„Freya" MPD-Client

Teil der Software Engineering II Studienarbeit WS 2011/2012, Inf 3

Christopher Pahl
Christoph Piechula
Eduard Schneider

16. Januar 2012

Teil I.

Prolog

Ziel dieser Studienarbeit ist die vollständige Bearbeitung einer vorgegebenen Aufgabenstellung nach einem selbst gewählten Vorgehensmodell. Die Aufgabenstellung schreibt vor, sich in einer Gruppe zusammen zu finden und gemeinsam ein Software-Projekt zu bearbeiten und dabei strukturiert und professionell vorzugehen.

Inhaltsverzeichnis

Teil II.

Definition des Projekts

1. Definition des Projekts

1.1. Rahmenbedingungen

- Persistente Datenspeicherung
  - Datei oder Datenbank (wenn schon bekannt)
- Netzwerk-Programmierung
  - Eine verteilte Architektur (z.B.: Client/Server)
- GUI
  - Swing
  - Web-basiert
  - Gtk+ (durch Nachfrage)

1.2. Prozess-Anforderungen

- Dokumentation aller Phasen(Analyse bis Testen)
- Auswahl eines konkreten Prozessmodells
  - Z.B. sd&m, M3, RUP, Agile Methoden ...
  - Begründung (warum dieser Prozess passt zu Ihrem System)
- Erstellung der Dokumente und UML-Diagramme
  - Visio
  - UML Werkzeuge (freie Wahl)
- Fertige Implementierung
  - Es kann mehr spezifiziert sein als implementiert
- Spezifikation von Testszenarien
  - und der Beleg der erfolgreichen Ausführung
- Lauffähiges System

1.3. Mögliche Themen

- CRM Systeme
  - Bibliothek
  - Musikshop
  - ...
- Kommunikationssysteme
- Chat-Variationen (Skype, etc.)
- File-Verwaltungs-Systeme (eigener Cloud-Dienst)
- ...

Portale

- Mitfahrgelegenheit
- Dating-Agentur ;)
- ...

[1]

Diese Arbeit ist wichtig, um den Studenten zu zeigen, wie man in einem Team zusammenarbeitet und nach Software-Engineering-Methoden qualitativ hochwertige Software erstellt. Es geht im Folgenden um einen Music-Player-Daemon-Client (Näheres bitte der Definition entnehmen). Dieses Thema wird behandelt, da es alle Rahmenbedingungen abdeckt und im Interesse der Autoren liegt. Die Besonderheit liegt darin, dass sich diese Software nach Fertigstellung auch wirklich anwenden lässt. Ziel ist die Erweiterung der Fähigkeiten im Bereich der Software Engineering sowie das Erlernen von Methoden für wissenschaftliches Arbeiten.

[1] Folie Anforderungen, Autor Prof. Dr. Philipp Schaible, WS 2011/2012, Inf 3

2. Wasserfallmodell mit Rücksprung

2.1. Definition

Das Wasserfallmodell ist ein lineares (nicht iteratives) Vorgehensmodell in der Softwareentwicklung, bei dem der Sotwareentwicklungsprozess in Phasen organisiert wird. Dabei gehen die Phasenergebnisse wie bei einem Wasserfall immer als bindende Vorgaben für die nächsttiefere Phase ein.

Im Wasserfallmodell hat jede Phase vordefinierte Start- und Endpunkte mit eindeutig definierten Ergebnissen. In Meilensteinsitzungen am jeweiligen Phasenende werden die Ergebnisdokumente verabschiedet. Zu den wichtigsten Dokumenten zählen dabei das Lastenheft sowie das Pflichtenheft. In der betrieblichen Praxis gibt es viele Varianten des reinen Modells. Es ist aber das traditionell am weitesten verbreitete Vorgehensmodell.

Der Name ,,Wasserfall" kommt von der häufig gewählten grafischen Darstellung der fünf bis sechs als Kaskade angeordneten Phasen.

