HOVEDPROSJEKT:

TITTEL

BeMiT

BeOS Musical Instrument Tracker

FORFATTER(E):

Balazs Halasy
Bjarte Sæverud
Morten Trillhus

Dato:

23.05.2001

SAMMENDRAG AV HOVEDPROSJEKT

Tittel:	BeMiT	Nr. :
	BeOS – Musikal Instrument Tracker	Dato : 23.05.01
Deltaker(e):	Balazs Halasy	
	Bjarte Sæverud	
	Morten Trillhus	
Veileder(e):	Ivar Farup	
Oppdragsgiver:	Ano Tech Computers	
Kontaktperson:	Andreas Lund	
Stikkord (4 stk)	Music, tracking, MIDI, BeOS, BeAPI	

Antall sider: 70	Antall bilag: 20	Tilgjengelighet (åpen/konfidensiell): Åpen

Kort beskrivelse av hovedprosjektet:

BeMiT prosjektet handler om utviklingen av et stabilt og brukbart rammeverk
(og tillhørende brukergrensesnitt) av et mulig musikk tracking system

Preface

Morten Trillhus, Bjarte Sæverud and Balazs Halasy started working with the BeMiT project in November 2000. To graduate as a computer engineer at Høgskolen i Gjøvik (Gjøvik College, located in Norway), the students have to do a final (graduation) project during their last semester. After consulting Dr.Ivar Farup and Ano Tech Computers (our "employer"), the three of us decided to make a framework for a music application. As suspected, the decision turned out to be a challenging and quite interesting assignment. Results and progress of our work is published in this document.

We would like to thank the following people:

Dr.Ivar Farup
-for being an inspiring and patient supervisor

Erik Hjelmås
-for help with the project report and Lyx

Andreas Lund and Anrfinn Grønberg
-for acting as contact personnel at Ano Tech Computers

John Kenneth Grytten
-for support, beta testing and constant encouraging

Kim Bekkevold
-for ideas and input

Vegard Skjefstad
-for supplying screen adapters free of charge

This project is the beginning of an object oriented music tracker framework; hence, a lot of work remains to be done. The goal of this project is making a framework available for the implementation of next generation music tracking systems.

Gjøvik / Norway, 23rd of May 2001

Balazs Halasy, Bjarte Sæverud and Morten Trillhus

Contents

1 Introduction **1**
- 1.1 Organization of this report . . . 2
- 1.2 The theory and history behind the "tracker" . . . 3
- 1.3 Explanation of the MIDI standard . . . 4
  - 1.3.1 What MIDI is (and isn't) . . . 4
  - 1.3.2 MIDI channels . . . 4
- 1.4 Problem domain . . . 5
  - 1.4.1 Problem definition . . . 5
  - 1.4.2 Main objectives . . . 6
- 1.5 Similar and former solutions . . . 7
- 1.6 The recipients . . . 9
- 1.7 The developers' background . . . 10
- 1.8 Method of work . . . 11
- 1.9 Development model . . . 12
  - 1.9.1 The Inception Phase . . . 13
  - 1.9.2 The Elaboration Phase . . . 14
  - 1.9.3 The Construction Phase . . . 15
  - 1.9.4 The Transition Phase . . . 15

2 Requirements **16**
- 2.1 Background . . . 16
  - 2.1.1 The tracker concept . . . 16
  - 2.1.2 The instrument . . . 17
  - 2.1.3 The track . . . 17
  - 2.1.4 Example . . . 17
  - 2.1.5 The pattern . . . 17
  - 2.1.6 The play-list . . . 18
  - 2.1.7 The song . . . 18
- 2.2 Operating environment . . . 18
  - 2.2.1 Dispersion of responsibility . . . 20
- 2.3 Functional requirements . . . 20
  - 2.3.1 Representation of data . . . 21
  - 2.3.2 The user interface . . . 23
  - 2.3.3 The pattern editor . . . 24

2.3.4 Framework functionality 26
2.3.5 Required menu items and/or buttons 27
2.4 Operational requirements 28
2.4.1 Passive mode 28
2.4.2 Edit mode 28
2.4.3 Play mode 28
2.4.4 Recording mode 29
2.4.5 Miscellaneous 29
2.5 Life cycle 29

3 Analysis **31**
3.1 Apportionment of liability 32
3.2 Brief description of RUP iterations: 33
3.2.1 Pattern display 33
3.2.2 Timing and synchronisation 33
3.2.3 Sound output 33
3.2.4 Player engine 33
3.2.5 User Interface 33

4 Design **34**
4.1 Pattern Display Engine (PDE) 34
4.2 Timer and synchronizing routines 35
4.3 Sound output 36
4.4 Player engine 36
4.5 User interface 37

5 Implementation **40**
5.1 RUP iterations and C++ components 42
5.1.1 Pattern display engine 42
5.1.2 Timing / Synchronisation 42
5.1.3 Sound output / MIDI 43
5.1.4 Player Engine 43
5.1.5 User interface 43
5.2 Representation of data 44
5.2.1 Method #1 46
5.2.2 Method #2 47
5.2.3 Method #3 47
5.2.4 Method #4 48
5.3 BeAPI 48
5.3.1 BApplication 49
5.3.2 BWindow 49
5.3.3 BView 49
5.3.4 BMidi 49
5.3.5 BMidiPort 50
5.4 Components of the BeMiT project 50
5.4.1 The main() routine 50

5.4.2 BeMiTApplication 50
5.4.3 MainWindow 51
5.4.4 PatternView 51
5.4.5 InputView 53
5.4.6 PlayListView 53
5.4.7 Song 55
5.4.8 Instrument 55
5.4.9 Pattern 56
5.4.10 PlayerEngine 57
5.5 Storage 60

6 Testing and quality assurance **62**
6.1 Testing of the BeOS and the BeAPI 62
6.2 Testing of individual routines 63
6.3 Testing during the integration process 63
6.4 Testing of the whole system 63
6.5 Strategy and tools 63
6.6 Beta-testing 64

7 End discussion **65**
7.1 The process 65

8 Conclusions **67**
8.1 Common goals 67
8.2 Private goals 67
8.3 What we have learned 67
8.4 Further work 68
8.5 Group evaluation 68

9 Signatures **69**

Main part

Chapter 1

Introduction

Computer aided composing has been around for a while. According to the leading producer of professional music software, Steinberg, almost everything within the music industry today is done with the help of modern technology. Since the first sequencers showed up during the 1980ties, development of different kinds of solutions has been enormous.

In this document we shall focus on a specific music composition software solution, referred to as a "tracker". A tracker is a program used to create music without the requirements of specialized, expensive equipment. There seems to be a need for the development of a basic, next generation tracker framework. Such needs (among other factors) have inspired our team to choose this kind of an assignment.

1.1 Organization of this report

After consulting our supervisor Dr. Ivar Farup, it was determined that we should use HiG's standard template for documenting graduation projects. However, since this project is considered to be a research project, we felt that it would be impossible to fulfil all the requirements of a template that was designed to handle "locked" and stable projects. The report is therefore written using a slightly modified template, which we feel fits our needs in a better way.

It is worth mentioning that we're inexperienced writers, we cannot guarantee that the readers' expectations have been fulfilled throughout the entire report. However, we have tried to do our best, combining several methods to illuminate the various problems (and appurtenant solutions). Finally, after a lot of (both internal and external) discussions, we decided to organize the report in the following way:

- Introduction
  An introduction to the theory and history behind music tracking and some basic explanation around the concepts of MIDI communication; Definition of the assignment; list of superior requirements.
- Requirement specification
  Requirements, limitations and functionality of the BeMiT framework and its surrounding test application.
- Analysis
  Superior analysis; List of the RUP iteration.
- Design
  Design aspects and the logical structure of a possible tracker framework.
- Implementation
  Details around the implementation of the BeMiT framework and its surrounding test application.
- Testing and Quality assurance
  What was done to ensure high quality regarding the BeMiT application?
- End discussion
  Discussion of both the result and the work that was done.
- Conclusion
  The authors' conclusion and final analysis.
- Signature
  The authors' signatures.
- Appendix
  Glossary, resource list, source code, reports, etc.

1.2 The theory and history behind the "tracker"

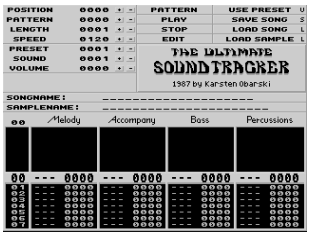

Figure 1.1: SoundTracker (1987)

The first "true" tracker ever to be developed was done in 1987 by a young and talented programmer, Karsten Obarski. He named the program "SoundTracker" (figure 1.1) and thus unknowingly founded the basis of a huge free music movement which developed in the following years and which is still going strong. Not only a revolution within the home studio, but later also on a professional basis.

Unfortunately, not everyone is familiar with the term "tracker". Most people know that there is some sort of "sequencing" software around, used by "those crazy musicians". For the record: A tracker is not sequencer (and vice versa). Sequencers represent musical data by actually recording and storing the length of notes within a melody. Trackers represent music in a slightly different kind of way, allowing the user to edit every piece of information almost instantly trough an (some people might say, "unusual") interface. Today, almost any platform (MacOS, Windows, Linux, etc.) has some sort of software that is based on the fundamental principles of music tracking.

The basic principles of tracking is simple, you have a number of columns (also referred to as tracks or channels) lined up side by side. Every track has an equal number of rows that are used for placing note events. Note events can be thought of as messages that describe what notes should be played, what instrument to use, etc. The location of events and messages throughout a combination of tracks make up melodies that may be intercepted as "music" by the human ear. Unlike sequencers, trackers are often used to produce and edit music in step-time, while sequencers are generally used to record "sequences" in real-time.

To begin with, trackers were based on reproducing digitised sound samples (with the use of a computer's built in sound chip); while sequencers were using a technology called MIDI to communicate with external devices such as synthesizers, drum machines and so on. Today, both sequencers and trackers are capable of producing sound in many ways, including the methods just mentioned. For the record: Trackers too can record sequences in real-time, but the major advantage (compared to sequencers) is the instant (some might say raw and brutal) access to music data .

1.3 Explanation of the MIDI standard

M.I.D.I. stands for Musical Instrument Digital Interface. A technology which gives musicians the ability to play more than one instrument at a time, sequence multiple performances for precise editing, score music as it is played and much more. The concept of MIDI is most easily understood as being a musical instrument communications network that enables MIDI compatible instruments to "talk" with each other through a common electronic language.

MIDI was first introduced in 1983 when, in a display of mutual cooperation seldom seen in modern industry, several major musical equipment manufacturers demonstrated how competing products could "play" together using MIDI code (which they had developed jointly). To the enormous benefit of all musicians many manufacturers further agreed to use the MIDI specifications for all their future products. MIDI is still alive (even tough the basic specification is more than 18 years old) and is frequently used by both professional and hobby musicians around the globe.

1.3.1 What MIDI is (and isn't)

The first barrier for the novice to overcome is the misconception that MIDI transmits sound. Unlike records or CDs, which are a reproduction of the sound itself, MIDI devices "talk" to each other using a code that describes every aspect of their specific "performance" (such as which keys were struck, how hard they were hit, how long they were held, and so forth). This performance data is transmitted as it is played (in real-time) using MIDI code, but it is the instrument receiving the code that produces the sound. A good way to think of MIDI is to compare it to a roll of music in a player piano. The "performance" is in the roll, but the "sound" is in the piano.

1.3.2 MIDI channels

The single most powerful aspect of MIDI is its ability to receive or transmit on one, all or any combination of 16 MIDI channels simultaneously. Each channel is capable of either transmission or reception of all MIDI messages independently of and simultaneously with all other channels. All MIDI devices receive, and most transmit MIDI messages (note on/off messages, preset changes, etc.) instantaneously.

1.4 Problem domain

Unfortunately, the solutions (trackers) of the past are dying. The community is literally screaming for new standards. There seems to be a strong need for a portable framework that can be used for further development on different kinds of platforms. As other software packages, trackers also need to be redesigned and rewritten using modern engineering methods like for example object oriented development. Since there is (at the moment) only one tracker-like application (a commercial product) for the Be operating system, and since we wanted to work with a modern, object oriented and preferably unknown environment, we chose to develop a tracker framework for the BeOS.

1.4.1 Problem definition

The goal is to construct a stable tracker framework with an integrated graphical user interface. This involves design and development of methods, source code, libraries and documentation that eventually will result in a foundation for a possible music tracking system. Our main objective is to build a stable and flexible framework that can easily be upgraded and/or expanded. Development of an appurtenant GUI is considered as low priority.

This project may be characterized as a research assignment, resulting in quite a lot of time-consuming experimentation. Keep in mind, that none of the developers have done anything like this before.

We wish to develop a solution that is capable of playing notes (music) using MIDI communication to "control" an external sound source, for example a MIDI compatible synthesizer. The test application (which demonstrates the framework) shall present a primitive, but working interface, including (among other things) a fully integrated pattern editor, giving the user the ability to test the functionality of the framework, and to actually compose / play music. However, our goal is not to implement a complete system, but to "understand" and to build a stable foundation for future development of a more advanced tool for musicians.

1.4.2 Main objectives

Theory:

- Understand the concepts of music tracking and the MIDI protocol
- Understand the Be operating environment and the BeAPI

Our work shall result in a framework (and a surrounding test application) that is capable of:

- Representing music data (instruments, patterns and songs)
- Instant rendering of music data (represented within a pattern editor on screen)
- Intercepting user generated messages (keyboard and mouse)
  (also acting on the basis of the intercepted messages)
- Editing music data on-the-fly, based on input from the user
- Playing music by traversing music data stored in memory and by sending messages to a MIDI compatible device
- Saving and Loading (disk/memory) of music data

1.5 Similar and former solutions

Tracker-like applications are to be considered as old news. Since the 1980ties, there have been several trackers around, written for different platforms. Unfortunately even the best solutions are pretty much dying these days. There are a number of reasons for this, some to be discussed in the current section.

