The Role of Design Information in Software Evolution

Walter Cazzola[1], **Sonia Pini**[2], and **Massimo Ancona**[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it

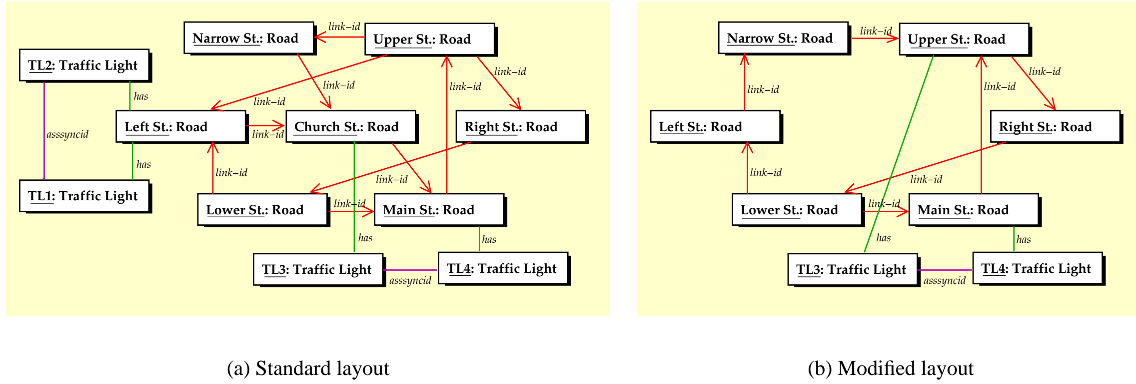
[2] Department of Informatics and Computer Science
Università degli Studi di Genova, Italy
{pini|ancona}@disi.unige.it

Abstract

Software modeling has received a lot a of attention in the last decade and now is an important support for the design process.
Actually, the design process is very important to the usability and understandability of the system, for example functional requirements present a *complete* description of how the system will function from the user's perspective, while non-functional requirements dictate properties and impose constraints on the project or system.
The design models and implementation code must be strictly connected, i.e. we must have correlation and consistency between the two views, and this correlation must exist during all the software cycle. Often, the early stages of development, the specifications and the design of the system, are ignored once the code has been developed. This practice cause a lot of problems, in particular when the system must evolve. Nowadays, to maintain a software is a difficult task, since there is a high coupling degree between the software itself and its environment. Often, changes in the environment cause changes in the software, in other words, the system must evolve itself to follow the evolution of its environment.
Typically, a design is created initially, but as the code gets written and modified, the design is not updated to reflect such changes.
This paper describes and discusses how the design information can be used to drive the software evolution and consequently to maintain consistence among design and code.

1 Introduction

In the last few years, methodologies to automate part of the or the whole software life cycle has been widely studied in the software system development. These methodologies can be used to create and/or maintain software, i.e. they are applicable to all the phases of the software life cycle.
Evolution and maintenance are phenomena more and more present in the software development area. Automatic techniques to support these phenomena are fundamental to improve the managing of unanticipated software evolution and the software efficiency.
The design process is very important to the usability and understandability of the system, for example functional requirements present a *complete* description of how the system will function from the user's perspective, while non-functional requirements dictate properties and impose constraints on the project or system.
The design models and implementation code must be strictly connected, i.e. we must preserve the correlation between the two views, and this correlation must exist for all the software life cycle. All system views must describe the same situation, but often, during the evolution a discrepancy between the two views can occur.
The software life cycle includes requirements analysis and specification, design, construction, testing, and maintenance (also called evolution). But often, the initial stages of development (the system specifications and the design) are ignored once the code has been developed. This practice causes several problems when the system must evolve, because the evolution of only one view of the system causes a *gap* between them, that could create confusion, misunderstanding and mistakes. For example, a kind of evolution could require to add new functionality not available in the earlier version of the system. When the change is not applied also to the design view, it is hard for the manager, programmer and customer to have the opportunity to plan future directions, goals, schedule and the necessary budget, since the design view could not provide an immediate and understandable global view of the system consistent with the code. Moreover, it also hinders the integration of new functionality.
When there is a gap between the views it is difficult or impossible to know all the necessary changes to apply, therefore the evolution cannot be planned *on-the-large*. It is necessary to have a *global view* of the system to apply all the evolutionary steps.
In our point of view, the global view may be well represented by the design information, because it is usually



