Meta-models and Infrastructure for Smalltalk Omnipresent History

Verónica Uquillas-Gomez, Stéphane Ducasse, Theo d'Hondt

Meta-models and Infrastructure for Smalltalk Omnipresent History

Verónica Uquillas Gómez [a,b] Stéphane Ducasse [b] Theo D'Hondt [a]

[a] *Software Languages Lab — Vrije Universiteit Brussel — Belgium*

[b] *RMoD Team — INRIA Lille-Nord Europe Research Center*
Laboratoire d'Informatique Fondamentale de Lille — Université Lille1 — France

Abstract

Source code management systems record different versions of code. Tool support can then compute deltas between versions. However there is little out of the box support to be able to perform queries and analysis over the complete history: for example tools have to build their own infrastructure to identify slices of changes and their differences since the beginning of the project. We believe that this is due to the lack of a powerful code meta-model as well as an infrastructure. For example, in Smalltalk often several source code meta-models coexist: the Smalltalk reflective API coexists with the one of the Refactoring engine or distributed versioning system such as Monticello. While having specific meta-models is an adequate engineered solution, it multiplies meta-models and it requires more maintenance efforts (*e.g.,* duplication of tests, transformation between models), and more importantly navigation tool reuse when meta-models do not offer polymorphic APIs. As a first step to solve this problem, this article presents several source code models that could be used to support several activities and proposes an unified and layered approach to be the foundation for building an infrastructure for omnipresent version browsing.

Key words: Meta-model, versioning, refactoring, Monticello, Git, SVN

1. Introduction

Source code management systems such as SVN or Git record different versions of code. Such source code is then processed by tools for detecting changes between versions and

Email addresses: vuquilla@vub.ac.be (Verónica Uquillas Gómez), stephane.ducasse@inria.fr (Stéphane Ducasse), tjdhondt@vub.ac.be (Theo D'Hondt).

providing conflict analysis as well support elementary merging. Currently, there is little support out of the box to be able to perform queries and analysis over the complete history: Tools have to build their own infrastructure, for example, to compare all the differences between all the senders of a given method in the past is not straightforward. To ease version history analysis we need adequate models that deal with changes, refactorings, versions, merging, and history in general. Now naturally the questions of their definition, the abstractions they use, and the APIs of such models are raised.

For example, in Smalltalk, several source code meta-models coexist in a weakly causal connected way [Mae87]: the Smalltalk reflective API coexists with the one of the Refactoring engine or distributed versioning systems such as Monticello. While having specific meta-models is an adequate engineered solution when developers want to abstract over different systems and be independent of idiosyncrasies of the underlying execution platform, in reality it multiplies the number of abstractions, it increases maintenance efforts and lower tool reuse when in presence of non-polymorphic APIs. We call this problem *the meta-models plague*, that is when multiple meta-models have different APIs which make difficult the conversion between them, maintenance by propagation and test assessment. This proliferation of meta-models puts the burden on the developer that has to maintain consistent models across tools. Resulting in many task duplications (*e.g.,* tests) for complying with the different APIs.

We believe that this is due to the lack of a powerful source code meta-model which could be extended and be the glue between several models as well as an adequate infrastructure [vdHL96]. As a first step to solve the mentioned problems, this article presents several source code models that could be used to support several activities and be the foundation to build an infrastructure for omnipresent version browsing. We stress the difference between the meta-models and their role. We present an unified meta-model, the *Ring*, that can be used for several purposes.

The contributions of this paper are: (1) comparison of SCM systems with a focus on the meta-model of the objects they manipulate, (2) comparison of code meta-models, and (3) proposal to merge those models in a unified and foundational model infrastructure, named Ring.

In the subsequent Section 2, we first describe some requirements for typical scenarios software engineers face. Then in Section 3 we present the meta-models of some key versioning systems. Section 4 describes the source code meta-models of well known Smalltalk projects. In Section 5 we introduce and sketch our proposed meta-model, followed by the discussion of several open questions about its infrastructure. Finally, we conclude this paper in Section 6.

2. Requirements for Source Code and History Modeling

The source code and its history are an invaluable resource for software engineers, developers and integrators that often need to analyze and understand the evolution of a system before performing actual maintenance or integration tasks. Different kinds of questions and actions should be supported by the tools and the underlying meta-models.

2.1. *Supporting Software Engineers*

By reasoning over the role of certain code entities in previous versions of the system, developers can better understand their current state, assess the required maintenance and avoid making the same mistakes over and over again. In the same way, integrators can speed the understanding of the changes and take better decisions of the integration process itself.

Based on our own experience, we present a list of specific questions that usually arise when analyzing the software evolution of a system (linear history) or when comparing forks of related systems (cross history). In addition, these questions are supported and extended by a recent work [FM10] that provides a list of 35 questions related to changes to the source code that developers commonly ask during maintenance tasks.

- *Co-Change analysis* [YMNCC04, ZWDZ04, XS04, LZ05, GJK03]: What are the entities that changed together with entity *Number* in *version 3*? Did the same entities change together in *version 4*? if not, what were the missing changes?
- *Queries as in the past*: What were the senders of the method *#asString* in *Squeak 3.9*?
- *Queries as in the present*: What are the senders of method *#readStream* in *version 3.1*?. What are the messages sent by method *#printOn:* in *version 2*?
- *Global analysis*: What is the whole history of method *#detect:ifNone:*? [FM10]. Was the method *#directoryNamed:* rename in the past? if yes, in which version? and which was its previous name?. What is the most queried history entity?. What is the meta-data that the history should keep?
- *Bug spot*: Was this method regularly changed over the last 15 years?
- *Forks analysis*: If the version of method *#isNil* changed in *Squeak 3.9*, should it be changed in *Pharo*?
- *Comparison profiler*: load one version and run it, load another version and run it. What are the differences or similarities of both running versions?

