



# Changing the engine while the garage is in motion: Porting to VW7

**Niall Ross, eXtremeMetaProgrammers**

**nfr@bigwig.net**

These are the slides of the talk at the Cincom Smalltalk User Conference in Frankfurt, December 5th-7th, 2006

# Overview

## Porting scenarios

## Porting techniques

- **Minimising code changes**
  - **Namespaces and Visibility**
  - **Class definition and initialization**
- **Synchronising old and new**
  - **Store for Glorp**
  - **Refactoring Browser Frameworks: Environments and refactorings**
  - **Putting them together**
- **Demos**
- **UI Tests**

# Porting Scenario

**The fantasy scenario (a.k.a the porting to Java/.Net scenario)**

- **“Sure, we’ll freeze all development while you port”**
  - (e.g. **Karen Hope’s talk at Smalltalk Solutions 2006**)

**Real life scenario (a.k.a the Smalltalk scenario)**

- **Frenetic development continues while you port**
- **One commonly suggested approach is to do trial runs then ‘suddenly port’**
  - **large complex evolving applications, tools, system extensions, tests**
    - **too many big tasks to do in one**
  - **platform-adapted performance and multi-user/multi-threaded behavior**
    - **may need to deploy in situ and revert**
  - **interruptions: the frenetic development may suddenly need the port team**
  - **commercial/politics: stakeholders will be ready and happy in their own time**

**The port preparation work decays as the system moves on.**

# An Alternative Approach

Accept you may not port suddenly.

- **minimise code change: defer or avoid everything you can**
- **keep the port up-to-date: replicate and reconcile**

**Be agile: work *with* the pressures, not against them.**

**Minimise early code change:**

- **Namespace resolution**
- **class definition**
- **defer initialization**
- **classes to globals**
- **(UI test style)**

**Replicating reconciled refactored code:**

- **Store for Glorp and the Refactoring Browser Framework**
- **replicating and reconciling**
- **refactoring**

# Namespaces

**Porting from a single-namespace source dialect to multi-namespace VW7:**

- every name in the source has a single resolution, so ...
- ... every name the port cares about *must* have a single valid resolution.

**In VW7, build a package with (only) a namespace with the right resolutions**

```
Smalltalk defineNameSpace: #MyApplication ... imports: ...  
and  
conflictingImports: key firstFound: bind  
  ^self == MyApplication  
  ifFalse: [super conflictingImports: key firstFound: bind]  
  ifTrue: [bind]
```

**and use DefaultPackageNamespaces (7.4.1 or from Cincom OR)**

**In source, define global MyApplication := Smalltalk and rewrite app references**

- resolution just works: `at:ifAbsent:`, etc.
- exploit `namesAndBindingsDo: for allClassesDo:`, etc.

**“NameSpaces are about *invisibility*.” (Georg Heeg, ESUG 2006)**

# Store: Class Definition and Initialization

Store began life before NameSpaces; it still has class-side definitions

- a few fixes needed: DatumDescriptor, class-side package setting, ...
- class definition protocol is in the base image, FileOut30 or easy to add  
— to ClassDescription *and* ExternalInterface *and* ExecuteCodeChange  
(Replicate / rewrite definitions between old-style and new-style databases.)

Has the system been ignoring warning 49 for years, or worse?

- use special blessing level(s) to defer initialization to outer bundle
- loaded code is easier to fix

Have the fix for ‘Store bundle content differences between multiple databases’

