SOFTWARE ARCHITECTURE OF THE ALGORITHMIC MUSIC SYSTEM IMPROSCULPT

THOR ARNE GALD SEMB

AUDUN SMÅGE

NTNU

SOFTWARE ENGINEERING MASTER THESIS.
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE,
FACULTY OF INFORMATION TECHNOLOGY, MATHEMATICS AND ELECTRICAL ENGINEERING.

ABSTRACT

This document investigates how real-time algorithmic music composition software constrains and shapes software architecture. To accomplish this, we have employed a method known as Action Research on the software system ImproSculpt.

ImproSculpt is real-time algorithmic music composition system for use in both live performances and studio contexts, created by Øyvind Brandtsegg. Our role was to improve the software architecture of ImproSculpt, while gathering data for our research goal.

To get an overview of architecture and architectural tactics we could use to improve the structure of the system, a literature study was first conducted on this subject. A design phase followed, where the old architecture was analyzed, and a suggestion for a new system architecture was proposed.

After the design phase was completed, we performed four iterations of the action resesarch cyclical process model, where we implemented our new architecture step by step, evaluating and learning from the process as we went along.

This project is a follow up of our previous research project, “Artistic Software” [3], that investigated how algorithmic composition was influenced by software.

PREFACE

This thesis was written as a part of our master degree at the Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU), during the spring of 2006.

We would like to thank our supervisor Letizia Jaccheri, at the Department of Computer and Information Science, for helpful advice and constructive feedback during the semester. We would also like to thank Øyvind Brandtsegg, at the Department of Music, for letting us be a part of such an extraordinary project as ImproSculpt, and providing us with both literature and personal expertise.

Trondheim, 16th of June 2006

Thor Arne Gald Semb

Audun Småge

Table of Contents

Figures

Tables

1 INTRODUCTION

Øyvind Brandtsegg, originally from a jazz improvisation background, has always had a strong desire to experiment and discover new ways of making music. This extended to experimenting with all kinds of synthesizers and music hardware available to him. Always looking for new ways to improvise in musical contexts, Brandtsegg picked up a piece of Macintosh software in 1996 called "Max" [14]. This provided a graphical programming interface, which could output MIDI signals to an acoustic piano for playback. In the years that followed, Brandtsegg wanted to accomplish something similar with audio instead of MIDI, but the available computer hardware just wasn't powerful enough for this task at the time. [3]

In 2000, Brandtsegg started working on the original ImproSculpt. It was developed in part as a compositional tool for a commissioned work from a choir, which put him in a position to realize some of the ideas he carried with him. Computers were now becoming powerful enough to handle a lot of the tasks Brandtsegg had visualized in software, and 1-2 years later, the choir work was performed with the first version of ImproSculpt. [3]

Originally created as a tool for himself, Brandtsegg quickly discovered that his ideas sparked a lot of interest from other composers and music technology enthusiasts, and it was published as an open-source distribution. At the time of writing, ImproSculpt has been used across the globe as both a compositional tool and a live performance instrument. As the author puts it; composing music with ImproSculpt is a kind of live performance in itself. [3]

1.1 Motivation

As the original ImproSculpt project matured, more features were continuously added, and the source code eventually grew large and unruly. Brandtsegg discovered that the task of adding new features became increasingly strenuous, as virtually the entire system was affected by even minor adjustments to the source code.

Eventually frustrated with the amounts of "collateral" work associated with further development of the system, Brandtsegg decided to give up developing the old source code any further, and rather start from scratch with a new and more structured approach. This is where our project met his.

With our education and experience within the field of software engineering and development, our intent was to help Brandtsegg avoid the same pitfalls in the new version of ImproSculpt. We set out to do this by developing an easily modifiable and maintainable software architecture for the system, and consequently implementing it on Brandtsegg's existing functionality.

1.2 Research context

"The general research context of this work is to empirically investigate software developed for artistic purposes (like installations, computer games, performance, music) in public or industrial settings."

The project stakeholders are our supervisor, Letizia Jaccheri, our client and author of ImproSculpt, Øyvind Brandtsegg, and the researchers and authors of this thesis, Audun Småge and Thor Arne Gald Semb.

1.3 Problem definition and project goals

"Provided that software has influenced algorithmic composition, the general research question in this work is: 'how does algorithmic composition constraint and shape software?' More specifically, which properties (qualities) will software for real-time algorithmic composition have? The concrete goal of this project is to design and implement an improved software architecture of the music algorithmic composition system ImproSculpt. Through participation to this development project, the candidates will get insights in the system as well as empirical data which can be used to answer the research questions."

This project is a follow-up of our previous research project *"Artistic Software"* [3], which investigated algorithmic music composition and how it had been influenced by software.

One of the challenges with this project has been finding a balance between devoting time to our research goal and working towards the concrete software goals. The two sets of goals have been, if not contradictory, then at least complementary. Our research focus has been observing how architecture is shaped by this kind of software, while the more concrete project goal has been to improve the software itself.

1.4 Limitation of scope

As we realized that we could not contribute much on the musical side, given Brandtsegg's vast expertise and experience in this area, we quickly decided to focus solely on the aspect we could improve upon as software engineers. The end-user functionality of ImproSculpt as was handed to us, was left completely unchanged. Instead, we have put our efforts into restructuring the architecture of the system, making sure the source code is flexible, easily modifiable and maintainable.

1.5 Reader's guide

1 Introduction
This chapter gives an introduction to the project, defining its context, motivation and scope, in addition to elaborating around our goals.

2 Research method
Our research method is outlined in detail.

3 Architecture prestudy
We take a closer look at software engineering theory, and study different aspects of software architecture.

4 Overview of ImproSculpt
A short description of ImproSculpt, the software system to be improved during this project. There is also a short explanation of the concept algorithmic composition, along with descriptions of the various technologies involved in the project.

5 Design
This chapter contains preliminary system analysis, the requirement specification for the planned improvements, and the specific plans to achieve these.

6 Action research
The iterative software development process we used is detailed here, containing both continuous analysis and concrete system changes.

7 Conclusions and further work
We summarize and discuss what has been achieved, both in terms of improving the software system ImproSculpt, and data gathered towards our research goal.

8 Bibliography
Literature used in our thesis.

2 RESEARCH METHOD

This project used an empirical research method known as "action research", primarily based on the article "Principles of Canonical Action Research" [6]. Avison identifies action research as "*...an iterative process involving researchers and practitioners acting together on a particular cycle of activities, including problem diagnosis, action intervention, and reflective learning*" [6]. In our case, we wanted to establish how live algorithmic music composition systems shape and constrain software architecture, while improving the software architecture of ImproSculpt.

The article suggests five principles for performing this kind of research, namely researcher-client agreements, the cyclical process model, the principle of theory, change through action, and learning through reflection. All principles represent aspects that are important to keep in mind when performing action research, and although the paper is primarily aimed at more thorough and formal research procedures, it was useful to us as more informal guidelines for conducting our research.

2.1 The process model

Figure 2.1 shows the Cyclical Process Model (CPM), which describes how the article sees the action research process being performed in practice:

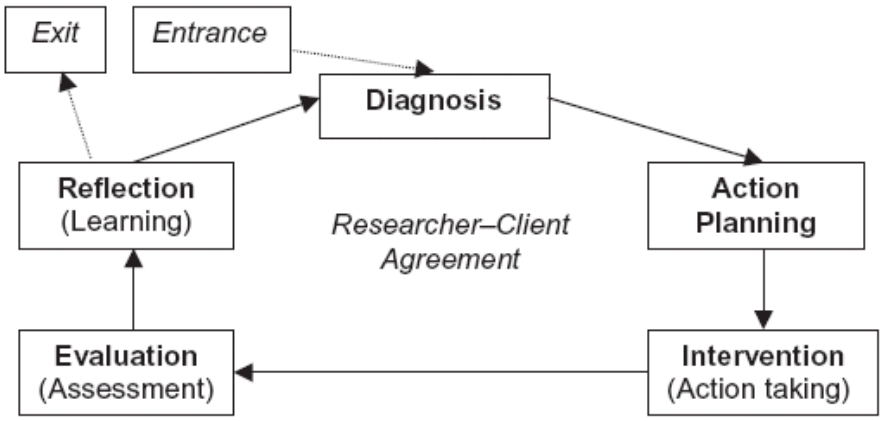


Figure 2.1 Cyclical Process Model

The phases of the CPM are shortly summarized as follows:

- **Diagnosis** - The diagnosis phase is about understanding what is wrong, insufficient, or simply sub-optimal about the current system. It's about cataloguing and evaluating those shortcomings, and finally, working out what should be done differently.
- **Action Planning** - The action planning stage is where one etches out the details regarding planned changes in the system, all based on what was established in the diagnosis phase.
- **Intervention** - Implementation of the changes detailed in the action planning phase.
- **Evaluation** - Analysis of how the intervention affected the project situation. Ideally, this phase involves both the client(s) as well as the researcher(s). Testing is usually also performed in this phase.
- **Reflection** - This is where one looks back at the performed iteration, and interprets the results of it. Often, new ideas and thoughts emerge, and the action researchers are able to learn from the process. Ideally, this is also where one decides whether to perform another iteration or not.

Figure 2.2 shows the Evolutionary Delivery Life Cycle, as suggested by Bass et al. [1].

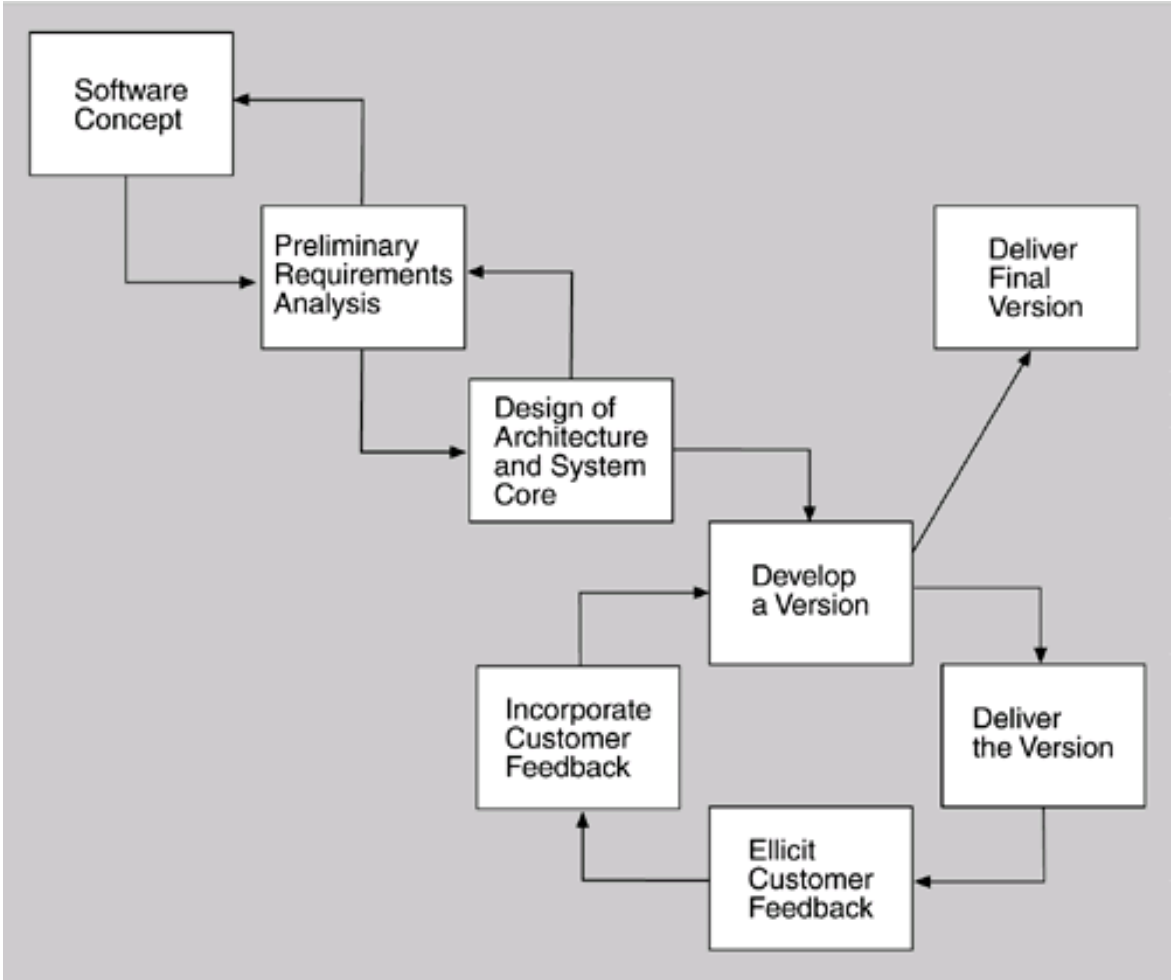


Figure 2.2 Evolutionary Delivery Life-cycle

Our approach to improving the ImproSculpt system has taken the shape of a combination of the Canonical Action Research Cyclical Process Model (Figure 2.1) [13] and the Evolutionary Delivery Life Cycle (Figure 2.2) [1]. While the CPM is a good and thorough model for the iterations themselves, it does not really go into detail regarding the preliminary analysis and design of the system. This is why we decided to use the waterfall-style entrance model from the EDLC model, while preferring the CPM for the iterations themselves. If looked at from a CAR model perspective, our design phase is therefore all part of the "Entrance" phase of the CPM.

2.2 Testing

At the end of each iteration, we planned on performing a series of tests to evaluate the status of the system. The actual testing was planned to take place in the evaluation phase of the iterations, while elaboration around these tests would take place in the reflection phase.

The following tests were planned:

- *Test A - Number of imports:*
  This test investigates the number of imports performed in the source code. This is interesting because it gives an indication of the number of communication paths and dependencies.
- *Test B - Maximum number of source files in a single package:*
  This test checks for the number of Python source files that are collected in the same folder of the file hierarchy. This test gives an indication of how well the system is modularized.
- *Test C - File rename ramifications:*
  The last test investigates the ramifications of renaming a central Python source file in the system. This will give some indication of improvements both in the locality of modifications and preventing ripple effects. What files this test will be performed on, and why, is elaborated upon in chapter 6.5.

