SHELL

- an application based on an XML enhanced source infrastructure

Title:
SHELL - an application based on an XML enhanced source infrastructure

Project period:
CIS4/KDE4,
February 1st, 2006 - August 16th, 2006

Project Group:
d635a

Students:
Wensheng Li and Kenneth Vittrup

Supervisor:
Kurt Nørmark

Statistics:
Copies: 4
Report pages: 53
Appendix pages: 33
Total pages: 86

Synopsis

SHELL is an experimental attempt to combine an XML based representation of Design Patterns with an XML based representation of source code. By merging these it is possible to derive structural knowledge from source code related to Design Patterns. Historically such structural knowledge must be explicitly provided by the programmer by providing details scattered throughout the source code.

It is finally concluded that until a better formalism for Design Patterns are discovered Design Patterns does not facilitate checking. However other forms of structural knowledge could be included as a substitute for Design Pattern if properly formalised.

Colophon

The layout of this report was done using XSL-FO. It was constructed using Apache FOP (xmlgraphics.apache.org/fop) with XSLT processing being performed by Saxon-B (saxon.sourceforge.net). All graphics included is SVG: UML diagrams were constructed by Poseidon for UML (www.gentleware.com), graphs were constructed by aiSee (www.aisee.com), and the remaining diagrams were constructed using Microsoft Visio 2003 (office.microsoft.com/visio).

Contents

1 Introduction

According to the ARKI research group the “future modelling tools and design patterns will form a framework, which enables developers to efficiently reuse existing design knowledge and do intelligent and grounded design decisions” Furthermore the group describes: “Future tools will inter-operate with knowledge bases, where design patterns and other design information is stored”. *[12]*

Composing new software is a major task requiring several computer scientists with different specialities working together towards a common defined goal for the final software. Computer scientists all contribute to the development process by supplying their knowledge and experiences: analysts, architects, designers, the project manager, and testers.

A variety of tools is available depending on the perspective laid on the system of the individual computer scientist. The tools help the computer scientist in creating, visualising, and documenting knowledge that should increase the quality of the final software. However even the most sophisticated tools are not able to express all the fine grained details making up all (or even the majority) of the knowledge of the computer scientist.

The quality of such tools depends on the amount of information initially available to the tool, how new information is introduced, and the quality of such information.

This project is inspired by the idea of a central data repository being able to express extensive information about source code. In order to test the usefulness of this idea the quality of information stored within the repository must be defined. The documentation related to Design Patterns is normally not documented or sparsely documented at best and is therefore reasonable to use for the attempt. The Design Pattern knowledge is stored in a central data repository that contains extensive information on the individual Design Patterns.

In order to demonstrate the usefulness of the idea presented by the ARKI research group an experimental development workbench is suggested. The workbench is constructed with innate support for expressing information related to Design Patterns beyond what is normally included and documented in the source code. Using the suggested workbench the software developer is able to create queries on the structure of the source code using the information on Design Patterns, stored in the data repository, as a reference.

Source code is formless but was depicted as a circle or sphere during the early development of the workbench. As the source code within this sphere was structured using XML it would be possible to draw parallels to additional information also structured in XML format. The additional information could be put in context of the actual source code stored within the sphere, in order to divert new (previously undocumented) information. The suggested workbench is the result of experiments describing how a shell of additional information surrounding the sphere providing such capabilities (hence the name SHELL).

While the computer scientist is developing using the workbench, SHELL is able to perform queries in the background on the current structure of the source code and the knowledge stored on Design Patterns. In addition SHELL is able to perform queries on how the knowledge on Design Patterns is applied in the structure of the source code.

1.1 Design Patterns

The term *Design Patterns* was introduced to encapsulate proven solutions to common (re)occurring problems in software design. The catalogue consists of twenty-three patterns divided into three major sections. *Creational patterns* gather Design Patterns that focuses on abstraction during the instantiation process. *Structural patterns* are Design Patterns that are concerned with how classes and objects interact and are used as building-blocks to form new and better structured software. *Behavioural patterns* focus on algorithms and assignment of responsibilities between objects.*[5]*

In addition to making know-how reusable, the use of Design Patterns "improve the structure of software, speed up implementation, simplify maintenance, and help avoid architectural drift".*[6]*

The term *template* is used to refer to any of the original twenty-three Design Patterns. In the book the presentation of each Design Pattern is mainly focused on explaining the domain of the problems in order to present a solution that is useful for the entire domain. The presentation of the individual solutions is highly dependant on OMT (Object Modelling Technique) diagrams to explain the interaction between classes and methods *[18]*. In general the diagrams used to explain the solution are not illustrated with the intent of realising the software, but on explaining the idea on the solution to the human reader.*[5]*

In the following figure the two Design Pattern templates Observer and Proxy are shown.

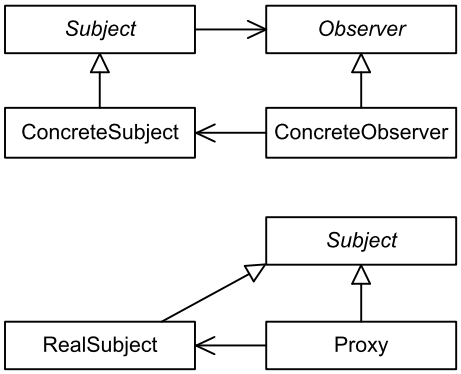


Figure 1. Observer Design Pattern (top) and Proxy Design Pattern (bottom).

A template can be *instantiated* by providing the proper combination of classes and methods (as defined in the template). This allows several instances of the same template to occur in a software solution. A benefit of this is that each class can be participating in several instances of Design Patterns, however in each instance of a Design Pattern a class cannot be participating more than once.

Whereas the origin of both templates and instances are conceptual, the implementation of a Design Pattern is one actual realisation (in a source program) of one instance of a Design Pattern.

The following figure illustrates how three instances of Design Patterns are introduced in an actual realisation. On the figure is illustrated one instance of the Proxy Design Pattern and two instances of the Observer Design Pattern. Each Design

Pattern instance is identified in a grey box. Separated by a colon, each box contains the name of a Design Pattern instance and identifies one participant class.*[7]*

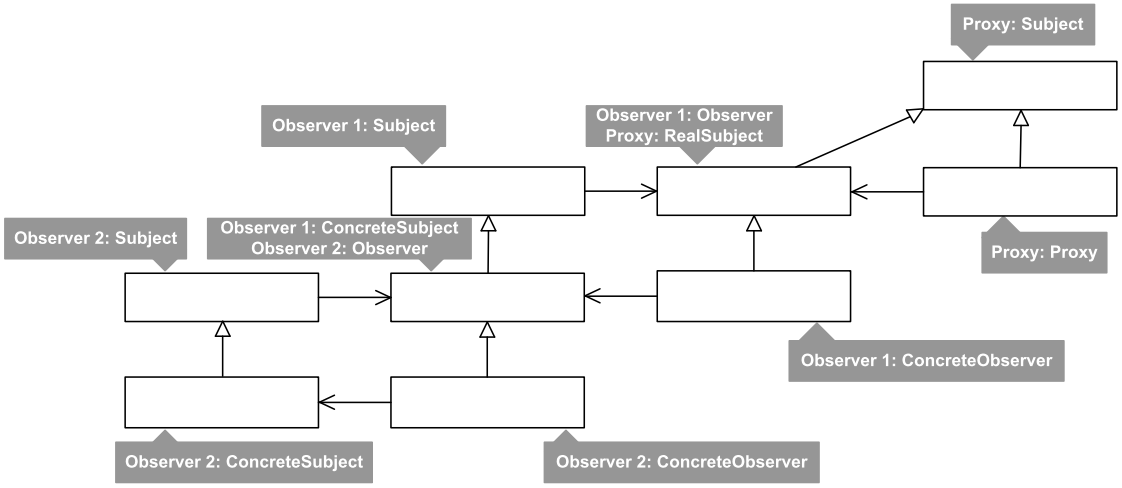


Figure 2. Diagram with two Observer Design Pattern instances and one Proxy Design Pattern instance.

In order to create automation for Design Patterns more information than what is presented in the original book on Design Patterns *[5]* is required. As an example of this neither of the Design Patterns described in the book contains information on cardinality. For some of the patterns this information will add valuable information, e.g. in any instance of the Observer Design Pattern there can be multiple participants of the class Observer.

Hence this project need to interpret the OMT diagrams presented in the book as UML. This will enrich the information contained within each individual pattern and is tailored towards automating the inclusion of Design Pattern information in source code.

Formalising a Design Pattern template is no trivial task due to an inherent focus on conveying information to humans. Some authors have tried to enrich the OMT notation used in the original book on design patterns while some have introduced a new language for expressing the coherence in Design Patterns. *[24] [26] [22] [25]*

In order to create a Design Pattern instance, the actual arguments for the template parameters need to be provided. In providing the actual arguments the fundamental constraints of the design pattern must be respected:

- A generalization between two classes in a Design Pattern template means that an inheritance or similar relationship must exist between the two corresponding classes in each Design Pattern instance. *[25]*
- An association between two classes in a Design Pattern template means that an association must also exist between the two corresponding classes in each Design Pattern instance *[25]*.

In 1997 the Object Modelling Group (OMG) developed the UML notation as a common architectural framework derived primarily from Booch OOD (Object-Oriented Design)*[16]*, Rumbaugh OMT *[18]*, and Jacobsen OOSE (Object-Oriented Software Engineering)*[17]*.

The OMG includes the *parameterized collaboration* in order to include information on Design Patterns in UML diagrams. In general a parameterized collaboration is a design construct that can be used repeatedly in different designs for expressing internal structures. However it should be noticed that Design Patterns contain more than structural information. The sample Design Pattern used to describe the parameterized collaboration is the Composite Design Pattern. The sample consists of three participants: Component, Leaf, and Composite. *[23]*

There are several limitations, that need to be considered when using the parameterized collaboration for describing Design Patterns. The following three extract from an article describe aspects of Design Patterns that can not normally be modelled using parameterized collaboration.

- “Design patterns having a variable number of participant classes (e.g. Visitor, Composite) cannot be precisely bound” *[22]*.
- “In the Visitor pattern, the number of visit methods defined by the visitor class must be equal to the number of concrete element classes.” *[22]*.
- “In design patterns an operation (or an attribute) is not necessarily a real operation. It defines a behaviour that must be accomplished by one or more actual operations.” *[22]*.

Several people have attempted to create better representation of source code, in order to include information on Design Patterns. Major policies for solving so are the visual extension and the language extension. The former normally uses UML

as a foundation and enhances the expressiveness with new formalism *[24] [25]*. The other approach is to create a new language, that contains the required information on each Design Pattern*[26]*. A similar approach is to create an XML prepresentation that contains the Design Pattern information named Design Pattern Markup Language (DPML) used for mining Design Patterns from source code, however not all Design Patterns can be expressed using this approach *[3]*.

In general the structural information contained within each of the Design Patterns determines if it is possible to formalise it. The typical Design Pattern used for evaluating the quality of expressing a pattern in a new approach is the Abstract Factory pattern. The reason that the Abstract Factory is useful for this is the cohesion between methods and classes: each participating class has a corresponding participating method.

1.2 Project defintion

SHELL is not designed to incorporate the common generic solution indicated by the original Design Patterns*[5]*. Instead the goal of SHELL is to provide a precise description of how the individual participants in each Design Pattern instances should participate depending on the used programming language*[22]*.

The underlying idea of SHELL is to store a formalised version of each of the Design Patterns in a central location. Following this the workbench is enhanced with a new underlying representation that better support searching and analysing the internal structures of the source code.

Theoretical this combination should makes it possible to derive information by recognising the predefined structures of Design Patterns without needing to parse the source code. Initially the focus is how it is possible to document the presence of Design Patterns in either source code or other documentation. The inclusion of Design Patterns in the Design Class Diagram is normally implicit given and sparsely documented, if documented at all. Next is the ability to check if a Design Pattern is correctly present in the source code. This would potentially allow searching for faulty or missing implementations of Design Patterns. Finally the workbench could be capable of automatically generating (part of) the Design Patterns.

The information stored in any source code is a reflection of the knowledge and work invested by the various programmers during the development process. The

aim of SHELL is to investigate if and how the OMT based models of Design Pattern models can be integrated into source code.

2 Internal structures of SHELL

The foundation of SHELL is the structured representation of XML. The following sections will describe how XML documents are useful for representing a variety of information and describe how XML documents can be modified using the operations described in section 2.3. Afterwards two possible XML representations relevant and useful for this project are explained: one representation for Design Patterns and one representation for Java source code.

Having described the XML foundation of SHELL, the exact problem addressed by SHELL is described in the final section of this chapter.

2.1 XML

XML is a W3C recommendation, which was created to structure, store and to send information. It was designed to separate the description of the content of data from the description of the layout of the data.

The content of an XML document consist of elements. Elements are enclosed in tags, signalling the start of an element <*element*> and end of the element <*/element*>. Between the starting and ending elements are the content of the element. The content of an element can be empty, containing text, or containing nested elements (denoted child-elements). The starting and ending elements of an element with no content can be substituted for <*element /*>.

Only the starting tag of an element can contain attributes; an attribute is made up of a unique name and a value. A controversial issue is if text should be put as content in elements, child-elements or attributes. In general using child-elements to contain text is considered better as this support the extensibility of XML. However it is often easier for the human to manually create elements where the majority of text is placed in attributes (less typing and more concise).

Each XML document can contain only one root element and the child-elements define the actual content of the document. The nested structure of XML documents allows parsing the document and creating a rooted tree graph. The nodes of the tree are the elements of the XML document, and there are no distinguishing between elements and attributes. Describing an XML document using a tree benefits as theories and terms from graph theory can be applied. In addition the tree provides better overview for visualising the structure of XML documents.

Visualisation of most XML documents is, in the remainder of this project, performed using directed trees, each with a single root element. These are included in the report in order to illustrate important concepts and specialities regarding XML documents. Attributes are only present in these diagrams when helping in understanding the diagram.

2.2 Representing contents using XML

XML can be used to structure and store various contents, some samples providing overview and showing the diversity of XML is presented next.

2.2.1 XHTML (Extensible HTML)

HTML (Hypertext Markup Language) is a W3C recommendation defining how a webpage for the Internet should be structured by using special predefined tags. It is an ancestor of XML defining both data and layout information as a single document. XHTML is a stricter and cleaner version of HTML defined in XML format.

2.2.2 XTM (XML Topic Map)

A topic map is a drawing showing relationships between topics. The drawing consists of topics (representing any concept from the problem domain), associations (representing relationship between topics), and occurrences (representing relationship between topics and sources of related information).

2.2.3 XSD (XML Schema)

XSD is a meta-language that is used to define the legal structures of elements an attributes in an XML document. XML documents that conform to the language described by the XSD are labelled “valid”; if it does not conform to the rules of the XSD it is labelled “invalid”.

2.2.4 SVG (Scalable Vector Graphics)

Vector graphics stored as XML is named Scalable Vector Graphics (SVG). An SVG file is a XML document using the SVG namespace. Each SVG file is made up of a single canvas. The height and width of the canvas can be specified as these are attributes of the canvas element. Nested within this canvas are the geometric figures that are to be displayed. It is interesting to note that such geometric elements can be nested within other geometric elements.

