provide support for cursors and scrollable cursors. A cursor represents a movable position in the result set. When using a cursor, the results of a query are held in a set of rows which may be fetched either in sequence or via random access.

A cursor is used for sequential access, while a scrollable cursor is used for random access. The scrollable cursor can fetch results moving either forward or backward from a given position, and the specified result may be indicated either via an absolute or relative row offset.

When using cursors, the rows in the result set are numbered starting with one. With a scrollable cursor, you can fetch the same rows several times, you can fetch a specific row, or a specific row relative to the current position.

For descriptions of the API and examples of its use, refer to the *Database Application Developer's Guide*, chapter 2, pp. 2-15 to 2-17.

Store

Applying Bundle Blessing Comments

When publishing a bundle, there was an option to apply the bundle's version/blessing comment to all of the sub-components that were being published at that time. This is now the normal behaviour. The **Apply to All** button has been removed, and unless the sub-component is explicitly given a separate comment, then the parent bundle's comment is applied.

New Blessing Levels

Two new blessing levels have been added to Store.

Replication Notice - The StoreForGlorp replicator adds this to versions that were replicated, noting when each was replicated, by whom, and the name of the source and target databases.

Obsolete - This is intended for components that are no longer in use or have been renamed.

User Server Timestamp for versions

Store now uses a UTC timestamp obtained from the database server (if available) as the timestamp for publishing. If it cannot get a timestamp from the server due to lack of capability (e.g., MS Access) or to an error, then it will use the local image's idea of what the current UTC time is. Lists of versions will now sort according to timestamp rather than according to sequence in the database.

Retain History on Rename

One of the difficulties with renaming a package or bundle is that the old version history is unavailable. Store now detects renames of these components and will automatically display both the newer and the older component. For this purpose, a rename means that the component was loaded under the old name, renamed in the image, and then published under the new name. Because the parent (trace) points to the commmponent under its old name, we can detect the rename and display both lists of versions.

Note that if you have development proceeding under both names, this can be confusing. It can also be confusing if someone looks under the old name. We recommend publishing a version under the old name with a blessing level of, e.g., Obsolete, and a comment that points them to the new name.

WebServices

This release includes some changes in simple types mapping. XMLTypesParser no longer creates the global simple types for user-defined local types.

The XML:

```
<xs:complexType name="readingsType">
    <xs:attribute name="parameterType" use="optional">
       <xs:simpleType>
       <xs:restriction base="xs:string">
          <xs:enumeration value="Raw"/>
       </xs:restriction>
    </xs:simpleType>
    </xs:attribute>
</xs:complexType>
```

is now mapped by the XMLTypesParser as:

```
<xs:complexType name="readingsType">
    <xs:attribute name="parameterType" use="optional">
       <xs:simple name="string" baseType="xs:string">
          <xs:enumeration value="Raw"/>
       </xs:simple>
    </xs:attribute>
</xs:complexType>
```

We used to create the below mapping for simple types:

```
<xs:attribute name="parameterType" use="optional" baseType="xs:string">
    <xs:enumeration value="Raw"/>
</xs:attribute>
```

The XMLTypesParser has not used this type of mapping for about 4 years. These changes no longer allow loading an XML to Object binding with this old mapping. The binding builder will raise an error and the error parameter suggests the supported mapping.

Internationalization

Per-process Locale and Message Catalogs

This new capability is useful primarily for applications executing on the server and serving a number of clients requiring potentially different Locales (e.g., WebToolkit or Seaside applications). With these changes it is now possible to associate a different Locales with each Smalltalk process. Use messages locale and locale: to get/set the locale of a process.

The expression Locale current returns the locale of the process executing the expression. If the process does not specify its own locale, the global Locale.CurrentLocale will be returned instead. There's now also a complementary Locale class>>current: message that can be used to set the locale of the currently active process.

With per-process locale it makes sense to have UserMessages translate in the context of the process locale as well. This behavior required further changes to the way MessageCatalogs are managed. Previously only single locale catalogs would be accessible at any given time. To make other catalogs available to processes with different locales there is a new MessageCatalogManager class. A single catalog manager instance maintains catalogs for a single language. A class side registry, MessageCatalogManager.Managers, maintains managers for all the currently available locales. The registry is keyed by language ID, e.g. “en” for English. See the MessageCatalogsManager class for details on how to create or find catalog managers and how to add new catalogs.

With catalog managers in place UserMessage>>asString can now translate in the language identified by the locale of the current process. As in previous releases, the search for translation uses similar set of fallback resolutions in case the corresponding manager, catalog or translation isn't found, ultimately returning the default string if none of those are found.

Net Clients

MIME Support

In last release we made significant changes in the HTTP framework to support more memory efficient approch of streaming message contents between the source and target streams. The HTTP framework is derived from a generic MIME framework, so in this release we promoted the streaming changes up into MIME level. This provides the same new features to MIME/Mail and brings the two layers back together using the same "stream stacking" approach throughout. Both MIME and HTTP frameworks can now stream large attachments directly to/from external files.

We also renamed some classes in MIME for better naming consistency. The MessageBuildHandler was renamed to MimeBuildHandler, and MessageBuildHandler is now deprecated. The new build-handler class hierarchy now looks as follows:

```
MimeParserHandler
    HttpBuildHandler
    MimeBuildHandler
        MailBuildHandler
        MailFileReader
```

The hierarchy of print-handlers has also changed. There are two fundamental output functions, which we call “printing” and “writing.” Printing displays MIME entities in inspectors or writes to log entries. In order to accomodate any kind of character stream to print on, printing avoids to switching modes and such. Consequently it requires a stream that accepts characters, i.e., a pure character stream or a stream in text mode. Writing is the fully spec-compliant output of messages as they are sent out into the network or stored into external files. Previously, both functions were handled by the same classes, causing a lot of conditional branching and resulting in two largely independent execution paths. We decided to separate them in this release and created two separate hierarchies for message printers and writers with the following structure:

```
MimeOutputHandler
    MimePrintHandler
        HttpPrintHandler
    MimeWriteHandler
        HttpWriteHandler
        SMTPWriteHandler
```

Note again that MessagePrintHandler is now deprecated in favor of MimeWriteHandler for writing and MimePrintHandler for printing.

Attachments

Similarly to the previously released HTTP attachment functionality, all mail attachments can now be automatically saved to an external file. This feature is configurable via the MimeParserHandler saveAttachmentsAsFiles: flag. (Note that the selector was corrected from the previous release, so use HttpBuildHandler saveAttachmentsAsFiles: for HTTP). However the default setting for mail attachments is to *not* save attachments (false), which is different from the HTTP default (true).

Mail attachment files are saved in a directory, which is by default named "mail-temp-files" and located in the image directory. Use the following expression to change the default directory:

```
MailBuildHandler defaultAttachmentDirectory: aString
```

Note that this is another change compared to the 7.5 release, where we used the defaultUploadDirectory selector; therefore, to set the directory for Http attachements (default is "http-temp-files") use:

```
HttpBuildHandler defaultAttachmentDirectory: 'myDirectory'.
```

The file names for attachments are based on the filename parameter in the **`Content-Disposition`** header fields. If a file with that name already exists a new name will be generated. Once the file name is determined the framework raises a notification, AttachmentFilename, allowing the application code to override the file name on the fly. If the notification is not handled the originally suggested file name will be used. Here is an example:

```
input :=
'From: zz@holcim.com
Content-Type: multipart/related;
      boundary="--11"

----11
Content-Type: text/plain; name="budd.txt"
Content-Disposition: attachment; filename="budd.txt"
Content-Transfer-Encoding: base64

QWxhZGRpbjpvcGVuIHNlc2FtZQ==

----11--
' readStream.
[ message := MailBuildHandler new
          removeContentTransferEncoding: true;
          saveAttachmentsAsFiles: true;
          readFrom: input.
] on: AttachmentFilename
      do: [ :ex | "The suggested a FATFilename('mail-temp-files\budd.txt')  is
          replaced by  image\my-budd.txt "
          ex resume: 'my-', ex filename tail ].
message parts first contents
```

If the file name provided by the AttachmentFilename notification already exists, the framework raises an error, AttachmentFileExists. This error is resumable allowing to specify a new file name as a resumption parameter. If it is resumed without a parameter or the new filename from the parameter also exists the corresponding file will then be deleted and reused for the attachment. Obviously, if this exception is not resumed, the attachment will not be saved and parsing of the enclosing message containing this attachment will end here, unfinished. An example:

```
input :=
'From: zz@holcim.com
Content-Type: multipart/related;
      boundary="--11"

----11
Content-Type: text/plain; name="budd.txt"
Content-Disposition: attachment; filename="budd.txt"
Content-Transfer-Encoding: base64

QWxhZGRpbjpvcGVuIHNlc2FtZQ==

----11--
' .
[ [ message := MailBuildHandler new
          saveAttachmentsAsFiles: true;
          readFrom: input readStream.
      ] on: AttachmentFilename
          do:  [ :ex | ex resume: 'temp-', ex filename tail ].
] on: AttachmentFileExists do: [ :ex | ex resume: 'anotherTemp.txt' ]
```

Changes in handling of non-ascii characters in header fields

AR 52145: "I: [MIME] improve handling of invalid header contents"

NonASCIICharException has been renamed to NonASCIICharacter.

NonASCIIFieldParamException has been removed. This error was raised while parsing a message with non-ASCII characters in header field parameters. This case is now covered by NonASCIICharacter as well. Error handling for non-ASCII characters was also extended to provide an option to accept the characters without raising an exception (acceptNonAsciiCharacters:). By default NonASCIICharacter will be raised.

Here is an example message containing the copyright character, which is not a valid in ASCII character. By default the NonASCIICharacter will be raised:

```
bytes :=
'Subject: Cincom©
Content-Type: text/plain

1234
' asByteArrayEncoding: 'ISO-8859-1'.
    stream := EncodedStream on: bytes readStream encodedBy:
        (StreamEncoder new: #'ISO-8859-1').
[ MailBuildHandler readFrom: stream
    ] on: NonASCIICharacter
        do: [ :ex | ex parameter ].
```

To make the parsing process proceed accepting the character as is, set the acceptNonAsciiCharacters option to true:

```
stream := EncodedStream on: bytes readStream
    encodedBy: (StreamEncoder new: #'ISO-8859-1').
mess := MailBuildHandler new
    acceptNonAsciiCharacters: true;
    readFrom: stream.
```

To parse individual header fields with non-ASCII characters, a new, extended read method can be used:

```
string := 'Received: (from Cincom©)  Tue, 18 Apr 89 23:29:47 +0900'.
HeaderField readFrom: string
    readStream acceptNonAsciiCharacters: true.
```

The same default behavior, raising the NonASCIICharacter exception, applies to parsing individual fields as well:

```
[ HeaderField readFrom: string readStream
    ] on: NonASCIICharacter do: [ :ex | ex parameter ].
```

When constructing a new mail message, header field values with non-ASCII characters are accepted. They will be properly encoded using the specified encoding (ISO-8859-1 by default) and processed as per the MIME standard when the message is written out. Note that if the default encoding cannot handle provided characters a different encoding should be explicitly specified using the headerCharset: method. In the following example the ISO-8859-2 encoding is used to encode the accented Czech characters.

```
message := Net.MailMessage new
  from: '"Žlůva Tůma Řízek"<xx@mail.com>';
  to:  '"Božidar Šlapetko"<yy@mail.com>';
  subject: 'hello';
  text: 'text';
  headerCharset: 'iso-8859-2';
  yourself.
stream := String new writeStream.
message writeOn: stream.
stream contents
```

In general you can parse MIME messages from any kind of stream (internal or external), however, to facilitate parsing MIME messages straight from external streams the parsing machinery is set up to work from bytes (not characters) at the lowest level. This assumption causes certain complications when working with internal streams on strings because the underlying collections (internal streams on top of ByteArrays don't have any issues and behave exactly the same as external streams). These "character streams" are automatically wrapped in a DecodedStream which converts characters to bytes using ISO-8859-1 (that's the character set prescribed by the MIME standard). Because of this it is possible for the DecodedStream to encounter a character that it cannot encode. Although such a message is technically invalid (such characters should be encoded differently using valid characters), it is still possible to finish parsing the message. The stream is set up with a special stream error policy, the ReplaceUnsupportedCharacters policy. This policy signals an UnsupportedCharacterReplacement notification for these characters and by default replaces them with ASCII character NUL (code 0). This allows the MIME parsers to recover and continue. This notification can also be trapped by an application level handler and resumed with a different replacement character if so desired. The example below replaces the (invalid) trademark character with an underscore:

```
string :=
'Content-Disposition: attachment; filename="Cincom™.txt"
Content-Type: text/plain; name="Cincom™.txt"

some bytes
'.
[ MimeBuildHandler readFrom: string readStream.
    ] on: UnsupportedCharacterReplacement
      do: [ :ex | ex resume: $_  ].
```

Mail Archive Support

AR52115: "I: Implement parsing mail archives"

The mail archive/file support also had to be adjusted due to the new streaming changes in the core MIME framework. However it also created an opportunity for some cleanup and improvements.

Fundamental difference between reading messages from a file/archive and reading them from a socket stream is the expected lifetime of the source stream. In case of sockets and other transient sources you need to make a complete copy of the contents to be able to work with it at a later time. In case of persistent source like an external mail file, the assumption is that it will be there throughout the lifetime of the messages that were read from it. So copying would be unnecessary waste. Therefore, the mail archive support uses StreamSegments pointing back to the original source stream. Consequently, the source stream has to be a fully positionable stream, e.g., a file stream.

Previously the two strategies were controlled by a boolean option, useSourceStream, on the MailBuildHandler. However, because they are so radically different, it makes more sense to split them into separate handlers. The option was removed and instead there is now MailBuildHandler for the transient sources, and MailFileReader for the persistent ones (mail archives).