(a) Standard layout

(b) Modified layout

Figure 1: Object diagrams: a) object diagram for UTCS before the evolution b) modified object diagram after the evolution.

graphic, and more intuitive and understandable than the code.

The paper describes and discusses how the design information can be used to drive the software evolution and consequently to preserve the consistence among design and code views.
There are several approaches that can be used to achieve consistent software evolution, such as to develop a comprehensive language that covers all the dimensions of software, or to develop a mechanism that integrates tools for different dimensions. Our approach is different, we propose to develop a middleware that uses a representation of software design strictly connected with the code of the system in question, and than that evolves together these two views.

2 System Evolution through Design Information

With the help of an example, we describe how the design information, in our case UML [1] specifications, can be used to evolve a software system.
The *Urban Traffic Control System* (UCTS) is a typical example of system subject to unpredictable evolution, since the requirements can dynamically change and the system should adapt itself, as soon as possible, to such changes. Examples of unexpected and hard to plan problems may be: road maintenance, traffic lights disruptions, car crashes, traffic jam and so on.
When designing *urban traffic control systems* (UCTS), the software engineer must model both mobile entities (e.g., cars, pedestrians, vehicular flow, and so on) and fixed entities (e.g., roads, railways, level crossing, traffic lights and so on).

The UTCS for a simple city, can be described by the object diagram showed in Fig. 1a, that defines the interconnections among roads and crossroads and a statechart that express the dependencies among traffic lights.
These diagrams well describe the system structure and behavior and its evolution should pass through these data to be well planned and integrated with the existing code. Therefore the information derived by all these diagram must go together with the system code as meta-data.
We suppose that the system evolves because of a car accident that temporarily blocks the traffic flow in *Church Street*. To face a similar event forces several small changes in the whole city structure and, consequently, to the traffic flow. Several streets must be followed in a different direction to allow cars of reaching every place in the city. Traffic lights governing the traffic in and out of the blocked street must be turned off.
All the changes required must be applied both in design and implementation view. The Fig. 1b show how the object model changes after this event. Using the two object diagrams, before and after the evolution, and the meta-data inside the code it is possible to propagate the evolution to the code.

3 Defining Meta-Data

To insert design information into the code of the system as meta-data, it is necessary to analyze what is a meta-data and how it can be used.

Meta-data literally *data about data*, or also *information about information* is a term used in several communities in different ways.
We can say that meta-data are structured information that describes, explains, locates or otherwise makes it easier to retrieve, use, or manage an information resource.
There are three main type of meta-data:

- *Descriptive meta-data* describes a resource for purposes such as discovery and identification, e.g., documentation.
- *Structural meta-data* indicates how compound objects are put together, e.g.,. they are used to describe the structure, layout and contents of an artifact.
- *Administrative meta-data* provides information to help manage an artifact, e.g., version control, location information, acquisition information.

