Requirements Analysis and Architectural Design of a Budget Planning Application

Henrik Edlund

2004-12-07

LITH-IDA-EX-D10--04/001--SE

Advisor: Prof. Mariam Kamkar
Abstract

This thesis presents the theory behind prototyping and user interviews as parts of requirements analysis. It further discusses how these were used in the development of a budget planning application.

The implemented Web-based solution uses a new component-based design rooted in a Model-View-Controller architecture. The application is used today with complete success at the Department of Computer and Information Science at Linköping University.
I would like to extend my gratitude and thanks to my advisor Prof. Mariam Kamkar for her encouragement and guidance throughout this thesis work.

I am also deeply grateful to my friends, you know who you are, for their enduring patience, belief and trust in me. You are all essential in my life.

This thesis was set in \LaTeX using the \LaTeX typesetting system on a computer running the Slackware Linux operating system. The typefaces used are from the Computer Modern family. Figure 4.1 and Figure 4.2 were drawn using Graphviz and retouched using Xfig.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>MVC class structure</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>Budget planning application components</td>
<td>16</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background

This thesis presents part of the work that was done to deploy a completely new Web-based budget planning application at the Department of Computer and Information Science at Linköping University. This project spanned several years and consisted of several steps.

First the project involved development of an advanced prototype that was used in actual budget planning for several years. Eventually the prototype was scrapped and a completely new application was developed, based on the experiences with the prototype as well as user interviews conducted.

When this is written, the new application has been fed an older budget for verification and historical purposes and then used to plan an entirely new budget with complete success.

This thesis report will focus on the user interviews and architectural redesign of the application which were conducted by the author.

1.2 Objective

The objective of this thesis work is to conduct user interviews in order to redesign and implement a budget planning application. There are several requirements of this system. The requirements range from software compatibility to making sure the budget data are accurate.

A prototype and user interviews are used for requirements analysis. Model-View-Controller is then used for architectural design. The application will be put into actual use at the Department of Computer and Information Science.

This thesis report will not go into details about the answers from the user interviews nor beyond the architectural redesign, although a few facts about the implementation will be given.
1.3 Methods

The methods used for requirements analysis are prototyping and user interviews, described in detail in Chapter 2 and Chapter 3. The architectural design was done according to Model-View-Controller and is discussed in Chapter 4.

1.4 Literature

As reference for prototyping, Pfleeger (2001) is used. User interviews are described in Preece et al. (1994). The sources for the Model-View-Controller description are Burbeck (1992), Fowler (2004), Gamma el al. (1995), Microsoft (2004), and Sun (2002).

This thesis depends on several Internet sources. All of these are either publications by reputable corporations or papers archived by universities. The source materials were selected and used as reference on the grounds of the information that needed to be retrieved.

1.5 Terminology

The audience for this thesis is those with at least an undergraduate degree in Computer Science. Terminology acquired during such studies will not be explained here.

Several abbreviations (acronyms or initialisms) are used without expansions in this thesis and will be explained here.

**CGI** Common Gateway Interface

**CSS** Cascading Style Sheets

**GUI** Graphical User Interface

**HTML** Hypertext Markup Language

**HTTP** Hypertext Transfer Protocol

**PDA** Personal Digital Assistant

**PHP** PHP: Hypertext Preprocessor\(^1\)

**WML** Wireless Markup Language

**UI** User Interface

**XHTML** Extensive Hypertext Markup Language

**XML** Extensible Markup Language

\(^1\)This is a recursive initialism.
1.6 Overview

As this thesis commences, it will begin with a background of the project and description of prototyping and the prototype. It will then move on to further requirements analysis using user interviews and after that focus on Model-View-Controller as a practice for architectural design. The thesis will conclude with a statement of results and possible future work.
2 Background

The budget process has made use of computers for many years, all the way back to the 1990s. This chapter will give a short background of how work used to be done and what led up to this thesis work.

2.1 In the Beginning of Time

For many years the department used around forty spreadsheet files that were sent around, one for each center. Once data had been input into these spreadsheets, they had to be sent to the economist, who in turn had to compile a division and department report in a spreadsheet of its own. The users and economist found this system problematic.

First there were no performance data available that fit the budget accounts. The budget accounts are not as detailed as the normal (performance) accounts used in the economy system, so when doing the budget, the users had a hard time comparing their projected performance with actual earlier performance statistics.

Another problem was that input required a special spreadsheet application only available for one operating system.\(^1\) With a multi-platform environment at the department this was problematic. With files stored locally or on the department file sharing network, access to them from home or on the road were usually not possible.

