

Design and Evolution of Software Architecture in Practice

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

With special focus on software architectural issues, we report from the first two major phases of a software development project. Our experience 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. However, the architecture 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 - that was helpful in this respect.

1. Introduction

Iterative development processes that take an evolutionary and incremental approach to application development are becoming standard in most object-oriented development processes. However, the development of applications that use these iterative approaches, lead to many new problems. Where the traditional waterfall processes provides an (assumed) precise and definite description of what is to be built, there is no equivalent firm foundation on which to base the initial development when using iterative processes. Furthermore, in doing iterative development, as the development continues, new and/or changing requirements must be addressed.

In the last couple of years the authors of this experience paper have been involved in a large project, that 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. The roles of the different team members may, somewhat simplistically, be described as follows: the ethnographer focused on current practice within customer service and related areas, the cooperative designer focused on the design of future practice and technological support together with users. The OO developers

developed the problem domain model and implemented the prototype. For a more thorough discussion of the process and its multi-perspective application development character see [7].

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 in 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 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. This can be due to changing requirements over time, due to increasing understanding of the problem domain, and due to further understanding of the technical ways of realising the system. In this way, not only the system itself but also the software architecture can be said to be prototyped.

2. Software Architecture

No definition of software architecture is commonly agreed upon [1], [5], [11], [21]. However, it is commonly agreed that software architecture is concerned with components of the system and their interrelationships. We will, therefore, in our discussion of architecture, focus on the *significant software structures, and their components and relationships*. As several authors remind us, e.g. [15], [5], [1], a software architecture can be observed from several specific views or viewpoints, each revealing different aspects. Examples include the module view, the process view, and the conceptual view. The module view describes what the important modules and sub-modules are, and it is useful for e.g. assigning work and defining encapsulation. The process view describes the running processes and their relation, and it is useful for e.g. performance analysis, and analysis of multi-user aspects. The conceptual view describes the major components in the architecture as perceived by the developers. This view is useful for e.g. understanding the problem domain and the current system in relation to that.

Then, to describe the software architecture of the system discussed in this article a slight variation of the notation described in [1] will be used. 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.

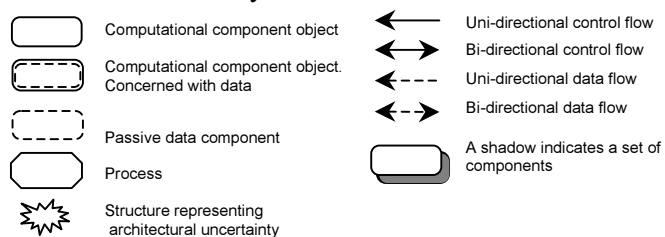


Figure 1. Architecture notation

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 [19], 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.

By using specific examples of the evolution and architectural refactorings that the developed system has undergone so far, this experience paper aims at explaining how a focus on architectural issues may support an experimental and evolutionary development process, that, although experimental and evolutionary, still aims at producing a sound system as seen from the software engineering perspective. Moreover, a concrete architectural strategy to support this process is pointed to.

3. The Development Process

The development of the initial prototypes and later the concrete evolutionary development was done for a large, global shipping company. Our work on this development cannot be regarded as an isolated piece of work though: it was part of a larger project within the company that concerned the development of a uniform and globally shared way of providing container transportation. The entire project can be described in a number of streams as illustrated in Figure 2 below.

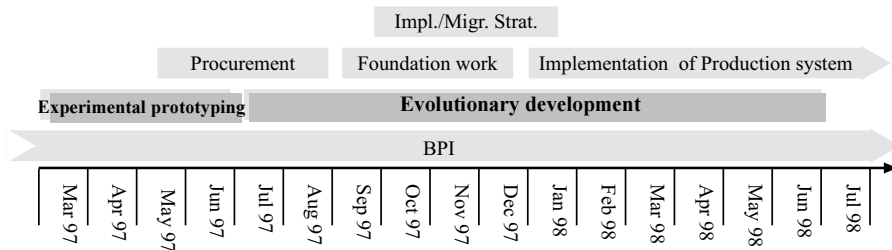


Figure 2. The streams of work within the overall project

Our work – referred to as the *Dragon Project* – is here seen as one stream (boldfaced), that consists of two major development phases. The shipping company, in cooperation with external consultants, carried out other streams of work. Examples of these include a global Business Process Improvement (BPI) project and various types of foundation work regarding network issues, migration issues, and overview of existing databases. These different streams interacted and, of course, influenced each other. That is, the overall process of the Dragon Project is something that should be seen as being related to and influenced by the other streams of work, with the common goal of developing a global customer service system for the company.

