

The ‘Domain Model Concealer’ and ‘Application Moderator’ Patterns: Addressing Architectural Uncertainty in Interactive Systems

Klaus Marius Hansen and Michael Thomsen
Department of Computer Science, University of Aarhus,
Aabogade 34, DK-8200 Aarhus N, Denmark
E-mail: {marius, miksen}@daimi.au.dk

Abstract

Designing architectures for interactive systems is difficult: Many system failures are due to the inability to handle social and technical changes that occur during development. We present two architectural patterns for interactive systems. We applied these to a short-term and a long-term development project. In the WebviseLT Project, the conceptual model was modified to handle various extensions and to meet an emerging standard. In the Dragon Project, the user interface evolved significantly over a period of almost two years. The application of these two architectural patterns demonstrates how focusing on change, or more specifically on architectural uncertainty, can be crucial to the success of a project.

1. Introduction

Brooks’ [7] vivid description of the great beasts trying to escape the tar pits evokes the problems faced by development teams entangled in failure. Because design of software architecture is the highest level of design in software development, architectural uncertainty causes significant risks. One way of addressing failure is through ‘risk management’ as in Boehm’s *Spiral Software Development Model* [5], an iterative and evolutionary software development model. Through ‘risk management’ one should carefully consider what the uncertainties are, in which way these could constitute risks and what should be done to handle them. To avoid failure, special attention should be paid to architectural uncertainties.

For the purpose of our discussion of software architecture, we will use the definition of Bass et al. [2]:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

In order to overcome the limitations of box-and-line diagrams encountered in some software architecture discussions, we will use a variation of the notation introduced in Bass et al. [2]. Solid lines denote control and processing, whereas dashed lines denote data (Figure 1). We introduce the starred symbol to denote an architectural uncertainty. Furthermore, a shadow on a component indicates one or more components of that kind.

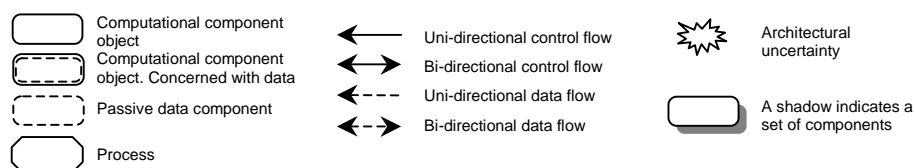


Figure 1. Software architecture notation.

1.1. Architectural uncertainty

We define uncertainty within a software architecture as follows:

An architectural uncertainty represents a lack of full understanding of an architectural structure.

Based on our experiences from two development projects, we discuss two important examples of architectural uncertainty: In the WebviseLT case, the Webvise hypermedia system was to be extended with typed links. This potentially required a change of the Webvise server's object-oriented model of hypermedia, while this model was about to be standardised. From the Webvise clients' view, the server's model thus presented an architectural uncertainty. In the Dragon case, the development process employed a high level of participation by users, which implied frequent changes to the user interface. These change requests forced us to continually evolve the part of the application that implemented the user interface. Thus, the user interface presented an architectural uncertainty with respect to the components that communicated with it.

The structure of the paper is as follows: In section 2, we present and discuss the two case studies of object-oriented system development; each addressing a different architectural uncertainty. Section 3 presents two architectural patterns, and section 4 discusses how these patterns were applied in each case to address the architectural uncertainty. Section 5 concludes.

2. Case studies

Each of the following case studies involves an architectural uncertainty that posed problems in the software development process.

2.1. WebviseLT: Extending the World Wide Web with typed links

The *WebviseLT Project* is a short-term research project that has currently run half a year. It extends the Webvise hypermedia system [23] that provides open hypermedia facilities for third party applications. In open hypermedia systems, structures such as links and anchors are external to the documents they link. This means that users may link from and anchor in documents they do not own. Unlike most other open hypermedia systems, Webvise also integrates with the World Wide Web. The World Wide Web is a single homogenous space of information, and its sheer size magnifies the classical problem of "being lost in Hyperspace" [14].

Link types, or more generally types for objects in hypertexts, provides the user with a manageable conceptual model in a hypertext [29]. Adding link types, also to the World Wide Web, is thus useful:

- for adding (user-defined) semantics to a hypertext,
- as a means of increasing support for user orientation by making the semantics of structures explicit, and
- as focal points for computer based analysis and synthesis of hypertexts.

Based on these considerations, part of the WebviseLT Project aims at extending the Webvise system with facilities for creation, manipulation, and use of typed links in media supported by Webvise, including World Wide Web documents [24].

Architectural uncertainty in the WebwiseLT Project: Figure 2 shows the Webwise Architecture. The main parts of the Webwise system are the Webwise Client processes, the Webwise Server process, and the external applications. The server, written in BETA [25], implements an object-oriented model of a hypertext consisting of objects such as links and anchors. The client provides a generic interface for general hypermedia functionality such as link following and browsing. Also, the client implements communication with external applications such as Microsoft Word or Microsoft Internet Explorer, effectively enhancing these with hypermedia functionality.