Our questions and the ones defined in [FM10] reinforce our claim that developers lack support for easily retrieving that information. We intend to take those questions into account when building our meta-model and infrastructure.

2.2. *Constraints*

The meta-model and infrastructure that we intend to build have a set of constraints that emerge from reuse and practical integration with the host environment, *i.e.,* Pharo [1] environment. We motivate the most important ones.

- *No layering and duplication of meta-models.* We do not want one runtime meta-model and one for the changes and versioning system. Having different meta-models is costly to maintain, to test, and to keep them in sync. Our goal is to defined a common core meta-model that can be extended for specific tasks. This may be at the cost of having some parts of the objects not used for certain scenario(s). To solve this problem, the meta-model should be able to be annotated with any additional information that is not handle beforehand.

[1] Pharo: http://www.pharo-project.org

– *Model update as cheap as possible.* Updating models is also a problem since desynchronization of the represented information may lead to subtle bugs. In addition since Smalltalk has its own reflective meta-model that is used by the runtime system [BDN+09] and it is causally connected (by causally connected we mean that the model reflects its subject in any circumstances [Mae87]). Therefore, the additional model should use the causal connection as much as possible. Note that taking such advantage is only possible between the current running runtime model and a past version. For history analysis between two versions in the past, it is not possible to use the causally connected reflective behavior of Smalltalk since we do not compare it with another model.
– *Tool reusability relying on common APIs.* Currently, it is common that new tools define their own meta-model with non-polymorphic API for representing entities. This hampers the reuse of tools manipulating entities, as their API might differ from each other. Having a common meta-model will ease the integration and reusability of those tools.

3. Source Versioning Systems

Versioning systems often store and manipulate the source code text without domain semantics. The relation between the source code meta-model and the versioning system meta-model might not be explicit. That means, the source code meta-model keeps each entity as a first-class object, while the versioning system meta-model often stores files of source code (containing different entities). The mismatch between the model used by the tools (like package class browsers in Eclipse) and the underlying versioning meta-model leads to extra efforts to connect two different abstractions. Having this gap is already the origin for having different APIs and transformations between those two models. Note that some versioning systems such as recently Monticello [BDN+09] but also Envy [TJ88, PK01] (in the past) manipulate classes and methods. In addition, a versioning system supports merging algorithms (*e.g.,* 3-way merging) and changes, that in turn interact with the source code meta-model. Resulting in other transformations of models and duplication of efforts in keeping them in sync.

While some specialized versioning systems manipulate a source code meta-model, not all of them do so. For example, Git or SVN can be used to version textual information and not only compiled code. Since meta-models do not necessarily keep a direct connection between the stored meta-model and the actual source code, this results in a need to understand versioning models and their links to the actual source code. For example using Git to directly manipulate methods/classes implies to build an extra infrastructure as it stores objects with a chunk of binary data representing the files that contain the definitions of method/classes, but not those entities independently.

3.1. *SVN*

Subversion [2] [CSFP09] is a centralized system for sharing information. At its core is a repository, which stores data centrally. The repository stores information in the form

[2] http://subversion.apache.org

of a filesystem tree –a typical hierarchy of files and directories. Any number of clients connect to the repository, and then read from or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.

The Subversion repository remembers every change ever written to it –every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.

Subversion supports two strategies for enabling collaborative editing and sharing of data, or in other words two versioning models.

- The Lock-Modify-Unlock model. This model is common in version control systems which address the problem of many authors overwriting each other's work. It allows only one person to change a certain file at a certain time. This exclusivity policy is managed using locks. A developer must *lock* a file before making changes to it. Once the lock is set other developers cannot lock it anymore, they can only read it, hence avoiding simultaneous changes.
- The Copy-Modify-Merge model. This model is used by several version control systems and is an alternative to locking. The idea is that each user creates a personal working copy (*i.e.,* a local duplication of the repository's files and directories). They can work independently on their private copies. As the end, those independent changes are merged together into a new, final version stored in the repository.

The disadvantage of the lock-modify-unlock model lies on the fact that files might be locked unnecessary when changes to the same file are not overlapping. Locked files need to be unlocked to allow other users to work on them. When unlocking files is procrastinated, other users' time might be wasted. Locking and changing interdependent files at the same time may result in semantically incompatible files.

In the copy-modify-merge model, the merging may need human intervention if changes overlap (creating *conflicts*). This might easily be solved without demanding a lot of time. The disadvantage of this model is that it is based on the assumption that files are contextually mergeable, *i.e.,* changed files are line-based text files which can be easily merged, excluding binary files where merging conflicting changes is often impossible.

Even though the lock-modify-unlock model is considered generally harmful to collaboration, it is still more appropriate when working with binary formats.

3.2. *Git*

Git [3] is a distributed revision control system with an emphasis on speed. The core of Git is composed of a collection of tools that implement a tree history storage and directory content management system [Cha08].

Git differs from most SCM systems (*e.g.,* Subversion, CVS, Perforce, Mercurial) on the way it stores data. Those SCM systems store the code deltas or diffs between one commit and the next when creating a new version of a project. Instead, when Git stores a new version of a project, it stores a snapshot of all the files in that project at a point in time. The snapshot is stored as a new tree – a bunch of blobs of content and a collection of pointers with which a full directory of files and subdirectories can be recreated. In Git, a

[3] http://git-scm.com

diff between two versions is calculated by running a new diff on the two trees representing those versions.

Git defines *objects* which represent the actual data. There are four main immutable object types that are stored in the Git *Object Database*, which in turn is kept in the Git *Directory*. Each object is referenced by the SHA-1 value of its content plus a small header.

