Towards Version Control in Object-based Systems

Jakob Reschke, Marcel Taeumel, Tobias Pape,
Fabio Niephaus, Robert Hirschfeld

Technische Berichte Nr. 121

des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam | 121

Jakob Reschke | Marcel Taeumel | Tobias Pape |
Fabio Niephaus | Robert Hirschfeld

Towards Version Control in Object-based Systems

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2018
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Das Manuskript ist urheberrechtlich geschützt.
Druck: docupoint GmbH Magdeburg

ISBN 978-3-86956-430-2

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URN urn:nbn:de:kobv:517-opus4-410812
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-410812

Version control is a widely used practice among software developers. It reduces the risk of changing their software and allows them to manage different configurations and to collaborate with others more efficiently. This is amplified by code sharing platforms such as GitHub or Bitbucket. Most version control systems track files (e.g., Git, Mercurial, and Subversion do), but some programming environments do not operate on files, but on objects instead (many Smalltalk implementations do). Users of such environments want to use version control for their objects anyway. Specialized version control systems, such as the ones available for Smalltalk systems (e.g., ENVY/Developer and Monticello), focus on a small subset of objects that can be versioned. Most of these systems concentrate on the tracking of methods, classes, and configurations of these. Other user-defined and user-built objects are either not eligible for version control at all, tracking them involves complicated workarounds, or a fixed, domain-unspecific serialization format is used that does not equally suit all kinds of objects. Moreover, these version control systems that are specific to a programming environment require their own code sharing platforms; popular, well-established platforms for file-based version control systems cannot be used or adapter solutions need to be implemented and maintained.

To improve the situation for version control of arbitrary objects, a framework for tracking, converting, and storing of objects is presented in this report. It allows editions of objects to be stored in an exchangeable, existing backend version control system. The platforms of the backend version control system can thus be reused. Users and objects have control over how objects are captured for the purpose of version control. Domain-specific requirements can be implemented. The storage format (i.e., the file format, when file-based backend version control systems are used) can also vary from one object to another. Different editions of objects can be compared and sets of changes can be applied to graphs of objects. A generic way for capturing and restoring that supports most kinds of objects is described. It models each object as a collection of slots. Thus, users can begin to track their objects without first having to implement version control supplements for their own kinds of objects. The proposed architecture is evaluated using a prototype implementation that can be used to track objects in Squeak/Smalltalk with Git. The prototype improves the suboptimal standing of user objects with respect to version control described above and also simplifies some version control tasks for classes and methods as well. It also raises new problems, which are discussed in this report as well.

Contents

1 Introduction **11**

2 Background **14**
- 2.1 Version control and object import/export in existing programming environments . . . 14
- 2.2 Goals for a new version control system for objects . . . 18
- 2.3 Prerequisites and prior knowledge . . . 19

3 An architecture for object versioning **27**
- 3.1 Storing objects in versions . . . 27
- 3.2 Live objects, snapshot objects, storage objects . . . 28
- 3.3 Preserving object identity across system boundaries . . . 29
- 3.4 Capturing and materialization of object graphs . . . 30
- 3.5 Differences between snapshots . . . 33
- 3.6 Storing objects outside of the programming environment . . . 39
- 3.7 Generic snapshot format for objects . . . 40

4 Object version control in Squeak/Smalltalk with Git **43**
- 4.1 Squot: Squeak's Object Tracker . . . 43
- 4.2 Git connectivity . . . 61
- 4.3 Graphical User Interface . . . 65

5 Evaluation and discussion **73**

6 Related work **84**
- 6.1 Existing version control approaches . . . 84
- 6.2 Tracking and transportation of objects . . . 90
- 6.3 User interface concepts . . . 92
- 6.4 Object merging . . . 92

7 Conclusion and outlook **94**

List of Figures

2.1 Monticello's model of snapshots and definitions 15
2.2 Git object types 20
2.3 Vocabulary issues for objects and snapshots of them 22

3.1 External referrer into a captured object graph 29
3.2 Example setting for object graphs, start objects, and object names . . 31
3.3 Different graph traversal strategy for a subgraph 32
3.4 Referrers and referents in object comparisons 36
3.5 Differences hidden behind added objects 37

4.1 Object containers, snapshots, artifacts, graphs, and image store in Squot 46
4.2 Relationships among object containers, artifacts, object graphs, and their respective difference classes 53
4.3 The Git browser 68
4.4 Context menu of the Git browser's object list 69
4.5 Change selection in the dialog to create new versions 70
4.6 Conversion between differences and trees for the graphical user interface (GUI) 72

5.1 Difficulties during object merging 76

Listings

4.1 Adding a package to an image store 43
4.2 Default implementation Object»captureWithSquot: 49
4.3 Materialization dispatch in SquotShadowMaterializer 51
4.4 The allocation of new objects by SquotObjectShadows is separated from filling in the slot values 51
4.5 The allocation of new difference objects for SquotDiffBuilder is separated from actually computing the differences. 54
4.6 Examples of the working copy application programming interface (API) 66

List of Abbreviations

API	application programming interface
CLOS	Common Lisp Object System
GUI	graphical user interface
IDE	integrated development environment
JSON	JavaScript Object Notation
STON	Smalltalk Object Notation
VCS	version control system

1 Introduction

For a long time, it has been good practice to keep past versions of source code to manage changes and track who performed them [55]. In most programming systems the source code of programs is saved in files. Hence, version control systems or their user interfaces are often based on files and interactions with files. That is, users add files and directories to the version control system and the subsequent changes to files and the directory structure constitute the differences between versions.

But when users engage in exploratory programming, the product of the users' work may not only be code. It can also be objects that have been interactively created, configured, and composed over time if the programming environment allows it. The amount of work and time spent to build objects might be as significant as the resources invested in writing code. A prime example where objects and code are equally important is the prototype-based language and programming environment *Self* [69].

Many Smalltalk systems can also be used for exploratory programming, using their extended set of interactive tools, such as object inspectors, a powerful debugger, and the ability to run snippets of code just about anywhere in the environment [21]. These facilities can also be used to construct objects without writing construction code in advance, like in Self. Yet, such arbitrary objects are often only second-level citizens in contemporary version control systems for Smalltalk, or they are not eligible for version control at all. Version control systems that support arbitrary content are usually file-based, whereas non-file-based version control systems tend to be specialized to some domain: most version control systems for Smalltalk, like *ENVY/Developer* or *Monticello*, focus on the tracking of class definitions, methods, and objects that store configurations (packages, applications), but not arbitrary objects.

A way to put such objects under version control anyway is to export them to a file or to encode them in a different kind of object that can be tracked by a version control system. Users of document-centric applications such as office software suites may look upon this as a natural workflow. But to save a new version of a set of objects in this way, users would have to collect all of these objects, export all of them from their native, directly manipulable form to a different representation, and only then actually create a new version with the version control system. This repetitive task should be facilitated by tools.

When objects must be converted before they can be put under version control, sometimes there are multiple options of how this can be done. For example, for any given purpose, multiple file formats can be available (e.g., XML or JavaScript Object

Notation (JSON)[1]), or multiple perspectives from which an object could be modeled (e.g., a particular plot of the graph of a function can be represented extensionally as an enumeration of its data points or as an image, or intensionally by a description of the plotted function and attributes such as the scaling and range of the coordinate axes).

Version control is also used to enable collaboration. When people make changes to local copies (or instances) of a system simultaneously, contemporary version control tools support users in managing and synchronizing different streams of development (branches). Internet platforms such as *GitHub*[2]have made collaborative, distributed development of software or other (file-based) artifacts easier, especially for free and open-source software. Often these platforms provide additional tools for project management, such as issue trackers or wikis.

Why should there not be similar support for collaboration on artifacts that are not text, or files, but arbitrary objects built in an exploratory programming environment? For example, when somebody creates an active essay [33] in an environment like Squeak/Smalltalk[3] [30], he or she might want to share it with others and accept contributions. If there is a team of co-authors, they will surely want to synchronize their work. Ideally, this should not be harder or more cumbersome than the sharing and synchronizing of source code among programmers.

The adoption of a new version control solution is not a lightly-taken decision. It must be trusted that the new system works correctly, so no data will be lost. We therefore propose a version control solution for arbitrary objects that works on top of an existing, already established version control system. This also makes the reuse of existing platforms possible. Otherwise, similar platforms might need to be developed and maintained for the new system. Every new platform would incur operational costs, which can be saved if an existing version control system and its platforms can be reused.

Contributions

In this report, we present an architecture to put arbitrary objects under version control. It allows for customization of how versions of objects are represented and how they can be stored in files (if that is necessary), so the most suitable representation can be chosen. The storage of versions is delegated to a backend version control system, so existing systems can be used for their maturity and familiarity to users, and their platforms can be reused for their utility and to save costs.

We do *not* propose a solution for synchronous collaborative editing of objects. That is, the proposed architecture assumes that the users of the exploratory programming environment need to synchronize their changes to objects explicitly. Changes by others do not become visible on the screen automatically, like it is the case in online

[1]http://www.json.org/ (last accessed November 5, 2017).
[2]http://github.com (last accessed November 5, 2017).
[3]http://squeak.org (last accessed November 5, 2017).

services like Etherpad[4] or Google Docs[5]. We rather assume that users work independently on different kinds of objects like Fabrik models [31], sets of UML diagrams, pictures and other static resources, classes and source code, and that they want to synchronize all these objects in combination at a time of their choosing.

Chapter 2 gives an overview of how version control has been approached so far in selected exploratory programming environments, and introduces the reader to background knowledge that might be needed to put some information from the following chapters into context properly. Chapter 3 describes the proposed architecture. Following that, chapter 4 introduces the prototype implementation of this architecture, which provides object version control in Squeak/Smalltalk with Git. In chapter 5, the prototype is evaluated. Observations and issues are discussed. Chapter 6 relates the prototype to a number of other version control approaches and solutions, ways to deal with objects that must be transferred between two programming environments, user interface concepts, and some work that can be used as a starting point to resolve issues of the prototype or to amend the features that are still missing from it. Finally, chapter 7 concludes this report and gives an outlook of how its results could find further applications.

[4] http://etherpad.org/ (last accessed November 5, 2017).
[5] https://www.google.com/docs/about/ (last accessed November 5, 2017).

2 Background

In this chapter, an overview on existing approaches to version control in exploratory programming environments is given, and coming from that, the goal of this report is refined. Further, an overview of existing concepts that are reused in the following chapters is given, such as version-control-related vocabulary, a brief introduction to the object model of Git repositories, and challenges that need to be solved.

When the term *users* is used in this report without further qualification, the users of an exploratory programming environment are meant. They could be programmers, but they might also use the environment to create documents, such as active essays, or other kinds of objects, without general programming expertise. *Users* are the ones that decide that some objects should be put under version control. More terms are introduced later in this chapter.

2.1 Version control and object import/export in existing programming environments

In most programming languages, all artifacts—both source code and resources that in combination define the delivered software—are stored in files. Thus, they are naturally compatible with file-based version control systems (VCSs) like Git. However, this is not the case for all programming environments. Self and most Smalltalk systems, being examples of exploratory programming environments, store their users' software product in a snapshot of the heap memory, together with the runtime environment and development tools. This snapshot is commonly called an (or the) *image*.

Such systems revolving around an image usually allow exporting the source code and other definitions of a software to the file system. In Smalltalks, this export operation is usually called *file out* [21]. The exported definitions can be imported into other images with a corresponding *file in* operation. The files created during a file-out can be versioned with any file-based VCS. But if that way is taken, the tools for version control are usually not available in the programming environment. Conversely, systems that do not provide own version control tools usually rely solely on file out and external VCSs, like *Cuis/Smalltalk* does [72, 73].

Many other Smalltalk systems do come with own version control systems. Squeak[1]and Pharo[2]use a VCS called *Monticello*. It describes Smalltalk packages declaratively in

[1]http://squeak.org/ (last accessed November 5, 2017).
[2]http://pharo.org/ (last accessed November 5, 2017).

terms of *definitions* of the contained classes, methods, auxiliary scripts and the class organization (categorization). All of these objects are essentially source code, or closely related to it. While Monticello's snapshot model could support non-code artifacts (see Figure 2.1), the Monticello tools are currently not built to support such artifacts well.

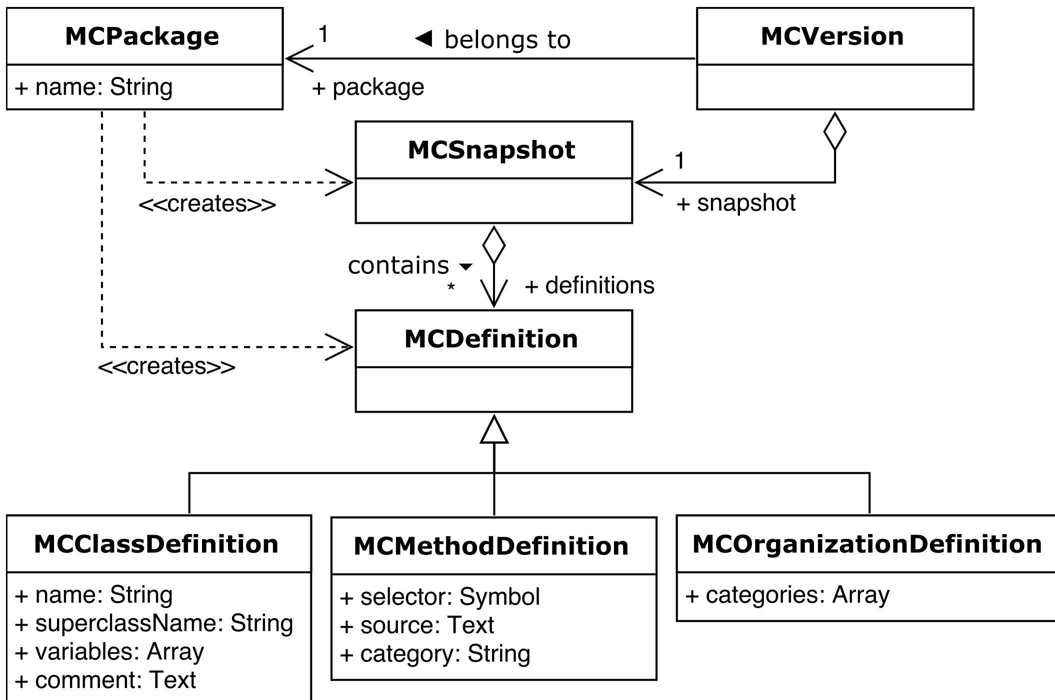

Figure 2.1: Monticello's model of snapshots and definitions (extract). New subclasses of MCDefinition could be added, but it might turn out that not all kinds of Monticello repositories and tools support them.

Another popular Smalltalk VCS is *ENVY/Developer*, often just called ENVY [51]. It is still available in *VA Smalltalk* and was previously also available in *Cincom VisualWorks Smalltalk*. It provides versioning for methods, classes, class extensions (which contain extension methods), applications (which group related classes and can have dependencies on other applications, similar to packages in Monticello), and configuration maps (that combine specific versions of applications into configurations). ENVY features a tight integration with the development tools. For example, most menus related to the versioned objects allow users to browse other editions of the selected object. But again, the entities listed above are all related to source code, not arbitrary objects. Files and directories can be added to applications [27, ch. 29], which stores them in the repository of versioned objects (called the *library* in ENVY). However, there seems to be no immediate way to track instances of domain objects with ENVY. *Dolphin Smalltalk* includes a version control facility called *Source Tracking*

System (*STS*) which is similar in use and functionality to ENVY [59]. It offers the possibility to add global objects (which can be accessed unqualified from methods and are usually stored in the Smalltalk dictionary) to a package, so these objects will be versioned together with the package. This will store the object in Dolphin's own binary object format, encoded as text, in the package file generated when the package is saved. Therefore, the serialization format is fixed by the version control framework and, as a human being, it is hard to make sense of the object just by looking at the package file (on GitHub, for example).

The focus of Smalltalk version control systems on source code is reflected in GUI building: most frameworks mandate that the GUI must be defined in code (Squeak's *ToolBuilder*, Dolphin's *MVP*), or design tools serialize the composed interfaces to source code (VA Smalltalk's visual parts, Squeak's Morphic designer[3]). The latter approach is also taken in other programming languages and frameworks, such as the form designer for Windows Forms in Microsoft's Visual Studio. Generating source code for the GUI does have advantages, such as being readable (if the API for GUIs and the source code generator are good). But on the other hand, it adds a transformation step between what is being built in a design tool and what is actually created from it. If the generated source code is subsequently modified manually (or by a merge on GitHub), this can lead to errors in the design tool, or the manual changes might be discarded when the design tool is used again.

There are facilities to export arbitrary objects from Smalltalk images, independent of version control. For example, there are general purpose object serializers and deserializers, such as Squeak's DataStream and its more advanced successor SmartRefStream [10], or *Fuel* as a fast binary object serializer [13, 14]. However, these are not integrated with version control facilities like Monticello, so they provide only a file-out and file-in equivalent for arbitrary objects.

Squeak's primary user interface, which is called *Morphic* and was ported from Self [25, 45], supports creating graphical objects interactively, without writing construction code. The graphical composites, called morphs, can be exported to files via a context menu, much like the file out operation for source code. But it is not integrated with a version control system. The serialization behind the scenes actually uses the previously mentioned SmartRefStream. Because Monticello does not track arbitrary objects or files, this poses a problem when an interactively-built morph should be versioned together with the source code that uses it. One possible workaround is to use two different systems: Monticello for the code, and a file-based VCS for resources in files, but it complicates configuration management and the installation of packages. Another workaround is to serialize resources into source code (e.g., generating a method that returns a byte array of the resource data), which can then be tracked with Monticello. But saving a new version in Monticello will not trigger the serialization of all resources into their designated methods, so there is still a disconnection between the VCS and the tracked objects.

[3] https://www.hpi.uni-potsdam.de/hirschfeld/trac/SqueakCommunityProjects/wiki/designer (last accessed November 5, 2017).

Moreover, any binary serialization output (whether encoded in text or not) is not as easily accessible on GitHub as text-based formats—primarily due to the fact that most humans cannot read the binary format. But automatic merging of such files is also not available [18, gitattributes, built-in merge drivers], so collaboration through pull-requests on GitHub can become more difficult.

In Self, objects are transferred from one image to another via the *Self Transporter* [68]. There are no classes in Self. Objects implementations are reused as prototypes, like in JavaScript. The prototype objects are fully-capable objects themselves, which blurs or even removes the gap between meta-objects and "actual instances". For this reason, the Self Transporter can transport any object. It does so by traversing a graph of objects—guided by annotations to objects and their slots to fill in important information about user intentions that would otherwise not be available in these objects—and writing files that contain Self source code expressions that rebuild the captured object graph when evaluated. The serialization format is therefore text-based and general enough to describe arbitrary objects, but it is specific to the Self language. Objects can be exchanged with file-based VCSs, like Smalltalk file outs, but the version control system is not integrated (or even visible) in the Self environment.

Another exploratory programming environment is the Lively Kernel [32]. Coming with a set of programming tools that should be familiar to Squeak/Smalltalk users, and another implementation of the Morphic framework, the Lively Kernel provides its users with an exploratory programming environment for JavaScript entirely inside the web browser. Lively components can be shared via a *parts bin* that uses Subversion for version control and publishing. On top of that, version control features like difference detection and merging have been integrated into the Lively Kernel environment [5][39, ch. 5.2, 5.3]. It employs a serialization based on JSON with support for object graphs (with cycles) and instance-specific behavior (i.e., functions that do not belong to a class). This is a very advanced solution for version control of arbitrary objects, but the serialization format is still fixed.

All of the systems described above have in common that objects and their meta-objects (e.g., classes and prototypes) and the programming tools *live* in the same environment. More technically, the programming environment and the manipulated objects share a single execution environment, such as an image or a web page. This is different from typical programming environments for languages like C, Java, or Python, where the execution of the program under development is, at development-time, usually short-lived in comparison to the programming environment: during a programming session, the source code is repeatedly compiled and run (for testing or debugging), then terminated for another iteration of the edit-compile-run cycle. When an integrated development environment (IDE), such as Eclipse, is not used, the programming environment may simply be a text editor or a shell. The implication is a stronger separation of the software artifacts (the code and resources) from the programming environment, and the ubiquitous use of the file system as a medium of data exchange between the two. In contrast, loading equivalent software artifacts into a programming environment such as Squeak/Smalltalk can have immediate side effects. Checking out a class for a Java program will certainly simply replace a single file of source code, and the program is probably not being run at the same

time. And even if it were, checking out the class (or rather its source code) in the file system will usually not affect the running program. Checking out a class in Smalltalk does imply changing an existing class definition and possibly compiling some methods, immediately affecting all existing (which is kind of equivalent to "running") instances of that class. This is a fundamental difference between tracking *live* objects and tracking ("dead") source code. This difference must be accounted for in a version control system for objects.

2.2 Goals for a new version control system for objects

The general aim is to put arbitrary objects under version control, not only meta-objects related to source code. To achieve this, source code and meta-objects should be strictly regarded as a special case of objects that can be put under version control. The diversity of domain objects and their possible repertoire of suitable data exchange formats should be accounted for by separating the serialization of objects from their captured snapshot representation. This separation is already realized in Monticello because the serialization format is determined by the repository type, not by the definition objects. But if the snapshot objects have no control over the serialization, there can be no flexibility in file formats for one kind of object. Moreover, certain version control operations, such as the handling of differences, should also be under the influence of domain-specific types. They may have special requirements for an operation (e.g., to produce differences that are at all useful for the consuming users) or the nature of a type might offer opportunities for improvements over a domain-unspecific, fixed set of procedures and tools. Another reason is that the knowledge and code about the representation of domain objects can stay close to the domain objects themselves, rather than being spread out into modifications to a Monticello repository type or custom merge drivers for Git, for example. In contrast to version control via import/export mechanisms such as the Self Transporter, file out, or saving objects to files manually, version control for objects should be controlled from inside the exploratory programming environment. This should make it possible to build tools that are as tightly integrated with the programming environment as demonstrated by ENVY/Developer.

On the other hand, the new system should not prompt for version control specific specialization too eagerly: it would be daunting for users if they had to supplement all their domain objects with suitable version control integration code first, before these objects could be tracked. Instead, an adequate solution that is already sufficient for many objects must be found. As far as possible, specializations should be an option, rather than a requirement.

One key requirement for version control systems is stability and robustness against data loss. After all, these systems are built to *preserve* past or concurrent alternatives. To meet the desire for stability or maturity, existing VCSs are preferred in productive use over experimental, new ones—until the new ones have matured and their technological advantages warrant a switch. Therefore, the new VCS for objects (which undoubtedly is experimental in the beginning) should be built on top of an exist-

ing VCS. In this way, the original, canonical set of tools of the reused VCS can be applied to repositories of the new system as well, filling the gaps in tooling for the new system and aiding in recovery scenarios. Moreover, the canonical tools can help to verify the correctness of the new system, as far as the underlying reused VCS is concerned (e.g., Git provides the git fsck tool to check the integrity of a repository).

Sharing versioned files (or objects) usually involves a central place to host and exchange the versioned data. While this is not technically necessary in distributed version control systems like Monticello, Git, or Mercurial, the presence of platforms such as *SqueakSource*[4], *GitHub*, or *Atlassian Bitbucket*[5]indicates that central repositories are very much desired. Many of them come with additional tools that complement the software development process, such as issue trackers or wikis. Reusing an existing VCS to base the new one on also means that the existing platforms can be reused. This frees the new system from the cost of implementing and, more importantly, maintaining and running an equivalent solution to host projects. Established and well-known platforms like GitHub and Bitbucket have proven their maturity, availability, and popularity in this regard.

Another reason to reuse an existing and especially a file-based VCS is that software may depend on resources that are better stored as external files (e.g., because they are large or infrequently accessed, not warranting storage in the programming environment), and that these external files should also be version controlled along with the artifacts resident in the image.[6] In file-based programming environments (e.g., in Java projects) this is supported naturally because there is no difference between source code and other files, as far as the VCS is concerned. Support for external files becomes even more important when multiple programming languages are used in a software project, and the different parts must be developed in different programming environments. For example, a project might have some modules written in Smalltalk and others in C, or there might be auxiliary shell scripts. Like resource files, the primarily file-based source code must be kept in synchronization with the Smalltalk parts as part of the software configuration management.

2.3 Prerequisites and prior knowledge

2.3.1 Git's object model and repository structure

Because the prototype implementation presented in chapter 4 uses Git, a brief introduction to Git's object model is given here.

At its heart, a Git repository (which is usually the *.git* directory in cloned directories) is a key-value store of hashed objects; the SHA-1 hashes of these objects are the

[4] http://squeaksource.com (last accessed November 5, 2017).
[5] https://bitbucket.org (last accessed November 5, 2017).
[6] This is also a weakness of Monticello because there is no easy way to connect external files with a Monticello version.

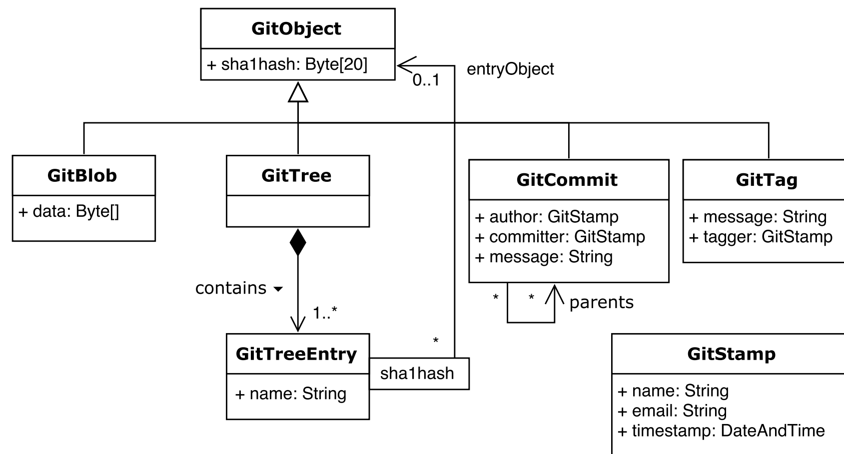

Figure 2.2: Git object types

keys, and the objects themselves are the values. There are four kinds of objects: blobs, trees, commits, and tags (see Figure 2.2). Blobs are made from tracked files and store some text or binary content as added (with `git add`) to the repository at some point in time. Trees represent directories and each tree contains a set of tree entries. Each tree entry refers (by hash) either to a blob, or to another tree (a subdirectory), or to a commit (in the special case of submodules). Trees and blobs together can capture a file system hierarchy. Modifying a file will create a new blob with a different hash than before. Because trees refer to their entry objects by their hashes, the tree for the directory that contains the file will also be modified. This changes the tree's hash in turn, so each change to one file ripples up through all parent trees up to the root tree. Commits form the version history in Git. Each commit refers to a tree, which captures the state of files and the directory structure when the commit was created. One commit can have an arbitrary number of parent commits. No parents mean the commit is "orphaned", which means that a new line of history starts there. One parent is the usual case when one commit is made as a successor of another. Commits with two or more parents are merges, where two or more divergent lines of history are reconciled. Tags refer to the tagged object (usually a commit) by hash and decorate it with additional data, such as the tag message or a cryptographic signature. [6, ch. 10]

From the structure of this model, certain performance characteristics can be derived. Because there are no direct history links that connect versions of blobs or trees with another, the detection of differences must always start from the root tree of a commit. Thus, the worst-case complexity to detect if there are any differences between two versions of a blob is linear in the number of path segments from the root tree to the blob. If any intermediate tree's hash is unchanged between the two compared versions, the blob must also be unchanged.[7] This effectively prunes the search

[7] All of this assumes that no collisions in the SHA-1 hash function appear, of course.

for differences. The absence of blob- and tree-specific history is also the reason why copy and rename detection must be performed by heuristics in Git. For comparison, *Mercurial* does keep own histories for each file and directory in its revlogs [42], and can therefore track renames and moves directly.