Successful products of the past run mostly on (so to speak) "ancient" operating systems. Pure assembly code written for specific CPUs and hardware was many times the programmer's only way to success. Speed, size and direct access to hardware were essential aspects of the past. Young and talented fanatics coded most of the trackers seen so far. Standardization and compatibility wasn't always in focus; different applications emerged from around the globe, competing within the tracker community. Some offering quite astonishing features. Most of these applications were public domain, i.e.: free of charge.

Today, commercial products of this kind may be purchased for around $50 - $200 (USD). These are mostly restricted to one operating system and do not always offer what the masses are looking for. There seems to be strong belief in that old-style/non-profit projects were much better than today's commercial "(l)imitations". Unfortunately - the success behind earlier applications was (is still) mostly kept in secret; release of source code seems to be out of the question. A short note on this: "Fasttracker II" - done by Swedish students, a milestone for PC musicians was discontinued around 1997; for reasons unknown the source is still not released to the public eye.

Discussion regarding "which software to use" is beyond our document's aim. We'll gently round up this section by presenting a list of earlier solutions along with short comments. The applications listed here are the ones believed to have made serious impact on the tracker-community:

SoundTracker
Author: Karsten Obarski
Release year: 1987
Coded in: Motorola 68000 assembly
Platform: Commodore Amiga under Amiga DOS / Workbench
Notes: Capable of playing 4 channels using 8 bit digitised audio

Title: ProTracker (v2.3)
Author: Peter Hanning
Release year: 1992
Coded in: Motorola 68000 assembly and x86 assembly
Platform: Amiga DOS/Workbench, later PC and Apple Macintosh
Notes: Capable of playing 4 channels using 8 bit digitised samples, built in sample editor

Title: OctaMED
Author: Teijo Kinunnen
Release year: 1991
Platform: Amiga Workbench (earlier versions for Amiga DOS), later on PC
Notes: Capable of playing more than 4 channels, offers MIDI
compatibility and software generated audio, offers also a
built in sample editor and score notation

Title: ScreamTracker
Author: Future Crew
Release year: 1994
Platform: PC, DOS
Notes: Capable of mixing 32 channels of 16 bit digital audio

Title: FastTracker II
Author: Magnus Hoegdahl and Fredrik Huss
Release year: 1994
Platform: PC, DOS under DPMI
Coded in: Borland Pascal 7.0 and x86 assembly
Notes: Capable of playing 32 channels of 16 bit interpolated digital audio,
basic MIDI compatibility, built in sample editor and CD ripper.
Considered as the most popular tracker ever developed, basicly
the best combination of ProTracker, OctaMED and ScreamTracker
(which all were major successes of the past).

Keep in mind that this list is strongly focused on popular projects of the past. It would be impossible to present a complete list of all tracker-related solutions. There are several trackers available today, but unfortunately many of these are either expensive commercial packages and/or discontinued projects. Actually, there is a promising open source application available these days, bearing the name "SoundTracker for Linux". Unfortunately, a closer look at the source reveals that it is done in pure C, not C++.

1.6 The recipients

The project itself is mainly aimed at developers who are involved and/or interested in development of tracker based music applications. We hope that the documentation of our results might help others, and that it will shed some light on a couple of "things". By "things" we mean information (and knowledge) that is generally hard to find, and/or hard to come up with.

The art of tracker design is not an easy matter. There seems to be a great deal of confusion among/between users and developers, also many pitfalls. We shall try to solve a couple of general (common) problems regarding design of a framework for such applications. The goal is to build a solid and developer-friendly framework that can be easily expanded to suit different needs within the community. Our mission is to help others to understand, and also help developers to get either started, or to get further.

We hope our work helps to recruit developers to the Be environment by shedding some light on operating system and API specific subjects. We also hope that both musicians and developers working with other operating systems realise the positive aspects of the BeOS, and that they also might get inspired and encouraged to do further development of our framework. Further development is also possible using other platforms. Our work shall result in an object-oriented implementation that is more or less considered to be portable.

It is strongly recommended that the reader understands object oriented programming and development. Formal education is considered to be unimportant, although some computer related education might help. Computer engineer students shall not have any problems understanding the contents of this document.

The reader must not be afraid of the unknown, and must be able to understand the concepts of complicated hierarchies and solutions. It is also recommended (but not required) that the reader has worked with (preferably done some sort of development on) various operating environments. We decided to write this document in English since we wanted to reach a broader range of public.

1.7 The developers' background

All members (three) of our development team are currently wrapping up three years of higher education within computer science. During these years, we feel that we have achieved some fairly good programming skills and acquired better understanding of how computer systems actually work. Among various languages, we feel that we have learned quite a lot of C and C++. All members of the development team have taken classes that relate to object-oriented analysis, development and programming. However, we were (are) certainly not expert programmers, but inexperienced students.

Balazs was already familiar with the tracker community; during the past years, he has used various applications for computer aided musical composition. He is one of the many who prefer using trackers instead of sequencers. Balazs was the one who took the first steps, doing some basic background analysis. Bjarte and Morten were contacted and asked if they were interested in doing something "different" for their graduation project at Høgskolen i Gjøvik. Fortunately Balazs managed to convince Bjarte and Morten that together as a team they might actually "pull it off".

None of the project members have done any development for the BeOS, actually, Morten and Bjarte haven't even seen the operating system before. Former (and positive) experience with alternative operating systems (such as Linux) encouraged the team to aim at the unknown. Since BeOS had it's own ready-to-use API and IDE, we thought of it as a good opportunity to expand our knowledge and to comprehend the fundaments of building a stable framework using a system (at the time) totally unknown to us. It is also worth to mention that neither Morten nor Bjarte have been using any kinds of music application before the start of this project.

1.8 Method of work

We decided that Balazs should be the project leader. The group felt that he was the most experienced member, a man who had the skills to run the show; also, he had been involved in the tracker community for a couple of years already.

Since none of us were completely comfortable with software development for the BeOS; and certainly not comfortable with tracker development, we decided to do much of the work together. At first, everything was done with the presence of all members. Research, planning, design, coding and experimenting had to be done in plenary so that everyone could understand the basic principles. We could not afford to leave any of the members behind (that would have resulted in a disaster later on). This kind of development method is considered to be a bit ineffective, since it requires a lot of time. Unfortunately, we were pretty much forced to work this way since we often felt that one person was not enough to solve complex problems on his own. Multi personnel research and experimenting was constantly essential, making parallel development almost impossible. However, when everyone had enough information and background (and when we felt that it was necessary), the team decided to split up. Individual work was done; later to be merged into the project. Documentation and design / coding of less critical parts of the system was also done on an individual basis. We would like to point out that only a fraction of the time used for research, design and planning was used to do the actual implementation. The complexity of the system required a great deal of analysis and reverse engineering.

A custom made blackboard located in our office played an important role throughout the entire project. It was used for frequent analysis, planning, organization and illustration of problems and solutions. Unfortunately, using the blackboard resulted also in information loss; we were not clever enough to document everything (that came up on the board) on solid media.

The BeMiT project was done using other forms of methods that are considered to be "traditional" when it comes to graduation projects at HiG. Most of the projects at HiG are carefully planned and carried out using the waterfall model of development. Since our assignment required a great deal of (constant) experimenting we simply couldn't follow a "locked" waterfall model.

We have worked as a team, gathered information from loaned books, IRC, mailing lists and discussions with “key men” at HiG.

1.9 Development model

We have chosen to use "The Rational Unified Process" to be the primary development model. The greatest advantage of this model is that it reflects an iterative development process, supporting both independent and dependent increments. It is obviously a good model to use for projects that require a great deal of just-in-time research and experimenting.

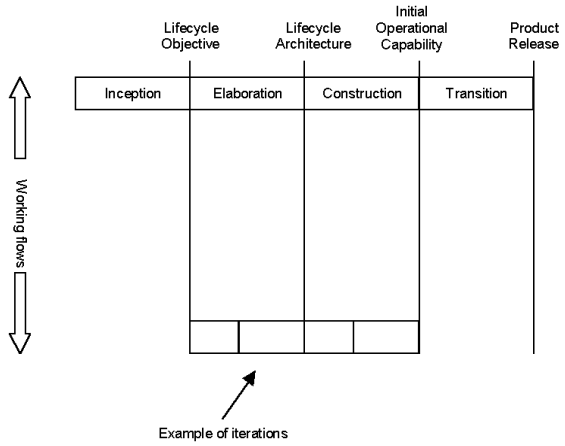

Figure 1.2: The Rational Unified Process

The Rational Unified Process is divided up into four major phases(figure1.2):

- Inception
- Elaboration
- Construction
- Transition

These phases are again divided into iterations, which are based upon the commonly known waterfall model (figure1.3).

Iterations make the development resemble a component driven process. The developers may focus on one component at the time, and analyse the current problem. This method, divide and concur, helps to focus on single problems and to find solution to these; instead of analysing and trying to solve all the problems at once. We feel that this kind of development suits our needs since it can easily handle unknown requirements and aspects that emerge during the process, instead of locking everything at the beginning. The process supports risk analysis, and almost automatically sheds light on the most critical components of a system. The developers can thereby easily prioritise and arrange such critical phases of the development process.

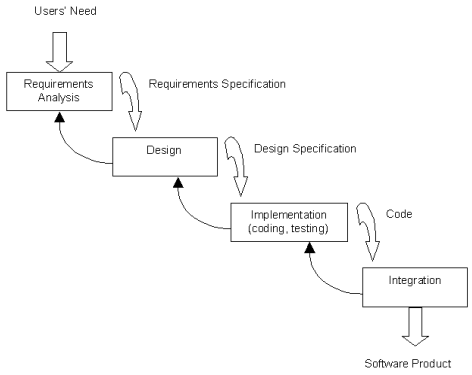

Figure 1.3: The Waterfall Model

1.9.1 The Inception Phase

The overriding goal of the inception phase is to achieve all the descriptions and requirements, which arise from all of the partners that are involved within the project.

The primary objectives of the inception phase include the following:

- Establish the project's software scope and boundary conditions, including an operational concept, acceptance criteria and descriptions of what is and is not the product.
- Estimate risks (Ranking all critical components to achieve more controlled and satisfied development process.)
- Estimate the time and make a schedule for the entire project.
- Discriminate the critical parts of the system, which will drive the system's functionality and will shape the major design.

The outcome of the inception phase is creation of the following artefacts:

- A vision document, that is, a general vision core project's requirements, key features, and main constraints.
- A project plan, which shows the phases and iterations.

Milestone: Lifecycle Objective

At the end of the inception phase is the first major project milestone, "Lifecycle Objective". The criteria of this milestone are that all the partners must have agreed on the core requirements and accepted the project plan.

1.9.2 The Elaboration Phase

The purpose of the elaboration phase is to analyse the problem domain, establish an architectural foundation, develop the project plan and eliminate the project's highest-risk elements. To accomplish these objectives, we must be able to have a "mile wide and inch" view if the main system. Architectural decisions must be made with an understanding of the whole system, its scope, major functionality, and non-functional requirements such as performance requirements.

The primary objectives of the elaboration phase include the following:

- Define, validate, and baseline the architecture as fast as practical.
- Baseline a high-fidelity plan for the construction phase.
- Resource on the chosen architecture will support the project's vision.

The outcomes of the elaboration phase are as following:

- Supplementary requirements that capture the non-functional requirements and any requirements that are not associated with a specific description from the partners.
- A software architecture description.
- A revised risk list.
- A development plan for the overall project, including the coarse-grained project plan, which shows iterations and evaluation criteria for each iteration.

Milestone: Life Cycle Architecture

At the end of the elaboration phase is the second important project milestone, 'Lifecycle Architecture'. At this point, we have to examine the detailed system objectives and scope, the choice of architecture, and resolution of the major risks.

The main evaluation criteria for the elaboration phase involve the answers to following questions:

- Is the vision of the product stable?
- Is the architecture stable?
- Do all partners agree that the current vision can be achieved if the current plan is executed to develop the complete system, in the context of the current architecture?

1.9.3 The Construction Phase

In this phase most of the development work is done. To simplify the process, the construction phase is usually split up in so called "iterations". During the iterations, critical parts of the system are implemented. The iterations follow the waterfall model. Every major part of the system is carefully analyzed, planned, designed and implemented.

Artifacts of the construction phase:

- Beta release.
- Description of the current release.

Milestone: Initial Operational Capability

After finishing up a construction phase and/or iterations, the development team would have a product ready for testing.

The following questions are to be asked:

- Is the product stable enough, is it possible to do further upgrading?
- Did we follow all the requirements that were set at the beginning?

1.9.4 The Transition Phase

The transition phase is usually used for writing reports and processing feedback from users of a system.

Artifacts of the transition phase:

- End product.
- Project document.

Milestone: Product Release

Our team tried to follow the Rational Unified Process during the entire process. Results and experiences are to be discussed at the final chapters of this document.

Chapter 2

Requirements

2.1 Background

2.1.1 The tracker concept

A tracker is an application designed for composition of musical pieces.

The basic concept is simple: you have a number of instruments, and you can arrange them on so-called tracks. A set of tracks is called a pattern, a combination of patterns form a song. (figure2.1).

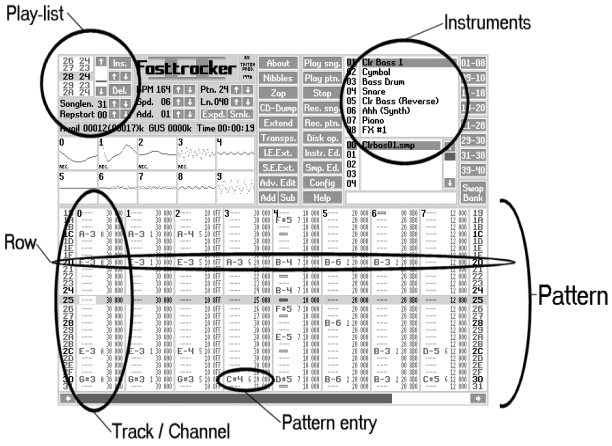

Figure 2.1: Essential parts of a typical tracker.

2.1.2 The instrument

An instrument within the application is basically an interface to a desired sound source, which can be a digitalized sample in memory, a MIDI channel/preset or real-time synthesized audio. Instruments are the sounds/sources that represent the instruments within a song.