2.3 The researcher-client agreement

Before the start of the action research iterations, a researcher-client agreement (RCA) was proposed, discussed and eventually agreed upon by both researchers and client. The purpose of such an agreement was to get a common understanding on both sides' about the motivation and context, and how this action research project was to be executed in practice. All parts agreed that the RCA be kept informal, as a kind of statement of intent. These were the major points:

- Research goal and context.
- A brief description of how we were to use action research in practice.
- What role our client has in this project, and what role the researcher has.

The final version of the RCA is in Appendix A (in Norwegian).

2.4 Data assessment

Data gathering methods were needed for the reflection and evaluation phases in the action research, and eventually for answering the research question. We decided to keep a daily log throughout the Action Research process, as we considered this to be a major asset in our data gathering process. We wrote down not only what we did, but why, what we thought about in the process, which ideas we had, what went according to plan, what didn't, and what new problems and challenges appeared. In retrospect, one tends to read more into events and decisions in light of all that has happened since. It occurred to us that having our thoughts then and there written down, could be valuable later.

During the reflection and evaluation phase, only the useful data were used. This eventually lead to an understanding of what kind of data was useful, and what was redundant. The final form of the log contained:

- Details of the changes that were implemented
- Why these changes were implemented
- The challenges that occurred during implementation of these changes
- Ideas relating to the design plans and architecture
- Thoughts related to the research question that appeared during implementation

3 ARCHITECTURE PRESTUDY

This chapter explains how we prepared for our project in terms of applied software engineering and architecture. It seems worth mentioning that some of the literature was read after the analysis and design of the system had begun, as we thought it important to start the process as early as possible. This resulted in delving deeper into certain literature on topics deemed important in the design process, which explains why there are occational references to the next chapter in the prestudy.

There are many schools of thought about software architecture, but several key concepts are shared by most of them; like quality attributes, tactics and architectural/design patterns. Having surveyed many different approaches, we decided to focus primarily on two books on the subject; *"Software Architecture in Practice"* by Bass, Clements and Kazman [1], and *"Design and Use of Software Architectures"* by Jan Bosch [2]. We will summarize here some of the most important theory and techniques we applied from these books.

3.1 What is software architecture?

Bass et al. [1] proposes the following definition of software architecture:

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

It is important to understand that any system has an inherit architecture regardless of descriptions, documentation or different opinions. Even the simplest of programs has an architecture, albeit probably not a very interesting one. There is, however, no one definite structure; software systems and architectures are comprised of several different structures. [1]

An architectural pattern (sometimes interchangeably used with "architectural style") is a description of element and relation types together with a set of constraints (on an architecture) on how they may be used. For instance, "client-server" is a pattern. Most known patterns have more or less fixed relationships with different quality attributes, in that they strengthen some while typically weaken others. Choosing an architectural pattern is often the first major design choice. [1]

Architectures are one of the most important means of communication between stakeholders, and manifesting early design decisions. A well documented architecture also makes it easier to manage changes, and work with evolutionary prototypes. [1]

3.2 Developing a software architecture

This chapter contains outlines of two suggestions for strategies regarding the creation of software architecture, one presented in "Software Architecture in Practice" by Bass, Clements and Kazman [1], and the other from "Design & Use of Software Architectures" by Jan Bosch [2]. We used elements from both of these strategies in our project.

3.2.1 Bass, Clements and Kazman's development method

Bass et al.[1] outlines 7 basic stages in the process of creating a software architecture:
- Creating the business case for the system
- Understanding the requirements
- Creating or selecting the architecture
- Communicating the architecture
- Analyzing or evaluating the architecture
- Implementing the system based on the architecture
- Ensuring that the implementation conforms to the architecture

3.2.1.1 Creating the business case for the system

Every project needs to assess the target market for the product, but there are other considerations to be made as well; what kind of users are targeted, how much knowledge is required, what kind and how expensive equipment is required to use the product, what are the system limitations it must work within, just to name some. Largely, these are not architectural decisions, but should be made together with architects to ensure feasibility. [1]

In our project, this phase was mostly concerned with establishing what kind of people Øyvind Brandtsegg wanted to be able to use and further develop the system. The primary focus, however, is that Brandtsegg himself is able to use it efficiently.

3.2.1.2 Understanding the requirements

The client usually knows what he wants the product to do, but it is always a challenge to elicit concretized system requirements from those wishes. Many tools are available for just this purpose, including scenarios and use-cases for simple "translation" into requirements, or such things as finite-state-machine models and formal specification languages for more rigorous approaches where this is required. [1]

In our project, this has been quite a long process, starting already last fall [3], where we had several discussions, informal meetings and a more formal interview to gain an understanding of the client's motivation and ultimate goals for the product. To finalize the requirements before our intervention, we arranged a miniature version of the Architectural Trade-off Analysis Method process [5], which is also discussed in Bass et al. [1].

3.2.1.3 Creating or selecting the architecture

This is the process of using the right architectural patterns, styles and tactics to meet the system requirements. The use of quality attributes and quality tactics are important tools to quantify goals, and make priorities clearer. In particular, it is important to have an articulated and prioritized list of quality attributes. [1]

3.2.1.4 Communicating the architecture

Often as big a challenge as developing the architecture, is communicating it to the stakeholders clearly and unambiguously. The importance of explaining it thoroughly to all parties involved should not be underestimated, as it is essential for the efficiency of the development and maintenance of the system. [1]

3.2.1.5 Analyzing or evaluating the architecture

In any design process, there are multiple candidates considered, and there are numerous tools available for rational selection and evaluation of these. We need to make sure that the stakeholders' needs are satisfied. This was also included in our ATAM-style process meeting, see chapter 5.4. [1]

3.2.1.6 Implementing based on the architecture

For the architecture to truly be effective, the development must follow its instructions closely. An explicit documentation of the architecture is vital in this respect, as well as good communication between architects and developers. In our project, we are both architects and developers, making communication a non-issue. It is, however, just as important to make explicit architecture documentation to be able to rationally evaluate and adjust it. [1]

3.2.1.7 Ensuring conformance to an architecture

This phase deals with upholding the original architectural ideas during later stages of maintenance. [1]

3.2.2 Bosch's development method

Jan Bosch [2] defines architecture as a *"...design method which explicitly address and balance the quality attributes of a system"*.

A 3-step architecture design method is suggested. The steps are roughly described as:

1. Functionality-based design: Design of a software architecture based on only functional requirements (not quality requirements).
2. Evaluation and assessment of software architectures: Evaluation of the functionality-based design with respect to the quality requirements. Each quality attribute is given an estimated value (qualitative or quantitative). If all quality attributes at least match the requirement specification, the architecture is finished. If not, go to step 3.
3. Architecture transformation: Improvement of the architecture by selecting quality attribute-optimizing transformations. This is followed by an evaluation (step 2).

In this project, we have taken inspiration from this methodology. But in our case, the functionality-based design (step 1) is already performed by our client. The interesting steps are therefore step 2 and 3, which will be further elaborated upon.

3.2.2.1 Evaluation and assessment of software architecture

Step 2 exists to evaluate the potential of the designed architecture - to find if the architecture satisfies the respective requirement levels in regard to quality attributes.

Bosch [2] describes two ways to assess a quality attribute value:

- Qualitative: Comparing two architectures to find which has the highest quality attribute value.
- Quantitative: Quantifying particular variables that represent an attribute.

4 practical approaches are suggested for assessing what quality attribute values an existing software system has:

- Scenario-based evaluation: This approach is based on using a set of scenarios to concretize the meaning of one specific requirement. It is largely dependent on the representativeness of the scenarios in question.
- Simulation: Main components are implemented and the rest are simulated.
- Mathematical modelling: A mathematical model is created and analyzed.
- Experience-based assessment: Logical reasoning by experienced engineers.

In our project, we found simulation to be a highly convenient means of assessing the quality attribute values. Because the system we were to change the architecture of was already operable, simulating potential architectural changes to test the quality attributes were quite valuable. Naturally, this was a good way in particular to find the quality attribute values on the system we were to change.

Experience-based assessment was also used for this purpose. This mainly concerned reasoning about architecture yet to be implemented.

3.2.2.2 Architectural Transformation (AT)

Step 3 is to perform changes to the architecture. Architectural Transformation requires a software architect to analyze the architecture in order to find the cause of bad quality attribute values. Roughly, the AT process consists of these 4 steps:

1. Find which quality requirements are unfulfilled.
2. Find in which components the quality attribute is unfulfilled.
3. Select transformation.
4. Perform transformation.

4 categories of architectural transformations are suggested by Bosch [2]:

- Impose architectural style: Use architectural styles that increase certain prioritized quality attribute values and perhaps decrease other non-prioritized values. Imposing an architectural style has a major architecture-wide impact. If several styles are merged, it is important that their constraints do not conflict.
- Impose architectural pattern: Impose patterns/rules on the architecture that specifies how the system will deal with one aspect of its functionality.
- Apply design pattern: An example of a design pattern is separating a concrete algorithm from a component to increase maintainability, perhaps at the cost of performance. Design patterns does not have such a large impact on the system as imposing architectural styles, as design patterns only will affect a limited number of architectural components. The danger of creating inconsistency by applying several patterns is therefore lower.
- Convert quality requirements to functionality: Add functionality that fixes a quality requirement. An example of this is implementing exception handling to fix reliability.

Architectural tactics for using in the architectural transformation step relevant to this project are mentioned in chapter 3.4.

3.3 Understanding quality attributes

Quality attributes are more or less quantifiable ways of describing a software system. While there is no definite standard, some of the most commonly used qualities are performance, usability, modifiability, reliability and security. Achievement of one quality attribute will most likely affect others, usually negatively.

"Functionality and quality attributes are orthogonal. This statement sounds rather bold at first, but when you think about it you realize that it cannot be otherwise. If functionality and quality attributes were not orthogonal, the choice of function would dictate the level of security or performance or availability or usability." [1]

This became an important understanding for our project, as we decided to focus solely on improving the system's quality attributes, while leaving the functionality unchanged.

The reasoning behind this is simple; we are developers of this system along with our client. However, his expertise is providing functionality to compose sounds and music, while ours is developing a solid and structured software system. While we are not in a position to be able to improve on Brandtsegg's musical and compositional algorithms, we can use our knowledge to make the system more reliable, faster, and more maintainable.

"Systems are often redesigned not because they are functionally deficient - the replacements are often functionally identical - but because they are difficult to maintain, port, scale, are too slow, or have been compromised by hackers." [1]

The reason Brandtsegg wanted our help is exactly this; the system became difficult to maintain, port and scale. We redesign the system with identical functionality, but improved quality attributes.

3.3.1 Performance

The performance attribute deals with how long it takes for the system to respond when an event occurs (latency). It is also concerned with how much resources, such as processing power, memory and storage space the system consumes during its execution.

3.3.2 Modifiability

This attribute is very closely related to maintainability, and is indeed often used interchangeably with it. In this document, we will treat modifiability and maintainability as a single quality attribute. Modifiability concerns changes in the system. Changes are often divided into three sub-categories: local, non-local and architectural. The local change is simply changing a single element of code, without having to adjust anything else. A non-local change has ramifications other places in the system without fundamentally changing the architecture, while architectural changes brings about major changes in the way the system works. A good modifiable system structure allows most changes to be local, as they are by far the least expensive.

3.3.3 Availability

Availability is concerned with component errors, their handling, and whether they lead to system failures or not. A failure has occurred when the user does not receive results consistent with the specifications of the system, while errors/faults should largely be invisible to the user and handled internally. Faults often lead to failures, if they are not corrected and/or masked.

3.3.4 Usability

The ease and efficiency with which an end-user can perform any given task on the system. Usability in general concerns both how easy it is for new users to learn about the system, and how efficiently experienced users can work with it.

3.3.5 Security

Security is concerned with securing the system from unauthorized access, illegal alteration, denial-of-service attacks and other types of activities where an attacker tries to harm the system. This quality attribute has low priority as the chances of attack on an off-line system with no confidential data is minimal.

3.3.6 Testability

Software testability refers to the ease with which software can be made to demonstrate its faults through testing.

3.4 Achieving qualities

To help achieve goals within specific quality attributes, there are several commonly used "quality tactics" to choose from; that is, certain design decisions that help achieve particular quality attributes. This chapter includes ways to improve quality attributes suggested by both Bass et. al. [1] and Bosch [2].

The focus is primarily on modifiability tactics, since preliminary analysis has indicated that it is the most important attribute to improve. We also have the following criterias on the described tactics to keep them relevant to this project:

- They can be implemented within the time frame of this project.
- They do not have a significant negative effect on the performance.

3.4.1 Bass, Clements and Kazman's architectural tactics

Here is a description of some common modifiability quality tactics described in *"Software Architecture in Practice"*. [1]

3.4.1.1 Localize modifications

These tactics deal with limiting the amount of source code involved in any given change in the system. Some of them are:

Maintain semantic coherence

All parts of the system that deals with one type of task should be in one part of the code. The goal is to ensure that all modules shoulder their responsibilities without excessive reliance on other modules. This makes the system more understandable and efficient, and actually helps in both localizing modifications and preventing ripple effects.

Anticipate expected changes

Write code with future extensions in mind. There could for example be any number of features which would logically be required or strongly desired at some point, but which fall outside the current project's scope. There are often precautions to be taken that make these extensions easier in the future, at little or no cost.

Generalize the module

Situations arise in most projects where one has the same, or very similar, logic applied two or more different places in the source code. Generalizing is about extracting this logic into a single function or module, to be called from the other parts of the source. This way, if a tiny piece of the logic is flawed, it can be changed in a single piece of code as opposed to all the places that utilize it.

Limit possible options

By restricting what parts of the system, or system dependencies, can be changed, one can limit the amount of considerations in development.

3.4.1.2 Prevent ripple effects

The ripple effect in this context is when one change leads to another, which leads to another, and so on. These tactics help isolate information in their own modules, so they are not affected directly from the outside.

