A Smart, Object-Oriented Change Management System for Smalltalk

Lynn Fogwell, 2011

I. Introduction

This paper describes a new change management system for Smalltalk. Change management is, unfortunately, a rather tedious - maybe even a boring - topic. It is probably as exciting as reading about a new type of septic system. It might be useful, but it is not a stunning demo. Nevertheless, I am very excited about this change management system because of what it could do for the Smalltalk community. I believe it can transform Smalltalk culture as dramatically as indoor plumbing transformed human culture. My hope is that this system can transform Smalltalk into a mainstream development environment.

II. Why Change Management?

In the 1990's, Smalltalk was shown time and again to be an extremely productive language. Studies showed that fewer programmers could accomplish more by using Smalltalk. Prolific programmers loved the Smalltalk language. Yes, it was different but it was productive and people who learned it, loved it. So what happened? Why is Smalltalk not mainstream now?

I believe there are two reasons. One was that Smalltalk was proprietary and expensive. Thankfully, Smalltalk is finally open source and SMOOCH is open source as well.

Secondly, I don't think Smalltalk could scale past six or so programmers. A long-time Smalltalker, Ward Cunningham, was once asked why such a great language like Smalltalk had never become mainstream. He was quoted as saying "In Smalltalk, it's just too easy to make a mess." I agree. I specifically believe that the lack of an appropriate change management system is the main reason that groups ended up with a tangled mess. The lack of appropriate change management also hampers reuse and repackaging for delivery.

Existing change management systems and concepts are not appropriate for Smalltalk. Existing check-in, check-out systems are too restrictive and imply a fixed assumption of the packaging beforehand. Also, Smalltalk cannot be seen as a collection of modules. The granularity of individual methods is much too fine, with methods reused and scattered across many classes, to be adequately addressed by a module level change management system.

This paper presents a different concept for Smalltalk change management.

III. SMOOCH Basics

The SMart, Object-Oriented CHange (SMOOCH) system consists of multiple Smalltalk images connected to a MySQL database. The images send changes to, and receive changes from the database as they develop and share code. Changes are the fundamental objects of the system and are described below. Any SMOOCH image may also connect to another, remote SMOOCH database as read only. From

there, they can browse and transfer changes from the remote database to their local database.

The SMOOCH image no longer uses changes and sources files. Every element is represented as a change object that resides in the database. Every Trait, Class, SharedPool, and CompiledMethod in the image holds their database key and can readily produce their change object (#fetchChange) from the database. CompiledMethods no longer have a file and offset pointer at the end of their byte array. They hold a database key that points to their change, which holds their source code.

The SMOOCH image was built on top of Pharo1.0-10508-rc2dev10.01.2, from early 2010. Pharo was the smallest and simplest image I could find that was well supported and available on all development platforms. Pharo is a good and stable system and ran very fast on my MacBook. I am currently using version 5.1.37 of MySQL.

Globally Unique Database Keys

The database key for every change is eight bytes long. The first five bytes are a timestamp and the last three bytes indicate a unique developer id. If I assume that a developer is only creating code on one image at a time, then I can guarantee that the key is globally unique. It represents a fixed time and place.

The five-byte time stamp represents hundredths of a second since January 1, 1980 (roughly the birth of Smalltalk 80). This gives centisecond resolution for the next 300 years. For keys that are requested faster than hundredths of a second (i.e. automated code generation), the IdentityManager borrows into the immediate future. Within any image, the timestamp of a generated key is guaranteed to be unique. When building changes from Pharo code, I used the existing timestamp strings when I found them. I have noticed Squeak code from developers going back to the late 1990's.

The three-byte developer id's are keys to a MySQL developers table that holds basic information on the developer: name, initials, e-mail address, and login. If the login field is not NULL, then the developer is considered *active* and can use this login name and their password to connect their image to the local MySQL database. SMOOCH users given Administrator access can set up these accounts through their image.