3.1. The Dragon Project Process

The process within our project can be divided into two separate phases: An experimental prototyping phase and an evolutionary development phase.

Experimental Prototyping. The first phase was primarily concerned with obtaining an understanding of the problem domain that the system should support. To this end, a number of software prototypes were constructed and evaluated in collaboration with the actual end-users.

The rationale, that ought to be well known, is that building computer system from paper descriptions alone, without any intermediate embodiments, is a problematic endeavour [13].

Concretely, and initially, in this project, a number of prototype versions were developed. These should illustrate and support the implementation of the global improvements suggested by the BPI stream. The concrete requirements resulting from the BPI stream were, however, quite vague. The whole first phase of the project was therefore very much occupied by narrowing down *what* a final system could be and by helping to determine the feasibility of the actual development.

To clarify and following [9], we characterise our approach as *experimental prototyping*, as our prototype versions were used to determine whether our ideas were adequate. This is only one characteristic of the initial phase, though. Other characteristics include: An *exploratory* style of prototyping (ibid.) to increase our limited understanding of the complex business of shipping; an *iterative* approach with short development cycles (14 days on average between reviews) to facilitate rapid development; a combination of *horizontal* and *vertical* prototyping (ibid.) to achieve both depth and width; and, finally, in general, usage of a wide variety of techniques from cooperative design [2], [12].

Evolutionary Development. The result after the first phase was that the company decided to embark on the implementation of the final system, their decision being based on the prototype. This led to the initiation of the second phase of the Dragon Project. Compared to the experimental prototyping phase, the character of the work in this phase naturally changed focus towards the development of the production version of the final system. Apart from the Dragon Project, the company, as mentioned previously, initiated different types of foundation work.

In this setting, the role of the Dragon Project was twofold. On one hand, the prototype served as specification (in a broad sense, encompassing e.g. division into components, client-side architecture and problem domain model, i.e., not a *requirement specification*) for the production version, and on the other, it served as "the big picture" for demonstration/teaching/design purposes. This meant that, in the second phase, emphasis had switched more towards elaborating the components/concerns previously identified, with particular focus on the client-side of the coming system. We characterise this part of the process as the *evolutionary development* phase. Naturally, the functional requirements were not all fully elaborated after the experimental prototyping phase, so many of the same techniques and approaches were applied in this phase. Nevertheless, a much clearer understanding of what the constituent parts of the system should be and how they should interact had been obtained. This made way for a more steady, evolutionary character of the work, with longer development cycles (reviews about every two months) and room for necessary software architectural considerations.

Work within the Phases. Within each of the phases, our work was organised in a number of development cycles that had a duration of approximately 14 days in the first phase and approximately two months in the second phase. Figure 3 below gives a schematic overview.

As the figure illustrates, a number of concerns were addressed within each cycle (i.e., between two reviews). The left side of the dashed vertical line shows the first phase, in which focus was on producing functional, but not very elaborate, implementations of major concerns. In the second phase, shown to the right, our focus shifted towards evolving and elaborating these.

In each cycle and with each of these concerns, a mixture of analysis, design, and implementation was made depending on the current understanding and possible experiences from an earlier cycle. This always involved continuous workshops with business representatives. Furthermore, each concern would be addressed by several members of the development team in parallel and seen from different perspectives, and several concerns were often treated concurrently.

To “control” the development, each cycle ended with a review, during which the current work was presented to some of the future end-users and management from the company. At these reviews, possible problems with the current solutions would be brought forward and a plan for what was to be done in the next cycle would be made.

A much more in-depth discussion of the development process, and the lessons learnt from it, can be found in [7].