Hide information

Information hiding is the tactic of making parts of the data in a module "private", so that it can only be accessed by internal procedures. Other modules wanting to alter this data will then have to do this through public functions, making sure the owner module has full control of all changes that occur.

Maintain existing interfaces

Make function calls and class constructors as general and flexible as possible, so that the system won't have to be redesigned entirely to make room for a single new feature. An example of this is writing functions to accept general "data" parameters, which can contain data types not necessarily known to the system author at the time of writing, as opposed to forcing the parameters to be "data type X".

Restrict communication paths

Make sure any given module shares data and communication paths with as few other modules as is practically possible. This tends to make the system structure simpler and more lucid, and will also prevent ripple effects.

Use an intermediary

If a change upsets the way module A communicates with module B, this can in some cases be solved by inserting an intermediary C to translate between A and B.

3.4.1.3 Defer binding time

These tactics deal with the ability to modify the behaviour of the system after it has been deployed; that is, without altering the source code itself. The most common way to achieve this is through elaborate configuration files, which enable the user to fine-tune the system's operation at start-up. A more costly method is run-time registration, which enables the user to plug in modules and alter system behaviour while the program is running. This typically impedes performance dramatically.

3.4.2 Bosch's architectural transformations

Bosch [2] offers some suggestions of different ways to perform architectural transformations to improve modifiability.

In these descriptions, only effects on performance, modifiability and availability are included since other quality attributes turned out to be of lesser relevance to this project, as later described in chapter 5.2. Both positive and negative aspects are associated with every transformation type. Positive aspects are marked with a + (plus), and negative with a - (minus).

3.4.2.1 Object-orientation

Object-orientation is an architectural style that concern organizing the system in terms of communication objects.

Performance advantages/disadvantages:

No general conclusions are drawn with respect to performance of object-orientation.

Modifiability advantages/disadvantages:

+ Well known for good modifiability as long as the right objects are used.

Availability advantages/disadvantages:

+ Processing units are separated given good object-orientation.
- Separating the system into classes can mean that it is harder for the system to identify that certain responsibilities are not fulfilled.

3.4.2.2 Implicit invocation

This is another architectural style that concerns organizing the system in components that generate events, and components that consume events (possibly containing data). This style is already used in the code.

Performance advantages/disadvantages:

- The architecture itself require a certain amount of computation.

Modifiability advantages/disadvantages:

+ Allows for run-time addition/removal of components, in addition to easy compile-time flexibility. Modifiability naturally depends on the modelling of the components.

Availability advantages/disadvantages:

+ Processing units are separated.
- No central or explicit specification of system behaviour means it is hard for the system to identify that certain responsibilities are not fulfilled.

3.4.2.3 Graphical user interface patterns

Certain patterns are associated with architectures containing a graphical user interface. Bosch [2] presents two main approaches: Model-View-Controller (MVC) and Presentation-Abstraction-Control (PAC). Both of these consist of a model (abstraction), view (presentation) and a controller (control)

Performance advantages/disadvantages in general:

- Generally requires considerable resources.

Modifiability advantages/disadvantages in general:

+ By separating domain model from presentation and control, each component can evolve independently.
- Both patterns increase complexity since functionality associated with domain concepts is divided over several components.

Availability advantages/disadvantages in general:

+ Domain-related computation is basically indipendent from other types of computation, meaning failure in one component do not necessarily affect the other components.
- Increased complexity tend to decrease reliability.

Model-View-Controller (MVC)

MVC means separating the architecture into a View, a Controller and a Model component. The View component is typically a user interface. The Controller's role is to govern the View. The Model is typically the domain functionality, which is where the input from the View is passed on to, and which eventually is displayed back on the View again, through the Controller.

Performance advantages/disadvantages:

- MVC tend to result in lots of update-messages between model and view.
- Access of data in view may require several messages to different parts of the model component.

Modifiability advantages/disadvantages:

- View and controller are often connected intimately, which causes changes to one of them to require change to the other as well.

Presentation-Abstraction-Control (PAC)

PAC means organizing the architecture into a hierarchy of cooperating agents that all contain a presentation, an abstraction and a control component. The control component is the external contact for an agent, as well as an internal coordinator that interacts with abstraction and presentation component.

Performance advantages/disadvantages:

- The control component tends to be the bottleneck for communication since all messages need to pass this.
- Requests that travel up and down the hierarchy often experience a large amount of overhead.

Modifiability advantages/disadvantages:

- PAC tends to result in a complex control component.

In our project, MVC seemed to fit the best, both because of the relatively few changes that were needed to implement this structure, and because it was more familiar to us than PAC.

4 OVERVIEW OF IMPROSCULPT

This chapter gives a description of ImproSculpt, the real-time algorithmic composition system which is the subject of software architecture improvements in this project. Further on, an explanation of algorithmic composition is given, as this is such a central concept to understand in order to realize how ImproSculpt works. Finally, the technology used by ImproSculpt is described.

4.1 What is ImproSculpt?

ImproSculpt is a semi-autonomous live performance instrument. The software is based primarily on sampling different input sounds, manipulating them in various ways, and playing back the results, all in real-time. The manipulation techniques are sometimes predictable, with fixed results from certain parameters, while others are within the realm of algorithmic composition, generating new sounds in a semi-autonomous stochastic fashion.

A quote from the author, Øyvind Brandtsegg, on using ImproSculpt [25]: "*Most of the time, the instrument might be able to bring you surprises... bringing in a new musical element. You're never 100% in control, it's more like you push things in a general direction, let it evolve, and then adjust the bits you do not like and refine the bits that already sound good.*"

This description is close to Chapel's definition of the Active Musical Instrument [24]: "*An Active Musical Instrument is a computer-based instrument for real-time performances / composition, who interacts actively with the musician. The system automatically proposes musical material in real-time, while the user's actions would serve to influence this ongoing musical output rather than have the task of initiating each sound. The instrument acts like a self-regulated system with a personal sonic behaviour, and its core is actually a real-time algorithmic music system.*"

Brandtsegg has used ImproSculpt in a variety of performance and studio settings, collaborating with vocalists, drummers and trumpeters, just to name some. Several recordings can be found at his web page [25].

4.2 Algorithmic composition

Algorithmic music composition, also known as automated composition, is the act of composing music with the aid of algorithms. Beyond that, the definition gets a little fuzzy. Is the system required to compose the music without any input from the composer? Is using randomly generated data as inspiration for a composition part of this concept? What exactly are the criteria for an algorithm in this context?

Rubén Hinojosa Chapel proposes these definitions:

- ***Algorithmic Music*** *is the music created by a computer system in an autonomous or semiautonomous way.*
- ***Algorithmic Composition*** *is a music composition procedure for creating algorithmic music.* [24]

"From a mathematical point of view, music composition could be defined as the process of selecting, from a finite set of elements (pitches, lengths or rhythms, timbres, dynamics, etc.) a subset, also finite. The elements of this subset should be combined and ordered according to a preconceived formal logic." [24]

Computers play an ever-increasing role in musical compositions, and algorithmic solutions to compositional strategies become more and more common. Miranda has detailed three distinct levels of abstraction when composing music with computers:

- **The microscopic level**
  Here, the composer is working with aspects of individual particles of sound such as frequency and amplitude, which are essentially physical sound attributes.
- **The note level**
  This is the level of an 'atomic element of music', where 'certain sound attributes [are] bundled together and [we] think of them as a note'.
- **The building-block level**
  This level is concerned with 'larger musical units lasting several seconds, such as rhythmic patterns, melodic themes and sampled sound sequence', which are collections of notes. I would add that either the building block level should also be concerned with relationships between these musical units, which is how we build larger scale structures and musical relationships, or a separate level of abstraction should be created to highlight such relationships. [26]

At the microscopic level, computers have been used a lot for sound synthesis and manipulation. At note level, the computer is in many ways analogous to a word processor in that it processes words, but not language. An interactive score sheet, if you will.

Algorithmic composition belongs at the building block level, concerning itself with rules, heuristics and mathematical formulae that generate sequences of notes, and to a certain extent, sound attributes.

What is an algorithm, then? Some would suggest that concepts such as rules, procedure and process are closely related to algorithms, and the many existing "rules" of harmony might lead us to consider a great many aspects of musical practice as algorithmic in nature.

David Cope has discussed at some length the question of whether every composer is algorithmic. He is quoted as saying "*Most composers apply rules, steps, or sets of instructions when composing music, especially when composing music in a particular style*", and Copley speculates that he does indeed mean "all" when he writes"most". Cope is also cited as saying "*constraints of almost any kind require algorithmic solutions*", referring to making music to fit within the definition of a particular style.

Cope implies that every significant composer in western art music adhered strictly to established musical forms in their most original work. It is a matter of little doubt that Haydn, Mozart and Beethoven, among others, were well schooled in established rules and practices in composition at the time, most notably those of Johann Fux (1660-1741), but it could just as easily be demonstrated that they paid much less attention to these rules as their compositional skills matured.

Another interesting question is raised: Is there a difference between an explicit or implicit use of an algorithm when composing? The key points here seem to be that implicit algorithms have evolved from a complex interplay of shared practices, rules and similar forms, and are in their very nature more dynamic. Explicit algorithms tend to be simpler and more constraining. Many common practices (implicit algorithms), however, are quite strict in their form, for instance "fugue". The distinction is still important, since the constraints are there to highlight and "frame" the music.

It would seem that although many significant compositions are mostly based on rules and established practices, the truly great compositions, and indeed most original work, seem to have clear deviations and unique ideas that transcend these rules. Maybe a more fitting definition of a compositional algorithm would be "*A set of steps for solving a problem using the minimum necessary number of rules.*" [27]

4.3 Technology

The previously published version of ImproSculpt is programmed directly in the Csound synthesis environment [8]. At the time of writing, Øyvind Brandtsegg is in the process of rewriting the ImproSculpt software in the Python [9] programming language with an interface to Csound, instead of only utilizing Csound's native opcodes. Both Csound and Python code is in itself platform-independent, but requires Csound and Python run-time environments installed on the desired platforms. Both of these are available for all popular operating systems, like Windows, Linux/Unix/BSD, MacOS, Amiga and many others. This chapter describes briefly how ImproSculpt behaves, and contains a more detailed description of the technologies relevant to the project.

4.3.1 Python

"*Python is an interpreted, interactive, object-oriented programming language.*" [9]

In many respects, Python can be compared to Java. It is strictly object-oriented, and based on interpretation, making it effectively platform-independent as long as a run-time environment is available for your choice of operating system. One of the things that separate the two, is that Python provides an interactive interpreter. You can actually give lines of code to the interpreter while a program is running, effectively programming on-the-fly, making debugging and experimentation easier.

Python is an open-source programming environment, certified by the Open Source Initiative (OSI) model [10]. The OSI standards are relied upon perhaps most notably by SourceForge, the world's largest open-source software development web-site [11]. The Python Software Foundation wishes to promote use of open-source software and the open-source community. Nevertheless, OSI exists to bring open-source to the commercial world, and no restrictions are put on products developed in the Python environment. This makes Python an attractive choice even for commercial products, and is utilized by such large corporations as Google inc., RealNetworks and Industrial Light & Magic [9].

4.3.2 wxPython

wxPython is a Python implementation of the C++ library wxWidgets (formerly wxWindows). It is essentially a cross-platform GUI toolkit extension for Python, but also provides a handful of other useful functions, like timer and thread modules [12].

4.3.3 Csound

Csound is an open-source programming environment for sound rendering and signal processing, with roots dating back as far as the 1950's. It has evolved through many leaps and changes since then, and has become one of the most flexible, potent and complex synthesis environments in existence today. The system features more than 450 "opcodes", which are essentially building blocks available to shape "instruments" or "patches". The idea is that the limitations to what you can create stem only from your imagination, knowledge and interest, but never from the language itself [8].

Csound is written in the C programming language, hence the name, and is distributed under the free software license GNU Lesser General Public License [21]. In its most simple usage, the programmer supplies two specifically formatted text files as input: One containing the "orchestra", meaning instruments and sound types, and another contain the score, which instructs Csound as to how the "orchestra" should be played [22].

In contrast to this preprogrammed playback, ImproSculpt utilizes the Csound environment in real-time, effectively changing the orchestra and score information continuously while running.

Csound, at the time of writing, has 18 active developers and is available for all recent Windows versions, Linux, MacOS and a few other platforms. Its development is based around SourceForge, using its forums, mailing lists and a dedicated chat room. [23]

4.3.4 Doxygen

"Doxygen is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavours) and to some extent PHP, C#, and D." [15]

Doxygen is one of the most widely used documentation generator tools for source code in open-source projects, especially those written in C/C++. In later years, it has been extended to support Python programs. It enables automatic generation of a full, searchable system API from comments in the source code, given a certain type of formatting. This documentation can be produced as HTML, PDF and various other formats. As long as the code comments follow the Doxygen standard, the output is very flexible and can be tailored in numerous ways at later stages.

Doxygen also directly supports graphical visualization tools, most notably GraphViz. *"[graphical visualization tools]... represent structural information as diagrams of abstract graphs and networks. Automatic graph drawing has many important applications in software engineering, database and web design, networking, and in visual interfaces for many other domains."* [16]

4.3.5 MIDI

> *"Musical Instrument Digital Interface, or MIDI, is an industry-standard electronic communications protocol that defines each musical note in an electronic musical instrument such as a synthesizer, precisely and concisely, allowing electronic musical instruments and computers to exchange data, or "talk", with each other. MIDI does not transmit audio - it simply transmits digital information about a music performance." [19]*

The important thing to take note of here, is that MIDI information is not sound in itself. MIDI can merely control a sound source, like a synthesizer, that ultimately translates MIDI to audio. In ImproSculpt, Brandtsegg wants to utilize MIDI as a control source for parameters in the system.

Many human interface devices use the MIDI protocol to communicate with audio hardware. By far the most common of these is the keyboard controller, but there are numerous others, like electronic drums, wind-based controllers and motion sensors [19]. In the project "Flyndra", Brandtsegg wants to utilize several climate variables like wind strength, temperature and humidity as input to the sound installation.