In this release, all the classes related specifically to processing of mail messages (MailFileReader, MailFileParser, MailMessage, etc) were moved to the Mail package. The MIME package now contains only the generic MIME aspects shared by both Mail and HTTP support.

The most significant change coming with the "streaming" approach is the way the message body source is constructed. It is now a stack of streams on top of the original body byte source. Each layer in the stack is responsible for some aspect of the decoding process (base-64, quoted-printable, character encoding etc). Body source contents will now always return fully decoded contents. There is no need to send removeTransferEncoding to the message anymore. For example:

```
input :=
'From: zz@holcim.com
Content-Type: multipart/related;
      boundary="--11"

----11
Content-Type: text/plain; name="budd.txt"
Content-Disposition: attachment; filename="budd.txt"
Content-Transfer-Encoding: base64

QWxhZGRpbjpvcGVuIHNlc2FtZQ==

----11--
' readStream.
message := MailFileReader readFrom: input.
message parts last contents.
```

We also added UnknownEncoding exception which is raised when message content-type specifies an unknown encoding (either because such encoding does not exist, or because corresponding encoder is not available in the image at that time). This exception can be resumed. It can be resumed with explicitly specified encoding, (e.g., resumeWith: #'US-ASCII') or resumed without parameter (resume) in which case the default encoding will be applied (ISO-8859-1). Here is an example (note the charset value in the Content-Type header field):

```
input :=
'From: Fred Foobar <foobar@Blurdybloop.COM>
Subject: afternoon meeting
To: mooch@owatagu.siam.edu
Content-Type: text/plain; charset=klingon

 Hello Joe, do you think we can meet at 3:30 tomorrow?
'.

"Default handling: if the exception is simply resumed, the message body
   source will use ISO8859-1"
[ message := MailFileReader readFrom: input readStream.
      ] on: UnknownEncoding
          do: [ :ex | ex resume  ].
message contents

"Overriding the default encoding: to force the message body source to use US-
   ASCII encoding"
[ message := MailFileReader readFrom: input readStream
      ] on: UnknownEncoding
          do: [ :ex | ex resume: #'US-ASCII' ].
```

Miscellaneous Cleanup

All the header field value setters will now consistently return the header field instances instead of the message entity. The methods that needed to change belonged mostly to MimeEntity:

- contentDisposition:
- contentDispositionSize:
- contentId:
- contentLength:
- contentTransferEncoding:
- contentType:
- mimeVersion:

For example:

```
part := MimeEntity newTextPlain.
(part contentDisposition: 'form-data')
    parameterAt: 'name' put: '23';
    fileName: 'mySystem.txt'.
```

Message parsing errors (invalid header fields) used to be reported in the Transcript. Now we raise an InvalidHeaderField notification instead.

Package comments and many class comments were revised and updated as well. The comments also contain a number of readily executable code samples that should help understand some of the aspects being discussed (see for example MailBuildHandler and MailFileReader).

Suppressing chunking for Http messages

52358: "[HTTP] add simpler way to suppress chunking"

In last release we made changes in the HTTP framework to support automatic chunking while writing Http messages. The only way for suppressing chunking was to make the chunk size big enough. In this release we have added shouldChunk option in HttpWriteHandler to simplify suppressing chunking. Messages are chunked by default, but this behavior can be controlled using the shouldChunk option in the following way:

- If shouldChunk is true (default)
    - if the message body size exceeds the size specified by the chunkSize option (defaultChunkSize is set to 4K), the message will be chunked;
    - if the message body size has fewer bytes than the specified chunkSize, the messages will not be chunked and will use the content-length header instead.
- If shouldChunk is false, the message will not be chunked regardless of body size and will use the content-length header instead.

Note that, when shouldChunk is false, the writer needs to be able to determine the exact, *final* byte size of the message body (i.e., if the body is to be compressed it has to be the compressed size). The size has to be known before it starts writing the body, so that it can inject the correct content-length field into the header. In general in this mode the body is first written into an internal stream to determine the correct byte count, then the header is finished with the right content-length and finally the body bytes are copied from the internal stream. As an optimization, if the body is simple (i.e. not multi-part) and the size of the body is known upfront, the writer will use that body size for the content-length field and then write the body bytes to the outging stream directly. This doesn't change the behavior in any way, it just may be useful to know that this particular case is handled in more efficient manner than the other non-chunked cases.

For how to control chunking, examine the examples in HttpWriteHandler class comments.

WebSupport

The Seaside project yielded some useful by-products, including improvements to the, so far rather Spartan, WebSupport package. The capabilities for submitting HTML form data were extended significantly, now supporting both simple “url encoded” format in a single-part HTTP request (content-type: application/x-www-form-urlencoded), or alternatively submitting each data entry as an individual part in a multipart HTTP request (content-type: multipart/form-data). These are now also available as extensions on bare HttpClient (and HttpRequest).

Multipart messages are used when form data contains entries with relatively large values, for example when a form has external files attached to it for upload to the server. More information about HTML forms can be found at http://www.w3.org/TR/html401/interact/forms.html#h-17.13.

The default behavior is to submit forms as simple requests. Form entries can be added individually using addFormKey:value: message, or set at once using formData: message which takes a collection of Associations. Note that formData: replaces any previous form content.

```
stream := String new writeStream.
(HttpRequest post: 'http://localhost/xx/ValueOfFoo')
    addFormKey: 'foo' value: 'bar';
    addFormKey: 'file'  value: 'myFile';
    writeOn: stream.
stream contents
```

An alternative way to post a form is through HttpClient. In this case the request is automatically executed and the result is the response from the server.

```
HttpClient new
    post: 'http://localhost/xx/ValueOfFoo'
    formData: (
        Array
            with: 'foo' -> 'bar';
            with:'file' -> 'myFile').
```

To force the form to submit as a multipart message, send beMultipart to the request at any point. Any previously added entries will be automatically converted to message parts. Note however that conversion of multipart messages back to simple messages is not supported, as it is not always possible without potentially losing information.

```
stream := String new writeStream.
(HttpRequest post: 'http://localhost/xx/ValueOfFoo')
    addFormKey: 'foo' value: 'bar';
    beMultipart;
    addFormKey: 'file'  value: 'myFile';
    writeOn: stream.
stream contents
```

File entries can be added using message addFormKey:filename:source:. Adding a file entry automatically forces the message to become multipart to be able to capture both the entry key and the filename.

```
stream := String new writeStream.
(HttpRequest post: 'http://localhost/xx/ValueOfFoo')
    addFormKey: 'foo' value: 'bar';
    addFormKey: 'text'  filename: 'text.txt' source: 'some text' readStream;
    writeOn: stream.
stream contents
```

Adding a file entry attempts to guess the appropriate Content-Type for that part from the filename extension. If it doesn't succeed the content type is set to default, i.e application/octet-stream. File names with non-ASCII characters will be automatically encoded using UTF8 encoding. UTF8 will also be used for the file contents if the source is a character stream (as opposed to byte stream).

Adding an entry to a multipart message returns the newly created part. That allows to modify any of the default settings or to add new ones. Here's an example changing the filename and file contents encoding to ISO8859-2:

```
stream := String new writeStream.
request := HttpRequest post:
  'http://localhost/xx/ValueOfFoo'.
part := request addFormKey: 'czech'
   filename: 'kůň.txt.txt'
   source: 'Příliš žluťoučký kůň úpěl ďábelské
     ódy.' withCRs readStream.
part headerCharset: #'iso-8859-2';
   charset: #'iso-8859-2'.
request writeOn: stream.
stream contents
```

There's also an API to parse messages containing forms in any of the supported forms. Just send #formData to the HTTP message. The result is a collection of associations, the same form as the input to the formData: message.

```
(HttpRequest post: 'http://localhost/xx/ValueOfFoo')
     addFormKey: 'foo' value: 'bar';
     addFormKey: 'file'  value: 'myFile';
     formData
```

File entry values will be entire message parts so that all the associated information can be accessed.

```
request := (HttpRequest post: 'http://localhost/xx/ValueOfFoo')
    addFormKey: 'foo' value: 'bar';
    addFormKey: 'text'  filename: 'text.txt' source: 'some text' readStream;
    yourself.
part := request formData last value.
part contents
```

Changed default cookie processing setting

AR 52053 changed the HttpProtocolInterpreter class method defaultEnableCookieProcessing from false to true. There are servers that send cookies in authentication or redirection responses. If the client request does not include these cookies, the server can stop communication. By setting this value to true, the default now allows the server to finish the client request.

GLORP

GLORP (Generic Lightweight Object-Relational Persistence) is an open-source project for mapping Smalltalk objects to and from relational databases. GLORP is now supported code.

GLORP is licensed under the LGPL(S), which is the Lesser GNU Public License with some additional explanation of how the authors consider those conditions to apply to Smalltalk. See the licensing information accompanying the software for more information.

Cincom currently plans to do a significant overhaul of the current database mapping facilities in Lens, using GLORP as one component of that overhaul.

Included on the CD is the GLORP library, its test suite, some rudimentary user-provided documentation, and some supplementary parcels. For more information, see the **glorp/** directory. Additional documentation is available at glorp.org.

WebSphere MQ Interface

Package MQ-XIF

Fix namespace of various MQ XIF classes

The MQ code as delivered with VisualWorks 7.5 had some defects. The following definitions should have been in namespace MQ, and now are:

- MQ.AbstractMQInterface
- MQ.MQTHAPIInterface
- MQ.MQTHAPIClientInterface
- MQ.MQInterface
- MQ.MQClientInterface

- MQ.AbstractMQInterface.DefaultStructs
- MQ.AbstractMQInterface.DefaultSizes

After moving the definitions to namespace MQ, all external interface dictionaries defined as #private import in VW75 are no longer private.

Fix invalid THAPI procedure definitions

THAPI interfaces were not properly defined. Decisive external procedures were omitting the "_threaded" keyword in the C pragma, so that external calls could not benefit from the threaded property. The procedures corrected are all from MQ.MQTHAPIInterface:

- MQCLOSE:with:with:with:with:
- MQDISC:with:with:
- MQOPEN:with:with:with:with:with:
- MQCONNX:with:with:with:with:
- MQCONN:with:with:with:

Now, all external procedures of THAPI MQ interfaces are marked as threaded.

Package MQ-Domain

Fixes

1. Fix prerequisite parcels: Should contain AT Profiling UI
2. Fix UHE #memberAt: when accessing a fresh AsynchronousMessage

   MQMessage>>messageType should take class based type if handle is not yet set
3. Fix MNU #contents sent to nil in attempt to access contents of a C pointer handle of an already closed queue manager

   - MessageQueue>>open:destinationQueueManager:
   - MessageQueue>>close
   - SenderQueue>>put:
   - ReceiverQueue>>getAnyMessageLike:

   These methods must access the queue manager handle carefully. If the queue or the queue manager has been closed by a concurrent process while another one is trying to open a queue, we should raise a MQNotConnectedError

4. Fix invalid handling of MQRC_HOBJ_ERROR during attempt to close a queue

   MessageQueue>>close must ignore this MQ exception. It originates from an obviously stale queue handle, and we cannot do anything about this.

5. Fix insufficient initialization in MQResult>>initialize

   If external interface is not set (which may happen during testing) it is ok to inline the known constant = 0 instead of leaving compCode nil.

6. Fix MQRC_CALL_IN_PROGRESS

   QueueManager>>disconnectExternal must be serialized to prevent concurrent processes from attempting to disconnect while another is still running an external call.

7. Fix QueueManager>>closeQueues (closing browse queues had been forgotten)

8. Fix ReceiverQueue>>getIncommingMessageWait: (use #ensure: to restore wait interval)

9. Fix invalid or inconsistent use of channel table mode

   RemoteQueueManager>>prepareConnectOptions must raise an exception if channel table should be used but prerequisites for channel table mode are not met. We use SystemNotification (Object>>notify:). Otherwise setup procedures can happily proceed without the chance of ever getting connected to MQ.

10. Fix invalid and incomplete implementation of message matching schemes

    Default matching scheme MQRO_COPY_MSG_ID_TO_CORREL_ID was effectively not supported. ReceiverQueue>>copyMatchFieldsFrom:to: had to be extended with code that puts an expected message id into the correl id of a query template.

11. Fix loss of known queues in queue manager

    QueueManager>>dismissInternal must not reset the queue dictionaries on disconnect. Queues must be kept for reopening after next connect.

12. Fix invalid length constant in access to #putDate

    Constant 20 was invalid, date field size is MQ_PUT_DATE_LENGTH (=8) only

13. Fix invalid length constant in access to #replyQueueManagerName

    Constant 60 was invalid, field size is MQ_Q_MGR_NAME_LENGTH (=48) only

14. Fix MQ.QueueManager>>connect

    Compare result of #acquireExternal with true, not nil; normally an exception prevents further processing after unsuccessful #acquireExternal, but when debugging it is possible to keep on going. The consequence is that the queue manager would be registered as successfully connected although it is not.

Review use of literal constants defined by external MQ interfaces

Clean up the code from use of constants defined by MQ XIF. Although Websphere MQ is a very mature product which is not very likely to be changed in constant definitions we prefer the use of “verbose” and self-explanatory expressions instead of literal constants, e.g. 48. A nice by-product of this step is the extinction of some flaws in improper use of constants (see “Fixes” section above).

- MQMessage initializeMessageDescriptor (48 => MQ_Q_NAME_LENGTH)
- MQMessage messageID (24 => MQ_MSG_ID_LENGTH)
- MQMessage messageGroupID (24 => MQ_GROUP_ID_LENGTH)
- MQMessage putDate (20 => MQ_PUT_DATE_LENGTH (=8))
- MQMessage setReplyQueueManagerName: (48 => MQ_Q_MGR_NAME_LENGTH)
- MQMessage setReplyQueueName: (48 => MQ_Q_NAME_LENGTH)
- MQMessage correlationID (24 => MQ_CORREL_ID_LENGTH)
- MQMessage userIdentifier (12 => MQ_USER_ID_LENGTH)
- QueueManager name: (48 => MQ_Q_NAME_LENGTH)
- ActionMessage replyQueueName (48 => MQ_Q_NAME_LENGTH)

Extensions

This section lists methods that have been added.