2.2.5 XMI (XML Metadata Interchange)

XMI is a standard of the Object Management Group (OMG) and is widely used for specifying UML diagrams in XML. The purpose of XMI is to allow interchange of UML diagrams between modelling tools.

2.2.6 DPML (Design Pattern Markup Language)

DPML was developed in order to encapsulate information on the original Design Patterns. It was developed in order to formalise Design Patterns for use in a data mining process. The intent was to mine applications of Design Patterns in source code *[3]*.

2.2.7 XDPML (XLink enhanced DPML)

XDPML was constructed by combining the DPML *[3]* with the XML linking technology named XLinks *[9]*. It was done by converting the standard internal references in XML documents (i.e. “id” and “idref” attributes) to XLinks. This resulted in an ability to include more information on the references between participating classes and methods in the individual Design Patterns. *[4]*

2.3 Operations on XML documents

By providing a style sheet it is possible to modify the information in any XML document (regardless of the actual stored information) according to the rules defined in the style sheet. The style sheet for XML is named XSL (Extensible Style Sheet Language) and provides a combination of the following different languages:

2.3.1 XSLT (XSL Transformations)

This, general purpose language for transforming XML documents, provides a processing application with the possibility of outputting a HTML, XML, or plain text document. Whereas XML is pure information, XSLT is the underlying logic providing essential information when transforming one XML document into another document.

An XSLT operation on an XML document is performed using an application capable of performing such an operation. Examples of such applications include xmllint (xmlsoft.org/xmllint.html), XMLStarlet (xmlstar.sourceforge.net), and XMLSpy (www.altova.com/xmlspy).

2.3.2 XPath

XPath defines the rules for navigating and traversing the elements and attributes of an XML document. The operation of traversing the tree is performed by XSLT (see above). The following gives the basic rules for understanding the XPath expressions in this report:

An XPath expression will attempt to match any elements or attributes in the XML document to the expression. The number of elements and attributes for an individual expression can be anything from no match to multiple matches (a collection). The intent of the following is not to explain every aspect and possibility with XPath, but to give a general introduction to the expressive power of XPath expressions. This is required for the reader to understand the XPath expressions used in the remainder of this report.

Navigating an XML document using XPath expressions are similar to navigating a unix/linux file system, where the character slash (“/”) is a special character used to differentiate between files and directories. In XPath the same character is used to separate between different levels of depth.

An absolute expression (any expression where the first character is a single “/”) defines the entire path from the root element to the specified element(s). A relative expression does not start with the root element of the file and defines the path onwards from the current element. Finally an expression where the first two characters are “//” will match anywhere in the file disregarding the nesting of the elements within the XML document.

Any (part of) an expression that starts by a “@” is an attribute. Any parts enclosed in square brackets (“[” and “]”) indicates a self explaining logical expression.

Some examples of XPath expressions are:

- b indicates the child element of the current element named “b”.
- /a/b indicates the element “b” that must be child of the root element “a” of the XML document.
- //a indicates the element “a” that could be present anywhere in the XML document.
- @a indicates the attribute “a” of the current element.

From this it should be clear that “/” and “@” are not valid characters for naming XML elements *[11]*.

2.3.3 XSL-FO (XSL Formatting Objects)

XSL-FO is a part of XML dedicated to creating visual output. Where most XML representations provide pure information and ignores possible output formats, XSL-FO relies heavily on XSLT and provides capability for outputting XML information on screen, paper and/or other media.

2.4 Design Patterns as XML documents

The following two sections describe two different ways of using XML to represent Design Patterns. Having presented the two possibilities a comparison will follow, indicating advantages and disadvantages of both.

2.4.1 Design Patterns as XMI documents

XMI specifies an open information model for exchanging information contained within UML, and both are specified by the OMG. OMG was founded in 1989 and "produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications" *[13]*. XMI integrates the three industry standards XML, UML, and MOF:

- XML is a W3C recommendation designed to separate the description of data from the description of how the data is presented. XML is extensible and has its own meta-language named XSD.
- UML is used to model applications on a conceptual level, among the information that can be modelled are "application structure, behaviour, and architecture" *[14]*.
- MOF is a "metadata management framework" that is used in all OMG standards *[13]*, in order to obtain homogeneous standards.

However XMI is not a standard for representing visual information and tools are constantly evolving to broaden the support for UML represented in XML.

XMI describes the information that make up the principal building blocks of UML. In particular interest for this project is the structural building blocks, as these have been extensively used to describe the intuition behind the use of individual design patterns.

In XMI central elements have a unique ID: classes, attributes, and operations. In addition to this a unique ID is present for all parameters and all primitive types used in the diagram.

Inheritance is central to all object-oriented thinking. In UML an inheritance is illustrated using an arrow. The arrow is directed towards the parent class from the child class. XMI expresses inheritance by creates a uniquely identified reference element containing references to the parent ID and to the child ID. This makes the linking bi-directional: the element that makes up the two classes both has a reference to an element specifying the relationship, while the relationship has references to both the parent and the child element.

XMI is strongly supported in the industry, a partial list of applications supporting XMI are: Poseidon for UML (www.gentleware.com), MagicDraw UML (www.magicdraw.com), Together (www.borland.com/us/products/together), and Umbrello UML Modeller (uml.sourceforge.net).

In summary XMI is an OMG standard for storing information on UML in XML format. By further using XSLT on the resulting XML file it is possible to extract central information from the XMI. In this project two different things are generated using XSLT. First it is possible to recreate the visual UML diagram, and store this in SVG format. Secondly (and more crucial for this project) is the ability to extract the information in a dedicated format, focussing on the "software design elements", "relationship among the elements", and the "behavioural description" *[6]*.

2.4.2 Design Pattern as Topic map

A topic map can represent information using topics (representing any concept, from people, countries, and organizations to software modules, individual files, and events), associations (which represent the relationships between them), and occurrences (which represent relationships between topics and information resources relevant to them). *[28]*

Topic map is originally used to structure documents in a certain problem domain. One of the common application examples of using topic map is to organize the documents in order to create a web site. In a typical web site, there are lots of several web pages belong to different titles or topics. The page and its sub pages in the same "topic" with similar content are modelled as a topic (the main page) and the occurrences of a topic (the sub pages).

Topic map has a standard XML-based interchange syntax called XML Topic Maps (XTM, Specification), as well as a de facto standard API called Common Topic Map Application Programming Interface (TMAPI), and query and schema languages are being developed within ISO. *[28]*

Notice that since XML topic map is the most widely used format of topic map, in this report the name topic map and the full name XML topic map are used interchangeably. The topic map that is used to organize XML document is called XML topic map. And it is in itself written in XML topic map.

In our project, Topic map is used to model the Design Pattern. Design pattern is considered as a domain knowledge, which requires to be organized in a very structural and standardized way.

2.4.2.1 Topic

First of all, topic is the heart of topic map, which is basically one of the things topic map is about. For example, when modelling the proxy Design Pattern, topics are, among others, "proxy", "subject", "real subject". Careful reader would observer that these are all participants in the proxy pattern. However, topics are more than just participants. A topic could also be operation in the participants, etc. In general, any entities with substantial meaning in the knowledge domain are topics. Therefore, a topic could be classes, interface, methods, and attribute.

Furthermore, in topic map, a topic could be typed in topic map indicating that a topic belongs to a certain type. For example, the "proxy", "subject" topics are both topic of "participant" type because they both represent participants the proxy pattern. "Request" topic is of type "operation". Interestingly, the type itself is topic. By recursion, the topic map is essentially extensible.

2.4.2.2 Association

Association models the relationship between topics. The relation can be binary relation between two topics or ternary relation among three topics. Conceptually, it could multiple relations among many topics. For example, the association for "Extend" is a binary association while the "methodCall" association is a ternary association.

In an association, each topic involved is said to play a "role" in the association. For example, the "subject" topic plays the role as "parent" while the "real subject" topic plays the role as "child" in the "extend" association.

Following is an example of association:

```
<association>
  <instanceOf>
    <topicRef xlink:href="#Extend"/>
  </instanceOf>
  <member>
    <roleSpec>
      <topicRef xlink:href="#Parent"/>
    </roleSpec>
    <topicRef xlink:href="#Subject"/>
  </member>
  <member>
    <roleSpec>
      <topicRef xlink:href="#Child"/>
    </roleSpec>
    <topicRef xlink:href="#RealSubject"/>
  </member>
</association>
```

Notice that the value in the "xlink: href" attribute in the element "topicRef" defining the type is itself another topic defined elsewhere in the same document. For example, "Extend", "Parent", "Child" are all topics themselves.

2.4.2.3 Occurrence

Occurrence in Topic map represents the information relevant to a specific topic. The occurrence is also typed giving the possibilities of assigning several different types of occurrences to a topic. The occurrence itself may be just an "inline" string representing what it is. For example, the "subject" topic has an occurrence of the type "modifier" which contains a string "abstract". On the other hand, the occurrence may also be an XLink, linking to another topic in the same documents or another part of external document.

```
<topic id="RealSubject">
  <baseName>
    <baseNameString<RealSubject>/baseNameString>
  </baseName>
  <occurrence>
    <instanceOf>
      <topicRef xlink:href="Modifier"/>
    </instanceOf>
    <resourceData<public>/resourceData>
```

```
    </occurrence>
    <occurrence>
      <instanceOf>
        <topicRef xlink:href="Method"/>
      </instanceOf>
      <resourceRef xlink:href="#Request"/>
    </occurrence>
  </topic>
```

2.4.3 Comparison

There are several advantages of using topic map to express design pattern.

One of the nice things about using topic map is the fact that there are lots of tools, libraries supporting topic map. Following are some of the examples:

- TM4J (www.tm4j.org) is an open source topic map engine project in Java. It consists of (a) TMNav - a Java/Swing desktop application for browsing topic maps. The eventual goal of this is to provide both a toolkit for the creation of topic map browsers and editors as well as reference implementations and (b) TM4Web - provides support implementations for integrating the TM4J Engine with commonly used web application frameworks such as Apache's Cocoon and Struts projects.
- The "Omnigator" is a free topic map browser that can display any topic map. Once the topic map is created, one can browse it from "Omnigator" immediately without any programming.
- TMQL (Topic Map Query Language) (www.isotopicmaps.org/tmql) is a standardization effort by ISO/IEC with the objective to create a standard for such a language (ISO 18048).

As long as the information about design pattern is there, SHELL can always do the checking by building XML parser which navigates through the pattern documents, no matter it is represented in topic map or XMI and read the JavaML files and do the comparison in order to do the verification.

One alternative is to build a parser, which does the checking work. SHELL is built with the XML DOM (Document Object Model) parser. XML DOM is a standard way of accessing and manipulating XML document. It basically represents the XML document as a tree structure and gives access to the structure through a set of objects.

Also, XPath and XQuery could be used as ways of obtaining relevant information quickly.

Luckily, with the design pattern information represented in topic map, SHELL will be able to query it using very concise Topic map query language statements instead of ad hoc and cumbersome XPath expression or XQuery expression.

For example, if SHELL are to check if the entire "RealSubject" participant instances classes actually implement its super classes "subject".

It would be tedious to use XPath expression every time such query is needed. Fortunately it is possible to make this query very easily by using the TMQL expression Exist topic NOT in (association.type="extend") WHERE topic.type=="real subject" topic.type=="subject")

This is just an example of some of the benefits SHELL gains by using Topic maps. There are lots of tools out there, which greatly relieve our burdens of parsing the XML.

In general, it is often hard to give overwhelming reason to explain why a technology is not as commonly used as anther. People with different background prefer different technologies. For example, people work in different problem domain has its own favourite programming language. The success of a technology largely depends on whether or not there are lots of ambitious people with great enthusiasm do their best to boost up the use of that technology by means of publishing paper, actively organizing and participating in communities and having conference.

Last but not least, XMI is not meant to be a complete solution to model the Design pattern problem but a way of visualizing the design information. In comparison, topic map is more than extensible and adaptable which is sufficient for capturing any the information in Design Pattern.

2.5 Source code as XML documents

Traditionally source code has been structured using American Standard Code for Information Interchange (ASCII) plain text format, however "This format does not reflect the underlying structure of the program" *[21]*. In order to actually incorporate further information into source code a stronger representation is required. When storing the source code of a program it can basically be done by storing a text representation or by storing some graph representation of the program.

Several people have worked on ways to use XML as an underlying structure for representation of source code. Most notably are CppML that is used for C++ source code representation *[20]* and JavaML is used for Java source code representation *[2] [1]*. As implied by the name, these are bound to the specific languages of C++ and Java. In order to avoid this binding to one language, work has been done in order to generalize this into OOML for generalizing object-oriented representations *[20]* and SrcML for a general-purpose representation for all source code *[19]*.

Historically the java source code representation is a stream of tokens in plain text format that are provided to a java compiler for further processing. JavaML is an example of using such an alternate representation of source code, based on XML *[2] [1]*. In order to create an XML based representation from a plain text representation the tokens of the later is encoded into the former, using a dedicated parser. For each java plain text file, the parser will provide a complementary XML based file. The advantage of this is the ability to utilise the innate structural information that is present in all XML files. An example of this use is the ability to use a DTD or XSD checking of the XML based code representation rather than parsing the plain text representation and creating a syntactical analysis.

The motivation for creating JavaML is that a new representation of Java source code could be useful as only sophisticated tools parse the source in order to obtain information on the underlying structure of the program. However, a multitude of tools do not parse the software, but relies solely on a syntactical analysis of the source code. With this in mind, JavaML was created in order to allow additional tools to operate on the source. The example given by Greg J. Badros on such a tool is "grep" (www.gnu.org/software/grep) but also using tools like "sed" (www.gnu.org/software/sed) and "awk" (www.gnu.org/software/awk) is possible.

JavaML makes it possible to create simple tools that do not need to parse and compile the Java source code in order to obtain information on the software. In many IDE's the parsing is an integrated part, performed simultaneously with the development. Modifications in JavaML can be performed by much simpler tools. The only restriction imposed on such tools is that JavaML must be able to validate the resulting XML document. Validation was originally done by DTD. A recent addition to JavaML is JavaML 2.0 where it is possible to replace the DTD checking with an XSD checking. *[1]*

The programming language Java allows the programmer to be very concise when writing source code. This has the advantage that the speed of development is

increased; however one of the main disadvantages is that some essential information is implicitly given, i.e. the programmer does not need to provide all these information.

With JavaML comes a JavaML compiler that analyses the AST (Abstract Syntax Tree) that is created by a compiler named Jikes (jikes.sourceforge.net).

The JavaML compiler does not only provide an XML structure of the java source code by marking up the original source code with XML tags. The internal logic in the program analyses much of the information, and converts much of the implicit information to explicit information in the XML output. Following is a partial list of such implicit information in the java source code, which is explicitly supplied in the XML encoded output:

- In Java a class that does not have an explicit stated parent, will inherit the class "Object". Depending on the programmer this information does not need to be present in plain text source, but will always be present in the corresponding XML source.
- In Java certain packages are included by default, e.g. "java.lang". The package of such default objects, methods, and primitive types does not need to be explicitly supplied in the plain text source representation. This is explicitly supplied in the XML source.
- Name binding with classes, variables, parameters, and methods are performed using the scoping convention of the java language, while the XML encoding assigns a global unique identifier to each, based on the name of the package, class, and any method and/or parameters.
- All references are provided an "idkind" attribute, with a value designating the nature of the referenced value.

