--FINAL VERSION--

Final version 3.4-- October 20, 1997 @ 16h39
To appear in the Communications of the ACM October'97 Issue on "Object-Oriented Frameworks"

Design Guidelines for Tailorable Frameworks

Serge Demeyer,[*] Theo Dirk Meijler,[†] Oscar Nierstrasz,[*] Patrick Steyaert[‡]

Since the early 1980s, object-oriented frameworks have demonstrated that programmers can encapsulate a reusable, tailorable software architecture as a collection of collaborating, extensible object classes. Such frameworks are particularly important for developing open systems in which not only functionality but architecture is reused across a family of related applications. Unfortunately, the design of frameworks remains an art rather than a science, because of the inherent conflict between *reuse* — packaging software components that can be reused in as many contexts as possible — and *tailorability* — designing software architectures easily adapted to target requirements.

To cope with this conflict, well-designed OO frameworks must provide a clean conceptual framework that clearly identifies *hot spots[6];* where tailorability is necessary and desirable and specifies *framework contracts[1]* that formalize exactly which parts of the framework are to be reused. This article presents three design guidelines that help identify hot spots and contracts, thereby balancing flexibility and tailorability.

Tailorable frameworks are particularly useful for the construction of so-called open systems. There exist various viewpoints on what precisely makes a system "open", a discussion beyond the scope of this article. For our purposes, the following open system requirements are sufficient.

- **Interoperability**: Open systems typically run on heterogeneous hardware and software platforms. Platform differences should be encapsulated in the system architecture to ease integration.
- **Distribution**: Open systems are physically distributed. Coordinating distributed services is nontrivial and requires reliable services. An open system guarantees the correct execution of system critical functions.
- **Extensibility**: Most open systems provide some form of extensibility, allowing end users to customize the system to address special needs. End-user customization requires the system's configuration to be adaptable without changing the internal implementation of existing system modules.

To cope with these requirements, we ask system designers to identify the *axes of variability* for their open system. Based on these axes, we specify the following three guidelines for designing an open system architecture (each design guideline introduces an extra level of tailorability for addressing an open system requirement).

Guideline 1 [Interoperability] Include in the design separate "axis-objects" so each such object represents a point on one of the axes of variability, thus encapsulating a degree of plat-

form independence. Objects in the initial system model must delegate responsibilities to the axis objects.

Guideline 2 [Distribution] Specify a framework contract for each of the variability axes. Extend the framework design with a "contract object" guaranteeing the correct execution of the corresponding contract.

Guideline 3 [Extensibility] Introduce a global "configuration object" representing the system configuration. By replacing that configuration object, an OO programmer can adapt the system's configuration without changing the implementation of the other objects.

To illustrate the practical value of the guidelines, the rest of this article explores their application to the concrete architecture of an open hypermedia system (see http://www.csdl.tamu.edu/ohs/). We stress that this is only an example and we refer readers to [2] for a discussion on the guidelines' general applicability. We also point out that such system is necessarily complicated, as it deals with difficult issues that cannot be illustrated with toy examples; we provide clarifying diagrams wherever possible.[1]

To apply the guidelines, we assume the existence of an initial model for the intended open system. Identification of the entities in such an initial model is another issue beyond the scope of this article but methodologies like Objectory's Use Cases[4] provide excellent support in this area.

Figure 1 shows the class structure of an initial open hypermedia system model, including a `Document` class, holding some `contents` in a certain multi-media format (e.g., HTML, GIF). A document also contains a number of `Anchors` representing the parts of a document that may be used as the source or target of a navigation operation. An anchor has a `value` that uniquely identifies it within the associated document contents (e.g., the position in a text, rectangle in a bitmap).

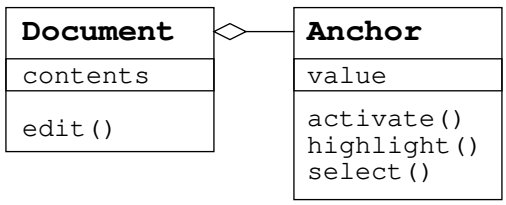

Figure 1 `Initial Model for an Open Hypermedia System`

1. All the diagrams employ the notation of the Unified Modelling Language (UML).

The `Document` and `Anchor` classes are two of the hot spots; tailoring the framework to the needs of the open hypermedia system, system designers must provide the appropriate subclasses. However, these subclasses must respect the fundamental rules of the framework, specified in the framework contract (see figure 2). Assuming the precondition of a displayed document whereby every contained anchor is highlighted, an activation of one of these anchors (`activate`) computes the target document and anchor of the corresponding navigation relationship, opens the document (`edit`), highlights the contained anchors (`highlight`) and finally selects (`select`) the target anchor within that a document.

