Documentation of Object Interaction[1]

Stefan Chiettini
Institut für Praktische Informatik
Johannes Kepler University Linz, A-4040 Linz, Austria

email: stefan.chiettini@ssw.uni-linz.ac.at

Abstract

The documentation of object-oriented systems usually consists of two parts: First there is the static part with the description of classes and methods. This part usually contains information about interfaces, inheritance relations, and aggregations. The second part, which is the topic of this paper, describes the dynamic behaviour of the system in a certain situation at run time. Common design and documentation techniques like OMT [Rum91] or UML [Fow97] introduce event *trace diagrams* (OMT) and *sequence diagrams* (UML) to visualize run time behaviour of interacting objects. These diagrams show the message sequence in a certain situation at run time. Their major weakness is that they are themselves static and therefore capable to illustrate only one special case typically called a 'scenario', not general behaviour of objects.
This paper proposes *behaviour diagrams* as an extension of existing diagrams to meet the requirements of modern documentation: structured documents with hypertext and multimedia capabilities extended with the possibility to interactively explore the documentation.

[1] This work was supported by the "Fonds zur Förderung der Wissenschaftlichen Forschung (FWF)" under project number P11350-TEC

1. Introduction

Modern object-oriented systems focus on reuse (code reuse in class libraries, design reuse in frameworks). In most cases the reusable classes are part of a huge code base that has to be understood in order to locate and use them correctly. The size and complexity of these code bases has increased to a level where it is impossible to understand their structure without appropriate documentation.

The reusable parts of the given system must be customized and/or parameterized to use them correctly, in many cases it is necessary to know not only the static structure but also the object collaboration at run time. When adapting a framework, for example, the programmer supplies only chunks of code (i.e. extensions) that are plugged into the framework and get called at certain times. For the correctness of the overall system, it is crucial to understand under which circumstances the extensions are called and what is the state of the system at those moments. It might also be necessary to know in which order extensions are invoked and which classes and methods are involved at all in a particular scenario. Since in most systems the reused parts are numerous and complex this is a hard task.

To help the user understand the context of the system, good documentation is therefore an indispensable part of any codebase providing reusable software. Modern documentation should meet many requirements. It should be detailed enough for maintenance programmers yet universal for new system users. Documentation should be easy to build, easy to maintain and easy to use.

Documentation Structure

To guarantee these requirements, documentation has to be structured to be usable for various user groups. Different entry points into the documentation have to be provided. Structure can be supported by using hypertext as a base for all documents. Multimedia (animation, video) should be used as additional presentation layer to support the learning process of the user. For more information see Section 2.

Documentation Of Different Aspects

Documentation has to contain information about the static structure as well as the dynamic behaviour of a software system (more information see Section 3). Information about the static structure is sufficient when using isolated interfaces or standalone objects that have no or little interaction with other objects (such as collection classes or mathematical classes). But today frameworks are more and more gaining importance in software engineering. Frameworks rely on complex objects interaction, in fact a large part of the design decisions is the design of the overall object structure and the interaction between these objects (not the interior of the objects). The primary goal for the user is to understand not only each of the objects but the complete system with all its different facettes at run time. This requires additional documentation about the system’s run time behaviour.

2. Documentation Structure

The documentation of object-oriented systems should not be presented in one large block at the deepest level of detail. Since there are many potential users of the system, documentation should be structured according to the required abstraction level for each user group [Fre96], [Meu97]. To guarantee appropriate information for all user groups, it is necessary to split the documents and to provide different entry points. Different entry points should be provided in the form of different start documents. Splitting the documents keeps irrelevant information away from particular users.

