Ring: a Unifying Meta-Model and Infrastructure for Smalltalk Source Code Analysis Tools

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. To ease version history analysis we need adequate models to represent source code entities. Now naturally the questions of their definition, the abstractions they use, and the APIs of such models are raised, especially in the context of a reflective system which already offers a model of its own structure.

We believe that this problem is due to the lack of a powerful code meta-model as well as an infrastructure. 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 or Store. 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 hinders navigation tool reuse when meta-models do not offer polymorphic APIs.

As a first step to provide an infrastructure to support history analysis, this article presents Ring, a unifying source code meta-model that can be used to support several activities and proposes a unified and layered approach to be the foundation for building an infrastructure for version and stream of change analyses. We re-implemented three tools based on Ring to show that it can be used as the underlying meta-model for remote and off-image browsing, scoping refactoring, and visualizing and analyzing changes. As a future work and based on Ring we will build a new generation of history analysis tools.

Keywords: Source code meta-model, versioning, refactoring, Monticello, Smalltalk

---

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

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 providing conflict analysis as well support elementary merging. Nowadays, 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 and history analysis on top of versioning systems [ZWDZ04], for example, to compare all the differences between all the senders of a given method in the past is not straightforward. Another example is how to support cross-forks merging. Such examples, however, should be based on source code models [LTP04]. To ease version history analysis we need adequate models to represent source code entities. Now naturally the questions of their definition, the abstractions they use, and the APIs of such models are raised; especially in the context of a reflective system which offers already model of its own structure.

In Smalltalk, several source code meta-models coexist in a weakly causal connected way[1][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 adequately 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 reduce 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. As a result developers will end adding or modifying the same behavior (*e.g.,* introducing a test) several times 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 source code models and tools as well as an adequate infrastructure [vdHL96]. As a first step to solve the mentioned problems, this article presents six source code models that could be used to support several activities and be the foundation to build an infrastructure for source code oriented analyses. We stress the difference between the meta-models and their role. We present a unified meta-model, named *Ring*, that can be used for multiple purposes.

The contributions of this paper are: (1) a comparison of six code meta-models, and (2) the proposal to merge those models in a unified and foundational model infrastructure, named *Ring*, and (3) the validation of Ring usage as meta-model for three tools.

[1] Causal connection is defined by Maes as: "A computational system is said to be causally connected to its domain if the internal structures and the domain they represent are linked in such a way that if one of the two changes, this leads to a correspond effect upon the other"

In Section 2, we first describe some requirements for typical scenarios software engineers face. Section 3 describes the source code meta-models of well-known Smalltalk projects. In Section 4 we introduce *Ring* our proposed meta-model. Section 5 presents a first validation where Ring is used as a meta-model of three existing tools: out-of-image code browser, Refactoring Engine scoping model, and Torch –a specialized visual tool for supporting source code changes integration. In Section 6 we discuss several open questions about its infrastructure. Finally, we conclude this paper in Section 7.

2. Requirements for Source Code and History Modeling

The source code and its history are an invaluable resource for software engineers, developers and integrators [LTP04]. They 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 their 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 Fritz and Murphy's work [FM10] that provides a list of 35 questions related to changes to the source code that developers commonly ask during maintenance tasks.

- *Queries as in the past*[2]: For example, what were the senders of the method asString in *Squeak 3.9*? Were they stable over their own history?
- *Queries as in the present*[3]: For example, what are the senders of method *readStream* in *version 3.1*? What are the messages sent by method printOn: in *version 2*?
- *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?

[2] Retrieve the original implementation of methods that were invoked in a past version regardless of their current implementation.

[3] Retrieve the last implementation of methods that were invoked in a past version.

- *Global analysis*: The following questions illustrate the kind of analysis to be supported by a meta-model: What is the whole history of method detect:ifNone:?[FM10]. Was the method directoryNamed: renamed in the past? if yes, in what version? and what was its previous name? What is the most queried history entity? What is the difference in usage of a given method between the different versions?
- *Bug spot*: Was this method regularly changed over the last 15 years?
- *Forks analysis*: For fork analysis, specific requests should be answered: for example, What are the previous changes that are required to merge a change with another branch? What are the users potentially impacted by a change in the source code and as well in the destination fork? More concretely, if the version of method isNil changed in *Squeak 3.9*, should it be changed in *Pharo* and what would possibly be the impact?
- *Comparison profiler*: after loading one version and running it, and loading another version and running it. What are the differences or similarities of both running versions?

Our questions and the ones defined in Fritz and Murphy's work [FM10] reinforce our claim that developers lack support for easily retrieving information to perform adequate analyses. We intend to take these questions into account when building our meta-model and infrastructure.

2.2. Constraints

The meta-model and infrastructure that we propose have a set of constraints that emerge from reuse and practical integration with the host environment, *i.e.,* Pharo[4] environment. We motivate the most important constraints:

- *No duplication of meta-models.* We do not want one source code runtime meta-model and one for change and versioning system. Having different meta-models is costly to maintain, test, and keep them in sync. Our goal is to define a common source code 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.
- *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 Smalltalk has its own reflective meta-model that is used by the runtime system [BDN+09] which is causally connected (meaning that the model reflects its subject in any circumstances). Therefore, the additional model should use the causal connection as much as possible. Note that

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

taking such advantage is only possible for analysis 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 for new tools to define their own meta-model with non-polymorphic API for representing entities. This hampers the reuse of tools manipulating source code entities, as their API might differ from each other. Having a common meta-model will ease the integration and reusability of those tools.

2.3. 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 that 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 reason 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 more 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[5][Cha08] or SVN[6][CSFP09] 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 building 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. In fact, in parallel to this work GitFS and Pharogenesis[7] were proposed to build Git repositories in Pharo proving the need of an extra infrastructure.

In a previous work [UGDD10a] we presented four versioning models as well as six code models. In the rest of this paper we focus on source code meta-models and their unification to support tool building.

[5] http://git-scm.com
[6] http://subversion.apache.org
[7] http://scg.unibe.ch/archive/projects/Lesk11a.pdf

3. A Smalltalk Source Code Meta-Model Zoo

While versioning focuses on how to merge and version 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 (*e.g.,* for managing changes, refactorings, merges and versions) that manipulate in some way the Smalltalk structural meta-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 dialect, 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. 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[8] software analysis platform, a model is created for a version or for the actual 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.

