Walter Cazzola, Sonia Pini, and Massimo Ancona. The Role of Design Information in Software Evolution. In Proceedings of the 2^{nd} ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05), pages 59–70, Glasgow, Scotland, July 2005.

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 previous 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 consistency among design and code.

1 Introduction

In the last few years, methodologies to automate part of the or the whole software life cycle have been widely studied in 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 present in the software development area, since an intrinsic property of software in *real world* environment is its need to evolve. The laws of software evolution [11] said 'a program that is used in a real-world environment must change, or became progressively less useful in that environment'.

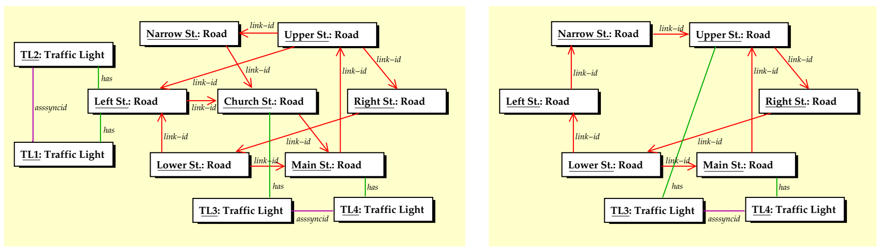
When a program evolves, it becomes more complex and automatic techniques to support these phenomena are fundamental to improve the management of unanticipated software evolution and the software efficiency.

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. At the beginning design view and implementation view are consistent since one is derived and developed form the other, and we must preserve the correlation between the two view, and this correlation must exist for all the software life-cycle. But often, during the evolution and maintenance phase a discrepancy between the two view can occur, because the initial stages of development (the system specifications and the design) are ignored once the code has been developed. The main problem is the fact that models are shown like only an intermediate step in the software development life-cycle. 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.

The term *gap* has been stated by Rumbaugh [16], 'too often, there is still a gap between concept and execution' as a problem area in his retrospective review of O-O methodology. In the past developers have tried different ways to link design to implementation, and to tackle this gap problem. One of the noted effort in this endeavor is by looking at the gap from program viewpoint, describing the mapping from O-O programming languages properties to their corresponding UML concepts.

To most people software is the code that is the and result of the software development process. When a company starts developing a new product, it typically uses a clean forward engineering scheme and goes (iteratively or not) through requirements analysis, high-level design, design and implementation phases. This development process changes when a first implementation is finished. From then on, the implementation receives more and more attention. This restricted view of software is one of the main causes of the many problems associated with software development and its evolution. 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.

This problem, where implementation and design evolve in different directions because they are no explicitly related, is also know as *drift/erosion*. It is well-known that uncontrolled change to software can lead to increasing evolution cost caused by deteriorating structure and compromised system quality [10]. For complex systems, the need to carefully manage system evolution is a critical task. Wide integrated software system must continuously change to satisfy the evolving requirements. Adapting such software can be very difficult, because of the software size and complexity and variety of users with conflicting requirements. When there is a gap between the views it become difficult or impossible to know all the necessary changes to apply, therefore the evolution



(a) Standard layout (b) Modified layout

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

cannot be planned *in-the-large*. To simplify the evolution process, 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 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 consistency among design and code views.

2 System Evolution through Design Information

With the help of an example, we describe how the design information, in our case UML [2] 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. Unanticipated software evolution is not something for which one can prepare during the design of a software system. Examples of unanticipated and hard to plan problems may be: road maintenance, traffic lights disruptions, car crashes, traffic jam and so on.

In the design phase, software engineers produce a lot of diagrams to identify domain concepts, to describe the business processes, and the physical structure of the environment, and so on.

The design information we consider can be divided in two categories: system structure and behavior.

- Structural Design Information is an explicit description of the structure of the system.
- Behavioral Design Information describe the computations and the communications carried out by the application objects, e.g. we consider object behavior, collaboration between objects, state of the object, and so on.

Structure and behavior of the system are modeled by class/object diagrams, sequence or collaboration diagrams and state diagrams.

