JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002
Vol. 1, No. 2, July-August 2002

Representing Design Patterns and Frameworks in UML — Towards a Comprehensive Approach

Yasunobu Sanada and Rolf Adams
Faculty of Information Sciences, Hiroshima City University, Japan

Design patterns and frameworks have become important concepts in object development. As well important is UML as the standard modeling language. But there is not sufficient support to model design patterns and frameworks in design class diagrams (DCDs) without using the extension mechanisms, that is, stereotypes, constraints, and tagged values. Some approaches have been developed to improve the representation by extending UML. But they are either not comprehensive, or not well-defined, or don't consider the granularity or complexity of DCDs. In this paper we present a more comprehensive and well-defined approach by using an example, distinguish between DCDs, detailed DCDs, and design pattern CDs, define UML profiles for the extensions, and outline how an UML tool can support the approach.

1 INTRODUCTION

Object-oriented techniques such as frameworks [13] and design patterns [8] make designs more flexible, extensible, and reusable. When documenting the design in a framework or when documenting the structure of a design pattern developers usually use UML [3, 2, 9] design class diagrams (DCDs). But standard UML DCDs often don't provide important information necessary to understand or extend the design.

To solve this problem several approaches have been suggested [1, 5, 6, 7, 10]. The most important approach is described in [1] and called UML-F. The authors suggest to use the UML extension mechanisms, that is, stereotypes, constraints, or tagged values, to represent the designs in frameworks. But their approach suffers from several shortcomings. First, they don't clearly distinguish between the three kinds of extension mechanisms and they don't define UML profiles for the extensions (until recently [4]). Second, their extensions are not comprehensive and there exist several more extensions that are useful in understanding the design. Third, they consider only partly the granularity of DCDs, such as distinguishing between DCDs and detailed DCDs. Finally, they don't consider the surface or complexity of DCDs,

Cite this article as follows: Yasunobu Sanada and Rolf Adams, ***Representing Design Patterns and Frameworks in UML — Towards a Comprehensive Approach***, in Journal of Object Technology, vol. 1 no. 2, July–August 2002, pages 143–154, http://www.jot.fm/issues/issue_2002_07/article3

that is, which mixture of extensions and diagrams result in an easy to understand DCD.

The approach described in [5] uses the extension mechanisms to improve the representation of so called configuration design patterns. Hence, this approach is used to represent the static structure of some design patterns. But it resembles partly the approach in UML-F and has also the problem that it is not comprehensive.

The approach described in [6] emphasizes the distinction between whitebox and blackbox hotspots or variation points. The approach is based on changing the visual appearance of DCDs and hence is not compatible with standard UML. Furthermore it is certainly not comprehensive.

Another approach, based on designing a system as a composition of design patterns, is described in [10]. Such an approach might be appropriate for some systems, but it is certainly difficult to apply for many kinds of systems. It requires detailed understanding of many design patterns and their interaction.

A similar but more general approach is described in [7]. The authors propose the use of role diagrams to explicitly document the interaction between classes. Such role diagrams are not part of standard UML. Furthermore they don't explicitly model variation points and their instantiation.

In this paper we present an approach that extends the work done before. It is aimed to be more comprehensive and well-defined. To illustrate our approach we use an example from an easy to understand application domain: a grade recording framework. First we design the system with common DCDs, then we design the system again with DCDs that contain our proposed extensions. After having hopefully convinced the reader that our extensions help in understanding or extending a design, we provide definitions of all extensions as UML profiles. The approach distinguishes between DCDs, detailed DCDs, and design pattern CDs, and this distinction should be reflected in an UML tool. So, we also explain how we changed an existing UML tool so that it supports this distinction.

2 EXAMPLE: DEVELOPING A GRADE RECORDING SYSTEM

Assume that we want to develop a framework for recording grades. Such a system can be used at any school, university, or company to record the grades. Here we don't consider developing the full requirements for this application. We assume that developing the full requirements or domain model might reasonably result in our parts of a design, although other designs are certainly possible.

In such a domain there exist certainly classes to represent students, teachers, lectures, tests, etc. Teachers might offer reports, written and oral examinations, and many other kinds of tests. A natural representation for such a structure uses the Composite pattern [8]. A partly unresolved issue is the procedure of how teachers compute the grades. Often teachers collect several grades and compute the final

grade by using a certain weighting function for the subgrades. But there is a lot of variety in how the final grade is computed. So, we assume that the class 'Test' has some varying method 'compute' to compute the final grade out of the subgrades. The initial design of the framework is shown in the DCD in Figure 1.

