Software Engineering in an Artistic Project: ImproSculpt Case Study

Anna Trifonova*, Øyvind Brandtsegg°, Letizia Jaccheri*
*Department of Computer and Information Science
° Department of Department for Art and Media Studies
Norwegian University of Science and Technology (NTNU)
7491 Trondheim, Norway
trifonova@idi.ntnu.no; obrandts@online.no; letizia@idi.ntnu.no

1. INTRODUCTION

In the context of SArt project at the software engineering group of the Norwegian University of Science and Technology (NTNU) a set of case studies are being performed on artistic project which are strongly software dependant. The shared objectives of these case studies are:

- identification of critical success factors
- identification of issues that must be addressed by software engineers and artists to improve the productivity of the code development process
- development of a reference body of case studies for the interested community
- documenting lessons learned from the analysis and personal team interviews

The research question we have addressed is "How can we offer technology to the artists through software engineering, i.e. providing better tools, processes?"

This case study has been carried out with the following steps:

1) analyze the available historical data about the project (i.e. students reports and theses)
2) analyze current version of the source code and documentation, as published in SourceForge
3) plan a questionnaire the team/artist(s)
4) analyze the filled-in questionnaire(s)
5) conduct follow-up (by phone, on site interviews or via e-mails)
6) write and publish report

The current report is an exploratory case study (Oates, 2006) of the development of an artistic software, called ImproSculpt, with focus on the software issues, challenges and lessons learned.

Further the text is organized as follows: Section 2 gives details of ImproSculpt and all his versions up-to-date (ImproSculpt Classic, ImproSculpt 4 and Flyndre). Each subsection describes the product itself with its functionalities and how it differs from the previous version. We have included details of development phases and important strategic decisions in the software design. Section 3 describes the development team involved and roles carried out in the project. In Section 4 we shortly discuss the software lifecycle, workflow and tasks. The development challenges are listed in Section 5 and the lessons learned throughout the project are given in Section 6. Finally, all used materials are listed in references (Section 7).

2. SOFTWARE CHARACTERISTICS

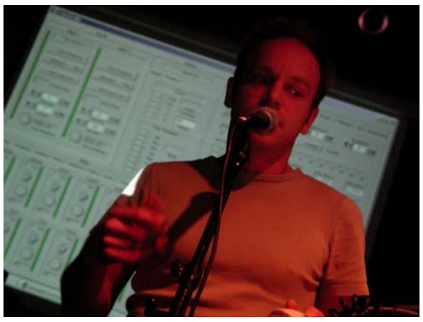
In this section we give a description of the ImproSculpt software, including overall goal, functionalities and methods used. We also give some details on the design and development choices, the tools and languages used.

ImproSculpt is software for live sampling and manipulation, algorithmic composition and improvised audio manipulation in real time. ImproSculpt might be considered also a live performance instrument. Øyvind

Brandtsegg has used ImproSculpt in a variety of performance and studio settings, collaborating with different musicians.

"Most of the time, the instrument might be able to bring you surprises... bringing in a new musical element. You are never 100% in control, it is 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."

(Øyvind Brandtsegg)



ImproSculpt comes in two basic versions; “Classic“and “4” (the number 4 representing the current major version number). The involvement of SArt project has started during the development phase of ImproSculpt4, the software subject to research was as such an intermediate version and also a moving target. Currently there are three published distributions of the software: ImproSculpt Classic, ImproSculpt 4 and Flyndre. More detailed description about each one is given further.

ImproSculpt consists of the following functionalities:

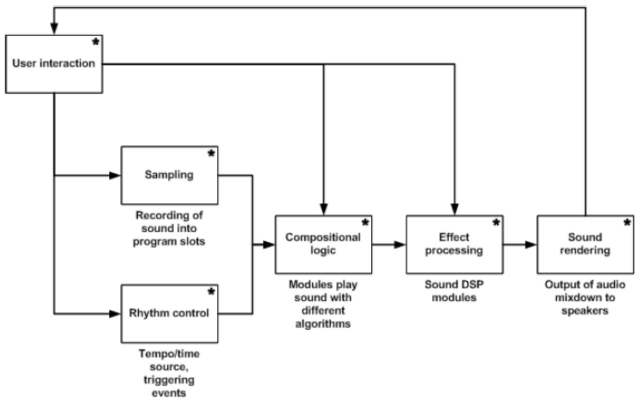

User interaction
Sampling
Recording of sound into program slots
Rhythm control
Tempo/time source, triggering events
Compositional logic
Modules play sound with different algorithms
Effect processing
Sound DSP modules
Sound rendering
Output of audio mixdown to speakers

Figure 1: ImproSculpt (Classic) process diagram

- The user interface of ImproSculpt lets the user interact with all parts of the process.
- Sampling is the recording of new sound material into ImproSculpt, usually through a microphone, but any sound source can be used. Even a feedback loop.
- Rhythm control is a process of triggering events in the program, based on the tempo and time signature dictated by the user. Specific triggers can also be programmed.
- Compositional logic uses the samples, or in some cases other sound material to produce sounds, different kinds of compositional techniques are implemented as separate modules.
- Effect processing is the application of filters to the sounds from the compositional modules, such as distortion, reverb and equalizer envelopes.
- Sound rendering sends the audio from the main ImproSculpt mixer to the platform running Csound, which typically routes it to the soundcard and subsequently loudspeakers, but that is beyond ImproSculpt’s control.

