JQuery: A Generic Code Browser with a Declarative Configuration Language

Kris De Volder

University of British Columbia, Vancouver BC V6T 1Z4, Canada,
`kdvolder@cs.ubc.ca`,
WWW home page: `http://www.cs.ubc.causers/~kdvolder/`

Abstract. Modern IDEs have an open-ended plugin architecture to allow customizability. However, developing a plugin is costly in terms of effort and expertise required by the customizer. We present a two-pronged approach that allows for open-ended customizations while keeping the customization cost low. First, we explicitly limit the portion of the design space targeted by the configuration mechanism. This reduces customization cost by simplifying the configuration interface. Second, we use a declarative programming language as our configuration language. This facilitates open-ended specification of behavior without burdening the user with operational details.

keywords: integrated development environment, program database, domain-specific language, logic programming.

1 Introduction

Customizability and extensibility are important design goals for modern IDEs. In a typical IDE we discern two levels of customizability. The first level is provided by GUI controls such as preference panes. These customizations are cheap[1]. Unfortunately their range is limited: one can chose between a finite set of predefined behaviors but one cannot define new behavior.

The second level of customizability is through a plugin architecture. For example, the Eclipse IDE plugin architecture allows one to implement IDE extensions in Java™ and dynamically link them with the IDE. Other modern IDEs offer similar mechanisms. Plugins add executable code to the IDE so new behavior can be defined. Unfortunately the cost of customization is comparable to developing a small GUI application.

We will refer to a customization mechanism's trade-offs, between open-endedness and customization cost, as its *customization-cost profile.* We argue that IDEs provide two extreme customization-cost profiles: one is open-ended

[1] We are interested in the cost in terms effort and expertise required of a customizer. In this paper, words like "cost", "cheap" and "expensive" should be interpreted in this sense.

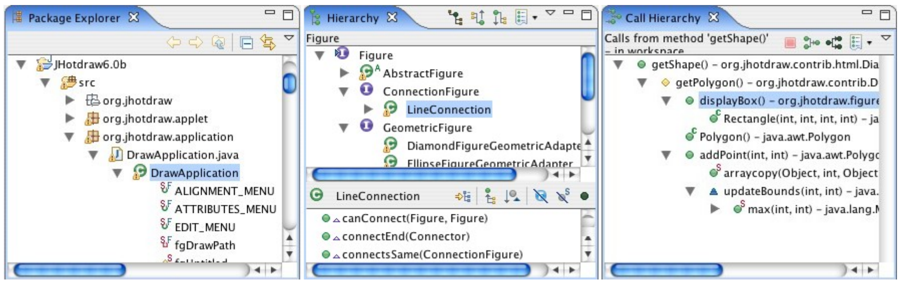

Package Explorer
JHotdraw6.0b
src
org.jhotdraw
org.jhotdraw.applet
org.jhotdraw.application
DrawApplication.java
DrawApplication
ALIGNMENT_MENU
ATTRIBUTES_MENU
EDIT_MENU
fgDrawPath
Hierarchy
Figure
Figure
AbstractFigure
ConnectionFigure
LineConnection
GeometricFigure
DiamondFigureGeometricAdapte
EllipseFigureGeometricAdapter
LineConnection
canConnect(Figure, Figure)
connectEnd(Connector)
connectsSame(ConnectionFigure)
Call Hierarchy
Calls from method 'getShape()' – in workspace
getShape() – org.jhotdraw.contrib.html.Dia
getPolygon() – org.jhotdraw.contrib.D
displayBox() – org.jhotdraw.figure
Rectangle(int, int, int, int) – ja
Polygon() – java.awt.Polygon
addPoint(int, int) – java.awt.Polygo
arraycopy(Object, int, Object
updateBounds(int, int) – java.
max(int, int) – java.lang.M

Fig. 1. Different Eclipse JDT Code Browsers

and expensive, the other is cheap but limited. We claim that it is also possible and useful to design configuration mechanisms with cost profiles in between these two extremes.

We present an approach that is flexible enough for open-ended customizations while remaining relatively cheap. There are two key ingredients. First, we explicitly limit ourselves to a portion of the design space referred to as the *targeted design space*. This reduces customization costs by allowing a simpler configuration interface. Second, we use a *declarative* programming language to facilitate open-ended specification of behavior without burdening the user with operational details.

