Modeling Heuristic Rules of Methods

Bedir Tekinerdoğan & Mehmet Akşit
TRESE project, Department of Computer Science,
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.
email: {bedir | aksit}@cs.utwente.nl, www server: http://wwwtrese.cs.utwente.nl

Abstract

In method-based software development, software engineers create artifacts based on the heuristic rules of the adopted method. Most CASE tools, however, do not actively assist software engineers in applying the heuristic rules. To provide an active support, the rules must be formalized, implemented and integrated within the framework of current CASE tools. In this paper we describe an approach for formalizing the heuristic rules of methods. We will illustrate the applicability of this approach using the OMT method and our experimental CASE environment.

1. Introduction

It is generally agreed that the success of a software development method largely depends on the availability of tools supporting it. Typically, a method is defined in terms of its artifacts, process and a set of heuristic rules. A considerable number of CASE environments have been developed during the past years. Most CASE environments provide tools for definition of artifacts and processes. Very few of them, however, provide support for the application of rules of methods. The lack to capture and support the heuristic rules of methods reduces the effectiveness of current CASE environments.

In this paper we provide a generic approach for formalizing and implementing heuristic rules of methods. We defined a knowledge acquisition technique to extract the rules of a method. In this approach, rules are expressed using production-rules. We developed an agent-based reasoning system to integrate the rules with the method artifacts. We integrated our tool into a commercially available CASE environment. This paper explains our approach and summarizes our experiences in implementing and utilizing it.

The paper is organized as follows: In section 2, we give an overview of the structure of current CASE tools. Section 3 describes the requirements for our generic rule modeling approach and the integration with CASE tools. The heuristic rule modeling is described in section 4 in which we show the rule elicitation, the rule analysis and the rule representation process. For illustration, we apply our approach to the analysis phase of the object-oriented software development method OMT. Section 5 shortly gives a description of our experimental CASE tool which supports the heuristic rules and which has been integrated with the Rational Rose [Rational 97] environment. Section 6 presents our evaluation
and the lessons learned. Finally, the last chapter gives conclusions. Appendix A provides a list of the formalized rules of the object model of the analysis phase of OMT.

## 2. Meta-Models for Object-Oriented Methods

During the last decade, a considerable number of methods have been introduced. Although most popular methods have a general character [Rumbaugh 91][Yourdon 89], some methods are targeted to specific application domains, such as real-time system design [Ellis 94]. Some methods are specifically defined for a given phase in software life cycle, such as requirement analysis [Wieringa 96]. Since it is difficult to define an ideal method for all application domains and all phases, most CASE environments support several methods.

Obviously, to support a method, it is necessary to model and implement the signification features of that method. To be able to support multiple methods, modern CASE environments adopt meta-models which can be tailored by method designers. A typical example is the meta model of the Unified Modeling Language (UML) [UML 97].

In general, a software development method can be defined in terms of three aspects [Budgen 93]:

**Representation Models:**

Representation models are descriptive forms that the software engineer can utilize for building models. Representation models reflect the properties of the artifacts in the system. For example, the method OMT introduces representation models for classes, associations, attributes, inheritance relations and state-charts.

In general, representation models can be associated with problem models and solution models, rather independently of any method. For example, most object-oriented methods adopt classes, attributes and operations. This means that the adoption of any method may not necessarily preclude a set of representation models. For instance, the representation models as defined by UML can be adopted by various different methods such as OMT [Rumbaugh 91] and Fusion [Coleman 94]. Recently, a number of researchers have been investigating formal semantics for representation models [Breu 97]. Although there are still a number of open issues to be studied, such as component formalization and composition [Nierstrasz 95], the representation models are possibly the best defined part of the current CASE environments.

**Software process:**

A software process is a description of the order of the phases of a method. For example, the method OMT advises the following process for defining the object models: requirement specification, noun extraction, class, association, attribute, and inheritance identification, and refinement.

Recently, there has been a considerable amount of interest in formalizing software processes. Objectives of process modeling can be summarized as process improvement and management, automatic guidance in performing process and automatic execution support [Berg 97]. Among many experimental systems, Merlin [Unkermann 94], MARVEL [Heineman 92] and OZ [Ben-Shaul 98] are
based on a rule-based expert system. The main difference is that these systems aim at supporting software development processes, whereas, our emphasis is targeted to supporting software engineer in applying the heuristic rules of methods.