MQMessage putTime

Answer the contents of MQ message field #PutTime

MQMessage reportOptionBits

Answer the report options mask from the MQ structure

MQResult isConnectionFailure
true if result is one of the major MQ connection errors

MQResult isCallInProgress
true on reason code MQRC_CALL_IN_PROGRESS

MQResult beFailed
initialize with completion code MQCC_FAILED

ActionMessage requiresPassCorrelId
true if message configured for copying its Correl ID to the Correl ID field of a reply

ActionMessage requiresMsgIdToCorrelId
true if message configured for copying its MsgID to Correl ID field of a reply

Removals

This section lists methods that have been removed.

- MQResult queueManager {accessing}
- MQResult isAtEnd {testing}
- MQResult isInError {testing}
- QueueManager releaseQueues {initialize-release}

Seaside Support

The Seaside web application framework is supported in 7.6.

What is Seaside?

Seaside is a framework for developing sophisticated web applications in Smalltalk. It is an open source, cross dialect project hosted at http://www.seaside.st.

Seaside provides a layered set of abstractions over HTTP and HTML that let you build highly interactive web applications quickly, reusably and maintainably. It is based on Smalltalk, a proven and robust language that is implemented by different vendors.

Seaside includes:

- Programmatic HTML generation. A lot of markup is boilerplate: the same patterns of lists, links, forms and tables show up on page after page. Seaside has a rich API for generating HTML that lets you abstract these patterns into convenient methods rather than pasting the same sequence of tags into templates every time.

- Callback-based request handling. Why should you have to come up with a unique name for every link and form input on your page, only to extract them from the URL and request fields later? Seaside automates this process by letting you associate blocks, not names, with inputs and links, so you can think about objects and methods instead of ids and strings.
- Embedded components. Stop thinking a whole page at a time; Seaside lets you build your UI as a tree of individual, stateful component objects, each encapsulating a small part of a page. Often, these can be used over and over again, within and between applications - nearly every application, for example, needs a way to present a batched list of search results, or a table with sortable columns, and Seaside includes components for these out of the box.
- Modal session management. What if you could express a complex, multi-page workflow in a single method? Unlike servlet models which require a separate handler for each page or request, Seaside models an entire user session as a continuous piece of code, with natural, linear control flow. In Seaside, components can call and return to each other like subroutines; string a few of those calls together in a method, just as if you were using console I/O or opening modal dialog boxes, and you have a workflow. And yes, the back button will still work.

Seaside also has good support for CSS and Javascript, excellent web-based development tools and debugging support, a rich configuration and preferences framework, and more.

Seaside in VisualWorks

With this release Seaside becomes a fully supported component of Cincom Smalltalk, including any of the additional components provided in the seaside/ directory of the installation. There are also additional, contributed components, ones that were developed by the Seaside community and are provided for convenience. Those can be found in the **`contributed/Seaside_Components`** directory. However, the contributed components are not supported by Cincom.

The Seaside package includes an almost unmodified version of the latest Seaside release (2.8), an integration layer setting Seaside up on top of the Opentalk based HTTP server and some development enhancements aimed to help new users get up and running quickly. For example, a Seaside server is started automatically as soon as the Seaside package is loaded and a web browser opens on the front page served by the server. There is also a Seaside menu added to the VisualLauncher providing easy access to Seaside related functions and also a new settings domain for easy management of most important parameters of a Seaside server. The Opentalk integration also provides few enhancements, e.g. WAExternalFileLibrary which provides efficient serving of external files directly by the Opentalk HTTP server.

Running Seaside with Object Studio 8

Seaside is also supported with Object Studio 8. However, there are few issues that users need to be aware of.

In order to avoid modifying Seaside as much as possible, it is running on top of a Squeak compatibility layer, which is loaded as a prerequisite. There are some some semantic conflicts between some of the methods overriden by the Squeak compatibility layer and their behavior in plain Object Studio 8. We will try to reduce the number of these differences in subsequent releases, but for now be aware that you might run into this kind of issue with Seaside loaded. Here are a few behavioral changes that we are currently aware of, that users are likely to run into with Seaside loaded:

- Character>>to: in Squeak returns a ByteString instead of an Interval.
- With Seaside loaded, date parsing (methods String>>asDate or Date class>>fromString:) reverts to the VisualWorks mechanism, which expects different input formatting than the Object Studio parsing does. Notably, a string like '2012-4-10' will not parse for OS8 when Seaside is loaded. Format rules are specified in the class Locale.
- Time parsing (methods String>>asTime and Time class>>fromString:) uses VisualWorks mechanisms as well. The comments from the previous point apply to Time parsing as well.

Add-On Components

Seaside-Resources

Provides a way to attach related information to a Seaside session (a locale, a database session, etc). Moreover a session will notify all its associated resources about important events through a set of callbacks that the resources may choose to act on. WASessionResource can be subclassed or serve as a template for all available callbacks.

A root component can choose to use a session with resources by configuring their session preference accordingly, customarily in a class side initialize method of the component class:

```
(self registerAsApplication: 'ApplicationName')
    preferenceAt: #sessionClass put: Seaside.WAResourcefulSession
```

Seaside-I18N

Provides a session resource capturing session's locale, named WALocale. A session locale information can be accessed using locale and locale: accessors. When a locale is set on a session, the session will also invoke SeasidePlatformSupport class>>setLocale: for every incoming request allowing to perform any platform specific locale setup before executing the request. For example in VisualWorks this updates the locale of the process executing the request.

Note however that session locale does not get set automatically. There are different strategies that an application may choose to employ to set the locale. Here is a sketch of a component that lets the user choose a locale for their session:

```
renderContentOn: html

    html form: [ | session |
        session := self session.
        html heading
                level: 1;
                with: #HelloWorld << #L10NTest >> 'Hello World!'.
        html div: [ html text: 'Current Locale: ', Locale current languageID, ' ',
  session resources printString  ].
        html select
            id: 'locales';
            list: (Locale availableLocales collect: [ :id | id copyUpTo: $. ]) asSet;
            selected: (session locale ifNotNil: [ :locale | locale id asString ]);
            callback: [ :value | session locale: value ].
        html submitButton text: 'Set Locale' ]
```

The component displays a UserMessage that will get translated based on current session locale when it is converted to a string. It also shows a drop down list of available locales that the user can choose from. When the choice is submitted the callback simply sets the chosen locale on the session.

Preview Components

A few additional components are included as previews.

Seaside-SUnitToo

A framework on top of SUnitToo for unit testing of Seaside components/tasks in the same manner as they run on the web. TestCase is automatically configured with a preinitialized component and a UserAgent representing a browser accessing the component. During test setUp the component gets registered with the Seaside server, and the agent hits the first page of the component. Therefore by the time the test method starts the first page is already available for inspection. A Page captures not just the response, but also its URL, the parsed DOM tree of the contents and a copy of the component (from the corresponding continuation) in the state that produced the page. XPath can be used to analyze and validate the parsed response contents (see Page>>find:).

Let's build a simple test case for the WACounter demo. We need to create a subclass of SeasideSUnitToo.TestCase, e.g. WACounterTest and specify the class of the component to test.

```
WACounterTest>>componentClass
	^WACounter
```

And here's an example test checking the function of the ++ link:

```
testIncrease
	"check current state of the component"
	self assert: self component count isZero.

	"find a link with text '++' on current page and simulate a click, i.e.,
	do a GET of the associated href"
	self clickLinkWithText: '++'.

	"check current value of the counter"
	self assert: self component count = 1.

	"find the heading element on current page and make sure the text
	corresponds to current counter value"
	self assert: (self find: '//h1/text()') first text = '1'
```

The agent caches the returned pages the same way a web browser would. It allows to look back and forward in the page history. It works the same way as web browser back/forward buttons, so currentPage does not have to be the last one in the history. Here's another sample test exercising the continuation mechanism in the face of the back button.

```
testClickBackClick
    "initial state"
    self assert: self component count = 0.

    "hit the increment link"
    self clickLinkWithText: '++'.

    "check new state"
    self assert: self component count = 1.

    "click the back button"
    self back.

    "current page should display 0"
    self assert: (self find: '//h1/text()') first text = '0'.

    "the snapshot of the component when it generated current page should
    have count 0"
    self assert: self pageComponent count = 0.

    "click incremement"
    self clickLinkWithText: '++'.

    "check new state of the component"
    self assert: self component count = 1.
```

Since both the client and the server live in the same image the agent also provides access to all interesting participants of the test: the instantiated component, associated application and session and also the HttpClient instance used to hit the server. The server used for testing is the shared variable SeasideServer class>>current. Tested components get automatically registered with the server during test setUp (under the componentTestingPath) and unregistered in test tearDown.

Finally, since most non-trivial web applications will likely use form posts, there's also the HtmlForm. It provides a number of helper methods emulating various types of form input.

```
testFormPost

    self newForm
        select: #( 'Dakar' 'Sydney' ) in: 'multi';"select in a multi-selection list"
        fill: 'Hello' in: 'input';"fill in an input field"
        click: 'Submit'"click the submit button"
```

The click: also performs the form post, so it needs to be last in the sequence. If a form doesn't have a submit button, HtmlForm>>post can be called directly as well. It is also possible to emulate file upload. Upload posts automatically as well.

```
testFileUpload

    self newForm upload: 'test.txt' in: 'upload'
        source: 'Hello World!' readStream.
```

Seaside-Glorp

This package is the connecting code for Seaside and Glorp. The key piece is GlorpResource which automatically associates a database session (Glorp) with a Seaside session, for the duration of the Seaside session. This, combined with the GlorpActiveRecord and WriteBarriers (which are automatically loaded as prerequisites), adds persistency for Seaside applications that is very simple to use.

Let's build a simple blog server as an example. The class Post will have a title, content and the time created attributes. To inherit the ActiveRecord functionality it will subclass from ActiveRecord. There are no special methods needed on the class beyond simple instance variable accessors for the attributes mentioned above.

Another required piece is an ActiveRecordDescriptorSystem subclass, let's call it BlogSystem. The only method that we will need is the following:

```
BlogSystem>>tableForPOSTS: aTable
    (aTable createFieldNamed: 'id' type: platform serial) bePrimaryKey.
    aTable createFieldNamed: 'title' type: (platform varchar: 200).
    aTable createFieldNamed: 'content' type: platform text.
    aTable createFieldNamed: 'created' type: platform timestamp
```

This method describes the structure of the database table that will be used to store the Post objects. The mapping between the objects and the table rows is derived automatically based on naming convention that are very much like the ones used by Ruby on Rails. With this code in place we can already persist some Post objects in the database. However we need to first provide the database login information before we can do that. Login can be specified via Settings (look for the Database node there), or through code like this:

```
        ActiveRecord defaultLogin: (
            Glorp.Login new
                database: PostgreSQLPlatform new;
                connectString:  'localhost_demo';
                username:  'user';
                password: '***';
                yourself)
```

With that and the database in place, we can create the database tables with a simple expression.

```
    BlogSystem recreateTables
```

And we can start storing objects

```
        1 to: 5 do: [ :i |
            Post new
                title: 'Factorial ', i printString;
                content: i factorial printString;
                created: Timestamp now;
                save ]
```

We can read the objects back in very easily as well.

```
    Post findAll
```

Now let's try to display the main blog page. We will need a WAComponent, let's call it Blog, configured with the GlorpResource.

```
    Blog class>>initialize
        | application |
        application := self registerAsApplication: 'Blog'.
        application preferenceAt: #sessionClass put:
      Seaside.WAResourcefulSession.
        application configuration addAncestor:
            SeasideGlorp.GlorpConfiguration new.
```

For the main page let's simply render all the posts.

```
    Blog>>renderContentOn: html

        html heading: 'Blog'.
        Post findAll
            do: [ :post |
                    html heading level: 2; with: post title.
                    html paragraph: post content.
                  separatedBy: [ html break ]
```

To demonstrate how to update objects, let's add an 'Edit' link to each post that will open an EditPost component.

```
Blog>>renderContentOn: html

    html heading: 'Blog'.
    Post findAll
        do: [ :post |
                html heading level: 2; with: post title.
                html paragraph: post content.
                html anchor callback: [ self call: (EditPost new post: post) ];
  with: 'Edit' ]
        separatedBy: [ html break ]
```

EditPost will use a form with simple input fields for title and content and Save/Cancel buttons.

```
EditPost>>renderContentOn: html

    html form: [
        html textInput text: post title; callback: [ :title | post title: title ].
        html textInput text: post content; callback: [ :text | post content: text ].
        html button callback: [ post save. self answer ]; with: 'Save'.
        html button callback: [ post session rollbackUnitOfWork. self answer ];
  with: 'Cancel' ]
```

Note that the Cancel action invokes a rollback, which will undo any other changes made in the form. We can use the same EditPost component to submit new posts as well.

```
Blog>>renderContentOn: html

    html heading: 'Blog'.
    html anchor callback: [ self call: (EditPost new post: (Post new created:
  Timestamp now; yourself)) ]; with: 'New Post'.
    Post findAll
        do: [ :post |
                html heading level: 2; with: post title.
                html paragraph: post content.
                html anchor callback: [ self call: (EditPost new post: post) ];
  with: 'Edit' ]
        separatedBy: [ html break ]
```

And finally, we can add a Delete link next to each Edit link to allow removing Posts as well.

```
Blog>>renderContentOn: html

    html heading: 'Blog'.
    html anchor callback: [ self call: (EditPost new post: (Post new
        created: Timestamp now; yourself)) ]; with: 'New Post'.
    (Post findAll sorted: [ :a :b | a id > b id ])
        do: [ :post |
            html heading level: 2; with: post title.
            html paragraph: post content.
            html anchor callback: [ self call: (EditPost new post: post) ];
                with: 'Edit'.
            html anchor callback: [ post delete ]; with: 'Delete' ]
        separatedBy: [ html break ]
```

To better manage database resources, this package also provides transparent support for database connection pooling. This functionality is based on work by Ramon Leon.

