Unanticipated Integration of Development Tools using the Classification Model

Roel Wuyts and Stéphane Ducasse

Software Composition Group
Institut für Informatik und angewandte Mathematik
Universität Bern, Switzerland

Abstract

The increasing complexity of software development spawns lots of specialised tools to edit code, employ UML schemes, integrate documentation, and so on. The problem is that the tool builders themselves are responsible for making their tools interoperable with other tools or development environments. Because they cannot anticipate all other tools they can integrate with, a lot of tools cannot co-operate. This paper introduces the *classification model*, a lightweight integration medium that enables unrelated tools that were not meant to be integrated to cooperate easily. Moreover, the tool integration is done by a *tool integrator*, and not by the *tool builder*. To validate this claim, we show how to integrate several third-party tools using the classification model, and how it forms the foundation for the StarBrowser, a Smalltalk browser integrating different tools.

1 Introduction

As software systems get increasingly more complicated, developers need to be able to rely on adequate programming languages and development environments. Therefore lots of specialised tools exist that help developers to cope with a certain aspect of software development. For example, development browsers provide sophisticated ways to edit and navigate source code, while UML tools allow developers to draw class diagrams and sequence diagrams. There is one problem however: these tools have to be used together in order to implement the systems that need to be finished. Hence they have

Email addresses: `roel.wuyts@iam.unibe.ch` (Roel Wuyts), `ducasse@iam.unibe.ch` (and Stéphane Ducasse).

to cooperate somehow, and this is most of the time problematic, due to the following reasons:

- Each tool typically uses its own GUI, and its own conceptual model to store and represent information. It is the tool user who has the responsibility to combine the results from different tools.
- It is hard to make the tool work on new kinds of items. Suppose, for example that we have a UML tool in the development environment that knows how to display a class diagram of all classes in a certain package. Adapting this tool to now take as input the results from another tool that does not work with packages is a major undertaking.

When tools properly work together, synergy occurs: the output of one tool can be used as input for another. This is what we call *integration of tools* in the scope of this paper. For example, integrating a UML tool in a development environment makes it possible to show a UML diagram of all the classes in a certain namespace. When integrating a software architecture tool in the same environment afterwards, it is possible to show UML diagrams of the classes in a layer of the architecture of a system. These combinations are endless, and allow a developer to navigate and combine the tools easily to help with the task at hand.

To allow the integration of tools some environments (like Microsoft's Management Console or the Open Source Applications Foundations' *Chandler*) have an *integration architecture* that tools can follow to cooperate with the platform and each-other. We call this form of tool integration *anticipated integration*: the tool builder has made sure that the tool adheres to a certain integration platform. While this has the advantage that the tool is not stand-alone anymore, it has the drawback that it can be very hard to make it compliant to an integration platform (due to the complexity and to the fact that the code of the tool is needed to make the changes). Moreover, it does not solve the problem that the tool cannot be integrated with other tools that were not built on the integration platform.

This paper tackles the problem of *unanticipated integration*: integrating unrelated tools that do not adhere to a certain integration platform. As solution we present the *classification model*, a lightweight grouping mechanism for objects that are acted upon by services. The design of the classification model combines a composite and visitor design pattern [4], which makes it easy to comprehend and extend. It serves as a *lingua franca* for tools, and provides a clear separation of concerns between the tools to be integrated, the glue code to do this, and the integration model. This has the important benefit that each of these distinct phases is done by different developers (the *tool builder*, the *tool integrator* and the *model builder*), who are not necessarily aware of each other.

We have implemented the classification model in the VisualWorks [2] and the Squeak [6] Smalltalk environments. The most important user of the model is the *StarBrowser* [1] that acts as a shell around existing Smalltalk development tools such as the object inspector, development browsers and UML tools, effectively integrating them.

The contributions of this paper are:

- the presentation of the classification model,
- the unanticipated integration of tools by customising the classification model,
- the practical application of the integration by building a concrete development tool, the StarBrowser.

The rest of this paper is structured as follows. Section 2 introduces the classification model in detail. Section 3 shows how the model can be customised to accommodate for new kinds of items or services. Then Section 4 shows how the classification model is used to integrate tools that were not designed to cooperate. Section 5 shows the StarBrowser, a browser that uses the classification model to integrate with the Smalltalk environment and other tools. Section 6 discusses several aspects of the classification model in more detail, while Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 The Classification Model

The classification model allows to group all kinds of entities and to uniformly manipulate these entities and groups. Particular about the model is that the grouping is independent of the manipulations, and that the model can be customised to either support particular groupings or to support particular manipulations or both. The following sections describe the model and discuss its design rationale.