**Method rules:**

Method rules aim at identifying, eliminating and verifying the artifacts. Most methods define rules in an informal manner. Nevertheless, method rules can be expressed using conditional statements in the form \textbf{IF} \langle\text{condition}\rangle \textbf{THEN} \langle\text{consequent}\rangle. The consequent part may be an action. For example, the method OMT advises the following rule to identify classes:

\begin{verbatim}
IF a tentative class <is relevant>
THEN the select the tentative class as a <class>
\end{verbatim}

Compared to the research activities on representation models and software processes, there has been very little effort in formalizing design heuristics. In the literature [Rumbaugh 91][Johnson 88][Riel 96] object-oriented design heuristics are generally explained in textual form.

In the area of expert system design, representation of heuristic rules have been extensively studied [Gonzales 93][Jackson 90]. For example, in the area of mechanical and electronic engineering, a number of expert systems have been developed to represent various kinds of expert’s knowledge [Lee 97][Maher 90]. In general, these systems are integrated into a CAD environment. To the best of our knowledge, there has not been any commercial CASE environment, which provides an integrated expert system support to guide software engineers actively in applying heuristic rules. Therefore, we think that it is worthwhile to investigate the possibility of using an expert system support for CASE environments.

3. **Our Approach**

3.1 **Requirements**

While designing the tool architecture, the following four requirements have been taken into account:

1. \textit{Integration with the popular CASE environments}: It is not our intention to develop a CASE environment from scratch but enhance a commercially available environment with heuristic rule support. However, this support must be generic enough to integrate it into different CASE environments.

2. \textit{Adopting the heuristic rules of a popular method}: Since our aim is not to develop a new method, the heuristics of a well-established method must be modeled. Nevertheless, the heuristic (knowledge) acquisition technique must be generic enough to apply to different methods.

3. \textit{Preserving the context of reasoning for each artifact}: During various phases of software development different kinds of artifacts are defined. It is important to collect and preserve the context and result of heuristic reasoning within the corresponding artifacts since, in the future, this information can be invaluable for maintaining the system.
4. **Active and specific support of design heuristics:** The support of software engineers in applying design heuristics must be actively guided and supported. It may be otherwise very difficult for the software engineer to select and apply the relevant rules to the corresponding artifacts.

3.2 **The Tool Architecture**

Figure 1 shows the architecture of our prototype. We have selected Rational Rose™ [Rational 97] as the CASE environment because of its popularity in the market and its availability in our laboratory. In Figure 1 the Rational Rose™ environment is represented as three vertical layers. To integrate our system into the Rational Rose™ environment, we had three options: to extend the artifacts of Rational Rose™ using meta-modeling techniques, programming using the scripting language of Rational Rose™ and/or to develop a separate application and link to Rational Rose™ using the OLE™ technology. Our version of Rational Rose™ did not provide full meta-modeling possibility. Since we needed to design a dedicated expert system, we decided to develop a separate application and link it to the Rational Rose™ environment using the OLE™ technology. The most convenient way was to link our artifacts to the repository of Rational Rose™. This is shown by the arrow.

![Figure 1. The tool architecture as implemented in the current prototype.](image)

In general, architecture of expert systems can be classified into two broad categories: traditional expert systems with a global interpreter and a knowledge base [Jackson 90], and distributed expert systems based on the agent technology [Agents 94]. Traditional expert systems have the advantage that they are easier to construct since there is only one interpreter active at a time. Secondly, interpretation is separated from the knowledge base thereby it is possible to utilize different interpreters for the same knowledge base. These two advantages are also the limitations of traditional expert systems. Namely, in most real-world systems interpretation is not global but distributed to multiple knowledge sources. In agent-based systems, knowledge bases are managed by multiple knowledge sources which cooperate together to achieve common goals.

To fulfil the third requirement, preserving the context of reasoning for each artifact, we decided to design an agent-based expert system. Each agent encapsulates the context of its reasoning. In addition, each agent has the capability to implement the most suitable interpretation for itself. This last property is particularly useful to fulfil the fourth requirement: active and specific support of design heuristics. Agent technology, however, requires a concurrent environment and well-defined protocols for cooperation. As shown by Figure 1, the bottom-layer of our tool provides a virtual environment of
concurrent agents. The second layer from bottom represents the agent-based. The layer artifact management and control provides tools for software engineers to control and manage the artifacts. The top-layer represents the user-interface of our heuristic rule support system.

The basic structure of each artifact agent is shown by Figure 2. Each specific artifact inherits from class Artifact thereby fulfilling the generic requirements for each artifact type. Every artifact has the following responsibilities: retrieving the necessary information from the environment, producing the artifact, controlling the properties and when the production is ready delivering the necessary information to the environment. These are represented by four rule components. Further each artifact has a set of properties and a representation of it.