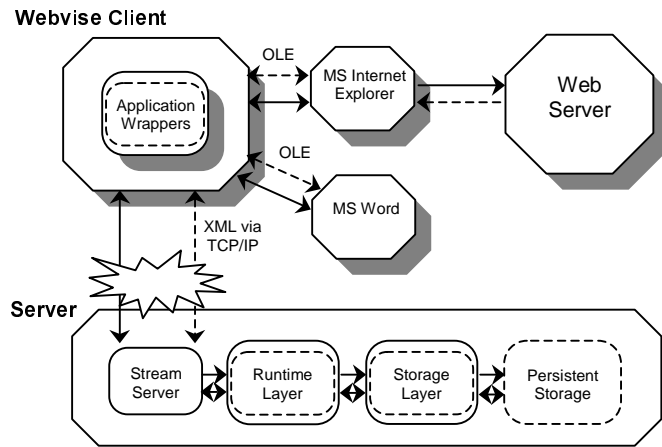


Figure 2. Webwise architecture

To support typed links, it would seem ideal to extend the storage layer of Webwise by creating a specialisation of the link class for each type. This is, however, not a very good idea since the future architecture of the server is uncertain: The server component will be replaced with the Construct system [35] in order to comply with a recent standard of the Open Hypermedia Systems Working Group (OHSWG, <http://www.ohswg.org/>). This architectural uncertainty represents a major obstacle to the introduction of typed links, since the server determines the conceptual view of a hypertext that the clients need to present to the users. Thus, it becomes important to introduce typed links in a way such that the developed functionality can coexist with both the original Webwise server and the new Construct server.

2.2. Dragon: Prototyping a global customer service system

The *Dragon Project* was initiated in 1997 as a joint project between a large, globally distributed container shipping company and University of Aarhus, Denmark. Over a period of more than 18 months, the goal of the project was to create a series of prototypes of a worldwide customer service system. The system should thus support handling of interaction with customers, e.g. in formulating prices for transport of containers (quoting), in booking containers, in arranging inland transportation of containers, in documenting ownership of cargo, and in notifying the consignee of cargo's arrival. The development process was iterative, exploratory, and multi-perspective [12].

Figure 3 shows how each concern of the project was treated iteratively. Booking, for example, was the subject of more than six reviews, with iterations in more than six sub-phases of development. Moreover, in-between each review, we explored several ways of treating the concerns. The development project involved several perspectives: object-orientation, participatory design [21], and ethnography [4].

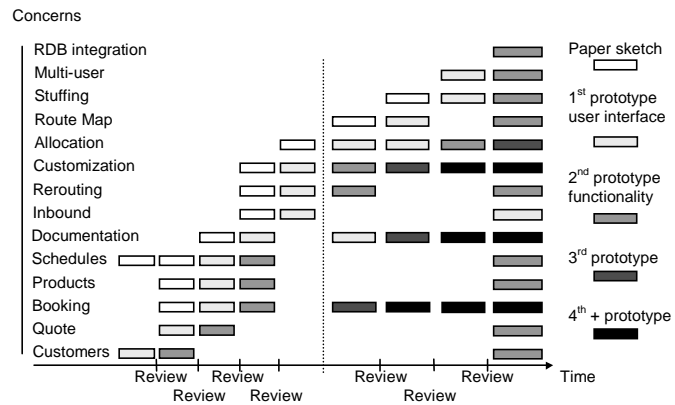


Figure 3. Evolution of concerns in the Dragon Project

The ethnographer worked on understanding and describing current practice, the participatory designer designed technological support for future practice with users, and the OO developers were responsible for the problem domain model, software architecture, and implementation [13].

Architectural uncertainty in the Dragon Project: Figure 4 shows a conceptual view of the initial architecture of the Dragon prototype. It consisted of four major components: Problem Domain Objects related to the shipping problem domain, Business Functions implementing functionality cross-cutting the Problem Domain Object structure, User Interface, and Persistent Storage.

The iterative and exploratory character of the development process meant that the user interface, problem domain model, and business functions all evolved throughout the project. For each area of the business that was covered, the problem domain model evolved horizontally to cover most phenomena of interest within a short period of time. In this way, the different areas of the problem domain model quickly became structurally stable; later changes tended to be much more vertical in the sense of elaborating on existing classes and relationships rather than introducing new ones. The user interface, however, continued to evolve more radically, as the participatory designer worked with users on new areas of the business. In this way, the instability of the user interface was an architectural uncertainty: The relative difference in stability was problematic, and the development of business functions, problem domain model, and user interface had to take this difference into account. Most important, the architecture had to be designed so that the evolution of the user interface would not overly affect the rest of the system.

