# Class-management using Logical Queries, Application of a Reflective User Interface Builder

**Roel Wuyts**
Programming Technology Lab
Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium
E-Mail : rwuyts@is1.vub.ac.be
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

January 15, 1996

## Abstract

Current browsers for object-oriented languages suffer from restricted query capabilities that only allow for class-oriented views on the classsystem. As a result, browsers are very poor in providing support for software engineering techniques that go beyond single classes, like frameworks, contracts and design patterns. This paper proposes the combination of a logical query language and user interface components as a foundation for non class-oriented, customizable browsers able to support recent and future object-oriented software engineering techniques. Validation of this proposal is done by building browsers for different domains in Smalltalk.

## 1 Introduction

Recent programming environments use browsers to permit browsing and editing of source code. Where in classic imperative programming languages like Pascal a simple editor suffices, object-oriented languages need more sophisticated tools due to the scattering of source-code all over the class-system. In such systems, the browser is the key to unlock the world of object-oriented programming.

Since software engineering techniques in the past were essentially class-based, so were browsers. The primary static relation between classes, i.e. inheritance, was also supported. However, recent techniques are shifting from single classes to more elaborate relations between classes, for example frameworks, contracts and design patterns [6, 5, 7, 9]. Current class-based browsers fail to accommodate these new insights, due to two problems. The first is the lack of a sophisticated query system that enables queries ranging over the whole class-system. The second problem is the lack of customizability of the queries and of the user interface used to present their results.

To address the raised problems, this paper proposes the use of a logic programming language as query-mechanism for questioning the class-system, and custom user interface components for building the user interface. The logic programming language enables strong queries, and cannot only range over classes, but over the full class-system including instance and class variables, methods and user-defined facts. It is also explicit, giving the user the power of adding facts and rules, and using own queries. The custom user interface components are used as pre-fabricated building blocks that are easily adapted using a reflective user interface builder.

VisualWorks Smalltalk [4, 8] was used as the programming environment for validating the proposed mechanism. First, a logic programming language was implemented to serve as query language. Then the already existing reflective User Interface Builder ApplFLab was used to build the custom user interface components. Afterwards, several browsers were constructed to browse classes and more complicated structures.

This paper is organized as follows. The two following sections will further introduce the two basic concepts in more detail. The next section is concerned with high-end customizable browsers. The last topic covered before the conclusion demonstrates the use of our concepts in building a browser for the Bridge design pattern.

# 2 The Logic Programming Language

Logic programming languages are declarative and multi-directional languages using logical terms to express facts, rules and queries. Facts and rules are used to write down information, while the queries allow to question this information. Using a logic programming language to express queries has the advantage that, although the queries are very powerful, the language is easy to understand and use.

Implementations of logic programming languages use the SLD-resolution algorithm to implement the inference mechanism that takes care of handling the queries. More elaborate information about use and implementations of logic programming languages can be found in [2, 10]. A small logic programming language based on the approach used in [1] was implemented in Smalltalk to use as query mechanism. We will first give some example facts, rules and queries that demonstrate the basic functionality[1]. To begin with, we will add facts for every class we want to take in account by giving the name of the class, the name of the superclass and an identifier (%classIncluded%) :

```
Dictionary Object %classIncluded%
Collection Object %classIncluded%
OrderedCollection Collection %classIncluded%
MySpecialCollection OrderedCollection
%classIncluded%
...
```

This adding of facts is only necessary because of the decision that was taken to separate the logic programming language from the Smalltalk class system. This separation ensures the generality of the query language, since it is not based on a specific language or class-system. We will now write a very simple rule to describe what a class is, and when a class is a direct subclass :

```
isClass (?class) = ?class ?X %classIncluded%
isDirectSubclass (?class ?super) = ?class ?super
%classIncluded%
```

In these rules, finding values for the variables simply comes down to matching patterns. The next rule that describes a hierarchy of classes is already more interesting :

```
inHierarchy(?root ?class) = isDirectSubclass (?root
?class) #or
(isDirectSubclass(?root ?class-super) #and inHierarchy(?root ?class-super))
```

This rule features inference, brackets, the logical operators #and and #or and recursion. Besides these facts concerning the class-system itself, users are also able to add facts specific for their situation, such as for example

```
MySpecialCollection author Mike
MySpecialCollection version 5 sub 8
MySpecialCollection not-tested
```

Having defined some facts and rules, it is time to ask some queries. To get for example a list of all the classes we can simply pose :

```
isClass(?X)
```

The different values for X will be the classes present. Facts and rules can be also be combined :

```
isDirectSubclass (OrderedCollection ?class) #and
?class author Mike
```

Given the facts present this query will return the name 'MySpecialCollection'.