Assume now that the software engineer wants to identify a tentative class. This results in the creation of an instance of class TentativeClass, which inherits from class Artifact. The initialization operation starts the interpretation process of the retrieval rules of this artifact. The retrieval rules search for an instance of class NounDelivery, because in OMT tentative classes are derived from nouns. Instances of class NounDelivery interact with the software engineer to detect the nouns in the requirement specification. If there is no instance of class NounDelivery in the system, it is created, and the similar process starts again. Now assume that the retrieval rules of a tentative class find a suitable instance of NounDelivery. Then the production rules are activated. These rules ask the software engineer whether the noun is relevant or not (see the appendix for the rule definition). If the noun is selected as relevant, then the delivery rules are triggered. This rule communicates with the environment about the creation of a new tentative class.

The design of this agent-based artifact system is a result of several years of research activities [Algra 94][Dongen 97][Evers 97][Hoogendoorn 97]. We consider its detailed description out of the scope of this paper.

Assume now that the software engineer wants to identify a tentative class. This results in the creation of an instance of class TentativeClass, which inherits from class Artifact. The initialization operation starts the interpretation process of the retrieval rules of this artifact. The retrieval rules search for an instance of class NounDelivery, because in OMT tentative classes are derived from nouns. Instances of class NounDelivery interact with the software engineer to detect the nouns in the requirement specification. If there is no instance of class NounDelivery in the system, it is created, and the similar process starts again. Now assume that the retrieval rules of a tentative class find a suitable instance of NounDelivery. Then the production rules are activated. These rules ask the software engineer whether the noun is relevant or not (see the appendix for the rule definition). If the noun is selected as relevant, then the delivery rules are triggered. This rule communicates with the environment about the creation of a new tentative class.

Assume now that the software engineer wants to identify a tentative class. This results in the creation of an instance of class TentativeClass, which inherits from class Artifact. The initialization operation starts the interpretation process of the retrieval rules of this artifact. The retrieval rules search for an instance of class NounDelivery, because in OMT tentative classes are derived from nouns. Instances of class NounDelivery interact with the software engineer to detect the nouns in the requirement specification. If there is no instance of class NounDelivery in the system, it is created, and the similar process starts again. Now assume that the retrieval rules of a tentative class find a suitable instance of NounDelivery. Then the production rules are activated. These rules ask the software engineer whether the noun is relevant or not (see the appendix for the rule definition). If the noun is selected as relevant, then the delivery rules are triggered. This rule communicates with the environment about the creation of a new tentative class.

The design of this agent-based artifact system is a result of several years of research activities [Algra 94][Dongen 97][Evers 97][Hoogendoorn 97]. We consider its detailed description out of the scope of this paper.

**Figure 2.** The agent-based artifact model.

4. **Heuristic Rule Modeling**

Our attempt to formalize software development methods is a special case of knowledge acquisition as described in traditional knowledge engineering techniques [Gonzales 93]. The domain for knowledge engineering hereby is the software development method for which we would like to define CASE support. We acquire our knowledge mainly by studying the available documentation on the methods.
and by interviewing experts of the corresponding methods [Hart 92]. Figure 3 represents the overview for this heuristic method rule acquisition process.

![Figure 3. Heuristic Method Rule acquisition](image)

Each method can be considered as having a *model description* and *method description*. In the model description the representation models are described. The method description part describes the process and the informal heuristic rules for instantiating these representation models. In OMT the model description part describes for example the representation model *class*. In the method description part heuristic rules for identification of classes are described. The method description may use additional representation models, which are not described in the model description. These may be graphical diagrams or temporary representation models like the tentative class in the OMT class identification process.

The method formalization process includes basically four steps. First the model artifacts are extracted from the model description. Second the informal heuristic rules from the method description are identified and formalized. Third, from the heuristic rules method artifacts are derived. Fourth, the formalized rules are assigned to the model and method artifacts.

We use the Object Modeling Technique (OMT) from [Rumbaugh 91] to illustrate our ideas. The modeling part of OMT consists of three kinds of models. The *object model* describes the static structure of a system in terms of objects and relationships corresponding to real-world entities. The *dynamic model* describes the control structure of a system in terms of events and states. The *functional model* describes the computational structure of a system in terms of values and functions.

The process of the OMT method consists of the following phases: *analysis, system design, object design* and *implementation*. The output of the analysis phase is a model that captures objects and their relationships, the dynamic flow of control and the functional transformation of data subject to constraints. The overall architecture of the system is determined during the system design phase. During the object design, the analysis models are elaborated, refined and then optimized to produce a practical design. The implementation phase realizes the design in terms of programming language constructs.