The developer id's are analogous to IP addresses. There is a centralized master developers table that any local SMOOCH database can download. This is similar to a DNS table that can be cached locally. This centralization cannot be avoided because developer id's must be globally unique in order to share code across development databases. A centralized table does have some advantages. If a developer's e-mail address changes, they can register the change in the master table and that change will eventually be propagated to the local development databases.

Finally, there is a range of local developer id's that can be used without bothering to register with the master table. This is similar to the 192.168.x.x IP address range. If a local developer wishes to share his code at some future time, he would need to first register and receive a global id, and then easily convert all his keys in the local database to his new id. They would then be ready to share their code across other repositories.
Three id bytes allows for over 16 million registered developers who could share code. I currently have around 193 developer ID's that I derived from the old author strings in Pharo.

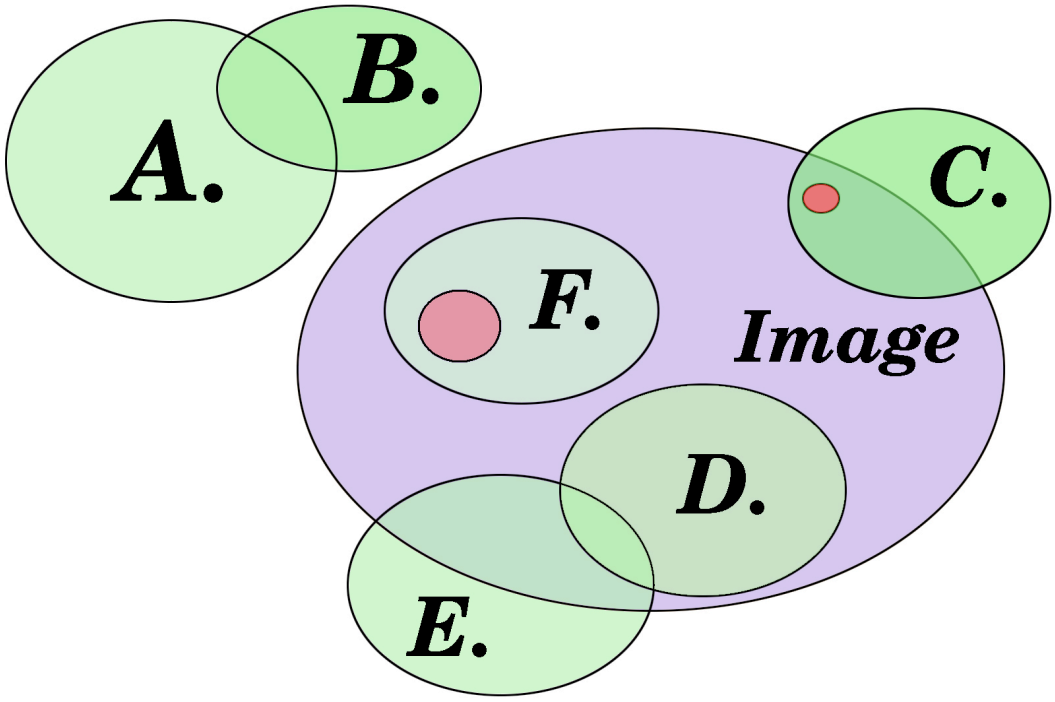
The log file
The image now writes to a .log file of the same name as the image (eg 'sample.image' and 'sample.log'). The log file is very terse and not critical to the image. If it is deleted, the image will create a new one and begin writing to it. It is mainly a collection of database keys to changes created in the image. It can also record changes that are *installed* into the image. Finally, the log records actions local to the image: DoIts and session information (QUIT, SAVE, SNAPSHOT). The main use for the log file is to recover from a crash. The RecoveryBrowser uses the keys in the log to look up the changes in the database. The changes relating to the local image actions are built directly from the log. In addition to looking back to the previous save, one can ask to see all changes since a given date and/or time (eg. yesterday at 10:30am).

The author has been using their current log file for three months and it is currently only 112 K bytes in size.