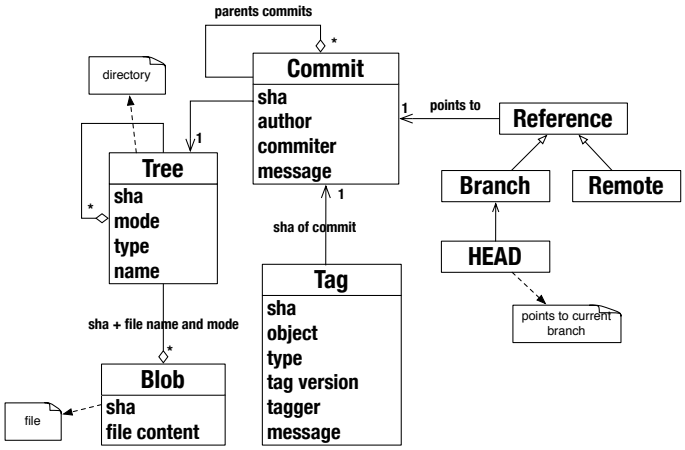

Fig. 1. The Git data model.

- **Blob**. The content of each file is stored as a *blob*. The files themselves –names and modes– are not stored within the blobs, just their content. Differently named files with the same contents will only store one blob and share it. Therefore, during repository transfers (*i.e.,* clones or fetches) only one blob will be transferred, then expanded it into multiple files upon checkout. The blob is totally independent of its location in the directory tree, and renaming a file does not change the blob that such file is associated with.
- **Tree**. The physical directories map to *trees*. A tree is a simple list of pointers to blobs and other trees, along with the names and modes of those trees and blobs. The content section of a tree object consists of a very simple text file that lists the mode, type, name and SHA-1 of each entry.
- **Commit**. The tree history is managed by the *commit* objects. A commit is similar to a tree. It points to a tree (representing the contents of a directory at a certain point in time) and keeps an author, committer, message and any parent commits.
- **Tag**. Commits can be referred to by *tags*, *i.e.,* permanent shorthand names. A tag contains an object, type, tag version, tagger and a message. Normally the type is commit and the object is the SHA-1 of the commit that is being tagged.

In addition to the immutable objects, mutable *references* also stored in Git as well. A reference is a pointer to a particular commit, similar to a tag, but easily moveable. References are used for controlling branches and remotes.

- Branch. A branch is just a file that contains the SHA-1 of the most recent commit for that branch.
- Remote. A remote is basically a pointer to a branch in another person's copy of the same repository (*e.g.,* by cloning a repository).

Figure 1 shows the data model with the objects and references stored in Git. Note an extra element, the HEAD file that points to the branch you are currently working on, and that is used as the parent of the next commit.

3.3. *Monticello 1*

Monticello 1 [4] is a distributed concurrent versioning system for some Smalltalk dialects such as Pharo, Squeak, GemStone and Cincom Smalltalk, in which classes and methods, rather than lines of text, are the units of change [BDN+09]. Monticello 1 (a.k.a. *MC1*) is organized around *snapshots* of a package, that are stored as *versions*. Snapshots are a declarative model of the Smalltalk code that makes up a package, which is composed of classes and methods, organized in various ways, and with dependencies.

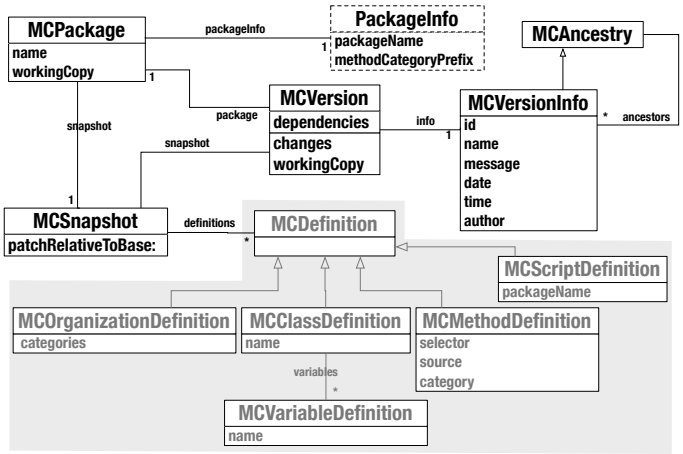

Fig. 2. Monticello 1 key classes for versioning (in grey the source code entities).

In Figure 2, we present an overview of the Monticello 1 models (*i.e.,* versioning model and source code model). The main entities of the versioning model are packages, snapshots, and versions. In addition, this model relies on an external packaging system, usually PackageInfo.

- Packages. A *package* is the unit of versioning. The classes and methods contained in a package are recorded and versioned together in a snapshot.
- Snapshots. A *snapshot* is the state of a package at a particular point in time. It includes definitions of classes, methods, variables, traits and categories.
- Versions. A *version* is a snapshot of a package. It also stores associated metadata as VersionInfo and the version's ancestry. Versions are often stored as mcz files, and represent the standard data used by the system.

In summary, MC1 records a series of snapshots of the code corresponding to a package as it evolves, as well as the ancestral relationships between snapshots. When loading a snapshot into an image, MC1 locates the differences between this snapshot and the state of its package in the image, and then makes the necessary changes to the image so that it

[4] http://www.wiresong.ca/monticello/v1

matches the snapshot. It uses the ancestry of snapshots to provide a merge operation, so that conflicts between two sets of changes can be detected, and non-conflicting changes can be applied automatically.

The source code model of Monticello 1 basically consists of definitions representing the elements of a program. It is connected to the versioning model through versions and snapshots. In Section 4.5 we provide detailed information about it and we show a more complete illustration in Figure 8.

In spite of the presented benefits, the versioning model that Monticello 1 uses, is based on the assumption that packages are well-defined and have relatively stable boundaries (*e.g.,* packages are not expected to be removed or renamed, or their classes will not move to other packages), which is not always the case. In addition, Monticello 1 limits the history to the level of packages and not to the level of independent entities. These issues are addressed by Monticello 2.

3.4. *Monticello 2*

Monticello 2 [5] (a.k.a. *MC2*) addresses the main problem encountered with Monticello 1, which is its unit of versioning –the package– that is too coarse for many situations that arise in normal development (*i.e.,* changes may only impact few methods or classes, but still the whole package needs to be versioned).