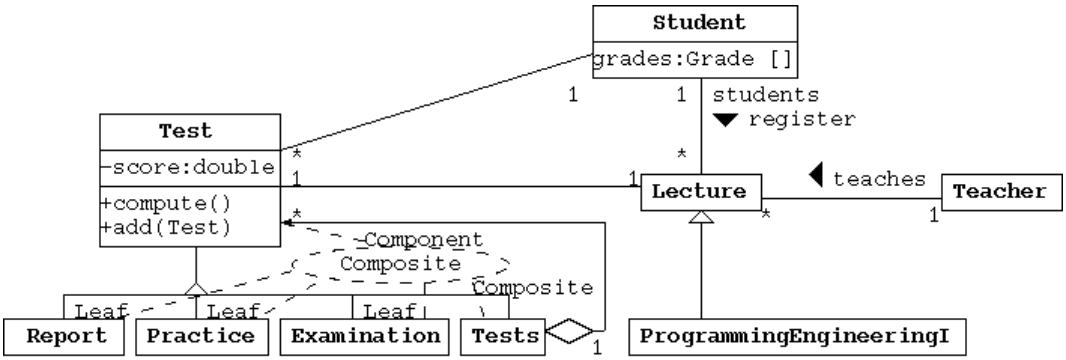


Figure 1: DCD of Grade Framework without Extensions

In the following we consider how the variation point in this design should be implemented. Different teachers usually use different weighting functions and even one teacher usually uses different weighting functions for different lectures. So, it is reasonable to assign the responsibility to implement the weighting function to the class 'Lecture' and use the Strategy pattern [8] to implement the method 'compute' in class 'Test'. The resulting design is shown in Figure 2. We call this DCD a detailed one because it cannot be refined anymore and the implementation of the variation point has been resolved.

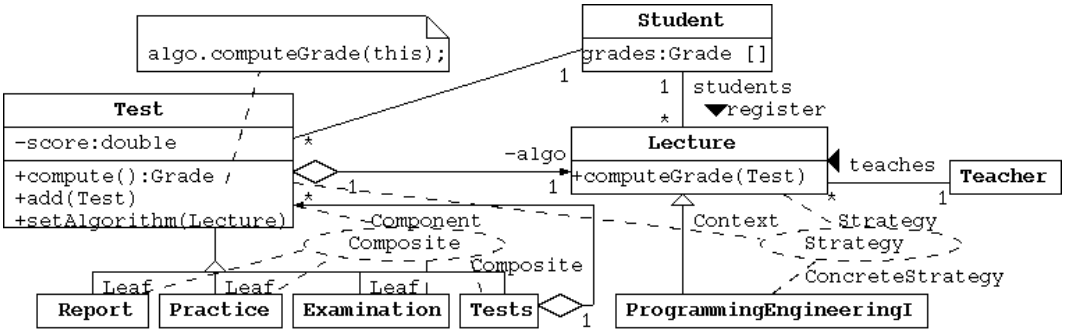


Figure 2: Detailed DCD of Grade Framework without Extensions

3 USING UML EXTENSIONS IN THE EXAMPLE

In this section we will show the same DCDs for our grade recording framework but add some information using the standard UML extension mechanisms, that

is, stereotypes, constraints, and tagged values. We explain which additional information these extensions reveal and argue that those information is important in understanding or extending the design. Finally we argue that the detailed DCDs in Figure 2 and 4 are both complex, that is, its visual appearance and understanding are complex. So, we introduce another extension, namely tagged values that indicate the roles of participants in design patterns. In cases where the appearance of a DCD becomes too complex by using collaboration diagrams, these tagged values might be preferable.