We do not intend to provide direct evidence for our claims in their most general sense. Instead, we describe a concrete illustrative example, the JQuery tool, for one particular class of IDE extensions, code browsers. Applying these ideas to other types of IDE extensions is interesting for future research but is outside the scope of this paper.

2 The Targeted Design Space

In this section we examine the browsers provided by Eclipse JDT (Java Development Tools) environment, which is representative for the state of the art. This analysis serves two purposes. First, it serves as a basis to establish explicit bounds on the targeted design space. Second, it provides a concrete example of the two extreme customization-cost profiles.

Eclipse JDT offers multiple browsers to view and navigate source code. The core JDT browsers are shown in Figure 1. Each browser allows a developer to view and navigate their code in a different way: the *Package Explorer* (left) shows program structure in terms of modular containment relationships; the *Type Hierarchy* (middle) shows code structure in terms of inheritance relationships; and the *Call Hierarchy* (right) shows the structure of the static call graph.

Each of the browsers provides very similar functionality: a pane with a tree-viewer displaying icons and textual labels that represent program elements (packages, types etc.). The elements are organized hierarchically based on a particular relationship that exists between the elements. Double clicking an element reveals its source-code in a Java editor. Each view also has some buttons at the top, allowing some control over the contents and structure of the view. For example, the call hierarchy allows inverting the direction of the call edges; the package explorer allows hiding private and static elements; the type hierarchy allows showing/hiding an extra pane displaying the members of the selected type.

We see that Eclipse JDT browser configuration exhibits the two extreme customization-cost profiles: GUI buttons provide cheap but limited control over a browser's behavior but customization beyond this point is costly: in the best case the browser's source code can be used to develop a new plugin.

We end this section by establishing explicit bounds on JQuery's targeted design space. We decided to focus only on the core functionality observed in the various JDT browsers: a single browser pane containing a tree widget, but no support for additional control buttons or information panes. We also limited ourselves to browsers for a single Java program. In particular, it was not our goal to support browsing across multiple versions (e.g. CVS) or browsing of non-Java artifacts (e.g XML configuration files, build scripts etc.). These limitations delineate the targeted design space.

As a general principle, a customizer should not need to specify things that do not vary within the targeted design space. This principle served as a design guideline to simplify the configuration interface.

3 JQuery from a User's Perspective

JQuery is Java browser implemented as an Eclipse plugin. Unlike the standard Eclipse browsers, JQuery is generic: it can be configured to render many different types of views. We will argue that JQuery's customization-cost profile fits somewhere in between the two extremes offered by modern IDEs. This means that JQuery offers a more cost-effective creation of open-ended browser variations than a plugin architecture. However, it also means that customizing JQuery requires more expertise than clicking GUI buttons. Users may be reluctant to learn the configuration interface. Therefore in this section we present an example illustrating how JQuery 3.1.5 can already be used "out of the box".

The example is a fictional scenario in which a developer is exploring the JHotDraw [1] code base. JHotDraw is an application that lets users draw and manipulate a variety of figures such as rectangles, circles, etc. The developer wants to find out how figures are implemented and to find an example of a class that manipulates figures. Figure 2 shows a screenshot of JQuery at the end of her exploration. We explain the exploration step by step.

The starting point for exploration is typically a general purpose browser, such as a package explorer or a type hierarchy. To this end, JQuery provides a menu to select one of several "TopLevel" browsers. In this example the developer chooses to start with a JQuery view similar to an Eclipse package explorer. She then navigates down into the `org.jhotdraw.figures` package and discovers a number of classes that seem to correspond to different types of figures. Assuming there is a common base type for figures she decides to examine the supertypes of