2.1.3 The track

A track (or channel) is a column where the user can place notes, capable of "playing" one note at the time. This way, you can create melodies using your own progressions of notes. However, this doesn't mean you need any knowledge of music theory. The composer can place notes without knowing which note it is or which scale he is using. The number of entries that are left empty (before insertion of a new note, or a note off command) determine the length of the previous note. Let's say you put an A note on a track and leave 7 empty entries before the next note. If, instead of leaving 7 entries, you leave 14, you'll have a note that lasts twice as long. Simple isn't it? Obviously the composer can use many tracks to create a song.

The first "real" tracker, ProTracker v1.0a supported only four tracks (since the Amiga sound chip, Paula, was capable of mixing only four channels of 8 bit digitised audio). Today, trackers support 16, 32, 64, etc tracks which are more than enough for even an extremely complicated song. Some trackers also support the use of MIDI communication, allowing the user to play notes using external hardware. Sometimes composers use one track for each instrument. For example the first track for the solo, the second for the bass, the third for the drums and so on, however, it is possible to combine several instruments in one track (as long as they don't overlap) - this feature ensures almost unlimited ways of organising a song.

2.1.4 Example

A basic drum set could for example be arranged by putting a bass drum at entries 0, 4, 8, 12 etc. of one track and putting some hihat at entries 2, 6, 10, 14 etc. of a second track. Of course you can also interleave several instruments on the same track, but they can't overlap, otherwise the previous note is stopped when the next one sets in (look at the screen shots for a more visual explanation).

2.1.5 The pattern

A set of tracks (channels) that are played at the same time is called a pattern. A pattern typically has an equal number of 64 entries per track; playing is made possible by cycling through these entries (also called rows) at equidistant time intervals.

2.1.6 The play-list

A combination of patterns form a song, this combination (in what order the patterns are to be played) is called a play-list.

2.1.7 The song

Finally, a complete song is a file containing various patterns and instruments, including a position list, which specifies playback order of the patterns.(figure 2.2).

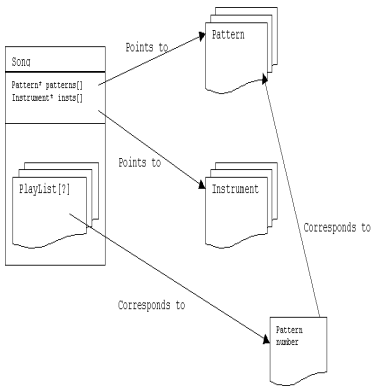

Figure 2.2: Representation of a song

2.2 Operating environment

The BeMiT (framework and its surrounding application) shall run on top of the Be Operating System. There shouldn't be any problem running the application on a 166 MHz IBM compatible (x86) PC with 32 Megabytes of RAM. For playback, a working soundcard driver (which supports MIDI communication) and a MIDI compatible device (external/internal) is required. The test application must be run inside a workspace (desktop) that is set to at least 800x600 pixels and has a color-depth of 32 bits per pixel.

The Be Operating System

The Be operating system (BeOS) is designed to take advantage of modern computer hardware, particularly machines with more than one CPU. The BeOS offers:

- Preemtive multitasking
- Protected address spaces
- Virtual memory
- Attributed file system
- Dynamically loaded device drivers
- Interapplicatoin messaging
- Built-in networking
- Real-time streaming of audio and video data
- Shared, dynamically linked libraries
- An object-oriented programming environment

The BeOS is designed for efficient multithreading from top to bottom. For example, the application framework creates a separate thread for each window. On systems with more than one processor, the kernel automatically splits assignments between the CPUs and gives priority to threads that need uninterrupted attention. The application programming interface (API) is designed to make the features of the BeOS easy to use. Written in C++, it includes numerous class definitions from which programmers can take much of the framework for their application(s).

Software Overview

The BeOS software lies in three layers:

- A microkernel that works directly with hardware and device drivers.
- Servers that attend to the needs of running applications. The servers take over much of the low-level work that would normally have to be done by each application.
- Dynamically linked libraries that provide an interface to the servers and the Be software that programmers use to build their application(s).

Applications are built on top of these layers, as illustrated in (figure2.3).

The API for all system software is organized into several "kits". Each kit has a distinct domain-there's a kit that contains the fundamental software for creating and running an application, a kit for putting together a user interface, one for accessing the file system, another for networking and so on.

What are the technical advantages of developing for BeOS?

Developing software for BeOS leads to better performance, lower latency, and a better experience for users. By taking advantage of the BeOS threading capabilities, highly responsive applications can be created that really show off the power of the underlying hardware. The kernel guarantees 250 microsecond thread scheduling latencies. Windows can't guarantee much less than 50 milliseconds. BeOS is designed to be responsive from the ground up. A user isn't stuck waiting because the kernel is busy doing something else. Even 50 millisecond latencies can lead to a glitchy/jerky UI. With a performance at 250 microsseconds, a poor UI experience is much less likely to happen.

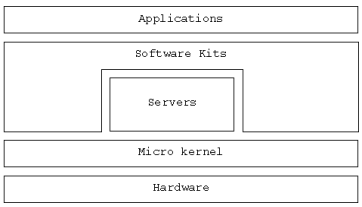

Figure 2.3: The Be software layers.

2.2.1 Dispersion of responsibility

- PC/CPU: responsible for timing and playing of notes via MIDI communication.
- Soundcard: responsible for acting as a MIDI interface.
- MIDI device: responsible for producing actual sound.

2.3 Functional requirements

The specified system is mainly meant to be a framework for a music application. Design and implementation is to be tracker oriented, this means that representation of data, functionality and user interface shall strongly resemble traditional tracker design. For convenience we'll split the framework in three major parts (only in this chapter).

Required (basic) aspects of the framework:

- Representation of data (Songs/Patterns/Instruments).
- Functionality (Player engine and such).
- User interface (GUI and message handling).

2.3.1 Representation of data

The system must represent different kinds of music related data. Representation and organization is to be done on a full-scale object oriented basis. Exclusions may occur in special cases, however, total design must result in a strong and perspicuous object oriented structure. Design and implementation must be done with future enhancements in mind. Memory shall in most cases preferably be dynamically allocated. Custom routines for garbage collection are to be used when/if needed. To minimize unnecessary waste, focus on optimized memory usage must be strong.

Data/information related responsibilities of the framework

- Representation and organization of instruments.
- Representation and organization of patterns.
- Representation and organization of play-lists.
- Representation and organization of songs (Multi document support).

Representation of a song

A song is basically a combination of various instrument definitions, patterns and a play-list. To fully represent a song, we would need the following (although minimal) set of attributes:

- The song's title.
- The composer's name.
- A text of description / comment (optional).
- A number of instrument definitions.
- A number of patterns.
- A play-list (order of patterns).
- The song's tempo.

Representation of an instrument

As previously defined, a song contains (among other things) a set of instruments. When composing/playing, instrument numbers are used inside patterns so that notes can be played using various (desired) instruments. An instrument (contained in a song) might be considered as an "interface" to a sound source. In our case, such an instrument definition would have to carry various MIDI attributes such as for example which MIDI channel to use. An instrument must have the minimal set of the following attributes:

- A unique identifier.
- Name of the instrument.
- Description/Comment (optional).
- Volume.
- MIDI channel.
- MIDI preset.

Representation of a pattern

The pattern is the "heart" of the system. A sequence of "patterns" ordered in a "play-list" will make up a "song". Generally, the pattern's name, description and actual music data needs to be represented.

A pattern shall (following the tradition) consist of entries ordered in columns (channels/tracks) and rows. The default number of rows is set to 64 (this is equal to 4 * 16 1/16th notes). Channels belonging to a specific pattern must consist of equal number of rows. Dynamic resizing of patterns ought to be implemented. The amount of channel(s) may vary from 1 to 64. The amount of row(s) may vary from 1 to 255. Each pattern may have its own dimensions and is considered as a block of independent music data. We simply follow the tradition, using a common set of attributes for each entry in a pattern. An entry shall hold information about what note to play, which instrument to use and an effect definition. In addition, we extend the standard tracker format by introducing a set of two extra attributes for each entry, panning and velocity. Coming up next, a brief description of an entry's attributes:

Note event:

The note event shall tell the system what note to play (for example a C or an F#) along with information about which octave to use. Basically, a note event may be a definition of a NOTE-ON, NOTE-OFF or simply just an empty "do nothing" message.

Instrument number:

The instrument number is used to pinpoint what instrument (a song can consist of many instruments) to use when playing a note, allowing asynchronomous combination of multiple instruments on a single or track and/or synchronomous combination of several instruments played on multiple tracks.

Panning:

This attribute is meant to simplify the placement (and navigation) of notes inside the sound picture. Unfortunately, MIDI doesn't support direct placement of notes. Panning of MIDI channels is possible, but the purpose of this attribute is to give musicians the ability to navigate single notes. If we navigate/pan port a MIDI channel, all notes that are played on the specific MIDI channel will be affected. Since the prototype of BeMiT will not have any support for other sources than MIDI (for example digitised sound samples), implementation of the panning attribute will be reserved for future upgrades.

Velocity:

A parameter, which holds information about what velocity to use when playing a note. This will be a somewhat similar, but enhanced implementation of the volume attribute known from previous (famous) trackers such as ScreamTracker and Fasttracker II. The velocity attribute shall be implemented in a compatible way to ensure easy conversion of songs made with earlier trackers (MOD/S3M/XM standard). Some sample-based effects may also be represented here allowing multi-processing (combination) of effects/messages within a single entry. For now, the velocity attribute shall only carry information about velocity. Reservation of other definitions is to be made, but there is no actual need for implementation of these at the current stage.

Effect:

The effect attribute shall hold information about special effects/messages, which are processed during playback. A total reservation of traditional (Protracker, ScreamTracker and Fasttracker II based) effects must be made (for backward compatibility) along with implementation of custom (BeMiT specific) effects/messages. Some of the traditional effects are sample based; such effects will only be reserved but not fully implemented.

An exhaustive overview/specification of the note numbers, instrument numbers, panning information, velocity and the effect attribute can be found here: [Appendix K, L, M, N, O, P]

2.3.2 The user interface

User interface is considered to have low priority. The main goal is to develop a framework prototype, not a fully integrated ready-to-use system. We must

however implement a basic interface, but keep in mind that this is done only to ensure demonstration of the framework's functionality.

The main interface shall be put inside a non-resizable window, with the dimensions of 800 (horizontal) and 600 (vertical) pixels. Standardized BeAPI specific components (such as BWindow, BViews, BButtons etc) shall be used for building parts of the user interface. The window shall be split in half (vertically). The upper portion of the window must include a drop-down menu along with some gadgets (buttons, text fields etc.). The lower portion of the window shall include a custom-made pattern editor. The application (actually the window itself) must be responsible for handling message events (keyboard and mouse). The user interface must be implemented in a way so that it does not disturb (or slow down) critical functionality.

Main aspects of the user interface:

- Point and click / window oriented graphical user interface.
- Fully integrated pattern editor.
- Message handler.

2.3.3 The pattern editor

The pattern editor is the heart of the user interface. It is used for displaying, editing and navigation inside the patterns of a song. The pattern editor may be a custom rendered block of graphics and shall more or less resemble traditional (earlier) pattern editor design. Since the pattern editor is a critical part of a tracker's GUI, focus around its development must be strong. At all times, the editor shall instantly display the contents of the current pattern at a given position. It must render all parameters of the pattern's entries: note events, instrument numbers, panning, velocity and effect parameters.

As previously mentioned, patterns are split into channels (tracks) and rows. Entries on the same row are to be displayed side by side (horizontally). Entries in the same channel shall be listed vertically. A single track carries sequential information about what note(s) to play, and shall be represented like this:

```
00 --- 00 0 00 000
01 G#5 01 0 60 000
02 --- 00 0 00 000
03 OFF 00 0 00 000
04 --- 00 0 00 000
```

Row#1 : [G#5] [Instrument=01] [Panning=0] [Velocity=60] [No effect]

Row#3 : [NOTE OFF command]

The first column displays the note event, the second column displays the instrument number, the third column is reserved for panning entry, the fourth

column displays the velocity to be used when playing notes, the fifth column is for displaying special effects / messages. In addition to displaying, all columns must be editable. A simple rectangle of some sort shall surround (pinpoint) the parameter in focus (the one to be edited). Editing and step-time composing shall be done via keyboard interaction.

The following mock up shows a possible display of nine rows and four channels of an empty pattern:

	Channel 1	Channel 2	Channel 3	Channel 4
Row 0	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 1	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 2	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 3	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 4	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 5	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 6	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 7	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 8	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000

The display is to be split in 3 parts. Upper, middle and lower, like this:

	Channel 1	Channel 2	Channel 3	Channel 4
Row 0	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 1	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 2	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 3	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 4	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 5	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 6	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 7	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000
Row 8	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000	— 00 0 00 000

The middle part shall display the current position inside a pattern, while the upper and lower portions respectively display previous and upcoming rows. A row is nothing more than a combination of pattern entries on different channels. Navigation will take action using "standard" navigation keys like the arrow keys and the TAB key. Take a closer look at [Appendix Q]for a complete list of basic (traditional tracker) keyboard definitions.

A final result of the pattern display engine must be quite configurable, allowing third party developers to tweak the rendered display to suit their needs.

2.3.4 Framework functionality

The framework must have routines that provide some basic tracker-functionality:

- Creation, deletion and navigation of patterns.
- Creation, deletion and navigation of songs.
- Creation, navigation and specification of instruments.
- Arrangement of play-list.
- Playback.
- Load/Save functions.

Playback

Playback of notes shall be done using MIDI communication. A synchronized routine/loop of some sort shall have the responsibility of traversing through a song's pattern data. While traversing, it shall detect notes, events, messages and accordingly send data to external device(s) via the MIDI standard. Notes that are on the same row must be played simultaneously. Playback must be accurate; latency, jitter and/or accumulation errors shall not be tolerated.

