

**Tanager: a case study of iterative development in
object-oriented analysis and design**

by

Robert John Lavey

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Gary T. Leavens, Major Professor
Doug Jacobson
Leslie Miller

Iowa State University

Ames, Iowa

2007

Copyright © Robert John Lavey, 2007. All rights reserved.

Table of Contents

Acknowledgements.....	iii
Abstract.....	iv
1. Introduction.....	1
2. Motivation for Tanager Project.....	2
2.1. Related Work	2
3. Iterative Versus Waterfall Methodologies	3
3.1. Waterfall Methodology	3
3.2. Iterative Methodology.....	3
4. Tanager Project.....	7
4.1. Project Definition.....	7
4.2. Inception Phase	8
4.3. Elaboration Phase 1.....	14
4.4. Elaboration Phase 2.....	27
4.5. Elaboration Phase 3.....	35
4.6. Elaboration Phase 4.....	47
5. Advantages and Disadvantages of an Iterative OOAD Process.....	57
5.1. Disadvantages of Iterative OOAD Process.....	57
5.2. Advantages of Iterative OOAD Process	58
6. Conclusions.....	60
7. Appendix A – Analysis/Design Documentation.....	61
7.1. Vision.....	61
7.2. Supplementary Specification	63
7.3. Glossary	64
7.4. Use Case Model	65
7.5. Domain Model	66
7.6. Class Model	67
7.7. System Sequence Diagram Specification	68
7.8. Sequence Diagram Specification	69
7.9. Operation Contract Specification.....	70
8. Appendix B – Sample Javadocs.....	71
8.1. CMediaPlayer.java	71
8.2. CPlaylist.java	74
8.3. CSongCollection.java	76
9. Appendix C – Sample Code.....	78
9.1. CMediaPlayer.java	78
9.2. CPlaylist.java	80
9.3. CSongCollection.java	82
References.....	84

Acknowledgements

I would like to thank Dr. Gary Leavens for all of his help over the past 2 years. I'm sure it has seemed to him like I would never finish (as it has seemed to me), but he has always been there with words of support and guidance. I know it's been difficult for him to mentor me remotely, and I appreciate the extra effort he has put forth.

I would like to thank Dr. Jacobson and Dr. Miller for agreeing to be on my POS committee and for showing an interest in my work.

I am also indebted to my colleagues at Hewlett-Packard for enduring my seemingly endless pursuit of a Master's degree. They have been supportive, understanding, and helpful as I have spent energy that would otherwise have gone into LaserJet printers.

Finally I would like to thank my family for their support, help, and love during this time. Now we can go to Disney World!

Abstract

The Tanager project presents a case study of implementing a digital music player using an iterative approach to object-oriented analysis and design (OOAD). Its purpose is to help those who are interested in learning OOAD see how an iterative approach can be used to break down the problem into manageable pieces and allow the development team to come up with an extensible product that fulfills the user's requirements. The main goal of the project is to present an example of an iterative OOAD process for implementing the user's requirements. This is accomplished by showing the analysis and design artifacts used to communicate with the users, the analysis and design artifacts used to communicate with other developers, and how all of these elements are used together to implement and test the final product.

1. Introduction

The increased popularity of object-oriented (OO) programming languages has exposed more and more software engineers to OO concepts. However, many have not been similarly exposed to processes for utilizing the features of those languages to produce well-designed code that not only solves their user's problems but allows for extensibility to handle additional user requirements. A process for starting with user requirements and winding up with well thought-out designs and code can be a mystery for engineers facing a large set of user features and a looming deadline. The temptation to dive in and start typing code must be suppressed, since good code begins with good design, and good designs can seldom be found while typing code. Good designs not only achieve the user goals, but they also allow for extension and re-use; a well thought-out design makes it easier to add user requirements that were unforeseen during the original analysis and design phases or to reuse design elements in different projects.

This case study attempts to show an iterative object-oriented software engineering process using a project that most readers will be familiar with. The Tanager project is a digital music player, and since most people are familiar with the operations of such a device, the user requirements can be easily understood. This case study will show how to break down those user requirements into manageable pieces; analyze those pieces to estimate a schedule; and iteratively design, implement, and test those pieces to complete the final product. Examples of the artifacts generated in each phase, the benefits of those artifacts, and who the artifacts are used by will be shown. These artifacts may be text documents, UML diagrams, object models, code, tests, or anything else generated during the process.

2. Motivation for Tanager Project

Textbooks on object-oriented analysis and design (OOAD) techniques invariably contain one or two examples. Larman's Applying UML and Patterns [1], for example, provides the NextGen Point-of-Sale system and a Monopoly simulation using the iterative Unified Process (UP) methodology. While these are good examples, as Larman states, "On a UP project, we would tackle the difficult, risky things first. But in the context of a book helping people learn fundamental OOA/D and UML, we want to start with easier topics." [1, p. 124]. The Tanager case study will attempt to show how and why a project would tackle those difficult, risky things first.

A short example from an area familiar to most people, using iterative development with one inception phase and three or four elaboration phases will be instructive for readers interested in understanding more about OOAD using iterative development. Fewer elaboration phases will not provide enough diverse development challenges, and more elaboration phases will rehash many of those same development challenges. The Tanager project shows how to analyze, design, and implement a digital music player using one inception phase and four elaboration phases. In the spirit of a typical iterative OOAD development project, the difficult, risky features are tackled first, and the simpler features are handled last. This will give the reader an additional example to draw from when they begin their own development project.

2.1. *Related Work*

Curtis Clifton's StickSync project [8] is another example of an object-oriented analysis and design project using iterative development that readers may be interested in reviewing. The goal of the StickSync project is to design a system that uses portable mass storage devices to synchronize multiple computers that may not be connected by networks. Mr. Clifton provides the artifacts from the OOAD process as well as a journal of the project's progress. This journal helps differentiate the process from the artifacts.

3. Iterative Versus Waterfall Methodologies

It is not one of the goals of this paper to give a detailed description of iterative OOAD techniques, since space does not permit such a discussion, and there are many excellent books on the subject such as Applying UML and Patterns [1] or The Rational Unified Process [2]. However, a brief description of iterative techniques and how they differ from the more traditional waterfall techniques will provide a foundation for the case study.

3.1. Waterfall Methodology

Waterfall methodologies such as the Object Modeling Technique [6] attempt to complete each phase of development for all user requirements before moving on to the next phase – design is only begun after a thorough and complete analysis of the entire set of user requirements, and implementation is only begun after a full and complete design. While working in the analysis phase, developers thoughtfully consider *what* needs to occur in the system without giving any consideration to *how* it might be done. Once all of the analysis activities are completed, the developers move on to the design phase and thoughtfully consider *how* they can design a solution to accomplish all the “*what*” tasks discovered during analysis. It is strictly discouraged that developers consider *how* a solution could be designed while they are considering *what* the user’s goals are.

Waterfall methodologies have the advantage that they allow developers to understand the whole problem and how all aspects of the problem will interact with each other during each phase. The results of each phase will handle all the use cases and scenarios as they are understood by the developers — no time is wasted designing a solution that will work for one set of user goals but would be inadequate for others. However, one disadvantage of these methodologies is that users don’t have the opportunity to verify the design and implementation as it progresses, and if there are misunderstandings between the users and developers, they will likely be found very late, when they will be much more difficult and expensive to correct.

3.2. Iterative Methodology

Iterative methodologies such as the Rational Unified Process [2] break problems down into manageable pieces that can be designed and implemented in shorter periods of time. There is usually a preliminary step, the inception phase, at the beginning of the project where user

requirements are gathered, a business case for the project is defined, a rough estimate of time and cost is prepared, and a “go” or “no go” decision is made. Typical artifacts from the inception phase are the initial revisions of the project’s Vision, Supplementary Specification, Use Case Model, and Glossary. These documents are further refined during the subsequent development phases. Following the inception phase is one or more elaboration phases, during which the development team will analyze, design, implement, and test a subset of the user requirements. During this phase, the development team may also perform deeper analysis on the user requirements that will be handled during the next phase. Typical artifacts from this elaboration phases are updates to the Vision, Supplementary Specification, Use Case Model, and Glossary, and new documentation specific to analysis and design is generated and refined: the Domain Model, System Sequence Diagram Specification, Sequence Diagram Specification, Operation Contract Specification, and Class Model.

3.2.1. Inception Phase

The purpose of the inception phase is to investigate the scope and business case for the project but not to define all the requirements. The requirements definition will be an evolutionary process that is done iteratively in each elaboration phase; some requirements, however, must be defined in this phase to get development started in the first elaboration phase.

The main goal of the inception phase is to determine whether or not the project is feasible: will it produce a product that fits into the goals of the business, and will the product be marketable? To make this determination, many functional and non-functional requirements must be analyzed.

Functional requirements include how the product will behave and how the user will interact with the product, and these requirements are captured primarily in the Use Case Model. The development team will identify use cases for all of the user requirements during this phase, and they will maintain a close interaction with the users to ensure their requirements are accurately captured. Although all of the use cases are identified in this phase, only 10-20% will be thoroughly analyzed, since the main goal of this phase is to investigate the feasibility of the project.

Non-functional requirements include user demographics, comparisons with existing products, licensing, and legal issues. These requirements are captured in the Supplementary Specification. The Vision document is used to capture a high-level view of the details from the Use Case Model and Supplementary Specification. It is used for a quick overview of the product.

One other important goal of the inception phase is to prepare the development environment for the subsequent elaboration phases. Examples of the tools needed by the development team are a documentation tool capable of drawing UML diagrams, an HTML editor, a word processing application, a software development platform (compiler, debugger, etc.), a test development platform, and a version control system. These tools must be in place before any development activities can begin.

3.2.2. Elaboration Phases

Each elaboration phase will concentrate on a subset of the use cases and scenarios identified for the whole project. Elaboration phases have a fixed time, and the number of scenarios that will be tackled in an elaboration phase depends on the complexity of the scenarios and the length of the phase. Each phase consists of complete analysis, design, implementation, and testing of that phase's scenarios, and the output is production-ready code. One additional important step in each elaboration phase is to refine the use cases to be handled in the following elaboration phases.

The use cases for each elaboration phase are thoroughly analyzed, and the results of the analysis are captured in the Use Case Model. The use cases are also dissected to determine the domain objects, which are captured in the Domain Model. The Domain Model also captures the associations between the domain objects and attributes of the domain objects. The Domain Model shows the user's view of the system and is useful for making sure all of the actors and other conceptual objects from the use cases are captured. The textual descriptions of the use cases are used to create high-level System Sequence Diagrams (SSDs), which are captured in the System Sequence Diagram Specification.

The Use Case Model, Domain Model, and System Sequence Diagrams show the user's perspective of the system, and are used primarily to communicate with the user during the analysis phase. However, they are also useful for the developers during design activities. The high-level interactions in the SSDs show how the actors interact with the system, and these high-level interactions are captured in the Operation Contract Specification, which defines the external interface to the system. The Use Case Model, Domain Model, and SSDs are used to develop the implementation classes, attributes and operations of the classes, and associations between the classes. These are captured in the Class Model; the Class Model is a representation of the implementation of the system. Sequence Diagrams are then developed from the use cases. The Sequence Diagrams describe the dynamic behavior of the system by showing how the

implementation classes interact with each other; these diagrams are captured in the Sequence Diagram Specification. The Sequence Diagrams show the operations and parameters required for each of the implementation classes, and they aid in the transition from design to implementation. All of these artifacts are useful for testing. The analysis documentation, with its user perspective, is helpful for understanding the overall functionality of the system and can be used to develop black-box system tests. The design documentation, with its implementation perspective, is helpful for understanding the internal workings of the system and can be used to develop white-box unit tests.

By focusing on a subset of use cases and scenarios, the developers are able to complete the analysis, design, implementation, and test during the fixed time period of each elaboration phase. One advantage of this methodology is that any misunderstandings between the development team and the users will be found early and corrected before other features become dependent on it. However, as each new phase is designed and implemented, it may force changes to the design of previous phases: simple designs used in early phases may be inadequate for the complexities of later phases.

4. Tanager Project

The premise behind the Tanager project was that a company wished to break into the hardware-based digital music player business. This company saw a hole in existing products: there was a very different set of features and a very different user interface between the hardware-based players and their related software-based players. Further, if a user had a wide variety to different media types, they may need multiple software-based players to be able to play them all. A player that can handle a wide variety of media types and has the same feature set and the same look-and-feel will fill that hole. This company anticipated creating code that would run on both software and hardware platforms, and it saw its opportunity to break into hardware-based players by first introducing a software-based player under the BSD license. This would reduce the risk of producing the hardware-based units, since they would be building on existing, proven designs, and there would be an existing user base.

4.1. *Project Definition*

The Tanager digital music player would be capable of performing the following tasks:

1. **Download Songs** - The user must be able to download songs to the player. The user should be able to download a wide variety of different formats without having to install additional components. They should also be able to delete songs that were previously downloaded.
2. **View Downloaded Songs** - The user must be able to view a list of all the songs they have downloaded to their player.
3. **Play Songs** - The user must be able to play the songs they have downloaded to their player. They should also be able to choose different ordering methods for the songs to play in: random ordering, alphabetical ordering, etc.
4. **Manipulate Playing Songs** - The user must be able to pause a song, skip over the current song to either the next song or the previous song, or restart the current song. They must also be able to change the volume of the playing song.

4.2. *Inception Phase*

The inception phase was started with a short brainstorming session where the development team¹ outlined their plans for the phase. The work that needed to be accomplished in the phase was to determine the scope of the project, how the project fit into the company's business model, what the basic functional and non-functional requirements were, and generally whether or not the project was feasible. One preliminary step for accomplishing these goals was to gather templates for the various pieces of documentation that would be produced: the Vision, Supplementary Specification, Glossary, and Use Case Model. The documentation templates were provided by a team member who had been using the Rational Unified Process (RUP), and the templates were modified slightly to meet the team's needs. The document templates are a framework that can be filled out during the OOAD process, and using this type of documentation framework makes it easier for the developers to know they have captured the necessary information and that it is in a standard format. This standardized format makes it easier for stakeholders and developers to understand documentation for many different projects. It was decided during this brainstorming session that a web site should be used to capture the artifacts of the development process and to capture the additional materials that were used during the process (such as the documentation templates).

The brainstorming session gave the team a direction and a goal for the inception phase. The first step in completing those goals was to write the preliminary version of the Vision document, which would provide the high-level view of the project.

The product differentiators had been defined as having a common look-and-feel and a common feature set on both software and hardware platforms, and the ability to play a wide variety of media types. They were formalized in the Vision document as the set of problem statements shown in Figure 4.1. The proposed solutions for the issues raised in the problem statement were formalized in the position statement shown in Figure 4.2.

¹ The development team consisted solely of the author, Robert Lavey, and all analysis, design, implementation, and testing was done by him. Dr. Gary Leavens and the author's wife assumed the roles of the stakeholders, and they helped by performing the user's review and approval duties.

The problem of	multiple user interfaces
affects	media player users
the impact of which is	confusion about how to operate different media players
a successful solution would be	software-based and hardware-based media players with identical user interfaces.
The problem of	multiple feature sets
affects	media player users
the impact of which is	confusion and time wasted trying to figure out how to utilize the player's features
a successful solution would be	software-based and hardware-based media players with identical feature sets.
The problem of	limited media type support
affects	media player users
the impact of which is	users must utilize many different media players
a successful solution would be	a single player that supports a wide range of media types.

Figure 4.1 - Tanager Problem Statement

For	media player users
Who	have many different media types they wish to play
Tanager	is a software-based media player
That	plays AIFF, AU, AVI, MIDI, MP2, MP3, QT, RMF, and WAV files
Unlike	Microsoft Windows Media Player, which cannot play QuickTime audio files, or iPod, which cannot play Windows Media Files
Our product	Tanager will play QuickTime audio files and many other types of audio files, and it allows the user to need to know only a single feature set and user interface.

Figure 4.2 - Tanager Position Statement

The stakeholders of the system were then defined: all those parties who are interested in the development of the system, including users and non-users. The users were defined as the Playlist Administrator (shown in Figure 4.3) and the Music Listener (shown in Figure 4.4), and no non-user stakeholders were defined. These stakeholder profiles helped remind the development team what the users expect from the system and how the users will be involved in the development process.

Description	Wants to be able to download songs to the system, remove previously downloaded songs, and view the list of downloaded songs.
Type	The Playlist Administrator could have any level of experience with this or other digital music players. They should understand the concept of downloading music to the player.
Responsibilities	<ul style="list-style-type: none"> • Turns system on & off • Downloads songs • Removes downloaded songs • Views list of downloaded songs
Success Criteria	Success is defined by the list of downloaded songs accurately showing those songs that have been downloaded and not removed.
Involvement	This stakeholder will be involved in the requirements-gathering process, and they will be involved in testing implemented solutions.
Deliverables	This stakeholder requires a user's guide describing the system's functionality.

Figure 4.3 - Playlist Administrator Stakeholder Profile

Description	Wants to be able to play music that has been previously downloaded to the system.
Type	The Music Listener could have any level of experience with this or other digital music players. They should understand the concept of playing music on a digital music player.
Responsibilities	<ul style="list-style-type: none"> • Turns system on & off • Plays songs • Skips songs • Repeats songs • Views list of downloaded songs • Adjusts the volume
Success Criteria	Success is defined by the user's ability to play songs that have been previously downloaded to the system.
Involvement	This stakeholder will be involved in the requirements-gathering process, and they will be involved in testing implemented solutions.
Deliverables	This stakeholder requires a user's guide describing the system's functionality.

Figure 4.4 - Music Listener Stakeholder Profile

The next step in achieving the inception phase goals was to write the initial version of the Supplementary Specification. In the Supplementary Specification, non-functional requirements such as performance, reliability, usability, and licensing are captured. The development team

knew they could not reasonably implement all of the media processing aspects of a digital media player, so they chose to use Sun's Java Media Framework (JMF) as the underlying audio media handler. The decision to use JMF, the software interface to JMF, and the JMF licensing requirements were all captured in the Supplementary Specification.

Next the development team began work on the initial version of the Use Case Model. All of the use cases were identified from the stakeholder's needs, and they were written in a brief format [1, p. 63]. Figure 4.5 shows an example of the Power On use case written in a brief format.

2.1. Power On

2.1.1. Brief Format

The Music Listener tells the Tanager product to power on. The Tanager system responds by powering on the system and displaying a message saying it's ready to accept commands. This use case ends when the system has powered on.

Figure 4.5 - Power On Use Case in the Brief Format

As the Vision, Supplementary Specification, and Use Case Model were written, many terms and acronyms were discovered that may not have been known to the reader, or which had ambiguous definitions that needed to be formalized. As each of these terms or acronyms was discovered it was captured in the Glossary.

At the end of the inception phase, the development team and stakeholders had another meeting to make a decision about whether or not the Tanager project should proceed. They reviewed the documentation that had been produced during the inception phase to understand the magnitude of the project and the business case. In addition, an informal schedule was generated, which showed all the work involved in completing the project and a very rough estimate of how long that work would take. Since the list of functional and non-functional requirements was manageable, the stakeholder's needs were well understood, and the informal schedule showed that the project could be completed in a reasonable amount of time, it was decided that the team would continue with the project.

At this point, the team needed to prepare for the analysis, design, implementation, and test steps in the first elaboration phase. This required further analysis of those use cases that were scheduled to be implemented in the first and second elaboration phases, obtaining any required tools or training, and setting up the development environment.

The use cases were prioritized such that work on the ones with the largest architectural coverage and the greatest risk would be done first. The team decided that the riskiest use cases were those that used functionality from the JMF. It was decided that the team would work on the use cases in this order:

1. **Elaboration Phase 1:** Power On, Power Off, and Download a Song (with no error checking)
2. **Elaboration Phase 2:** Download a Song (error checking scenario), Play Music, and Pause Music
3. **Elaboration Phase 3:** View Playlist, Delete a Song, and Volume Adjustments
4. **Elaboration Phase 4:** Select Playlist Order, Skip to the Next Song, Restart the Current Song, and Skip to the Previous Song

The Power On use case was chosen to be first, because it required a major portion of the architecture to be designed, implemented, and tested. It is always a good idea to have a large portion of the architecture implemented in the first elaboration phase, so it will be thoroughly exercised in all of the elaboration phases, and the existing architecture can be built upon in later elaboration phases. Download a Song was chosen for this phase because it is required for most of the rest of the use cases. The Play Music use case would have been a good choice for this phase, since it is the most architecturally risky, but it could not be tackled until there were songs in the system that could be played.

To be ready to implement the use cases that were going to be handled in the elaboration phases, the team decided on a process where they would write the fully-dressed format for each use case that would be implemented in the next phase and to write the casual format for each use case that would be implemented in the phase after that. To be prepared to begin elaboration phase 1, the team first wrote its use cases in the casual format [1, p. 63] and then refined its use cases by writing them in the fully-dressed format [1, p. 67]. Figure 4.6 shows the casual format for the Power On use case that will be implemented in elaboration phase 1, and Figure 4.7 shows the fully-dressed format for the Power On use case. To be prepared for elaboration phase 2, the team wrote its use cases in the casual format, and those use cases would be refined in elaboration phase 1 by writing them in the fully-dressed format. Each of these refinements to the use cases was captured in the Use Case Model.