IV. New Change Management Concepts

SMOOCH promotes changes to rich objects that can be loaded, queried, and unloaded without ever being installed. Loaded changes can be asked many types of questions. Whole configurations can be loaded just to compare them with what's currently installed in the image. Loaded configurations can answer the changes that currently conflict with the image (installed changes). Most importantly, configurations can answer whether or not they can successfully install.

Overlapping configurations
A single change can now reside in multiple configurations. This means that configurations can be viewed as venn diagrams. One can also view the installed changes and loaded configurations as venn diagrams as well.



The above figure represents six configurations that have been loaded into the image (A - F). Configurations A and B are uninstalled, configurations C, E, and F are partially installed and configuration D is completely installed. The current installation status of configurations are always displayed in the change management interface. The small red circles, above, indicate conflicts between the configuration and the installed versions. This means that a different version of the same element exists between a configuration and the image, or between two configurations. I can ask to '**Find Conflicts With Image...**'. The developer can also ask to '**Find Orphan Installed Changes...**' which will return all installed changes that are not found in any loaded configuration (all of the purple, non-intersecting part of 'Image' above). This is useful in discovering how your image compares to a known, full-image configuration. All possible sections of the venn diagram, above, can be queried and browsed in the change management interface.

Loading tens of thousands of change objects can increase the size of the image, but the performance does not seem to suffer much. The base image file, with no changes loaded, stands at under 11 megabytes. If one loads the 26 configurations that make up the entire base image (around 37,000 changes), the image file grows to around 16 megabytes. Of course, unloading the configurations and saving the image will return the image file to its 11 megabyte size.

Locking

In SMOOCH, changes installed in an image can be locked. This means that they cannot be altered in any way. If a method version is locked, the developer cannot recompile it. They cannot install a new change over it. They cannot even move it to a different protocol. The locking mechanism in SMOOCH insures stability of key

components of your image. A developer can install new configurations and be assured that they will not unknowingly overwrite key classes and methods.

Version information is always displayed in the SMOOCH class browsers. Locked versions are displayed in **red** and quickly noticed during browsing. Classes, methods, or entire categories can be quickly locked and unlocked from the browser.

Smart Changes

A change represents a piece of code that is independent from any image. Once they are loaded, a developer can ask a change to install into an image (or deInstall from the image). One can also ask each change a number of questions:

Are you currently installed in this image?
Could you install into this image?
Can you be de-installed?
Do you conflict with whatever is already installed?
If installed, could you possibly alter existing behavior?
What are your external references?
Are you a local change or are you from a remote repository?

The changes are considered smart because they do extensive checking before installing themselves into an image. To guarantee a successful install, a methodVersion first checks if he #canBeInstalled. He checks the following:

Is my Class/Trait installed?
Do all of my references resolve? (he keeps a list of them)
Is there an existing method present and if so, is it locked?

Ask a ClassVersion if he can be installed and he checks:

Is my superlass installed?
Is the superclass type compatible with my type?
Do my Pool/Trait references resolve?
Is there a name space conflict (is my name being used as a Global or Trait)?
Does my state (instvars, classVars, classInstVars) conflict with any superclass?
If there is an existing class with my name installed,
    is it locked?
    does it have subclasses that have a compatible type?
    does my state conflict with any subclass?

These behaviors become more interesting when I apply them to groups of changes. Groups of changes answer the same questions as any individual change. Configurations with thousands of individual changes answer the same questions as above:

Are you currently installed in this image? (may answer ‘partially’)
Could you install into this image? (modeled with a ProjectedImage)
Can you be de-installed? (also modeled with a ProjectedImage)
Do you conflict with whatever is already installed?
If installed, could you possibly alter existing behavior?
What are your external references?

Are you a local change or are you from a remote repository?

If a change says it can be installed, it is guaranteed to install. If it can't, it will inform the user why. The straightforward questions to individual changes produce complex and rich answers when the same question is propagated across many linked configurations containing thousands of changes. Best of all, if a Configuration says it can be installed, it *will* install. No more walkbacks in the middle of a file-in.

