Object-Oriented Architectures for Electronic Commerce

Concepts and Development Heuristics

Wolfgang Pree

C. Doppler Laboratory for Software Engineering
Johannes Kepler University Linz, A-4040 Linz, Austria
Voice: +43 70-2468-9432; Fax: +43 70-2468-9430
E-mail: pree@swe.uni-linz.ac.at
http://www.swe.uni-linz.ac.at/wolf

Abstract. The Web is exciting because of its potential to be ubiquitous soon. So electronic commerce will be tightly coupled to this medium and people will buy goods over the Internet instead of going to stores. The precondition for electronic commerce is that existing information systems are enhanced to support the Web. Another exciting opportunity will be the distribution of software components over the Internet. Though software components comprise just one specific product, this will have a profound impact on how software systems wil be built in the future. The paper sketches the key role of semifinished object-oriented architectures (= frameworks) for both, the enhancement of existing information systems and plug-and-play componentware. It includes a discussion of the concepts underlying framework technology and of some design heuristics known so far.

Keywords. electronic commerce, Web, frameworks, software architectures, component-based software development, design patterns, software reuse, Java, hot spots

1 Electronic commerce over the Web

Steve Jobs summarized the potential of the Web in a recent interview in Wired (Jobs, 1996), asking the interviewer: “Who do you think will be the main beneficiary of the Web? Who wins the most?” Jobs answers his question and states that those people win the most “who have something to sell! It is more than publishing. It is commerce.” He continues to sketch the advantage of the Web for businesses. It acts as a democratizer where even very small companies cannot be discerned from big ones. If the Web becomes ubiquitous, and this is what he believes it is going to be, it neutralizes the advantage of big companies which have invested lots of money in creating distribution channels.

Jobs forecasts the impact of electronic commerce over the Web, estimating that significantly more than 10 percent of the goods and services in the U.S. will be sold at this marketplace. His conclusion is that “eventually, it will become a huge part of the economy.”

The next sections present the concepts of frameworks and outline how framework technology can help to Web-enable existing information systems. The role of frameworks is put into the context of future development stages of the Web.

2 Framework concepts

Object-oriented programming languages are used in many software projects in a manner similar to module-oriented languages with classes as a means for implementing abstract data types. Inheritance helps to adapt and thus reuse building blocks that do not exactly match the requirements. So adaptations are possible without having to change the source code.

In order to develop the full potential of object-oriented software construction in the realm of constructing reusable architectures, an *appropriate combination* of the basic object-oriented concepts (i.e., object/class definition, inheritance in connection with polymorphism and dynamic binding) is necessary.

The key idea behind this approach is to find good abstractions of concrete classes. *Abstract classes* as discussed below represent such abstractions. They form the basis of frameworks, which are reusable application skeletons. Frameworks represent the highest level of reusability known today, made possible by object-oriented concepts: Not only source code but architecture design—which we consider as the most important characteristic of frameworks—is reused in applications built on top of a framework. Overall, frameworks enable a degree of software reusability that can significantly improve software quality.

Abstract classes. The general idea behind abstract classes is clear and straightforward:

- Properties (that is, instance variables and methods) of similar classes are defined in a common superclass.
- Classes that define common behavior usually do not represent instantiable classes but abstractions of them. This is why they are called *abstract classes*.
- Some methods of the resulting abstract class can be implemented, while only dummy or preliminary implementations can be provided for others. Though some methods cannot be implemented, their names and parameters are specified since descendants cannot change the method interface. So an abstract class creates a *standard class interface* for all descendants. Instances of all descendants of an abstract class will understand at least all messages that are defined in the abstract class.

  Sometimes the term *contract* is used for this standardization property: instances of descendants of a class A support the same contract as supported by instances of A.
- It does not make sense to generate instances of abstract classes since some methods have empty/dummy implementations.

The implication of abstract classes is that other software components based on them can be implemented. These components rely on the contract supported by the abstract classes. In the implementation of these components, reference variables that have the static type of the abstract classes they rely on are used. Nevertheless, such components work with instances of descendants of the abstract classes by means of polymorphism. Due to dynamic binding, such instances can bring in their own specific behavior.

The key problem is to find useful abstractions so that software components can be implemented without knowing the specific details of concrete objects.

Frameworks. Abstract classes form the basis of a framework. If abstract classes factor out enough common behavior, other components, that is, concrete classes or other abstract classes, can be implemented based on the contracts offered by the abstract classes. A set of such abstract and concrete classes is called a *framework.*

The term *application framework* is used if this set of abstract and concrete classes comprises a generic software system for an application domain. Applications based on such an application framework are built by customizing its abstract and concrete classes.