The first MIDI specification was developed in Japan in 1983, but is still a widely used standard in music hardware and software, despite the introduction of numerous newer protocols in recent years [18]. The most notable of these standards is the OpenSound Control protocol, or OSC. While being implemented by several recent software and hardware systems, among them Csound, it is supported by few mainstream musical applications and standalone instruments. OSC is technically far superior to the MIDI protocol, which suffers from severe limitations as a result of being more than 20 years old [20].

5 DESIGN

This chapter will first give a short introduction to how the original system was built up, followed by a requirement specification, before describing what plans and intentions we had for the new architecture. It is important to mention that this was the plan before the ATAM meeting and the actual action research, and that this architectural plan was subject to continuous change throughout the entire project.

5.1 Architecture of the original system

Brandtsegg's original system, as was handed to us in March 2006, consisted of a single directory of numerous files of Python source code, Csound effects and instruments, MIDI files and audio files.

Each Python file had a designated original purpose, but some of them seemed to have evolved outside their responsibilities with time. For example, the user interface calculated parts of the Markov melody chain.

There were also a lot of unnecessary interdependency. Serving as prime examples, the `eventCaller` file was referenced from 5 other files, and `markovMelody` from 4.

It became apparent to us that this new approach to ImproSculpt was quickly growing as entangled and unruly as the old version. There were significant structural improvements over the pre-Python venture, but there was still a long way to go architecture-wise. To illustrate the communication paths in the original code, we created Figure 5.1. The arrows pointing to a file indicate that the file is either using an object of the file, or a function or a variable in that file; that is, it depends on that file. Already, changing almost any part of the program involved changing several others.

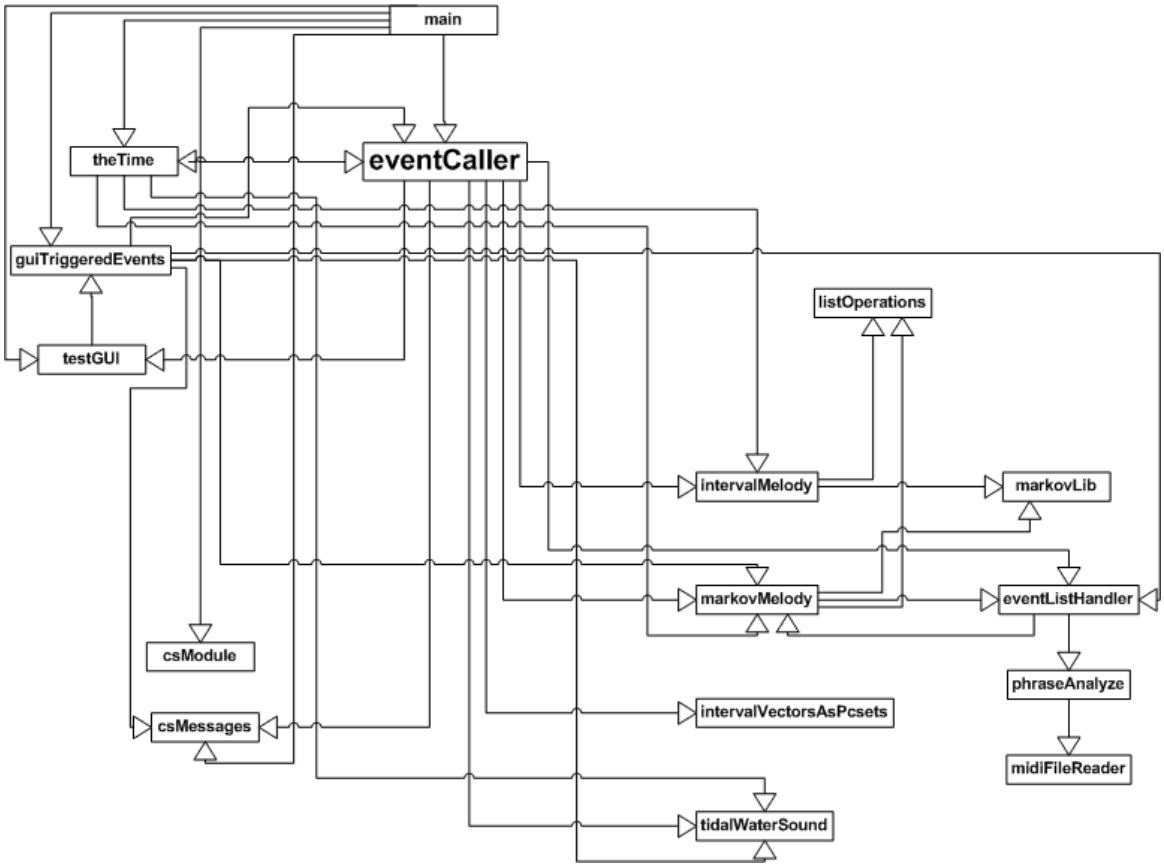


Figure 5.1 Communication paths in the original source code.

We will not go into detail about all of the source code, but discuss briefly those files that were most important to the structure of the system.

`Main_Flyndra.py`
The main file is the entrance point of the system, and had many important tasks. It imported and instantiated most of the system classes, saving only those that handled compositional logic. Most of these instances were then passed, directly or indirectly, to `eventCaller`. The main file started all of the system's threads, ran the main application loop, and stopped the threads upon exit. The file name had the suffix "Flyndra", because at the time of giving us his source code, Brandtsegg was in the process of adapting his ImproSculpt system to a sound installation of that name.

`eventCaller.py`
This class had a part in everything. Depended upon by 3 other files, and itself invoking 9 files, `eventCaller` was clearly the single most important element of the system. It communicated with the user interface, it handled several bits of compositional logic, invoked all the compositional modules, communicated with the Csound engine both directly and indirectly, in addition to controlling the time and event queue module. There was no one distinct task for this file; it was, in short, a little bit of everything. It is also worth noting that, in addition to containing the `EventCaller` class, the `eventCaller.py` file contained a number of dubious references, even including an instance of its own EventCaller class.

`testGui.py + guiTriggeredEvents.py`
These two files were responsible for creating the visual part of the GUI and handling its interaction events, respectively. The latter communicated directly with most parts of the system, including Csound, the event queue and various compositional modules. It also contained a certain amount of compositional logic.

`csMessages.py`
A series of functions designed to handle communication between the Python code and the Csound engine. This file did not contain a class, but rather a series of translation functions to be called from other parts of the system wanting to communicate with Csound.

`csoundModule.py`
This module takes the shape of a class thread, whose sole purpose it is to continuously call `PerformKsmps`, which is required to make Csound process any sound output. Its behaviour is largely dictated by the Csound API.

`timedEvents.py`
This file contains the class `TheTime`, which we later renamed the file after. It consists of a timer and an event queue system. Other parts of the system adds events to various parts of the timeline, and `TheTime` traverses this timeline in accordance with the current tempo in BPM [17]. When it encounters an event scheduled for the current time, it sends this event to its parser for processing. The parser function interprets the event, and makes function calls to the appropriate system modules.

Aside from these, the remaining source files were mostly compositional logic and utilities, such as modules for creating Markov chains, harmonizing notes and chords, and processing audio.

5.2 Requirement specification

The purpose of this chapter is asserting the requirements for the final system, along with more accurately describing the principal problems to be solved. The requirement specification will create a common understanding of what the system should be like, and eliminate misunderstandings. Thus, this document also formally specifies what is to be done, in what order, and how important each element is to the result.

This requirement specification contains the non-functional requirements (not the functional requirements) for ImproSculpt. The purpose of this project is to improve the architecture with respect to a set of quality attributes, main focus being on the modifiability. Functional requirements therefore fall outside this project's scope.

By describing the system's non-functional requirements, we get an overview of what quality attributes have high and low priorities. This serves as a guideline when architectural changes are to be performed on ImproSculpt in order to raise quality attributes that are below the requirement threshold.

5.2.1 Business requirements

The business requirements are taken directly from Brandtsegg's expressed wishes and needs for ImproSculpt.

BR1 The system must be able to sample, process and play back sounds in real-time
BR2 Sounds must be able to be processed by any or all of the modules at any time
BR3 The system must be easily expandable with new sound processing modules
BR4 The system must be easily maintainable
BR5 The system user interface must be efficient

5.2.2 Quality attributes and requirements

The quality requirements are based on the business requirements, but through interpretation and reasoning, they have been concretized and divided into more specific requirements primarily belonging to one of the several quality attributes.

The quality attributes are listed and explained here in prioritized order, in accordance with our communication with Brandtsegg. The quality requirements are listed in the appropriate quality section. From most important to least important, the list of quality attributes are:

1. Performance
2. Modifiability
3. Availability
4. Usability
5. Testability
6. Security
7. Safety

Performance
The performance attribute deals with how long it takes for the system to respond when an event occurs (latency). It is also concerned with how much resources, such as processing power, memory and storage space the system consumes during its execution. Since ImproSculpt is a real-time processing system, this quality attribute has a very high priority.

QR1 The system must handle audio input, processing and output with very low latency.
QR2 The system must run well on today's market mid-range computers.

Modifiability
Modifiability concerns changes in the system. Changes are often divided into three sub-categories: local, non-local and architectural. The local change is simply changing a single element of code, without having to adjust anything else. A non-local change has ramifications other places in the system without fundamentally changing the architecture, while architectural changes bring about major changes in the way the system works. A good modifiable system structure allows most changes to be local, as they are by far the least expensive.

QR3 The system must have a modular object-oriented design that is easy to understand.
QR4 Changes made to the system should be as local as possible (little to no ramifications for other parts of the system)
QR5 The system must be easily expandable with new processing modules
QR6 Csound changes should not affect ImproSculpt drastically, or vice versa

Availability
Availability is concerned with component errors, their handling, and whether they lead to system failures or not. A failure has occurred when the user does not receive results consistent with the specifications of the system, while errors/faults should largely be invisible to the user and handled internally. Faults often lead to failures, if they are not corrected and/or masked.

QR7 As the system will be used in live performances, it is of great importance that the system produces no fatal errors of such nature that the program halts or crashes (i.e. failures).
QR8 The system must be able to handle "wrong" or illogical usage. That is, it must be able to handle unexpected input from the user without crashing.
QR9 The system must be able to catch internal errors in the software in such a manner that the system never terminates because of these.
QR10 (The system must have an error rate of under 1 per 100 time units, where one time unit corresponds to 10 seconds interaction with the system. An error is an action from the system that differs from the expected behaviour. The errors mentioned here are non-critical, in that they do not cause the system to halt, but may produce wrong or unexpected output.)

Usability
The ease and efficiency with which an end-user can perform any given task on the system. Usability in general concerns both how easy it is for new users to learn about the system, and how efficiently experienced users can work with it. The latter is far more interesting than the former in our current context.

Testability
Testability is about revealing a software system's faults through testing. To understand what went wrong and where, it is important for each component to have readily available methods for observing and controlling its internal state at any given time. In short, testability refers to the probability that the software system will fail on its next test execution, assuming it has at least one fault.

Security
Security is concerned with securing the system from unauthorized access, illegal alteration, denial-of-service attacks and other types of activities where an attacker tries to harm the system. This quality attribute has low priority as the chances of attack on an off-line system with no confidential data is minimal.

Safety
Safety is concerned with negative effects the system can have on real world entities. This could for example be an air traffic control system not detecting foreign planes, which could have fatal consequences. This quality attribute has low priority as we are dealing with a system which can do minimal harm to its environment.

5.3 Our design plans

The requirement specification in chapter 5.2, as well as several dialogues with the client, implied that the main concern for the new architecture was to improve modifiability, since the requirements for performance and availability already were satisfied by the current system. Therefore, we concerned ourselves mainly with the numerous modifiability quality tactics from the prestudy. After surveying the different tactics with regard to how they would affect the requirements, how difficult we believed their application would be in this context and the work load involved, we ended up with a list of possible ways to improve the architectural structure of ImproSculpt.

From Bosch [2], an architectural style that fits our project is object-orientation. Using this style on the system implies organizing the system in terms of communication objects by implementing the files as classes and use instances of these classes. The old code is a mixture of files that are classes and files that are not. By using object-orientation, we will in theory improve QR2 directly, and hopefully QR3, QR4 and QR5 as well since all these requirements will be fulfilled with good modifiability.

The concept of architectural drivers is defined in Bass et al. [1] this way:
"An architecture is shaped by some collection of functional, quality, and business requirements. We call these shaping requirements architectural drivers".

Setting aside the functional requirements for the reasons already elaborated over in chapter 3.3, we considered architectural drivers in the design process. To identify our architectural drivers, we identified the highest priority business goals. BR1, BR2 and BR5 are mostly concerned with functionality, and we are thus left with BR3 and BR4 which are both concerned with the modifiability of the system. We chose to utilize a number of modifiability quality tactics suggested by Bass et al.[1], primarily those concerned with localizing changes and preventing ripple effects.

One of the more important tactics we wanted to utilize in the new architecture was maintaining semantic coherence, as described in chapter 3.4. By polarizing responsibilities into separate packages in the system, we would immediately localize modifications, and thus immediately make future architectural changes easier.

Inspired by the Model-View-Controller pattern detailed in chapter 3.4, we saw clear benefits in separating the user interface (View) of the system from the controller and model. One particular reason for this was that our client wanted to use a MIDI device to operate the system in addition to the graphical user interface. A separation of View from Controller would therefore make the use of several different input methods easier. As for the rest of the system, we wanted to separate source files containing compositional logic, sound processing, Csound interface code and utility functions in their own respective packages. The utility package was meant to consist of a collection of classes and functions that are generalized and without dependencies. Examples of utility functions are general mathematical functions and file operations. No data is stored in utility modules; functions are given a set of parameters, and return results after processing them.

There are numerous advantages of utilizing packages to organize the code and file structure, not the least of which is the surveyability of the system from a development viewpoint. We examined the source code, and were able to roughly divide the files into the packages in Table 5.1:

UI testGui.py guiTriggeredEvents.py	**Audio Processing** tidalWaterSound.py	**Csound** csMessages.py csModule.py
Control Main_Flyndra.py eventCaller.py timedEvents.py	**Composition** eventListHandler.py intervalMelody.py markovMelody.py markovLib.py phrazeAnalysis.py ruleChorale.py	**Utilities** midiFileRead.py listOperations.py

Table 5.1 Our first ideas for packages.