V. Defining terms

A Smalltalk ***element*** is defined as a class, trait, shared pool, or method. Any creation of, or change to an element will generate a new, independent change, called a version. The version records the time stamp and author as its key and contains everything it needs to reproduce the element in any image. There is no link in a version to any previous version. The version history of an element can be produced with a database query, where all versions will be returned in time stamp order. The following version events are described for each element:

ClassVersion

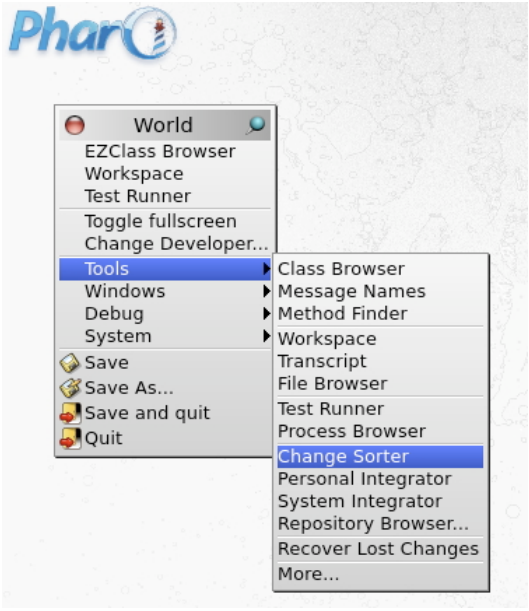
A class version records all of the state of the class. A new class version holds the following information and a new class version is created if any of the following are altered:
superclass - any change to the name of the superclass
class type - any change to normal, variable, variableByte, etc.
instance variables - any added or removed
class variables - any added or removed
class instance variables - any added or removed
pool dictionaries - any added or removed
referenced traits - any trait names added, altered, or removed
referenced meta-traits - any meta trait names added, altered, or removed
category - the class is moved to a new category
comment - adding, altering, or removing a comment

In the change management system, a class is defined by its name. This is tricky when a developer renames a class in the image. A rename action in the browser will produce a **removeClass** of the old classVersion and produce a new classVersion based on the new name. This will also produce new classVersions for all methods and for any subclasses. If any methods or subclass versions were locked, the rename function will be aborted.

TraitVersion

A traitVersion is defined by its name. It holds no state so it is able to reconstitute itself with the following information: category, comment, referenced traits, referenced meta-traits. Again, if any of this information is altered, a new TraitVersion is generated.

PoolVersion
For change management, the SharedPool in Squeak is a more attractive alternative to Dictionary objects. Pool variables are defined as class variables in a SharedPool subclass and therefore changes to pool variables are caught as a new PoolVersion. A new pool version will also be created if there is a category change or any comment change. Because of this Squeak mechanism, PoolVersions also support methods.



MethodVersion
MethodVersions define methods for both Classes and Traits, on both the instance or class side. A method version is defined by the class or trait that holds it and by the selector. A new methodVersion is generated if new source code is compiled, or if the method moves to a new protocol.
MethodVersions hold all external references found in the source code. That is: instance, class, pool, or global variables. SMOOCH can predict if a method will install without having to compile. This is critical when a developer asks a 5,000 change configuration if it can be installed. SMOOCH builds a model of the install and, using the variable references, can quickly and accurately predict the success of an install.

Configurations
Configurations are collections of the change versions described above. They are considered a static description of a unit of functionality. If one wanted to capture the functionality of the class Browser, they would include the ClassVersion and all MethodVersions of the class. It is the Configuration that captures both the state and behavior of a class. Configurations can easily hold dozens of classVersions and thousands of methodVersions.

Configurations have an owner and can only be opened and locked by them. Configurations are identified by their name and are locked by version and release. They can be opened for editing (called an edition) and multiple editions can be open at once on the same version/release. Configurations remember the parent version/ release they came from.