2.1. Power On

2.1.1. Casual Format

The music listener tells Tanager to power on. The Tanager system responds by powering up the system and telling the user that it is ready to accept commands. The music listener can now interact with Tanager to play music, modify the music downloaded on the system, or modify the music playback options.

Figure 4.6 - Power On Use Case in the Casual Format

2.1. Power On

2.1.1. Fully- Dressed Format

2.1.1.1. Brief Description

This use case describes the user turning on the system.

2.1.1.2. Scope

The Tanager system.

2.1.1.3. Level

User-goal.

2.1.1.4. Primary Actor

Music Listener.

2.1.1.5. Stakeholders and Interests

2.1.1.5.1. Music Listener

The Music Listener wants the system to boot up without errors and without crashing the computer it's running on; for the system to restart at the last known state, if the system can find a valid last known state; and for the system to be in a state where it can accept commands to play music.

2.1.1.6. Preconditions

None.

2.1.1.7. Postconditions (Success Guarantee)

The system has booted up and is available for the user to interact with, and the system state has been saved to non-volatile memory.

2.1.1.8. Main Success Scenario

1. The user tells the system to power on.
2. The system checks its non-volatile memory to determine its last known state, finds a valid last known state, and initializes itself to that last known state.
3. The system saves its current state to non-volatile memory.

2.1.1.9. Extensions

- 2a. No valid last known state is found in non-volatile memory.
 1. The system does not find a valid last known state in its non-volatile memory, so it initializes to a default state.

Figure 4.7 - Power On Use Case in the Fully-Dressed Format

Since the Tanager project was going to be freely-distributed software, cost was a concern to management, and the team needed to mostly use freely-available tools. Tanager was to be implemented in Java, so Eclipse was the logical choice for a development environment, and it included JUnit, so the team also got the test environment as part of the package. Similarly, Concurrent Versions System (CVS) was the logical choice for version control, since it integrates with Eclipse well. The team needed an OOAD tool to generate the UML documentation for the project, so several different tools such as ArgoUML and Poseidon for UML were investigated. Poseidon for UML was the clear winner, since it implemented all of the necessary UML diagrams (ArgoUML didn't support sequence diagram, for example), the Community Edition was free, and the tool can generate Java code stubs from the class model. Finally, the development team already owned a license for Microsoft Word, so it would be used for word processing and HTML editing.

In this phase, the development team established the vision and business case, captured their findings in the Vision, Supplementary Specification, Use Case Model, and Glossary, refined the analysis of the use cases for the first and second elaboration phases, and set up the development environment. The team was now ready to begin the elaboration phases to design, implement, and test the use cases and to refine the use cases to be handled in future elaboration phases.

4.3. *Elaboration Phase 1*

4.3.1. Overview

Based on the schedule created in the inception phase, the plan for this phase was:

1. Implement the basic scenario for the Power On use case: power on to default state. No checking for saved system state will be done.
2. Implement the basic scenario for the Power Off use case: system state will not be saved.
3. Implement the basic scenario for the Download a Song use case: download a song with no error checking. No checking for existence of music file will be done.
4. Write the casual format for the View Playlist, Delete a Song, and Volume Adjustments use cases (to be further refined in elaboration phase 2).
5. Write the fully-dressed format for the Play Music and Pause Music use cases (to be implemented in elaboration phase 2).

Steps 4 and 5 are similar to the work the development team did refining use cases during the inception phase, so we will not go over it again here or in the subsequent elaboration phases.

Since the development team was starting from scratch on the architecture, there were fewer use cases to be implemented in this phase. However, the architecturally complex Power On use case was one of those to be developed, so the basic framework of the architecture could be constructed in this phase and built upon in subsequent elaboration phases.

4.3.2. Analysis

The team began this first elaboration phase by analyzing the use cases to gather the domain objects. They did this by using the technique of pulling all the nouns from the use cases to get a list of potential domain objects. Figure 4.8 shows the use cases with their nouns highlighted.

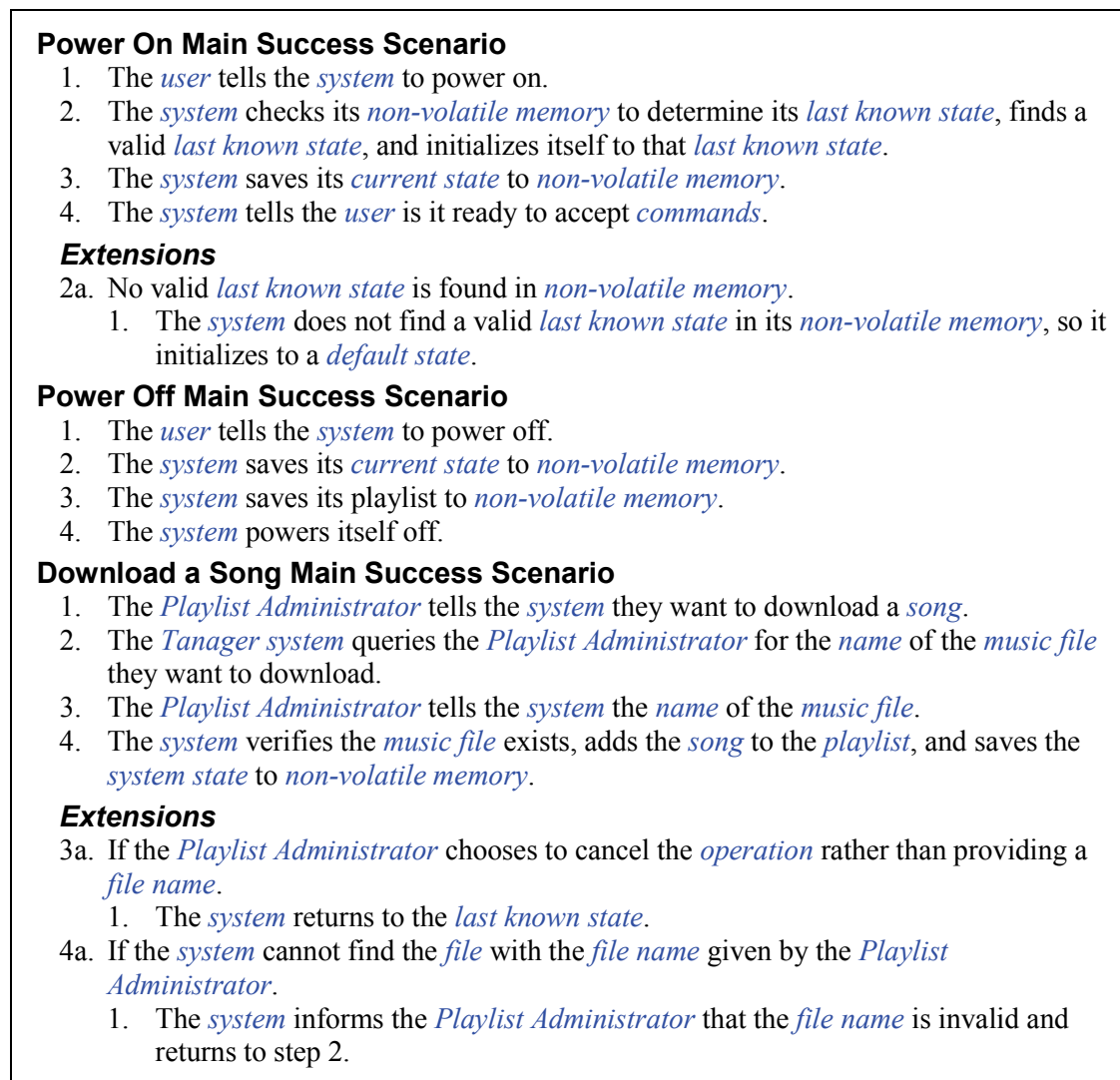


Figure 4.8 - Elaboration Phase 1 Potential Domain Objects

The team then went through the list of potential domain objects and removed those that were not part of the domain or were attributes of domain objects. The results of this activity are shown in Figure 4.9.

Potential Domain Object	Result
<i>user</i>	domain object: Music Listener or Playlist Administrator
<i>system</i>	domain object: Music Player
<i>non-volatile memory</i>	domain object: Non-Volatile Memory
<i>last known state</i>	domain object: State (“last known” is an attribute)
<i>current state</i>	domain object: State (“current” is an attribute)
<i>commands</i>	not in domain
<i>default state</i>	domain object: State (“default” is an attribute)
<i>song</i>	domain object: Song
<i>name</i>	attribute of music file
<i>music file</i>	not in domain
<i>playlist</i>	domain object: Playlist
<i>operation</i>	not in domain

Figure 4.9 - Potential Domain Object Resolution

The use cases were then further analyzed to discover any associations between the domain objects. There are many methods for finding associations between objects. Larman has a good checklist of common associations [1, p.155] for finding associations within use case descriptions, and another technique is to look for verbs that link domain objects. For example, the Power On use case says, “The user tells the system to power on.” which indicates an association between the user (the Music Listener or the Playlist Administrator) and the system (the Music Player). That is modeled in the domain model as an association named “Powers-on.” Figure 4.10 shows the use cases with the domain objects and potential domain object associations highlighted, and Figure 4.11 shows how the potential domain object associations were resolved. Using the domain objects, associations, and attributes, the team generated the diagram shown in Figure 4.12 and captured that in the Domain Model.

Power On Main Success Scenario

1. The *user* **tells** the *system* to **power on**.
2. The *system* **checks** its *non-volatile memory* to determine its *last known state*, finds a valid *last known state*, and **initializes** itself to that *last known state*.
3. The *system* **saves** its *current state* to *non-volatile memory*.
4. The *system* **tells** the *user* **is it ready** to accept commands.

Extensions

- 2a. No valid *last known state* is found in *non-volatile memory*.
 1. The *system* does not find a valid *last known state* in its *non-volatile memory*, so it **initializes** to a *default state*.

Power Off Main Success Scenario

1. The *user* **tells** the *system* to **power off**.
2. The *system* **saves** its *current state* to *non-volatile memory*.
3. The *system* **saves** its *playlist* to *non-volatile memory*.
4. The *system* **powers itself off**.

Download a Song Main Success Scenario

1. The *Playlist Administrator* **tells** the *system* they want to **download** a *song*.
2. The *Tanager system* **queries** the *Playlist Administrator* for the name of the music file they want to download.
3. The *Playlist Administrator* **tells** the *system* the name of the music file.
4. The *system* **verifies** the music file exists, **adds** the *song* to the *playlist*, and **saves** the *system state* to *non-volatile memory*.

Extensions

- 3a. If the *Playlist Administrator* chooses to cancel the operation rather than providing a *file name*.
 1. The *system* **returns** to the *last known state*.
- 4a. If the *system* cannot find the file with the file name given by the *Playlist Administrator*.
 1. The *system* **informs** the *Playlist Administrator* that the file name is invalid and returns to step 2.

Figure 4.10 - Elaboration Phase 1 Potential Domain Object Associations

Potential Domain Object Association	Result
<i>user</i> tells <i>system</i> to power on	Music Listener Powers-on Music Player Playlist Administrator Powers-on Music Player
<i>system</i> checks <i>non-volatile memory</i> to determine its <i>state</i>	State Reads-data-from Non-Volatile Memory
<i>system</i> saves its current <i>state</i> to <i>non-volatile memory</i>	State Writes-data-to Non-Volatile Memory
<i>system</i> tells <i>user</i> it is ready to accept commands	This is an embodiment of the observer/observable pattern and isn't shown in the domain model
<i>system</i> initializes to a default <i>state</i>	Self association, not shown in domain model
<i>user</i> tells <i>system</i> to power off	Music Listener Powers-off Music Player Playlist Administrator Powers-off Music Player
<i>system</i> saves its <i>playlist</i> to <i>non-volatile memory</i>	Playlist Writes-data-to Non-Volatile Memory
<i>Playlist Administrator</i> tells the <i>system</i> they want to download a <i>song</i>	Playlist Administrator Adds-songs-to Playlist
<i>system</i> queries the <i>Playlist Administrator</i> for the name of the music file to download	This describes an parameter of the download operation
<i>Playlist Administrator</i> tells for <i>system</i> the name of the music file to download	This describes an parameter of the download operation
<i>system</i> verifies the music file exists	This describes an parameter of the download operation
<i>system</i> returns to a default <i>state</i>	Self association, not shown in domain model
<i>system</i> informs the <i>Playlist Administrator</i> that the file name is invalid	This describes an parameter (return value) of the download operation

Figure 4.11 - Potential Domain Object Association Resolution

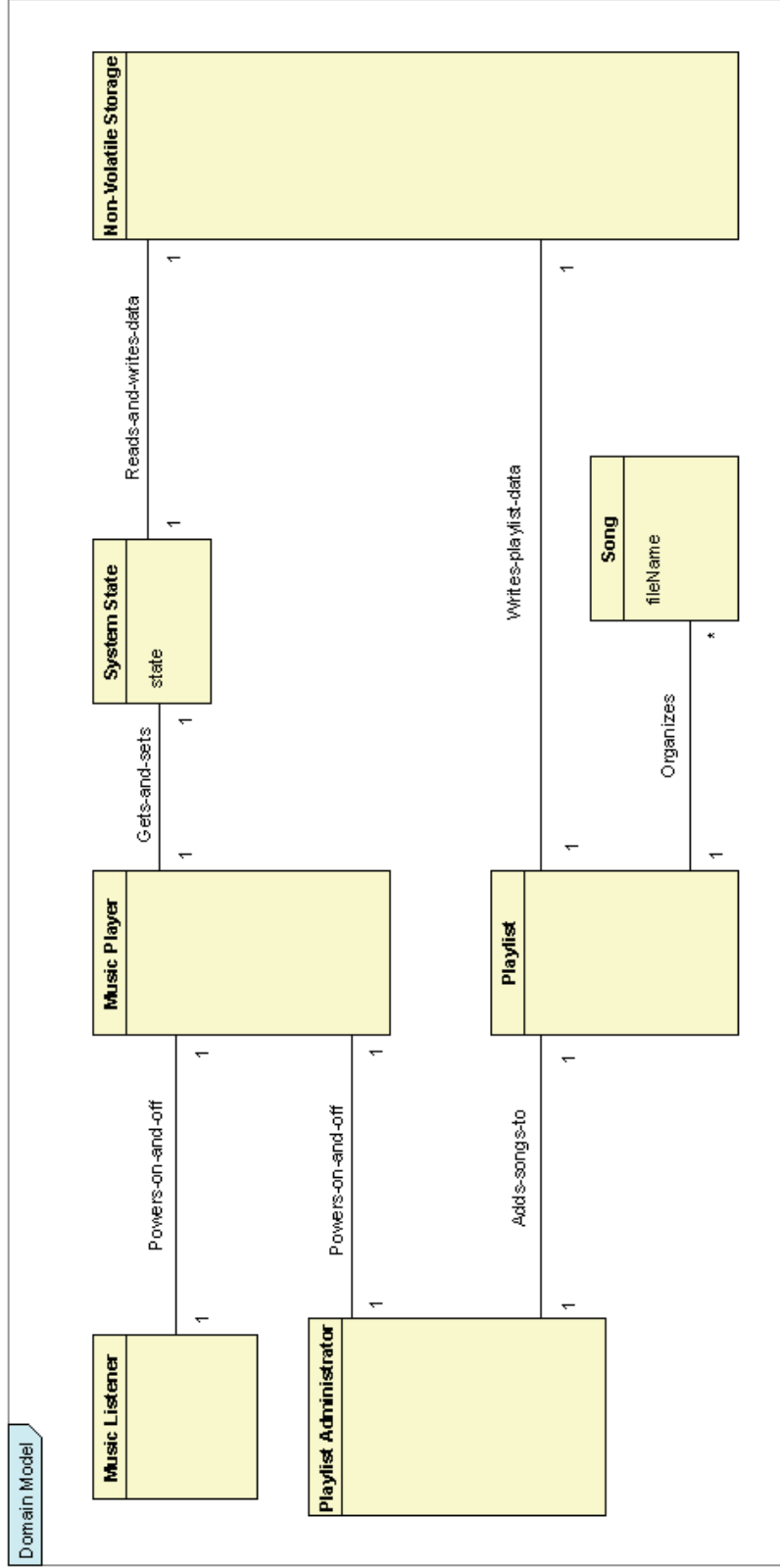


Figure 4.12 - Elaboration Phase 1 Domain Model

4.3.3. Design

4.3.3.1. System Sequence Diagrams and Operation Contracts

Using the use case descriptions and the Domain Model, the development team then began work on the SSDs and operation contracts. The SSDs are simple UML diagrams showing how actors interact with the system. These diagrams are important, since they show the external interface for the system under design. The operations discovered in the SSDs are further refined into operations contracts, which show the pre- and post-conditions of the domain model before and after execution of the operation.

Figure 4.13 shows the SSD for the Power On use case, and Figure 4.14 shows the operation contract for the PowerOn operation discovered in this SSD. These are captured in the System Sequence Diagram Specification and Operation Contract Specification, respectively.

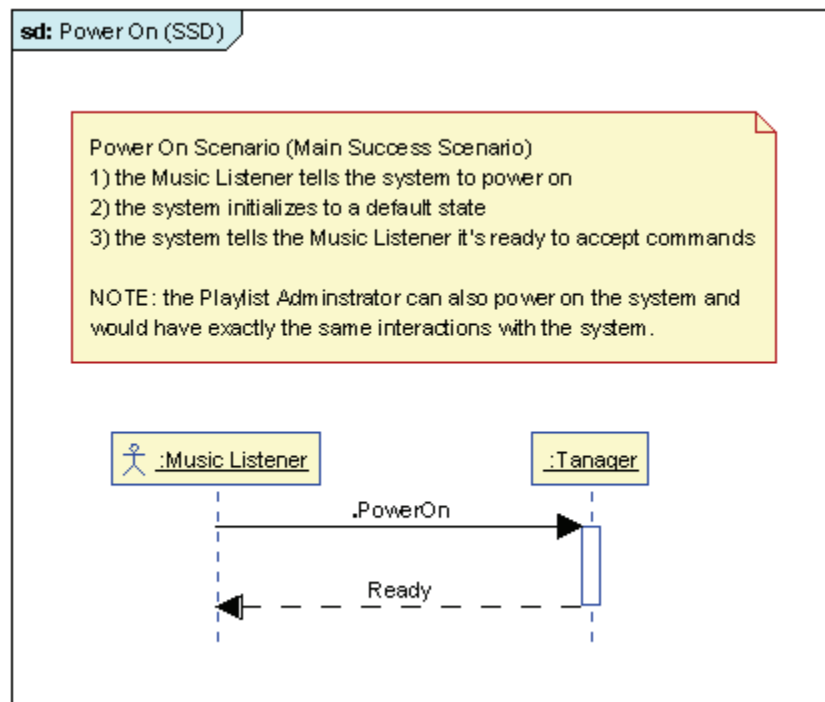


Figure 4.13 - Power On System Sequence Diagram

Operation	PowerOn()
Cross References	Use Cases: Power On
Preconditions	None
Postconditions	<ul style="list-style-type: none"> • A MusicPlayer instance, musicPlayer, was created • A SystemState instance, systemState was created and was associated with the MusicPlayer instance • A Playlist instance, playlist, was created and was associated with the MusicPlayer instance • A SongCollection instance, songCollection, was created and was associated with the Playlist instance • A Song instance was created for each file found in the FileStore, and each Song instance was associated with the SongCollection instance

Figure 4.14 - PowerOn Operation Contract

4.3.3.2. Sequence Diagrams and Class Diagram

The Tanager development team used the use cases, Domain Model, SSDs, and operation contracts to generate the sequence diagrams and the class diagram. The sequence diagrams are a refinement of the SSDs. However, where the SSDs show the interactions between actors and the system, the sequence diagrams show the interactions between classes in the implementation. For example, the SSD from Figure 4.13 would be refined into a sequence diagram that showed how the system would process the `PowerOn()` call to produce the changes in the system specified by the `PowerOn()` operation contract. Rather than changes to domain objects, which the operation contract describes, the sequence diagram shows how classes in the implementation (the Class Model) are changed when the operation is executed. There is often a matching class in the class model for each object in the domain model. However, there are usually many more classes in the Class Model, since the domain has to be modeled in software with greater detail than the Domain Model shows. The Power On sequence diagram snippet shown in Figure 4.15 shows how the dynamic behavior of the system is modeled in the Class Model, and how that relates to the Domain Model. The `CTanagerUI` class models the Music Listener and Playlist Administrator domain objects, the `CMusicPlayer` class models the Music Player domain object, and the `TanagerSongCollection:ObjectInputStream` class instance models one use of the Non-Volatile Storage domain object. Figure 4.15 also shows how the SSD is refined as a sequence diagram:

notice that the call into the system that starts the sequence diagram is `powerOn()`, but that there is a much greater level of detail in the sequence diagram.

