

The Genesis of Pharo:
The Complete History of Pharo with Git

Max Leske
Software Composition Group
University of Bern, Switzerland

March 22, 2011

Abstract

In the Smalltalk dialects Squeak and Pharo it is difficult to view changes to source code that occurred across major releases. This poses a problem for development as well as for code analysis: Developers often need to be able to revert code to a previous version and for code analysis there should ideally be a single source of information. We solve the problem by building a GIT repository with the complete source code history of Pharo.

To build the repository we propose two tools: GITFS and PHAROGENESIS. GITFS is an implementation of GIT written entirely in Smalltalk. With GITFS we can create, read and manipulate GIT repositories. PHAROGENESIS uses GITFS to build a GIT repository from the source code that PHAROGENESIS extracts from the source files.

We also show how GIT can be used to provide a single point of access for all source code from Squeak 1 through Squeak 4 and Pharo 1 using GITFS. The source code database we built with PHAROGENESIS is available on github (<http://www.github.com/pharogenesis/pharogenesis.git>).

Acknowledgments

I thank Prof. Dr. Oscar Nierstrasz for his support and for heading a great research group. I am especially grateful that he opened up the world of Smalltalk to me and my fellow students.

My thanks also go to Lukas Renggli who never doubted that I could complete this work. He was always there to answer any question and point out alternatives when I got stuck. Lukas is an inspiring tutor and I consider myself lucky to have worked with him.

I would also like to thank Jorge Ressoa who was my tutor on the final stretch. Jorge always got me back onto my feet when I felt that I would not be able to finish in time. He also never got tired of rereading my thesis, restructuring it and making helpful comments. Thank you Jorge for being so patient.

Special thanks also to my friend Matthias Wüthrich for his great feedback on intelligibility.

Last but not least I want to thank my girlfriend Eliane and all my friends for the support and the countless times they had to hear a “no” from me when they wanted to go out.

Contents

Contents	ii
1 A Discontinuous History	1
1.1 How Squeak Forgets the Past	1
1.2 Our Solution to Squeak's Discontinuous History	4
2 Rebuilding History with GitFS	5
2.1 The Design of GitFS	5
2.2 Git Internals	6
2.3 Filesystem internals	8
2.4 GitFS Architecture	9
2.4.1 Implementation of Plumbing	9
2.4.2 Implementation of Porcelain	10
3 Modeling The History With a Git Repository	12
3.1 Reading *.changes and *.sources files	12
3.2 Extracting The Source Code	13
3.3 Representing The Class Hierarchy as a Directory Tree	15
3.3.1 Encoding File Names	16
3.4 How To Use Pharogenesis	16
3.5 A Method Version Browser For Pharogenesis	18
4 Conclusion	22
5 Future Work	23

1 A Discontinuous History

The Smalltalk platforms Squeak [Blac07a] and Pharo [Blac09a] store their source code in a way that makes it difficult to view code changes that occurred before the last major release of the platform. The package manager MONTICELLO [Monticello] offers only a partial solution to this problem.

1.1 How Squeak Forgets the Past

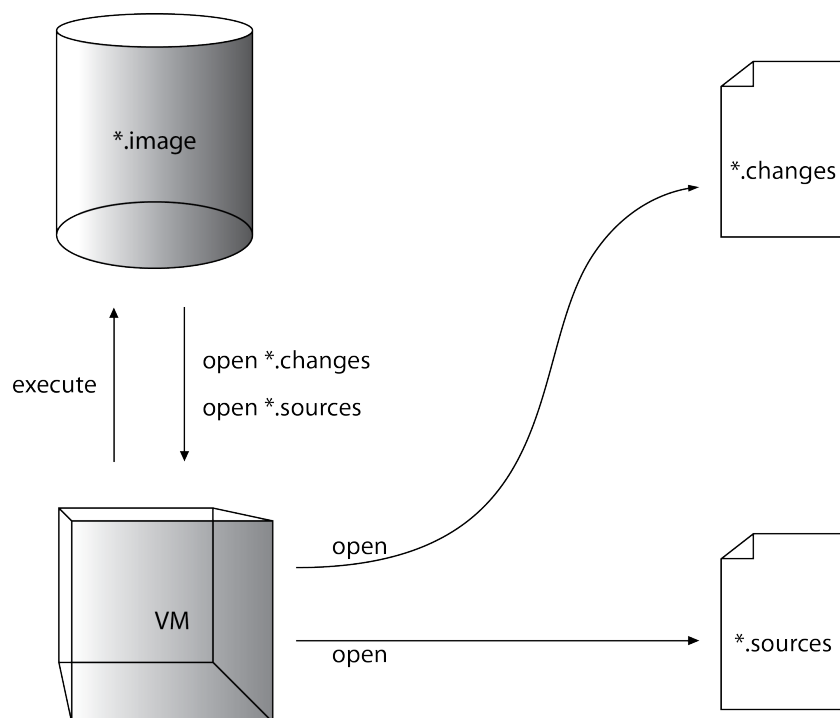


Figure 1: The image is executed by the virtual machine (VM). For source code lookup the image asks the VM to open the ***.changes** and ***.sources** files on the file system.

The Squeak project emerged from Smalltalk-80 in 1995 as a development environment for educational software. Pharo is a fork from Squeak 3.9 that came to life in 2008. Both Squeak and Pharo depend on three files to keep track of changes (see figure 1):

- The ***.image** file is a snapshot of the compiled source code and running programs that are executed by the virtual machine.
- The source code is stored in a file called ***.sources** and can be shared by multiple images.

- Changes to the source code are stored in a file called ***.changes** that is specific to an image. Changes include additions and removals.

Some mechanisms that were originally introduced to reduce memory and storage footprints of source code are still in use in Squeak and Pharo. One such legacy is the “changes file mechanism”: Since its first release Squeak has provided a mechanism, based on the ***.sources** and ***.changes** files that allows users to track changes made to the image. Today, code browsers still use the same mechanism to provide a history of changes and a means to revert to a previous version of the code. Unfortunately, the changes file mechanism cannot be used (without additional work) to list versions older than the last major release.