As mentioned earlier: A pattern typically consists of an equal number of entries per track; these entries are cycled through at equidistant time intervals. If a note is already playing on a track (channel), a NOTE OFF message must be sent to turn off the currently playing note before a new note is played, making it impossible to play more than one note at a time (per channel). This is done to prevent note hanging and to preserve the traditional style of tracking. If no instrument number is defined, the last instrument and its "effect configuration" shall be used. Since much of this is common practice, we are not going to dig deeper at this point. Detailed description and "secrets" shall be revealed in the coming chapters (design and implementation).

Loading and saving

Basic storage routines shall be implemented so that songs can be saved to disk and/or loaded into memory. Compression may be implemented but is not required at this stage.

2.3.5 Required menu items and/or buttons

Overview of required menu-items and/or buttons which are to be implemented to enable use of functions within the framework:

General functionality:

- Quit/exit application.
- About popup.

Song related functionality:

- Creation.
- Loading.
- Saving.
- Closing.
- Navigation between songs (Multi Document).
- Editable song-title field.
- Editable composer field.

Pattern related functionality:

- Creation.
- Removal.
- Navigation between patterns.

Play-list related functionality:

- Insertion/Deletion of positions.
- Inc/Dec of patterns in positions.
- Navigation between song positions.

Instrument related functionality:

- Insertion of instruments.
- Selection of MIDI channel.
- Selection of MIDI preset.
- Selection of Volume.
- Navigation between instruments.
- Definition of Instrument name.

2.4 Operational requirements

The framework shall use minimal resources on the target computer.

Normal operation is divided into 4 modes:

- Passive mode.
- Edit mode.
- Play mode.
- Recording mode (experimental).

2.4.1 Passive mode

Upon start-up, the program is idling and is so to speak doing nothing; what it actually shall do is listening for user input. Things like editing a composer's name, loading/saving songs and editing instrument definitions are to be carried out while in idle mode. While in this mode, actual song data cannot be edited.

2.4.2 Edit mode

While in editing mode, the program shall carefully listen to specific input from the user and alter actual song-data (inside the current pattern) on demand. When changes/insertions/selections occur, the pattern display must be updated instantly.

2.4.3 Play mode

Play mode may be turned on or off by interaction from the user. Entering play mode starts the actual player routine that is responsible for streaming MIDI messages based upon real-time interpretation of data within the song (within the patterns). The pattern display is updated frequently so that it is constantly synced with the notes being played. While in this mode, actual song data cannot be edited.

2.4.4 Recording mode

Basically a primitive combination of the edit and play modes. While playing the song/patterns, the user is allowed to edit the song data by entering notes through the keyboard (in real-time). This is an experimental mode and is not to be considered as a fully implemented real-time recorder.

2.4.5 Miscellaneous

A song's tempo may vary between 20 to 250 BPM (Beats Per Minute - usually number of quarter notes per minute); however, it is likely that the test application only will operate at a constant tempo of 125 BPM. (depends on how much time we have left to implement tempo change specific messages etc.) The timer system shall be accurate, maximum 10ms of latency is tolerated. It must also be able to handle 100 calls (worst case) per second. (100 Hz = 2*(250 BPM)/ 5 - more on this later.) Combination of notes within the same row must be played instantly. Throughput of actual MIDI data will in any case be limited by hardware restrictions, however, the application itself must be able to send a constant stream of non-latent MIDI data to the underlying software layer. The speed of the stream may be (if all 64 channels are used within a pattern, and the song is going at the maximum speed) 19200 bytes per second (100 Hz [250 BPM] x 3 bytes per MIDI event x 64 channels).

Since we've decided to use MIDI communication for playing notes, no memory will be required to store digitised sound. Instrument definitions will (usually) only eat a couple of hundred bytes; whereas patterns might consume maximum 100 000 bytes each, but then again, they are to be allocated dynamically ensuring the usage of only actual required amount of memory.

Lack of proper error handling (and error messages) may characterize some parts of the test application. Throughout the project, we shall focuse on making the framework work (the goal is to make it as fast and stable as possible); therefore, time used for debugging the test application might be somewhat reduced.

Just for the record: There are no security requirements within this project. The test application (and the framework) may be used by anyone, anywhere. We haven't implemented any authentication module, multi user support, parental lock, or any other kind of (annoying) registration requirement.

2.5 Life cycle

It is impossible to do a fully functional and errorless system during a one-semester graduation project. If the life cycle of this project is to exceed beyond our development process, we have to make sure that it is easily expandable. When our work is done, the source code is to be published on the Internet so that both BeOS and other developers (also users) may use it for experimentation and for implementation/integration of customized systems. We have tried to do our best making further development as easy as possible. The development of open source projects is characterized by a release under a special license that is

adjusted for the purpose of the software. It is common practice that everyone has the right to use, alter and re-publish the system. We haven't actually written any licence ourselves, but the idea is to release our work to the public, so that other people can take advantage of what we have accomplished during the last months.

We have already received a couple of questions regarding release date and details around the framework, most of the enquiries come from the Be development community. Our team believes that the project might just be a possible fundament for an upcoming tracker solution of the future. For the record: Balazs (our project leader) is planning to do further development during this summer, his main objectives are (among others) the implementation of more effects and functionality and the implementation of sample (digitised sound) support - this involves also the development of a portable software based digital sound mixer.

Chapter 3

Analysis

We have focused on finding the components that must be present for the realisation of a working solution. A summary shall be formed, reflecting the modules to be constructed (and also how they relate), creating a picture of the moments that are revealed within the requirement specification. We have chosen to use object-oriented analysis to be able to express the relations between the components of the system.

In the beginning of the Rational Unified Process a superior analysis of the entire task is made. The process solves problems by segmenting a complete problem into so-called iterations. Detailed analysis are conducted in the beginning of every iteration. The analysis and the design chapters within our document are somewhat melted together. Layout might therefore deviate from the reader's expectations. However, we have tried to express the substantial parts of our analysis. Using the RUP method has given us freedom that was essential for solving such kind of an assignment. The "traditional" waterfall method may have resulted in slowdown and instability during development.

During the early phases of development, we decided to split the assignment into five major parts. This division reflects how our team has analysed and understood the concepts of a possible music tracking system. We believe this is a general way of describing the most essential modules that must exist within any kind of tracker application. Each module is to be considered critical. Removal of one (or more) would result in a faulty solution that may not qualify or fulfil the expected needs. The modules described in this chapter must work together in harmony thus forming a possible structure of a tracker framework. Such module based analysis, design and development is pretty much preferred from the developers' point of view since it generally helps maintaining overview and at the same time simplifies implementation and testing.

3.1 Apportionment of liability

Long lasting discussions and analysis of similar products resulted in a way of dividing the design of a tracker framework. From our point of view, a music tracking system usually consists of these basic moments:

- Visualisation of music data (an interactive pattern editor of some kind).
- Precise timing and synchronisation routines.
- Realisation of audible sound with the help of digital sound samples, MIDI communication or computer generated tones.
- An interactive graphical user interface granting access to the program's functionality.

These moments are reflected as iterations in our RUP development model:

- Pattern Display Engine (PDE).
- Timing/Synchronizing routines.
- Sound output (in our case, MIDI communication).
- Player engine.
- User Interface.

The following use case (figure:3.1) illustrates the relation between the chosen RUP iterations.

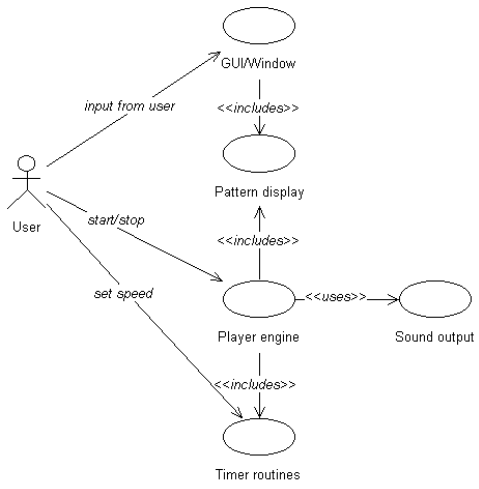

Figure 3.1: Use Case

3.2 Brief description of RUP iterations:

3.2.1 Pattern display

Takes care of visualisation of music data.

3.2.2 Timing and synchronisation

Takes care of timing and synchronisation of output.

3.2.3 Sound output

Takes care of outputting sound via a chosen source (in our case by using MIDI communication).

3.2.4 Player engine

Takes care of proper playback. (Translates song data to actual sound.)

3.2.5 User Interface

Takes care of user interaction and provides a graphical user interface.

Chapter 4

Design

4.1 Pattern Display Engine (PDE)

From the user's point of view, the pattern display is the heart of the application. It is via the pattern editor the musician inserts/edits note events - forming a composition. While in play/record mode, the display is precisely following the concurrent song-positions thus visualizing (in real-time) the notes and events that are processed by a player engine running in the backgorund.

From the developer's point of view, the PDE itself is considered to be a "dumb", non-interactive module that is only responsible for displaying song data at desired positions. It has no direct interaction with the user (even if it looks that way), but is mainly responsible for rendering and dumping humanly readable song data to a buffer and/or to on-screen coordinates; speed is a critical factor here, since a song could have a nasty tempo requiring the display to be refreshed quite frequently. However, the PDE must not flood the CPU so that that other, vital parts of the system (for example the player engine itself) are blocked. The PDE must be a dynamic implementation allowing (among other factors) frame skipping. Frame skipping is a phenomenon that occurs when the PDE is unable to keep up with the player engine or (tough unlikely, but possible) is exposed to extreme interaction (abuse) from the user. When the target system's CPU is overloaded (read: the system CPU is too slow) it is unable to process the rendering of the requested frames. Frames that can't be processed are to be automatically discarded so that the CPU can be used for other (more critical) parts of the system. The PDE may either use system specific methods or (preferably) custom rendering routines for visualising song data. By visualising we mean the translation of raw music data (stored in memory) and the proper representation of these within a pattern editor.

4.2 Timer and synchronizing routines

On any given platform, one would require some sort of timing routines capable of delivering a set of ticks (or pulses) that are to be used by the player and pattern display engines for proper playback and visualisation. It is the player engine's responsibility to scan through the notes of a pattern (song) at equidistant time intervals. The player engine has absolutely no idea of when to play the notes, therefore (to ensure proper synchronisation) a superior timer routine must be running in the background.

Unlike sequencers, which only use BPM (Beats Per Minute, quarter notes per minute) for tempo settings, trackers use two variables to represent tempo: SPEED and BPM. Developers have asked us to explain more about this subject because (according to them) people don't seem to get it at first. SPEED and BPM are certainly different, and we need a SPEED variable and a BPM variable to store them. Every music application that deserves to be called a tracker follows the traditional implementation and proper use of these variables.

SPEED is the number of ticks (or times the timer handler is called) between each time a new row (within a pattern) is to be processed. The default (traditional) SPEED setting within a tracker is always 6. This means the timer should tick 6 times for every row of the pattern processed. Now imagine if the speed was 3, it would take half of the time to process the rows, therefore making the song play twice as fast. The composer might not set a speed and just use that default speed. More explanation around this subject is to be found in the description of the player engine.

BPM (also called "tempo" within trackers), is the rate at which the timer (pulse generator) should be running. Computer systems use seconds, milliseconds or microseconds for time measurement. The following formula helps us to calculate the exact frequency (in Hz) of a timer at a given BPM definition, thus converting the "tempo" so that it can be used to for example set a computer's clock:

$$Frequency[Hz] = \frac{BPM * 4 * 6}{60} = \frac{BPM * 2}{5}$$

BPM is the desired tempo, 4 is because the patterns within the tracker system consist of quarter notes, 6 because the default SPEED of a song is 6, and finally dividing it by 60 to convert the value from beats per minute to ticks per second.

Say, if the tempo of a song is set to 125 BPM (the default BPM), the timer must tick at 50Hz, that is - it must be called/processed/woke up exactly 50 times per second with equal intervals (pauses in between the ticks). It is very important that the timer is running precisely; otherwise playback (and recording) may have strange results. Floating values/results of this formula are usually rounded up/down.

This timing theory (combination of speed and BPM) may seem a bit strange at first; however, it has proven to be very comfortable for musicians. By for

example changing the speed variable, the composer is able to instantly alter the playback resolution at any part of the song. Let's say there is no need for better resolution than 1/8th notes throughout a song, except a special drum roll at the end. The musician could calmly cycle through most of his song at a "low" resolution and when needed tweak the resolution (SPEED variable) to suit his needs. Once the set up of a proper timer routine is completed, developers may focus on actual sound output. It is also worth to mention that a pattern usually consists of 4 measures, a row (pattern entry) thus usually represents a 1/16th note.

4.3 Sound output

This part of the system is generally (roughly said) designed to act as an interface between the software and the hardware that is responsible for generating sound. How it actually works and what it does depends completely on the requirements and the operating environments of a system. In this particular project we choose to use MIDI communication for realising sound output, which meant that we would have to develop routines involving code for initialising and realising actual sound by communicating with MIDI compatible devices. Systems that rely on digitised sound samples or computer synthesized tones, would require the implementation of a software sound mixer, having the responsibility for mixing sounds/samples and piping them to hardware. In other words, the sound output part of a tracker-based solution is the module that is responsible for generating actual sound in some way or another. Discussion around implementation of different sound output methods is beyond this document's aim.

4.4 Player engine

From the developer's view: the "real" heart of the system. The player engine is responsible for proper playback of a song. It is traversing through the rows of a pattern translating every bit of information (events stored as raw music data) and realising these by interacting with the sound output module. The player engine is also strongly connected to the timer module, which acts as a background pulse generator.

From the timer's viewpoint, the player module is nothing more than a slave. It is the timer module that decides the frequency of actual playback; the player engine is only responsible for translating (and playing) music data at given intervals.

Now lets take a look at the player logic. The SPEED of a song is the base on how a song is played. Each row of a pattern is updated every SPEED number of clock ticks, so if a speed is 6, then we only update each row every 6 clock ticks. So on a speed like 3, the row is going to be updated every 3 ticks and will play twice as fast as speed 6. In-between we would update certain tick sensitive effects, like portamento, volume slides and vibrato.