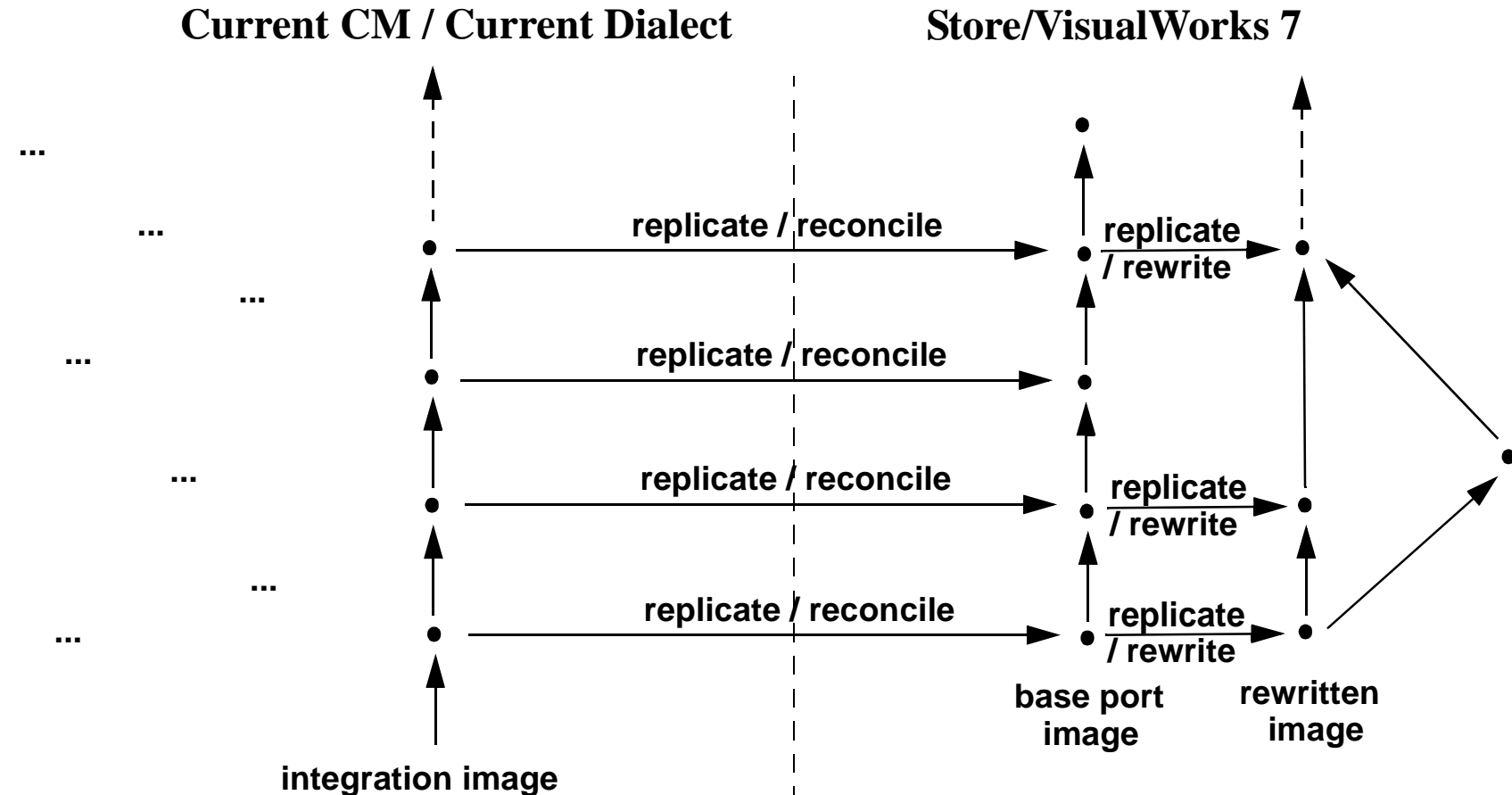
## Classes to Globals

Mapping between classes, globals and namespaces can avoid rewriting code

- CharacterEncoderPool is a NameSpace in VW7, needs to be a class in VW3 backports

# Replicate and Reconcile

Reconcile each integrated system increment onto the port.



