Redesign for Flexibility and Maintainability: A Case Study

Christopher Ackermann[1], Mikael Lindvall[1,], Greg Dennis[2]
Fraunhofer Center for Experimental Software Engineering Maryland[1]
Massachusetts Institute of Technology[2]
cackermann@fc-md.umd.edu, mikli@fc-md.umd.edu, gdennis@mit.edu

Abstract

In this paper, we analyze software that we inherited from another party. We analyze its architecture and use common design principles to identify critical changes in order to improve its flexibility with respect to a set of planned extensions. We describe flexibility issues that we encountered and how they were addressed by a redesign and re-implementation. The study shows that basic and well-established design concepts can be used to guide the design and redesign of software.

1. Introduction

The objective of software maintenance is to modify the existing software product while preserving its integrity [4]. While the development process may be long and expensive, it is dwarfed by software maintenance. Often, this period lasts an average of 10 years [1] while the cost constitutes 60%-80% of the entire software budget [2]. The original software developers are often not available, for example, because software development and maintenance is frequently outsourced to different organizations. Thus, it is important to facilitate maintenance of software by developers that were not involved in the original development.

A crucial factor for maintainability is the quality of the software architecture. Software architecture is the structure of the system comprising software elements, the externally visible properties of those elements, and the relationships among them [3]. Well structured, clearly defined, and adequately documented systems are easier to understand, change, and test; and consequently, they are easier to maintain [5]. Structures that allow for easy and quick changes are considered flexible. Flexibility is the degree to which a system supports possible or future changes to its requirements. The more complex the task of adapting a system to modified requirements, the less flexible is the system. Flexibility is thus defined relative to a set of anticipated categories of changes.

In this paper, we describe how we analyzed a working software prototype of the Tactical Separation Assisted Flight Environment (TSAFE), which we used as the basis for a software test bed. At first, the design of the system seemed reasonable, but upon investigating the feasibility of implementing some new features, we discovered several critical issues related to flexibility that required modifications. We analyzed these issues and how they impacted the flexibility of the software system. We analyzed and resolved these issues in order to reduce future maintenance effort.

2. TSAFE

The system we redesigned and re-implemented is a prototype of TSAFE as specified by NASA Ames Research Center and implemented by Greg Dennis at MIT [6]. TSAFE was proposed as a principal component of a larger Automated Airspace Computing system that shifts the burden from human controllers to computers. The TSAFE prototype checks conformance of flights to their flight plans, predicts future trajectories, and displays results on a geographical map. We call the original prototype TSAFE I and the redesigned version TSAFE II. They have different structures but their external GUI and behavior are identical.

3. TSAFE I

The requirements of TSAFE I (and TSFAFE II) are summarized as follows. The system shall continuously read radar flight data from a server. Based on the radar flight data, the system shall compute, for each flight, the expected trajectory, conformance status, and snap back point. The factors determining the trajectory are: the current position, the speed and the heading of the flight. The conformance status indicates whether a flight is satisfactorily conforming to (i.e. following) its planned route as defined in the flight plan and a set of thresholds. The snap back point represents the point on a flights' planned route that is closest to the current flight position. When a flight is conforming to the flight plan, the computed trajectory shall be based on the assumption that the flight will converge to the planned route and the computed trajectory shall therefore follow the plan. A graphical user interface shall display the air traffic on a map of a geographical area. The display shall be updated at a fixed time interval. The user shall be able to select geographical area at system launch. During run-time, the user shall be able to alter

1534-5351/09 $25.00 © 2009 IEEE
DOI 10.1109/CSMR.2009.60

thresholds and other GUI-oriented settings and be able to select flights and flight plans to be displayed.

Since TSAFE I was a prototype, it only implemented the most basic functions from the original NASA description. We identified several features that could arise as new requirements in the future. Some potential change scenarios were already mentioned in the TSAFE I specification [6]. Other change requests emerged from our reasoning about the demands on TSAFE I when operating in a real environment.

FIG reader. Add a feature to read Feed Input Generator (FIG) files, which store recorded flight data.

LOS detector. Add a Loss of Separation Detector (LOS) detecting flights that are too close to each other.

Dynamic map. Add capabilities to view a different part of the airspace during runtime.

Textual client. Add a command line-based user interface.

4. Analyzing TSAFE I

The following describes how we conducted the analysis of TSAFE I in order to derive a basic understanding of the software in preparation of implementing the change requests.

Conceptual View. Figure 1 shows the conceptual view of TSAFE I as provided in the software documentation [6]. From the description we could infer that the Client component initiates communication with all other components and that the Parser updates the Database continuously.

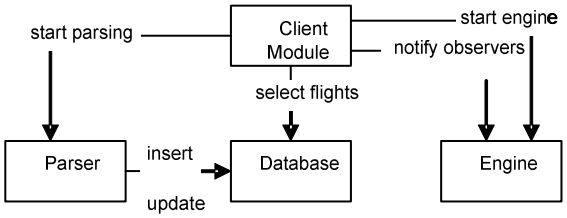

Figure 1: Conceptual view.