Opentalk

Opentalk-Component-Base removed

We've had OpentalkSystem for a few releases now, along with its global registry of brokers. The only reason why Opentalk-Component-Base stuck around was because the two Opentalk tools (Console and Monitor) were still relying on the manually managed ComponentHolder from that package. In this release we retargeted the tools onto the automatically managed registry in the OpentalkSystem and were therefore able to remove the ComponentHolder package altogether. This also makes the tools a bit easier to use since you don't need to add/remove brokers manually anymore.

WebToolkit

The WebSiteConfiguration object model within the WebToolkit parcel has been significantly refactored to make configuring a headless development image and/or production runtime image easier and more robust. A development image no longer automatically configures itself as an application server. You can manually configure the Application Server from the Web menu in a development image, or it will lazy initialize (as the runtime image always did) when the first request is received at the server. You can also use the Web menu to clear the configuration from your development image.

This change eliminates the soft hooks back to the development environment that often annoyingly remained in a runtime image. It also means that you can create a headless development image, or for that matter a headful development image, which will start and configure itself exactly as a production runtime image does. This makes it much easier to test your server application.

As always, you must save your image containing your desired IPWebListener or Smalltalk HTTP Server instances, or construct your application to create these servers before the first request arrives to be served if you are relying on the Application Server's lazy initialization.

The biggest behavioral impact of this change is due to the fact that Application Server configuration relies on the parent/child relationship between the global configuration and the web sites in the configuration's sites collection. You can no longer configure a WebSite before it is added to the configuration's sites collection. The basic WebToolkit behavior accounts for this change, but if you have built applications that rely on WebToolkit internals, in particular the class side API of WebSite or WebConfigurationManager, you will need to update your application to work with this new version of the WebToolkit parcel.

The class side behavior of WebSite to manage the global configuration has been moved to WebSiteConfiguration, which now caches its singleton instance as the global configuration.

The WebConfigurationManager class has been removed. The behavior relating to configuration and event interception has been moved to WebSiteConfiguration. A few things that were possible under the old configuration scheme are no longer available globally on the class side and must be performed on the global configuration instance instead. The behavior relating to logging has been moved to a new class, WebSiteLoggingManager, without changing any of the functionality.

The WebSiteConfigurator class has been renamed WebSiteConfigurationPages which is more representative of the fact that this class is essentially a servlet that provides the content for the Application Server administrator's Configuration Pages.

There is also a new WebToolkit subsystem, VWAppServerSystem, which handles all the system start, snapshot and quit events. In addition it exposes properties for a new settings page in the Web Settings section where you can install the name of the primary global configuration file (where Application Server configuration starts), and where you can allow/disallow the use of a new parameter, -wtkcnf, to set the configuration file name from the command line.

In a development image, the Web menu now has a new option (Clear Configuration) to explicitly clear the cached Application Server configuration. This is especially useful when using a headful development image to test your server. Clear the configuration in order to simulate the startup sequence of a headless runtime server image.

Due to the large scale of the changes in this release, we have provided a compatibility parcel, WebToolkit-Compatibility-API which can be loaded into your image along with the new WebToolkit parcel. This compatibility parcel implements placeholders for most of the obsolete class side API of WebSite and WebConfigurationManager, and should assist you in upgrading your current applications.

Extend support for subclassing HttpApplication, HttpSession, PageModel

There is now support for subclassing HttpApplication to provide your own desired Web Application behavior, or to facilitate hosting multiple Web Applications within a single VisualWorks Application Server. In addition, you can subclass HttpSession and/or PageModel to provide additional session management or page compilation support, as required, for your application classes.

The behavior and global data stored in the contents dictionary of your HttpApplication subclass is available application-wide, as are the session behavior and session variables along with your extensions to page compilation behavior.

Each of these subclasses can be specified in the Application Server's configuration files. When specified in the global configuration file, they apply to all sites on the Server. If you wish to enable site-specific application, session or page compilation behavior, specify your subclass in the relevant site configuration file. An applicationClass, sessionClass or PageModelClass defined in a site-specific configuration file takes precedence over a class defined in the global configuration file. Place these definitions in the **`[configuration]`** section of the INI file, using the following syntax :

```
applicationClass = VisualWave.MyApplication
sessionClass = VisualWave.MySession
pageModelClass = VisualWave.MyPageModel
```

Note that because of the way asQualifiedReference works, a dotted name must be used to specify your subclass.

Implement reset for buffered responses

Following the introduction of response buffering in the previous release, you can now clear the response buffer after starting to write to it. This can be particularly useful when it becomes necessary to forward a request after some page content has already been written to the buffer.

DLLCC

External-Interface-Pragmas Package

This package seeks to solve the long standing problem of having to use a class definition override to add libraries for an ExternalInterface subclass to use. It also seeks to provide a simpler way of specifying different libraries for different platforms.

Consider the following method added to an ExternalInterface:

```
unixLibrary
    <library: #linux>
    <library: #solaris>

    ^'libcairo.so'
```

The method returns a string specifying the library name. The <library:> tags indicate for which platforms this method should be executed. The arguments that show up in the <library:> tag are simplified forms of the platformMoniker. See ExternalInterface class>>knownPlatformIDs for a list.

The method name itself is arbitrary. Using this facility an extension package can add libraries to search to an external interface. Or add support for more libraries on other platforms.

There may be some refinement of this feature in the next release.

Compression

The Compression-Zip parcel provides facilities for reading file archives in zip format that have been compressed using the deflate method (the default). An archive is represented by an instance of Archive containing a collection of ArchiveEntries describing the compressed files. In general the contents of a file can be accessed through a decompression stream. The right kind of stream can be obtained from an ArchiveEntry using the readStream message. Archives, entries and decompression streams also have potentially interesting attributes associated with them, for example compressedSize, uncompressedSize, fileName, lastModificationDate, etc. See the accessing protocol on corresponding classes for a full list of available attributes.

Here is an example of how to get contents of all files in an archive:

```
archive := OS.Zip.Archive filename: 'archive.zip'.
[ archive entries collect: [ :each | each contents ] ] ensure:
    [ archive close ]
```

Here is how to access contents of the last file in an archive as a stream:

```
archive := OS.Zip.Archive filename: 'archive.zip'.
[ archive entries last readStream upToEnd ] ensure: [ archive close ]
```

Both archives and entries understand message extractTo: which takes a directory (String or Filename) as an argument, and extract the entire contents of the receiver into that directory.

```
(OS.Zip.Archive filename: 'archive.zip') extractTo:
    '.'  "The archive is closed automatically."

archive := OS.Zip.Archive filename: 'archive.zip'.
[ archive entries last extractTo: '.' ] ensure: [ archive close ]
```

The directory structures embedded in the archive will be respected and reconstructed as needed.

Note that only the basic (commonly used) archiving capabilities are supported, none of the extended features like 64-bit extensions, encryption, multiple volumes, etc are available.

For details about the zip format, see

http://www.pkware.com/documents/casestudies/APPNOTE.TXT

Documentation

This section provides a summary of the main documentation changes.

Basic Libraries Guide

General updates and corrections.

Tool Guide

No changes

Application Developer's Guide

General updates and corrections.

COM Connect Guide

No changes

Database Application Developer’s Guide

Major update, edit, and reoranization, including new examples

DLL and C Connect Guide

Several updates, particularly in chapter 8, “Platform Specific Information.” This document needs a comprehensive update and edit, so some information may still be out of date.

DotNETConnect User’s Guide

No changes

DST Application Developer’s Guide

No changes

GUI Developer’s Guide

Minor updates

Internationalization Guide

No changes

Internet Client Developer’s Guide

Significant updates and corrections, particularly relating to refactorings and improvments to MIME and mail support.

Opentalk Communication Layer Developer's Guide

No changes

Plugin Developer’s Guide

No changes

Security Guide

Minor updates and corrections, particularly in sections on ASN.1.

Source Code Management Guide

No changes

Walk Through

No changes

Web Application Developer’s Guide

Minor updates

Web GUI Developer’s Guide

No changes

Web Server Configuration Guide

Minor updates

Web Service Developer’s Guide

No changse

3

Deprecated Features

By deprecating certain features, we remove them from the system. These are made available for a limited time as parcels in the **`obsolete/`** directory, to provide you the opportunity to port applications away from using the features before they are removed altogether. This directory is on the default parcel path.

Virtual Machine

Thunking DLLs

Because the relevant Windows platforms are no longer supported (95/98/ME), the vwft32.dll and vwft16.dll files are deprecated. They are included with this release, but will be removed from the next release.

MacOS 9

7.6 is the last release to support MacOS 9.

User Interface

InputSensor

mousePointFor: and mousePointForEvent: are obsolete and will be removed from a future release. Use cursorPointFor: instead, for both of these.

Notebook Widget

The Notebook widget is being deprecated, and will be removed in the next major release. We suggest using the Tab Control widget in its place.

4

Preview Components

Several features are included in a **`preview/`** and available on a "beta test," or even pre-beta test, basis, allowing us to provide early access to forthcoming features. Several are described in the following sections. Browse the directory contents for last minute inclusions.

Universal Start Up Script (for Unix based platforms)

This release includes a preview of new VW loader that runs on all Unix and Linux platforms. This loader selects the correct object engine for an image, based on the image version stamp. Formerly, the only loader of this sort was for Windows.

The loader consists of two files and a readme in **`preview/bin`**. Installation and configuration information is provided in the readme.

This loader is written as a standard shell script which allows it to be used to launch VW on virtually any Unix based platform. This opens up the possibility of having a centrally managed site-wide installation of an arbitrary set of VW versions allowing users to simply run their images as executables without any user specific setup required. The loader figures out which version of VW and which specific VM is needed to run the image being launched, using information provided in the INI file).

For installations using only final releases (not development build releases), a single entry line in the INI file for each VW version will suffice to serve any Unix based platform for which a VM is available at the specified location.

Base Image for Packaging

`preview/packaging/base.im` is a new image file to be used for deployment. This image does not include any of the standard VisualWorks programming tools loaded. The image is intended for use as a starting point into which you load deployment parcels. Then strip the image with the runtime packager, as usual.

64-bit Image Conversion

The ImageWriter parcel is still capable of converting arbitrary 32- bit images to 64-bit images. However, due to an unresolved race condition, occasionally it may create an image that brings up one or more error windows. These windows can safely be closed, and if the 64- bit image is saved again, they will not return.

However, they may be problematic in a headless image or an image that has been produced by the Runtime Packager. For such cases, re-saving or recreating the original 32-bit image, and then converting it again may avoid the race condition. Alternatively, converting the image to 64 bits before applying the Runtime Packager or making the image headless may also be helpful.

ImageWriter empties all instances of HandleRegistry or its subclasses. Since these classes have traditionally been used to register objects which must be discarded on startup, emptying them during the image write is safe. But if your code is using HandleRegistry or a subclass to hold objects which are intended to survive across snapshots, ImageWriter may disrupt your code. Running ImageWriter before initializing your registries may solve this problem. We would also like to know more about how you use HandleRegistry, in order to improve ImageWriter's ability to transform images without breaking them.

Tools

With the Trippy basic inspector now being much more robust, work was done on integrating this with the debugger. It has not been finished yet, but may be loaded. At this writing, this causes the inspectors located in the bottom of the debugger (the receiver and context fields inspectors) to be basic trippy inspectors (not the entire diving/previewing inspector, just the basic tab). This makes operations between the two the same, and provides the icon feedback in the debugger. The stack list of the debugger also shows the receiver type icons.

Smalltalk Archives

Even though Smalltalk Archives are a supported feature of ObjectStudio 8, they are supported only in preview for VisualWorks at this time.

A Smalltalk Archive is a file containing a collection of parcels, compressed (using tar). The archive specifies a load order for the parcels, and supports override behavior.

Unlike publishing a bundle as a parcel, a Smalltalk Archive preserves package (and parcel) override behavior and package load order. Accordingly, Smalltalk Archives are a good alternative to publishing a bundle, in some circumstances.

Smalltalk archive files have a .store filename extension.

To load an archive, use the File Browser to locate the file, then right-click on it and select **Load**.

The archive loads with the parcel and bundle structure it had when it was saved. Database links might or might not be preserved, depending on the settings at the time it was saved.

To publish a Smalltalk Archive, select the bundle (or package) in the System Brower, the select **Package → Publish as Parcel…** The options in the publish dialog are the standard publish options, except for the Store options section. Because you want to save as a Smalltalk Archive, leave that checkbox selected. If you are using the Smalltalk Archive for deployment, and so do not need the database links, uncheck the **With database links checkbox**; otherwise, leave it checked.

The archive is published, by default, in the current directory, typically the image directory. The file name is the bundle name with a .store filename extension.

WriteBarriers

The immutability mechanism in VW can be used to detect any attempts to modify an object. All it takes is marking the object as immutable and hooking into the code raising the NoModificationError. The ability to track changes to objects can be useful for number of different purposes, e.g., transparent database updates for persistent objects, change logging, debugging, etc. While the mechanism itself is relatively simple, it is difficult to share it as is between multiple independently developed frameworks.

WriteBarriers (loadable from **`preview/parcels/WriteBarriers.pcl`**) allow multiple frameworks to monitor immutability exceptions at the same time. This framework makes object change tracking pluggable through subclasses of Tracker. A Tracker must implement a couple of methods:

isTracking: *anObject*
Answer true if the tracker is tracking *anObject*

privateTrack: *anObject*
Register and remember *anObject*`

privateUntrack: *anObject*
Forget about the object you were tracking

applyModificationTo: *anObject* **selector:** *selector* **index:** *index* **value:** *value*
We've accepted that we are tracking the object and a change has been made to it, what do we want to do? The default behavior is to apply the change to the object.