**Also replicate base system: can't load but can compare your changes, dialects' changes**

# Glorp

**A framework for connecting SQL databases to Smalltalk**

- **declarative, no impedance mismatch**
- **natural Smalltalk style of coding**

```
session read: StorePackage where:
```

```
  [:each | each name = aName &  
    (each currentBlessingLevel = self replicationBlessingLevel)].
```

```
session transact: [session register: newPundle].
```

**See Alan's talk for (much) more detail**

## Store for Glorp

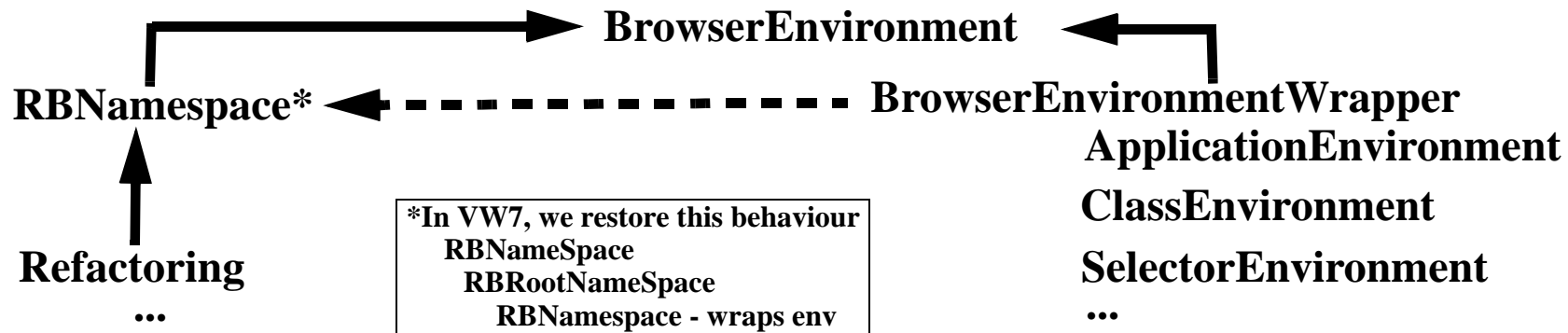
**Using Glorp to read and write to Store databases:**

- **available in several dialects**
- **wholly distinct from the Store code as far as the image is concerned (be aware of this)**

# Refactoring Browser Frameworks

The RB is in most dialects. Three of its frameworks can be exploited:

- **RB UI: shadow browse code in Store databases**
  - **BrowserNavigator** views an environment
- **Refactoring framework: export code refactored in RBNamespace but *not* in image**
  - **RBNamespace\*** presents a refactored view of code in an environment
- **BrowserEnvironment framework: define what code is to be exported**
  - **BrowserEnvironment**: the image; all other environments are wrappers
  - **BrowserEnvironmentWrapper**: view subsets (of subsets of subsets...) of the image
  - **RBNamespace** already looks very like a wrapper ...



# BrowserEnvironments

The easiest way to create code units because their UI and tools come free.

- use the environments for the dialect's code units if the mapping is easy
  - `ApplicationEnvironment`, `PundleEnvironment`, ...
- build environments using standard RB functions, `ParseTreeSearchers`, hand-coding
  - `ClassEnvironment`, `SelectorEnvironment`, `ParseTreeEnvironment`, ...
- dialect lacks bundle-equivalent: use `MultiEnvironment`
- dialect lacks extensions: use `MultiEnvironment`

Some protocol additions and fixes

- RB environments care who includes a class rather than who defines it

Lazy creation of `StoreForGlorp` pundles in background replication process

```
self replicationManager
  addToReplicationList:
    (anEnvironment onBaseEnvironment: self lineUpEnvironment)
  whenDone: ...
```

```
ConvertingReplicator>>pundleMatching: nonPundleFromNonStore
  ^super pundleMatching: nonPundleFromNonStore asPundle
```



# Where to Refactor

## Run refactorings in the integration image

- rewriting code to be dialect-neutral is the ideal solution
- feed back to the integration image via shadow browser or FileOut30

but often not possible

## Run refactorings in the image before replicating, then discard

- allows easy browsing
  - fully reversible: executing refactoring changes returns their undo equivalent
- ```
undoRefactoring := aRefactoring execute.
```

but the refactorings must not break the image.

## Run refactorings only in an RBNamespace

- replication environments wrap anRBNamespace
  - replicated code reports refactored class names, method names and code
- ```
(...Refactoring model: anRBNamespace ... ) primitiveExecute.  
"or use a CompositeRefactoring"
```

# Refactorings

The experience of rewriting code between dialects suggests refactorings for those dialects.

