Combining Behavioural and Structural Software Descriptions

Kim Mens, Tom Mens
{ kimmens | tommens }@vub.ac.be
in cooperation with Patrick Steyaert
Department of Computer Science, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

Position paper presented at the ECOOP '97 Workshop on
Object-Oriented Software Evolution and Re-engineering
by Koen De Hondt

Question: What is the best way to combine structural and behavioural descriptions of software to facilitate reuse and evolution of object-oriented systems?

1 Introduction

To answer the above question, we first need to agree upon a clear definition of structure and behaviour of a software system.

1.1 Behaviour

When reusing software components, a description of their behaviour is desirable to assess the interactions with other components. *The behaviour of a system describes the way in which it functions or operates.*

1.2 Structure

Besides a description of the behaviour, we also need a good description of the structure of a software system. Well-structured software is more comprehensible and easier to adapt, making it more reusable. *The structure describes how the different parts in a software system are arranged.* For example, an object-oriented system is structured into different classes forming class hierarchies, and objects are associated with particular classes by means of an instantiation relation.

1.3 Other considerata

Apart from making a distinction between behavioural and structural descriptions, other considerations need to be made:

- A first distinction can be made between *declarative descriptions* (*what* the system does) and *operational descriptions* (*how* the system functionality is achieved).
- From a design point of view another distinction can be made. *Essential descriptions* describes those parts of the behaviour that are crucial to the system design, for example, message sends describing an important interaction protocol. The remaining descriptions are called *implementation-specific.*

- Finally, a system can also be described statically or dynamically. *Static descriptions* look at compile-time aspects, while *dynamic descriptions* take run-time aspects into account. For example, consider the method dependencies between classes. Without looking at run-time aspects it is possible to extract from the code which message sends *might* occur. In order to find all message sends that *will* occur, extensive data-flow analysis is needed.

2 Current Approaches

The Demeter method for adaptive software [Lie96] is a design notation in which a loose coupling between behavioural and structural information is achieved. The structure is provided by *class structures* and the behaviour is described as independently as possible from this (implementation-specific) structure by means of *propagation patterns*. In this way the assumptions that need to be made about the data structures when the behaviour is specified can be minimised.

Reuse contracts [SLMD96] are a design notation describing the structure and essential behaviour of an object-oriented system. Reuse contracts document, for example, the important self sends that should be made by the methods of a class. This information is used to detect conflicts during evolution, but only at compile-time. This means that reuse contracts (currently) express only static behaviour.

Design patterns [GHJV94] are usually considered as an implementation technique, but can also be viewed as a means of describing the structure and essential behaviour of system parts. While design patterns primarily describe static behaviour, they often deal with dynamic aspects as well.

The object-oriented language Eiffel [Mey88] supports *programming by contract*. *Preconditions* are used to check that input arguments are valid and that an object is in a reasonable state to perform a requested operation. Similarly, *postconditions* verify whether a method has successfully performed its duties, thus "fulfilling its contract" with the caller. While pre- and postconditions can describe both essential and implementation-specific behaviour, in most cases checking preconditions and postconditions can be done only at run-time, so it is a way to deal with the dynamic behaviour. Unlike the previous approaches, pre- and postconditions describe the behaviour in a declarative way.

Larch/C++ [Lea97] is an interface specification language for C++. Like in Eiffel, the behaviour is described in a declarative way by means of pre- and postconditions. Larch/C++ also supports inheritance of specifications and behavioural subtyping.

State diagrams are another notation for describing the dynamic behaviour of a system. Because flat state diagrams lead to a combinatorial explosion of states and transitions when the system evolves, more structured variants like statecharts [Har88] or nested state diagrams have been developed. Lower nesting levels deal with implementation-specific details, while higher levels express more essential behaviour.

OMT [RBP+91] can be viewed as a system modelling technique that combines three different views. The object model represents the static, structural aspects of a system (using object and class diagrams). The functional model specifies the system behaviour in a declarative way. The dynamic model represents the temporal, operational behaviour of a system using event traces and nested state diagrams. Again nesting mechanisms are used to distinguish essential and implementation-specific aspects.

3 Discussion

In this section we discuss the viewpoints, advantages and shortcomings of the different approaches with respect to software reuse and evolution, by looking at how structural and behavioural software descriptions are combined.

3.1 Structure versus behaviour

The structure and behaviour of a software system are strongly related. By structuring the system and code in a certain way, a particular behaviour may become easier to design or implement. Conversely, to obtain a particular behaviour, sometimes the structure needs to be adapted.