The individual algorithms used in ImproSculpt are in themselves not very complicated, but the system gains complexity through combining all of these in various ways. The most interesting results are typically achieved by combining several modules and effect filters. Several recordings can be found the composer’s web site (Brandtstegg, 2007).

The ideas behind ImproSculpt, the main implementation of code and the ongoing development of the system are mainly done by the composer and programmer Øyvind Brandtsegg. In addition, several software engineering students from NTNU have participated and helped throughout the project as part of their study and theses. The Logo of ImproSculpt was created by a volunteer artist, who responded to a communication (i.e. request) posted by the students in an OSS forum (www.linuxforum.org). Two artists offered their logo proposals and the current one was selected by Øyvind Brandtsegg.

No single sponsor can be identified for this work. However, parts of the software were developed for commissioned (i.e. sponsored) artworks, including several art installations, music compositions and live performances.

The published versions of ImproSculpt has earned its developer Øyvind Brandtsegg much respect, in the Csound community in particular. ImproSculpt Classic was used to test the functionality of Csound5 before release, as it was one of the most intricate and complex Csound orchestras to date.

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, like (Ratkje, 2005) and (North, 2007). As the author puts it, composing music with ImproSculpt is a kind of live performance in itself.

ImproSculpt is an open source project and is licensed under the GNU General Public License (GPL) schema. It is uploaded on SourceForge at http://sourceforge.net/projects/improsculpt/. This type of license was chosen mainly due to its popularity. The artis Øyvind Brandstegg stated "What I want from the license is to give us protection against theft of the code into commercial closed source projects". The developers believe that the GPL license will best guarantee such high protection.

Current version of ImproSculpt is developed with Csound and Python (see section 2.2). TortoiseCVS has been chosen as CVS client and Putty was used for generation of public/private SSH key needed to access SourceForge and upload ImproSculpt. Furthermore, the information about ImproSculpt, including links to SourceForge, documentation and installation instructions is published in Wiki. pmWiki was chosen by the students responsible for this task, because it uses files to store the data. Most Wiki implementations use MySQL or other database but they did not have knowledge about how to create and manage such database.

2.1 ImproSculpt Classic

ImproSculpt Classic was the first version of the software, which was developed entirely by the artist.

In 2000, Brandtsegg started working on the original ImproSculpt. It was developed in part as a compositional tool for a commissioned work from by Midt-Norsk Solistensemble (a vocal choir). The commission was not specifically for the software, but for a musical piece (in 2000) and the artist wrote the software to enable the performance of the piece. This put him in a position to realize some of the ideas he carried with him. Computers were 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.

The code structure of ImproSculpt Classic in itself is flat, consisting of a single large Csound script file with nearly 5500 lines of code. There are no distinct classes or objects, other than those inherent in Csound itself.

The classic version have also been used for several art installations, for example TONO's "lydløype" 2003, and "Hermetrollet" at Ringve Museum, Trondheim. For each work a custom version of the software was developed and the new (useful) functionalities were incorporate into the current development version.

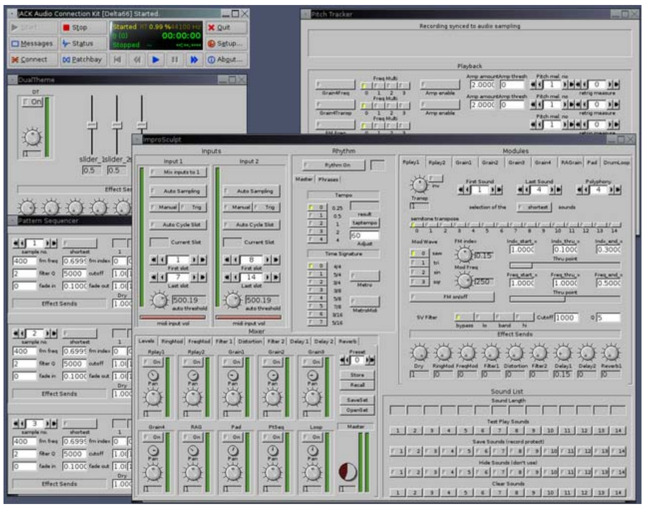


Figure 2: ImproSculpt (Classic) interface

2.2 ImproSculpt4

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.

Brandtsegg himself states in an interview that he "never planned for the project to become so big", which is why there was no long-term planning of architecture or program structure, it just evolved from several smaller sound modules. He also mentions that it has become nigh-on impossible to add new modules and composition techniques to the program, since any small change depends on changing a lot of different code.

Brandtsegg started the work on developing a new version of ImproSculpt in 2004, with more modular structure, and plans for increased modifiability. The aim was to separate parts for sound synthesis, mapping, compositional algorithms and user interface, in such a way that a change in one place does not necessitate changes elsewhere. Øyvind Brandtsegg rewrote the ImproSculpt software in the Python programming language with an interface to Csound, instead of only utilizing Csound's native opcodes.

