

Architectures of Prototypes and Architectural Prototyping – The Dragon Experience

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

Department of Computer Science, University of Aarhus, Building 540,
Ny Munkegade, DK-8000 Aarhus C, Denmark

E-mail: {toby, damm, marius, ess, miksen}@daimi.aau.dk



Abstract

This paper reports from experience obtained through development of a prototype of a global customer service system in a project involving a large shipping company and a university research group. The research group had no previous knowledge of the complex business of shipping and had never worked together as a team, but developed a prototype that more than fulfilled the expectations of the shipping company.

The prototype should:

- complete the first major phase within 10 weeks,
- be highly vertical illustrating future work practice,
- continuously live up to new requirements from prototyping sessions with users,
- *evolve over a long period of time to contain more functionality*
- *allow for 6-7 developers working intensively in parallel.*

Explicit focus on the software architecture and letting the architecture evolve with the prototype played a major role in resolving these conflicting constraints. Specifically allowing explicit restructuring phases when the architecture became problematic showed to be crucial.

1. Introduction

It should be well known to all developers that designing good, stable and reusable object-oriented software is hard. And it should also be well known that doing it under circumstances where the requirements to the software are not known, where the area

of work the software is going to support into is not known and where time is a scarce resource does not exactly make things easier. These are exactly the circumstances under which the software development project that this experience paper reflects upon was carried out.

What we describe in this paper is our experiences acquired during a 14-month prototyping project. When doing prototyping in an experimental, evolutionary manner, as we did, a number of challenges must be faced. First of all the development must be extremely fast, to shorten the cycle from the point where the developers gather new requirements from the users to the point where the prototype supporting these requirements can be put to the test. Secondly these developments projects are typically characterized by an extreme degree of uncertainty. Given that the prototyping is usually done in an environment where nobody really knows “what it is we need”, it is very likely that what one day was the perception of the system will have completely changed shortly after. Thirdly these projects are typically characterized by a high degree of parallelism, where several people need to work on the same parts of the system in parallel, e.g. a cooperative designer redesigning a part of the UI while another developer is changing the corresponding code.

What our experience suggest is that when developing under these challenges one is more or less forced to base the prototype on a good and solid system architecture. If the architecture is robust it is a lot easier to add more and more functionality, screens etc. as the prototype evolves to cover more ground. If the architecture is flexible it is easier to allow experiments with the prototype when trying out new ideas. If the architecture is well defined and divided into sensible parts it is much easier to develop on parts of it in parallel.

This is not something new though. Experienced developers will probably have no problem saying that a system architecture should be robust, flexible, well defined and divided into sensible parts. What is problematic is that when doing evolutionary prototyping achieving these properties becomes so much harder to obtain exactly for the reasons that urged us to obtain them.

What we present in this experience paper is a practical example of how we managed this “walking on a tightrope”. We present how we to begin with outlined an initial architecture that had most of the desirable properties, and how working within it was done for the first three to four months of the project. We then present how this architecture became too restraining and even in some ways problematic as our understanding of the system, the requirements and the work processes was increased. Then we present how the system was restructured to reflect the new understanding and to once again give a robust and flexible ground to work upon. It is our experience that allowing these explicit restructuring phases are worthwhile even when doing extremely fast development such as Rapid Application Development, and that focussing only on the graphical user interface and “hacking functionality” will not pay off in the long run, especially when doing long-term or evolutionary prototyping.

The structure of the paper will be as follows: We start with a short project description and then give a characterization of prototyping and especially the approach we took. When then give a short description of what we understand by the term software architecture and with this then lead onto the main part where we discuss our architecture as described above. Finally we discuss current status, future work and give preliminary conclusions.

2. Project description

The project, which is now known as the Dragon Project, was started in late January 1997, having the DEVISE research group and a globally distributed shipping company (which cannot be named for commercial reasons) as partners. The project, which is scheduled to end in June 1998, has so far resulted in a number of prototypes of a Global Customer Service System (GCSS). Through a business process improvement (BPI) project, the company has conceived the concept of GCSS as a means of supporting new and improved business processes. A major role of the prototype was to give a “proof of concept” of these processes.

The project has involved many iterations of prototyping, working within a highly distributed company (offices in some 70 countries) and a business with a high degree of complexity which makes the project a fertile ground for evaluation of tools and techniques for rapid evolutionary prototyping and user involvement in system development.

Participants in the project include, from the university: one project coordinator, one participatory designer, one usability expert, three full-time and three half-time object-oriented developers and one ethnographer. Participants from the business include senior management, business representatives from the major continental regions, administrative staff, customer service personnel and members from a private consulting company.

2.1 The problem domain

Shipping in the context of the Dragon Project may be described as the actual transport of cargo in containers. The prototype attempts to cover the provision and delivery of customer services, which the transport of containers relies upon. Work here includes the handling of interactions with customers in formulating prices for transport (quoting), in booking containers, in arranging inland haulage, in documenting ownership of cargo, in notifying the consignee of cargo’s arrival, and so on.

Currently, no global system or formal practice exists, yet GCSS should support coordination between sites employing some two and a half thousand people in over 250 offices in 70 countries on six continents. The main design issue in many respects thus became to develop a prototype supporting a global practice as well as a customer service that, while streamlined (i.e. operating with fewer resources), is both effective and efficient and at the same time respects local needs in the various regions.