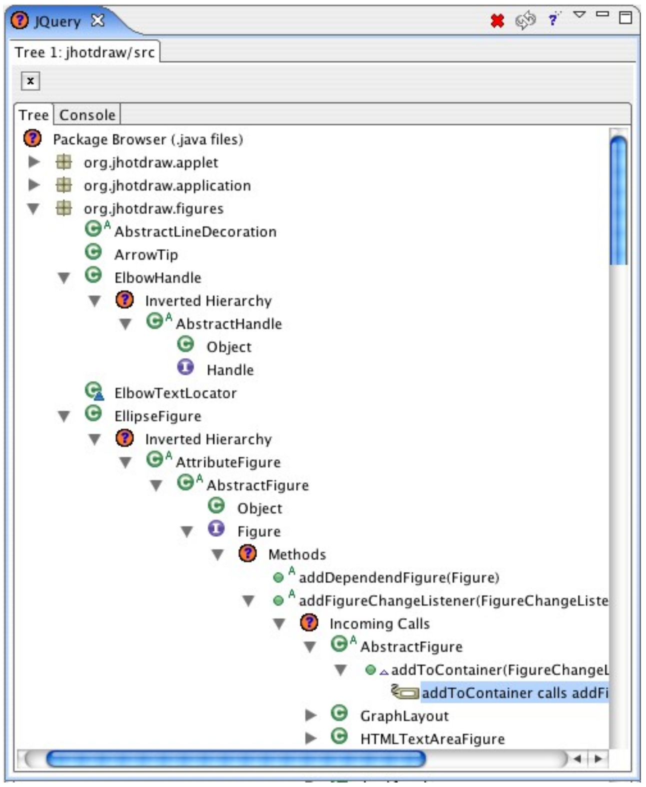


Fig. 2. Exploring Figures Implementation in JHotDraw.

`ElbowHandle`. In JQuery, the browser's view can be extended to reveal relationships not yet shown. Right-clicking on a node brings up a contextual menu of node-specific queries. The developer right-clicks on `ElbowHandle` and selects "Inheritance >> Inverted Hierarchy", which reveals the supertypes of `ElbowHandle` in an up-side-down inheritance hierarchy. Double clicking on a node brings the corresponding source code into view in the editor pane. Our developer inspects the source code of the supertypes of `ElbowHandle` and concludes that they are not what she is looking for. She then retraces her steps and tries browsing the supertypes of `EllipseFigure` instead. Among these she finds an interface called `Figure` which is what she was looking for. She decides to investigate what operations can be performed on Figures, expanding the view with the list of methods it defines. Finally, to find examples of classes that use `Figure`s she decides to find all the places in the code that make calls to `addFigureChangeListener()`. This concludes the example.

This example adopted from our previous paper [11] shows how an exploration task may involve following several different types of relationships back and forth. It illustrates that JQuery supports this kind of "mixed-relationship browsing" by allowing the insertion of sub-browsers at any node in the tree. Our previous paper focused on the merits of this particular GUI design. The current paper focuses on the merits of its generic implementation and declarative configuration interface. We believe that it is one of the merits of the generic implementation to have enabled the conception of the “mixed-relation browser” GUI. Indeed, the predecessor of JQuery, QJBrowser [13], was also generic but it did not support mixed-relationship browsing. QJBrowser’s generic browser model made adding that ability a straightforward next step.

4 Levels of Configurability

Although JQuery rests on top of a general purpose logic programming language called TyRuBa [8], it is not required for a JQuery user to be an expert TyRuBa programmer. It was illustrated in Section 3 how using JQuery “out of the box” requires no knowledge of logic programming. Furthermore, JQuery’s configuration interface can be divided into two levels which we will refer to as the *basic* and the *advanced* configuration level. We draw the line between the levels based on how a user interacts with the configuration mechanism. A basic user only uses the JQuery GUI. An advanced user also edits configuration files in a separate text editor. Since configuration files are TyRuBa include files, advanced user’s need to be familiar with programming in TyRuBa. Basic users are only exposed to TyRuBa via a dialog box in which they can edit a query. It suffices they have a basic understanding of TyRuBa expression syntax, but they do not need to know about inference rules or writing logic programs.

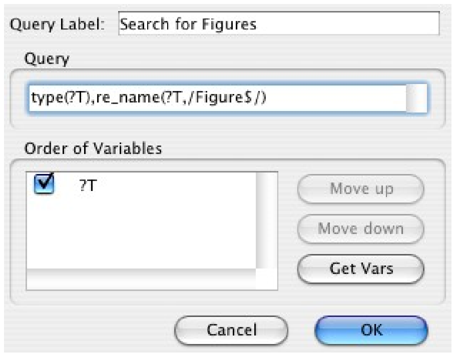
In the next two sections we will discuss both configuration levels in more detail. Each section begins with an explanation of the configuration interface and concludes with an argument placing its customization-cost profile at a different point between the two extremes.