As mentioned earlier, we wanted to restrict which parts of the system that could communicate with each other. Figure 5.2 shows how the communication paths were in the original system viewed with our concocted packages, and Figure 5.3 shows how we wanted the communication paths to be after the architectural changes.

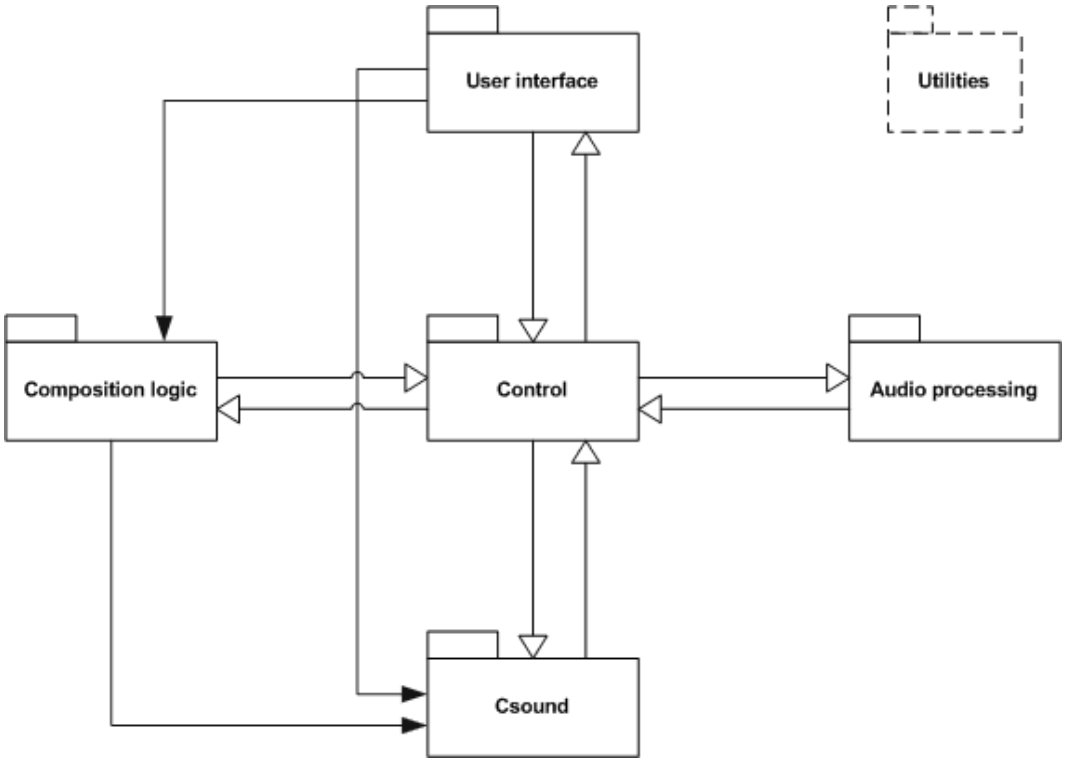


Figure 5.2 Communication paths between the packages in the old system.

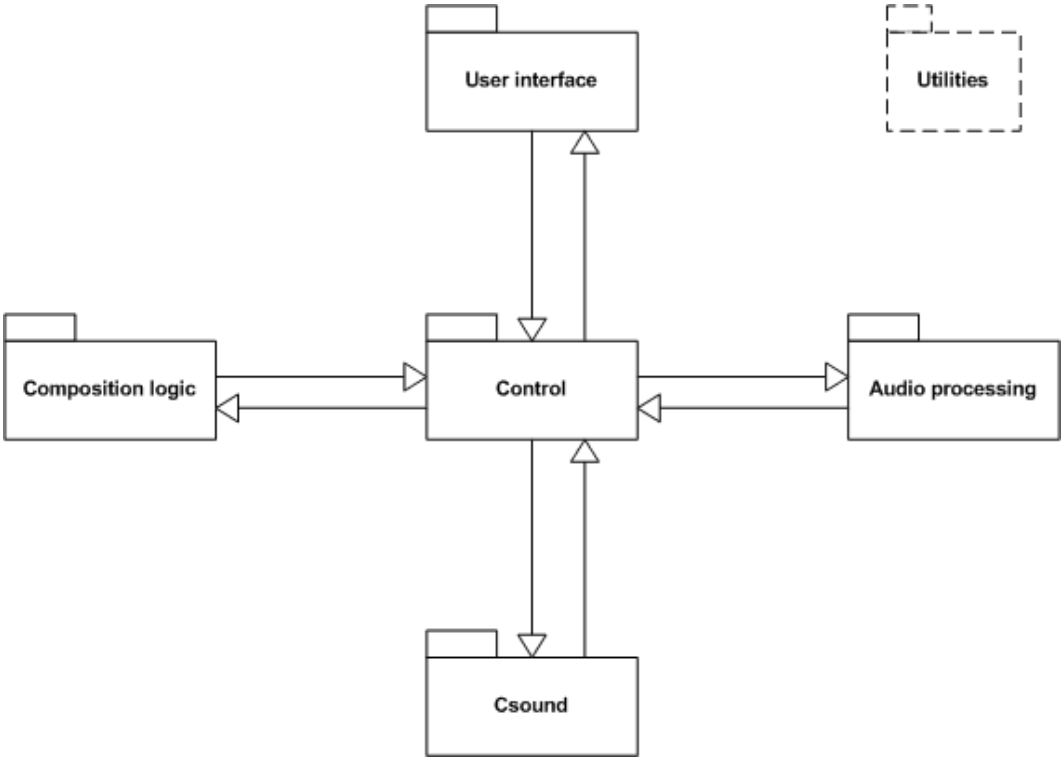


Figure 5.3 Desired communication paths after the architectural changes.

By only letting a central control logic communicate with the other parts of the system, the code will be easier to understand and easier to modify and maintain because of fewer dependencies. As we saw it, a notable flaw in the original code was that several files contained elements that belonged in other parts of the system. If the files were to fit in our architectural model, we would have to restructure a lot of code and separate logic that did not belong together.

All of these architectural changes were designed to improve the system's modifiability. It should be noted that some of these tactics often reduce performance levels, for example generalization. While the performance levels of the system already were adequate, and did not need improvement, we monitored the system's performance closely to make sure it didn't drop noticeably. If there were any reductions in performance, we would have had to reconsider that particular modifiability tactic. The architectural suggestions made in this chapter are further discussed with our client in the ATAM, described in detail in chapter 5.4.

5.4 ATAM

ATAM is short for Architectural Trade-off Analysis Method, and is basically a template for a meeting between different stakeholders in a development team. One of the primary strengths of the method is ensuring that most thoughts and ideas from the project's stakeholders are discussed and considered. In addition to being essential input for the design process, this ensures a common understanding among the stakeholders. There are also phases designed to create scenarios and elicit specific requirements for the project.

The expected results of an ATAM meeting are:
- A concise summary of the new architecture.
- Clear and concise business requirements regarding the system.
- Quality requirements in the form of scenarios.
- How different architectural choices affect the quality attributes of the system.

The original ATAM template suggests a meeting that lasts for several days, but is aimed towards bigger projects than this. Keeping this in mind, we created a tailored scaled-down version of the process, scheduled to last for roughly two hours. The meeting consisted of several steps where each step provides output to the next one.

The following chapters describe, phase by phase, how we carried out the ATAM meeting, and eventually what results the ATAM meeting produced. The only present stakeholders in this meeting were ourselves, and our client, Øyvind Brandtsegg.

5.4.1 Phase 1: Brief ATAM presentation (duration: 5 minutes)

We used 5 minutes to describe the entire ATAM process. A document describing ATAM was sent to all stakeholders via e-mail one week before the meeting, to prepare everyone and to shorten the time used for this phase during the actual meeting.

5.4.2 Phase 2: Identifying requirements (duration: 10 minutes)

The goal of this phase was to get all the requirements on the table. We started off by presenting the list of requirements made in an earlier requirement specification which were based on conversations with our client. After this presentation, a discussion followed where the goal was to come up with requirements that were still missing, and adjust the already existing ones.

The result of this phase was a list of new system requirements:
- Sounds must be able to be processed by all processing modules at any time.
- The composition logic must have a standard function that chooses and returns one or more notes used to continue a sequence, based on one of, or a combination of, compositional algorithms, with regard to potential parameters given to the function.
- The control rate of the system should be 1/4 milliseconds.
- The delay from UI to an effect being performed should be lesser or equal to 10 milliseconds.

5.4.3 Phase 3: Presentation of the architecture (duration: 20 minutes)

In this phase, we presented and afterwards discussed the architectural changes we wanted to perform on the system. They were based on earlier input from our client, which we had interpreted and concretized to the best of our ability. Preparing a suggestion for an architecture prior to the meeting ensured we had a platform to work with, adjusting it as necessary rather than creating a new draft from scratch. This was done with our limited time frame in mind. The architecture suggested in this phase is described in chapter 5.3.

Following this presentation was a short discussion where Brandtsegg could comment on the architecture proposition. One change we agreed upon were that the classes we had dedicated to the audio processing package in reality had the same role as the composition classes, and that they therefore belonged in the composition package as well.

5.4.4 Phase 4: Identify quality tactics (duration: 10 minutes)

This phase concerned identifying the quality tactics that are used in our approach of improving the architecture, presented in phase 3. The relationship between these tactics and the business requirements are also discussed here. The tactics presented here were the same tactics described in chapter 5.1.

5.4.5 Phase 5: Generating a quality tree (duration: 30 min)

In this phase, we were to produce a collection of scenarios to describe quality requirements (non-functional requirements) for the system, based on the business requirements from phase 2. The goal was to establish whether our focus on modifiability was in accordance with stakeholder interests.

As with the business requirements, we had produced a collection of scenarios prior to the meeting to get a head start and save time. We soon discovered, though, that the use of scenarios was an unnecessarily formal and time-consuming way of describing the system requirements in detail, as all stakeholders understood the system from a very technical view. Besides, most of the scenarios we produced described functional requirements which, as explained in chapter 5.2, are outside the scope of this project. The conclusion of this phase was that the non-functional requirements were thoroughly described in phase 2, and no further elaborations on these were strictly necessary. These requirements concerned either performance, modifiability or availability, and since performance and availability were good enough already, the focus on modifiability was justified.

5.4.6 Phase 6: Analysis of architectural approaches

This phase concerned finding and analyzing architectural approaches to satisfy the most important requirements from phase 5. A collection of suggested architectural tactics is described in chapter 5.3. Since the focus of quality attributes did not change in phase 5, these tactics can still be used.

5.4.7 Results

After the ATAM process had completed, we had produced several results that could be used further on in our project:

- A supplement of quality requirements regarding the system, as described in phase 2.
- A concise summary of the new architecture as well as a common understanding and an agreement about this architecture.

At this point, all stakeholders understood how different architectural choices affect the quality attributes of the system. Since the architecture and architectural tactics did not change drastically during this meeting, they stayed the same as described in chapter 5.3 except for the minimal changes described in phase 3.

With the ATAM finished, the design was now created. The next step in this project would therefore be the actual implementation of this architecture, which the next chapter describes in detail.

6 ACTION RESEARCH

One of the most important aspects of utilizing an iterative process, is that one gets to stop and survey one's efforts every now and then, as well as adjusting the details of one's plans. This way, if the premise is imperfect, or even dead wrong, one has the ability to discover and correct this at a reasonably early stage. At the end of each iteration, the cyclical process model allows the action researchers to take a breather, and distance themselves a bit from the work context. At this point, the evaluation and reflection phases allow them to critically review what has been done so far, and carefully consider if the project is still on track, and whether the plan is still sound.

In our project, we have focused considerably on the time frames for each iteration, as opposed to only setting certain objectives for each iteration. This has several reasons; one of them being the unpredictable nature of intervening in someone else's system, which may result in unexpected consequences. Another is that we wanted to make sure we got enough evaluation phases where we could take a look at what we were doing and had done.

6.1 First iteration

After the design phase was finished, as detailed in chapter 5.3, we entered the Cyclical Process Model (Figure 2.1) iterative process. Approximately 50 work hours from each of us were spent during the first iteration, which is significantly more than later iterations. This is due, in part, to the difficulty of fully understanding the system. As we grew more comfortable with the code, modifications became easier to perform.

6.1.1 Diagnosis

Our approach has combined the Evolutionary Delivery Life Cycle with the Canonical Action Research Cyclical Process Model. As a result, our entrance and preliminary diagnosis is considerably more elaborate than suggested in the CPM. The diagnosis phase of the first iteration is therefore, in reality, part of our design process (see chapter 5). However, as the design plans encompassed all the planned changes for the entire system, we used this phase to limit the scope of the first iteration.

Keeping in mind that the available time was limited, we wanted to make sure we were able to finish at least three iterations for the reasons mentioned earlier. This meant we had to prioritize what problems to solve first.

The most pressing issue in our eyes was the difficulty of obtaining an overview of the source code. All of the files were in a single directory, and their relation with each other only became apparent after familiarizing ourselves thoroughly with their content. We spent much time going through the files, method by method, to get a clear idea of how the different parts related to each other.

Brandtsegg has previously expressed a desire to be able to control ImproSculpt through other means than the graphical user interface provided. Most importantly, this includes a MIDI keyboard, but also extends to more experimental devices like motion sensors. With this in mind, we set out to generalize the user interface as much as possible.

6.1.2 Action planning

A good way to help future developers obtain an overview of the system, is separating the related source code into their own packages. This involves both isolating groups of files, and altering source code to make sure all relevant tasks are in their appropriate classes and files. This is related to the modifiability quality tactics of localizing modifications; more specifically, maintaining semantic coherence (see chapter 3.4).

To make the GUI more easily interchangeable with other control devices, we wanted to make a separate package for user interfaces. We also decided to apply the quality tactic of restricting communication paths between the user interface and the rest of the system. This separation of the GUI, or "view", is also in accordance with the principles of the Model-View-Controller pattern [2].