An important reason to create descriptive meta-data is to facilitate the discovery of relevant information by describing an artifact with meta-data simplify its understandability by a program, promoting the interoperability. For our scope, we need to identify an artifact and to link up it with its design information. These meta-data could be automatically derived (extracted) from the design models of the system, and then automatically inserted into the code in the right places. To interleave the design information with the related code, is the better way of rendering the code well documented and of granting the consistency and a prompt update of the design and implementation views. There are two ways to obtain a high coupling between design information and system code, the first consists of deriving the design information from the system code, e.g. by using tools for reverse engineering it is possible to obtain the UML diagrams from code, the second consists of deriving the skeleton of the program from the design information, e.g. tools as Rational Rose, and Poseidon permits of generating the code directly from the UML diagrams.
An implementing mechanism to link up design information and system code could be the meta-data facility present in a lot of programming languages. In general, meta-data describe the implemented code, by storing information regarding classes, methods, and types.
Several modern programming languages provide the programmers with a facility for annotating the code with meta-data. In the case of the Java programming language, for example, this facility allows developers to define custom *annotation types* and to *annotate* fields, methods, classes, and other program elements with *annotations* corresponding to these types. Development and deployment tools can read these annotations and process them producing additional Java programming language source files, XML document, or other artifacts to be used in conjunction with the program containing the annotations.
Our idea consists of using the Java meta-data facility to link up the design information to program elements. In particular, we think to use as design information the UML diagrams.

4 UML as Meta-Data

The UML is *de facto* the standard (graphical) language used during the design process, therefore our project consider its diagrams as a good representation of the system design information.
Our scope is to simplify the evolution/maintenance mechanism. That is, to render the changes required by the evolution immediately available both to the design models and to the implementation, all that we will have as direct consequence the maintenance of the consistency among the design and the code.
In our view, the UML diagrams and the code are seen as different views (design view and implementation view) on a software system, so that consistency between the views is preserved by modeling a coherent refactoring of these views. To realize our project, in particular we need to identify which diagrams are interested by the evolution and also which pieces of software these diagrams describe, in other words, we need a precise mapping between the two views mentioned above. The UML diagrams are, typically, available at design time, to maintain the mapping between the design and implementation view and then the consistency among design models and implementation model during the evolution phase, all this information must be available also at loading and run-time.
Our proposal consists of decorating the system code with the design information. In this way, we obtain a twofold

advantage: to render the design information available at run-time; and, to create a mapping between the design and the implementation view. The decoration will be realized by using Java annotations.
Since UML is a graphical language it is difficult to deal automatically with its diagrams, therefore, we have to convert them into a textual representation to use them as meta-data.
We adopt, as most of the UML tools, the XML Metadata Interchange (XMI [6]) as handling form for the design information. XMI provides a translation of UML diagrams in a text-based form more suitable for run-time manipulation. The XMI standard gives a guideline for translating each UML diagram in XML. Each diagram is assimilated to a graph whose nodes are the diagram's components (e.g., classes, states, actions and so on), and arcs represents the relation among the components. The graph is decorated with XML tag describing the properties of the corresponding UML component.
An example of the translation between UML diagram and XML is showed in the following listing.

```
<UML:Object xmi.id = 'Im169f2c98m10436f02a32mm7cfb'
           name = 'TL2' visibility = 'public' isSpecification = 'false'>
    <UML:Instance.classifier>
       <UML:Class xmi.idref = 'Im13db344bm1041dfafc5emm7ec5'/>
    </UML:Instance.classifier>
    <UML:Instance.linkEnd>
       <UML:LinkEnd xmi.idref = 'Im169f2c98m10436f02a32mm7cdb'/>
    </UML:Instance.linkEnd>
</UML:Object>
<UML:Object xmi.id = 'Im169f2c98m10436f02a32mm7cec'
           name = 'Left St' visibility = 'public' isSpecification = 'false'>
   <UML:Instance.classifier>
      <UML:Class xmi.idref = 'Im13db344bm1041dfafc5emm7c0a'/>
   </UML:Instance.classifier>
   <UML:Instance.linkEnd>
      <UML:LinkEnd xmi.idref = 'Im169f2c98m10436f02a32mm7cdc'/>
   </UML:Instance.linkEnd>
   <UML:Instance.ownedLink>
      <UML:Link xmi.id='Im169f2c98m10436f02a32mm7cdd' name='has' isSpecification='false'>
         <UML:Link.connection>
            <UML:LinkEnd xmi.id = 'Im169f2c98m10436f02a32mm7cdc' isSpecification = 'false'>
               <UML:LinkEnd.instance>
                  <UML:Object xmi.idref = 'Im169f2c98m10436f02a32mm7cec'/>
               </UML:LinkEnd.instance>
            </UML:LinkEnd>
            <UML:LinkEnd xmi.id = 'Im169f2c98m10436f02a32mm7cdb' isSpecification = 'false'>
               <UML:LinkEnd.instance>
                  <UML:Object xmi.idref = 'Im169f2c98m10436f02a32mm7cfb'/>
               </UML:LinkEnd.instance>
            </UML:LinkEnd>
         </UML:Link.connection>
      </UML:Link>
   </UML:Instance.ownedLink>
</UML:Object>
```