The ImproSculpt4 version uses Python[1], wxPython[2], Csound[3], Numpy[4], Midi[5], psyco[6] and Pyro[7], TickyClav[8]. It contains about 26 000 lines of code (both Python and Csound) from which about 95% are developer written new code and 5% are reused old code (with maybe some refurbishment). The documentation standard for

[1] Python - a dynamic object-oriented programming language, http://www.python.org/

[2] Python implementation of the C++ library wxWidgets (formerly wxWindows) – a cross-platform GUI toolkit extension for Python, which also provides a handful functions, like timer and thread modules.

[3] Csound - a programming language designed and optimized for sound rendering and signal processinghttp://www.csounds.com/

[4] An OSS package for Numeric Operations in Python http://sourceforge.net/projects/numpy

[5] MIDI – Musical Instrument Digital Interface – industry-standard electronic communication protocol that defines each musical note… MIDI does not transmit audio – it simply transmits digital information about a music performance.

[6] Psyco, the Python Specializing Compiler, available at http://sourceforge.net/projects/psyco/

[7] Pyro - Python Remote Objects – and OSS Distributed Object Middleware for Python, available at http://sourceforge.net/projects/pyro/

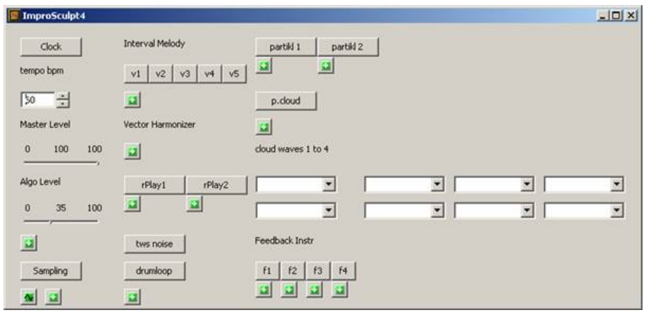
[8] TickyClav - VST instruments used in ImproSculpt - freeware by Big Tick Audio Software. The VST instrument plugins are only guaranteed to work under windows, however it is relatively easy to disable the loading of the VST plugin and this has only small ramifications on ImproSculpt usability and flexibility. Available online at http://bigtick.pastnotecut.org/index.php?action=PROD&pcode=120

ImproSculpt4 is Epytext format, using Epydoc[9] as the autodoc tool. The soundfonts currently loaded by ImproSculpt are all freely available[10] (e.g. Dsix magic, Steinway Grand Piano, etc.).

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/Uniz/BSD, MacOS, Amiga and many others. Python provides 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.

In ImproSculpt, Brandtsegg wants to utilize MIDI as a control source for parameters in the system. Csound provides an implementation of the MIDI standard.

Øyvind Brandtsegg has chosen to use Python. Ideally, Brandtsegg wishes to make the software so accessible that other musicians using ImproSculpt can add to and extend it, as it is an open-source project.

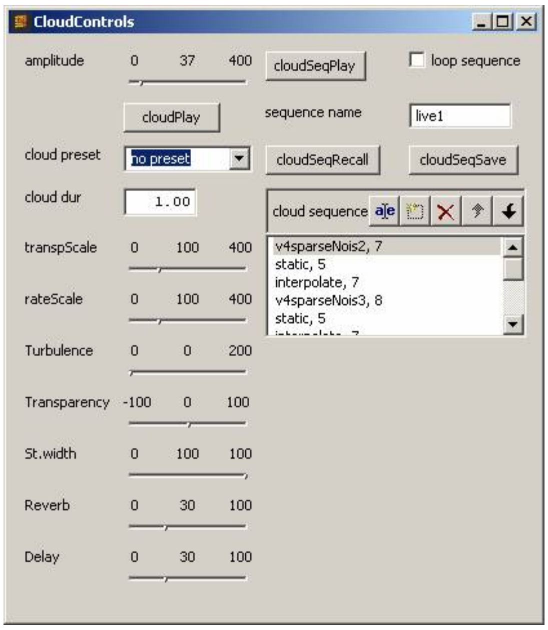


tempo bpm: 50

Master Level: 0 100 100

Algo Level: 0 35 100

Figure 3: ImproSculpt 4 main interface



amplitude 0 37 400

sequence name: live1

cloud preset: no preset

cloud dur: 1.00

transpScale 0 100 400

rateScale 0 100 400

Turbulence 0 0 200

Transparency -100 0 100

St.width 0 100 100

Reverb 0 30 100

Delay 0 30 100

v4sparseNois2, 7
static, 5
interpolate, 7
v4sparseNois3, 8
static, 5

Figure 4: ImproSculpt 4 interface for Cloud Controls

2.2.1 *Architecture*

In March 2006, two students from NTNU, as part of their pre Master study (Småge and Semb, 2005) and thesis work (Semb and Småge, 2006), entered the project with the goal to use their software engineering knowledge and “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

[9] Epydoc - Automatic API Documentation Generation for Python, available at http://epydoc.sourceforge.net/
[10] HammerSound web site - http://www.hammersound.net/

implementing it on Brandtsegg's existing functionality". The students state that, although they were interested in music and composition themselves, they realized they could not contribute much to the music side, given the vast expertise and experience in this area of Øyvind Brandtsegg. Thus their effort had been focused on restructuring the software architecture, leaving the end-user functionality as it was handed to them. In other words, they were concentrated on the non-functional requirements and worked on improving the ImproSculpt architecture according to a set of quality.