6.1.3 Intervention

One of our first accomplishments was separating the graphical user interface from the rest of the system, by creating a new user interface package called `ui`, incorporating the files `testGui.py` and `guiTriggeredEvents.py`, both of which later became just `gui.py`. Instead of communicating with several different modules concerned with compositional logic, the Csound interface module and the Csound engine itself as it had done earlier, we made sure that the GUI file only made calls to the `EventCaller` class. There was also some compositional processing logic and logic storage in the GUI file which we moved to `EventCaller` for the time being. We did this to minimize the dependency on the user interface, so that it may be easily replaced with other control systems in the future.

6.1.4 Evaluation

From doing some testing on the system, we saw that there already were major improvements on the system modifiability. The number of imports in all of the code had been reduced from 73 to 59.

The number of modifications that needed to be performed after changing the filename of `csMessages.py` (later known as `cs/messages.py`) did not reduce drastically, though. From the 17 changes in the original code, we were now down to 15. The same test run on `eventCaller.py` showed no changes at all. The number of ramifications regarding changing `gui.py` on the other hand, changed from 5 in 4 different files, to only 2 in a single file.

Also, the last test indicated some improvement as the number of source code files in one folder was reduced from 19 to 15.

6.1.5 Reflection

One reason why the number of imports had been reduced, is the restriction of communication paths between some packages. Additionally, some logic has been moved to other files and classes where they seemed to belong. This means for example that the `gui` now only contains user interface logic, and not part of the Markov melody logic which was previously executed before passing values along to other Markov-related modules.

When performing the ripple-effect test on the system, `gui.py` had the largest improvement. This has to do with gui being one of the main focus areas in this iteration. A lot of non-gui logic were moved to other files. Direct communication between `gui` and `csMessages` was also removed completely. The small reduction in the ripple-effect of changing the filename of `csMessages.py` was connected with these changes as well.

The last improvement associated with the reduction of source code files in one folder is connected with the introduction of packages, this time by introducing the ui package.

6.2 Second iteration

The second iteration ran for approximately 35 work hours each, making it somewhat shorter than the first.

6.2.1 Diagnosis

We kept the Model-View-Controller structure in mind, and shifted focus towards the Model part in this iteration, after having separated the View from the rest of the system in the previous iteration. This was still in accordance with our original design plans, but we now proceeded to carry them out on parts of the system that fell outside the scope of the first iteration.

Additionally, some issues appeared during the first iteration that we wanted to address. Most notably, we observed that `csMessages.py` was referenced from all over the source code, which made modifying it quite arduous with regard to the rest of the system.

At this point, we considered how many changes were required if we were to add a new compositional module. Apart from the new code itself, we would have to make changes to `EventCaller` for processing of user input, and the user interface for the same reason, which is to be expected. However, it would also require changing `timedEvents`, which is a sort of timed event queue system. This seemed wrong to us, as this was a general processing module that shouldn't be interpreting anything.

6.2.2 Action planning

With regard to the Model-View-Controller, the Model of the system was the compositional logic along with the Csound-related code. These two sections of code were different enough to warrant two separate packages, which we decided to name `comp` and `cs`.

We saw a strong need to apply the quality tactic information hiding to the Csound interface code, `csMessages`, since we wanted to prevent ripple effects throughout the system when modifying it. To achieve this, we needed to impose a proper class structure on `csMessages`.

There was a significant amount of compositional logic in `EventCaller` at this point, which really belonged in the Model part. We decided to rewrite and move the relevant code lines to their respective composition modules in the `comp` package.

To generalize `timedEvents`, we decided to move the function `parseEvent` to `EventCaller`, and thus make `timedEvents` a generalized module which wouldn't have to be changed when adding new composition modules. Three quality tactics were involved in this plan; generalize the module `timedEvents`, semantic coherence with regard to moving logic to where it naturally belongs and anticipate expected changes, as we were making future expansion of the system easier.

6.2.3 Intervention

We started work on the second intervention by creating a new package for Csound-related Python code, and called it `cs`, incorporating the files `csMessages.py` and `csModule.py`.

At this point, we made a proper class of the previously unstructured `cs.messages`, and changed the old filename references to class instance references. This way, we drastically reduced the amount of reference points in accordance with localizing changes and preventing ripple effects. This change illustrated very clearly to us the ripple effect that will manifest itself in unstructured source code. As expected, making `cs.messages` into a proper class forced us to alter code segments across the entire system.

We started seeing a tendency that `Main_flyndra.py` became smaller, as we realized more and more code was rather out of place there. Much of the instantiation work was moved to `EventCaller`.

A lot of the Markov melody compositional logic was moved from `EventCaller` to the `MarkovMelody` class, where it belonged.

To make the system more surveyable, we decided to make two new directories for the `inc` files and the MIDI source files. The `inc` files are part of the Csound scripting language, and not directly related to the Python code.

`eventListHandler` was unnecessarily referenced from `eventCaller`, so we moved the relevant code to `markovMelody`.

6.2.4 Evaluation

We had to perform this evaluation without our client present because of practical reasons, so we relied on our tests to indicate what had improved.

After running the same tests as we did in the first iteration, we could observe further improvements to the system architecture.

The most drastic change appeared when we tested how many source code changes were required if we changed the names of `cs.messages` or `eventCaller.py`, as detailed in Test C - File rename ramifications. As for `cs.messages`, we needed to change the code in 15 places after the last iteration, while the number of changes now was reduced to just 2. The improvement was even larger for `eventcaller.py`, as the number of changes needed was reduced from 18 to only 1. `gui.py` had already experienced most of its possible improvement after the first iteration, but still there was a reduction of ripple-effects from 2 to 1.

All the other tests indicated improvements as well. While the code after the first iteration used 59 imports, the new code used only 46, which is an improvement of 13 imports. The largest source code package had shrunk from 15 to 10, with the `comp` package now being the largest.

6.2.5 Reflection

From the evaluation, we saw that the largest improvement was in our test for ripple effects after changing the name of `cs.messages` or `eventCaller.py`. This was a direct result of our ongoing process in making the system object-oriented, which in this iteration meant making `cs.messages` into a class, and making sure previous file references now instead referenced instances of the class. Many of these references were in `eventCaller.py`.

There were other improvements made to the system as well. This is a result of the continuous implementation of the tactics introduced in the first iterations, rather than a result of the new tactics introduced in this iteration.

Because of the time limit we had set on this iteration, we did not get to implement the tactics suggested in the action planning phase to the entire system. For this reason, another iteration was necessary.

6.3 Third iteration

The third iteration cost us roughly 25 work hours each, making it the shortest iteration. This is presumably because the implementation has become easier as our understanding of the code has grown. Additionally, we could already reap the benefits of our improvements to modifiability, speeding up implementation even further. The original plan was to let all iterations be of roughly the same length, but because of these unforeseen time adjustments, iterations became progressively shorter.

6.3.1 Diagnosis

There were still difficulties regarding navigation in the code. There were still a lot of source code files that were not separated into packages. Another problem not yet addressed was the mismatches between the class and file names.

After having separated the View and the Model in the earlier iterations, the only part of the Model-View-Controller pattern left separating was the Controller.

As mentioned in the design chapter, the code documentation was incomplete and needed to be fixed, as this is both quick to implement and an effective tool for giving potential new developers an easier way to obtain an overview of the system.

Still, a lot of the tactics applied in the earlier iterations lacked in some parts of the system. There were a lot of reference points between the same classes which caused ripple effects. Since this iteration was planned to be the last one changing the original source code, finishing the implementation of our planned architecture was very important.

6.3.2 Action planning

The way we wanted to separate the Controller from the rest of the system was similar to how we separated the View and the Model in earlier iterations, namely by creating packages and stripping the controller classes for non-controller code.

In addition to creating packages for the controller classes, we now wanted to separate the utility files/classes into an own package in order to further tidy up the code. These utilities were classes containing general functions that take a set of parameters and return the result with no data being stored, as mentioned in the design chapter.

We decided to comment the code in compliance with the standards of the documentation generator tool Doxygen [15], as this was a popular and flexible tool suited for our purposes.

The last planned action in this iteration concerned class dependencies. There were still too many interdependencies after the second iteration, so we decided to revisit the entire source code to remove as many surplus references as we could find.

6.3.3 Intervention

As previously mentioned, a lot of this iteration concerned finalizing architectural ideas introduced in earlier iterations. The following modifications concerned this:

- A new package named `control` was created with the purpose of containing the classes which make up the Controller part of the Model-View-Controller pattern. This new package incorporated `eventCaller.py` and `theTime.py`.
- Almost all filename references were changed to proper class instance references.
- A great deal of logic was moved from the main file to `EventCaller`.
- To finalize the process of making the system object-oriented, `EventListHandler` and `MidiFileReader` were made into classes.
- A few dependencies were removed between classes, specifically `PhraseAnalysis` and `MidiFileReader`.

One new aspect introduced in this iteration was the documentation, which was added throughout the code so as to support documentation generating tools such as Doxygen (see 4.3.4).

6.3.4 Evaluation

Using the same tests as in the earlier iterations, we still noticed improvements. The number of imports were reduced by 4 this time, from 46 to 42, which is a considerably smaller reduction than in the earlier iterations. The number of ripple effects when changing the filename of `cs.messages` was now only reduced by 1, but considering this reduction was from 2 to 1, this was still large improvement. `gui.py` and `eventCaller.py` were already on a minimum regarding this test. Our last test indicated improvement as well, as the folder with the largest collection of source code files now only contained 8 Python source files, which is an improvement of 2 since the second iteration.

This was the first iteration where we were able to meet Brandtsegg after the intervention to discuss the changes made to the system. There was mostly positive feedback from this meeting. The primary focus of discussion during the meeting was documentation tools, more specifically what tools to use.

6.3.5 Reflection

All in all, the improvements in this iteration were not as large in quantity as with the earlier iterations. The number of imports, for example, were only reduced by 4 this time, as opposed to earlier improvements of 13 and 14. This could indicate that we were perhaps getting closer to our architectural goals for this project. This is most definitely the case with the number of ripple effects appearing, which in this iteration only had an improvement of 1. But since the possible improvements only were 2, this still means a 50% improvement.

The only thing that needed some more attention from these three iterations now, as evaluation indicates, was the documentation. Normally, the next iteration would therefore just concern this, but as explained later, the conditions were changed as new code was introduced.

6.4 Fourth iteration

When we started working on the first iteration, we worked on project code from 10th of March. While implementing our architectural changes on the project, we did not consider Brandtsegg's ongoing changes, and the three first iterations are thus based on the same functional code base. We felt this was necessary to be able to effectively carry out our plans. This iteration is therefore somewhat different than the earlier ones, because instead of working on the same code as in iteration 1-3, we were now to adapt new code to our earlier changes.

After our meeting with Brandtsegg 23rd of May, we went to work on a fourth iteration to implement the changes that were made to the system's functionality while we were doing the first three. The time spent on this iteration was roughly 15 work hours each.

6.4.1 Diagnosis

Since communication with our client during the earlier action research iterations was almost non-existent due to practical reasons, the new code basically contained the same problems as the code fixed before the fourth iteration. Additionally, some new issues appeared when inspecting how to adapt the new code. A couple of code lines in the new code caused changes in several similar functions in the existing code, and it was obvious that these functions needed be generalized. Brandtsegg also wanted guidance regarding code commenting and documentation tools, as the evaluation of the last iteration explains.

6.4.2 Action planning

As stated in the diagnosis, some of the methods affected by the new code needed to be generalized. This tactic had until now not been used, so we wanted to inspect the original code as well to try and find more situations where generalization could improve the system and localize changes, as according to the tactic described in the design chapter.

To find out what documentation tool to use, we wanted to do some informal testing on some of the most familiar tools to see what they were capable of and how difficult they were to use so that we could recommend one of them in the end.

The rest of the problems that appeared in this iteration were problems already addressed earlier on, so the only actions used on these problems were actions used in earlier iterations as well.

6.4.3 Intervention

Most of this iteration concerned, as previously indicated, applying changes from the three first iterations to the new code. These changes included updates of several compositional modules and various new minor features.

A change worth mentioning for reasons discussed later in the reflection phase of this iteration, is the separation of MIDI control functions from `cs.messages` to a new `midiControl` class in the `ui` package.

We ended up sticking with Doxygen as our choice of documentation tool for the project. It was easiest to use, and seemed to be the most flexible and widely used tool available. Since the earlier documentation comments in the code were made to work with Doxygen, no code updates were needed.

Another new aspect introduced in this intervention was the generalization of some method groups with similar functionality into a single method.

6.4.4 Evaluation

It would be meaningless to run the same tests on the code after this iteration and compare with the earlier results, as a significant amount of functional code from Brandtsegg had been added, and the only changes in this iteration were performed on this new code.

We also had a meeting with Brandtsegg after this iteration. This time, the feedback was positive as well, and no new problems or tasks emerged. It would seem that the adaptation of the new code had worked satisfactorily.

6.4.5 Reflection

The implementation of the new MIDI interface class highly indicated that the Model-View-Controller pattern worked as intended. This implementation only involved making a new class in the `ui` package, and adjusting the control to handle input from this package. These were originally functions that our client expressed having trouble placing because he did not feel like the belonged anywhere. With our new package hierarchy, they fitted perfectly in the `ui` package. This can be said about all the other new code as well, which is a confirmation that the package separation was meaningful.

The adaptation of the new code did not take particularly long to implement, as we after 3 days of programming were up to speed. The relatively short time required to implement these changes, we feel, is to some degree a testament to the benefits of our structural changes.

6.5 Testing and results

This is a summary of the tests we used to evaluate our progress throughout the iterations. We will present some comparisons between the different stages, and discuss the changes. Note that it only makes sense to compare the first three iterations, as our fourth iteration actually added several new features like MIDI control.

6.5.1 How we tested

Our changes to ImproSculpt have been purely structural, and none of them are actually visible to the end-user. Our focus has been quality attributes that helps developers of the system maintain and modify the code. Therefore, the tests have been chosen from a developer's point of view, not the end-user. The challenge of executing the tests ourselves, as developers, is maintaining objectivity. We have attempted to achieve this by producing as quantifiable results from as rigid tests as possible.