3.2. Tools and Code

The concrete results achieved is a number of prototype versions that has evolved into a rather large application consisting of over 300 files containing over 100,000 lines of code, and having well over 50 screens. The development platform was Windows NT, and the main software engineering tools used were: a CASE tool, a graphical user-interface builder, a code editor, a persistent store, a relational database (IBM DB2), and concurrent versioning system [8]. The first four tools are part of the Mjølnir System [14], [6], [18] and the programming language used was BETA [16].

3.3. Related processes

In many ways, Barry Boehm’s *Spiral Software Development Model* [3] resembles our process. The spiral model is both iterative and evolutionary. It is also based on the notion of development cycles, and these cycles also end with a review, in which plans for the next cycle are made. Furthermore, Boehm thoroughly describes how “risk management” controls the process. There is no explicit focus on software architecture in Boehm’s article, but through his description of risk management, an indirect advice can be found: If the current architecture is considered a risk or an uncertainty as each cycle is planned, it should be given priority on the following cycle.

The *Unified Software Development Process* [10] describes a development process divided into a number of overall phases of which the first two, *Inception* and *Elaboration*, to some extent, are similar to our *Experimental Prototyping* and *Evolutionary Development* phases. The Inception phase is concerned with the definition of scope and should include a description of a *candidate architecture* and demonstrate the proposed system via prototypes. The Elaboration phase more thoroughly analyses the problem at hand and should result in an *executable architecture baseline*, a *software architecture description*, and an *80% complete domain analysis model*. The Unified Software Development Process is furthermore described as an *architecture-centric process*. We agree with this focus on architecture. Among others, we believe that our project provides empirical evidence on the importance of architectural focus and that a combination of rapid prototyping and architectural focus is feasible.

Rapid, iterative prototyping is in the industry often manifested in the form of RAD (Rapid Application Development). In RAD the development consists of a number of iterative development cycles, and there is also much focus on the use of prototypes. It seems though that RAD assigns no special role to architecture. In the standardised RAD described by DSDM [22] eight principles underlying RAD are described; none of these relate to software architecture.

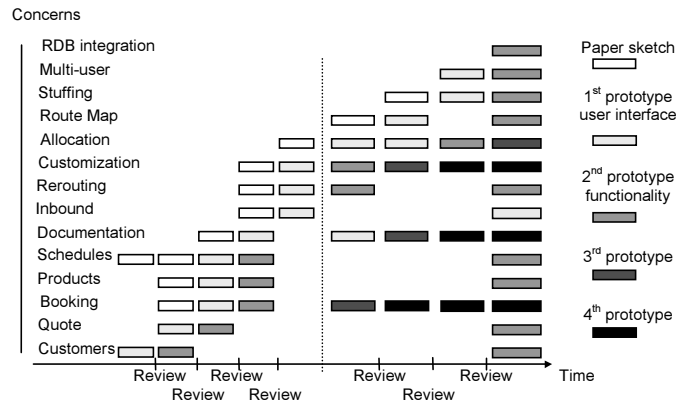


Figure 3. The evolution of central concerns

4. Phase One: Experimental Prototyping

In section 3, it was described how the development was partitioned into two major phases. In this section, the role played by our software architecture in the first phase is described. More specifically, we look into the way, in which the concrete architecture provided a stable foundation for the experimental prototyping, while, at the same time, it allowed for the necessary room for experimentation. Furthermore, we describe the important criteria of having an efficient work organisation.

4.1. Initiating Development

The development was scheduled to begin with a two-week cycle. There was no concrete, specific goal of this first cycle, and we quickly became aware that our understanding of the business of shipping was quite limited, so two strands of work were initiated within the first cycle. One group worked on creating graphical user-interface screens, and another group worked on creating an initial problem domain model. The two groups worked in parallel, and there was little coordination between the groups. This was mainly a practical choice due to the strict time constraints, but it also proved to be a sensible one: In this way, many and diverse pieces of knowledge about the business domain were quickly gained.

4.2. Elements of an Architecture

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.

From an architectural/software-engineering point of view, we could, at this early point in time, – only two weeks into the development – identify the first two architectural elements: one component, consisting of the initial user-interface screens, and one component, consisting of the problem domain model (see figure 4).