Configurations hold their changes in a predefined order that reflects the install order. TraitVersions are first, and ordered by references. PoolVersions are next (alphabetical order.) ClassVersions follow PoolVersions, and they are sorted by class hierarchy order. Finally, the methodVersions are ordered by class name and selector.

Configurations also allow the developer to specify install pre-actions and post-actions. These are optional DoItExecutions that are executed at the appropriate time. The developer can also define de-install pre-actions and post-actions. DoItExecutions

in the pre-actions perform a #signaledInstall, which means that, if they return false, it will stop the install or de-install of the configuration.

Configurations are hierarchical. They can reference prerequisite configurations that need to be installed first.

Remove Events
The abstract superclass RemoveEvent represents the common behavior of four significant change events: RemoveClass, RemoveTrait, RemovePool, and RemoveMethod. Unlike versions, remove events are not held by configurations. If a remove event is sorted into an open configuration, it will act on it by removing the corresponding version from the configuration, if it holds one. Remove events are held by changeSequences. Remove events display like all other change objects in the interfaces. They will show as installed if there is no corresponding element in the image. Installation of a remove is not trivial. It will be blocked if the corresponding element is locked. ClassRemove's will not happen if there are subclasses. TraitRemove's will stop if there are any trait references in the system. Asking a ChangeSequence if it 'Can be installed' will catch any of this before any install is attempted.

DoItExecution
DoItExecution's are typically image only and are therefore stored in the .log file instead of the database. If a doItEvent is shared in a changeSequence or as a pre or post action in a configuration, then it will be committed to the database. DoItExecution's will never show as installed because it is impossible for the change management system to know what that means. DoItExecution's do have a receiver context that they will attempt to execute under when they are installed.

ChangeSequence
While configurations represent a static definition of functionality, ChangeSequences represent an ordered collection of delta changes. Delta changes can be versions and can also be remove events or doIts. They are a set of changes that take an image from state A to state B. The order of the changes in a changeSequence matters. The class **ImageChanges** is a special instance of a ChangeSequence that collects all changes made in the developer's image. If I make five changes to a method and then remove it, the ImageChanges will contain five methodVersions and then a RemoveMethod change. This is similar to the change set and change sorter idea in earlier Smalltalks.

Additionally, a developer can *derive* a changeSequence that will take him from one configuration to another. This is useful to quickly see the differences between version 1 and version 2 of a configuration. If version 2 has two less methods and a change to another, the derived change sequence will contain a MethodVersion and two RemoveMethod changes. In other words, if I have version 1 installed, I can install the derived, three-change ChangeSequence and consider myself to have version 2 loaded. If I compare two completely different configurations, the resulting

changeSequence will have a collection of removes for the first configuration and then all of the version events of the second configuration.

Editions
Class and Trait editions are not changes, but placeholders. A class Edition is held by a configuration. A single class edition represents the classVersion and all methodVersions currently installed for that class. Sometimes a developer knows that the currently installed class is exactly what he wants in a given configuration. They can assure they have the currently installed configuration by adding a classEdition to the configuration and then converting it to class and method versions before committing the configuration to the database.

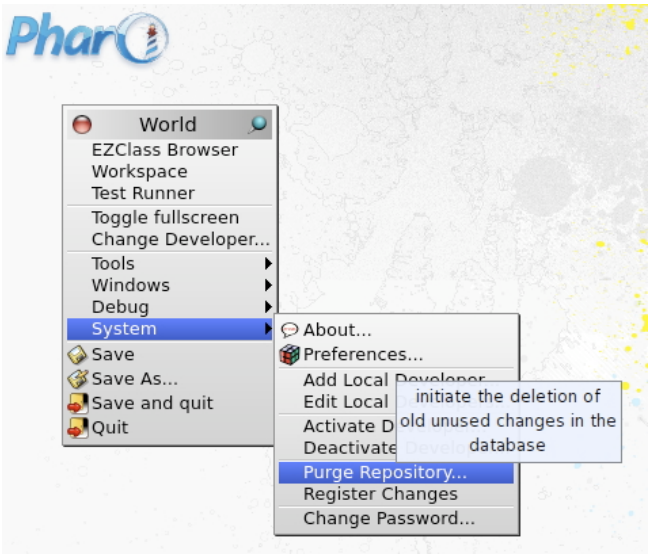
VI. Usage Scenarios

Communicating day to day work
Daily communication among developers assumes two things. First, their images are largely the same. They have mostly the same stuff installed and therefore only need to communicate the differences, or delta changes of their recent work.