Note that the framework does not provide a mechanism for keeping track of which objects each tracker is managing. Instead, it leaves the options open. Frameworks may already have a registry of objects that they want to track (e.g., a persistency framework will likely cache all persitent objects to maintain their identity) in which case a separate registry for the corresponding tracker would waste memory unnecessarily.

A deliberate limitation of WriteBarriers is that they will refuse to track any previously immutable objects. Trackers can decide to not apply the modification and emulate the original immutability that way, and refusing to track immutable objects reduces complexity of the solution.

Here's a sketch of a tracker that will announce a Modified announcement for any modification that occurs. Let's assume that this AnnouncingTracker will have its own registry for tracked objects in the form of an IdentitySet (inst var. objects). The first three required methods are fairly obvious:

```
privateTrack: anObject
    objects add: anObject

privateUntrack: anObject
    objects remove: anObject ifAbsent: []

isTracking: anObject
    ^objects includes: anObject
```

The modification callback needs to call super so that the modification is actually applied, but in addition makes the announcement as well. Note that Tracker subclasses Announcer to make Announcement use easy.

```
applyModificationTo: anObject selector: selector index: index value: value

    super applyModificationTo: anObject selector: selector
        index: index value: value.
    self announce: (Modified subject: anObject)
```

To make use of the tracker, it has to be instantiated, which automatically registers it in the global registry, Tracker.Trackers. Any objects to be tracked by it have to be explicitly registered with it using the #track: message.

```
tracker := AnnouncingTracker new.
tracker when: Modified do: [ :ann | Transcript space; print: ann subject ]
string := 'Hello' copy.
tracker track: string.
string at: 1 put: $Y.
```

The last statement will trigger the Transcript logging block. To stop tracking an object use the #untrack: message.

```
tracker untrack: string
```

And to deactivate the tracker altogether use the #release message.

```
tracker release
```

Sparing Scrollbars

Sparing Scrollbars extends VisualWorks widgets to optionally set scrollbars to be dynamic and only appear when necessary. When loaded on top of the UIPainter, the dataset, list, table, tree, text editor, hierarchical view, or view holder widgets, or the window itself may be specified to use this feature in the **Details** page of the UIPainter tool. Sparing Scrollbars includes some test and sample classes which demonstrate the usage of this new behavior.

The Sparing Scrollbars component can be loaded from the parcel **`preview/SparingScrollbars.pcl`**.

Multithreaded COM

Composed of a DLL and three parcels, the multithreaded COM option for COM Connect introduces the ability to perform non-blocking COM calls in VisualWorks. This improves responsiveness of COM servers implemented in VisualWorks using COM Connect. It currently operates for VisualWorks versions 7.3 to 7.6 on Window XP platforms with 32Bit Intel Architecture.

This option changes the threading model of an existing VisualWorks to free-threaded, but synchronization is still performed via VisualWorks DLLCC mechanisms. Threaded COM call-outs and all in-bound calls will be routed through the multithreaded COM interface. Non-threaded callouts will still go through the original COM primitives provided by the VisualWorks virtual machine.

Additional notes and usage instructions appear in **`/preview/multithreaded com/DLLCOM.pdf`**.

COM User Defined Type (UDT) Support

Prior to this release, COM Connect was unable to call COM functions that use user-defined parameter types (structures). The COM UDT Support preview component adds support for such data types for 7.6.

The COM UDT support component consists the parcel files and a PDF file in **`preview/com udt support/`**.

Unicode Support for Windows

Extended support for Unicode character sets is provided as a preview, on *Windows 2000 and later* platforms. Support is restricted to the character sets that Windows supports.

The parcels provide support for copying via clipboard (the whole character set), and for displaying more than 33,000 different characters, without any special locales.

The workspace included in **`preview/unicode/unicode.ws`** is provided for testing character display, and displays the entire character set found in Arial Unicode MS.

First, open the workspace; you'll see a lot of black rectangles. Then load **`preview/unicode/AllEncodings.pcl`** and instantly the workspace will update to display all the unicode characters that you have loaded. You can copy and paste text, for example from MS Word to VW, without problems.

If there are still black rectangles, you need to load Windows support for the character sets. In the Windows control panel, open **Regional and Language Options**. (Instructions are for Windows XP; other versions may differ slightly.) Check the **Supplemental language support** options you want to install, and click **OK**. The additional characters will then be installed.

To write these characters using a Input Method Editor (IME) pad, load the **`UnicodeCharacterInput.pcl`**.

Scripting and Command Line Support

This release includes a preview support for improved scripting and command-line driven operation. It has been possible to run VisualWorks from the command line using options like **`-doit`** for some time, but these were not always as convenient for script type of operations as they might be.

The parcel ScriptingSupport adds a number of enhancements to support this type of operation. If this parcel is loaded, then basic usage would be something like

```
visual scripting.im scriptfile.st
```

where **`scriptfile.st`** contains a sequence of Smalltalk code. It does not need to be file-out format. On Windows, it is desirable to use the console VM, so that standard in/out/error are available.

You can also load the ScriptingSupport parcel as part of the command line. If you do this, you probably want to specify **`-nogui`** so that the image runs headless.

```
vwntconsole.exe visual.im -nogui -pcl ScriptingSupport scriptFile.st
```

When running from a script file, anything on the command line following the name of the script file are treated as arguments, and are bound into the `arguments` variable visible to the script.

You can also run commands directly from the console, using command-line options more similar to those found in traditional scripting languages, e.g.

```
visual scripting.im -e "MyServer startUp"]
```

Using the **`-i`** option will give an interactive command shell for evaluating Smalltalk expressions.

All of these commands will run in a slightly different environment than normal Smalltalk execution. The environment behaves more like a Workspace, where, by default, all of the name spaces in the image are visible. Ambiguous expressions will be resolved to choose the first name that matches. There is also an experimental extension operator, **`&`**, which makes it easier to assemble collections when writing short scripts.

All of the script code is run as a `doit` in an object called a `ScriptRunner`. This supports some additional methods for working with scripts. The `include:` method reads in another file as a script. The `use:` method loads a parcel with a given name. The `print:` method prints an object to standard out. Note that output is to the stdout stream. The Transcript is suppressed, because many of the things that appear on the Transcript are unsuitable for printing in a command-line script.

For a full list of the command line options available, run the image with the **`-h`** parameter. For more information on the capabilities that this adds, see the ScriptingSupport package.

Grid

A Grid widget combines elements of the Table and Dataset widgets for a simpler and more flexible interface of viewing and editing tabulated forms. This release includes a preview of a new Grid, based on the Grid from the Widgetry project. It currently supports the following features:

- Multiple row sort by column with or without a UI
- Multiple and single selection options by row or individual cell
- Interactive row or column resize
- Scroll and align column, row, or cell to a particular pane position (e.g., center, left, right top, bottom)
- UIBuilder canvas support

Planned features include:

- A `SelectionInGrid` model. Currently one may directly access, add, remove, and change elements of the Grid. Direct access will always be available.
- Drag-and-drop rows or columns to add, remove, or sort elements
- A tree column

- Completion of announcements, trigger events, or callbacks
- Specific OS look support for column headers. Currently only a default look is supported.
- The column and row headers may be set to not scroll with the Grid.

Further information on usage and supporting classes with examples appears in **`preview/Grid/grid.htm`**.

Store Previews

Store for Access

The StoreForAccess parcel, formerly in “goodies,” has been enhanced by Cincom and moved into preview. It is now called StoreForMSAccess, to distinguish it from the former parcel.

The enhancements include:

- A schema identical that for the supported Store databases.
- Ability to upgrade the schema with new Cincom releases (e.g., running `DbRegistry update74`).
- Ability to create the database and install the tables all from within Smalltalk, as described in the documentation.
- No need to use the Windows Control Panel to create the Data Source Name.

The original parcel is no longer compatible with VW 7.4, because it does not have the same schema and ignores the newer Store features.

While MS Access is very useful for personal repositories, for multi-user environments we recommend using a more powerful database.

Store for Supra

In order to allow Store to use Supra SQL as the repository, the StoreForSupra package provides a slightly modified version of the Supra EXDI, and implements circumventions for the limitations and restrictions of Supra SQL which are exposed by Store. The Store version of Supra EXDI does not modify/override anything in the base SupraEXDI package. Instead, modifications to the Supra EXDI are achieved by subclassify the Supra EXDI classes.

Circumventions are implemented by catching error codes produced when attempting SQL constructs that are unsupported by Supra SQL and inserting one or more specifically modified SQL requests. The Supra SQL limitations that are circumvented are:

- Blob data (i.e. LONGVARCHAR column) is returned as null when accessed through a view.
- INSERT statement may not be combined with a SELECT on the same table (CSWK7025)
- UPDATE statement may not update any portion of the primary KEY (CSWK7042)
- DELETE statement may not have a WHERE... IN (...) clause with lots of values (CSWK1101, CSWK1103)
- When blob data (i.e. LONGVARCHAR column) is retrieved from the data base, the maximum length is returned rather than the actual length
- Supra SQL does not have SEQUENCE

StoreForSupra requires Supra SQL 2.9 or newer, with the following tuning parameters:

```
SQLLONGVARCHAR = Y
SQLMAXRESULTSETS = 256
```

StoreForSupra installation instructions

1. Install Supra SQL
2. Create a Supra database
3. Use XPARAM under Windows to set the following
    - set Password Case Conversion = Mixed
    - set Supra tuning variable SQLLONGVARCHAR = Y
    - set Supra tuning variable SQLMAXRESULTSETS = 256
4. Start the Supra database
5. From the SUPERDBA user, create the Store administration user with DBA privileges.
    - User ID BERN is recommended, password is your own choice.
    - Sample SQL for creating the Store administration use:

      ```
      create user BERN password BERN DBA not exclusive
      ```

6 Load the StoreForSupra parcel.

7 To create the Store tables in the Supra database, run the following Smalltalk code from a workspace (You will be prompted for the Supra database name, the Supra administration user id and password.)

```
Store.DbRegistry goOffLine installDatabaseTables.
```

8 To remove the Store tables from the Supra database, run the following Smalltalk code from a workspace

```
Store.DbRegistry goOffLine deinstallDatabaseTables.
```

Security

OpenSSL cryptographic function wrapper

The OpenSSL-EVP package provides access to most of the cryptographic functions of the popular OpenSSL library (http://www.openssl.org). The functions currently available include

- symmetric ciphers: ARC4, AES, DES and Blowfish
- hash functions: MD5, SHA, SHA256, SHA512
- public ciphers: RSA, DSA

The API of this wrapper is modelled after the native Smalltalk cryptography classes so that they can be polymorphically substituted where necessary. Since these classes use the same name they have to live in their own namespace, Security.OpenSSL. The intent is that each set of classes can be used interchangably with minimal modification of existing user code.

Along these lines, you can instantiate an instance of an OpenSSL algorithm same way as the native ones. For example:

```
| des ciphertext plaintext |
des := Security.OpenSSL.DES newBP_CBC
        setKey: '12345678' asByteArray;
        setIV: '87654321' asByteArray;
        yourself.
ciphertext := des encrypt: ('Hello World!' asByteArrayEncoding: #utf_8).
plaintext := (des decrypt: ciphertext) asStringEncoding: #utf_8
```

An alternative way to configure an algorithm instance is using cipher wrappers. The equivalent of the #newBP_CBC method shown above would be the following.

```
des := Security.OpenSSL.BlockPadding on: (
    Security.OpenSSL.CipherBlockChaining on:
        Security.OpenSSL.DES new ).
```

Note that while the APIs look the same the two implementations have different underlying architectures, so generally their components should not be mixed. That is, OpenSSL wrappers merely call the OpenSSL library with some additional "flags", whereas the Smalltalk versions augment the calculations. In general, it won't work properly to use a Smalltalk cipher mode wrapper class around an OpenSSL algorithm and vice versa.

The only thing that is different with hash functions is that the OpenSSL version does not support cloning, so #copy will raise an error. Consequently, it is currently hard to use them with HMAC, which uses cloning internally. We have yet to modify the HMAC implementation to avoid that. However the wrapper already provides SHA512 which is not yet available with the Smalltalk library.

The wrapper also supports 2 public key algorithms, RSA and DSA. The keys for these algorithms are more complex than the simple byte sequences used with symmetric ciphers. However, the wrapper is written so that the API uses the exact same kind of objects for both the Smalltalk version and the OpenSSL version. Similarly for DSA signatures, both versions use DSASignature instances. Here's an example.

```
| message keys dsa signature |
message := 'This is the end of the world as we know it ...' asByteArray.
keys := Security.DSAKeyGenerator keySize: 512.
dsa := Security.OpenSSL.DSA new.
dsa privateKey: keys privateKey.
signature := dsa sign: message.
dsa publicKey: keys publicKey.
dsa verify: signature of: message
```

The current version of the wrapper should support usual OpenSSL installations on Windows and Linux and various Unixes out of the box. There is only one interface class, with platform specific library file and directory specifications in it. If you get a LibraryNotFoundError when trying to use this package, you may need to change or add these entries for your specific platform. You need to find out what is the correct name of the OpenSSL cryptographic library on your platform and where is it located, and update the #libraryFiles: and #libraryDirectories: attributes of the OpenSSLInterface class accordingly. More information can be found in the *DLL and C Connect User's Guide* (p.51). To obtain the shared library for your platform, see http://www.openssl.org/source. Note that the library is usually included with many of the popular Linux distributions, in many cases this package should just work.

A note about HP platforms. If your version of the openssl library doesn't contain (export) the requested function, the image can hang. On Windows, an exception is thrown instead (object not found). The workaround is to verify that your version of the library has the functions you need. For example, the "CFB" encryption facility wasn't available until version 0.9.7.e. And the sha256 and sha512 are only available after 0.9.8 and higher.