2.1 Overview

The classification model is built on the following main concepts, shown in Figure 1:

- *items.* An *item* is anything tangible as an object in the software development environment, such as a class, a namespace, an image, an HTML file, . . .

[1] The StarBrowser can be freely downloaded from the StarBrowser Web page at http://www.iam.unibe.ch/∼wuyts/StarBrowser/index.html

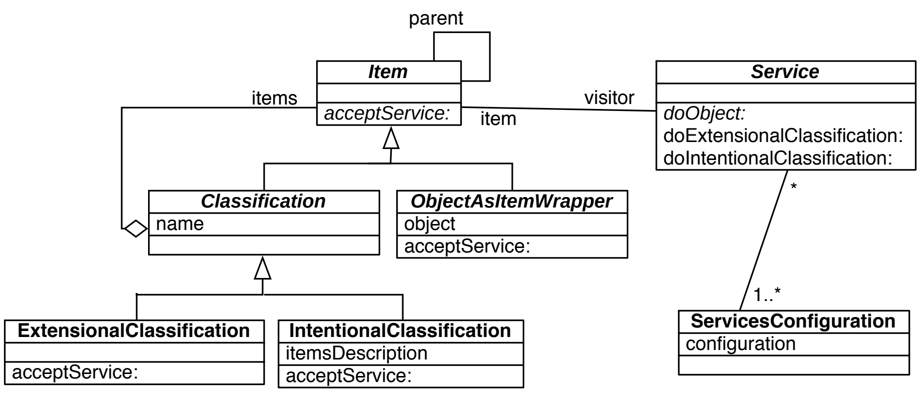


Fig. 1. Diagram showing the core concepts of the classification model: *items*, *classifications*, *services* and the *service configuration*.

- *classifications.* A *classification* is a group into which items are classified, and basically is a container for items. Items do not have to be of the same type, and it provides behaviour for enumerating and managing the items it contains. For example, a classification ‘Popular Classes’ can contain the most browsed classes or the classification ‘My Architecture’ can contain the items that make up your software architecture.
- *services.* A *service* implements an action that can be performed on items. Depending on the kind of item, a service can perform a different action. But it should at least provide a default action that is able to process any kind of item that is passed. Examples of services can be a service that gets the children of an item, a label for an item, or a preferred editor.
- *service configuration.* This is a registry where services can register under a certain name to be retrieved by tools.

There are no restrictions on the number of items in a classification and an item may be classified into more than one classification. Since a classification is an item itself, it can be part of other classifications as well. We differentiate between two kinds of classifications:

- *extensional classifications* enumerate items. For example, while browsing the code in a system, a developer can add classes and methods of interest into an extensional ‘Favourites’ classification.
- *intentional classifications* compute their items according to a description. For example, we can define a classification as consisting of a certain class and all its subclasses. Or we can describe a classification that consists of all the senders of a certain methods in the context of another classification.

Classifications support a full range of set operations (unions, subtractions and intersection) to make it easy to recompose their elements. For example, suppose that we have two classifications: an extensional classification ‘Favourites’ containing a collection of classes we are interested in, and an intentional classification ‘My classes’ that calculates all the classes in my namespace. We can then intersect ‘Favourites’ and ‘My classes’, and obtain a classification ’My favourite classes’.

Services are registered in a services configuration under a service name they specify. Anybody in need of a service retrieves it from the services configuration by its service name. Different services can register under the same name, in which case one of them becomes the ‘active’ service. When asking for a certain service, clients do not know exactly which service implementation they get. For example, they can ask for a service returning icons for items. If the services configuration has different services for returning icons, the active one will be returned. Swapping the active one for another will result in the clients using different icons without the need to change a single line of code in the client. Note that the provider of the service has to make sure that services with the same name are compatible. Unit tests are provided that enforce this compatibility between services registered with the same name.

2.2 Design Rationale

We made two important design decisions for the classification model. The most important one was to split the behaviour of items in two: the behaviour dealing with managing items (adding, removing, enumerating, ...) is implemented on the items themselves. All other behaviour regarding items (their icons, labels, editors, . . .) is implemented using the services, using the *Visitor* design pattern. There were two motivations for choosing a visitor:

- *Keep the interface small.* In the Smalltalk implementation, an Item can be any kind of Smalltalk object. Since we did not want to clutter the interface of the class *Object* with all kinds of item-specific methods, we only add one method (the method *acceptService:*) to enable a *Visitor* to implement the services.
- *Swap services at runtime.* The services can be changed at runtime, which is not possible if the behaviour is directly implemented as methods on the item classes.