5 Basic Configuration: “Instant” Browser Definitions

5.1 The Basic Configuration Interface

The key insight that sparked the JQuery design is that a code browser can be thought of as not much more than a tree-viewer widget for displaying and navigating query results. This is the main principle underlying JQuery browser definitions, consisting of two parts. The first part is a logic query executed over the *source model*, a database containing facts about the browsed program’s structure. The purpose of the query is to select the elements to be displayed in the browser. We will refer to it as the *selection criterion*. The second part is an ordered list of (a subset of) the variables bound by the query. Its purpose is to define how to organize the query results into a tree. We will refer to it as the *organization criterion*.

A basic user can access and change the definition of any browser or sub-browser by double-clicking on its root node. Alternatively they can create a new


Query Label: Search for Figures

Query

```
type(?T),re_name(?T,/Figure$/)
```

Order of Variables

[✓] ?T

Fig. 3. Dialog box for editing browser definitions.

browser by selecting "New TopLevel Query" from the menu. In each case they are presented with a dialog box like the one shown in Figure 3.

We now look at some concrete examples that serve two purposes. First, they help clarify the meaning of selection and organization criteria. Second, they illustrate we can a create a broad variety of useful browsers.

The first example is shown in Figure 3. Its selection criterion finds all types (classes or interfaces) in the program whose name ends with "Figure". This is a useful query, exploiting a naming convention in JHotDraw to find classes that implement the `Figure` interface.

The query language used to express the selection criterion is the declarative logic programming language TyRuBa [8]. TyRuBa expression syntax is similar to Prolog's. One notable difference is that identifiers for denoting variables must start with a "`?`" character. This is convenient because it allows us to use names of Java methods, classes and variables as constants even when they start with a capital.

JQuery defines a number of TyRuBa predicates that provide access to the source model. In this example two such predicates are used: the `type` predicate is used to find all types declared in the program; the `re_name` predicate is used to restrict to types whose name matches the regular expression `/Figure$/`. These predicates are implemented by JQuery either by storing facts into the TyRuBa database or by means of logic inference rules. From a user's perspective this distinction is irrelevant. All that is required is that users are familiar with the semantics of the available predicates. The list of source model predicates is fairly extensive. A representative selection is shown in Table 1. For a more complete list we refer to the JQuery documentation [14]. These predicates provide access to a wealth of information about Java program structure: declaration context, inheritance hierarchy, location and targets of method calls, field accesses, where objects are created, location of compilation errors, method signatures, JavaDoc tags, etc.

Predicate	Description
package(?P)	?P is a package.
type(?T)	?T is a type defined in the program.
interface(?T)	?T is an interface defined in the program.
method(?M)	?M is a method defined in the program.
field(?F)	?F is a field defined in the program.
method(?T,?M)	?M is a method defined in type ?T.
returns(?M,?T)	Method ?M has return type ?T.
name(?E,?n)	Element (=package, type, method, field) ?E has name ?n.
re_name(?E,?regexp)	Element ?E has a name that matches ?regexp.
subtype(?Sub,?Sup)	?Sub is a direct supertype of ?Sup.
subtype+(?Sub,?Sup)	Transitive closure of subtype.
child(?E1,?E2)	?E2's declaration is directly nested inside ?E1.
reads(?reader,?field,?loc)	?field is read from ?reader at source location ?loc.

Table 1. Selected predicates from JQuery's source model.

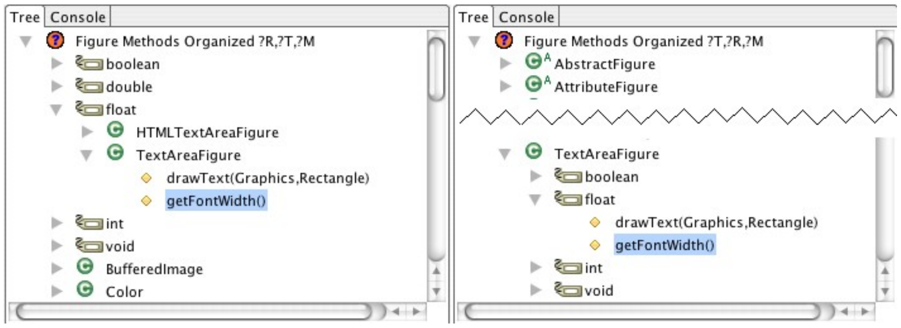


Fig. 4. Example: two different ways of organizing methods

The previous example's selection criterion only binds a single variable, making it ill-suited to illustrate the organization criterion. So let's extend the selection criterion as follows:

```
type(?T),re_name(?T,/Figure$/),method(?T,?M),returns(?M,?R)
```

This query additionally selects the methods (for each selected type) and the methods' respective return types. The results of this query can be organized in several ways. For example we can organize them primarily based on what type they are declared in by specifying a selection criterion `?T, ?R, ?M`. Alternatively we can organize them primarily based on return type by specifying `?R, ?T, ?M`. The resulting browsers are shown side by side in figure 4. To show the correspondence between both browsers, we have selected the same method (`getFontWidth()` from class `TextAreaFigure` returning a `float`) in each.

