

Software Architectural Evolution in the Dragon Project

Michael Christensen, Christian Heide Damm, Klaus Marius Hansen, Elmer Sandvad & Michael Thomsen

Department of Computer Science, University of Aarhus, Aabogade 34, DK-8200 Aarhus C, Denmark.
{toby, damm, marius, ess, miksen}@daimi.au.dk

Abstract

Our experience from the first two major phases of a software development project suggests that explicit focus on software architecture in these phases was an important key to success. More specifically: Demands for stability, flexibility and proper work organisation in an initial prototyping phase of a project are facilitated by having an explicit architecture, which, however should also allow for certain degrees of freedom for experimentation. Furthermore, in a following evolutionary development phase, architectural redesign is necessary and should be firmly based on experience gained from working within the prototype architecture. Finally, to get it right, the architecture needs to be prototyped, or iterated upon, throughout evolutionary development cycles. In this architectural prototyping process, we address the difficult issue of identifying and evolving functional components in the architecture and point to an architectural strategy: a set of architectures, their context and evolution, which was helpful in this respect.

1. Introduction

In the last couple of years the authors of this experience paper have been involved in a large project, that, among others, used an iterative and evolutionary approach to development. The project involved a group of university researchers and a global container shipping company and spanned a 14-month development period. During this period, the university team initially went through a number of iterations on a prototype of a global customer service system. When the prototype was approved, the development continued in a number of development cycles in which the prototype was extended both horizontally, to include all the important areas of business, and vertically, to contain a more substantial functionality. The research group consisted of one ethnographer, one cooperative designer, and six OO-developers. For a more thorough discussion of the process and its multi-perspective application development character see Christensen et al. (1998).

This paper focuses on the software engineering process, more specifically on software architecture. In our experience an explicit focus on the software architecture of the system that was being built, can be seen as one of the key explanations as to how we, technically, were able to accomplish the task set out in this project. We will try to reflect on our concrete experience and present two central lessons:

An **explicit architecture** is essential; even in initial prototyping cycles. It is also important in order to provide a well-defined structure within which, e.g., functionality, user-interface, and a problem domain model can be created and recreated when evolving a prototype in response to new requirements. It is important as it provides the necessary flexibility to experiment with alternative solutions in areas that are not well understood and it can also facilitate parallel work thereby allowing for faster development.

It is rarely possible to design the final architecture of the system during the initial phase of the development. We will, instead, argue that **architectural evolution** is necessary, partly, as the requirements change over a longer period of time, and, partly, as the developers' understanding of both the problem domain, and the technical ways of realising the system, supporting this problem domain, increases. In this way, not only the system itself but also the software architecture can be said to be prototyped.

2. Software Architecture

When referring to software architecture we refer to a system's *significant software components, their structure, and their relationships*. To describe the software architecture of the system discussed in this article and to overcome meaningless box-and-line diagrams we use a slight variation of the notation described in Bass et al. (1998). Generally, solid lines denote control and processing, whereas dashed lines denote data. For our purposes, we have added the symbol "Emerging structure" to be able to describe the process of architectural creation and evolution.

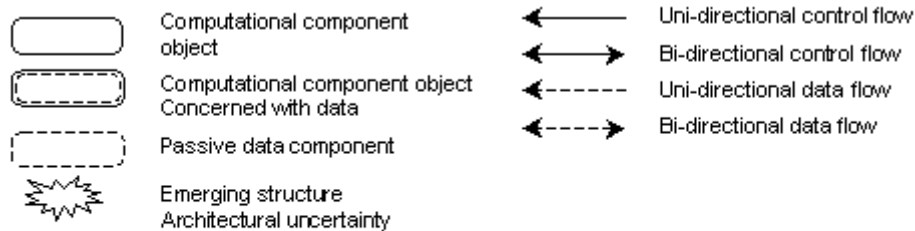


Figure 1. Architecture notation

2.1 Evolution and Architectural Refactoring

We make a clear distinction between the terms 'refactoring' and 'architectural refactoring'. Whereas refactoring is considered with semantic-preserving transformations of objects and classes (Opdyke, 1992), architectural refactoring is considered with function-preserving transformations of architectural structures. This means that a software system, after an architectural refactoring, will compute the same functions with respect to some problem domain. Pushing it a little, the reasons for using refactoring are concerned with code-technical transformations, whereas architectural refactoring needs to be seen in a broader software engineering perspective: technical as well as social considerations must be taken into account in order to understand the reasons for doing this.