Secondly, communication among developers working on the same project assumes that no one update of code will be very large. Daily communication should be on the order of dozens of changes, not hundreds.

These assumptions are related. ChangeSequences are meant to efficiently synchronize images that are already similar. They assume a similar starting state and then sequentially update or remove code to bring the target image into the same state of the original developer. If developers are communicating with hundreds of changes, they have likely lost the efficiency of changeSequences and need to communicate with configurations.

A unlikely analogy may be found in video compression where very large amounts of data need to be communicated. Video compression makes use of the assumption that the differences between any given video frame and the following few frames is minimal. They can save a great deal of space by only storing the delta changes from a given key frame. This system breaks down when there are a lot changes between frames (eg. a moving camera.) Tracking and storing delta changes then becomes very inefficient and the codec is better of to simply record the data of each frame.



ChangeSequences are good at communicating day to day work. They

should be named for the bugs they fix or the features they implement. After installation, they may be kept around for integration into configurations.

Packaging your configurations for release

Class and Method Ownership is an important part of change management. Much of group programming involves dividing responsibility and ownership of different pieces of code. This is also true in Smalltalk, but must be viewed in a different way. Every Smalltalk programmer owns his entire image. The Smalltalk programmer has the freedom and power to quickly and easily alter any method in his environment. This is one of the many appeals and power of Smalltalk. This freedom to experiment should never be inhibited by forcing any packaging overhead while working in the image. One should never have to open a version or even think about versioning before adding a halt to any method.

The enforcement of ownership and limits, comes when it is time to package and integrate your code. When the functionality the programmers have created needs to be formally shared outside of their group. The **personal integration task** is when a programmer sorts changes from change sequences into *their* configurations. Different programmers may have written methods and fixes in your classes. Those were communicated in the day to day change sequences. But only the owner of the configuration can add those method changes to their configuration. The configurations enforce ownership and most ownership issues are detected and resolved at that point.

The product of a development project are a set of configurations, not change sequences. Configurations are a more static description of functionality, meaning that they are less dependent on the state of the image they will install into.

Building hierarchies of configurations

System integration is a process that deals with configurations and all of their prerequisites. Larger projects may have many configurations that are divided up by functionality but need to be installed in a particular order or based on particular options. The **system integration task** is seen as a separate and distinct task from building the individual configurations. From the system integration window, nested prerequisites can be viewed as a tree structure and all loaded at once. Conflicts can then be detected throughout the configuration hierarchy - without anything ever being installed. For example, two configurations may have created two versions the same particular String method and assumed ownership of the extension. The system integrator will catch such things.

A set of linked configurations have the ability to completely define an entire image. The **SMOOCHbase** configuration, with its prerequisites, completely defines the entire base image. If one has an image in an unknown state, they can load a known image configuration hierarchy and then run queries to understand the differences between their image and the known configuration.

Importing configurations from remote repositories
To make code sharing convenient, any developer may access, read-only, any other open SMOOCH repository on the internet and browse their fixed configurations. They access a remote MySQL database as a 'guest' user. A developer may only browse and adopt (import) configurations from remote repositories that have been fixed to a version and release. Only when a configuration is fixed and unchangeable, can it be propagated outside of a local repository.
When a developer adopts a configuration from a remote site, they are adopting it for their entire group. That configuration will now be accessible in the local database by everyone in the group.
The Smooch login names of 'root' and 'guest' are reserved. The Smooch administrator can turn guest access on and off from the World menu (System>'Allow Guests'.)