In general, a given framework anticipates much of a software system's design. This design is reused by all software systems built with the framework.

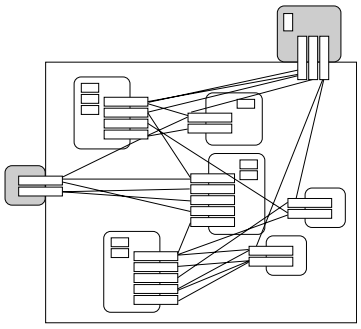

Figure 1 Framework with flexible hot spots.

In other words, a framework defines a *high-level language* with which applications within a domain are created through *specialization* (= adaptation). Specialization takes place at points of predefined refinement that we call *hot spots*. Figure 1 illustrates this property of frameworks with the flexible hot spots in gray color. The overall framework comprises the standardized, i.e., frozen, domain aspects intertwined with the hot spots.

3 Frameworks as enabling technology for electronic commerce

Let us first gaze into the future in our crystal ball and take into consideration future development stages of the Web that will have an impact on electronic commerce. Steinberg (1996) envisions three further major stages of the Web beyond the current situation where almost all documents are simply downloaded from servers to clients and displayed on the client side:

(1) First, clients and servers will become smarter. So end users will easily be able to access information systems. This marks the beginning of electronic commerce. The fun-to-use package tracking system of FedEx gives a taste of what will happen at this stage. The internal FedEx information system is connected to the Web as follows: Customers who send packages via FedEx receive a package code. This code number can be entered into a Web page offered by FedEx. The Web page represents the front end to the internal information system. So the customer can easily keep track of where the package is currently in the world and if it is delivered on time.

(2) Second, not just data but programs will be exchanged between servers and clients. Steinberg (1996) sketches an example: "A stockbroker's Web site might send out an applet [little application] that acts as front end for displaying a ticker tape at the top of your screen."

(3) In the third stage, applets will turn into intelligent agents that are sent out like servants and gather information.

In all these stages framework technology will play a key role. In order to make clients and servers smarter, a huge amount of software has to be written. Only appropriate frameworks that significantly reduce the amount of code that has to be written will allow software developers to cope with these demands. The first frameworks for this purpose are NeXT's WebObjects for the server side and Apple's Cyberdog for the client side. These frameworks are described below.

Applets are examples of software components that configure themselves automatically on the end user's client. Again, frameworks will be the core technology underlying such applets. As Sun's Java language, Microsoft's Internet Studio (originally code-named Blackbird) and General Magic's Telescript are specifically designed for building applets, various frameworks will be based on them in the future.

Intelligent agents are quite similar to applets, but more active. Today most of the agents are built from scratch in research labs. Thus when agents become the vogue on the Web, we can expect appropriate agent frameworks.

Frameworks for the first stage. We pick out NeXT's WebObjects as representative framework for making information systems accessible over the Web. This framework preimplements a generic software system that allows a Web server to access information in internal databases. WebObjects transforms this information into HTML documents. For example, a bookstore could have its books in a large Informix database. A WebObject allows to submit a search request from the Web server to this database. The matching records are returned and repackaged in an HTML document which is then displayed on the Web.

The framework significantly speeds up the development process of such more complex Web sites. For example, it took FedEx four month (Jobs, 1996) without an appropriate framework to connect its information system to the Web, which is a long time for a simple feature.

Even more challenging is real electronic commerce. If customers should not only be able to browse through the goods offered by a company, the internal order-management system and collection system have also to be connecetd to the Web. Again appropriate frameworks that help to tie up the Web with database systems greatly reduce development efforts.

Apple's Cyberdog focuses on the client side. Currently one has to use special purpose Web browsing tools such as Netscape's Navigator. Apple's Cyberdog comprises a framework that Web-enables every application. The end user can define active, embedded Web-links into any type of document. Apple exemplifies the power of this concept: A report on earthquakes might have an embedded Web-link that displays up-to-date seismographic data.

Frameworks for the second stage. Let us pick out Microsoft's Internet Studio and Sun's Java to illustrate how framework technology supports this level of Web animation. Internet Studio allows small programs to be sent from the server to the client where they are executed. For example, a program to display video data could be sent along with the data from the server to the client. The program immediately starts running on the client and shows the movie while the data are downloaded. The advantage is that the Web-browser is always up-to-date. Internet Studio is based on Microsoft's proprietary standard OLE to integrate different types of objects, such as spreadsheets, text documents, and drawings. The small programs (=applets) are socalled OLE controls. (Remark: Apple's Cyberdog is based on OpenDoc.)