The UTCS for our 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.

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 to maintain the consistency between the views of the system.

The evolved system, as well as the original system, can be or better should be modeled by using, for example, UML diagrams. If this is the context of the evolution, then the original diagrams can determine the code to be adapted, the evolved diagrams specify how the code has to be adapted, and moreover the difference between such diagrams before and after the evolution represent the evolution itself.

Applying evolution at high level of abstraction, like at UML level has the advantage that software designers don't have to worry about the syntax of all possible programming language, and have a global view of the system. And as Blaha et al. [1] had said "Models allow a developer to focus on the essential aspects of an application and defer details".

The Fig. 1b show how the object model changes after this event. By comparing the diagrams in Fig. 1 it is possible to understand how the UCTS is configured after the evolution.

When the models are evolved it is necessary to map this evolution into the implementation code. To realize this issues, in our opinion, we must have got the design information available not only at design time, but also at compile and run time, to obtain this result our idea is to maintain this design information inside the system and to link every component of the UML diagrams to the respective portion of code using meta-data facility.

In this way, 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.

With the increased interest in evolution and maintenance, UML vendors seek ways to support software developers in applying maintenance. The problem with such tools is that the UML meta-model is inadequate to maintain the consistency between the model and the code while one of them gets evolved [17].

In the past developers have tried different ways to link design to implementation, and to tackle this problem, to date, no existing, general purpose methodology for this issue. We propose an infrastructure to dynamic adapt software system, called RAMSES [4]. Our project involves a reflective middleware whose aim consists of consistently evolving software systems, both design information and implementation code, against run-time changes. This middleware provides the ability to change the system at run-time without stopping it, by using its design information. Many important applications must run continuously and without interruption. This is especially true of mission critical applications, such as telephone traffic control system, and air traffic control system. Our goal is to show that dynamic software evolution can be achieved in a practical manner that is flexible, efficient, robust, and easy to use.

RAMSES (Reflective and Adaptive Middleware for Software Evolution of Systems) performs two phases to carry out our dynamic self-adaptation. In the first phase, the RAMSES's meta-level extracts the design information as XMI (XML Meta-data Interchange [13]) schema from the base application and it reifies them in the meta-level to constitute the meta-data. Whereas, in the second phase, RAMSES's meta-level plans the dynamic adaptation of the base-level system, get the run-time events, evolves the meta-data against the detected event, checks the consistency, and finally reflects the modified data to the base-level.

In this paper, we treat only the role of the design information in software evolution.

3 Defining Meta-Data

In our opinion, a simple way for maintaining consistency is that design information are linked with the source code. To synchronize design and implementation, our proposal consists of using a mechanism that permits to express design as a set of meta-data over the implementation. To back our idea we have this definition: *Software design is an abstraction of implementation*. This definition is consistent with descriptions of design that can be found in software engineering literature [7], and moreover this definition said that design is explicitly related to the implementation. To express 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 express design information over the implementation. In particular, we think to use as design information the UML diagrams, or more just to use the textual representation of 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 considers 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 affected 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 Meta-data Interchange (XMI [13]) 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 interconnection. 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 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 maintain consistency between design and implementation

Our goal, consists of transform system models and code to implement required modifications, and propagate the transformation effect across the views, this can be faced using model-driven approach [9], i.e. to tackle the problem of evolving complex software systems by raising the level of abstraction from source code to models, and then maintain the consistency between model and code.

Since software designers think about evolution at the design level, and since design information provide an immediate and understandable global view of the system, it is quite natural to exploit the UML and its unified meta-model for expressing evolution.

Model-driven engineering is a software engineering approach that promotes the usage of *models* and *transformations* as primary artifacts. In our context, we can said design information is a model and implementation code is a model; and following this reasoning evolution and consistency between design and code is simply a model transformation.

We could use, in first phase of the evolution, *horizontal transformation* [6] on design models to describe the evolution of the system, and *vertical transformation* [6] in last phase of evolution to propagate the evolution at the implementation in order to maintain consistency among the models. A crucial aspect of model transformation is *model consistency*, since UML model is typically composed of many different diagrams, the consistency between all these diagrams need to maintained when any of them evolves.