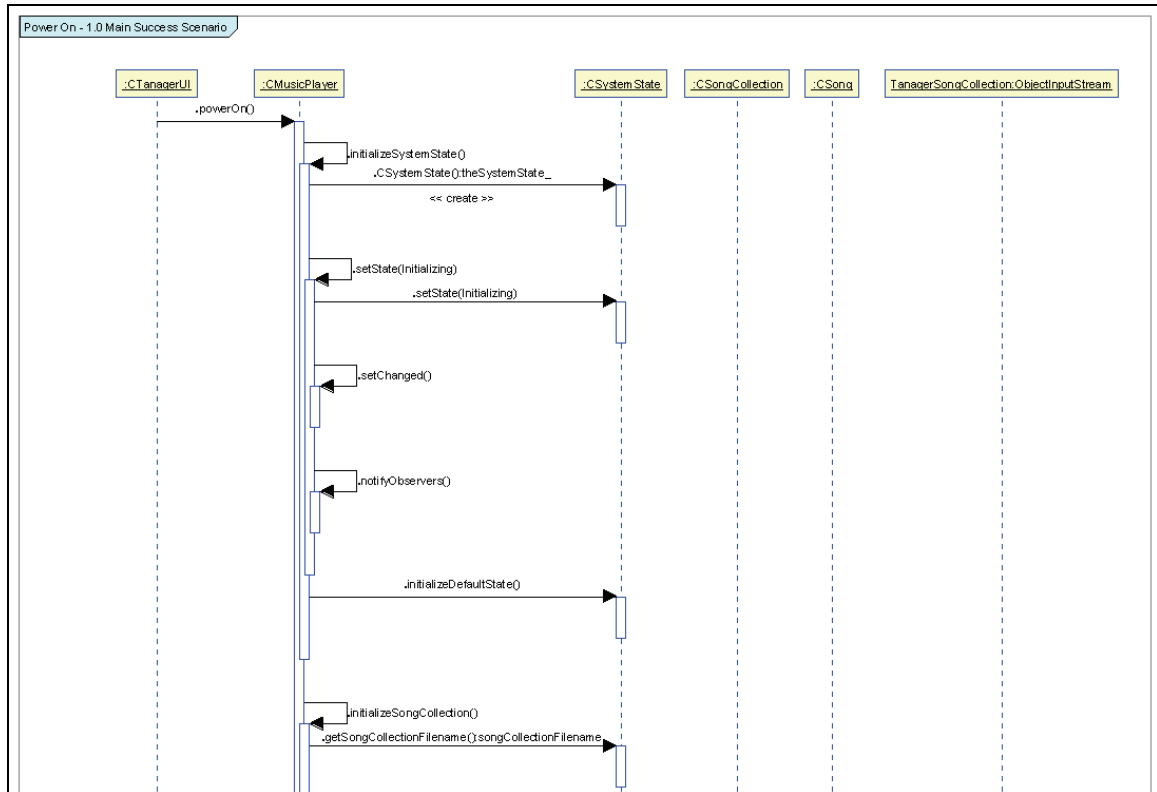


Figure 4.15 - Elaboration Phase 1 Power On Sequence Diagram

Each lifeline (the vertical lines with the square boxes at the top) represents a class instance from the class model, and each message (the horizontal lines) represents an operation from the class pointed to by the arrow. Messages may go from one class instance to another, or they may go from a class to itself.

The development team drew sequence diagrams for each use case scenario that was to be implemented during elaboration phase 1 and they captured those sequence diagrams in the Sequence Diagram Specification. The sequence diagrams were drawn using Poseidon, since it was faster to update electronic versions of the diagrams than hand-drawn versions. The very first iteration of each sequence diagram was drawn by hand, however, since it's easier to brainstorm possible interactions on paper than using a drawing tool. Once the preliminary diagram was drawn, it was transferred to Poseidon and updated there. The team also captured the classes and class operations in the Class Model using Poseidon. As each new class was added, the documentation for the class was also entered into Poseidon, so accurate javadocs could later be

generated along with the class definitions. Similarly, the operations discovered in the sequence diagrams were added to Poseidon along with their semantics and other documentation, so the operation stubs and their javadocs could be generated before implementation.

Figure 4.16 shows the Class Model at the end of elaboration phase 1. Compare the Class Model with the Domain Model shown in Figure 4.12, and you'll see that they are very similar.

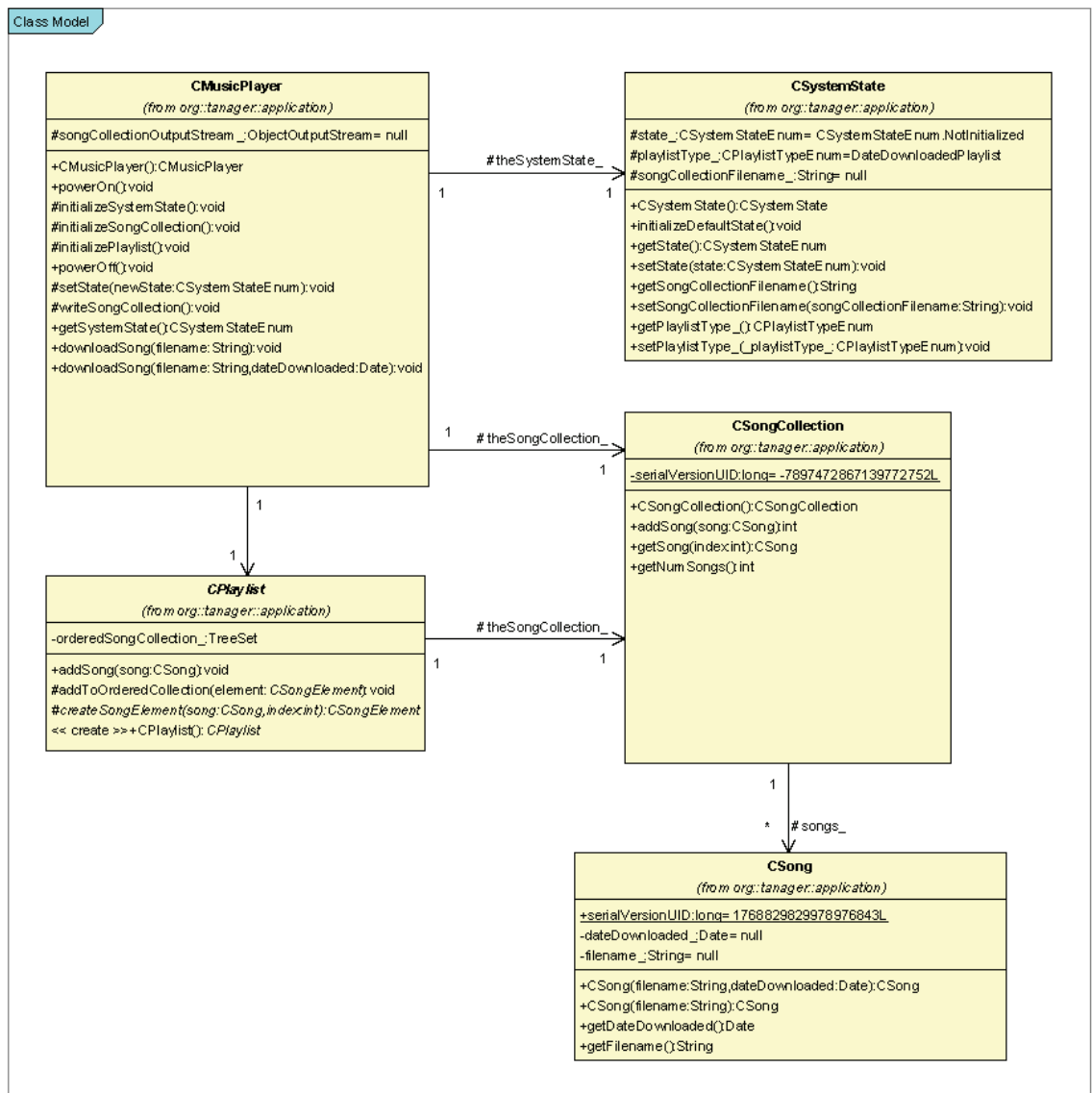


Figure 4.16 - Elaboration Phase 1 Class Model

4.3.4. Implementation

Once all of the sequence diagrams were finished and the Class Model was updated with the classes, associations, operations, and attributes, the Tanager development team generated the Java classes, attributes, and operation stubs using Poseidon. The operations were then implemented by the developers according to the operation semantics entered into Poseidon during the sequence diagramming process.

Figure 4.17 shows the code that Poseidon generated for the `CMusicPlayer.powerOn()` operation. Figure 4.18 shows the javadoc generated from the `CMusicPlayer.powerOn()` code. Figure 4.19 shows the `CMusicPlayer.powerOn()` operation after implementation.

```

/**
 * <p>
 * This method is called by the UI layer
 * to power on the Tanager System.
 * </p>
 * <strong>Semantics:</strong>
 * <ol>
 * <li>call <code>InitializeSystemState()</code></li>
 * <li>call <code>InitializeSongCollection()</code></li>
 * <li>call <code>InitializePlaylist()</code></li>
 * <li>set the state to Idle and notify any Observers
 * of the change</li>
 * </ol>
 *
 *
 * @poseidon-object-id [Ib77928m10992590c55mm748b]
 */
public void powerOn() {
    // your code here
}

```

Figure 4.17 - Poseidon-generated code for `CMusicPlayer.powerOn()`

Method Detail

powerOn

```
public void powerOn()
```

This method is called by the UI layer to power on the Tanager System.

Semantics:

1. call `InitializeSystemState()`
2. call `InitializeSongCollection()`
3. call `InitializePlaylist()`
4. set the state to Idle and notify any Observers of the change

Figure 4.18 - Javadoc for `CMusicPlayer.powerOn()`

```
/**
 * <p>
 * This method is called by the UI layer
 * to power on the Tanager System.
 * </p>
 * <strong>Semantics:</strong>
 * <ol>
 * <li>call <code>InitializeSystemState()</code></li>
 * <li>call <code>InitializeSongCollection()</code></li>
 * <li>call <code>InitializePlaylist()</code></li>
 * <li>set the state to Idle and notify any Observers
 * of the change</li>
 * </ol>
 *
 *
 * @poseidon-object-id [Ib77928m10992590c55mm748b]
 */
public void powerOn() {

    //
    // TODO - need try...catch here for initialization exceptions
    //
    initializeSystemState();
    initializeSongCollection();
    initializePlaylist();

    setState(CSystemStateEnum.Idle);

}
```

Figure 4.19 - `CMusicPlayer.powerOn()` after implementation

4.3.5. Testing

The development team was limited on the resources they could apply to testing, so a large suite of regression unit tests was out of the question. The team was faced with having to implement a prototype user interface (UI) to provide demonstrations to the stakeholders. They turned that to their advantage by using the UI to drive their application, which allowed them to perform black box system-wide exploratory testing using the use cases and sequence diagrams for their test plans. For this first elaboration phase, however, it was important to develop a base set of regression tests, since the functionality implemented in this phase would be built upon in all subsequent phases. The team needed to be sure that changes in those subsequent phases didn't break the base functionality, and regression testing with unit tests is the most efficient way of doing that. They decided that the set of unit tests that would satisfy this requirement would be one test that would exercise the system's ability to power-on to a default state, one test that would exercise the system's ability to download a single song, and one test that would exercise the system's ability to download multiple songs and have them ordered correctly in the playlist. The exploratory tests that were run exercised the system's ability to power-on without crashing or hanging and to download songs without crashing or hanging. Since there was no way to view the playlist at this stage of development, there was no way to verify the downloaded songs using exploratory testing, but the unit tests adequately covered that.

4.3.6. Results

During this phase, the development team laid the architectural groundwork for the Tanager project by implementing the use cases with the most architectural coverage and by implementing a set of regressions tests to exercise that architecture. They created the initial versions of the Domain Model and Class Model, and they created SSDs, operation contracts, and sequence diagrams. Once the analysis and design steps were finished, they generated code from the class model using Poseidon and implemented the operations using the semantics as a guideline. After completing the implementation, testing it with exploratory and unit test, they demonstrated it for the stakeholders. During this phase they also refined the use cases for the next elaboration phases and archived the artifacts from this phase.

4.3.7. Additional Documentation

The full set of artifacts and documentation produced by the Tanager development team during elaboration phase 1 can be found on their web site at

http://www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_01/ .

4.4. *Elaboration Phase 2*

4.4.1. Overview

Based on the schedule created in the inception phase, the plan for this phase is:

1. Implement the alternate scenario for the Download a Song use case: unsupported song content type is downloaded.
2. Implement the basic scenario for the Play Music use case: play music when system is not paused.
3. Implement the basic scenario for the Pause Music use case: pause music when music is playing.
4. Implement the alternate scenario for the Play Music use case: play music when music is paused.
5. Write the casual format for the Select Playlist Order, Restart Current Song, and Skip to Next Song use cases (to be further refined in elaboration phase 3).
6. Write the fully-dressed format for the View Playlist, Delete a Song, and Volume Adjustments use cases (to be implemented in elaboration phase 3).

The development team established the architecture for the Tanager project in the first elaboration phase, and it would be built on in this second elaboration phase. The most risky use case, Play Music, was to be implemented in this phase, but the Download a Song use case that verified the song type was supported was a prerequisite to playing the downloaded song. Another risky use case, Pause Music, was also scheduled for this phase. The risk in these two use cases comes from the fact that the development team would be interacting with JMF to play and pause music, and none of the members had experience with that package. By implementing these use cases early in the project, the risks would be mitigated by the testing that would occur during the rest of the project.

4.4.2. Analysis

The team began this phase by analyzing the use cases to gather the domain objects. They again used the technique of pulling all the nouns from the use cases to get a list of potential domain objects. Figure 4.20 shows the use cases with their nouns highlighted, and Figure 4.21 shows the results of analyzing the potential domain objects. Although no new domain objects were identified from the list, several new attributes of existing objects were identified.

Play Music Main Success Scenario	
1.	The <i>user</i> tells the <i>system</i> to play <i>music</i> .
2.	The <i>system</i> checks that it is not in a <i>paused state</i> , and it begins playing the <i>currently-selected playlist</i> from the <i>beginning</i> .
Extensions	
2a.	If the <i>system</i> is in a <i>paused state</i> (the <i>Music Listener</i> had previously paused the <i>music playback</i>).
1.	The <i>system</i> begins playing the <i>currently-selected playlist</i> from the <i>point at which it was paused</i> .
Pause Music Main Success Scenario	
1.	The <i>user</i> tells the <i>system</i> to pause <i>music playback</i> .
2.	The <i>system</i> stops playing and saves the <i>point at which playback stopped</i> .

Figure 4.20 - Elaboration Phase 2 Potential Domain Objects

Potential Domain Object	Result
<i>user</i>	domain object: Music Listener or Playlist Administrator
<i>system</i>	domain object: Music Player
<i>music</i>	not in domain
<i>paused state</i>	domain object: State (“paused” is an attribute)
<i>currently-selected playlist</i>	domain object: Playlist (“currently-selected” is an attribute)
<i>music playback</i>	not in domain
<i>point at which it (the playlist) was paused</i>	domain object: State (“paused song” is an attribute) domain object: Song (“paused point” is an attribute)
<i>point at which playback stopped</i>	domain object: State (“paused song” is an attribute) domain object: Song (“paused point” is an attribute)

Figure 4.21 - Resolution of Potential Domain Objects

Figure 4.22 shows the use cases with the domain objects and potential domain object associations highlighted, and Figure 4.23 shows how the potential domain object associations were resolved. Using the domain objects, associations, and attributes, the team generated the diagram shown in Figure 4.24 and captured that in the Domain Model.

<p>Play Music Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>user</i> tells the <i>system</i> to play music. 2. The <i>system</i> checks that it is not in a <i>paused state</i>, and it begins playing the <i>currently-selected playlist</i> from the beginning. <p>Extensions</p> <ol style="list-style-type: none"> 2a. If the <i>system</i> is in a <i>paused state</i> (the <i>Music Listener</i> had previously paused the music playback). <ol style="list-style-type: none"> 1. The <i>system</i> begins playing the <i>currently-selected playlist</i> from the <i>point at which it was paused</i>. <p>Pause Music Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>user</i> tells the <i>system</i> to pause music playback. 2. The <i>system</i> stops playing and saves the <i>point at which playback stopped</i>.

Figure 4.22 - Elaboration Phase 2 Potential Domain Object Associations

Potential Domain Object Association	Result
<i>user</i> tells <i>system</i> to play music	Music Listener Plays-music-on Music Player
<i>system</i> checks that it is not in a <i>paused state</i>	Music Player Gets State
<i>system</i> begins playing the <i>currently-selected playlist</i>	Music Player Plays-songs-in Playlist
<i>user</i> tells <i>system</i> to pause music	Music Listener Pauses-music-on Music Player
<i>system</i> stops playing the <i>currently-selected playlist</i>	Music Player Pauses-songs-in Playlist
<i>system</i> saves the <i>point at which playback stopped</i>	Music Player Sets State

Figure 4.23 - Potential Domain Object Association Resolution

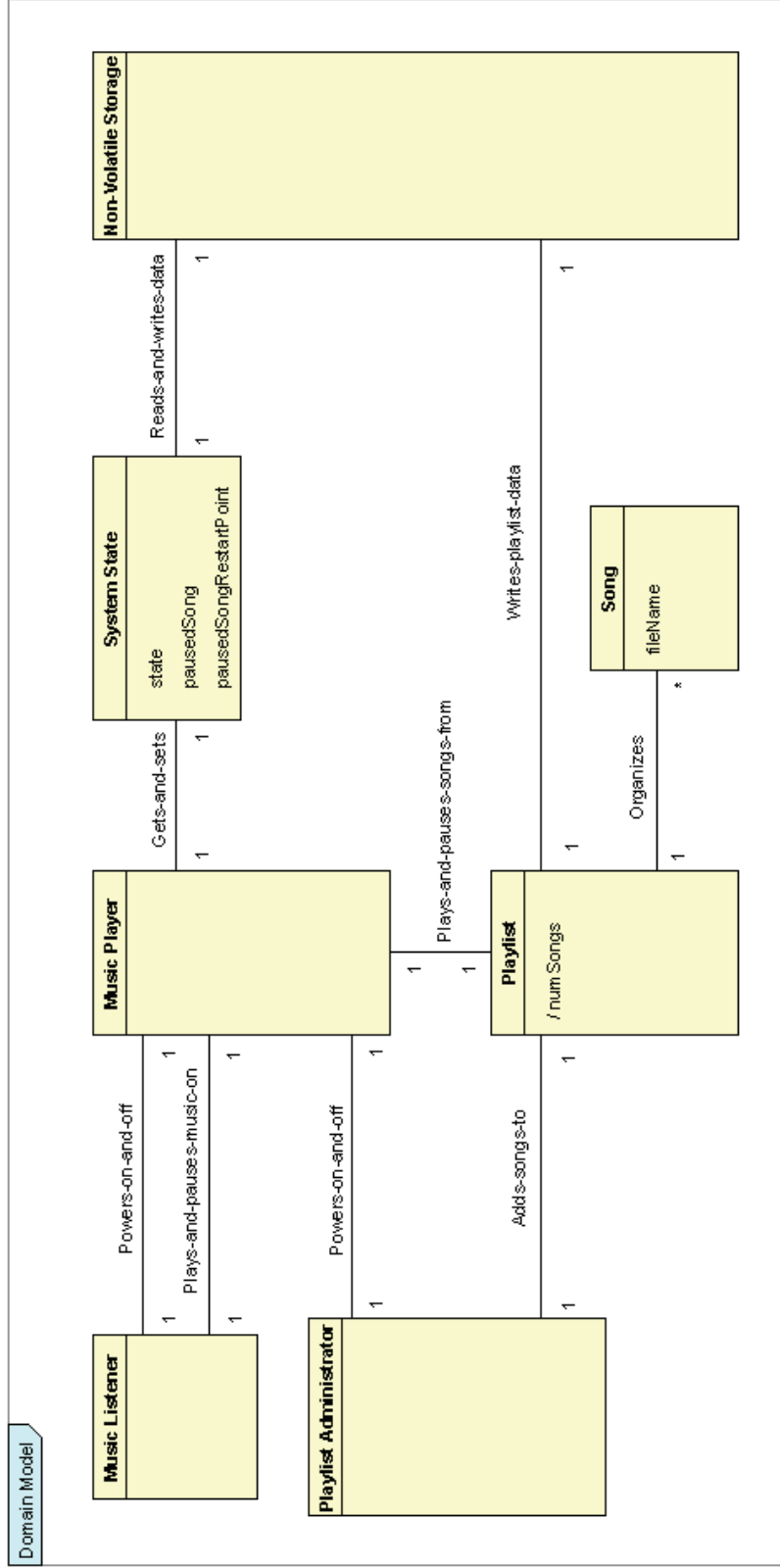


Figure 4.24 - Elaboration Phase 2 Domain Model

4.4.3. Design

The team then created SSDs and operation contracts from the use cases and the Domain Model, and they used these artifacts to create the sequence diagrams and update the class diagram. Since this phase included very complex use cases, the sequence diagrams were also complex, and many new classes and associations were added to the class diagram. The team was able to extend the existing architecture to accommodate the new functionality, and no refactoring was necessary.

Figure 4.25 shows the structure of the class model created for elaboration phase 2.