2.2 Project history

The major events in the Dragon Project are summarized below in Figure 1. As the table illustrates an often reoccurring event is the review. A review of some kind has, on average, been held every two weeks during the 10 months of effective prototype development of the total 14 months. Before each major review a “frozen” version of the prototype was made and a presentation was then made of this version at the review. These reviews had as purpose to assess whether or not the prototype was on

the right track in terms of scope and functionality, and also to ensure that the prototype at all times kept a "global perspective" in so much as it should be acceptable by all the different regions in the company.

Besides these formal sessions there were also a large number of prototype sessions involving actual end users coming from both managerial and daily customer service positions. These were done both locally in Aarhus, and as the table illustrates in a number of different local offices. For a further discussion of the development process see [3].

Month	Event
February	Start of phase one; meeting at Copenhagen
March	Start of development with business representatives, BPI representatives, and ethnographer
March	Major review
March	Prototyping sessions with regional business representatives
March	Ethnographer goes to Hong Kong and US
April	Major review
May	The prototype is presented to the company's executive body, where it is very well received.
May	Phase one ends, and a major version of the prototype is delivered.
August	Phase two of the project is planned
September	Meeting at Copenhagen – development infrastructure in place
September	A major restructuring is made
October	Prototyping sessions in UK
October	Prototyping sessions in Sweden
November	Prototyping sessions in Singapore and Malaysia
December	Review based on experiences from Asia
January	Major review
January	A major restructuring is made
February	Prototyping sessions in USA
March	Major Review
July	End of phase two

Figure 1. A short history of the Dragon Project

2.3 Tools and code

The concrete results achieved so far in a bit more than a year 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, implementing over 15 individual, fully functional components, and having well over 50 screens.

The development platform is Windows NT, and the main software engineering tools used are a CASE tool, a GUI builder, a code editor, a persistent store and a concurrent engineering tool. The first four tools are part of the Mjølner System ([1]; <http://www.mjolner.com>) and the fifth tool used was concurrent versioning system [5]. The programming language used is BETA [12]. Further description of the tools will be given where relevant.

3. Prototyping

The usage of prototypes in computer science has its roots in an acknowledgement of that building computer system from pure paper descriptions without any intermediate embodiments is a problematic endeavor. Recognizing this, prototypes were introduced in computer science inspired by their success in engineering disciplines [8]. As Grønbaek also points out in [8] the term is not used in a coherent way though. First of all prototypes are seldom used in the same way as their counterparts in engineering. Where the prototypes in engineering are introduced late in the process just before mass production where they are supposed to represent the final product, software prototypes are usually used much earlier in the process and they are often not supposed to represent any final or complete stage.

As an overall classification we can describe software prototypes according to the following characteristics [7]:

- *Horizontal vs. vertical prototypes.* Horizontal prototypes focus on what is visual in the user interface, and the underlying functionality is often “hacked”, simulated or even left out. Vertical prototypes have selected areas where the functionality is implemented in (almost) the detail of the final system.
- *Exploratory, experimental and evolutionary prototyping.* In exploratory prototyping the prototype is used as a means of requirements gathering, in experimental prototyping the prototype is used to determine whether an already perceived idea is adequate and in evolutionary prototyping the prototype is gradually developed into the final system either in an incremental way in which more and more is added stepwise or by iteratively refining what has already been covered.

3.1 The RAD approach

One of the most widely used prototyping approaches used in the software industry is the Rapid Application Development (RAD) method [9], [16]. This method takes the evolutionary approach to prototyping, and focuses on dividing the whole system development into a number of cycles where the system is extended in an incremental way. In this way the RAD method only focuses on prototypes that are highly vertical, as each of the cycles are supposed to deliver a fully functional unit.

The experimental approach can also be used as an element in RAD projects, by using the so-called usability prototypes, which has as purpose to ensure that the final system “will be as easy and intuitive to use as possible”.

3.2 Our approach

The approach to prototyping that we took in the project used several of the characteristic forms described above in an intermixed manner. First of all it was *evolutionary*. The prototype has not been used for quick “code-and-throw-away” experiments, but has rather been used over a longer period of time where it has been gradually refined and extended. These extensions were both incremental as we added more and more areas covering new work processes, and evolutionary as we refined what was already there.

Being evolutionary in the sense that we aimed for the final system, the prototype was also very *vertical*. Not only did we implement a large amount of functionality; we also focused a lot on architectural elements such as an object model and complex storage issues. There were also a few elements of the prototype that were constructed only in a *horizontal* way though, but these were in the minority.

Finally we also used *exploratory and experimental prototyping*: As the ethnographer on the project provided input to the understanding of current customer service practice the participatory designer explored how the prototype could support the future work through numerous experiments.

Needless to say having used all these different approaches the development was highly *parallel*, and at any time a number of different competencies would be working on the prototype in a number of different manners. Furthermore the development was characterized by a very *rapid approach*. As described above reviews, being minor or major, were often held and the always-pressing deadline of the next review encouraged and even sometimes pressed the developers to rapid development.

4. What is architecture?

In [2], Beck suggested different definitions of software architectures including

- How one explains a working system (post hoc)
- A characteristic relationship between entities in a system (academic)

The project’s approach to software architectures can be seen as being both post hoc and academic. Although explicit architectures has been decided on (academically) throughout the project the very nature of evolutionary, rapid prototyping made adhering to a strict architecture not always desirable. This has called for (post hoc) analyses of the architecture at various stages of the project, revealing the need for evolution and restructuring of the architecture.