In the following we will extract heuristic rules from the object model description of OMT’s analysis phase. Section 4.1 describes the elicitation of model artifacts. Section 4.2 describes the acquisition of
the informal heuristic method rules and the formalization of these rules. In section 4.3 method artifacts are extracted from the formalized rules. Finally, section 4.4 handles the integration of the identified artifacts with the formalized rules.

### 4.1 Identifying the model artifacts

Identifying the model artifacts is done by studying the model description part of the method. In general, the models of a method are well defined and as such can be straightforwardly derived to a certain extent. However, usually each phase of a method uses different representation models. It may not always be clear from the model description in which phase each representation model will be used. Therefore, it is worthwhile to read the corresponding method description as well.

**OMT**

Remind that OMT provides models for the object model, dynamic model and functional model phases. Our purpose here is the formalization of the object model in the analysis phase. We extracted the following model artifacts for the object model of the analysis phase of OMT:


### 4.2 Model rules

The rule modeling process is explained in the following sub-sections. First, in section 4.2.1 the informal heuristic rules of the method are elicited. Second, section 4.2.2 describes how to analyze and abstract the rules. Third, in section 4.2.3 the abstracted informal rules are formalized.

#### 4.2.1 Extract Rules

For extracting heuristic rules we need to study the method description part of the given method and if needed consult the method experts. Rules can be found by looking for imperative statements, like do, select, group, eliminate etc. Some rules may not be explicit in the method description, but implicitly assumed. Further, rules may refer to model artifacts as well as method artifacts. We need to consult with method experts to verify the identified rules and to identify the missing rules.

**OMT**

OMT identifies classes by first extracting nouns from the problem statement, which provides the tentative classes, and then selecting the right classes according to some heuristic design rules. After reading the corresponding description for this tentative class identification and the class identification we can extract the following informal heuristic rules and steps:

**Tentative Class identification**

1. Extract the nouns from problem statement.
2. Write down classes from the application domain.
3. Select classes from your general knowledge.
4. Write down every class that comes to mind.
**Class identification**

5. If two classes are redundant then keep the most descriptive one.
6. If a class has nothing to do with the problem, it should be eliminated.
7. If a tentative class is vague then eliminate it.
8. If a tentative class describes an operation that is applied to objects and not manipulated in its own right, then it is not a class. An operation that has features of its own should be modeled as a class.
9. The name of a class should reflect its intrinsic nature and not a role that it plays in an association.
10. Constructs extraneous to the real world should be eliminated from the analysis model.

**4.2.2 Analyze and abstract rules**

The goal of this step is to eliminate the unnecessary rules and keep only the right rules. If two rules express the same information, then we eliminate the least descriptive one. Further we rewrite unclear rules. Some rules may refer to application domain. Other rules may just refer to the general and intuitive knowledge of the software engineer. In that case we consider the context whether to include the informal heuristic rule or not.

**OMT**

For comprehensibility we restate some rules, like rule 1, which is restated in the following list. Other rules are redundant. For example in the above list of rules rule 3 and rule 4 refer to the software engineer’s knowledge and therefore we only keep rule 3. In the informal rules the term class is used, although only tentative classes are meant. We therefore, replace the term class with the term tentative class. Further, we try to write every rule in the "If <condition> Then <consequent>" format so that later the formalization process can be easier. Finally we add an implicit rule which is needed to keep the remaining tentative classes as the right classes. As a result we have the following steps and informal rules for the tentative class identification and class identification:

**Tentative Class identification**

1. Select the nouns in problem statement as tentative classes.
2. Select tentative classes from the application domain.
3. Select tentative classes from your general knowledge.

**Class identification**

4. If a tentative class is redundant then eliminate it.
5. If a tentative class is irrelevant then eliminate it.
6. If a tentative class is vague then eliminate it.
7. If a tentative class is an attribute then eliminate it.
8. If a tentative class is an operation then eliminate it.
9. If a tentative class is a role then eliminate it.
10. If a tentative class is an implementation construct then eliminate it.

**4.2.3 Formalize rules**

The previous informal rules will be formalized using the following format:

```
IF <CONDITION> THEN <CONSEQUENT>
```

The condition part represents a for the action in the consequent part. The condition and the consequent part include artifacts, which are inspected and manipulated respectively. Note that some rules may only consist of an action.
In general, rules may apply to properties of artifacts. As the name suggests itself properties describe the characteristics of the artifact. Enclose part of the rules which describe a property of an artifact between characters '<' and '>'.

**OMT**

The previous informal rules for tentative class identification and the class identification process can be formalized as follows:

**Tentative Class identification**

1. *Extract* nouns from the problem statement.
2. *Extract* tentative classes from application domain knowledge.
3. *Extract* tentative classes from general knowledge.