6.5.1.1 Test A - Number of imports

All valid import statements across the source code are counted. A simple idea, yet able to describe interdependency in the system. One of our most important modifiability quality tactics is to restrict communication paths in the system, and this test seems the best way to measure the amount of these paths.

Test ID	A
Name	Number of imports
Quality attribute	Modifiability
Quality tactic(s)	Restrict communication paths
Code revision	**Number of Imports**
Original code	73
After iteration 1	59
After iteration 2	46
After iteration 3	42

Table 6.1 Test A - Number of imports

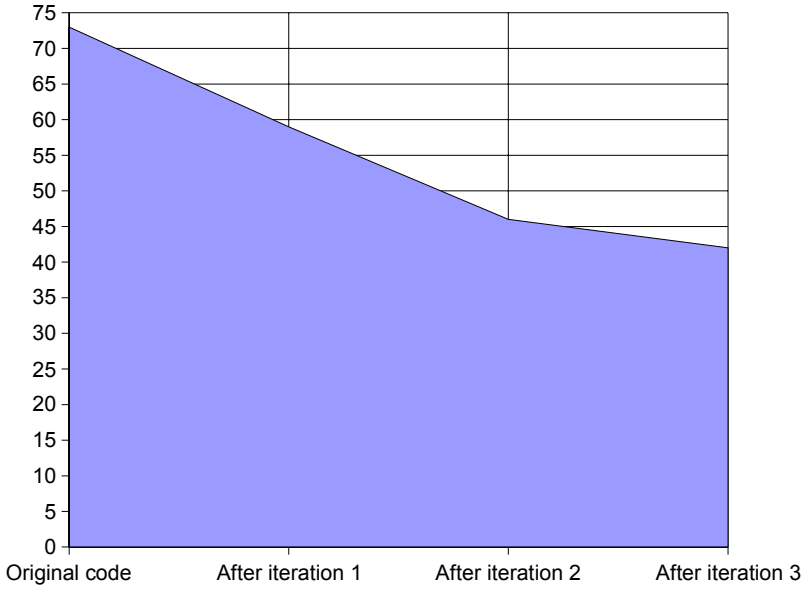


Figure 6.1 Test A - Number of Imports

As is evident from Table 6.1 and Figure 6.1, the number of imports across the source code have been drastically reduced. We believe this is a very important improvement with regard to modifiability. For instance, the GUI now only has a single line of communication with the rest of the system, where it had 3 before. This makes it a lot simpler to replace the graphical user interface with other means of control, an idea which Brandtsegg has shown interest in on several occasions.

6.5.1.2 Test B - Maximum number of source files in a single package

This test deals with how easily a newly introduced developer can understand and get up to speed on the source code. It summarizes to a certain extent how effectively we have managed to modularize the system, and increased the source code's transparency.

Test ID	B
Name	Maximum number of source files in a single package
Quality attribute	Modifiability
Quality tactic(s)	Maintain semantic coherence
Code revision	**Number of source files (what package)**
Original code	19 (root)
After iteration 1	15 (root)
After iteration 2	10 (comp)
After iteration 3	8 (comp)

Table 6.2 Test B - Maximum number of source files in a single package

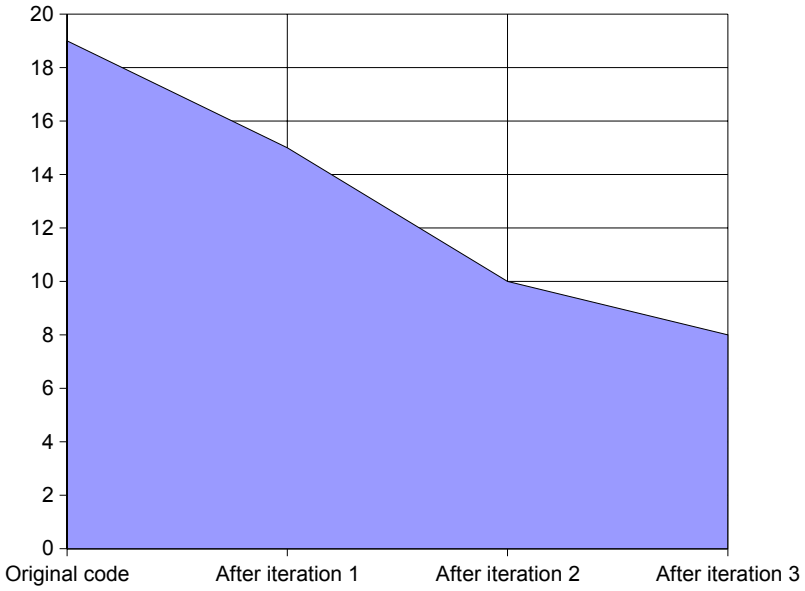


Figure 6.2 Test B - Maximum number of source files in a single package

6.5.1.3 Test C - File rename ramifications

This test deals with localization of changes. We went about this test by changing the file name of the module in question, and then count how many modifications across how many source files were necessary to be able to compile the program successfully again. We decided to choose one module from each element in the Model-View-Controller; `csMessages` from the Model, the GUI from the View and `eventCaller` from Controller, and thus dividing the results into three sub-tests, as detailed in Table 6.3 through Table 6.5.

Test ID	C1
Name	File rename ramifications - csMessages.py
Quality attribute	Modifiability
Quality tactic(s)	Localize modifications
Code revision	**Number of changes necessary**
Original code	17 across 4 files
After iteration 1	15 across 3 files
After iteration 2	2 in a single file
After iteration 3	1 in a single file

Table 6.3 Test C1 - File rename ramifications - csMessages.py

Test ID	C2
Name	File rename ramifications - gui.py
Quality attribute	Modifiability
Quality tactic(s)	Localize modifications
Code revision	**Number of changes necessary**
Original code	5 across 4 files
After iteration 1	2 in a single file
After iteration 2	1 in a single file
After iteration 3	1 in a single file

Table 6.4 Test C2 - File rename ramifications - gui.py

Test ID	C3
Name	File rename ramifications - eventCaller.py
Quality attribute	Modifiability
Quality tactic(s)	Localize modifications
Code revision	**Number of changes necessary**
Original code	18 across 3 files
After iteration 1	18 across 3 files
After iteration 2	1 in a single file
After iteration 3	1 in a single file

Table 6.5 Test C3 - File rename ramifications - eventCaller.py

As is apparent from these tables, our architectural changes to the system have made changes a lot more local. While testing for a single renaming of one file doesn't provide an accurate overview of the entire system, testing one important file from each of the three areas of the MVC model does indicate to a certain extent how the code structure has changed.

The GUI test, Table 6.4, deserves special mention, because of the restructuring during our intervention. In the original source as of March 2006, the GUI consisted of two separate files, which were merged into one file during the first iteration. To make this comparison fair, we used the ramifications of only one of the GUI files for the original code ramifications, namely `guiTriggeredEvents.py`. In all subsequent iterations, `gui.py` was used.

Figure 6.3 shows a cumulative progression of the ramifications of the three files throughout our action research process. We see this as representative for the system as a whole, as we have extended our architectural tactics to all parts of the source code.

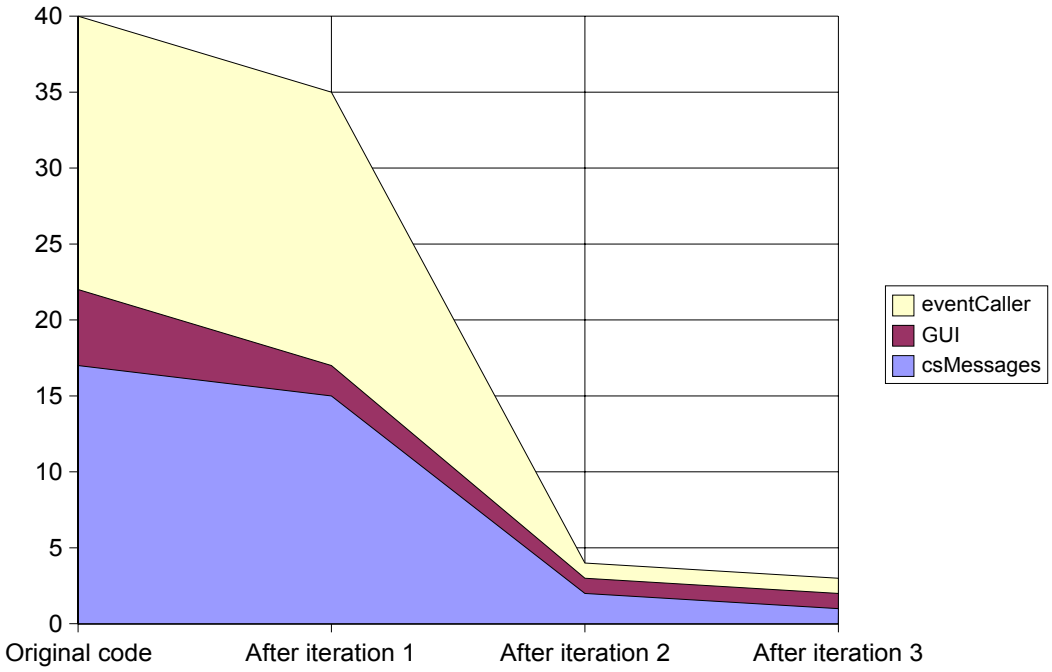


Figure 6.3 Test C - File rename ramifications

7 CONCLUSIONS AND FURTHER WORK

As action researchers, we experienced a duality in our project; we acted as both software engineers and researchers. On the one hand, we wanted to focus on the concrete software goal of the project, namely to develop and implement a new and better architecture for Brandtsegg's ImproSculpt. On the other hand, we wanted to gather data for our research question. These two goals, while not diametrically opposite, are often in conflict with each other when it comes to devoting time and energy to them.

This chapter is divided into two different sections as a result of this duality. The Development goal evaluation looks at ImproSculpt before and after implementing our architecture, what has changed, and how it behaves now, all from a software engineer's point of view. The Research goal evaluation summarizes and discusses what data we have gathered with regard to our research question, and what we can extrapolate from this.

7.1 Development goal evaluation

The formal project definition states that:

"The concrete goal of this project is to design and implement an improved software architecture of the music algorithmic composition system ImproSculpt."

We approached this goal by studying literature about applied software architecture, thorough analysis of the system in its original state, a design phase where we developed a new architecture, and finally, implementing the architecture through the action research iterative process.

Throughout the entire iterative action research process, and specifically during the intervention phases of the last iterations, we noticed improvements to the system modifiability. New changes became easier to implement, they involved fewer source files, and there were fewer ripple effects throughout the system.

With reference to Figure 5.1 depicting the communication paths and dependencies in the original system, we created Figure 7.1 to illustrate how the system dependencies are now, after our interventions. As is apparent, the communication paths have been reduced drastically. There have been numerous minor changes in design plans during the iterative process, but our architectural ideas have proven to be sound, and have in general terms been implemented according to plan.

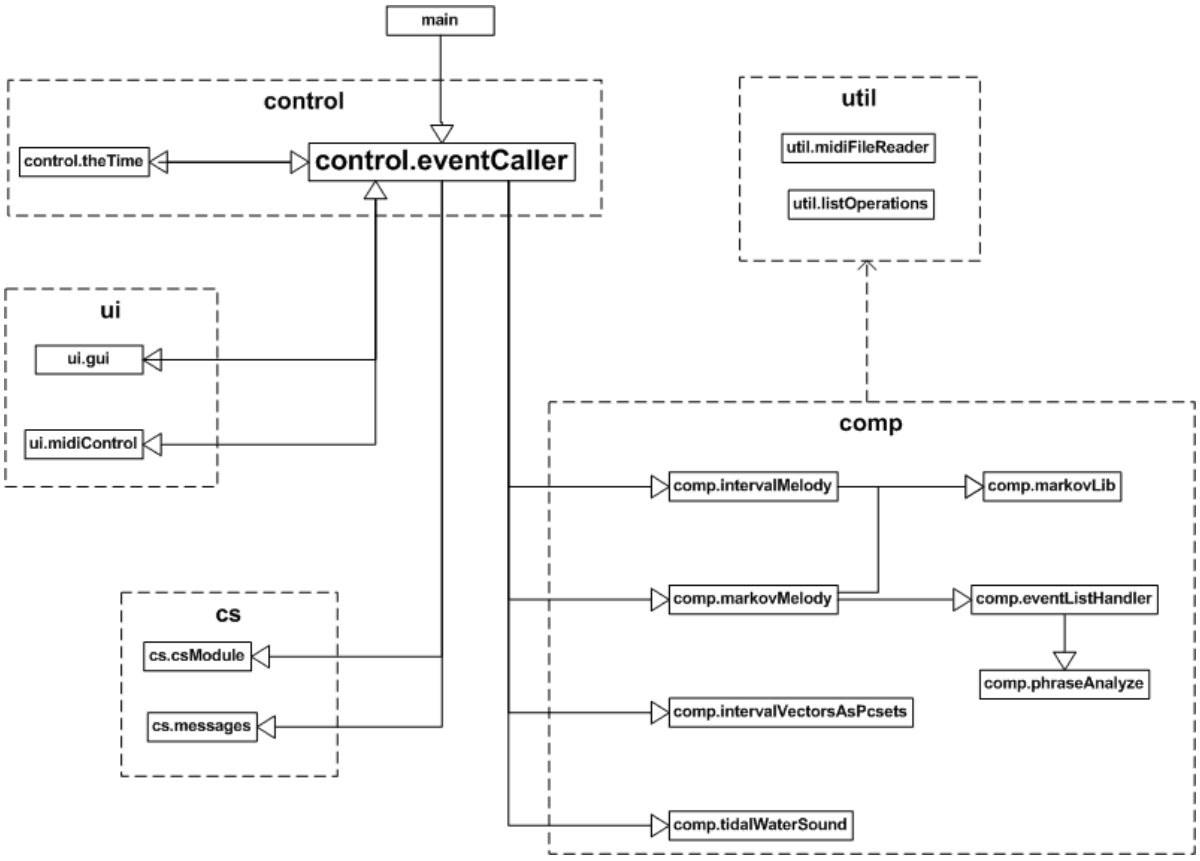


Figure 7.1 Communication paths after our interventions.