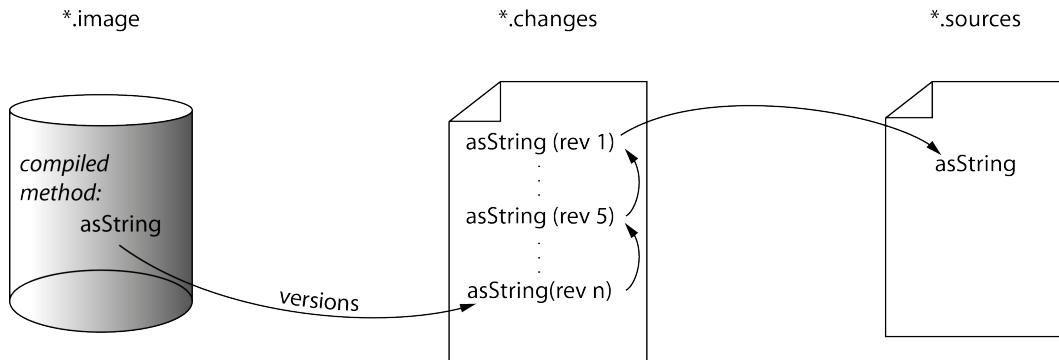



Figure 2: The source code displayed in code browsers is fetched from the ***.changes** file (latest revision). To display earlier versions the version browser follows the pointers through the ***.changes** file and ultimately to the initial version in the ***.sources** file.

The changes file mechanism works as follows (see figure 2): The compiled method in the image points to its source code. The source code will either be in the ***.changes** or ***.sources** file depending on whether the method has been changed. Every method version in the ***.changes** file points to its ancestor and the oldest version points to the original source code in the ***.sources** file. It is thus possible to retrieve every version of the method by following the chain of pointers. The downside to this approach is that it is not possible to see changes that were made *before* the creation of the ***.sources** file. The ***.sources** file is simply a snapshot in time of the source code and is recreated for every major release.

The Squeak ***.changes** files were not designed to store information about changes that occurred before the last major release of the platform. As a consequence, there is no way to list all the changes that were ever made to a specific fragment of source code without writing a program for this purpose. Our solution provides an infrastructure that can be queried directly.

With the release of Squeak 3.9 developers began to version the Squeak source code with MONTICELLO. MONTICELLO uses packages to version source code. A package is

a package  always contains the full source code of its members.

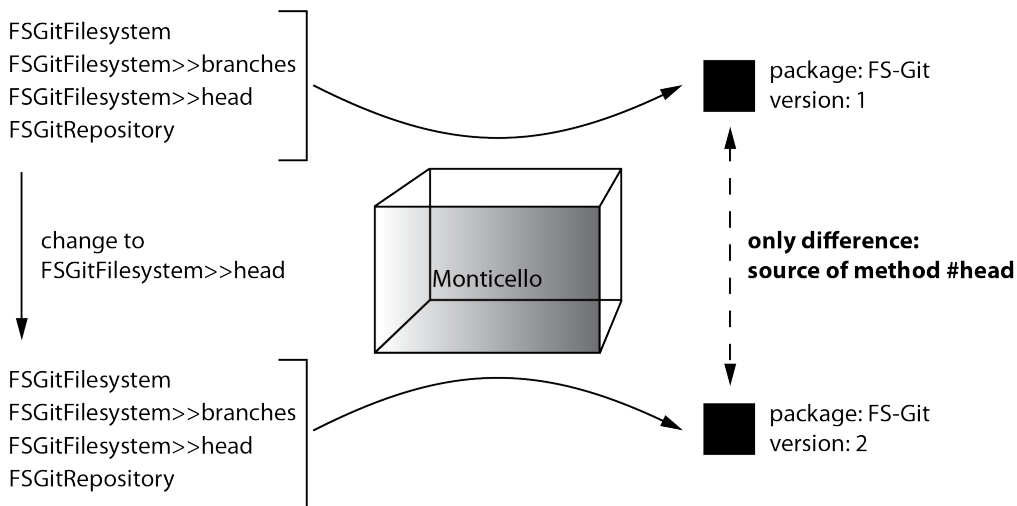


Figure 3: Every MONTICELLO package version contains the full source code of its members. The two versions of the package “FS-Git” have the exact same contents except for the change in the method `#head` (e.g. the source code of the method `#branches` is stored completely in both package versions).

a group of classes, a package version a snapshot of that group. Each package version contains the full source code for each class which means that there are no dependencies between different package versions. Any change to the source code of a package causes MONTICELLO to create a new snapshot (see figure 3). MONTICELLO helps developers to work around the problem of discontinuous history: it is much easier to find old versions of code than before because the source code is stored independently of the image (`*.changes` files are specific to an image). Nonetheless, MONTICELLO has the following problems:

- The source code that existed before the introduction of MONTICELLO is not available in MONTICELLO packages.
- To access a single item in a specific package version the complete package version has to be loaded.
- The duplication between package versions is often high due to small changes between versions.
- Renamed methods and classes are not tracked.
- Code moved between packages is not tracked.

- MONTICELLO is tied to a Smalltalk environment. There are no applications outside of Smalltalk environments that know how to access a MONTICELLO package. This restricts code analysis with tools that are not written in Smalltalk because extra work is needed to access package contents.

1.2 Our Solution to Squeak’s Discontinuous History

We solve the problem of the discontinuous Squeak history with the version control software “GIT”. Using GIT we build a repository with all the information in the ***.changes** and ***.sources** files of all previous Squeak and Pharo versions. This approach has the following advantages:

1. Unlike MONTICELLO packages, GIT commits allow access to single entities inside the commit without the need to load the entire commit data. This is an advantage because:
 - less data needs to be loaded than with MONTICELLO to access a single entity. Therefore, accessing a single entity takes less time and uses less image memory.
2. It saves disk space: Every entity is only stored once thus different commits can reference the same entity without duplication.
3. GIT is independent of Smalltalk. It is available on various platforms and a multitude of tools and services exist that help users to manage GIT repositories. These are some of the available tools and services:
 - “github”, “gitorious”: hosting services for GIT repositories
 - GIT grep: a built-in command that lets you search source code across all versions
 - GIT clone: a built-in command that lets you query data locally rather than on a remote machine
 - “GitX”, “Git Extensions”: graphical user interfaces that visualize changes

The name of the GIT repository we are going to build, and at the same time the name of the tool we use to generate the data for that repository, is PHAROGENESIS.