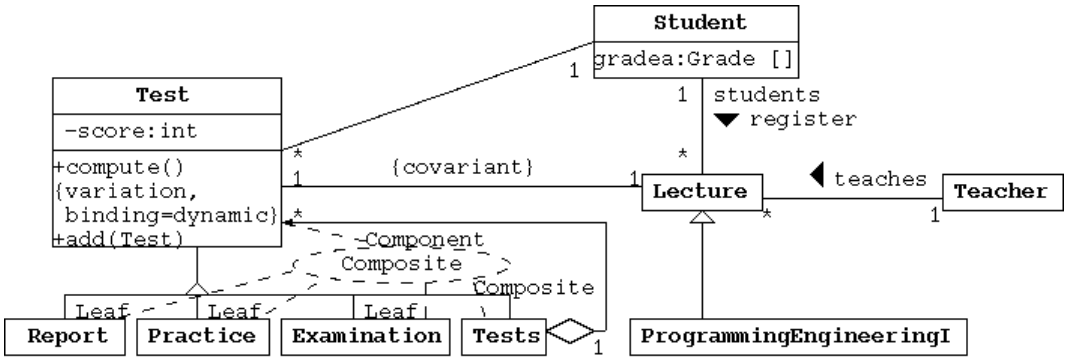


Figure 3: DCD of Figure 1 with Extensions

Figure 3 shows the DCD of Figure 1 but contains two extensions. These extensions provide us the following information:

variation, binding=dynamic These tagged values show that the method 'compute' is a variation point in the design and that its instantiation is dynamic, that is, it is necessary to support changing its instantiation at runtime.

covariant This constraint shows that the classes 'Test' and 'Lecture' are covariant classes, that is, they work cooperatively and adding a subclass to 'Lecture' might result in adding a subclass to 'Test', or vice versa.

The detailed DCD of Figure 2 with the added extensions is shown in Figure 4. The extensions provide us the following information:

≪Application Class≫ An application specific class is 'Programming Engineering I', that is, this class is not part of the framework, rather of the framework instantiation.

extensible=false When we create a new application specific class, such as 'Programming Engineering I', the interface of the class must not be extended, that is, the framework instantiator must not define new attributes or methods.

≪Hook≫ The method 'computeGrade' of class 'Lecture' is a hook method, that is, the framework instantiator must overwrite this method if adding a subclass.

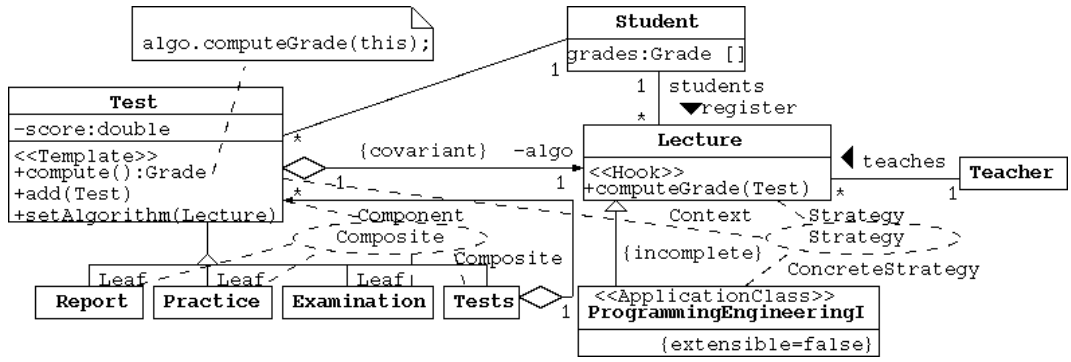


Figure 4: Detailed DCD of Figure 2 with Extensions

≪**Template**≫ The method 'compute' of class 'Test' is a template method that uses the hook method 'computeGrade' of class 'Lecture' to implement the algorithm for computing a grade.

incomplete The number of subclasses of class 'Lecture' is not fixed, that is, the framework instantiator can add new subclasses during framework instantiation.

covariant Same as in Figure 3 described before.

Another important aspect in representing designs as DCDs is the complexity of the layout and the visual appearance. In some cases, such as in Figure 2, using collaboration diagrams to illustrate design patterns, might result in a DCD that is difficult to understand. So, we suggest to use in such cases tagged values as an alternative. The resulting detailed DCD is shown in Figure 5. In case some kind of automatic or semi-automatic layout is supported in a tool, the visual appearance depends on the layout algorithm. But we think that there are always cases where tagged values are preferable to collaboration diagrams.