Diagrammatically the playing of a song looks like this.

```
SPEED IS 6
tick#
0: UPDATE ROW #0  <- update/play the notes in a pattern's row.
1: ---  \
2: ---   \
3: ---    >- certain effects are updated here
4: ---   /
5: ---  /
0: UPDATE ROW #1 <- update/play the notes in a pattern's row
1: ---  \
2: ---   \
3: ---    >- certain effects are updated here
4: ---   /
5: ---  /
etc..
```

Studying this example reveals that only every sixth tick results in the update of the next row of note events within a pattern. However, the ticks in-between are not to be forgotten, since these are normally used for special effect purposes, enabling the musician to for example do smooth volume slides. A basic design of a tracker framework must therefore feature such an implementation allowing future upgrades and granting almost unlimited expandability of the effect system.

Logically a very basic representation of playing a song looks like this:

```
STATIC TICK = SPEED   //declaration, start it off at SPEED, not 0, as we
                      //want straight into the 'if tick >= speed condition'
TICK = TICK + 1       //now increment the tick counter
if TICK >= SPEED      //if the tick # is bigger or equal than SPEED then
  update_row          //update the CHANNEL number of notes for the new row
  tick =0             //reset tick to 0
  ROW = ROW + 1       //jump to the next row
else update_effect    //else we update the tick based effects
```

We must also to take into account that there are only Y (default 64) rows in a pattern, if we hit 64 then the player engine must jump to the next pattern and start at row 0 again. I say 64 because row 63's effects have to be played out before jumping to the next pattern.

4.5 User interface

A user interface module is essential for the actual utilization and demonstration of the framework's functionality - including the output of the previously described pattern display engine. The user interface is commonly responsible for presenting graphical gadgets like windows, buttons, menus, text fields and so on, as well as being responsible for the detection of user activity. Building a good user interface for a tracker would certainly take a lot of time. Ergonomic and configurable GUI design, integration and testing is considered to consume around 70 to 80 percent of the total time used to develop common applications.

Unfortunately, our team will have very little time for building the framework in focus, development of a surrounding user interface is considered to have very low priority. Secondly, developing a user interface on top of a functioning and stable framework is always easier than first developing a fancy interface and then trying to implement a fully functional framework. Anyone with some programming knowledge could stitch together a satisfying user interface, but not everyone is capable of building a good framework. As soon as we take away detailed and complex functions, the basis of tracker framework becomes simple. A set of common components show up in almost any type of tracker design. After analysing a handful of solutions, we have concluded that a framework's surrounding application must have the minimal set of components:

1. Pattern editor (an interactive implementation of the PDE's output).
2. Some sort of a graphical system including a set of buttons and/or menu items which grant access to basic functionality of the tracker's framework: administration of songs, patterns, instruments and play-lists.
3. Input event handler (keyboard and mouse).

The pattern editor is to be located in the lower half of the screen/window used for hosting the test application. Menus, buttons and other graphical gadgets are to use the upper half of the application's workspace. This is the common arrangement of tracker interface design, and we do not intend to change it.(figure 4.1)

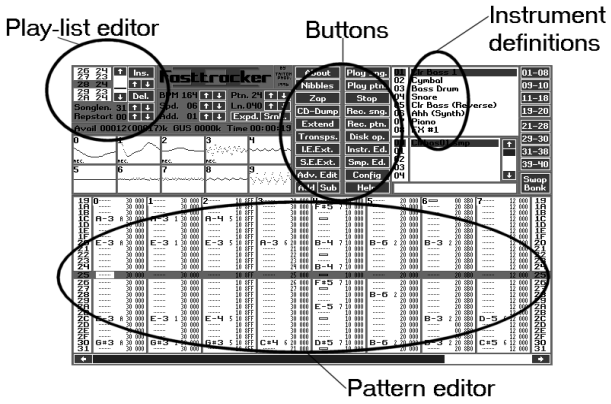

Figure 4.1: FastTracker2

Our team has decided to implement a basic tracker framework (and a surrounding application) by going through the previously described RUP phases one by one, starting with the most critical part of the system:

- Development of a Pattern Display Engine (PDE).
- Development of timer/synchronising routines.
- Establishment of MIDI communication (sound output).
- Development of a player engine.
- Development of a surrounding user interface.

Some people might think the player engine is the one that would be the most critical part of a tracker application. Fortunately, the theory behind (and actual implementation) of a basic player engine seems fairly easy compared to the theory behind a fast and customisable pattern display engine. Therefore, we have chosen to develop the display engine first, also because we had to have something to rely on when testing the rest of the system. Without the implementation of a proper pattern display engine, we would constantly sit blindfolded.

After reading this section, one must understand that all of the described modules have to be in place and function in harmony so that an actual tracker solution can be realised. A simple chain of sentences proves our theory: If we take away the PDE, the musician would not be able to see and edit song data. If we disable the timer module, synchronised playing and/or recording would be impossible. Removing the sound output module would result in fully functional, but silent system (not much usable, unless the musician is deaf). Turing off the player engine would (again) result in a static and muted system. If we remove the user interface, the system remains as a perfect framework, but becomes unusable for the musician. In the next chapter we'll strongly focus on how our team has managed to implement a working tracker framework including a surrounding user interface.

Chapter 5

Implementation

During the implementation phase, have we have tried to focus on how to construct the system to fulfil the requirements as good as possible. Along the entire process, time consuming discussions have emerged, slowing us down minute by minute. We hope these discussions have leaded us in a correct direction. There are several alternatives when it comes to assigning responsibility to different parts (objects) of the system. We have tried to maintain as low coupling as possible, designing and implementing the hierarchy in a way that it is structured, portable and easy to understand.

The entire implementation process is based on the previously determined iterations within the RUP model; Pattern display engine, timing routines, MIDI communication, player engine and user interface. However, as the audience might discover, during implementation, borders between iterations are sometimes slightly fainted. When you have no clue about the actual possibilities of the specified target platform, it is merely impossible to divide a problem into "correct" domains and solutions. Some (if not all) of our results are based on ever lasting experimentation within the different problem domains.

Until now, we have documented an all round solution (a rough step by step guide) for how to make a music tracker. We haven't mentioned any details or secrets behind an actual implementation. In this chapter however, we shall try to shed light on how our team of developers have been able to give birth to a basic music tracker framework for the BeOS platform, using C++ and the BeAPI.

We decided to use a development tool called the "BeIDE", which is more or less a comfortable front-end for the well-known GCC compiler. The use of the BeAPI has forced us to work in a special way, among other things splitting the GUI into windows and views etc (more about this later).

At the moment, the BeMiT project consists of the following list of files(figure5.1):

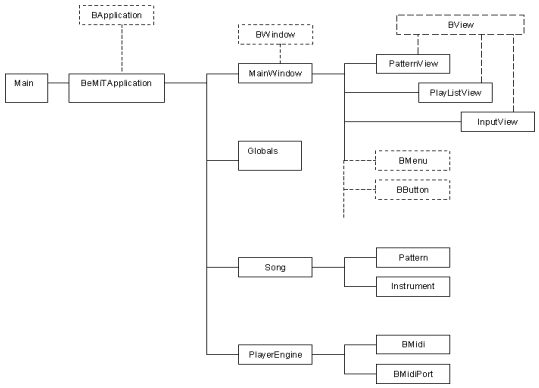

Figure 5.1: Project hierarchy (C++)

- Globals.h
  This files contains a set of global declarations, constants, etc.
- Main.cpp
  Only the main() function is located in this one, our program starts here.
- BeMiTApplication.h / BeMiTApplication.cpp
  A definition of "our" application, derived from the BApplication.
- Instrument.h / Instrument.cpp
  Responsible for representation and functions related to instruments.
- Pattern.h / Pattern.cpp
  Responsible for representation and functions related to patterns.
- Song.h / Song.cpp
  Responsible for representation and functions related to songs.
- MainWindow.h / MainWindow.cpp
  This is our main window; containing menus, buttons, and views.
- PatternView.h / PatternView.cpp
  Responsible for rendering pattern data.
- InputView.h / InputView.cpp
  Responsible for handling input messages (keyboard etc.).
- PlayerEngine.h / PlayerEngine.cpp
  Responsible for actual playback (and recording).
- PlayListView.h / PlayListView.cpp
  A simple interface displaying play-list organisation.

5.1 RUP iterations and C++ components

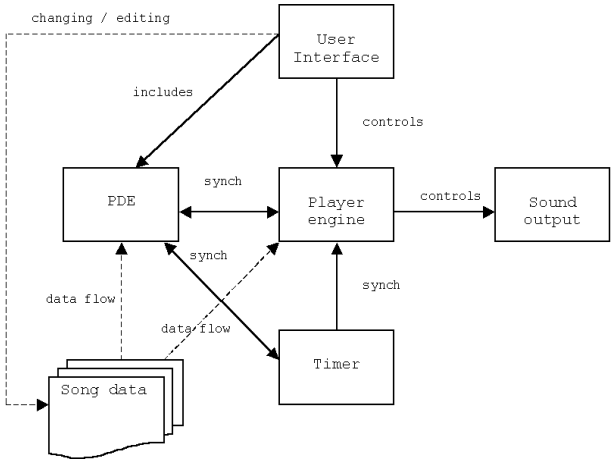

Figure 5.2: RUP block diagram

Our RUP model consisted of these iterations(figure 5.2):

- Pattern Display Engine.
- Timing / synchronising routines.
- Sound output (MIDI communication).
- Player engine.
- User interface.

The following text describes relations between the RUP iterations and our C++ model:

5.1.1 Pattern display engine

"PatternView" is the main object related to this iteration, it includes the "BeMiTApplication", which is responsible to control the songs. Actual pattern data is extracted from song objects, translated and rendered to a buffer that is to be displayed on screen.

5.1.2 Timing / Synchronisation

This iteration is implemented and integrated inside the "PlayerEngine" part of the system.

5.1.3 Sound output / MIDI

A BeAPI specific MIDI object holds the necessary functions related to MIDI communication. This iteration is implemented and integrated inside the "PlayerEngine".

5.1.4 Player Engine

The "PlayerEngine" part of the C++ implementation holds functionality related to the Player Engine iteration. It includes BeMiTApplication, which is responsible to control the songs. Actual pattern data is extracted from song objects, translated, and at equidistant time intervals (with the help of the integration of the Timing iteration) streamed to a designated MIDI object.

5.1.5 User interface

This iteration reflects the integration of all the graphical user interface and input components: "MainWindow", "PatternView", "PlayListView" and "InputView".(figure 5.3).

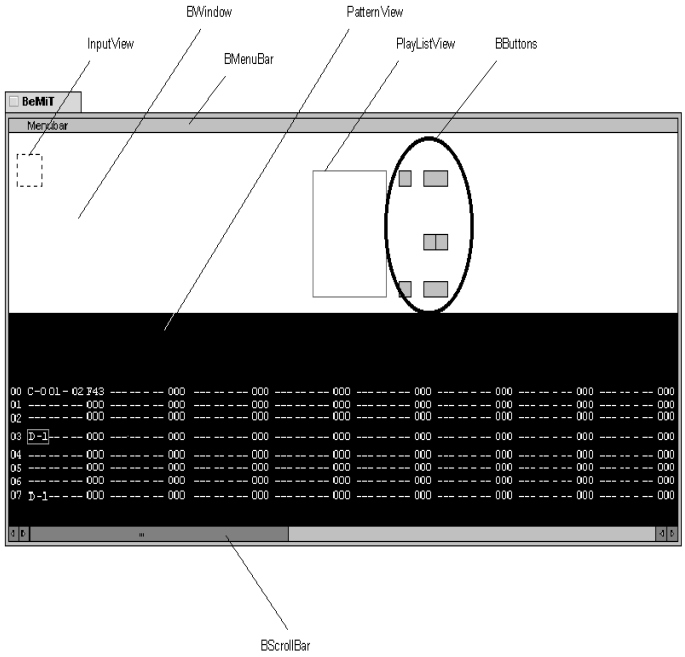

Figure 5.3: Window

Before we go for the implementation of the RUP iterations including a deeper description of the C++ model, we must construct some basic classes that allow

structuring and representation of musical data within the framework.

5.2 Representation of data

Since we've decided to do this project inside an object-oriented environment, it would be likely for us to use the power of object oriented development for the representation and arrangement of both data and functionality. As earlier told, a song consists of different patterns, which all have a set of tracks (channels), and which again have a set of equal number of entries. Within the requirement specification, we have decided that entries should consist of five parameters: note event, instrument number, panning information, velocity definition and an effect definition. A close examination of the MIDI protocol and earlier tracker solutions has played a very important role when dimensioning the actual size for the mentioned data carriers.

For compatibility reasons, note events within a pattern entry must be able to represent every single MIDI note event. We have decided to assign actual MIDI note values to the correct/corresponding byte values within our note event implementation chart. This frees us from having to convert our way of representation to match the MIDI standard during playback/recording etc. Since we managed to fit the MIDI standard along with the definition of special cases (such as the note off event), all (possible) note events can be stored within the range of 127 different combinations. However, because of possible expandability we have decided to use a byte (8 bits, 256 combinations) to represent note events, reserving values 128 to 255 for future use. Please refer to[Appendix K].

In the past, songs were based on having 16, 32, 64 or 128 different instrument definitions. The BeMiT framework shall not be humiliated because of limitations, therefore we have decided to use 8 bits (one byte) for the representation of the instrument number. This gives us the ability to define and address 255 different instruments from within a song. I say 255, because number "zero" is always used for special purposes (more about this later). Now, we haven't actually defined the usage of all 255, we have only implemented 128 and reserved the rest for future upgrades. Please refer to[Appendix L].

It is unfortunately impossible to pan-port single MIDI notes because of the limitations within the MIDI protocol. However, panning of computer synthesized sounds and/or digital sound samples (with the use of a digital software mixer) must be possible within a future upgrade. This parameter is meant to be used for pan-porting single notes/samples/voices while the song is playing, giving the composer the ability to place whatever he wants wherever he wants within the sound picture. Since the panning definition takes up one byte, we could use the first 128 values for defining a chain of position definitions from left to right (this would give the musician a very good resolution); hence the rest of the remaining 128 values could be used for special effects like pan-slides, fading etc. We have set up an example showing how the panning definition could be used, but this is just a suggestion, leaving a vast number of possibilities within a future upgrade. Actually, such a definition could be used for anything. However,

it is meant for panning purposes. Please refer to[Appendix M].