2 Rebuilding History with GitFS

Our goal is to use PHAROGENESIS to build a GIT repository which holds the source code of Squeak and Pharo and the changes made to it. This requires a way to create and modify GIT repositories from Smalltalk. We propose a pure Smalltalk implementation of GIT called GITFS.

2.1 The Design of GitFS

We built GITFS with the following design objectives in mind:

1. Provide simple read and write access to GIT repositories through an object oriented library.
2. GITFS has to be fast and efficient.
3. Implement GITFS entirely in Smalltalk to make it extensible and maintainable.
4. Provide full compatibility with the GIT command line tools.

The original implementation of GIT was written in C, Bourne Shell and Perl. One option to design GITFS is to simply execute GIT commands externally and process the results. This would relieve us of having to know how GIT works internally (e.g. how objects are stored). However, there are downsides to executing GIT externally:

- GIT would have to be installed for GITFS to be operational.
- The indirection of creating input data for GIT commands, executing GIT commands externally and parsing the output data makes this approach slower than if we implement the functionality in Smalltalk.
- GITFS might not work with other versions of GIT than the one used for development if certain implementation details change.

GIT provides commands which are not normally used in day to day work. Those commands include maintenance tasks and special operations on branches. It is not our goal to implement every command GIT offers. Instead we want to provide a system that is useful but still easily maintainable and that provides the functionality we need to create and use the PHAROGENESIS repository. Hence, GITFS will not include every GIT command.

We want to be able to open a view on a GIT repository using the standard tools in Pharo. A possible model for such a view is a file system. Using a file system view the user can work with files and directories just as would be the case outside of the image. The problem here is that we do not want the user to work on the files directly because we have no way to hide or prevent access to GIT specific files. The FILESYSTEM [Putn10a] library by Colin Putney solves this problem for us. With FILESYSTEM we can create

a virtual file system to which the user has access. From the point of view of the user this virtual file system is identical to the one the GIT repository actual resides in. The difference is that as developers we have full control over what the user sees and how we interact with the repository.

We conclude:

- It makes sense to replicate GIT in Smalltalk.
- We will implement only important and often used GIT operations in GITFS.
- We will use FILESYSTEM to allow users to work with files and directories.

The following two sections give an overview of how GIT and FILESYSTEM are built and how they work.

2.2 Git Internals

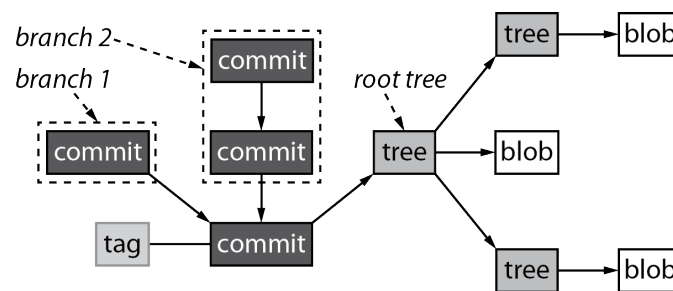


Figure 4: In the above illustration every GIT object type has a different color scheme. A tag references a commit; a commit references a tree (root tree) and a parent commit; a tree reference trees and blobs. Two commit sequences (“branch 1” and “branch 2”) share a common parent commit.

A GIT repository is initialized for a specific directory in a filesystem: the base directory. The “base directory” usually contains a directory named “.git” that represents the repository. The “.git” directory contains the commits that have already been made together with special files that GIT needs to operate. Every file and directory inside the base directory (except for the contents of “.git”) can be versioned with the repository. The content of the base directory (not including the “.git” directory) is called the “working copy”.

GIT uses four kinds of objects: “blobs”, “trees”, “commits” and “tags” (see also figure 4). A blob stores the user data; often blobs hold the contents of a file or, in the case of PHAROGENESIS, source code. Blobs do not store any additional information. Every blob is referenced by a tree object. Each tree has a list of entries and each of these entries maps a name to another tree object or a blob. Thus trees and blobs can mimic

the working copy where trees are equivalent to directories and blobs are equivalent to files. We call the tree that knows the entries of the base directory the root tree.

A commit object represents a snapshot in time of the working copy. Every new commit is the child of one or more parent commits (with the exception of the first commit). Each commit contains:

- a reference to the root tree
- references to its parent commits (except for the first commit)
- information on the author of the changes and the time of the modification
- information on the committer of the changes and the time of the commit
- a message describing the changes

A tag is an object that points to a specific commit. Tag objects make it easy to mark commits. They can also contain arbitrary additional information.

A sequence of commit objects is called a “branch” and every branch has a unique name. One use of branches is to develop different systems with a common base. Pharo for example is a branch from Squeak: both share common source code. Branching works by simply allowing multiple commits to have the same parent commit (see also figure 4).

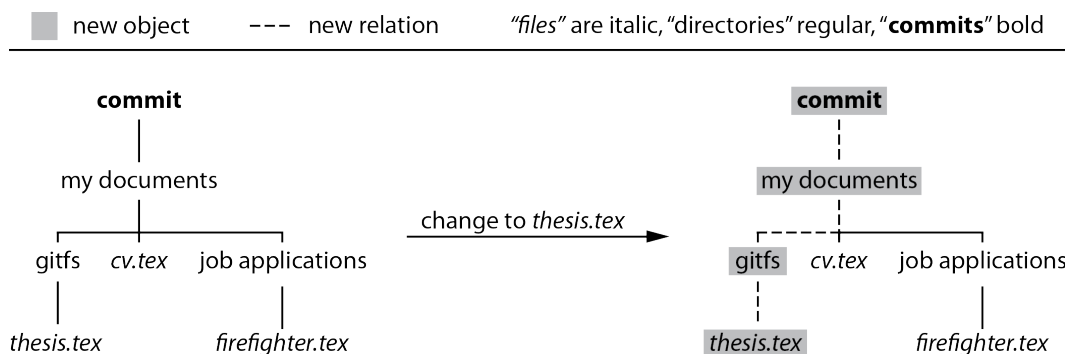


Figure 5: The signature of a GIT object depends on the signatures of the referenced objects. If the file “thesis.tex” is changed new objects will be created for all the objects directly or indirectly referencing the file.