Retrospectively, we can ask ourselves why these two components were chosen. Much of the reason for this can be found in the developers’ background. First of all, the development team had a common background in the Scandinavian approach to object-oriented development. In this approach, object-orientation is not seen as just providing a pleasing technical environment. It more importantly also provides a conceptual framework [16], [17] that underlies development. The creation of a problem domain model therefore becomes an essential part of development, since the problem domain model illustrates our understanding of the most important concepts in the problem domain.

Furthermore, part of the development team has a background in cooperative design [2], [12]. As the essence of cooperative design is the creative involvement of the user in the design of the computer system, the user-interface naturally becomes important. That lies in the simple fact that most users are not concerned with the code and other technicalities; they are simply

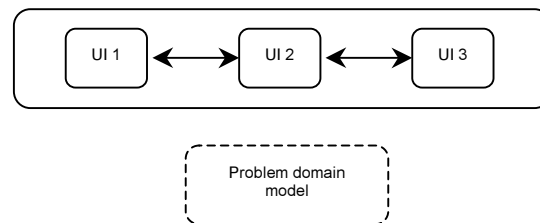


Figure 4. Elements of initial architecture

concerned with the way in which the system is operated, and how the system performs the tasks they expect.

4.3. Designing an Architecture

The review after the two weeks concluded that the development was on the right track. Together with suggestions for changes and next steps, it was decided that we should add functionality to what was covered so far and extend both the user-interface and the problem domain model to cover another business domain – booking.

As an implication of the decision to start adding functionality, it was considered how the existing parts (user-interface and problem domain model) could be combined. That is, the task was to design a software architecture that could contain both the existing elements and the functionality to be added. This architecture 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, and
- it should support an efficient work organisation, allowing all developers to work intensively on the prototype in parallel.

Figure 5 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.

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. The business functions are the functions that are directly visible to the user; they implement functionality in the problem domain.

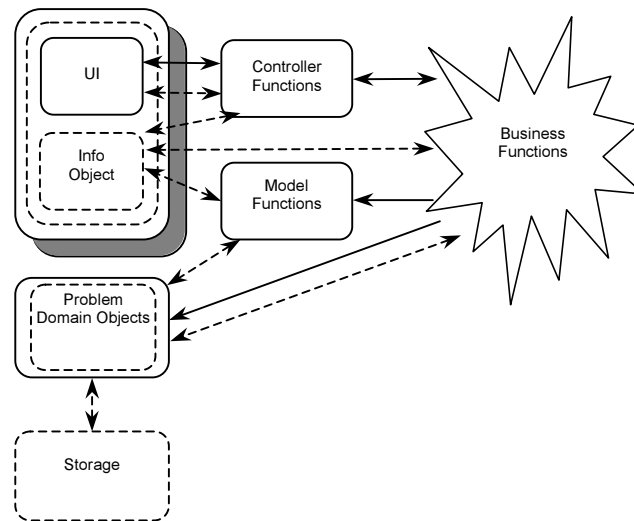


Figure 5. Initial architecture

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). An elaborated discussion of this architecture can be found in [23].

The left and middle part of the architecture diagram shows the well-defined parts of the architecture. Please recall that the user-interface and problem domain model units were constructed in the first cycle, and note that they remain elements in the new 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.

Furthermore, the architecture serves as a vehicle for communication and explanation. This was for example experienced when a new developer was introduced to the prototype late in the experimental prototyping phase: as a result of the well-defined architecture, he obtained an understanding of the relatively large prototype quite easily.

It should, however, be noted that the well-defined architecture illustrated in the diagram was not achieved in full detail instantly after its definition. Although principal decisions about the overall structure were made immediately, some smaller decisions were not made, that allowed each developer room enough to find what he considered the best solution. As further understanding was achieved, these variations would be discussed and the developers would agree on a shared solution. In this way as the prototype evolved, even some overall structural decisions evolved.

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, that 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.