**Class identification**

4. IF tentative class <isRedundant> 
   THEN eliminate tentative class.
5. IF tentative class <isIrrelevant> 
   THEN eliminate tentative class.
6. IF tentative class <isVague> 
   THEN eliminate tentative class.
7. IF tentative class <isAttribute> 
   THEN eliminate tentative class.
8. IF tentative class <isOperation> 
   THEN eliminate tentative class.
9. IF tentative class <isRole> 
   THEN eliminate tentative class.
10. IF tentative class <isImplementationConstruct> 
    THEN eliminate tentative class.
11. Select remaining tentative classes as right classes.

The other rules can be derived in a similar way. Appendix A gives a complete list of the formalized heuristic rules for producing the object model in the analysis phase of the OMT method.

**4.3 Discovering method artifacts**

We can identify method artifacts from the formalized rules by analyzing the condition and/or the consequent part of the rules. Artifacts are the entities in the rules, which are manipulated, that is, selected, eliminated or changed.

**OMT**

For example from the previous rule 1 we can identify the artifact NOUN. This artifact has not been identified as a model artifact before but is only described in the method description of OMT. Further it is considered essential for identifying tentative classes. Another example is the artifact TENTATIVE CLASS. The model only describes the artifact CLASS. The artifact TENTATIVE CLASS is only needed for supporting the class identification process at the method level. The list of all the identified method artifacts are as follows:
Note that we also included the artifacts SOFTWARE ENGINEER and APPLICATION DOMAIN. The artifact SOFTWARE ENGINEER represents the software engineer’s intuition and the artifact APPLICATION DOMAIN represents the application domain, which is referred to by many rules.

4.4 Assign design rules to artifacts

Each artifact model defines its own production rules. After we have identified, abstracted and represented different rules, we need to assign these rules to the corresponding artifact types. We have assumed that each rule can be (re)written in the format: IF <condition> THEN <consequent>. The condition part includes retrieval artifacts. The consequent part includes the artifact, which is being processed, that is, produced, created, or refined. We assign each rule to the artifacts appearing in the consequent part of the rule.

OMT

Consider for example the following rule:

1. IF tentative class <isImplementationConstruct>  
   THEN eliminate tentative class.
2. Select remaining tentative classes as right classes.

Rule 1 is assigned to the artifact TENTATIVE CLASS rule 2 is assigned to artifact CLASS.

4.5 Define Artifact specification

4.5.1 Define retrieval artifacts

The production of each artifact requires another artifact. These artifacts which are necessary for producing a new artifact and thus need to be retrieved in time are called sub-artifacts. These sub-artifacts can be extracted from the rules, which have been assigned to the artifact before. Retrieval artifacts are found by looking at the artifacts in the condition part of the assigned rules.

OMT

In the previous rules we can infer that for producing TENTATIVE CLASS artifact NOUN is needed. NOUN artifacts are extracted from the artifact PROBLEM STATEMENT. It is clear that each artifact may serve as a sub-artifact. We can represent these dependencies in an artifact graph as it is shown in Figure 4. In the artifact graph the nodes represent the artifacts and the arrows represent the dependencies between artifacts. The arrows are directed from the sub-artifact to the artifact, which needs to be produced. For example artifact PROBLEM STATEMENT is a sub-artifact of artifact NOUN. The applied rule for producing artifact NOUN is the rule with number 1. Note that in Figure 4 the software engineer is abbreviated to SE and the application domain to AD. As we can see in Figure 4 many rules refer to these artifacts.
### 4.5.2 Identify properties

Properties can be identified from the corresponding rules for the artifact by inspecting the condition and the consequent part of each rule. Properties characterize an artifact or describe some quality factor of the artifact. The properties of an artifact are shown between the characters ` '<` and ` '>`. 

**OMT**

Consider for example the following rule:

1. IF tentative class `<ImplementationConstruct>` THEN eliminate tentative class.

From this rule we can extract the property `isImplementationConstruct` for the artifact `TENTATIVE CLASS`.

### 4.5.3 Detecting the flaws of a Method

In our attempt to formalize the heuristic rules of the analysis phase of the OMT method we encountered the following problems:

1. **Vague, ambigue and imprecise rules**: Some rules are vague, ambiguous and imprecise and are not directly amenable for formalization. An example is the rule "Extract tentative classes from general knowledge". Our experience is that OMT relies very often on the general knowledge and intuition of the software engineer. In order to solve this problem we introduced the artifacts `APPLICATION DOMAIN` and the artifact `SOFTWARE ENGINEER`. The rules referring to the intuition of the software
engineer are put in the artifact SOFTWARE ENGINEER. The rules referring to the application domain are put in the artifact APPLICATION DOMAIN. The rules for these artifacts rely directly on the response of the software engineer and do not provide additional automated support.