In Monticello 2, a new versioning model has been incorporated. It does not have packages as the fundamental unit of versioning. Instead, the unit of versioning is individual program elements – classes, methods, instance variables, etc. This means that Monticello 2 can be used to version arbitrary snippets of code. These might correspond to packages, change sets, or any other method a programmer chooses to separate "interesting" code from the rest of the image.

Rather than maintaining the version history of packages, Monticello 2 keeps version history for each element. Having such history allows users to perform tasks that are not possible with Monticello 1 (*e.g.,* access the whole history of a particular method). With this model package boundaries are not a restriction anymore. Packages can be created, renamed or destroyed, elements can move back and forth between packages, elements can even belong to more than one package at a time. Since the version history is attached to the element, it is not affected.

As shown in Figure 3, the ancestry information is now linked at an entity-based level and not at the level of the package as in Monticello 1.

– Elements. These are representations of specific parts of the program (*e.g.,* classes, methods, variables).
– Variants. A *variant* describes the state of a particular element. RemovelVariant defines the state of elements that might be not present in the image. DefinitionVariant defines a set of properties of an element.
– Versions. A *version* represents the state of an element at a particular point in time. A version associates a variant of an element with the ancestry of that element (*i.e.,* set of versions that precede this version), and it is identified by a hashstamp.
– Hashstamps. A *hashstamp* is a unique identifier given to each version.

[5] http://www.wiresong.ca/monticello/v2

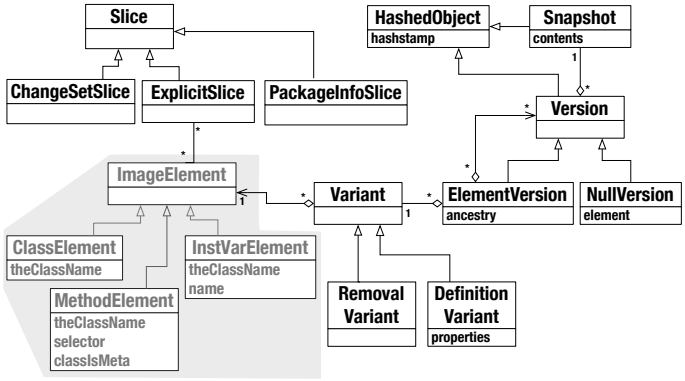

Fig. 3. Monticello 2 key classes for versioning (in grey few source code entities).

– Slice. A *slice* groups elements together. They are responsible for defining which elements are part of the slice and which are not. Slices are independent and can overlap. Elements can belong to many slices at the same time or to none. Different types of slices are supported: PackageInfoSlice for elements defined in a given package, ChangeSetSlice for elements associated with a given ChangeSet, ExplicitSlice for a particular collection of elements, and UnionSlice for elements included in one or more slices.
– Snapshot. A *snapshot* captures the state of a slice. Snapshots record the versions' hashstamps of the slice's elements. It is equivalent to a version in Monticello 1, and it is the unit at which code is moved between images.

The repositories are similar to those in Monticello 1, but have different protocols and performance characteristics. Several types of repositories are defined: *memory* stores objects in the image, *file* stores objects in a file, and *directory* stores objects in several files within a directory.

Another important consequence of the new versioning model (element-based version history) is that merges can be performed on individual elements. Although Monticello 1 supports cherry-picking, it does so in an awkward and non-intuitive way. In Monticello 2, cherry-picking is the norm, and merging an entire package is just a special case.

4. Source Code Meta-Models

While versioning focuses on how to version and merge between versions, it is important to look at source code models. If we take for example Smalltalk, there are several source code meta-models with different purposes (for managing changes, refactorings, merges and versions) that manipulate in some way the Smalltalk source code model. Most of the time such meta-models are overlapping or included in each other. This overlap often exists for a good reason: for example the Refactoring Engine was developed in VisualWorks and should work on any other Smalltalk dialects, therefore the authors preferred to extract and build their own representation instead of extending the existing one. A similar concern exists for Monticello.

Another important concern that we should pay attention to is that Smalltalk is a reflective language [Riv96, Duc99]. This means that it has a causally connected representation of itself [Mae87]. Such causal connection between the model of Smalltalk and its

execution is a really powerful mechanism that supports tool building. Now, when new models are populated to represent views of the Smalltalk runtime, the question of the causal connection is key: should tool builders recreate the model each time the runtime changes? How do they maintain consistency across models? For example in the Moose software analysis platform a model is created for a version or the current code, but if such code changes the model needs to be recreated. Moose keeps immutable models as it focuses on being able to manipulate source code written in different languages; Smalltalk being one among others (Java, C, C++) [NDG05]. But since Moose is implemented in Smalltalk, it could be possible that for a single version analysis we could use the casual connection to the actual source code, and avoid recreating the model when changes happen.

We start studying FAMIX the metamodel of Moose since its goal is to capture the code structure of different object-oriented languages.

4.1. *FAMIX*

FAMIX 3.0 [6] is a family of meta-models oriented towards enabling software analysis. These models were developed in the context of the Moose [7] analysis platform [NDG05]. The meta-models are implemented in Smalltalk, and provide a rich API that can be used for querying and navigating. The core of FAMIX [DTD01] is a language independent meta-model that provides a generic representation of the static structure of programs written in multiple object-oriented and procedural programming languages, such as Smalltalk, Java, C, and C++.

The core meta-model consists of a set of classes that represent source code at the program entity level. Such classes map onto the different elements in a program (*e.g.,* classes, methods, attributes, comments), and of the associations between these elements (*i.e.,* inheritance definitions, invocations of methods, accesses to attributes by methods, references to classes by methods). Figure 4 shows the FAMIX-Core meta-model. While the meta-model is fairly complete, it can be easily extended in order to incorporate other language extensions.

Key points. There are two important points in the design of FAMIX that are worth stressing.

