Handling Complexity of Large Software Systems by Mapping Objects and Classes to Conceptual Architectural Elements

Peter Tabeling, Bernhard Gröne
Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany
{peter.tabeling, bernhard.groene} @hpi.uni-potsdam.de

Abstract

Today, the overall program code of an industrial software system often comprises several thousand classes and interfaces. While UML class diagrams and packages can principally help to gain a better overview of the program code, other techniques are needed to define a corresponding architectural model of the (intended) running system. This applies especially to object-oriented systems where models of runtime object structures have no explanatory power because they represent volatile snapshots with excessive fine granularity. This paper presents techniques to map such structures to coarse-grained and stable architectural models.

Keywords: Software Architecture, Conceptual Architectural Models, Fundamental Modeling Concepts, FMC.

1. Introduction

1.1. System Architecture vs. Code “Architecture”

In the literature, the term “Software Architecture” is often used to identify the “high level, important structures” of a software system. However, only few authors clearly state what kind of structures they refer to. In this context, we should distinguish at least two types of structures, namely (1) *system* structure and (2) *code* structure. The latter consists of code fragments and their relationships. In case of object oriented systems, these are primarily classes, their associations and dependencies. In order to grasp the code “architecture”, these should be further organized by means of packages, for example.

But structures of the code itself must not be confused with structures of the first category, i.e. the structures of the running system which comes into existence when the code is executed. On a very low level, these are large, highly dynamic object structures which are difficult to describe and understand due to their volatility and the fragmentation of the system’s functionality into thousands of small objects. This is reflected by the relatively wide-spread opinion that objects are not abstract enough to describe conceptual system architectures.[1] Instead, “components” and “connectors” are considered to be the first-order abstractions of architectural models.[2]

1.2. The Problem of Mapping Objects and Classes to Architectural Components

To obtain a meaningful architectural model of the system, object structures must be reduced to more abstract, conceptual components and connectors. While deriving a conceptual system architecture from objects and classes is more the “reengineering side” of the problem, the opposite direction is equally difficult and important when developing large systems. In order to to avoid non-traceable transitions from architectural models to software design (and vice versa), practitioners need some rules to perform such mappings.

2. Mapping Policies

In the following, we present different possibilities to map objects and classes to architectural models, depending on typical usages of objects. We use the Fundamental Modeling Concepts (FMC) [2][3][4] as the basis to describe conceptual architectural elements. FMC distinguishes between active components, called *agents*, and passive components, called *channels* or *storages*. While all pro-

1. The term “*conceptual* system architecture” is used here to express the focus on high-level, not hardware-related, system structures.
2. “[...] separating components form connectors, raising them both to visibility as top-level abstractions [...] also raises them in the conciousness of the designer.” [1] p. 19.

cessing is done by agents, storages can be used by them to store data and channels can be used to transmit information between agents. Figure 1 shows the corresponding notation (more information about FMC can be found in [2],[4].)

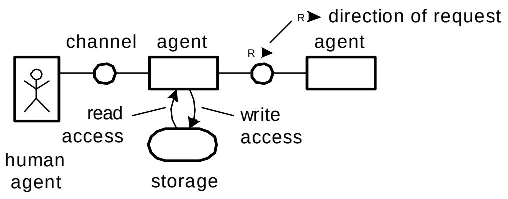

Figure 1. FMC Notational Elements

When mapping objects to agents or storages, three basic mapping types can be differentiated - simple (one-to-one-) mappings, hierarchical (many-to-one-) mappings and many-to-many-mappings. We call the different mappings „views“, because each type of mapping represents a certain way to interpret or “look at” objects.

2.1. Simple Mappings

Object Agent View. Considering an object to be an agent is a common interpretation in object-oriented design. An object agent is an active and abstract component of the system. Calling a method is interpreted as sending a message to a receiver object which carries out the desired operation and responds with an answer. Only the object agent has access to the attribute data associated with the object. Methods describe which messages the object can handle, what operations on its data it can perform and which messages it sends to other objects (see, for example, [5], p.6).

A typical example for this view is an event dispatcher object: GUI toolkits usually use a singleton object to dispatch GUI events (e.g. mouse moved or button pressed) to other objects which have been registered as handler for certain event types. Figure 2 shows this scenario. Objects are modeled as agents with internal storage holding the object‘s attribute data. „Knowing“ other objects by their object ID (reference) is symbolized by a channel which can be used to exchange requests and answers between objects.