The following is a sample plain text java source code.

```
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello World!");
  }
}
```

Next is the corresponding XML source code after encoding by the JavaML compiler. Note that the underlined words in the XML encoding are tokens from the original source code, that can be found in the original java source code above.

```
<java-source-program>
  <java-class-file>
    <class name="HelloWorld" id="LHelloWorld;" idkind="type">
      <modifiers>
        <modifier name="public"/>
      </modifiers>
      <superclass name="Object" idref="Ljava/lang/Object;" idkind="type"/>
      <method name="main" id="LHelloWorld;main([Ljava/lang/String;)V" idkind="method">
        <modifiers>
          <modifier name="public"/>
          <modifier name="static"/>
        </modifiers>
        <type name="void" primitive="true"/>
        <formal-arguments>
          <formal-argument name="args" id="LHelloWorld;arg9" idkind="formal">
            <type name="String" dimensions="1" idref="Ljava/lang/String;" idkind="primitive-type"/>
          </formal-argument>
        </formal-arguments>
        <block>
          <send message="println" idref="Ljava/io/PrintStream;println(Ljava/lang/String;)V"
idkind="method">
            <target>
              <field-ref name="System.out" idref="Ljava/lang/System;out" idkind="field"/>
            </target>
            <arguments>
              <literal-string value="Hello World!"/>
            </arguments>
          </send>
        </block>
      </method>
    </class>
  </java-class-file>
</java-source-program>
```

These two representations are not equivalent as converting the XML representation to the corresponding plain text representation yield a loss of such explicit information. Both representations have advantages and disadvantages as summarised in the following.

- For plain text source an advantage is that the programmer does not need to worry about certain details. This gives the programmer freedom and allows the programmer to be very concise when developing, hence making development faster. The main disadvantage is that mistakes and misunderstandings with the implicit given information will only be revealed during the compilation process, or a test phase following the main development of the software.
- For XML source the main advantage is that it is possible to extend the representation with explicit information. It is possible to hide this inclusion, as this will not affect the internal working of the program. The disadvantage

is the XML syntax is tedious, and nearly impossible, for programmers to both read and write.

The amount of explicit information present in the source code is important for SHELL, as this limits the amount of information from the Design Patterns that potentially could be included.

Extraction of information about the source code is done by analysing and traversing the structure of the XML tree using XPath expressions. By displaying the above XML source code as a tree it will next be illustrated how such expressions are constructed.

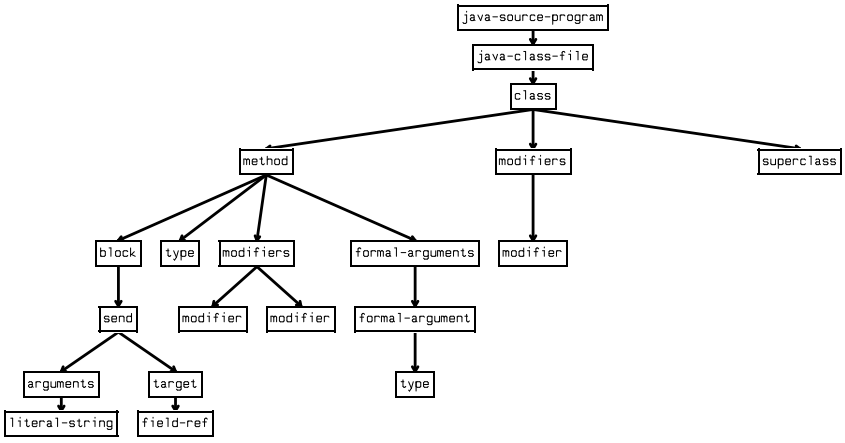


Figure 3. Tree-view of a JavaML file

The following gives some samples of the usefulness of using XPath expressions in each of the following: Classes, Inheritance, Defining methods, and Calling methods.

2.5.1 Classes

The name of the current class is can be matched using the XPath expression /java-source-program/java-class-file/class/@name. By exchanging @name for @id in the expression the unique identifier for the class is matched.

A collection of the modifiers applied to the current class can be matched by the XPath expression /java-source-program/java-class-file/class/modifiers/modifier. The individual modifiers can be accessed by an identifying condition in square

brackets following this XPath expression. E.g. by appending /[@name="public"] to this expression, the expression will match a value if the "public" modifier is present in the XML source code structure.

2.5.2 Inheritance

The element "superclass" contains a reference to the class that it inherits (its parent class). The XPath expression /java-source-program/java-class-file/class/superclass matches information on the parent. By appending /@name to this expression the name of the parent is matched, while appending /@idref matches a unique identifier for the parent class.

Note that classes in Java by default inherit the class "Object".

2.5.3 Defining methods

Each method of the class has a corresponding "method" element in the XML structure. The XPath expression /java-source-program/java-class-file/class/method matches a collection of the methods of the class. The individual modifiers can be matched by an identifying condition in square brackets following this XPath expression. Each method has a @name attribute that contains the name of the method and a @idref attribute that contains a unique identifier for the method.

By appending /modifiers/modifier to this expression a collection of the modifiers applied to the method is retrieved. The individual modifiers can be matched by an identifying condition in square brackets following this XPath expression.

By instead appending a /type to the above XPath expression the return type of the method is retrieved. The return type has a @name attribute that contains the name of the return type and possible a @idref attribute that contains a unique identifier for the return type.

By appending a /formal-arguments to the above XPath expression a collection of the arguments for the method is matched. The individual arguments can be obtained by further appending /formal-argument to this expression. Each argument has a @name attribute that contains the name of the argument and possible a @idref attribute that contains a unique identifier for the argument.

2.5.4 Calling methods

Calling a method on another class is always done by a block inside a method inside the current class. It has two parts: an argument and a target object.

The argument is a collection of the actual arguments of the method call.

The target is a unique identifier for the method being called.

2.5.5 Intuition

Assume an instance of the Proxy Design Pattern is present in a source code. The participant class "Subject" is named "Subject"; the participant class "Proxy" is named "Proxy", while the participant class "RealSubject" is named "RealSubject". The participating operation "Request" is named "Request".

This gives that the classes and methods in the following UML diagram must be part of the Design Class diagram:

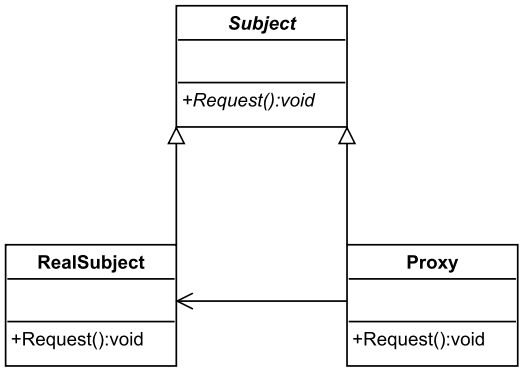


Figure 4. Proxy Design Pattern.

Note in the following that this is a strict interpretation of the original Proxy Design Pattern and does not reflect the fact that new language features since have been

introduced into the design and construction of modern languages (e.g. *C#*). Examples of such features also include interfaces and inheriting from non-abstract classes.

Having encoded the source to XML using the JavaML compiler, the following XPath expressions are used to locate implementation details on the Proxy Design Pattern:

The class "Subject" must be declared abstract, i.e. one of the modifiers applied to the class must contain the keyword "abstract", this is done using the expression /java-source-program/ java-class-file/class[@name="Subject"]/modifiers/modifier[@name="abstract"].

The classes "RealSubject" and "Proxy" must both inherit the class "Subject", this is done using the expressions /java-source-program/java-class-file/class[@name="RealSubject"]/superclass[@name="Subject"] and /java-source-program/java-class-file/class[@name="Proxy"]/superclass[@name="Subject"].

All three classes must contain an operation named "Request", this is done using the expressions/java-source-program/java-class-file/class[@name="Subject"]/method[@name="Request"], /java-source-program/java-class-file/class[@name="Proxy"]/method[@name="Request"], and /java-source-program/java-class-file/class[@name="RealSubject"]/method[@name="Request"].

The method "Request" in the class "Subject" must be declared abstract, using the expression /java-source-program/java-class-file/class[@name="Proxy]/method[@name="Request]/modifiers/modifier[@name="abstract"].

The method "Request" in the class "Proxy" must call the method "Request" in the class "RealSubject", this is done using the expressions /java-source-program/java-class-file/class[@name="Proxy]/method[@name="Request]/block/send/[@message="Request"] and /java-source-program/java-class-file/class[@name="Proxy]/method[@name="Request]/block/send/[@target="RealSubject"].

2.6 Problem statement

The central question addressed by this project is: *“When developing a software development workbench that facilitates checking for the presence of Design Pat-*

terns, which advantages (if any) can be obtained by using a richer structured XML representation for the underlying storing of both source code and Design Pattern knowledge?"

Some questions derived from this central question are listed below:

- *"Do the Design Pattern descriptions contain sufficient information to facilitate checking?"*
- *"Do the Design Pattern descriptions contain sufficient information to facilitate checking once these are stored as XTM?"*
- *"What information from the Design Patterns literature can (not) be expressed using XPath expressions on JavaML?"*

3 Design and realisation of SHELL

In the previous chapters the baseline was on elaborating the idea and possibilities that theoretically could be obtained if XML was introduced as the underlying structure of a development tool. In this chapter the focus will be on exploring these ideas and possibilities even further. This is done by gradually developing a workbench dedicated for such experimentation.

The original OMT diagrams are a universal toolkit dedicated to be used for any object-oriented programming language. As it is generic and language-independent it does not specify any details on the actual implementation details of Design Patterns in various programming languages. An OMT diagram can specify that one class should inherit another class, but not what it actually means to inherit another class using the current programming language. As examples of inheritance in different programming languages are Java that uses the keyword "extends" while C++ uses the notation "::"*[8] [27]*.

By using XPath expressions on JavaML files, it is possible to determine if the JavaML classes and methods conforms to the requirement set by each of the employed Design Pattern instances.

3.1 User scenarios (Use Cases)

In this section is described how the workbench be like and how can it be used from the end user" perspective. The people that will use the SHELL workbench can be roughly categorised into three: the programmers, the software designers, and the software testers (QA).

3.1.1 Programmer

The programmer is the person in the software development process who actually types the source code line by line and is obligated to correctly implement the software strictly according to the software system design specification. The software system design information is very often conveyed from the software designer (described next) to the programmer using UML diagrams.

The programmer has the most detailed knowledge of the implementation including the structure of the source code. The Design Patterns already have been instantiated whenever the implementation is finished. Afterwards, the programmer

is responsible for relating the source code and the system design by creating the mapping file.

3.1.2 Software designer

The software designer is responsible for the overall design of the software system, including creating and editing the UML diagrams. In order to create the appropriate design, there are two options. One option is to select an existing design pattern e.g. the 23 patterns from the GOF. And the other option is to create a customized design pattern to suit the needs.

In the first case, the software designer selects the most appropriate standard pattern from the pattern catalogue. Next the participating classes in the chosen Design Pattern will be presented.

In the later case the software designer is obligated to draw a Design Pattern diagram from scratch. After creating or editing the participating classes or methods in a Design Pattern, it is saved as a customised Design Pattern that would appear in the Design Pattern catalogue for later use by programmers and other software developers.

3.1.3 Software testers

When the programmer has written the source code and created the mapping, the software tester will verify if the code is a correct implementation of the design. The software tester has the obligation of reporting the verification results to the programmer in order to notify her errors, if any.

3.2 Tool-chain

The underlying structure of SHELL is a set of representations and possibilities of combining these using dedicated tools (most notably XSLT transformations). The following figure illustrates how these representations and tools are combined into

a complete tool-chain. The individual elements on the illustration will be presented in the following sections.

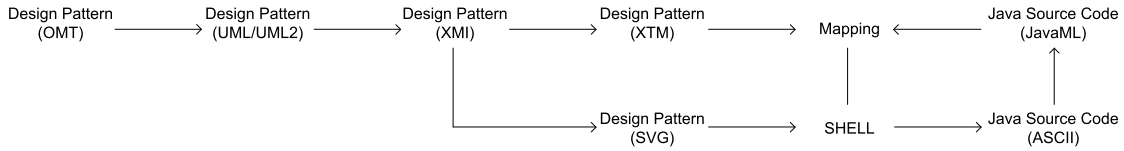


Figure 5. Tool-chain with representations and conversions.

3.2.1 UML

The Design Patterns were presented using OMT *[5]*. OMT was since adapted as a part of the UML and later as a part of UML2. What makes this interesting is that a subset of UML2 is the XMI specification *[15]*. Hence it is possible to create an XMI representation of each of the Design Patterns.

3.2.2 XMI

The generated XMI files are primarily a standardised way of describing the structural parts of Design Patterns using a Design Class diagram. In more general terms XMI is used for making a XML representation for each of the UML2 diagrams: class diagrams, interaction diagrams, state diagrams etc. *[14]*

3.2.3 SVG

By XSLT operations on the XMI file it is possible to derive the information needed to create an SVG representation of the UML information. Essentially this is a SVG version of the UML2 Design Class diagram *[14]*. In SHELL the operation of extracting SVG diagrams from XMI files is performed by the open source tool uml2svg (uml2svg.sourceforge.net). It is possible to include such SVG graphics in the workbench using Apache Batik, as stated on the homepage of the project: "Batik is a Java(tm) technology based toolkit for applications or applets that want to use images in the SVG format for various purposes, such as viewing, generation or manipulation." (xmlgraphics.apache.org/batik)

3.2.4 XTM

A problem with the chosen approach is that the XMI representation only includes the information that can be derived from the original OMT diagrams. Hence it contains no language-specific details that can be used in SHELL. In order to add this additional information the stronger representation of XTM is used by SHELL. By employing XTM to describe Design Patterns it is possible to combine both the universal information from the OMT diagrams and language-specific details, refer to section 2.3.2 for an explanation of how XPath expressions can be used for this purpose.

3.2.5 Java files to JavaML

A sub process invoked by SHELL is the JavaML compiler tool from JavaML, which converts each Java plain text files to a corresponding JavaML representation. The JavaML compiler relies on a Java compiler (i.e. jikes), and will return an error if the source code could not be compiled, i.e. a syntactical or semantically error was present in the plain text source code.

3.2.6 Mapping file

In order to make use of XTM and JavaML files a new XML document is created. This file contains a one-to-many mapping of the Design Pattern Templates (in XTM format) to the actual implementation of the Design Pattern Instances (in JavaML format). Beyond the root element the mapping XML file contains a child element for each Design Pattern instance present. The element "pattern" has an identifier for the XTM file containing the Design Pattern Template. Child elements of the "pattern" element are the classes (participants) and methods (participating methods) of the Design Pattern. Each of these contains two parts: one part for identifying the JavaML file and one part for identifying the corresponding participating class or participating method within the Design Pattern XTM file. Note that there are no restrictions on the number of "pattern" elements that can be included in the mapping file. This has the effect that any number of Design Pattern instances could be implemented in the actual source code (including zero). The number of participating classes and methods in the individual Design Pattern instance are restricted as described in the corresponding XTM file.

Mapping is a mechanism binding the Design Pattern to the Java class. It establishes the connection between participanting classes and methods and the actual Java implementation of classes/interfaces and methods. The mapping is a many-

to-many relationship. This indicates that a class can be participating in several Design Pattern instances.