Also, on all platforms, remember that the openssl library uses pointers to memory areas which are valid only while the image is still running. After an image shutdown, all pointers are invalid. Your code should therefore discard OpenSSL objects, and generate new ones with each image restart. Even though your old objects will be alive at startup, (a return from snapshot), the pointers are invalid, and the openssl library no longer remembers any of its own state information from the previous session.

Opentalk

The Opentalk preview provides extensions to 7.2 and the Opentalk Communication Layer. It includes a preview implementation of SNMP, a remote debugger and a distributed profiler. The load balancing components formerly shipped as preview components in 7.0 is now part of the Opentalk release.

For installation and usage information, see the readme.txt files and the parcel comments.

Opentalk HTTPS

This release includes a preview of HTTPS support for Opentalk. HTTPS is normal HTTP protocol tunneled through an SSL protected socket connection. Similarly to Opentalk-HTTP, the package Opentalk-HTTPS only provides the transport level infrastructure and needs to be combined with application level protocol like Opentalk-XML or Opentalk-SOAP.

An HTTPS broker must be configured with a SSLContext for the role that it will be playing in the SSL connections, i.e., #serverContext: for server roles and #clientContext: for client roles. Also, the authenticating side (which is almost always the client) needs to have a corresponding validator block set as well. The client broker will usually need to have the #serverValidator: block set to validate server certificates. The server broker will only have its #clientValidator: block set if it wishes to authenticate the clients. Note that the presence or absence of the #clientValidator: block is interpreted as a trigger for client authentication.

Here's the full list of all HTTPSTransportConfiguration parameters:

clientContext
The context used for connections where we act as a client.

serverContext
The context used for connections where we act as a server.

clientValidator
The subject validation block used by the server to validate client certificates.

serverValidator
The subject validation block used by the client to validate server certificates.

Note that the same broker instance can be set up to play both client and server roles, so all 4 parameters can be present in a broker configuration. For more information on setting up SSLContext for clients or servers please refer to the relevant chapters of the *Security Guide*.

This example shows how to set up a secure Web Services broker as a client:

```
context := Security.SSLContext newWithSecureCipherSuites.
broker :=
    (BasicBrokerConfiguration new
        adaptor: (
            ConnectionAdaptorConfiguration new
                isBiDirectional: false;
                transport: (
                    HTTPSTransportConfiguration new
                        clientContext: context;
                        serverValidator:
                            [ :name | name commonName = 'Test Server' ];
                        marshaler: (
                            SOAPMarshalerConfiguration new
                                binding: aWsdlBinding)))
    )   newAtPort: 4242.
```

This example shows how to set up a secure Web Services broker as a server:

```
context := Security.SSLContext newWithSecureCipherSuites.
"Servers almost always need a certificate and private key, clients only when
client authetication is required."
"Assume the server certificate is stored in a binary (DER) format."
file := 'certificate.cer' asFilename readStream binary.
[ certificate := Security.X509.Certificate readFrom: file ] ensure: [ file close ].
"Assume the private key is stored in a standard, password encrypted PKCS8
format"
file := 'key.pk8' asFilename readStream binary.
[ key := Security.PKCS8 readKeyFrom: file password: 'password' ] ensure:
    [ file close ].
context certificate: certificate key: key.
broker :=
    (BasicBrokerConfiguration new
        adaptor: (
            ConnectionAdaptorConfiguration new
                isBiDirectional: false;
                transport: (
                    HTTPSTransportConfiguration new
                        serverContext: server;
                        marshaler: (
                            SOAPMarshalerConfiguration new
                                binding: aWsdlBinding)))
    )   newAtPort: 4242.
```

This release also includes a toy web server built on top of Opentalk as contributed code, and is not supported by Cincom. It is, however, quite handy for testing the HTTP/HTTPS transports without having other complex infrastructure involved. So here is another example how to set up a simple secure web server as well:

```
| resource ctx |
resource := Security.X509.RegistryTestResource new setUp.
ctx := Security.SSLContext
      suites: (Array with: Security.SSLCipherSuite
            SSL_RSA_WITH_RC4_128_MD5)
      registry: resource asRegistry.
ctx rsaCertificatePair: resource fullChainKeyPair.
(Opentalk.AdaptorConfiguration webServer
      addExecutor: (Opentalk.WebFileServer prefix: #('picture')
          directory: '$(HOME)\photo\web' asFilename);
      addExecutor: (Opentalk.WebFileServer prefix: #('ws')
          directory: '..\ws' asFilename);
      addExecutor: Opentalk.WebHello new;
      addExecutor: Opentalk.WebEcho new;
      transport: (Opentalk.TransportConfiguration https
            serverContext: ctx;
            marshaler: Opentalk.MarshalerConfiguration web )
) newAtPort: 4433
```

Once the server is started, it should be accessible using a web browser, for example https://localhost:4433/hello.

Distributed Profiler

The profiler has not changed since the last release and works only with the old AT Profiler, shipped in the **`obsolete/`** directory.

Installing the Opentalk Profiler in a Target Image

If you want to install only the code needed for images, potentially headless, that are targets of remote profiling, install the following parcel:

- Opentalk-Profiler-Core

Installing the Opentalk Profiler in a Client Image

To create an image that contains the entire Opentalk profiler install the following parcels in the order shown:

- Opentalk-Profiler-Core
- Opentalk-Profiler-Tool

Opentalk Remote Debugger

This release includes an early preview of the Remote Debugger. Its functionality is seriously limited when compared to the Base debugger, however its basic capabilities are good enough to be useful in many cases. The limitations are mostly related to actions that open other tools. For those to work, we have yet to make the other tools remotable as well.

The remote debugger is contained in two parcels.

The Opentalk-Debugger-Remote-Monitor parcel loads support for the image that will run the remote debugger interface. The monitor is started by sending:

```
RemoteDebuggerClient startMonitor
```

Once the monitor is started, other images can “attach” to it. The monitor will host the debuggers for any unhandled exceptions in the attached images.

To shutdown a monitor image, all the attached images should be detached first and then the monitor should be stopped, by sending:

```
RemoteDebuggerClient stopMonitor
```

The Opentalk-Debugger-Remote-Target parcel loads support for the image that is expected to be debugged remotely. To enable remote debugging this image has to be “attached” to a monitor, i.e., to the image that runs the remote debugger UI. Attaching is performed with one of the “attach*' messages defined on the class side of RemoteDebuggerService. Use detachMonitor to stop forwarding of unhandled exceptions to the remote monitor image.

A packaged (possibly headless) image can be converted into a “target” during startup by loading the Opentalk-Debugger-Remote-Target parcel using the **`-pcl`** command line option. Additionally it can be immediately attached to a monitor image using an **`-attach [host][:port]`** option on the same command line. It is assumed that the Base debugger is in the image (hasn't been stripped out) and that the prerequisite Opentalk parcels are also available on the parcel path of the image.

Testing and Remote Testing

The **`preview/opentalk`** subdirectory contains two new parcels, included for those users who expressed an interest in the multi-image extension to the SUnit framework used to demonstrate the Opentalk Load Balancing Facility:

- Opentalk-Tests-Core contains basic extensions to the SUnit framework used to test Opentalk. Version number 73 6 is shipped with this release.
- Opentalk-Remote-Tests-Core contains the central classes of the remote testing framework and some simple examples. Version number 73 9 is shipped with this release.

The framework these packages implement is known to have defects and is evolving. Future versions will differ, substantially.

The central idea behind the framework is that since SUnit resources are classes, there is no reason why references to remote classes cannot be substituted for them in a test case.

The are two central classes in the framework.

OpentalkTestCaseWithRemoteResources
This is the superclass of all concrete, multi-image test cases. It contains an instance variable named 'resources' that is populated with references to remote resource classes. The references are constructed from the data returned by the method resourceObjRefs, which any concrete test case must implement. The class has a shared variable named CaseBroker that contains the broker in which the resource references are registered. This request broker is the one used by all multi-image test cases to communicate with remote resources.

OpentalkTestRemoteResource
This is the superclass of all concrete remote resources. It has a shared variable named ResourceBroker that holds the broker through which test resources communicate with test cases. Concrete resources register themselves with this broker, using their class name as an OID, so that test cases may programmatically generate references to them.

Since multi-image tests usually involve resources that start up brokers and exchange messages of their own, care must be taken in any test to determine that the communication exchange under test has completed before any asserts are evaluated. Also, since the exchange between resources may be complex, the assert: messages are usually phrased in terms of the contents of event logs. Much use is made of the Opentalk event and event logging facilities. Test may create event logs of their own, or analyze the remote event logs created by a remote resource.

The current scheme assumes that there will be only one resource per image, but you may construct a resources with arbitrary complexity.

The drill for configuring a multi-image test is now overly complex, because port numbers are derived from the suffix of the image name, expected to consist of two decimal digits. Port numbers are also hard-coded in the method resourceObjRefs. This is the wrong way to do things that we intend to move to a scheme where the test case image starts its broker on a well known port, and resource images register with the test case image on startup.

That said, the current drill goes as follows. The essentials are also discussed in the class comment of CaseRemoteClientServer.

1 Make sure that the machines you intend to use are not already listening on the default ports used by the multi-image testing framework. The CaseBroker, if you follow our recommendations, will come up on port 1800, and resource brokers will come up in the range 1900-1999. If your machines are already using these ports, alter the class-side method basePortNumber in OpentalkTestCaseWithRemoteResources or OpentalkTestCaseRemoteResource, as appropriate. The following directions will assume that you did not need to change an implementation of basePortNumber.

2 Write your resource class or classes. You may use any of the concrete classes under ResourceWithConfiguration as models.

3 Write your test case class. You may use any of the concrete classes under CaseRemoteClientServer as models.

4 Save your image.

5 Remind yourself of how many resources your test case employs. For example, class CaseRemoteClients1Servers1 requires three images. You can check this by examining its implementation of resourceObjRefs. Two references are set up, one for a client and one for a server. The third image will be the one that runs the test case. So, if your image is named **`otwrk.im`**, clone copies of it now, named **`otwrk00.im`**, **`otwrk01.im`** and **`otwrk02.im`**. All the image names must end in two digits. The name ending in “00” is conveniently reserved for the test running image, making its broker come up on port 1800. All the images derive the port of their broker from their image name. In this case, the resource images will start their brokers on ports 1901 and 1902.

6 After saving the images, reopen them, and start the relevant brokers. Remember that in the test case image you only want to start the CaseBroker. In the resource images, you start their ResourceBroker. The class-side protocol of OpentalkTestCaseWithRemoteResources and OpentalkTestCaseRemoteResource both contain start up methods with useful executable comments, if you like doing things that way. (You will use only one image, and start both kinds of brokers in it, only when you intend that everything run in the same image. And that setup is very useful in debugging.)

7 Run your tests, from the test case image, and run them one at a time. The framework has known difficulties running a test suite.

If you ever find that your event logs show record of, say, 50 messages, when your test only sends 30, then the preceding test run—which you probably thought you had successfully terminated by, say, closing your debugger—was still going strong. Clean up as necessary and start again.

Opentalk SNMP

SNMP is a widely deployed protocol that is commonly used to monitor, configure, and manage network devices such as routers and hosts. SNMP uses ASN.1 BER as its wire encoding and it is specified in several IETF RFCs.

The Opentalk SNMP preview partially implements two of the three versions of the SNMP protocol: SNMPv1 and SNMPv2. It does so in the context of a framework that both derives from the Opentalk Communication Layer and maintains large-scale fidelity to the recommended SNMPv3 implementation architecture specified in IETF RFC 2571.

Usage

Initial Configuration

Opentalk SNMP cares about the location of one DTD file and several MIB XML files. So, before you start to experiment, be sure to modify 'SNMPContext>>mibDirectories' if you have relocated the Opentalk SNMP directories.

Broker or Engine Creation and Configuration

In SNMPv3 parlance a broker is called an “engine”. An engine has more components that a typical Opentalk broker. In addition to a single transport mapping, a single marshaler, and so on, it must have or be able to have

- several transport mappings,
- a PDU dispatcher,
- several possible security systems,
- several possible access control subsystems,
- a logically distinct marshaler for each SNMP dialect, plus
- an attached MIB module for recording data about its own performance.

So, under the hood, SNMP engine configuration is more complex than the usual Opentalk broker configuration. You can create a simple SNMP engine with

```
SNMPEngine newUDPAtPort: 161.
```

But, this is implemented in terms of the more complex method below. Note that, for the moment, within the code SNMP protocol versions are distinguished by the integer used to identify them on the wire.

```
newUdpAtPorts: aSet
| oacs |

oacs := aSet collect: [ :pn |
    AdaptorConfiguration snmpUDP
        accessPointPort: pn;
        transport: ( TransportConfiguration snmpUDP
            marshaler: ( SNMPMarshalerConfiguration snmp ) )].

^(( SNMPEngineConfiguration snmp )
    accessControl: ( SNMPAccessControlSystemConfiguration snmp
        accessControlModels: ( Set
            with: SNMPAccessControlModelConfiguration snmpv0
            with: SNMPAccessControlModelConfiguration snmpv1 ) );
        instrumentation: ( SNMPInstrumentationConfiguration snmp
            contexts: ( Set with: (
                SNMPContextConfiguration snmp
                    name: SNMP.DefaultContextName;
                    values: ( Set with: 'SNMPv2-MIB' ) ) ) );
        securitySystem: ( SNMPSecuritySystemConfiguration snmp
            securityModels: ( Set
                    with: SNMPSecurityModelConfiguration snmpv0
                    with: SNMPSecurityModelConfiguration snmpv1 ) );
            adaptors: oacs;
            yourself
        ) new
```

As you can see, it is a bit more complex, and the creation method makes several assumptions about just how you want your engine configured, which, of course, you may change.