The `ui` package represents the View in the Model-View-Controller pattern. As described in chapter 3.4, the View communicates with the Controller, represented in our architecture by the `control` package. When the View receives input from the user, either through the `gui` or the `midiControl` in the current system, the input is sent to the Controller. The responsibility of the `eventCaller` is to govern the Controller's response to this input, and invoke the correct class or classes in the `comp` or `cs` package. In many ways, the `control` package now works as the central hub of the program, and the `EventCaller` class especially handles communication between the other packages. The `comp` and `cs` packages represent, as previously mentioned, the Model. When the Controller has received the data needed from the Model, the data can be sent back to the View. This is typically the case with View classes that give feedback to the user, for example the `gui`.

In chapter 5.1, we described some of the most important modules of the system, and what their respective responsibilities were. Let us look at some of the major changes in these files and their roles:

`Main_CsFlyndra.py -> main.py`
The main file has seen a lot of changes, and is now more of a very minimized initialization module. Being noticeably smaller than it was, main instantiates `eventCaller` and the user interface modules, before starting the main program loop of wxPython. The actual system threads are now started and stopped within eventCaller, but `main.py` calls the start and stop functions before entering and after exiting the main program loop, respectively.

`eventCaller.py -> control/eventCaller.py`
EventCaller has probably seen the biggest changes of all. It has been trimmed a bit, mainly by moving all compositional logic to their respective modules. Even so, it is more of a central hub than ever, as it has become the GUI's only path of communication. As a result, it now handles calls that the GUI previously sent directly to various other modules, such as Csound, the event queue and various compositional modules.

`testGui.py + guiTriggeredEvents.py -> ui/gui.py`
These files have been merged into the new `gui.py`. All compositional logic has been moved to the Model part of the system, facilitating a change of user interfaces in the future.

`ui/midiControl.py`
We created this module based on the MIDI code Brandtsegg had added while we were performing our first three iterations. The class we created contains code largely moved from `csMessages.py`, which we felt would be more suitable for our user interface package. The purpose of this module is handling input from a MIDI device, such as a keyboard for example, and sending those signals to the ImproSculpt system logic. This includes controlling elements of the system also available from the GUI, hence its belonging in the same package.

`csMessages.py -> cs/messages.py`
This module has the same basic functionality as before, acting as an interface between the Python code and Csound. It has, however, been turned into a restrictive class with emphasis on the quality tactics information hiding and restrict communication paths. It now receives all its calls from eventCaller.

`csoundModule.py -> cs/csModule.py`
Apart from its inclusion in the new `cs` package, the module itself only saw minor adjustments.

`timedEvents.py -> control/theTime.py`
While `timedEvents.py`, now `theTime.py`, previously distributed the current event to the corresponding module, it has now became a completely generalized module which does not need to be altered when adding new system features. This was accomplished by moving the actual parser logic to `eventCaller.py`.

As the comments in the original source code were unorganized at best, and absent more often than not, we chose to make sure the entire code comply with the Doxygen [15] commenting standard, see chapter 4.3.4. This is a widely used and flexible tool, and the output from it can be tailored later, as long as the comments are in place in the source code.

The changes described in this chapter show that several improvements have been made to ImproSculpt. Shortly after finishing the implementation, we came in contact with Brandtsegg, who could tell us about positive experiences with the new system architecture, and that he had already implemented a new composition module in it. This was a welcome confirmation of how the architecture worked in practice.

7.2 Research goal evaluation

The research question for this project is as follows:
"Provided that software has influenced algorithmic composition, the general research question in this work is: 'how does algorithmic composition constraint and shape software?' More specifically, which properties (qualities) will software for real-time algorithmic composition have?"

The most important data, with regard to the research question, was gathered during our initial prestudy and design phase. At this point, we were creating and designing an architecture to satisfy the needs and requirements from our customer. During this process, we had to firmly establish and concretize what those requirements were.

After the iterations of the Cyclical Process Model started, there was a very limited amount of new research data gathered. Most of the architectural choices made in the design phase remained valid throughout the entire process, and thus, there were few new constraints and little more shaping of the software architecture during the CPM. There are numerous possible explanations for this; our first draft of the architecture may have been largely sufficient, and we may have had too much control over the development process and the final structure.

From our initial prestudy and design process, we concluded that the three most important quality attributes for real-time live algorithmic music composition systems were performance, modifiability and availability. This conclusion was based on the fact that all of the requirements we elicited from our client belonged to one of these quality attributes. The particular focus on modifiability might not be as applicable to other software systems of this type in general, as it is directly related to this project and the state of the source code we received. Modifiability was the only quality attribute that did not satisfy the requirements specified in the design phase.

Ultimately, we concluded that the three most important quality attributes in software for real-time algorithmic music composition were performance, availability and modifiability, while most other common quality attributes were of lesser importance.

7.3 Further work

This project has improved the modifiability of ImproSculpt drastically, without impeding it in regard to the other quality attributes. The non-functional aspects of the system has been improved to an acceptable level, and perhaps the next focus could be on the functionality.

Brandtsegg has during meetings expressed the need for a better and more effective graphical user interface, perhaps with the use of 3D technology. There are also a huge range of algorithms and techniques for algorithmic music composition, which could be interesting to implement as new modules in ImproSculpt. Some of these techniques are genetic algorithms, neural networks, cellular automata and fractals. For a closer look at these methods, see the descriptions in our previous research project, “Artistic Software” [3].

Further work could also be done within the research context. This project is based on qualitative action research on a single system, and one could consider running a series of more rigid, quantitative tests on various related software systems to gather data. Alternatively, there is the possibility of carrying out similar action research on another real-time composition system, to gather more empirical data for performing qualitative comparisons.

8 BIBLIOGRAPHY

[1] Len Bass, Paul Clements and Rick Kazman. *"Software Architecture in Practice, Second Edition"*. Boston, MA: Addison-Wesley, 2003.

[2] Jan Bosch. *"Design and Use of Software Architectures"*. Boston, MA: Addison-Wesley, 2000.

[3] Thor Arne G. Semb and Audun Småge. *"Artistic Software"*. IDI, NTNU, 2005.

[4] Øyvind Brandtsegg. *"A Sound Server Approach to Programming in Csound"*. Csound Journal, Fall 2005.

[5] *"The Architecture Trade-off Analysis Method (ATAM)"*. `http://www.sei.cmu.edu/architecture/ata_method.html`. Carnegie Mellon University, last modified 4th of May 2006.

[6] David Avison, Francis Lau, Michael Myers and Peter Axel Nielsen. "*Action research*". Communications of the ACM, volume 42, issue 1, pages 94 - 97, 1999.

[7] Robert M. Davison, Maris G. Martinsons and Ned Kock. "*Principles of canonical action research"*. Information Systems Journal, Vol. 14, No. 1, pp. 65-86, 2004.

[8] cSounds.com, website of Csound. `http://www.csounds.com`, last visited May 2006.

[9] Python Software Foundation, 2005 http://www.python.org

[10] Open Source Initiative (OSI). `http://www.opensource.org`, last visited May 2006.

[11] SourceForge.net. `http://sourceforge.net`, last visited May 2006.

[12] wxPython's website. `http://www.wxpython.org`, last visited May 2006.

[13] Robert M. Davison, Maris G. Martinsons and Ned Kock. "*Principles of canonical action research*". Information Systems Journal 14, Blackwell Publishing, 2004.

[14] Wikipedia, *"Max (software)"*, The Wikimedia Foundation Inc. `http://en.wikipedia.org/wiki/Max_%28software%29`, last visited June 2006.

[15] Doxygen's homepage. `http://www.doxygen.org`, last visited June 2006.

[16] GraphViz' homepage. `http://www.graphviz.org`, last visited June 2006.

[17] Wikipedia, *"Beats per minute"*, The Wikimedia Foundation Inc. `http://en.wikipedia.org/wiki/Beats_per_minute`, last visited June 2006.

[18] Doan, T.T. *"Understanding MIDI"*. Potentials, IEEE, Vol. 13, Issue 1, pages 10-11. Published February 1994.

[19] Wikipedia, *"Musical Instrument Digital Interface"*, The Wikimedia Foundation Inc. `http://en.wikipedia.org/wiki/Midi`, last visited June 2006.

[20] Matthew Wright, Adrian Freed and Ali Momeni. *"OpenSound Control: State of the Art 2003"*, Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03), Montreal, Canada.

[21] *"GNU Lesser General Public License"*, Free Software Foundation, Inc. `http://www.gnu.org/licenses/lgpl.html`, last visited June 2006.

[22] Wikipedia, *"Csound"*, The Wikimedia Foundation Inc. http://en.wikipedia.org/wiki/Csound, last visited June 2006.

[23] Csound project page at SourceForge.net. `http://sourceforge.net/projects/csound`, last visited May 2006.

[24] Rubén Hinojosa Chapel. "*Real-time Algorithmic Music Systems From Fractals and Chaotic Functions: Toward an Active Musical Instrument*". Doctoral Pre-Thesis Work, Universitat Pompeu Fabra, Barcelona, 2003.

[25] Øyvind Brandtsegg's website, `http://oeyvind.teks.no`, last visited December 2005.

[26] Eduardo Reck Miranda, *"Composing music with computers"*. Music Technology Series. Oxford, UK: Focal Press. 2001.

[27] Peter Copley, Andrew Gartland-Jones, "Musical Form and Algorithmic Solutions", Proceedings of the 5th conference on Creativity & cognition, London, United Kingdom, Poster papers, Pages: 226 - 231

Appendix A - RCA

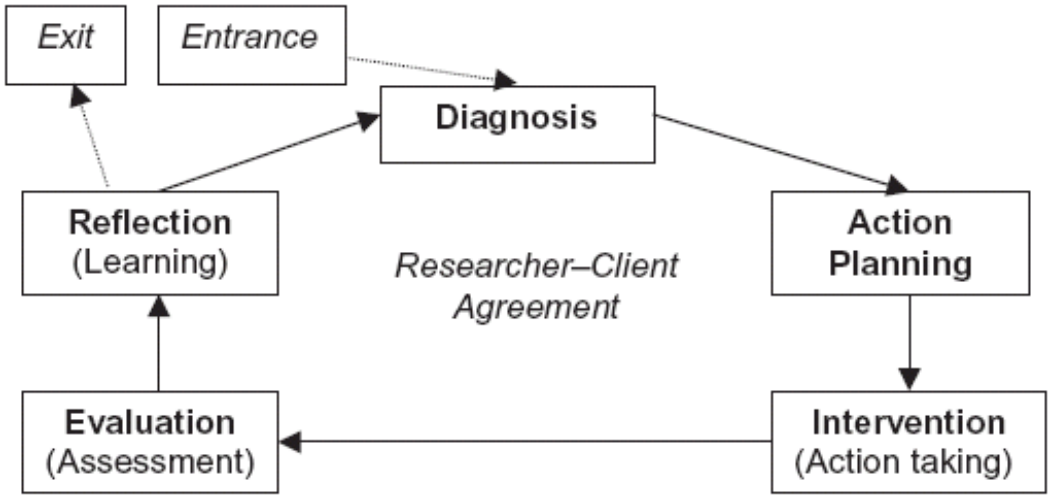
Researcher-Client Agreement i tråd med artikkelen "Principles of Canonical Action Research" (Davison, Martinsons, Kock)

Formål med forskningsprosjekt

Vårt formål med dette forskningsprosjektet er å belyse hvilke krav som stilles til, og restriksjoner som legges på, software-arkitektur i systemer for real-time algoritmisk komposisjon. Dette utgjør forskningsmålet i vår Master-oppgave på IDI ved NTNU.

Gjennomgang av forskningsmetode

Vår arbeidsmetode i dette prosjektet kommer til å følge en syklisk modell som vist i figur under.



Vår plan er å gjennomføre de iterasjonene som er nødvendige for at kunden og vi blir fornøyd med den implementerte arkitekturen. Likevel må siste iterasjon være ferdig i starten av juni siden forskningsprosjektet vårt har deadline 16. juni og etterarbeid er nødvendig.

Hvor lang tid en iterasjon vil ta avhenger av arbeidsmengden som må gjennomføres, noe som avklares i Diagnose-fasen til en iterasjon. Den skal likevel ikke overskride 1 måned da hyppig kommunikasjon mellom utvikler og kunde er en nødvendighet.

Entrance: Ved inngang til CAR-syklusen har vi gjennomført en ATAM og har en arkitektur klar som vi vil implementere på ImproSculpt.

Diagnosis: Her gjennomfører vi en analyse av hva som skal endres på.

Action planning: Planlegging av hvordan vi skal gjennomføre nye endringer.

Intervention: Her utfører vi de faktiske endringene.

Evaluation: Her evaluerer vi hva slags effekt endringene fikk, om det påvirket systemet som forventet. I denne fasen kan vi ta et møte med kunden vår slik at endringen blir belyst fra alles ståsted.

Reflection: I denne fasen reflekterer vi over hvorfor endringene eventuelt ikke fikk de effektene som var planlagt, og om noe kunne vært gjort annerledes.

Exit: Forhåpentligvis har vi fått gjennomført de arkitektur-endringene som trengs innen vi avslutter Action Research-prosessen i starten av juni. Vi er uansett nødt til å sette en strek ved dette tidspunktet.

Produktet som skal komme ut av denne Action Research-prosessen skal være en ny arkitektur på ImproSculpt og en implementasjon av denne arkitekturen. I løpet av prosessen skal vi også ha samlet nok informasjon til å belyse vårt overordnede forskningsmål.

Roller

Øyvind Brandtsegg, Kunde

- Opphavsmann til ImproSculpt.
- Fortsetter utvikling av det funksjonelle ved systemet.
- Vil bidra med evaluering av arkitekturforslag som fremmes.

Audun Småge og Thor Arne Gald Semb, Forskere

- Har som forskningsprosjekt å utbedre arkitekturen til ImproSculpt.
- Vil i prosjektet stå for design og implementasjon av forbedret arkitektur.