Erweiterungen des einfachen Modells (Wasserfallmodell mit Rücksprung - gestrichelt, siehe 2.1) führen iterative Aspekte ein und erlauben ein schrittweises ,,Aufwärtslaufen" der Kaskade, sofern in der aktuellen Phase etwas schieflaufen sollte, um den Fehler auf der nächsthöheren Stufe beheben zu können. [1]

[1] Zitat aus: http://de.wikipedia.org/wiki/Wasserfallmodell

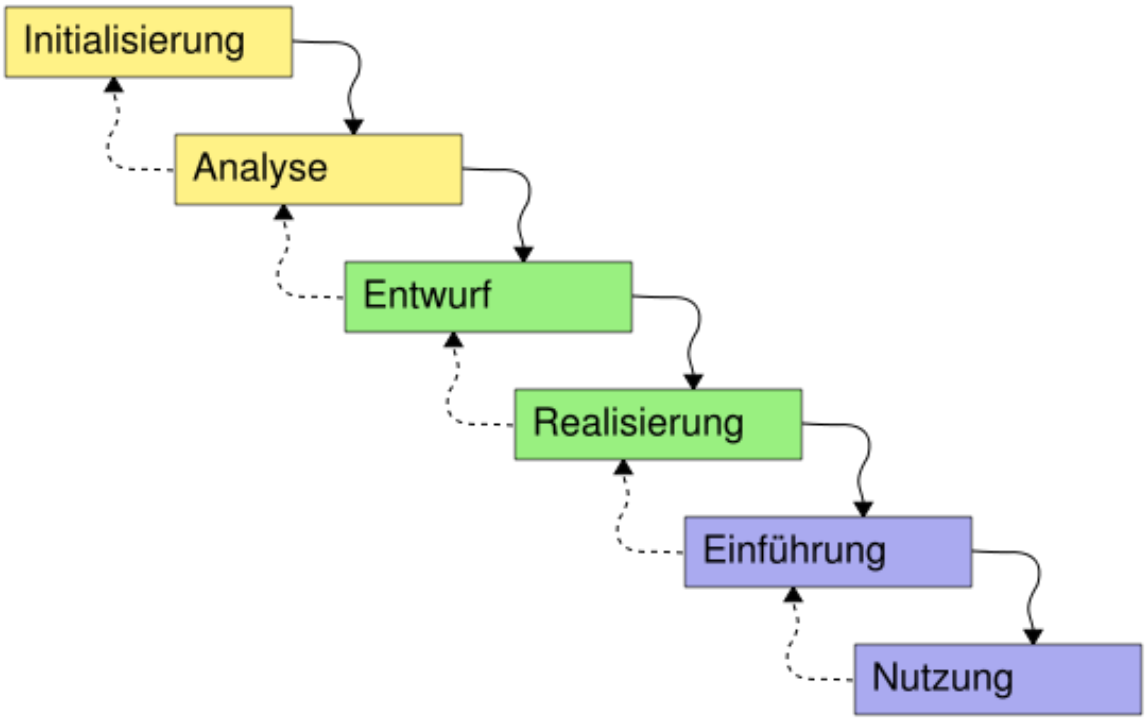


Abbildung 2.1.: Wasserfallmodel mit Rücksprung

2.2. Warum dieses Modell?

Wir haben uns für das Wasserfallmodell mit Rücksprung [2] entschieden, weil dieses Modell alle Phasen der Entwicklung klar abgrenzt und sich optimal auf einen professionellen Softwareentwicklungsvorgang abbilden lässt. Dieses Modell ermöglicht eine klare Planung und Kontrolle unseres Softwareprojekts, da die Anforderungen stets gleich bleiben und der Umfang einigermaßen gut abschätzbar zu sein scheint. Um ein paar Nachteile dieses Modells auszuhebeln, haben wir uns für die erweiterte Version mit Rücksprung entschieden. Beispielsweise sind die klar voneinander abgegrenzten Phasen in der Realität oft nicht umsetzbar. Daher sind wir somit flexibler gegenüber Änderungen.

[2] Wasserfallmodell mit Rücksprung, (Bild-Quelle)

3. Richtlinien

3.1. Programmierrichtlinien

- Modulare Gestaltung.
- Sauberkeit ist wichtiger als Performance.
- Durchgehende Anwendung des Model-View-Controller Pattern.
- ,,make" sollte keine Warnungen ausgeben, die man leicht umgehen könnte. (-Wall und -Wextra wird genutzt)
- ,,make test" soll vollständig durchlaufen.
- Sinnvolle Variablenbenennung.
- Keine nichtkonstanten globalen Variablen.
- Klassenmethoden (*static*) nur in Ausnahmefällen bzw. nur mit guten Gründen.
- Valgrind darf keine Laufzeitfehler bringen, die nicht von Gtk oder anderen Bibliotheken stammen.
- ,,camelCase" bei Objektnamen, C-Style (function_name()) bei Funktionsnamen - Präzise Namen.
- Tabstop = 4 Leerzeichen, Allman-Stil.