First, FAMIX does not only represent structural source code entities such as *packages, classes, methods* but also it represents explicitly information that is extracted from the methods' abstract syntax trees and attached to the correct semantic level: a class *refers* to another class (Reference), a method *accesses* attributes (Access) and a method *invokes* another one (Invocation). This way FAMIX offers a finer-grained representation of a program than a simpler meta-model and it does so in a language independent manner. Fact extractors which by definition have the knowledge of the targeted language, then produce such a language independent information in terms of FAMIX models. The second point is the decoupling between Package and Namespace. Namespaces are scoping entities and packages deployment entities. This decoupling makes sure that FAMIX can model any kind of situations at the package level.

[6] http://www.themoosebook.org/book/internals/famix
[7] http://www.moosetechnology.org

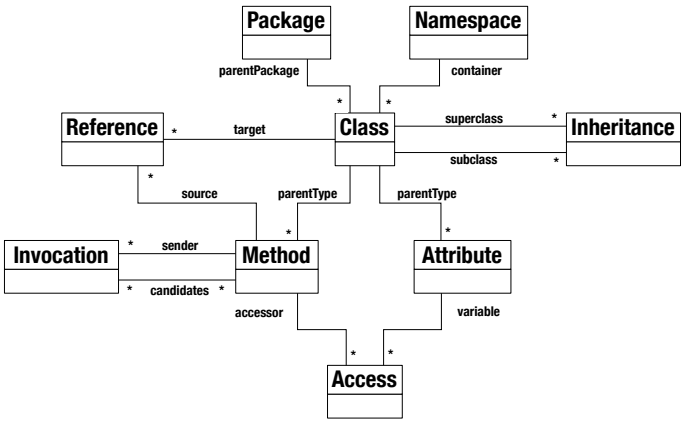

Fig. 4. An overview of the FAMIX-Core language independent meta-model.

4.2. *Refactoring Browser*

The Refactoring Browser [8] – known as *RB* [BR98, RBJ97, Rob99] – is a powerful Smalltalk browser which enables developers to perform several automated refactorings on Smalltalk programs. The refactorings can be classified into three kinds: class, method, and code-based refactorings. RB also offers other productivity enhancements for programmers: Smalltalk Code Critics, a tool that analyzes code for detecting bugs or possible errors; and the Rewrite tool for expressing the rewriting of code through recognition of expressions (pattern matching) on ASTs.

RB defines different models, each having a particular purpose. The following three models are the main ones. The *refactoring model* represents specific refactoring operations. The *changes model* represents changes associated with refactorings. The *source code model*, which is relevant for our study maps the program elements to entities which are manipulated for the rest of RB models. In addition, the RB source code model models a delta on the current system and it is supposed to be polymorphic with the Smalltalk meta-model.

The complete source code model is shown in Figure 5. Note that, it is a very simple model which is only mapping classes, methods and namespaces. Other elements, such as variables are not being modeled as first-class objects. A namespace is associated with an environment (BrowserEnvironment) which provides an API for the browsers and queries.

4.3. *Smalltalk Runtime and Structure Meta-model*

Smalltalk itself defines a meta-model for representing entities at structural and runtime level [GR89]. An excerpt of this meta-model extracted from Pharo is shown in Figure 6. The main root class in Smalltalk is Object which defines common behavior for the rest of the classes. Classes and metaclasses derive from ClassDescription where instances variables are maintained in an array. Classes' methods are kept in a suitable form for interpretation by the virtual machine (*i.e.,* instances of CompiledMethod) and contained in a dictionary

[8] http://www.refactory.com/RefactoringBrowser

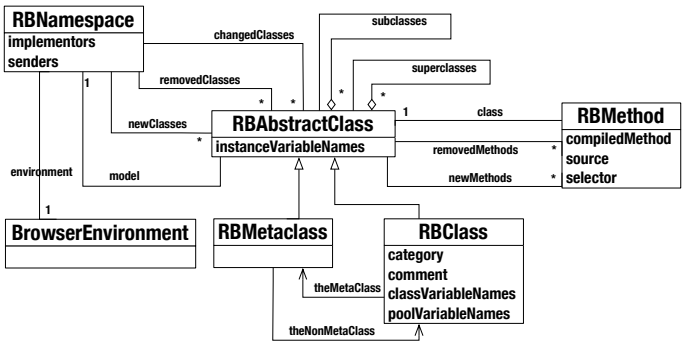

Fig. 5. RB declarative source code model.

(methodDict). Classes are organized in categories, or what is commonly known as packages. The class categories are kept in SystemOrganization, an instance of SystemOrganizer. The protocols of a class are managed by ClassOrganizer. Finally, any entity knows the environment (*i.e.,* namespace) in which it is visible, such environment is unique and is represented by an instance of SystemDictionary.

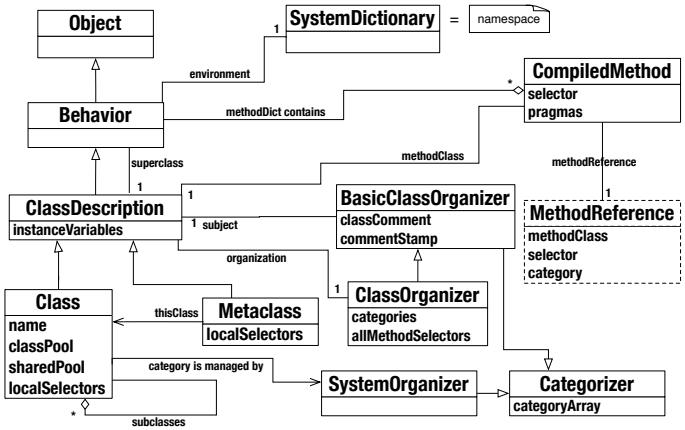

Fig. 6. Smalltalk (Pharo) core model (with dashed border an attempt to add a representational object for CompiledMethod).