As can be seen, the implemented language is rather classic. An exception is the feature that makes it possible to use Smalltalk blocks as a predicate for rules or queries. This is the only place where Smalltalk can be used in the logic programming language. Such use of a Smalltalk block takes the form of

```
[[Smalltalk block] arguments ]
```

the arguments being logical variables. To demonstrate how a Smalltalk block could be used as predicate, we make a rule for abstract classes that states that a class is abstract if it has at least one method that returns 'self subclassResponsibility' (as is common practice in the Smalltalk community) :

```
abstract(?abs) = isClass(?abs) #and [[:absName
((Smalltalk at: absName)
whichSelectorsReferTo: #subclassResponsibility)
isEmpty not] ?abs]
```

Almost the same result could be accomplished by adding a fact and a rule of the form :

```
name-of-class %abstract%
abstract(?abs) = isClass(?abs) #and ?abs %abstract%
```

The former formulation has the advantage of being smaller, since in the latter facts need to be supplied for every abstract class. However, the latter is more general for two reasons. First of all, it is independent from Smalltalk since it only uses logical facts and rules. Secondly the smalltalk block is just a predicate, and thus serves only as a filter to reject some elements and keep others.

The next section will introduce the custom user interface components and the user interface builder that is used to create and manipulate these components, ApplFLab.

# 3 The User Interface Components

The reflective user interface builder (UIB) used is ApplFLab (Application Framework

[1] a note concerning the notation : variables are directly preceded by a question mark, the logical operations are #or and #and

Laboratory), a UIB based on Parcplace's VisualWorks. Although the VisualWorks UIB is a proven development tool that is well integrated with the underlying Smalltalk development environment, it lacks a profound mechanism for reusing user interface components [12, 11]. ApplFLab provides this reuse ability through user interface components, applications in which part of the domain knowledge has to be specified when the component is used in an application. User interface components can be nested, resulting in larger components for which again specifications can be given. Take for example an application that displays a list, and then waits untill the user presses a continue button beneath the list. Reusing this application is fine, but not on an as-is basis : it might be necessary to change the label of the button to OK. ApplFLab provides user-friendly tools that enable the programmer of the user interface component to express that the label of the button should be filled in when the user uses the component.

For the tools and browsers described in the next sections, two layers of components were constructed. First, some base components were made (lists, buttons, text fields and text editors), based on existing VisualWorks components, but with simple input/output behavior making it easy to link components with one another. More advanced components were built using these base components, representing higher level reusable parts of browsers. The most important of these components are the QueryList, Classlist, MethodList and EditSpace. These components can be used as prefabricated browser building blocks or can be adapted to address more specific needs.

Using the basic components, tools were build to add, change and remove facts, rules and queries. Since these tools were constructed with the user interface components, they can easily be adapted to the taste of the user.

The next sections show the combination with the logic programming language on two domains. To begin with, class browsers are made that are far more powerful than the browsers that are standard provided, thus showing the validity of the concept. Afterwards a browser for a design pattern is made, showing how new programming techniques can be supported by browsers using the combination of logic query language and user interface components.

# 4 Class Browsers

The first class browser built was a simulation of the System Browser, which is a standard tool in the VisualWorks Smalltalk environment that enables the programmer to have a look at all the classes available, their definition and their methods (see figure 1). This browser can be simulated using only four queries and five user interface components, thus showing the generality of the concept.