Evolution, as opposed to architectural refactoring, is concerned with change in functions. Evolution of functions presupposes both flexibility and stable structures. The flexibility provides the ability to change within the stable structure, while the stable structure at the same time helps in achieving important non-functional requirements.

3. Phase One: Experimental Prototyping

The development was scheduled to begin with a two-week cycle. One group worked on creating graphical user-interface screens, and another group worked on creating an initial problem domain model.

At the end of the first cycle, two artefacts were delivered: A first design of the graphical user-interface, comprising of a few screens, and a problem domain model covering a subset of the problem domain (called 'quoting'). The user-interface design was illustrated by means of a horizontal prototype, i.e. a running application with virtually no functionality, except for functioning user-interface controls. The problem domain model was presented as a UML class diagram, and was, at this point in time, not a part of the running prototype. Nevertheless, it was already more than just a diagram: by using a CASE tool, code had been generated and updated incrementally, as elements were added to the model (Christensen et al., 1996).

After the first review the initial architecture of the prototype was outlined by two senior developers under consideration of several factors:

- the architecture had to offer a fairly stable structure, in which the prototype could evolve during the first phase,
- the structure had to be flexible enough to allow for a high degree of experimentation within rapid development cycles,

- it should support an efficient work organisation, allowing all developers to work intensively on the prototype in parallel.

Figure 2 illustrates the initial architecture. The figure uses the notation introduced in the architecture section and is drawn based on a "conceptual view": The figure illustrates the conceptual components - the major components as perceived by the developers - in the architecture, and the data and control flow between them.

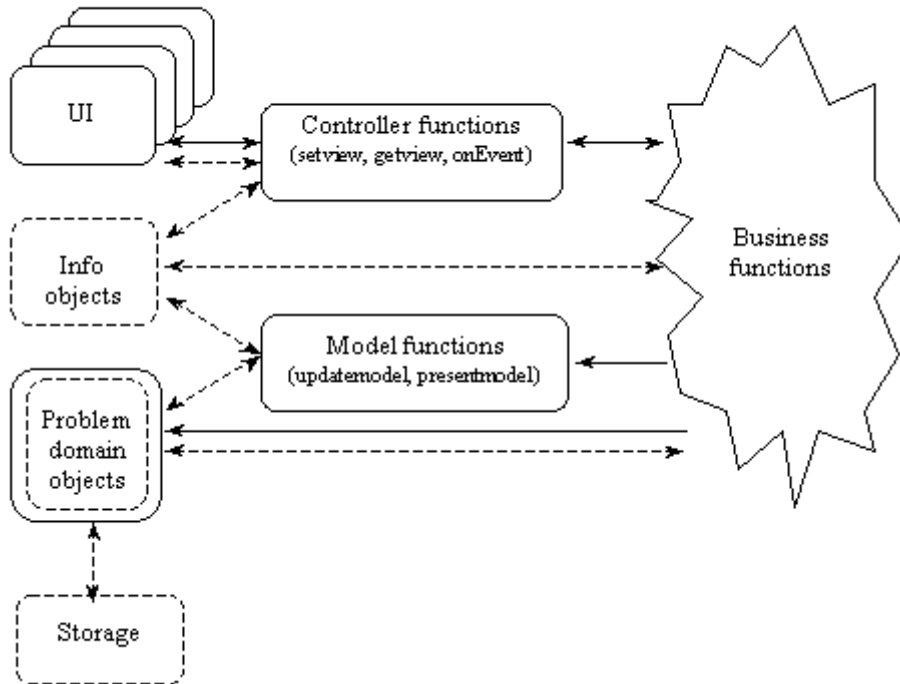


Figure 2. Initial architecture

The user-interface components (UI) hold the basic user-interface functionality. They are divided into several components according to divisions found in the problem domain. Between the Problem domain objects (or just Domain objects) and the user-interface components, the "Info objects" are found. These act as an interface between the Domain objects and the user-interface components. The Info objects are data structures that "mirror" the data in the user interface. Introducing this layer between the UI and the Domain objects de-couples the direct dependence between them. The Controller functions and Model functions components shown in the middle also act as interfaces; the Controller functions mediate between the user-interface components, the Info objects and the Business functions, and the Model functions mediate between the Domain objects and the Info objects. Ideally the Info objects layer is used as follows: when a Business function is activated via the UI (onEvent), the relevant content of the UI is mapped into the Info objects (getView). The corresponding Domain objects may need to be updated (updateModel) and the relevant parts of the Info objects are used in the Business function. After execution of the Business function the Info objects are updated (presentModel) followed by an update of the UI (setView).