3.1.1. Begründung

Einhaltung dieser Programmierrichtlinien sorgen für ein einfaches, übersichtliches und einheitliches Arbeiten. Jeder sollte sich ohne größere Umstände in den Code eines anderen einlesen können. Dies gewährleistet eine hohe Wartbarkeit der Programm-Codes und beugt außerdem Fehlern vor. Das Programm ist leicht erweiterbar ohne große Anpassungen vornehmen zu müssen.

3.2. Toolauswahl

- CMake (Buildsystem)
- g++ (C++ Compiler)
- Valgrind (Memorydebugger)
- git (Hosting auf Github) [1]
- Glade (GUI-Designer)
- doxygen (Interne Dokumentationsgenerierung)
- Devhelp (Dokumentationsbrowser)

3.2.1. Begründung

CMake wurde ausgewählt da es eine solide und vor allem einfache Syntax bietet, und zudem leicht anpassbar ist. Git dient zur Versionsverwaltung. Github wurde dabei als Hostingplattform ausgewählt, da das Hosting dort für Open-Source-Projekte frei ist. Glade bietet eine solide Trennung von der grafischen Oberfläche zum Kontrollkern des Programms. Außerdem kann mit Glade sehr einfach eine grafische Oberfläche erstellt werden. Da man sich in C++ im Gegensatz zu Java um sehr viele Sachen selbst kümmern muss, wurde Valgrind als Memorydebugger ausgewählt, um das Aufspüren von Fehlern zu vereinfachen.

3.3. Bibliotheken

- gtkmm3 (C++ Wrapper für Gtk+) [2]
- libmpdclient (Lowlevel MPD-Bibliothek für C) [3]
- libxml2 (XML Parser Library für C) [4]
- libnotify (Anzeige von Desktopnachrichten) [5]
- Avahi-glib (Interface zum Avahidaemon) [6]

[1] https://github.com/studentkittens/Freya
[2] http://www.gtkmm.org/de/index.html
[3] http://www.musicpd.org/doc/libmpdclient/files.html
[4] http://xmlsoft.org/index.html
[5] http://developer.gnome.org/libnotify/
[6] http://avahi.org/wiki/WikiStart#WhatisAvahi

3.3.1. Begründung

C++ wurde aufgrund persönlicher Interessen der Autoren gewählt. Außerdem gibt es für Java nur wenige oder sehr alte Bibliotheken für dieses Projekt. Gtkmm3 bietet ein dynamisches Layout und ist leichtgewichtiger als Qt, außerdem ist es einfacher in der Handhabung und lässt sich auf den meisten Desktopumgebungen besser integrieren. Swing ist aus Sicht der Autoren nicht geeignet. Libmpdclient ist eine eigene lowlevel C-Bibliothek, die unabhängig vom eigentlichen MPD-Server ist. Libxml2 liefert einen standardisierten Xml Parser und ist sehr leichtgewichtig, sowie auf einer großen Zahl von Systemen vorhanden. Libnotify liefert Benachrichtigungen über interne Events und ist auf den meisten Linux Distributionen verbreitet. Avahi-glib ist ein Interface für den optionalen Avahidaemon. Avahi dient als Server-Borwser, kann allerdings nur MPD Server finden, die sich per Zeroconf am Avahidaemon registriert haben.
Primäre Entwicklerplattform ist ein GNU/Linux-System nach Wahl. Der Client soll dabei nach Möglichkeit portabel gehalten werden, um später auf andere vom MPD Server unterstützte Plattformen portiert werden zu können.

4. Definition

Der MPD ist eine Client/Server-Architektur, in der die Clients und Server (MPD ist der Server) über ein Netzwerk interagieren. MPD ist also nur die Hälfte der Gleichung. Zur Nutzung von MPD, muss ein MPD-Client (auch bekannt als MPD-Schnittstelle) installiert werden. Als Netzwerkprotokoll wird das MPD eigene Protokoll verwendet. [1]

4.1. Definition des MPD

Der Music Player Daemon (kurz MPD) ist ein Unix-Systemdienst, der das Abspielen von Musik auf einem Computer ermöglicht. Er unterscheidet sich von gewöhnlichen Musik-Abspielprogrammen dadurch, dass eine strikte Trennung von Benutzeroberfläche und Programmkern vorliegt. Dadurch ist die grafische Benutzeroberfläche auswechselbar und auch eine Fernsteuerung des Programms über das Netzwerk möglich. Die Schnittstelle zwischen Client und Server ist dabei offen dokumentiert und der Music Player Daemon selbst freie und quelloffene Software.