2.2 The Prototype

In 2001 the economist devised a Web-based (any platform, any browser) budget planning system and during September 2001 a fully working prototype was developed by Daniel Costello, an employee of the department. This prototype was then used for the budget planning work that took place during October to November of 2001.

A prototype is a version of a system that can be used to help the user identify key requirements of a system (Pfleeger, 2001). It can also be used to demonstrate\(^1\)

\(^1\)The spreadsheet application was Microsoft Excel.
feasibility of a design or approach. This process is often iterative; after evaluation with user feedback, another prototype is built.

One often uses prototyping in order to design a good user interface (Pfleeger, 2001). The prototype can be used to show the user what functions could be made available and since the user interface is a bridge between the application domain and the software developer, it can be used to identify issues and assumptions that would not have been made clear using another approach to requirements analysis.

As the department used the prototype, more and more ideas and requirements accumulated, some from users but most from the economist. These features and fixes were added to the prototype which continued to be used for actual budget planning work in 2002 and 2003 as well. But the prototype lacked a lot of features, did not always behave as expected and showed signs of being just a prototype.

Requirements analysis often yields a wish list of what the customer would like to see, but it is uncertain whether that list is complete or not (Pfleeger, 2001). Prototyping can be used to help customers determine what they actually want, if their requirements are realistic and if it is possible to solve their problem.

There are two approaches to prototyping: evolutionary and throw-away (Pfleeger, 2001). When software is developed in order to learn more about a problem or to explore feasibility or desirability of possible solutions, but not intended to be used as delivered software, it is called a throw-away prototype. Software developed in order to learn about a problem or to form the basis for some or all of the delivered software is called an evolutionary prototype.

Both of these approaches are sometimes called rapid prototyping as they are meant to build parts of the proposed system so that the requirements can be judged on grounds of necessity, desirability or feasibility (Pfleeger, 2001). Here rapid means that a prototype is built before the design has been decided, in contrast to engineering when the prototype is built when the design is complete. Rapid prototyping is used as a tool to reach a final design.

Studies show that products developed using prototypes performed about as well as those developed using traditional techniques (Pfleeger, 2001). The same studies show that 45 percent less effort was expended and 40 percent fewer lines of code were generated by the developers who used prototypes. The systems developed using prototypes performed almost the same when it came to speed and efficiency, compared to those developed the traditional way. Another study found that prototyping was the best at identifying problems with requirements.

The prototype of the budget application was a mix of both evolutionary and throw-away. With it, a great deal about the problem was learned and the prototype actually became the system used for three real budget planning sessions. During and in between these sessions it evolved and changed more and more as the requirements and users’ expectations changed and were increasingly realized. However, being a prototype initially designed and developed during a very short amount of time, it had some serious design and implementation problems.
In late 2003, as Daniel Costello left the department, a decision was made to throw away the prototype and start over from scratch. A lot of great experience and ideas had been gained from developing and using the prototype, though it was plagued by early design decisions which were hard to correct at this stage. By starting over from scratch, features impossible to add to the prototype could be added and by devoting more initial design time a better, more open and future-proof design could be accomplished.

2.3 Setting the Stage

The role of the user becomes more and more important as software becomes a greater part of critical areas of our lives (Pfleeger, 2001). It is important to learn how the user thinks about problems and solves them, so that our software can support and encourage the appropriate user behaviors. There are many examples of software that prevents users from performing their tasks. By focusing on the need of the user as well as the need of the organization, software can be built that is more responsive and useful.

This meant that at this stage, it was time to move beyond the requirements mainly set down by the economist and move the focus to the users. Now there was enough time to perform user interviews to gather the opinions and requirements of the user and see if these could be accomplished and merged with the earlier requirements.
3 User Interviews

At the requirements stage of design, users can express their opinions of existing work practices (Preece et al., 1994). Checking the opinions of the users during various stages of design is essential and can save a lot of time by ensuring that unusable, unnecessary and unattractive features are avoided. Interviews provide ways of gathering data on users’ preferences and are qualitative compared to the quantitativeness of questionnaires.

3.1 Types of Interviews

According to Preece et al. (1994) there are two main kinds of interviews: structured and unstructured (also called flexible). Structured interviews have questions that were determined before the interview and the interview follows a fixed structure with no exploration of individual attitudes. This type of interview is often used when you want to make statistical statements about a group of interviewees.