2. Unconnected artifacts: Some artifacts appeared to be unconnected with the rest of the artifacts. In the formalization of the analysis phase, for example, we could not identify rules for providing the connection of the artifact DATA DICTIONARY and the artifact MODULE with the other artifacts. The connection of the artifacts is necessary to provide the traceability among the artifacts and as such ease the maintainability.

3. Cyclic paths: Artifacts may be connected through cyclic paths, which means that the applied rules may infinitely circulate through the corresponding artifacts. Although this problem cannot be identified in the artifact graph of Figure 4 we can look for cyclic paths to detect this kind of flaws of methods. The solution for this flaw is simply to break the cycle, by eliminating or restating the applied rules.

5. Tool support

Our experimental CASE tool for supporting the formalization of heuristic design rules uses tools to define artifact types, to instantiate artifact types and to execute formal heuristic rules of artifacts. In Figure 5 the execution of a heuristic rule for class identification is shown. The upper tool, which is the heuristic design rule tool, shows the software engineer the list of identified tentative classes and asks whether the corresponding tentative classes are irrelevant or not. Hereby an example of an ATM system is given. The software engineer can select yes or no. In case of a positive reply (yes) then the action of the rule will eliminate the tentative class from the tentative class list. If the software engineer selects no, then the tentative class will be kept in the repository. After answering this question for all the tentative classes the software engineer can commit the decisions made by pressing the Okay button. This will trigger the rule for selecting all the tentative classes as right classes. The software engineer may also cancel the design decisions by pressing the Cancel button. This will restore the original list of tentative classes. The connection with Rational Rose tools is made by pressing the Representation button. The Rational Rose tool is shown in the lower tool of Figure 5. This will open Rational Rose tool and show the selected classes in the UML representation.
6. Evaluation and lessons learned

To evaluate our system, we have been carrying out a number of experiments. We think that to increase the effectiveness of current CASE tools, heuristic rule support is necessary. However, we have also experienced a number of difficulties. We will now evaluate our prototype with respect to the requirements presented in section 3.1.

1. Integration with the Rational Rose™ was largely successful. The OLE™ technology allows client-server communication between our system and Rational Rose™. Although we did not try our implementation with another CASE environment, we avoided specific protocols as much as possible. However, we had a difficulty to implement an efficient triggering mechanism into Rational Rose™ to automatically update its repository when an external model is changed. The second problem in our current prototype is that the user interfaces of our tool and the Rational Rose™ environment are completely separated. Although our user interface conforms to the interface conventions, it would be preferable to integrate the user interfaces. We think that user interface integration is possible by using the scripting language of Rational Rose™.
2. Our knowledge acquisition process was largely successful in modeling the heuristic rules of the OMT method. We had a reasonably accurate representation of OMT. The software engineers could conveniently interact with the system. There are, of course, a number of open issues related to representation of heuristic rules. The problems that we experienced in this context can be grouped into two categories: problems related to the OMT method, and problems related to the adopted knowledge representation and reasoning techniques. In section 4.5.1, the problems related to the OMT method was summarized as imprecise rules, cyclic dependencies and unconnected artifacts. This experimentation shows that formalizing the heuristic rules of a method can be useful in identifying the flaws of a method. Related to the adopted knowledge representation technique, we experienced the following problems:

- Although it closely represents the OMT rules, adopting two-valued logic based reasoning has a number of problems. Firstly, two-valued logic based rules, such as defined in OMT, force the software engineers to take abrupt decisions such as selecting an entity as a class or not. In practice, however, especially in the early phases of software development, it is very difficult to make hard commitments. Some entities for instance, can be partially considered as classes, attributes or both. The software engineers may prefer to defer their decisions and commit only when sufficient information is available. Deferring design decisions is not supported by current methods.

- Secondly, two-valued logic based reasoning does not match the intuition of software engineers very well. Software engineers may like to use expressions like this class is fairly relevant. In OMT, an entity is either a class or not a class, but cannot be a fairly class.

In fuzzy-logic, software engineers may use expressions like a class is fairly relevant. Fuzzy-logic allows specifying overlapping alternatives and deferring the abrupt choices. Currently, we are investigating fuzzy-logic based rules in representing the heuristic rules of methods.