The problem is that Internet Studio does not include security features. So a virus can also be downloaded as applet. Sun's Java language and environment offer a better solution for several reasons: Java does not allow the programmer to manipulate arbitrary memory locations. It is platform-independent (OLE is currently restricted to Windows) and Java applets cannot only be sent from the server to the client but also vice versa. So Java's class library (in particular, the user interface part AWT and classes for programming in a networked environment) together with the run-time environment comprise a framework for writing applets.

Applets can be useful for electronic commerce in various situations. For example, a bank could offer calculator applets to allow its customers compare the different banking products.

Frameworks for the third stage. Applets that travel the Internet and filter information are often referd to as agents. With regard to electronic commerce, agents could help customers compare numerous offers and find the best ones. The idea of having personalized news papers could also become reality through agents.

General Magic has designed Telescript as language and environment to support agent programming. The main difference between Telescript and Java is that Telescript applets can store data in items on a computer even when they leave it. This is not possible in Java. On the other hand, Telescript is not (yet) an open standard.

Componentware. So far framework technology was discussed as means to easily connect information systems to the Web and to animate Web pages. This helps to sell any product electronically. The Internet could also imply a breakthrough for selling one specific product, namely software. Instead of selling current monolithic applications, a component market could arise. Frameworks are also the long-term player in this area: As discussed in Section 2, well-designed frameworks predefine most of the overall architecture, that is, the composition and interaction of its components. (Well-designed means that a framework offers the domain-specific hot spots for adaptations.) So applications built on top of a framework reuse not only source code but also architecture design—which we consider as one of the most important characteristics of frameworks.

Many existing frameworks give an impressive example of the degree of reusability that can be achieved if these systems are well-designed. For example, GUI frameworks with excellent design (see, for example, Lewis et al. 1996) deliver a reduction in source code size (that is, the source code that has to be written by the programmer who adapts the framework) of *80% percent or more* (Weinand et al., 1989) compared to software written with the support of a conventional graphic toolbox.

The more advanced framework designs available today already point out possible directions in which future frameworks will go. These pioneering frameworks rely mainly on *object composition*. In such frameworks most of the adaptations are done by just plugging together objects instead of modifying behavior by means of inheritance. To those who have already experienced the ease and power of such adaptations, it is obvious that future frameworks will rely mainly on composition. The underlying concepts are described, for example, in Gamma *et al.* (1995) and Pree (1996). We cannot deny the inevitability of this transition. Several implications result from this trend:

- Componentware will indeed mean distributing software components that can be plugged into software systems. The underlying technology will be frameworks whose behavior is modified and/or extended by composition.

  Of course, the world-wide network infrastructure will strongly boost such a component market.

  Note that in the long term this trend remains quite independent of the answer to the question of which of the current or future defacto standards (OLE/COM, Corba/OpenDoc) for integrating (distributed) components will dominate.
- Tools will become available that allow end users to configure software systems by handling such framework components.
- Currently software components are quite monolithic. In many cases components represent full-fledged applications. Expect a much finer level of granularity of software components.

The subsequent section discusses some heuristics how to design frameworks independent of a particular domain.

4 Hot-spot-driven framework design

A framework-centered software development process comprises the creation and reuse of frameworks. What we call *hot-spot-driven approach* guides developers in the process of systematically incorporating experience captured and expressed in design patterns (Gamma *et al.* 1995, Pree 1995, 1996).

Successful framework development requires the explicit identification of domain-specific hot spots. The various aspects of a framework that cannot be anticipated for all adaptations have to be implemented in a generic way. As a consequence, domain experts have to be asked:

- Where is flexibility required? Which aspects differ from application to application in this domain? A list of hot spots should result from this analysis.
- What is the desired degree of flexibility of these hot spots, i.e., must the flexible behavior be changeable at run time?