Git's object model actually does not contain any notion of differences; Git has a state-based history model [35]. All differences are computed on demand by Git's tools. For comparison, the earlier VCSs *SCCS* and *RCS* stored revisions of files by recording the differences from one revision to another, [20, 55, 66, 67]. A contemporary VCS that is based on changes is *Darcs* [56].

A Git repository contains more than the key-value store of objects. Most importantly, each repository has a dictionary of *references*, which are mutable named pointers to Git objects. They are stored in the *refs* subdirectory of a Git repository in the file system. The references located under *refs/heads/* are the local *branches*. The mirrored references of remote repositories, which are called *remote-tracking branches*, are stored under *refs/remotes/<remotename>/*. They are created and maintained by `git fetch`.

2.3.2 Diverse representations for objects

There can be more than one way to represent one kind of object. Take a `Form` in Squeak, for example, which is a rectangular picture that can be rendered on a graphical medium. When it should be stored in a file, many different image formats could be used to do so. Or a custom binary format like that of `DataStream` could be used, trading interoperability with other programs working with images for a possible performance benefit during serialization and deserialization.

Another example are compiled methods in Smalltalk. They can be represented as they are, as literals and bytes (the byte codes), or by the source text. Capturing the former may be brittle because the literals could change (when global variables are rebound or classes are replaced) and it is specific to the byte code set employed by the virtual machine. But restoring a compiled method from such a snapshot could be much faster than compiling the source code again. Some common requirements for snapshot and serialization formats, among which a trade-off must be made, are performance, portability and interoperability, expressiveness and completeness (i.e., that no information is lost), and human-readability.

The most suitable form of representation might not even depend only on the type of object, but also on the use case of the representation. In the compiled method example, for collaborative development on GitHub, classes and methods are best represented in the form in which they are written by the developers (i.e., as source code). However, if the use case is to distribute the software (transfer it to another image) it can be more beneficial to share a binary representation of the same classes and methods for performance reasons. The latter is done for loading methods in *Orwell* [65]—a predecessor of ENVY [51, p. 2]—to save the compilation time, and VisualWorks/Smalltalk *Parcels*, which offer a way to load and unload sets of classes and methods fast, but in a controlled manner [47].

2.3.3 Object lifecycles and vocabulary issues

In the previous section, it was noted that *live* objects may need to be treated differently than files, because *loading* such objects may cause side effects. Therefore, it should be possible to inspect other *versions* of objects without *loading* them. Monticello solves this by introducing another kind of *meta-objects*, the MCDefinition and subclasses, that are bundled together in an MCSnapshot. Unfortunately, the objects captured by MCClassDefinition (i.e., classes), are *meta-objects* themselves (for their instances). So, how to call the objects in a snapshot? If they are contrasted with *live* objects, are they supposed to be *dead* objects? At least, there is no real difference to the execution environment. ENVY calls them *shadows* [51, pp. 237 ff.]. How does *loading* relate to the *deserialization* of data from files?

Even without the issues connected with all the terms printed in italics in this section so far, different VCSs have used different terms for equivalent operations or the same term for different operations in the past. For example, while Subversion and Git coin *commit* (as a verb) to create a new *revision* (in Subversion) or *commit* (as a noun, in Git), *SCCS* and *RCS* coined *checkin* for the same operation [20, 55, 66, 67]. In ENVY, it is *version* (as a verb), and in Monticello it is *save*. Even back in 1990, the lack of consistent terminology has already been noted [1]. Since, today, there is no agreement among Subversion, Git, and Mercurial on how a *branch* should be constituted [35], and there are multiple explanations of what *checkout* does in Git alone, the situation does not seem to have improved much.

For an overview of selected terms associated with the lifecycle of objects under version control, have a look at Figure 2.3. The terms used in the remainder of this report

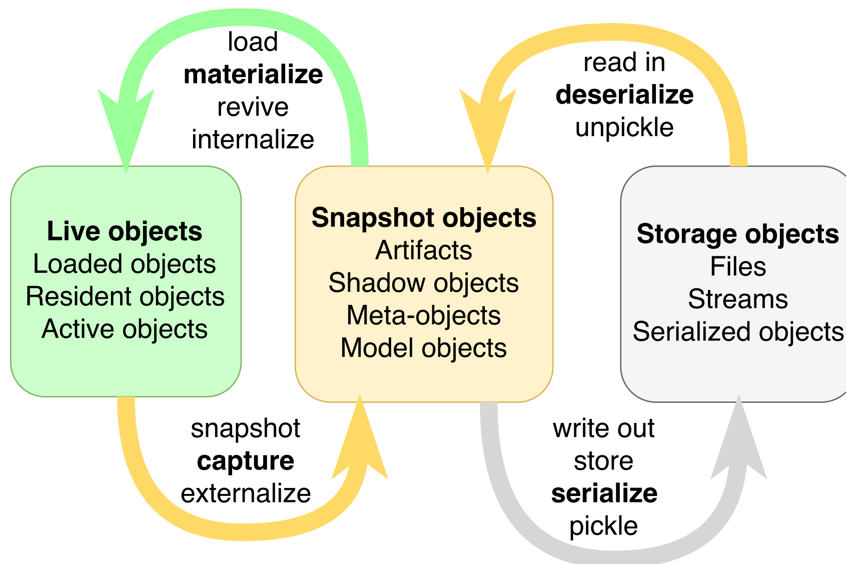

Figure 2.3: A multitude of terms is available to describe operations of converting objects between different representations. The ones chosen in this report are printed in bold.

(with some deviations in chapter 4, which describes the prototype implementation of the proposed design), are the following:

live object An object that would exist even without any support for support for version control in the programming environment.

snapshot object An object that represents a live object for the purpose of version control.

to capture an object Convert a live object to a snapshot object.

to materialize an object Convert a snapshot object to a live object. This may have side effects on other live objects, as noted earlier in this chapter.

tracked object A live object that is currently considered for version control and that can be captured at some point.

captured object Usually a live object that has been captured.

to serialize an object Convert an object to a series of bytes, for storing the object to a stream, which can end up in a file.

to deserialize an object Convert a series of bytes that was generated by serializing an object back to an object.

version (without a referent noun) An object that describes a set of object graphs at some point in time, with metadata such as the author who created this version. Such versions form the version history in a repository. The Git equivalent would be a commit.

version (of an object) An object as present in a *version* as defined above.

edition (of an object) An object in a state that needs not necessarily be present in any version. It could be a live object with changes not persisted in a version, or a modified snapshot object that was derived from applying only *some* of the differences between two versions to a version of an object. The set of versions of an object is always a subset of the set of editions of an object.

to apply differences to an object Transform an object from one edition into another edition of itself. Synonym: *to patch* an object.

merge The operation of combining three editions of each object in a set of objects into one edition, and the result of that operation.

The choice of *version* and *edition* are partially based on their usage in ENVY, where versions are immutable editions that have been given a name (usually including a number) [51].

The color code used in Figure 2.3 will also be used for other figures in this report when applicable: live objects are green, snapshot objects are yellow, and objects that result from serialization or are involved in that process are gray.

2.3.4 Object graphs

In general, one single object does not have a significant meaning on its own.[8] Instead, what can make a given, seemingly interesting object meaningful is a graph of objects that is reachable from it. For example, given a user interface form that contains a box for text entry and a button to accept the entered text, the form alone would not be meaningful without its contained objects (the text box and the button).

On the other hand, not all objects that are reachable from a given object might be relevant for the purpose of tracking this object for version control. For example, in Squeak's implementation of Morphic, each morph has a reference to its containing morph, the owner. If the user chooses to track one morph, its owner might not be relevant for versioning the morph; the owner can change whenever the user puts the morph into another space of the programming environment. The change of owner is therefore only relevant to organize the environment for *one* user, not for all of them and, subsequently, not for tracking and sharing the morph. Further, if the owner reference were always followed unconditionally, tracking any morph that happens to be visible on the screen in Squeak would mean to track the whole *world* of visible morphs and their associated objects. By following the chain of owner references, the root morph that transitively contains all morphs visible on the screen can be reached from every visible morph. Tracking every morph on the screen as a consequence of tracking a single morph is certainly unexpected and undesired.

While the owner reference of the initially tracked morph may be irrelevant, the owner references of morphs contained in the tracked morph *are* relevant because otherwise the part-of relationships among these morphs would not be captured (or inconsistently captured). So, whether a particular instance variable is relevant for capturing must in general be decided per instance, not per type.

2.3.5 User intentions missing from object graphs

The generalized variant of the issue raised above with the example of morph owners has already been documented for the Self Transporter. According to [68], the following information is generally missing from graphs of extensionally constructed objects, but it is required when these objects should be transported from one system to another. Therefore, the information must be supplemented somehow:

- To which package a part of an object belongs (different parts could belong to different packages),
- whether a reference from one object to another should be captured as is or whether the referent should be replaced by a different value in the transport representation,

[8] Numbers and primitive values such as nil, true, and false may all be complete on their own, but it is not useful to track any of them standing alone. They only make sense when put into a context: *What* is true or false?

- whether a referenced object is a global object that is assumed to be already present in the target system, thus making it preferable to capture only a symbolic reference to the object instead of the object itself,
- whether the identity of an object matters when it is restored, and
- whether an object should be recreated from an abstract expression rather than from a complete snapshot representation.

The first point is applicable if packages are put before objects, that is, the entities that are being tracked are packages, which happen to contain objects. As per the goals set out above, the reverse should be assumed: a package is just a special kind of composed object (that may refer to only parts of other objects such as classes or prototypes, to address the issue raised in [68]). Therefore, it might not be necessary to annotate parts of objects with packaging information. However, it is important to be able to control the relevant "reach" of an object, depending on the intention for tracking it. It should be possible to ignore a portion of an object's state when capturing a graph of objects, for example the owner reference of a morph that has been selected for tracking, as discussed above.

The second point applies, for example, to transient state (such as caches) which should probably not be put under version control.

The third point indicates that some objects provide a context for others, but this context is fixed by the programming environment and should neither be transported out of the environment, nor modified when other versions are loaded into the environment. In Smalltalk, one example of such an object is the global Smalltalk dictionary, which usually manages the bindings of all globally accessible identifiers, including class names. While loading different versions of packages might actually modify these bindings, the mapping object itself does not usually need to be tracked and should not be tracked, as it is bound to the running instance of the programming environment (e.g., a Smalltalk image).

The fourth point derives from the need that multiple references to the same object must be transported such that when the references are restored in the target environment they will all refer to the same object again. However, there are exceptions, such as value objects, for which this requirement does not necessarily apply. For example, a point of x and y coordinates in Self does not have an identity, so it is irrelevant whether two references to a particular point will be stored and restored as references to two equal points or as references to the same point object.

The fifth point from the list above derives from the capturing approach of the Self Transporter, which transforms object graphs into an evaluable stream of Self expressions. These could either allocate a new Self object and initialize it by assigning the results of other expressions to its object slots, or instead use special message sends to construct an object (such as the `@` message to construct points, i.e., `x @ y`). Since we do not want to dictate a particular serialization format or even prescribe the in-memory representation of snapshots for all kinds of objects, this point can be reformulated into "[It must be defined] whether a special type of snapshot should be used to represent an object."

All of these issues require that objects or whole object graphs must be complemented by additional information. In the remainder of this report, such information will be called *object metadata* (or only *metadata* for short).

3 An architecture for object versioning

In this chapter, we present our framework solution to track objects and store them in existing version control systems. It is described how object graphs can be captured and rematerialized, how object identity can be preserved in this process, how differences between two editions of object graphs can be handled, and how it can be supported to have different formats in which snapshots can be stored in an existing VCS. Finally, a generic way to capture any kind of object is presented, so that users do not have to provide own solutions for snapshots and differences for the types of all objects that they want to track.

3.1 Storing objects in versions

In order to put anything under version control from inside of an exploratory programming environment, there must exist a connector component to a version control system. It should be able to access the version history of a repository, create new versions, and possibly manage independent development streams, such as branches. How this connection can be established without focusing on a particular VCS has already been treated by related work and is not in the scope of this chapter (some details will be given in chapter 4). However, some assumptions must be established on how objects can be put into versions and how they can get out again. This part of the architecture is based on a subset of an abstraction for version control systems called Pur [35].

A *version* describes a revision of a set of objects. Versions can have any number of parent versions. This relationship forms the version history in a repository.

Each version contains a *snapshot* of object graphs and their associated metadata. In contrast, the Pur architecture deliberately does not define what a snapshot consists of because it depends on the particular application that is seeking to connect to a version control system. However, Pur defines an entity named *store* that can create snapshots and restore snapshots, updating the objects in the store. What "restore" means depends on the particular type of the store. The example implementation *Pur for Newspeak* presented in [35] defines two stores: an *image store* to capture classes and methods from the Newspeak environment into a snapshot and to load them back from a snapshot, and a *file store* to write snapshots to files and to read such files back into a snapshot.

In the following sections of this chapter, we will further define the snapshots of our version control solution for objects and what they must be able to do.

3.2 Live objects, snapshot objects, storage objects

The distinction of stores and versions from their snapshots implies that there can be at least two or three kinds of forms to represent the same object. Three realms of objects will be defined into which these forms are categorized. This makes communication about the different forms and about crossing the boundaries of the realms easier. The three realms were already shown in Figure 2.3, but not formalized.

The objects that the users usually interact with and that make up the programming environment are attributed to the *live realm*. Domain objects belong to this realm, as well as their meta-objects that define their types and behaviors, such as classes and methods in Smalltalk and functions (closures) in JavaScript. A defining property for live objects is that they do not exist for the purpose of version control. They would still exist when version control is not even attempted.

The *snapshot realm* contains objects that stand in for editions of objects that originally came from the live realm, either from the the running system or from another system. Snapshot objects can represent either current editions (i.e., their captured state is equivalent of that of live objects in the running system) or editions from other points in history. Objects in the snapshot realm reside in the memory of the running programming environment, just like live objects, but they solely exist for the purpose of version control. Objects for differences between editions, and other objects that explicitly deal with snapshots are also attributed to the snapshot realm.

The third realm includes all forms of objects that are intended to be stored outside of the programming environment. Since the most pressing reason to export objects in the context of version control is persistence, this realm will be called the "storage realm". But even when snapshots from the version control solution were to be transferred directly between two running programming environments, the representation "on the wire" between the two processes would belong to the storage realm, according to this nomenclature.

Converting objects from the storage realm to the live realm always goes via the snapshot realm. The same holds for the inverse direction.

There are additional objects in the proposed design that solely exist for the purpose of version control, but they are not snapshots because they do not represent live objects. They deal with objects of one particular realm or operate at the boundary between realms, converting objects from one realm into another. Since these objects are in a way aware of version control and of the different involved realms, they will be called infrastructure objects. Examples of these are stores, versions, and repositories. If they deal with objects of one realm specifically, they will be attributed to the realm on which they operate. If an object operates in multiple realms, they will be attributed to the realm that is not the snapshot realm. For example, serializers that convert snapshots to storage data are attributed to the storage realm; an object assuming the role of a store for live objects would be attributed to the live realm; and an object that computes differences between snapshots would be attributed to the snapshot realm. Containers of objects in the snapshot realm (rather than containers that can convert objects to snapshots on demand), belong to the snapshot realm. For example, the

snapshot that is associated with a version (and contains other snapshots), belongs to the snapshot realm.

3.3 Preserving object identity across system boundaries

In subsection 2.3.5 several issues were already raised that must be solved when attempting to transfer objects from one environment to another. If an object should be tracked for version control, additional requirements arise. It must be possible to discover an object in a target environment if it already exists there. Also, it must be possible to identify that different snapshots of an object belong to the same original object. Otherwise, restoring a graph of objects from a snapshot would either have to rely on the structure of the graph to identify corresponding objects (e.g., starting from a well-known object, the object reached via a certain path will be assumed to correspond to a snapshot object in the snapshot graph via the same path), or it would have to rematerialize the whole object graph from scratch (based on the snapshot) and only a well-known references were redirected from the “old” graph’s objects to the “new” ones. This might be sufficient in some cases, but in general there may be objects outside of a captured graph that have references to non-well-known objects inside the graph (see Figure 3.1 for an illustration). These references would become stale if the objects of the captured graph are not updated in-place.

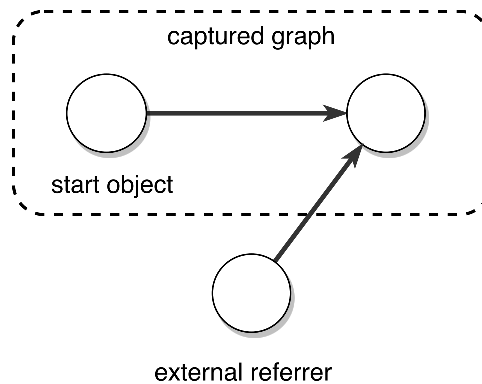

Figure 3.1: External referrer into a captured object graph

To identify objects in different snapshots, we propose that *names* are assigned to objects if the identity of these objects needs to be preserved. By default, object identity needs to be preserved for all objects, but some objects can be exempt: identity can be ignored for value objects, and for objects that can be identified and accessed in the target environment based on their properties (e.g., a PackageInfo object in Squeak is identified and accessed by its name, so it is unnecessary to assign another, technical name to it). Object names must be globally unique, even across the boundaries of the programming environment. They can take any suitable form for in-memory

and external storage, but they should make it easy to avoid collisions, should not depend on the state of an object (e.g., they should not be hash values), and should not consume much memory (to keep the overhead low). One possibility are UUIDs,[1] although these are hard to read and remember for humans.

3.4 Capturing and materialization of object graphs

In this section, we describe how graphs of live objects can be captured to create graphs of snapshot objects, and how the inverse operation, materializing live objects from snapshot objects, can be performed. All of this does not assume any particular application domain.

3.4.1 Composition of version snapshots and object graphs

The snapshot of a version or a store is a collection of object graphs. Each graph is associated with object metadata that saves decisions about the capturing or storage, which might also be needed to recover the graphs properly. In addition to that, a key is assigned to each graph. This key uniquely identifies the graph in the snapshot and in the store that creates the snapshot. It is also used to access a graph in a snapshot.

Each object graph stores the bidirectional mapping between object snapshots and the names of their captured objects. Additionally, each graph has a start object, from which all other objects in the graph can be reached. New graphs are introduced to a store by telling the store to track an additional object (under a given key) that is the start object of the graph. The other objects in the graph are then derived from the relationships among objects, guided or restricted by the metadata that is configured in the store. Snapshot objects can refer to each other, so all snapshots (nodes) in the graph should be reachable from the snapshot of the start object of the graph.

To capture the snapshot of a store, the store must enumerate the live object graphs that are known to it, convert them to the snapshot realm and collect them into the overall snapshot, together with the metadata (see Figure 3.2).

3.4.2 Abstract algorithm to capture object graphs

To capture an object graph, beginning from a given start object, the graph has to be traversed and an appropriate snapshot has to be created for each encountered object. All objects encountered that do not already have names from a previous capture operation (or because they have been materialized from another snapshot) must have new names assigned unless their identity does not matter. These names should be persistently kept in the store that tracks live objects, so future operations on the same graph of objects can look up the named live objects if they still exist. The names

[1] Universally Unique IDentifier, https://tools.ietf.org/html/rfc4122 (last accessed November 5, 2017).

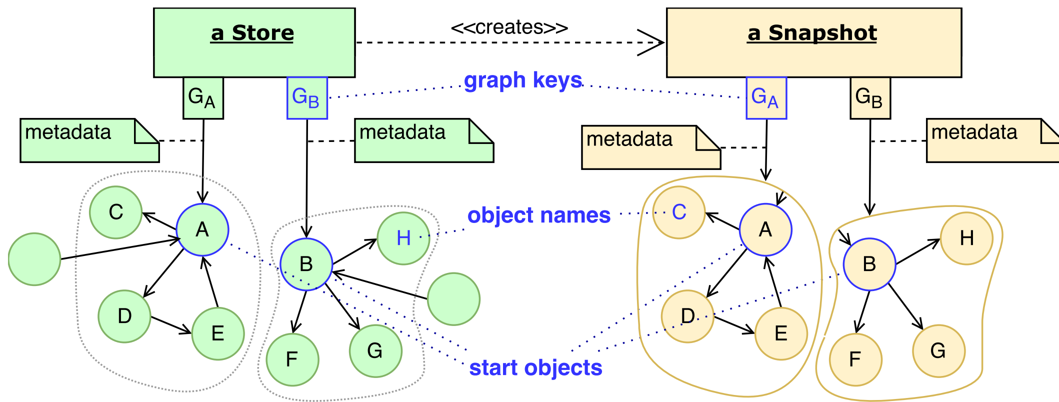

Figure 3.2: Example setting for object graphs, start objects, and object names. The store captures all objects that are reachable from its known start objects.

are also assigned to the respective snapshot objects, so the corresponding live object of a snapshot object can be looked up and vice versa.

The graph traversal can be realized with an exhaustive search algorithm (e.g., breadth-first search for graphs). The algorithm should include multiple-path pruning, so each object is captured only once. The live objects that are being captured must be able to direct the graph traversal to related objects, so the traversal can progress through the graph.

When a live object is encountered during the traversal, a message is sent to the live object, instructing it to convert itself into its preferred type of snapshot. This is the opportunity for a live object to decide that it should be replaced in the snapshot graph by another object (e.g., by a symbolic reference to itself).

Live objects that know that they are the root of a sufficiently independent substructure in the overall graph could also decide at this point to start another (independent) graph traversal that works differently for this subgraph (see Figure 3.3). They must take care not to miss out on the multiple-path pruning of the main traversal. If the snapshots of the objects in this subgraph should be registered normally (with object names) in the snapshot graph that is built by the outer graph traversal, a way to pass the inner snapshots out to the main graph must be implemented. Alternatively, the very aim of the separate traversal might be to encapsulate the results of the special traversal in a single (composite) snapshot object.

Live objects (and auxiliary objects involved in the capturing traversal) must further be able to access the object metadata themselves, so gaps of missing information as described in subsection 2.3.5 can be filled with this metadata. For example, when one attribute of an object should be captured with a default value instead of the actual value, this needs to be looked up in the metadata.

There might be more than one way to capture one type of live object. For example, a CompiledMethod in Smalltalk could be captured in source code form or in its compiled byte code form (or in the form presented below in section 3.7). To support the choice that an object should be captured with a different type of snapshot than the

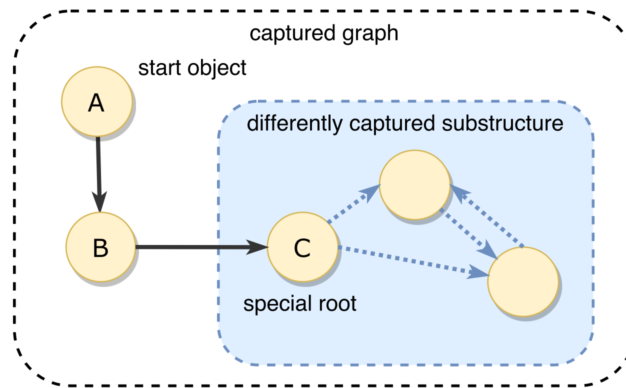

Figure 3.3: Different graph traversal strategy for a subgraph. The different line styles indicate that the relationships among the objects in the subgraph could be of a very different nature than the relationships outside of the subgraph (e.g., references from pointers vs. references derived from a naming convention).

one the object usually prefers, the necessary override information must be added in the metadata—associated with the particular live object to which the override pertains—and it must be interpreted by either the graph traversal or the live object itself. For example, the metadata could contain an association of the live object with a different message that will be sent to it for capturing, when the object is encountered.

3.4.3 Abstract algorithm to materialize object graphs

When snapshots of object graphs can be created, it must also be possible to convert them back to live objects. Like capturing, this can be realized with a traversal of the object graph (now the graph of snapshot objects), starting from the snapshot of the start object. As during capturing, each object should only be rematerialized once.

When a snapshot is to be materialized, a message is sent to it to convert it back to its original live object. Depending on the kind of snapshot this may involve, for example, creating a new instance of the type of the captured object, or compiling source code. If the captured live object had itself replaced by another during capturing, at this step the replacement object will be rematerialized instead of the captured object. To get the original live object back, another message is sent to the materialized object, essentially telling it to "bring itself back to live". A symbolic reference to a globally accessible object would at this point resolve itself and return this global object. Live objects can also perform other post-materialization tasks in response to this message if necessary, such as notifying observers of changes.

When snapshots are materialized, it is possible that a live object for the snapshot already exists (i.e., there is a live object with the same name known to the store). In this case, the snapshot object should be instructed to materialize itself *into* the existing live object if possible. This attempts to ensure that references to the live object do not become stale by materializing a new live object.

3.4.4 Containers of tracked objects

Stores and the snapshots produced by them contain objects in different forms (as indicated by the different realms), but ultimately they always contain graphs of objects. In a way, the same holds for versions, which contain the objects in their snapshots. Retrieving the snapshot can involve delegating to a store that retrieves objects from the storage of the backend VCS.