When the students took ImproSculpt to work on its architecture it "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." (Semb and Småge, 2006)

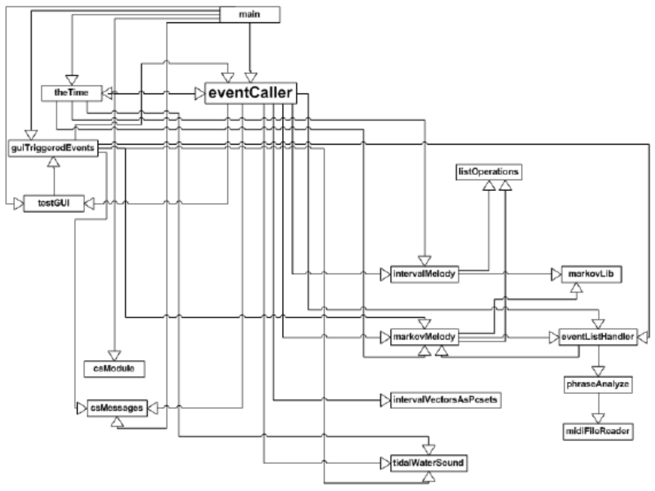


Figure 5: Communication paths in the original source code.

The students were supposed to follow Canonical Action Research method. They did not follow the methodology strictly, however used some of the main steps - reflection and evaluation. They have kept a daily log of their activities, thoughts and decisions which was later analyzed.

Through a series of discussions, informal meetings and more formal interviews the two students tried to gain understanding on who the potential users of the system will be and what are the requirements, client's motivation and ultimate goal for the product. They have used a "miniature version of Architectural Trade-off Analysis Method process[11]" (Semb and Småge, 2006). In order to modify the architecture they have prioritized the quality attributes according to their importance for ImproSculpt (performance, modifiability, availability, usability, testability, security, safety). The prioritized list was aggreed with the artist.

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
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

[11] The Architecture Tradeoff Analysis Method (ATAM) http://www.sei.cmu.edu/architecture/ata_method.html

quality attributes. The quality attributes were prioritized and agreed with Brandtsegg. On Table 1 we list them from most important to least important.

Many of the above mentioned quality attributes (e.g. performance, availability) were already at a satisfactory level. In this context the students concentrated their effords on impriving modifiability. The tactics they used is of ”polarizing responsibilities into separate packages in the system”, thus inproving the object orientation and mainaining semantic coherence (Semb and Småge, 2006).

Table 1: Prioritized list of quality attributes in ImproSculpt

Quality Attribute	**Details**
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. 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.*
2. Modifiability	Modifiability concerns changes in the system. Changes are often divided into three subcategories: 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*
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 dØs 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.)*
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. The latter is far more interesting than the former in our current context.
5. 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.
6. 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 offline system with no confidential data is minimal.
7.	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.

The students discovered that relatively few changes were needed to be done on the current version of the software in order to acchieve Model-View-Controller (MVC) architecture pattern. As the artist also wanted to utilize MIDI (and interfaces like MIDI keyboard and sensors, as in Flyndre) the separation of the user interface (View) from the controller was even more logical. Furthermore, the whole system would profit from a good separation between the the compositional logic, sound processing, Csound interface code and the utility functions.

The simple separating of files into packages greatly improved the comprehensibility/clearnes of the communication pat diagram, as it is shown on Figure 6. 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. However, there was a notable flaw in the original code - several files contained elements that belonged in other parts of the system. Additional work (i.e. restructuration of code that did not belong together) was needed for acheiving the desired clear separation, which is depicted on Figure 7.

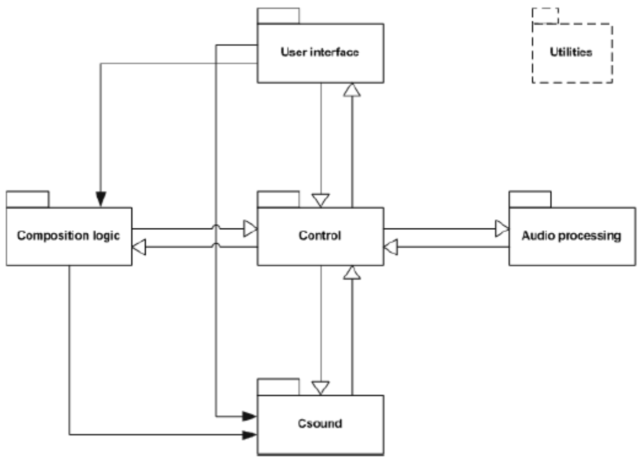


Figure 6: Communication paths between the newly created packages in the old system.

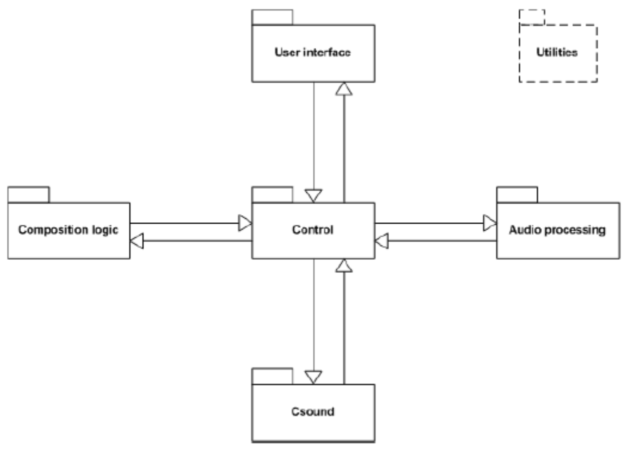


Figure 7: Desired communication paths after the architectural changes.