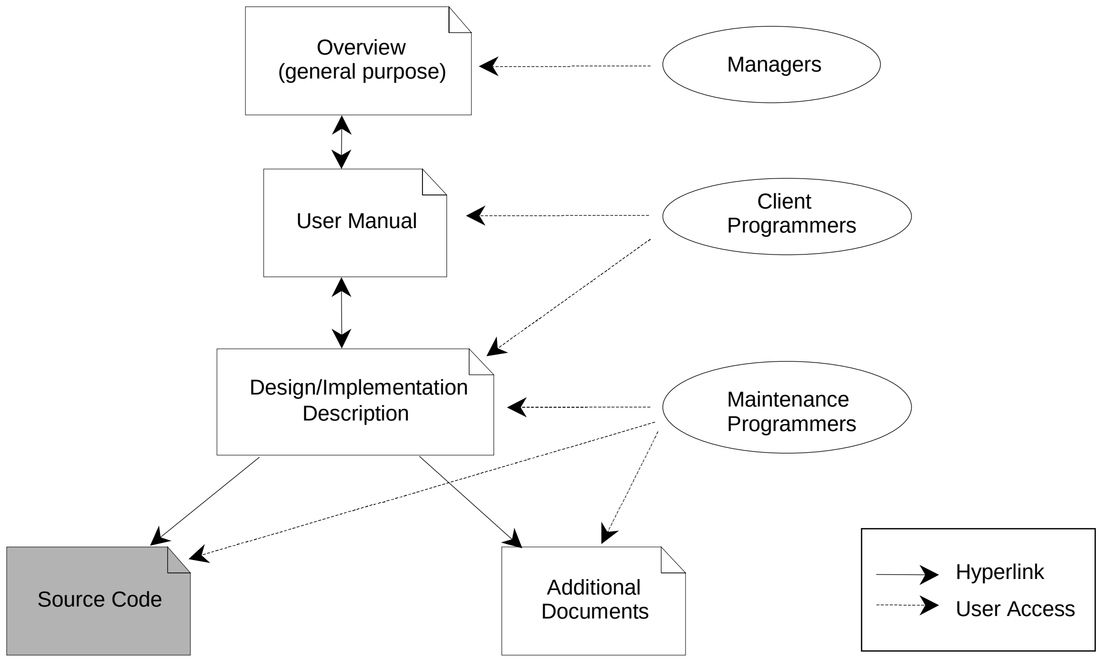

Figure 1: Documentation Structure for Different User Groups

The structure shown in Figure 1 meets the requirements of the different user groups:

- *Project managers* and *executive managers* who have to decide whether a system shall be used for a particular project or not. These users need an accurate yet short description of the purpose and the functionality of the system. They will start their investigation of the documentation at the point where the overall purpose of the system is described. In several cases they may be interested in how to use the system, too, but will rarely need detailed information about its internal structure.

- *Client programmers* new to the system need a description on how to use it in order to produce working applications in a short time. Advanced users will also need direct access to detailed documentation of the system context (static and dynamic documentation).

- *Maintenance programmers* need detailed documentation, too, but also require access to the source code (the most finegrained level of information). Links to the source code typically are a special form of hyperlink. Eventually existing additional documents (design specifications, results of analysis phases or standard documents like RFCs or specifications of standards) should be integrated into the documentation for fast and easy access.

3. Documentation Aspects

This section gives an overview of existing documentation techniques and tools. They are grouped by their focus:

- documentation of static structure ('Static Documentation')
- documentation of run time behaviour ('Dynamic Documentation')

3.1. Static Documentation

The following information is part of static documentation:

- Class descriptions (purpose, general description, inheritance relations).
- Method and attribute descriptions (interface, parameters, return values).
- Relations and cooperations between classes.

This sort of documentation is essential for the reuse of classes. A vast number of tools exists to support the generation of static documentation. UML editors may be used to document inheritance relations between classes. Descriptions of classes, methods, and attributes are supported by hypertext authoring tools. Pattern languages [Joh92] are useful when structuring documents. HTML enables online availability of software-documentation on the internet.

Some tools for commercial use on this wide field are

- javadoc (www.javasoft.com)
- Rational Rose™ (www.rational.com)
- SNiFF+™ (www.takefive.co.at)
- ObjectManual™ (www.obsoft.com)
- CDOC Professional™ (www.swbs.com)

These tools analyse source code and extract meta information about inheritance relations, aggregations, type information and embedded comments. They generate class trees, hyperlinked indexes, and keyword databases. Sometimes links to the appropriate position in the source code are generated. All these tools rely on the static structure of the software system, many parse source code to automatically create documentation. Hypertext has proven its usability in these tools and is a quasi-standard for good documentation now, though it is often extended with a separate query mechanism for keyword search. Documents are structured and indexed, in frameworks sometimes *cookbooks* [Pre95] are provided to explain how certain adaptations have to be done.

3.2. Dynamic Documentation

The first step in learning how a system works is to know the classes, their capabilities and limitations. The next step is to understand how the objects interact at run-time. Especially frameworks rely heavily on complex object interaction, including calls to abstract methods where hotspots [Pre95] are located. In many cases, an extension of the framework requires detailed knowledge about the interior of a framework. In these situations, it is essential to know the involved classes, their state at a given time, and their behaviour.