Next a simple browser was build that enables to walk through the class-system by applying queries, and includes a back-track facility (see figure 2). The facts and rules are those used in the logic programming language section. The idea is to provide a standard set of facts, rules and queries, and let the user extend or modify these, using the tools described in the previous section to tailor the functionality of the browser. One can think of information concerning versions, authors or frameworks. Customized queries can then immediately be formulated and used, for example

isClass (?class) #and abstract(?class) #and framework(?class BrowserFramework)

that returns all the abstract classes in the framework 'BrowserFramework', given the two rules abstract and framework that respectively return whether ?class is abstract and whether or not it belongs to a certain framework.

The programmer is now able to paint a browser using some of the browser user interface components. The result is then a class-oriented browser with customizable functionality - new facts, rules and queries can be edited and applied - and customizable interface.

To demonstrate the extendibility, the simple browser was extended to take methods into account. This merely comes down to adding facts of the form

class methodName %methodIncluded%

and some more rules

hasMethod(?class ?method) =
?class ?method %methodIncluded%
sameProtocol (?class1 ?class2 ?methods) =
hasMethod
(?class1 ?methods) #and hasMethod(?class2 ?methods)

This last rule can be used to compare methods from classes. This is very important in method-oriented browsers to enable the comparison of protocols of classes.

Once the functionality is extended by supplying facts, rules and queries that take methods into account the interface can be extended with for example a MethodList component. The resulting browser is depicted in figure 3.

The resulting browser demonstrates that the customizability offered by combining our explicit logic programming language and user interface components is endless. Next section will demonstrate this by constructing a browser for the bridge design pattern.

# 5 Bridge Pattern Browser

Design patterns are solutions to common design problems that have evolved over time and are elegant and well-designed [9, 3]. An example is the bridge design pattern, that describes a system in which an abstraction is decoupled from its implementation such that both can vary independently. This gives rise to a abstraction hierarchy, with as root the so called 'abstraction class', and a implementation hierarchy where the 'implementor class' is the root. The abstraction and implementor class are bridged by an aggregation relation. This aggregation is the first part of the bridge pattern, and can be implemented in different ways : using an instance variable, a dictionary with associations between abstraction classes and implementor counterparts, or a global variable. For this different implementations, different 'types' of aggregation were defined : instance-variable, dictionary and global-variable. The second part of the bridge pattern is formed by the methods of the abstraction class, the protocol. Methods of this protocol are used on the implementor side to implement operations.

Current programming environments provide almost no tools that support new programming techniques like design patterns. To demonstrate that a logic query language and user interface components can be used to create browsers that support such techniques, a browser for the bridge pattern was build. Assuming that we have the facts and rules of the previous section, only one kind of fact is necessary to obtain such browser, i.e. for each bridge design pattern used we state the following :

BridgePattern Example Window Xwindow instance-variable myReference

This fact defines a bridgePattern with name Example, using Window as the abstraction class, Xwindow as the implementor class, and an aggregation of type instance-variable using myReference to do the reference. We can then define some rules that facilitate working with this fact :

allBridgePatterns (?bridge) = BridgePattern ?bridge ?abstraction ?implementor ?inst ?ref
bridgeParticipants (?bridge ?abstraction ?implementor) = BridgePattern ?bridge ?abstraction ?implementor ?inst ?ref
bridgeAggregation (?bridge ?inst ?ref) = BridgePattern ?bridge ?abstraction ?implementor ?inst ?ref

We can now obtain the protocol of the bridgepattern, the abstraction and implementorhierarchies and the used reference types using following rules :

protocol (?bridge ?prot) = bridgeParticipants (?bridge ?abstraction ?implementor) #and protocol (?abstraction ?prot)
abstractionHierarchy (?bridge ?absClass) = bridgeParticipants (?bridge ?abstraction ?implementor) #and inHierarchy (?abstraction ?absClass)
referenceTypes (?type) = bridgeAggregation (?bridge ?type ?ref)

The user interface that was constructed for this browser resembles the OMT-like scheme given in [3]. It is shown in figure 4. This example shows how creating a highly sophisticated browser can be done using just one fact, some rules, and a user interface built using some custom components.