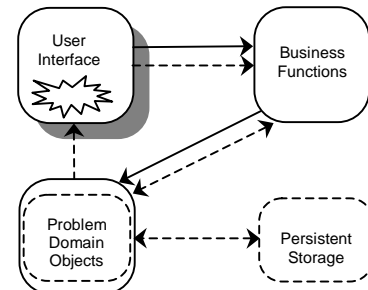


Figure 4. Conceptual view of the Dragon prototype architecture

3. Architectural patterns for interactive systems

Each of the case studies highlights an architectural uncertainty. In each situation, the uncertainty and potential risks were, in part, handled by designing a software architecture that anticipated the possible change implied by the uncertainty. In this section we describe these architectures. To give a more detailed understanding of the architectures, and to allow the architectures to be used in other contexts, we present them as *architectural patterns*. Architectural patterns, like design patterns [20], present a common solution to a common problem within a given context, which matches well with the notion of architectural uncertainty. Architectural patterns are not new. Well-known architectural patterns include those presented by Buschmann et al. [9], and Shaw [32].

3.1. ‘Domain Model Concealer’ and ‘Application Moderator’

This section presents the ‘Domain Model Concealer’ and the ‘Application Moderator’ patterns. Both pattern descriptions assume that an object-oriented approach to software development is used. In object-oriented development, the system must not only be technically sound, but it must also maintain a real-world reference, i.e. it must contain a proper model of what is referenced [27], [25], [34]. This model is often referred to as the *problem domain model*. Since we are dealing with interactive systems, another important part of the application is the user interface. These two parts of the application should be separated into two different parts or components, making each easier to understand and maintain.

In our pattern descriptions below, when discussing sample implementation, we will refer to the same *Financial History* application. This facilitates comparison of the two patterns. The, somewhat artificial, application enables tracking of financial expenses. The simple problem domain model is shown in Figure 5 and the user interface is shown in Figure 6.

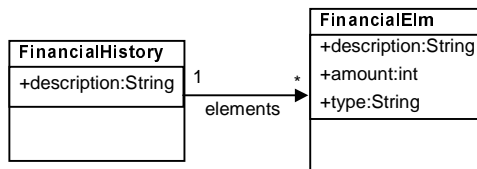


Figure 5. Problem domain model for the Financial History application

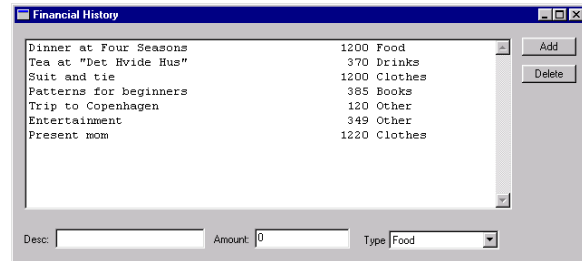


Figure 6. User interface of the Financial History application

3.2. Pattern format

The pattern format used to describe the architectural patterns in this article varies somewhat from the format of Buschmann et al. [9] and Shaw [32]. The former can be too verbose to efficiently communicate the essence of the patterns, and the latter can be too short to give sufficient understanding to actually apply the patterns. Figure 7 presents the pattern format, which may be viewed as a condensed version of the format in Gamma et al. [20] or as a slight variation on Brown et al.'s [8] 'deductive mini-pattern' template.

Name and thumbnail: What should the pattern be commonly known as? Followed by a short description.

Problem: What is the architectural problem that the pattern faces, and what are the main forces behind this problem?

Solution: What is the effective solution of the stated problem? This section includes a description of the pattern's high-level static structure in the form of a Unified Modeling Language (UML [31]) class diagram using packages.

Sample implementation: How could this pattern be applied? This section includes UML class diagrams.

Consequences: What are the benefits and liabilities of applying this pattern?

Figure 7. Architectural pattern format

3.3. Domain Model Concealer

The Domain Model Concealer architectural pattern divides interactive applications into a problem domain related part (the Problem Domain Model), an interface to this model (Domain Model Concealer), a User Interface part containing the user interface, and a User Interface Logic part that implements user interface related functionality by communicating with the Problem Domain Model through the Domain Model Concealer.

Problem: How does one design the architecture of an interactive system so that the user interface functionality is separated from the problem-domain related functionality? How does

one ensure that frequent changes to the Problem Domain Model do not require frequent changes to the whole application?

The following forces should be balanced:

- User interface functionality should be separated from problem domain related functionality, so that each part is easier to understand and maintain.
- Changes to the problem domain model should have no or minimal impact on the rest of the application.

Solution: The Domain Model Concealer pattern divides the application into four parts: User Interface, User Interface Logic, Domain Model Concealer, and Problem Domain Model. Optionally, a Testing component can be implemented. The overall structure is shown in Figure 8.

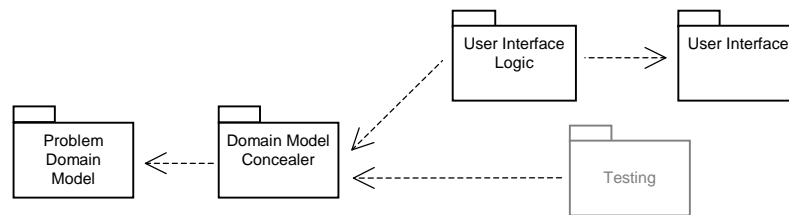


Figure 8. Overall structure of the Domain Model Concealer pattern

The User Interface component implements the user interface and consists of declarations and instantiations of the widgets in the user-interface of the application. The Problem Domain Model contains the domain-related classes and functions, and possibly other functionality. It can be implemented independently of the User Interface component. The Domain Model Concealer contains an interface to the Problem Domain Model component. The Domain Model Concealer is thus dependent on the Problem Domain Model component but independent of the User Interface component.