As reported earlier the performance levels of the system already were adequate, and did not need improvement. Throughout their work the students monitored the system's performance closely to make sure it didn't drop noticeably.

An ATAM meeting was organized between the two students and the artist. The ATAM phases that concerned functional requirements (e.g. scenarios generation and discussion) were skipped, as the students were focusing on the non-functional requirements. A list of new system requirements was derived in addition to the previous ones:

- 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.

Considering the proposed architecture the artist clarified that the classes that the students have placed in separate audio processing package "in reality had the same role as the composition classes, and that they therefore belonged in the composition package as well".

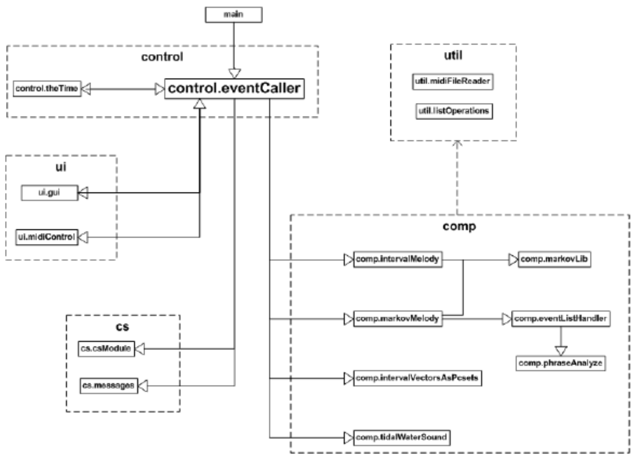


Figure 8: Communication paths after students' interventions.

The ui package represents the View in the Model-View-Controller pattern. 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.

The two students report spending much more time in understanding the system and analysing it and planning the changes, than on performing the actual changes. "As we grew more comfortable with the code, modifications became easier to perform". After performing packaging of the original files they have iteratively modified many of the files. One of the strategies was to shift pieces of code from one package to another (where it logically belonged). Furthemore, information hiding was improved by imposing proper class structure on some packages (i.e. csMessages). Several pieces of code were rewritten and generalization of some method groups with similar functionality into a single method was introduced. On the last stage of their work the students added comments to the code, in order to facilitate automatic documentation generation.

Due to insufficient communication with the artist and the fact that the artist was continuously updating and upgrading the code, at the final stage the students had to redo some of the work they had performed on the new/changed files by Brandtsegg. The modified system was submitted to the artist when no more problems or task emerged.

Several tests were performed in each iteration step to evaluate the modifications made: Number of imports, maximum number of source files in a package, file name ramification. The figures below depict the improvements achieved with each iteration.

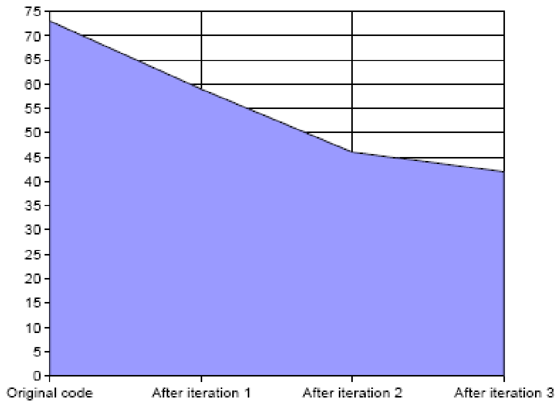


Figure 9: Number of Imports

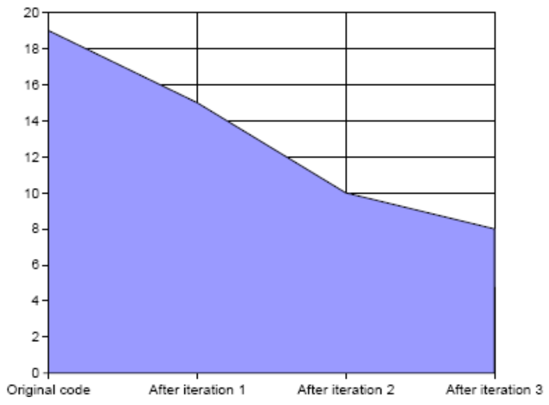


Figure 10: Maximum number of source files in a single package

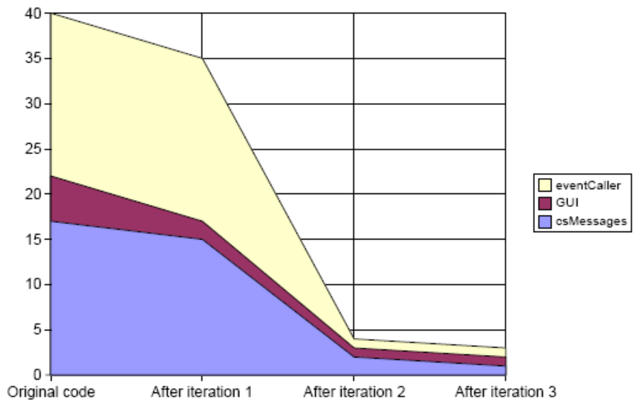


Figure 11: File rename ramifications

2.2.2 Current state of ImproSculpt4

The architecture has been further developed by Brandtsegg since the end of the Semb/Småge project. New composition modules have been added, and the modularisation have been further extended. The gui and the audio processing code have been split into separate applications, communication via network interface, and these two parts of the system can now be run on separate computers if needed. The code for the GUI has also been modularized and generalized to enable more reuse of code. New audio processing methods have been implemented, and an extensive midi control mapping included.