Therefore, at a higher level of abstraction, stores, versions and their snapshots can share a common set of operations. For example, it should be possible to compute the differences between two stores, between two versions, between a store and a version, between a store and another snapshot, and between a version and another snapshot (each time resulting in a set of differences for object graphs). Consequentially, another such operation would be to apply a set of differences on top of a store or on top of the snapshot obtained from a store. Applying differences to a version would be somewhat ambiguous because it is unclear what the result should be: a new version or only a different snapshot? When such operations are performed on stores, in contrast to snapshots, these operations will affect objects in the live realm or the storage realm, introducing the side effects mentioned in the previous chapter. Applying an operation to a snapshot would constrain the operation to the snapshot realm.

To generalize such operations, we introduce the abstract type *object container* as the union of stores, versions, and snapshots obtained from them. To realize some of these operations when performed on different concrete types (e.g., when a version is compared to a store), the object realms need to be crossed transparently by converting objects to and from the snapshot realm whenever necessary. At the same time this means there is potential for optimization, since containers of the same type (e.g., versions from the same VCS backend) may implement shortcuts that avoid the unnecessary conversion of some objects.

3.5 Differences between snapshots

After having described how objects can be converted between the live realm and the snapshot realm, in this section we will motivate the need to describe and compute differences between object graphs, describe how it can be done (independent of a particular application domain) and how these differences can be applied to object graphs.

3.5.1 About the importance and granularity of differences

It would probably be possible to build a simple version control system that does not have a concept of differences between two snapshots. In fact, the Git object model does not include differences, as mentioned in the previous chapter, and Pur also omits them. But the tools that come with Git can compute differences for trees and

blobs and the example implementation "Pur for Newspeak" presented in [35] also includes differences.

Differences are important, not only for the users to consume, but also for optimizing certain operations. For example, when a new version is to be saved, first computing the differences between the working copy and its parent version makes it possible to ignore all objects that were not changed. The rest of the save operation can take advantage of this knowledge. The unchanged objects may not need to be serialized again (depending on the storage mechanism of the backend VCS) and can be reused for caching of the newly created version. If a tool offers to create the new version with only a subset of the changes made to the working copy (as `git add –patch` does), it might need to reevaluate which objects need to be serialized after the selection of changes has been performed. But it is guaranteed that the initially unchanged objects require no further processing. Based on the assumption that only a small part of a system changes from one version to another, processing only the differences can mean that fewer objects must be processed overall per operation.

Implementing differences requires an investment of additional development time (in comparison to building a purely snapshot-based system). But the effort can be worth it for both the user experience and the performance of the version control system.

There are multiple levels at which differences can be computed.

1. At the object container level: Which object graphs have changed?

2. At the object graph level: Which objects have changed?

3. At the object level: Which parts of an object have changed?

The simplest form of difference is replacement as a whole. It can be used as a fallback whenever differences are not implemented at one of the three levels. For example, if a particular domain object is captured in a form of snapshot for which fine-granular differences are not implemented, the object can be stored and restored by replacing the whole object with a new edition that was created from the snapshot, as described in the previous sections. In programming environments that support an identity replacement operation, such as Smalltalk's `become:`, this replacement may even be trivial. An alternative approach is to overwrite the existing object entirely, modifying it in-place. When this is not always possible (e.g., because it might not be possible to change the size of an existing object), it might be necessary to not only track the object itself, but also the relevant places that reference it.

If the system does not implement the detection and application of differences at the object graph level, the whole graph must be replaced. If differences are not available at the object container level, all tracked objects must be processed with each operation that transfers objects from one realm to another.

For object graphs, there are two additional trivial forms of differences at the level of object containers: addition and removal of graphs to and from an object container.

3.5.2 Abstract detection of differences

The structure of differences depends on the structure of the specific snapshots. However, the structure of the collection of differences in an object graph and the structure of differences between object containers are amenable for generalization. The general principle of detecting changes is described in this section.

Remember that an object graph refers to the snapshot of a start object and that it includes a bidirectional mapping between object names and snapshots. Consequentially, the differences between two object graphs can be expressed as the collection of differences between the corresponding snapshots in the compared graphs; that is the differences between the start objects plus the differences for any objects with the same name.

To collect the individual differences for the snapshots in an object graph, another form of graph traversal can be performed, but this time the traversal happens not in a single graph of live objects, but simultaneously in two or three graphs of snapshots that should be compared. One of the graphs is the *left*-side graph, containing snapshots "before" certain changes that should be detected, the second is the *right*-side graph, which captures the situation "after"[2] the changes, and an optional third graph would be the *base* graph, which contains snapshots from the base version of a merge, or more generally the base version of a three-way difference of graphs.

Assume two (or three) graphs that contain some snapshots with names that exist in either graph, but with different snapshot contents for the same name in some cases. Beginning from the snapshots of the start objects in each graph, two (or three) snapshots are compared in each step. The determination of the local differences (i.e., changes that apply to the memory of a single object) is up to the implementation of the snapshots being compared because they must know best their own structure. To that end, one of the snapshots is told to compare itself to the other snapshot (e.g., the left-side snapshot receives a message to compare itself to the right-side snapshot, based on the base snapshot) and the result must be some kind of differences object. We can assume that the comparison will somehow iterate over the relevant relationships of the snapshot object (the "referrer") with other snapshots and that the other ends of these relationships may need to be compared among the two (or three) graphs. Thus, per relationship two (or three) other snapshots (the "referents") will be reached in this way (see Figure 3.4).

If the referent snapshots all have the same object name, then the live referrer object was in relation with the same live referent object at the time of each capturing. This means that there is no change in this particular relationship. The referents must then be compared in a subsequent traversal step, to detect differences deeper in the graph.

If the referents have different names, then the relationship has changed (e.g., an instance variable has been reassigned) and, thus, a difference for the referrer exists. While users might be interested in the differences between these two (or three)

[2] The terms *before* and *after* are set in quotation marks because they imply a chronological ordering which needs not be accurate for all comparisons.

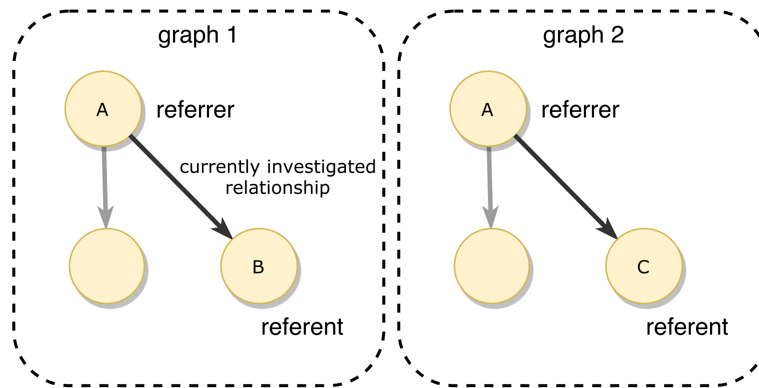

Figure 3.4: Two graphs being compared, currently inspecting a particular relationship from the two editions of the object named A. In graph 1, B is at the end of this relationship; in graph 2, it is a different object C. B from graph 1 and C from graph 2 are the referents in this relationship, while A from graph 1 and A from graph 2 are the referrers. The change in name from B to C means that the two captured editions of A related to different objects.

distinct referent objects, sometimes this comparison does not make sense (e.g., when two of the referents are of completely unrelated types). The differences between the referents could be computed, but they would only be informational for users and must not be applied to an object graph because the left-side referent did not actually mutate into the right-side referent.

If one of the referents does not have an object name, meaning that the identity of the captured object is not tracked, further information is needed to determine the correct course of action. If this is the snapshot of an immutable value object, then the difference applies to the referrer. If the snapshot is of an object that should be mutated and that supports fine-granular differences in *each* of the two (or three) compared graphs, meaning that the referent object would not need to be replaced at the referrer, then there is no difference for the referrer. But if some of the referent snapshots have names and others do not, or some are mutable and others are not, then the difference must in case of doubt be applied to the referrer.

It could be mandated that all snapshots must be able to answer the mentioned questions: is it immutable, does it support fine-granular differences, and does it support fine-granular differences also when compared to the specific other snapshot(s)—they could be of incompatible types. But the more queries or operations all snapshots must support, the more difficult it will be for users to create correct domain specific snapshot implementations. Another solution to find the correct course of action in such cases is to keep the snapshots from all involved graphs and present options on how to proceed to the users who might want to apply these differences. Change the referrer or change the referent? Attempt to compare the referents or not? Unfortunately, this makes the user experience of such cases similar to that of a proper conflict. It also means that the difference finding graph traversal must be able to resume after user input has been provided. Yet another solution would be to

annotate each relationship that can run into these problems with the correct answer to these questions (thus, defining a strategy for comparisons) in the object metadata. Since this might be daunting for users if it had to be done in advance (e.g., when an object is enlisted for tracking for the first time) the two latter solutions should be combined: if one user has made a choice for a particular relationship while looking at differences, it should be offered to persist the choice in the metadata for future difference computations. The simplest solution is to insist that objects without names need not be kept (they opted out of having their identity tracked, after all), so the change should always be applied to the referrer.

If a three-way difference is computed and there are three different names for the snapshots in a set of referents (or two names and one nameless snapshot), then the relationship was changed both from the base to the left-side and from the base to the right-side, but not to the same object in both cases. This is a conflict and must be appropriately recorded in the differences for the referrer.

When the referent from the right-side graph does not have a corresponding snapshot in the left-side graph, a new object has been introduced to the graph. This must be noted in the differences by copying the snapshot into the differences, so it could be added to a graph to which these differences should be applied. The memory overhead of a new set of objects in the differences could be optimized by replacing references to snapshots that have a name and that already exist in the left-side graph with symbolic references by the name. If there is a difference for the referenced object, then it will be recorded under this name anyway. For the case when an existing object from the left-side graph can only be reached via the added object in the right-side graph, the relationships of the added object must be followed (see Figure 3.5). For each referent with a name that already exists in the left-side graph, the difference traversal must continue with these same-named snapshots from either side.

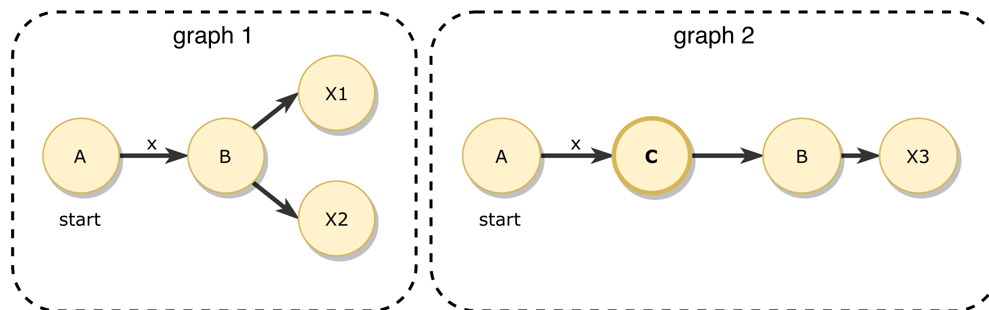

Figure 3.5: Two graphs being compared. In the right graph an object C has been added and it replaces B at the end of the reference x from A. B can only be reached from A via C in the right graph. The changes to B are "hidden" behind the added object. The difference detection graph traversal must therefore follow the relationships of C or it will not detect the changes in B.

If a three-way difference is computed and a name can be found via this mechanism in the left-side and right-side graph, but not in the base graph, then the *same* object was added in both changes from the base (which must likely have been done by back-porting objects to the base version and continuing history from there). Differences between the left-side snapshot and the right-side snapshot are then automatically conflicts.

If the right-side graph does not contain an object from the left-side graph, that object has been removed from the graph because it is no longer referenced. Whether this requires a special mention in the differences between the two graphs depends on how the object store to which these differences should be applied behaves. In a system that provides automatic garbage collection, changing the references such that the removed object is not referenced anymore in the target graph would be sufficient, so no special mention of the deletion is needed. This is also safe for uncaptured objects (outside of the graph) that refer to the object that has been "removed" (actually it is only no longer part of the captured graph). If the objects in a store are not subject to garbage collection (e.g., files on disk may not be deleted automatically), the removal of an object should be noted explicitly in the differences, so the removal can be executed when the differences are applied. For the latter case, the removal of a named object can be detected by marking each named object in the left-side graph when an object with the same name is encountered in the right-side graph. All objects in the left-side graph that are not marked at the end of the traversal are not present in the right-side graph and have thus been removed. With these semantics, objects without a name are always "removed" from the target graph and replaced by others (which may be equivalent).

When one snapshot is told to compare itself to another, the snapshot implementation can decide to start an own graph traversal for differences, much like the the possibility for live objects to start their own capturing graph traversal for a subgraph. This makes most sense when objects pursue this way both for capturing and for comparisons.

3.5.3 Abstract application of differences

Applying differences relates to detecting differences like materialization relates to capturing. But since the differences between two snapshot graphs are made up of the collection of differences to named objects (as defined above), no further object graph traversal is needed this time. Instead, the individual differences must simply be applied to their respective objects in the left-side graph.

Sometimes, objects appear in the right-side graph that do not exist in the left-side graph. In this case, these objects have been added and this addition must be reproduced when the differences are applied.

If a store on live objects implements the application of differences and new objects must be created, they must be materialized similar as described in subsection 3.4.3, but with a variation. If a materialized object refers to another named object, this referent object must be looked up in the target graph as usual. But if it already exists, it does not need to be materialized (which would be the case in the abstract

materialization algorithm above). Instead, only the reference to this object must be materialized in the referrer, while the existing referent object is only subject to a change if there is an own difference for it.

3.6 Storing objects outside of the programming environment

The previous sections primarily deal with live objects and their snapshots. When the objects eventually have to leave the programming environment, to be shared with other programmers or authors, the third of the introduced realms, the storage realm, comes into play.

When snapshots are exported from the programming environment, they must be converted into a representation that suits the target storage or communication medium. For one type of snapshots, there might be more than one form of representation. For example, a snapshot of a formatted text could be converted into Markdown[3], HTML, or some specialized XML format. For some objects, users may want to have control over the export format, but for others, they might not care.

We note that there can be a variety of storage strategies[4] for each snapshot type. These strategies are implemented in *serializers* and *deserializers*. The responsibility of serializers is to convert graphs of snapshots (not live objects, although implementations may choose to be relaxed about this) into storage objects (e.g., files), and deserializers should do the inverse: create snapshots from storage objects. For each type of serializer, there should be a matching type of deserializer.

Snapshot types should define a preferred type of serializer that is generally suitable for the objects that they represent. For example, snapshots for character strings could refer to a serializer that outputs the text in plain text files with a Unicode encoding. So in case the user is not interested in the particular storage format, a reasonable default will be chosen.

Which serializer is used for a graph of objects must be recorded in the object metadata because users may choose a different type of serializer than the default one. This information both helps the store choose the correct serializer and it is a hint for the store how storage objects can be deserialized. If deserializers can answer the question "Can you read the output of this serializer?", then a store can choose a suitable deserializer from a pool of available deserializers based on the information about the serializer.

Because the object metadata must be accessible before the correct deserializer is known, the format of the metadata must be determined by the store, not by users or the snapshots in an object graph.

A store must therefore perform the following steps to serialize an object graph:

[3] https://daringfireball.net/projects/markdown/ (last accessed November 5, 2017).
[4] cf. strategy pattern [16, pp. 315 ff.]

1. Look up the chosen serializer according to the metadata (if there is none defined, use the preferred serializer of the start object of the graph and add that information to the metadata).
2. Instruct the serializer about the key of the object graph to be stored (the serializer may derive the final storage location from the key).
3. Invoke the serializer with the object graph.
4. Write out the metadata to the storage medium.

The steps to deserialize storage objects to snapshots are:

1. Read in the metadata from the storage medium.
2. Based on the information about the serializer contained therein, look up a suitable deserializer.
3. Instruct the deserializer about the location of the storage objects.
4. Invoke the deserializer to obtain a graph of snapshots.

Finally, a store may need to find the storage locations of objects graphs in the first place. How it does that is basically implementation-defined, but a good strategy is to maintain a dictionary that connects graph keys with locations. The store would have to store this dictionary in a well-known location and format.

3.7 Generic snapshot format for objects

The previous sections described an abstract framework for version control of diverse types of objects. In this section, a proposal is made, how objects can be captured and compared when there are no special types of snapshots available for them. Thus, this proposal attempts to make all objects trackable for version control.

3.7.1 The structure of objects

In general, objects combine data with behavior. How this combination is realized depends on the object model of the programming language. In Smalltalk, behavior is defined in classes and their methods. While each object belongs to exactly one class, it only stores its own state in the memory allocated for it, but not the behavior. In JavaScript and Self, methods can be assigned to objects directly, although most objects inherit their methods from their respective prototype object. The prototypes themselves are interesting candidates for version control, of course. Since methods and functions are first-class objects in JavaScript and Self, they can be regarded as a special kind of state because all objects will only have references to the method objects anyway, which makes them indistinguishable from "non-behavioral" state without looking at the referent.

One can say that objects are composed of a number of *slots*. This term is borrowed from Self and from the Common Lisp Object System (CLOS) to denote "a component of an object that can store a value" [52, 26.1 Glossary]. There may be different types of slots, such as instance variables, indexable fields, or unordered items of a collection. Typically, each slot references another object, but in theory a slot could also signify a meaning on its own (for example, a mark or tag on the object). Slots can have an identifier, such as a variable name, symbol, or index, but they do not need to. However, it must be possible to look up a slot in an object. In a bad case, the lookup could include a search through the items of a collection.

The snapshot of an arbitrary object is the collection of captured slots of that object, and the name assigned to the object if its identity should be kept. Specializations of this schema can be made to accommodate special kinds of objects, such as primitive values. For concrete examples, please see subsection 4.1.8. Slots could refer to other object snapshots directly or by name (if the referent has one). In programming languages with strong typing, it makes sense to keep a reference to the type of the original live object that was captured and to the type of the live object that was put as a replacement at the request of the captured object (e.g., a symbolic reference). Thus, when the snapshot is rematerialized into a live object, the correct type can be instantiated.

To make these types of snapshots applicable to the programming environment, the message that tells an object to convert itself into a snapshot must be implemented for a suitable root object, such as in the root class for all objects in the environment (if there is one).

The default implementation of the capturing message should enumerate all slots in the live object (which needs capabilities for introspection in the programming language) and add slots of the appropriate type to the snapshot under construction. If the live slots reference other objects, these should be handed back to the graph traversal as described previously, so the search through the object graph can be continued.

When the snapshot slots are created, it should be looked up in the object metadata whether the slot's value should be replaced with a default value or if the slot should not be captured at all, as described in subsection 2.3.5. Actually, this way to influence the capturing of objects is formulated very specifically for this low-level form of object capturing. Live objects that are captured with a higher level of abstraction would need an equally more abstract description of a defaulting mechanism to achieve the same goal.

To materialize an object snapshot, a new (uninitialized) instance of the type of the object of which the slots have been captured must be created. If the captured live object was replaced for capturing, the needed type is the type of the replacement object. If the captured live object was not replaced, the type is the same as the type of the captured live object. Each slot must then be materialized into the fresh object. What must be done to that object depends on the type of each slot. For example, a slot for an instance variable should assign its materialized value to the represented instance variable (which needs capabilities for reflection in the program-

ming language). After all slots have been materialized in the new object, it should be complete.

3.7.2 Differences in single objects

How objects can be mutated depends on the programming language. Commonly (at least in languages that support imperative programming) a slot value can be changed, so the slot refers to a different object after the change. The addition and removal of slots is possible if the set of slots for a particular type is not fixed (i.e., slots can be added or removed from an object of this type without changing the identity of the object or having to create a new object).

The differences between two object snapshots can therefore be described by the collection of changes to the slots of the object (reassignments, additions, removals). If an object is replaced by another one everywhere in the system, there might also be another type of difference that denotes "object replacement". But in most cases, it is sufficient and simpler to determine that slots referring to the "replaced" object are changed to refer to the replacement.

When an object snapshot is told to compare itself to another one (as described in subsection 3.5.2), it must iterate over its slots and match them with the slots of the other snapshot. For this reason, it is necessary that slots can be looked up in another object snapshot. If no matching slot can be found in the other snapshot, the slot was either added or removed. In both cases, this must be added to the collection of slot changes.

If a matching slot is found, the two slots are compared. Should the slots be of a kind that references another snapshot, the referents of the slots are passed to the difference detection graph traversal as described in subsection 3.5.2. Should it be determined that there is a local change to the referrer, which in this case means that a different object has been assigned to the slot, this reassignment is added to the collection of slot changes.

When such differences are applied to an object, each slot change in the collection must be applied to the object. Depending on the type of slot change this can mean that a slot's value is changed to a different object, or that a slot is added or removed.

4 Object version control in Squeak/Smalltalk with Git

In this chapter, we will describe our prototype implementation of the architecture introduced in the previous chapter. It has been realized in Squeak/Smalltalk and connects to Git as the backend VCS. The part that is independent of Git has been named *Squot* and the part that connects with Git has been named *Squit* (as a mixture of the names Squot and Git). The implementation is hosted on GitHub.[1]

4.1 Squot: Squeak's Object Tracker

4.1.1 Object containers and object graphs

The origin of all version control for objects are the original live objects, of course. As per the design from the previous chapter, these must be captured by a store. This is the responsibility of a SquotImageStore. The objects in it are stored in the Squeak image and they are live while Squeak is running. To start tracking an object graph, one object is chosen by the user together with a key. The chosen object will become the start object of an object graph during capturing and the chosen key will become the graph's key in the store snapshot.

Because the only implemented backend VCS at the moment is Git, which is a file-based VCS, the keys of graphs are also the paths to the files or directories in which the graphs will be stored. For example, if the package "Squot" should be put in the directory *src/Squot.package*, then the user would add it to the store as shown in Listing 4.1. Because graph keys are unified with storage paths, whenever

Listing 4.1: Adding a package to an image store

```
anImageStore add: (PackageInfo named: 'Squot') at: 'src/Squot.package'.
```

"graph keys" were mentioned in chapter 3, the corresponding code in Squot will mention "paths" instead. These paths are character strings, not objects that are used to represent file system paths (e.g., FSPath from the *FileSystem* API). Because graph

[1] https://github.com/hpi-swa/Squot (last accessed November 5, 2017).

keys only need to be unique in a store (unlike object names, which should be globally unique), the double role as paths is unproblematic.

Thus, the image store keeps a bidirectional mapping between paths and start objects. It supports to track the same start object at multiple paths, but this has not been used practically so far. Because an object can have multiple forms of snapshots and serialization formats, the possibility to add an object more than once has been considered. The implication would be redundant representations of the same object in the version history, of course.

In addition to the objects and their paths, an image store also holds onto the metadata for each path. Because the terms introduced in chapter 2 were not fixed from the beginning, variables that refer to object metadata are mostly named *storeInfo* or *additionalInfo* in Squot. The holders of metadata for an object graph are instances of SquotTrackedObjectMetadata. This is essentially a dictionary because the required attributes in the metadata cannot be known in advance. There are some well-known keys that correspond to pieces of metadata presented in chapter 3 (e.g., the name of the serializer to be used is stored at the symbol key *#serializer*).

The image store must also remember the names of objects that are not the start objects of a graph, otherwise these objects would get new names assigned each time the graph is captured. For this purpose, the image store keeps a dictionary of SquotObjectGraphs and an *object registry*. The difference is that one SquotObjectGraph stores the names of live objects and snapshots only for one graph, whereas a SquotObjectRegistry is used to keep the names of all live objects (and only the live objects) encountered while capturing from an image store. Therefore, objects that appear in multiple graphs (which leads to redundancy among the snapshots) can at least get the same name in each graph.

There are some kinds of metadata that are only relevant for an image store, and they must not be persisted in the snapshots of versions. This is called *transient store info* in Squot. Live objects are given the opportunity to initialize the metadata when they are added to an image store. When a class overrides the method squotAddTransientStoreInfoTo:,[2] its instances may add transient information to an image store's metadata, and they also have the liability to remove such transient metadata when they receive the message squotRemoveTransientStoreInfoFrom:. Other kinds of metadata must be transformed when snapshots are created. For example, the information that an instance variable of a live object should not be captured at all is stored in an IdentityDictionary that associates live objects with arrays of the ignored instance variable names. The live object must not be referenced from metadata belonging to a snapshot. Therefore, such kinds of metadata have their live object references replaced with the names of those live objects in the default implementation of squotRemoveTransientStoreInfoFrom: in the class Object. The names are converted

[2] All messages that could be implemented as extension methods to classes that do not belong to Squot have the word "Squot" in their names to avoid name clashes. Squot is not the first package in the domain of version control and object converting, and it will probably not be the last.

back to live objects by the default implementation of squotAddTransientStoreInfoTo: after the objects have been rematerialized when they are added to the image store.

When the snapshot of an image store is captured, the result is a SquotSnapshot, which is also the type of the snapshots of versions. To distinguish the snapshots of stores and versions from the snapshots of objects, the latter are called *shadows* in Squot. This follows the tradition of ENVY/Developer which calls non-resident (i.e., not loaded) editions shadows [51, pp. 237 ff.]. They are implemented differently in Squot, however. Shadows in ENVY inherit from their live classes (e.g., EmShadowClass is a subclass of Class), which is not the case for any of the current classes of shadows in Squot. It is not forbidden to have shadow classes inherit from their corresponding live classes either—they only must make sure that multiple shadows of the same object can coexist without issues. The deserialization of shadows from storage objects should therefore be free of side effects to the programming environment (e.g., methods should not be compiled when shadows of them are read in).

Each SquotSnapshot has a dictionary of SquotArtifacts.[3] An artifact is the combination of an object graph, its key (called path as described above), and the associated metadata. The name *artifact* was chosen before it was decided that an artifact should always contain a graph of objects (instead of a single object, for example). For the same reason, the graph of an artifact is accessed with the contents message. Apart from the structural definition above, the true definition of *artifact* in Squot is: an artifact is an element of an object container. All classes that implement the protocol of object containers allow querying instances for their artifacts, which answers a dictionary of objects that understand the protocol of SquotArtifact. The keys in these dictionaries are the paths of the artifacts.

Since image stores are also object containers, they can also produce a variant of artifacts that are called SquotLiveArtifact. They act like adapters on the live objects in the image store, holding the image store metadata and the live start object. When the contents of a SquotLiveArtifact are accessed, it will capture the graph starting from the live object and answer a graph of shadows.