A second decision was to wrap objects in a wrapper class *ObjectAsItemWrapper*, a subclass of class *Item* that implements the method *acceptService:* as follows:

```
ObjectAsItemWrapper≫ acceptService: aService
    ^self object acceptService: aService
```

Therefore any kind of object can be wrapped as an item provided it implements the method *acceptService:*, where it determines what method needs to be

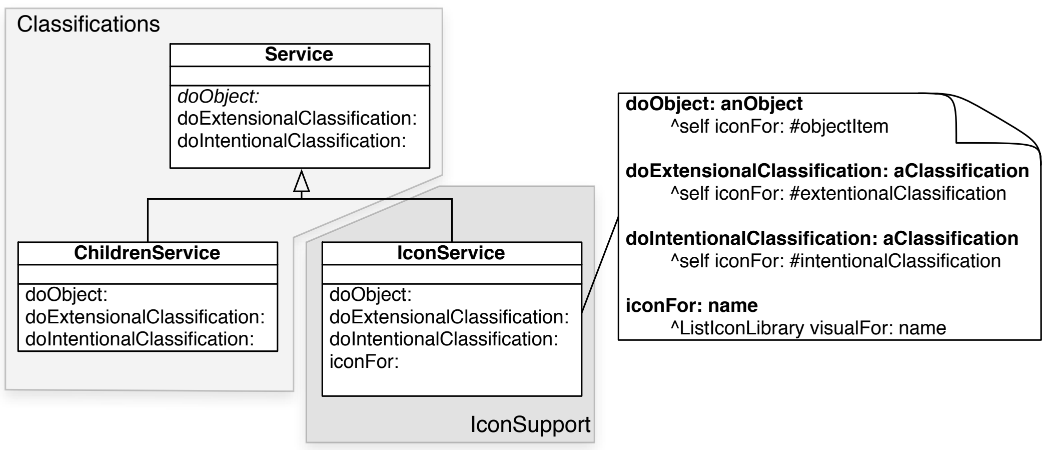

Classifications
Service
doObject:
doExtensionalClassification:
doIntentionalClassification:
ChildrenService
doObject:
doExtensionalClassification:
doIntentionalClassification:
IconService
doObject:
doExtensionalClassification:
doIntentionalClassification:
iconFor:
IconSupport
doObject: anObject
^self iconFor: #objectItem
doExtensionalClassification: aClassification
^self iconFor: #extentionalClassification
doIntentionalClassification: aClassification
^self iconFor: #intentionalClassification
iconFor: name
^ListIconLibrary visualFor: name

Fig. 2. Adding a new *Icon* service by subclassing the *Service* class. *IconService* is put in its own package (package *IconSupport*, in dark grey). Only the service classes are shown in the *Classifications* package (in light grey).

called on the service to process that item.

The result of both these design decisions is a lightweight model. The next section describes how this model can be customised to accommodate for new kinds of items and services.

3 Customising the Classification Model

The classification model described in Section 2 can be customised in two orthogonal ways: extra services can be added and new kinds of items can be supported. This section explains both of these customisations. In the next section we then see how this is used by the tool integrator to support unanticipated integration of tools.

3.1 Adding New Services

Services define the actions that can be performed on items, following the well-known *Visitor* design pattern. Because they are de-coupled from the items, new services are added by subclassing existing services, in the "classical" *Visitor* scheme. For example, suppose that we are building an application that needs to show items in some GUI, and that wants to show icons for each item. Then this application needs to know which icon to use for each item. This is done by adding a new service as subclass of the existing class Service, and overriding the methods for which icons need to be returned. Figure 2 shows