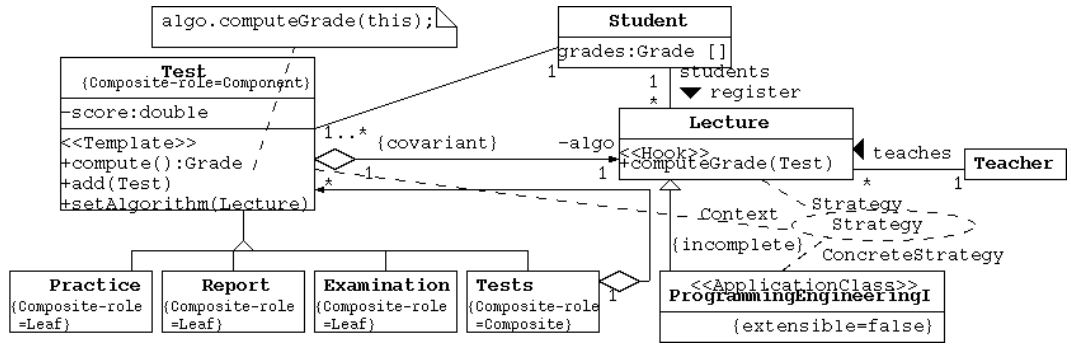


Figure 5: Alternative Detailed DCD for Detailed DCD in Figure 4

4 UML PROFILES FOR ALL EXTENSIONS

After having explained some of the extensions using an example in the previous sections, we provide a complete definition of all extensions in this section. That is, we provide explanations of all extensions in natural language and define UML profiles [9] for them.

First we provide an overview of all extensions and classify them according to their purpose or use. Figure 6 shows all extensions and classifies DCDs in three kinds: DCDs, detailed DCDs, and design pattern CDs. The extensions for each kind of DCD vary although some are in common. The distinction between a DCD and a detailed DCD have been described in the example given before. A detailed DCD results from a DCD by resolving the variation points. A design pattern CD is used to describe the static structure of a design pattern. We don't give an example of a design pattern CD here, but we developed one for the design pattern to implement enumerated types in Java as described in [12]. This pattern can alternatively be used to represent the different kinds of tests in our example.

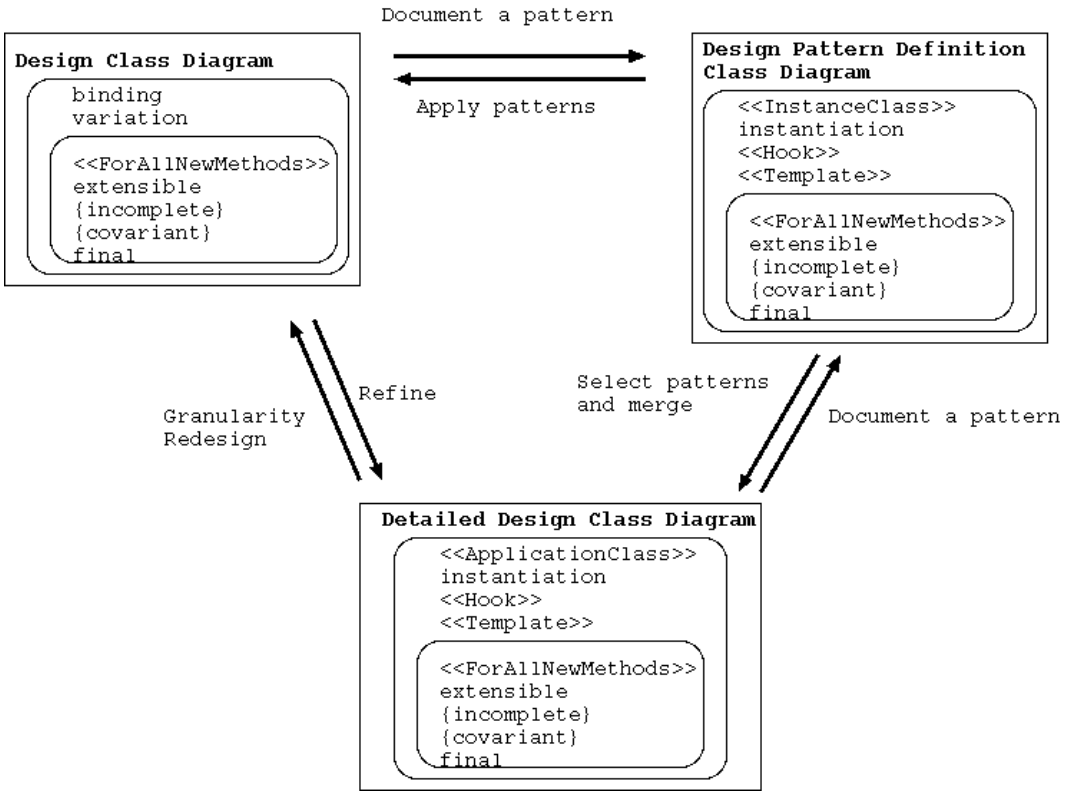
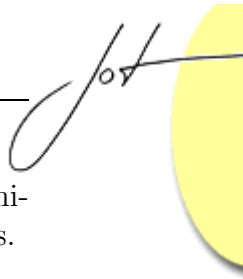