In this basic implementation, the velocity entry's role is to represent the actual velocity of a note event (if any). The MIDI standard uses 7 bits (0-127) for representing such values. For compatibility reasons, we have decided to use one byte to represent the velocity entry, since future upgrades (digital sound sample support etc.) must be able to offer so called "volume column effects" known from the famous ScreamTracker and FastTracker II standards. Therefore, the rest of the 127 bit positions are strictly reversed for such purposes. Please refer to[Appendix N].

Representing the effect entry takes up two bytes, and has almost always done so (study of earlier file formats reveal this). The first byte is used to tell what kind of effect is to be initiated; the second is used for defining a value for that particular effect. Say, if the first byte was used to tell that we wanted to do a volume slide, the second byte is likely to be used to define the speed of the actual slide. Or, another example: say if we wanted to do a position jump within a pattern, the first byte within the effect definition would tell the system that it must do a position jump, while the second would carry information about where to jump. Please refer to[Appendix O and P].

By now, anybody may guess that it will take us six bytes to represent a single entry within a track:

- Note event (1 byte)
- Instrument number (1 byte)
- Panning information (1 byte)
- Velocity definition (1 byte)
- Effect command (1 byte)
- Effect parameter (1 byte)

It may seem like we are using up too much memory for representing entries, but this is not the case. Future upgrades would always demand a bunch of reserved bit positions, something that is extremely granted by our way of entry representation. Note events and instrument numbers could have been merged into a single byte using values 0 to 127 for note numbers and 127 to 255 for velocity definitions. However, under runtime, the program would have to constantly pack and unpack data (unnecessary CPU usage). Besides, it would be difficult to implement new definitions etc. *Conclusion:* To maintain the possibility of future upgrades and to spare the player engine for unnecessary data conversion routines, we have decided to use 6 bytes per entry.

As earlier told, a pattern consists of channels, which again consist of equal number of entries. There are several ways to organise this, however, it is clear that patterns within a song must be represented as independent objects. Channels could be represented as objects too, also entries. This would unfortunately

result in an exaggerated object-oriented structure slowing down both development and execution of code. We have stated that our team wants to implement a solution using object-oriented technology, however, we must be aware of the pitfalls.

Representing entries as objects might sound like a good idea at first. Unfortunately it isn't. Addressing an object would require at least 4 bytes because of the flat 32bit architecture of protected mode on most of today's platforms. Now, if a pattern consisted of 64 rows (default) and 32 channels, a pattern object would require 64 rows x 32 channels x 4 bytes per pointer = 8192 bytes just to be able to address it's own entry-objects, which again consist of (at least) 6 bytes each. Total memory usage would then result in something like: 8192 bytes x 6 bytes per note = 49152 bytes ~ 50Kb (!) for representing a single pattern of normal size (64 rows x 32 channels). This is unfortunately unacceptable.

Patterns within a song may have different dimensions; one could have 20 rows and consist of 10 channels, another having 64 rows and consisting of 64 channels and so on. Since patterns may hold maximum 256 number of rows, we would have to allocate maximum-number-of-channels x 256 different event pointers. Dynamic allocation techniques lead to fragmented blocks of data and require more time for processing (when inserting/deleting elements). The "original" C++ language (without a Standard Template Library package) doesn't seem to support dynamic allocation of data/elements/pointers. Without dynamic allocation technology, we would end up with 64 channels x 256 rows x 4-bytes per pointer = 65 536 bytes used to only represent our event pointers regardless of the pattern's dimensions. If we want our structure to be like this, then every pattern object must be able to address the maximum allowed dimension of events even if the pattern isn't using the maximum dimension; Keep in mind that we would like to be able to resize patterns on the fly and to have "poly-sized" patterns within a song.

Obviously, too much object-orientation would lead to both unnecessary memory usage and complex programming; hence we must unfortunately ditch the idea of doing a 100% object-oriented implementation when it comes to the representation of music data. As earlier mentioned, our development team has done a lot of research regarding this subject. We have concluded that data could (more or less) be represented using four different methods (including the method described above). These methods will be briefly mentioned here:

5.2.1 Method #1

Song data is represented as three objects (full object orientation): patterns, channels and entries (as discussed above).

```
class entry {
 uint8 note;
 uint8 instrument;
 uint8 panning;
 uint8 effect command;
 uint8 effect parameter;
```

```
}
class channel {
 entry* notes[MAX_NUMBER_OF_ROWS];
}
class pattern {
  channel* channels[MAX_NUMBER_OF_CHANNELS];
}
```

Pros and cons of Method #1:

+Structured code
+Easy to understand
+Easy to expand (add more parameters within an entry)
-Overcomplicated object orientation
-Difficult to write fast routines that can handle this type of representation
-Requires much memory
-Fragmented data

5.2.2 Method #2

Song data is represented as two objects: patterns and channels. Patterns would have pointer arrays to channel objects, which would store channel data as blocks (black box method).

```
class channel {
 uint8* data;  // pointing to block of data (NUMBER_OF_BYTES_PER_ROW * rows)
}
class pattern {
 channel* channels[MAX_NUMBER_OF_CHANNELS];
}
```

Pros and cons of Method #2:

+Somewhat easy to understand and structured code
+Easy to add/remove channels
-Requires still a bit more memory (than actually needed)
-Fragmented data
-Difficult to implement player/display routines that support this geometry

5.2.3 Method #3

Song data is represented as two objects: patterns and rows. Patterns would have pointer arrays to row objects, which would store row data as blocks (black box method, basicly Method #2 upside down).

```
class channel {
 uint8* data;  // pointing to block of data (BYTES_PER_ENTRY * channels)
}
class pattern {
 channel* row[MAX_NUMBER_OF_CHANNELS];
}
```

Pros and cons of Method #3:

+Easy to add and remove rows

+Somewhat easy to understand and structured code

-Requires still a bit more memory (than actually needed)

-Fragmented data

-Difficult to implement player/display routines that support this geometry

5.2.4 Method #4

Represent pattern data inside independent pattern objects, which contain all their respective data in one block (full black box method).

```
class pattern {
 uint8* data; // pointing to block of data(BYTES_PER_ENTRY * channels * rows)
}
```

Pros and cons of Method #4:

+Easy to set and get raw data, player/display methods only need to scan trough the one

block.

+Fast access to data

+Minimal fragmentation of data

-Less upgradeable

-Not so easy to understand

In addition to these four methods, we have also considered to make a chain of linked entries that are connected to patterns (etc). Unfortunately this idea was also scratched when we discovered how uncomfortable such an implementation might get. To make our implementation fast and easily traversable, the team decided to use method number four for representing song data within the framework.

Now, once the representation of song-data is successfully determined, it is time to reveal the secrets related to actual implementation of the BeMiT project. We shall only maintain focus on critical parts of the system; it is almost impossible to discuss every little detail in this report. Before essential modules are discussed, we feel that it is necessary to shed light on the basic classes we've used (within the BeAPI) to build our application. Developers familiar with the BeAPI might skip this part.

5.3 BeAPI

The application programming interface (the BeAPI) is designed to make the features of the BeOS easy to use. Written in C++, it includes numerous class definitions from which developers can take much of the framework for their application. By using the BeAPI, developers can use standard (ready-to-use and customisable) components when building applications for the BeOS.

5.3.1 BApplication

The BApplication class defines an object that represents our application, creates a connection to the App Server, and runs our application's main message loop. An application can only create one BApplication object; the system automatically sets the global "be_app" object to point to the BApplication object we create. BApplication is the main object that launches all the other objects that are related to the application. The primary purpose of the BApplication object is that it makes and maintains a connection with the "Application Server". It is the "Application Server" that takes care of the low-level work such as handling interactions between windows and monitoring input from data entry sources such as the keyboard and the mouse.

5.3.2 BWindow

BWindow is the heart of the graphical user interface of the Be operating system. A BWindow object represents a window that can be displayed on the screen, and that can be the target of user events. You almost always create own BWindow subclass(es) rather than using direct instances of the API's BWindow. BWindow objects draw windows by talking to the App Server. If we would want to take over the entire screen or draw directly into the graphics card's frame buffer (by-passing the App Server), it is possible to use BDirectWindow or BWindowScreen objects; more on this later, when we focus on the implementation of the pattern display engine. The function "MessageReceived(BMessage *message)" is a hook function that fetches all the messages targeted to the desired window.

5.3.3 BView

A BView object represents a rectangular area within a window. The object draws within this rectangle and responds to user events (mouse clicks, key presses) that are directed at the window. Most of the classes in the "Interface Kit" within the BeAPI inherit from BView, buttons, text-fields, menus, etc. We shall use some these classes off-the-shelf, but for some parts of the BeMiT implementation, we'll also need to create BView subclasses. The "PatternView", "InputView" and "PlayListView" are such subclasses. A window usually contains a set of "BView" objects. All the views are redrawn when the window is updated. A selected view (in focus) has the ability of receiving messages. Only one of the views can be in focus. Such an offering gives us the opportunity to for example switch between views to change the way keyboard messages are handled etc. The main goal is to be able to work with independent tasks within the same window.

5.3.4 BMidi

BMidi is an abstract class that defines protocols and default implementations for functions that most of the other Midi Kit classes within the BeAPI inherit.

The functions that BMidi defines fall into four categories:

- Connection functions. The connection functions let you connect the output of one BMidi object to the input of another BMidi object.
- Performance functions. BMidi objects that generate MIDI data (by reading from a port or file, as examples) implement the Run() hook function. Run() is the brains of a MIDI performance; Start() and Stop() control the performance.
- "Spray" functions. The "spray" functions send MIDI messages to each of the objects that are connected to the output of the sprayer. There's a spray function for each type of MIDI message; for example, SprayNoteOn() corresponds to MIDI's Note On message.
- "MIDI hook" functions. The "MIDI hook" functions define the object's response to a particular type of MIDI message. There's a hook function for each MIDI message (NoteOn(), NoteOff(), etc.). The hook functions are invoked automatically as "upstream" objects call the corresponding spray functions.

5.3.5 BMidiPort

A BMidiPort object reads and writes MIDI data through a MIDI hardware port. A MIDI hardware port has an input side (MIDI-In) and an output side (MIDI-Out); we can use a single BMidiPort object to communicate with both sides. Also, we can create and use any number of BMidiPort objects in our application, multiple BMidiPort objects can open and use the same hardware port at the same time.

5.4 Components of the BeMiT project

5.4.1 The main() routine

Our C++ implementation begins execution with the help of a routine called main(). In the world of the BeOS, this routine must declare a pointer to, and create an instance of a "BApplication". In our case we have to declare and initialise a new instance of the "BeMiTApplication", our (special) application.

5.4.2 BeMiTApplication

BeMiTApplication was the first class we had to create by inheriting the BeAPI's BApplication class. This class is responsible for the construction and control of the main window, the player engine and the set of songs that are to be allocated in memory. It is also responsible for initiating load and save sessions related to songs. Variables and instances are initialised within the constructor. Songs

are contained, accessed and arranged within an array of song-pointers. The formerly mentioned global pointer "be_app" can from now on (after creation of the BeMiTApplication) be used to access the "BeMiTApplication" from different parts of our application.

5.4.3 MainWindow

"MainWindow" is derived from BWindow and includes "PatternView", "InputView", "PlayListView" and other objects related to the graphical user interface (BScrollBars, BButtons, BMenus, BFilePanels, etc.). Objects are added into the window's "workspace". "MainWindow" is also responsible for receiving and interpreting messages related to interaction (when for example buttons are pressed) and is controlling its underlying objects. As earlier mentioned, we're not focusing on doing a fancy GUI since this project is about constructing a framework. The GUI (except the pattern editor) is implemented for one purpose only: for being able to "reach" and demonstrate the framework's functionality. The framework (as it is today) offers quite a bit of functionality, unfortunately not everything is reachable from within the GUI since our team didn't have any time left.

5.4.4 PatternView

This is one of the modules that took most time to develop and also the module we are most proud of. We've been able to construct (from scratch) a fully functional, fast, scaleable and portable 32bit pattern viewer that is based on custom fonts.

The PatternView can be considered as partially being the heart of our tracker's GUI. It is mainly responsible for (when asked) interpreting song data and (as fast as possible) rendering understandable data for on-screen display. The "PatternView" displays a portion of a pattern at the desired position. Since patterns can be larger than the screen is physically able to display, the PatternView has to calculate what and how to do the actual rendering. The layout itself is pretty much standardised: tracks (channels) side by side (horizontally) divided into rows (vertically).

The entire display is divided into three major parts: an upper part, middle part and a lower part. These parts illustrate the pattern throughout time. The middle part of the display shows the current (desired) position, while the upper part displays rows that are located (in time) before this one. The lower part displays rows that are located (in time) after the middle part. Just a quick note on this, to gain understanding: the player engine will play entries that are located on the same row simultaneously.

Implementing a pattern viewer can (more or less) be done in two ways: Either by using operating system specific functions to build graphics (text), or by building custom graphics and using operating system specific functions to display these. The easiest way would be to use some sort of "DrawString",

"DrawText" or similar routines for rendering the rows of a pattern. Unfortunately, such routines within the BeAPI don't seem to sync to the vertical retrace of the screen. Multiple updates/refresh calls result in annoying flickering. The second disadvantage with built in string display routines is that they are only able to display a limited set of characters within limited space.

After experimenting a bit with the BeAPI's DrawString routine we decided to construct our own (custom) pattern render engine. This resulted in approximately a month of research where we experimented with the implementation of a BDirectWindow and the use of a custom BView's DrawBitmap routine. Both implementations were based on using custom designed fonts / image maps. Since most modern GUIs run in 32bit colour workspace, we decided to do the implementation in 32bits per pixel resolution. The BDirectWindow lets us address the entire workspace (not just the window) in a very fast way by giving the application direct access to the framebuffer by bypassing the App Server. However, programming the BDirectWindow is *not* easy! We found it rather difficult to implement; and to make it work together with the rest of the GUI. After a lot of experimentation, we decided not to use the direct access to the framebuffer. Tough, we learned a lot about low-level graphics programming while experimenting.