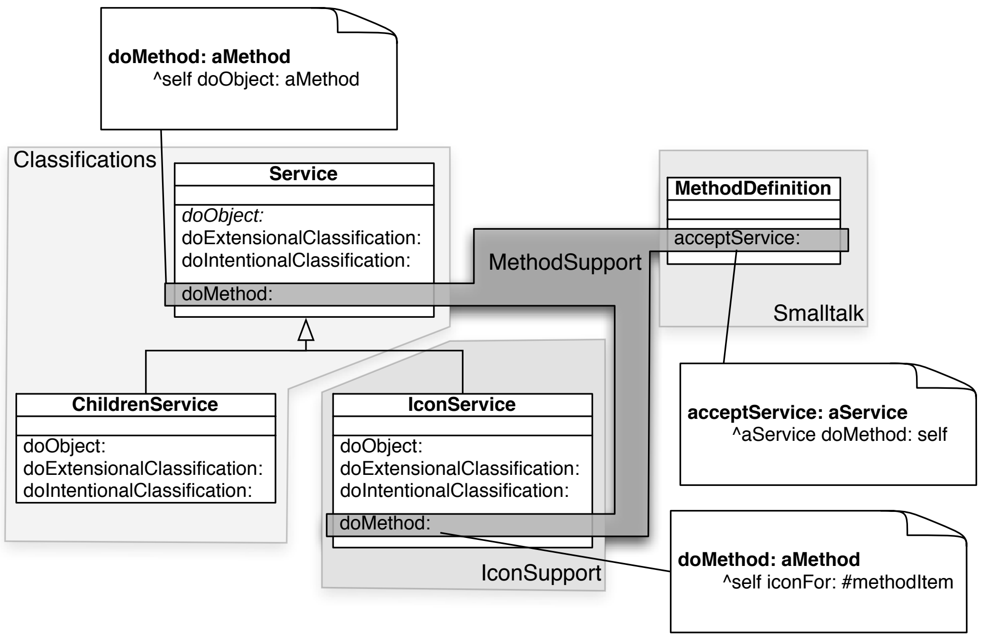


Fig. 3. *MethodSupport* is a package that extends the *Classifications* and *IconSupport* packages to add support for methods. It contains three class extensions.

the implementation needed.

3.2 Supporting Custom Items

In Section 3.1 we added an icon service to return icons for items. However, it only supports three kinds of items: objects, extensional classifications and intentional classifications. Now suppose that we want to add support for another item, say a method, so that we can return a specific icon for it. To do this, we need to do two things: extend the class implementing a method with an *acceptService:* method and extending the services that want to take advantages of methods with a new Visitor method (for example *doMethod:*). This is shown in Figure 3.

Note that the implementation of *doMethod:* on class *Service* simply calls *doObject:*. As a result, all services that do not need to support methods explicitly, will process methods as objects. For example, there is a service called *ItemChildren* that returns the 'children' of an item. Asking for the children of a method item by sending *doMethod:* to an instance of class *ItemChildren* will result in sending *doObject:* to that instance, hence processing a method as a generic object.

4 Unanticipated Tool Integration

Sections 2 and 3 introduced the classification model and showed how it can be customised to support new kinds of services and items. In this section we show how this is used to integrate the models of tools that were not designed to cooperate together.

4.1 Integration Overview

As said in the introduction, we consider tools to be integrated when the output of one tool can be used as the input for another tool. To solve the problem of unanticipated integration, somehow a format has to be introduced to integrate tools, but without changing these original tools. The idea is to use the classification model as a first-class glue format to bridge the tools to be integrated. Using the classification model for this purpose has the advantage that items are manipulated uniformly (meaning that making the output of one tool 'compatible' with the classification model immediately allows that tool to work with all tools that have compatible input). Another important advantage is the set operations available on classifications, meaning that once a tool is integrated it can be used to calculate unions or differences with results from other tools. While trivial at first, this makes it easy to make semantic operations that combine the output of different tools.

To integrate tools using the classification model, tool output has to be made tangible as items and a translation has to be provided from items to whatever is needed as input:

- *Disguise output as items.* The output of a tool, some object, has to be tangible as an item. If the classification model already knows about that object, nothing needs to be done. If the classification model has no support for that kind of object, it needs to be customised. For example, suppose that we have a tool (the Smalltalk system itself) that produces classes as output. Since the classification model does not know about classes, we extend it. As described in section 3, this boils down to adding the method *acceptService:* on the class *ClassDescription*, and adding the method *doClass:* on the class *Service*.
- *Create service for input.* To make a tool work with items as input, we create a service that translates items to the input needed by the tool. For example, as we will see when integrating the UML editor *Advance*, we create a service that maps items to so-called 'subjects', used internally by *Advance*.

It is important to note that the tools themselves do not need to be changed. It is just the classification model that gets customised. Moreover, the responsibility for integrating the tools does not lie with the tool developer, but with the tool integrator. This is a major difference between the approach allowed by the classification model and an up-front integration architecture that tools should comply with.

4.2 Example: Integrating Advance *and* SmallBrother

We illustrate the unanticipated integration of tools with a concrete example showing how to combine the output of *SmallBrother*, a coding assistant that tracks browsing behaviour, with *Advance*, a UML tool developed by IC&C and shipped with the VisualWorks Smalltalk environment [2].