The left and middle part of the architecture diagram shows the well-defined parts of the architecture. The user-interface and problem domain model units were constructed in the first cycle, and they become elements in the initial architecture (the user-interface component is transferred directly, and the classes of the problem domain model are instances in the 'problem domain objects' component).

The right part of the diagram, containing the business functions, is represented by the "emerging structure" type. The intention is to express architectural uncertainty. Since the business functions were unknown from the start, they could not be put in a well-defined box with well-defined interfaces to the other components. We needed a space in the architecture where we could experiment with the business functions in an unrestricted and flexible way. This architectural uncertainty will be further discussed in the following section on flexibility.

A Stable Structure for Evolution. The desire for a stable structure, in which the prototype can evolve, can be seen as a way of reducing the complexity of the system. By defining a detailed structure within which the coding must be done, this complexity is reduced: E.g., defining the "communication protocol" between the user interface and the problem domain objects via info objects gives a clear and simple way of doing this communication.

In this way, the architecture provides an overview of the quickly growing prototype and constitutes a consensus about "how to do things". Furthermore, it ensures a "uniformness" as to how the functionality is implemented, and e.g. prevents developers pressured of time from "quick-and-dirty" solutions, like implementing the functionality directly in the code for the user interface.

Flexibility for Experimentation. Another criterion in the design of the architecture was flexibility, as it should allow evolutionary development of and experimentation with all parts of the prototype. This requirement is reflected in the large number of interfaces between the components in the architecture. These interfaces provided independence between e.g. the user-interface and the problem domain objects, which made it possible to experiment with each of these components without affecting the other components.

When it comes to the business functions, it was impossible to define an independent component that contained these, since the business functions were not well known. Furthermore, since the business functions are bridging between the user interface and the problem domain objects, they need to be able to access the user-interface (via the info objects) as well as the domain objects. And from the starting point and until very late in the prototyping process, it is not known exactly which part of the user-interface and which domain objects are involved.

Consequently, in order to allow for flexible experimentation with business functions, business functions need to have easy access to all components: user-interface, controller functions, info objects, model functions, and domain objects. The solution is an open architecture with white-box components. Black-box components can not be introduced before the business functions have been found and developed. To indicate that the architecture has room for flexible experimentation with functionality, the business functions component is shown as an emerging structure in the diagram. As explained above, the business functions were intentionally not very structurally elaborated in the architecture in the prototyping phase. Instead the functions emerged in the boundary of the controller functions in the event handlers of the user interface. This structure was indeed very flexible but as will be described later, its use should be limited to the prototyping phase.

Efficient work organisation. As mentioned previously, another important criteria for designing the architecture was that it should facilitate an efficient work organisation. In our context, it should support roughly the following work activities in phase one:

- creation of the user-interfaces by the cooperative designer,
- creation of programming interfaces (info objects and controller functions) to the user-interface modules,
- programming of advanced graphical user-interface elements that could not be created directly using the graphical user-interface builder,
- programming of functionality (business functions and model functions),
- programming of the problem domain model,
- programming of components simulating legacy systems.

These activities had to be performed concurrently in order not to slow down the development. The architecture facilitated this. By making a strict division of the user-interface, the business functions, and the problem domain model, with well-defined interfaces, the dependencies were reduced in a way such that problems were seldomly experienced. As an example, the info objects interface isolated changes in the user-interface from having an effect on the domain objects and visa versa.

People had areas of responsibility according to major functional areas of the problem domain. One developer was, for instance, responsible for the customer-related functionality, another for the booking-related functionality and yet another for creating programming interfaces.

4. Phase Two: Evolutionary Development

We go beyond the discussion of architecture as a means of providing flexibility, stability, and proper work assignments in the prototyping phase and discuss why and how the prototype was used in the second phase of the project. In doing so, we will discuss the concrete architectural refactorings made, and the reasons for undertaking such restructurings.