Engine Use

Engines are useful in themselves only as lightweight SNMP clients. You can use an engine to send a message and get a response in two ways. The Opentalk SNMP Preview now supports an object-reference based usage style, as well as a lower-level API.

OR-Style Usage

If you play the object reference game, you get back an Association or a Dictionary of ASN.1 OIDs and the objects associated with them. For example, the port 3161 broker sets up its request using an object reference:

```
| broker3161 broker3162 oid ref return |

broker3161 := SNMPEngine newUdpAtPort: 3161.
broker3162 := self snmpv0CommandResponderAt: 3162.
broker3161 start.
broker3162 start.
oid := CanonicalAsn1OID symbol: #'sysDescr.0'.
ref := RemoteObject
      newOnOID: oid
      hostName: <aHostname>
      port: 3162
      requestBroker: broker3161.
^return := ref get.
```

This expression returns:

```
Asn1OBJECTIDENTIFIER(CanonicalAsn1OID(#'1.3.6.1.2.1.1.1.0'))->
      Asn1OCTETSTRING('VisualWorks®, Pre-Release 7 godot
      mar02.3 of March 20, 2002')
```

Object references with ASN.1 OIDs respond to get, set:, and so forth. These are translated into the corresponding SNMP PDU type, for example, a GetRequest and a SetRequest PDU in the two cases mentioned.

Explicit Style Usage

You can do the same thing more explicitly the following way, in which case you will get back a whole message:

```
| oid broker1 entity2 msg returnMsg |

oid := CanonicalAsn1OID symbol: #'1.3.6.1.2.1.1.1.0'.
broker1 := SNMPEngine newUdpAtPort: 161.
entity2 := self snmpv1CommandResponderAt: 162.
broker1 start.
entity2 start.
msg := SNMPAbstractMessage getRequest.
msg version: 1.
msg destTransportAddress: ( IPSocketAddress hostName: self
      localHostName port: 162 ).
msg pdu addPduBindingKey: ( Asn1OBJECTIDENTIFIER value: oid ).
returnMsg := broker1 send: msg.
```

which returns:

```
SNMPAbstractMessage:GetResponse[1]
```

Note that in this example, you must explicitly create a request with the appropriate PDU and explicitly add bindings to the message's binding list.

Entity Configuration

In the SNMPv3 architecture, an engine does not amount to much. It must be connected to several SNMP 'applications' in order to do useful work. And 'entity' is an engine conjoined with a set of applications. Applications are things like command generators, command responders, notification originators, and so on. There are several methods that create the usually useful kinds of SNMP entities, like

```
SNMP snmpv0CommandResponderAt: anInteger
```

Again, this invokes a method of greater complexity, but with a standard and easily modifiable pattern. There as several examples in the code.

MIBs

Opentalk SNMP comes with a small selection MIBS that define a subtree for Cincom-specific managed objects. So far, we only provide MIBs for reading or writing a few ObjectMemory and MemoryPolicy parameters. A set of standard MIBS is also provided. Note that MIBs are provided in both text and XML format. The Opentalk SNMP MIB parser required MIBS in XML format.

If you need to create an XML version of a MIB that is not provided, use the 'snmpdump' utility. It is a part of the 'libsmi' package produced by the Institute of Operating Systems and Computer Networks, TU Braunschweig. The package is available for download through http://www.ibr.cs.tu-bs.de/projects/libsmi/index.html, and at http://rpmfind.net.

Limitations

The Opentalk SNMP Preview is raw and has several limitations. Despite them, the current code allows a user, using the SNMPv2 protocol, to modify and examine a running VW image with a standard SNMP tool like ucd-snmp. However, one constraint should be especially noted.

Port 161 and the AGENTX MIB

SNMP is a protocol used for talking to devices, not applications, and by default SNMP uses a UDP socket at port 161. This means that in the absence of coordination between co-located SNMP agents, they will conflict over ownership of port 161. This problem is partially addressed by the AGENTX MIB, which specifies an SNMP inter-agent protocol. Opentalk SNMP does not yet support the AGENTX MIB. This means that an Opentalk SNMP agent for a VisualWorks application (only a virtual device) must either displace the host level SNMP agent on port 161, or run on some other port. Opentalk SNMP can run on any port, however many commercial SNMP management applications are hard-wired to communicate only on port 161. This places limitations on the extent to which existing SNMP management applications can now be used to manage VisualWorks images.

OpentalkCORBA

This release includes an early preview of our OpentalkCORBA initiative. Though our ultimate goal is to replace DST, DST will remain a supported product until OpentalkCORBA matches all its relevant capabilities and we provide a reasonable migration path for current DST users. So, we would very much like to hear from our DST users, about the features and tools they would like us to carry over into OpentalkCORBA.

For example, we do not intend to port any of the presentation-semantic split framework, or any of the UIs that essentially depend upon it, unless there is strong user demand. Please contact Support, and ask them to forward your concerns and needs to the VW Protocol and Distribution Team.

This version of OpentalkCORBA combines the standard Opentalk broker architecture with DST's IDL marshaling infrastructure to provide IIOP support for Opentalk. OpentalkCORBA has its own clone of the IDL infrastructure residing in the Opentalk namespace so that changes made for Opentalk do not destabilize DST. The two frameworks are almost capable of running side by side in the same image. The standard base class extensions, however, like 'CORBAName' can only work for one framework, usually the one that was loaded last. Therefore, if you want to load both and be sure that DST is unaffected, make sure it is loaded after OpentalkCORBA, not before.

This version of OpentalkCORBA already offers a few improvements over DST. In particular, it supports the newer versions of IIOP, though there is no support for value types yet. A short list of interesting features and limitations follows:

- supports IIOP 1.0, 1.1, 1.2
- defaults to IIOP 1.2

- does not support value types
- does not support Bi-Directional IIOP
- doesn't support the NEEDS_ADDRESSING_MODE reply status
- system exceptions are currently raised as Opentalk.SystemExceptions
- user exceptions are currently raised as Error on the client side
- supports LocateRequest/LocateReply
- does not support CancelRequest
- does not support message fragmenting
- the general IOR infrastructure is fleshed out (IOPTaggedProfiles, IOPTaggedComponents, IOPServiceContexts) and adding new kinds of these components amounts to adding new subclasses and writing corresponding read/write/print methods
- the supported profiles are IIOPProfile and IOPMultipleComponentProfile, and anything else is treated as an IOPUnknownProfile
- the only supported service context is CodeSet, and anything else is treated as an IOPUnknownContext
- however it does not support the codeset negotiation algorithm yet; correct character encoders for both char and wchar types can be set manually on the CDRStream class
- the supported tagged components are CodeSets, ORBType and AlternateAddress, and anything else is treated as an IOPUnknownComponent

IIOP has the following impact on the standard Opentalk architecture and APIs:

- there is a new IIOPTransport and CDRMarshaler with corresponding configuration classes
- these transport and marshaler configurations must be included in the configuration of an IIOP broker in the usual way
- the new broker creation API consists of the following methods
    - #newCdrIIOPAt:
    - #newCdrIIOPAt:minorVersion:
    - #newCdrIIOPAtPort:
    - #newCdrIIOPAtPort:minorVersion:

- IIOP proxies are created using Broker>>remoteObjectAt:oid:interfaceId:
- there is an extended object reference class named IIOPObjRef
- the LocateRequest capabilities are accessible via
  - Broker>>locate: anIIOPObjRef
  - RemoteObject>>_locate
- LocateRequests are handled transparently on the server side.
- A location forward is achieved by exporting a remote object on the server side (see the example below)

Examples

Remote Stream Access

The following example illustrates basic messaging capability by accessing a stream remotely. The example takes advantage of the IDL definitions in the SmalltakTypes IDL module:

```
| broker stream proxy oid |
broker := Opentalk.BasicRequestBroker newCdrIiopAtPort: 4242.
broker start.
[     oid := 'stream' asByteArray.
      stream := 'Hello World' asByteArray readStream.
      broker objectAdaptor export: stream oid: oid.
      proxy := broker
          remoteObjectAt: (
              IPSocketAddress
                  hostName: 'localhost'
                  port: 4242)
          oid: oid
       interfaceId: 'IDL:SmalltalkTypes/Stream:1.0'.
      proxy next: 5.
] ensure: [ broker stop ]
```

“Locate” API

This example demonstrates the behavior of the “locate” API:

```
| broker |
broker := Opentalk.BasicRequestBroker newCdrIiopAtPort: 4242.
broker start.
[	| result stream oid proxy found |
	found := OrderedCollection new.

	"Try to locate a non-existent remote object"
	oid := 'stream' asByteArray.
	proxy := broker
		remoteObjectAt: (
			IPSocketAddress
				hostName: 'localhost'
				port: 4242)
		oid: oid
		interfaceId: 'IDL:SmalltalkTypes/Stream:1.0'.
	result := proxy _locate.
	found add: result.

	"Now try to locate an existing remote object"
	stream := 'Hello World' asByteArray readStream.
	broker objectAdaptor export: stream oid: oid.
	result := proxy _locate.
	found add: result.
	found
] ensure: [ broker stop ]
```

Transparent Request Forwarding

This example shows how to set up location forward on the server side and demonstrates that it is handled transparently by the client.

```
| broker |
broker := Opentalk.BasicRequestBroker newCdrIiopAtPort: 4242.
broker start.
[     | result stream proxy oid fproxy foid|
      oid := 'stream' asByteArray.
      stream := 'Hello World' asByteArray readStream.
      broker objectAdaptor export: stream oid: oid.
      proxy := broker
          remoteObjectAt: (
              IPSocketAddress
                  hostName: 'localhost'
                  port: 4242)
          oid: oid
          interfaceId: 'IDL:SmalltalkTypes/Stream:1.0'.
      foid := 'forwarder' asByteArray.
      broker objectAdaptor export: proxy oid: foid.
      fproxy := broker
          remoteObjectAt: (
              IPSocketAddress
                  hostName: 'localhost'
                  port: 4242)
          oid: foid
          interfaceId: 'IDL:SmalltalkTypes/Stream:1.0'.
      fproxy next: 5.
] ensure: [ broker stop ]
```

Listing contents of a Java Naming Service

This example provides the code for listing the contents of a running Java JDK 1.4 naming service. It presumes that you have Opentalk-COS-Naming loaded. To run the Java naming service, just invoke 'orbd -ORBInitialPort 1050' on a machine with JDK 1.4 installed.

Note that this example also exercises the LOCATION_FORWARD reply status, the broker transparently forwards the request to the true address of the Java naming service received in response to the pseudo reference 'NameService'.

```
| broker context list iterator |
broker := Opentalk.BasicRequestBroker newCdrIiopAtPort: 4242.
broker passErrors; start.
[	context := broker
		remoteObjectAt: (
			IPSocketAddress
				hostName: 'localhost'
				port: 1050)
		oid: 'NameService' asByteArray
		interfaceId: 'IDL:CosNaming/NamingContextExt:1.0'.
	list := nil asCORBAParameter.
	iterator := nil asCORBAParameter.
	context
		listContext: 10
		bindingList: list
		bindingIterator: iterator.
	list value
] ensure: [ broker stop ]
```

List Initial DST Services

This is how you can list initial services of a running DST ORB. Note that we're explicitly setting IIOP version to 1.0.

```
| broker dst |
broker := Opentalk.BasicRequestBroker
		newCdrIiopAtPort: 4242
		minorVersion: 0.
broker start.
[	dst := broker
		remoteObjectAt: (
			IPSocketAddress
				hostName: 'localhost'
				port: 3460)
		oid: #[0 0 0 0 0 1 0 0 2 0 0 0 0 0 0 0]
		interfaceId: 'IDL:CORBA/ORB:1.0'.
	dst listInitialServices
] ensure: [ broker stop ]
```

Seaside Support

Several additional components in support of Seaside are included as preview code. These are described with the “Seaside Support” notes in chapter 2.

Internet Browser Plugin

The new VisualWorks Plugin is available as a preview for VW 7.5 remains in preview for 7.6 without further change.

Beginning with the VisualWorks Plugin for VW 7.5, we provide both an ActiveX Control for Internet Explorer and a Gecko Plugin for use in Gecko 2.0 enabled browsers such as Netscape 8, Mozilla 1.7.5+ and Firefox 1.0, 1.5 and 2.0. For the time being, the Plugin is still only available on Windows platforms.

The Plugin parcel for VW 7.5 is not backward compatible with any of the old Plugin parcels. Please see preview/plugin/readme.txt for more information.

International Domain Names in Applications (IDNA)

RFC 3490 “defines internationalized domain names (IDNs) and a mechanism called Internationalizing Domain Names in Applications (IDNA) which provide a standard method for domain names to use characters outside the ASCII repertoire. IDNs use characters drawn from a large repertoire (Unicode), but IDNA allows the non-ASCII characters to be represented using only the ASCII characters already allowed in so-called host names today. This backward-compatible representation is required in existing protocols like DNS, so that IDNs can be introduced with no changes to the existing infrastructure. IDNA is only meant for processing domain names, not free text” (from the RFC 3490 Abstract).

Limitations

The current implementation in VisualWorks

- doesn't do NAMEPREP preprocessing of strings (currently we just convert all labels to lowercase)
- doesn't properly implement all punycode failure modes
- needs exceptions instead of Errors
- needs I18N of messages

USAGE

You can convert an IDN using the IDNAEncoder as follows:

```
IDNAEncoder new encode: 'www.cincom.com'
    "result: www.cincom.com"
```

or

```
IDNAEncoder new encode: 'www.cìncòm.com'
    "result: www.xn--cncm-qpa2b.com"
```

and decode with

```
IDNAEncoder new decode: 'www.xn--cncm-qpa2b.com'
    "result: www.cìncòm.com"
```

This package also overrides the low level DNS access facilities to encode/decode the hostnames when necessary. Here's an example invocation including a Japanese web site.

```
host := (String with: 16r6c5f asCharacter with: 16r6238 asCharacter), '.jp'.
address := IPSocketAddress hostAddressByName: host.
    "result: [65 99 223 191]"
```

The host name that is actually sent out to the DNS call is:

```
IDNAEncoder new encode: host
    "result: xn--0ouw9t.jp"
```

A reverse lookup should also work, however I wasn't able to find an IP address that would successfully resolve to an IDN, so I wasn't able to test it. Even our example gives me only the numeric result:

```
IPSocketAddress hostNameByAddress: address
    "result: 65.99.223.191"
```

5

Microsoft Windows CE

WinCE devices have been supported since 7.3. Because separate documentation has not been developed or provided elsewhere, this section repeats the information provided in the releases here from the previous release.