The above portion of XMI code translates part of the object diagram showed in Fig. 1a. In particular, it describes the object named `TL2` and `Left St` and their inter-connection. The instances description of a class is grouped into the XMI tag `UML.Object`. The two occurrences showed in the above snippet describe respectively the the object `TL2` and `Left St` in Fig. 1a. The name of the instance is contained in the attribute `name`, whereas the type of the instance is contained in the sub-tag `Class`. The `xmi.idref` refers to description of the corresponding class into the class diagram. The `has` association is described through the tags `UML:Instance.linkEnd` that specify which instances are involved into the association and the tag `UML:Instance.ownedLink` that describes the nature of the association.

5 How to Use Meta-Data for Evolution

To annotate the code with design information we have to extract from each UML diagram its XMI description that represents the perfect reification of the design information at run-time.

The meta-data provided by a single UML diagram are many and, above all, refer to different part of code, e.g., a class diagram describes every class in the system and their relations, and this information encode both the class definitions and the definition of some of their attributes, that have to be annotated. Since the main elements of a sequence diagram are objects and messages, from them it is possible to extract information regarding the instances of a class, and the interactions among them, e.g., creation, invocation of methods, destruction and so on. All this information is inserted into the body of methods as annotations.
The right positioning of the annotations (i.e., from UML design information to Java meta-data) is possible by

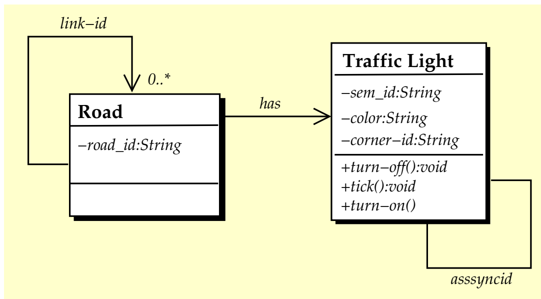


Figure 2: An UTCS class diagram fragment

```
@Retention(RetentionPolicy.RUNTIME)

public @interface MESSAGE{
        String XMI_ID();
        String XMI_name();
        OBJECT Link-start();
        OBJECT Link-end();
}
```

Figure 3: declaration of annotation type MESSAGE

mapping the UML model components to the OMG Interface Description Language (IDL) and achieved by applying the meta-object facility (MOF)-IDL mapping. The existence of this IDL representation of UML means that each UML element, such as associations, classes, actions, operations and so on, has an IDL description. The last step to complete the mapping consists of applying an IDL to Java mapping (e.g., Java _IDL).

To realize the insertion of the XMI code into the right code place, we use Java annotation facility. An annotation is a tag that we insert into the source code. It does not alter the semantics of the code, but instead allows an external application of recognizing and interpreting the tag for its purpose.

The following listing shows the annotation type CLASS declaration, this kind of annotation will decorate the classes of the system, and the values of the attributes of each annotation derives by the corresponding class diagram.