3.1. FAMIX

FAMIX 3.0[9] is a family of meta-models oriented towards software analysis. These models were developed in the context of the Moose 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 1

[8] http://www.moosetechnology.org

[9] http://www.themoosebook.org/book/internals/famix

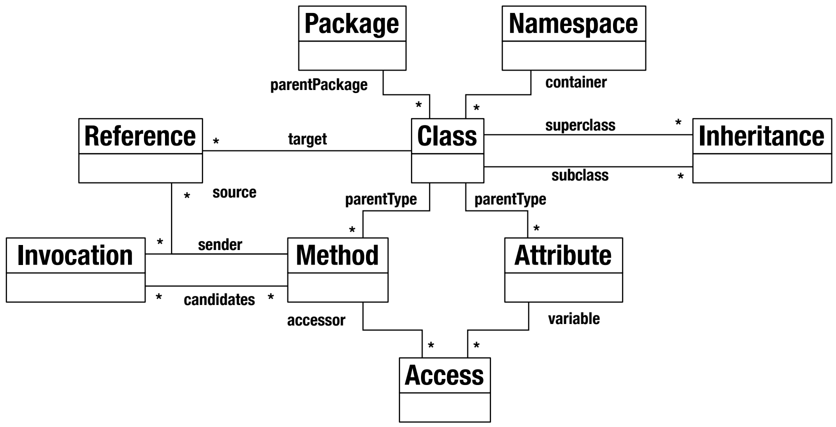

Figure 1: An overview of the FAMIX-Core language independent meta-model.

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.

1. 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* other methods (Invocation). In 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, produce a language independent information in terms of FAMIX models.

2. FAMIX provides decoupling between Packages and Namespaces. Namespaces are scoping entities that provide a lexical scope for the contained entities, while packages scoping are entities that describe the physical structure of a system (*i.e.,* deployment entities). This decoupling makes sure that FAMIX can model any kind of situations at the package level.

3.2. Refactoring Browser

The Refactoring Browser[10] –known as *RB* [BR98, RBJ97, Rob99]– is a powerful Smalltalk browser which enables developers to perform several automated refactorings on Smalltalk programs, such as: pushing up methods, renaming variables, splitting classes, etc. The refactorings can be classified into three

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

groups: class based refactorings, method based refactorings, 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 scoping model*, which is relevant for our study, identifies the program elements that are manipulated for the rest of RB models. In addition, the *RB source code scoping model* models a delta on the current system and it is supposed to be polymorphic with the Smalltalk meta-model.

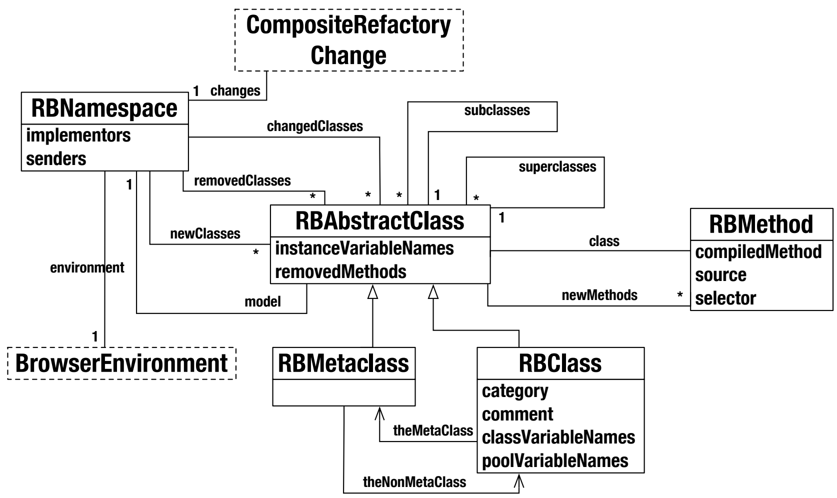

Figure 2: Refactoring engine declarative source code scoping model.

The complete source code model is shown in Figure 2. Note that, it is a very simple model, only mapping classes, methods and namespaces. Other elements, such as variables, are not modeled as first-class objects. Two external classes to this model are BrowserEnvironment and CompositeRefactoryChange (shown with dashed border), both are associated to RBNamespace. The first represents the environment in which a namespace is defined and the second allows a namespace to control changes and refactorings.

3.3. 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 3. 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 instance 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 (methodDict). Classes are organized in categories, or what is commonly known as packages. However, this model only keeps category names in SystemOrganization, an instance of SystemOrganizer. The protocols of a class (*i.e.,* method categories) are managed by ClassOrganizer. Finally, every entity knows the environment (*i.e.,* namespace) in which it is visible, such environment is unique and is represented by an instance of SystemDictionary.

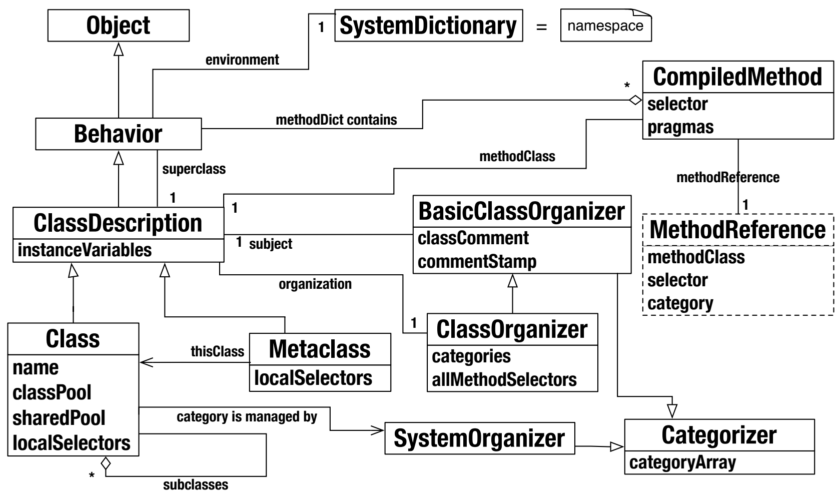

Figure 3: Smalltalk (Pharo) core model (with dashed border an attempt to add a representational object for CompiledMethod).

Key points. There are three points to stress about the Smalltalk model.

1. 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.

2. The 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.

3. Figure 3 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 of different versions of a method. Originally MethodReference was not polymorphic with the static API of a CompiledMethod. This resulted in tools duplication and level mixing (UI and execution).

3.4. Ginsu

Ginsu[11] 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 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.

Ginsu maps the elements defined in Smalltalk code to semantic objects. A semantic object represents the semantics 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.