Most of the interesting work in this phase involved the interactions between the Tanager objects and the JMF. Tanager objects like the `CMusicPlayer` and `CSong` needed to instantiate and manipulate objects in the JMF, such as `Manager` and `Player`. The development team spent quite a bit of time analyzing the JMF to understand which objects would be needed and how to interact with those objects. The JMF objects are not part of the domain, so they are not shown in the Domain Model or Class Model, but they are shown in the sequence diagrams, since the Tanager objects interact with them. Figure 4.26 shows a sequence diagram where a `CSong` object interacts with the JMF `Manager` and `Player` objects.

4.4.4. Implementation

The class model was updated in Poseidon as each sequence diagram was created and refined. After all the analysis and design activities were complete, the class model was generated from Poseidon, and the code stubs were filled out according to their semantics. The team added the JMF package to the Eclipse development environment, so they could build their code using the JMF objects and interfaces. Luckily, JMF behaved like its documentation said it would, and the interaction between the Tanager objects and the JMF objects was as the team expected.

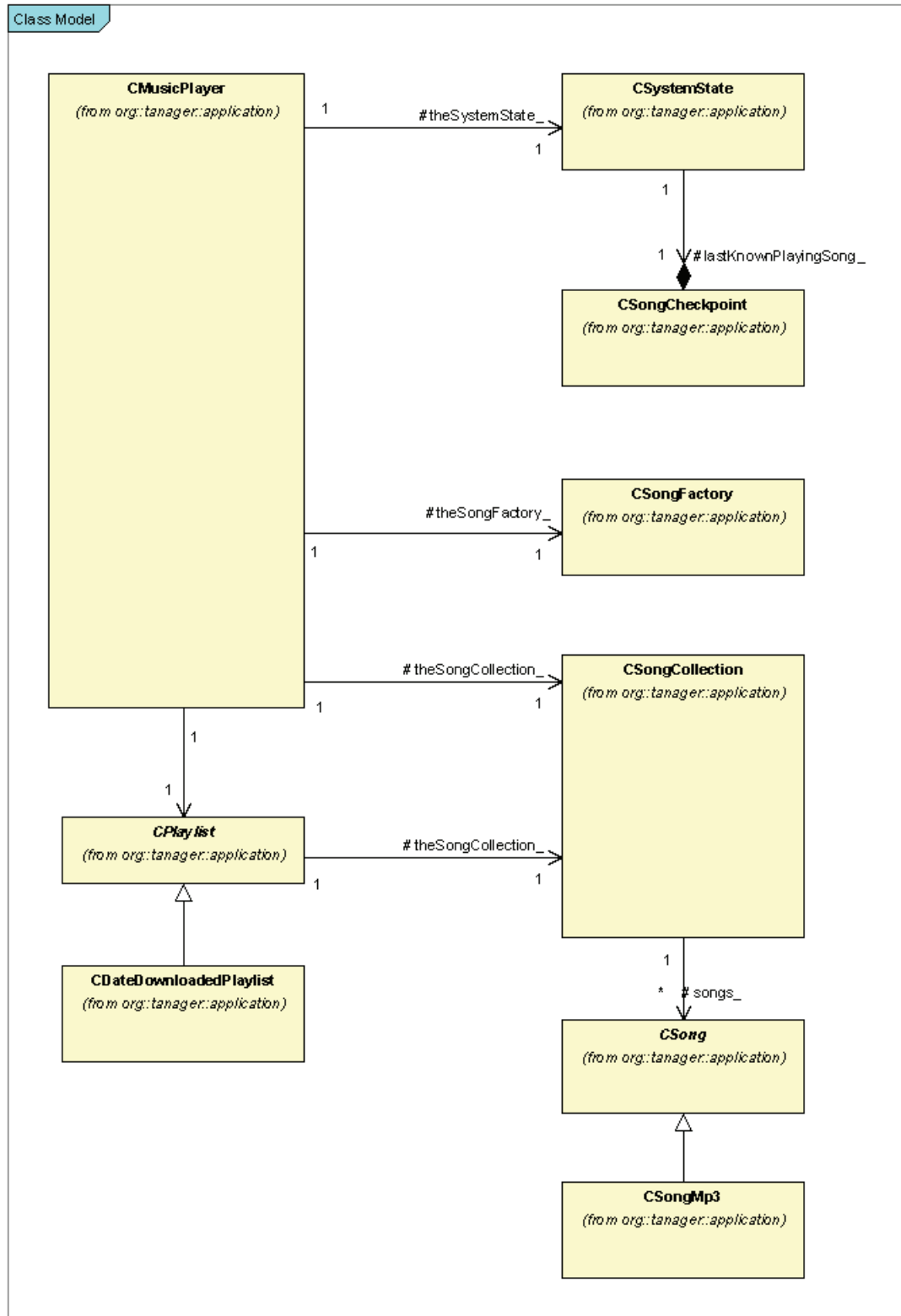


Figure 4.25 - Elaboration Phase 2 Class Model

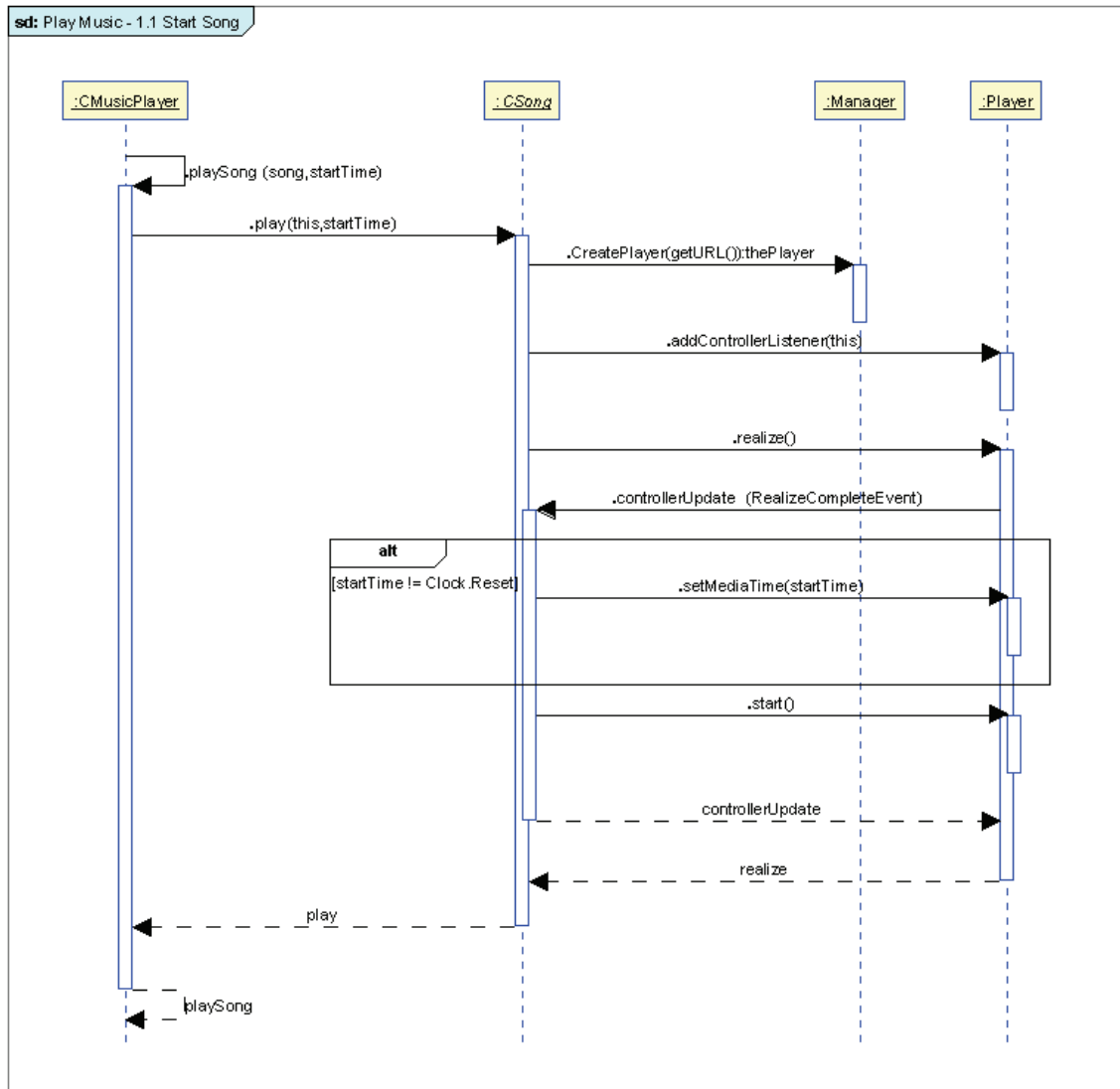


Figure 4.26 - Start Song Sequence Diagram

4.4.5. Testing

The development team added a single unit test to the regression suite during this phase. The new test exercised the system's ability to play a downloaded song. They verified that all of the regression unit tests executed successfully, and then they performed exploratory testing using the use case definitions. When they had fixed the defects found during testing, they met with the stakeholders to demonstrate the results of elaboration phase 2 and to solicit feedback. During this meeting, the stakeholders and development team realized they had missed an important use case: Stop Playing Music. The users indicated they would like to be able to stop playing music, and if playback was restarted, it would start at the beginning of the playlist. This use case hadn't come out in any of the discussions up to this point, but by having the stakeholders involved in the development process, something that might have been lost completely or missed until the end of the project was caught early enough to get it into the schedule. The team wrote the brief, casual, and fully-dressed use case formats at the end of elaboration phase 2, and scheduled the implementation of this newly-discovered use case for elaboration phase 3.

4.4.6. Results

The Tanager development team was able to implement the risky use cases of playing and pausing music, and wrote regression tests to exercise this functionality. They also discovered a new use case while working with the stakeholders at the end of the phase and were able schedule the work on this new use case.

4.4.7. Additional Documentation

The full set of artifacts and documentation produced by the Tanager development team during elaboration phase 2 can be found on their web site at

http://www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_02/ .

4.5. *Elaboration Phase 3*

4.5.1. Overview

Based on the schedule created in the inception phase and revised in elaboration phase 2, the plan for this phase is:

1. Implement the Stop Music use case.
2. Implement the View Playlist use case.
3. Implement the Delete a Song use case.
4. Implement the Volume Adjustments use case.
5. Write the fully-dressed format for the Select Playlist Order, Restart Current Song, and Skip to Next Song use cases (to be implemented in elaboration phase 4).

In the previous elaboration phase, the development team discovered a new use case, Stop Music, which needed to be implemented in this phase, since it used new functionality from the JMF. The team had decided that the riskiest use cases were those that used the JMF, so they wanted to handle them in the early elaboration phases. The remaining use cases to be handled in this phase either also used functionality from the JMF (Volume Adjustments) or had architectural implications (Delete a Song).

4.5.2. Analysis

As in previous phases, the development team analyzed the use cases to discover any new domain objects, attributes, or associations. The potential domain objects are shown in Figure 4.27 and the resolution of the potential objects is shown in Figure 4.28. The team found no new domain objects, but did identify some new attributes.

<p>Stop Music Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>user</i> tells the <i>system</i> to stop music playback. 2. The <i>system</i> stops playing. <p>View Playlist Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>Playlist Administrator</i> requests the <i>playlist</i> from the <i>system</i>. 2. The <i>system</i> returns a <i>list of all the downloaded songs</i> ordered by the <i>current playlist</i>. <p>Delete a Song Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>Playlist Administrator</i> retrieves a <i>list of all the songs in the current playlist</i> from the <i>system</i>. 2. The <i>Playlist Administrator</i> tells the <i>system</i> which <i>song</i> they want to delete from the <i>playlist</i>. 3. The <i>system</i> deletes the chosen <i>song</i> from the <i>playlist</i>, rebuilds the <i>playlist</i>, and saves the <i>system</i> state to <i>non-volatile memory</i>. <p>Extensions</p> <ol style="list-style-type: none"> 3a. If the <i>Playlist Administrator</i> chose the <i>currently-playing song</i>. <ol style="list-style-type: none"> 1. The <i>system</i> informs the <i>Playlist Administrator</i> that the <i>currently-playing song</i> cannot be deleted. 3b. If the <i>system</i> is <i>paused</i>, and the <i>Playlist Administrator</i> chose the <i>paused song</i>. <ol style="list-style-type: none"> 1. The <i>system</i> informs the <i>Playlist Administrator</i> that the <i>paused song</i> cannot be deleted. <p>Volume Adjustments Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>Music Listener</i> requests the <i>current volume setting</i> from the <i>system</i>. 2. The <i>Music Listener</i> modifies the <i>volume setting</i> and tells the <i>system</i> the <i>new volume setting</i>. 3. The <i>Tanager system</i> verifies the <i>volume setting</i> is between 0 and 20 and saves the <i>new volume level</i> to <i>non-volatile memory</i>. <p>Extensions</p> <ol style="list-style-type: none"> 3a. If the <i>new volume setting</i> is greater than 20. <ol style="list-style-type: none"> 1. The <i>system</i> snaps the <i>volume setting</i> to 20. 3b. If the <i>new volume setting</i> is less than 0. <ol style="list-style-type: none"> 1. The <i>system</i> snaps the <i>volume setting</i> to 0.
--

Figure 4.27 - Elaboration Phase 3 Potential Domain Objects

Potential Domain Object	Result
<i>list of all the downloaded songs</i>	domain object: Playlist
<i>list of all the songs in the current playlist</i>	domain object: Playlist
<i>currently-playing song</i>	domain object: State (“currently-playing” is an attribute)
<i>current volume setting</i> <i>current volume level</i>	domain object: State (“volume” is an attribute)

Figure 4.28 - Resolution of Potential Domain Objects

Figure 4.29 shows the use cases with the domain objects and potential domain object associations highlighted, and Figure 4.30 shows how the potential domain object associations were resolved. Using the domain objects, associations, and attributes, the team generated the diagram shown in Figure 4.31 and captured that in the Domain Model.



Figure 4.29 - Elaboration Phase 3 Potential Domain Object Associations

Potential Domain Object Association	Result
<i>user tells system to stop music</i>	Music Listener Stops-music-on Music Player
<i>Playlist Administrator requests the playlist from the system</i>	Playlist Administrator Gets-playlist-from Playlist
<i>Playlist Administrator retrieves the playlist from the system</i>	Playlist Administrator Gets-playlist-from Playlist
<i>Playlist Administrator tells the system the song they want to delete</i>	Playlist Administrator Deletes-songs-from Playlist
<i>the system rebuilds the playlist</i>	Not an association - this is the playlist rebuilding itself after a song is deleted, and is not shown as an association
<i>the system informs the Playlist Administrator that the currently-playing song cannot be deleted</i>	Not an association - this is a return parameter to the delete song operation
<i>the system informs the Playlist Administrator that the paused song cannot be deleted</i>	Not an association - this is a return parameter to the delete song operation
<i>Music Listener requests the current volume setting from the system</i>	Music Listener Gets-volume-from Music Player
<i>Music Listener modifies the volume setting</i>	Not an association - this is an internal UI operation
<i>Music Listener tells the system the new volume setting</i>	Music Listener Sets-volume-in Music Player
<i>Tanager system verifies the volume setting</i>	Not an association - this is an internal operation

Figure 4.30 - Potential Domain Object Association Resolution

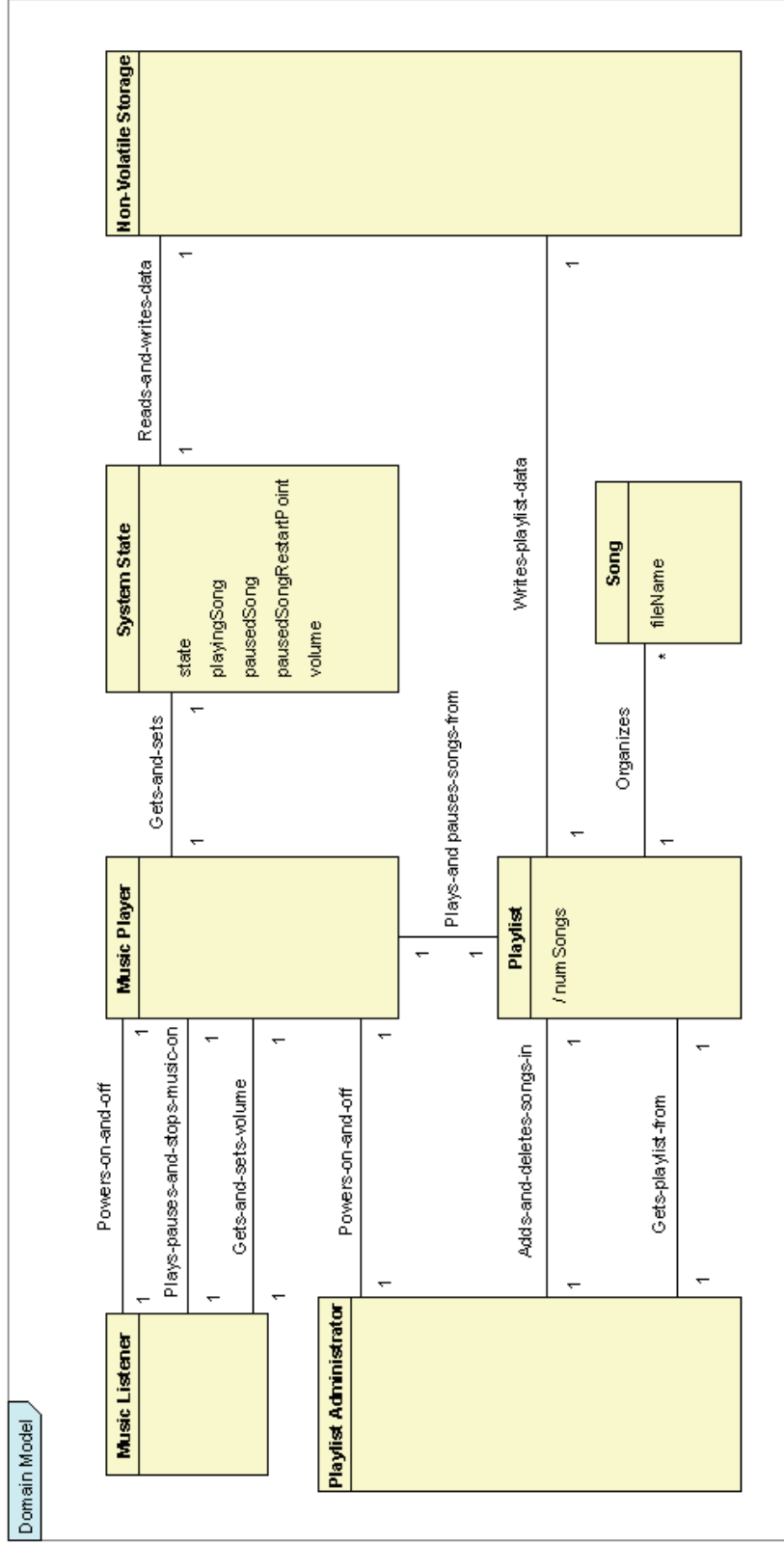


Figure 4.31 - Elaboration Phase 3 Domain Model

4.5.3. Design

During this phase, the development team realized that even though they were devoting as much time to developing the UI layer as they were to developing the application layer, they had not been as rigorous about applying an OOAD process to the UI development. They had been drawing scenario diagrams and had been capturing the UI classes in the Poseidon class model, but they had not been applying the same OOAD process as they had been using for the application layer. Their team barely had the resources to get the application layer designed and implemented, but they knew they had to do a better job on the UI. They decided that they did not have the resources to do a full-blown OOAD process for the UI and still meet their schedule, but they would try to spend more time practicing good processes.

The event that triggered this realization in the team was that they needed to refactor the menu system. They had initially implemented the menu as part of the Download a Song use case, but it had been implemented in a very simple way as a popup menu. User feedback, however, told them that it would be better if the menus were displayed in the main display screen of the Tanager UI rather than as a popup. Based on this user feedback, and since there were several use cases to be designed and implemented in this phase that would be using the menus, the team decided that they would refactor it to make it fit in the main UI window and to be more extensible. The menu's analysis and design had previously been driven by application-specific use cases, but two new UI use cases, Enter Menus and Exit Menus, were created. These new use cases were added to the Use Case Model in a new UI section. Rudimentary analysis was done on these use cases, sequence diagrams were generated, and the class model was updated. Figure 4.32 shows a simplified portion of the UI class diagram containing the refactored classes. The `CScrollablePane` abstract class provides the base functionality for handling the control buttons and mouse double-clicks, and classes with this base type can be displayed in the UI's main window. A `CMenu` abstract class is derived from `CScrollablePane`, and `CMainMenu` is derived from `CMenu`. `CMainMenu` provides the Tanager UI's menu functionality, and since it's derived from `CScrollablePane` it can be displayed in the UI's main window. Similarly, the `CPlaylistViewer` class is used by the View Playlist use case and the `CSelectablePlaylistViewer` class is used by the Delete a Song use case, and both of them can be displayed in the UI's main window.