For our next example, we note that not all JHotDraw's figure classes follow the naming convention our first example relies on. This fact can be verified by formulating a selection criterion that finds violations of the naming convention:

```
    name(?IFigure,Figure), subtype+(?IFigure,?Figure),
    NOT( re_name(?Figure,/Figure$/) )
```

The resulting JQuery browser, displaying the values of all violating `?Figure`s, could be very useful to refactor the offending classes and make them respect the naming convention. Note that it would be very difficult to accomplish this task with the standard Eclipse browsers and searching tools: even though Eclipse has a fairly extensive search facility, it is insufficient to formulate a precise enough query.

As a last example, assume that the developer who wishes to fix her code to adhere to the naming convention only has ownership of specific packages in the code base. The browser above is not as useful because it does not help her distinguish violating classes under her control from other classes. There are several ways in which she could get around this. If she knows how to characterize the packages under her control, for example based on a naming convention, then she could refine the query to remove classes not under her control. Alternatively, she could decide to organize the offending classes based on their package by changing the browser definition as follows:

```
  selection:    name(?IFigure,Figure), subtype+(?IFigure,?Figure),
                NOT( re_name(?Figure,/Figure$/) ),
                package(?Figure,?P)
  organization: ?P, ?Figure
```

The resulting browser lets the developer quickly see all packages that contain offending classes and navigate to those packages under her control.

These are only a few examples. Many other useful browser's can be defined by selecting and organizing Java code elements based on different combinations of properties and relationships such as their names, inheritance relations, field accesses, declaration context, method call targets, object creation, method signatures, JavaDoc tags, etc. The possibilities are endless.

We conclude this section by noting that sub-browser definitions, for which we gave no explicit example, work in exactly the same way except that their selection criterion may use a special `?this` variable to refer to the element at which the sub-browser is rooted.

5.2 The TyRuBa Language

We now discuss some specifics of the TyRuBa language. TyRuBa has a static type-and-mode system that was heavily inspired by Mercury's [10]. The TyRuBa inference engine follows a tabled evaluation [2] strategy. Both these features help making the query language more declarative than more conventional (i.e. Prolog-like) logic programming languages. The details of the TyRuBa language, its type and mode system and its tabled execution are beyond the scope of this paper. We refer the interested reader to [8] for more details. We will assume that the typical PADL reader has at least a passing familiarity with these more advanced logic programming language features and limit our discussion to how they may affect JQuery users and their required level of expertise.

Evidently, the type-and-mode system adds considerable complexity for users who need to understand it and provide additional type and mode declarations when defining predicates. However, these complications mostly affect advanced-level users, since basic users are not expected to declare or implement predicates. Most important for the basic user is how these features affect the formulation of logic expressions. The type and mode system actually helps rather than complicates this. First, the type-and-mode checker rejects some ill-formed queries with an error message. In a more Prolog-like language such queries might execute without returning any results, producing an empty browser. For example the following query which has its arguments `?loc` and `?f` accidentally switched will be rejected by the type checker:

```
field(?f),name(?f,"foobar"),reads(?m,?loc,?f)
```

Another advantage is that the mode system absolves the user from worrying about subexpression execution order. For example in the following query subexpressions can be executed in either order and this will yield the same result (the Figure interface in the JHotDraw code base):

```
type(?Figure), name(?Figure,Figure)
```

However, it is preferable to execute the `name` subexpression first because that will quickly retrieve all objects with name `Figure` (a small number) and then iteratively single out the ones that represent types. The other execution order will retrieve all types (a large number) and then iteratively test whether their name equals `Figure`. The TyRuBa mode system uses some simple heuristics based on the number of bound parameters and a predicate's declared modes to make an educated guess about the best execution order.