Summarised, when considering reuse and evolution of a software system, a good description of both the structure and the behaviour of the system is important, and there is a strong correlation between the two. But the question remains what is the best way to combine the structural and behavioural software descriptions. To answer this question, we take a look at how current approaches try to solve the problem.

Separate structural and behavioural model. Methodologies like OMT use separate models for describing the behavioural and structural aspects of a system. This leads to a better understanding of the system by providing complementary views. However, as the interaction between both models is limited, it is difficult to see how changes to the behaviour affect the structure and vice versa.

Loose coupling of behaviour and structure. Loose coupling, as advocated in [Lie96], is beneficial for reuse as the behaviour usually does not need to be adapted when the structure evolves. Conversely, when the behaviour itself evolves one needs to investigate only the impact on the corresponding parts of the structure. Also note that, although an approach such as the Demeter method describes behaviour and structure separately, it does not have the problem mentioned in the previous point, as the behavioural description explicitly describes which parts of the structure it affects.

Single combined model. Reuse contracts and design patterns provide a single formalism in which both the structure and (essential) behaviour of a system can be described. In this way, the strong correlation between structure and behaviour is made explicit.

3.2 Essential versus implementation-specific descriptions

There is a delicate trade-off between how much essential and how much implementation-specific behaviour should be included in the design. Too little implementation-specific behaviour can lead to conflicts during system evolution that are hard to detect and difficult to solve. The more behaviour is documented, the more conflicts can be detected. However, too much (implementation-specific) behavioural information is undesired as it clutters the design with unnecessary implementation details. The following approaches can be distinguished.

No explicit essential behaviour. Many approaches do not explicitly distinguish essential and implementation-specific behaviour, making it hard to recognise or understand the "core" behaviour of a system.

Mainly essential behaviour. Reuse contracts and design patterns take the opposite approach and mainly focus on the structure and essential behaviour of a system, while ignoring implementational aspects.

Layering. A third possibility is the use of layering or nesting mechanisms to achieve a gradual transition from high level essential behaviour descriptions to low level implementation-specific behaviour descriptions. Nested state diagrams are an example of this.

3.3 Static versus dynamic descriptions

When a system is documented by or annotated with static behavioural descriptions, many evolution conflicts can be detected already at compile-time. Dynamic behaviour descriptions can aid in finding the remaining evolution conflicts that can be detected only at run-time. Most of the current approaches focus either on static behaviour descriptions (reuse contracts, design patterns) or on dynamic behaviour descriptions (Eiffel, state diagrams) but do not combine both, unless in separate models (OMT). An alternative approach would be to combine both kinds of behaviour somehow in a single model.

3.4 Declarative versus operational descriptions

The alternative of operational descriptions is making use of declarative descriptions, as is the case with pre- and postconditions in Eiffel or formal specification languages like Larch. At this point it is not yet clear which alternative (if not both) is most promising with respect to software evolution. Both approaches have their own specific benefits and disadvantages.

Declarative specifications are sometimes advocated as being "more abstract than operational ones, leading to specifications that are easier to understand and are less likely to capture accidental implementation details such as invocation order." While this is clearly an advantage it can also be regarded as a shortcoming, as the relation between the abstract specification and the concrete implementation is sometimes hard to find. This is not the case with operational descriptions. As a result, operational descriptions can be used more easily for automatic code generation and to detect and solve possible conflicts in the implementation.

4 Conclusion

We started out with the question of how to combine structural and behavioural descriptions of software so that reuse and evolution are facilitated. Instead of proposing a particular solution we investigated how current approaches tackle the problem. This lead to a taxonomy of considerations that need to be taken into account when assessing the quality of an approach with respect to software evolution:

1. Structure versus behaviour;
2. Essential versus implementation-specific descriptions;
3. Static versus dynamic descriptions;
4. Declarative versus operational descriptions.

For each pair of complementary descriptions, the following questions need to be answered in the context of software evolution. *Which of the alternatives is best (if not both)? If both are needed, what is the best way to combine them, or to integrate them into a model? What are the repercussions of the made choices on reusability?* Answering these questions is far from easy, as they strongly interact with each other.

We think these are important questions the object-oriented software reuse community should think about to get a better insight in the software evolution process.

References

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addisson-Wesley, 1994.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[Lea97] G. T. Leavens. *Larch/C++ Reference Manual*, 1997. http://www.cs.iastate.edu/ leavens/larchc++.html.

[Lie96] K. J. Lieberherr. *Adaptive Object-Oriented Software. The Demeter Method with propagation patterns.* PWS Publishing Company, 1996.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[SLMD96] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 268–285. ACM Press, 1996.