Unstructured, or flexible, interviews are the other type and here a topic is decided but the interviewer may go beyond this and explore personal attitudes (Preece et al., 1994). This interview type is less formal than the structured interview and is often used for design and requirements gathering, or to measure users’ opinions about some idea.

In the area between these two archetypes of interviews there are semi-structured and prompted interviewing (Preece et al., 1994). Semi-structured interviewing is when the interviewer has a set of questions that can be used to direct the interview if the interviewee is silent or unsure about what to say.

Prompted interviewing is used when more information is needed from the interviewee (Preece et al., 1994). Here the interviewer may ask the interviewee to continue or to explain something in more detail. An option is to show the interviewee an item, such as a screen design, in order to promote further discussion or generate new ideas for discussion.

Preece et al. (1994) suggest that prompting, questions or working with pairs of users are ways to avoid silences.
3.2 Interviewing Prototype Users

In February 2004, the users of the budget planning application prototype were called to interviews. Interviewees were offered a number of available one hour slots and were to pick the ones where they were available to be interviewed. Their choices were then condensed so that optimally two persons would be interviewed during the same time slot. Some extremely busy persons had to be interviewed alone.

By using pair interviews, one could let one person think while the other talked and also let them spark each other with ideas for things to say. Also being interviewed with a coworker would give the interviewee more comfort and confidence towards the interviewer.

The semi-structured type of interviewing was used as base. Interviewees were in advance mailed a number of basic questions to consider and were called upon to go back and explore the prototype based on these questions. The questions, here translated from Swedish into English, were:

1. How has the prototype improved budget planning compared to earlier?
2. Is there something missing in the prototype that was available earlier?
3. The prototype is divided into screens.
   a) Are these screens logical and do they help and ease the budget planning?
   b) Do the screens reflect the way you work?
   c) Is the navigation between screens good?
   d) What could be done differently in order to increase the understanding, overview and pace of work for you?
4. Consider each screen. (Feel free to print each screen, draw on it and bring to the interview.)
   a) What is good and what is bad about it?
   b) What can be improved?
   c) What should be redone?
5. Is there something missing in the prototype that you would want to have?
6. Anything else?

During the interview, prompted interviewing was used, mainly by having print outs of the prototype screens where ideas could be further examined via drawing or pointing. This way new ideas for discussion were also generated.
The interview followed the questions sent out before the interview but allowed the user, in its semi-structured way, to wander off the given path. As long as the user kept talking and had things to say, the user was not formally interrupted so that the next question could be addressed.

The users’ comments were noted down by the interviewer in a paper pad. The same pad was used for each interview so that it could be noted how many had expressed certain ideas and what the key points were.

The notes were then condensed into a list of requirements. This list was approved and prioritized in cooperation with the head of the department. It would be okay not to implement the lowest prioritized items in time for the upcoming budget planning session, but instead to save them for later implementation, if time became short.

The most commonly requested and highly prioritized features were:

- Addition of a project level so that the budget work could optionally be performed on projects instead of on centers (of which projects are parts)
- Improved history presentation with several years of performance and budget data, available both on reports and at input
- Improved reporting, both on screen and on paper print outs
- Ability to input and update values without refreshing the entire screen
- Flexible exporting of data to other applications
- Approve/reject procedure when handling hiring of employees between centers

\(^1\)These notes are in the private possession of the author.
4 Architectural Redesign

This chapter will cover a description of the architectural design of the prototype and how Model-View-Controller was used in the architectural redesign of the application.

4.1 The Prototype

The prototype was developed in a very short time. With these time constraints the initial design was far from optimal, but proved itself in that it could be used for three budget planning sessions.

The programming language for the prototype was PHP. The user interface was mixed with business logic; several variables that would change between budget planning sessions were set in the program code. Some variables were, however, stored in database tables. Report templates were written in external text files in a comma-separated spreadsheet-like format. A MySQL database was used for raw storage of data, but without integrity constraints and with no business logic. Some database tables had a column for defining budget planning sessions, while others did not.

This approach of course offered a fast way for rapid prototyping, but eventually this design became unworkable. Each budget planning session (year), a copy of the entire database, program code and report templates had to be made as these did not support several sessions in all places. The small support for multiple sessions built-in did not matter, copies had to be made anyway. Then each variable had to be found in the program code and updated.

The prototype served its purpose perfectly as a prototype; it made it possible to show the users examples on interface design and also showed the possibilities of a Web-based application.