# 6 Conclusion

To address two problems faced by browsers in recent programming environments, the absence of a sophisticated query language and the lack of customizability, this paper proposes the use of a logical query language and custom user interface components. Not only does this combination prove to be powerful thanks to the logic programming language, the open-endedness ensures support of different programming techniques. To claim this statement, browsers were build in Smalltalk that demonstrate the power and customizability on different domains. Such browsers are not only keys to unlock the rich world of object-oriented programming, they are the master key to open just those doors the programmer wants to enter.

# 7 Acknowledgements

I wish to thank following persons for their important contributions that made this work possible : prof. dr. Theo D'Hondt, dr. Patrick Steyaert, Serge Demeyer, Koen De Hondt, Wim Codenie and Carine Lucas.

# References

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and interpretation of*

*computer programs.* MIT Press, Cambridge, 1985.

[2] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, Reading, Massachusets, 1995.

[4] P.D. Gray and R. Mohamed. *Smalltalk-80 : A Practical Introduction.* Pitman, 1990.

[5] R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural composition in object-oriented systems. pages 169–180. OOPSLA-ECOOP'90, ACM Press, 1990. New-York.

[6] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1, 1988.

[7] R.E. Johnson and V.F. Russo. Documenting frameworks using patterns. pages 63–76. OOPSLA, ACM Press, 1992. New-York.

[8] Parcplace Systems. *VisualWorks Tutorial*, 1992.

[9] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, Reading, Massachusets, 1994.

[10] L. Sterling and E. Shapiro. *The art of Prolog.* The MIT Press, Cambridge, 1988.

[11] Patrick Steyaert, Koen De Hondt, Serge Demeyer, and Niels Boyen. Reflective application builders. In Chris Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press Inc, Boca Raton, Florida, 1996.

[12] Patrick Steyaert, Koen De Hondt, Serge Demeyer, and Marleen De Molder. A Layered Approach to Dedicated Application Builders Based on Application Frameworks. In D. Patel, Y. Sun, and S. Patel, editors, *Proceedings of the 1994 International Conference on Object-Oriented Information Systems*, pages 252–265. Springer-Verlag, 1995.

# 8 Figures

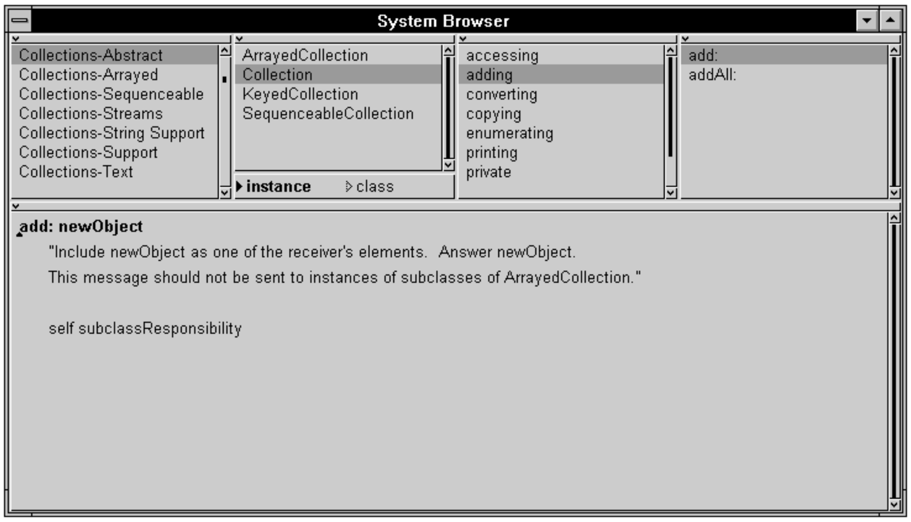


Figure 1: SystemBrowser.