Furthermore we will use the following concepts in the discussions of the architecture of the prototype [13], which divide the units of the architecture into

- *Component units*, i.e. collections of parts of a program that constitute a bounded part of the programs and that have a well-defined responsibility
- *Process units*, i.e. a coherent sequence of operations during execution of a system

5. Initial Architecture

Actual coding of the prototype started two to three weeks into the project. In the light of the given time constraints it was clear that the code from the beginning would have to be structured in a way so that all six developers could work in parallel on component units that had as few dependencies as possible. This obviously made architecture a very explicit issue right from the start of development. The consequence of this was that no actual coding started before the initial architecture described in the following had been outlined by two senior developers. It was also clear, though, that initially we neither could nor should aim for a fully detailed architectural design. The reasons for this were that we did not have the time to go into too much detail and that even if we had the time the very nature of the project was exactly that we were designing for a system of which we had *very* little idea of form and behavior. As it turned out this initially outlined architecture took its form rather fast and more or less stayed as our basis for the first three months of the project.

To give a better impression of why and how the architecture evolved in minor and major cycles the following will describe the initial architecture and the role of the constituent parts: The initial architecture consisted of an object model, a persistent store, a main program and for each major part of the problem domain (e.g. customer or outbound handling) three units that worked together in a way similar to the Smalltalk model-view-controller (MVC) user interface framework [10]. An example of this is the Customer Functions unit, the Customer UI unit and the Customer Controller unit, respectively, as shown in Figure 2.

The object model. One of the implications of combining a rapid evolutionary prototyping approach with object-orientation has been a more explicit focus on constructing a model of the problem domain. We did not start out *only* focusing on the design of screens and functionality, as has perhaps been the tendency in prototyping projects, but we also from day one initiated an object modeling activity. However, when doing rapid prototyping it is probably more relevant than ever to avoid the pitfall of “analysis-paralysis”. Prototyping *is* requirements gathering. Therefore one should not attempt to build more than a preliminary model before actual coding starts. In our case such a model was constructed in a couple of weeks and was then gradually extended as further domain knowledge was gathered through the prototyping process. In fact when coding started the extent of the domain model was about twenty-five classes covering basic business areas such as quoting (pricing) and booking and only the most basic attributes had been added. Three months later

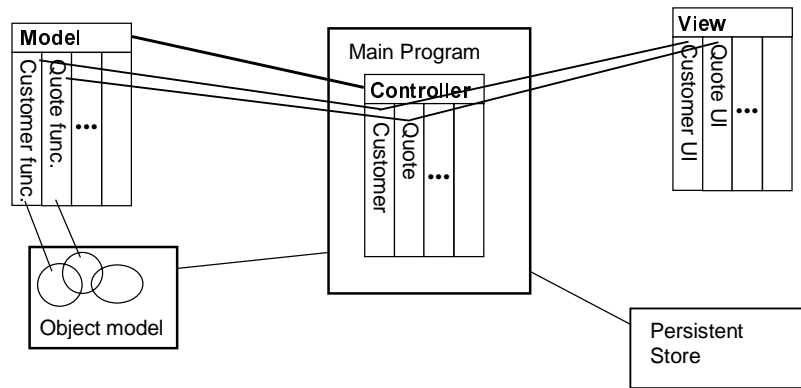


Figure 2. Initial architecture of the prototype

the model contained 40 classes and a substantial number of attributes, operations and relationships¹.

As can be seen from Figure 2, the object model of the problem domain forms a separate entity of its own right in the prototype. It was kept *purely* as a model of the problem domain encapsulated from implementation specific classes. By doing this, the object model served as a constant common frame of reference between members of the developer group and to some extent also between developers and members of the business.

The persistent store. Being able to demonstrate a prototype to users using real data is important in most prototyping projects, [4]. In our case e.g. real sailing and customer data was used throughout the process. This made users able to relate the prototype to current practices and, on top of that, using the *same* underlying data in demonstrations facilitated traceability of changes in views and/or functionality to users. Consequently making data persistent was an important aspect to be dealt with even in early design and implementation of our prototype.

In BETA, persistence is an 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, with the storage mechanism being limited to usage in a single-user setting, other storage mechanisms were explored and utilized at later stages so that the prototype was able to support and demonstrate multi-user scenarios.

The MVC units. The most important part of the main program was the controllers that combined the corresponding views and the models. Technically speaking, the controllers were specializations of the views, in which events from the views were bound to the corresponding functionality located in the model units. Note that the word “model” is used in two ways: as designating the object model and as designating

¹ For further details see “Evolution of the object model” below.

the MVC model unit, relating the functions and the underlying object model. The Customer Functions unit, e.g., operates on the part of the object model that relates to customers.

The user interface. A basic guideline in the design of the user interface was that both work processes and basic material handled within these processes should be visual. On the other hand the architecture was also meant to be representative in the sense that component and process units should reflect the work processes and material of future practice. As discussed later this guiding “representative principle” showed its strength in a number of situations.