Graph transformation seems to be a suitable technology and associated formalism to specify and apply model transformations for the following reasons: first graphs are a natural representation of models that are intrinsically graph-based; last graphs transformation theory provides a formal foundation for the automatic application of model transformations. Table 1 show a direct correspondence between software evolution and graph transformation.

Evolution	**Graph transformation**
Software Artifact	column Graph
Evolution	Graph Production
Composite Evolution	Composition of Graph Production
Evolution Application	Graph Transformation

Table 1. Correspondence between evolution and graph transformation.

6 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 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.

We go to explain how to use the annotations for our scope. Java allows the programmer to define and use user-defined annotation types. The facility consists of a syntax for declaring annotation types, a syntax for annotating declarations, APIs for reading annotations, and a class file representation for annotations. To create an annotation we need to define an annotation type first. Annotation types are defined like interfaces with an '@' (at) sign before the interface definition, and annotations are specified in the program source by using the '@' sign, followed by the annotation name.

The following listing shows the Java 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.

```java
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.

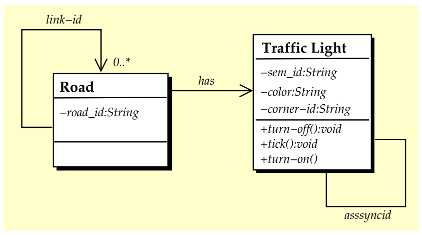


Fig. 2. An UTCS class diagram fragment

```
@Retention(RetentionPolicy.RUNTIME)

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

Fig. 3. declaration of annotation type MESSAGE

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 annotating only the declarations whereas the UML diagrams have a finer granularity. The sequence diagrams 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 benefiting of our experience on [a]C# [5].

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 analyzes 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.

7 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. Michael R. Blaha and William J. Premerlani. A Catalog of Object Model Transformations. In *Proceedings of 3rd Working Conference on Reverse Engineering (WCRE'96)*, pages 87–96, Montrey, CA, USA, November 1996. IEEE Computer Society Press.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.
3. 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.
4. Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. RAMSES: a Reflective Middleware for Software Evolution. In *Proceedings of the 1st ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04)*, in 18th European Conference on Object-Oriented Programming (ECOOP'04), pages 21–26, Oslo, Norway, on 15th June 2004.
5. 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.

6. Robert France and James M. Bieman. Multi-View Software Evolution: a UML-Based Framework for Evolving Object-Oriented Software. In Gerardo Canfora and Anneliese Amschler Andrews, editors, *Proceedings of 17th IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 386–397, Florence, Italy, November 2001. IEEE Computer Society.
7. Adele Goldberg and Kenneth S. Rubin. *Succeeding With Objects: Decision Frameworks for Project Management*. Addison-Wesley, 1995.
8. 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.
9. Stuart Kent. Model Driven Engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, LNCS 2335, pages 286–298, Turku, Finland, May 2002. Springer.
10. Krzysztof Kowalczykiewicz and Dawid Weiss. Traceability: Taming Uncontrolled Change in Software Development. In *Proceedings of the IV KKIO Conference*, Tarnowo Podgórne, Poland, 2002.
11. Meir M. Lehman. Laws of Software Evolution Revisited. In Carlo Montangero, editor, *Proceedings of the 5th European Workshop on Software Process Technology (EWSPT'96)*, LNCS 1149, pages 108–124, Nancy, France, October 1996. Springer.
12. 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.
13. OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at `http://www.omg.org`, January 2002.
14. 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.
15. James Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
16. James E. Rumbaugh. Modeling through the years. *Journal of Object-Oriented Programming*, 10(4):16–19, July-August 1997.
17. Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Enabling and Using UML for Model Driven Refactoring. In Serge Demeyer, Stéphane Ducasse, and Kim Mens, editors, *Proceedings of the 4th International Workshop on OO Reengineering*, pages 37–40, Darmastadt, Germany, July 2003. Universiteit Antwerpen.