Structural View. Since the documentation did not include any structural design documents, we recovered the high level architecture from the source code using the file structure as a guide for identifying the architecture components. The structural view is shown in Figure 2.

The `data` package was used by all components (but the database), which led us to believe that the `data` package was a passive library. The `feed` package contained one class called `FeedParser` and a package called `asdi`. The `main` package contained several classes, which were connected to a number of classes in other packages and a package called `gui`. All other packages did not contain any sub packages but only a number of classes. In the following sections, we will describe architectural characteristics that are of interest for the remainder of this paper.

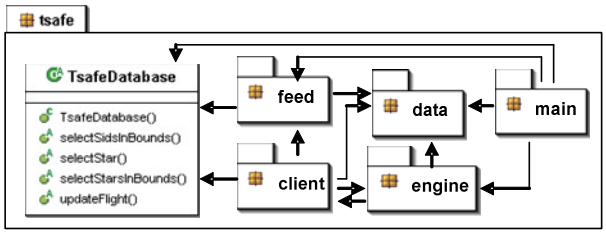

Figure 2: The structure of TSAFE I.

Program Flow. The program flow was retrieved manually by observing the program running in the debugger. Since TSAFE I used two main threads that ran in parallel, two separate program flows could be identified. Thread 1 parsed the feed source continuously and stored the flight data in the database. Thread 2, the main thread, updated the flight data on the gui every 3 seconds. This timer was located in the `client` component, which queried the `database` for flight data and passed the flight data to the `engine` component. When the `engine` component completed its computations, it sent the results back to the `client` and the `client` displayed the flight data on the GUI.

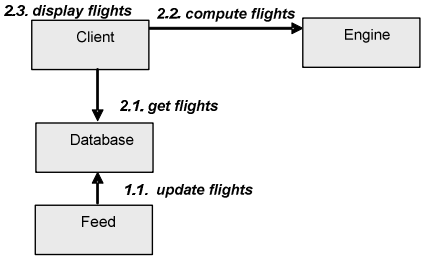

Figure 3: The program flow of TSAFE I.

Figure 3 illustrates the parsing and the main thread. The actions of the parsing thread are labeled 1.x. The actions of the main thread are labeled 2.x. The variable x represents the logical order of the actions.

4.1. Problems with implementing changes

When assessing the impact the changes might have on the program and estimating the change effort, we discovered several issues.

During the first step of the change impact analysis we derived a general understanding of the program. We identified the main functions and where the source code would have to be modified to implement the change requests. The conceptual view provided us with a first glance of the program's functional components and the

architecture recovery gave us more information about the implementation. The program flow analysis was tedious but allowed us to better understand the dynamics of the program.

Conceptual View. The first point that confused us was the naming of the components in the conceptual view. The client seemed to coordinate the entire program, thus, acting as a mediator. It was also unclear what role the engine would play in the program.

Structural View. The structure of the implementation surprised us for a number of reasons. First, the names of the conceptual components were not all represented in the high-level structure. Instead, other components were present, for example, a single class with unclear role. The names were not intuitive. For example, the parser function was implemented in the `feed` and the runtime data structures were located in the `data` package. The `main` package contained a somewhat random collection of classes and packages.

Program Flow. The program flow seemed to follow the conceptual view.

Since the goal was to assess the change impact for the new features of TSAFE, we conducted an analysis that focused on these tasks and encountered a number of flexibility issues.

Implementing the FIG Reader. This change request was difficult to analyze due to the lack of cohesion of the functions the FIG reader had to interact with. The FIG reader would read data from a file and save it to the runtime database. The classes that implement the database were located in different packages.

Implementing the LOS detector. The main criticism regarding the conditions for implementing the LOS detector were the lack of cohesion of the functions the LOS detector had to interact with and the misuse of design patterns. The location of the LOS implementation would be in the `engine` package because it contains all other computation. While the structure of the engine package was rather intuitive, the decomposition of the calculator function was confusing. The `calculator` class contained methods for conducting simple calculations, some of which are needed by LOS. The calculator was implemented using a Template Design Pattern in which an interface defines the methods that are accessible and a concrete class implements them. However, the calculator classes were not located in the same package. Furthermore, there was no need for a Template Design Pattern because the calculator function was not planned to be extended (according to the documentation). The convoluted structure also caused low cohesion and unnecessary high coupling.

Implementing the Dynamic Map feature. The greatest challenge with implementing this change request would be changes to the program flow. The information about the area that is displayed in the graphical user interface i.e., the bounds, were read from a settings file in the beginning of the program and then locally stored by the database, computation, and client. The dynamic map requirement means that when the user changes the boundaries in the client, they must be propagated to all other components.

Implementing the Textual Client. The `client` component contained not only the display of the information but also the responsibility of driving and mediating the entire program. Since there was no component that was solely responsible for user interaction, it was difficult to identify how a textual client could be added. This caused low cohesion in and high coupling of the `client` components, making change in general more difficult.