Connecting an image to a SMOOCH repository
An existing SMOOCH image can be connected to any SMOOCH repository. Once connected though, there may be installed changes in the image that are not present in the repository. The developer can easily check for the presence of all installed changes in the repository by executing:

```
LocalRepositoryManager verifyAllImageKeys
```

The process takes about 30 seconds over a LAN connection.

VII. Strengths/Advantages

The following sections clarify what the author sees as the key advantages to the SMOOCH System described in this paper.

Separates functional grouping from packaging
Unlike Monticello, class categories and method protocols should reveal a functional organization that is separate from how things are packaged. The two concepts should not be intermingled.

The String method #hasSuffix: belongs in the protocol 'testing' because it makes a comparison and returns a boolean. That is a functional grouping. This method may have been written as part of a file system package, but I do not want it hidden in the protocol 'file system extensions'.

There are thousands of classes and methods and users need to learn their function. New users need elements strictly organized based of what they do. It is not helpful to confuse this with how they are packaged or why they were written. Packaging information is a necessity but it has no place in the functional organization. Good functional organization is one tool that a programmer should have in tackling the steep learning curve of Smalltalk.

Start with a small base image

Smalltalk has a steep learning curve that is made worse by images with literally thousands of classes. Pharo was much leaner that Squeak and it still had 3,200 classes (and traits). Thousands of classes are overwhelming to a newcomer and the classes end up acting as noise. It is more difficult for the programmer to find the fundamental classes they need to know about. They will wade through many more methods when chasing senders and implementors. The learning curve in Smalltalk has always been a challenge, but having literally thousands of classes and scores of thousands of methods in an image turns a steep learning curve into an unscalable cliff. The Smooch base is now down to 1,260 classes. Eventually, I want to get the base image under 1,000 classes. That would be similar to the old ParcPlace image - which I always thought was too big.

What does a small Smalltalk base, or starting, image *need* to have? The base should contain the classes that define the Smalltalk language (eg. the “Blue Book” classes) as well as the fundamental classes that make up the Integrated Development Environment (IDE). These would include exception handling, the unit testing framework, external communication, change management, a multilingual framework, etc. Finally, the base image needs a single, simple, windowing framework with a small footprint. Years ago, I learned how to build Smalltalk interfaces by looking at the code and comments of the base image interfaces. If one sees the base image windows as interface teaching tools, then lightweight, hand-built windows become preferable to automated window builders.

To further support learnability, the base should remain very readable, with no pieces of old broken framework, duplicate, alternative browsers, or legacy methods. The base image should also be very stable and altered very infrequently. Highly reused packages are the very ones that should change the least - and the base image is the most reused package of all. A small base minimizes necessity of frequent changes.

Finally, if change management actually works, then the developer should be able to take a base image and quickly install the configurations they are interested in. They should be able to de-install configurations just as easily. SUnit Test configurations are a classic case of install, run the tests, and de-install. Test configurations should never be installed in the shipped, base image.

Repackaging is an indispensable part of reuse

A Smalltalk programmer can never predict how a class or framework they have written will be reused or re-factored. One of the strengths of Smalltalk is in the ability to reuse objects in novel and creative ways. In the same way, programmers can never know how their code will be repackaged.

For example, one programmer writes a package to do versioning. One of his classes may parse a string into a versioning hierarchy. A second programmer wants to use that parsing class to encode and decode a string into a completely different type of hierarchy. The second programmer wants to use the one class, and he is not

interested in the 20 other classes in the original package. What is he to do? He wants to add the one class to his package. He would also like the freedom to have both packages easily loaded together in the future.

Repackaging is an essential part of real reuse. The way something is packaged largely determines the way it is meant to be used. For reuse to work, changes must have the ability to reside in more than one package. Multiple packages with overlapping, identical classes and methods need to be allowed. Multiple packages with conflicting classes and methods need to be easily detected.