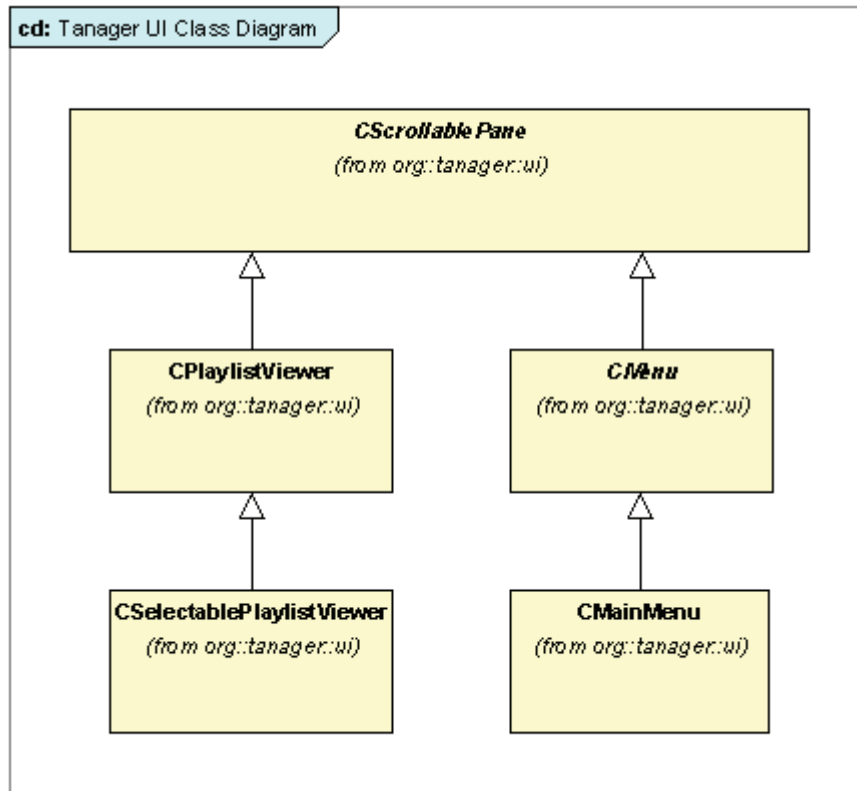


Figure 4.32 - Tanager UI ScrollablePane and Derived Classes

With the addition of the Stop Music use case, the design of the UI layer became complex enough to warrant the use of a formal state machine. Figure 4.33 shows the UI state machine diagram after all the UI design work from elaboration phase 3 was complete. The development team had implemented the application layer's state machine as simple switch/case statements in those methods that required state handling, but that design would be inadequate for the complexities of the UI state machine. The team decided that using the State Pattern was a better solution for the UI state machine, since it allowed each state to handle only those events that they were interested in, and the responsibilities and code turmoil were contained to each state's implementation class.

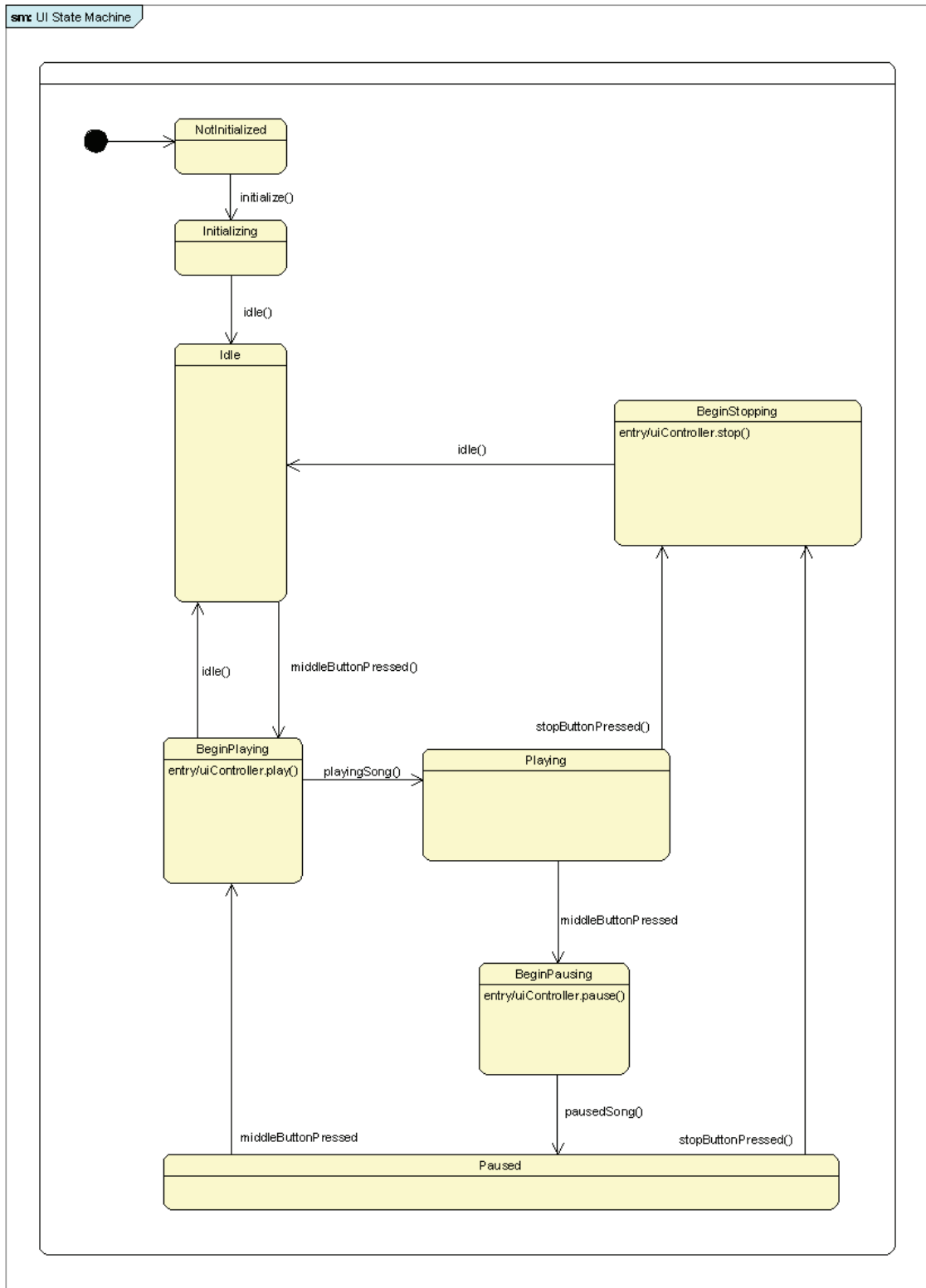


Figure 4.33 - Tanager UI State Machine

Figure 4.34 shows the UI code that executed when the user pressed the middle button (the play/pause button) before the team applied more rigorous design efforts on the UI. The sequence diagram in Figure 4.35 shows how the UI code handles a middle button press using the State Pattern.

```
public void playPauseButtonPressed() {  
    Icon playPauseIcon = middleButton_.getIcon();  
    if (playPauseIcon == playIcon_)  
    {  
        middleButton_.setIcon(null);  
        theTanagerUI_.play();  
    }  
    else if (playPauseIcon == pauseIcon_)  
    {  
        middleButton_.setIcon(null);  
        theTanagerUI_.pause();  
    }  
}
```

Figure 4.34 - Old Method of UI Play/Pause Button Handling

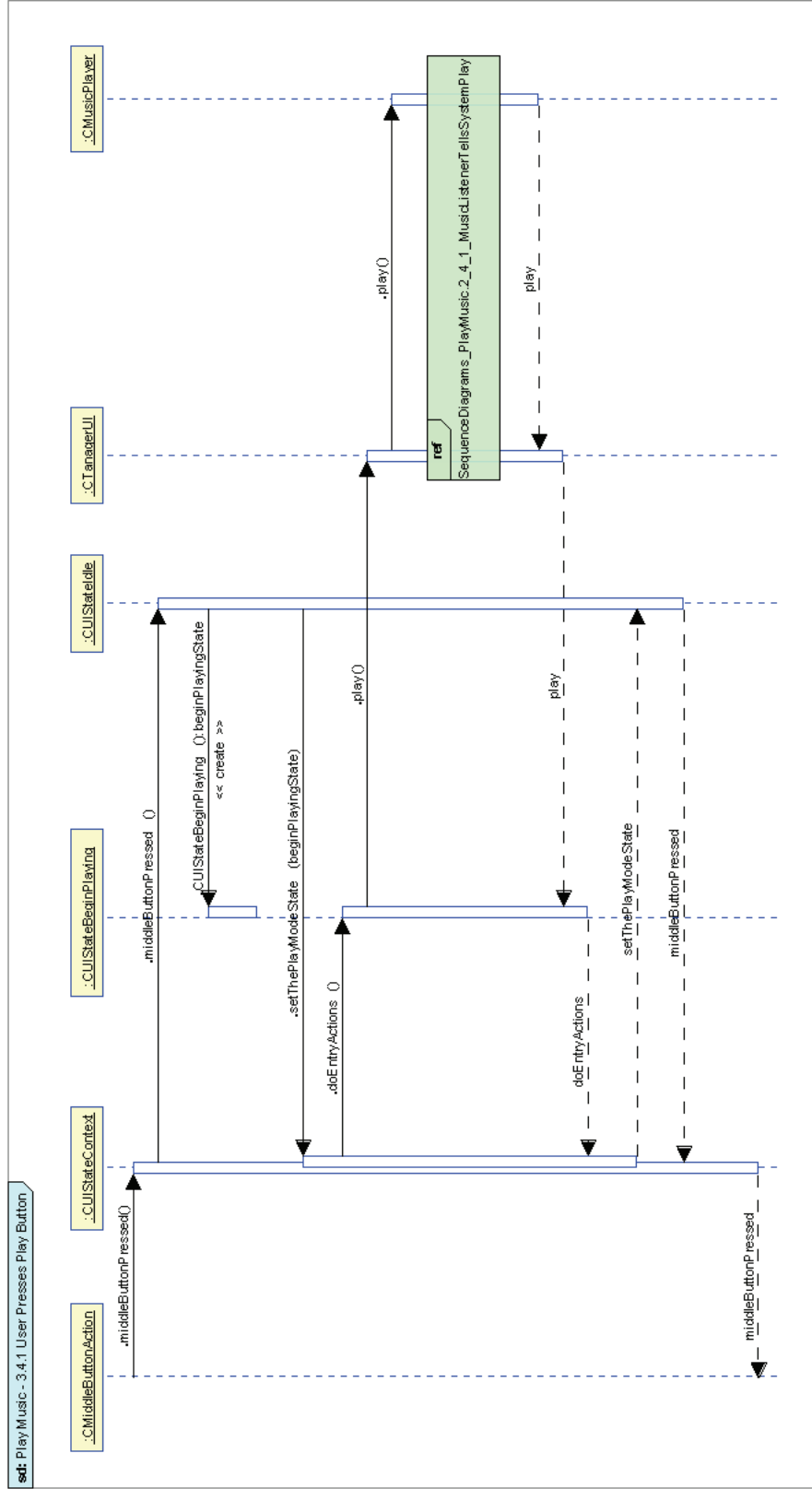


Figure 4.35 - User Presses Play Button Sequence Diagram

4.5.4. Implementation

As with the previous elaboration phases, the class model was updated as the design progressed, and when the design step was complete, the development team generated the class model from Poseidon. All of the refactoring in the design phase meant that the team had to be very careful to update the semantics documentation of those operations that changed, and they had to keep track of which operations needed rework during implementation. CVS was a great help during this process, since the team could use that tool to see which operations had their semantics documentation changed since elaboration phase 2. This provided a list of operations whose implementation needed to be updated to match its new semantics.

4.5.5. Testing

A new unit test was added during this phase to exercise the system's ability to delete downloaded songs. This unit test was added to the regression suite, and the development team made sure that the whole regression suite passed before moving on to exploratory testing. As was expected, due to the large amount of code turmoil, the team found several defects. None were serious enough to require re-design, however.

After the defects found during regression testing had been fixed, the team moved on to exploratory testing of the View Playlist, Stop Music, Delete a Song, and Volume Adjustments use cases. All worked as expected, but the team found it time-consuming to download multiple songs from the same directory. Each time they selected the menu item to download a song, the UI displayed a file chooser that started at their home directory. Since the test files were in a directory many levels below their home directory, it was a cumbersome task to navigate to the test file directory. The team realized that users would face this same problem, so they decided to redesign the sequence diagrams to save and retrieve a "last known downloaded song path" to use when displaying the file chooser to the user. Figure 4.36 shows the UI's Download a Song sequence diagram. The only changes that were necessary for this change are highlighted in red. The team then met with the stakeholders and demonstrated the results from elaboration phase 3. The stakeholder's feedback was positive, and no new requirements were generated.

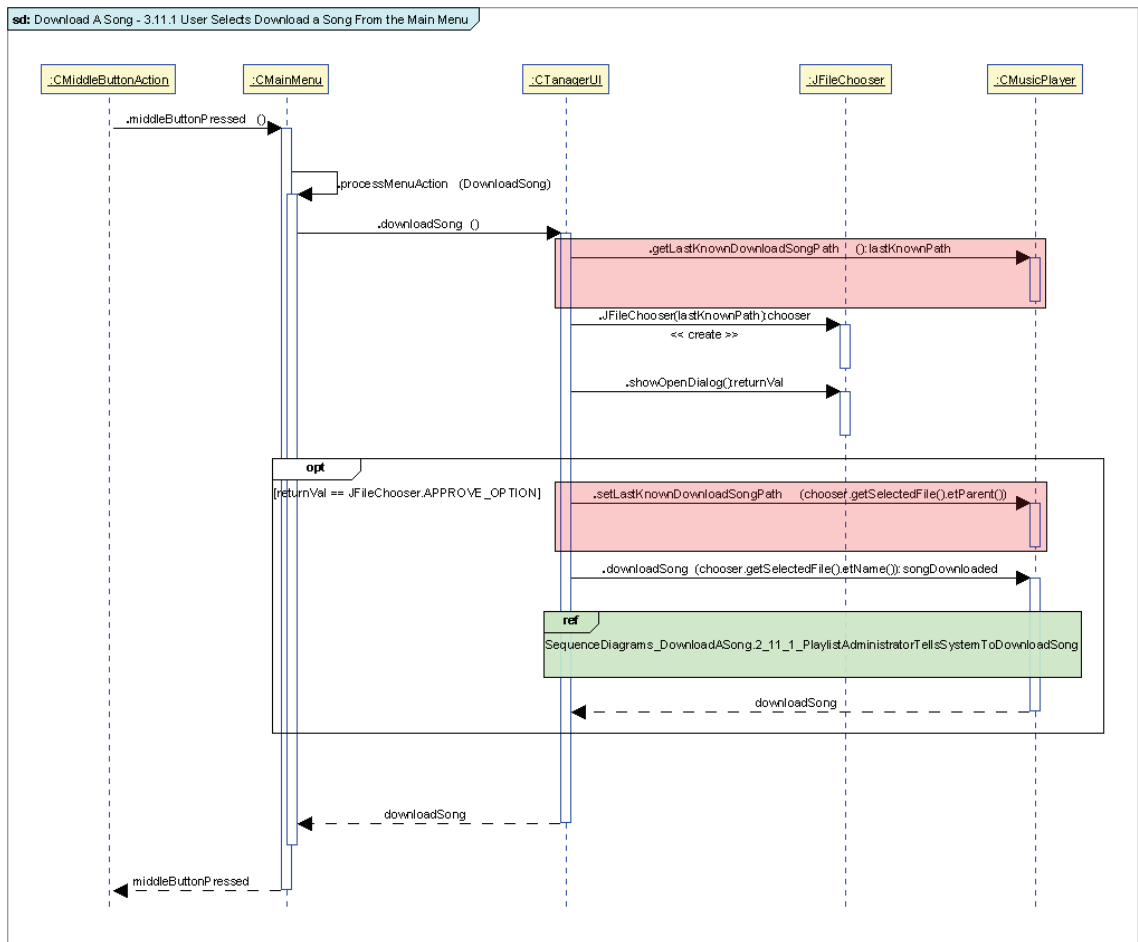


Figure 4.36 - Modified Download a Song Sequence Diagram

4.5.6. Results

Much of the work in this elaboration phase involved the UI layer rather than the application layer. Ideally, the UI would be handled as a separate project, but there weren't enough resources on the development team to allow that. Even though the Tanager project was to focus on the application layer, the same team was implementing the application and UI layers, and many UI concepts were working their way into the application layer's analysis and design documentation. During this phase, a concentrated effort was made to separate the UI and application artifacts. All of the UI analysis and design artifacts were captured in separate sections of the Tanager documentation, even though they weren't officially part of the Tanager project.

4.5.7. Additional Documentation

The full set of artifacts and documentation produced by the Tanager development team during elaboration phase 3 can be found on their web site at http://www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_03/.

4.6. *Elaboration Phase 4*

4.6.1. Overview

Based on the schedule created in the inception phase, the plan for this phase is:

1. Implement the Select Playlist Order use case.
2. Implement the Restart Current Song use case.
3. Implement the Skip to Next Song use case.
4. Implement the Skip to Previous Song use case.

The development team faced a lot of design and implementation in this phase, but since this was the final elaboration phase, there were no more use cases to analyze for subsequent phases.

4.6.2. Analysis

For this final elaboration phase, the development team began by analyzing the use cases to discover any new domain objects, attributes, or associations. Figure 4.37 shows the potential domain objects, and Figure 4.38 shows the resolution of the potential objects. The development team found no new domain objects, and only identified one new attribute. This was expected by the team, since they attacked those use cases that had the greatest architectural coverage in the earliest elaboration phases. Figure 4.39 shows the use cases with the domain objects and potential domain object associations highlighted, and Figure 4.40 shows how the development team resolved the potential domain object associations. Using the domain objects, associations, and attributes, the team generated the diagram shown in Figure 4.41 and captured that in the Domain Model.

<p>Select Playlist Order Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>user</i> tells the <i>system</i> to rebuild the <i>playlist</i> using the <i>specified ordering</i>. 2. The <i>system</i> rebuilds the <i>playlist</i> in <i>specified order</i>. 3. The <i>system</i> saves its <i>playlist</i> to <i>non-volatile memory</i>. <p>Restart Current Song Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>Music Listener</i> requests that the <i>system</i> restart the <i>current song</i>. 2. The <i>Tanager system</i> restarts the <i>current song</i> from the beginning. <p>Skip to Next Song Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>Music Listener</i> requests that the <i>system</i> skip to the <i>next song</i>. 2. The <i>Tanager system</i> stops playing the <i>current song</i>. 3. The <i>system</i> retrieves the <i>next song</i> in the <i>playlist</i> and begins playing it. <p>Skip to Previous Song Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>Music Listener</i> requests that the <i>system</i> skip to the <i>previous song</i>. 2. The <i>Tanager system</i> stops playing the <i>current song</i>. 3. The <i>system</i> retrieves the <i>previous song</i> in the <i>playlist</i> and begins playing it.