Every GIT object has a unique identifier we call the “signature”. The signature of an object is a 40 character hexadecimal hash (SHA-1) that is calculated using specific rules for each of the four object types:

- the signature of a blob is the hash of its contents;
- the signature of a tree is the hash of its entries. An entry consists of a name (e.g. a filename) and a signature (blob or tree signature);

- the signature of a commit is the hash of the commit information and the signature of the root tree;
- the signature of a tag is the hash of the tag information and the signature of the referenced commit.

The calculation rules ensure that every change will produce a different signature for every object associated with the change (see figure 5).

Every object is stored in a file. The first two characters of the signature (“00” - “ff”) denote the parent directory; the remaining 38 characters become the filename. These files are referred to as “loose objects”. The contents of loose objects are compressed with a deflating algorithm.

The object signature has another purpose apart from uniquely identifying objects: The signature is used as a reference mechanism. Whenever one object references another this means that it stores the signature of the other object. This mechanism is also used by special references. A branch reference for example points to the last commit of a commit sequence. A repository can have an arbitrary number of branches. The “HEAD” reference points to the tip of the active branch.

GIT repositories have the tendency to produce many files (each change to a blob entails new tree and commit objects). More files mean more fragmentation and longer access times. For these reasons GIT provides the possibility to pack a repository. The packing algorithm stores loose objects into a single file with a *.pack extension. The packing technique is “delta based” which means that similar objects will not be stored fully but as differences to each other (loose objects are always stored fully). The packing algorithm also produces an index file (*.idx) which contains pointers into the *.pack file to accelerate lookups. The packing of a repository can be initiated by executing `$ git gc` (gc stands for “garbage collection”) on the command line. `$ git gc` will also remove all the loose objects that have been duplicated in the *.pack file (“collect the garbage”).

The architecture of GIT uses two layers called “plumbing” and “porcelain”. The plumbing layer itself is already a fully functional GIT. However, plumbing is complex to use because its purpose is to structure the program code. Porcelain is built on top of plumbing to simplify access to GIT. The porcelain layer can be thought of as the user interface to GIT.

2.3 Filesystem internals

There are four core classes in FILESYSTEM: `FSFilesystem`, `FSPath`, `FSReference` and `FSDirectoryEntry`. `FSFilesystem` is an abstract class that defines methods every filesystem inherits or needs to implement. `FSPath` is responsible for handling resource locators. A path like “/directory/file” tells the filesystem where to find “file”. `FSPath`

can also discern filesystem specific delimiters like the UNIX delimiter “/” and the Windows delimiter “\”. `FSReference` combines path and filesystem objects. By using a reference it is possible to point to a file or directory without having to explicitly specify the filesystem. The class `FSDirectoryEntry` holds meta information about a node in a filesystem. The meta information includes data like the last modification date and the size of a file.

The `FILESYSTEM` package includes two implementations ready to use: `FSDiskFilesystem` and `FSMemoryFilesystem`. The former allows access to disk filesystems while the latter creates a filesystem in the image memory. To the user there is little difference in accessing either of them because both filesystems inherit from `FSFilesystem`.

By subclassing `FSMemoryFilesystem` we can show the user a filesystem view on a repository over which we have full control. Additionally we can override certain methods where we need to change or add behaviour.

2.4 GitFS Architecture

The Architecture of `GITFS` mimics the architecture of `GIT`: one layer implements the `GIT` functionality (plumbing) and a second layer serves as the user interface (porcelain).

2.4.1 Implementation of Plumbing

We base our implementation of plumbing on `GIT FOR SQUEAK` [[Garn10a](#)]. `GIT FOR SQUEAK` is an implementation of the `GIT` plumbing layer for the Squeak platform written entirely in Smalltalk. It can create and modify `GIT` repositories and read `GIT` packs.

`GIT FOR SQUEAK` uses `FileDirectory` to access disk filesystems (`FileDirectory` is the interface used in Squeak and Pharo to interact with disk filesystems). For `GITFS` we replaced references to `FileDirectory` with references to `FILESYSTEM` for two reasons:

- The implementation of `FileDirectory` is very close to the filesystems it accesses. Additionally, `FileDirectory` uses strings instead of distinct objects to represent filenames and directories. The implications of these two points are:
 - users of `FileDirectory` require knowledge of the filesystem that is being accessed;
 - `FileDirectory` is not reusable or extensible. New functionality cannot be added in an efficient way and the interface cannot be reused in a different context.
- We do not want `GITFS` to use two different libraries that share functionality. Doing so would increase maintenance costs due to a higher error rate during development.

We introduced two additional changes to GIT FOR SQUEAK:

- GITFS uses object references where possible to support the Smalltalk object model. The original GIT system and GIT FOR SQUEAK both use GIT signatures to establish references.
- GITFS loads objects lazily whenever possible. Lazy loading means that only as much data is loaded as has been explicitly requested. This helps us to minimize the amount of data that needs to be fetched. Not using lazy loading can be a problem for large GIT repositories because, for the user, it can lead to long periods of waiting. GIT FOR SQUEAK does not support lazy loading of objects.

With our implementation of plumbing we achieve:

- that the GITFS plumbing layer is in itself a full implementation of the GIT functionality we expect from GITFS;
- that we are able to inspect the plumbing layer using the standard tools in Pharo because we implemented support for the Smalltalk object model;
- a performance enhancement over having every object in memory, by loading GIT objects lazily.

2.4.2 Implementation of Porcelain

Our porcelain implementation is special in that it acts as an adaptor between the FILESYSTEM protocol and the GIT protocol. By subclassing `FSMemoryFilesystem` with a class named `FSGitFilesystem` we get the functionality and protocol of memory filesystems for free. This section describes how we implemented the communication with the plumbing layer and which changes to the behaviour of `FSMemoryFilesystem` were necessary.

`FSGitFilesystem` delegates messages to a cache. The cache is an instance of `FSMemoryFilesystem` and it is the filesystem the user actually interacts with. The cache contains accessed and newly created paths. When a user wants to access a path that is not yet in the cache then that path is loaded into the cache by `FSGitFilesystem` before the request is handed to the cache. Using an `FSMemoryFilesystem` object as cache and letting the user operate on that cache saves us from having to implement our own cache data structure. It also simplifies the delegation of messages. There are only three scenarios where `FSGitFilesystem` adds behaviour to `FSMemoryFilesystem`:

- File modifications need to be registered (modifications include the creation of files). When committing, we need to store the modified paths as GIT objects in the repository. Iterating over all entries of the cache might yield many paths that have been read but not modified. To improve performance we therefore register if a path has been modified.