Figure 6: Three Kinds of DCDs with their Relationships and Extensions

The different kinds of UML class diagrams don't necessarily correspond to different iterations in an iterative and adaptive development process [14]. In our grade recording example the given DCD and detailed DCD can be produced in one iter-



ation in case the variation point is considered an important part of the core architecture of the system. But they can as well be produced all in different iterations.

UML Profile for Design Patterns

Description of Stereotypes in the UML Profile for Design Patterns

We define four stereotypes in the UML profile for design patterns: ≪InstanceClass≫, ≪ForAllNewMethods≫, ≪Template≫, and ≪Hook≫ (see Table 1).

Table 1: Stereotypes for Design Patterns

Stereotype	**Base Class**	**Parent**	**Tags**	**Constraints**
InstanceClass ≪InstanceClass≫	Class	N/A	extensible instantiation final	None
ForAllNewMethods ≪ForAllNewMethods≫	Constraint	N/A	None	None
Hook ≪Hook≫	Method	N/A	None	None
Template ≪Template≫	Method	N/A	None	None

InstanceClass Its meaning is the same as in [5]. Generally, instance classes model the varying concept encapsulated by the pattern. New instance classes are defined during the pattern instantiation. So, they are a kind of classes and based on an existing model element, Class, hence it should be a stereotype. Required tags are `extensible`, `instantiation`, and `final`. If stereotype ≪InstanceClass≫ has tag `final=true`, a tag `instantiation=replace` must be specified. The reason is, in order to create a pattern instance, no descendant classes of an instance class can be created. Otherwise, the value of the `instantiation` tag is not determined and an `extensible` tag should be specified.

ForAllNewMethods The stereotype ≪ForAllNewMethods≫ has the same meaning as in [1] and indicates that the constraint is meant to hold for all newly introduced methods. Usually constraints are described in OCL, a programming language, or in natural language, and restrict a certain implementation. By using the stereotype ≪ForAllNewMethods≫ we can restrict all future implementations that use a framework.

Template and Hook They indicate the roles of methods in the pattern. Template and hook methods often appear as the gimmick of extensibility in design

patterns. Template methods define abstract behavior or generic instantiation in interaction between classes and hook methods supply the concrete implementation. Treatment of template and hook methods is different from usual methods, so it should be a stereotype.

Description of Tags in the UML Profile for Design Patterns

We define three tags in the UML profile for design patterns (see Table 2):

Table 2: Tags in the UML Profile for Design Patterns

Tag	**Stereotype**	**Type**	**Multiplicity**
extensible	N/A	UML::Datatypes::Boolean	1
instantiation	InstanceClass	UML::Enumeration: {replace, extend}	1
final	N/A	UML::Datatypes::Boolean	1

extensible When a new instance class is created, we can add new attributes and new methods. Usually, if an instance class has tagged value `instantiation=replace`, `extensible=true` is also specified. And in the case that the tagged value `instantiation=extend` is attached and the instance class is used without accessing through a reference to the base class or interface, the tagged value `extensible=true` must be used.

instantiation The tag indicates how to instantiate classes. The tag can have one of two values: `replace` or `extend`. The value `replace` represents, that when we apply a pattern to design and create new instance classes, instance classes are replaced. If a design pattern is designed in order to inherit a pattern instance class, we specify the value `extend`.

final This keyword has the same meaning as the Java language keyword `final`. A final class has no descendent classes. It cannot be used in some programming languages, but we want to indicate this. An instance class with tag `final` is an end leaf class and we cannot inherit the class. Since we can create no descendent instance class, we use `instantiation=replace` and don't use `instantiation=extend`.