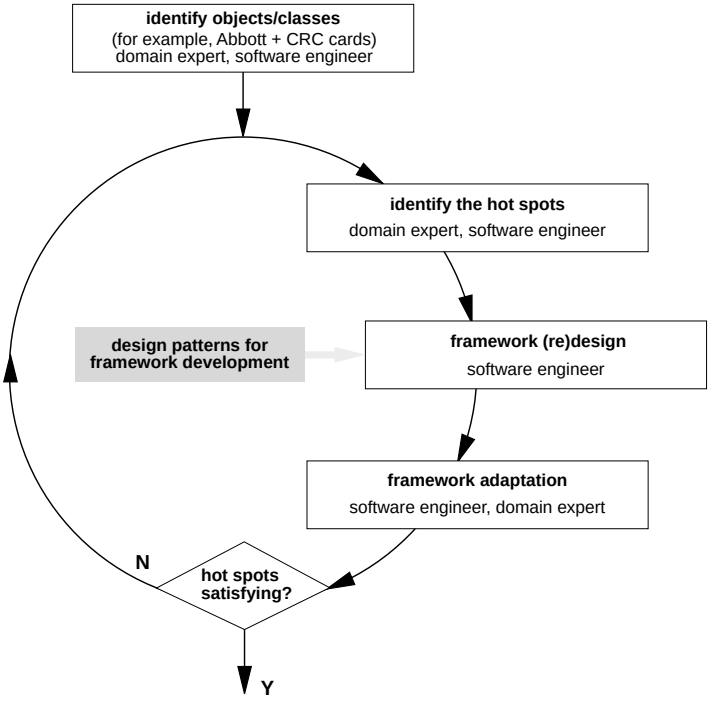

Figure 2 Hot-spot-driven approach.

Figure 2 schematically depicts the hot-spot-driven approach. State-of-the-art OOAD methodologies, such as those proposed by Booch (1994), Coad and Yourdon (1990), Jacobson (1993), and Wirfs-Brock *et al.* (1990), support the initial identification of objects/classes and thus a modularization of the overall software system. This initial step primarily requires domain specific knowledge. Software engineers assist in this activity. Of course, this first step is already an iterative one where object models have to be refined until they meet the domain-specific requirements.

Hot spot identification. Current OOAD methodologies neglect the importance of identifying hot spots as a basis for framework development. The schematically depicted framework development process in Figure 2 starts with the identification of hot spots. Again, the knowledge of domain experts is considered to be the most valuable source in order to successfully complete this step. Software engineers might guide domain experts

so that they describe hot spots in an adequate manner. As stated above, typical questions domain experts should be asked are: Which aspects differ from application to application? What is the desired degree of flexibility? Must the flexible behavior exist at run time? The software engineer can generate from the perspective of his current design more specific questions using the following rules:

- Having identified a class, find out from the domain expert how and whether objects would differ from application to application (whether there are subclasses).
- For a collection of type T ask whether it would be possible to have elements of various kinds in it. These may be subtypes of T.
- For a method defined for type T, find out for each subtype the dependencies of the method.

Domain experts might either identify *functions or data* as hot spots.

For example, if a framework for rental software systems should be developed, that can easily be customized for hotels, car rental companies, etc., a domain expert would identify the rate calculation aspect as a typical function hot spot in this domain. Rate calculation in the realm of a hotel has to encounter the room rate, telephone calls, and other extra services. In a car rental system different aspects will be relevant for rate calculation. Overall, the idea is to produce a reservation system that can be adapted to specific needs by plugging in specific rate calculation components.

The domain expert also has to assess the required flexibility of an identified hot spot, i.e., whether the hot spot behavior has to be adaptable at run time. For example, if a rental system framework should be targeted at rental companies with a world-wide operation, run-time adaptability of the rate calculation is important. Every stand-still of the system costs lots of money.

Framework (re)design. After domain experts have initially identified and documented the hot spots, software engineers have to modify the object model in order to gain the desired hot spot flexibility. In this step design patterns for framework-centered software development as discussed in Gamma *et al.* (1995), and Pree (1995, 1996), assist the software engineer (see Figure 2). Pree (1996) describes how frameworks can be constructed in a systematic manner once the hot spots have been identified. For example, run-time flexibility of a system function implies that this functionality goes into an abstract class. In general, more flexibility makes the class/object model more complex. So flexibility has to be injected in the right doses. Design patterns just describe how to make object-oriented architectures more flexible. This might be dangerous as it leads to unnecessary complex solutions. Thus we consider hot spot identification as a precondition in order to exploit the potential of design pattern approaches. We conclude with a summary of heuristics how to identify hot spots.

Hints for detecting hot spots. In practice, most domain experts are absolutely not used to answering questions regarding a generic solution. The current project culture forces them to do requirements analysis and system specifications that match exactly one system. Vague or generic statements are not welcome. Below we outline ways to overcome this obstacle.

- *Take a look at maintainance.* Most of the software systems do not break new ground. Many software producers even develop software exclusively in a particular domain. The cause for major development efforts that start from scratch comes from the current system which has become hopelessly outdated. In most cases the current system is a legacy system, or, as Adele Goldberg (1995) expresses it, a *millstone*: you want to put it away, but you cannot as you cannot live without.