Another issue that enabled experimentation was the fact that we did not have to spend much time on handling storage in the prototyping phase. Making data persistent was an important aspect to be dealt with even in early design and implementation of the prototype. The persistence in the Mjølner system is orthogonal and transparent. This means that any object can be stored and that given a *persistent root* all reachable objects will be made persistent transparently. In this way, achieving persistence in the first versions of the prototype was straightforward. However, despite that, we did spend some time on storage: it was an explicit requirement in the first phase that the Dragon Project at some point should experiment with multi-user functionality, and abstractions preparing for this were made at the beginning of phase one. These abstractions, however, were not unproblematic. They were made prematurely, and in this way, constrained the initial architecture unnecessarily.

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 and well-defined division of the user-interface, the business functions, and the problem domain model, with interfaces, the dependencies were reduced in a way such that problems were seldomly experienced. As an example, the info objects interface prevented most changes in the user-interface from having an effect on the domain objects and vice 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.

The user-interface was often used as a means of organising and/or coordinating activities between the cooperative designer and the OO developers. The cooperative designer mostly made the initial user-interface design in the graphical user-interface builder, and then in collaboration with the OO developers further elaborated it. Due to reverse engineering capabilities of the graphical user-interface builder, it was possible for the cooperative designer to make changes to the user-interface throughout the process, even after the generated code had been changed by the other developers.

Finally, the problem domain model served as a constant common frame of reference between members of the development group and to some extent also between developers and

members of the business. It forms a separate entity of its own right in the prototype and it was kept *purely* as a model of the problem domain isolated from implementation-specific classes. Also, the model evolved and since diagrams were used for discussion purposes, the reverse engineering capabilities of the CASE tool became important in the development process.

5. 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.

5.1. The Transition to Evolutionary Development

After the first phase it was decided to extend the project for a year. This naturally changed the scope and focus of development. Although the prototype should, preferably, be used for continued experiments with business functionality, 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.

During the transition we evaluated whether the prototype, resulting from phase one, should be transferred into the evolutionary development phase. We had several options, including: To continue development of the prototype from the first phase; throw away the existing code, only transferring functional requirements; or totally discard the results from the first phase. Given that we were to use the same programming language and tools, that were used in the prototyping phase; that the code and architecture produced generally was of reasonable quality; and that the prototype had been well-received by the business, we decided to continue from the results of the first phase. This was not without problems though.

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 that “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 part used the booking functionality. The problems became apparent, as a “compound” function was needed that could perform a number of existing functions on a bulk of business objects. This compound function did not bring about any changes to the problem domain model, but only required reuse of a large part of the existing functionality located in different units. However, the existing architecture did not provide the proper abstractions for the reuse (this actually meant that the compound function initially, due to constraints of time, was implemented using copy-n-paste reuse of the existing code).

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 [20].

The exact timing of this architectural refactoring depended on a number of (social) factors: in our specific setting, the new scope made it *desirable* to have a new architecture, and the “spare time”, in between the two phases, made it *practical* to do the restructuring at that point in time. Another very important fact was that it was now *possible* to make the actual changes based on a greater understanding of both the problem domain and the “solution domain”. Of course, the economics played an important role too; it was expected that the restructuring would pay off in the long run.