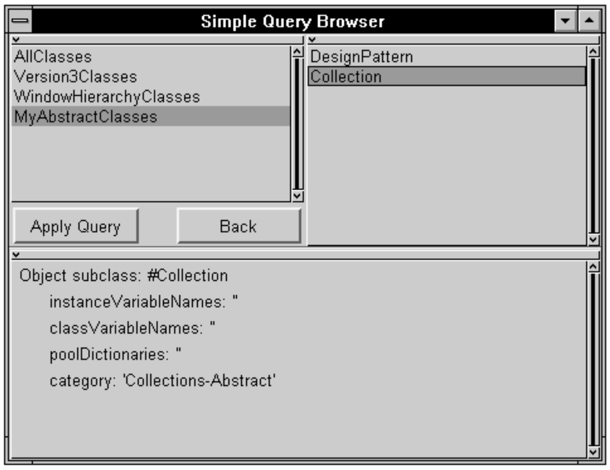


Figure 2: SimpleQueryBrowser.

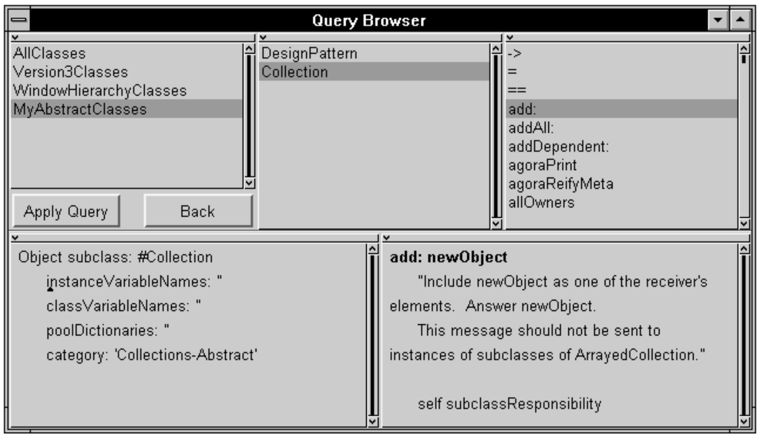


Figure 3: QuerySystemBrowser.

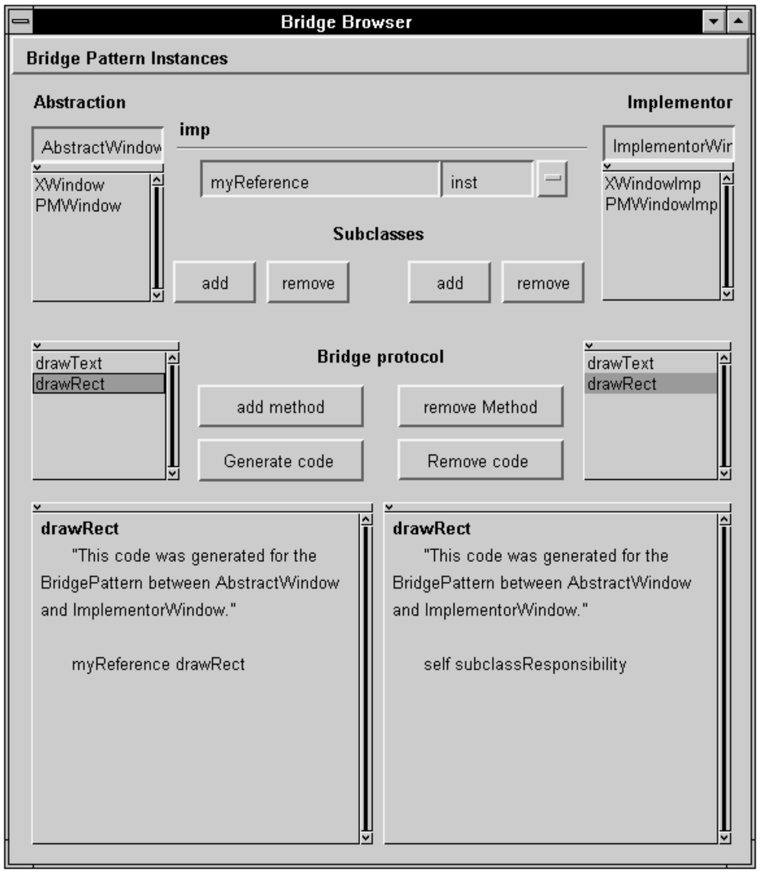


Figure 4: BridgePatternBrowser.