SmallBrother. *SmallBrother* tracks the methods browsed by a developer while working. Every-time a method is selected, this is intercepted and the class, selector, and a time-stamp are kept in a database. This database can then be queried for information regarding the history and browsing behaviour. We can for example evaluate the following piece of code to get the 20 most recent methods that were browsed:

```
MethodHistory uniqueInstance recentMethods: 20
```

Other things we can ask for are, for example, the number of times a method was browsed, or what classes have been used a lot. The results of these queries are collections of objects. Therefore the mapping is easy in this example: an intentional classification is used that computes its items:

```
IntentionalClassification name: 'Recent Methods'
                           description: [MethodHistory uniqueInstance recentMethods: 20]
```

Advance. *Advance* is a UML tool for the VisualWorks Smalltalk environment. To let *Advance* work with items, we create a new service, class *AdvanceEditor*, with the methods shown in Figure 4. This class implements methods that visit items (*doExtensionalClassification:*, that calls *doClassification:*, and *doParcel:*). These methods convert items to an internal *Advance* representation, the *subject*. The other two methods, *createSubjectForItem:* and *doForSubject:* are auxiliary methods that generate a subject (which is a class) on the fly, and open an *Advance* diagram tool on the subject.

We want to stress that we are not the developers of the *Advance* tool, and that none of the existing *Advance* tools had to be changed to make them classification model compatible. As tool integrators we just implemented some glue code to convert items to subjects, and pass these subjects to *Advance*.

```
AdvanceEditor≫ doParcel: aParcel
   "Perform the service defined by the receiver on the pundle passed as argument."
   | subject |
   subject := self createSubjectForItem: aPundle add: aPundle definedClasses.
   ^self doForSubject: subject

AdvanceEditor≫ doExtensionalClassification: aClassification
   "Perform the service defined by the receiver on the classification passed as argument."
  ^self doClassification: aClassification

   AdvanceEditor≫ doClassification: aClassification
   "Perform the service defined by the receiver on the classification passed as argument."
   | classesInClassification subject classItemsInClassification |
   classItemsInClassification := (aClassification select: [:item | item isBehavior]) items.
   classesInClassification := classItemsInClassification collect: [:item | item unwrappedItem].
   subject := self createSubjectForItem: aClassification add: classesInClassification.
   ^self doForSubject: subject

AdvanceEditor≫ createSubjectForItem: item add: classes
   "Generate a subject class to make Advance happy, and add the classes to it."
   |  subjectClass subject subjectName |
   subjectName := self subjectNameForItem: item.
   subjectClass := self createSubjectClassNamed: subjectName.
   subject := self advance subjectEnvironment makeClass: subjectClass
subjectNamed: nil.
   subject addClasses: classes.
   ^subject

AdvanceEditor≫ doForSubject: subject
   "Build an Advance diagram using the given subject, and open a painter for it.
       Use this painter as the new editor and return it."
   | diagram diagramName painter |
   diagram := subject fakedDiagram.
   diagramName := diagram name = self advance diagramSpecClass fakeName
          ifTrue: [nil]
          ifFalse:
                    [self diagramSelectionIsValid ifFalse: [^self updateWithNotification].
                               diagram name].
     painter := self advance diagramPainterClass new.
     self doForEditor: painter.
     painter openDiagramOn: subject name: diagramName.
     ^painter
```

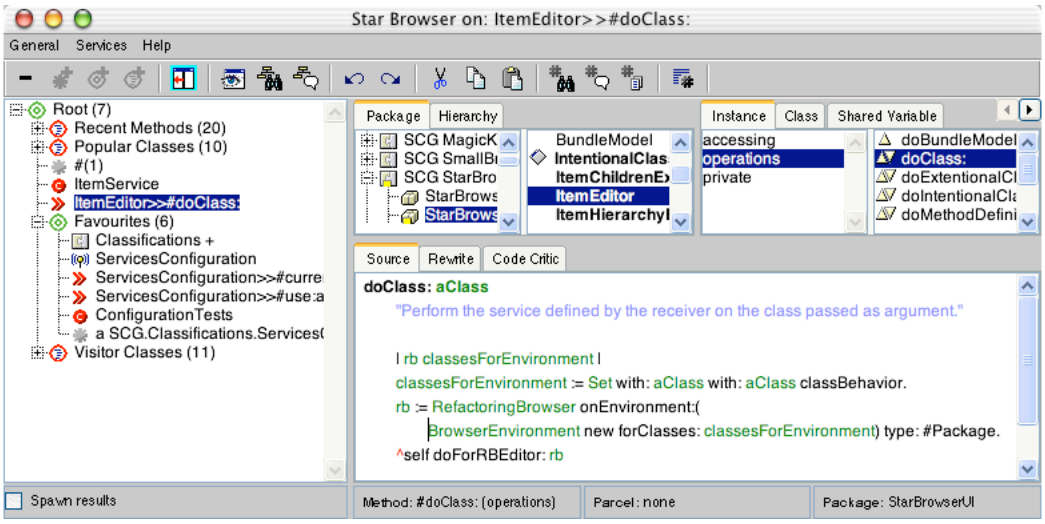
Fig. 4. The implementation of the class *AdvanceEditor* that allows one to use the *Advance* UML tool with items.