- File deletions need to be registered. The cache has no knowledge of deleted files but we need to remove the deleted entries from the GIT tree objects before committing.
- Objects need to be lazily loaded. For large repositories it would take a long time and use much memory to cache the entire working copy. By lazily loading objects from the repository we can improve performance and decrease memory load. Lazy loading requires hooks in certain methods to load objects in the background before the user interacts with the cache. This is what we accomplish with the cache in `FSGitFilesystem`.

Modifications and deletions are registered in two separate dictionaries where paths are mapped to the GIT tree object they belong to. An additional dictionary ensures quick lookup of accessed trees.

When the message `#commit:` is sent to an `FSGitFilesystem` object we first prepare the data in porcelain and then let plumbing handle the interaction with the repository. The data preparation involves the following steps:

1. Create blob objects from modified files. The blobs will not be modified again before the commit so we pass them to the plumbing layer to store them.
2. For every stored blob update the container tree object: the tree objects are sorted to allow us to first process the tree objects farthest away from the root tree. This is necessary to ensure correct signatures (remember that the signature of an object changes if the contents of that object are modified). The respective entries in the tree are then created or replaced.
3. For every removed path update the concerned tree object.
4. Pass all tree objects to the plumbing layer to store them.
5. Build the commit object referencing the updated root tree object. Pass the commit to the plumbing layer.

With our implementation of porcelain we achieve:

- that the user interfaces with a file system;
- a better user experience due to a simplified interface to GIT (compared to plumbing);
- that implementation details of GIT are not visible to the user;
- a performance enhancement over having every object in memory, by loading GIT objects lazily.

3 Modeling The History With a Git Repository

PHAROGENESIS is our solution to the problem of discontinuous history in Squeak and Pharo. It uses GITFS to create a GIT repository that models the complete history of the source code in Squeak and Pharo. A prebuilt repository using the setup described in this thesis is available on github (<http://www.github.com/pharogenesis/pharogenesis.git>).

The data that is passed to GITFS is prepared in two steps:

1. We extract the source code from the ***.sources** and ***.changes** files.
2. We build a directory structure that can be versioned with GIT.

In the following we will elaborate on both steps of the data preparation.

3.1 Reading *.changes and *.sources files

Source code changes in the image are represented by the class `ChangeSet`. We obtain a collection of the changes that are stored in a file by asking the `ChangeSet` class to scan that file:

```
| changes filename |
filename := 'SqueakV10.sources'
changes :=
  FileStream
    oldFileNamed: filename
    do: [ :sourceStream |
      ChangeSet
        scanFile: sourceStream
        from: 0
        to: sourceStream size ].
```

The answered collection contains `ChangeRecord` objects. A `ChangeRecord` is the representation in the image of a single change. A change is the difference between two versions of source code. The content of a change is the new source code (the code that was accepted by the user in the image). The following information is stored in a `ChangeRecord` object:

- the type of the change. There are four different `ChangeRecord` types:
 - “preamble”: The preamble is a section, specific to a change, in a changes file that contains arbitrary Smalltalk code. It can be used to store additional information to a change or perform initialization tasks. Generally the contents of the preamble are not interesting and hence we ignore them.
 - “classComment”
 - “method”

- “doIt”: a “doIt” is an execution directive. Class creations for instance are encoded as “doIts”.
- the class the change occurred in.
- truth values that tell if the change occurred on instance or class side of the concerned class.
- the method category (only if the change is of the type “method”).
- the method selector of the changed method (only if the change is of the type “method”).
- a stamp describing time, date and author of the change (not always present, sometimes differently formatted or partly missing).
- the actual change in string representation (the representation depends on the change type).

3.2 Extracting The Source Code

The information we want to extract is stored in several files. Fortunately, every ***.changes** file includes the contents of all previous ***.changes** files with respect to the latest ***.sources** file. This means that we only need to include the ***.sources** file and the latest ***.changes** file of every release to ensure that we have all the changes. PHAROGENESIS therefore expects the input files in the following order (see also figure 6):

1. ***.sources** file of the first release.
2. most recent ***.changes** file to the first release.
3. repeat 1. and 2. for releases 2 through $n - 1$.
4. ***.sources** file of release n .

When processing each file we iterate over every **ChangeRecord** object that we received from the **ChangeSet**. Depending on the type of the change (not the **ChangeRecord** type) we need to handle the data differently. We created a class hierarchy to mimic the change types which lets the object decide how to treat the data instead of having to manually check the type. Each **ChangeRecord** object is wrapped into an appropriate object of the mimicking hierarchy. There are five different relevant change types (not **ChangeRecord** types):

- meta class creation (**ChangeRecord** type “doIt”)
- class creation (**ChangeRecord** type “doIt”)
- class comment (**ChangeRecord** type “classComment”)
- method creation / modification (**ChangeRecord** type “method”)

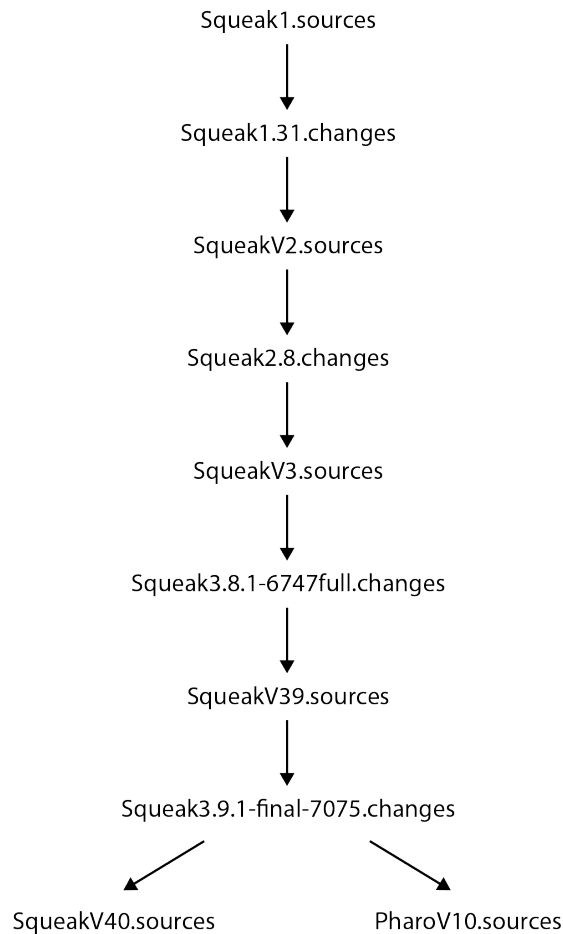


Figure 6: The GIT repository is built with the information gathered from a series of ***.changes** and ***.sources** files. This figure schematically illustrates which files we used and how the repository is structured.