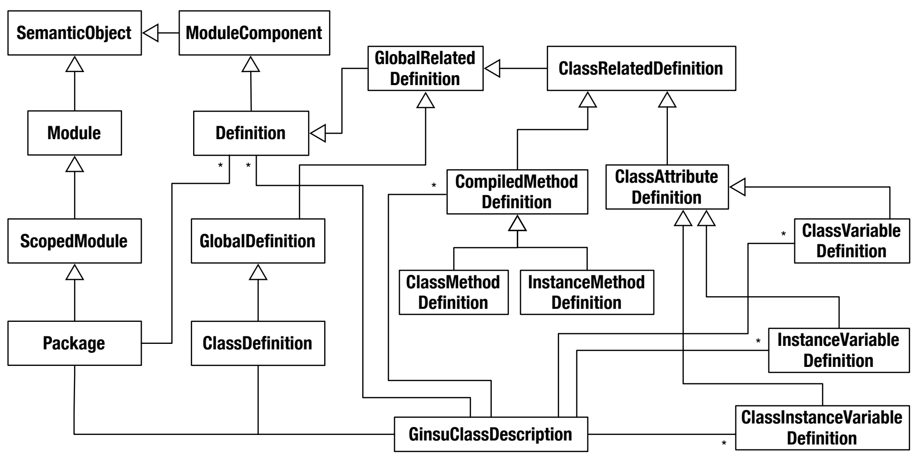

Figure 4: Ginsu key classes.

The key classes defined in the semantic model are shown in Figure 4. 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 browser) interact with class descriptions.

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

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

3.5. Monticello 1

Monticello 1[12] is a distributed concurrent versioning system for 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 composed of classes and methods, organized in various ways, and with dependencies.

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 stored on a repository. It also stores associated metadata as VersionInfo and the version's ancestry. Versions are often .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 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.

Source Code Model. 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. Figure 5 shows the key entities of the Monticello 1 models (*i.e.,* versioning model and source code model) and the link between both models. Note that the versioning model is displayed in grey to differentiate it from the source code model.

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: MCPackage (as PackageInfo –an external class), MCClassDefinition maps a class contained in a package, MCOrganizationDefinition maps the categories names in which classes are defined (*i.e.,* packages names). Subclasses of

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

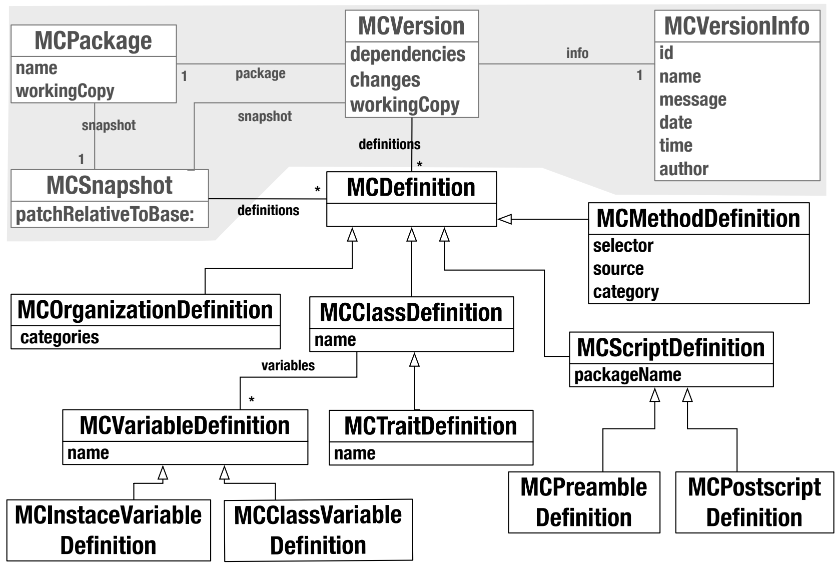

Figure 5: Monticello 1 key classes for source entities (in grey the versioning entities).

MCVariableDefinition represent variables of classes, and they are accessed by class references. MCMethodDefinition maps structural data of methods (selector, source code). Finally, MCScriptDefinition subclasses represent the pre/post conditions required by packages.

The source code model is not complete. For example, there are no definitions to represent class extensions as first-class objects, but instead naming convention in method categories is used.

In spite of the presented benefits of Monticello 1, its versioning model 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 that 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. Note that such issues are orthogonal to the source code model used and are out of the scope of this article. Nevertheless, these issues are addressed by Monticello 2.

3.6. Monticello 2

Monticello 2[13] (a.k.a. *MC2*) 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 6 presents the key classes defined in the source code model of MC2. It also shows the link

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

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.

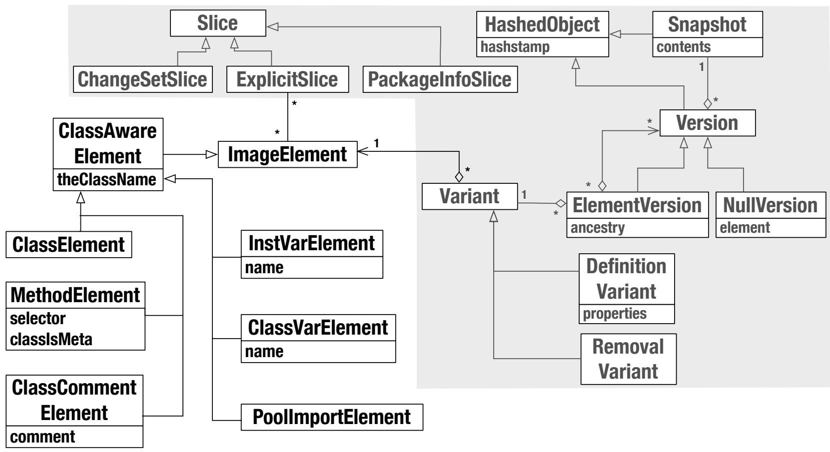

Figure 6: Monticello 2 key classes for source entities (in grey the versioning entities).

The main source code entities in Monticello 2, which map to program elements, derive from ImageElement. In Monticello 2, an *element* is more fine-grained than a *definition* in Monticello 1; 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.

The versioning model in Monticello 2 does not have packages as the fundamental unit of versioning. Instead, the unit of versioning is the individual program elements – classes, methods, instance variables, etc. This means that MC2 can be used to version arbitrary snippets of code. These might correspond to packages, change sets, or any other way a programmer chooses to separate "interesting" code from the rest of the image. Rather than maintaining the version history of packages, MC2 keeps version history for each element. Having this history allows users to perform tasks that are not possible with MC1 (*e.g.,* access the whole history of a particular method). With this versioning 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.

4. Towards a Unifying and Foundational Model Infrastructure

Section 2 shows that lots of questions are about history, and that history requirements are linked to the source code model. In this article we stress the importance of getting a well-designed source code meta-model which we call

Ring. In the future, Ring will serve as foundation for a solid history meta-model. In addition, we want that existing tools use Ring to avoid code and task duplication.