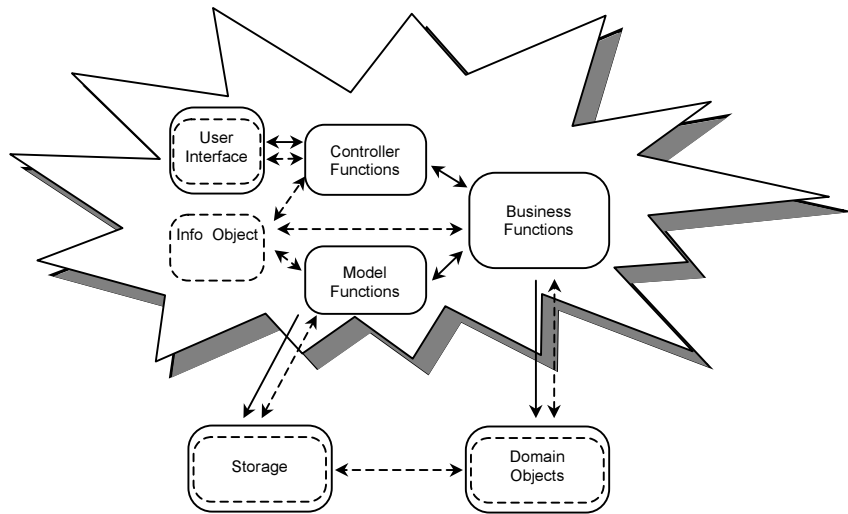


Figure 6. New architecture

The Problems Resolved. The experienced problems were solved with a new architecture, depicted in figure 6. 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 6 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. During a trip to Asia, e.g., the power of this new architecture was experienced, as it was actually possible to add major components to the system “on the fly”. In Singapore the system was modified ac-

ording to the discussions in prototyping sessions. To give an impression of the time frame: the visit to Singapore took three days. On the first day the system was presented and discussed in a number of sessions, with different representatives of the business and at different detail levels. Based on these sessions a number of changes to the prototype were decided. The next day, two developers made the changes in the prototype while the ethnographer and cooperative designer continued their work with the business representatives. On the third day the new version of the prototype was presented. This process was repeated in Malaysia. In this way two major component objects were added: A “query overview” component object providing overview of data related to major objects in the application domain and a “pending tray” component object supporting many collaborative work processes within customer service in the company.

However, this architectural refactoring was not entirely unproblematic. The trade-offs to be considered in such situations will be discussed in the next section.

5.2. 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, that 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.

Not all changes were equally problematic. As the persistent store used provided for transparent, orthogonal persistence it was a relatively straightforward process to retarget the database component of the system. This meant that after that transition a transparent, heterogeneous storage mechanism was available. In this way programming to relational databases and persistent stores was possible at the same time. Also, initial identification of problem domain components and their boundaries were relatively easy after the first phase because of the increasing understanding of the major areas of business functionality during the first phase. Conceptually the candidates for problem domain area components had gradually been formed during the first phase. This architectural refactoring was, due to the use of a syntax directed code editor with semantic browsing and abstract presentation possible within short time. The abstract presentation provided overview and the fact that the editor was syntax directed made syntactically valid transformations of implementation code possible. Semantic “adjustments”, however, had to be done using the semantic browsing facilities of the editor. Furthermore the transformation introduced a number of errors that called for renewed testing of the prototype. The architectural refactorings were thus somewhat problematic seen in the context of rapid

system development: A tool for reverse and forward engineering of the architecture or a more direct incorporation of architectural support in the tools used would be very useful.

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 7. 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 the well-defined architectural component symbols, because the architectural uncertainty has been resolved.

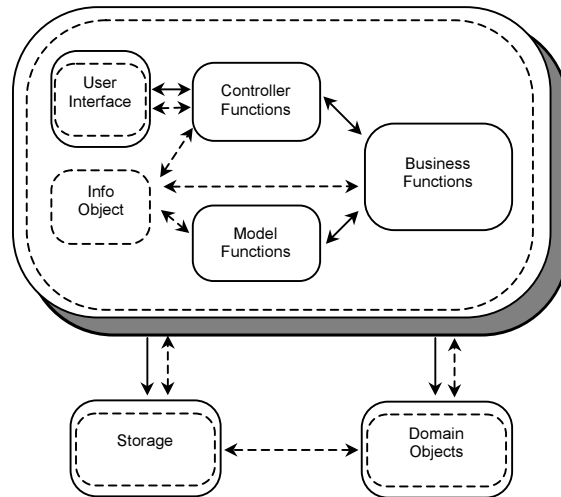


Figure 7. Architecture of June 1998

6. Conclusions

Aspects of the changes in architecture that have occurred during the first two major phases of the Dragon Project have been discussed. Figure 8 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, that 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 - that

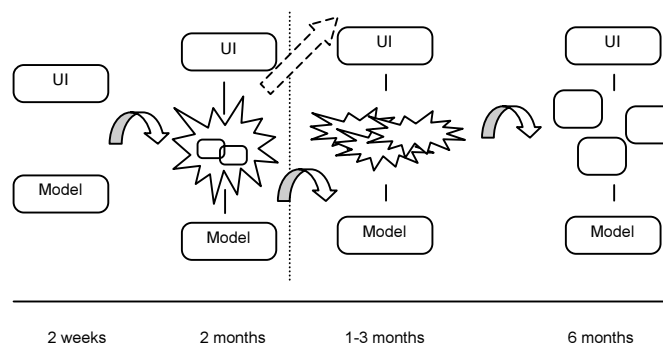


Figure 8. Evolution of architecture

supports 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 experi-

ments with component topology during evolutionary development. The Emergent structure symbol indicates where the experiments take place.