The User Interface Logic implements the functionality offered by the User Interface component by collaborating with the Problem Domain Model component through the Domain Model Concealer interface. This makes the User Interface Logic component dependent on both the Domain Model Concealer and the User Interface components. Finally, the Testing component allows testing of the Problem Domain Model component via the Domain Model Concealer component.

Sample implementation: An application of the Domain Model Concealer pattern may proceed as follows; the steps should not necessarily be taken sequentially. Part of the class structure of a resulting implementation is illustrated in Figure 9.

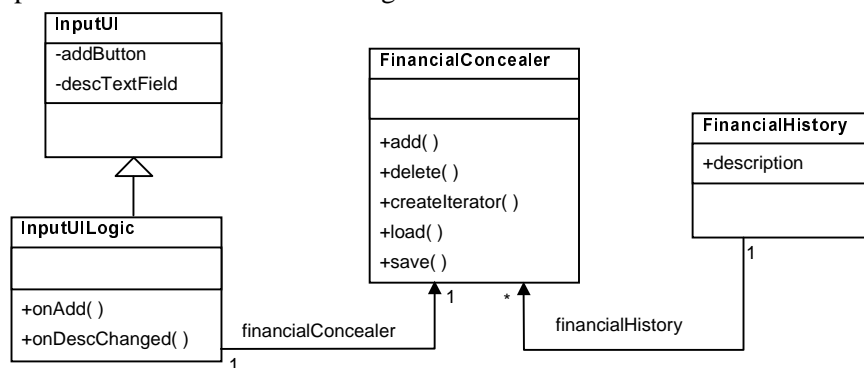


Figure 9. Class diagram for the Financial History application

1. *Implement the Problem Domain Model*

The Problem Domain Model is as described above.

2. *Implement the Domain Model Concealer*

The Domain Model Concealer is implemented as the `FinancialConcealer` class. In the Financial History application, operations to add financial elements (`add`), delete financial elements (`delete`), and iterate over financial elements (`createIterator`) are needed. The Concealer should hold a reference (`financialHistory`) to the object in the Problem Domain Model of which it is currently showing the state. Furthermore, methods to set the state of the Concealer (`load`) and to set the state of the Problem Domain Model through the Concealer (`save`) are needed.

3. *Implement the User Interface and User Interface Logic Components*

The User Interface component (`InputUI`) can be implemented using a user interface toolkit in the normal way. It contains the declarations and instantiations of the widgets and corresponds to what may be generated by a user interface builder. The User Interface Logic component (`InputUILogic`) implements the actual functionality of the User Interface by subscribing to events in the User Interface. To communicate with the Problem Domain Model it uses the associated `FinancialConcealer`.

4. *Optionally implement the Testing Component*

A Testing component can be created to systematically test the Problem Domain Model via the Domain Model Concealer.

Consequences: This architectural pattern has both benefits and liabilities.

Benefits:

- Separation of the parts that require domain knowledge and the parts that require user interface toolkit knowledge is supported.
- Change in technology is supported, since the User Interface and User Interface Logic components can be substituted by new components when the user interface toolkit changes.
- Testing of the Problem Domain Model and the Domain Model Concealer is facilitated.
- The User Interface and User Interface Logic components are shielded from changes in the Problem Domain Model, since they do not interface with it directly.

Liabilities:

- As the Problem Domain Model changes, either more work is involved in mapping between the Problem Domain Model and the Domain Model Concealer, or the Domain Model Concealer must be changed, which may require changes to the User Interface Logic.
- Multiple view consistency is not accommodated. The Observer pattern [20] can provide this by having the Domain Model Concealer as subject and several User Interface Logic components as Observers.
- Frequent changes to the User Interface also imply frequent changes to the User Interface Logic component.

3.4. Application Moderator

The Application Moderator architectural pattern divides an interactive application into a problem domain related part (the Problem Domain Model), a user interface

component (User Interface), an abstract interface to the user interface (User Interface Mirror), and an Application Moderator component that couples the User Interface Mirror with the Problem Domain Model and other functionality.

Problem: How does one design the architecture of an interactive system such that the user interface functionality is separated from the problem domain related functionality and such that changes to the user interface require minimal change to the rest of the system?

The following forces should be balanced:

- User interface functionality should be separated from problem domain related functionality, so that each part is easier to understand and maintain.
- Changes to the user interface should have as little impact on the rest of the application as possible.

Solution: The Application Moderator pattern divides the application into five components, Problem Domain Model, Application Moderator, User Interface Mirror, and User Interface. Optionally a Testing component may be implemented. The overall structure is shown in Figure 10.