3. The agent-based architecture has been proved very suitable for preserving the context of reasoning for each artifact. Each agent can easily encapsulate its context. A difficulty that we experience with the agent-based approach is to trace the dependencies among agents. We are currently building dedicated tools for this purpose.

4. Each agent provides an active and specific support for design heuristics. In this approach, however, we encountered the problem of managing the user-interface. If each agent communicates with the user independently, then the user interface becomes too complex. We will therefore adopt specific user interface agents to coordinate the interactions with the software engineer.

7. Conclusions

Although many CASE tools provide support for processes and representation models of methods, they lack support for the heuristic rules of methods. In this paper we defined a knowledge modeling method for formalizing the heuristic rules of methods. The approach is aimed to be used for a wide range of well-established methods and for integration with existing CASE tools. For illustration purposes we
applied our approach to the analysis phase of OMT and formalized the heuristic rules of it. We have built an experimental CASE tool, which we have integrated with the Rational Rose tool. We experienced that our experimental CASE tool helped the software engineers in applying heuristic rules in designing their systems. Further, with our approach we showed that it is possible to detect the flaws of a method. As an example, we listed some problems of OMT. These problems were identified as imprecise rules, cyclic dependencies and unconnected artifacts.

The problems that we encountered in representing the heuristic rules can be grouped as problems related to the method being formalized and the problems related to the adopted knowledge representation and reasoning technique. We suggested the use of fuzzy-logic for reasoning in design heuristics since fuzzy-logic allows specifying overlapping alternatives and deferring the abrupt choices.

The agent-based architecture has been proved very suitable for preserving the context of reasoning for each artifact. Each agent can easily encapsulate its context. A difficulty that we experience with the agent-based approach is to trace the dependencies among agents. We are currently building dedicated tools for this purpose.

References


Appendix A: Analysis Rules of OMT

The following rules represent OMT rules from the analysis phase:

Tentative class

Tentative classes are defined by extracting nouns from the problem statement:

Tentative Class identification

1. Extract nouns from the problem statement
2. Select nouns as tentative classes.
3. Extract tentative classes from application domain knowledge.
4. Extract tentative classes from general knowledge.

Class identification

Class

The identified tentative classes are used to identify classes:

5. IF tentative class <isRedundant> THEN eliminate tentative class.
6. IF tentative class <isIrrelevant> THEN eliminate tentative class.
7. IF tentative class <isVague> THEN eliminate tentative class.
8. IF tentative class <isAttribute> THEN select tentative class as Attribute.
9. IF tentative class <isOperation> THEN eliminate tentative class.
10. IF tentative class <isRole> THEN eliminate tentative class.
11. IF tentative class <isImplementation> THEN eliminate tentative class.
12. Select remaining tentative classes as right classes.

Data Dictionary

13. Prepare a Data Dictionary for all modeling entities.

Tentative Association

Tentative associations are derived from identified verbs from problem statement:

14. Extract all the static verbs and verb phrases from problem statement.
15. Extract implicit verb phrases from the problem statement.
16. IF a verb phrase <includesPhysicalLocation> THEN the verb phrase is a tentative association.
17. IF a verb phrase <includesDirectedAction> THEN the verb phrase is a tentative association.
18. IF a verb phrase <includesCommunication> THEN the verb phrase is a tentative association.
19. IF a verb phrase <includesOwnership>
   THEN the verb phrase is a tentative association.
20. IF a verb phrase <includesSatisfactionOfCondition>
   THEN the verb phrase is a tentative association.
21. Extract tentative associations from application knowledge.

Association
Right associations are derived from the tentative associations and from the application domain:
22. IF a tentative association <includesEliminatedClass>
   THEN eliminate tentative association or restate tentative association.
23. IF a tentative association <isIrrelevant>
   THEN eliminate tentative association.
24. IF a tentative association <isImplementationConstruct>
   THEN eliminate tentative association.
25. IF a tentative association <isTransientEvent>
   THEN eliminate tentative association.
26. IF a tentative association <isDerivedAssociation> and <multiplicityConstraintNonImportant> and not <useful>
   THEN eliminate tentative association.
27. IF a tentative association <representsConditionOnAttribute>
   THEN eliminate tentative association.
28. Select remaining tentative associations as right associations.
29. Add missing overlooked tentative associations from application domain.

Binary and Ternary Association
The following rules are used to distinguish between binary and ternary associations:
30. IF an association <includesTwoClassTerms>
   THEN select tentative association as binary association
31. IF an association <includesThreeClassTerms>
   THEN select tentative association as ternary association

Aggregation and Qualified Association
Binary associations may be transformed to aggregations. Ternary associations may be transformed to binary associations or qualified associations.
32. IF a binary association <isPartOfRelation> and <isTransitive> and <antiSymmetric>
   THEN select binary association as aggregation.
33. IF a ternary association <includesDescriptiveDependentTerm>
   THEN select ternary association as binary association.
34. IF a ternary association <can be qualified by a name>
   THEN select ternary association as qualified association.