4.1 The Transition to Evolutionary Development

After the end of the first phase of the project, a decision was made to extend the project for a year. In this second phase development continued on the prototype developed in the first phase, although the scope and focus of development changed. Among other things other areas of the system were to be explored and developed. These areas included multi-user functionality, database issues, and the general identification and creation of "black-box" components and their topology. Thus, the requirements of the architecture changed.

The Problems. The initial architectures enabled us to focus on experimentation with business functions. However, the prototype grew in size and complexity: From containing some over 150 files with around 50,000 lines of code after the experimental prototyping phase, the prototype evolved to contain over 300 files and around 100,000 lines of code at the end of the Dragon Project. This growth introduced a number of problems that forced us to refactor the prototype architecture. Two major classes of problems were:

- The prototyping sessions and the reviews produced new requirements to the prototype which "demanded" changes to the architecture, e.g. in order to be able to reuse code better,
- Work within the existing architecture had shown annoyances, e.g., unclear separation of responsibilities and inconvenient dependencies between different parts of the prototype that could be rectified with a new architecture.

An example of the first type of problem is that, in the initial architecture, it was difficult to reuse business functionality. Until late in the first phase of the project, the required functionality was usually local to the corresponding part of the prototype: e.g., only the booking-handling part used the booking-handling functionality. However, the existing architecture did not provide the proper abstractions for the reuse (which actually meant that a compound function initially was implemented using copy-n-paste reuse of the existing code - due to constraints of time).

An example of the second type of problem was that the turn-around time (edit/compile/run cycles) in the initial architecture increased as the prototype grew bigger due to unpractical dependencies between parts of the prototype: certain changes to the model, the controllers or the business functions caused recompilation of large parts of the prototype. The main program with the controllers and the business functions became a bottleneck, and as the prototype grew, the recompilation time became a limiting factor in the rapid development process.

The most flexible part of the prototype - the emerging business function components - had thus become problematic at this point. However, the *identification of business functions* enabled us to make clearer abstractions on the problem domain. In this way, the flexibility, inherent in the initial architectures, actually became a forcing as well as an enabling factor in the change. In other words, the "white-box" components of the initial architecture allowed for a high flexibility in the development process, but caused problems as the system grew. Moreover, the identification of business functions enabled the construction of "black-box" components (Parnas, 1972).

The Problems Resolved. The experienced problems were solved with a new architecture, depicted in figure 3.

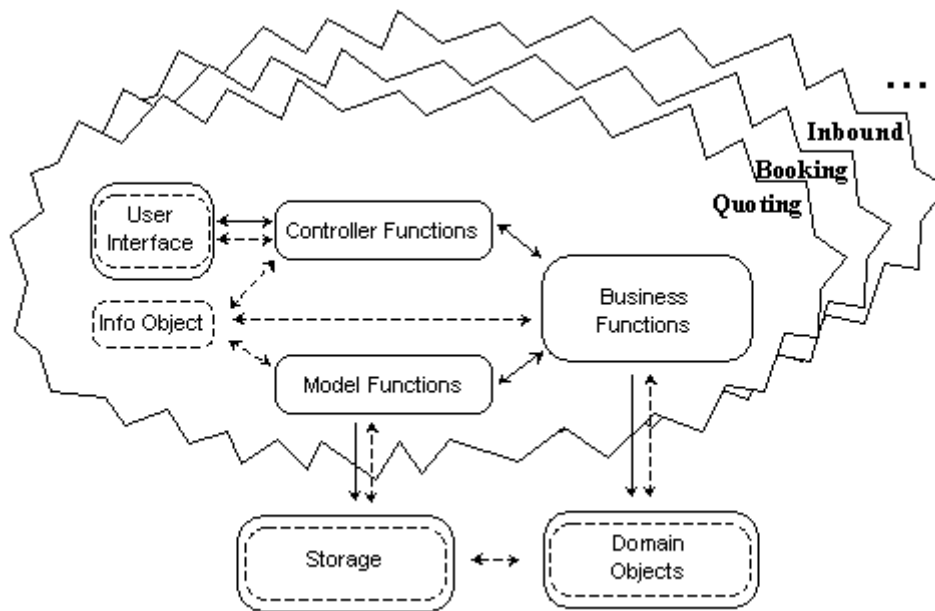


Figure 3. New architecture

The major change was made in the role of the business functions. From focussing almost entirely on the business functions, the architectural focus changed so that is now included the role of *problem domain area components*. More specifically, the goal was to develop black-box components, corresponding to the major problem domain areas, and to look into the division of work and responsibility between those components. In this way, it can be said that initial architectural work in the evolutionary development phase focused on the component topology.