To provide appropriate information on object interaction, *event trace diagrams* (Figure 2) where introduced in the OMT-Method [Rum91]. They display objects as vertical lines representing the timeline. Messages sent between objects are horizontal arrows. Event trace diagrams are frequently used in object-oriented literature to illustrate object interaction (e.g. [Gam95], [Jac94]). The major weakness of these diagrams is, that no information about the control flow is shown. To fix this problem, messages in UML *sequence diagrams* [Fow97] may contain *conditions* and *iteration markers* (Figure 3). Some UML tools (like Rational Rose™) use bars instead of dashed lines to represent the control scope of an object.

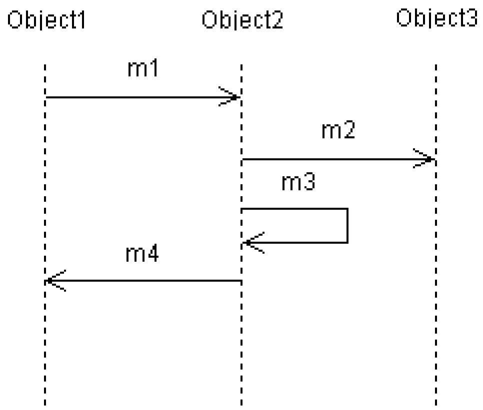

Figure 2: Event Trace Diagram in OMT

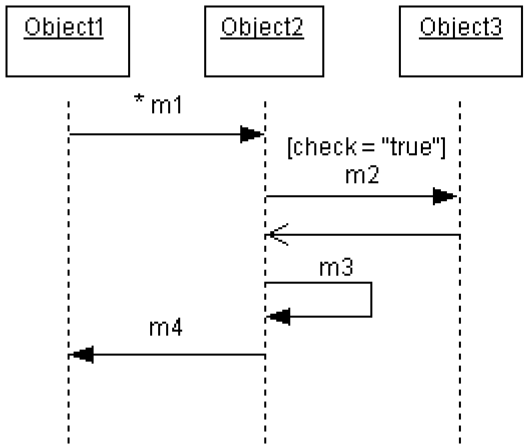

Figure 3: Sequence Diagram in UML

The diagram in Figure 3 shows that message `m1` is sent many times and that message `m2` is sent only if condition `[check = "true"]` holds. This information about the control flow is not present in Figure 2.

What is missing in event trace diagrams as well as in sequence diagrams, is information about the exact actions that take place in the method bodies. A possible question may be: ’what actions different from message sends are performed between the receipt of message `m1` in `Object2` and the sending of message `m2` to `Object3` ? ’.

Another difficulty is to show dynamically bound method calls.

```
Shape shp;
...
shp.draw();
...
```

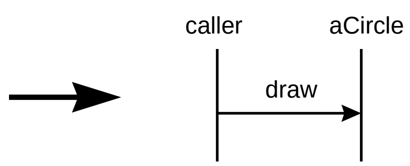

Figure 4: A Dynamically Bound Method Call

The scenario illustrated in Figure 4 may be part of some sort of graphics editor. `Shape` may be the superclass for several concrete shapes like circles, lines, or rectangles. An event trace diagram only shows one special case (called *scenario* in OMT and *use case* in UML), for examle the case when `shp` is of dynamic type `Circle`. The information that `shp` could also be some other subtype of `Shape` is lost.

Even with the UML extensions added, documentation of the control flow is still too coarse grained. Iteration markers do not show how many iterations are performed. Additionally it is almost impossible to show the following situations:

Repetition of message sequences:

```
while ... {
  obj1.m1();
  obj2.m2();
  obj3.m3();
}
```

Conditional message sequences:

```
if ... {
  obj1.m1();
  obj2.m2();
}else
  obj3.m3();
```

The only possibility in a sequence diagram is to separately place the same iteration marker or condition on all affected messages. This obscures the fact that the messages form a sequential block. `else` - branches need to be marked with the inverse condition of the `if` - statement.

The conclusion is that information about general behaviour like conditional messages and dynamically bound method calls cannot be packed into a diagram which is itself static and is therefore able to illustrate only one particular scenario. To visualize the fact that receiver objects of a dynamically bound method call may be of different static types, a diagram must be embedded into an interactive environment. In such an environment the user would switch between different receiver objects and explore several independent paths through the flow of control.

4. An Extension of Sequence Diagrams

One of the problems stated above is that we don't see what happens inside the method bodies. The objects are just represented by vertical bars or dashed lines. Our basic idea is therefore to widen the bar to a textbox to show the control flow inside. The contents of the box shows pseudocode at an appropriate abstraction level. At the one extreme it could be just a sequence of message calls, at the other extreme it could be almost as detailed as source code. Figure 5 shows the general layout of such a diagram compared to an equivalent sequence diagram.