Tentative Attribute
Tentative attributes are derived from the nouns in the problem statement, the tentative classes and from the application domain knowledge:
35. Extract tentative attributes from application domain knowledge.
36. IF a noun in the problem statement is <followedByPossessivePhrase>
   THEN select noun as a tentative attribute.
Attribute

The next step is the production of attributes from the tentative attributes.

37. IF a tentative attribute is <Relevant>
    THEN eliminate the tentative attribute.
38. IF a tentative attribute <isForImplementation>
    THEN eliminate the tentative attribute.
39. IF a tentative attribute <isDerivedAttribute> and not <useful>
    THEN eliminate the tentative attribute.
40. IF a tentative attribute <isIndependent>
    THEN select attribute as a class
41. IF a tentative attribute <dependsOnContext>
    THEN eliminate the tentative attribute.
42. IF a tentative attribute <isName> and <selectsObjectSet>
    THEN eliminate the tentative attribute.
43. IF a tentative attribute <representsIdentifier>
    THEN eliminate the tentative attribute.
44. IF a tentative attribute <dependsOnPresenceOfLink>
    THEN eliminate the tentative attribute.
45. IF a tentative attribute <representsInvisibleInternalStateOfObject>
    THEN eliminate the tentative attribute.
46. IF a tentative attribute <affectsMostOperations>
    THEN eliminate the tentative attribute.
47. Select remaining tentative attributes as right attributes.

Inheritance

The next step in OMT’s analysis phase is to organize classes by using inheritance. Inheritance can be either used for generalization or specialization. During generalization common aspects of existing classes are abstracted into a superclass. Specialization refines existing classes into specialized subclasses. Two classes have inheritance relation if they share the same attributes, associations or operations. The following rules deal with producing inheritance artifacts.

48. IF class A <hasCommonAttributes> with class B
    THEN define generalization inheritance for class A and class B.
49. IF class A <hasCommonOperations> with class B
    THEN define generalization inheritance for class A and class B.
50. IF class A <hasCommonAssociations> with class B
    THEN define generalization inheritance for class A and class B.
51. Define generalization inheritance for class A and class B from application domain.
52. Define inheritance specialization relations from existing taxonomy in real world and AD.
53. Define inheritance specialization by looking for noun phrases composed of various adjectives on the class name.
54. IF inheritance <hasExcessiveRefinements>
    THEN inheritance <isWeak>
55. IF class in inheritance <isWeak>
    THEN inheritance is <isWeak>
56. Refine attributes and classes from inheritance relations if needed.

Object Model

After the production of classes and associations it is advised by OMT to draw the initial object model:
57. Define/draw the initial object model with the given associations.

The initial object model is refined with the identified attributes:

58. Refine the object model with the given attributes.

After identification of inheritance relations the object model is further refined:

59. Refine the object model with the inheritance relations.

60. Verify that access paths exist in the object model for likely queries.

Finally the object model is iterated with the following rules:

61. IF inheritance is <asymmetric>
   THEN add class to inheritance.

62. IF class <hasDiscordantAttributes> and <discordantOperations>
   THEN split the class so that each class is coherent.

63. IF association <isAsymmetric>
   THEN add class to association.

64. IF class <playsTwoRoles>
   THEN split class.

65. IF operation <hasBadTargetClass>
   THEN add new class.

66. IF association A <hasSameName> and <samePurpose> with association B
   THEN generalize the associations and add super class.

67. IF class in association <isRole> and <roleShapesSemanticsOfClass>
   THEN add class for the role.

68. IF class <lacksAttributes> and <lacksAssociations> and <lacksOperations>
   THEN eliminate class.

69. IF operation <hasMissingAccessPaths>
   THEN add association so that queries can be answered.

70. IF association <isRedundant>
   THEN eliminate association.

71. IF association <hasTooBroadRoleNames>
   THEN move the association up in the hierarchy.

72. IF association <tooNarrowRoleNames>
   THEN move the association down in the hierarchy.

73. Sign of incorrect placement of attributes.

Module

The last step in OMT’s object model in analysis phase is the grouping of classes into modules:

74. Extract modules from application domain.

75. IF module <isNotLogicalSubsetOfClass>
   THEN eliminate module.

76. IF class A is <tightlyCoupledWith> class B
   THEN put class A and B in the same module.