5 The StarBrowser

The main application of the classification model is the StarBrowser. The StarBrowser is a VisualWorks Smalltalk [2] [2] development browser. By itself it provides only a toolbar, an interface to display classifications as a tree, a part where editors for these items can be shown, and a mechanism to allow a user to switch services using the ServicesConfiguration. It extends the classification model to support classes, methods, namespaces, packages, bundles, and parcels. All its other functionality is implemented in a number of services:

• editor: The editor service is responsible for adding an editor on the currently

[2] See http://www.cincom.com/scripts/smalltalk.dll//Home.ssp for more information and downloads.



```
doClass: aClass
    "Perform the service defined by the receiver on the class passed as argument."

    | rb classesForEnvironment |
    classesForEnvironment := Set with: aClass with: aClass classBehavior.
    rb := RefactoringBrowser onEnvironment:(
        BrowserEnvironment new forClasses: classesForEnvironment) type: #Package.
    ^self doForRBEditor: rb
```

Fig. 5. StarBrowser in VisualWorks editing a method selected in the classifications tree.

selected item in the classifications list. This editor is embedded on the right of the classifications list. It integrates tools to edit all kinds of source code entities with the Refactoring Browser [10] and objects with the *Trippy* object inspector. The application returned by the editor service is integrated in the StarBrowser using VisualWorks' subcanvas technology. The toolbar of the editor (if there is one) is merged with the toolbar of the StarBrowser.

- icon: The icon service is responsible for showing the icon of an item in the classifications tree.
- label: The icon service is responsible for showing the label of an item in the classifications tree.
- menu: The menu service is responsible for returning the operate menu that users get when they right-click on an item in the classifications list.

Figure 5 shows the StarBrowser in action. The left tree view shows the classifications tree that this browser is opened on. Right of the tree is an editor for the currently selected item, as given by the editor service. It currently shows a Refactoring Browser on the selected method item in the tree.

Extensional classifications are manipulated using drag'n'drop: items are just dragged from any kind of Smalltalk tool (stand-alone or embedded in the StarBrowser) and dropped at their desired location. That way extensional classifications are used to group items of interest. The extensional classification 'Favourites' from Figure 5, for example, groups items we used while working on the *ServicesConfiguration* class. It contains the *Classifications* bundle, the *ServicesConfiguration* class, two methods we were frequently editing at the time of taking the screenshot, the SUnit class with the unit tests for the class, and an instance of *ServicesConfiguration* so that we could directly test new implementations. Keeping all these items together helps to reduce the complexity of the development process.