Graphs of shadows are instances of SquotShadowGraph, and they associate object names with shadows. In contrast, SquotObjectGraphs additionally associate names with live objects and should therefore not be stored in versions.

Because there is no static type checking in Smalltalk, classes need not declare that they implement a particular interface or protocol. Instead, Squot defines traits [15] for behavior that is common for multiple classes. This allows grouping of related messages and capabilities without introducing mandatory abstract superclasses.

[3] The naming is somewhat debatable if we look at the definition of *artifact* in the Oxford Advanced Learner's dictionary: "(technical) an object that is made by a person, especially sth of historical or cultural interest" [26, p. 72]. In a way, Squeak really deals with "objects that were created by a person", namely the Smalltalk objects which were created by users during their exploratory programming sessions. And since version control deals with the history of objects, there is also "historical interest", although the meaning deviates somewhat from the intention of the Oxford dictionary definition.

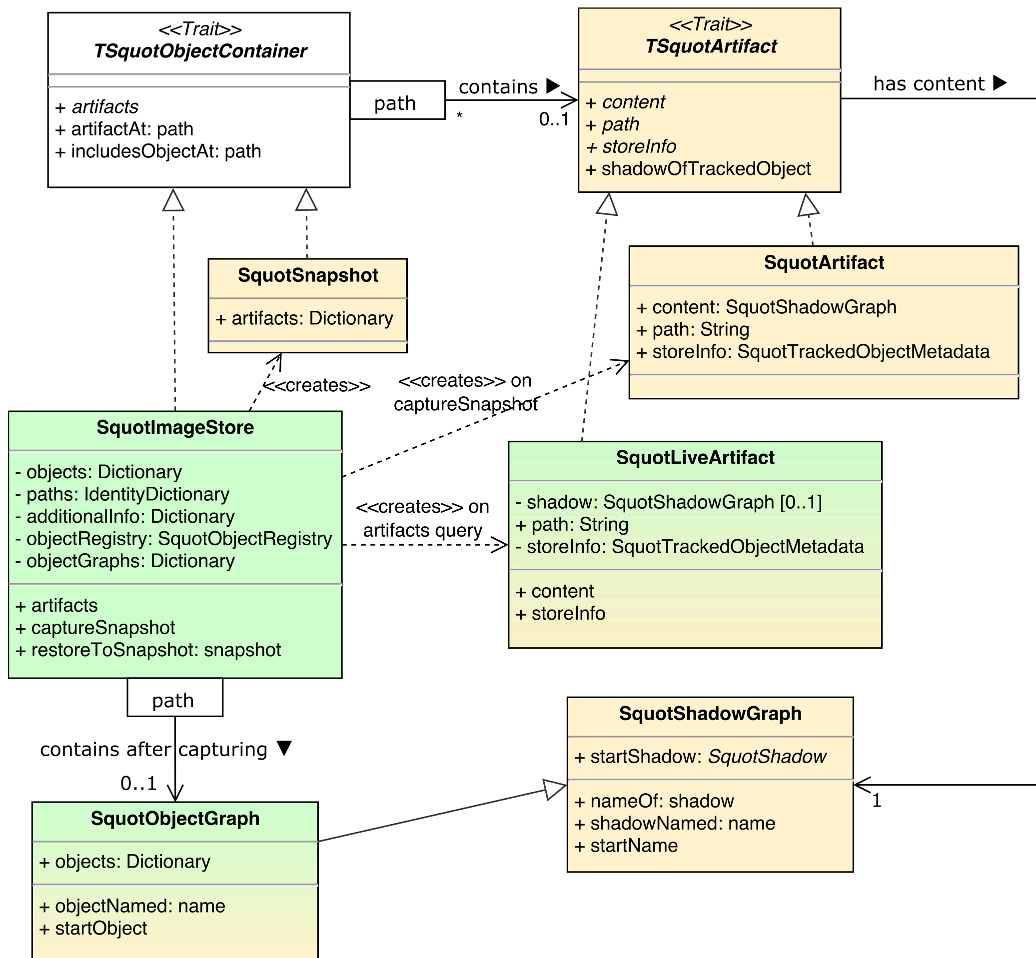

Figure 4.1: Relationships among object containers, SquotSnapshot, SquotImageStoree, artifacts, and graphs of shadow objects (snapshot objects). The abstract methods in the traits are required by the trait.

Naturally, some of these traits correspond with the types that are proposed in chapter 3, such as object containers.

4.1.2 Generic object snapshots

Following section 3.7, generic snapshots for all kinds of objects, implemented in SquotObjectShadow, contain a collection of slots, the class of the original live object that should have been captured, and the class of the object that was actually captured (in case the live object wanted to be replaced).

Two types of slots must be considered for Smalltalk objects:

1. instance variables
2. indexable variables (for variably-sized classes, such as Array)

Since instance variable names cannot be integers, both kinds of slots can be stored together in one dictionary. The classes of slots have an abstract superclass SquotShadowSlot, which is a subclass of Association (a key-value pair). Instance variables are represented with SquotInstVarSlot and indexable variables with SquotVariablePartSlot. The key of a SquotInstVarSlot is the instance variable name. For a SquotVariablePartSlot, the key is the index. In both classes, the value of the slot is the shadow of the object to which the respective variable is bound. The lookup of a slot in an object shadow sends instVarSlotNamed: or variablePartSlotWithIndex:, respectively, to a SquotObjectShadow that will perform the lookup in its slots dictionary.

Some objects must be captured differently because of their special identity requirements: the primitive types SmallInteger and, since the advent of the Spur memory manager[4]with Squeak 5.0, Character and SmallFloat64. They could each be captured as an "empty" SquotObjectShadow, but we would still need to know which shadow is which integer, for example. For ordinary objects, this is solved by assigning names to them. However, we want to avoid names for primitive objects because they are value objects and should not need an additional name. Thus, a different class, SquotPrimitiveValue, is used for the shadows of primitive objects (also for true, false and nil). It is a simple wrapper around the primitive object. This means a shadow object will reference a live object, but in this case this is fine because the primitive objects have no side effects, cannot change their identity, and are already present in every Smalltalk system.

While this is already sufficient to capture and materialize nearly[5] every object in Squeak/Smalltalk, some types should be handled differently. There are collections whose indexable variables all contain the same primitive type (e.g., ByteArray, WordArray, ByteString, and WideString). Creating one SquotPrimitiveValue wrapper for each slot would be a huge waste of memory, so there is another shadow class SquotBitsObjectShadow that simply wraps a copy of such collections.

The reason why not all objects can be captured by simply making a copy of them is that snapshots would be indistinguishable from live objects by the meta-object protocol. For example, the snapshot copies of domain objects would be included in queries such as allInstances to their classes. Applications that leverage the meta-object protocol to perform operations on live domain objects would also modify the snapshots,

[4]http://www.mirandabanda.org/cogblog/2013/09/05/a-spur-gear-for-cog/ (last accessed November 5, 2017).

[5]Another exception is CompiledCode (the superclass of CompiledMethod), which has indexable byte variables (the bytecode), but the first indices are not accessible like indexable variables because they actually contain pointers (the literals). There is an extra shadow class SquotCompiledCodeShadow for them that currently implements proper capturing, but not materialization.

which contradicts the purpose of snapshots to preserve state. Primitive objects and bits objects are defined by the Squeak standard library, are domain independent, and should not be subject to modification after having been retrieved by an application via the meta-object protocol (because it would not only affect objects of the application, but other parts of the system as well). If application developers consciously decide to introduce new domain specific bits types (with variableByteSubclass:, variableWordSubclass:, etc.), rather than using the ones that ship with Squeak, then these developers should also be entrusted with properly overriding the capturing and materialization behavior for this type.

4.1.3 Capturing

All trackable live objects must understand the message captureWithSquot and answer an appropriate snapshot. The default implementation added to Object (which is the base class of nearly all objects) starts a traversal of the live object graph, which is implemented in SquotObjectCapturer. Immediate objects (SmallInteger et al.) are just wrapped with SquotPrimitiveValue, no traversal is started from them.

The capturer performs a breadth-first search starting from the start object of the live graph. In the course of this, it builds a new SquotObjectGraph. To affirm that object graphs should be considered immutable after capturing, the mutator methods used to build a graph are only defined in a subclass named SquotMutableObjectGraph, which is used by the capturer. When new objects are encountered, the capturer assigns a name to them. Names are both registered in the object graph and in the object registry supplied by the capturing image store. Additionally, the captured object and the snapshot are decorated[6] with a SquotNameDecorator. This helps to see the names easily in the debugger, but the decoration is lost in a double dispatch (because when the decorated object supplies self as a message argument or return value, the decorator is no longer applied). Therefore, the definite source for existing object names is the object registry.

When the capturer encounters an object, it sends captureWithSquot: to the object, with the capturer as argument. This message should be understood by all trackable objects in addition to the parameterless captureWithSquot. The answer, which should be the created shadow, is added to the object graph under construction. The default implementation of captureWithSquot: in Object can be seen in Listing 4.2 below: The response to squotShadowFactory should be the preferred shadow class of the object. Most objects will answer SquotObjectShadow, but some differ (see subsubsection 4.1.8.1).

Remembering subsection 2.3.5, globally accessible objects should be able to replace themselves with symbolic references for capturing. They can do so by returning a replacement object from the squotReplacement: method. The default implementation is to delegate to another method objectForDataStream:, which is used in Squeak for the DataStream class. DataStreams (in practice its subclasses) are used to save ob-

[6] cf. decorator pattern [16, pp. 175 ff.]

Listing 4.2: Default implementation Object»captureWithSquot:

```
captureWithSquot: aCapturer
    | replacementOrSelf |
    self class isImmediateClass ifTrue:
        [↑ aCapturer capturePrimitiveValue: self].
    self class isBits ifTrue: [↑ aCapturer captureBits: self].
    replacementOrSelf := self squotReplacement: aCapturer.
    ↑ aCapturer
        capture: replacementOrSelf
        as: replacementOrSelf squotShadowFactory
```

jects to the disk [10]. Objects that want a symbolic reference must create a DiskProxy instance with the necessary information to find the object in the target environment, send replace:with: to the argument of objectForDataStream: (which usually is a DataStream), and return the DiskProxy from objectFromDataStream:. Since Squot needs the same mechanism for object capturing, it reuses this message and lets the capturer mimic the necessary part of DataStream's protocol. The Fuel object serializer defines a similar message fuelReplacement for the same purpose.

In capture:as: the capturer will 1) instantiate the new shadow object and 2) tell it to initialize itself based on the live object. Separating these two steps is one way to deal with cycles in the object graph: the uninitialized shadow can already be associated with the live object in a dictionary, so on a recursive encounter with the same object, the (possibly not yet fully initialized) shadow can be used already. For SquotObjectShadow the initialization means that the live object must enumerate all its relevant instance variables (which can be controlled by overriding the class-side method squotRelevantInstVars) and the indexable variables and adding them to the shadow. This will create the appropriate slots and tell the capturer to capture the slot's value. If a slot's value must be replaced by another value according to the object metadata supplied from the image store, the replacement will happen at this point. If the slot value has already been captured in a previous step, the resulting shadow will be assigned to the slot. Otherwise, the live value is enqueued to the search frontier in the capturer, but it is not captured yet (doing so would result in a depth-first search); and the slot is remembered, so the correct shadow can be filled in later. The name-decorated live object is used as the slot's value in the meantime. Deferring the final filling-in of references is a second way to deal with cycles in the object graph because it means that a shadow can be initialized without running into an endless recursion when a live object in a cycle is captured. Instead, the following of references can be left to the search algorithm, which provides proper multiple-path pruning.

After the object graph traversal has come to an end, the remembered slots whose values were not yet filled with the correct shadows are finally redirected. The shadows are taken from the aforementioned dictionary that associates live objects with their shadows. The same dictionary is used to capture each object only once: when an object is already present as a key in the dictionary, it will not be captured again.

In the current implementation, the names assigned to objects are arrays that contain a UUID and slot identifiers on the path via which an object was reached from the start object. Because breadth-first search always reaches a node via the shortest path from the search origin, it was preferred over depth-first search for capturing. The path component of the name is not necessary to distinguish objects, but it makes debugging easier (e.g., in the unit test cases) because UUIDs alone are hard to remember for most humans.[7] If an object moves in the graph or is replaced where the path in the name leads, the path will become outdated, of course. But since it is only a nice-to-have and the UUID on its own should suffice to uniquely identify an object, this does not impede the functionality. The path part of the name is created in the capturer by remembering the slot from which an object was first encountered. The slots understand a message to provide a readable name for this purpose (usually it is simply the slot key, i.e., the variable name or index). The referrer-slot information is stored in dictionaries next to the object names. Objects are also decorated with their referrer with a SquotCreatorDecorator.

Shadow objects always refer to another directly via pointers, an indirection to only refer via the object name has not been implemented so far, as the need for it did not arise yet. The name indirection would make it possible to create partial object graphs that do not contain shadows for all objects in the graph.

4.1.4 Materialization

The snapshot traversal is, like the traversal for capturing, implemented in a separate class, called SquotShadowMaterializer. A recursive depth-first search was chosen here because it was more straight-forward to implement. When the materializer is created, a SquotObjectRegistry, a target SquotObjectGraph, and a source SquotShadowGraph can be supplied. The source graph is used to retrieve object names, so they can be added to the registry and to the target graph if an object does not yet exist there. If no target graph is supplied, the names are only kept in the object registry.

To materialize a shadow graph, rematerialize: is sent with the start object shadow to the materializer. If a shadow subsequently wants to have another shadow rematerialized (as the search progresses), it will also send rematerialize:. In this method of the materializer, it is checked that each shadow is materialized only once.

When a shadow is to be materialized and it has an object name, it will be looked up in the object registry to find an existing live object with the same name. If one is found, the materializer will attempt to "overwrite" the live object in-place (the shadow is materialized *in* the existing live object). How this is accomplished is determined by the shadow, as can be seen in Listing 4.3. If the in-place materialization is not possible (e.g., because the object would have to change its size, which is impossible in Squeak) the shadow should materialize itself into a new object instead and the old one will be replaced with the new one via becomeForward:. If no object with the name exists

[7] Projects like https://github.com/jamesmunns/human-hash-rs (last accessed November 5, 2017) seem to support this claim.

yet, according to the registry, it is at the shadow's discretion to materialize itself into a new object. Primitive objects do not have names, they will simply be unwrapped from their SquotPrimitiveValue.

Listing 4.3: Materialization dispatch in SquotShadowMaterializer

```
findOrCreateAndRematerialize: aShadow
    | existing |
    self sourceNameOf: aShadow ifPresent: [:name |
        existing := objectRegistry objectNamed: name ifAbsent: [].
        (existing notNil and: [existing squotShouldKeepIdentity])
            ifTrue: [↑ self materialize: aShadow in: existing]].
    ↑ aShadow squotMaterializeWith: self

materialize: aShadow in: anObject
    | materializedObject |
    convertedObjects at: aShadow undecorated put: anObject.
    (anObject shouldHaveNameAssignedBySquot and: [sourceGraph notNil])
        ifTrue: [objectRegistry nameOf: anObject
                    ifAbsentAssign: [sourceGraph nameOf: aShadow]].
    materializedObject :=
        (aShadow materializeAs: anObject with: self)
            squotReactivateWith: self.
    anObject becomeForward: materializedObject copyHash: false.
    ↑ anObject
```

SquotObjectShadows tell each of their slots to materialize themselves in the given live object, with the materializer. The slots, in turn, instruct the materializer to rematerialize their values, then modify the instance variables or indexable variables of the live object accordingly.

Again, object graphs with cycles are cared for by separating the creation (allocation) of a new object from initializing (materializing into) it. The created (or existing) object will be registered before the materialization proceeds (see Listing 4.3 and Listing 4.4).

Listing 4.4: The allocation of new objects by SquotObjectShadows is separated from filling in the slot values (cf. Listing 4.3). This makes it possible to materialize object graphs with cycles.

```
squotMaterializeWith: aMaterializer
    | object |
    object := self createMaterializedInstance.
    ↑ aMaterializer materialize: self in: object
```

4.1.5 Providing context information

As already indicated in chapter 3, many operations need access to context information such as object metadata or the names of objects. It is not always obvious which operation will need to have access to which information: the tracked objects and their snapshots can influence these operations and one object might need other information than another.

There is more than one way to provide context information in Squeak. The obvious one is to pass everything that is needed in message arguments, which, however, clutters the message signatures. This approach was taken to pass around the capturer and materializer, for example. Since these are already available as an argument, a different way to make further context accessible would be to use the capturer and materializer as facades to access the other necessary information. This in turn would clutter the interfaces of these classes with messages that are only marginally relevant to the rest of their implementation (low cohesion).

A third way are dynamic variables. The concept is well defined in Common Lisp [52, 5.3 Macro defparameter, defvar], which allows the value-binding of variables declared as "special" to be controlled by the dynamic execution context, rather than the lexical environment [52, 3.8 Declaration special]. A simplified explanation is that dynamic variables are like implicit additional arguments to all functions or methods invoked in the scope of the dynamic binding. The object metadata and the image store's object registry are made available via dynamic variables in Squot (as subclasses of DynamicVariable). The object registry is further given directly to capturers and materializers.

4.1.6 Differences

On page 34 three levels of differences are introduced: object container level, object graph level, object level. The corresponding classes of differences in Squot are:

1. SquotPatch—at the object container level
2. SquotObjectGraphDiff—at the object graph level
3. classes that can understand the protocol of the trait TSquotDiff (SquotObjectDiff, for example)—at the object level

Because of the additional SquotArtifact in Squot's compositional hierarchy of snapshots, there is also a SquotArtifactDiff. A SquotPatch[8] contains one SquotArtifactDiff for each changed artifact. Each SquotArtifactDiff then wraps one SquotObjectGraphDiff, which maps object names to their differences. For an overview see Figure 4.2.

[8]The names might not be ideal because the distinction of "patches" from "diffs" is not obvious before reading their documentation or reading how they are used in the code. But the names SquotSnapshot and SquotPatch are at least reminiscent of the Monticello classes MCSnapshot and MCPatch, which serve similar roles: versions have a snapshot and patches contain the differences between two such snapshots.

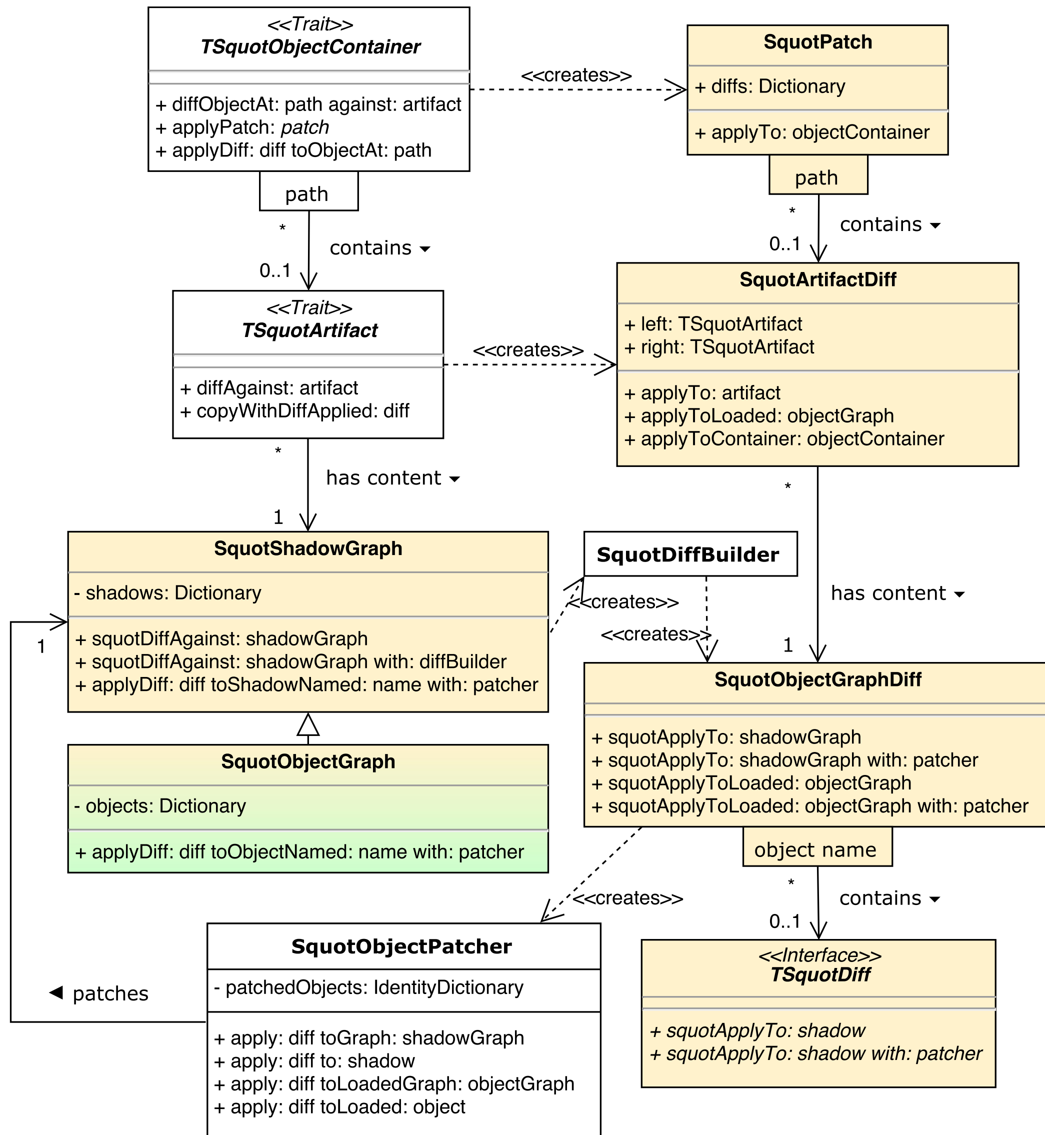

Figure 4.2: Relationships among object containers, artifacts, object graphs, and their respective difference classes

The graph traversal for the computation of differences is conducted by instances of SquotDiffBuilder. While traversing two or three graphs simultaneously, it constructs a SquotObjectGraphDiff. In each step, a left-side shadow is compared with a right-side shadow, and optionally a base shadow. Their object names are compared as described in subsection 3.5.2 to decide how the difference detection should proceed. It currently relies on the presence of name decorators on the shadows and remembers names in an own dictionary before performing double-dispatches with shadows. There is one specialty for the treatment of the start object shadows: when their names differ, a SquotObjectReplacement will be generated as the difference, signifying that a different object is now used to span the graph. For all other objects encountered via references, different names will lead to differences in the referrer, not in the referents.

At several places the resulting differences are tested with the message squotHasChanges, to which the response should be false if no differences were found. SquotDiffOfEqualObjects serves as a null-object for many kinds of differences. It always denies to have changes. If no changes are detected for an object, there is no need to record anything about it in the differences between two object graphs (except for some information described later in this section).

Cycles in the compared graphs are dealt with by splitting the creation of difference objects from their initialization (see Listing 4.5).

Listing 4.5: The allocation of new difference objects for SquotDiffBuilder is separated from actually computing the differences.

```
diffFrom: leftShadow to: rightShadow
ifNew: newBlock andInitialize: initBlock
    | diff left |
    left := leftShadow undecorated.
    diff := objectDiffs at: left
        ifPresent: [:existing | ↑ existing]
        ifAbsentPut: newBlock.
    initBlock cull: diff cull: leftShadow cull: rightShadow.
    diff squotHasChanges ifTrue:
        [graphDiff
            addDiff: diff
            forObjectNamed: (shadowNames at: left ifAbsent: nil).
        self noteChanged: left].
    ↑ diff
```

Notably, the multiple-path pruning in this traversal works by only checking that the left-side shadow has not been encountered yet. The rationale is that the changes to a single object do not depend on the path via which the object was reached in the graph. The object must have changed in the same way wherever it is referenced. Anything else would rather mean that a former reference to this object was in fact changed to another object (with another name).

Analogous to capturing, the comparison of slot-based snapshots works by delegating the comparison to the contained slots. The results are saved in an instance of SquotObjectDiff, which keeps differences for all changed slots for one object. If the names of the values of two compared slots are different or one of the values does not have a name, a SquotSlotReassignment will be recorded. It remembers both the left-side and the right-side value. In effect, user-defined strategies to deal with unnamed or differently-named objects as proposed on page 36 are currently not implemented. Making changes to the referrer is always preferred to making changes to the referent. This avoids the mutation of value objects, but it might not always be what the user wants. If the names of the compared slot values are equal, no difference is recorded for the compared slots, but the referents are added to the search queue of the SquotDiffBuilder for further comparison. At this level of abstraction, slots cannot be added or removed from a Squeak object (because once allocated, an object cannot change its size on the heap). So the only type of slot difference up to this point are reassignments.

Applying differences to shadows is initiated by sending squotApplyTo: to a difference object with the target shadow as argument. The graph traversal needed to patch object graphs is coordinated by a SquotObjectPatcher. It eventually sends squotApplyTo:with: to each difference object (with the target as first argument and with the patcher as second argument) and remembers the answer (which should be the modified shadow) for each shadow to which a difference has been applied. Shadows should be considered immutable (after having been constructed), so they will usually copy themselves and apply the changes to the copy, which is then returned.

The messages used to apply differences to snapshots, stores, artifacts, and object graphs can be seen in Figure 4.2.

SquotObjectDiffs simply apply all their slot differences to the target. On a SquotSlotReassignment, the slot's value is replaced by the right-side shadow that is stored in the reassignment object.

When differences are applied to a SquotImageStore, the live objects must be patched, of course. Currently, difference classes implement a method with the prefix applyToLoaded to apply changes directly to live objects. For SquotObjectDiffs, this will apply the differences to a shadow first and then materialize the patched shadow in the live object. The patcher assumes the role of the materializer in the arguments. When an object does not exist in the graph yet, or when it does not have a name, then it will be fully materialized with a SquotShadowMaterializer. This is only a stub implementation, however, because it does not correctly handle the case when a new object must be added to the graph. Neither are the differences beyond added objects detected, as demanded on page 37, nor is the materialization of new objects "shallow" as described on page 38. Instead, the whole subgraph reachable from the added object will be materialized. This stub implementation works when the added objects are simple nameless objects that do not transitively refer to any named objects (e.g., it works for Points). The name-object mappings for both shadows and live objects beyond the added one will not be correct. At least, the graphs generated by applying such differences are structurally correct, so, Squot's bookkeeping aside, the generated objects should work.