Der MPD kann wegen seines geringen Ressourcenverbrauchs nicht nur auf Standartrechnern sondern auch auf einem abgespeckten Netzwerkgerät mit Audioausgang betrieben werden und von allen Computern oder auch Mobiltelefonen / PDAs im Netzwerk ferngesteuert werden.

Es ist auch möglich den Daemon und den Client zur Fernsteuerung lokal auf dem gleichen Rechner zu betreiben, er fungiert dann als normaler Medienspieler, der jedoch von einer Vielzahl unterschiedlicher Clients angesteuert werden kann, die sich in Oberflächengestaltung und Zusatzfunktionen unterscheiden. Mittlerweile existierten auch zahlreiche Clients, die eine Webschnittstelle bereitstellen.

Der MPD spielt die Audioformate Ogg Vorbis, FLAC, OggFLAC, MP2, MP3, MP4/AAC, MOD, Musepack und wave ab. Zudem können FLAC-, OggFLAC-

[1] http://www.musicpd.org/doc/protocol/index.html

, MP3- und OggVorbis-HTTP-Streams abgespielt werden. Die Schnittstelle kann auch ohne manuelle Konfigruation mit der Zeroconf-Technik angesteuert werden. Des Weiteren wird Replay Gain, Gapless Playback, Crossfading und das Einlesen von Metadaten aus ID3-Tags, Vorbis comments oder der MP4-Metadatenstruktur unterstützt. [2]

4.1.1. Der MPD kann:

- Musik abspielen
- Musik kontrollieren und in Warteschlangen reihen
- Musik Dateien dekodieren
- HTTP-Streaming
  - Eine HTTP-URL kann zur Warteschlange hinzugefügt oder direkt abgespielt werden.

4.1.2. Der MPD kann nicht:

- Album-Cover speichern
- Funktionen eines Equalizers bereitstellen
- Musik Taggen (Informationen aus dem Web suchen)
- Text für Playlist-Dateien parsen
- Statistische Auswertungen machen
- Musik visualisieren
- Funktionen eines Remote-File-Servers bereitstellen
- Funktionen eines Video-Servers bereitstellen