4.1. The Ring meta-model intuition

One key idea behind Ring is to define a meta-model and infrastructure that provide a common API at structural and runtime level and allow 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 partially polymorphic model with the Smalltalk meta-model but this implies duplication of information (*e.g.,* tests).

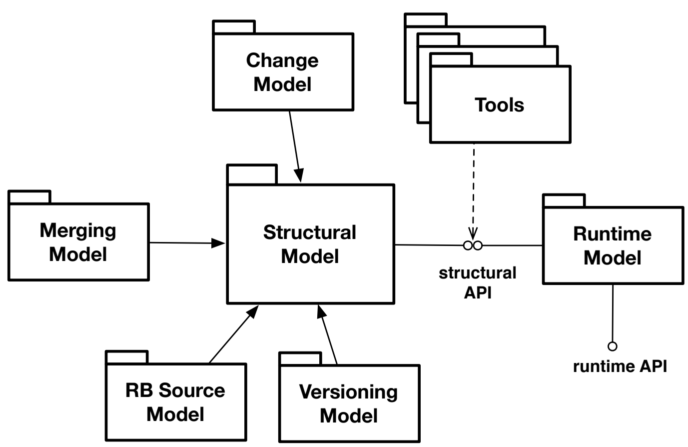

Figure 7: The Ring overview.

Figure 7 shows our proposal for the Ring meta-model and infrastructure, and how its components and tools should interact. 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, for example a code browser should browse entities (*e.g.,* classes, methods) loaded in image or in external code files (*e.g.,* changesets). Finally, the rest of the main models (*i.e.,* change, RB, versioning and merging models) should use and extend the structural model.

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 8 (left side). The Smalltalk runtime model (right side) was presented in Section 3.3. Defining Ring in layers can facilitate reuse and integration with other tools and models. We envision to define a solid *core source* model and in the future a *history* model and a *merging* model built on top. The *core source* model will only know about the essential structural entities such as classes, methods and comments.

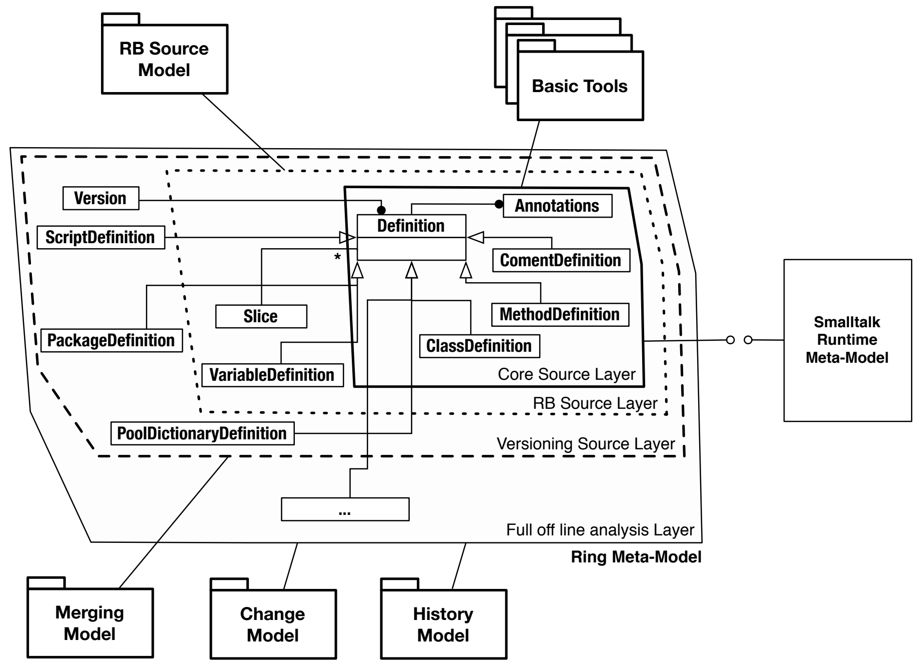

Figure 8: The layered structure of Ring.

In the context of merge support, change and history understanding [UGDD10b], we want the largest layer of Ring to be used to store all the versions of all the entities of the system to support advance queries and merging algorithms. Finally, we want to offer such storage as a web service.

4.2. The Ring meta-model first design

In this section we present a concrete implementation of Ring[14]. This is our first design and was based on an exhaustive analysis of four source code meta-models presented in Section 3: 1) Smalltalk structural and runtime meta-model, 2) Refactoring source code scoping model, 3) Ginsu, and 4) Monticello 1 source code model.

So far we have implemented two complete layers of Ring (*i.e.,* Core source layer and RB source layer), and partially the third layer. All the classes are named using the prefix **RG** which stands for *Ring*.

[14] http://www.squeaksource.com/Ring

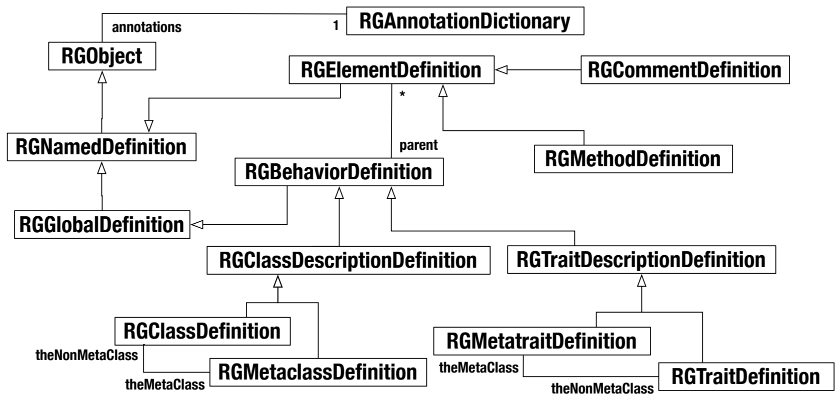

Figure 9: The Ring core source layer/model.

Core source layer/model. The first layer of Ring is shown in Figure 9. The core source layer comprises the main entities present in the source code (*i.e.,* classes, methods and comments). This also implies that our first layer corresponds to the *core source model* as well. The root class in Ring is RGObject, a subclass of Object. We incorporated in RGObject the annotations feature proposed by Ginsu that allows users to add properties to an object without altering its structure. Annotations are managed by RGAnnotationManager and kept in an RGAnnotation-Dictionary object. Entities in the source code are treated as definitions in Ring, and thus we defined RGNamedDefinition, from which most of our classes are derived.