- method removal (**ChangeRecord** type “doIt”)

ChangeRecord objects of type “doIt” need special treatment because they can contain arbitrary Smalltalk code. Additionally, for the “doIt” type the **ChangeRecord** object does not know all relevant properties such as the name of the class the change has occurred in. We retrieve this information by parsing the “doIt” and extracting the required data using AST pattern matching. Then we store the new knowledge in the wrapper object. This makes the information available wherever it is needed.

The extracted source code is written to an **FSGitFilesystem** object. At this point **GITFS** takes over and performs the commit of the altered working copy.

3.3 Representing The Class Hierarchy as a Directory Tree

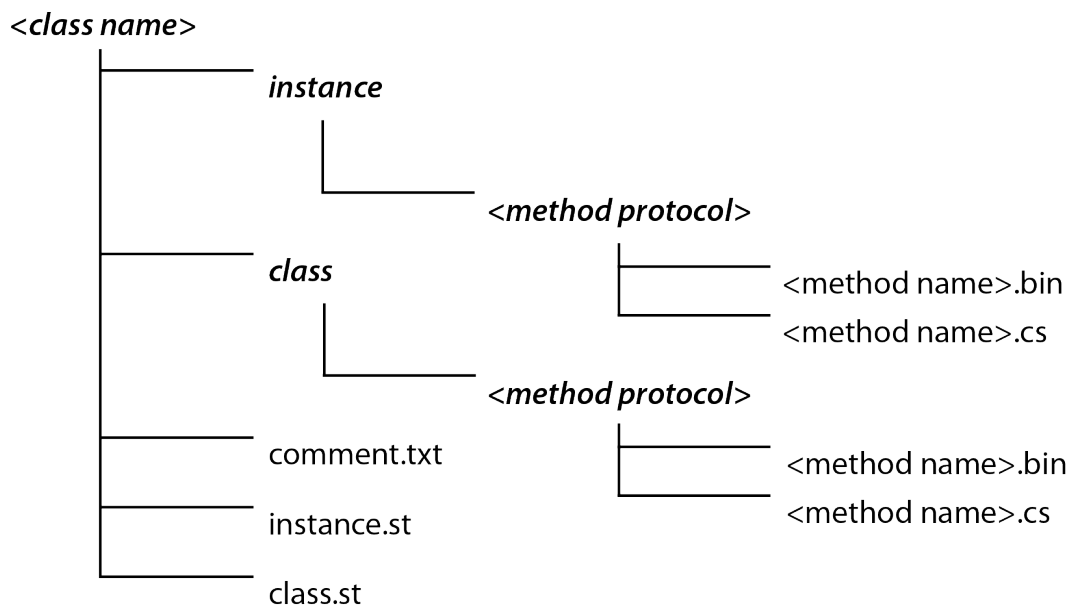


Figure 7: Information is grouped by 1. class name; 2. instance or class side; and 3. method protocol. Directories are bold and italic, the chevrons denote variable parts.

We model the class hierarchy as a tree of directories and files (see figure 7). To GIT the directories are tree objects and the files are blobs that contain plain text. The base directory contains a directory for each class name. Each class directory contains two directories named “instance” and “class” to separate instance side from class side methods. Besides those two directories the class directory also contains three files: “comment.txt” (“.txt” denotes textual content) stores the class comment if there is one, “instance.st” (“st” is the acronym for “Smalltalk”) contains the class definition and “class.st” the metaclass definition.

The directories separating the class side from the instance side methods further group methods by method protocol. The method source finally is stored in a file named after the selector and ends with “.cs” (“cs” is the acronym for “ChangeSet”).

For every method there is a second file called “<method name>.bin” which stores the following data:

- the name of the author
- the time of creation
- the name of the commit that occurred *before* the one that introduced the current method version

The first three entries allow us to access information about a method version without having to know the commit that introduced said version (remember that commit information in GIT is only held by the commit object). This means that we can look at an arbitrary commit and view method metadata without having to lookup the commit object that holds that metadata. The last entry is an optimization for looking up method versions: Instead of traversing every commit in the GIT repository and comparing signatures, we can jump from commit to commit doing only as many lookups as there are different versions of the method. Considering that the PHAROGENESIS repository holds approximately 60000 commits the performance enhancement for a method with six versions is 1:10000.

3.3.1 Encoding File Names

Method names can contain special characters such as “:”, “\” and “/”. These characters are used for the encoding of paths in filesystems and therefore lead to unexpected results when checked out of a GIT repository. We decided to encode the selectors by replacing certain special characters with their hexadecimal representation. Hashing the selectors is another possible solution but hashed selectors are difficult for humans to read when manually browsing the repository. The following replacements are performed:

- “-” with “-2D-”
- “:” with “-3A-”
- “/” with “-2F-”
- “\” with “-5C-”

We use the hyphen character to mark a character replacement which means that we need to encode the hyphen character as well.