UML Profile for Frameworks

Description of Stereotypes in the UML Profile for Frameworks

We define four stereotypes in the UML profile for frameworks (see Table 3). The only difference to design patterns is the use of ≪ApplicationClass≫ instead of ≪InstanceClass≫:

Table 3: Stereotypes for Frameworks

Stereotype	Base Class	Parent	Tags	Constraints
ApplicationClass ≪ApplicationClass≫	Class	N/A	extensible instantiation final	None
ForAllNewMethods ≪ForAllNewMethods≫	Constraint	N/A	None	None
Hook ≪Hook≫	Method	N/A	None	None
Template ≪Template≫	Method	N/A	None	None

ApplicationClass The meaning is the same as in [1]. It indicates application specific classes and classes that exist only in the framework instance. When design patterns are used in the framework instantiation process, classes with stereotype ≪InstanceClass≫ may become application specific classes.

Description of Tags in the UML Profile for Frameworks

We define five tags plus a special set of tags in the UML profile for frameworks (see Table 4). The meaning of three tags is the same as or analogous to the meaning in the profile for design patterns. The meaning of the additional tags is as follows:

Table 4: Tags in the UML Profile for Frameworks

Tag	Stereotype	Type	Multiplicity
variation	N/A	UML::Datatypes::Boolean	1
extensible	N/A	UML::Datatypes::Boolean	1
binding	N/A	UML::Enumeration: {static, dynamic}	1
instantiation	ApplicationClass	UML::Enumeration: {replace, extend}	1
final	N/A	UML::Datatypes::Boolean	1
PatternName-role	N/A	UML::Datatypes::String	1

variation It is the same as in [1]. It means that the method implementation is the varying concept that the pattern encapsulates. Or, in other words, the method implementation depends on the framework instantiation. We add the meaning that the method is a variation point that should be resolved in a detailed DCD.

binding This tag indicates whether runtime instantiation is required for variation points, and whether runtime change is used to realize variation points. Each variation point should be marked by the binding tag and the value `static` or `dynamic` (but not both, they are exclusive). UML-F has boolean tags `static` and `dynamic`. They are exclusive values, but it should be one tag (`binding`) and two assignable values (`static` and `dynamic`). This is the only addition to the variation point, so it is a tagged value. The tag `binding` should be specified with the tag `variation`.

PatternName-role These tags specify the roles of the participants in patterns. The name of the tags has a fixed form: name of the pattern plus `-role`. The values are strings and show the role name.

5 SUPPORT IN AN UML TOOL

We distinguish in our approach between three kinds of DCDs as shown in Figure 6. An UML tool should reflect this distinction and their relationships and provide an appropriate functionality. For example if an UML tool shows a DCD with a variation point it should be possible to easily switch to a detailed DCD where a variation point is instantiated, for example by selecting a menu item. Similar the opposite direction should be supported. The user might not be familiar with a certain design pattern and hence it should also be easy to switch forward and backward to a design pattern CD or full definition of a design pattern.

We implemented some part of the above functionality by extending an open source UML modeling tool, Argo/UML [11]. We added support of the standard UML extension mechanisms and the possibility to switch between DCDs and associated detailed DCDs. The output of a detailed DCD is shown in Figure 7.