As Daniel Costello left the department and the author was to conduct extensive user interviews, it was a consensus that it would be appropriate to start over from scratch, only bringing with us the experiences of the prototype to a new application but no program code or database schemas.
4.2 Model-View-Controller

The Model-View-Controller (MVC) architecture has its roots in Smalltalk (Sun, 2002). The MVC triad of classes is used to build user interfaces in Smalltalk-80 (Gamma et al., 1995). Three objects make up MVC: Model, View and Controller. The Model is the application object, the View is its screen presentation and the Controller defines how the user interface handles user input. Before MVC was introduced, designs of user interfaces often mixed these three objects together. By separating them you increase flexibility and reuse. MVC can be mapped to several design patterns and Gamma et al. (1995) describes this in more detail.

Each object in MVC is specialized for its task (Burbeck, 1992). The View manages the output of the application. Several Views can be attached to a Model and new Views can be created without rewriting the Model (Gamma et al., 1995). The Controller handles input (typically keyboard and mouse), telling the Model or the View to change as appropriate (Burbeck, 1992). Finally, the Model manages the behavior and data of the application domain, answers requests (usually from the View) about its state and listens to instructions (usually from the Controller) to change its state.

Both the Controller and the View depend on the Model, but the Model depends on neither the Controller nor the View (Microsoft, 2004). This separation allows the Model to be built and tested independent of the Controller and View. Figure 4.1 shows the structural relationship between the three objects.

![MVC class structure](image)

This concept is straightforward to map into the domain of enterprise applications (Sun, 2002). The Model then represents enterprise data and the business rules that control access to and updates of this data. The Model aims to be a good software approximation of a real-world process.

The View displays the content of the Model (Sun, 2002). It accesses enterprise data through the Model and specifies how data should be displayed. The View is responsible for keeping itself updated when the Model changes. This is done by either the push or the pull model. In the push model, the View registers itself with the Model for change notifications and in the pull model the View is responsible for calling the Model when it needs to retrieve the most current data. The push
model is also known as the active model (Microsoft, 2004) and the pull model as the passive model (Burbeck, 1992; Microsoft, 2004).

Interactions with the View are translated by the Controller into actions to be performed by the Model (Sun, 2002). Stand-alone GUI clients would have button clicks or menu selections as user interactions. A Web application has GET and POST HTTP requests. The actions the Model performs are business processes or state changes to itself. Depending on the user interaction and the result of the Model action, the Controller will select a new View for response.

By using an MVC architecture it is possible to have multiple Views of the same Model (Sun, 2002). New types of clients can be supported by simply writing a new View and some Controller logic to wire them to the existing Model.

MVC is used today in for example the Swing GUI components for Java (Fowler, 2004). The Swing architecture is rooted in MVC but collapses the View and the Controller into a single UI object. MVC is also a recommended practice when developing ASP.NET applications (Microsoft, 2004). A Document-View variant merges the Controller with the View and the Document then corresponds to the Model role in MVC. This variant is present in many existing GUI platforms, an example being the Microsoft Foundation Class Library (MFC) in the Microsoft Visual C++ environment. The tradeoff with Document-View is that the View and the Controller are more tightly coupled.

In a Web-based application the user interface logic tends to change more frequently than the business logic (Microsoft, 2004). New user interface screens are added or existing layouts are altered. One of the advantages of a Web-based application is that the user interface can change at any time without the need to redistribute the application. If user interface logic is combined with business logic it is very easy to introduce errors in the business logic when doing even minimal user interface changes.

Often an application displays the same data in different ways, depending on audience (Microsoft, 2004). The user interface is also more device-dependent than the business logic. A clean separation of these two parts will make it easier to migrate or extend the application support for new clients, like PDAs and cell phones.

Web applications use the passive model of MVC (Microsoft, 2004). The browser (View) displays the presentation and responds to user input, but it does not detect changes in the data (Model) on the server. The user explicitly has to request a refresh. The server-side components handling the HTTP request are the Controller.

### 4.3 The New Component-Based Design

The new architectural design that was done by the author focused on a clear separation by using the MVC paradigm. No better suited architecture has been found. The application is split into a number of components, each of which can be replaced
or reused by another application. In Figure 4.2 these components are shown with their dependencies drawn.

In this design, the Model is implemented by using a PostgreSQL database and all business logic is put there. This means that data integrity is guaranteed by the database component by using various constraints. All retrieval or change of data is done by calling stored procedures (functions). As the Model is dependent on neither the Controller nor the View, it can be developed and tested independent of these.