Data mining and analysis of finished projects
Changes stored in the database provide numerous opportunities to analyze the development process. Every single change in SMOOCH is timestamped and kept. SQL queries can quickly and easily answer a number of questions about the changes. One can look at which class or subsystem caused the most intermediate changes. A set of queries could graph when, during the week, the most development work was done. One can look at the number of changes generated per developer.

After any analysis is finished, there is a mechanism for purging unreferenced changes in the database. All changes are deleted that are not referenced in any Configuration, ChangeSequence, or in any image used by the group.

Never again get a walkback error when installing
Currently, Smalltalk cannot guarantee a successful installation of a package. Worse, file-ins can often end in an error walkback, leaving part of the package installed. Aborted installs can be hard to back out of.

SMOOCH can *guarantee* a successful install by choosing the 'Can be installed?' menu command. Even attempting an install without testing for success first will never give a walkback window. Every individual change tests if it can be installed before actually installing. If it cannot, it halts the install process and pops up a modal dialog to explain why. The developer can then easily select de-install to remove the changes installed to that point.

A configuration may have overwritten existing changes in the image. If 'logging installs' is activated, the developer can de-install the installed changes, which will remove the installed change and re-install any previous change.

These features, along with locking, are meant to make installs safe, easy, and convenient. They are meant to encourage experimentation with the work of others.

Reuse - Blessing and Curse
The *majority* of SMOOCH development time was spent figuring out existing frameworks in Pharo. Duplicate packages were untangled and de-installed. Methods in many classes were re-sorted into consistent protocols. Odd methods in Object and other fundamental classes that didn't seem to be used were cautiously deleted. The

developer's focus is on making sure every class and every method has a clear and appropriate use in a base Smalltalk image. This task is not finished.

The less-is-more argument has already been made. I now need to explain my obsession with simple, understandable code.

Reuse is a double edge sword - it has the potential for great time savings, but reusable packages also have the potential to waste thousands of hours of time. A reasonably well written package may have protocol and comments that are somewhat confusing, it may take a developer an extra 20 minutes to figure it out. That is not terrible, but if reuse is a reality, you can multiply that twenty minutes by literally hundreds of programmers, going through the same learning delay. That can add up to hundreds of wasted man hours.

In addition, developers incur costs in even investigating reuse. The build versus reuse decision itself requires time. There will necessarily be configurations that I install and investigate that I don't use. All of this argues for a very high standard for resuseable packages - especially for popular packages.

The battleground for style

I have worked in a number of Smalltalk development groups where conflicts arose over appropriate coding style and frameworks. The author understands the subjective nature of this issue. There often isn't a right and wrong way to do to things - just different. The Unix operating system has a similar issue in that it can be constantly redefined and expanded. There end up being various 'flavors' of Unix that are the adopted starting point for various groups (Sun/OpenBSD, Solaris, HP/UX, etc)

SMOOCH is an attempt to be the platform where various flavors can be born. This is another reason for an obsession with a small base image. A small base gives a smaller base of contention. There is less to correct and reconfigure when you install your flavor of Smalltalk frameworks and interfaces on top. Also, with common change management, configurations can port more easily across different 'flavors' of Smalltalk images.

VIII. Immediate Goals

I have been using the databased SmOOCh changes for about a year, and I am comfortable with relying on them on a large scale. The parts of the Smooch image that need immediate work are as follows:

1. Repair File in and File out, so that code can go between Smooch and other Smalltalks.

2. Continue to pair down the base image: rewrite the Debugger window.

3. Repackage many of the Pharo frameworks as configurations to load into the base. This requires a familiarity with the framework code that I do not have.

4. To share Smooch with as many people as possible, and see where it goes. I have been programming on an island by myself for far too long.

5. I plan to use Smooch as my development environment for consulting. The best thing people can do for Smooch is to simply use it as a productive development tool.

Finally, I would love feedback and advice. Feel free to e-mail your questions, comments, and criticisms.

Thanks for your time,
Mr. Lynn Fogwell
Raleigh, North Carolina, USA
fogwell@bellsouth.net