  As a consequence, companies try the development of a new system in parallel to coping with the legacy system. This offers the chance to learn from the maintainance

problems of the legacy system. If you ask domain experts and/or the software engineering crew where most of the effort was put into maintaining the old system, you'll get a lot of useful flexibility requirements. These aspects should become hot spots in the system under development.

Often, a brief look at software projects where costs became outrageuos in the past, is a good starting point for such a hot spot detection activity.

- *Investigate scenarios/use cases*. Use cases (Jacobson, 1993, 1995), also called scenarios, turned out to be an excellent communication vehicle between domain experts and software engineers in the realm of object-oriented software development.

  They can also become a source of hot spots: Take the functions incorporated in use cases one by one and ask domain experts regarding the flexibility requirements. If you have numerous use cases, you'll detect probably commonalities. Describe the differences between these use cases in terms of hot spots.

- *Ask the right people.* This last advice might sound too trivial. Nevertheless, try the following: Judge people regarding their abstraction capabilities. Many people get lost in a sea of details. Only a few are gifted to see the big picture and abstract from irrelevant details. This capability shows off in many real-life situations. Just watch and pick out these people. Such abstraction-oriented people can help enormously in hot spot detection and thus in the process of defining generic software architectures.

5 Outlook

Though object/framework technology is touted as yet another one and only path to true knowledge, it simply is not. Object/framework technology tries to unify the lessons learned in programming methodology over the past 30 years. Nevertheless, too many problems are still encountered. For example, if the interface of (abstract) framework classes is changed applications built on top of the framework are rippled. This is commonly known as *fragile base class problem*.

Furthermore, tools for better understanding frameworks and for testing components added to a framework would be crucial. Currently such tools are not adequate or even not existing.

Nevertheless, we think that these obstacles can be removed and that framework technology will be the *enabling technology that underlies future software systems*. This is already true for software that helps to connect information systems to the Web, thus forming the basis of electronic commerce. In essence, component-based software development might have the meaning then that companies develop components that specialize frameworks. As frameworks are well suited for any domain where numerous similar applications are built from scratch again and again, they will exist for numerous domains ranging from commercial and technical software systems to advanced applications such as intelligent agents. In many cases, frameworks will be adapted by just plugging in various components, distributed over the Web, so that end users are able to do configuration jobs.

For sure, framework development requires a radical departure from today's project culture. Framework development does not result in a short-term profit. On the contrary, frameworks represent a long-term investment. The proposed hot-spot-driven approach is aimed at exploiting the potential of design patterns for framework development. Experience has proven that the explicit identification of hot spots together with a systematic transformation of the corresponding domain object model contribute to finding appropriate methods and abstractions faster so that the number of redesign iteration cycles is reduced.

References

Booch G. (1994). *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummings

Coad P. and Yourdon E. (1990). *Object-Oriented Analysis*. Englewood Cliffs, NJ: Yourdon Press

Gamma E., Helm R., Johnson R. and Vlissides J. (1995). *Design Patterns—Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley

Goldberg A. (1995). *What Should We Learn? What Should We Teach?* Keynote speech at OOPSLA'95 (Austin, Texas); video tape by University Video Communications (http://www.uvc.com), Stanford, California

Jacobson I., Christerson M., Jonsson P. and Overgaard G. (1993). *Object-Oriented Software Engineering*. Wokingham: Addison-Wesley/ACM Press

Jacobson I., Ericsson M. and Jacobson A. (1995). *The Object Advantage*. Wokingham: Addison-Wesley/ACM Press

Jobs S. (1996) *The Next Insanely Great Thing!*, Wired, 4.02

Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J., Schmucker K. (1995) *Object-Oriented Application Frameworks*. Manning Publications, Prentice Hall

Pree W. (1995). *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley/ACM Press

Pree W. (1996). *Framework Patterns*. New York City: SIGS Books.

Pree W. and Sikora H. (1996). *Application of Design Patterns in Commercial Domains*. Tutorial at the OOPSLA'96, ECOOP'96 and TOOLS USA '96 Conferences.

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall

Steinberg S. (1996) *Get Ready for Web Objects*, Wired, 4.02

Wirfs-Brock R., Wilkerson B. and Wiener L. (1990). *Designing Object-Oriented* Software. Englewood Cliffs, NJ: Prentice-Hall

Weinand A., Gamma E. and Marty R. (1989). Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, **10**(2), Springer Verlag