Supported Devices

Virutal machines for Microsoft Windows CE are intended for use on CE devices as an application deployment environment. Typically, an application is developed in a standard development environment, and prepared for deployment on a CE device. The image, VM, and any supporting files, are then copied to the CE device and executed.

VisualWorks has been successfully tested on the following hardware:

- Simpad SLC with StrongARM-SA-1110, Windows CE .NET Version 4.0
- skeye.pad with StrongARM-SA-1110, Windows CE .NET Version 4.1
- HP iPAQ H2210 with Intel PXA255 XScale, Windows Pocket PC 2003 (Windows Mobile 2003)
- Tatung WebPAD with Geode GXm, Windows CE .NET Version 4.10

There are, however, limitations. Refer to “Known limitations” below for details.

Distribution contents

There are two directories with virtual machines for the different processors:

- **`bin\cearm`** – for StrongARM and XScale processors,
- **`bin\cex86`** – for Pentium-compatible processors like the Geode.

Each directory contains three executables and a DLL:

- **`vwntoe.dll`** – the DLL containing the virtual machine.
- **`vwnt.exe`** – the GUI stub exe which is normally used to run GUI applications. It uses **`vwntoe.dll`**.
- **`vwntconsole.exe`** – the console stub executable which is normally used to run console applications. It uses **`vwntoe.dll`**.
- **`visual.exe`** – the single virtual machine, which is used for single-file executable packaged applications.

The assert and debug subdirectories contain versions of these executables with asserts turned on for debugging. The debug engines are not optimized and so can be used with the Microsoft eMbedded Visual C++ debugger. Refer to the engine type descriptions in the *Application Developer's Guide*, Appendix C, for further information.

Prerequisites

Windows CE VMs require a few additions to the standard image. These are provided in the parcel **`ce.pcl`**. On the PC, prior to the deployment to your CE machine, load this parcel into your image.

This parcel contains two major changes:

- A new SystemSupport subclass for CE – This is necessary because the name of the DLLs differs from other Windows versions and they contain different versions of the called functions. For example, only Unicode versions of most functions are provided and some convenience functions are missing.
- A new filename subclass, CEFilename – CE does not have a "current working directory" concept, so only absolute paths are supported. Therefore CEFilename stores the current directory and expands relative paths into absolute paths.

Developing an Application for CE

In general, developing an application for deployment on a CE device is the same as for any other application. The notable differences have to do with screen size, especially on small PDA-type devices, and filename handling, because CE does not use file volumes or disk drive letters.

Before beginning development, load the CE parcel (ce.pcl) into the development image. The changes it makes only take effect when the image is installed on the CE device, so you can develop as usual on your standard development system.

Filenames

WinCE does not use relative file paths or volume (disk) letters. This is transparent during development, because the CEFilename class handles converting all paths to absolute paths when the application is deployed on a CE device. No special development restrictions need to be observed.

DLL names

Similar, DLL names are modified appropriately when installed on a CE device.

Window sizes and options

CE devices come in a variety of screen sizes. For the larger devices, with a screen size of 640x400, the limitations are not extreme. However, on the smaller devices, such as a Pocket PC with a screen size or 240x320, the size greatly affects your GUI and application design.

As a deployment environment, you generally should have all development tools, such as browsers closed, and possibly removed from the system, though this is not required.

However, when testing and debugging it is convenient to have all of these development resources available, and this can present serious difficulties.

Also, especially for smaller devices, select an appropriate opening position for the GUI, in the canvas settings. Opening screen center is generally a safe choice.

Input devices

The input side limitations are also worth mentioning. Typically you only have a touch sensitive screen and a pen for it. There is no keyboard, hence no modifier keys. You have no mouse buttons where VisualWorks prefers to have three. So moving the pen somewhere always implies a pressed button. You can open the 'soft input panel', i.e. a small window with a keyboard in it. But it is not really comfortable to enter longer texts this way and this window needs some of your valuable screen space. So whenever you expect textual input, you should leave some free room for the keyboard. (At 240x320, a full screen work space contains 10 lines of text plus title bar, menu bar, tool bar buttons and the status bar at the bottom. The Keyboard window covers the lines 8 to 10 and the status bar.)

The CE parcel adds code which interprets holding the pen for approx 1.3 seconds as a right button press to open the operate context menu. This behavior can be turned on and off in the look and feel section of the settings window. On pocket PC, but not on the CE web pads, users are trained to expect this behavior.

.NET access

While WinCE .NET uses the features of the Microsoft .NET platform, the DotNETConnect preview does not support their use.

Deploying on a CE Device

Load the CE parcel (**`$(VISUALWORKS)\bin\winCE\CE.pcl`**) into your development image. This provides the features described above (see "Prerequisites").

Deployment preparation is, otherwise, the same as usual, though there may be practical considerations. On many devices

Starting VisualWorks on CE

There are several ways to start VisualWorks on Windows CE:

- In the command shell, execute:

  ```
  visual [options] visual.im
  ```

  (Not all CE environments have a command shell interface.)

- Double-click on **`visual.exe`**. This starts VisualWorks with the default image, **`visual.im`**..

  By default, the vm attempts to open an image with the same name as the vm and in the same directory. So, you can rename the vm to match your image name and execute it in this way.

- Double-click on an image file. This works only if the **`.im`** extension is associated with VisualWorks in the registry of the CE device.

  If you are developing on the CE device, you can evaluate this expression in a workspace:

  ```
  WinCESystemSupport registerVisualworksExtension
  ```

- If you have packaged the vm and image as a single executable file (e.g. using ResHacker provided in the **`packaging/win`** directory), you can simply run the executable.

- Create a short-cut to read e.g.

  ```
  "\My Documents\vwnt" "\My Documents\visual.im"
  ```

  The default CE Windows explorer can be used to create associations by copying an existng short-cut (e.g., Control panel), renaming it, and editing its properties. On CE machines that lack the standard explorer, you can find free tools to edit associations.

Known limitations

Sockets

- Non-blocking calls are not yet supported.
- Conversion of hostnames to IP addresses, service names to ports, etc., is not implemented. Use addresses instead, e.g., 192.109.54.11 instead of www.cincom.com.

File I/O

- File locking does not exist on CE (prim 667)
- Delete, rename, etc., do not work on open files (prim 1601,1602,..)
- “ `'\' asFilename fileSize` “ fails with FILE_NOT_FOUND_ERROR.

Windows and Graphics

- Animation primitives not working properly (prims 935-937)
- Only full circles are supported by the OS; arcs and wedges are converted to polylines
- No pixmap <-> clipboard primitives

User primitive

- As yet there is no support for user primitives or primitive plugins.

6

Installer Framework

The Installer Framework is a supported component, installable from the packaging/installer directory. Until full documentation can be provided, the following notes are provided.

The VWInstallerFramework parcel provides the basic functionality for the installer, while the VWInstaller parcel serves as an example of customizing this framework for an individual company and product. The installer application is a wizard with a set of pages that are displayed in sequence. Creating a custom installer is largely a matter of changing the **`install.map`** file for that installation. See the **`install.map`** files on either the Commercial or Non-commercial CDs for examples. These can be hand-edited to suit your particular installation needs.

Customizing the install.map File

Dynamic Attributes

The first item in this file is a dictionary containing version information about the particular distribution to be installed. Edit this section as appropriate for your needs. Many attriburtes are self explanatory, but others may require some explanation.

#defaultTargetTail
The default name of the installation subdirectory, which the user can change at install time.

#imageSignature
Used for updating VisualWorks.ini file at install time (auto update of this file is currently a no-op).

#installDirectoryVariableName
The name of the system variable (or registry key) representing the installed location of the product. For VisualWorks, this is $VISUALWORKS. This can be changed as necessary.

#mapVersion
This can be used by the installer to identify older or newer install.map formats.

#requiresKey
Setting this value to true will display the KeyVerifierPage, and will only proceed with the installation once a proper product key has been supplied by the user. VisualWorks installations no longer require this, but the feature remains for those who want it.

#sourcePathVariableName
The name of the system variable (or registry key) representing the location from which the product was installed. For VisualWorks, this is $SOURCE_PATH. This can be changed as necessary.

#variablePath
The path in the Windows registry to use for setting variables on that platform (see Win95SystemSupport.CurrentVersion).

There is also a section of dictionary entries with integer keys and string values of the form “VM *”. The integers represent bytes from the engine thumbprint of the running installer, and are used to identify to the installer the name of the default VM component for the platform on which the installer is run.

Components

Each component is listed in **`install.map`** with various attributes. Many of these are self explanatory, but others require some explanation.

#target: #tgtDir
Although the VisualWorks components are all installed to the main installation directory, the framework anticipates that a need might arise for some components to be installed to a different location. The symbol #tgtDir resolves to the installation directory chosen by the user. However, one could add other symbols, along with supporting code, to allow multiple target directories. For example, if the same installer were to install ObjectStudio and VisualWorks, the symbols #osTgtDir and #vwTgtDir could be used if methods by these names were implemented to answer the appropriate directories.

#environmentItems:
These represent system variables (or Windows registry entries) to be set when the containing component is installed. In the VisualWorks installation, only the Base VisualWorks component contains these.

#startItems:
These describe the attributes necessary to create a Windows shortcut, such as in the start menu or on the desktop. On Unix these attributes are used to create a small script to launch the newly installed image and VM.

#sizes:
A collection of the uncompressed sizes of all the files in the archive, for determining disk space requirements at install time.

License

The presence of the optional license string in **`install.map`** determines whether the LicenseVerifierPage will be displayed. This string is present in the Non-Commercial installer application, and so the page is displayed, but not in the Commercial installer.

Customizing the Code

The wizard application is called InstallerMainApplication, and the wizard pages are subclasses of AbstractWizardPage. These pages are only displayed when listed in InstallerMainApplication>>subapplicationsForInstall.

Some pages are conditionally displayed, as determined by implementors of #okToBuild. For example, CheckServerPage is only displayed if the server has not yet been checked, or if available updates have not yet been applied. Also, as mentioned earlier LicenseVerifierPage is only displayed if the **`install.map`** to be installed contains a license string.

To change the GUI of either the wizard or its pages, simply subclass and tailor the window or subcanvas spec to suit your needs. Then reference your subclass in #subapplicationsForInstall and it will become part of your installer.

The graphic at the top of the wizard window can be changed by implementing #defaultBanner in a class method of your subclass of InstallerMainApplication.

Once your customizations are done, you can strip your install image from the launcher by selecting **Tools → Strip Install Image**.

Creating Component Archives

The packaging tool (**`goodies/parc/PackingList.pcl`**) that automatically packages our product. However, it is very tailored to our particular build processes, and is not recommended for general use. It runs on a linux box, and creates component archives by first staging all the files in a directory structure and then invoking the following code:

```
UnixProcess
    cshOne: ('tar --create --directory="<1s>" --file="<2s>" --owner=0 --totals
        --verify --same-order <3s>'
            expandMacrosWith: directoryString
            with: fileString
            with: contentString)
```

Note that any Mac files with resource forks must be added to the archive in MacBinaryIII format (*.bin) to be installed properly later.

Local Installations

The scripts and structure of our CDs serve as examples of a working packaged CD. Any archive could be installed from another part of the CD if its #path: attribute is adjusted in the install.map file.

Cincom uses and recommends CDEveryWhere (www.cdeverywhere.com) to create hybrid CDs for distribution that run on Win, Mac, and Unix/linux.

Remote installations

Your wizard subclass should implement #configFileLocation, which answers anFtpURL. This XML file should reside on your server and list the current installer image version, available patches, and available products to install. An example from our NC download site follows:

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

<configuration>
    <installerImageVersion>'1.1'</installerImageVersion>
    <installerParcelVersions>
        '#()'
    </installerParcelVersions>
    <applicationsToInstall>
        '#(#(''VisualWorks 7.1 Non-Commercial'' ''vwnc7.1'')
        #(''VisualWorks 7.2 Non-Commercial'' ''vwnc7.2'')
        #(''VisualWorks 7.2.1 Non-Commercial'' ''vwnc7.2.1''))'
    </applicationsToInstall>
</configuration>
```

In the above example, the last application listed is 'VisualWorks 7.2.1 Non-Commercial', which is the string that will appear in the drop down list of available versions. The string following that, 'vwnc7.2.1', is the subdirectory on the ftp server which contains the application. This subdirectory is flat, unlike the CST CD directory structure, and contains the **install.map** and archive files. The same **install.map** file can work unchanged for CD and remote installations. For remote installations, only the tail of the component archive file is used, since it is assumed that the FTP server does not need the deeper directory structure of the CST CDs.

In addition to the default configuration file location hard coded into your wizard class, users can also keep a local config file, named **installerConfiguration.xml**, which can list alternate local install sources or remote servers. For example, the following local config file lists two additional servers from which one could install any products available there:

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

<configuration>
    <additionalConfigFiles>
        '#(''ftp://anonymous:foo@myServer//remoteInstall/
            installerConfiguration.xml''
        ''ftp://anonymous:foo@theirServer//remoteInstall/
            installerConfiguration.xml'')'
    </additionalConfigFiles>
</configuration>
```

This may be useful anywhere frequent installations might be performed, such as a QA or Tech Support computer lab.