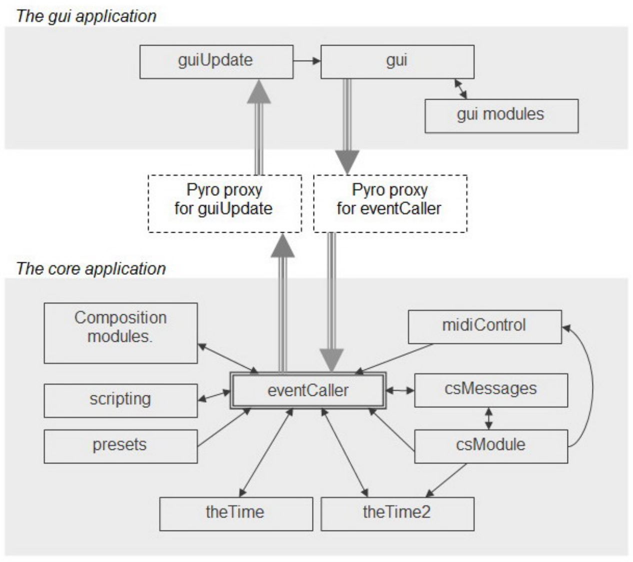


Figure 12: Current ImproSculpt 4 architecture

The core application consist of a central event caller, communicating with the audio processing module (csound), the composition modules, and the modules handling timed automation. The gui application handles the graphical user interface. The two applications communicate via Pyro (Python Remote Objects) network proxies, and thus can be run on separate computers.

2.2.3 OSS

ImproSculpt was initially created for personal use of its creator: Øyvind Brandtsegg. He quickly discovered that ImproSculpt was a useful and interesting tool for other composers and music technology enthusiasts on the globe. For this reason, Brandtsegg decided to declare ImproSculpt as an open-source distribution. Thus, he uploaded the ImproSculpt's source code on the net. Making the software publically available was supposed to help the artist, whose main focus is the compositional part, to recruit people to help, to improve, to give ideas from others points of view, to find bugs. Although ImproSculpt was declared Open Source Software by Øyvind few years ago (when the source code has been accessible on internet), a lot of management tasks are required before call it Open Source Project.

Two NTNU students as part of their pre-thesis study (Collet and Ramirez, 2006) were involved in the publishing of ImproSculpt as OSS. At that time the software was available as "ImproSculpt bundle (Flyndra, ImproSculpt Classic and Unstable ImproSculpt)". Students' intervention included licensing of the ImproSculpt source code, adding the appropriate terms and conditions documents in the software packages, publish the software on SourceForge, setting up a CVS environment and Wiki.

SourceForge was chosen because it provides most or even all necessary tools for managing OSS. In addition, it was expected that the publication in SourceForge will be a good "advertisement" of the ImproSculpt, because it will be "visible and accessible to everyone". Meanwhile, the Wikis were supposed to increase the attractiveness of the project for potential developers by offering detailed and well organized information. The students' report that at the beginning of their intervention the information related to ImproSculpt was scattered. As their supposition was that the potential developers will be also end-users, a great portion of the effort was planed to be on improving installation documentation, thus that it will be easy for anybody to get started. The two students had planed to establish a backup policy. They have added "credit" file with the names of all contributors to the project.

2.3 Flyndre

Flyndre is an interactive art installation. It is a sculpture located in Inderøy, Norway. On the right one can see the physical construction of the artwork which is placed in the natural site. The physical sculpture was made by Norwegian artist Nils Aas in 1999, and Aas did respond positively to Brandtsegg's request (in 2003) of adding a sound installation to the existing sculpture. People can walk around the sculpture or sit nearby to watch it and listen to its music. It has an interactive sound system that has the goal to reflect the nature around the sculpture. To implement this goal the produced sound changes depending on parameters like the local time, light level, temperature, water level, etc. The triggering parameters from the environment are captured by sensors and influence the generation of the music. These parameters are integrated by interaction rules into the music which is played by the sculpture all the time. Some of the rules are static (i.e. fixed results are produces by certain parameters), while other rules are dynamic (e.g. algorithmic composition). The 'content' in this interactive installation is the music. It includes samples pre-defined by the composer, which are combined with and overlaid by the dynamically generated sounds. The compositon techniques involved was designed to allow the software to take musical choices on behalf of the composer, taking into account the input parameter values at the time of decision. Such musical decisions are run on a per minute basis in the software.

Currently ImproSculpt in Flyndre uses Python, wxPython, Csound, CsoundVST[12], Doxygen[13], MIDI.