It is also possible that the object metadata is different from one snapshot to another. Therefore, differences should be computed and applied for the metadata as well. But since changes to the metadata have been rare so far, this has not been thoroughly implemented yet. Instead, the metadata entries from the right side are simply added to the metadata from the left side, possibly overwriting entries.

4.1.6.1 Deep changes

Let us assume a merge situation where in the base version there is a form that contains a button. In the left-side form, the button's label has been changed, whereas in the right-side form, the button has been deleted. The situation should be presented to the user with a conflict in the form for the button because with an automatic solution either the label change or the deletion would be lost. To make this feasible, it must be noted in the differences to the form between its base snapshot and its left-side snapshot (where the label of the button changed) that *something* has changed in the button slot. Otherwise, when the differences to the form are inspected and there is no change when going to the left side (after all, the button was changed, not really the form itself), it might be decided that the deletion of the button can safely be performed, since the change does not conflict with any other change to the form. The crux is that changes to objects that are *part of* a parent object can actually be significant to the parent object as well.

Squot notes this by adding SquotSlotTransientChanges to the slot changes in SquotObjectDiffs for the parent objects. To do that, whenever the differences builder encounters the values of slots, it remembers the referrers of these values. When differences to an object are found, the difference builder walks back the chain of referrers and adds the transient changes for the slots. When these differences are applied to objects, they do nothing, but they can be used in three-way differences to detect conflicts. They also make difference objects navigable like shadow objects, so all differences in a graph can be reached from the difference object for the start object of the graph. This is helpful for inspecting the differences with the Smalltalk toolset and it could also help when visualizations of the differences must be built.

An alternate implementation approach would be to look for the deep changes only when three-way differences are actually computed for a merge. Some kind of reverse search through the object graph would be needed to find the transitive referrers to objects that have actual changes. Navigating two-way differences like a graph is not possible in this variant, but it could be reproduced by computing the transient changes belatedly like for merges.

4.1.6.2 Addition and removal of object graphs

In the beginning it seemed to be a good idea for certain operations to describe the addition of an artifact or object, or the removal of one, as the differences between *nothing* and the present object. One example application is the displaying of differences to users. Squot has SquotAbsentArtifact and SquotAbsentValue (the "Absent types") as null-objects to represent *nothing* for such purposes. However, these objects lack knowledge about the type of the absent thing and, hence, cannot know what "nothing" really means for the purpose of comparison. This was compensated by

double-dispatching the absent object with the other, present object, such that the type can be determined from the present object. To have further control over the handling of absent objects, customization hooks have been introduced such as a constructor absentPrototypeForSquot in Object which must be overridden if an empty instance is not the same as an uninitialized instance, or if it is at all ill-defined what an empty or uninitialized instance of a class would be (Boolean and Number answer SquotAbsentValue here). Ultimately, the Absent types only deferred the problem of dealing with additions and removals from the place of their detection to other places in the code; it turned out simpler in most situations to directly handle additions and removals specially instead of hiding them in specially constructed regular differences. Thus, SquotArtifactAddition and SquotArtifactRemoval were introduced later as kinds of SquotArtifactDiff.

4.1.7 Serialization and deserialization

The most common target of export operations is the local file system. For each artifact, one or more files and directories should be created. Squot includes the SquotFileSystemStore to operate on a file system directory that contains serialized artifacts. It stores object metadata next to the files of an artifact in a dot-file that ends with *.squot-contents*. Additionally, a *table of contents* named *.squot* is maintained in the store's directory, so that it needs not search the directory tree for the relevant files. Both metadata and the table of contents are stored in Smalltalk Object Notation (STON) [70].

To use Squot even on existing Smalltalk projects that do not have any of the metadata files, the file store will perform a search for files named *.filetree* to find at least packages stored with Squeak's *FileTree*[9]implementation of the Cypress Smalltalk code interchange format[10]. In order to contribute to such projects using Squot, but without forcing the Squot metadata files on anybody else, a file store can be told to not store the metadata.

The directory on which a file store operates can also contain files that are not relevant to Squot, such as a readme text or configuration files used by other services like *smalltalkCI* [49]. The file store currently ignores any files and directories not related to artifacts according to the table of contents, but it generates a single SquotUnrecordedFilesArtifact as a representative for all of them.

When a file store should write an artifact, it will look up the serializer class to be used in the artifact's metadata. A serializer object is then created to write in the directory of the file store and the artifact is given to it for writing. Because the artifact paths (the keys of object graphs) are file or directory paths already, the serializer will create its output at this location. Afterwards, the metadata will be stored (unless the file store was configured not to do so). When all artifacts have been written out, the table of contents is created. It contains the paths of all stored artifacts.

[9] https://github.com/dalehenrich/filetree (last accessed November 5, 2017).
[10] https://github.com/CampSmalltalk/Cypress (last accessed November 5, 2017).

When the file store is asked to enumerate its artifacts, it will read the table of contents and the metadata for each artifact. There it finds the information about each artifact's serializer. To find a compatible deserializer, the file store class has a registry of deserializer classes and each of these responds to a message canReadObjectSerializedBy:. Deserializer classes are responsible to register themselves in the registry when they are installed. The first deserializer class that responds to be able to read the output of the given serializer is selected to read the artifact. An instance of the detected deserializer is created to read in the object graph from the files.

When a file store is asked for all of its artifacts, the deserialization is done lazily, in case not all object graphs in the file store will actually be needed (but only the paths and metadata, for example). To support this, there is SquotLazyArtifact, which will invoke the conversion to shadows only when the getter method for the object graph is invoked. That method is memoized, so the deserialization will not need to be invoked again when the graph of the lazy artifact is accessed multiple times. Because it is sometimes necessary to extract the shadows early, all artifacts understand a capture message that does nothing for regular artifacts, but invokes the conversion immediately for lazy artifacts.

Because the STON serializer will not differentiate between snapshot objects and live objects, it is especially important that the metadata does not reference any live objects anymore when the artifacts are being serialized. That is why references to live objects in the metadata are replaced with object names when the artifacts of an image store are captured.

4.1.7.1 General purpose serializers and deserializers

Since Squot provides general purpose shadow objects, it also provides general purpose serializers for them. These serializers can also be used for other shadow types, since they can serialize and deserialize any object.

Squeak's own facility to store objects to files and restore them later is the SmartRefStream. It traverses object graphs, serializes each object only once and for each type of object it encounters, it also stores the class layout and the names of the instance variables. When the schema of a class changes, this can be detected and the stream will try to migrate instances. The SquotSmartRefStreamSerializer and the corresponding -Deserializer are simple adapters around SmartRefStream. The serializer will create a file at the artifact's path, create a SmartRefStream on the file and put the SquotShadowGraph of the artifact into the stream. The deserializer implementation is equally simple: a SmartRefStream is created on the file at the artifact's path and the first object is extracted from it, which should be the SquotShadowGraph serialized earlier.

Another serialization implementation on which Squot already depends for the metadata, and which produces more human-readable files than SmartRefStream, is the serializer and deserializer for STON. SquotStonFileCodec is serializer and deserializer both in one class,[11] since only two message sends to the STON class must be

[11] Ideally, there would be similar portmanteau for serializer-deserializer. Codec is a portmanteau of coder-decoder.

adapted. As in the SquotSmartRefStreamSerializer, the SquotShadowGraph from the artifact is converted as is and written to a file at the artifact's path.

4.1.8 Example specializations for some kinds of objects

The aim of the design implemented in Squot is to be flexible about the formats of snapshots and differences, so that domain specific solutions can be implemented when desired. In this subsection, we present some examples of specializations that deviate from the standard procedure of capturing, serialization, or materialization.

4.1.8.1 Hashed collections

While the slot-based shadow model is indeed sufficient for nearly all objects in Squeak, it does not always produce the most useful results. For example, collections are not treated specially at all, as far as their capturing has been described so far. This means that their structural implementation details are exposed in the shadow graphs and that additions and removals to variable-length collections are not detected as such. For HashedCollection, which is the superclass for Set and Dictionary in Squeak, this is particularly troublesome.

HashedCollection implements a hash table for constant-time element lookup, addition and removal. It uses an internal array as the hash table and an integer tally to remember the size of the collection. These could be captured separately as described so far, but the differences computed for these snapshots would be confusing and inappropriate for the task: elements could be moved in the hash table (because the identity hashes of objects with the same name need not be equal in two different Squeak images), or the hash table could be replaced by a new one (because it had to be grown), which would be visible when comparing regular object shadows, but it would not constitute a relevant change to the collection. The movement of retained elements in the hash table can also obfuscate actual additions and removals of other elements.

Therefore, a specialized shadow class SquotHashedCollectionShadow (which subclasses SquotObjectShadow) is used for hashed collections. It keeps a separate collection for the elements of the tracked collection, each captured in a SquotHashedCollectionSlot (whose key is irrelevant). The lookup of such slots is realized by searching the items of a SquotHashedCollectionShadow. This has different performance characteristics than the live hashed collections, but the snapshots should not be handled like live collections anyway. They serve a different purpose, after all.

When two such shadows of hashed collections are compared, the elements of one are looked up in the other to detect additions and removals. These are recorded as SquotSlotAddition and SquotSlotRemoval with the elements' slots. Additions and removals are suddenly possible because the shadows for hashed collections treat these collections as proper collections, which is a higher level of abstraction than before. On this more abstract level, collections *can* actually change their size.

Dictionaries receive another slightly different treatment. A Squeak Dictionary is made up of Association objects. Two dictionaries can share a single association. This means that if the value of the association is changed in one of the dictionaries, it

will also change in the other dictionary. This is used, for example, in the dictionaries for variable bindings, most prominently global variables in the Smalltalk pseudo-dictionary.[12] While this might not be relevant to most dictionaries used in domain objects, the feature is generally available and must be reflected in the shadows. Therefore, the elements of a dictionary shadow are the shadows of the contained associations. These in turn carry the shadows for their keys and for their values.

4.1.8.2 Packages (Smalltalk code)

For some domain objects or infrastructure of the programming environment, objects for snapshots and differences already exist. Existing version control systems for Smalltalk code, such as Monticello in Squeak, come with a set of modeling types to capture the definitions of classes and methods, and possibly more. For Monticello, the relevant classes are MCSnapshot, MCPatch, MCDefinition, subclasses of the latter, and MCPatchOperation and its subclasses. Monticello also provides a class to compile the definitions captured in its snapshots, called MCPackageLoader. Further, Monticello has several types of repositories that define a storage format for the snapshot types. To not duplicate the implementation effort that has been invested in such models already, the architecture presented in this report supports the reuse of such objects, provided that their interfaces can be adapted suitably.

The traditional representation of a package in Squeak is a PackageInfo object. It has a name and is usually registered in a PackageOrganizer.[13] Each PackageInfo can be queried for the contained Behaviors (which includes classes) and all the methods that belong to the package. Which belongs and which does not is determined by a naming convention for the class categories and method protocols [50]. Hence, it was decided to track Smalltalk source code by adding a PackageInfo object to an image store. So, a PackageInfo is the live object from which Smalltalk code is captured, as a special case of object capturing.

To capture a package, there is SquotPackageShadow. It wraps an MCSnapshot and the name of the package. The snapshot is obtained from the MCPackage with the same name (which uses the PackageInfo to collect all the behaviors, methods and scripts belonging to the package and turns them into MCDefinitions). When two SquotPackageShadows are compared, a SquotPackageDiff is created, which wraps an MCPatch. During merges, the latter is substituted for a SquotMonticelloMergeDiff because an MCPatch cannot contain conflicts. Applying package diffs to package shadows applies the wrapped Monticello patches to the wrapped Monticello snapshots. Materializing a package shadow or applying the diffs to loaded packages uses the MCPackageLoader to perform the operation.

This implementation makes use of the permission in the architecture that live objects may capture their relatives independently of the main object graph traversal, as described on page 31. The relationship of packages, classes, and methods is not

[12] Smalltalk used to delegate to a SystemDictionary and since Squeak 4.5 it delegates to an Environment, which in turn has a dictionary for its bindings.

[13] There is a default package organizer singleton, but each Environment also has its own organizer.

established via object references, but via a naming convention in the globally accessible system organization and the organization of methods within classes. The other reason to go down this path is, as already mentioned, the reuse of existing capturing code in Monticello.

For serialization, the Cypress/FileTree format already implemented for Monticello is used. This ensures compatibility with the existing Smalltalk projects that have used FileTree to share the code via Git on GitHub. The features of MCFileTreeWriter and MCFileTreeStCypressWriter are used for serialization in SquotCypressCodeSerializer and those of MCFileTreeStCypressReader are used for deserialization in SquotCypressCodeDeserializer. Because of the way the connection to Git is made, which is described later in this chapter, the FileSystem API [3] must be used instead of Squeak's FileDirectory API. Normally, FileSystem would only be used when Monticello is executed on Pharo/Smalltalk (instead of Squeak), so the readers and writers have been subclassed to make the necessary adjustments to use FileSystem also on Squeak. At a later time, a cleaner reimplementation of the de-/serialization to the same target format could be approached.

4.1.8.3 Texts in text files

While the two previous examples of specializations both introduced new shadow types, this time we will only customize the serialization format. Most software repositories will contain a readme file and it might be desirable to edit it from within Squeak. Squeak can load text files into workspaces and save them back, but this is decoupled from version control, of course. Moreover, a working copy of the readme would be needed in the file system (instead of only in Squeak's memory) to interact with it. It would be more consistent if the text file, or rather its textual contents, could be tracked as objects, without compromising on the storage format being plain text.

To convert between String objects and plain text files, there is SquotTextFileCodec, which is the squotDefaultSerializer for SquotBitsObjectShadow if it contains a String. The codec assumes that the object graph of the artifact that should be serialized contains only the start object, which is the character string. Hence, writes out only the start object. Accordingly, it creates a new object graph with a single bits object shadow on deserialization.

4.2 Git connectivity

In this section we describe the missing parts to bring objects from Squeak to Git repositories, for example on GitHub.

4.2.1 FileSystem-Git

FileSystem-Git is originally a project for Pharo/Smalltalk that combines two facilities:

1. an implementation of the Git object model, repository layout and auxiliary structures (such as pack files) that allows inspecting and manipulating of Git repositories in a file system
2. an adaptation for the FileSystem API to provide virtual file systems on the trees and blobs of Git commits

While the former enables the low-level interaction with a Git repository (that usually is the local *.git* directory of a clone), the latter provides the possibility to browse and modify Git trees and blobs with anything that uses the FileSystem API; such clients do not even have to know that they are browsing the contents of a Git commit. Checking out a working copy outside of the Smalltalk environment is not necessary. Moreover, new commits can be created from a Git file system after its files or directories have been modified. In traditional Git terms, the low-level implementation can be thought of as the "plumbing". The FileSystem layer on top provides some kind of "porcelain". Though, it does not provide all the features of the tools that come with the official Git distribution.[14]

The implementation of the Smalltalk plumbing was originally started as *Git for Squeak* [17]. Later it has been extended and used in *GitFS* [38], which already added the connection to the FileSystem API, before it was included under the name *FSGit* in the original repository of the FileSystem package.[15]Since then, it was separated from FileSystem again, as the latter was adopted as the standard file access library in Pharo [3], renamed to *FileSystem-Git* and moved to SmalltalkHub.[16]

In the meantime, the library has been abandoned as the provider of Git connectivity in Pharo. It has been superseded by a binding to the library *libgit2*[17] [37] (which is implemented in C) for maintenance and performance considerations [36]. Nevertheless, we have chosen to pick up the pure-Smalltalk implementation of Git and backport it to Squeak because the bindings to *libgit2* use a variant of the foreign function interface that is currently not available in Squeak. Because of the differences between the FileSystem API in Pharo and the version of it that is available for Squeak (e.g., classes have been renamed), some changes have been made to the FileSystem library for Squeak as well. All changes to FileSystem and FileSystem-Git are currently hosted together with Squot in its repository on GitHub.

The implementation of FileSystem-Git was already quite complete when it was picked it up for Squot. But besides the porting to Squeak, support for some Git features had to be added, some bugs had to be fixed, and optimizations have been implemented (which are mostly about handling large blobs and pack files more efficiently). Some notable contributions are:

[14] http://git-scm.org (last accessed November 5, 2017).
[15] http://www.squeaksource.com/fs.html (last accessed November 5, 2017).
[16] http://smalltalkhub.com/#!/~FileSystemGitDev/FileSystem-Git (last accessed November 5, 2017).
[17] https://libgit2.github.io (last accessed November 5, 2017).

- support to push Git objects to a remote repository
- computation of recursive tree differences (i.e., generating a hierarchy of changed trees and blobs to focus on objects that changed between two commits)
- implementation of a merge-base finding algorithm
- tolerance for tree entry modes other than file and directory, such as executable file or submodule (no implementation to deal with submodules has been added, though)
- basic read and write access to the Git *index* (staging area)
- appending to *reflogs*[18]
- support for packed references (i.e., references stored in the *.git/packed-refs* file)
- access to the configuration local to the repository (*.git/config*)

There still is some functionality missing that would be required to ingest really all Git repositories, such as extensions of the index format [19] or replaced objects [18, git-replace]. Also, the implementation of pack file deltification (which significantly optimizes the disk space occupied by a Git repository) has been postponed, since it is not absolutely necessary for working with a Git repository. There is currently no interface to access the Git *stash*; even without a special interface, to edit the stash in a way compatible with the canonical *git stash* command, more comprehensive support for editing the reflog would be needed [18, git-stash].

Instances of GitRepository are used to access Git objects (i.e., blobs, trees, commits, and tags), resolve and manipulate references (and thus, branches) and access the repository's configuration. On top of a GitRepository, a FileSystemGitRepository can be created, which can create file systems on branches and commits. These file systems are of the class GitFilesystem, which uses a GitStore as the backing store (which is a FileSystem store, not a store that creates snapshots from objects as the term is employed otherwise in this report). Each GitStore starts from a single Git commit and its associated tree. When entries are accessed in a GitFilesystem, the tree is traversed down to look up the blob or tree at the requested path. When a blob or tree is accessed, it is copied from the repository as a file or directory to a MemoryFileSystem that serves as cache and as temporary storage space. When files are opened in the Git file system, they are actually opened in the memory file system. Whenever a blob is opened for writing or when a directory is created, it will be remembered by a GitStore's GitModificationManager. When a directory entry is deleted, the deleted path will be remembered as well. Eventually, when a new commit is created from the GitFilesystem, new blobs and trees will be created in the repository for the remembered and deleted paths, from the leaf blobs up to the root tree. A new commit will be created in the repository with that new root tree.

[18] https://git-scm.com/docs/git-reflog (last accessed November 5, 2017).

4.2.2 Squit

On the one hand, it was clear from the beginning of the implementation of Squot, that it should connect to Git repositories. On the other hand, it was decided that Git should not be hard-coded into the design, so that it could be replaced in the future if a new, widely-used version control solution with platform support emerges and it would warrant a connection to Squeak. Pur is already mentioned in the previous chapter as an abstraction for version control systems in which history can be modeled as a directed acyclic graph of versions [35]. Squot and Squit are based on the design of Pur, although Squot does not adhere strictly to the interfaces that Pur defines.

A Pur *repository* is an object that contains *versions*, which form history by the parent-child-relationship among themselves. History can be accessed through *historians* of a repository. A historian represents one view of the history inside the repository, by referring to one tip version, and the historian can change and carry forward that history by updating its reference to another version. Historians represent label-based branches (that are like Git branches). A repository can create new historians and delete existing ones.

Should the need arise, it should be possible to connect Squot to a different VCS than Git by only implementing suitable adapters for repositories, historians, and versions.

Following that nomenclature, Squit defines the classes SquitRepository, SquitHistorian and SquitVersion. A Squit repository wraps a FileSystemGitRepository and adapts or delegates all operations that query or alter the Git repository to it. Moreover, it provides a place to store credentials needed to connect to remote repositories, and it keeps a cache of SquitVersions. Each Squit historian wraps a Git reference (e.g., *refs/heads/master* for the master branch) and it knows its Squit repository, and the low-level GitRepository, to perform updates to the Git reference when a different version is assigned to the historian, or to delete the reference when the historian is deleted from its repository.

A SquitVersion knows the GitCommit that it wraps and the containing SquitRepository. As an object container, it also provides methods for comparison with other object containers. When two SquitVersions are compared to each other, the computation is optimized by only comparing those object graphs whose files were modified according to the tree differences obtained from FileSystem-Git.

But even if an operation were performed that involves a SquitVersion and a different kind of object container, all artifacts in a SquitVersion are extracted lazily by default. In fact, Squit uses the previously described SquotFileSystemStore to create trees and blobs, since FileSystem-Git provides access to them like regular files and directories.

When a new SquitVersion must be created, which happens with a SquitVersionBuilder for reasons given below, a GitFilesystem is created on the first parent commit. On the root directory of this file system, a file store is used to update all files and folders belonging to Squot artifacts. After the files have been written, a new commit is prepared from the GitFilesystem, which stores the new trees and blobs in the repository, but not yet the commit. The author information, the message for the change log and possibly additional parent commits must be added by the SquitVersionBuilder

first because a Git commit cannot be changed after it was stored in the repository once; its hash would change as well. Finally, the completed commit is stored in the repository, and a new SquitVersion that wraps the new commit is created.

To update a Git reference to point to the new commit, the newly created SquitVersion must be assigned to a SquitHistorian. Or the message to create a new version must have been sent to the historian in the first place, then it will automatically update its reference. The latter way is actually preferred because the reference update will be identified as a commit operation in the reflog.

In order to merge object graphs, a suitable merge-base version needs to be found among any versions for Squot, just like a Git merge needs to find a merge-base among commits. The merge-base algorithm implemented for FileSystem-Git does not really depend on any Git specifics—it is rather a special-purpose search algorithm that operates on directed acyclic graphs—so it is desirable to reuse the implementation of the algorithm in Squot. But neither Squot should depend on FileSystem-Git, nor should FileSystem-Git depend on Squot. Therefore, the merge-base algorithm has been outsourced into a separate package simply called *VersionControl*, on which both FileSystem-Git and Squot depend.

4.2.3 Working copies

Now we can read versions from a repository and create new ones, and we can capture snapshots from an image store and restore snapshots to it. Connecting the two is the responsibility of a SquotWorkingCopy, which combines one store with one historian (and, hence, a repository). Working copies can be registered in a class instance variable of SquotWorkingCopy, so they remain globally accessible (and will not be garbage collected when users accidentally close the only workspace that held a reference to the working copy).

The working copy is actually intended as the primary scripting interface for Squot users (see Listing 4.6 for examples). It understands messages to add and remove objects (which are delegated to the store), add new versions to the historian based on the store's current objects, perform merges, switch to another historian etc.

Being one form of the user interface, usability aspects come into play. Git has a reputation of being more complicated to use than other VCSs. Analyses of the causes of that have been conducted by De Rosso and Jackson, leading to a redesign of the Git user interface named Gitless [11, 12]. Some of their ideas have been incorporated in SquotWorkingCopy, such as automatic saving and restoring of uncommitted changes upon switching a branch, and not allowing the so-called *detached HEAD* state of Git.

4.3 Graphical User Interface

To make Squot and the new Git connectivity for Squeak more easily tangible, a GUI has been created using Squeak's ToolBuilder API.

Listing 4.6: Examples of the working copy API

```
"add an object and create a new version"
aWorkingCopy
    add: (PackageInfo named: 'MyPackage')
    at: 'src/MyPackage.package'.
aWorkingCopy saveNewVersionInteractivelyIfCanceled:
    [Transcript showln: 'User canceled the save dialog'].

"merge another branch"
master := aWorkingCopy loadedHistorian.
branch := aWorkingCopy repository historianNamed: 'other-branch'.
aWorkingCopy
    mergeVersionInteractively: branch version
    ifCanceled: [Transcript showln: 'User aborted the merge'].
"at this point, no merge version has been created yet,
 but the branch version is remembered as the second parent version"
aWorkingCopy saveNewVersionMessage: 'merge other-branch' interactive: false.

"change the active branch and modify an object there"
aWorkingCopy switchTo: branch.
"new versions would now be created on the branch"
aMorph := aWorkingCopy objectAt: 'MyActiveEssay.morph'.
aMorph position: 200 @ 200; addDropShadow.

"go back to master, keep the unsaved changes"
aWorkingCopy switchAndMoveUnsavedChangesOverTo: master.
aMorph position. "=> 200 @ 200" aMorph hasDropShadow. "=> true"
"...and then change your mind"
aWorkingCopy discardUnsavedChanges.

"pick something from the branch: a dialog will be opened to choose changes"
aWorkingCopy
    loadArtifactInteractively: (branch version artifactAt: 'test/testdata.ston')
    ifCanceled: [Transcript showln: 'User canceled the load dialog'].
```

4.3.1 Git browser

The primary tool to interact with a working copy and the history of the connected repository is the *Git browser*. It is not named after Squot because all user interface elements, such as buttons, are labeled with Git terminology (i.e., *branch* instead of *historian*, *commit* instead of *version*, *checkout* instead of *materialize* or *load*, etc.). The reason behind this is that Squot was provided to a course of about 80 bachelor students who had to work on software projects and host them on GitHub during the months leading up to this report. They should not be burdened with learning two different vocabularies for version control simultaneously. Since the target platform was GitHub, it was obvious that Git is involved, so hiding Git from them was less of an option than hiding Squot from them. For students who have already been familiar with Git, it also meant less of an overhead to learn the new tools.

The Git browser is divided into six panes (see Figure 4.3):

1. a list of projects (working copies)
2. a list of branches (historians) in the selected project
3. a searchable list of commits (versions) that is a linearized history view of the selected branch
4. a text with information about the selected commit
5. a list of objects (actually artifacts) contained in the selected version
6. a central ribbon of buttons that can be used to trigger common actions on the selected branch or the selected version (depending on the action)

In addition to the buttons, many more actions can be accessed through the context menus of each of the four lists. The actions in the context menu of the branch list allow users, for example, to switch the working copy to the selected historian, or to remove that historian from the repository. The action to add a new project (i.e., create a new working copy) or to clone from a remote repository can also be found in the context menu of the list of projects. Adding new objects to the set of tracked object would also happen via the project list menu, but because of the target audience mentioned above, there are only actions to track packages (i.e., Smalltalk code), currently. Other kinds of objects must be added by sending the appropriate messages to the SquotWorkingCopy object from a workspace or any other Smalltalk code editor.

The model class for the Git browser is SquitBrowser and belongs to the Squit package accordingly. Most of the features it provides can be generalized from Squit to Squot, but the Git terminology does not fit the names used by Squot; and some features are really specific to Git (e.g., dealing with remote-tracking branches).