Finally, we went back to an earlier solution, an implementation of a custom BView, which is responsible for the rendering and screen dumping of pattern data by using customized 32bit fonts designed by one of our developers. The fonts were drawn using Deluxe Paint and converted/exported using Paint Shop Pro to raw bitmap formats. One bitmap for each parameter of a pattern entry was made: Note events, instrument numbers, panning definitions, velocity values, effect commands and effect parameters. In addition, we also use a bitmap that represents position numbers. All the bitmaps contain a list of image blocks that are vertically separated by a single scanline. These bitmaps are loaded into memory when the PatternView is created. Upon orders, the PatternView traverses through given pattern data and uses memcpy(...) to copy and paste correct image scanlines at correct coordinates to a previously defined 800x350x32bit off-screen buffer. To be able to keep track of pattern data, image buffers and so on, we use several pointer operations. For example: a pointer to actual pattern data is sequentially incremented as we traverse through the data, making interpretation a fast and easy operation. The pinpointing of correct image blocks (source) and the destination coordinates are also done with the help of pointers. The image buffer is finally dumped to the BView's surface by calling DrawBitmap(). Last, but not least: For navigation we draw a small, outlined rectangle that is able to jump between different parameters within the pattern's entries.

The main code inside PatternView is strictly based on variables (rather than static declarations) ensuring a very powerful and customisable rendering engine. By editing the variables declared at the beginning of the PatternView, developers can change every little detail around how the display engine should act when rendering patterns. Developers (in later versions, users also) will have the opportunity of replacing the custom fonts, changing the dimension etc; it would

only be necessary to feed PatternView with correct data that describes the image maps etc. It is also worth to mention that the PatternView automatically displays the correct portion of a given pattern, everything is calculated, it only needs to be manually adjusted to the environment that it is to be implemented in.

When displaying patterns larger than the screen can handle, we simply skip (by incrementing the pattern data pointer) past the "uninteresting" parts of a pattern buffer. A BScrollBar within the "MainWindow" (targeted to the "PatternView") is used for navigating horizontally (channels/tracks) inside large patterns. An override of the call-back function "ScrollTo()" makes instant the detection of desired scrolling possible. Since image operations are done within the same memory space (in this case, main memory) they are processed very fast. The only thing that slows down our implementation is the final call to DrawBitmap, which has to wait for the vertical retrace (of the target monitor/screen adapter) and also process copying of data from one memory pool to another (that is from main memory to framebuffer memory).(figure5.4).

5.4.5 InputView

The "InputView" is derived from the "BView" class; tough it is not displaying any graphics. This class is only responsible for handling keyboard events. This is the view that is constantly in focus within our application's main window. Every keyboard message is carefully interpreted. Based on the intercepted messages (which keys were hit, in what combination and so on) the "InputView" acts in a corresponding way by changing variables, calling functions etc. It is here where most of the actual user interactions take place. The "InputView" works as an interface between the user and the application's framework.

5.4.6 PlayListView

The "PlayListView" is also derived from the "BView" class. It is responsible for displaying a song's play-list on screen using BeAPI specific standard components. This is a very primitive interface, coded in hurry for the only purpose of being able to test and demonstrate play-list related functionality: navigation, organization etc. It simply displays the current play-list position (within the currently selected song's play-list) and its surrounding positions by dumping parts of a song's play-list array into the view. The selected song-position is emphasized using a distinctive red colour. Our primitive display is limited to show only five items (song-positions), however, it is possible to reach every song-position by using the play-list related buttons. These are located to the left of the "PlayListView" (implemented inside "MainWindow") and are used for navigation and organization of the current song's play-list. We can navigate through a song's play-list (the patterns that make up the song) and it is also possible to change/edit the order. The refresh function is responsible for rendering the display; it can be called (by using a correctly casted "be_app" pointer along

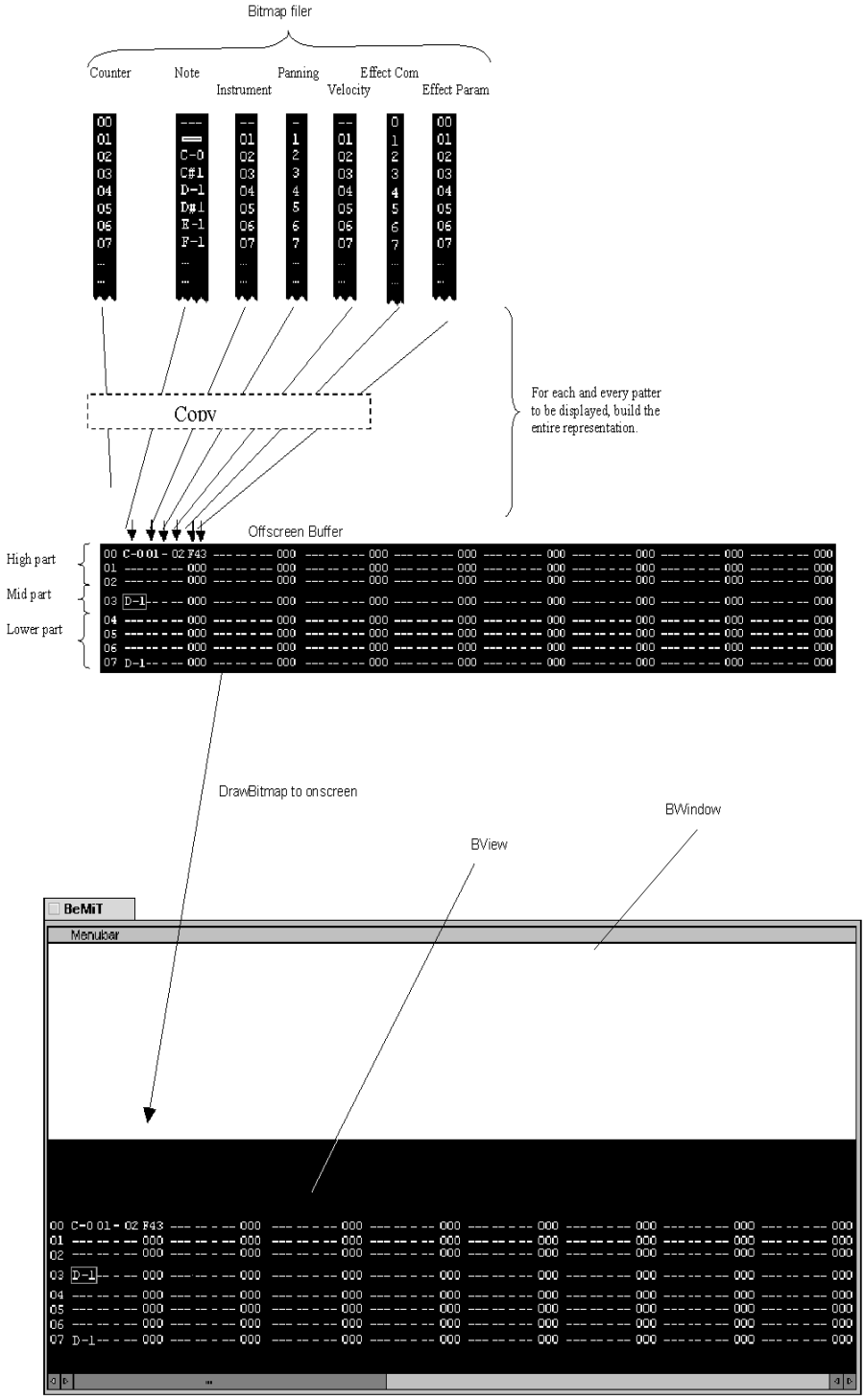

Figure 5.4: Building an pattern

with a correct pointer chain that points to the refresh function) from anywhere within the application.

5.4.7 Song

As previously mentioned, a song consists of patterns, instruments and a play-list. A song object must know about its currently "selected" pattern and instrument. This is handled by the variables CURRENT_PATTERN and CURRENT_INSTRUMENT, which are used to index the array of pattern pointers and instrument pointers within a song. In addition it must also know the exact position within a pattern (the location of the cursor). A song's play-list is represented as a simple array of bytes. Stored inside this array is a sequence of pattern numbers, this sequence makes up a complete song. The song is totally responsible for the arrangement (creation, deletion, navigation, etc.) of its own play-list, patterns and instruments. Upon creation, a song object declares and allocates it's first pattern and instrument, which cannot be deleted. Not surprisingly, a song's title, composer, speed and BPM parameters are also stored inside the respective song object. Saving and loading functionality is also implemented, the song is responsible for saving/loading its own parameters only (title, composer, play-list, etc.), patterns and instruments are responsible for saving/loading them selves. This is simply done to preserve independency. (figure 5.5).

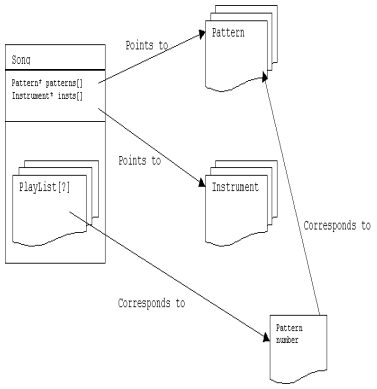

Figure 5.5: Song

5.4.8 Instrument

The "Instrument" class is mainly responsible for storing information about the parameters that belong to instruments within a song. Each instrument is an independent instance (object) of this class and can only belong to a single song. In addition, it offers a set of basic functions for loading and saving instruments.

Future upgrades might include cut/copy/paste functions, a wider range of parameters etc.

At this point, the BeMiT framework contains only necessary parameters and most basic functionality. All parameters are to be user-editable. Name is the name of the instrument, description is meant for user-defined description purposes, volume holds the instruments (global) volume setting (we follow the MIDI standard, 0-127), tune and fine-tune is to be used for tuning/fine tuning (transpose) purposes, MIDI channel tells the player engine which MIDI channel to use when playing this instrument. MIDI preset (when changed) tells the framework to switch to this preset within the instrument's MIDI channel. (When changed, a MIDI program change message is to be dispatched from the application.) MIDI bank is to be used for bank switching on instruments that offer more than 127 patches to choose from (the bank parameter would then be used to switch between various instrument pools within a physical instrument, for example a synthesizer). Many of these parameters are only implemented but never used within the current version of the BeMiT framework, however, their presence indicate how easy it is to implement such parameters.

Every single instrument-object is responsible for saving/loading it's own data with the help of a file pointer that is sent to the load/save functions. (figure5.6).

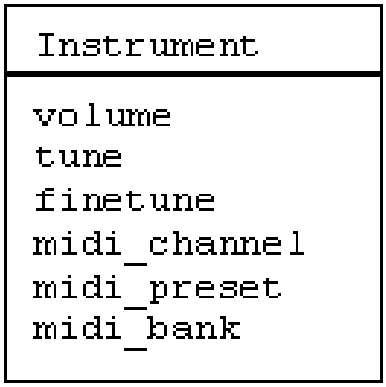

Figure 5.6: Instrument

5.4.9 Pattern

The "Pattern" class is responsible for holding all information that belongs to a pattern, its name, description, number of channels, number of rows and the music data itself. Patterns (within a song) are totally independent of each other; a single pattern can only belong to one song. A block of allocated memory is used for the representation of raw music data (multiple representation methods were discussed earlier in this chapter). We simply use memalloc(...) to allocate the exact number of bytes that are needed for a pattern, which is the product of the following factors: "NUMBER_OF_BYTES_PER_ENTRY",

"NUMBER_OF_CHANNELS" and "NUMBER_OF_ROWS". If we would like to allocate data for a pattern that consists of 32 channels, 64 rows and that carries 6 bytes per entry the correct size of the required buffer would be: 6 bytes per entry * 32 channels * 64 rows = 12 288 bytes.

The constructor(s) take care of allocating a correctly sized buffer based upon either the default settings (defined in "Globals.h") or parameters that are sent to the constructor(s). It is worth to mention that we've successfully developed functions for resizing patterns, however there was unfortunately no time left to implement this functionality in the framework's surrounding application. Patterns can be resized on-the-fly by calling the functions "set_rows(...)" and "set_channels(...)". When resizing, the current block of pattern-data is automatically copied into a new buffer (that is to hold the data, with correct size); the old pattern buffer is de-allocated. When a pattern is enlarged, all of its current information is preserved, when it is shrunk, data loss may occur (this is perfectly normal) since a smaller buffer is unable to hold as much information as the original one. Every pattern-object is responsible for saving/loading it's own data with the help of a file pointer that is sent to the load/save functions. (figure5.7)

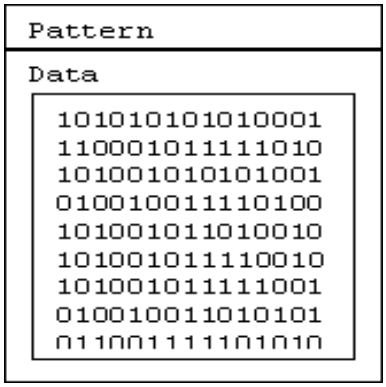

Figure 5.7: Pattern

5.4.10 PlayerEngine

The "PlayerEngine" is (as earlier told) the "real" heart of the framework. This is where information becomes music. Actually, the code itself is fairly short, and may seem simple and "obvious". For your information: it took a great amount of time to do the "behind the scenes" research that finally led to a useful solution. Gathering and understanding this kind of information doesn't seem to be easy. However, we managed to design and implement what we mean is a "correct" player routine.

The "PlayerEngine"s main responsibility is to read, translate and play a pattern's rows at equidistant intervals. For convenience, we decided that the

"PlayerEngine" is to be responsible for synchronising playback of notes in accordance with the "PatternDisplay". This means that it is not only responsible for playing music, but also responsible for refreshing the pattern editor so that it constantly follows and visualises the rows being processed (played). We have to keep in mind that the "PatternView" must not interfere with playback. Notes must be played precisely at exact time intervals; playback cannot be slowed down (or jittered) because of an unstable, slow, or CPU demanding display routine.