Flyndre relies on software in several of its main parts – 1) for controlling the numerous sensors that are capturing environmental parameters, 2) for incorporating the captured parameters into music that is played by the sculpture, 3) for maintaining an archive of the music generated in the past, 4) for online presentation of the artwork, the artist, the music archive and the software. The web site (part of it in HTML and part in Flash) is an important part of the whole project and includes on-the-fly animated Flash application that displays the current parameters of the environment and a live audio stream playing the current music from the sculpture. The archive of previous music as played by the sculpture is also accessible through the web site in MP3 format. The archive typically contains a one-minute sample per day.

At the controlling core of the sound installation there is a custom version of the software ImproSculpt. The artist has started the software project that implements the functionalities of Flyndra. He has been the sole responsible for development of the modules which implement the composition logic (e.g. the sampling algorithms or the integration of particular sounds which are generated based on the environmental parameters). The generated music reflects the composer's aesthetical values and/or the desired [by the artist] effect/impression on the audience.

Students from NTNU, as part of Experts in Team course (Garioud et al., 2006) have developed the technical framework for the networking and the sensors systems in collaboration with Brandtsegg and Soundscape Studios, Trondheim (i.e. for capturing parameters by the sensors and for transferring them via the Internet to the sound processing station).

3. PROJECT AND TEAM

Øyvind Brandtsegg, originally from a jazz improvisation background, has always had a strong desire to experiment and discover new ways of making music. Øyvind Brandtsegg is a musician and composer that take a special interest in electronics and computers related to the making of sound. This extended to experimenting with

[12] Extended version of Csound, particularly aiming at making the engine more extendable. This implementation features Python wrappers for Csound API, along with build-in Python interpreter. It is able to run as a Python extension module, or a VST plug-in.

[13] A documentation system (i.e. a tool for generating documentation) for C++, C, Java, Objective-C, Python, IDL (Cobra and Microsoft flavours) and to some extend PHP, C#, and D.

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". 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. He is quite acquainted with the utilization of sensor technology to make for a more physical way of interacting with the computers. He favors bigger physical movements than what you normally would do when using a computer mouse, and feels one should be able to actually dance more while making music.

> *"You might argue that I'm not a "real" programmer. This is true, as I do not yet know how to write for example in C or C++ or such. And I did not participate in making Csound, I'm just sort of a very heavy user. Anyway, I find that I can build most of the things I need using Csound, and when there's need for additional opcodes it is always possible to ask for it on the Csound mailing list.*
>
> *So, my programming might be called "semi-programming" or patch-making or whatever. In the end it's really the design bit that is interesting, what you can make it do more than how it is actually made."*
>
> *(Øyvind Brandtsegg, 2005)*

Currently Øyvind uses Python in addition to Csound to write the stuff that he needs. In collaboration with NTNU students Thom Johansen and Torgeir Strand Henriksen he had contributed to the development of Csound - the partikkel opcode[14]. In this sense he might be called a "real programmer" now.

Brandtsegg has a very exploratory and open-minded approach to music:

> *"I have worked and played with synths, delay machines and 4-track recorders since the mid-80's. For me, when I've bought a new piece of music hardware, it's always been like 'yes, this is cool, but can it do more?' […] and sometimes, when I see a new instrument in a music store, there is a feature that makes me go 'oh, neat, I don't have that'. Then I find out how to reproduce that function, and implement it in my software."*
>
> *(Øyvind Brandtsegg)*

Øyvind is the main developer and owner of ImproSculpt and its variations.

The artist Johan Bakken has proposed several variations for ImproSculpt logo. He had also worked on the web page of Flyndre previously.

The following people have been working on parts of the ImproSculpt project: Thom Johansen, Torgeir Strand Henriksen, Thor Arne Gald Semb, Audun Smege, Rune Flobakk, Thibault Collet, Maximo Ramirez Robles (all NTNU students with supervisors Sigurd Saue and Letizia Jaccheri).

The csound community as a whole has contributed greatly to the development of ImproSculpt. Snippets of code have been compied from publicly available sources and from discussions on the csound user and developer lists. Specific acknowledgements may occur in the csound source code whenever unmodified code has been copied from other authors.

On SourceForge there are three registered developers of ImproSculpt. Project Manager is Øyvind Brandtsegg, who is the composer behind the software. He is also the main developer. Additionally, two NTNU students (i.e. Maximo Ramirez Robles and Thibault Collet) have participated as developers as part of their master thesis adopting action research methodology (Davison et al., 2004). Maximo Ramirez Robles has the role of Web Designer.

[14] http://www.csounds.com/manual/html/partikkel.html

4. SOFTWARE LIFE CYCLE, WORKFLOW AND TASKS

The project has started in seven years ago (i.e. in 2000) and since then it has been continuously evolving (i.e. in development phase). The artist predicts at least 2-3 more years of continuous development of the software, or even unpredictably more. Meanwhile Flyndre installation is planned to exist for 10 years (i.e. until 2013), thus the utilization of ImproSculpt should be maintained.

The software development model might be classified as agile. No particular agile method was strictly and consciously followed, but rather the agile principles can be observed. The development team was consisting of either the artist only or of the artist and one/several students and in some cases there were external collaborators, thus always a small team. All participants were highly motivated to deliver working product. In the periods when more than one developer was involved there was close collaboration and information exchange. No formal deadlines and milestones were defined. New functionalities were developed and tested constantly in iterations where a stable version of the software was produced for the specific performances or installations. Creation of comprehensive documentation has not been a priority.