Figure 3 illustrates how the object model and the functions are very visible in the user interface. By selecting an upper tab the user works with a unit of the application relating to an area of the problem domain – in this case outbound handling. Visualized in the user interface is part of the object model so that the user works on material pertaining to the object model. Also visible are process buttons showing state of

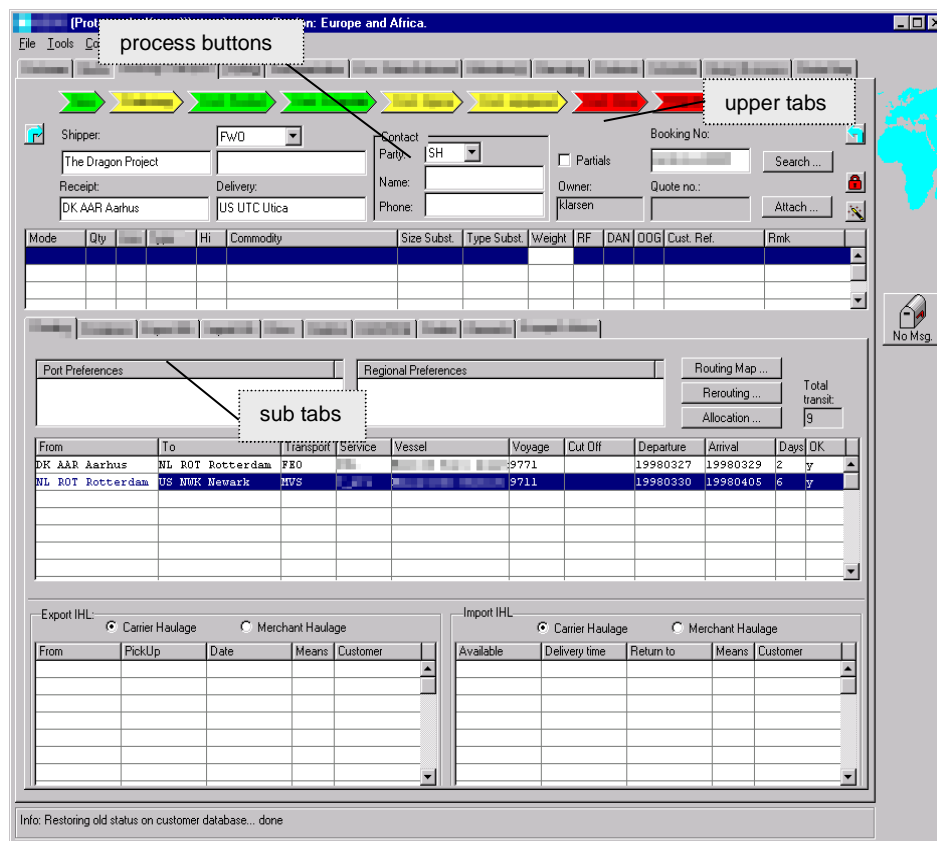


Figure 3. Screenshot of outbound handling. (Image blurred for confidentiality.)

completion of a work process and at the same time providing an interface to functions of the prototype.

This retrospective description of the initial architecture may indicate that we got it right in the first try. However, as aforementioned after the first major iteration, three months into the project, some problems in the basic structure implied that a restructuring was necessary (discussed below).

6. The process: working within the architecture

6.1 Organizing work

The strict division of the user interface, functions and the object model has been successful in reducing dependency and allowing multiple developers to work on the prototype concurrently.

In the first phase of the project work was roughly organized according to the following scheme:

- the cooperative designer created and designed user interfaces using a GUI builder
- two developers were responsible for creating programming interfaces to the user interface modules and for programming advanced GUI elements that could not directly be created using the GUI builder.
- three developers programmed functionality and made necessary changes to the problem domain model
- one developer programmed components that were simulating interfaces to legacy systems.

Furthermore, in accordance with the representative nature of the architecture, people had areas of responsibility according to major areas of the problem domain. One developer was for instance responsible for the customer functions unit, another for the booking functions unit and yet another for creating programming interfaces to both customer and booking UI units. In a sense the initial schemes for division of labor could be said to utilize the well-defined architecture both vertically (UI/Functions/model) and horizontally (according to representative units). Later as the architecture evolved and developers got a firmer understanding of all component and functional units this work organization changed into a "vertical only" division. One developer would e.g. be responsible for everything having to do with 'customer': customer user interface, programming interface to this, customer function unit and the part of the model dealing with customers.

In the concrete organization of work each developer has his own copy of the prototype code, which means that he can develop concurrently with and independently of the others. The code is exchanged via a common repository using

the standard Concurrent Versions Engineering tool (CVS; [5]). Because of the division of labor according to the architecture, merge conflicts very seldom occur. The use of a concurrent engineering tool, like CVS, has been crucial in the project, because 6-7 developers had to work on the same application in many iterations over a short period of time often exchanging code many times a day.

As became evident when a new developer was introduced to the prototype relatively late in the process the well-defined architecture also made it easy to get a quick understanding of the structure of the relatively large prototype. This was further eased by the fact that the division into the individual unit followed the natural division into individual areas found in the problem domain.

In a setting like ours, a further, not unimportant, aspect of work organisation is the physical location of a development team. In the first phase of development, the development team was located in company offices at a customer service site. While not actually located with customer service staff, access to practice was greatly facilitated. Furthermore, the transmission of knowledge was greatly facilitated through locating all the developers in one room. This co-location supported internal awareness and coordination of development activities. It is an organization of work we have maintained in the following phases although we are now located at the university.

6.2 Evolution of the user interface

An inherent characteristic of evolutionary prototyping is that functionality is added or changed many times. Since the functionality is accessed through the user interface, it is necessary to be able to make changes to the user interface and the underlying functionality, while at the same time preserving other parts of the user interface and functionality. In other words: it has been important to be able to make incremental changes to the user interface. The architecture and the GUI builder helped in this process.