Figure 4.37 - Elaboration Phase 4 Potential Domain Objects

Potential Domain Object	Result
<i>specified ordering</i>	domain object: Playlist (“ordering” is an attribute)
<i>next song</i>	domain object: Song
<i>previous song</i>	domain object: Song

Figure 4.38 - Resolution of Potential Domain Objects

<p>Select Playlist Order Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>user</i> tells the <i>system</i> to rebuild the <i>playlist</i> using the <i>specified ordering</i>. 2. The <i>system</i> rebuilds the <i>playlist</i> in <i>specified order</i>. 3. The <i>system</i> saves its <i>playlist</i> to <i>non-volatile memory</i>. <p>Restart Current Song Main Success Scenario</p> <ol style="list-style-type: none"> 1. The <i>Music Listener</i> requests that the <i>system</i> restart the <i>current song</i>. 2. The <i>Tanager system</i> restarts the <i>current song</i> from the beginning. <p>Skip to Next Song Main Success Scenario</p> <ol style="list-style-type: none"> 3. The <i>Music Listener</i> requests that the <i>system</i> skip to the <i>next song</i>. 4. The <i>Tanager system</i> stops playing the <i>current song</i>. 5. The <i>system</i> retrieves the <i>next song</i> in the <i>playlist</i> and begins playing it. <p>Skip to Previous Song Main Success Scenario</p> <ol style="list-style-type: none"> 6. The <i>Music Listener</i> requests that the <i>system</i> skip to the <i>previous song</i>. 7. The <i>Tanager system</i> stops playing the <i>current song</i>. 8. The <i>system</i> retrieves the <i>previous song</i> in the <i>playlist</i> and begins playing it.
--

Figure 4.39 - Elaboration Phase 4 Potential Domain Object Associations

Potential Domain Object Association	Result
<i>user tells system to rebuild the playlist using the specified ordering</i>	Playlist Administrator Sets-ordering-of Playlist
<i>the system rebuilds the playlist</i>	Not an association - this is the playlist rebuilding itself after a new ordering is selected, and is not shown as an association
<i>Music Listener requests that the system restart the current song</i>	Music Listener Restarts-songs-on Music Player
<i>system restarts the current song</i>	Music Player Restarts-songs-from Playlist
<i>Music Listener requests that the system skip to the next song</i>	Music Listener Skips-songs-on Music Player
<i>system stops the current song</i>	Music Player Stops-songs-from Playlist
<i>system retrieves the next song from the playlist and begins playing it</i>	Music Player Plays-songs-from Playlist
<i>Music Listener requests that the system skip to the previous song</i>	Music Listener Skips-songs-on Music Player
<i>system stops the current song</i>	Music Player Stops-songs-from Playlist
<i>system retrieves the previous song from the playlist and begins playing it</i>	Music Player Plays-songs-from Playlist

Figure 4.40 - Potential Domain Object Association Resolution

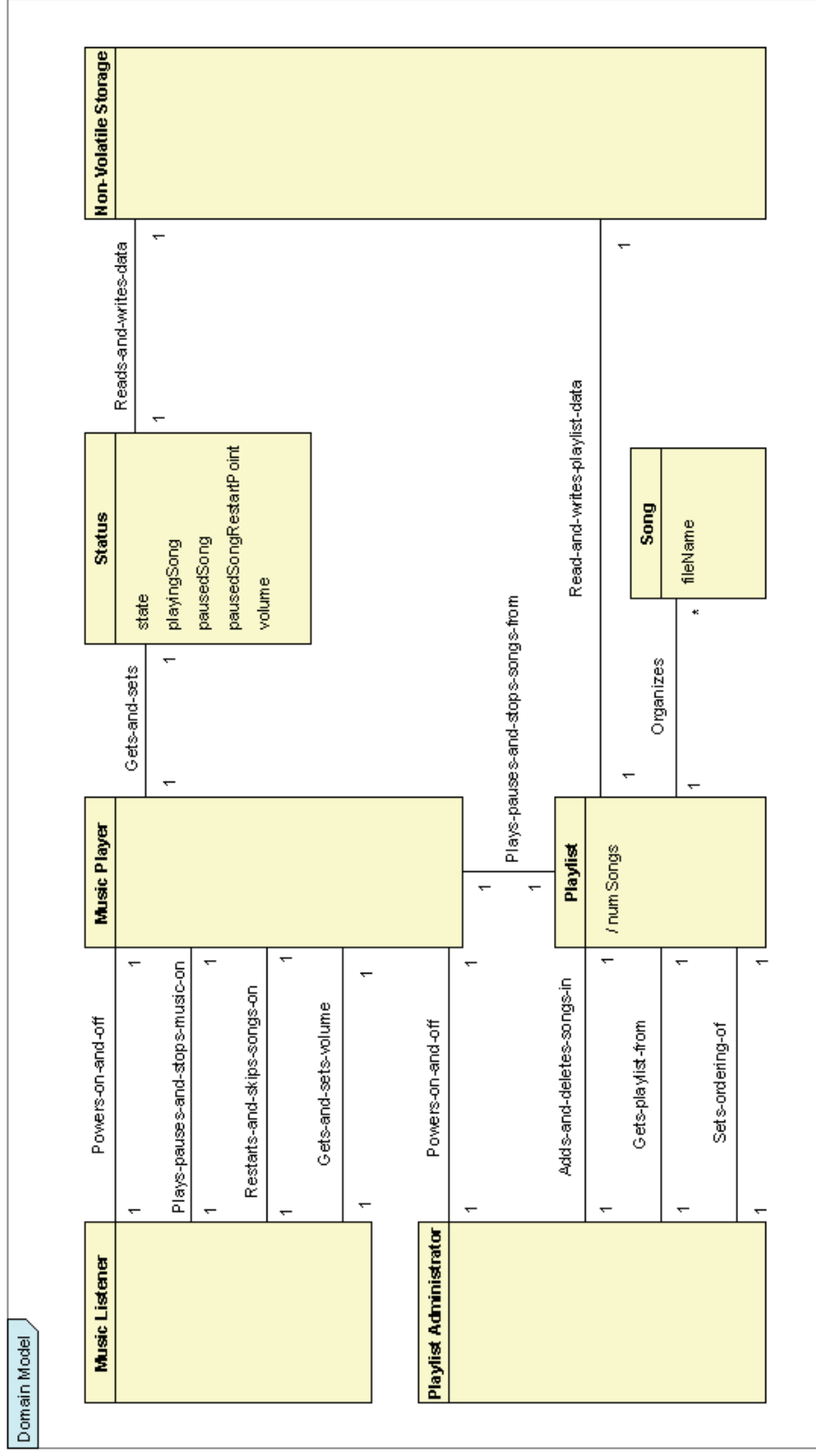


Figure 4.41 - Elaboration Phase 4 Domain Model

4.6.3. Design

The majority of the use cases that were handled in this phase used application layer design elements from previous phases: stopping songs, selecting songs, and playing songs. Figure 4.42 for example, shows a portion of the Skip to Next Song sequence diagram. Each of the green boxes references a sequence diagram that was designed in a previous elaboration phase and was reused in this design.

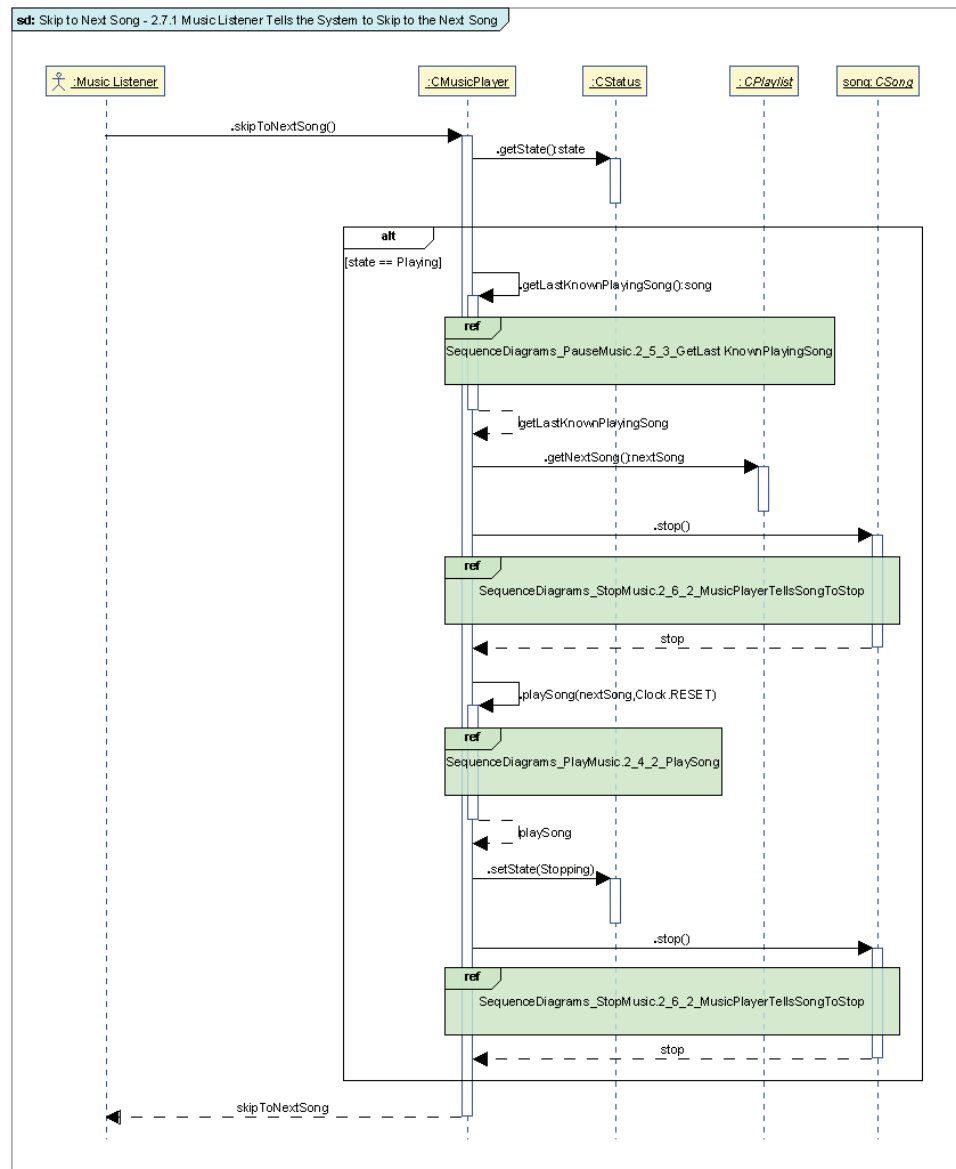


Figure 4.42 - Skip to Next Song Sequence Diagram

The only architecturally interesting application layer use case for this phase was Select Playlist Order. All previous phases used the default `CDateDownloadedPlaylist`, which is derived from the abstract `CPlaylist` class. For this phase, the development team derived a new class, `CRandomPlaylist`, from `CPlaylist` and designed a way for the `CMusicPlayer` to switch between the different playlist subclasses. Figure 4.43 shows the portion of the Class Model that includes the playlist classes. Figure 4.44 and Figure 4.45 show the sequence diagrams the team designed for the Select Playlist Order use case. As the sequence diagrams for this phase were developed, the team updated the class model and especially the operation semantics documentation.

4.6.4. Implementation

Once the design step was complete, the team generated the class model from the Poseidon tool and implemented the new classes and their operations and attributes, and they updated the existing classes, operations, and attributes as necessary.

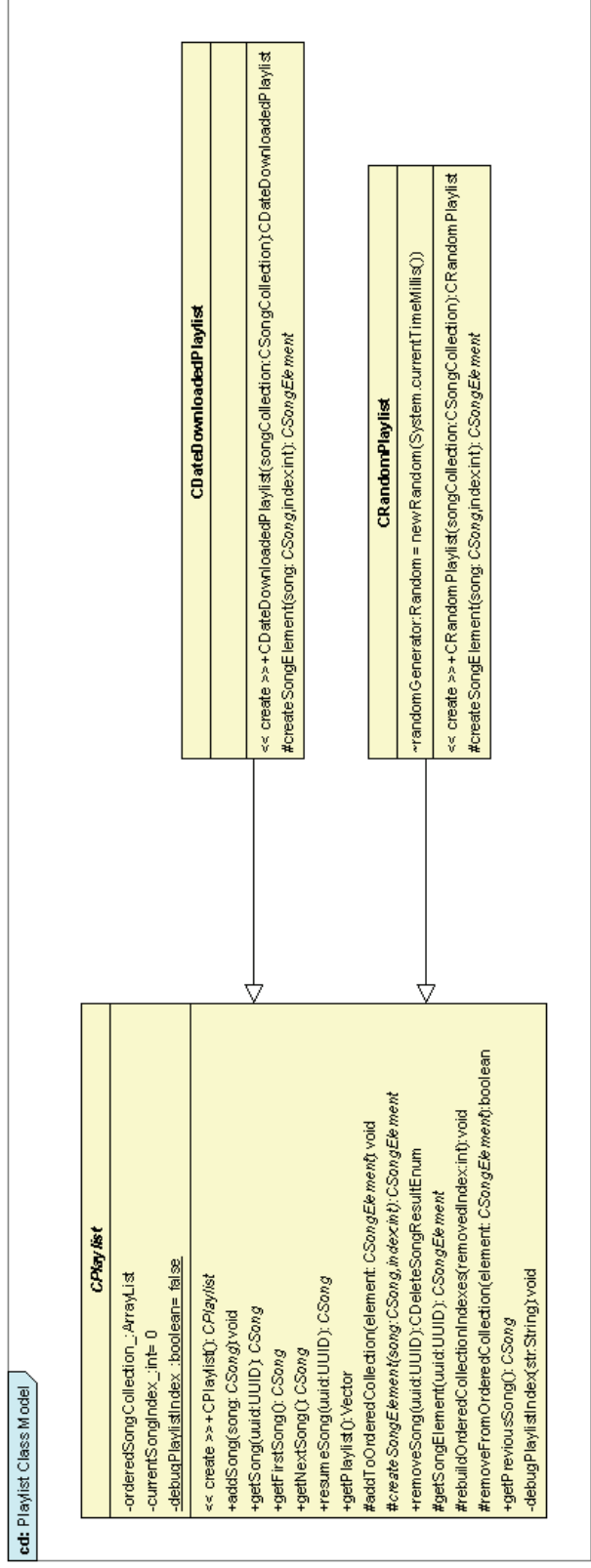


Figure 4.43 - Playlist Class Diagram

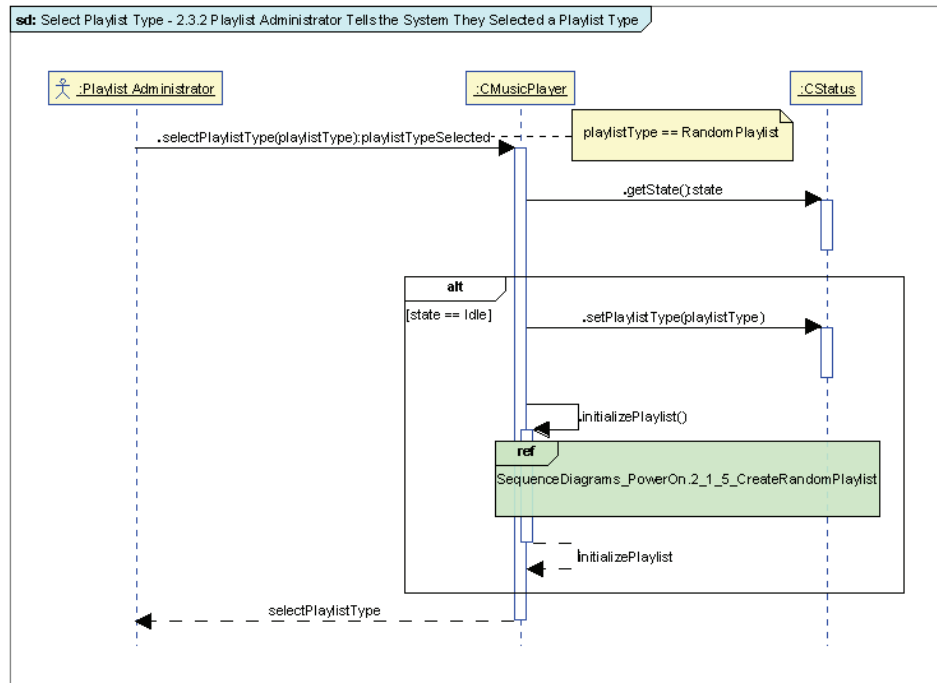


Figure 4.44 - Playlist Administrator Selects Playlist Ordering Sequence Diagram

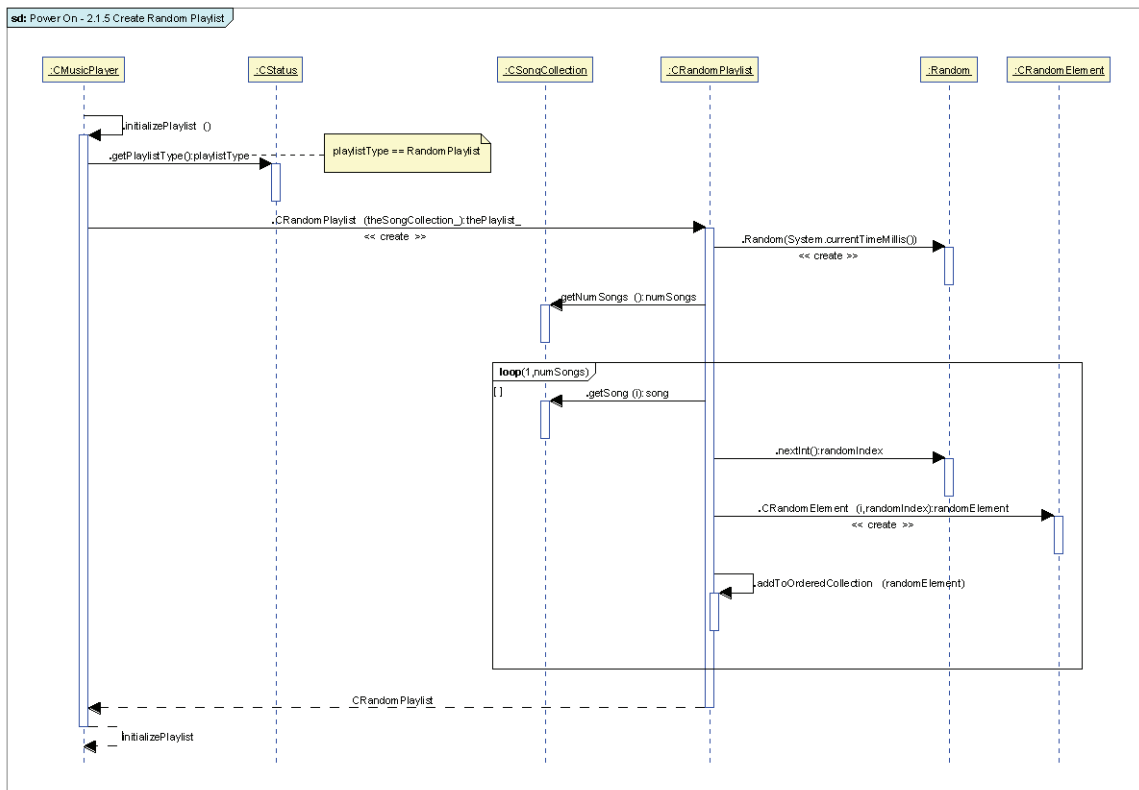


Figure 4.45 - Create Random Playlist Sequence Diagram

4.6.5. Testing

No new unit tests were implemented in this phase, since most of the implementation used existing functionality. The team ran the regression test suite, and after they fixed the minor defects that they found, they began performing exploratory testing on the use cases implemented in this phase. During the exploratory testing of the Restart Current Song use case, the team found a defect where pressing the left button while in the menus and playing a song caused the menus to be exited *and* the song to be restarted. While in the menus, the left button should jump to the previous menu, or it should exit the menus if the top level menu is displayed; it should not affect the play song. The team analyzed the failure and discovered the button handlers for the playing song screen were not removed while the menus were being displayed. This stemmed from their menu handling re-design in elaboration phase 3 (see Section 4.5.3 above). The button handlers for the menus were added, but the button handlers for the playing song screen were not removed. This caused the button press actions to be handled by the menu and the playing song screen. The team redesigned the playing song screen object and added the ability to add and remove the button handlers as necessary.

Once the redesign was complete, all the tests from the regression suite passed, and the team had completed exploratory testing, they scheduled a meeting with the stakeholders to demonstrate the results. While reviewing the final implementation, the stakeholders found a defect that had been missed by the team's exploratory testing. The defect occurred when the user pressed the menu button, selected View Playlist from the main menu, and then pressed the menu button again. The expected behavior was that the user would exit the menus and be able to interact with Tanager normally. However, the defect caused the system to not respond to any key presses after the user exited the menus. The user could not enter the menus by pressing the menu button, and they could not play music by pressing the Play button. The development team investigated and discovered that the menu button was not removing its button handler at the correct time. As a result, the menu button handler would be added multiple times and never removed, which caused menu button events to be handled multiple times. The team added code to remove the menu button handlers at the appropriate time, and that fixed the defect. The stakeholders were happy with the final implementation of the code, and they and the team celebrated until the wee hours of the morning.

4.6.6. Results

During this phase, the development team implemented the last of the use cases. As they expected, much of their design and implementation reused elements that had been completed in previous phases.

When the team performed their project-end review, however, they found that they had not been maintaining accurate user documentation throughout the project. Since the team had been working directly with the users when demonstrating the Tanager music player, the lack of user documentation was not an issue. However, once the development process was over, the users were left without any accurate documentation. As a result, the team had to go back through each of the previous phases and add user documentation that was accurate for that phase. The team decided that they needed to add updated user documentation as one of the goals for the end of each elaboration phase.

4.6.7. Additional Documentation

Additional samples of the documentation produced by the Tanager development team during elaboration phase 4 can be found in the appendices. The full set of artifacts and documentation can be found on the Tanager web site at

http://www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_04/ .

5. Advantages and Disadvantages of an Iterative OOAD Process

5.1. *Disadvantages of Iterative OOAD Process*

The main disadvantage to an iterative OOAD process is the amount of training that may be necessary. All developers on the team will have to understand the iterative process they will be using. This may be as simple as a few informal sessions with one of the team members leading the discussions, or it could be an intensive multi-day training session with a training consultant. Either way, the team members will need to understand the steps involved in the process and the motivation behind them. Additionally, there may be training requirements for UML or OOAD tools. Good team communication will require that all members are speaking the same notational language and that they understand the same meaning for these notations.

Another disadvantage for team members who are not used to a formal design process is the amount of documentation that is captured during each phase. Without studying the process and perhaps even using it for a while, it is not apparent how the documentation can make the analysis, design, implementation, and testing efforts easier.

Developers who have been successful using a waterfall methodology will have slightly different challenges. Switching their thinking from a waterfall methodology to an iterative one may be even more difficult than training team members with no experience in any methodology. Since they have been successful with the waterfall techniques, it may not be apparent to them how they will benefit from learning a new methodology. Further, they may become frustrated when they have to refactor their design after finding that a simple design used in an early elaboration phase is inadequate for the complex scenarios in a later phase. In their old waterfall methodology, they would have seen that the more complex design was required before implementing the simple design, and they would not have needed to refactor. For example, the Tanager development team initially used a very simple mechanism for handling the UI state machine, but that had to be redesigned using the State Pattern in elaboration phase 3. If they had done all of their analysis up-front, they may have anticipated that need and not wasted energy on a solution that would not work for all the use cases.