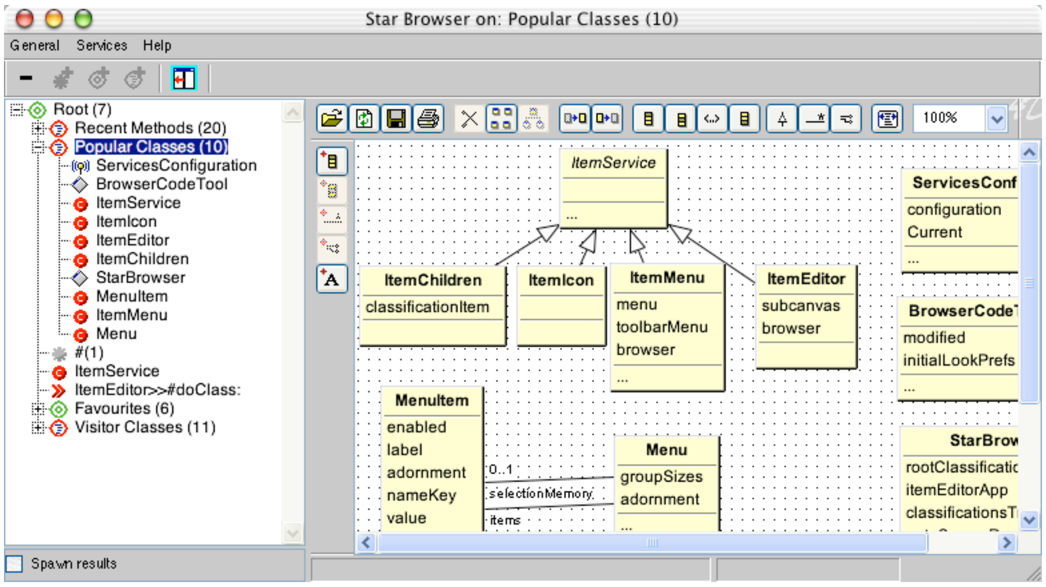


Fig. 6. StarBrowser showing the *Advance* UML Editor on the Popular Classes classification. This shows how the results from one specialised tool (*SmallBrother*) can be used by another one (*Advance*) by integrating them using the classification model.

Besides being used for constructing working views on a system, classifications are also used as working contexts for some widely used commands. For example, the senders or implementers of a method can be looked for within the context of a classification.

5.1 Advance *and* SmallBrother *in the StarBrowser*

As shown in Section 4, we integrated the *Advance* UML tool in the classification model. The reason for doing so was that we could show a class diagram for any kind of item that gets selected in the classifications tree. For example, selecting a namespace shows a class diagram for all the classes in this namespace. Or showing a classification shows all the classes in that classification.

We also integrated *SmallBrother*, so that we could add the history information as classifications in the tree. We decided to use intentional classifications that calculate their items by querying the *MethodHistory* instance, as shown in Section 4. However, we wanted this classification to refresh itself whenever the user browses a method. So we made a subclass (called *ObservingClassification*) that observes models using the VisualWorks dependency mechanism. Since MethodHistory is a model, it can be observed by an ObservingClassification and it will refresh when needed.

Once the service for *Advance* is selected as the current editor, and we have added one of the classifications that wraps *SmallBrother*, the tools are effectively integrated. Figure 6 shows an *Advance* diagram of the Popular Classes classification. Of course this means that *Advance* can be used to display class diagrams on all kinds of items, and that lots of other tools can take advantage of this.

5.2 Other Tools

We showed how we integrated two external tools, *Advance* and *SmallBrother*, in the StarBrowser. Besides these tools, other tools were integrated in the StarBrowser (by us or by independent parties):

- *CodeCrawler* [7] is a language independent tool that combines software visualization and software metrics to help with the understanding of software systems. We also integrated a tool that shows *class blueprints* [8] (visualising the internals of classes) whenever items containing classes are edited.
- *Conan* [1] is a tool that supports *concept analysis* (a technique to group different objects with common relationships) in the context of reengineering of software systems. The StarBrowser is used to browse the concepts and elements found by Conan.
- *Intentional Software Views* offer a simple, intuitive and lightweight model that facilitates software understanding and maintenance. The model is implemented in a logic programming language. To shield the developers from the implementation details or syntactic peculiarities that this implies, an intuitive user interface was developed using the StarBrowser [9] .
- *Soul* is a logic programming language living in symbiosis with its implementation language (Smalltalk). The novel way of integrating these two languages from different paradigms allows one to write logic programs that can do full logic reasoning on and using objects [11]. The StarBrowser was extended with support for showing the results of Soul queries as classifications, and work is in progress to integrate the Soul predicates browser in the StarBrowser as well.
- *Pictures.* Another extension integrates a picture viewer in the StarBrowser that shows pictures graphically when they are selected. This is useful for keeping graphical information in your work context, or when making presentations.
- *SUnit* is the Unit testing tool in Smalltalk. Using the StarBrowser, an advanced GUI for SUnit is being built to ameliorate the built-in platform-independent tool.

6 Discussion

This section we discuss the porting of the StarBrowser to a completely different Smalltalk enviroment (the Squeak environment), and then how the Smalltalk mechanism of class extensions is essential in the implementation of the classification model.

6.1 The Squeak Port

The VisualWorks implementation was ported to a second Smalltalk environment, the open-source Squeak[3] system [6]. After doing an initial port, containing the model and a very simple browser, the Squeak version was taken over by another developer, Ned Konz. In a couple of days, Ned had significantly extended our initial crude implementation to a level where it nearly provided the same functionality as the VisualWorks version. Afterwards support was added for Squeak-specific items like *Morphs*, *SqueakMap entries* and *DVS packages*, and tools like an e-mail browser. This is an indication of the ease with which the lightweight classification model can be put to good use even by developers who did not know the model before.

6.2 Packaging using Method Additions

As explained before, we identified three different actors that play a role when integrating tools: the tool builder, the tool integrator and the model builder. In order to support this separation in practice, it is absolutely necessary that each actor can package its own code separately. Hence the tool integration package has to be a separate entity that customises the classification model.