Because the building of the commit list can produce a noticeable delay if there is a moderate number of commits in the selected branch, only the top 30 commits are displayed at first. The rest of the list is collected asynchronously and appended to the displayed list when finished. Being able to work immediately only with the most recent 30 commits is sufficient under the assumption that the most recent

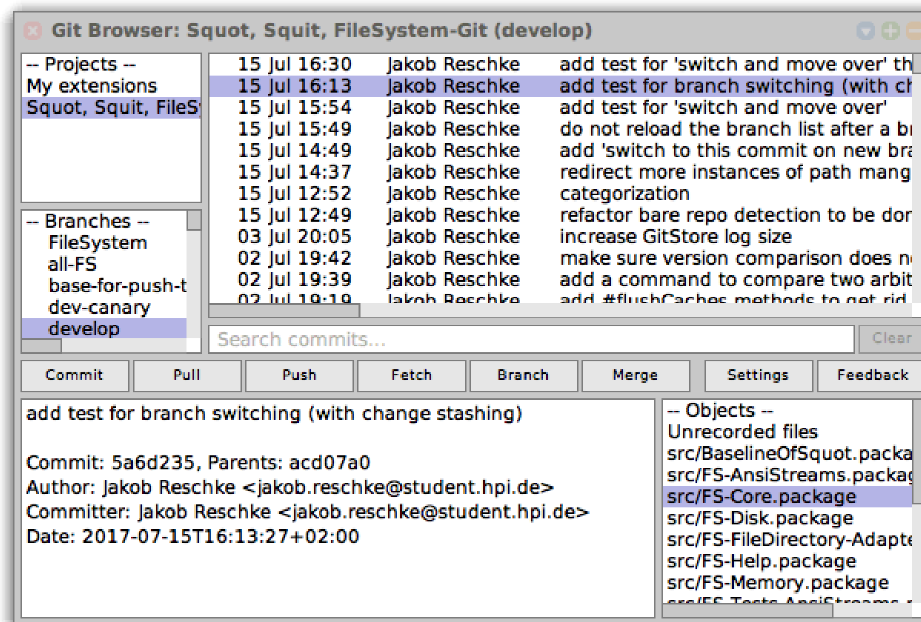

Figure 4.3: Screenshot of the Git browser. A working copy of the Squot repository itself is currently viewed. The commit list shows commits on the *develop* branch.

commits are in most cases also the ones of most interest (e.g., having a look at the recent work of co-developers to see if their commits should be merged into one's own working copy). The object list is loaded asynchronously in its entirety because building it used to be slow if the contents of a version was voluminous (e.g., the Squot repository itself currently includes 16 packages, some of which have more than a dozen classes and lots of methods). This was later solved by having SquitVersion and SquotFileSystemStore load their artifacts lazily, as described above, because all that is needed to display the list in the Git browser are the paths of the artifacts, not the snapshots of any contained objects.

The context menu of the object list can be used to trigger actions that affect only the selected object. It also allows the corresponding live object to add menu items that are specific to the type of the object. For example, four menu items that can be seen in Figure 4.4 allow users to edit the scripts associated with a PackageInfo object. This menu is not an ideal location for such actions because, now, it contains actions for both the historical edition of an artifact and for the loaded edition. There is no separate GUI list for the objects in the working copy, so this is the best place to hook up such actions currently. The actions on the live object are added by sending squitBrowserSupplementObjectMenu: with the menu under construction to the live object. The *Browse edition in selected version* action offers a way to inspect the historical edition of the artifact without materializing it first. Which tool would be best for this task depends on the type of the object, of course. Therefore, the action is first dispatched to the artifact (sending browse), which will usually delegate to a shadow object by sending browseFromSquot: with the artifact as argument (so the path or the

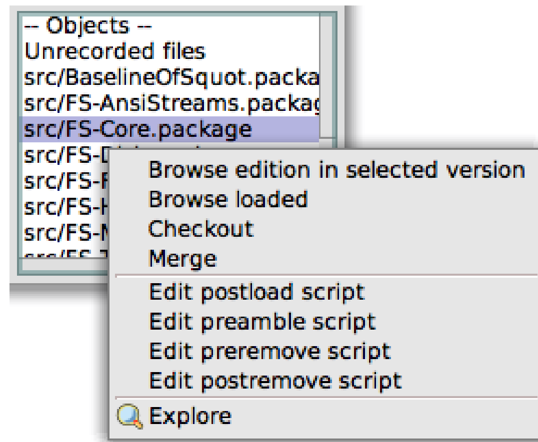

Figure 4.4: Context menu of the object list. Some menu items concern the historical edition of an artifact while others pertain to the live object in the working copy's image store. In this case, the script editing commands are supplied by the live PackageInfo object. The other actions are always available.

metadata can be accessed if necessary). The *Browse loaded* action does the same, but sends browse to the artifact in the working copy's image store, so the live object will receive browseFromSquot: in the end. The default implementation for both live objects and shadows in Object opens an inspector on the live object. The live PackageInfo opens a system browser on all the classes in the loaded package. SquotPackageShadow opens an MCSnapshotBrowser on its Monticello snapshot (which looks similar to the system browser for loaded classes).

4.3.2 Presentation and manipulation of differences

Whenever objects should be stored, materialized, or merged in the course of an interactive operation on the working copy, a dialog will appear that allows users to select and reject pieces of the differences to be applied. For example, the dialog for creating a new version/commit offers such a selection (see Figure 4.5). This introduces another aspect to differences not described previously, which is the ability to modify them before applying. To create the tree view on the left in Figure 4.5, the SquotObjectGraphDiff from each artifact must be transformed into a tree first. In the example, only changes to packages are displayed, which have a tree structure anyway (primarily that is: packages above classes above methods). For other kinds of objects and their differences a different form than a tree might be more beneficial. However, this was the most practical way to implement the GUI for the time being. Via the context menu of the tree view, nodes can be included or excluded from the current operation. Unless the dialog is canceled, the exclusions must be propagated to the SquotObjectGraphDiff or parts of the differences contained in it. Therefore, a two-way conversion between differences objects and a hierarchical (tree) form for their presentation is needed.

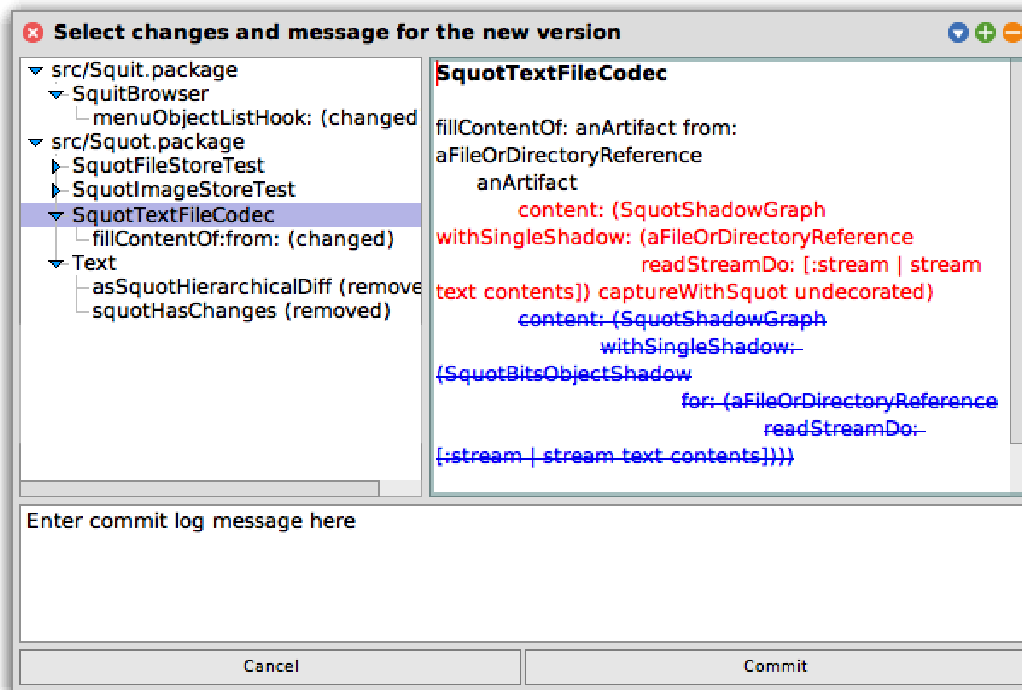

Figure 4.5: Change selection in the dialog to create new versions/commits. The tree on the left side presents differences in a tree structure. The right pane shows a textual representation of the changes included in the selected node on the left. Each node can be excluded from the new version via the context menu of the tree.

The protocol for this is as follows: Each SquotArtifactDiff can be asked for a list of top-level nodes. The class for such nodes is SquotDiffNode. Each node can have a title—used as the node label in the tree view—, a collection of child nodes, a flag whether the change is currently included or excluded, and optionally a reference to the relevant part in the object differences graph (for later reference, we will call such a parts *difference particles*). In three-way merges, nodes with conflicts also store which choice a user made to resolve the conflict (or if no choice has been made yet). To prevent the change selection dialog from being affirmatively closed before all conflicts have been resolved, nodes can be asked whether any of them or any of their descendant nodes still requires a choice to be made (unresolved conflicts require a choice). Currently, the nodes are already created when the SquotArtifactDiff is initialized (i.e., they are *always* created when diffs for artifacts are produced). This is done by sending asSquotHierarchicalDiff to the diff object obtained by comparing two (or three, in a three-way diff) artifacts. For example, SquotPackageDiffs produce their tree structure (which can be seen in Figure 4.5) from the wrapped MCPatch or SquotMonticelloMergeDiff. For SquotObjectShadows, only a stub implementation is available so far. Only changes to the slots of the start object can be excluded because no nodes are created and processed for the related objects.

The user selection must eventually be brought back into the various diff objects. To achieve that, the messages adjustedBy: and squotAdjustToResemble: are sent to artifact diffs and object graph diffs, respectively, with the corresponding top node as the argument. A SquotObjectGraphDiff will make a copy of itself and then delegate the operation to the diff of the start object. It is the responsibility of the object diffs to check whether their node has been excluded (deactivated) or not (because only they know how to handle this situation), to update themselves and the graph diff accordingly, and to continue the adjustment with the child nodes for diffs of the corresponding related objects. For package diffs it removes Monticello patch operations from the wrapped Monticello patch when necessary. Monticello conflicts are resolved with the choices made according to the nodes. Because the same diff object is responsible for both building the difference nodes and applying them back, it can make use of the difference particle in each node. In the case of package diffs, the Monticello patch operations are stored as difference particles in the leaves of the tree. An example for a package diff and its hierarchical representation can be seen in Figure 4.6.

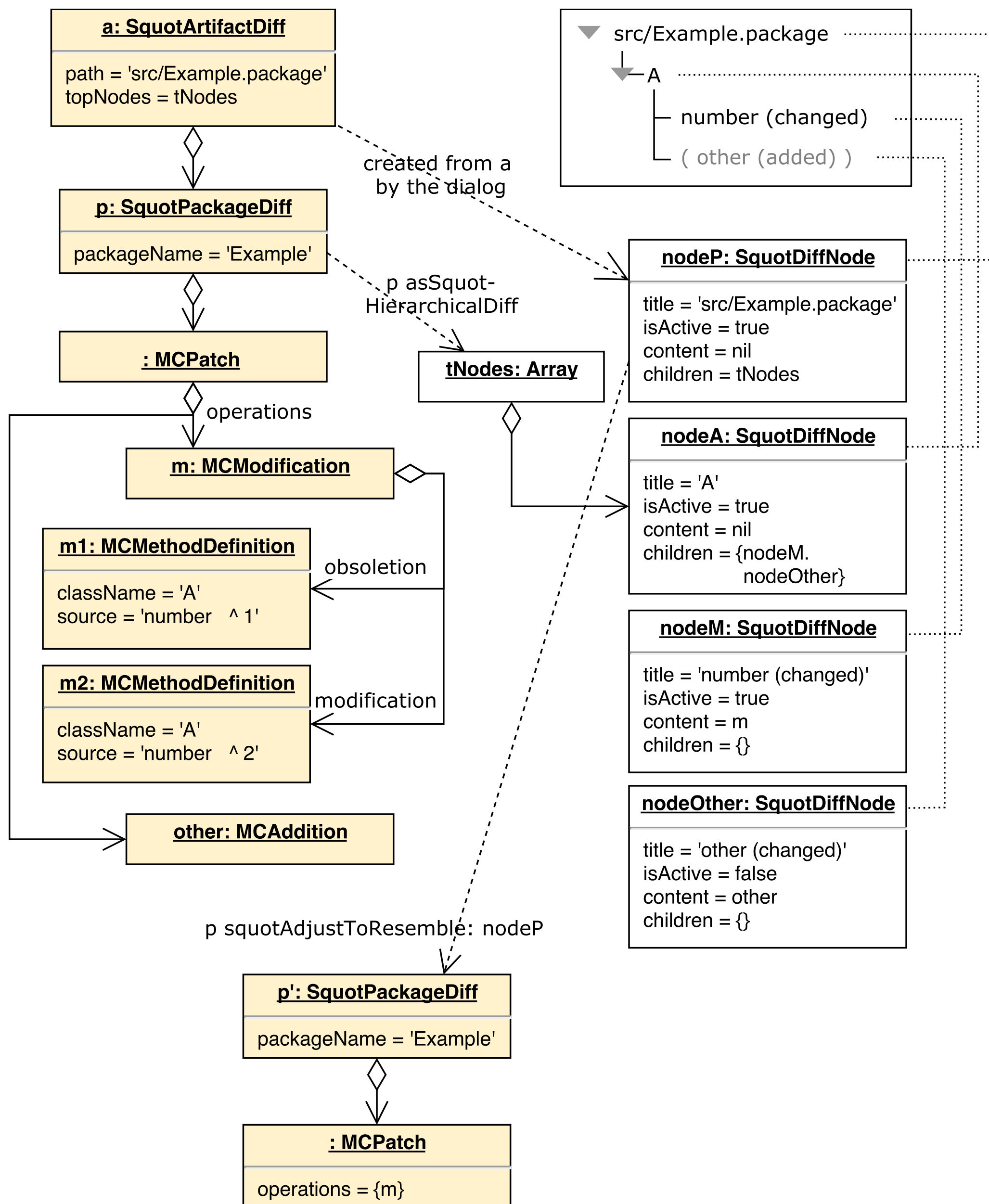

Figure 4.6: Example setting for the conversion between differences and trees for display in the GUI. The SquotObjectGraphDiff is omitted from the picture to save space. It goes between the SquotArtifactDiff and the SquotPackageDiff.

5 Evaluation and discussion

The prototype *Squot* introduced in the previous chapter has been used productively during the past months. Several groups of students have used it to manage the source code of their Squeak projects. These projects are all hosted on GitHub. Also, Squot is used to manage its own code. In this chapter, Squot and the architecture from chapter 3 are evaluated with regard to the other systems and the goals described in chapter 2. Observations, issues and possible improvements are discussed.

Squot and Squit come with a suite of unit tests that check a variety of cases, such as the capturing, materializing, and comparison of primitive objects, collections, cyclic data structures, global variable bindings; the preservation of object identity; snapshot creating and restoring for image stores and file stores, and the application of patches to them; and basic working copy tasks such as creating new versions, reverting to older versions, and performing merges. The test suite does not claim to cover all combinations and corner cases. Some of the tests use a scenario where the window (i.e., the morph) of a workspace with variable bindings is tracked, modified and restored. This combines many different kinds of objects and circular dependencies in the object graph, as well as metadata influences on the capturing, so this test running successful gives an impression of the functionality of Squot's object tracking.

The example specializations presented in subsection 4.1.8 show that:

- the general (slot-based) approach to object snapshots is extensible to support types with special requirements,
- domain objects can opt out of the general approach and implement their own,
- existing classes for snapshots and differences (e.g., those of Monticello) can be reused and adapted for Squot, and
- the format of snapshots and the format of the generated files can be customized separately—to use a different file format, it is not always required to change the kind of snapshot used for an object.

In comparison to the previous workflow to host Squeak code on GitHub (using Monticello-FileTree and the canonical Git command line tools) the process has been streamlined. Instead of having to save all packages belonging to a project (e.g., one core package and a test package) separately in Monticello and, on top, create a commit with Git, it can all be done in a single action from the Git browser. If separate commits for the different packages of a project are desired, the other packages can be deselected in the commit dialog. No further interaction with the canonical Git command line is required. Moreover, the path to a package inside the repository needs no longer be known to load the package from Git. For Monticello-FileTree, a

repository had to be created on the folder which contains the package subfolders as immediate entries. Because of Squot's table of contents, which works a little bit like a makefile, packages can be located in different subfolders and still be loaded in one action (e.g., when there are external libraries in a *vendor* or *lib* subfolder, and the actual source code is in a different *src* subfolder).

Merges were a complicated task with FileTree: First, a Git merge had to be performed, supplying arguments on the command line to make Git resolve all conflicts by choosing the incoming ("their") changes, rather than the local ones ("mine"). Second, a merge had to be performed in Monticello. Because the code directory on which FileTree operates would contain all the changes not already present in Squeak, the incoming changes would not go missing. After the merging with Monticello, a new Monticello version had to be saved to the FileTree repository, to write the merge results back to the files. Finally, the merge commit created during the previous Git merge had to be amended (i.e., overwritten), so that it would contain the properly merged files. Not only is this a cumbersome procedure, there are also multiple opportunities to make mistakes and eventually lose changes. Certainly, it is not ideal for any newcomers to Git and Squeak. With Squot, merges have become much simpler. Actually, the process of dealing with conflicts and reviewing the incoming changes has not changed and remains a complex task for beginners in itself. But the surrounding procedure is now simplified to invoking the merge action on a branch or commit. After the conflict resolution has been performed, the user is prompted to create a commit to persist the merge in the version history. No interaction with the Git repository outside of Squeak is necessary (and no command line switches need to be remembered to produce the correct results).

However, Squot does not currently regard any files that are not associated with tracked objects. This is a problem during merges, because these files need to be merged as well—especially in projects that have modules both in Smalltalk and in other languages. But also pure Squeak/Smalltalk projects can have files that are not of immediate interest for the image: Readme texts, license files, and configuration files for external services, such as Travis CI[1]in combination with smalltalkCI [49]. One solution would be to track all such files as objects in Squot (e.g., a SmalltalkCISpec object), but the more general and less cumbersome solution would be to also merge the non-Squot files. Squot already has a placeholder object for all these files (SquotUnrecordedFilesArtifact), which could be extended to take care of merging. It might be tricky to thoroughly separate the files produced by Squot serializers from the unrelated files, though.

Since we implemented a connector to one backend VCS only (Git), Squot does not prove that the design will also support other VCSs. But the implementation of snapshots (shadows), the image store, file store, and working copies do not depend on methods only defined by Squit. Moreover, since Squot is based on Pur, for which connectors to Git and Mercurial have been shown to work [35], we believe that the implementation of connectors for additional VCSs is feasible.

[1] https://travis-ci.org/ (last accessed November 5, 2017).

While other objects than packages have not been tracked in the student projects, it is now possible to commit objects together with packages. The separation between saving the code and exporting objects to the disk has been lifted. Objects and source code can be tracked in the same Git repository with a unified workflow.

Because Git's object model puts the version history on top of whole directory structures (i.e., each commit can contain multiple objects), each commit already describes a configuration of the objects stored therein. Further configuration management for the objects in the repository is not necessary; by checking out objects from a specific commit (or materializing objects from a version, in Squot terms), configurations can be restored in the Squeak object memory in a single operation. Squot does not currently handle dependencies on external packages or objects (i.e., such hosted outside of a project repository). Neither does Monticello, as this task is delegated to another library called *Metacello* [2]. Metacello can also be used to install packages hosted in a remote repository into an image. Since Squot uses the FileTree/Cypress format to serialize Smalltalk code, Git repositories managed by Squot can be accessed with Metacello like any other Git repository that contains Smalltalk code in the Cypress format. However, Metacello does not know anything about Squot's capability to track arbitrary objects and, hence, cannot install other objects than packages. To remedy this, an integration with Metacello could be worthwhile in the future.

One problem associated with changing the structure of classes—such as adding or renaming instance variables—is migrating the state of existing objects to the new schema. Assuming that users will properly migrate the state in their own programming environment, the tracked objects will also be properly migrated in other users' environments when they update objects from a version that was created after the schema change. These objects have been captured after receiving the treatment of the migration, after all. But objects that are not tracked with Squot cannot benefit from that, however. Therefore, it is still advisable to take other measures to migrate existing objects, such as migration code in preamble (pre-load) and post-load scripts of packages.

In the previous chapter, some missing features have already been mentioned. Some other tools are still missing from Squot. For example, there are no special browsers for graphs of object shadows (they can be explored with the object explorer, but it will reveal implementation details), and there is only a stub implementation to convert their differences to trees for the selection of changes. Also, any object metadata can only be edited by evaluating the necessary Smalltalk expressions in workspaces, inspectors, and object explorers, since there is no GUI tool to do that. Such tools should be added in future versions of Squot, to make the tracking of objects similarly simple to tracking packages.

While the merging of packages hosted in Git repositories has been simplified as described above, merging for arbitrary objects (with slot-based shadows) is not implemented yet. The reason is that further research must be conducted first on *how* objects can be merged correctly. This problem is not solved in this report, but the example in Figure 5.1 briefly introduces one of the issues. Some basic assumptions in line-based text merging via three-way differences do not always make sense in object graphs, such as: if the base and one of the other two alternatives for a comparison

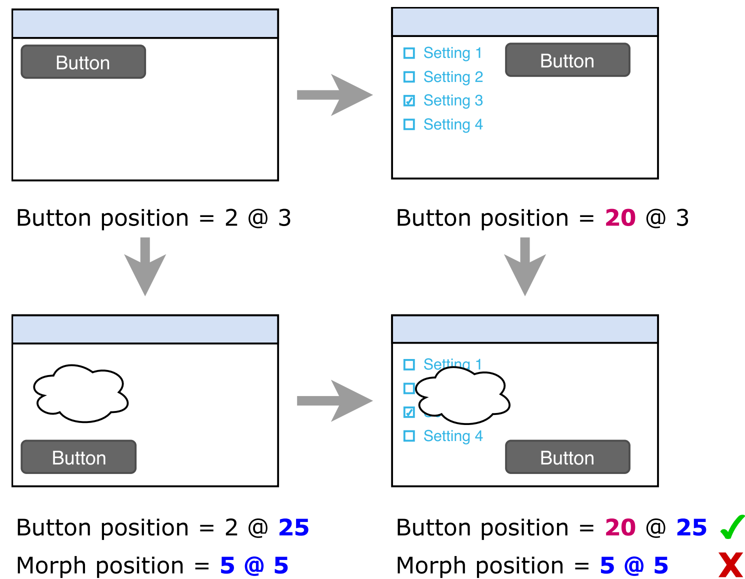

Figure 5.1: Merging even small object graphs correctly can be difficult. In this example, a window and its contained widgets are tracked. Initially, the window contains only a button. On one branch, a sidebar is added to the window, which moves the button to the right. On the other branch, a new morph is added above the button; the button is moved downward. When the two branches are merged, the position point of the button could be merged component-wise without a conflict. In this example, the component-wise merge yields the correct result. If the button had instead been moved to the far right of the window in the upper-right branch, the button would no longer be aligned with the morph from the lower-left branch, which could be an intention of the author of the lower-left branch. But since the automatic merge cannot know whether the morph should be moved to the right like the button, an automatic merge would probably result in a wrong position of the morph anyway, without indicating a problem or conflict. Layout containers could help in this particular case to make each user's intentions about the layout explicit.

atom (a line) are equal, the solution is the third alternative. It could be a good idea to delegate the determination of a valid merge strategy to the merged objects themselves, which is also done for determining suitable snapshot types. However, to determine a useful default implementation for merging (or to determine whether such a default exists), it should be investigated how conflicts are resolved manually in real-world merge scenarios for objects.

Squot also does not currently provide a general solution to trace the history of individual objects deep inside of an object graph. Similarly to Git, each version/commit must be inspected for changes to a particular object/path. In contrast to Git, appropriate tools to hide this circumstance do not exist yet in Squot. While object names make it easy to retrieve objects from a given graph, not all objects get names assigned. For example, the package snapshots created by Monticello for SquotPackageShadows do not assign object names to method snapshots and do not collect them in a data structure that supports a constant-time lookup.[2] If the tracing of inner object history is requested in the future, the interface of snapshot objects could be extended to offer the lookup, given one snapshot edition and its origin object graph, of the equivalent edition that is reachable from the receiver snapshot. Object graphs could try to look up the object by its name and if that fails, delegate the look up to the snapshot of the start object. Given a Monticello method definition, a SquotPackageShadow could then try to locate an edition of the method in its wrapped package snapshot. For slot-based snapshots, the object could be searched in its original graph and the detected path to it from the start object could be used to find an object in the lookup graph. An architecturally simpler alternative that puts additional burden on custom capturers, however, would be to demand that all objects that should be traceable must be registered with a name in the object graph, even if they are value objects or have intrinsic names already. Instead of UUIDs, names derived from the objects in a well-known manner could be used in such cases (e.g., a method's class name and the method selector). Other VCSs like Mercurial have built-in per-file history support in the repository. Taking advantage of that would require optional features in the repository interface and a way to extract the connection between a snapshot object and a file from the used serializer.

In general, when the features of the backend VCS are considered, there is a trade-off between abstraction and interface simplicity on the one side and performance on the other side. Pur defines a minimal interface to interact with VCSs, and even its example implementation in Newspeak already defines some extensions [35]. A prominent VCS feature that warrants to be exploited is the listing of changed objects/files, for which there are probably appropriate data structures in place. Finding all changed objects could also be implemented in Squot without consulting the version control backend, but it would mean to deserialize all objects from a version in the backend to snapshots first and then inspect all of these snapshots for changes. Under the assumption that only a small portion of the tracked objects changes from one version

[2] An MCDefinitionIndex can be created from a set of definitions, but its creation accesses each definition in the snapshot, so it does not help for looking up a single definition only once.

to another, this would mean a huge overhead without the help of the version control backend.

For operations that could utilize features of the backend VCS, an abstract reimplementation of these features can be beneficial when a new connector to a different VCS is developed. Like a polyfill[3], the abstract implementation could be used until the new VCS connector provides a more efficient solution itself. However, providing the abstract solution also implies a redundant implementation and, after connectors have been completed, dead code. It still remains a useful strategy for features that might be available in only a few VCSs, but which could be utilized for significant performance improvements if available.