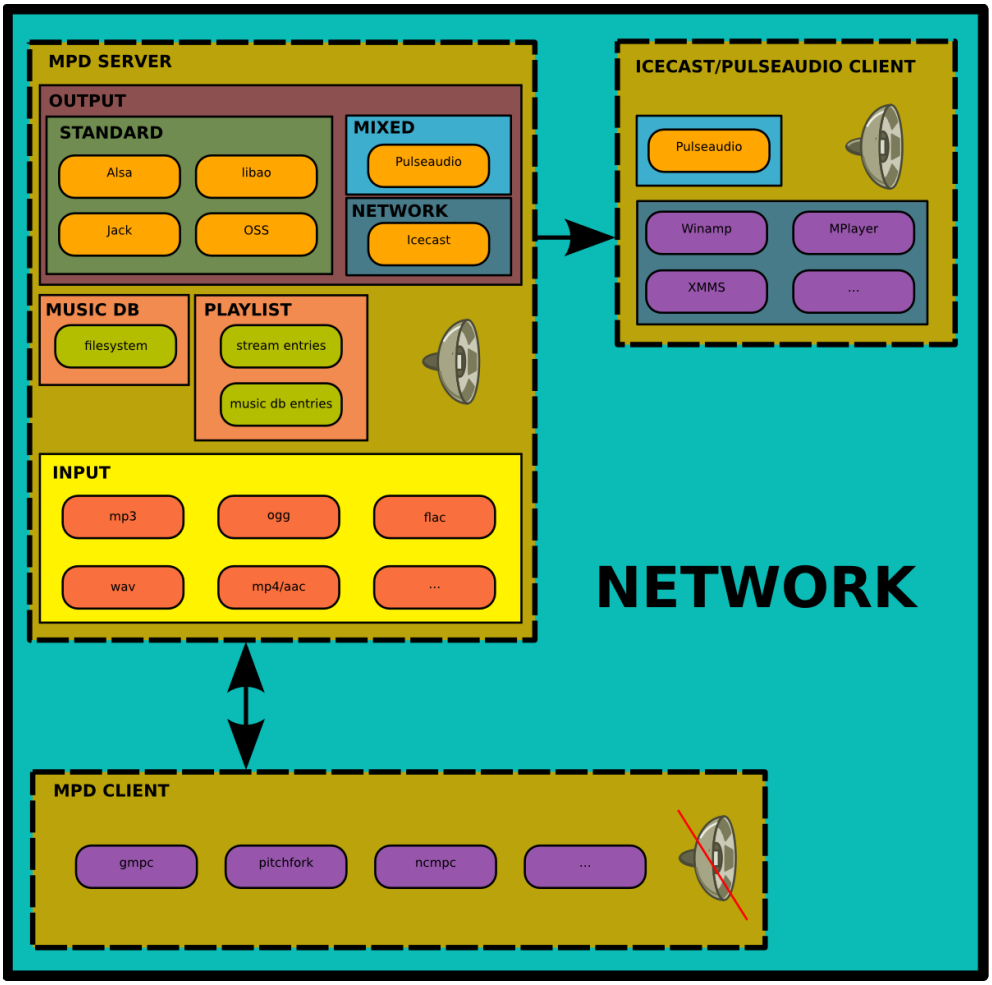
4.2. Definiton des MPD-Client

Der Music Player Daemon Client ist nun die Schnittstelle zum MPD. Über diesen Client kann der MPD gesteuert werden. Es gibt viele verschiedene Clients mit unterschiedlichsten Funktionen, da der Client nicht auf den Funktionsumfang des MPD begrenzt ist. Das heißt im Klartext,

[2] Zitat aus: http://de.wikipedia.org/wiki/Music_Player_Daemon

dass der Client zwar nur die Funktionen über das Netzwerk steuern kann, die vom MPD implementiert sind aber nicht, dass er deshalb auch keine lokalen Dienste bzw. Funktionen anwenden kann. So kann ein Client beispielsweise alle Funktionen lokal implementieren, die unter dem Punkt ,,Der MPD kann nicht" erwähnt wurden.

4.3. Grafische Übersicht



[3] Der MPD-Server bekommt als Input *mp3, ogg, flac, wav, mp4/aac,...* Musik-Dateien die entweder in einer Musik-Datenbank oder in Playlisten gespeichert sind. Der Standardoutput des MPD ist Alsa, libao, jack oder OSS, die Musik kann aber auch über einen Icecast oder Pulseaudio Clienten ausgegeben werden. Der MPD-Client steuert den MPD-Server, dekodiert aber selbst keine Musik.

[3] Bild-Quelle

Teil III.

Lastenheft

5. Lastenheft

Bei der Erstellung dieses Lastenheftes wurde sich an folgendem Beispiel orientiert:
`http://www.stefan-baur.de/cs.se.lastenheft.beispiel.html?glstyle=2010`

5.1. Zielbestimmungen

Welche Ziele sollen durch den Einsatz der Software erreicht werden? Dem einzelnen Benutzer soll das Abspielen von Musik über eine Netzwerkverbindung ermöglicht werden. Die Musik wird dabei vom MPD Server in einer Datenbank gespeichert. Der Rechner, auf dem dieser Dienst läuft, ist auch primär für die Audioausgabe zuständig. Desweiteren sind andere Audioausgabequellen, wie z.B. PulseAudio, möglich. Die Client-Rechner sollen die Ausgabe steuern und Abspiellisten auf dem Server verwalten können. Die Bedienung soll für alle Benutzer sehr einfach und komfortabel über einen lokalen Client realisiert werden. Bei jedem Start des Clients wird der Zustand des MPD-Servers ,,gespiegelt".

Standardmäßig sollen den Benutzern folgende Funktionen zur Verfügung stehen:

- Abspielen und Steuerung von Musik (Play, Stop, Skip, ...)
- Erstellung und Verwaltung von Playlisten
- Anzeige der MPD-Datenbank
- Verwaltung der der Audioausgabegeräte

Weitere Funktionen müssen modular integrierbar sein, allerdings müssen sie noch nicht implementiert werden. Einige Beispiele für weitere Funktionen wären:

- Finden von Album-Informationen (Unter Verwendung von libglyr)
- Speichern von Serveradressen (Adressbuchartig)
- Dynamische Playlisten

Die Systemsprache soll auf Englisch festgelegt werden.

5.1.1. Projektbeteiligte

Wer soll an dem Projekt teilnehmen?

- Christopher Pahl
- Christoph Piechula
- Eduard Schneider

5.2. Produkteinsatz

Für welche Anwendungsbereiche und Zielgruppe ist die Software vorgesehen? Der MPD-Client ist nicht auf bestimmte Gewerbe beschränkt, jeder soll diesen Client verwenden können. Die Software soll unter der GNU General Public License (GPLv3) vom 29 Juni 2007 stehen.