3.3 System overview

Having introduced the underlying representation of the infrastructure the following presents functionalities contained in the workbench, ranging from verification to pattern catalogue.

3.3.1 Verification

In general SHELL is able to verify the presence/absence of classes in employed Design Pattern instances and whether the participating classes and methods are implemented in accordance with the Design Pattern template.

- The class modifier "abstract" are checked if this is also the case in the corresponding JavaML class.
- The attributes are checked if all the required attributes are present.
- Classes that inherit other classes are checked if this is also the case in JavaML.

In addition to these checks the presence/absence of methods in the Design Pattern and whether the methods are implemented in accordance with the Design Pattern template.

- A proper mapping occurs when a class and the containing method are both present in the JavaML.
- If a method in a participating class properly calls another participating method, this can be located in both the same class as the calling method or another participating class.
- If the return type of the participating method is correct (i.e. if the type is an object present in the Design Pattern, or a Java primitive type).

Verification is only possible when the mapping relationship is properly established.

3.3.2 Class viewer

3.3.2.1 A list of the classes/interfaces

The user has access to a list view of all the classes and interfaces created. The fully qualified name of the classes and interfaces will appear.

3.3.2.2 A list of the methods

The user has access to a list view of all the methods created, including method signature (method name and parameters), return type and the class or interface in which the method resides.

3.3.3 Pattern catalogue

The pattern catalogue contains not only standard pattern *[5]* but also customized pattern. Customized pattern could be as simple as a modification of an existing Design Pattern, a combination of the several existing standard patterns or it could be a more complex Design Pattern designed for a specific problemdomain.

It is recommended that the customized pattern to be testet before being added to the Design Pattern catalogue.

3.4 System implementation

In this section the details of the implementation of the workbench is explored, this includes screenshots of the intented user interface. The conversion from Java to JavaML is introduced firstly as this is the foundation for the latter implementation. The main focus is on the mapping creation and the verification parts, which are key to the implementation of SHELL.

3.4.1 Java to JavaML conversion

Whenever the Java plain text source code is loaded or saved in the workbench, it will be automatically converted into the corresponding JavaML code - the user is not aware that this conversion is performed. This reflects the idea that the JavaML underlying structure should be transparent to the user.

The user has the option of manually convert the Java source into JavaML and gain access to the JavaML files with a read-only privilege. As the JavaML file is

the underlying infrastructure on which the workbench is based, it is most important to ensure its integrity and consistency. Therefore, the user can not make any change directly to the JavaML source code. Inappropriate alteration would cause serious error.

3.4.2 UML editor

A UML editor is embedded as part of our workbench. From the pattern catalogue, the software designer can select the Design Pattern intended to be used. And then the programmer drops the selected Design Pattern template onto the UML editor. The overall structure of the selected Design Pattern is then shown in the UML editor. In most case, it will be a class diagram. This becomes the starting point of the design. The programmer edits the UML diagram by filling in more details into the templates and by replacing the generic classes/methods by the specific. For example, the "request" method in the Proxy Design Pattern should be replaced by a similar actual method in the Design Pattern implementation. Any third party UML editor that is based on XMI can be integrated into SHELL.

3.4.3 Topic map creation

To create topics and associations that captures the UML data model as the Design Pattern information, SHELL converts the XMI into Topic map by using XLST transformations. The Topic map can manually be read and edited by any software designer. Usually these two approaches are used in combination. The Topic map has to be stored persistently. And SHELL ensures that the generated Topic maps conform to the desired schema.

3.4.4 Mapping creation

The mapping creation process is divided into three steps, as presented in the following sections:

3.4.4.1 Mapping step #1 (JavaML Viewer)

The programmer selects the Java source file from the window as shown in *Graphics B.3.* (located in Appendix B.). The structure of source code is evident through the viewer, in this view all the implemented classes/interfaces as well as all the implemented methods are shown. Next, the programmer selects the classes/interfaces and the methods, which are relevant to the current Design Pattern.

All the classes/interfaces and methods can be shown from this window with only a little effort thanks to the JavaML. By parsing the JavaML as XML document, the information can be easily queried. For example, in order to obtain all the classes name, one has to query the "id" attribute of all the element named "class" as following:

```
XmlNodeList className = xDoc.GetElementsByTagName("class");
foreach(XmlNode n in className)
{
  Add(n.Attributes.GetNamedItem("id").InnerText);
}
```

The interface and method name can also be queried similarly.

Notice that it is possible for select more than one JavaML file and has the information shown in the JavaML viewer. This meets the requirement that there are usually more than one Java source code file involved in each Design Pattern implementation.

3.4.4.2 Mapping step #2 (Topic map viewer)

In the second window as shown in *Graphics B.4.* (located in Appendix B.), the programmer is presented with an overview of all the participanting classes and methods in the Design Pattern template. The Topic map API makes this a trivial task.

3.4.4.3 Mapping step #3 (Mapping view)

Have the list of all the classes/interfaces and methods from the "JavaML viewer" and the participants and operation from the "topic map viewer", the program establishes the mapping between in the third window as illustrated in *Graphics B.5.* (located in Appendix B.). The mapping can be added, removed, saved and loaded from an existing mapping file.

3.4.5 Implementing verification

In this section, the implementation details of verification are explained respectively for each type of different verification. It is described stepwise and is transparent to the implementation programming language details.

3.4.5.1 Inheritance check

The process of inheritance checking is as follows:

1. For any particular Design Pattern instace, examine the corresponding Topic map file and look for all the "Association" of type "Extend", where the inheritance relation information for that pattern are stored.
2. For each of all the "Assertion" of type "Extend", get the "parent" and "child" pair.
3. Examine the mapping file, look for the corresponding classes for each "parent" and "child" pair and store them as "parent class" and "child class" pair.
4. For each "parent class" and "child class" pair, examine the corresponding JavaML file, check if the JavaML element "class" has en id of the "child class" has a sub-element named "super-class" whose id equals to the id of the "parent class"

A similar approach is used when checking the implementation of an interface, namely if a class inherits (implements in Java terminology) another interface appropriately.

In step (1) and (2), the TM4J (Topic Map for Java) API is used in order to conveniently retrieve information necessary for the checking process.

3.4.5.2 Check method call

The process of inheritance checking is as follows:

1. For any particular pattern, examine the corresponding Topic map file and look for all the "Association" of type "Methodcall", where the method calling relation information for that pattern are stored.
2. For each of all the "Assertion" of type "Methodcall", get the "caller", "calle" and "methodcalled" triple.
3. Examine the mapping file, look for the corresponding classes for the "caller" and "callee" and the actual methods for the "methodcalled" in the triple. And then store them appropriately.
4. For each "caller and "callee" pair, examine the corresponding JavaML file, check if the JavaML element "class" has an id of the "caller".
5. For each JavaML element "class" with an id equal to the "caller" check if it has a sub-element named "send", if the "target" attribute equals to the

id of the "callee", and if the "message" attribute equals to the id of the "methocalled".

3.4.5.3 Check health

The process of inheritance checking is as follows:

For any particular pattern, examine the corresponding Topic map file. For each topic, look for all the "Occurrence", where the constraints of the topics are stored. If the "Occurrence" is of type "attribute" as specified by the "instanceOf" element", then:

a) Get the attribute value from the content (innertext) in "resourceRef" element.

b) Examine the corresponding JavaML file, check if the JavaML element "filed"has an attribute "name "with the same value as the attribute value read from previous step.

If the "Occurrence" is of type "method" as specified by the "instanceOf" element", then:

a) Get the method value from the content (innertext) in "resourceRef" element.

b)Examine the corresponding JavaML file, check if the JavaML element "method"has an attribute "name "with the same value as the attribute value read from previous step.

3.4.5.4 Check cardinality

The cardinality can only be checked when the instance creation is manifested by the returning of the created objects values in a method. This is due to the fact that SHELL is constrainted by ignoring the method body. The initialization of objects within the method body can therefore not be examined.

4 Experiments

The next sections contains a description of the experimental works conducted with the workbench, with a main focus on the verification module described in the previous sections, as it is a core part of the entire workbench. The goal of this experiment is to investigate whether the tool is both sound and complete. *Sound* means the tool is able to do something of value in certain case and *complete* means the tool is able to do something of value in all cases.

The experiment defines a foundation for the discussion and conclusion in later chapters.

4.1 Experiment 1: The verification module

In this section experiments on the verification module is performed. The three verification functionalities are experimented with, in order to ensure they perform as expected. As the starting point, a Java source file with correct pattern implementation is fed into the workbench. The workbench reports no error information as expected. After this a series of well-defined intentionally erroneous Java source codes are fed to the workbench to discover if it reports the errors as expected way.

First of all, the check inheritance function is tested by changing the following statement:

```
class Order implements OrderIF
{
...
}
```

into:

```
class Order {
...
}
```

Of course, this code has to be re-converted into JavaML and fed into the workbench again.

The class "Order" participates as "real subject" and the interface "OrderIF" participates as "subject" in the Proxy Design Pattern - this has already been specified in the mapping file by the programmer. According to the Topic map Design Pattern template the "real subject" should extend or implement the "subject" participant.

Therefore this is an implementation error and is reported by the workbench as shown in *Graphics B.8.* (located in Appendix B.).

SHELL makes the following conversion:

```
<class name="Order" id="LOrder;" idkind="type">
  <superclass name="Object" idref="Ljava/lang/Object;" idkind="type"/>
    <implement interface="OrderIF" idref="LOrderIF;" idkind="type" />
```

into

```
<class name="Order" id="LOrder;" idkind="type">
  <superclass name="Object" idref="Ljava/lang/Object;" idkind="type"/>
```

Obviously, the JavaML element <implement interface="OrderIF" idref="LOrderIF;" idkind="type" /> is missing. The absence of this statement is detected by the Xpath expression /java-source-program/java-class-file/class/implement. Similarly, the inheritance error caused by removing the "extend" keyword from the statement can also be checked. As can be seen above, the checking inheritance function is satisfactory.

Secondly, the method call checking function is tested by removing the required method as follows, from the following conversion:

```
class OrderProxy implements OrderIF {
  ...
  public Vector getAllOrders() {
  ...
  Vector v = order.getAllOrders();
  ...
  return v;
}
```

into:

```
class OrderProxy implements OrderIF {
  ...
  public Vector getAllOrders() {
  ...
  Vector v = null;
  ...
  return v;
}
```

The "Proxy" class "OrderProxy" should explicitly calls the request method "getAllOrder()" in the "real subject" class as one of the constraints detailed in the Proxy

Design Pattern template. If the method is not called, as in this case, the workbench will raises a method call error as shown in *Graphics B.9.* (located in Appendix B.).

Looking at the underlying JavaML for the this Java source code, the change can easily be detected. The calling to a method is manifested as:

```
<send message="getAllOrders" idref="LOrder;getAllOrders()Ljava/util/Vector;" idkind="method">
  <target>
    <var-ref name="order" idref="LOrderProxy;var1225" idkind="local-variable"/>
  </target>
  <arguments/>
</send>
```

The "message" attribute of the "send" element is the name of the method being called and "idref" attribute is the fully qualified name of the method. The "target" subelement tells the callee class where the method is residing. The callee class is "order" in this case. If the "getAllOders" method is not called, the above "send" element will simply disappear. The function for checking the health is tested. As described in the previous section, there are three different types of health to be checked, namely the modifier, the attribute and the method. In order to verify if the attribute is present, one has to check if the "class" element has a subelement "field" whose id equals to the attribute it should contain.

```
class OrderProxy implements OrderIF {
  private Order order = new Order();
    public Vector getAllOrders() {
    ...
    Vector v = order.getAllOrders();
    ...
    return v;
  }
}
```

In this case, the "OrderProxy" as a "Proxy" class should have a reference to the "RealSubject" class "order". In the JavaML file, the reference to the "order" class is:

```
<field name="order" id="LOrderProxy;order" idkind="field">
  <modifiers>
    <modifier name="private"/>
  </modifiers>
<type name="Order" primitive="false> </field>
```

If this is missing, the checking health will fail and report an error. However, in reality, this is not always the right way to verify it. If one class must contain a reference to another class according to the Design Pattern specification, this class

may have that reference as a filed as in the above example or as a local variable. Effectively, it is the same in both cases. However, in the latter case, the workbench will generate en error. This indicates that it doest not make any big sense to check if an attribute of a class is present or not as it may appear as a local variable functioning a reference to the other class and is effectively the same. There is no such problem to check the present of the method declaration in a given class. In the above example, the method "getAllOrders ()" is required to be part of the class "OrderProxy". The workbench will simply check if the following code is present or not:

```
<method name="getAllOrders" id="LOrderProxy;getAllOrders()Ljava/util/Vector;" idkind="method">
  ...
<method>
```

4.2 Experiment 2: Design Patterns

In this section a check to test whether the tool is *complete* is performed. The completeness of the tool is defined as the fact that it works without any faultiness in all possible Design Pattern implementations. However it is not possible to enumerate all the Java Design Pattern implementations and test all of these in SHELL. The simplest solution is to group all the different implementation and select a few representative candidates in each of the groups: creational, structural and behavioural Design Patterns*[5]*. In this experiment SHELL is tested using the Abstract Factory, Proxy and Observer Design Patterns, respectively. As experiment with the Proxy Design Pattern has been done in the previous section, the following will on ly test the Abstract Factory Design Pattern and Observer Design Pattern here. Input: Two non-trial Java source codes, each of which has the above-mentioned two Design Pattern implementations.

4.2.1 Checking using the Observer Design Pattern

From the experiment on this Design Pattern, there are certain aspects that the tool can check successfully. For example each "Observer" must contain an "update” method; while each "Subject" must contain an "addObserver” method as well as a "notifyAllObserver" method. Interestingly SHELL is able to check if the "update" method in the "Observer" is called by the "notifyAllObserver" method in the "Subject" class. These are some of the constraints on the Observer Design Pattern. However, they are not all and hence there are problems with the verification of the Observer Design Pattern implementation.

Since there is no strict inheritance relation among the participants in Observer Design Pattern as in the Proxy Design Pattern, for example, the "Observer" and "subject" do not inherit one another; the function "check inheritance" can not be of any use. The more important point to verify in the Observer Design Pattern is that the "Subject" always notify its "Observers" when its state changes. This cannot be verified by simply checking the existences of certain inheritance relation or method call. There is also requirement for the method call operation taking place in a certain manner, for instance, a certain sequence.

Thus, the verification of tool is not sufficient for Observer Design Pattern as it doest not capture the Design Pattern's most essential aspect. Given a Java source code implementation of Observer Design Pattern, the implementation is NOT necessarily correct even when all the verification reports no error.

And also, an unrelated method call operation inserted between the state change of the "Subject" and the call to the "notifyAll" should be allowed and not be considered as an error. However, it is very difficult for a tool to detect this and check if the "notifyAll" is called at the proper place or not. This aspect is considered as ordered interaction among objects in Design Pattern, which is normally expressed by UML sequence diagrams.

Our workbench doest not currently support sequence diagrams. Thus, the behaviour Design Pattern cannot be fully verified.