5.2. Advantages of Iterative OOAD Process

The main advantages of an iterative development process are that production-ready code is implemented early, and the stakeholders are involved early. The main success scenarios of the use cases with the most architectural coverage are analyzed, designed, and implemented first. This implementation is then demonstrated for the stakeholders, and the stakeholders provide feedback. Any misunderstandings between the stakeholders and developers or any changes to the stakeholder's wishes can be addressed before architectural complexities make it more difficult to change the design.

By having production-ready code implemented early, the interactions with collaborating teams are tested early. These collaborating teams may be separate testing teams or teams that are implementing collaborating functionality. For example, if the Tanager project had separate teams implementing the application and UI layers, the teams could schedule their elaboration phases to implement matching functionality and integrate their efforts at the end of each phase. Problems in the integration effort would be found early enough to correct with minimal effort. Along those same lines, a separate testing team could schedule their test implementation in coordination with the elaboration phase schedule, and they would be ready to test the code early. Code defects that are found early are much less expensive to fix than those found late in a project, so testing early is important.

Iterative development allows the team to concentrate on a subset of the problem rather than having to think about all possible aspects of the system. One advantage to concentrating on a subset of the user requirements is the team is less likely to miss details in those requirements, and they are less likely to make mistakes during the process. Also, by focusing on a subset of the use cases during each phase, the development team is less likely to over-design a solution. For example, the Tanager development team used a simple switch/case statement mechanism for the application layer state machine, which proved to be adequate for all of the use cases, even though the state machine grew with each phase. If they had analyzed all of the use cases in the beginning and designed a solution based on that analysis, they may have wound up wasting precious time over-designing a solution that wasn't necessary. This just-in-time development means that simple designs are used whenever possible, and they are only replaced with more complex designs when it becomes necessary.

Another advantage of focusing on a subset of the problem is that the team is less likely to miss details in the user requirements or make mistakes implementing them. Since the team does not have to think about the whole set of user requirements, their full concentration will be on the details of the subset of use cases for any given phase.

The documentation produced during the iterative development allows collaborating teams and test teams to understand the implementation at a high level and in detail, if necessary.

Collaborating teams that are producing UI and application code for the same product can use the Use Case Model, System Sequence Diagram Specification, and Operation Contract Specification to define the interface between the two layers. Both teams would only be interested in that high-level interface of their collaborator, since the collaborator is treated as a black-box object.

However, a test team that is implementing unit tests would be interested in the low-level details found in the Sequence Diagram Specification and Class Model. All of this documentation is available before any implementation begins in an elaboration phase, so the collaborators can also start on their work early.

6. Conclusions

The Tanager case study shows that using an iterative development process ensures that the user requirements are met, and that the resulting design is simple where it can be, yet complex where it needs to be.

This case study also shows that developers do not have to use the waterfall technique of analyzing and designing the whole system to be able to get reuse from the designs and implementation. The Tanager developers were able to reuse many parts of the design and code, even though they did not consider how the designs may be used in future phases. For example, they designed and implemented the functionality for playing, pausing, and stopping songs in elaboration phases 1 and 2, and they were able to reuse the designs and implementation in the Restart Current Song, Skip to Next Song and Skip to Previous Song use cases in elaboration phase 4.

The iterative development process is not difficult and can be learned and used by any developer. As the Tanager case study demonstrates, the resulting product meets the user's requirements, is well documented, and is extensible: all qualities of a well-designed system.

7. Appendix A – Analysis/Design Documentation

This appendix presents sample documentation from Tanager’s elaboration phase 4. The full documentation for Tanager can be found on the Tanager web site. The artifacts from each phase of Tanager’s development can be accessed separately:

Inception Phase: www.cs.iastate.edu/~rjlavey/Tanager/Inception/

Elaboration Phase 1: www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_01/

Elaboration Phase 2: www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_02/

Elaboration Phase 3: www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_03/

Elaboration Phase 4: www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_04/

7.1. Vision

1 Introduction

The vision for Tanager is that it will be a full-featured media player that will support multiple media types (AIFF, AU, AVI, MIDI, MP2, MP3, QT, RMF, WAV).

1.1 Purpose

The purpose of this document is to collect, analyze, and define high-level needs and features of the Tanager system. It focuses on the capabilities needed by the stakeholders and the target users, and why these needs exist. The details of how the Tanager system fulfills these needs are detailed in the use-case and supplementary specifications.

1.2 Scope

This document will summarize the high-level requirements of the Tanager system and the business case for the project. More detailed information can be found in the [Use Case Model](#) and [Supplementary Specification](#).

1.6 Overview

This document will summarize the goals and problems of the Tanager project at a high level. It will describe the product goals, the various stakeholders and users and their responsibilities, the features of the product, and the nonfunctional requirements of the product.

Figure 7.1 – Sample sections from Tanager Vision

2	Positioning
2.1	Business Opportunity
	While there are many media players on the market today, few of them support a large range of media types. Users that wish to play many different media types must use several different players, each of which has different feature sets and user interfaces. Users report that dealing with different feature support and multiple user interfaces for each product they use is confusing, and we believe it is unnecessary. The Tanager product will be our first step in providing software-based and hardware-based products that support a wide range of media types and have identical feature sets and user interfaces. The Tanager product will be a software-based media player, which plays a wide variety of media types, and it will serve as the base for a line of follow-on hardware-based products.
3	Stakeholder and User Descriptions
3.3	User Profiles
3.3.1	<i>Playlist Administrator</i>
3.3.2	<i>Music Listener</i>
4	Product Features
4.1	Download Songs
	The user will be able to download songs to the Player. The user will be able to download a wide variety of different formats without having to install additional components. They will also be able to delete songs that were previously downloaded.
4.2	View Downloaded Songs
	The user will be able to view a list of all the songs they have downloaded to their player.
4.3	Play Songs
	The user will be able to play the songs they have downloaded to their player. They will also be able to choose different ordering methods for the songs to play in: random ordering, alphabetical ordering, etc.
4.4	Manipulate Playing Songs
	The user will be able to pause a song, stop a song, skip over the current song to either the next song or the previous song, or restart the current song. They will also be able to change the volume of the playing song.

Figure 7.2 – Sample sections from Tanager Vision Sections

7.2. *Supplementary Specification*

<p>3. Usability</p> <p>3.1. Human Factors</p> <p>Since Tanager will be running on the user's computer and sharing desktop space with other programs, it should take up as little space as possible while still allowing the user to accurately press buttons and read the display.</p> <p>4. Reliability</p> <p>4.1. Accuracy</p> <p>It is of paramount importance that the user's music files are played accurately. There must be no noticeable skipping or stuttering while songs are playing.</p> <p>5. Performance</p> <p>5.3. Resource Utilization</p> <p>The Tanager system must not over-utilize the user's computer system resources and cause slow-downs with other executing programs.</p> <p>10. Free Open Source Components</p> <p>10.1. Java Media Framework</p> <p>Sun Microsystems' Java Media Framework will be used as the underlying audio media player for the Tanager system. The Java Media Framework is licensed under the JMF 2.1.1 Binary Code License: http://java.sun.com/products/java-media/jmf/2.1.1/license.html.</p> <p>11. Interfaces</p> <p>11.1. User Interfaces</p> <p>The user interface will consist of:</p> <ul style="list-style-type: none"> • an on/off button • an "up" button to be used for menu traversal • a "down" button to be used for menu traversal • an "up volume" button • a "down volume" button • a "forward" button for skipping to the next song • a "back" button for repeating the current song or skipping to the previous song • a "menu" button for accessing the menus • a "select" button for choosing menu items

Figure 7.3 – Sample sections from Tanager Supplementary Specification

7.3. Glossary

<p>A</p> <p>AAC — Advanced Audio Coding File Format</p> <p>A <u>lossy digital audio compression</u> scheme, which achieves better sound quality than <u>MP3</u> for the same file size.</p> <p>AIFF — Application Information File Format</p> <p>A file which contains the caption, icon, capabilities, and MIME priority support information for an application.</p> <p>AU</p> <p>Sun Microsystem's audio file format.</p> <p>Audible</p> <p>Audible.com's audio file format, used for audio books (see http://www.audible.com).</p> <p>AVI — Audio Video Interleaved</p> <p>A container format for video with synchronized audio. An AVI file can contain different compressed video and audio-streams.</p> <p>D</p> <p>Download</p> <p>The process of copying a song from one device to another. In the case of the Tanager Music Player, it is the process of copying a song to the Tanager device.</p> <p>DRM — Digital Rights Management</p> <p>Digital Rights Management is a web-based licensing system for audio and video files.</p> <p>J</p> <p>JMF — Java Media Framework</p> <p>The Java Media Framework API enables audio, video and other time-based media to be added to applications and applets built on Java technology.</p>

Figure 7.4 – Sample sections from Tanager Glossary

7.4. Use Case Model

2.4.	Play Music
2.4.1.	Fully- Dressed Format
2.4.1.1.	Brief Description This use case describes the user playing downloaded songs on the system.
2.4.1.2.	Scope The Tanager system.
2.4.1.3.	Level User-goal.
2.4.1.4.	Primary Actor Music Listener.
2.4.1.5.	Stakeholders and Interests
2.4.1.5.1.	Music Listener The Music Listener wants the system to play downloaded songs in the order dictated by the selected playlist.
2.4.1.6.	Preconditions The system has been previously booted up and one or more songs have been downloaded.
2.4.1.7.	Postconditions (Success Guarantee) The downloaded songs are playing in the order dictated by the currently-selected playlist.
2.4.1.8.	Main Success Scenario <ol style="list-style-type: none"> 1. The user tells the system to play music. 2. The system checks that it is not in a paused state, and it begins playing the currently-selected playlist from the beginning. 3. When the playing song ends, the system gets the next song in the playlist and plays it.
2.4.1.9.	Extensions <ol style="list-style-type: none"> 2a. If the system is in a paused state (the Music Listener had previously paused the music playback). <ol style="list-style-type: none"> 1. The system begins playing the currently-selected playlist from the point at which it was paused. 3a. If there are no more songs in the playlist. <ol style="list-style-type: none"> 1. The system returns to an Idle state.
2.4.1.10.	Special Requirements None.
2.4.1.11.	Frequency of Occurrence Every time the user wants to play songs using the Tanager system: several times each day.