The second example is similar but more complex and illustrates that some execution orders are expressly prohibited by the mode system.

```
re_name(?drawMethod,/^draw/), interface(?IFigure),
name(?IFigure,Figure), subtype+(?IFigure,?Figure),
method(?Figure,?drawMethod)
```

In this query, the execution of the `re_name` subexpression must be postponed at least until the `?drawMethod` variable is bound to an actual value. The TyRuBa mode system will pick an ordering that satisfies this constraint, as well as the preference to execute the `name` subexpression first.

Our experience suggests it is more intuitive for users unfamiliar with a logic programming language to write conjuncts in any order and think of them as semantically equivalent, than to consider the operational semantics of different execution orders.

5.3 Configuration-cost Profile

We end this section with an analysis of the configuration-cost profile for the basic-level user. Recall that the customization-cost profile of a mechanism is a characterization of its cost versus flexibility trade-off. We therefore perform an analysis in terms of a) limitations of the mechanism b) cost of the mechanism. We will argue that both in terms of cost and limitations the basic-level configuration interface fits somewhere in the middle of the spectrum.

We can divide the user's cost into two kinds of effort: effort to learn the query language and effort to actually formulate a query. Both of these costs are clearly higher than respective costs to click on GUI buttons which requires little effort to learn or to use.

Both of these costs are at the same time considerably lower than similar costs associated with developing a plugin in Java. Specifically, learning to use a plugin architecture is very costly because the APIs and XML configuration files associated with a plugin architecture have a complexity that is orders of magnitude higher than that of the JQuery query language. As a point of comparison, we determined the minimal subset of Eclipse 3.1 public APIs that is required for the compilation of JQuery. This subset declares 241 public types and 2837 public methods. In contrast, the JQuery query language defines only 13 types and 53 predicates over those types. We believe the difference outweighs the advantage that plugin developers may gain from being able to program in the familiar Java language. A similar order-of-magnitude difference is apparent in the effort of formulating a query versus implementing a plugin. A typical query is a handful of lines of declarative code; the implementation of a plugin ranges in the thousands of lines. For example the implementation of JQuery itself consists of 11019 commented lines of Java code, not including the implementation of the query engine.

Of course, the reduction in complexity implies a loss of flexibility. The basic-level configuration interface does not provide full control over all aspects of a browser's behavior, or not even over some configurable aspects of the tool that are only accessible at the advanced level.

Some limitations that apply specifically to basic-level users are as follows. First, edits to a browser definition through the GUI only affect the current instance of the browser but not new instances created later. Second, the structure of JQuery's menus can not be changed by basic users because this requires editing the configuration files. Third, basic users cannot define browsers with recursive structure such as for example a type-hierarchy or call-hierarchy browser.

There are also limitations that are a result of the limits of the source model. Basically, if information about the program structure is not present in the source-model or derivable from it, than no amount of creativity can produce that information. These limitations affect basic and advanced users alike. A plugin implementor on the other hand has direct access to extensive APIs and if that is not sufficient they have the option of implementing their own program analyzer.

In conclusion, the above analysis puts JQuery's basic-level configuration cost profile clearly in the middle between GUI controls and plugins.

6 Advanced Configuration

In this section we discuss the level of configurability available to advanced users willing to learn the TyRuBa programming language and the structure of JQuery's configuration files. Compared to basic users advanced users gain additional abilities:

1. to effect permanent changes to an existing browser's definition.
2. to define new browsers and add them permanently to JQuery's menu hierarchy.
3. to define recursive browsers.
4. to extend the query language with new predicates.

6.1 The Advanced Configuration Interface

The principle behind the advanced configuration interface is that whenever JQuery needs to make a decision about what to display or do, and this is supposed to be configurable, JQuery launches a query, asking the TyRuBa engine what it should do. Thus, the configuration interface takes the form of a set of predicates that are declared by JQuery but implemented in configuration files. The configuration files are TyRuBa include files that get loaded dynamically by JQuery. These files provide logic facts and rules that determine a significant portion of JQuery's functionality. To save space we limit ourselves to discussing two illustrative examples.