4.2.2 Checking using the Abstract Factory Design Pattern

Again, the tool is able to verify the inheritance, method call and health constraints enforced on abstract factory pattern. For example, the "concreteFactory" must be a subclass of the "abstractFactory" and the "concreteProduct" has to inherit the "abstract product". In order to verify the correctness of a creational patter, one has to ensure that the required objects are instantiated properly. Identify the creation of objects is only possible when the initialization of objects is explicitly returned from the method, which can then be examined by checking the return statement.

However, in the implementation of creational Design Patterns, one is not restricted to return the created instances explicitly in the return statement.

In the implementation of Abstract Factory in this experiment, it could be the case that only an object" reference is returned from the method where the "concrete product" is created. Even though a "concrete product" type of object is expect-

ed according to the design. And this return "object" can be upcasted to "concreteproduct" later on. Even though with a tiny problem, the result of the checking module meets the overall expectation. Thus, the tool is *sound* in the current scenario. In other words, it works smoothly in the context of the Java Proxy Design Pattern implementation. Provided a Java source code with a Proxy Design Pattern implementation a pass through all these checking yields that the implementation is correct i.e. a Proxy Design Pattern instance is present.

To conclude, the tool performs not very satisfactorily in either behaviour Design Pattern or creational Design Pattern. In some sense, the tool can only verify the structural aspect of behaviour Design Pattern and creational Design Pattern. Neither behavioural Design Pattern nor creational Design Pattern implementation is completely correct if only the structure is correct. Therefore, the tool is *sound* but not *complete*

5 Discussion

Having created the SHELL workbench and conducted a series of experiments the focus of this chapter is to evaluate the usefulness of the initial idea and the created workbench.

SHELL is restricted by the individual Design Patterns and to which extend it is possible to formalise these. As an example of this, problems occurred when using the workbench to express Design Patterns with cardinality constraints (e.g. Abstract Factory Design Pattern). However Design Patterns that does not contain such cardinality constraints often relies more on structural aspects. For such Design Patterns it is possible to convey more information using the structurally oriented OMT diagram (e.g. Observer Design Pattern).

Each Design Pattern template defines the requirements that the individual Design Pattern instances must fulfil. The SHELL the fulfilment of these requirements can be checked. This is done by checking various aspects of the participating classes and methods: inheritance, method calls, health, and cardinality. Even as it is possible to check such aspects individually it must be noted that only combined will the Design Pattern operate as originally presented.

Conducting the experiments in the previous chapter it became apparent that SHELL performs best with the Structural Design Patterns followed by the Creational and Behavioural Design Patterns. It is reasonable to believe that this is closely related to how much information it is possible to illustrate using UML.

In reality, when using the workbench, there are several transformation stages are involved in the workflow, which indicates that whenever any change occurs in the starting end of the flow, e.g., the programmer modifies the Java source code or the software designer edits the UML diagram, the entire transformation process has to be redone all over again. This will significantly also affect the scalability of the infrastructure.

6 Conclusion

To recap from a previous section, the central question posed by this project is *When developing a software development workbench that facilitates checking for the presence of Design Patterns, which advantages (if any) can be obtained by using a richer structured XML representation for the underlying storing of both source code and Design Pattern knowledge?"*

This report document that it is possible to introduce structural knowledge into a development workbench. And the benefit of this is that the software designer has all the necessary information, from design document and the sourced code, integrated into one single tool. The documents and the source codes are no longer heterogeneous information floating around but is one single infrastructure instead.

And the structural knowledge is highly standardized and categorized so that anyone can adopt the knowledge by simply grapping one from the catalogue. It also promotes the reusability of the structural knowledge. The customization of the structural knowledge makes the infrastructure flexible enough to be of practical use.

Most attempts to formalise Design Patterns is based on UML. However UML still does not perform well with the Behavioural Design Patterns. It is possible that this problem could be relieved if more focus was on the dynamic run-time aspects of Design Patterns: Sequence Diagrams and Object diagrams.

In general it is possible to include structural information as presented in this project. It is reasonable to conclude that a successful inclusion of Design Pattern knowledge would allow all lesser complex structural information to be included. However the definition of some Design Patterns is still too weak to be properly formalised for inclusion.

The fact that SHELL did not successful included all Design Patterns does not necessarily limits the inclusion of all structural information. The level of abstractness and quality of formalisation of structural information dictates the usefulness of such information.

One must have a deep understanding of Design Pattern knowledge before using SHELL - it cannot replace the software designer’s own understanding of the usage of Design Patterns. The software designer is the authority on Design Pattern issues, including which pattern to use, when and how.

7 Further Work

Sophisticated tools and development suites like eclipse (www.eclipse.org) parse the source code while the developer types in order to obtain information on potential syntactical issues. Often this parsing functionality is provided in a separate thread of the development suite, and is included in order to provide the developer with hints to problems and mistakes that would otherwise only be revealed as errors and warning during compilation. Examples include typos and names disallowed by the compiler and/or programming language.

SHELL does not, in the current form, focuses on the visual presentation of information to the user. The following describes how it is possible to use the Design Patterns stored as SVG to improve the interaction the user has with SHELL.

The idea is that the programmer is presented with a familiar UML diagram of each instance of a Design Pattern. Using this diagram as a reference the programmer is able to navigate to the actual information stored in the underlying structure.

As an example the illustration in *Graphics B. 10.* (located in Appendix B.) contain two panels: a navigation panel (on the left) and a text editor panel (on the right). In the navigation panel a SVG illustration of the Proxy Design Pattern is shown. When the user clicks a participating class or method in this illustration, the text editor will change to display the corresponding class or method as indicated in the mapping file.

7.1 Extended information on SVG

Vector graphics is used to store a series of geometric primitives (point, lines, curves and polygons) in mathematical form. The benefit of this is that it is possible to zoom both in and out, without ever affecting the quality of the displayed image (as is the case with raster graphics). In general vector graphics has advantage over raster graphics when describing images made up of such geometric figures.

The notation of UML is made up of the geometric figures. By viewing a variety of UML diagrams, it becomes obvious that the entire notion of UML is based on lines, polygons, ellipses and text labels.

Vector graphics stored as XML is named SVG and uses a dedicated SVG namespace (http://www.w3.org/2000/svg). Each SVG file is made up of a single canvas where the height and width are specified as attributes. Nested within this canvas

are the geometric elements that are to be displayed. It is interesting to note that such geometric elements can be nested within other geometric elements.

7.2 Converting XMI to SVG

The open source project uml2svg (uml2svg.sourceforge.net) focuses on extracting the visual elements stored within XMI as SVG. The XMI input to the uml2svg tool must be structured in XMI and be constructed to support the UML 2.0 Diagram Interchange Specification*[14]*. With this in mind it is crucial to understand that not all XMI formats are usable as input for the uml2svg tool. If the information is indeed compliant with the above standards, a transformation using the uml2svg on the XMI will output a SVG file.

Each class within the resulting SVG file is made up of a box. Within this box are two lines: one for separating the header compartment of the class from the attribute compartment, and one for separating the attribute compartment from the method compartment. The header compartment contains a text label. The attribute compartment and the method compartment are made up of text labels. Any of these text labels could be italic.

Relationships between classes are created by a single line from one class to another class. Each end of this line is marked with an identifier. This identifier is used to create the arrow at the end (when required).

7.3 Appending required information to the SVG

SVG does not contain any information related to linking. By including both the namespaces of SVG and XLinks, it is possible to store both vector graphics and linking information in a single file.

As illustrated in *Figure 6* and *Figure 7* a linking element is inserted by grouping the elements that will make up the body of the link and supplying the target of the link as an attribute on this linking element. The figures illustrates how the former

figure is converted into the later figure, where the nodes “d1” and “d2” constitutes a link. The target of this link is set as an attribute on the linking element.

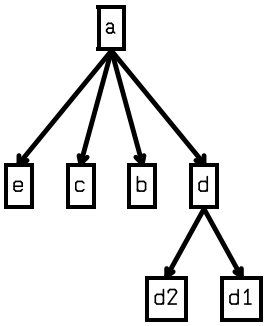


Figure 6. Tree-view of SVG without linking.

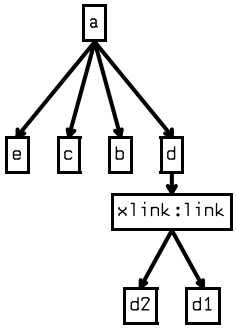


Figure 7. Tree-view of SVG with linking.

The linking information to the proper JavaML file can be located in the mapping file by using name coincidence, as both the SVG file and the mapping file originates from the same XMI file.

As presented elsewhere certain aspects of design patterns are not easily represented. This constitutes a problem with the design patterns where several classes and methods can have a single role. In that case the mapping done is not a mere one-on-one correspondence but a more complex many-to-one correspondence. Simple linking is not powerful enough to express this fact as such a type of link is made up of only one source and one target.

When a more complex pattern needs to be introduced a more sophisticated way of structuring this is required. It is possible to use the more powerful extended linking, which allows a many-to-one correspondence to occur *[10] [9]*.

7.4 Summary of Sample Userinterface

A problem arises with Design Patterns where the number of participating classes and methods are not set by the Design Pattern template. In this case the mapping done is not a mere one-on-one correspondence but a more complex one-to-many correspondence. An example of this problem is the fact that there can be any number of “Proxy” participating classes in a single instance of the Proxy Design Pattern (as long as there are an equal number of “RealSubject” participating classes).

A similar problem is that the individual Design Pattern templates can have any number of actual instances represented in the source ode.

In essence the navigation from UML to JavaML code would operate similar to the imagemaps that exist for use on the Internet. However using the SVG and XLink combination yield a powerful alternative to such imagemaps that can be applied in further development of SHELL.

8 Bibliography

[1] Gabriel David Ademar Aguiar and Greg J. Badros. JavaML 2.0: Enriching the Markup Language for Java Source Code. XATA 2001, 2004.
[2] Greg J. Badros. Javaml: a markup language for java source code. Comput. Networks, 33(1-6):159–177, 2000.
[3] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from C++ source code. In ICSM ’03: Proceedings of the International Conference on Software Maintenance, page 305, Washington, DC, USA, 2003. IEEE Computer Society.
[4] Wensheng Li and Kenneth Vittrup. Towards an XML Enhanced Source Infrastructure.
[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of reusable object-oriented software. ISBN 0-201-63361-2. Addison-Wesley.
[6] Tichy, W. F. 1997. A Catalogue of General-Purpose Software Design Patterns. In Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems (July 28 - August 01, 1997). TOOLS. IEEE Computer Society, Washington, DC, 330.
[7] John M. Vlissides. Pattern Hatching: Design Patterns Applied. ISBN 0201432935. Addison-Wesley.
[8] SUN Microsystems. The java language specification, third edition, February 2004. http://java.sun.com/docs/books/jls/third edition/html/j3TOC.html.
[9] W3C. XML linking language (XLink) version 1.0, June 2001. http: //www.w3.org/TR/xlink.
[10] Improving Web linking using XLink. David Lowe and Erik Wilde.
[11] W3C. Extensible Markup Language (XML) version 1.0 (Third Edition), February 2004. http://www.w3.org/TR/REC-xml.
[12] arki.uiah.fi/4GDesign/pos.
[13] www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF
[14] www.omg.org/technology/documents/modeling_spec_catalog.htm#UML
[15] www.omg.org/technology/documents/modeling_spec_catalog.htm#XMI
[16] Object-Oriented Analysis and Design with Applications (2nd Edition). ISBN 0-8053-5340-2.
[17] Object-Oriented Software Engineering: A Use Case Driven Approach by I. Jacobson Publisher: Addison-Wesley Professional; 1st edition (June 30, 1992) Language: English ISBN: 0201544350
[18] Object-Oriented Modelling and Design James Rumbaugh ISBN: 0-13-629841-9

[19] Jonathan I. Maletic, Michael L. Collard, and Andrian Marcus. Source code files as structured documents. In IWPC ’02: Proceedings of the 10th International Workshop on Program Comprehension, page 289, Washington, DC, USA, 2002. IEEE Computer Society.

[20] E. Mamas and K. Kontogiannis. Towards portable source code representations using XML. In WCRE ’00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE’00), page 172, Washington, DC, USA, 2000. IEEE Computer Society.

[21] G. McArthur, J. Mylopoulos, and S. K. K. Ng. An extensible tool for source code representation using XML. In WCRE ’02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE’02), page 199, Washington, DC, USA, 2002. IEEE Computer Society.

[22] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise Modelling of Design Patterns

[23] James Rumbaugh, Ivar Jacobsen, and Grady Booch. The Unified Modeling Language Reference Manual. ISBN 0-201-30998-X. Addison-Wesley.

[24] Structural Modelling of Design Patterns: REP Diagrams

[25] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design Pattern Application in UML.

[26] Language Support for Design Patterns using Attribute Extension

[27] Bjarne Stroustrup. The C++ Programming Language (3rd Edition). ISBN 0-201-88954-4. Addison Wesley.

[28] en.wikipedia.org/wiki/Topic_map

Appendix A: Library of Design Patterns

I Adapter

When you need to implement an expected interface, you may find that an existing class performs the services a client needs but with different method names. You can use the existing class to meet the client's needs by applying the adapter pattern. The intent of Adapter is to provide the interface a client expects, using the services of a class with a different interface.

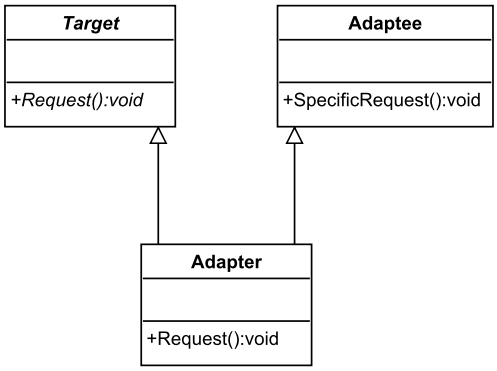


Figure A.1. Adapter Design Pattern template

II Facade

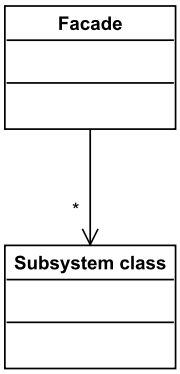


Figure A.2. Facade Design Pattern template

III Composite

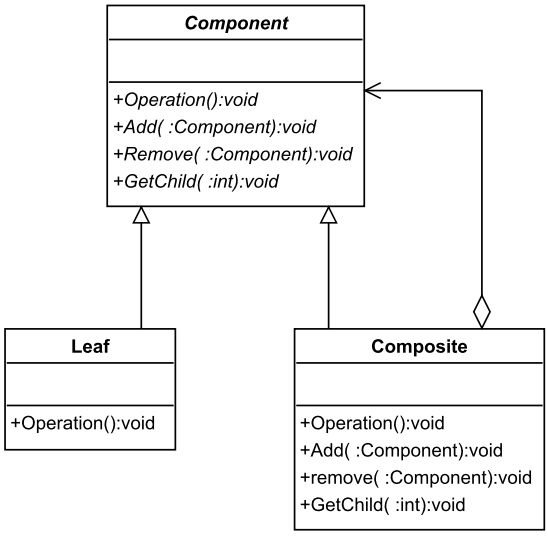


Figure A.3. Adapter Design Pattern template

IV Bridge

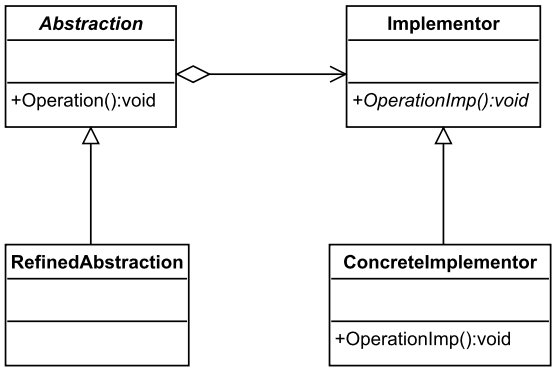


Figure A.4. Bridge Design Pattern template

V Singleton

The intent of the singleton pattern is to ensure that a class has only one instance and to provide a global point of access to it.