Abstract Data Type View. Another common interpretation can be called the ”abstract data type view“. Here, an object is seen as a storage for an abstract data type which is described by the corresponding class. For example, Bertrand Meyer presents this interpretation of objects—he defines a class as „an abstract data type equipped with a possibly partial implementation“ [[6], p.142]. The class not only lists the operation types (i.e. the method signatures) of the abstract data type, it may also provide the implementation of the data storage (i.e. the attribute storages) and the implementation of the operation types (i.e. the method bod-

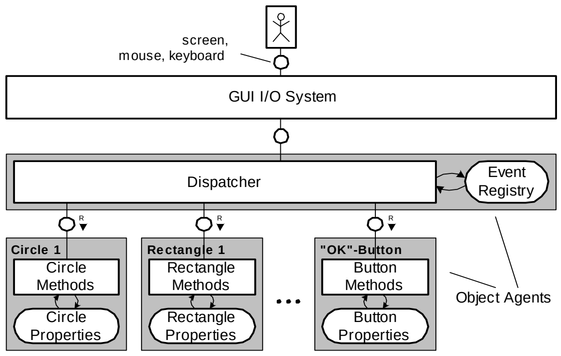

Figure 2. Object Agent View: A Dispatcher (Block diagram)

ies). From this point of view, an object can be seen as a passive system component (a storage), and a method call can be interpreted as an operation which is performed on the object by other system components (i.e. the caller of the method), rather than a message causing the object to perform the operation on its own.

Figure 3 shows a typical example of this type of view. The ”Vector“ class defines an abstract data type ”Vector“ with operation types (scale, add, rotate etc.) as well as the implementation of the Vector data and the operation types. Following the data type view, an instance of the ”Vector“ class can be seen as a storage which holds an abstract ”Vector“ value.

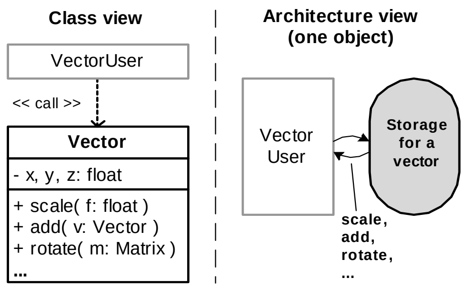

Figure 3. Abstract Data Type View (Class Diagram / Block Diagram)

The simple mappings discussed above actually do not reduce complexity and can be inappropriate in case of multi-threaded systems. Nevertheless, they are typical interpretations of objects which can be found quite often and can be used beside hierarchical and many-to-many-mappings.

2.2. Hierarchical Mappings

High–level Abstract Data Type View. A quite obvious way to model a set of objects is to extend the idea of the abstract data type view. There are cases where the definition of a single class is not sufficient for the realization of an abstract data type. For example, a data type ”tree“ could be needed which stores arithmetic expressions as binary trees. For this purpose, one might define the classes as shown in Figure 4. Instances of classes derived from ”node“ would represent a tree's nodes and a ”tree“ object would be the placeholder of the whole tree. This object provides methods to access the whole object structure, such as calculating the value of the tree (i.e. the value of the corresponding expression).

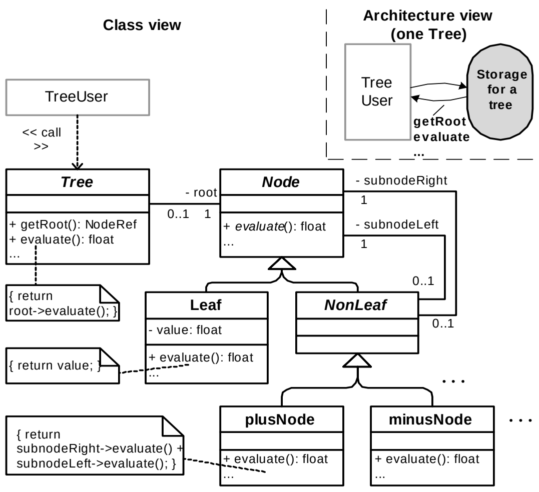

Figure 4. High–Level Abstract Data Type View: Tree Storage (Class Diagram / Block Diagram)

The basic idea behind the tree–related classes is to provide the possibility to store trees. Hence, at a higher level, the complete object set holding a certain tree can be viewed as a single storage for an abstract data type ”tree“ - see upper right corner of Figure 4. This view yields a single storage as an abstract, compact model of an object set. This model remains valid even if the underlying object structure consists of many objects and changes over time - in the architectural model, only the current value (i.e. the tree) in the storage is changed. The operation types defined for the abstract data type (e.g. tree evaluation) are not implemented by a single method but the combined methods of several classes (here: the ”evaluate“ methods).

High-Level Object Agent View. It is sometimes reasonable to combine many objects to one agent. Take, for example, a persistency service which consists of a singleton object of the persistency manager class and a set of class agents, one singleton object for each persistent class.