One potential problem with the current implementation is that objects cannot be shared among multiple object graphs. If one object is referenced in multiple graphs, these graphs will overlap and create redundant snapshots (and storage data) for the objects reachable from the common object. At least the redundant snapshots will have equal names due to the shared object registry in an image store. Shared objects would be "rejoined" when they are materialized. But it could also lead to inconsistencies when changes to common objects are committed for only one of the involved graphs. The currently proposed solution is to let users make sure that the graphs spanned by the tracked start objects do not overlap. Because there are no special tools to easily explore the reach of an object yet, ensuring this can be difficult. Independent building blocks of the system under construction must be carefully identified and then suitable start objects must be chosen. For source code, such slicing has existed for a long time in the form of packages (under various names in other environments), but for exploratively-built objects it might be less clear where the separating lines can be drawn. On the one hand, reflecting about the structure of the system and paying attention to loose coupling is advantageous; on the other hand, it slows down and distracts during an exploratory programming session—particularly in the earlier stages when the final result is far from clear. Version control might be desired for these early exploratory steps anyway. Tracking many parts in a single graph could encourage tighter-than-necessary coupling. Without tools for tracing the history of inner objects, it may also imply less awareness for changes—not least because only a single object is selected for tracking and the rest of the spanned graph results implicitly.

If multiple parts of the system are tracked independently (i.e., in separate object graphs), it could be confusing where the proper place to compose them is and how (or if) this composing part should be tracked (without including all the other parts). One reason for that is that it is unclear *where* live objects should be put when they are materialized from a snapshot—unless these live objects have already existed and are visible on the screen, for example. The materialized objects are added to an

[3] "Shims and polyfills are libraries that retrofit newer functionality on older JavaScript engines: A shim is a library that brings a new API to an older environment, using only the means of that environment. A polyfill is a shim for a browser API. It typically checks if a browser supports an API. If it doesn't, the polyfill installs its own implementation. That allows you to use the API in either case." [54, p. 405].

image store, but the image store is (probably) not part of the software or document under development and out of reach for this reason. The materialized objects might need to be assigned to some class variable or associated with content holders, such as graphical tools. There are several possible ways to address this problem. One is to introduce a concept of "places" that can hold tracked objects (like places and generalized references in Common Lisp [52, ch. 5.1]). The place where an object should be materialized would be stored in the tracked object's associated metadata. In some cases, these places might be transient (e.g., a workspace that exists only in the environment of one user), in other cases they might need to be persisted (e.g., when the object should be assigned to a class variable). A different approach is to mandate that container objects that wrap or hold the actual objects of interest should be tracked instead of the actual object. Looking at the simple example of tracking a piece of text, this could mean to track a StringHolder model which contains the String or Text object, instead of tracking the latter directly. Workspaces can be opened on the model to view and edit the text. Tracking objects in class variables could be explained as tracking the class object with a special capturing mode (i.e., one which captures a subset of the state of the class, instead of its source code). The capturing mode would be recorded in the metadata. The difference between the "places" and the "containers" approaches is that, in the first, the location of an object is stored in the metadata, whereas in the second, the location itself is tracked as an object. When containers are tracked that include more than one object (e.g., when there are multiple class variables in one class) this would lead to larger object graphs again where different parts of the system could be combined unnecessarily. Sometimes there is no suitable container object for the desired location of an object. For example, there is no special object that represents a live class variable in Squeak.[4] A "synthetic" container that represents the particular class variable and manages to update it could be introduced and tracked. A StringHolder object to wrap a text might also be regarded as a synthetic container.

Coming back to the problem of graph separation with respect to shared objects, it could also be addressed by introducing "external references" among object graphs. It would be necessary to assign a "home graph" to each shared object (or even slots, which would be similar to the package annotations described for the Self Transporter [68]). If an object from one graph is reached while capturing a different graph, an external reference to the object would be created instead of capturing the object. An external slot would simply be omitted from the snapshot. Because we do not yet have experience on how version control for objects would primarily be used in practice, it is difficult to know whether support for shared objects and external references is necessary.

Another concern, not entirely unrelated to the proper separation of object graphs, is that not all objects contained in a version should always be materialized in the

[4] The association object that is the binding of the variable in the class pool dictionary comes closest as a candidate for tracking. But this is a crude, hard-to-explain solution and it exploits the implementation details of both class pools and dictionaries in Squeak.

image. Currently, Squot assumes that an object graph should be deleted (i.e., untracked) if it is absent in a working copy's store, but present in the parent version upon which a new version should be based. If a repository contains packages for different platforms, so not all of them can be loaded at the same time,[5] support for unmaterialized object graphs without signifying their deletion becomes a requirement. It could be necessary to make the untracking of object graphs explicit in the state of a store to achieve this.

To speed up the creation of new versions, it would be beneficial if serializers were able to write only those files (or even parts of files) that must actually be changed. In other words, serializers should be able to work with differences, just like object containers. Currently, all implemented serializers always write out (and overwrite) all files belonging to an object graph. Reading only the subset of changed files from other versions could be difficult because a file store operates on only one tree of files and directories and can, therefore, only access the files of one single version. Hence, differences-aware deserializers would have to cooperate with other objects, such as a version or a repository, to obtain the necessary information about changed files. Or such data would have to be provided as input data to deserializers. Both approaches would require a VCS-neutral interface to communicate differences in trees of files and directories, to keep deserializers independent from concrete VCS backends.

Creating new versions could also be sped up on the other end of the process, at the live objects. Capturing an object graph currently involves traversing the whole graph each time. If it were possible to capture just the objects that have changed since the last capture operation, time could be saved, especially for large object graphs (or packages with many methods) with only small changes from one version to the next. Evolving a Smalltalk program can feel particularly swift because each method is compiled just after it has been edited. The software is compiled incrementally in small steps, which prevents slowing down the programmer with lengthy compilation steps. For classes and methods, this could be leveraged by also capturing them whenever they are saved. Changes to code are announced via a global notification mechanism in Squeak and Pharo (and probably in other systems as well). Squot's image store could hook onto such notifications at the request of tracked objects, to realize incremental capturing. Of course, no such notifications are signaled when an instance variable of an arbitrary object changes, so incremental capturing might only be available in some domains. Or a language extension that intercepts assignments to properties of objects would have to be used [53].

When FileSystem-Git was extended and later deployed with Squot and Squit for production use, some lessons have been learned regarding the priority of some of Git's features. Future attempts to work on a Git implementation or maybe even other VCS might benefit from these insights. When Git pushes objects from one repository to another, references (e.g., branches) at the remote are updated. In the canonical

[5] The repository of Metacello is an example for this. It contains packages specific to Squeak and Pharo, respectively, encapsulating the use of APIs specific to each, such as file access. The Pharo variant would not work in Squeak and vice versa.

Git implementation, it is detected when updating a remote reference would make some commits that are currently reachable from the reference no longer be reachable after the push. This is the case if someone else pushed onto a branch while one was working on the same branch locally. The canonical Git implementation will reject the push, so no commits are lost at the remote. References can be updated anyway in this case with a *forced push*, which can be commanded with a command line flag. But technically, the force-push prevention is not visible in the Git protocol to synchronize objects. It must be detected and handled by the connected parties (client and server), which both can deny a forced push, but the situation is not formally advertised in their communication. Usually, Git servers do not prevent forced pushes and on GitHub such prevention must be enabled explicitly in the repository settings. The initial released version of Squot and Squit did not have a detection for destructive pushes in FileSystem-Git, so all pushes were essentially performed like forced pushes. This obviously lead to problems of people overwriting each others changes on GitHub, so the detection of destructive pushes has been added soon after. Users now have to confirm the push when a forced push would be necessary.

Another feature that was absent in the beginning was the clone operation. The reason for its absence was that it can be reproduced by either 1) creating a new, empty repository, adding a remote repository, and fetching the objects; or 2) by cloning a repository with the canonical Git or any GUI for it, and adding the new local repository in the Git browser. While none of these steps require any extraordinary Git expertise and the steps were demonstrated in an introductory lecture, the difficulties during the first steps of users with any new tools should not be underestimated. Providing a built-in clone operation from the very beginning would have avoided some support requests in the start phase of the student projects.

Later during these projects, another issue emerged. For some users, more and more redundant remote-tracking branches started to accumulate in the Git browser (e.g., next to the expected *master* and *origin/master* branches, branches like *origin/-origin/master* started to appear). The most probable explanation is that these users worked on the remote-tracking branch (*origin/master*) instead of the corresponding local branch (*master*).[6] This was possible because for both local and remote-tracking branches the same actions were made available by the Git browser. Technically, there really is no difference, except for the reference names (local branch references start with *refs/heads/*, while remote-tracking branch references start with *refs/remotes/*). But users should never check out (i.e., base a working copy on) a remote-tracking branch because its reference will be overwritten by the git fetch operation. The canonical Git client automatically creates and checks out a local branch when the user attempts to check out a remote-tracking branch. The lesson learned is, thus, that remote-tracking branches (or in general, branches that should not be modified by

[6] The error pattern subsequently emerged because of branch name heuristics applied when no upstream branch was configured for a given local Git reference. This lead to the repeated prepending of the remote name *origin* to these references.

users) must be treated differently than local branches in the user interface, even if there is no technical difference between the two.

If users want to make heavy use of pull requests on GitHub, it is advantageous if as few "technical" conflicts as possible arise (i.e., conflicts caused only by encodings and file formats, not by conflicting user actions). But even without conflicts, errors can slip in during automatic merges. In one such instance, when a project's *.squot* table of contents file was merged on GitHub, the last item was removed from it. The table of contents is stored in the STON format, which is based on JSON, where object entries and collection elements must be separated by commas. The last item of an object must not have a trailing comma, however. The automatic merge performed on GitHub removed the last element from the table of contents, but it did not remove the trailing comma from the preceding line. Consequentially, the file was no longer a well-formed STON document and the STON parser used by Squot refused to accept the file. The error was fixed by manually removing the comma.

However, this little detail in the file format revealed once again the impact of the choice of file formats on the version control experience. When comparing binary formats to text formats, this is obvious. But the automatic mergeability of different text formats varies as well. To facilitate automatic merges with general-purpose line-based text difference tools (as employed by GitHub), text files should

1. provide enough context to separate changes, so changes to different parts of the file will not be erroneously intermixed easily,
2. not contain complex nested structures that cannot be detected and guaranteedly preserved by line-based diff algorithms,
3. not require changes to adjacent items when one item is changed, added, or removed,
4. have a stable ordering of any encoded collection items, so swapping lines or blocks of text is always a significant change, and
5. be comprehensible by humans, so merge errors can be fixed easily when they occur.

STON and JSON do not fulfill the second and third demand. Even source code with nested blocks surrounded by any kind of brackets does not fulfill the second demand, but after all, automatic mergeability is not as important as the legibility of human-authored text files. The point is that complex nesting unamenable to line-based diff tools should be avoided for automatically generated files when it is not absolutely necessary for fulfilling the purpose of these files. Because Squot's table of contents currently does not store complex data, but only a flat list, a simpler file format without separator characters (except for line breaks) could be used in the future, to prevent merge errors. Alternative solutions would be to modify the STON parser to be more relaxed about the commas, or to declare merges with tools other than Squot itself as unsupported. The latter option is unpractical because one of the goals of Squot was to make it easier to use Squeak/Smalltalk projects with GitHub; denying the use of some of GitHub's features is therefore contradictory.

These observations about text file formats are also relevant when new file formats for the serialization of objects are designed or existing formats must be selected. A related aspect not covered in the list above is that data that should not be merged under any circumstances is best stored in separate files. Git will not merge content from blobs at different paths. This is one of the reasons why FileTree/Cypress stores each method in a separate file, even though it makes it more difficult to browse the source code directly in the files and on GitHub, compared to using the tools in Squeak (which are not available on the GitHub web page, of course).

6 Related work

In this chapter, a selection of other work on version control systems and object tracking, capturing, and serialization solutions are given and put in relation to our prototype implementation *Squot*. In addition, some work on user interfaces for version control is referenced and some starting points for the merging of objects, which is not yet generally available in Squot, are given.

6.1 Existing version control approaches

Pur is mentioned multiple times in this report as an abstraction for version control systems [35]. Looking at Subversion, Git, and Mercurial, it derives common concepts and transforms them into an abstract framework for version control with a state-based history model and directed acyclic graphs to describe version history. The abstract framework must be complemented with both a backend implementation for a particular VCS and with a frontend implementation that defines stores, the structure of snapshots, and how versions are created. Pur was a starting point for Squot, although Squot does not strive to be a full implementation of Pur. Image store, file store, working copy and snapshot types are the frontend implementation in Squot (snapshots being composed of artifacts, object graphs, and shadow objects). The backend implementation is Squit, which is an implementation of the Pur interfaces with Git as the target VCS.

Orwell is a version control and configuration management system for Smalltalk described in 1988 [65]. Its goal was to make development of Smalltalk software in teams productive. Orwell provides versioning of methods, classes, applications and configurations, all of which are stored in a single object database. The traditional *changes* file that accompanies a Smalltalk image is replaced by a *doIt log*, since the code for method and class editions is stored in the object database instead. Next to the textual representations, their compiled forms are also stored in the object database. The reason for this is to speed up the loading of other object editions by saving the time needed to compile the code. However, Orwell does not support to track arbitrary Smalltalk objects, as Thomas and Johnson note in their conclusion: "Our current solution eliminates persistent objects [i.e., globals or objects stored in pool dictionaries] by placing the responsibility for their creation with the class/application owner [who must write code to initialize such persistent objects]. Ideally it should be possible to manage such objects in the same database." [65, p. 141] Squot explicitly declares to make the tracking of any object possible.

As already described in chapter 2, Smalltalk systems such as VA Smalltalk, VisualWorks, Dolphin, Squeak, and Pharo provide specialized version control systems that

the building blocks of Smalltalk packages (or applications) directly, instead of files. VA Smalltalk comes with *ENVY/Developer* [51], VisualWorks offers *Store* [7], Squeak and Pharo provide *Monticello* [48], and Dolphin Smalltalk comes with a *source tracking system*, which offers a similar user experience to that of ENVY [59]. Some of them are well-integrated into their respective programming environments (e.g., by offering to browse other editions whenever a versioned object is encountered), but most of them focus on classes and methods and do not support to track arbitrary objects (or only in a fixed serialization format).

GitFileTree is an extension for Monticello in Pharo/Smalltalk that improves the integration of Git and FileTree [23]. It provides a special kind of Monticello repository that interprets the Git history of the selected FileTree directory to list Monticello versions. The standalone FileTree only ever shows one version per package because only one version can exist in a directory at one point in time. Operations on a GitFileTree repository automatically invoke Git commands, such as `git commit` when a new version is saved. Thereby, GitFileTree repositories feel less disconnected from Git than the regular Squeak-FileTree combination described above. Squot achieves the same goal, but in a different way. By using FileSystem-Git, Squot does not depend on the canonical Git being installed and it does not require external process call support in Squeak (usually provided via the *OSProcess* package[1]that is not included in the Squeak trunk).

Iceberg is a new set of tools for Pharo/Smalltalk that strives to greatly simplify the use of Git and GitHub for Smalltalk projects [28]. But like Monticello, Iceberg currently seems to focus on the versioning of Smalltalk code and has no support for tracking or serializing arbitrary objects.[2] In contrast to Squit, Iceberg uses Pharo's foreign function interface (FFI) to bind to the C library *libgit2* to work with Git repositories. Iceberg also strives for a better integration of GitHub features directly in the Pharo programming environment, which has not been a focus of Squot so far. Like Squot, Iceberg currently only connects to Git, but could handle other VCSs if appropriate connectors would be implemented.

CoVer adds version control to a collaborative hypermedia editing system with asynchronous editing [24]. Within it, Haake and Haake propose a collaboration model based on tasks, reminiscent of the contemporary *Rational Synergy* by IBM.[3]All versions of a tracked object are combined into a *multi-state object* to track the identity of an object. Thus, each object has its own history, which is more similar to ENVY or Mercurial than to Squot, Monticello, or Git. Moreover, Squot does not include any model for collaboration; it only provides the technical means on top of which many different collaboration workflows could be organized—much like Git. While CoVer deals with hypertext documents, the authors state that "any application domain that can be modelled by interrelated objects can be mapped to hypertext" [24, conclusion]. This could mean that hypertext could be used as a general snapshot format (though, it might also incur serialization, lifting the distinction of snapshots from storage

[1] http://www.squeaksource.com/OSProcess.html (last accessed November 5, 2017).
[2] This was concluded by reviewing the Iceberg classes in the Pharo 6 distribution.
[3] http://www.ibm.com/software/products/ratisyne (last accessed November 5, 2017).

objects) or that their approach on hypertext can be transferred easily to other kinds of object graphs.

COOP/Orm is an environment that provides fine-grained version control and configuration management for sets of documents [43] based on a framework for fine-grained version control for software development [44]. Each document is modeled and edited as a tree of information units, which could be classes and methods, or chapters and paragraphs, for example. Versions are created for documents, but the history of individual information units can be inspected as well. Configurations are created with explicit bindings between versions of documents. It even allows multiple versions of the same document to appear in a configuration. The latter is not possible with Squot in a single working copy because only one edition of an object can be checked out at the same time in one working copy—like in most file-based VCSs and ENVY. On the other hand, COOP/Orm imposes to model documents as a tree structure, while the equivalent to documents in Squot are object graphs that, as the name says, need not be trees. Merging is described as a two-level process: first, the node structures in the document object trees must be compared and merged, which can result in conflicts, and second, these conflicts are attempted to be resolved by comparing and merging the data of the conflicting nodes, which can again result in conflicts. In both steps, a set of default rules determines the outcome of merges. The results are presented to users who can either accept or modify the proposed merge. Remaining conflicts must always be resolved by the users. The same can be observed in Git: first, changes in trees are identified by comparing the hashes of tree entries, and second, diff and merge tools are invoked on blobs at the same path. For changes to trees and lines of text, Git has decision tables to merge automatically or escalate conflicts to users.

In general, we observe that the older VCSs put the focus of "revisions" on smaller parts like files, documents, or classes and methods [4, 20, 43, 51, 65, 67] and require higher-level "configurations" to compose particular versions of them. Newer VCSs like Subversion, Monticello, Mercurial, and Git rather have revisions (or versions, change sets, and commits, respectively) [35] that behave more like the configurations of the previous systems, and the contained objects/files can be addressed in terms of these higher-level revisions (e.g., "the file at path $x/y/z$ in revision r"). Monticello is actually somewhere in between because a system is often described in terms of multiple packages, so configurations are still needed on top of versions. The need for additional configurations in the other VCSs primarily arises when a system must be composed with other systems or libraries that are hosted in external repositories, in which case facilities like Metacello [2], Maven[4], npm's package.json[5], or Git submodules are used to describe dependencies. Squot follows Pur's style of versions in this regard, which is equivalent to that of Git, Mercurial, and Subversion.

The term *configuration* is also used to describe which implementation of a particular interface should be used to instantiate a system—most importantly when there

[4]https://maven.apache.org/ (last accessed November 5, 2017).
[5]https://docs.npmjs.com/files/package.json (last accessed November 5, 2017).

are multiple implementations for a single interface. This seems to be the primary meaning employed by Ambriola et al. for their comparison of version control and configuration management generations in 1990 [1].

HistOOry is an object versioning system that can efficiently record the state of selected fields of objects whenever a snapshot is created. It is also meant to be a general-purpose language extension for Squeak/Smalltalk and Pharo/Smalltalk for all applications that might need object versioning [53]. This sounds quite similar to Squot, but there are some differences. HistOOry stores all versions in memory (i.e., in the image) and does not intend to connect to a VCS with external storage. Sharing objects and their versions with other users also seems not to be a goal of HistOOry, while supporting collaboration is an explicit goal of Squot. HistOOry can create views on the snapshots of an object that are polymorphic with the live object, which is not the case for shadows in Squot (it is not impossible to achieve in Squot, but it would need a different implementation of the shadows). Finally, HistOOry has a fairly efficient implementation to detect modifications to arbitrary objects whenever they happen, without requiring modifications to the Smalltalk virtual machine. It involves byte-code patching of accessor methods for versioned instance variables and introduces an indirection to property access via the current Process, which controls which version determines the current state of an object. This could allow for operation-based change tracking and could avoid traversals of whole object graphs to create new snapshots if this approach were combined with Squot.

CoExist proposes continuous versioning as a means to relieve programmers from the risks of changing code [60, 61]. New snapshots are created automatically whenever a program is changed in the Squeak environment by using a tool, such as a system browser. For retrieving a previous state of the environment, a timeline of changes and a versions browser are provided. To achieve good performance and short response times, meta-objects such as classes and compiled methods are stored with each snapshot (i.e., a state-based approach). These instances are shared among snapshots if they were not modified from one version to another. Other versions can be inspected and browsed by opening another environment inside the current one. In the other environment, code browsing tools will display the state of the chosen version and objects will behave like when the chosen version was created. In order to work with classes from multiple versions simultaneously, CoExist works with a modified version of Squeak's virtual machine, in which classes are late-bound in compiled methods. Normally they are bound whenever a method is saved and compiled, but this prevents methods to work with past versions of classes without recompiling the methods. Similarly to HistOOry, versions are kept in-memory and sharing them with other users is not a use case of CoExist. Squot only captures objects on demand (usually when new versions should be created), but also requires more time to do so. Like most Smalltalk VCSs, CoExist focuses on the source code of programs and the definitions of classes. Arbitrary objects are not supported. Sharing and collaboration are not the goals of CoExist, rather it is to provide fast recovery means (undo and redo operations) to encourage changes during exploratory programming. It is meant as a complementary tool next to a VCS, so Squot and CoVer could work well together.

Expanding on *CoExist*, Steinert et al. implemented object versioning for the *Lively Kernel* [62]. In contrast to *CoExist*, arbitrary objects are considered because the direct manipulation of objects is central to Lively, and JavaScript has no classes, like Self. Their approach is to replace references between objects with versioned references. Depending on a context setting, the references resolve to different objects as they did when a version was created. The approach of augmenting references with version information is similar to that of HistOOry. The versioned references in Lively were implemented using ECMAScript 6 proxies. Because this implementation shares the goals of *CoExist*, the same differences to Squot in that regard apply. At the time when [62] was published, the early browser implementations of ECMAScript proxies displayed bad performance, making their ubiquitous use in Lively objects unfavorable.

Lively Kernel includes a *parts bin* of reusable graphical objects (morphs) [40]. Thomschke and Lincke describe that the parts in the bin are versioned using Subversion, but additional tools in Lively have been created to detect and display differences and perform merges [5, pp. 44 ff.][39, pp. 103 ff.]. One motivation to have extra tools, even though Subversion provides such functionality already, is that the morphs are serialized in JSON, which is not easily comparable and mergeable with line-based diff tools. Thomschke and Lincke describe the challenges involved in tracking and merging morphs and their associated object graphs and provide solutions. They identify the need to trace the identity of each morph, so morphs can be recognized even when they are moved in a hierarchy of submorphs, for example. The tracing of morphs is made possible by assigning unique identifiers to them, which is similar to the object names assigned by Squot. Further, they establish that morphs can have multiple representations: (1) visual, (2) JavaScript objects, (3) "linearized" JavaScript objects—which are object graphs with cycles broken up by introducing indirect references to other objects, so the graph can be serialized in JSON—, and (4) the serialized JSON string. In the terms of this report, these correspond to (2) live objects, (3) snapshots, and (4) storage objects. The visual representation is domain specific; it is available because morphs are graphical objects. It is also a filtered view on these objects because not every property of a morph is visible; some cannot be seen. In each of these representations, differences can be computed and merges can be attempted. Some representations are more suitable for the task than others. In Squot these findings resonate 1) by providing the possibility to have domain specific snapshots, 2) by performing change detection and merging on these snapshot objects, and 3) in its difference detection algorithm, which distinguishes objects from references to them, like the linearized form in Lively. Additionally, the serialization format is customizable in Squot, so a format can be chosen that is more suitable for consumption by users or for the merge tools of the backend VCS (and ideally, both), if such a format is available.

WebCards by Dannert is a modern recreation of Apple's *HyperCard* in the *Lively Kernel* [9]. In comparison to the original, it features synchronous collaboration of multiple users on the same card in a stack. Cards are composed of morphs, which

are stored automatically in a *CouchDB*[6] (a non-relational document-based database) when they are first added to a card. Thus, each morph gets a document ID assigned in the CouchDB. Morphs are serialized in JSON after a linearizing preprocessing, as already mentioned above. One core concept of WebCards is that all user actions are recorded in *command objects*.[7] This enables the synchronous collaboration via operational transformations and it provides an undo capability because the commands can be reverted. Because the commands are also stored in the CouchDB, asynchronous collaboration is also possible. In combination with undo, this creates an implicit version control mechanism in based on the command objects. The CouchDB document IDs that are assigned to each morph on a card are similar to the object names assigned by Squot. Difference objects have commonalities with command objects, but Squot's differences do not constitute versions—they are only computed on-demand, since Squot uses a state-based version control approach. In Squot, it is not necessary to create a command for every single trackable operation that can be applied to an object, but for the same reason it does not easily support synchronous collaboration, versions must be created explicitly, and the undoing of operations involves more work (loading a previous version and deselecting all changes that are not related to the operation that should be undone). Creating differences for every user operation is also either inconvenient (when it limits the actions that users may perform) or complex (language extensions like HistOOry are necessary). Generally, state-based storage models in version control systems imply that versions can be loaded in $O(1)$ time, while change-based storage can imply $O(n)$ worst-case time where n is the size of the version history. However, for small histories and sufficiently large object graphs or due to lengthy deserialization procedures, change-based storage can still be faster in practice, which is why Squot tries to efficiently derive differences whenever it can. Git tries to get out the best of both worlds as well with its pack file heuristics: recent versions (which are more likely to be accessed, which is also observed by Dannert) are stored independently for faster access, while older versions of similar blobs are stored with a delta compression scheme to save space [8].

An example of a programming environment that does not primarily involve source code is *Simulink*[8]. It can be used to model systems as hierarchical blocks and also provides simulation capabilities. With *Simulink Project*, integrations with existing version control systems are provided, including Subversion and Git [57, 58]. Tools for resolving conflicts in models (i.e., not text-based content) are also available [46]. While this version control integration is naturally tied to the Simulink product, it shares some characteristics with Squot's architecture: multiple backend VCSs are supported and multiple kinds of objects (non-textual models and textual files) can be tracked and merged within the programming environment. Because Simulink is a proprietary product with undisclosed source code, we cannot compare its version control solution with Squot in more technical detail.