Thus, the flexibility in the architecture at this point had to be mostly at the level of component boundaries. This is illustrated in figure 3 by now using the Emerging structure at the component level instead of at the Business functions level, because the architectural uncertainty now is on the component level. The flexibility allowed for concrete experiments with component topologies during the first part of phase two, and led to a reasonable structuring of the software architecture.

As it turned out, this architectural refactoring did pay off in terms of faster turn-around time, higher levels of abstraction, and the ability to identify "black-box" components.

4.2 Architectural Refactorings in Evolutionary Development

The success of the first architectural refactoring meant that *explicit refactoring phases* were incorporated in the project plan after the first three hectic months of the project. In this way each major review would also contain a review of architecture. Since the focus before a review was mostly on getting the system to work according to the planned changes, and the length of cycles was short (especially in the first phase of the project), we had to relax rules and conventions in order to meet the deadlines, e.g., by employing copy-n-paste reuse of code. This could then be rectified after a review; either by refactoring the architecture or refactoring code.

Please recall from section 2.1 that we distinguish between 'refactoring' and 'architectural refactoring'. In the Dragon Project refactorings could be done as a part of the ongoing development: One person, typically the person who made some shortcuts in the code during the hectic days before a review, did local refactorings. Architectural refactorings, which could be done independently in isolated parts of the prototype, were delegated to all the developers and they could do them when convenient.

The major architectural refactorings concerned the whole prototype, and hence development could not go on as usual. This meant that the role of deciding if, when, and how an architectural refactoring should be made had to be separated from ordinary development. Also, the holder(s) of the architectural role had some time set aside for doing the restructuring without doing ordinary development at the same time. In

this way the major restructurings in reality meant temporary interruptions of development. The restructuring usually took between a couple of days and a week. Several refactorings took place during the evolutionary development phase meaning that the architecture of the system as delivered June 1998 can be depicted as in figure 4. Components and their boundaries have now stabilised in the sense that they represent useful abstractions of the problem and use domain. Notice that the Emergent structure symbols have been substituted by well-defined architectural component symbols, because the architectural uncertainty has been resolved.