The GUI builder was used to create the user interface in a direct manipulation graphical editor. The user interface can be modified via its graphical representation or via textual code. Initially the user interface was used for discussions only, although code was generated automatically. Changes to the physical appearance of the user interface, i.e. the types of user interface controls used and their concrete layout, were typically made in the graphical editor. The basic functionality of the user interface and the interface to the object model and functionality layer were, in contrast, typically changed using the code editor. Due to reverse engineering and incrementality it was easy to alternate between using the two tools.

The user interface was often used as means for organizing and/or coordinating activities between the cooperative designer and the OO developers. The cooperative designer most often made the initial user interface design and the OO developers in collaboration then further elaborated it with the cooperative designer. Again due to reverse engineering it was possible for the cooperative designer to make changes to the user interface throughout the process, even very late in a prototype cycle.

6.3 Evolution of the object model

Changes in functionality may of course require changes to the object model, especially when new areas of the problem domain are introduced in the prototyping process. The model evolved along two dimensions: *horizontally* and *vertically*. Horizontally the model was gradually extended to cope with new areas in the problem domain. Proceeding in this manner, both the model and the prototype were mutually elaborated. This was also the case vertically. The first iterations of a class or collection of related classes focused on data and the relationships between the classes. In the following iterations the focus moved to high level functionality (the business functions), which introduced methods to the classes. Development of new business functions occasioned many amendments to the relevant parts of the model. Amendments primarily consisted in the addition of attributes and methods, modification of existing attributes and adjustments in class relationships. Concurring with [4], it is in our experience very important that construction of a problem domain model from the beginning is considered part of the prototyping process: The object model evolves as the application evolves and as a part of the application.

As the object model was created in the CASE tool using the UML notation [14] the corresponding BETA code was automatically and incrementally generated. Although code was thus available at any time, the early versions of the object model were only used in the form of UML diagrams, as the emphasis was on communication and discussion. Like the user interface the object model can be modified via the graphical representation or via the textual code. As emphasis moved to code and redesign cycles, object model changes were mostly made in the code editor - only structural changes requiring overview of the object model were made in the CASE tool. When the main emphasis is on coding, it is often more convenient to do the changes in the textual representation, especially when the changes are at a detailed level. Using the reengineering capability, the UML diagram was recreated from time to time and posters of it were made for discussion.

7. Evolution of the architecture

As described above the initial architecture was very helpful in organizing development work. The division of labor was well defined because it was connected to the architecture and the initial architecture more or less stayed unchanged in the first phase of the project. However, as development went on and the prototyping sessions gave a better understanding of the problem domain and especially the required functionality it became evident that the architecture was pushed to its limits. We identified two types of problems that called for a new architecture:

- 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
- work within the existing architecture had shown annoyances, as e.g. unclear separation of responsibility and inconvenient dependencies between different

parts of the prototype, or inconsistent naming of functions, 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 functionality across component units. Until late in the first phase of the project, functionality required was very local to the corresponding part of the prototype, outbound handling functionality e.g. was only used local to the outbound handling unit. The problems became apparent, as a sort of “compound” function that could perform a number of existing functions on a bulk of business objects was needed. This compound function did not occasion any major reconstruction of the object model, but only required reuse of a great part of the existing functionality located in different units. However, the existing architecture did not provide the proper abstractions for the reuse, because the functions were not explicit in the architecture. Due to constraints of time the initial implementation of the compound function employed copy-n-paste reuse of existing functionality.

An example of the second type of problem is that turn-around time (edit/compile/run cycles) in the initial architecture increased as the prototype grew bigger due to awkward dependencies between the parts of the prototype. This had the effect that recompilation time was high because changes to the interface of a UI or Controller unit (of e.g. the outbound handling module) caused recompilation of not only the corresponding Functions unit, but also all the other Functions units. The main program with the controllers became a bottleneck, and as the prototype grew bigger the recompilation time became a limiting factor in the rapid prototyping process.

Because the first phase was very time-pressed we did not have the time to even think about restructuring the architecture, but after the first phase we took the time to improve the architecture.

7.1 Explicit restructuring phases

After the first hectic three months of the project the initial architecture was pushed to it's limits and we had to redesign the architecture before we could continue. Therefore an explicit restructuring phase was put into the plan.

In the second phase of the project we went through several restructurings. There were two major restructurings, which affected the overall structure of the prototype. The other restructurings were either local to certain parts, or global but could be accomplished incrementally with respect to the different parts, i.e. the changes could be made in one part independently of whether the changes had been made in other parts.

In the first major restructuring we designed and implemented a component architecture (see Figure 5 in the next section). In this architecture the UI, Controller and Functions units of a particular module (e.g. outbound handling) were gathered together in a single component having an abstract interface. Consequently *other* Functions and Controller units no longer depended on the UI unit of that particular module, eliminating the need for recompilation of those other units when changing the particular UI unit. Also, the new architecture enabled us to implement the

mentioned compound functions using the cleaner abstractions, which were “immediately” present in the new architecture.

The hectic nature of the project influenced the way the restructurings were accomplished. The project was organized with a number of iterations, each having a major review at the end, which was considered very important. Consequently, the focus before a review was on getting the prototype to work according to the planned changes. In the beginning of each cycle we took the time to code according to our conventions and make the proper abstractions. However, since the cycles were very short (especially in the first phase of the project or when development was a part of an on-site prototyping session) we had to relax the rules and conventions in order to meet the deadlines e.g. by employing copy-n-paste reuse of code. In general the last minute coding still respected the overall architecture though.