An inherent dilemma of experimental prototyping and evolutionary development lies in the mixing of 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.

7. Acknowledgements

This work was supported by the Danish National Centre for IT-Research (CIT, <http://www.cit.dk>), research grant COT 74.4.

We would also like to thank Michael E. Caspersen, Henrik Bærbak Christensen, Jørgen Lindskov Knudsen, and Janne Christensen for valuable comments that improved this paper.

8. References

- [1] Bass, L., Clements, P., Kazman, R. *Software Architecture in Practice*. Addison Wesley Longman, 1998.
- [2] Bjerknes, G., Ehn, P., & Kyng, M. *Computers and Democracy: A Scandinavian Challenge*, Aldershot: Avebury, 1997.
- [3] Boehm, Barry W. *A Spiral Model of Software Development and Enhancement*. COMPUTER. Vol. 21, No. 5, May 1988.
- [4] Brooks, F.P. *The Mythical Man-Month: Essays on Software Engineering. Anniversary Edition*. Addison-Wesley, 1995.
- [5] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture. A System of Patterns*. Jon Wiley and Sons, 1996.
- [6] Christensen, M. Sandvad, E. Integrated Tool Support for Design and Implementation. Nordic Workshop on Programming Environment Research (NWPER'96), Aalborg, Denmark.
- [7] 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.
- [8] Gnu, *Concurrent Version System*. <ftp://archive.eu.net/gnu/>.
- [9] Floyd, C. *A Systematic Look of Prototyping*. In R. Budde, K. Kuhlenskamp, L. Mathiassen, & H. Züllighoven (eds.), *Approaches to Prototyping*. Springer Verlag, 1984.
- [10] Jacobson, I., Booch, G., Rumbaugh, J. *The Unified Software Development Process*. Addison Wesley, 1999.
- [11] Garlan, D., Shaw, M. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [12] Greenbaum, J., & Kyng, M. *Design at Work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associates, 1991.
- [13] Grønbaek, K. *Prototyping and Active User Involvement in Systems Development: Towards a Cooperative Prototyping Approach*. Ph.D. Thesis, Aarhus University, Denmark, 1991.
- [14] Knudsen, J.L., Löfgren, M., Madsen, O.L., Magnusson, B. (Eds.) *Object-Oriented Environments. The Mjolner Approach*. Prentice Hall, 1993.
- [15] Kruchten, P. *A 4+1 View Model of Architecture*. IEEE Software 12 (6), 1995.
- [16] Madsen, O.L., Møller-Pedersen, B., Nygaard, K. *Object-Oriented Programming in the BETA Programming Language*, ACM Press, Addison Wesley, 1993.
- [17] Madsen, O.L. Open Issues in Object-Oriented Programming – A Scandinavian Perspective. SOFTWARE – Practice and Experience, vol. 25, no. 54, 1995.
- [18] <http://www.mjolner.com>
- [19] Opdyke, W. *Refactoring Object-Oriented Frameworks*. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.
- [20] Parnas, D.L. *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, vol. 15, no. 12, December, 1972.
- [21] <http://www.sei.cmu.edu/architecture/definitions.html>.
- [22] Stapleton, J. *The Dynamic Systems Development Method Manual v2.0*. The DSDM Consortium, 1995, <http://www.dsdm.org>.
- [23] Hansen, K.M., Thomsen, M. The 'Domain Model Concealer' and 'Application Moderator' Patterns: Addressing Architectural Uncertainty in Interactive Systems. Proceedings of the 31th Conference on Technology of Object-Oriented Languages and Systems (TOOLS Asia '99)