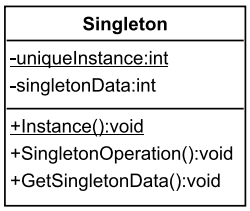


Figure A.5. Singleton Design Pattern template

VI Observer

The intent of the observer pattern is to define a one-to-many dependency such that when one object changes state, all its dependents are notified and updated automatically.

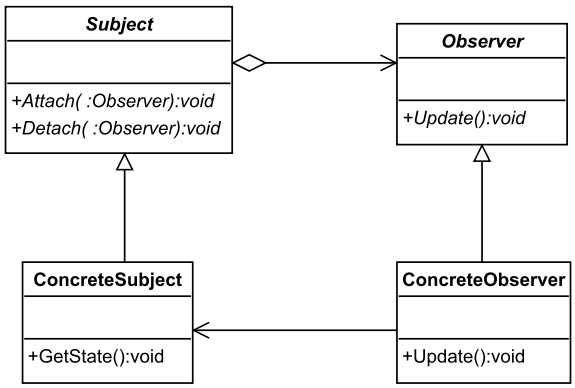


Figure A.6. Observer Design Pattern template

VII Mediator

The mediator pattern defines an object that encapsulates how a set of objects interact. This promote loose coupling, keeping the objects from referring to one another explicitly, and lets you vary their interaction independently.

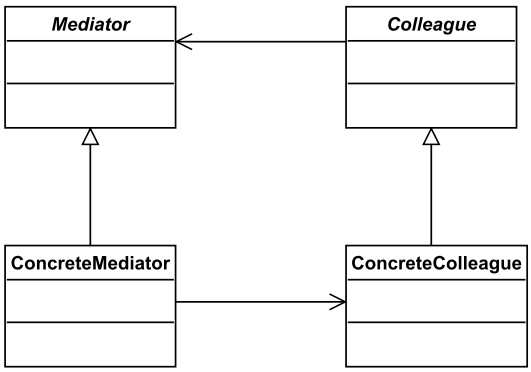


Figure A.7. Mediator Design Pattern template

VIII Proxy

The intent of the proxy pattern is to provide a surrogate, or placeholder, for another object to control access to it.

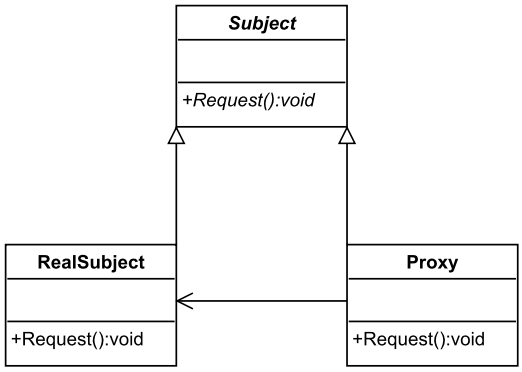


Figure A.8. Proxy Design Pattern template

IX Chain of Responsibility

The intent of the chain of responsibility pattern is top avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. To apply this pattern, chain the receiving objects and pass the request along the chain untill an object handles it.

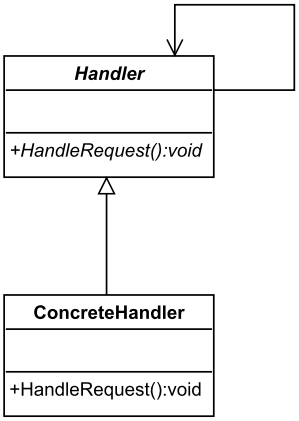


Figure A.9. Chain of Responsibility Design Pattern template

X Flyweight

The intent of the flyweight pattern is to use sharing to support large numbers of fine-grained objects efficiently.

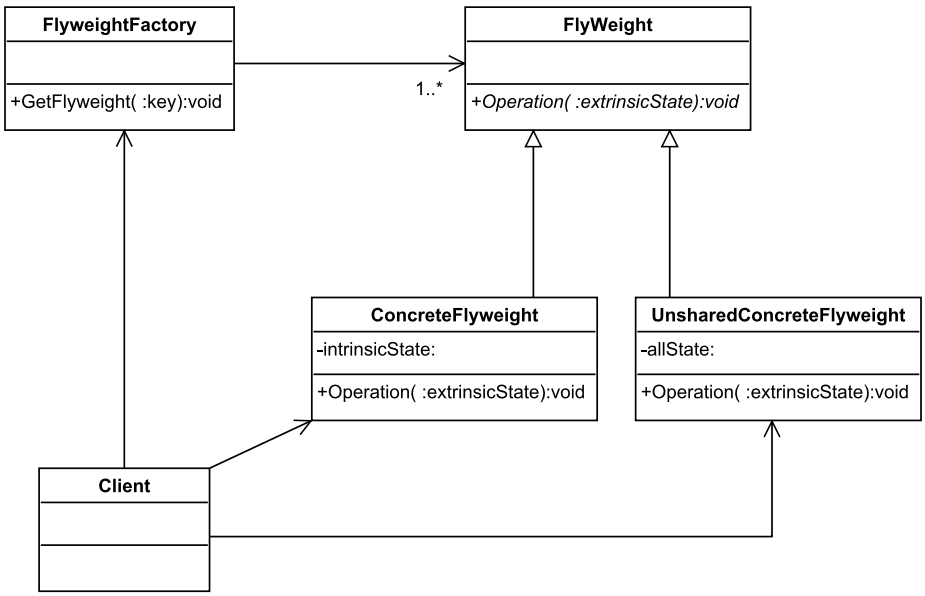


Figure A.10. Flyweight Design Pattern template

XI Builder

The builder pattern moves the construction logic for an object outside the class to instantiate.

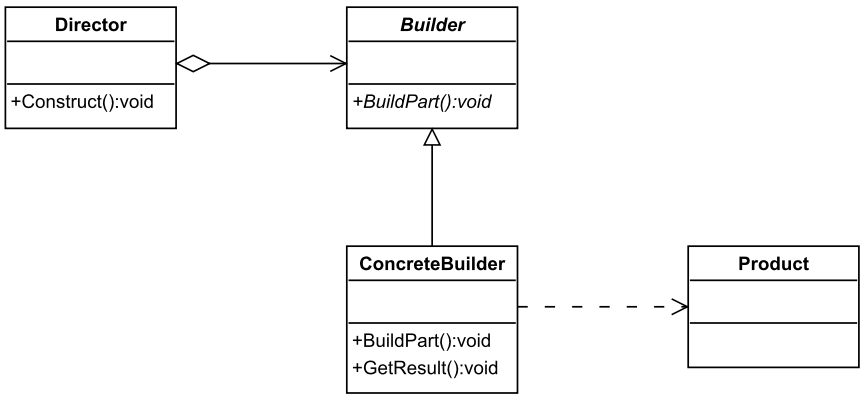


Figure A.11. Builder Design Pattern template

XII Factory Method

The Factory method pattern lets a class developer define the interface for creating an object while retaining control of which class to instantiate.

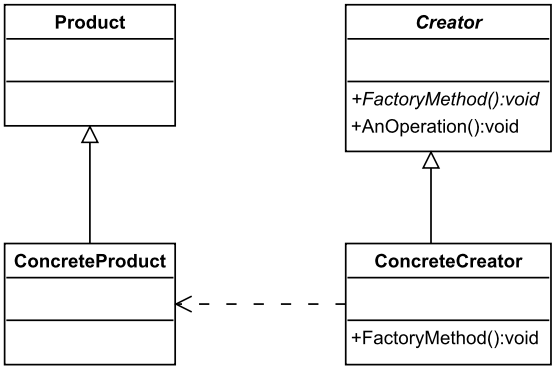


Figure A.12. Factory Method Design Pattern template

XIII Abstract Factory

The intent of this pattern is to provide for the creation of a family of related, or dependant, objects.

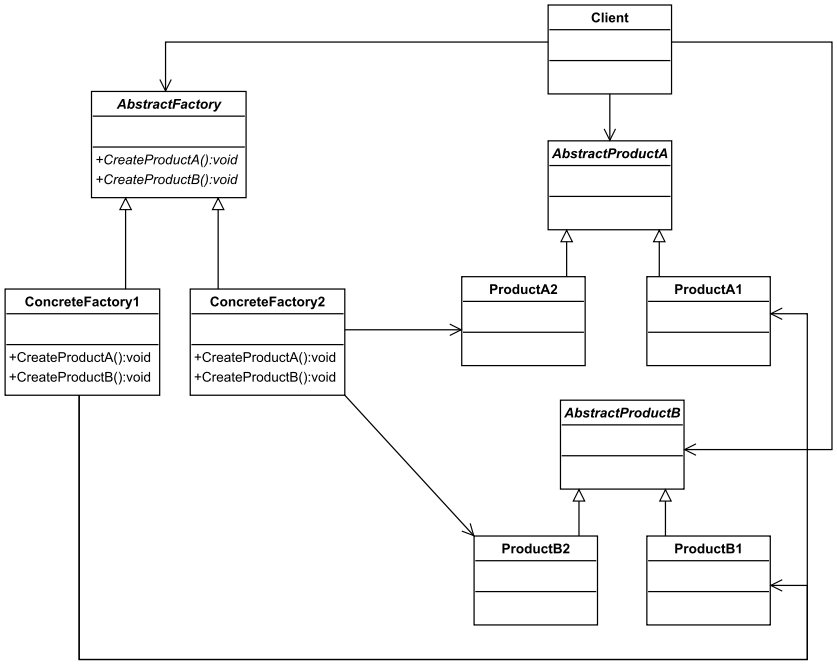


Figure A.13. Abstract Factory Design Pattern template

XIV Prototype

Instead of bringing forth new, uninitialized instances of a class. the intent of the prototype pattern is to provide new objects by copying an example.

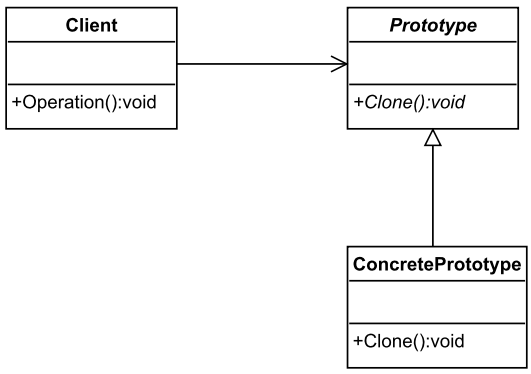


Figure A.14. Prototype Design Pattern template

XV Memento

The intent of the Memento pattern is to provide storage and restoration of an object's state.

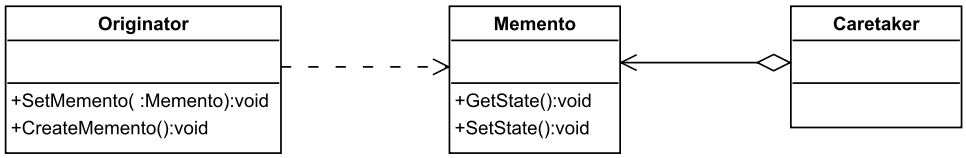


Figure A.15. Memento Design Pattern template

XVI Template Method

The intent of template method is to implement an algorithm in a method, deferring the definition of some steps of the algorithm so that other classes can redefine them.

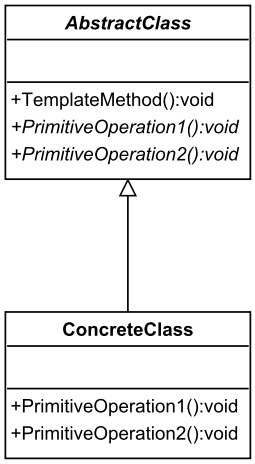


Figure A.16. Template Method Design Pattern template

XVII State

The intent of the state pattern is to distribute state-specific logic across classes that represent an object's state.

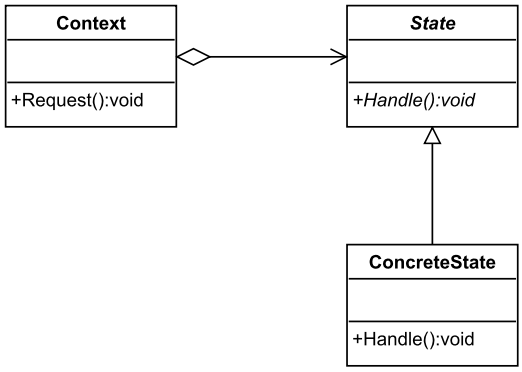


Figure A.17. State Design Pattern template

XVIII Strategy

The intent of strategy is to encapsulate alternative strategies, or approaches, in separate classes, each of which implement a common operation. The strategic operation defines the input and output of a strategy but leaves implementation up to the individual classes. Classes that implement the various approaches implement the same operation and are thus interchangeable, presenting the same interface to clients.

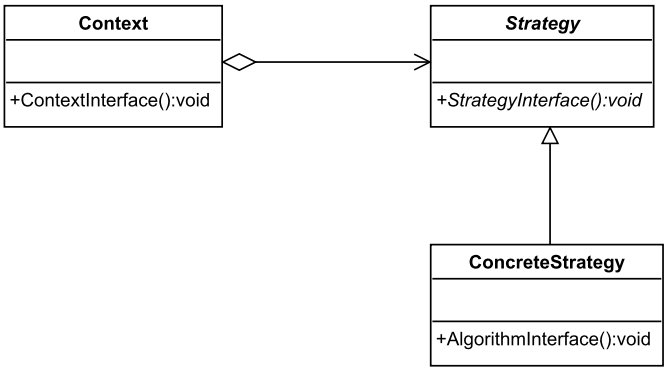


Figure A.18. Strategy Design Pattern template

XIX Command

The command pattern establishes a method signature, most often execute () or perform (), and lets you define various implementation of this interface (polymorphism in java).

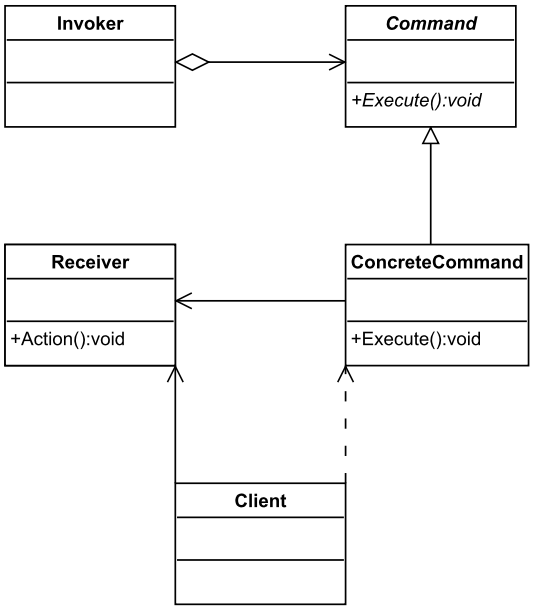


Figure A.19. Command Design Pattern template

XX Interpreter

An interpreter object conducts the execution, or evaluation of a collection of rules, letting you build expression evaluators and command languages.