The class includes the BeAPI's MIDI components for MIDI communication purposes, in this case for streaming MIDI data to an external MIDI compatible device. The constructor sets up a BMidiPort that acts as a software interface between our application and a working MIDI device driver within the operating system. The start and stop functions take care of starting and stopping actual playback. Since the BeAPI offers quite accurate timing with the implementation of threads, and since developers within the community stated that "threads is the way to go" - we decided to implement timer/synchronising routines based on threads. To satisfy the tracker standard (to be able to have "ticks" between rows) we would need at least two threads for playback.

The first thread needs to pulsate accurately at a high rate (ticks within a song, for maintaining high resolution), while a second thread (in some way synchronised to the first one) would act as a slave and be responsible for actual playback. The first thread is only serving as a time source, with the one purpose of pulsating at a given rate; for accuracy, nothing is to be done within its main loop. Several discussions emerged about how we should synchronise these threads; we also consulted our supervisor Dr.Ivar Farup and Mr.Øyvind Kolloen for help. There seemed to be several possible solutions:

1. Synchronising of threads with the help of signals.
2. Synchronising of threads with the help of variables (local/global).
3. Synchronising of threads with the help of semaphores.
4. Synchronising of threads with the help of the BLocker component.

Together we looked at several methods and solutions within Java, Linux etc. The BLocker could unfortunately not be used (although its a wonderful tool) because it is apparently blocking a thread while waiting. Blocking of threads result in severe and unnecessary CPU usage that could lead to imprecise synchronisation. Variables (both local and global) seemed slow, in addition these could contain false values upon retrieval, leading to dangerous deadlocks. Signal communication between threads seemed quite easy to begin with, but we quickly discovered that a complex implementation would have to be made, since signals may contain (and transfer) various data, tokens, etc.

None of the alternatives mentioned so far seemed to fit our needs. The goal was to simply synchronise two threads. After even more experimentation, we found that semaphores might just do the trick. They seemed quite fast

compared to the other alternatives. The advantage of the semaphore system within the BeOS is that if a thread can't acquire a semaphore (because the semaphore is yet to be released by the previous acquirer), the thread blocks in the "acquire_sem(..)" call. While it is blocked, the thread doesn't waste any cycles, allowing the CPU to work with other things while the thread is a state of waiting. Suddenly we were able to synchronise the threads in a fashionable way by constantly releasing a semaphore within the loop of the first thread, and by waiting for the release in the second thread. Upon release, the second thread (in this case, our play thread) would instantly pick up the semaphore released by the first thread, and apparently synchronised it to the first thread. The following pseudo code shows a primitive implementation of this technique:

```
Tread1() {
  while(true) {
    snooze(time);
    release(semaphore);  // continous release of the semaphore
  }
}
Tread2() {
  while(true) {
    catch(semaphore);   // wait here (but don't block!)
                        // until the semaphore is released
    Do_something();
    Do_something();
  }
}
```

The first thread (called timer thread) is configured to run with real-time priority, while the second thread (play thread) uses urgent priority within the Be environment.

Please read the design chapter to understand the theory behind ticks, rows and timing within a tracker. Once the semaphore is picked up by the play thread (the second thread) it increases its tick counter. When the tick counter reaches the song's SPEED variable, a new row is processed within the current pattern; this results in dumping actual MIDI data (based upon the data stored inside the pattern buffer) to the external device. Notes (and messages) on the same row are processed instantly and (almost) simultaneously by a fast inner loop within the play thread. The song's tempo (BPM) determines the frequency of semaphore releasing within the first thread.

In addition to the previously described threads, we have also set up a display thread. This is configured to run with "display priority" - a special priority designed for graphical operations within the Be environment. The third thread is responsible for calling the "PatternView"'s refresh routine. Figure 5.8 shows an overview of our (three) threads. BeMiT's play thread simply releases a (new) semaphore every time it is ready to process a new row. The display thread is constantly lurking in the background, waiting for the right moment to update the "PatternView"; thus making the synchronisation of graphics and sound become a reality.

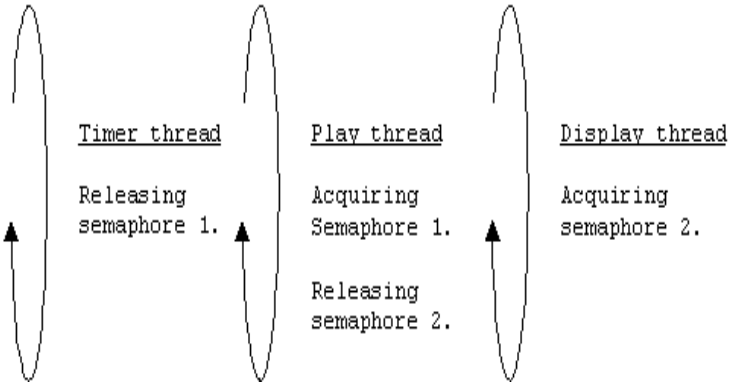

Figure 5.8: Threads

5.5 Storage

Every object that is carrying song related data (songs, patterns and instruments) is responsible for loading and/or saving it's own attributes. When loading/saving is initiated, a song is only responsible for it's own parameters. The song object throws responsibility to its pattern and instrument objects (calls their load/save functions) so that they can load/save themselves. The BeMiT application is responsible for all songs, and thereby holds the main file functions. This kind of responsibility division helps us to maintain expandability within the framework.

Song objects are responsible for saving/loading these parameters:

1. Length of the song's title (1 byte)
2. Title (null terminated string, max 255 bytes)
3. Length of the composer's name (1 byte)
4. Composer (null terminated string, max 255 bytes)
5. Length of description (2 bytes)
6. Description (null terminated text, max 65535 bytes)
7. BPM (1 byte)
8. Speed (1 byte)
9. Number of instruments (1 byte)
10. For each instrument allocated/defined within the song, call the specific instrument's save/load function. (Instruments are responsible for loading/saving themselves).

11. Length of the play-list (1 byte)
12. The play-list itself (array of bytes)
13. Number of patterns within the song (1 byte)
14. For each pattern allocated/defined within the song, call the specific instrument's save/load function. (Patterns are responsible for loading/saving themselves).

Instrument objects are responsible for saving/loading the following:

1. Length of the instrument's name (1 byte)
2. Name (null terminated string, max 255 bytes)
3. Length of the instrument's description (2 bytes)
4. Description (null terminated text, max 65535 bytes)
5. Volume value (1 byte)
6. Tune (1 byte)
7. Finetune (1 byte)
8. Midi channel (1 byte)
9. Midi preset (1 byte)
10. Midi bank (1 byte)

Pattern objects are responsible for saving/loading the following:

1. Length of the pattern's name (1 byte)
2. Name (null terminated string, max 255 bytes)
3. Length of the pattern's description (2 bytes)
4. Description (null terminated text, max 65535 bytes)
5. Number of channels (1 byte)
6. Number of rows (1 byte)
7. Raw pattern-data, size is calculated by multiplying the number of bytes per event with the number of channels and rows.

This file format presented here is far from optimal; compression routines may be implemented by for example packing zero-only or repetitive entries, channels and/or rows into chunks, and refer to these with the help of special escape sequences within the file. There are many ways of compressing pattern-data (also song/instrument data), unfortunately we didn't have time to implement and/or test our ideas.

Chapter 6

Testing and quality assurance

Frequent testing and tweaking was preformed during the entire development process to assure the quality of our work. Testing was carried out in four different levels:

- Testing of the BeOS and the BeAPI.
- Testing of individual routines.
- Testing during/after the integration processes.
- Testing of the whole system.

6.1 Testing of the BeOS and the BeAPI

To be able to build a fast and stable framework (and a surrounding application) we had to test the responsiveness of different methods and components within the BeOS and its BeAPI. Such testing has led to suitable choices between numerous alternatives and made us discover some of the possibilities within the Be environment. This strategy has prevented us from implementing "wrong" and slow solutions.

Examples of testing at this level:

- Testing and comparison of Drawstring and DrawBitmap.
- Testing and comparison of DirectWindow and BWindow.
- Testing and comparison of semaphores and the BLocker.

6.2 Testing of individual routines

Time critical (custom made) routines were tested using the BStopWatch debugger tool of the BeAPI. CPU usage was also taken into consideration.

Examples of testing at this level:

- Testing and efficiency improvement of our team of custom threads.
- Testing and efficiency improvement of custom made graphical routines.
- Testing and debugging of hidden functionality.

6.3 Testing during the integration process

At the end of each iteration we have carefully tested that the components, both the previously developed modules, and the ones to be integrated were functioning and interacting as expected.

6.4 Testing of the whole system

The entire implementation was tested at the end of the development process. Our purpose was to ensure that all components were functioning and interacting correctly. We have also compared the final product to the requirements for being able to do a proper conclusion.

6.5 Strategy and tools

Constant focus on CPU usage, efficiency, execution speed and debugging ensured that the team was moving along the "right track". We have used several tools when it came to debugging and system tuning. Variables, timing results and debugging information was dumped to standard output; the GUI (window system) was not used for such purposes during development.

We have also maintained close contact with other developers to ensure the quality of the source code. Input from developers, users and our supervisor have sometimes led to faster implementation, and also automatically ensured improvement of quality. Please refer to the [Appendix C] for a complete set of quailty assuring rules, backup routines and emergency protocols.

6.6 Beta-testing

Unfortunately, beta-testing was only done on an individual basis. Because of exams during the finalization phase, we were interrupted in our construction of a user friendly beta release. After a set of exams, we decided to focus on the documentation instead of doing a beta-testing cycle. Remaining time was insufficient for carrying out a complete beta-testing phase with analysis of feedback. We also felt that the product was not completely ready for mass distribution.

Chapter 7

End discussion

7.1 The process

Since we chose to do a special project (which was more or less defined by ourselves), we had to describe the assignment already during November, last year. Search for a willing employer was carried out. We contacted Ano Tech Computers (www.atc.no), a firm that was interested in our project and willing to support it. ATC became our employer. The project idea vas evaluated and approved by HiG; our team could start working with the assignment.

To begin with, a complete analysis was preformed so that the problem domain could be segmented into smaller parts. Estimation of time required for each operation resulted in a Gantt diagram [Appendix D]. We made a formal contract which included rules, backup routines and emergency protocols. To prevent conflicts, a plan of action was also written. These documents were signed by all members. Fortunately, this document was only used in cases of minor emergencies.

Our team scheduled three meetings with the supervisor. The appointments were included in the Gantt diagram. During the elaboration phase we decided to schedule an extra meeting because of miscellaneous purposes. The previously defined iterations were more or less carried out as planned. Much of the work was done in plenary to ensure quality, controlled implementation and integration. Only minor fractions of the development could be carried out on an individual basis because of the complexity of this assignment. Because of several reasons the development model was not always followed in detail. Use case driven development was not always our highest priority. Instead of using use case driven development, we focused on analysing problem domains using custom methods. Upon agreement of possible solutions, actual code was implemented, tested and integrated. Tests were conducted to see if the previously requirements were fulfilled, and also for quality assuring purposes.

We started by exploring the possibilities within the Be environment. Basic understanding of the operating system and the API was necessary and helped

us starting development of the BeMiT application. Example code (included in BeOS) was carefully studied and tested from within the BeIDE. Simultaneously, the group started to analyse the basic concepts of earlier music tracking systems and the MIDI standard. Day by day, the team worked its way through, until a satisfying state was reached. The Rational Unified Process turned out to be a great tool when it came to carrying out this assignment. It helped us to constantly maintain control and provided stable overview during the entire process.

Chapter 8

Conclusions

8.1 Common goals

This project has resulted in a solid framework that has the stability and possibilities required by an operational tracker. The code is prepared so that it is easy to read, understand, modify and upgrade. People who are interested in tracker development (and who have carefully read this document - although not required) shouldn't have any problems with further development of the framework. A lot of research has been done to ensure that standards are implemented correctly. Even tough parts of the system uses BeOS/BeAPI specific solutions, it is easy to understand (and replace) these if the framework is to be ported to another platform. The project contains a set of carefully prepared variables and functions that together form a great (and standardized) fundament.

8.2 Private goals

It was determined that focus should be aimed at creating a fundament of knowledge, so that all of the team members could contribute within the project. Everybody had to participate during the entire development process. This focus has been positive and has helped to build a working team. The assignment has forced all of the members to use their knowledge and to extend these. All team members had to "stretch out" to be able to solve problems; this has been a huge factor of motivation.

8.3 What we have learned

The developers have learned to work as a team rather than individuals. The team has learned that it is necessary to summarize every aspect of knowledge and qualification to form a synergetic effect that is able to match a given challenge. Method of work was planned and carried out using the RUP development

model. Planning and organizing has lead to new and important experiences regarding problem solving on a larger scale. Great deal of time and resources were used to ensure knowledge possession for each and every team member regarding the BeAPI, tracker theory - and the MIDI standard. Strong focus on experimentation, optimization and knowledge acquirement has resulted in a report that might be weaker than expected. The team has learned that it is very important to constantly document everything during work.

8.4 Further work

A framework for a tracker system has been developed, founding the basis for further development. The components within the system are implemented in a way that it is easy to expand these and to upgrade the functionality so that a complete, working music tracker system can be made. The implementation is somewhat portable, making it possible to use the framework under different environments.

8.5 Group evaluation

The team has functioned very well throughout the whole project, there hasn't been any severe disagreements. Motivation and willingness to contribute has been satisfying.

The team has used a custom made blackboard for segmentation of problems, distribution of information, planning and problem solving. Thursdays were used to hold status meetings and to exchange important information/knowledge.

The supervisor has been extremely engaged, helpful and motivating and has provided suggestions around solutions even without knowing the target environment.

Chapter 9

Signatures

Graduation project BeMiT, BeOS Musical Instrument Tracker
at Gjøvik Collage 23.05.2001

Balazs Halasy	Bjarte Sæverud	Morten Trillhus