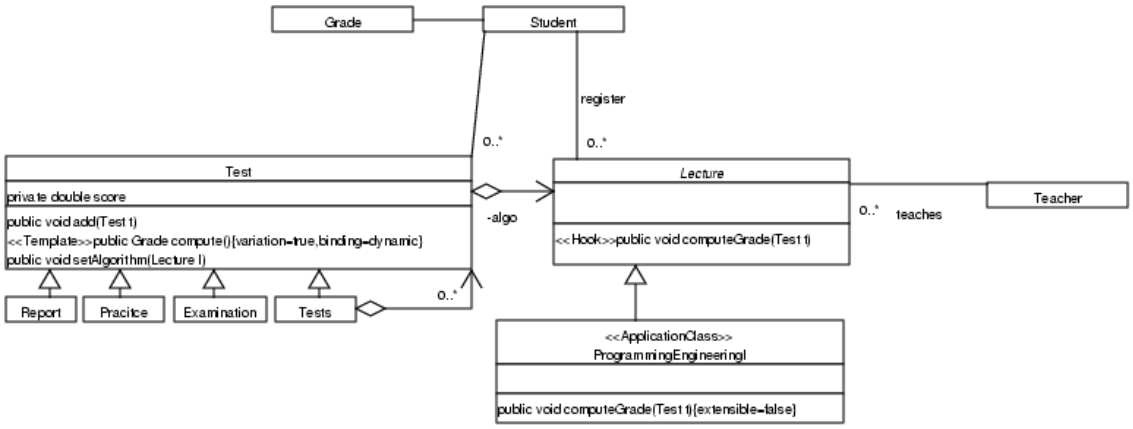


Figure 7: The Output of a Detailed DCD with Argo/UML

6 CONCLUSIONS

Representing design patterns and frameworks in UML with DCDs is not adequate when using no extensions. In this paper we present several extensions based on using the standard extension mechanisms of UML, that is, stereotypes, constraints, and tagged values. Our contributions are:

- providing a well-defined, more comprehensive set of extensions,
- using an example to illustrate the usefulness of our extensions,
- classifying DCDs in usual DCDs, detailed DCDs, and design pattern CDs,
- providing an outline of how UML tools should support our approach.

Future work should collect experience in using our approach.

REFERENCES

[1] Marcus Fontoura and Wolfgang Pree and Bernhard Rumpe, 'UML-F: A Modeling Language for Object-Oriented Frameworks', *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 63–82, Springer, LNCS, Vol. 1850, 2000

[2] James Rumbaugh and Ivar Jacobson and Grady Booch, 'The Unified Modeling Language Reference Manual', Addison-Wesley, 1998

[3] Grady Booch and James Rumbaugh and Ivar Jacobson, 'The Unified Modeling Language User Guide', Addison-Wesley, 1998

[4] Marcus Fontoura and Wolfgang Pree and Bernhard Rumpe, 'The UML Profile for Framework Architectures', Addison-Wesley, 2002

[5] Marcus Fontoura and Carlos José P. de Lucena, 'Extending UML to Improve the Representation of Design Patterns', *Journal of Object-Oriented Programming, Vol. 13, No. 11*, pages 12–19, 101communications, March 2001

[6] Nadia Bouassida and Hanêne Ben-Abdallah and Faïez Gargouri, 'A UML based Design Language for Framework Reuse', *In Proceedings of the 7th International Conference on Object-Oriented Information Systems (OOIS 2001)*, pages 211–221, Springer, 2001

[7] Dirk Riehle and Thomas Gross, 'Role Model Based Framework Design and Integration', *In Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 117–133, ACM Press, 1998

[8] Erich Gamma and Richard Helm and Ralph Johnson and John Vissides, 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley, 1995

[9] 'OMG Unified Modeling Language Specification V.1.4', September 2001, http://www.uml.org/

[10] Sherif M. Yacoub and Hany H. Ammer, 'UML Support for Designing Software Systems as a Composition of Design Patterns', *In UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 149–165, Springer, LNCS, Vol. 2185, 2001

[11] Argo/UML homepage, http://argouml.tigris.org/

[12] Paul A. Cairns, 'Enumerated Types in Java', *Software-Practice and Experience, Vol. 29, No. 3*, pages 291–297, Wiley, 1999

[13] J. van Gurp and J. Bosch, 'Design, Implementation and Evolution of Object Oriented Frameworks', *Software-Practice and Experience, Vol. 31, No. 3*, pages 277–300, Wiley, 2001

[14] Craig Larman, 'Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process', Prentice Hall, 2002

ABOUT THE AUTHORS

Yasunobu Sanada received the B.E. and M.E. degrees in Information Science from Hiroshima City University, Japan, in 2000 and 2002 respectively. He joined Software Research Associates, Inc., Japan in 2002.

Rolf Adams received the 'Diplom' degree in computer science from the University of Kaiserslautern, Germany, in 1987 and the 'Dr. rer. nat' degree from the University of Karlsruhe, Germany, in 1992. Since 1994 he is an Associate Professor at Hiroshima City University in Japan. His email address is adams@ce.hiroshima-cu.ac.jp.