Note that for classes and methods we have defined two hierarchies on top as we needed to define the basis for class-like entities (*i.e.,* traits[BS04]) and other elements of classes (*e.g.,* comments, variables). Two main definitions separate classes from methods, RGBehaviorDefinition for class-like entities and RGElement-Definition for elements such as methods or comments. A behavior definition can have multiple element definitions, and each element definition knows the class in which is defined by its unified property parent. We propose parent to solve the current main problem of having multiple names for this property among several source code models, such as methodClass, parentClass, belongsTo, className, etc. The same problem happens when referring to the class of a variable.

A class and a metaclass are also separated definitions represented by RGClass-Definition and RGMetaclassDefinition respectively. A class knows its theMetaClass and a metaclass knows its theNonMetaClass. Both have similarities and inherit from RGClassDescriptionDefinition. Note in the diagram that the same logic and hierarchy are applied for traits.

Methods and comments are element definitions (concretely RGMethodDefinition and RGCommentDefinition) that inherit from RGElementDefinition. An element knows in which class is defined by its parent property.

The small core source model is sufficient to re-implement the simple browser that supports (external) file-based code browsing available in Pharo and Squeak known as *File Contents Browser*. In addition, as the core also includes comment definitions, it is also sufficient to manipulate changesets.

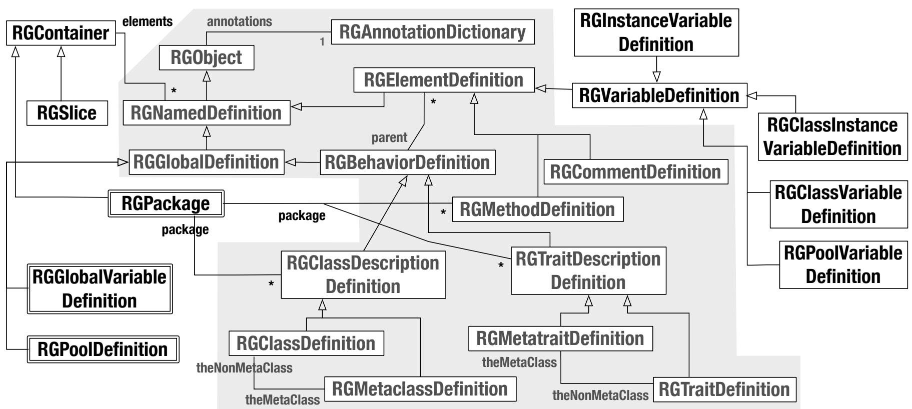

Figure 10: Adding Package, Variables and the Ring RB source scoping layer.

RB source scoping layer. The second layer and a partial third later of Ring are shown in Figure 10. The RB source scoping layer is the basis for re-implementing the Refactoring engine source code scoping model explained in Section 3.2. In this layer we have added definitions for variables as well as slices. Note that the diagram shows the core source layer with grey background to distinguish from the new classes.

Package and shared definitions belong to the third layer and they are shown with a double border in the diagram. Even thought that layer is not finished yet we are introducing those classes in this section because we make use of them in one of our validations.

Variables are all subclasses of RGVariableDefinition which inherits from RGElementDefinition. They can represent instance variables, class variables, class instance variables and pool variables. Pool variables do not represent shared pools of Smalltalk but their usage relation with classes.

A package is represented by RGPackage and may contain many RGBehaviorDefinition or RGElementDefinition objects, such as classes, traits or methods (in case of class extensions). A class knows the package in which it is packaged by means of the property package. Similarly a method knows its package when it extends a class. This follows our API unification when representing the relationship *B is the parent of A* by means of the unique message parent which is mapped to semantics language elements: a class is the parent of a method, a comment or of a variable. Therefore we have two relationships: parent for language driven relationship definitions and package for deployment and ownership representation.

RGSlice represents a group of elements as RGPackage. In fact, both inherit from RGContainer. But a slice is more general than a package as it can group several kind of elements, not only classes or methods, but also variables, senders, etc. Every Ring object is linked to an environment. Thus, a slice is used to identify elements on which particular operations should be performed (*e.g.,* rewrite, rule checking, browsing,...). We also have an initial representation of Namespace, which will be included in the next version of Ring. For now the environment is pointing to Smalltalk globals (the default Smalltalk namespace).

Finally, we also have shared definitions as subclasses of RGGlobalDefinition. We emphasize RGPoolDefinition as the mapping definition of shared pools in the system.

5. Validation

In this section we present how we migrated three existing tools to be based on Ring as their source code meta-model: we show the code file browser, the refactoring engine source scoping model, and the Torch dashboard.

5.1. External Code File Browser: Out-of-Image Code Browsing

Smalltalk IDEs allow developers to browse and load the contents of external source code files – change set files (.cs) or smalltalk source files (.st). Before loading source code files into an image, developers usually browse the contents of such files first to be sure that those files contain the needed source code.

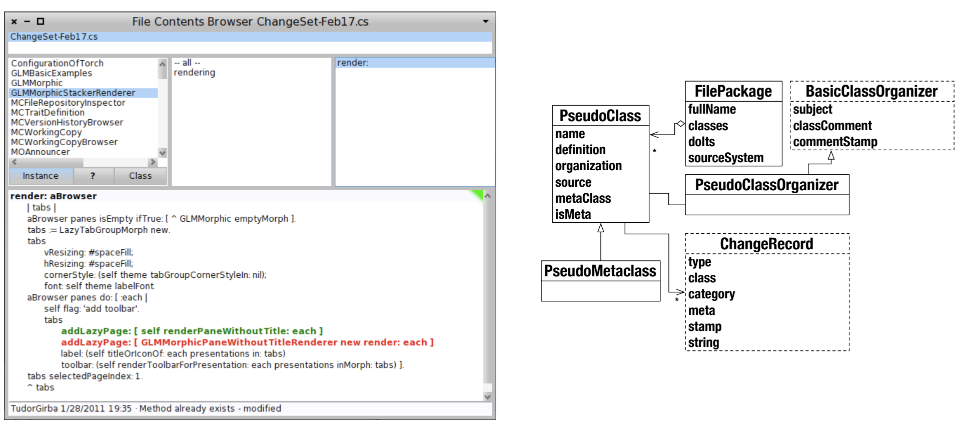

Figure 11: The current FileContentsBrowser and its meta-model.

For this task, the Pharo environment provides users with a browser FileContentsBrowser shown in Figure 11 (left). The source code is represented with a meta-model created for this particular browser, such model is known as the *pseudo classes* shown in Figure 11 (right).