Figure 5 shows the persistent objects, the persistency service, the transaction service and the database. The task

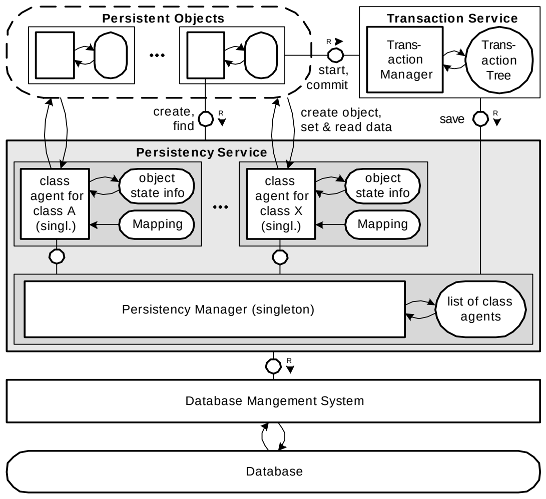

Figure 5. High-level Object Agent View: Persistency Service (Block Diagram)

of the persistency service is to create or load persistent objects from the database and to save altered object data whenever the transaction service requests it in case of a commit. The persistency manager just keeps a list of the classes, while the class agents read and write object data and create objects; they have all information about their class and know which objects have been altered. Class agents are therefore factories for persistent objects and part of the persistency service. The persistency manager is the representative of the persistency service. (This architecture has been chosen for the project ”Object Services“ at SAP in 1999 [7].)

2.3. Many-To-Many Mappings

Functional View. The views presented above have in common that one architectural element (storage or agent) is mapped to many objects. However, this is not appropriate if the architectural model primarily represents a functional decomposition. Figure 6 shows an architectural model of a simple graphic editor where each component provides a certain functionality, namely editing, displaying, printing and persistency. The central storage holds all data describing the graphic drawing currently being modified. The agents rely on certain components of the underlying platform, e.g. the file system.

While this model is very useful for presenting a system overview, a one–to–one or one–to–many mapping of architectural elements to objects would not result in an appropriate code structure. Following object–oriented design principles, we should define a class ”GraphicObject“ with subclasses for the different types of graphical elements, i.e. rectangles, circles etc. - see the class view in Figure 6. The

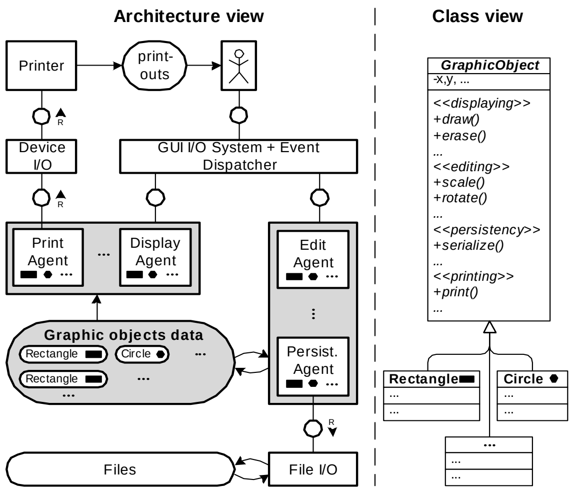

Figure 6. Functional View: Graphic Editor (Block Diagram / Class Diagram)

implementation of displaying, editing and other operations depends on the implementation of the graphic element data. Hence, each of the classes should not only define a storage format for graphic data but also contain methods corresponding to the various operation types. (Of course, additional classes had to be defined beside these classes.)

The example shows a many–to–many mapping of objects to architectural elements. The collective object attribute data is mapped to the ”graphic objects data“ storage, and all methods implementing a certain functionality (e.g. editing methods) are mapped to a corresponding agent (e.g. the ”edit agent“). We call this mapping the ”functional view“ because the architectural model reflects the functionality provided by the program code.

3. Conclusions

The different views presented above foster a mostly systematic mapping of object structures/classes to architectural models. This mapping can be done following certain criteria which are described in [8]. By applying these mappings - especially the one-to-many and many-to-many mappings - large, highly dynamic object structures can be „concentrated“ into a coarse-grained and stable architectural model which is easier to understand and describe. Additional types of mappings are discussed in [8] which help modeling multithreaded systems.

4. References

[1] Nenad Medvidovic, Richard N. Taylor, *A Classification and Comparision Framework for Software Architecture Description Languages,* Dept. of Information and Computer Science, University of California, Irvine, 1997

[2] Frank Keller, Peter Tabeling et. al. *Improving Knowledge Transfer at the Architectural Level: Concepts and Notations,* International Conference on Software Engineering Research and Practice, Las Vegas, June 2002

[3] Frank Keller, Siegfried Wendt, *FMC: An Approach Towards Architecture-Centric System Development,* IEEE Symposium and Workshop on Engineering of Computer Based Systems, 2003

[4] Siegfried Wendt et al. *The Fundamental Modeling Concepts Home Page,* fmc.hpi.uni-potsdam.de

[5] A. Goldberg, D. Robinson, *Smalltalk-80: The Language,* Addison Wesley, 1989

[6] Bertrand Meyer, *Object-Oriented Software Construction,* 2nd Ed., Prentice Hall, 1997

[7] Bernhard Gröne et al. *Object Services für R/3 Release 99 - Konzepte,* SAP AG Walldorf, Germany, 1999

[8] B. Gröne and P. Tabeling, *Mappings Between Object-Oriented Technology and Architecture-Based Models,* International Conference on Software Engineering Research and Practice, Las Vegas, CSREA Press, June 2003