A CGI component (part of Controller) acts as the gateway between the browser component (part of View) and the database component. The CGI component consists of several Perl scripts, which are called from the browser component depending on context and current presentation.

After the CGI component has called the relevant database function, it in turn accesses a template component (part of View). The raw data retrieved from the database component is processed by the template component and returned to the CGI component, which in turn delivers HTML to the browser component.

While the returned HTML is presentable on its own, the browser component also retrieves program code from the Javascript component (part of View) and style sheets from the CSS component (part of View). The Javascript code improves the interactiveness of the View by adding interface elements and functionality which users usually only find in classic GUI environments. The style sheets give the
HTML rendered by the browser a better layout, adding colors and spacing in order to improve the visual presentation as well as printing.

This design allows internal changes in the Model without affecting any other component. The database could completely be replaced as long as the CGI component is able to call the same functions and retrieve the same data. The CGI component, here written in Perl, could also be replaced by scripts or programs written in another language, without it affecting any other component.

By replacing, or extending, the template component, presentable/structured data can be sent to the View in other forms than HTML, for example in text form, or as WML (for cell phones), XHTML or XML. As it is the template component that embeds calls for retrieving both Javascript program code and CSS style sheets, these components can also be replaced if needed by other client-side scripting languages or style sheet languages.

This design ensures all business logic remains in the Model while all user interface logic remains in the View. A Controller handles interaction between the Model and the View.
5 Implementation

After the requirements analysis and architectural design phases were completed, implementation took place during June through October 2004. Implementation was performed by the author in cooperation with Andreas Lange. The completed application was delivered in time for the budget planning session that started in October. Some figures on the size of the implementation are given below.

The database component consists of 13,226 lines of SQL code. This code defines 42 database tables, 123 database indices, 154 database functions and 116 constraints of which 56 are foreign key constraints, 26 check constraints and 34 triggers. 35 new data types are defined.

The CGI component is made up of 14 scripts and a supporting module, totaling 1,550 lines of Perl code. Adding up to 2,354 lines of Template Toolkit code, the template component consists of 17 files. The Javascript component has 14 files and 3,923 lines of code, while the CSS component covers 13 files and 1,111 lines of style sheet directives.

There are three auxiliary scripts that create new budgets, import performance data and allow the economist to control database variables from a spreadsheet file. These make up 5,399 lines of Perl code total.
6 Results

The aim of this thesis work was to complete the requirements analysis by conducting user interviews, then redesigning and implementing the budget planning application. This work constitutes the result and accomplishment. The implementation fulfills the requirements laid down by the economist and users, although some low prioritized features are still unimplemented.

The budget planning application was used October through November 2004 to plan the budget for 2005 at the Department of Computer and Information Science at Linköping University. It operated with complete success and will continue to be used for many years to come.
7 Conclusion and Future Work

7.1 Conclusion

This thesis has shown that by using prototyping and user interviews, a better understanding of the requirements can be accomplished. The users have expressed nothing but praise for the delivered budget planning application and they are eagerly awaiting the upcoming additional features. The economist greatly appreciates the complete reporting system.

The component-based design and clear MVC structure make the application easy to maintain and extend for many years to come. The reusability of each component was directly displayed in November 2004 when in just a few days a salary allotment application was developed by reusing components, altering them slightly where needed, from the budget planning application.\(^1\)

7.2 Future Work

Even though the users are satisfied, the next step now when the budget planning session is coming to its end is to conduct follow-up interviews with the users and the economist. It is important to learn in more detail about the experiences the users and economist had with the application and see what can be altered or extended to benefit them the most.

The features that were not implemented due to lack of time should of course also be added so that the application becomes complete and the list of wanted features becomes empty.

\(^1\)The salary allotment application was developed by Andreas Lange.
References


Kravanalys och arkitekturell design av en budgetplaneringsapplikation

Requirements Analysis and Architectural Design of a Budget Planning Application

Henrik Edlund

This thesis presents the theory behind prototyping and user interviews as parts of requirements analysis. It further discusses how these were used in the development of a budget planning application.

The implemented Web-based solution uses a new component-based design rooted in a Model-View-Controller architecture. The application is used today with complete success at the Department of Computer and Information Science at Linköping University.

Budget Planning, Model-View-Controller, MVC, Prototyping, User Interviews
Upphovsrätt

Detta dokument hålls tillgängligt på Internet eller dess framtida ersättare under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press, se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/.

© 2004 Henrik Edlund