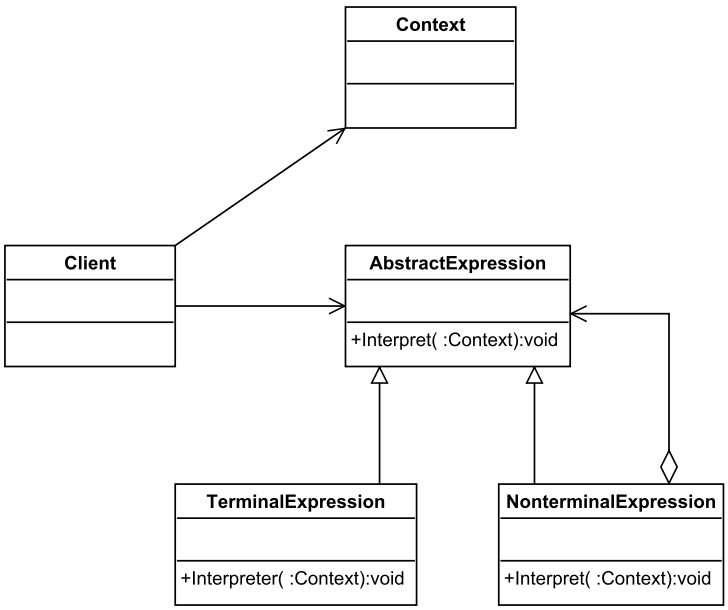


Figure A.20. Interpreter Design Pattern template

XXI Decorator

The intent of decorator is to let you compose an object's behavior dynamically.

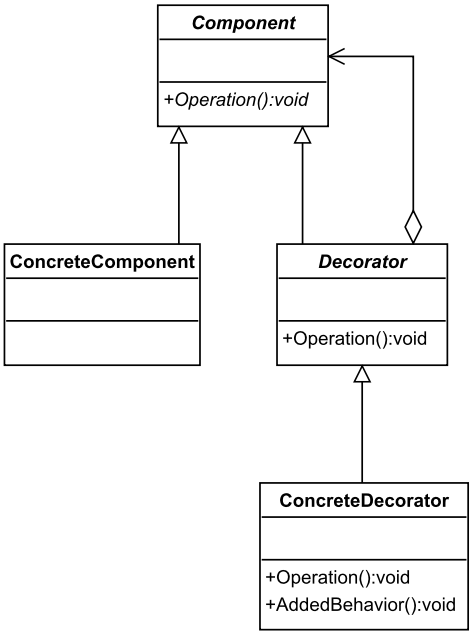


Figure A.21. Decorator Design Pattern template

XXII Iterator

The intent of the iterator pattern is to provide a way to access the elements of collection sequentially.

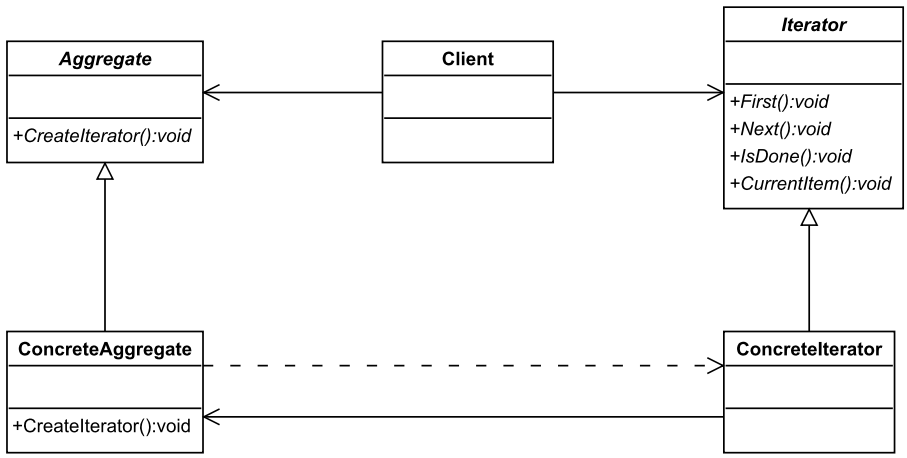


Figure A.22. Iterator Design Pattern template

XXIII Visitor

The intent of visitor is to let you define a new operation for a hierarchy without changing the hierarchy classes.

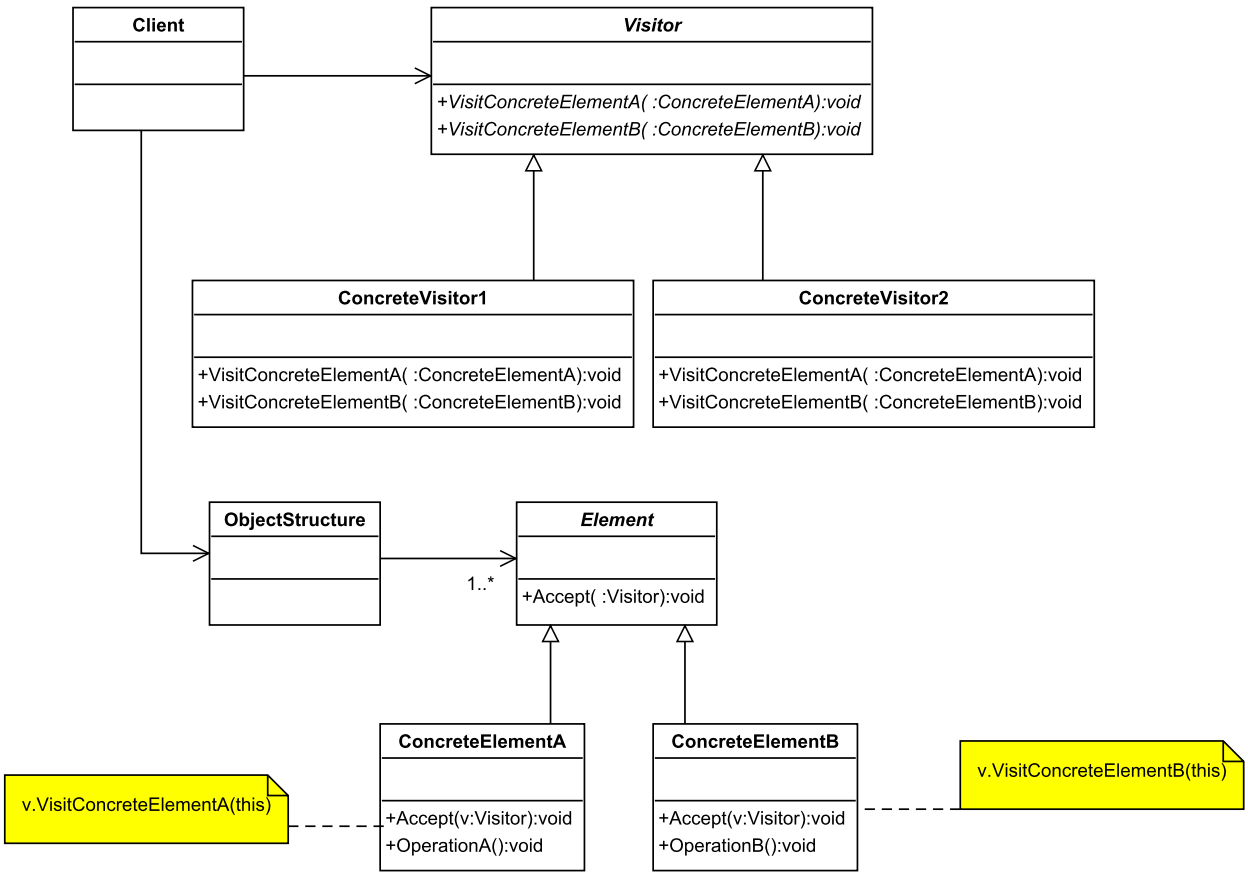
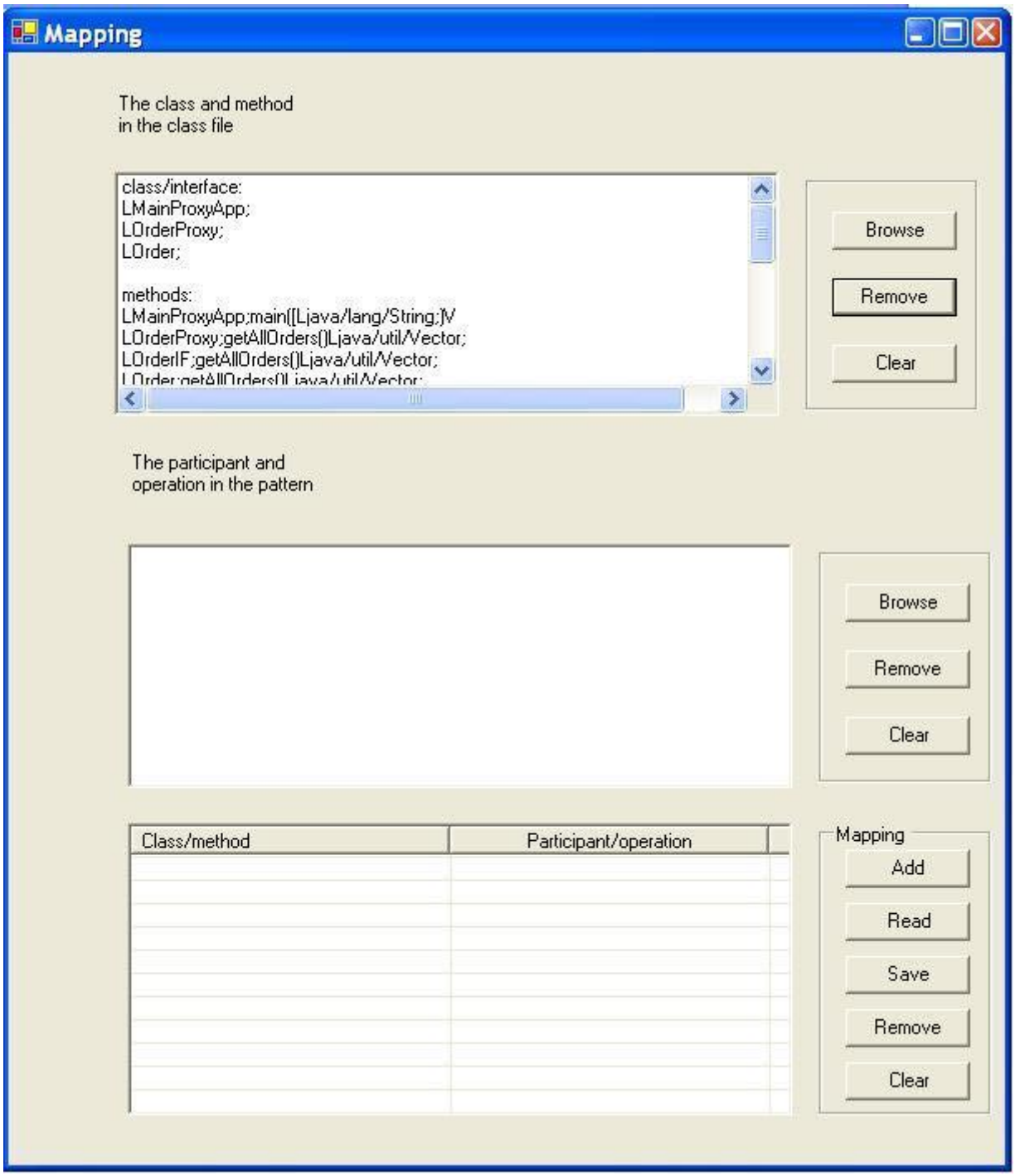


Figure A.23. Visitor Design Pattern template

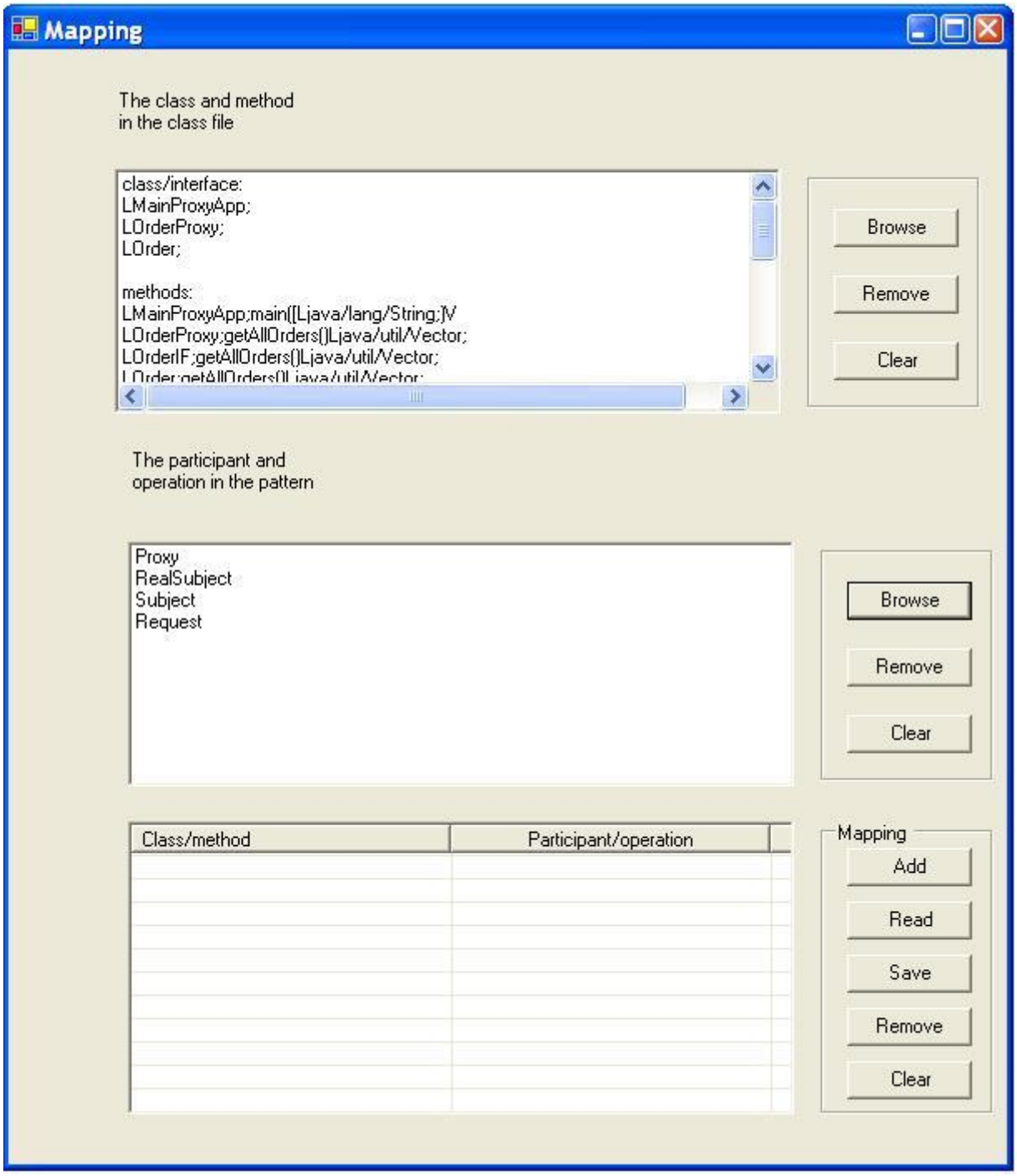
Appendix B: Miscellaneous Graphics

Graphics B.1. Graphics showing the pattern catalogue.