3.4 How To Use Pharogenesis

PHAROGENESIS is most easily obtained using Gofer:

```
Gofer new
  url: 'http://www.squeaksource.com/GitFS';
  package: 'ConfigurationOfPharogenesis';
  load.
((Smalltalk at: #ConfigurationOfPharogenesis) project version: #stable) load.
```

For this example we put all the ***.sources** and ***.changes** files we want to import into a single directory named “files” in the working directory. We then need to decide where to put the repository. PHAROGENESIS will create a new folder called “pharogenesis” for the repository so we will use the working directory as the target for the example (the repository will then be generated in “<working directory>/pharogenesis”):

```
source := FSDiskFilesystem current working / 'files'.
target := FSDiskFilesystem current working.
```

Now we initialize a new `Pharogenesis` object. We then tell it which directory to write the repository to and where to find the files:

```
Pharogenesis
  createRepositoryAt: target
  fromFilesIn: source.
```

The names of the files that need to be in the “files” directory are listed in the method `#namesOfSources` of class `Pharogenesis`:

```
^ #(
  'SqueakV1.sources'
  'Squeak1.31.changes'
  'SqueakV2.sources'
  'Squeak2.8.changes'
  'SqueakV3.sources'
  'Squeak3.8.1-6747full.changes'
  'SqueakV39.sources'
  'Squeak3.9.1-final-7075.changes'
  'SqueakV40.sources'
  'PharoV10.sources'
).
```

PHAROGENESIS includes a hard coded hook that checks for the Pharo sources and automatically creates the branch at the correct position. For each `*.sources` and `*.changes` file a tag is written. The tagged commits can be accessed through an `FSGitRepository` like this:

```
repository := FSGitRepository on:
  (FSDiskFilesystem current working / 'pharogenesis').
squeak28 := repository tagNamed: 'Squeak2.8.changes'.
```

To list all the tags in the repository we can send `#tagNames`:

```
squeakAndPharoVersions := repository tagNames.
```

The repository contains two branches named “squeak” and “pharo”. The “squeak” branch contains all the commits up to and including the tag “SqueakV40.sources”. The “pharo” branch contains only one commit whose parent is the commit tagged “Squeak3.9-final-7075.changes”. We can send `#switchToBranch:` to switch between branches:

```
repository switchToBranch: 'pharo'.
```

The repository available on github (<http://www.github.com/pharogenesis/pharogenesis.git>) has been packed and garbage collected. GITFS does not yet support these operations which is why we used the command line:

```
$ cd pharogenesis
$ git gc --aggressive
```

Building the repository takes up to 10 hours depending on the hardware and creates over 500'000 files. Running the GIT garbage collector will take a comparable amount of time because each file is read and the contents rewritten to the pack file before the object file is deleted. The effort is worthwhile though because the repository will be significantly reduced in size and access times will be shorter.

Although building the repository takes a long time, it has to be built only once and can then be used very efficiently.

number of object files after creation:	517758
number of object files after garbage collection:	0
size of *.changes + *.sources files parsed:	105.7 MB
size of repository after creation:	4.2 GB
size of repository after garbage collection:	64.3 MB
size of pack file:	49.8 MB
size of pack index file:	14.5 MB

Table 1: Observations on the effect of GIT garbage collection on the number of files and the size of the repository.

With PHAROGENESIS we provide a tool that can create a GIT repository from the contents of ***.changes** and ***.sources** files. PHAROGENESIS also demonstrates how GITFS can be used to interact with GIT.

3.5 A Method Version Browser For Pharogenesis

To show how PHAROGENESIS can be used we added a button to OMNIBROWSER [Berg08c] that allows us to browse the PHAROGENESIS repository for method versions. OMNIBROWSER is the standard code browser currently used in Pharo.

The easiest way to look at the example is to download the pre-built one-click application from <http://scg.unibe.ch/jenkins/job/Pharogenesis/lastSuccessfulBuild/artifact/Pharogenesis.zip>.

To launch the one-click image do the following depending on your platform:

- Apple Mac OS: double click on “Pharogenesis.app”
- Linux: double click on “Pharogenesis.app/Pharogenesis.sh”. If the shell script should not be executable you can issue the following commands on the command line to launch PHAROGENESIS:

```
$ cd Pharogenesis.app
$ chmod gu+x Pharogenesis.sh
$ ./Pharogenesis.sh
```

- Microsoft Windows: double click on “Pharogenesis.app/Pharogenesis.exe”

To load the example into an existing image you can use Gofer:

```
Gofer new
  squeaksource: 'GitFS';
  package: 'PharogenesisOB';
  load.
```



Figure 8: A click on the button labeled “Versions (Git)” shows all the versions of the method #add of class Bag that were found in the GIT repository.

When the extension is loaded a click on the button “Versions (Git)” in OMNIBROWSER will browse a GIT repository for method versions instead of the *.changes file (see figure 8). The default location of this repository is a folder called “pharogenesis”. If you use the one-click image the “pharogenesis” directory should reside in the same directory as the one-click application. For every other image the default location is the image working directory. To use the prepared repository from “github” we call GIT from the command line:

```
$ cd <path to image or one-click application>
$ git clone http://github.com/pharogenesis/pharogenesis.git
$ cd pharogenesis
$ git checkout -b origin/pharo
```

The above commands will create a directory named “pharogenesis” and then clone the repository from “github” into “pharogenesis”. GIT only clones the default branch from a repository. The final command is therefore necessary to download the branch “pharo”. Because we used “git checkout” we also switched to “pharo” as our working branch (“HEAD” points to the latest commit on the branch “pharo”).