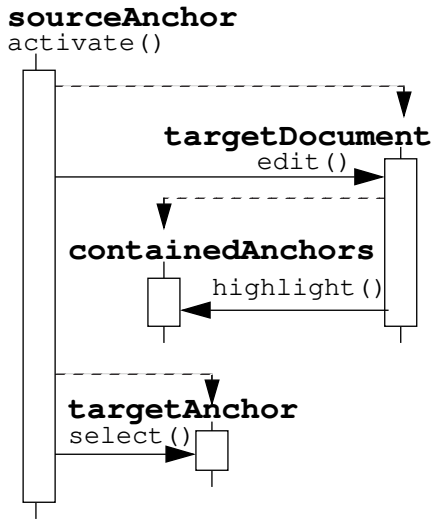

Figure 2 The initial framework contract

Axes of Variability

To turn an initial system model into an architecture for an open system, designers should first identify the axes of variability. How to do this is also beyond the scope of this article, so we again refer to well-known OO methods like Objectory [4] and OORAM [5].

For our open hypermedia system, we propose three important varying characteristics: *storage* (how hypermedia documents are stored — in, say, http, ftp or a file), *presentation* (how they are viewed — through, say a browser or JPEG viewer); and *navigation* (how they may be linked — by, say, embedded references, CGI-scripts). Based on this analysis, figure 3 shows the three variability axes: the *storage* axis, enumerating all possible document repositories; the *presentation* axis, enumerating all possible viewer applications; and the *navigation* axis enumerating all possible kinds of linking relationships.

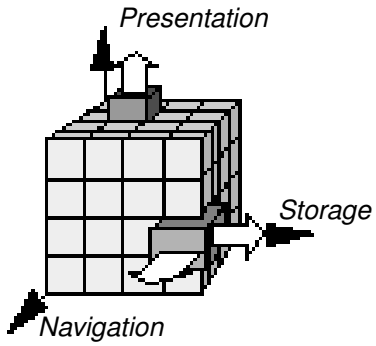

Figure 3 The hypermedia axes of variability

We now turn to the question how to incorporate these axes of variability into a tailorable framework architecture.

The Interoperability Requirement

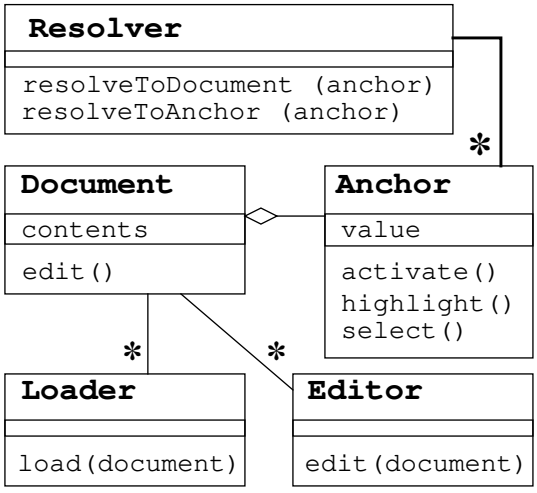

Figure 4 `Resolver`, `Editor` and `Loader` classes, representing the variability axes

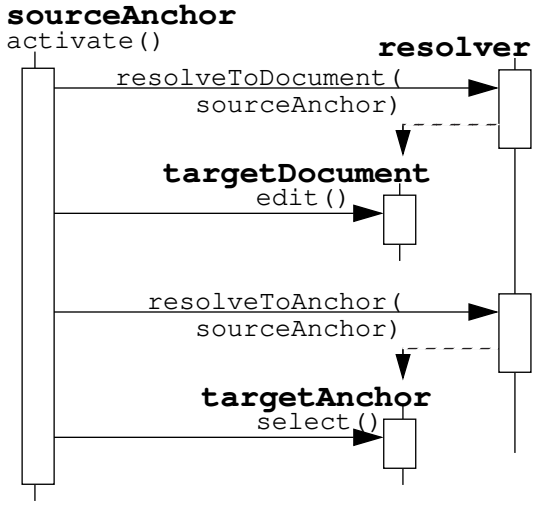

Figure 5 The Navigation contract

The application of the first guideline introduces three new classes (see figure 4): one for the storage axis (`Loader`), one for the presentation axis (`Editor`), and one for the navigation axis (`Resolver`). These extra classes yield three hot spots for the hypermedia framework. By introducing these hot spots, we must split the initial framework contract into three finer-grained contracts. The *navigation contract* (see figure 5) states that an anchor receiving the `activate` message, must send a `resolveToDocument` message to the associated resolver to create the target document and display it to the user (`edit`); afterward, the activated anchor must create (`resolveToAnchor`) and select (`select`) the target location. The *storage contract* states that a document receiving the `edit` message, must send a `load` message to the associated loader to ensure that the document contents are properly loaded from the storage device and that all associated anchors are created. The *presentation contract* states that a document receiving the `edit` message must — after being loaded — send an `edit` message to the associated editor to display the contents to the end user and highlight all the anchors.