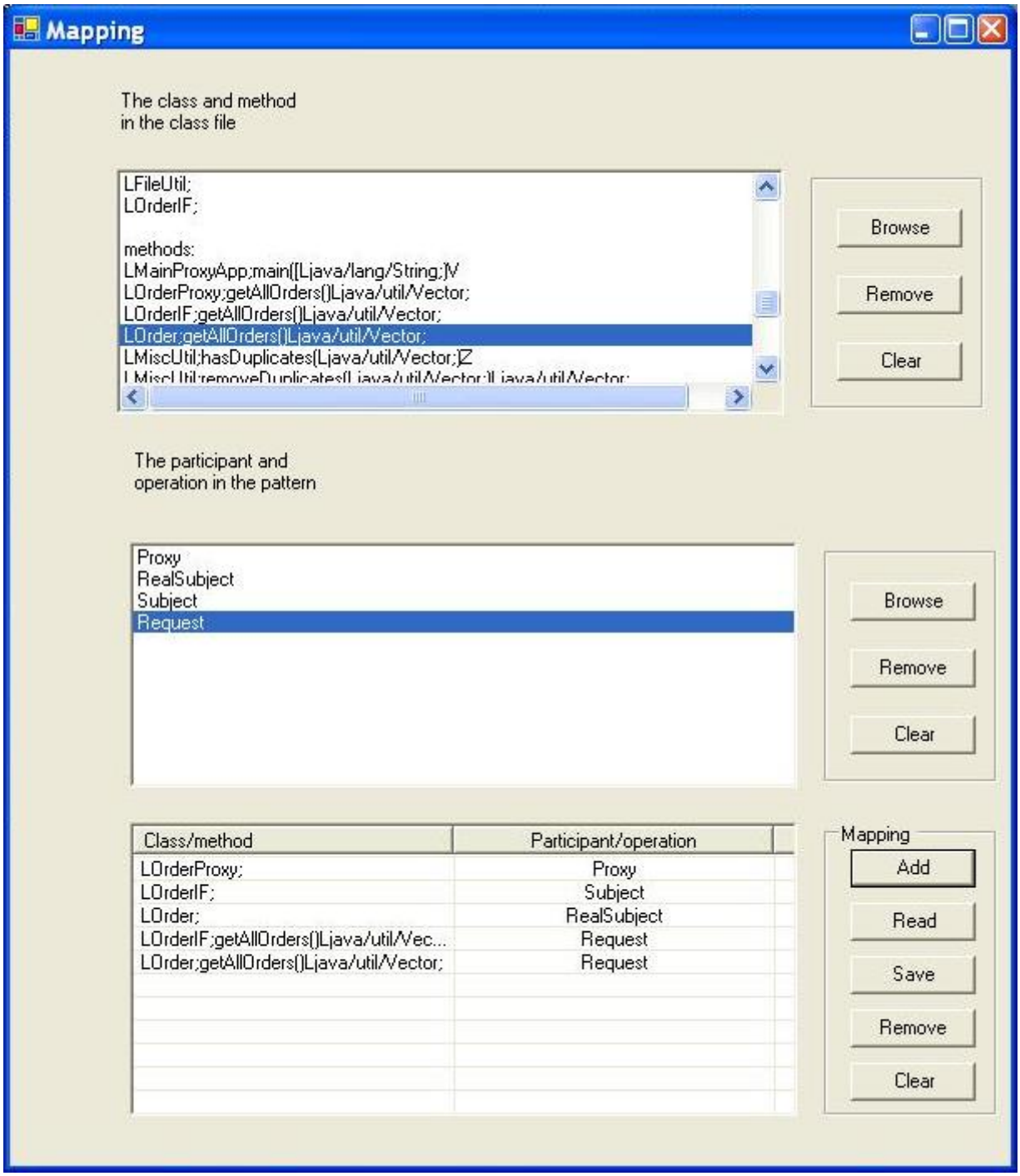
Graphics B.2. Java to JavaML conversion.



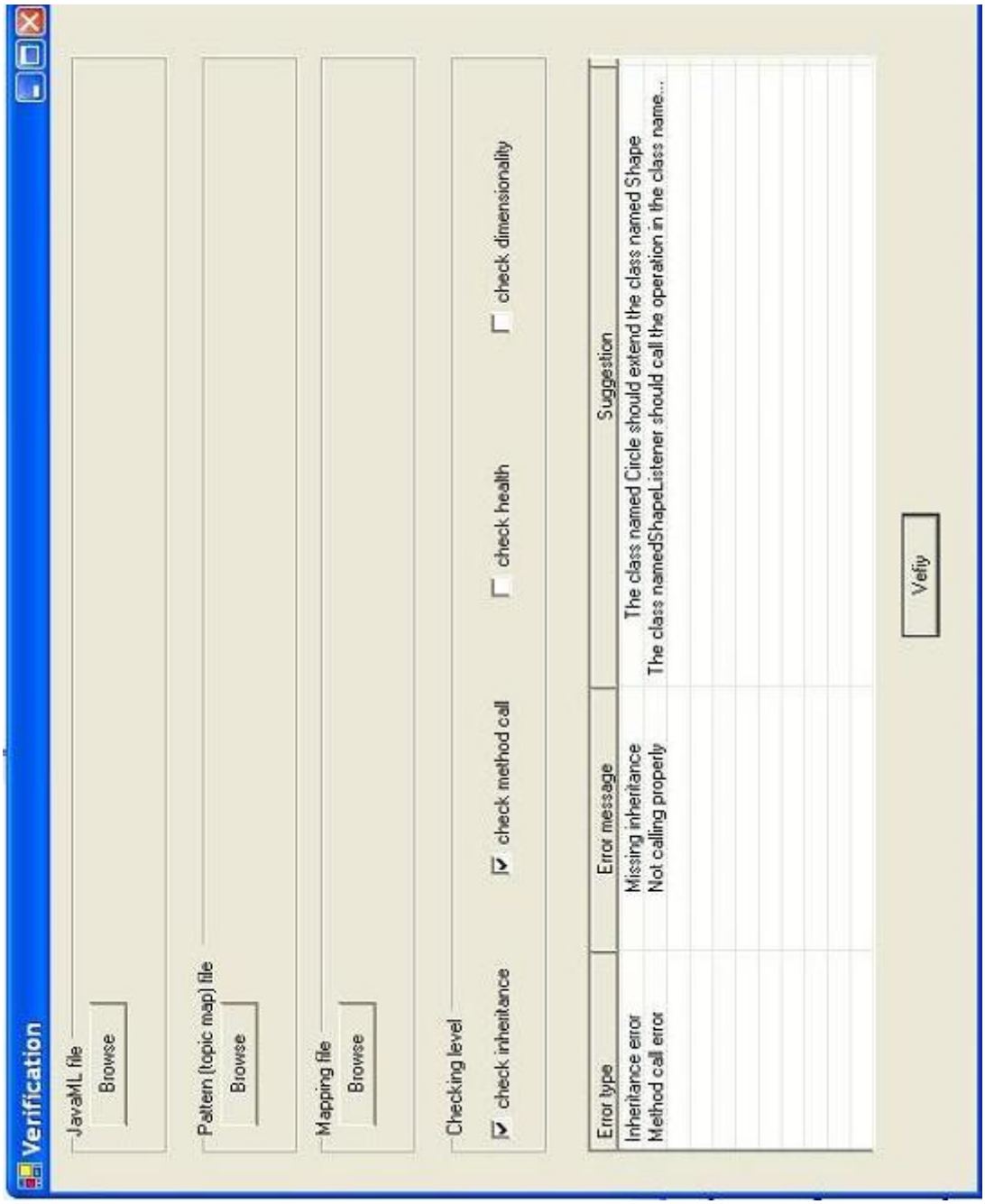
Graphics B.3. Mapping step #1.



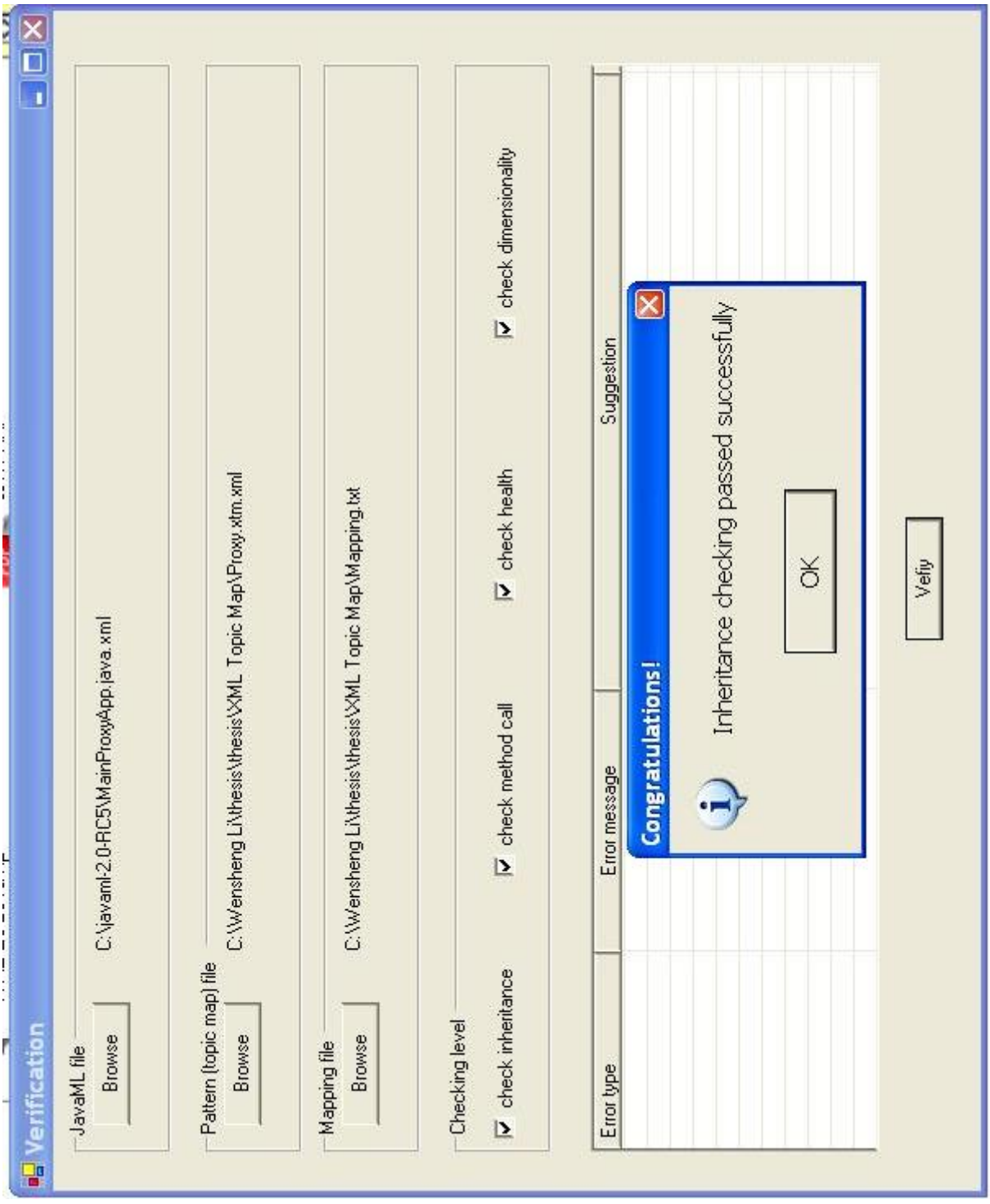
Graphics B.4. Mapping step #2.



Graphics B.5. Mapping step #3.



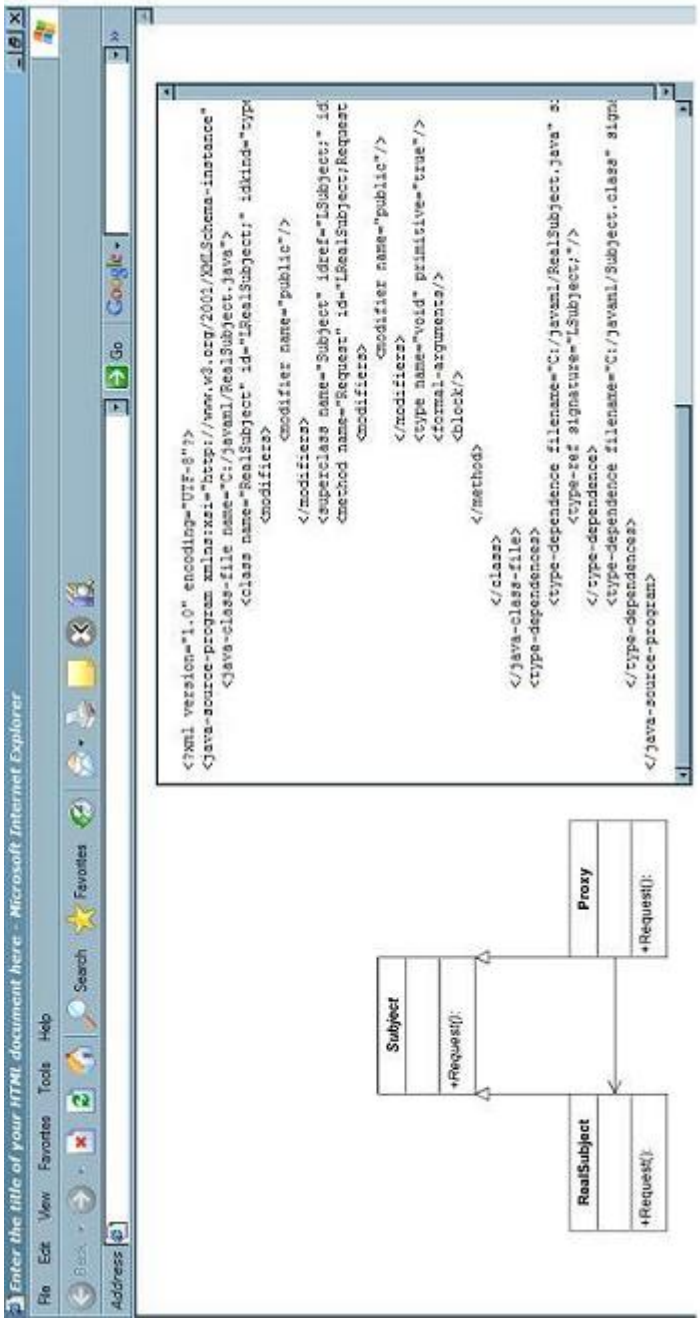
Graphics B.6. Verification.



Graphics B.7. Verification all succeeded.

Graphics B.8. Implementation error.

Graphics B.9. Method call error.



Graphics B.10. Possible outcome of further development.