Key points. There are several points to stress about the Smalltalk model. First, the model is causally connected with its execution. Therefore there is no problem related to the synchronization of the model when a runtime entity changes. Second, the meta-model is really influenced by the information mandatory for the language execution. For example, instance variables are not first-class objects but just strings. This is a problem when we need to map meta-models targeted to program representations or versioning. Finally, Figure 6 shows the class MethodReference that can be considered as a hack to support a representation of compiled methods. This hack was needed to support tools browsing different versions of a method. Originally MethodReference was not polymorphic with the static API of a CompiledMethod. This resulted in tools duplication.

4.4. *Ginsu*

Ginsu [9] is a cross-dialect semantic model and toolkit for partitioning Smalltalk code into packages. Each package should have a clearly defined scope and prerequisite structure. One of the goals of Ginsu is to be able to build analyses about code that is currently not executing or living in a Smalltalk runtime image [BDN+09]. This goal is similar to the one of FAMIX but without the language independent aspect and with a stronger focus on Smalltalk.

In particular, Ginsu maps the elements defined in Smalltalk code to semantic objects. A semantic object represents the semantic of a Smalltalk program. Semantic objects (SemanticObject) are categorized as modules or components (subclasses of Module and ModuleComponent). Packages are mapped to modules and the rest of the elements (*e.g.,* classes, methods, variables, etc.) to components. A particular definition (such as: ClassDefinition, InstanceMethodDefinition, ClassVariableDefinition, etc.) exists for each kind of component.

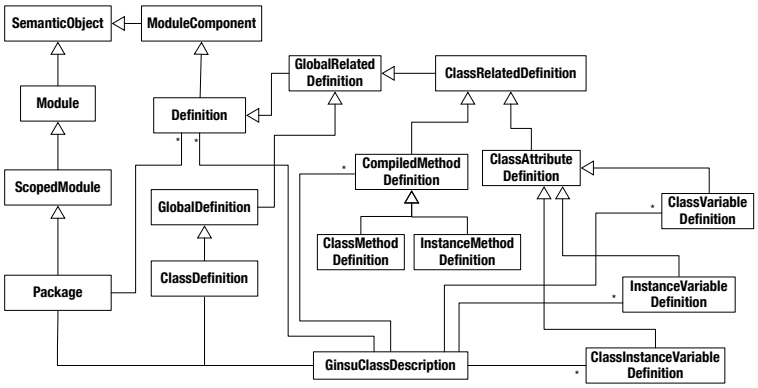

Fig. 7. Ginsu key classes.

The key classes defined in the semantic model are shown in Figure 7. Note that a package contains a set of definitions, the key idea of Ginsu. An interesting property of Ginsu is the ability to annotate any semantic object. Annotations are easily maintained in a dictionary attached to each semantic object. In addition, the model defines the GinsuClassDescription which is associated with a class definition, a set of definitions, and a package. The Ginsu browsers (*i.e.,* PackageSystem and PackageSupport) interact with class descriptions.

Another nice property of Ginsu is that when a semantic object is built for an entity that currently exists in the runtime, Ginsu delegates queries to the living entity. This approach tries to get as much as possible out off the natural causal connection of the underlying model.

4.5. *Monticello 1 and 2*

This section presents the underlying source code meta-model of Monticello 1 and 2. We show the key entities of the meta-model of Monticello 1 in Figure 8. Note that the

[9] http://sourceforge.net/projects/ginsu

versioning model is displayed in grey. This allows us to present the link between both models.

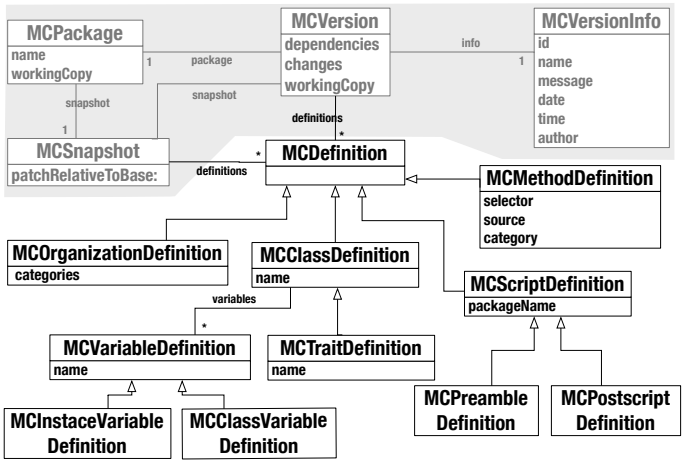

Fig. 8. Monticello 1 key classes for source entities (in grey the versioning entities).

A source code entity *definition* represents an element of the program (*i.e.,* class, method, variable, trait, category, script). The source code model required by MC1 is simple: Package (as PackageInfo – an external class), ClassDefinition maps a class contained in a package, OrganizationCategory maps the categories names in which classes are defined (*i.e.,* packages names). Subclasses of VariableDefinition represent variables of classes, and they are accessed by class references. MethodDefinition maps structural data of methods (selector, source code). Finally, script definitions represent the pre- and post-conditions required by packages.

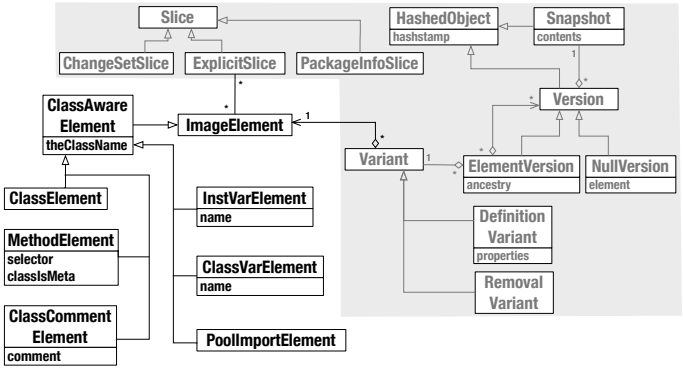

Fig. 9. Monticello 2 key classes for source entities (in grey the versioning entities).