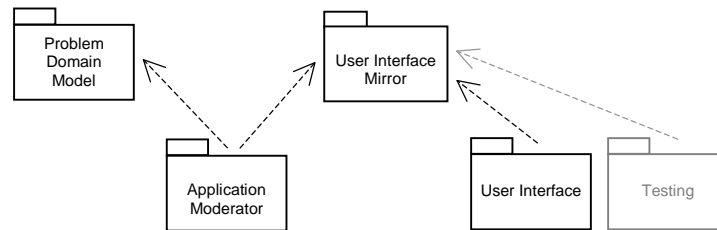


Figure 10. Overall structure of the Application Moderator pattern

The Problem Domain Model contains problem domain related data and functionality. The User Interface Mirror contains an abstract interface to the User Interface component. It consists of data members that reflect the state of the widgets in the user interface and event members that represent events that occur in the user interface, such as a button press. Furthermore, it contains two abstract methods, `setState` and `getState`, which should be refined to write the state of the concrete widgets into the data members (`getState`) and conversely (`setState`). The User Interface contains the concrete widgets, implements the `getState` and `setState` methods as described above, and calls event members as appropriate.

The Application Moderator connects the Problem Domain Model with the User Interface by subscribing to the events in the User Interface Mirror and thus moderates their communication. Upon invocation of the events, it can access the current state of the user interface by calling `getState` and then read the data members or it can set the state of the user interface by setting the data members and then call `setState` or it can do a combination of both.

Finally, the architecture allows for effective testing of most of the application. This is done by creating a Testing component that systematically sets the state of the User Interface Mirror, calls one or more event methods and then tests if the data members are in a correct state.

Sample implementation: An application of the Application Moderator pattern may proceed as follows; the steps should not necessarily be taken sequentially. Part of the class structure of a resulting implementation is illustrated in Figure 11 and Figure 12.

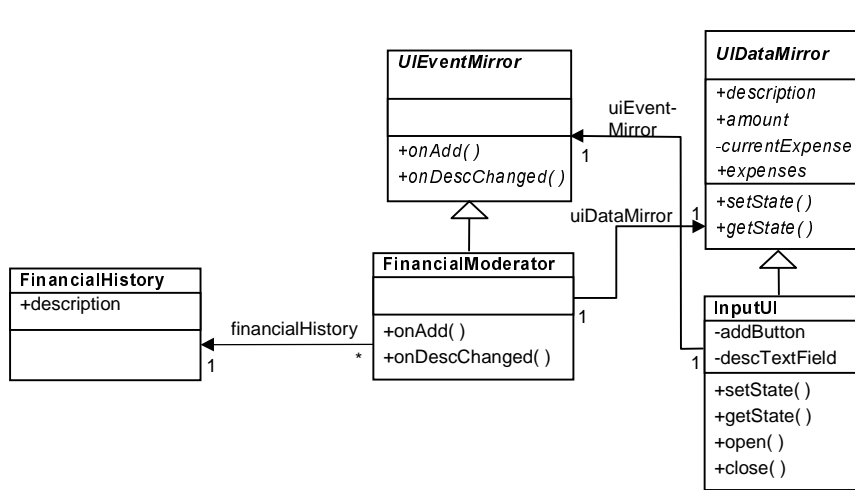


Figure 11. Class diagram for the Financial History application



Figure 12. FinancialExpense data interface

1. Implement the Problem Domain Model

The Problem Domain Model is as described above.

2. Implement the User Interface

The User Interface component (InputUI) can be implemented using a user interface toolkit and corresponds to what may be generated by a user interface builder.

3. Implement the User Interface Mirror

The User Interface Mirror reflects the User Interface and acts as an interface to it. The User Interface Mirror is in this sample implementation divided into two classes: UIDataMirror and UIEventMirror. The UIDataMirror provides a data interface (Figure 11) that reflects the data displayed in the user interface. The description and amount attributes correspond to the two text fields in the user interface of the Financial History application. The expenses attribute is a list of FinancialExpense objects (Figure 12). Each element of this list corresponds to an element in the list view of the application. CurrentExpense models the current selection of the list view. Moreover, the getState and setState operations are defined in the UIDataMirror class. The event members of the User Interface Mirror are implemented as a number of abstract methods on the UIEventMirror, with one function for each event of interest in the user interface. The onAdd method, e.g., corresponds to the event that the addButton was pressed.

4. Refine the User Interface component

The User Interface component, implemented in step 2, is refined to implement the setState and getState operations and to call the event methods in the User Interface Mirror. In this case the User Interface component binds the button-press events of the buttons to call the corresponding event members in the UIEventMirror associated through its uiEventMirror association.

5. Implement the Application Moderator

The Application Moderator connects the User Interface Mirror and the Problem Domain Model. In our example, the FinancialModerator binds the events in the user interface by implementing the event members in the UIEventMirror interface and reading and writing to

the data members in the `UIDataMirror`. Generally, the implementation of the Application Moderator should:

- Add methods, which map data between the Problem Domain Model and the data members in the User Interface Mirror,
- Refine the abstract methods of the User Interface Mirror that represent events of interest,
- Implement the general application functionality inside the appropriate event methods, and
- Implement multiple view consistency if needed.