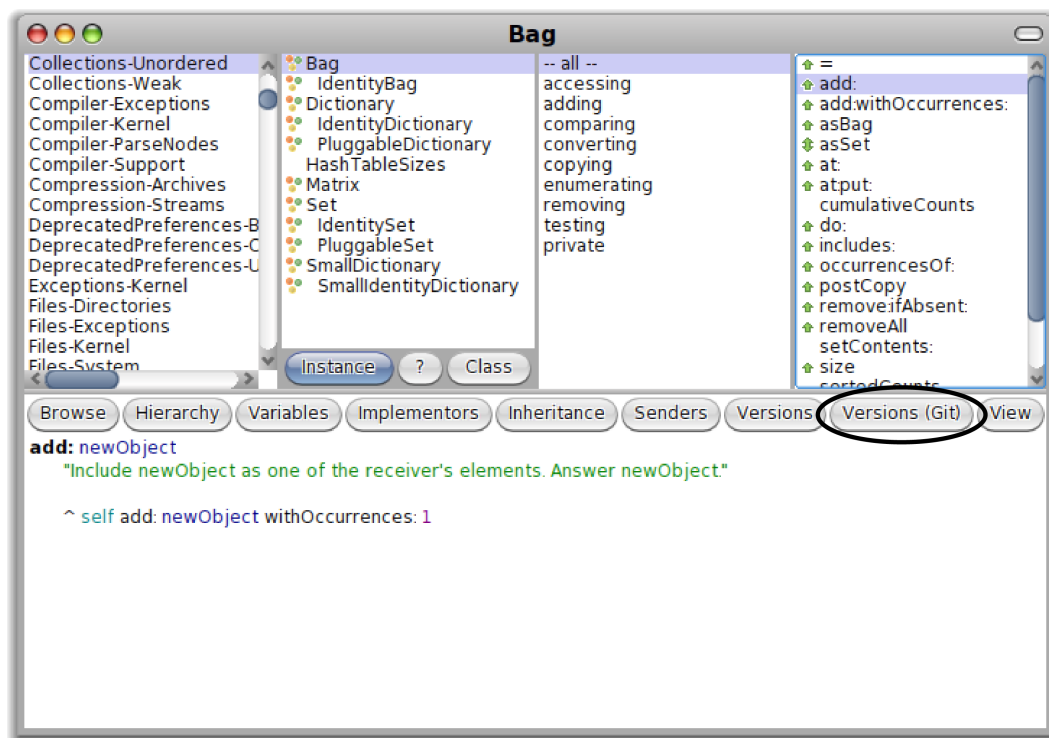


Figure 9: To browse method versions in OMNIBROWSER, select a method name from the list and click the button labeled “Versions (Git)”.

Now that the repository has been cloned into the working directory a click on the button “Versions (Git)” in OMNIBROWSER will search the GIT repository in the “pharogenesis” directory (see figure 9). The following code is an excerpt showing only the most important steps involved in finding the method versions in the repository:

```

1  searchFor: aPath
2      | commit nodes methodReference done |
3      nodes := OrderedCollection new.
4      commit := self defaultRepository head.
5      methodReference := commit referenceTo: aPath.
6
7      done := false.
8      methodReference exists ifFalse: [ done := true ].
9      [ done ] whileFalse: [
10         nodes add:
11             (OBGenesisMethodVersionNode

```



```

12         on: methodReference
13         inClass: theClass).
14
15     (self hasParentVersion: methodReference)
16     ifTrue: [
17         commit := self nextCommit: methodReference.
18         methodReference := commit referenceTo: aPath ]
19     ifFalse: [ done := true ] ].
20
21     nodes isEmpty ifTrue: [ nodes add: OBGenesisMethodVersionNode new ].
22     self sortVersionsByDateDescending: nodes.
23     ^ nodes

```

The message argument “aPath” is the path (an `FSPath` object) to the location of the explored method in the directory tree. The path is identical for every commit. Every version of the method we are exploring (the one that was selected when the click on “Versions” occurred) is added to the collection “nodes” (lines 3, 10). The variable “commit” is initialized with the latest commit on the active branch (branch “pharo”; line 4) and tracks the commit that is being analyzed.

The code inside of the while loop follows the commits mentioned in the method data files and adds a method version for each commit to “nodes” (lines 9 through 19). The method `#nextCommit:` (line 17) answers the commit that contains the parent version of the current method version as found in the method data file.

Before answering the collection of method versions, we sort the method versions by time stamp (remember that the commits are not in chronological order).

4 Conclusion

Our goal was to provide better access to the history of Squeak and Pharo code. To this end we implemented two tools called GITFS and PHAROGENESIS. GITFS provides an easy way to create and manipulate GIT repositories. By extending FILESYSTEM, GITFS presents users with a view they are used to. Using GITFS, custom tools can be built to analyze and visualize the Pharo and Squeak code history without having to deal with the shortcomings of ***.changes** and ***.sources** files. We proved this by implementing PHAROGENESIS on top of GITFS.

PHAROGENESIS uses GITFS to build a GIT repository of the changes found in ***.changes** and ***.sources** files. With PHAROGENESIS we provide a single source for access to the Squeak and Pharo code history. We also showed how the PHAROGENESIS repository can be used to find information on code history.

5 Future Work

GITFS is well suited for the task of building the PHAROGENESIS repository. We do not feel that it is necessary or even appropriate to duplicate GIT completely with GITFS. Nonetheless, there are areas which should be improved. The implementation of the following items will greatly increase the value of GITFS:

- merge operation for commits. Branches are a convenient way for people to work on different parts of an application. These parts need to be merged at some point which makes the merge operation an important feature.
- support for “reflogs” (log file that records all the changes and can be used to recover lost changes).
- support for “.gitignore” (a file with regular expression entries to exclude files and directories from commits).

We designed GITFS with the idea that developers should use it for their own projects. There are especially two projects we would like to see: One is a solution that enables users to hold on to their local changes across fresh images (remember that a ***.changes** file is specific to an image). A side effect of this solution would be additional safety against loss of code. The other solution we imagine is the integration with MONTICELLO. GITFS could be used to provide better version control of packages or even to replace the current packaging mechanism with a widely accepted version control technology.

The PHAROGENESIS repository in its current form does not guarantee that the commits are ordered by time stamp. The reason for this is that the time stamps that are associated with each change are unreliable: some stamps are missing completely and others are incomplete or obviously erroneous. A future repository should reflect the creation time of each change to allow for better analysis.

We built the PHAROGENESIS repository for all the changes up to and including Squeak 4 and Pharo 1. For the future we would like to see the repository also spanning the gap to the newest releases. An automated process could use PHAROGENESIS or a similar tool to write new changes to the repository.

Like MONTICELLO, our solution does not track moved code (e.g. methods moved between two classes) and renames. GIT has a built-in command name “mv” that enables tracking of moves and renames. If this command were implemented in GITFS, PHAROGENESIS could be adapted to find renames and moves in the ***.changes** and ***.sources** files so that such information could be written to the repository.

References

- [**Berg08c**] Alexandre Bergel et al. “Creating Sophisticated Development Tools with OmniBrowser”. In: *Journal of Computer Languages, Systems and Structures* 34.2-3 (2008), pp. 109–129.
- [**Blac09a**] Andrew Black et al. *Pharo by Example*. Square Bracket Associates, 2009.
- [**Blac07a**] Andrew Black et al. *Squeak by Example*. <http://SqueakByExample.org/>. Square Bracket Associates, 2007.
- [**Monticello**] Avi Bryant and Colin Putney. *Monticello*. <http://www.wiresong.ca/Monticello>.
- [**Garn10a**] Tony Garnock-Jones. *Git for Squeak*. <http://www.squeaksource.com/Git>.
- [**Putn10a**] Colin Putney. *Filesystem*. <http://www.wiresong.ca/filesystem>.