Monticello 2 defines a new meta-model that aims at overcoming some limitations encountered while using Monticello 1. In particular, the unit of versioning of MC2 is not limited to only packages but also to any individual elements (*i.e.,* classes, methods, variables, comments). Figure 9 presents the key classes defined in the source code model of MC2. It also shows the link of such elements to the versioning model, which appear in

grey. Note that the versioning model accesses the source code elements through slices and variants.

The main source code entities in MC2 which map to program elements derive from ImageElement. In MC2, an *element* is more fine-grained than a *definition* in MC1, for example a comment is also represented as an element (ClassCommentElement). Elements are mostly related to a class and thus are defined as subclasses of ClassAwareElement. Class elements (*e.g.,* variables) can be referred to directly, rather than by implication of the class reference.

5. Towards a Unifying and Foundational Model Infrastructure

Section 2 shows that lot of questions engineers face are about history and that history is linked to source code meta models. In this paper we stressed the importance of getting an omnipresent history meta-model. We will work on such a meta-model called the *Ring* meta-model.

5.1. *The Ring meta-model*

One key idea behind the Ring is to define a meta-model and infrastructure that provides a common API at structural and runtime level and allows existing and new tools to interact and integrate directly with the host environment, *i.e.,* Pharo. The second key idea, is that the Ring meta-model should become the foundational model in Pharo, and thus other models should use, refer or extend it. In particular, we focus on the merging model and the versioning model of Monticello as well as the change model and the refactoring model of RB as clients of the Ring meta-model. Currently, those models work mostly independently of each other. In addition, they often define non-polymorphic APIs which makes working on maintenance tasks cumbersome. The RB source code model is the only polymorphic model with the Smalltalk meta-model but this implies duplication of information (*e.g.,* different names, tests).

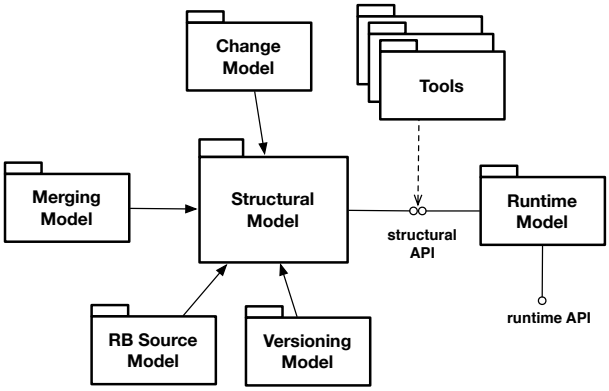

Fig. 10. The Ring overview.

Figure 10 shows our proposal for the Ring meta-model and infrastructure, and how it should interact with other components and tools. Note that the structural and runtime

models share a common API which can be referred by basic tools. This will ease the reuse of such tools. In addition, the rest of the main models (*i.e.,* change, RB, versioning and merging models) should use and extend the structural model.

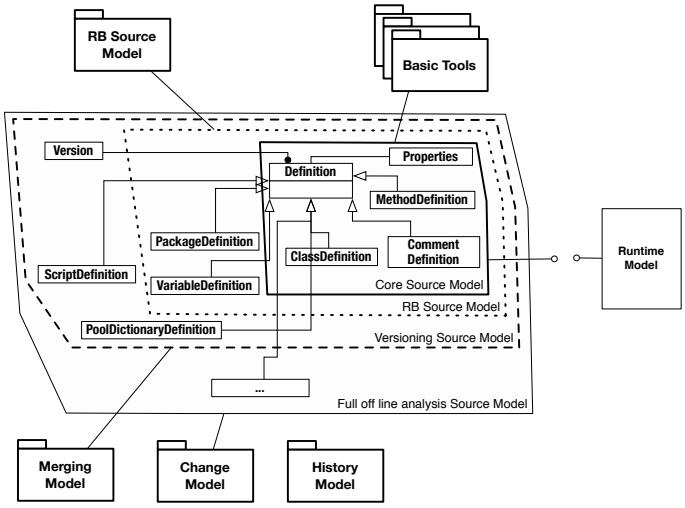

Fig. 11. The Onion structure of the Ring.

The Ring meta-model will not be defined as one big meta-model, the goal is to divide it in layers as shown by Figure 11. These layers can facilitate the reuse and integration with other tools and models. We envision to define a *core source code* model, a *history* model and a *merging* model. The core source code model will only know about the few entities in the system, such as classes, methods, and comments.

Note that up until now we did not really talk about the infrastructure we want to put in place. In addition to have a layered meta-model that different tools can extend. We want the largest version of the Ring meta-model to be used to store all the versions of all the entities of the system to support advance queries and merging algorithms. We want to offer such storage as a web service.

5.2. *Open Questions*

The goals of the Ring meta-model and infrastructure are clear. However, there are several open questions that we need to consider, answer and evaluate before we start building it, as they will impact its design and architecture.

Expensive Queries: How do we manage queries like *find all the senders* in a given version in the past and at which cost (memory and speed)? Indeed, if we only store source code entities like in Ginsu, it may be heavy to compute queries like finding the senders of a particular message which requires to build a representation of the complete AST for the whole system. The question of storing intra-methods elements (*i.e.,* AST nodes) or like in FAMIX (*i.e.,* invocations, references, accesses) has to be assessed in terms of memory and speed of each of the solutions.

Version ids: identifying a version (or group of elements committed together) and its elements sounds trivial, however, we encounter the case where the elements of a version are not aware of their *id*, and they are looked up by matching the commit comment and the commit timestamp with the ones of the versions which have been committed previously or at the same time. Performing such a search may not be efficient especially if we want to provide a flexible querying infrastructure. At the same time, if any element attached to a version is aware of the version id, we should be careful defining the format of *ids* since they may have to be distinct over multiple repositories.