[6] http://couchdb.apache.org/ (last accessed November 5, 2017).
[7] cf. command pattern [16, pp. 233 ff.]
[8] http://www.mathworks.de/products/simulink/ (last accessed November 5, 2017).

6.2 Tracking and transportation of objects

Vegdahl writes about the challenges involved in moving objects between Smalltalk images and the implementation of a solution that supports structures with circular references [71]. The proposed solution is similar to Squeak's ReferenceStream and the object references in STON in that it assigns numbers to objects (the numbers are unique in the stream). All but the first reference to an object are replaced by the number of that object. The described implementation already contains the concept of "unique objects" that should not be serialized. Rather, the corresponding unique object in the target system should be put in its place on deserialization. This is applied to global variables, including classes. Vegdahl also describes the need for special handling of Sets because it would not be useful to serialize their internal (implementation-defined) structure. Like in Squot, the objects to be serialized are themselves in control of the format in which they are written out, although there is no intermediate snapshot representation.

The *Self Transporter* is the primary means to move objects between *Self* images [68]. Because Self is a prototype-based language, there are no classes, so the Transporter is not specific to Self code, unlike most of the Smalltalk version control systems mentioned above. The Transporter captures objects and transforms them into Self expressions that recreate the captured objects when evaluated. Ungar notes in his paper that object graphs are missing some information about the intentions of the programmers who built the objects, and that these intentions must be recovered somehow for a proper transportation of objects from one system to another. In Self, the missing information is stored in annotations to objects and their slots. The Transporter does not provide own facilities for version control. Rather, it is proposed to share the exported files via any file-based VCS. In this regard, it could be described as an advanced file-out and file-in mechanism for arbitrary objects in Self. The insights obtained from developing the Transporter heavily influenced the generic object snapshots in Squot and guided the design of object capturing in general.

Parcels are a deployment mechanism for objects and source code in VisualWorks Smalltalk [47]. It employs a pickling method for fast loading of a parcel and its contained objects into a VisualWorks image. Further, it supports mutual dependencies among parcels, features partial loading of parcels (in case some prerequisite classes are not present in the image when a parcel is loaded), and appropriate teardown when a parcel is unloaded. While there can be multiple versions of a parcel, parcels itself do not provide a version control system—for this, *Store* is used in VisualWorks [7]. Squot is currently not intended to be a deployment facility and focuses on version control instead. However, partial loading is an interesting feature that could be investigated for Squot in the future.

Fuel is a fast binary object serializer for Pharo/Smalltalk and Squeak/Smalltalk [13, 14]. It expands on the fast pickling format of *Parcels*. To make object materialization more efficient, objects are clustered in a separate analysis phase before these objects are actually serialized. As a result, object allocation and materialization can be done iteratively and in bulks. Like any good serializer, Fuel also addresses at least some of the issues of object transportation, such as transient state and global objects. Fast

serialization and deserialization would also benefit Squot, but it must be weighed against the portability and readability of the file format. Fuel streams are generally not intended to be portable. Nevertheless, an adapter to Fuel could be added to Squot as an alternative to the SquotSmartRefStreamSerializer. Fuel's object graph analysis has many common tasks with Squot's capturing. It could be investigated in the future whether there is any potential for code reuse here.

Tanker is a project for Pharo/Smalltalk to export and import classes, traits, and their methods efficiently. Fuel is used for serialization and materialization, to avoid the overhead of compilation source code on each import [64]. Thus, Tanker is similar to Orwell and ENVY in this regard, but Tanker currently does not provide version control. The need for faster imports/materialization could play a role in future extensions of Squot or uses of FileSystem-Git. Monticello version files also include both a binary snapshot and the source text of a package. A similar double storage could be built into Squot, but it must be determined how to balance fast saving and loading with the tool and platform support of GitHub, for example. The latter will have problems with the handling (especially merging) of binary snapshots. It might also be problematic if the binary snapshots get out of synchronization with the primary (maybe textual) snapshots that can be handled by users outside of Squeak if necessary. Moreover, the binary serializations of Fuel and the compiled bytes of methods are not portable across all different Smalltalk dialects and virtual machines. Therefore, binary serializations for faster loading might be more interesting for company- or group-internal software repositories, complementary to repositories that contain publishable and portable serializations.

Khoshafian and Copeland describe different levels of identity support in programming and database systems [34]. They note that Smalltalk has a strong concept of object identity already (with object oriented pointers), but identity is limited temporally and to the running instance of the Smalltalk system. Moreover, object oriented pointers are not stable in Squeak because it uses direct pointers and a compacting garbage collector instead of the object table described for Smalltalk-80, which assigns an integer to each object [22, 30]. For comprehensive object identity support, Khoshafian and Copeland propose to introduce object surrogates that are "system-generated, globally unique identifiers, completely independent of any physical location" [34, p. 413]. The object names that Squot assigns to objects are similar to such surrogates, but they are not assigned to each object unconditionally and not already at the time when an object is created. Instead the names are assigned when objects are captured for the first time and only when it is deemed necessary to assign a name (although objects must opt out from having a name assigned to avoid it). More importantly, several snapshot objects might carry the same name in Squot to denote that they represent the same live object (whether that live object actually exists in the running system or not).

6.3 User interface concepts

Tam and Greenberg describe a "theoretical framework for asynchronous change awareness in collaborative documents and workspaces" [63]. In it, they collect which kind of information should be made available to users to make them aware of each other's changes in a collaborative environment where changes are performed asynchronously—which is the case with Squot and Git. Not much work to address change awareness has been put into Squot, except for the displayability and manipulability of differences in tree form. If more sophisticated tools are designed for Squot and domain-specific snapshots in the future, such a change awareness framework and related work of it could be a valuable guidance.

De Rosso and Jackson have conducted a conceptual design analysis of Git (primarily of its command line user interface) [12] to find out what conveys the wide-spread impression that Git is complicated (which is even reiterated in a presentation for Iceberg[9]) or at least more complicated than other VCSs. They further attempt a redesign of Git, called *Gitless*, to address some of the identified shortcomings [11]. Squot already picks up some of their ideas as described in subsection 4.2.3. Since, in general, object graphs are more complicated than hierarchically structured source code and it can be harder to build good user interfaces about them, every possible step should be taken to avoid unnecessary complexity in Squot's version control user interface and concepts, when it does not mean to reduce its power.

6.4 Object merging

Ignat and Norrie describe and compare operation-based and state-based merging in the case of asynchronous collaborative editing of graphical objects [29]. They conclude that operation-based merging has advantages in solving conflicts that would otherwise require manual resolution by users and that it is also more efficient for large documents (i.e., when the number of objects is high). Nevertheless, Squot basically assumes state-based merging because it fits more naturally with the snapshot-based (instead of change-based) version control model and there is no API in Squeak to capture all operations that are performed on an arbitrary object (though, HistOOry's indirect references could probably be extended for that purpose). If domain objects have the capability to record the trace of operations, they could choose to capture these operations in the snapshots and store them in the backend VCS. They could then use the operations in the snapshots to perform operation-based merging. However, the serialized operations may be unpractical to handle for users (unless both the final state and the operations that lead there are stored), and they may be hard to merge on GitHub where only line-based text merge tools (and manual editing) are available. Despite being state-based, Squot tries to make use of the

[9] https://www.slideshare.net/esug/iceberg-bringing-next-generation-source-versioning-to-pharo, slide 13 (last accessed November 5, 2017).

efficiency advantage of operation-based approaches by computing the differences between snapshots first (which can sometimes avoid to process the whole snapshot, when information from the backend VCS can be utilized) and applying those instead of processing whole snapshots. Thus, different "operations" than originally performed by users are generated, so there is no advantage for merging as in truly operation-based merges. Another difference between Squot and [29] is that in the latter, the versioned (graphical) objects are trees, while Squot has to deal with more general graphs of objects.

Lindholm describes how three-way merges can be performed for XML documents [41]. Even though XML infosets[10]are also trees, and thus only a subset of what Squot should eventually support to merge, this paper and its related work are further sources to consider when a merge implementation for arbitrary objects in Squot will be devised. XML documents are described as "ordered trees with labelled nodes" [41, p. 3]. Labels are already provided by Squot in the form of object names, so the findings about XML merges could be useful when merging object graphs that include collections that have an ordering.

[10]http://www.w3.org/TR/xml-infoset/ (last accessed November 5, 2017).

7 Conclusion and outlook

We set out to devise a solution to the problem that, in exploratory programming environments that are built around objects, regular (non-code) objects are either precluded from specialized version control solutions, or that version control is not integrated with the programming environment because an export and import step separates the version control system from the programming environment. Existing version control technology should be reused to provide insurances for robustness and tool support. Ultimately, version control for objects should be as practical and accessible as it is for files today, or even better.

With the architecture presented in this report, a first step towards that goal has been made. It provides a framework for modeling other editions of objects and comparing them with each other. A generic solution to track arbitrary kinds of objects is provided, so the effort of versioning a new objects is kept low. When a more specialized way to handle different editions of an object is needed, the architecture provides variation points, so that custom software can supply their own formats of snapshots and serializations. The prototype implementation *Squot* proves that the architecture is functional and that the status-quo on object version control could be improved in Squeak/Smalltalk. By building on the ideas of Pur [35], the architecture is not tied too closely to Git to be portable to another backend version control system.

The journey towards a product that makes version control for objects feel as natural as it has become for files has not come to an end yet; there is still room for improvement. To provide a fully-functional version control experience for objects, merging facilities for arbitrary objects must still be implemented. In general, a number of tools needs to be added to Squot to make snapshots of object graphs and differences between them easier to understand. Other open issues are discussed in chapter 5. The prototype could be extended with additional, useful features, such as support for ignoring certain changes to tracked objects when they are captured. This would allow users to keep some changes (e.g., customizations and hacks) private in their own working copies of these objects.

To further validate the applicability and generality of the proposed architecture, an implementation in another exploratory programming environment could be undertaken. While the *Lively Kernel* already supports to manage versions of objects, it could be examined how the solution in this report can be mapped to support and extend Lively's approach to that challenge. To port Squot to a different Smalltalk system would likely be a smaller step; whether or how Squot can help to exchange libraries of objects between different Smalltalks could be investigated. It could also be interesting to see whether additional requirements arise when interoperability is considered or how snapshots and file formats might change to suit that need. More abstract forms of snapshots could be needed, to materialize different kinds objects

depending on the target environment. It could be possible to capture a graphical tool that displays data such that it could be rematerialized in a different environment using a different implementation of windows and widgets. Possible questions regarding the used file formats are: Would the formats tend to be more general and be used for diverse kinds of objects (think of XML, ASN.1,[1] or RDF[2]), or would many domain specific formats emerge?

The availability of a working library to directly manipulate Git repositories from Smalltalk (which is fulfilled by both FileSystem-Git and Pharo's binding to libgit2) paves the way for further ideas on how the two technologies might complement each other. For example, it might be investigated whether Git repositories (or other better-known version control systems) can be used to replace the *changes* file (that accompanies each Squeak image) to an advantage, like ENVY libraries replaced changes files. Another area for experimentation is whether fine-grained version control (as defined in some related work [43, 65], where all objects can have their own history) can somehow be combined with Git's history model and, thus, be made easily usable with tools and platforms like GitHub.

[1] Abstract Syntax Notation One.

[2] Resource Description Framework.

References

[1] V. Ambriola, L. Bendix, and P. Ciancarini. "The evolution of configuration management and version control". In: *Software Engineering Journal* 5.6 (1990), pages 303–310.

[2] A. Bergel, D. Cassou, S. Ducasse, J. Laval, D. Henrichs, and M. M. Peck. "Managing Projects with Metacello". In: *Deep into Pharo*. Switzerland: Square Brackets Associates, Aug. 2013, pages 147–188. ISBN: 978-3-9523341-6-4.

[3] A. Bergel, D. Cassou, S. Ducasse, J. Laval, and M. Leske. "Files with FileSystem". In: *Deep into Pharo*. Switzerland: Square Brackets Associates, Aug. 2013, pages 15–30. ISBN: 978-3-9523341-6-4.

[4] B. Berliner. "CVS II: Parallelizing software development". In: *Proceedings of the USENIX Winter 1990 Technical Conference*. Volume 341. 1990, page 352.

[5] C. Calmez, H. Hesse, B. Siegmund, S. Stamm, A. Thomschke, R. Hirschfeld, D. Ingalls, and J. Lincke. "Explorative authoring of Active Web content in a mobile environment". In: *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam*. 72. Universitätsverlag Potsdam, 2013. ISBN: 978-3-86956-232-2.

[6] S. Chacon and B. Straub. *Pro git*. 2nd edition. Apress, 2014.

[7] *Cincom VisualWorks Store Repository*. URL: http://www.cincomsmalltalk.com/main/developer-community/store-repository/ (visited on 2017-09-02).

[8] *Concerning Git's Packing Heuristics*. URL: https://www.kernel.org/pub/software/scm/git/docs/technical/pack-heuristics.html (visited on 2017-09-09).

[9] J. Dannert. "WebCards-Entwurf und Implementierung eines kollaborativen, graphischen Web-Entwicklungssystems für Endanwender". Master's thesis. Hasso-Plattner-Institut, Universität Potsdam, 2009.

[10] *DataStream*. URL: http://wiki.squeak.org/squeak/4310 (visited on 2017-08-11).

[11] S. P. De Rosso and D. Jackson. "Purposes, concepts, misfits, and a redesign of git". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM. 2016, pages 292–310.

[12] S. P. De Rosso and D. Jackson. "What's wrong with git?: a conceptual design analysis". In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM. 2013, pages 37–52.

[13] M. Dias, M. M. Peck, S. Ducasse, and G. Arévalo. "Clustered serialization with fuel". In: *Proceedings of the International Workshop on Smalltalk Technologies*. ACM. 2011, page 1.

[14] M. Dias, M. M. Peck, S. Ducasse, and G. Arévalo. "Fuel: A fast general purpose object graph serializer". In: *Software: Practice and Experience* 44.4 (2014), pages 433–453.

[15] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. "Traits: A Mechanism for Fine-grained Reuse". In: *ACM Trans. Program. Lang. Syst.* 28.2 (Mar. 2006), pages 331–388. ISSN: 0164-0925. DOI: 10.1145/1119479.1119483.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[17] T. Garnock-Jones. *Git for Squeak*. URL: http://www.squeaksource.com/Git.html (visited on 2017-08-10).

[18] *Git documentation*. URL: https://git-scm.com/docs/ (visited on 2017-08-20).

[19] *Git index format*. URL: https://www.kernel.org/pub/software/scm/git/docs/technical/index-format.html (visited on 2017-08-10).

[20] A. L. Glasser. "The evolution of a source code control system". In: *ACM SIGSOFT Software Engineering Notes* 3.5 (1978), pages 122–125.

[21] A. Goldberg. *Smalltalk-80: The Unteractive Programming Environment*. Edited by M. A. Harrison. 1984. ISBN: 0-201-11372-4.

[22] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[23] T. Goubier. *GitFileTree*. 2014. URL: https://pharoweekly.wordpress.com/2014/02/05/gitfiletree/ (visited on 2017-09-21).

[24] A. Haake and J. M. Haake. "Take CoVer: Exploiting Version Support in Cooperative Systems". In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. CHI '93. Amsterdam, The Netherlands: ACM, 1993, pages 406–413. ISBN: 0-89791-575-5. DOI: 10.1145/169059.169320.

[25] *History of Morphic*. URL: http://wiki.squeak.org/squeak/2139 (visited on 2017-08-27).

[26] A. S. Hornby. *Oxford Advanced Learner's Dictionary of Current English*. Edited by S. Wehmeier, C. McIntosh, and J. Turnbull. Seventh edition. Oxford University Press, 2005.

[27] *IBM Smalltalk User's Guide*. International Business Machines Corporation, 2002.

[28] *Iceberg*. URL: https://github.com/pharo-vcs/iceberg/ (visited on 2017-09-21).

[29] C.-L. Ignat and M. C. Norrie. "Operation-based versus state-based merging in asynchronous graphical collaborative editing". In: *Proc. 6th International Workshop on Collaborative Editing Systems, Chicago*. 2004.

[30] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. "Back to the future: the story of Squeak, a practical Smalltalk written in itself". In: *ACM SIGPLAN Notices*. Volume 32. 10. ACM. 1997, pages 318–326.

[31] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle. "Fabrik: a visual programming environment". In: *ACM SIGPLAN Notices*. Volume 23. 11. ACM. 1988, pages 176–190.

[32] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. "The lively kernel a self-supporting system on a web page". In: *Self-Sustaining Systems*. Springer, 2008, pages 31–50.

[33] A. Kay. *Active Essays*. URL: http://web.archive.org/web/20060710213801/http://www.squeakland.org/whatis/a_essays.html (visited on 2017-09-09).

[34] S. N. Khoshafian and G. P. Copeland. "Object Identity". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '86. Portland, Oregon, USA: ACM, 1986, pages 406–416. ISBN: 0-89791-204-7. DOI: 10.1145/28697.28739.

[35] M. Kleine, R. Hirschfeld, and G. Bracha. "An abstraction for version control systems". In: *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universtität Potsdam*. 54. Universtitätsverlag Potsdam, 2012. ISBN: 978-3-86956-158-5.

[36] M. Leske. *libgit2 bindings for Pharo*. URL: https://theseion.github.io/libgit2-pharo-bindings/ (visited on 2017-08-10).

[37] M. Leske. *libgit2 bindings supersede FileSystem-Git*. URL: http://forum.world.st/Pharo-Git-cons-tp4793645p4793986.html (visited on 2017-08-10).

[38] M. Leske. "The Genesis of Pharo: The Complete History of Pharo with Git". Bachelor's thesis. Bern, Switzerland: Software Composition Group, University of Bern, 2011.

[39] J. Lincke. "Evolving Tools in a Collaborative Self-supporting Development Environment". PhD thesis. Hasso-Plattner-Institut, Universität Potsdam, 2014.

[40] J. Lincke, R. Krahn, D. Ingalls, M. Roder, and R. Hirschfeld. "The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content". In: *System Science (HICSS), 2012 45th Hawaii International Conference on*. IEEE. 2012, pages 693–701.

[41] T. Lindholm. "A three-way merge for XML documents". In: *Proceedings of the 2004 ACM symposium on Document engineering*. ACM. 2004, pages 1–10.

[42] M. Mackall. "Towards a better SCM: Revlog and mercurial". In: *Proc. Ottawa Linux Sympo* 2 (2006), pages 83–90.

[43] B. Magnusson and U. Asklund. "Fine grained version control of configurations in COOP/Orm". In: *Software Configuration Management* (1996), pages 31–48.

[44] B. Magnusson, S. Minör, and U. Asklund. "Fine-Grained Revision Control for Collaborative Software Development". In: *In Proceedings of the 1993 ACM SIGSOFT Conference on Foundations of Software Engineering, Los Angeles CA*. 1993, pages 7–10.

[45] J. Maloney. "Morphic: The self user interface framework". In: *Self 4.0 Release Documentation*. Sun Microsystems Laboratories, 1995.

[46] *Merge Simulink Models from the Comparison Report - MATLAB & Simulink*. URL: https://www.mathworks.com/help/rptgenext/ug/merge-simulink-models-from-the-comparison-report.html (visited on 2017-09-09).

[47] E. Miranda, D. Leibs, and R. Wuyts. "Parcels: a fast and feature-rich binary deployment technology". In: *Computer languages, systems & structures* 31.3 (2005), pages 165–181.

[48] *Monticello*. URL: http://wiki.squeak.org/squeak/1287 (visited on 2017-09-26).

[49] F. Niephaus, D. Henrichs, M. Taeumel, T. Pape, T. Felgentreff, and R. Hirschfeld. "smalltalkCI: A Continuous Integration Framework for Smalltalk Projects". In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*. IWST'16. Prague, Czech Republic: ACM, 2016, 3:1–3:9. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991044.

[50] *PackageInfo: Declarative Categorization of Squeak Code*. URL: http://wiki.squeak.org/squeak/PackageInfo (visited on 2017-08-17).

[51] J. Pelrine, A. Knight, and A. Cho. *Mastering ENVY/Developer*. New York, NY, USA: Cambridge University Press, 2001. ISBN: 0-521-66650-3.

[52] K. Pitman, editor. *Common Lisp HyperSpec*. 2005. URL: http://www.lispworks.com/documentation/HyperSpec/Front/index.htm.

[53] F. Pluquet, S. Langerman, and R. Wuyts. "Executing code in the past: efficient in-memory object graph versioning". In: *ACM SIGPLAN Notices*. Volume 44. 10. ACM. 2009, pages 391–408.

[54] A. Rauschmayer. *Speaking JavaScript: An In-Depth Guide for Programmers*. O'Reilly Media, Inc., 2014.

[55] M. J. Rochkind. "The source code control system". In: *IEEE Transactions on Software Engineering* 4 (1975), pages 364–370.

[56] D. Roundy. "Darcs: distributed version management in haskell". In: *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*. ACM. 2005, pages 1–4.

[57] *Set Up Git Source Control - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/ug/set-up-git-source-control.html (visited on 2017-09-09).

[58] *Set Up SVN Source Control - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/ug/set-up-svn-source-control.html (visited on 2017-09-09).

[59] *Source Tracking System*. URL: http://www.object-arts.com/downloads/docs/index.html?source_tracking_system.htm (visited on 2017-09-02).

[60] B. Steinert. "Built-in recovery support for explorative programming". PhD thesis. Universität Potsdam, 2014.

[61] B. Steinert, D. Cassou, and R. Hirschfeld. "Coexist: overcoming aversion to change". In: *ACM SIGPLAN Notices*. Volume 48. 2. ACM. 2012, pages 107–118.

[62] B. Steinert, L. Thamsen, T. Felgentreff, and R. Hirschfeld. "Object Versioning to Support Recovery Needs: Using Proxies to Preserve Previous Development States in Lively". In: *Proceedings of the 10th ACM Symposium on Dynamic Languages*. DLS '14. Portland, Oregon, USA: ACM, 2014, pages 113–124. ISBN: 978-1-4503-3211-8. DOI: 10.1145/2661088.2661093.

[63] J. Tam and S. Greenberg. "A framework for asynchronous change awareness in collaborative documents and workspaces". In: *International Journal of Human-Computer Studies* 64.7 (2006), pages 583–598.

[64] *Tanker*. URL: https://rmod.inria.fr/web/software/Tanker (visited on 2017-09-02).

[65] D. Thomas and K. Johnson. "Orwell—a configuration management system for team programming". In: *ACM SIGPLAN Notices*. Volume 23. 11. ACM. 1988, pages 135–141.

[66] W. F. Tichy. "Design, implementation, and evaluation of a revision control system". In: *Proceedings of the 6th international conference on Software engineering*. IEEE Computer Society Press. 1982, pages 58–67.

[67] W. F. Tichy. "RCS—a system for version control". In: *Software: Practice and Experience* 15.7 (1985), pages 637–654.

[68] D. Ungar. "Annotating Objects for Transport to Other Worlds". In: *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '95. Austin, Texas, USA: ACM, 1995, pages 73–87. ISBN: 0-89791-703-0. DOI: 10.1145/217838.217845.

[69] D. Ungar and R. B. Smith. "Self: The Power of Simplicity". In: *SIGPLAN Not.* 22.12 (Dec. 1987), pages 227–242. ISSN: 0362-1340. DOI: 10.1145/38807.38828.

[70] S. Van Caekenberghe, S. Ducasse, and J. Fabry. *STON: a Smalltalk Object Notation*. URL: https://ci.inria.fr/pharo-contribution/job/EnterprisePharoBook/lastSuccessfulBuild/artifact/book-result/STON/STON.html (visited on 2017-08-17).

[71] S. R. Vegdahl. "Moving structures between Smalltalk images". In: *ACM SIGPLAN Notices*. Volume 21. 11. ACM. 1986, pages 466–471.

[72] J. Vuletich, K. Dickey, and H. Hirzel. *Managing your code in Cuis*. URL: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/blob/master/Documentation/CodeManagementInCuis.md (visited on 2017-08-27).

[73] J. Vuletich and H. Hirzel. *Using Git and GitHub to host and manage Cuis code*. URL: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev/blob/master/Documentation/CuisAndGitHub.md (visited on 2017-08-27).

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
120	978-3-86956-422-7	**Squimera : a live, Smalltalk-based IDE for dynamic programming languages**	Fabio Niephaus, Tim Felgentreff, Robert Hirschfeld
119	978-3-86956-406-7	**k-Inductive invariant Checking for Graph Transformation Systems**	Johannes Dyck, Holger Giese
118	978-3-86956-405-0	**Probabilistic timed graph transformation systems**	Maria Maximova, Holger Giese, Christian Krause
117	978-3-86956-401-2	**Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud" 2016**	Stefan Klauck, Fabian Maschler, Karsten Tausche
116	978-3-86956-397-8	**Die Cloud für Schulen in Deutschland : Konzept und Pilotierung der Schul-Cloud**	Jan Renz, Catrina Grella, Nils Karn, Christiane Hagedorn, Christoph Meinel
115	978-3-86956-396-1	**Symbolic model generation for graph properties**	Sven Schneider, Leen Lambers, Fernando Orejas
114	978-3-86956-395-4	**Management Digitaler Identitäten : aktueller Status und zukünftige Trends**	Christian Tietz, Chris Pelchen, Christoph Meinel, Maxim Schnjakin
113	978-3-86956-394-7	**Blockchain : Technologie, Funktionen, Einsatzbereiche**	Tatiana Gayvoronskaya, Christoph Meinel, Maxim Schnjakin
112	978-3-86956-391-6	**Automatic verification of behavior preservation at the transformation level for relational model transformation**	Johannes Dyck, Holger Giese, Leen Lambers
111	978-3-86956-390-9	**Proceedings of the 10th Ph.D. retreat of the HPI research school on service-oriented systems engineering**	Christoph Meinel, Hasso Plattner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich, Emmanuel Müller
110	978-3-86956-387-9	**Transmorphic : mapping direct manipulation to source code transformations**	Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, Robert Hirschfeld
109	978-3-86956-386-2	**Software-Fehlerinjektion**	Lena Feinbube, Daniel Richter, Sebastian Gerstenberg, Patrick Siegler, Angelo Haller, Andreas Polze

ISBN 978-3-86956-430-2
ISSN 1613-5652