Figure 7.5 - Sample sections from Tanager Use Case Model

7.5. Domain Model

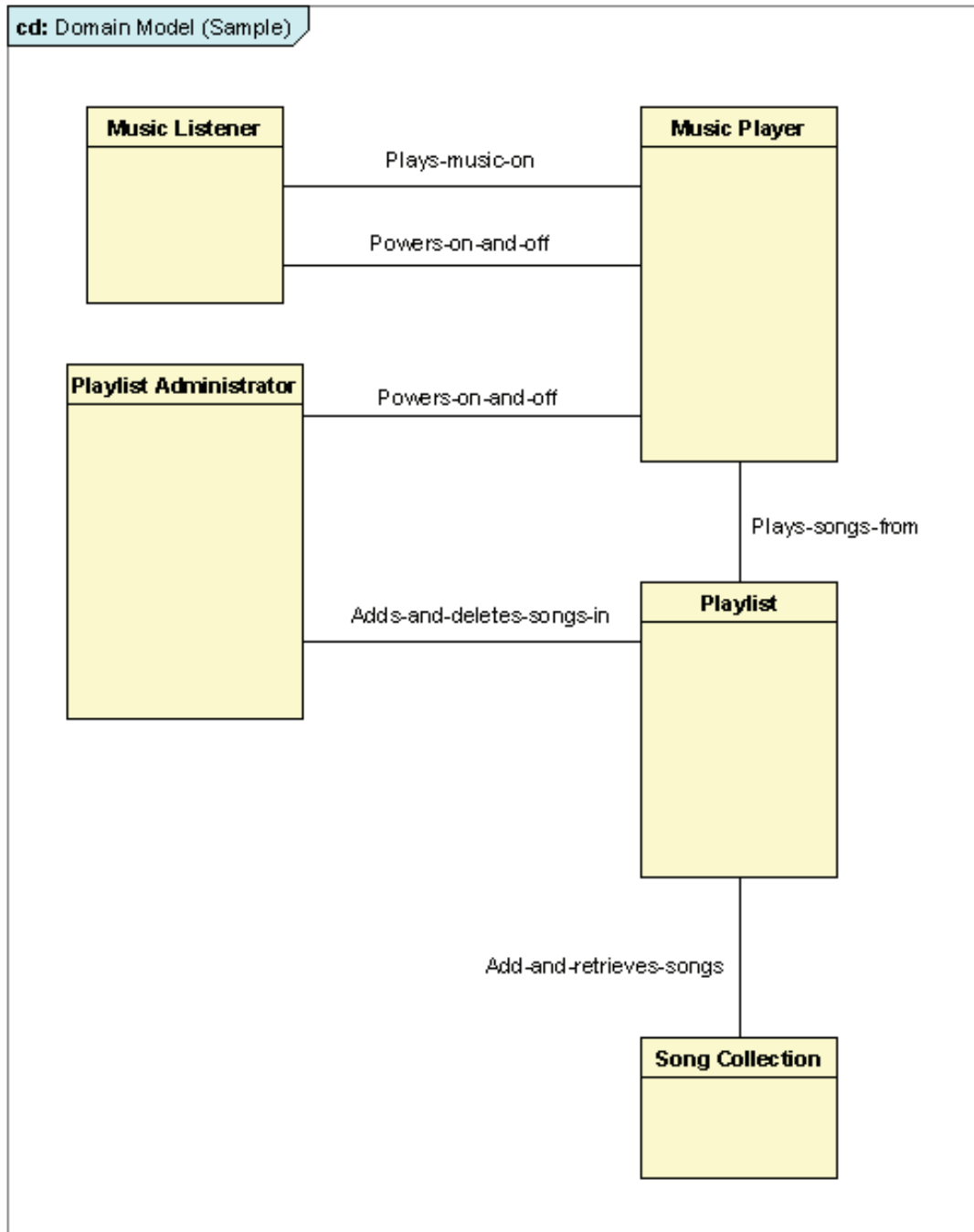


Figure 7.6 - Sample from Tanager Domain Model

7.6. Class Model

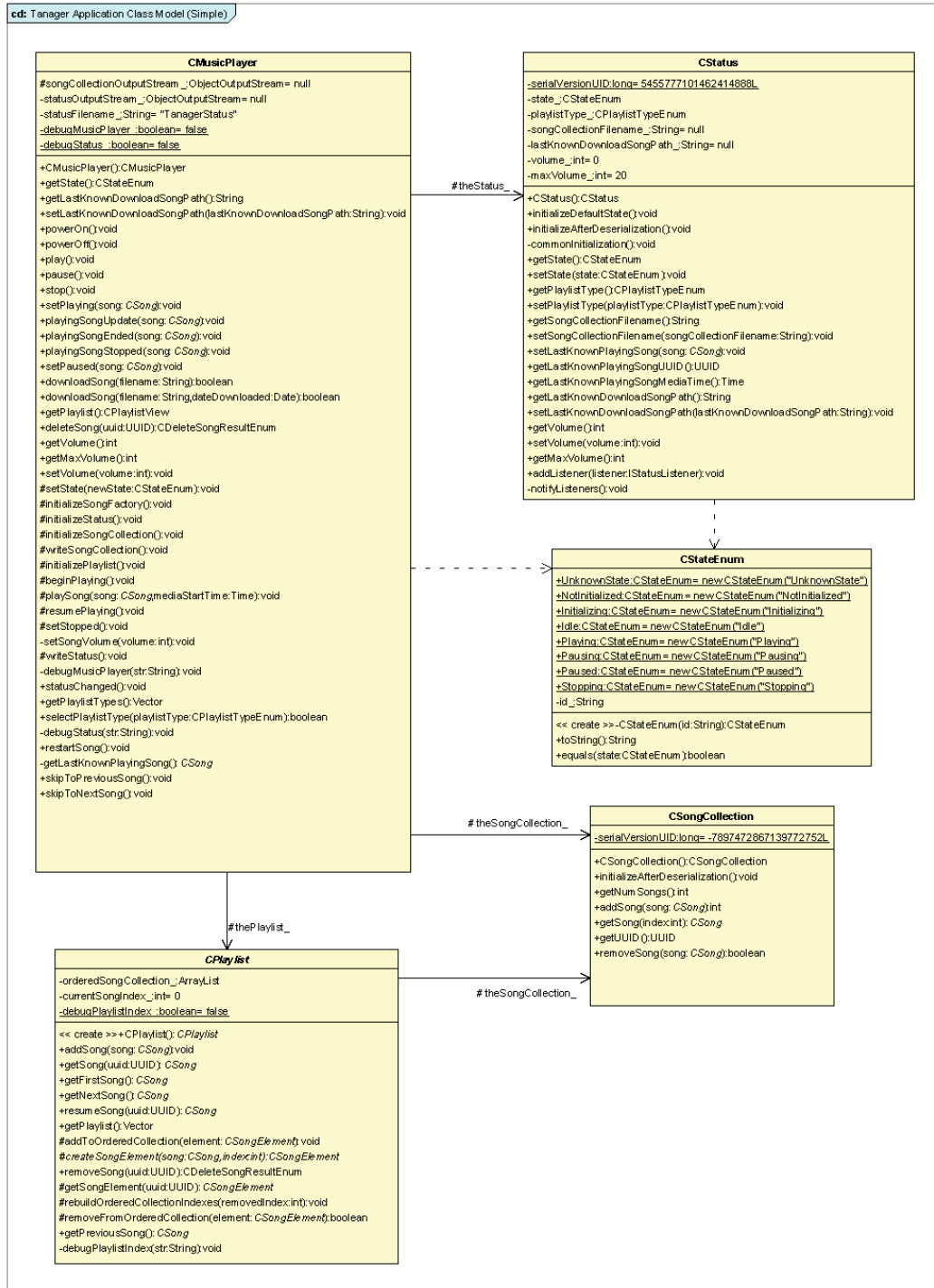


Figure 7.7 - Sample from Tanager Class Diagram

7.7. System Sequence Diagram Specification

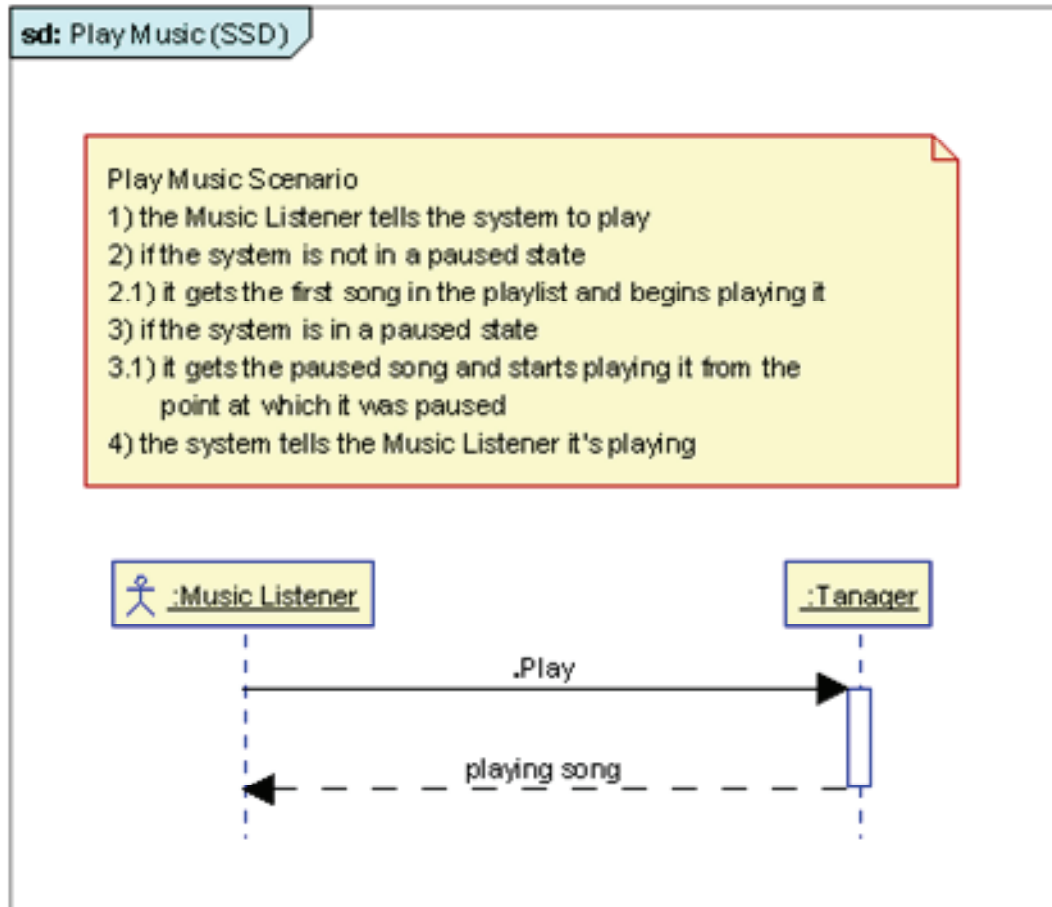


Figure 7.8 - Sample from Tanager System Sequence Diagram Specification

7.8. Sequence Diagram Specification

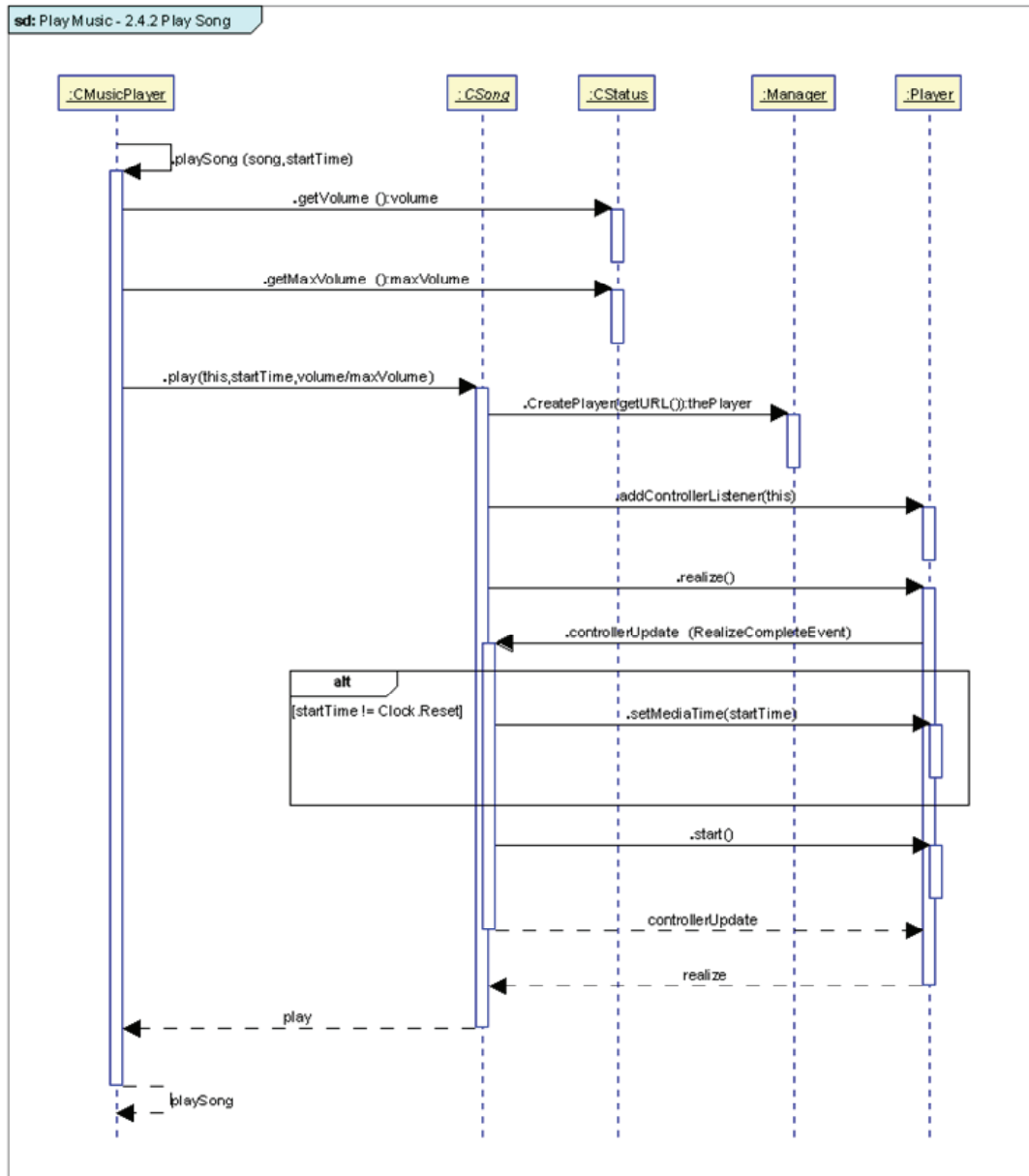


Figure 7.9 - Sample from Tanager Sequence Diagram Specification

7.9. Operation Contract Specification

3.7. Play	
Operation	Play()
Cross References	Use Cases: Play Music
Preconditions	systemState.state is Paused or Idle
Postconditions	<ul style="list-style-type: none"> • systemState.state was changed to Playing • If the system was in the Paused state, the paused song has resumed playing from the point at which it had been paused. • If the system was in the Idle state, the first song from the playlist has begun playing • The SystemState instance, systemState was saved to non-volatile storage

3.14. Stop	
Operation	Stop()
Cross References	Use Cases: Stop Music
Preconditions	systemState.state is Playing or Paused.
Postconditions	<ul style="list-style-type: none"> • If the systemState was Playing, the song that was playing has stopped • systemState.state was changed to Idle • The SystemState instance, systemState was saved to non-volatile storage

Figure 7.10 - Samples from Tanager Operation Contract Specification

8. Appendix B – Sample Javadocs

This appendix presents sample Javadocs from Tanager’s elaboration phase 4. The full Javadocs for Tanager can be found on the Tanager web site. Each elaboration phase has its own set of Javadocs that show the state of the project at the end of that phase:

Elaboration Phase 1:

www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_01/TanagerEclipseProject.zip

Elaboration Phase 2:

www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_02/TanagerEclipseProject.zip

Elaboration Phase 3:

www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_03/TanagerEclipseProject.zip

Elaboration Phase 4:

www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_04/TanagerEclipseProject.zip

8.1. *CMusicPlayer.java*

org.tanager.application

Class **CMusicPlayer**

```

java.lang.Object
├── java.util.Observable
│   └── org.tanager.application.CMusicPlayer

```

All Implemented Interfaces:
[IStatusListener](#)

```

public class CMusicPlayer
extends java.util.Observable
implements IStatusListener

```

This class is the controller for the Tanager system. It uses the Observer Pattern to decouple its behavior from the UI layer.

Version:
\$Revision: 1.4.2.14 \$

Author:
Bob Lavey

Figure 8.1 - Sample javadoc from Tanager *CMusicPlayer.java*

Field Summary	
protected CPlaylist	thePlaylist_ This is a Poseidon-generated association with the CPlaylist object.
protected CSongCollection	theSongCollection_ This is a Poseidon-generated association with the CSongCollection object.
Constructor Summary	
CMusicPlayer ()	This constructor is used to initialize the CMusicPlayer to a default state.
Method Summary	
void	pause () This method is used to tell the system to pause playing.
void	play () This method is used to tell the system to begin or resume playing.
Field Detail	
theSongCollection_	protected CSongCollection theSongCollection_ This is a Poseidon-generated association with the CSongCollection object.
thePlaylist_	protected CPlaylist thePlaylist_ This is a Poseidon-generated association with the CPlaylist object.

Figure 8.2 - Sample javadoc from Tanager CMusicPlayer.java

Constructor Detail
<p>CMusicPlayer public CMusicPlayer() This constructor is used to initialize the CMusicPlayer to a default state.</p>
Method Detail
<p>play public void play() throws CMusicPlayer.CResumePausedSongException This method is used to tell the system to begin or resume playing. Semantics:</p> <ol style="list-style-type: none"> 1. if the state is Idle <ol style="list-style-type: none"> a. call our beginPlaying() method 2. if the state is Paused <ol style="list-style-type: none"> a. call our resumePlaying() method <p>Throws: CMusicPlayer.CResumePausedSongException</p>
<p>pause public void pause() throws CMusicPlayer.CPauseException, CMusicPlayer.CSongNotFoundException This method is used to tell the system to pause playing. Semantics:</p> <ol style="list-style-type: none"> 1. if the state is not Playing <ol style="list-style-type: none"> a. throw a CPauseException 2. call our getLastKnownPlayingSong() method to get the last known playing song 3. if getLastKnownPlayingSong() returned null <ol style="list-style-type: none"> a. throw a CSongNotFoundException 4. call the Status's setState() method to set the state to Pausing 5. call the last known playing song's stop() method <p>Throws: CMusicPlayer.CPauseException CMusicPlayer.CSongNotFoundException</p>

Figure 8.3 - Sample javadoc from Tanager CMusicPlayer.java

8.2. CPlaylist.java

org.tanager.application

Class CPlaylist

java.lang.Object

↳ org.tanager.application.CPlaylist

Direct Known Subclasses:

[CDateDownloadedPlaylist](#), [CRandomPlaylist](#)

```
public abstract class CPlaylist
extends java.lang.Object
```

CPlaylist is an abstract class that encapsulates the concept of a playlist of music files.

A CPlaylist-derived object is a specific ordering of the songs contained in the Song Collection. For example, there may be a playlist that orders the songs by date downloaded, there may be another that orders the songs by song title, and there may be yet another that orders the songs by artist.

Version:
\$Revision: 1.4.2.7 \$

Author:
Bob Lavey

Field Summary

private java.util.ArrayList	orderedSongCollection_ Represents a ordered collection of songs.
protected CSongCollection	theSongCollection_ This is a Poseidon-generated association with the CSongCollection.

Constructor Summary

CPlaylist ()	This constructor is used to initialize a CPlaylist object to a default state.
------------------------------	---

Figure 8.4 - Sample javadoc from Tanager CPlaylist.java

Method Summary	
void	<p>addSong (CSong song)</p> <p>This method is used to add a song to the song collection and to the playlist.</p>
Field Detail	
<p>theSongCollection_ protected CSongCollection theSongCollection_ This is a Poseidon-generated association with the CSongCollection.</p>	
<p>orderedSongCollection_ private java.util.ArrayList orderedSongCollection_ Represents a ordered collection of songs.</p>	
Constructor Detail	
<p>CPlaylist public CPlaylist () This constructor is used to initialize a CPlaylist object to a default state. Semantics:</p> <ol style="list-style-type: none"> 1. instantiate an empty ordered song collection 	
Method Detail	
<p>addSong public void addSong (CSong song) This method is used to add a song to the song collection and to the playlist. Semantics:</p> <ol style="list-style-type: none"> 1. call the Song Collection's addSong () method passing the given song 2. call createSongElement () to create an appropriate CSongElement-derived object 3. call addToOrderedCollection () passing the CSongElement-derived object <p>Parameters: song - the song to be added to the songCollection and the playList</p>	

Figure 8.5 - Sample javadoc from Tanager [CPlaylist.java](#)

8.3. CSongCollection.java

<pre>org.tanager.application</pre>	
<h2>Class CSongCollection</h2>	
<pre>java.lang.Object └─org.tanager.application.CSongCollection</pre>	
<p>All Implemented Interfaces:</p> <pre>java.io.Serializable</pre>	
<hr/> <pre>public class CSongCollection extends java.lang.Object implements java.io.Serializable</pre>	
<p>This class encapsulates the concept of an unordered collection of songs.</p>	
<p>Version: \$Revision: 1.4.2.4 \$</p>	
<p>Author: Bob Lavey</p>	
<p>See Also: Serialized Form</p>	
<hr/>	
<h2>Field Summary</h2>	
<pre>private java.util.Collection</pre>	<pre>songs_ Represents the unordered collection of songs.</pre>
<hr/>	
<h2>Constructor Summary</h2>	
<pre>CSongCollection ()</pre>	<pre>This constructor initializes the CSongCollection to a default state.</pre>
<hr/>	
<h2>Method Summary</h2>	
<pre>int</pre>	<pre>addSong (CSong song) This method adds a song to the song collection</pre>

Figure 8.6 - Sample javadoc from Tanager CSongCollection.java

Field Detail
<p>songs_ <code>private java.util.Collection songs_</code> Represents the unordered collection of songs.</p>
Constructor Detail
<p>CSongCollection <code>public CSongCollection()</code> This constructor initializes the <code>CSongCollection</code> to a default state. Semantics:</p> <ol style="list-style-type: none"> 1. instantiate the unordered song collection
Method Detail
<p>addSong <code>public int addSong(CSong song)</code> This method adds a song to the song collection Semantics:</p> <ol style="list-style-type: none"> 1. call <code>add()</code> on the song collection, passing the given song 2. if <code>add()</code> returns true <ol style="list-style-type: none"> a. return the index at which the song was added 3. if <code>add()</code> returns false <ol style="list-style-type: none"> a. return -1 <p>Parameters: <code>song</code> - the <code>CSong</code> element to be added to the song collection Returns: if the add operation was successful, returns the index at which the element was added; otherwise returns -1</p>

Figure 8.7 - Sample javadoc from Tanager `CSongCollection.java`

9. Appendix C – Sample Code

This appendix presents sample code from Tanager’s elaboration phase 4. The full code for Tanager can be found on the Tanager web site. Each elaboration phase has its own archive of code that shows the state of the project at the end of that phase:

Elaboration Phase 1: www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_01/Javadoc/index.html

Elaboration Phase 2: www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_02/Javadoc/index.html

Elaboration Phase 3: www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_03/Javadoc/index.html

Elaboration Phase 4: www.cs.iastate.edu/~rjlavey/Tanager/Elaboration_04/Javadoc/index.html

9.1. *CMusicPlayer.java*

```

/**
 * Copyright (c) 2006-2007, Iowa State University
 *
 * This file is part of the Tanager project and is
 * licensed under the BSD license.
 */
package org.tanager.application;
/**
 * This class is the controller for the Tanager
 * system. It uses the Observer Pattern to
 */
public class CMusicPlayer extends java.util.Observable implements
org.tanager.application.IStatusListener {
    /**
     * This is a Poseidon-generated association
     * with the <code>CSongCollection</code>
     * object.
     */
    protected org.tanager.application.CSongCollection theSongCollection_ =
null;
    /**
     * This is a Poseidon-generated association
     * with the <code>CPlaylist</code> object.
     */
    protected org.tanager.application.CPlaylist thePlaylist_ = null;
    /**
     * This constructor is used to initialize the
     * <code>CMusicPlayer</code> to a default
     * state.
     */
    public CMusicPlayer() {
        playlistTypes_.add(CPlaylistTypeEnum.DateDownloadedPlaylist);
        playlistTypes_.add(CPlaylistTypeEnum.RandomPlaylist);
    }
}

```

```

/**
 * This method is used to tell the system to
 * begin or resume playing.
 * Semantics:
 * 1) if the state is Idle
 * 1a) call our beginPlaying()method
 * 2) if the state is Paused
 * 2a) call our resumePlaying() method
 */
public void play() throws CResumePausedSongException {
    CStateEnum state = theStatus_.getState();
    if (state.equals(CStateEnum.Idle)) {
        beginPlaying();
    }
    else if (state.equals(CStateEnum.Paused)) {
        resumePlaying();
    }
}
/**
 * This method is used to tell the system to
 * pause playing.
 * Semantics:
 * 1) if the state is not Playing
 * 1a) throw a CPauseException
 * 2) call our getLastKnownPlayingSong() method
 * to get the last known playing song
 * 3) if getLastKnownPlayingSong() returned null
 * 3a) throw a CSongNotFoundException
 * 4) call the Status's setState() method to set
 * the state to Pausing
 * 5) call the last known playing song's stop()
 * method
 */
public void pause() throws CPauseException, CSongNotFoundException {
    CStateEnum state = theStatus_.getState();
    if (state != CStateEnum.Playing) {
        throw new CPauseException();
    }
    //
    // try to find the last known playing
    // song in the playlist
    //
    CSong song = getLastKnownPlayingSong();
    if (song == null) {
        throw new CSongNotFoundException();
    }
    theStatus_.setState(CStateEnum.Pausing);
    song.stop();
}
}

```

9.2. CPlaylist.java

```

/**
 * Copyright (c) 2006-2007, Iowa State University
 *
 * This file is part of the Tanager project and is
 * licensed under the BSD license.
 *
 */
package org.tanager.application;
import java.util.*;
import org.tanager.application.CSong;
import org.tanager.application.CSongElement;
/**
 * <code>CPlaylist</code> is an abstract class
 * that encapsulates the concept of a playlist of
 * music files.
 * A <code>CPlaylist</code>-derived object is a
 * specific ordering of the songs contained in the
 * Song Collection. For example, there may be a
 * playlist that orders the songs by date
 * downloaded, there may be another that orders
 * the songs by song title, and there may be yet
 * another that orders the songs by artist.
 */
public abstract class CPlaylist {
    /**
     * This is a Poseidon-generated association
     * with the <code>CSongCollection</code>.
     */
    protected CSongCollection theSongCollection_ = null;
    /**
     * Represents a ordered collection of songs.
     */
    private ArrayList orderedSongCollection_;
    /**
     * Represents the index of the current song.
     */
    private int currentSongIndex_ = 0;
    /**
     * This constructor is used to initialize a
     * <code>CPlaylist</code> object to a
     * default state.
     * Semantics:
     * 1) instantiate an empty ordered song collection
     */
    public CPlaylist() {
        orderedSongCollection_ = new ArrayList();
    }
}

```

```

/**
 * This method adds the given
 * <code>CSongElement</code> to the ordered
 * song collection.
 * Semantics:
 * 1) call the ordered song collection's
 *    <code>add()</code> method passing the
 *    given <code>CSongElement</code>
 * 2) sort the ordered song collection</li>
 * @param element the
 *    <code>CSongElement</code> to be
 *    added to the songCollection
 */
protected void addToOrderedCollection(CSongElement element) {
    orderedSongCollection_.add(element);
    Collections.sort(orderedSongCollection_);
}
/**
 * This method removes the given
 * <code>CSongElement</code> from the
 * ordered song collection.
 * Semantics:
 * 1) call the orderedSongCollection's
 *    <code>remove()</code> method passing the
 *    given <code>CSongElement</code>
 * 2) return the result of the
 *    <code>remove()</code> method call </li>
 * @return whether or not the
 *    <code>CSongElement</code> was
 *    removed
 * @param element the
 *    <code>CSongElement</code> to be
 *    removed from the songCollection
 */
protected boolean removeFromOrderedCollection(CSongElement
element) {
    return orderedSongCollection_.remove(element);
}
}

```

9.3. CSongCollection.java

```
/**
 * Copyright (c) 2006-2007, Iowa State University
 *
 * This file is part of the Tanager project and is
 * licensed under the BSD license.
 */
package org.tanager.application;
import java.io.*;
import java.util.*;
import org.tanager.ui.IScrollablePaneListener;
/**
 * This class encapsulates the concept of an
 * unordered collection of songs.
 */
public class CSongCollection implements Serializable {
    /**
     * Represents the unordered collection of
     * songs.
     */
    private java.util.Collection songs_ = null;
    /**
     * This constructor initializes the
     * <code>CSongCollection</code> to a default
     * state.
     * Semantics:
     * 1) instantiate the unordered song collection
     */
    public CSongCollection() {
        songs_ = new Vector();
    }
    /**
     * This method returns the number of songs in
     * the collection.
     * @return the size of the unordered song
     *         collection
     */
    public int getNumSongs() {
        return songs_.size();
    }
}
```

```

/**
 * This method adds a song to the song
 * collection
 * Semantics:
 * 1) call add() on the song collection,
 *    passing the given song</li>
 * 2) if add() returns true
 * 2a) return the index at which the song was
 *     added
 * 3) if add() returns false
 * 3a) return -1</li>
 * @param song the <code>CSong</code>
 *         element to be added to the song
 *         collection
 * @return if the add operation was
 *         successful, returns the index at
 *         which the element was added;
 *         otherwise returns -1
 */
public int addSong(org.tanager.application.CSong song) {
    int returnVal = -1;
    if (songs_.add(song)) {
        returnVal = (songs_.size() - 1);
    }
    return returnVal;
}
/**
 * This method removes a song from the song
 * collection
 * Semantics:
 * 1) call <code>remove()</code> on the
 *    song collection, passing the given song
 * 2) return the result of the
 *    <code>remove()</code> method </li>
 * @return whether or not the song was
 *         removed
 * @param song the <code>CSong</code>
 *         element to be removed from the song
 *         collection
 */
public boolean removeSong(CSong song) {
    return songs_.remove(song);
}
}

```

References

- [1] Larman, Craig. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall PTR, Upper Saddle River, NJ, 2005.
- [2] Krutchen, Phillippe. *The Rational Unified Process: An Introduction.* Addison-Wesley, Boston, MA, 2003.
- [3] Fowler, Martin. *UML Distilled.* Addison-Wesley, Boston, MA, 2004.
- [4] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.
- [5] Shalloway, Alan, and James R. Trott. *Design Patterns Explained.* Addison-Wesley, Boston, MA. 2004
- [6] Martin Marietta. *Object-Oriented Analysis.* Mastering a World of Technology. Advanced Concepts Center of Martin Marietta, 1995.
- [7] Martin Marietta. *Object-Oriented Designs.* Mastering a World of Technology. Advanced Concepts Center of Martin Marietta, 1995.
- [8] Clifton, Curtis. *The StickSync Project.* 8 Oct, 2004. Iowa State University. 07 Mar, 2007 <<http://www.cs.iastate.edu/~leavens/ComS362/sticksync/>>.