After a review, when there was less pressure on development, each programmer typically spent some time on going through his code to clean it up. Furthermore, the development group discussed whether the actual architecture of the prototype was good enough, and whether it could be improved. Actually, these discussions of the architecture continued way into the next iteration, but issues arising late in the iteration were typically postponed till after the next review.

7.2 Practically accomplishing restructuring

As mentioned above, there were both major and minor restructurings of the prototype in the project, and the practical accomplishment of a particular restructuring of course depended on the size and nature of that restructuring.

The major restructurings concerned the whole prototype, and hence development could not go on as usual. It was necessary to set aside some time to do the restructuring without doing ordinary development at the same time. In this way the major restructurings in reality meant temporary interruptions of development. Depending on the nature of the restructuring, a number of developers were assigned to the job, and they focussed entirely on this particular job. The restructuring usually took approximately between a couple of days and a week.

The minor restructurings could be done as part of the ongoing development. Local restructurings were done by one person, typically the person who made some shortcuts in the code in the hectic days before a review. Global restructurings that could be done independently in isolated parts of the prototype were delegated to all developers and the developers could choose to do them when it was convenient in their current work.

7.3 Tool support in the restructuring process

Important in doing the major restructurings was the structure editor, see Figure 4, with its abstract presentation of code and editing operations that can be used on arbitrary abstraction levels [15].

Abstract presentation gives overview. In the screen dump in Figure 4 a part of the booking component is presented. The three dots (...) which is called a contraction indicate that details are left out. When double-clicking on a contraction the details are shown. In the example `theCanvas` is an instance of a specialization of a class `bookingView`. If this contraction would be opened 94 similar declarations with contractions would appear, and the fully detailed code in this window is 923 lines of code.

High-level operations. Editing in the structure editor can be performed at any abstraction level. This means that it is easy to move even very large blocks of code and furthermore the language-oriented structure editor ensures that only syntactical complete parts of the code are moved or copied. As an example think of cutting, copying or pasting the whole definition of `theCanvas` by simply selecting the one-line abstraction.

Semantic browsing. In the restructuring process semantic errors were inevitable, but semantic browsing made it easy to correct the errors. The semantic browsing facility

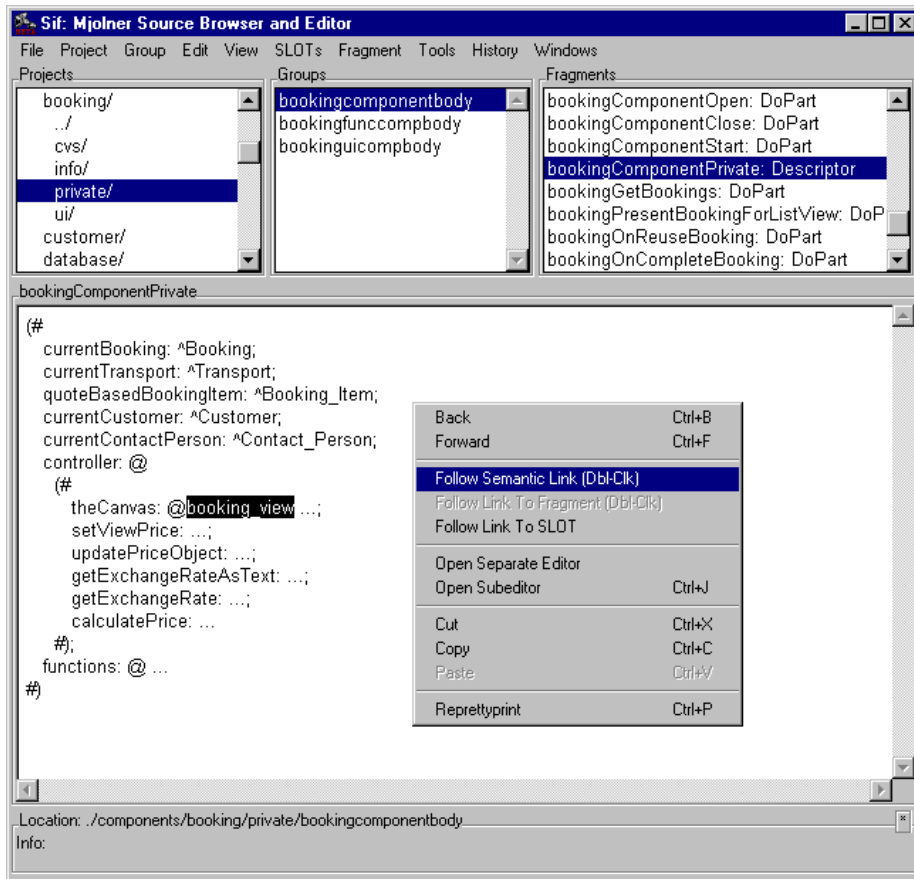


Figure 4. Abstract presentation of the parts of the booking component

lets the programmer follow a hyper link from a name in the program to it's declaration. Semantic browsing also helped in predicting the consequences of planned restructurings. In the screen dump the popup menu shows how the semantic link can be followed to the definition of `bookingView`.