6. *Optionally implement the Testing component*

An optional Testing component can be implemented by creating another specialisation of the User Interface Mirror that systematically calls the event members (e.g. in `UIEventMirror`) and then tests the state of the data members (e.g. in `UIDataMirror`).

Consequences: This architectural pattern has both benefits and liabilities.

Benefits:

- The User Interface component and the User Interface Mirror components are independent of the Problem Domain Model component.
- The Problem Domain Model component is independent of the User Interface component.
- The architecture supports effective testing. Regression testing of large parts of the application, e.g., can be done efficiently via a Testing component.

Liabilities:

- The interface to the User Interface represented by the User Interface Mirror component takes some time to develop and maintain.
- The architecture results in a minor overhead in terms of function calls and data conversion.

3.5. Pattern discussion

Known uses: Three-tier client-server architectures [33] divide (interactive) applications into three tiers: Database, Domain, and Client tier, according to External, Conceptual, and Internal schemas. In an interactive application with a thin client the representation of the domain on the client in many ways resemble a Domain Model Concealer, and the Domain and Database tiers resemble the Problem Domain Model in the Problem Domain Model Concealer Pattern. Fowler [19] discusses an ‘Application Facade’ architectural model in a non-pattern format. In this architecture, the Application Facade acts as a Domain Model Concealer and is used in the same way as in the Domain Model Concealer pattern. In section 4.1, we discuss another use of the Domain Model Concealer pattern.

The Devise Open Hypermedia System [22] uses the Application Moderator pattern extensively to divide user interface and the hypermedia data model in the client (runtime layer) of the system. Each dialogue in the system communicates with the non-client layers through a mirror of the hypermedia data model as in the Application Moderator pattern. In Microsoft Visual C++ [28], the Document/View architecture uses the Application Moderator to separate the user interface (View) and the domain model (Document). The class wizard in the environment generates implementations of the `setState` and `getState` methods of what corresponds to the User Interface Mirror. In section 4.2, we show another use of the Application Moderator pattern.

Relation to other architectural patterns for interactive systems: A large number of architectures have been proposed for interactive systems. Examples include Seeheim [30], Model-View-Controller (MVC [26]), Presentation-Abstraction-Control (PAC [15]), Arch/Slinky [3] and PAC* [11]. However, few of these have been treated in a pattern form. The most well known patterns are probably the MVC and the PAC patterns described by Buschmann et al [9].

Both the Application Moderator and Domain Model Concealer patterns complement the Model-View-Controller (MVC) pattern. The MVC pattern is mostly concerned with obtaining multiple view consistency. Whereas it decouples the Model from the Views and Controllers via the Observer, it allows the Views and Controllers direct access to the Model. Neither the Domain Model Concealer nor the Application Moderator allows this. In the Domain Model Concealer pattern, the Concealer shields the Problem Domain Model. In the Application Moderator pattern, only the Moderator, and not the User Interface Mirror, has access to the Problem Domain Model.

The Presentation-Abstraction-Control (PAC) pattern is different from the other three patterns. PAC has as its dividing principle a vertical split of the application such that it is divided into separate agents according to divisions found in the problem domain. Each of the agents contains their own user interface and functionality.

A main difference between the Domain Model Concealer and Application Moderator lies in what is decoupled. The Application Moderator architecture adds an interface layer to the user interface, thereby shielding the clients of the user interface from changes in it. The Domain Model Concealer architecture does almost the opposite: It adds an interface to the Problem Domain Model and in that way hides changes from its dependants.

4. Managing architectural uncertainty

This section discusses the architectural uncertainties introduced in section 2 and explains how these were handled in the two concrete cases with the help of the patterns presented in the previous section.

4.1. Deferring conceptual choice in WebwiseLT: Applying the Domain Model Concealer pattern

A common problem encountered in system development is changes in technology imposed by external driving forces. In the WebwiseLT case, this was experienced: The emerging OHSWG standard meant that the server component of the Webwise system was to be replaced. Since the server implements an object-oriented model of hypermedia, the introduction of typed links became problematic. Two choices were possible:

- Wait for the standard, then design the extensions, and subsequently implement the design, or
- proceed, temporarily ignoring the potential changes caused by the standard.

Both choices are problematic. Waiting would cause the project to fall behind schedule and ignoring the upcoming changes would lead to a potential re-implementation. We followed a third path. We decided to develop new functionality while the hypermedia model was evolving, and applied the Domain Model Concealer architectural pattern. Figure 13 shows the introduction of a Link Type Concealer as part of the Webwise architecture. The only way components in the Webwise client application can use link types is through the Link Type Concealer. This concealer thus provides both link type browsers and editors with a stable view

of link types. Also, external applications, such as Microsoft Internet Explorer, are enhanced with link type functionality via the Application Wrappers' use of the Link Type Concealer.