Annotations: Being able to annotate entities makes a model extensible. This is one of the advantages of the Ginsu and FAMIX meta-models. However, they allow annotations in different ways and for different purposes. In Ginsu, for example, any entity defines annotations by means of a dictionary, which allows one to annotate (*i.e.,* add properties) even at runtime. The situation is the same in Moose. In addition, when an undefined property is loaded from a file, it is automatically added to the dictionary of properties. This ensures that extended properties are not lost when loading or saving a model. Our meta-model has to have the same mechanism to support its extensibility.

Meta-model vs. Database schema: We want to define a meta-model that is more than what is stored. This means, we want to define and store entities that are able to produce more data by computation. For example, a slice of changes is computed as diff between two versions. On the other hand, keeping pre-processed data will definitely speed up the querying of data, especially if we take into account that our histories may considerably grow. But if we are able to reconstruct such information, then several questions arise. The answers to the following questions are not clear to us and we will have to define scenarios to assess them: Should we keep all the information in the history? Is the speed more important in our infrastructure? Which are the entities that should be part of the source code meta-model?. One alternative could be to store data that will be frequently searched (*e.g.,* deltas). Another alternative could be to only store computed data of histories (in particular heavy data for searching such as senders and references), and to process such information on demand for the current implementation. In any case, we have to ensure that tools should be able to manipulate all that information.

Core code model API: We intend to encourage tool reusability by relying on a common API of the main entities (*i.e.,* classes, methods, comments) which basic tools may refer to. This avoids having non-polymorphic APIs for representing entities among different tools. Related to the API the question that arise is: Are we considering all the definitions that external tools may need? A typical problem is related to instance variables. Indeed instance variables are not first-class entities in the Smalltalk reflective API even though they are important information for a number of tools. Bridging both worlds and making sure that for example a visitor can navigate both structures (runtime and declarative model) requires some thoughts.

Meta-Models Extensibility: How do we provide an extensible support for class extensions? Smalltalk supports class extension [BDNW05], *i.e.,* developers are able to add methods to classes and package them in different packages than the ones to which

the classes belong to. This is a simple mechanism that allows developers to add behavior to existing entities without subclassing them. The discussion here is how the Ring infrastructure can provide an extensible support to manage class extensions, as well as state extensions. Annotations are a possible solution. An interesting scenario is to see that exists a fundamental difference between Monticello 1 and Monticello 2: in MC1 one method can belong to only one package, while in MC2 a method may belong to multiple packages. It is not clear whether we should consider the possibility of such kind of changes in advance, but if is possible we should evaluate the cost of making those kind of changes.

Unifying Models: Can the reflective model and the declarative model be merged? As shown in Figure 10, the declarative model and the runtime model are independent of each other but they implement a common API. The question of knowing whether the runtime entities know their representation is an interesting question from the perspective of a reflective model having another separate and unconnected representation [DDL09]. The inverse is simpler, keeping track of the runtime representation of entities from the declarative definitions makes it easy and efficient (*e.g.,* Ginsu takes advantage of this). Now the question is if we cannot simply have either reflective entities that can be disconnected and played the role of declarative ones. This would simply merge both models as an optimal implementation of the Ring. The question of the API is then central.

6. Conclusions

In this paper, we have presented needed requirements for modeling source code and history of a system. In particular, those requirements stress the importance of supporting software engineers and integrators. Additionally, the requirements are complemented with a set of constraints that need to be taken into account.

Several versioning models and source code models have been presented. Some of them showed the connection between the versioning model and the code model (as in the case of the Monticello implementations). Each of those models have some benefits which, together with the requirements, gave us a background for proposing an unifying and foundational model infrastructure, named the Ring.

The Ring is conceptually presented as a code meta-model as well as an infrastructure for providing support to perform queries over the complete history, and as a means to encourage navigation tool reuse by providing an API on which external tools can rely on. Even though the Ring has not been implemented yet, we believe that we provide enough insights about it, its requirements and its scope.

Acknowledgments. We gratefully acknowledge the sponsoring of ESUG (the European Smalltalk User Group) http://www.esug.org/

References

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example.* Square Bracket Associates, 2009.

[BDNW05] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4):107–126, December 2005.

[BR98] John Brant and Don Roberts. "Good Enough" Analysis for Refactoring. In *Object-Oriented Technology Ecoop '98 Workshop Reader*, LNCS, pages 81–82. Springer-Verlag, 1998.

[Cha08] Scott Chacon. *Git Internal*. PeepCode, 2008.

[CSFP09] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion (for Subversion 1.6)*. O'Reilly Media, June 2009.

[DDL09] Stéphane Ducasse, Marcus Denker, and Adrian Lienhard. Evolving a reflective language. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST 2009)*, pages 82–86, Brest, France, aug 2009. ACM.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[FM10] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 175–184. ACM, 2010.

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 13–23. IEEE Computer Society, 2003.

[GR89] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[LZ05] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Software Engineering Notes*, 30(5):296–305, September 2005.

[Mae87] Pattie Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, January 1987.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[PK01] Joseph Pelrine and Alan Knight. *Mastering ENVY/Developer*. Cambridge University Press, 2001.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[Riv96] Fred Rivard. Reflective Facilities in Smalltalk. *Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996*, February 1996.

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[TJ88] Dave Thomas and Kent Johnson. Orwell — A configuration management system for team programming. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, volume 23, pages 135–141, November 1988.

[vdHL96] Peter van den Hamer and Kees Lepoeter. Managing design data: The five dimensions of cad frameworks, configuration management, and product data management. In *In Proceedings of the IEEE*, volume 84, pages 42 – 56. IEEE CS Press, January 1996.

[XS04] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *SEKE '04: Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering*, pages 123–128, 2004.

[YMNCC04] Annie Ying, Gail Murphy, Raymond Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9):573–586, 2004.

[ZWDZ04] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.