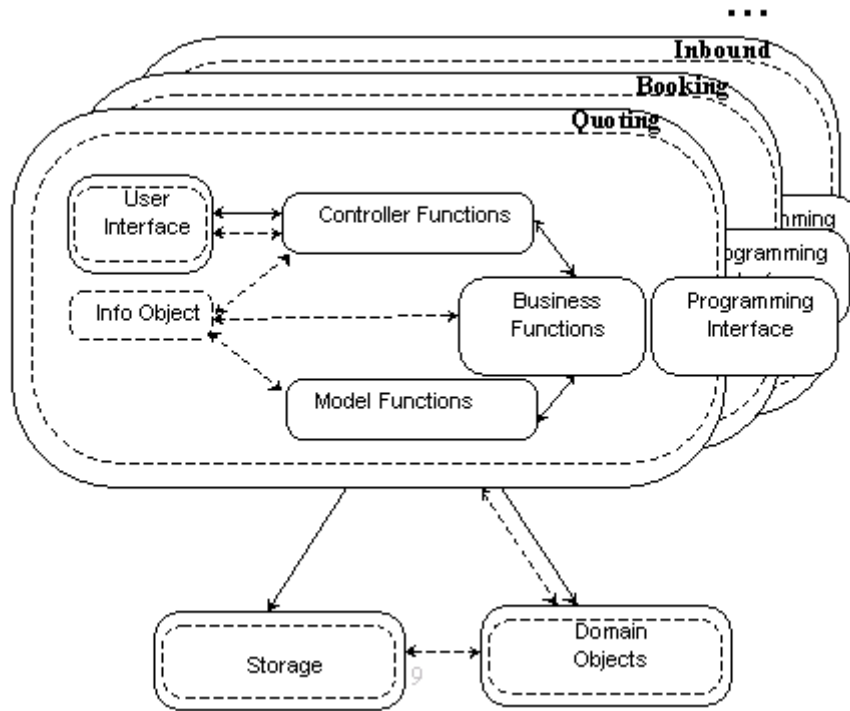


Figure 4. Architecture of June 1998

Conclusions

An inherent dilemma with experimental prototyping and evolutionary development lies between user involvement and software engineering. User involvement - and cooperative design in particular - provides for efficient analysis and design of functions pertaining to a problem domain. Object-orientation is then concerned with the implementation of emerging ideas of future work practices. The dilemma lies in allowing for flexibility and experiments with future work practice, while at the same time preserving the benefits of object-orientation and traditional software engineering. We address just that: The reconciliation of experimental development practice and software engineering practice goes through a strong focus on software architecture throughout the entire development process. In doing so, aspects of the changes in architecture that have occurred during the first two major phases of the Dragon Project have been discussed. These may, very coarsely, be depicted as in figure 5.

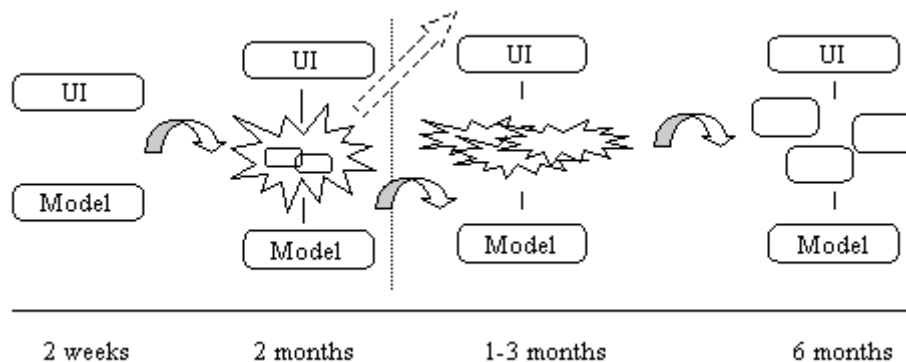


Figure 5. Architecture changes in phase one and two

This picture should convey two messages: *Firstly*, architectural focus in an experimental prototyping phase should very much be placed on the procurement of flexibility, stability and well-defined foundations for discovering functionality, adding new user-interface screens, etc. *Secondly*, evolutionary development may either discard or retain a successfully developed prototype. In either case, focus is much more placed on architectural prototyping: discovering and experimenting with a proper component topology, experimenting with legacy systems, distribution issues, etc. Still, both architectural and functional insights, gained in an experimental prototyping phase, provide rich and important input to the evolutionary development phase. Moreover, still being able to experiment with user functionality is important for several reasons: Although a proof of concept, a first prototype may not cover the whole scope of the final system in width; politically, organisations may need a continually evolving prototype to ensure user buy-in, etc. Continued experiments may also provide important insights, which also have architectural consequences.

Our project was characterised by dealing with complex human work practice, an unknown problem domain, and the need for rapid work, a context that we do not believe to be unique for our experience. Thus, we have suggested an *architectural strategy*, a set of architectures, their context and evolution, which support experimental prototyping and evolutionary development in a context as ours. This strategy focuses on *stability* and *flexibility* in design of a computer system. Stability must e.g. provide for efficient parallel work in experimental prototyping and provide for efficient prototyping of a final architecture during evolutionary development. Flexibility must e.g. provide for the experiments with business functions in experimental prototyping and for experiments with component topology during evolutionary development. The Emergent structure symbol indicates where the experiments take place.

References

(Bass et al., 1998). Bass, L., Clements, P., Kazman, R. Software Architecture in Practice. Addison Wesley Longman.

(Christensen et al., 1996). Christensen, M. Sandvad, E. Integrated Tool Support for Design and Implementation. Nordic Workshop on Programming Environment Research (NWPER'96), Aalborg, Denmark.

(Christensen et al., 1998). Christensen, M., Crabtree, A., Damm, C.H., Hansen, Klaus. M.H., Madsen, O.L., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., Thomsen, M. The M.A.D. Experience: Multiperspective Application Development in evolutionary prototyping. Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP '98), Brussels. Springer Verlag.

(Opdyke, 1992). Opdyke, W. Refactoring Object-Oriented Frameworks. Ph.D: Thesis, University of Illinois at Urbana-Champaign.

(Parnas, 1972). Parnas, D.L. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, vol. 15, no. 12, December.