However, customisations of the classification model to support new kinds of items depend on adding methods to existing Service classes. For example, Figure 3 showed that the customisation of the classification model to support methods is done by adding three methods to existing classes. Hence we need a packaging mechanism that allows us to create a package for these methods.

Smalltalk has a package mechanism that supports *method additions*. A method addition is a method that is defined in a package, but that belongs to a class that is not defined in that package[4]. In other words, it is a method that can

[3] See http://www.squeak.org/ for information and downloads.
[4] In VisualWorks we use *parcels* or *bundles and packages*. In Squeak we use *changesets*. They all support class extensions.

be loaded into a system to extend some existing class, and is exactly what we need to support the packaging customisations of the classification model.

When a language does not support method additions (such as for example C++ or Java), the design of the classification model becomes much more complicated. The visitor pattern used for the services then has to be replaced by a design that allows customisations to be made purely by subclassing or delegation.

7 Related Work

Conceptually, the classification model is a direct descendant of the *software classification model* [5]. The main difference is that the classification model uses a visitor to represent the actions that can be performed on items, which did not exist in the *software classification model.* As a result, it is easier to add operations, and services can be changed on the fly.

A lot of environments offer integration features that tools can use to integrate with the environments and/or each other. For example, the Microsoft Management Console (MMC)[5] integrates management tools in Windows. The tools have to be developed as snap-ins, and cannot work as stand-alone applications. Another example is *Chandler*[6], a tool to let users store and organise diverse kinds of information (like e-mails, news, or mp3 files). It is set up as an extensible platform, where users can contribute so-called *parcels.* Parcels are python scripts that can use the facilities of Chandler. The major difference with the approach taken by these environments and our approach is that they support anticipated integration, and that the tool developers have to make their tools compliant to the architecture, not the tool integrators.

Regarding unanticipated tool integration, not much work seems to have been done. Apart from the Eclipse IDE [3], we are not aware of another model that supports unanticipated tool integration that does not require the tool developers themselves to make changes.

The Eclipse environment follows the same concept as the StarBrowser. It consists of a tiny core (the plug-in loader), with most of the environment contributed by plug-ins. For example, the Eclipse IDE and the Java Development Environment together consist of around 60 large plug-ins. Plug-ins for Eclipse have to conform to certain interfaces, and are glued together through *extension points.* These extension points are basically observers, and plug-ins are

[5] http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/winxppro/proddocs/sag_MMCConcepts0_0.asp
[6] http://www.osafoundation.org/Chandler_Compelling_Vision.htm

thus integrated using a synchronous message passing model. This differs a bit from the StarBrowser and the classification model, where items and services are used, and class extensions allow for the customisation. While intrinsically there is thus not much difference between the two, in practice it is much easier to customise the classification model than to write a Eclipse plug-in. The reason is that much more coding is needed to implement extension points, because the events have to be implemented in such a way that they are not blocked, do not lead to errors, etc. In the classification model, the customisation is much simpler.

8 Conclusion

This paper tackled the problem of unanticipated integration of tools, where tools that were not designed to cooperate should be integrated. To solve this problem, the paper presented the *classification model* and showed how this model can be used for unanticipated tool integration. The advantages of the classification model are that it is lightweight, which make it easy to extend, and that the integration does not need to modify the tools themselves. This is achieved by 'glue code' that sits between the tools and expresses the mapping to and from items, the foundation of the classification model.

We showed the Smalltalk implementation of the classification model and its major client, the StarBrowser, an extensible browser that integrates different tools. By means of concrete examples we showed how the StarBrowser uses the classification model to integrate with existing Smalltalk development tools and third party tools like the Advance UML tool or the *SmallBrother* coding assistant.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1). We also like to thank Koen De Hondt for valuable comments on classifications in general and on this paper in particular, and the StarBrowser users for their comments and support.

References

[1] Gabriela Arévalo. Understanding behavioral dependencies in class hierarchies using concept analysis. In *LMO 03: Langages et Modeles a Objets*. Hermes, 2003.

[2] Visualworks application developer's guide, 2002. Cincom.

[3] Eclipse Platform: Technical Overview, 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[5] Koen De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. Ph.D. thesis, Vrije Universiteit Brussel,Departement of Computer Science, Brussels - Belgium, December 1998.

[6] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326, November 1997.

[7] Michele Lanza. Codecrawler - lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, page to be published. IEEE Press, 2003.

[8] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

[9] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proc. of SEKE 2002*, pages 289–296. ACM Press, 2002.

[10] 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.

[11] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.