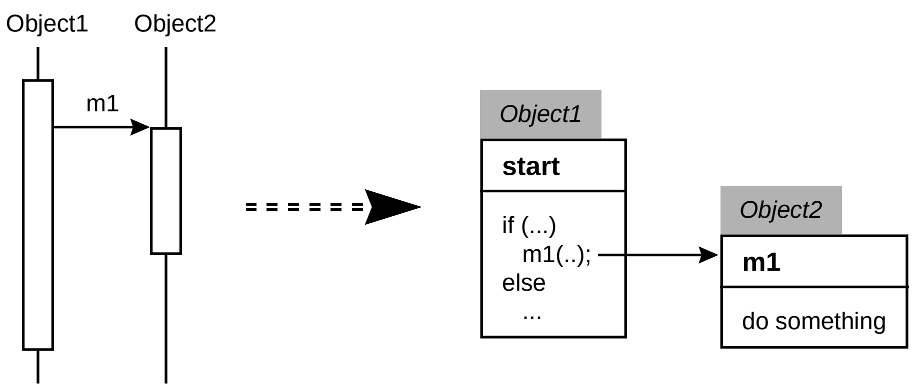

Figure 5: Sequence Diagrams Extended

To get a maximum level of usability and flexibility, hypertext with embedded text elements (e.g. links to other boxes) is used to display the information in a box. The text elements are similar to those used in the Oberon System [MK95]. The hypertext is highly extensible and completely free for editing - no special format or syntax is required by the system.

The most important fact cannot be illustrated using printed paper - such diagrams are interactive. Not all boxes are visible from the beginning. Method `start` contains a link that can be clicked with the mouse and pops up the referenced box (in Figure 5 this is `Object2`). A click on the link to an already visible box hides the box. This enables the user to explore parts of the diagram and hide paths that are no longer interesting.

4.1. Behaviour Diagrams

The author calls this extension of sequence diagrams *'Behaviour Diagrams'*, since such diagrams are showing not only the state of an object system at a certain time (just like a snapshot), but the general behaviour of such a system.

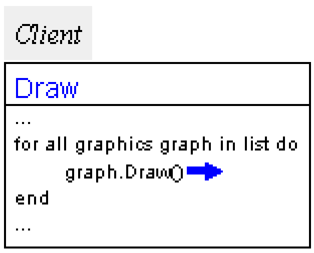

Figure 6: Initial Situation

Figure 6 illustrates a typical situation in a behaviour diagram. Only one starting object is visible (maybe the client object of some graphics editor). This object represents the root for all further investigation of the diagram. Multiple roots and different root types (method calls or user interaction) are possible. Clicking on the *LinkElem* pops up the referenced object (see Figure 7).

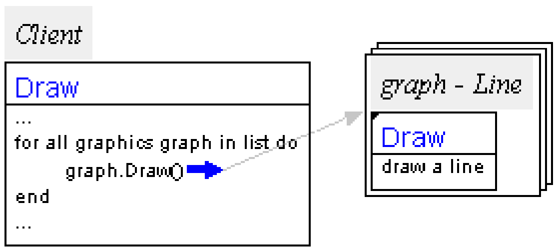

Figure 7: Possible Scenario: A Line is Drawn.

Figure 7 shows the scenario when a single line is drawn. The additional borders around object `graph` indicate that the `Line`-object is not the only scenario referenced by the link. In fact the `Line`-object is only the top of a set of objects implementing the interface of a common base class. Alternative objects are hidden behind the top object and can be switched to the foreground (see Figure 8).