Having applied the first design guideline, the document and anchor objects in the initial hypermedia model delegate the variant behaviour to the corresponding axis objects—`resolver`, `loader`, `editor`. This way, all platform-dependent aspects are encapsulated into the axis objects, thus addressing the interoperability requirement.

The Distribution Requirement

Reliability is crucial in a distributed system. Crucial system services —log maintenance, locking, authority control— require the system to monitor all activities of a certain kind and perform additional checks and bookkeeping. This requirement conflicts with the interoperability and extensibility requirements, according to which the system has to cooperate with external software and may be adapted at run time. Guaranteeing reliable services in such a dynamic environment is difficult, but our second guideline results in an architecture that monitors crucial system services, independent of the participating objects. We illustrate this need through the problem of maintaining a log of all navigation operations.

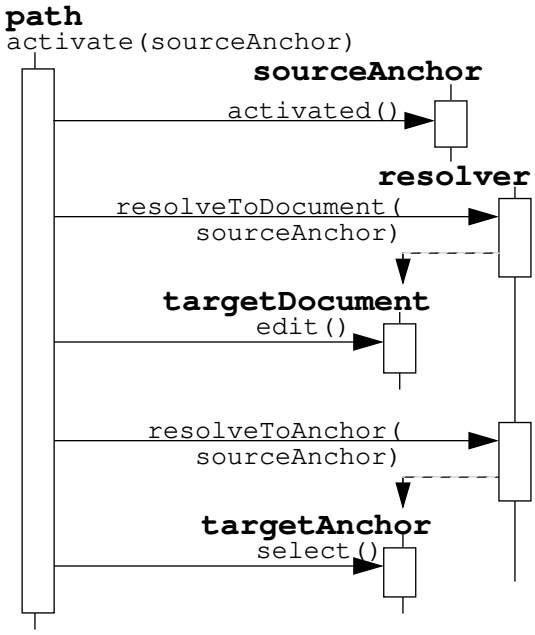

Figure 6 The `path` object, representing the navigation contract

Figure 6 shows the result of applying the second guideline for the navigation contract in figure 5. Here, the `path` object takes complete control of the execution of the navigation contract, including an extra notification (`activated`) of the source anchor. This `path` object is a hot spot of the framework, providing an ideal location for wrapping additional logging behaviour around the execution of the navigation operation, independent of the participating `resolver`, `anchor` and `document` objects. We similarly reify the storage contract and guarantee that all read/write operations are monitored; such hot spot can be used to guarantee a systemwide locking strategy. Also, the reified presentation contract can monitor all editing operations and implement authority control. We conclude that applying the second guideline allows us to monitor execution of system-critical services, addressing the need for reliability in distributed systems.