- capture as many dialect differences as possible in Refactorings
- some are trivial and specific: rename this class, that method
- some are general

```
"Convert all class names in specs to qualified names."  
... (aClass compiledMethodAt: aSymbol) resourceType = #canvas  
...  
rewriter := ParseTreeRewriter new  
  replace: '#literalNode'  
  withValueFrom:  
    [:aNode | RBLiteralValueNode value:  
      (ObjectNameWrapper objectNameWrapperFor:  
        (Smalltalk at: aNode value)) asQualifiedName]  
  when: [:aNode | aNode isLiteralValueNode  
    and: [...]]].  
^(rewriter  
  executeTree: (RBParser parseMethod: aSpecSourceString);  
  tree) newSource
```

# Performance

## To replicate and reconcile:

- **1000+** application package environments, each with an average of
  - **15** classes
  - **225** ordinary application methods
  - **6 large** application methods (can be > 64k)
  - **2.5** base overrides / extensions

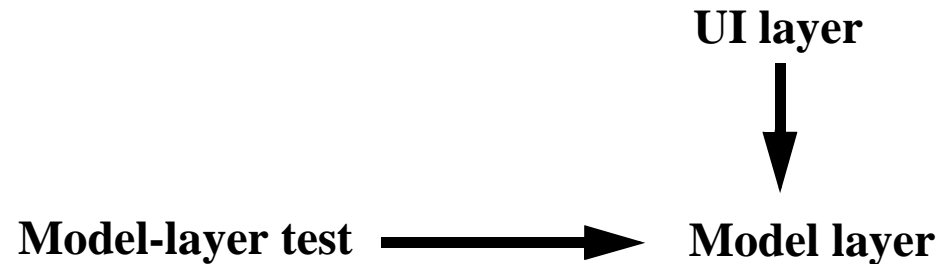
## takes

- **3 hours:** performant client machine, dedicated db server machine
- **6+ hours:** low-spec with other tasks
- **can also do**
  - **multi-threaded package replication, bundles built post-hoc**
  - **exploit source system CM**

**Overnight replication and build.**

# (More) Portable UI Tests

**The UI-layer and top-level model-layer tests act similarly on the model layer.**



## **In use, model layer**

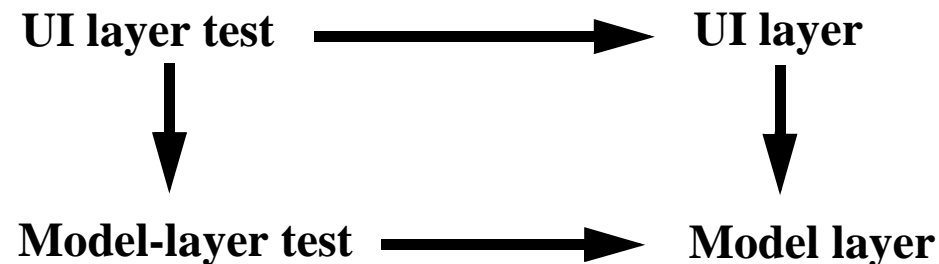
- **gets its values from the UI**
- **returns its results to the UI**

## **In test, model layer**

- **gets its values from the test**
- **returns its results to the test**

# (More) Portable UI Tests

Complete the commutative diagram ...



... by creating UI-layer tests as subclasses of top-level model-layer tests

```
SomeFunctionTest subclass: #SomeFunctionUITest
```

Inherit model-layer tests as UI tests

- override value setters/getters to set/get values in/from widgets
- override operation invocations to press buttons, select menu picks, ...

Add UI tests with very little (and reusable) extra code

- this little code is all the extra UI porting your tests need

# Summary

**A port needs to be as agile as a standard development (or more so :-)**

- **Large complex systems may have large complex ports**
- **Visible customers are developers, not end users: the bottom of the food chain**

**Keep your port synchronised**

- **Exploit configuration management**

**Keep NameSpaces invisible**

- **if you're starting without them, this must be doable**

**Solve problems where and when it is easiest**

- **which image / CM system knows what a rewrite / refactor needs**