The first example is the definition of the "Inverted Hierarchy" sub-browser shown in Figure 2 and its corresponding menu item in JQuery's GUI. The relevant predicate in the configuration interface is declared as follows:

```
// -------------------------------------------------------------------
// A menuItem is defined follows:
//
// menuItem(?this, label, queryStr, [varName0, varsName1, ...])
//         :- applicabilityExp.
// -------------------------------------------------------------------

menuItem :: Object, [String], String, [String]
MODES
   (BOUND,FREE,FREE,FREE) IS NONDET
END
```

We stress that the *declaration* of this predicate is *not* in the configuration files, but rather is provided by JQuery as part of its definition of the configuration interface. The role of the declaration is to establish a contract between the JQuery GUI which calls it, and the configuration files that provide the implementation. The "Inverted Hierarchy" menu item is defined by the following configuration rule:

```
menuItem(?this, ["Inheritance", "Inverted Hierarchy"],
         "inv_hierarchy(?this,?IH)", ["?IH"])
:- Type(?this).
```

When a user clicks on an element in the GUI, JQuery calls the `menuItem` predicate, binding the `?this` parameter to the element. The second parameter will be bound by the rule to a list of strings representing a path in the menu/sub-menu hierarchy. In this case it indicates the creation of a menu item "Inverted Hierarchy" in the "Inheritance" menu. This rule's condition appropriately restricts its applicability to elements of type `Type`. The second and third parameters, also bound by the rule, correspond to the selection and organization criterion for the sub-browser created by invoking this menu. Their meaning is as described in Section 5. However, for simplicity, we neglected to mention that when variables are bound to list values, they are "unfolded" to construct consecutive levels of the tree. This mechanism enables the definition of recursive browsers. In this example, the `inv_hierarchy` auxiliary predicate, recursively constructs paths in the inverted-hierarchy tree:

```
inv_hierarchy :: Type,[Type]
MODES   (B,F) IS NONDET    END

inv_hierarchy(?T, []) :- NOT(subtype(?,?T)).
inv_hierarchy(?Sub,[?Super|?R])
   :- subtype(?Super,?Sub), inv_hierarchy(?Super,?R).
```

Note: the `?` variable in TyRuBa is similar to Prolog's `_` variable.

As a second example, we show how this mechanism makes it possible to dynamically construct menu structures dependent on properties of the element clicked on:

```
menuItem(?this, ["Members","Methods...", ?name],
         {child(??this,??M), method(??M),name(??M,?name)}, ["?M"], )
:- Type(?this), method(?this, ?M), name(?M, ?name).
```

The text within {} is a string template where variables inside the braces are substituted by their values. Variable substitution can be prevented by escaping them with an extra ?). Interesting in this example is how the variable `?name` bound by the rule's condition is used to construct the menu label as well as the selection criterion.

6.2 Customization-cost Profile

It should be clear by now that the advanced configuration level represents yet another point in the configuration-cost profile spectrum which is situated somewhere in between that of the basic-level and the plugin architecture.

In comparison to the basic-level, this mechanism offers strictly more flexibility than the basic level and at a strictly higher cost.

In comparison to plugins and in terms of flexibility, in spite offering a lot of flexibility to specify new behavior in logic rules and queries, JQuery is bound by the limitations of the targeted design space outlined in Section 2. Other limitations such as those caused by what is (not) reified by the source model also still apply. In terms of cost, an advanced user must be fluent in the TyRuBa programming language. This requires considerable effort in learning a non familiar and unconventional programming language. Assuming that the language barrier can be overcome, the customization cost is an order of magnitude below that of implementing a plugin. As a point of comparison, the most complex browser definition in JQuery's configuration files defines the "Method Hierarchy" sub-browser showing how a given method is overridden repeatedly in different classes in its class hierarchy. It defines two auxiliary predicates (to construct a hierarchy of alternating methods and classes in two steps) and consists of a total of 24 lines of TyRuBa code. As another point of comparison, the two default configuration files "`topQuery.rub`" and "`menu.rub`" which define a total of 15 TopLevel browsers and 54 sub-browsers are 73 and 352 lines of TyRuBa code respectively (this includes blank lines and comments). Even when taken together there is still an order of magnitude difference from the typically thousands of lines of Java code required to implement a browser plugin.

7 Related Work

JQuery derives much of its flexibility and functionality from the expressive power of the underlying query engine. The idea of using structural queries — in a logic language or another sufficiently powerful query language — as a basis for constructing software development tools is not new. Some examples of other systems based on structural source code querying are SOUL [17], ASTLog [7], GraphLog [6], Coven [4] and Stellation [5]. SOUL is a logic query language integrated with the Smalltalk development environment. ASTLog is a logic query language for querying C++ abstract syntax trees. GraphLog is a logic based graphical query language in which both queries and query results are represented as Graphs. Coven and Stellation are software configuration management tools, equipped with an SQL-like query language for the retrieval of software units. In all these tools, software queries can be used by developers in the process of exploring code. However, they do not support the use of the query language in that same way that JQuery does as a means to configure browsers, sub-browsers and the menu hierarchy.