The Extensibility Requirement

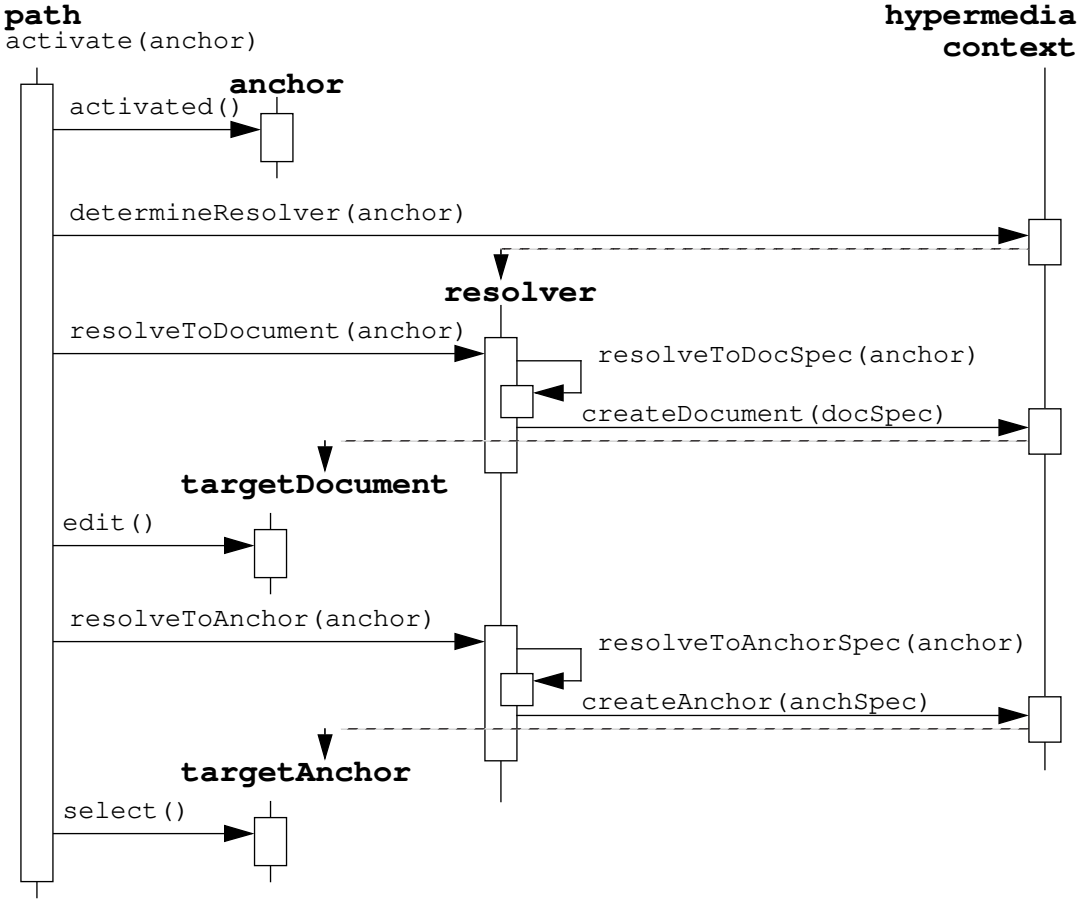

Figure 7 `hypermediacontext` object, representing the system configuration

Rephrasing the third guideline, we must refactor all operations that determine the system configuration into a single configuration object. In a system designed according to our guidelines, these are all methods that create objects[1] from the initial model (see figure 1) plus all the methods that make a connection between an object of the initial model and a contract object (the first guideline). Figure 7 shows the result of applying the third guideline for the navigation contract. When the `path` object receives the request for a navigation operation (`activate`), it asks the `hypermediacontext` object to identify which `resolver` is supposed to handle the navigation operation (`determineResolver`). This `resolver` is then asked to compute the targets of the navigation operation (`resolveToDocSpec/resolveToAnchorSpec`), returning a document and anchor specifier. Each of these specifiers is passed to the `hypermediacontext` object, which instantiates one of the existing `Document` (`createDocument`) or `Anchor` (`createAnchor`) classes. Similarly, there are two methods that decide on the `loader` (`determineLoader`) or `editor` (`determineEditor`) that is supposed to handle a document.

1. To that extent, the configuration object behaves like an Abstract Factory [3]

This single hypermedia context object provides the framework hot spot in which an OO programmer can tailor the system configuration without altering the rest of the system. Thus, the third guideline addresses the extensibility requirement.

Conclusion

The proposed design guidelines cover only some of the state of the art in framework design. But because of the way they are formulated, they fit nicely with the other techniques available today —i.e., design patterns, open implementations, class refactoring—making them especially attractive.

The fact that the guidelines provide concrete design solutions for such issues as interoperability, extensibility and distribution makes them useful for coping with the growing demand for openness. The search for more openness is inevitable in an environment in which software is evolving dramatically and the World-Wide Web's growing popularity means new technical requirements every day. Since they build on our hypermedia experience, we are confident that these guidelines, will address the needs of current and future generations of open systems.

References

[1] Codenie, W., De Hondt, K., Steyaert, P., and Vercammen, A., Evolving Custom-Made Applications into Domain-Specific Frameworks. *Communications of the ACM 40, 10* (October 1997).

[2] Demeyer, S. *Zypher: Tailorability as a Link from Object-Oriented Software Engineering to Open Hypermedia*. Ph. D. dissertation, Vrije Universiteit Brussel, Department of Computer Science, 1996 — Belgium.
See http://dinfwww.vub.ac.be/ and http://iamwww.unibe.ch/~demeyer/.

[3] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*. Addison-Wesley, 1995.

[4] Jacobson, I. *Object-Oriented Software Engineering, a Use Case Driven Approach*. Addison-Wesley, 1992.

[5] Reenskaug, T. and Wold, P. and Lehne, O. A. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, Greenwich CT, 1996.

[6] Schmid, H. A., Systematic Framework Design by Generalisation. *Communications of the ACM 40, 10* (October 1997).

———————————————

(*) Serge Demeyer `demeyer@iam.unibe.ch` is a research assistant in the Software Composition Group at the University of Berne in Switzerland.

(†) Theo Dirk Meijler `tdmeijler@research.baan.nl` is a researcher in the research department of the Baan Development company in Ede, The Netherlands.

(*) Oscar Nierstrasz `oscar@iam.unibe.ch` is a professor heading the Software Composition Group at the University of Berne in Switzerland.

(‡) Patrick Steyaert `prsteyae@vub.ac.be` is a research assistant in the Programmin Technology Lab at the "Vrije Universiteit Brussel" in Brussels, Belgium.