5. CODE DEVELOPMENT CHALLENGES

- The application is a moving target. At times, the specifications change faster than the implementation time. This was especially relevant during the early phases of development. Currently, the specification changes are limited to the "local module" scope, and is easier to deal with.
- Utilization of multiple CPUs or multi-core CPUs. Audio processing is done on a buffer of audio samples, this buffer must be synchronized with the previous and next buffer. This makes multi-CPU processing difficult. Realtime performance is a continuous challenge and can never be good enough. Current solution is to run the different parts of the system as separate applications (allowing audio processing one dedicated CPU/core).
- Timing problems. Designing a stable and precise clock for timed automation. The main problem is that some decisions has to be calculated at the moment when the answer is needed (to take any external changing parameters into account). The calculation does sometimes require time, but we do not want to get the result too late. This conceptual problem can only be solved in a satisfactory way by optimizing the calculations and getting faster computers

6. LESSONS LEARNED

- The artist is the initiator, project-leader, software owner, main developer and principle decision maker during the development of the artistic software.
- The collaboration of the artist with the SE students was fruitful for both sides. If the project was to be repeated the artist would still collaborate with students. However, the artist realized to have better results the students need to ne given more specific (i.e. concrete and well defined) tasks.
- The functional requirements for the system were designed and developed by the artist and the software developers were not competent to influence on them.
- The high-level functional requirements were rather abstract (e.g. live sampling and manipulation) and with quality attributes defined [definable] mostly within the art domain. The specific functional requirements were often changing [growing with time], as new ideas for musical expression appeared on day-to-day basis (i.e. the artist was continuously coming up with new ideas). There was no planning of functional requirements in advance, due to the experimenting nature of the artist. For the same reason there is no formal project schedule, apart from deadlines of commissioned works.
- The lack of software engineering knowledge started to show-up as the project grew bigger. The lack of clear architecture and more particularly the lack of modular separation led to difficulties in adding new functionality. It also did nor facilitate inclusion of other (new) code developers.
- Modularization and re-use of code were very important in ImproSculpt, as they makes it easier to maintain and easier to read.
- Without the application of SE knowledge the errors made in the first software version (even if they had been realized) are probably going to be repeated in the new one.
- OSS was requested/needed because of the desire for wider distribution of the software, help from a bigger community and sharing ideas with interested people. However it was also needed as an intellectual property protection mechanism.
- The choice of technology (programming languages, standards, etc.) was done by the artist. The artist

took care that the selected tools/languages are, as much as possible, platform independent.

- During the following years after students' intervention the artist has continued his solo work on ImproSculpt and the software has evolved and changed several times. The artist, however, has consulted and reused whenever possible the modular architecture proposed by the software engineers and he found it very useful. In addition, the artist found helpful the publishing of the software as open source in SourceForge and the utilization of the Wiki and the Concurrent Versioning System (CVS) introduced by the software engineers.
- For the artist, who is also the main developer, the flexibility was more important than ease of use. For him it is important that everything is possible, and then limit the complexity for each specific case of using the application. In this way as little as possible is hidden from him and he has a wide range of possibilities open.
- Potential users might be scared off by a huge and complex application that can do "everything". Users want small and neat applications that are easy to understand quickly.

7. REFERENCES

Brandtstegg, Ø. (2007) Øyvind Brandtstegg web site. Online: http://oeyvind.teks.no, last visited 19/11/07.

Collet, T. & Ramirez, M. (2006) IMPRO SCULPT: Open Source - Artistic Software Norwegian University of Science and Technology (NTNU), Trondheim, Norway, Report for TDT4705 - Software Engineering, Depth Study, 2006, available online at http://www.idi.ntnu.no/grupper/su/fordypningsprosjekt-2006/When_OS_meets_AS.pdf.

Davison, R., Martinsons, M. G. & Kock, N. (2004) Principles of canonical action research. *Information Systems Journal,* 14**,** 65-86.

Garioud, A., Lebegue, C., Mayende, G., Gundersen, H. M., Hæreid, M. L. & Engum, T. (2006) Flounderphonics. Norwegian University of Science and Technology (NTNU), Trondheim, Norway, Product Report for Expert in Team - Art and Software Village, 2006, available online at http://www.idi.ntnu.no/~letizia/eit2006/reports/ProductReportG2.pdf.

North, M. (2007) Web page of Ottawa Proccessed - a three piece set of improvisations. Online: http://www.sonus.ca/curators/thomson/north.html, last visited 14/12/2007.

Oates, B. J. (2006) *Researching Information Systems and Computing*, SAGE Publications Ltd.

Ratkje, M. S. K. (2005) "ADVENTURA ANATOMICA" - CD.

Semb, T. A. G. & Småge, A. (2006) Software Architecture of the Algorithmic Music System ImproSculpt. *Department of Computer and Information Science, Faculty of Information Technology, Mathematics and Electrical Engineering.* Trondheim, Norway, Norwegian University of Science and Technology (NTNU)**,** 70.

Småge, A. & Semb, T. A. G. (2005) Artistic Software. Norwegian University of Science and Technology (NTNU), Trondheim, Norway, TDT4735 - Pre Master study, 2005, available online at http://www.idi.ntnu.no/grupper/su/fordypningsprosjekt-2005/smage-semb-fordyp05.pdf.