There are numerous tools (e.g. Rigi [12], SHriMP [16], Ciao [3], SVT [9] and GraphLog [6]) that provide different ways to visualize the structure of a software system. JQuery is related to these tools in that a code browser is one kind of visualization. SVT [9] is most closely related. It is a configurable software visualization framework that relies on Prolog as a configuration language. However, JQuery's differs in that its configuration language is more declarative and that its targeted design space is (deliberately) more limited. Consequently

JQuery strikes a completely different balance between cost and flexibility of its configuration interface.

8 Discussion and Future Work

One area for future research is extending and broadening the targeted design space: to other types of IDE extensions and to be less Java specific.

A second area for future research is the generation of efficient implementations from declarative browser specifications. In this paper, we have talked about configuration cost entirely in terms of user effort. We have not considered runtime and memory efficiency of the browsers. Although efficiency was never the main concern for our work, it has become clear that it affects the practical usability of a tool like JQuery. Indeed, such concerns have been an ongoing issue and the TyRuBa query engine has had considerable work put into it in order to meet the demands of producing browsers from realistic code bases in an interactive environment. JQuery produced browsers, in its current implementation, cannot compete with hand-crafted plugins. However, we believe that the declarative nature of the specification should provide ample opportunities for automatic optimizations and compilation. This is a very interesting area for future research that could tap into a wealth of knowledge from databases and logic programming languages.

9 Conclusion

We discussed how modern IDEs offer two levels of configurability that have cost-profiles at the extreme ends of a spectrum. One mechanism is GUI-based and is very easy and cheap to use but offers limited flexibility. Another is very hard and expensive to use but allows complete control by linking in executable imperative-style code.

We argued that the space in the middle between those two extremes is also interesting and can be accessed by designing a configuration interface targeted for a particular domain on top of a declarative programming language. We illustrated this approach by presenting the JQuery tool, a highly configurable Java code browsing tool that employs a declarative logic programming language at the core of its configuration mechanism. We argued that JQuery achieves a very open-ended configuration model that is still orders of magnitude cheaper in terms of user effort than a typical plugin architecture.

Acknowledgments

This work was supported in part by IBM Canada, NSERC and the University of British Columbia. We thank the anonymous PADL reviewers for their constructive criticism. We thank Doug Janzen and Rajeswari Rajagopalan for their many contributions without which this paper would not have been possible.

References

1. JHotDraw. http://www.jhotdraw.org/, 2002.
2. Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.
3. Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proc. Int. Conf. Software Maintenance, ICSM*, pages 66–75. IEEE Computer Society, 1995.
4. Mark C. Chu-Carroll and Sara Sprenkle. Coven: brewing better collaboration through software configuration management. In *Proceedings of the eighth international symposium on Foundations of software engineering for twenty-first century applications*, pages 88–97. ACM, 2000.
5. Mark C. Chu-Carroll, James Wright, and David Shield. Aspect-oriented programming: Supporting aggregation in fine grained software configuration management. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 99–108. ACM, November 2002.
6. Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 138–156, New York, NY, USA, 1992. ACM Press.
7. R.F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.
8. Kris De Volder. Tyruba website. http://tyruba.sourceforge.net.
9. Calum A. McK. Grant. *Software Visualization In Prolog.* PhD thesis, Queens College, Cambridge, December 1999.
10. F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The mercury language reference manual, 1996.
11. Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM Press.
12. H. Muller, K. Wong, and S. Tilley. Understanding software systems using reverse engineering technology. In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994.
13. Rajeswari Rajagopalan. Qjbrowser: A query-based approach to explore crosscutting concerns. Master's thesis, University of British Columbia, 2002.
14. Jim Riecken and Kris De Volder. Jquery website. http://jquery.cs.ubc.ca.
15. Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proc. of International Conference on Software Engineering*, 2002.
16. M.-A. D. Storey, C. Best, and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proc. of International Workshop on Program Comprehension (IWPC '2001)*, 2001.
17. Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceeding of TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.