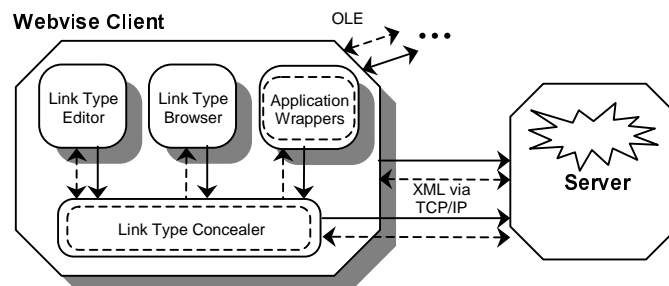


Figure 13. Introducing link types in Webwise

Currently, the link type hierarchy itself and link types are encoded in general attribute/value pairs stored in the hypertext objects. This has two consequences: First, the standardised model also specifies general attribute/value pairs on hypermedia objects. This means that after replacing the server, the only parts of the link type that has to be changed are the actual requests to the server. Second, this scheme should allow for a graceful introduction of first class types that will eventually extend the storage layer of the new standardised server.

4.2. Parallelism in the Dragon Project: Applying the Application Moderator pattern

User involvement in system development generally takes one of two forms. Either

- user involvement is primarily concentrated on requirements gathering at the beginning of a project, and acceptance testing at the end of a project, or
- user involvement and user testing is on-going and involves iterative development of the system.

Both approaches are problematic: User testing only in the end of a development project often causes missed deadlines due to users' rejection of the developed system. On-going user involvement, on the other hand, often causes missed deadlines due to evolving user requirements.

In the Dragon Project, user involvement was on-going, with change caused by evolving or emerging requirements that came from three major sources:

- Usability studies,
- participatory design workshops with users, and
- presentations to and discussions with managers and end users.

The frequent changes to the user interface throughout system development constituted a major architectural uncertainty. Figure 14 shows how the Application Moderator pattern addresses this by isolating user interface changes. The Business Function, Problem Domain Objects, and User Interface components are now no longer communicating directly. Instead, all communication with the user interface now goes through the User Interface Mirror component. This means that if changes to the user interface do not require changes to User Interface Mirror, the rest of the application will not be affected by the changes.

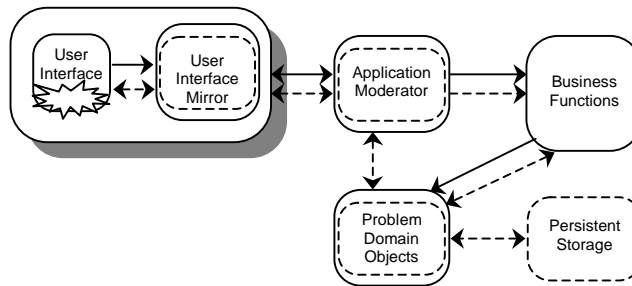


Figure 14. Isolating user interface changes in Dragon

In our case, changes to the user interface often required none or only small changes to the rest of the application. For example, several versions of a set of allocation user interfaces resulting from regional differences existed over time using the same User Interface Mirror. The choice of architecture was helpful: Although the extra interfaces were somewhat problematic to introduce in the context of rapid development, it more than paid for the extra work in terms of increased stability. The result was that the series of prototypes all were developed on schedule.

5. Conclusion

Software systems must evolve and change because of changing user requirements and external factors, such as emerging standards. This often leads to architectural uncertainties, which must be identified and addressed. This paper reports our experience from two system development projects that addressed architectural uncertainty, by constructing architectures anticipating change.

The two presented architectural patterns were used to isolate and thus handle the architectural uncertainties and both helped in the two projects being finished successfully and on time. Generally, the two patterns are concerned with the coupling of user interfaces and problem domain models of object-oriented applications. While no pattern can remedy the impact of every change, they can provide some independence: The ‘Domain Model Concealer’ pattern can help in shielding from changes to the problem domain model, and the ‘Application Moderator’ can decrease the implications of changes to the user interface. Thus, the architectures documented in the patterns can increase the flexibility of the application and help in handling certain kinds of architectural uncertainty.

6. Acknowledgements

Part of this work has been carried out in Center for Object Technology (<http://www.cit.dk/COT>) which is a project that has been partially funded by the Danish National Centre for IT Research (<http://www.cit.dk>) and the Danish Ministry of Industry. Thanks to our colleagues that participated in the two projects. Thanks also to Henrik B. Christensen, Aino Cornils, Ole L. Madsen, Henrik Røn, and Lennert Sloth for valuable comments that improved this paper. Special thanks to Wendy E. Mackay for improving our English.