Fragmentation. Support for automatic division of program parts into interface and implementation parts was useful. This is a BETA-specific functionality, which is supported using the fragment system of the Mjølner environment.

8. Current architecture

After a number of iterations of changes to the architecture of the prototype, the architecture has evolved into what is schematically shown in Figure 5 below. Changes in functionality are now less problematic, as the architecture now allows us to deal with complex issues emerging from prototyping sessions.

It should be noted however that we do not consider the current architecture the final architecture. The architecture discussions also included e.g. use of pre- and post conditions, coding and naming conventions. Until now we have not had the time to implement all of this.

8.1 Component Objects

The single most important feature of the current component architecture is its division into relatively independent *component objects*. Typically a component object represents one or more work processes in the problem domain. Each component object contains all functionality, controller, user interface (if any) and relations to other components necessary to implement the functionality pertaining to that

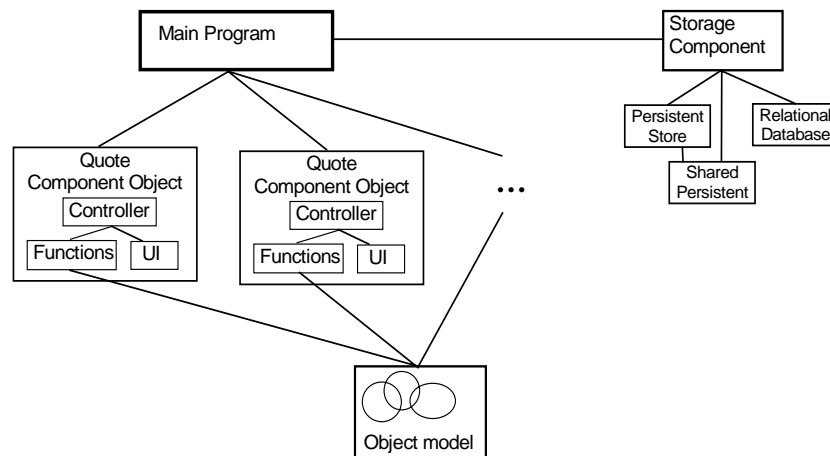


Figure 5. Current architecture of the prototype.

component object, i.e. each component object constitutes a whole.

To the outside, component objects consist of two abstract interfaces: one containing the operations used by other component objects, and one used by the main program only in order to manage the component (e.g. doing initialization and displaying the component object's user interface).

The component object architecture reduces the dependency between the different parts of the prototype: changes in one component object typically only affect the component object itself. Only if the changes are made to one of the interfaces of the component object will other parts be affected. In both cases it is inevitable that the parts using the changed interface need to be recompiled. Conversely, only the parts actually using the changed interface need recompilation.

An interesting future experiment will be to have the prototype use COM-objects [11] integrating the prototype with existing Windows NT applications while still maintaining a focus towards the object model and thus the problem domain.

8.2 Storage component object

As explained earlier, there was a need for making objects instantiated from the object model persistent right from the start. However, the actual medium of storage changed during the project from a single user persistent store to shared (multi-user) persistent store and, partly, to a relational database. This change of storage media during the development process made it desirable to obtain an encapsulation of the persistence making the actual media used for storage transparent to the application.

With the current architecture a component object of the prototype, namely the *storage component object*, thus handles the problem of storing the objects. Initial access to persistent objects goes through the storage component object, but after that, following references to other persistent objects is transparent independent of storage media. And whenever changes are made to persistent objects, the storage component object's *transaction* mechanism is used, making saving very easy.

The possibility for changing storage mechanism "on the fly" has also made it possible to provide the company with numerous stand alone version of the prototype using the single user persistent store.

8.3 Fine Points regarding the current Architecture

During a trip to Asia, e.g., the power of the component object architecture was experienced, as it was actually possible to add major component objects to the system "on the fly". In Singapore the prototype was modified according 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 prototype 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.

The success of these major additions was largely due to the fact that we during the project have striven to reach the following two goals with the prototype architecture:

- *The architecture should consist of components with low independence.* As the prototype grew larger and more complex turn-around time rose because of the dependencies of the architectural units. The following restructuring moved the architecture of the prototype towards a more separated architecture that proved extremely useful in the later phases of the project. A strict separation of user interface, object model, storage, and components was necessary in the prototyping project.
- *The architecture should be representative.* Main elements of the process and component architecture can be said to represent concepts in the model domain. Just as the object model reflects the problem domain, the user interface reflects the use domain and the functionality reflects the work processes, the components of the architecture has a direct relation to problem domain tasks as e.g. quoting and outbound handling.

9. Status and Future Work

After five major iterations, the prototype was appraised and approved by the company’s highest executive body. This means that a production version of the customer service system is now being developed. In this context the prototype is being used as the primary input for requirements specification and the object model developed as part of the prototype is being used as one of the main resources for defining the data layout of the production version. Furthermore an enterprise architecture has been defined in a stream running parallel to the prototyping stream and an implementation strategy² has been outlined. The implementation strategy is amongst others based on the component objects identified in the prototyping project.

Further work in the context of the prototype includes investigations of converting the current component object architecture into an architecture using COM-objects³ [11], which will also allow for tighter integration with existing applications on the Windows platform.

A number of experiences have been made with the tools used. In order for multiple developers to work on the same graphical representation of the object model the CASE-tool must be further developed to handle these situations. Furthermore, initial

² In this connection “implementation strategy” is the strategy for *what to develop when and when and where to install and use it* in the real world setting.