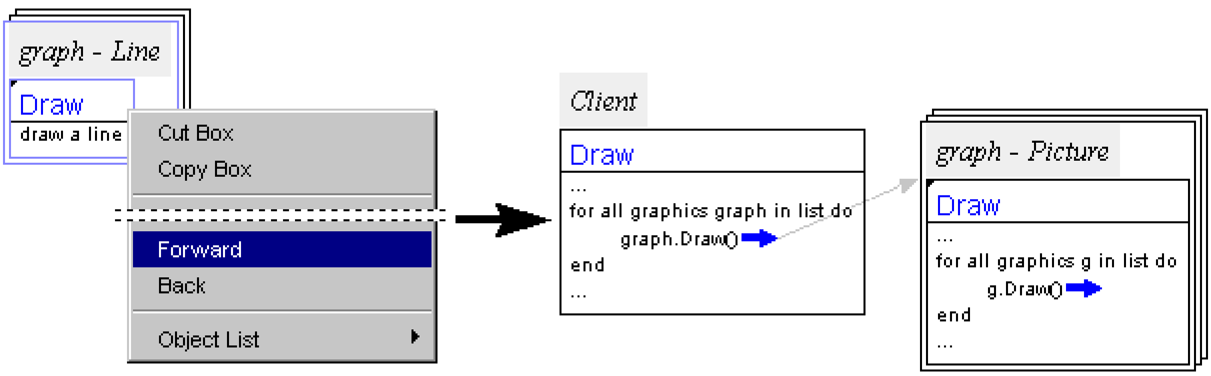

Figure 8: Switching an Object to the Foreground

Since the LinkElem in this example references more than one objects, it represents a dynamically bound method call. The user may select the call that is actually performed and therefore enters new paths in the diagram. All objects in a stack of alternatives must share a common base class and implement some common interface used by the client object which only works with the base class interface. To illustrate this fact, the designer of the behaviour diagram can mark methods of the common base class interface. Links to methods marked in such a way are handled appropriately, e.g. they stay open when switching to an alternative object, whereas links to methods special for certain objects are closed when switching to another object. Methods marked this way are called *common methods*. The method `Draw` in the `graph` object is such a common method. A special mark in the upper left corner of a method indicates that it is a common method.

4.2. Browsing in Behaviour Diagrams

Initially only the first object in the calling chain or the initiating user input is visible. Given this starting point, the user follows the links of his interest and doing so builds up a path through the diagram. The timeline which was fixed in sequence diagrams is replaced by a directed graph, which can be traversed by the user in the order of the control flow. Links that are not interesting for a particular user can be omitted. The irrelevant information stays invisible and therefore does not consume valuable space on the screen.

Lack of space on the screen is one of the main problems when using graphical notations. The fact that behaviour diagrams show additional information in the method bodies makes it difficult to place many boxes on the screen at the same time. To reduce consumtion of space, box positions are automatically chosen by the system, placing them near related boxes and avoiding overlapping boxes or arrows. Methods can be displayed in a collapsed state, where only the method header is visible (see Figure 9).

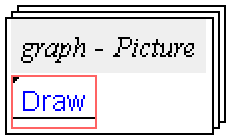

Figure 9: Collapsed *graph*

Considering these measures and especially the fact that there is only a small part of the complete diagram visible at the same time, behaviour diagrams remain small and easy to survey, even if they consist of lots of boxes. One should also take into account that the user focuses on a single method most of the time. When following a link, the user's focus moves to the new method similar to hypertext browsing where most of the time the actual document is in the center of the user's attention.

5. Conclusion

- Behaviour diagrams show what actions are performed inside the methods (the level of detail is chosen by the designer of the diagram). This enables the designer to illustrate repetitions of message sequences and conditional messages much better than in UML diagrams.

- Behaviour diagrams are interactive. They illustrate not only a snapshot of a certain situation but the general behaviour of a set of objects. This includes base class information of common methods and enables the designer to illustrate dynamically bound method calls. Additionally the user browses such a diagram in the way a hypertext would be browsed. Usually there is only one path of interesting objects visible, unnecessary information is hidden and cannot obfuscate the information actually needed.

- Behaviour Diagrams are easy to build and maintain. This is important since building documentation is a time consuming and unloved task which is often neglected by software engineers.

6. Bibliography

[Fow97] Fowler, Scott: *UML Distilled - Applying the Standard Object Modeling Language*, Addison-Wesley 1997.

[Fre96] French, Knight, Powell: *Hypertext Structures and Software Documentation*, Technical Report CS-96-04, Department of Computer Science, University of Virginia.

[Gam95] Gamma, Helm, Johnson, Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.

[Jac94] Jacobson: *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley 1994.

[Joh92] Johnson: *Documenting Frameworks using Patterns*, OOPSLA 1992.

[Pre95] Pree: *Active Guidance of Framework Development*, Software - Concepts and Tools 1995.

[Meu97] Meusel, Czarnecki, Köpf: *A Model for Structuring User Documentation of Object Oriented Frameworks Using Patterns and Hypertext*, ECOOP 1997.

[MK95] Mössenböck, Koskimies: *Active Text for Structuring and Understanding Source Code*, SOFTWARE - Practice and Experience, July 1996.

[Rum91] Rumbaugh, Blaha, Premerlani, Eddy, Lorensen: *Object-Oriented Modelling and Design*, Prentice-Hall 1991.