PseudoClass and PseudoMetaclass represent classes. On one had, both classes are not related to the ones that define classes in the Smalltalk structure meta-model, and do not fully implement the same API. On the other hand, a part of the class data (*i.e.,* comment, stamp and method categories) is indeed managed by PseudoClassOrganizer, a subclass of BasicClassOrganizer that is defined in the Smalltalk structure meta-model. In addition, for methods there is no pseudo method definition, but instead ChangeRecord objects are stored into a dictionary that corresponds to the property source of a pseudo class.

As expected this small meta-model is chaotic, and maintenance on it is avoided at such extend that traits are not being loaded and presented in the FileContentsBrowser.

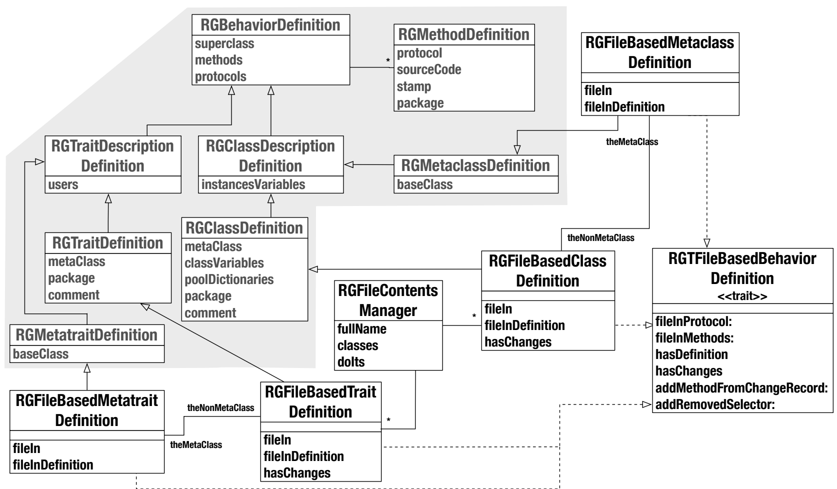

Figure 12: Ring proposal for replacing the pseudo classes.

Our proposal for browsing source code file contents is shown in Figure 12. The pseudo classes are replaced by Ring subclasses of classes defined in the *core source layer*. The source code of files is mapped to classes, traits and methods defined in the Ring core model. Additional behavior for browsing source code (*i.e.,* loading data from change records and filing the source code in the image) is needed. For this, we had two alternatives, subclassing class and metaclass definition, or creating method extensions. We opted for the first option and have created RGFileBasedClassDefinition and RGFileBasedMetaclassDefinition subclasses of RGClassDefinition and RGMetaclassDefinition respectively. The same subclassing was done for traits. As the new subclasses have common behavior, we also introduced of a new trait RGTFileBasedBehaviorDefinition that implements such behavior.

We also simplify the original FilePackage and replaced it by RGFileContents-Manager. RGFileContentsManager is dedicated to read files, to provide the loaded data to the browser, and to offer the file in operation of the whole file.

Finally, the new FileContentsBrowser shown in Figure 13 was implemented from scratch. It uses *Glamour*[15], an engine for building dedicated browsers.

[15] http://www.moosetechnology.org/tools/glamour

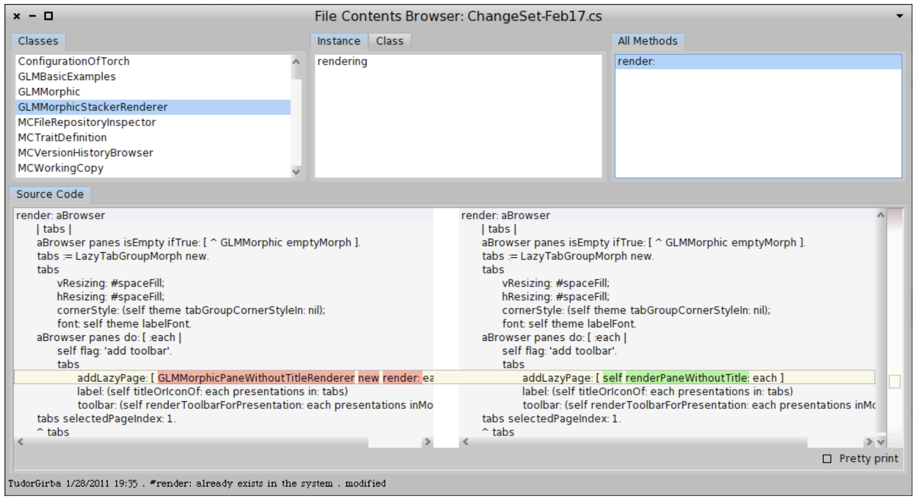

Figure 13: The new FileContentsBrowser.

5.2. Refactoring Engine Source Scoping Model

The Refactoring Browser and its current declarative source code scoping model were introduced in Section 3.2. In our validation we redesigned its source code model extending the second layer of Ring.

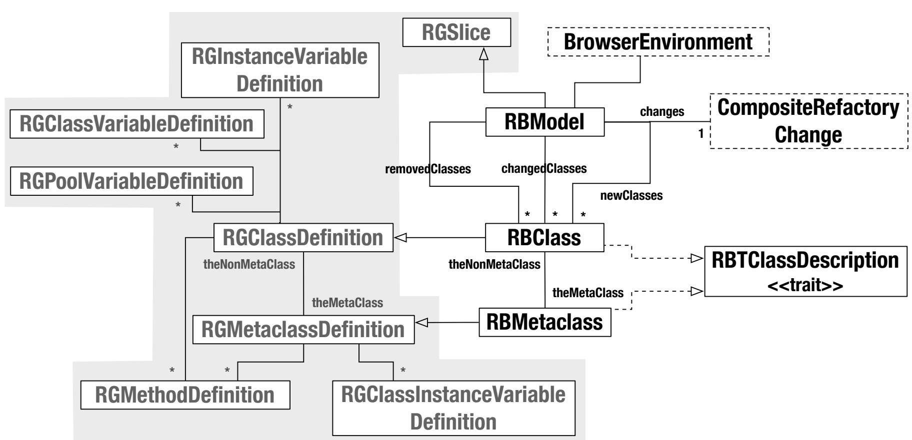

Figure 14: Refactoring engine new declarative source code scoping model.

Figure 14 shows the new proposal for the source scoping model. The new *RB* classes are subclassing existing Ring classes such as RGClassDefinition, RGMetaclassDefinition and RGSlice. The old RBMethod class is not needed anymore, instead the Ring RGMethodDefinition class instantiates method objects. Additional parsing behavior required by method objects was added as method extensions of RGMethodDefinition.