³ Currently another part of DEVISE research group are investigating and implementing support for usage of COM from within the BETA language.

investigations, [6], has shown that generation of the programming interfaces used between object model, functions and user interface can be generated automatically.

10. Conclusion

As we pointed out in the introduction developing software prototypes is not an easy venture. We also claimed that part of the secret in achieving a success, from a software-engineering point of view, lies in a strong focus on the architecture of the prototype.

We have identified and discussed a number of reasons for and properties of a good architecture for software prototypes:

- The evolutionary character of the development means that one cannot go about doing code-and-hack cycles, as the prototype should eventually evolve into the finished product. This requires a well-founded product that can best be based on a robust architecture.
- The experimental and exploratory style of our development approach means that the prototype must be sufficiently flexible to provide input to prototyping sessions, and furthermore the prototype must be able to provide this input quickly.
- As time is a scarce resource development must go on in parallel between the different competencies – this requires an architecture that is well structured and modular. These modules are as we experienced best divided into parts that make sense in terms of the problem domain.

But this is only one side of the solution though. The real problem lies in actually accomplishing this architecture. As the DSDM RAD manual puts it: *nothing is built perfectly the first time* ([16], p. 1-1).

What we have described is our approach, which shoves to be successful: One should in the beginning of the project not be too concerned about getting the architecture right. A preliminary architecture can be outlined in such a way that it structures and supports the work without being too time demanding in the first crucial phases of the project.

What we furthermore experienced and have discussed is that if this architecture becomes too restraining – and reflecting on our experience it probably will – then it is possible both technically and time-wise to restructure it. This requires though that the project managers allow the time for doing this, which should be a minor problem as it in our experience fully pays off in the long run. Also note that when doing these restructurings good tool support is very important.

Finally the evolution of the architecture can be considered as a prototyping process on the architecture itself. This architectural prototyping process has two objectives: 1). To provide the best possible framework for the prototyping of future practice, where the focus is on work processes and 2). To investigate the necessary properties of the final architecture in the production system. Examples of these properties are

exploration of storage mechanisms, transactional units, locking mechanisms etc. and integration with legacy systems.

Acknowledgements

This work was made possible by the Danish National Centre for IT-Research (CIT; <http://www.cit.dk>), research grant COT 74.4. We would also like to express our sincere thanks to all the people within the company who made this project possible.

Sincere thanks are also due to Ole Lehrmann Madsen for constructive comments on earlier drafts of this paper.

References

- [1] Andersen, P. Bak, L., Brandt, S., Knudsen, J.L., Madsen, O.L., Møller, K.J., Nørgaard, C., Sandvad, E. The Mjølner BETA System. In Knudsen, J.L., Löfgren, M., Madsen, O.L., Magnusson, B. (Eds.) *Object-Oriented Environments. The Mjølner Approach*. Prentice Hall, 1993.
- [2] Anderson, B., Shaw, M., Best, L., Beck, K. *Software Architecture: The Next Step for Object Technology (PANEL)*. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*.
- [3] 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*. To appear in *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP '98)*, Brussels, Belgium.
- [4] Connell, J., Shafer, L. *Object-Oriented Rapid Prototyping*. Prentice Hall, 1995.
- [5] Gnu, *Concurrent Version System*. <ftp://archive.eu.net/gnu/>, 1998.
- [6] Damm, C.H., Hansen, K.M., Thomsen, M. (1997) *Issues from the GCSS Prototyping Project – Experiences and Thoughts on Practice*, Department of Computer Science, Aarhus University, 1997.
- [7] Floyd, C. *A Systematic Look of Prototyping*. In R. Budde, K. Kuhlenkamp, L. Mathiassen, & H. Züllighoven (eds.), *Approaches to Prototyping*. Berlin: Springer Verlag, 1984, pp. 1-18.
- [8] Grønbæk, K. *Prototyping and Active User Involvement in Systems Development: Towards a Cooperative Prototyping Approach*. Ph.D. Thesis, Aarhus University, Denmark, 1991.
- [9] Kerr, J. and Hunter, R. - *Inside RAD: How to build fully functional computer systems in 90 days or less*, McGraw-Hill, 1994.
- [10] Krasner, G.E., Pope, T.P. *A Cookbook for using the Model-View Controller User Interface Paradigm in Smalltalk-80*. In *Journal of Object-Oriented Programming*, pp. 26-49, August/September 1988.
- [11] *The Component Object Model Specification*, Microsoft Corporation, 1995.
- [12] Madsen, O.L., Møller-Pedersen, B., Nygaard, K. (1993) *Object-Oriented Programming in the BETA Programming Language*, ACM Press, Addison Wesley, 1993.
- [13] Mathiassen, L., Munk-Madsen, A., Nielsen, P.A., Stage, J. *Objektorienteret Design*. (In Danish), Marko, 1995.
- [14] Rational Software Cooperation *UML Notation Guide Version 1.1*, <http://www.rational.com/uml/html/notation>, 1998
- [15] Sandvad, E. Hypertext in an Object-Oriented Environment. In Knudsen, J.L., Löfgren, M., Madsen, O.L., Magnusson, B. (Eds.) *Object-Oriented Environments. The Mjølner Approach*. Prentice Hall, 1993.
- [16] Jennifer Stapleton. *The Dynamic Systems Development Method Manual v2.0*. The DSDM Consortium, <http://www.dsdm.org>, 1995.