Summary. The problems we encountered were due to a number of design and documentation issues. First, the conceptual view was not consistent with the actual structure of the system. Second, low cohesion of the system's functions made it difficult to understand how they are implemented. This emerged not only as a problem when the function itself had to be changed (e.g. the user interface) but also when the new feature was supposed to use services provided by a function (e.g. the calculator and database functions). Third, strong coupling between components, made it difficult to recognize interfaces in the structural view. Fourth, design patterns that were misused had a negative effect on the flexibility of the system as it made it difficult to understand. Design patterns were misused because they were implemented in a wrong way (e.g. observer pattern between `client` and `engine`) or they were simply not needed (e.g. calculator). Lastly, understanding and changing the program flow emerged as cumbersome and time consuming.

5. TSAFE II

The analysis of future requirements detected several issues related to lack of flexibility. The goal of the redesign was to first fix those problems and then to create structures to accommodate the implementation of these requirements.

Renaming. The package called `feed` was renamed to `parser` to match the structural with the conceptual view. The engine was renamed to `computation` to better represent its functionality.

Relocation. A new package was created for the database classes. The `ServerMediator` class was introduced to take over the role of the driving component from the client and all the methods were moved from the client to the new class.

Interfaces. Each package now has one class responsible for the inter-package communication.

Dependencies to the `common_datastructures` package are excepted from this rule.

Design Pattern. In order to decouple the GUI from the program logic a client-server architectural style was introduced. It clearly separates the program logic (`server`) from the display functionality (`client`). A mediator pattern now coordinates the sub-packages of the `server` package to clearly identify the driver of the program and minimize inter-sub-package coupling.

Program flow. The main program flow remained the same with the difference that the `ServerMediator` is now responsible for coordinating all activities. Furthermore, an additional flow was added from the client through the `ServerMediator` and to the server sub-packages that operate on the bounds to accommodate updating the bounds data needed to implement the Dynamic Map change request.

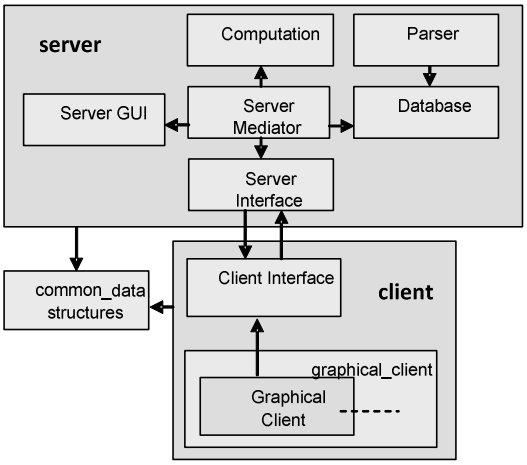

Figure 4: The high level structure of TSAFE II.

6. Conclusion

The high level design provided in the specification of the software system we inherited seemed to be well structured. Each component appeared to have well defined responsibility, and intra-system communication seemed to be conducted through the exchange of a few messages. When planning the implementation of new features – based on a thorough analysis of the static structure and the dynamic behavior of the system – we discovered that the documentation did not match the implementation. The responsibilities of the components were ambiguous, main functionalities were not located in one component but spread out, and the communication between the components was difficult to understand. We addressed these issues through a redesign, and re-implemented the system accordingly.

Programmers often inherit software systems so the issues that this paper describes are not unusual. Our analysis can therefore be useful for programmers who encounter similar situations. However, such issues could even be avoided in the first place, when developing a software system from scratch, but requires that the same kind of analysis is conducted based on early artifacts such as design and architecture documents. We believe that by identifying and avoiding the issues described in this study, during design or during maintenance, one may reduce the maintenance costs since fixing such issues may be complex and time consuming. Such issues may even make the software degenerate and prohibit necessary change.

The detailed description of the changes and the reasoning based on basic design principles can be useful when applying a redesign to other software systems that lack flexibility. The modifications can also serve as examples of how to prepare software systems for the implementation of future requirements.

Acknowledgements

We thank Jens Knodel for helping with an early version of this paper and providing us with valuable suggestions. This work is sponsored by NSF grant CCF0438933, “Flexible High Quality Design for Software.”

References

[1] T. Tamai and Y. Torimitsu, “Software Lifetime and its Evolution Process over Generations”, Proceedings of 1992 Conference on Software Maintenance, Nov. 1992.

[2] Stark, G.E., “Measurements for managing software maintenance”, Proceedings of the International Conference on Software Maintenance, pp. 152 – 161, 1996.

[3] Bass, L., Clements and P., Kazman, R., “Software Architecture in Pratice”, Addison-Wesley, 1998.

[4] Garlan, D., “Software architecture: a roadmap”, Proceedings of the conference on The future of Software engineering, p.91-101, 2000.

[5] Lindvall M., Tesoriero R., and Costa P., "Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture", In Proceedings of International Symposium on Software Metrics, IEEE, 2002, pp. 77-86.

[6] Dennis G., "TSAFE: Building a Trusted Computing Base for Air Traffic Control Software." Masters Thesis MIT, 2003.