The new RBClass and RBMetaclass classes make use of a newly introduced trait, RBTClassDescription that defines the global behavior of the Refactoring Engine to deal with changes, refactoring and conditions. Note that the old model did not define classes for mapping variables. Instead collection of names (*i.e.,* string and symbols) were used to represent instance variables, class instance variables, class variables and pool dictionaries. In the new proposal variables are first-class objects represented by the subclasses of RGVariableDefinition. This change in particular had a high impact in the API used by the Refactoring Engine, in addition to changes to the *source code model*, both the *refactoring model* and the *changes model* needed some minor changes as well.

Finally, the old RBNamespace class had few changes in the new model. In the Refactoring engine a RBNamespace object deals with changes associated to its environment, and keeps such changes in separated groups depending on their change status (*i.e.,* new, removed, changed). We did make a few adaptations in its API, renamed it to RBModel and inherited it from RGSlice. The link to an environment is not affected as every ring object knows the environment in which it is working. As for the Refactoring Engine, it continues being a BrowserEnvironment instance. Since Squeak/Pharo do not have explicit namespace at the language level, considering the original namespace as a kind of slice is not a contradiction, but in future versions, we will use the final implementation of RGNamespace instead of RGSlice.

5.3. Torch dashboard: Visually Supporting Source Code Changes Integration

Torch is a visual tool for understanding source-code changes [UGDD10b]. Torch supports integrators in taking decisions about the integration of changes before performing the actual merging, and offers developers a means to understand and control their changes before publishing them. Torch provides an overview of how a Smalltalk program was changed, and aims at aiding its users in understanding these changes.

Torch characterizes the changes based on structural information, authorship and symbolic information. Figure 15 shows the Torch dashboard that presents different structural representations of source code changes using visualizations.

In Figure 16 we show the current source meta-model of Torch. This model did not extend an existing one but followed some conventions from the Monticello 1 source code meta-model. As the dashboard is integrated in the Monticello browsers, its data comes from Monticello definition objects (*i.e.,* MCDefinition and subclasses).

The main characteristic of Torch is that it maintains a unique object representing a particular entity (*e.g.,* a class, a method, a package). Objects in Torch are stateful and thus each object knows which of its properties have changed and what are the old and new values. The following example clarifies the logic in Torch. If we compare two versions in which a class has changed its superclass, Monticello provides two MCClassDefinition objects for that class. Torch converts both objects into one TCClass object and set its status to modified. Finally, the subclass instance variable keeps a TCChangedElement object that contains the old and new subclass.

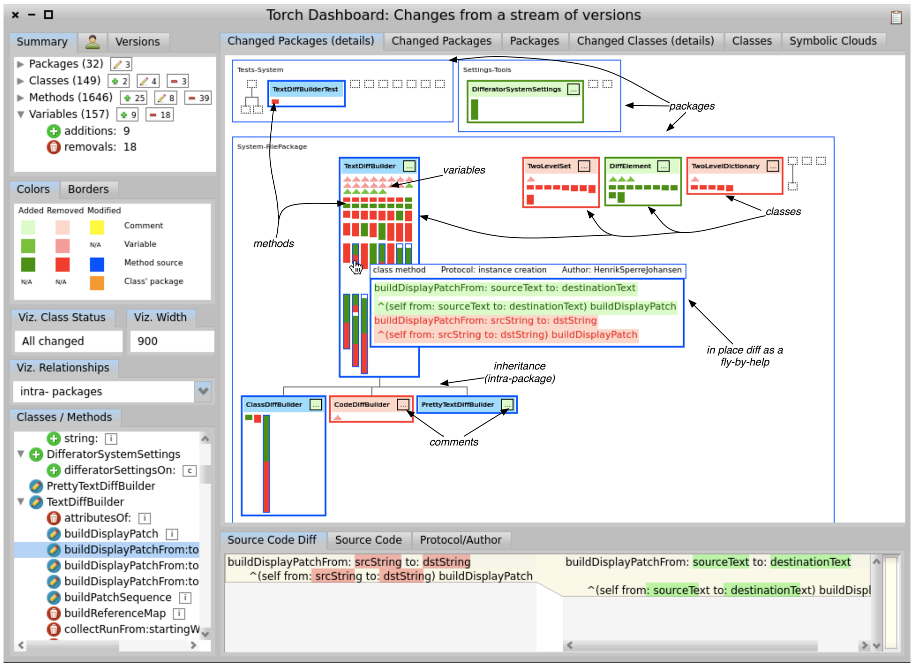

Figure 15: The Torch dashboard.

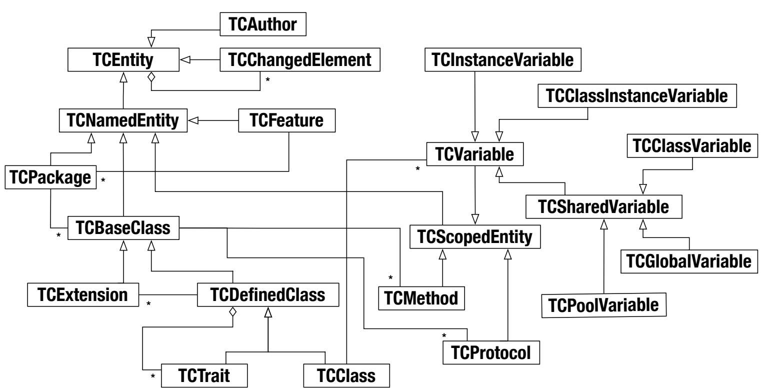

Figure 16: The actual Torch source meta-model.

Another characteristic of the classes defined in the Torch meta-model is that they also know how to be drawn in the dashboard. Mixing the business logic with the drawing logic has increased the complexity of such classes.

For this validation, we have re-implemented the source meta-model of Torch using Ring – including our partial third layer. Moreover, we have also separated the drawing logic of entities in a new layer. In Figure 17 we show the classes of the new source meta-model (classes with white background) and how they relate