7. References

- [1] Andrews, K., Kappe, F., Maurer, H. (1995). Serving Information to the Web with Hyper-G. In *Computer Networks and ISDN Systems*, 27(6).
- [2] Bass, L., Clements, P., Kazman, R. (1998). *Software Architecture in Practice*. Addison Wesley Longman.
- [3] Bass, L., Faneuf, R., Little, R., Mayer, N., Pellegrino, B., Reed, S., Seacord, R., Sheppard, S., Szczur, M.R. (1992). A Metamodel for the Runtime Architecture of an Interactive System. In *SIGCHI Bulletin*, 24(1).
- [4] Blomberg, J., Suchman, L., Trigg, R. (1994). Reflections on a Work-Oriented Design Project. In *Proceedings of Third Biennial Conference on Participatory Design (PDC '94)*. Chapel Hill, North Carolina, USA.
- [5] Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5).
- [6] Booch, G., Jacobson, I., Rumbaugh, J. (1998). *The Unified Modeling Language User Guide*. Addison-Wesley.
- [7] Brooks, F. P. (1975). The Tar Pit. In *The Mythical Man-Month*. Addison-Wesley.
- [8] Brown, W.J., Malveau, R.C., McCormick III, H.W., Mowbray, T.J. (1998). *Anti Patterns. Refactoring Software, Architectures, and Projects in Crisis*. Prentice Hall.
- [9] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley.
- [10] Bush, V. (1945) As We May Think. *Atlantic Monthly*, July 1945.
- [11] Calvary, G., Coutaz, J., Nigay, L. (1997). From Single User Architectural Design to PAC*: a Generic Software Architecture Model for CSCW. In *Proceedings of Conference on Human Factors in Computing Systems (CHI'97)*. Atlanta, Georgia, USA.
- [12] Christensen, M., Crabtree, A., Damm, C.H., Hansen, Klaus, M.H., Madsen, O.L., Marquardsen, P., Mogensen, P., Sandvad, E., Sloth, L., Thomsen, M. (1998a). The M.A.D. Experience: Multiperspective Application Development in evolutionary prototyping. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP '98)*. Brussels, Belgium.
- [13] Christensen, M., Damm, C.H., Hansen, K.M., Sandvad, E., Thomsen, M. (1998b). Architectures of Prototypes and Architectural Prototyping. In *Proceedings of the Eight Nordic Workshop on Programming Environment Research (NWPER'98)*. Bergen, Norway.
- [14] Conklin, J. (1987) Hypertext: An Introduction and Survey. In *IEEE Computer* 20(9).
- [15] Coutaz, J. (1987) PAC, an Object-Oriented Model for Dialog Design. In *Proceedings of IFIP INTERACT'87*.
- [16] Damm, C.H., Hansen, K.M., Thomsen, M. (1997). *Issues from the GCSS Prototyping Project – Experiences and Thoughts on Practice*. Technical Report, Department of Computer Science, University of Aarhus.
- [17] Engelbart, D. (1984). Authorship Provisions in AUGMENT. In *Proceedings of the 1984 COMPCON Conference*.
- [18] Fowler, M. (1997). *Analysis Patterns*, Addison-Wesley.
- [19] Fowler, M. (1999). *Application Facades*. [Online]: <http://www.awl.com/cseng/titles/0-201-89542-0/apsupp/appfacades.pdf>.
- [20] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns. Elements of Reusable Software*. Addison-Wesley.
- [21] Greenbaum, J., Kyng, M. (1991). *Design at Work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associates, Hillsdale New Jersey.
- [22] Grønbaek, K., Hem, A.H., Madsen, O.L., Sloth, L. (1994). Designing Dexter-based and Cooperative Hypermedia Systems. In *Communications of the ACM*, 37(2).
- [23] Grønbaek, K., Sloth, L. (1999). Ørbæk, P., Webwise: Browser and Proxy Support for Open Hypermedia Structuring Mechanisms on the WWW. In *Proceedings of The Eight International World Wide Web Conference (WWW8)*. Toronto, Canada, May 11-14, 1999.
- [24] Hansen, K.M., Yndigegn, C., Grønbaek, K. (1999). Dynamic Use of Digital Library Material - Supporting Users with Typed Links in Open Hypermedia. In *Proceedings of The Third European Conference on Digital Libraries (ECDL'99)*.
- [25] Madsen, O.L., Møller-Pedersen, B., Nygaard, K. (1993). *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley.
- [26] Krasner, G.E., Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-8. In *Journal of Object-Oriented Programming*, 1(3).
- [27] Meyer, B. (1998). *Object-Oriented Software Construction, Second Edition*. Prentice Hall.
- [28] Microsoft (1999). Microsoft Visual C++ Integrated Development Environment. [Online]: <http://msdn.microsoft.com/vstudio/>.
- [29] Nanard, J., Nanard, M. (1991). Using Structured Types to Incorporate Knowledge in Hypertext. In *Proceedings of Hypertext 91*.
- [30] Pfaff, G.E. (1985). *User Interface Systems. Eurographics Seminars*. Springer-Verlag.
- [31] Rumbaugh, J., Jacobson, I., Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley.
- [32] Shaw, M. (1996). Some Patterns for Software Architectures. In *Pattern Languages of Program Design 2*. Addison-Wesley.
- [33] Tsichiritzis, D.C., Klug, A. (1978). *The Ansi/X3/SPARC DBMS Framework: Report of the Study Group on Database Management Systems*. Information Systems, 3.
- [34] Waldén, J., Nerson, J.-M. (1994). *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*. Prentice Hall.
- [35] Wiil, U.K., Nürnberg, P.J. (1998). Collaboration in Open Hypermedia Environments. In *Proceedings of the Fourth Workshop on Open Hypermedia Systems (OHS 4)*.