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CLASS)
public @interface CLASS{
        String XMI_ID();
        String XMI_name();
        ATTRIBUTE[] attributes();      //array of annotation type ATTRIBUTE
        ASSOCIATION[] associations(); //array of annotation type ASSOCIATION
        METHOD[] methods();            //array of annotation type METHOD
}
```

Note that the annotation type declaration is itself annotated. Such annotations are called meta-annotations. The first (`@Retention(RetentionPolicy.RUNTIME)`) indicates that annotations with this type are to be retained by the virtual machine so they can be reflectively read at run-time. The second (`@Target(ElementType.CLASS)`) indicates that this annotation type can be used to annotate only class (type) declarations.

The following annotation is derived from the class diagram showed in Fig. 2.

```
@CLASS(XMI_ID="Im13db344bm1041dfafc5emm7c0a",
        XMI_name="Road",
        attributes={@ATTRIBUTE(XMI_ID="Im13db344bm1041dfafc5emm7bf6",
                                 XMI_name="road_id"),
                     @ATTRIBUTE(XMI_ID="Im13db344bm1041dfafc5emm7be4",
                                 XMI_name="road_link")},
        associations={@ASSOCIATION(XMI_ID="Im13db344bm1041dfafc5emm7bba",
                                    XMI_name="has",
                                    multiplicity="Im13db344bm1041dfafc5emm7bbe",
                                    associationEnd="Im13db344bm1041dfafc5emm7ec5")},
        ...
)

public class Road{
        private String road_id;
        private String road_link;
        private Traffic_Light[] hastrafficlights;
...
}
```

The declaration of the annotation type MESSAGE showed Fig. 3 will be used to decorate the pieces of code, statements and so on, which map the message exchanged among objects, the values of the attributes of each annotation derives by the corresponding sequence and collaboration diagrams.

The Java annotation mechanism is not completely adequate for our purposes, because it permits of annotating only the declarations whereas the UML diagrams have a finer granularity. The sequence diagram have information about blocks of statement, and then the linked annotations would to be inserted inside the bodies, the the present mechanism of Java does not allow this.
To overcome this problem, we are extending the Java annotation mechanism and therefore the Java language to support custom annotations on arbitrary code blocks or statements. This new Java dialect, called @Java extends the syntax of the Java language to allow a more general form of annotation. To carry out this job we are benefitting of our experience on [a]C# [3].
Obviously, the mechanism to insert the annotations into the application code is completely transparent to the developer because it is realized as a preprocessor that analyze the design information and annotates on-the-fly the code by byte-code instrumentation.
In this way any kind of evolution could be developed at the design level (i.e. at the design view of the system), simply modifying all the necessary diagrams and dynamically realized by retrieving the related annotations and instrumenting the code according to the planned evolution.

6 Conclusions

This paper presented an approach to use the design information for the dynamic software evolution. This approach is based on some key concepts. The first concept is to maintain a strict correlation between the design information and the application code, in an automatic way. The second is to map all the evolutionary steps both in the design view and in the application code, so that the previous requirement is always satisfied. Into our work, we have used as design information UML diagram, and as programming language Java. The correlation between the two views of the system is realized thanks to the XMI description extracted by the UML diagrams, and thanks to the annotation facility of Java programming language.

References

[1] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.

[2] Walter Cazzola, Antonio Cisternino, and Diego Colombo. [a]C#: C# with a Customizable Code Annotation Mechanism. In *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*, pages 1274–1278, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.

[3] Walter Cazzola, Sonia Pini, and Massimo Ancona. AOP for Software Evolution: A Design Oriented Approach. In *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*, pages 1356–1360, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.

[4] Timothy J. Grose, Gary C. Doney, and Brodsky Stephan A. *Mastering XMI: Java Programming with XMI, XML, and UML*. John Willy & Sons, Inc., April 2002.

[5] Bennet P. Lientz, E. Burton Swanson, and Gail E. Tompkins. Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6):466–471, June 1978.

[6] OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at `http://www.omg.org`, January 2002.

[7] Jim Pierce, Michael D. Smith, and Trevor Mudge. Instrumentation Tools. In Anthony Finkelstein, editor, *Fast Simulation of Computer Architectures*, chapter 4. Kluwer Academic Publishers, Boston, MA, USA, 1995.

[8] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.