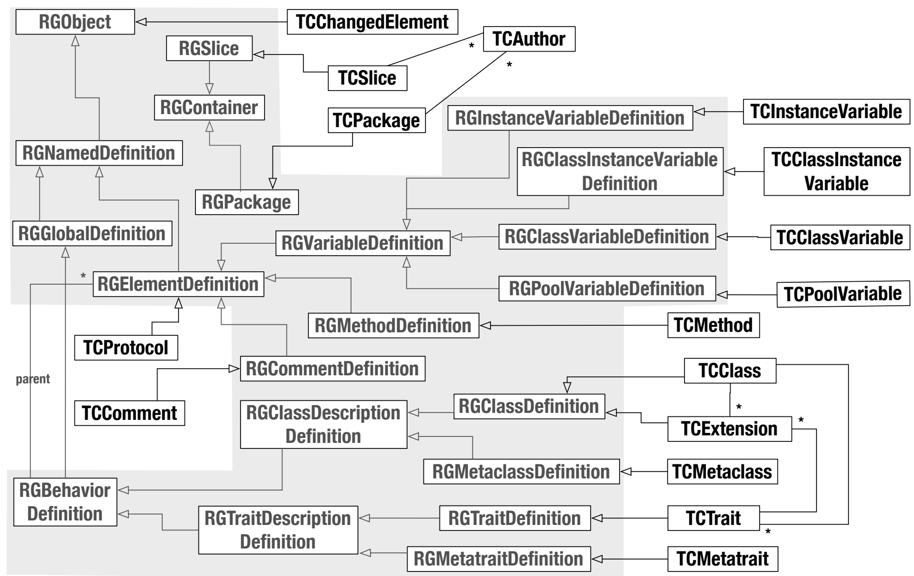

Figure 17: The key classes of the new Torch source meta-model.

to the Ring classes (classes with grey background). Note that in this diagram we are only showing the inheritance relationships between classes to simplify it. Associations and aggregations relationships are semantically the same as in the original meta-model shown in Figure 16, *e.g.,* a class (TCClass) has 0..* variables (subclasses of RGVariableDefinition).

We kept the logic of unique stateful objects and changed properties linked to them, but this behavior is defined in a new trait TCTObject. This trait is used by all the entities that need state. A newly introduced class TCSlice replaced the old TCFeature as slice is a better concept for representing a group of entities. Two classes have been introduced in Torch, TCProtocol and TCExtension that inherit from RGElementDefinition and RGClassDefinition respectively. TCProtocol maps the method categories, and TCExtension –a class alike definition– groups method extensions of a class per external package.

In addition, common existing behavior was grouped in traits to avoid code duplication. TCTComment and TCTStamp take care of comments, authors and timestamp in classes and methods. TCTClass defines the specific behavior of classes and traits, and TCTClassAlike the behavior of class extensions.

The actual classes in Torch define additional properties non-existing in Ring classes. Those properties are caches that exclusively avoid repeated data processing when drawing (*e.g.,* allSuperclasses keeps all the superclasses in the inheritance chain of a class present in the data). In the newly proposed source meta-model, classes do not define additional instance variables, but keep extra information as annotations a feature of Ring.

To simplify the integration of Torch with Ring, we also separated from classes

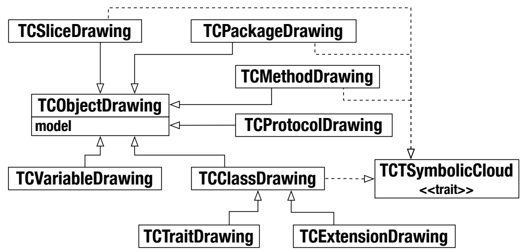

Figure 18: The drawing layer of Torch when using Ring.

the logic that is responsible for drawing the entities in the dashboard and moved to a new layer. Our Ring specializations only care for business logic. The drawing layer is shown in Figure 18. Note that it involves less classes than the new source meta-model. For each entity that needs to be drawn a class have been defined (*i.e.,* slice, package, class, trait, extension, method, variable). We call them *drawing classes* and they inherit from TCObjectDrawing that knows the model (*entity* that will be drawn). Some entities are able to display symbolic clouds (*i.e.,* a particular kind of visualization in Torch) and for such behavior the trait TCTSymbolicCloud is used.

6. Open Questions

The goals of the Ring meta-model and infrastructure are clear. However, there are open questions that we will need to address when modeling the complete history of the system (since 1996) and when building tools to help merging and manipulating versions.

6.1. Current Ring Concerns

Revisiting design choices. We will continue stabilizing Ring based on other validations. Now the following concerns will have to be addressed.

- **Namespace**. Even though Pharo does not have namespace support at the language level, it has a namespace to support class lookup. Modeling namespace will enable remote browsing and other facilities. Therefore this is already a definition being implemented.
- **Slice/Package interaction**. We have to experiment more to understand how slice and package interact.
- **Compiled method**. CompiledMethod has some method extensions for MethodReference that we do not want in Ring. We are considering to define a subclass of RGMethodDefinition to deal with compiled methods as a temporal solution for tools that use such a reference. But our goal is to

remove MethodReference and make some adjustments in the Smalltalk runtime model as well. This API unification will allow us to interact with compiled methods without any definition on top.

- **Layer names vs Model names**. We are discussing about the names set to layers. We want to highlight that a layer in Ring is not always a full model for a particular tool, for example, the *RB source scoping layer* is the foundation layer for the *RB source scoping model* implemented in our second validation. But as names are almost the same this can bring confusion.

Unifying Models. Can the reflective model and the declarative model be merged? As shown in Figure 7, 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 play the role of declarative ones. This would simply merge both models as an optimal implementation of Ring. The question of the API is then central.

Core source model API. We intend to encourage tool reusability by relying on a common API of the main entities (*i.e.,* classes, methods, variables) 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 arises 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 thought.

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 these kind of changes.

6.2. Representing versions

Since one of the goals of Ring is to provide a solid foundation to support history analyses and merging, we should consider the following points.

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 needs 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 every 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.

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 some 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 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 are able to manipulate all that information.

7. Conclusion

In this paper, we have presented needed requirements for modeling the source code and history of a system. In particular, these 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.

Six 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 these models have some benefits which, together with the requirements, gave us a background for proposing a unifying and foundational model infrastructure, named the Ring.

We presented Ring, a unifying meta-model and its implementation. We validated the first design and implementation of Ring by migrating three existing tools to use such meta-model. These validations showed how such unrelated tools replaced their source code meta-model by using and extending the common API proposed by Ring.

As result of the validations the roadmap of Pharo 1.4 already includes the integration of Ring. We will provide support to the integration process but the engineers of Pharo will lead the integration of Ring with the tools included in the core distribution. Two of our validations corresponding to core tools (*i.e.,* Refactoring Engine Source Scoping Model and the Code File Browser) will be considered. The third validation, the Torch dashboard using Ring is already released.

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.

[BS04] Andrew P. Black and Nathanael Schärli. Traits: Tools and methodology. In *Proceedings ICSE 2004*, pages 676–686, May 2004.

[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.

[LTP04] Timothy Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*, volume 94, pages 7–18, 2004.

[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.

[UGDD10a] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Meta-models and infrastructure for smalltalk omnipresent history. In *Smalltalks'2010*, 2010.

[UGDD10b] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Visually supporting source code changes integration: the torch dashboard. In *Working Conference on Reverse Engineering (WCRE 2010)*, October 2010.

[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 Press, 2004.