Siehe dazu für Details: `http://www.gnu.org/licenses/gpl.html`

5.2.1. Anwendungsbereiche

Einzelpersonen verwenden dieses System überall da, wo mit einem unixoiden Betriebssystem Musik abgespielt werden soll. Damit wären z.B. Personal Computer, Musikanlagen, Laptops und evtl. sogar diverse Smartphones möglich.

5.2.2. Zielgruppen

Personengruppen, die komfortabel von überall aus auf ihre Musik und Playlisten zugreifen wollen, ohne diese jedes mal aufwändig synchronisieren zu müssen (z.B. durch Abgleich von Datenträgern).

Aufgrund der für das System vorgesehenen Betriebsumgebung sind ebenso grundlegende Kenntnisse im Umgang mit Unixoiden Betriebssystemen nötig, zudem muss der Benutzer die Systemsprache Englisch beherrschen.

5.2.3. Betriebsbedingungen

Das System soll sich bezüglich der Betriebsbedingungen nicht sonderlich von vergleichbaren Systemen bzw. Anwendungen unterscheiden und dementsprechend folgende Punkte erfüllen:

- Betriebsdauer: Täglich, 24 Stunden
- Keinerlei Wartung soll nötig sein
- Sicherungen der Konfiguration müssen vom Benutzer vorgenommen werden

5.3. Produktumgebung

5.3.1. Software

Softwareabhängigkeiten sollen durch den Entwickler bestimmt werden. Dies gewährleistet, dass der Entwickler diesbezüglich nicht eingeschränkt wird und somit mehr Möglichkeiten hat. Nach Möglichkeit soll allerdings portable Software bevorzugt werden.

5.3.2. Hardware

Das Produkt soll möglichst wenig Anforderungen an die Hardware stellen, da die Software eventuell auch auf sehr Hardwareschwachen Geräten (wie z.B. Smartphones) verwendet werden soll.

5.3.3. Orgware

Es soll nach Möglichkeit keine Orgware für den reinen Betrieb vonnöten sein. Der Nutzer der Software soll sich um möglichst wenig Nebenläufiges zu kümmern haben, wie dem Starten anderer Dienste vor der Benutzung des Clients.

5.4. Produktfunktionen

Welche sind die Hauptfunktionen aus Sicht des Auftraggebers?

5.4.1. Allgemein

Beim ersten Start des Systems soll eine Standard-Konfiguration geladen werden und die Verbindungseinstellungen zu einem MPD-Server müssen vorgenommen werden. Bei jedem weiteren Start soll die Konfiguration geladen werden, die vom Benutzer erstellt wurde. Falls keine Konfiguration gefunden wurde, oder diese beschädigt ist, so wird auf eine vom Client bereitgestellte Standardkonfiguration zurückgegriffen. Der Benutzer soll sämtliche Einstellungen zu jeder Zeit über die Oberfläche des Clients ändern können. Natürlich sollen alle üblichen Musikabspielfunktionen vorhanden sein. Dazu gehören *Play, Stop, Previous, etc. und Next.*Aber auch erweiterte Funktionen wie *Repeat, Consume und Random*sollen einstellbar sein. Der Benutzer soll über die Software direkten Zugriff auf das virtuelle Dateisystem des Servers haben, um nach Musik zu suchen und diese abspielen zu können. Aus dem Dateisystem heraus soll der Nutzer ebenfalls die Möglichkeit haben, Musik-Dateien direkt zu Playlisten und zur Warteschlange hinzuzufügen. Verbindungseinstellungen müssen auf möglichst einfache Art und Weise vorgenommen werden können, wenn möglich sollte dem Nutzer eine Liste von verfügbaren Servern angezeigt werden.

Dem Nutzer soll ermöglicht werden, dass er nach bestimmten Titeln, Alben oder Interpreten suchen kann, da es mit dieser Software möglich ist, auch sehr große Musik-Datenbanken zu steuern. Administratorfunktionen müssen nicht implementiert werden, das sie vom Unix-System übernommen werden.

5.5. Produktdaten

Welche Daten sollen persistent gespeichert werden?
Die vom Benutzer vorgenommenen Verbindungs- und Client- Einstellungen sollen auf dem Rechner lokal und persistent gespeichert werden. Nur so kann das Laden und Übernehmen dieser Einstellungen nach jedem Start gewährleistet werden.

Außerdem soll eine Log-Datei auf den einzelnen Rechnern angelegt werden, die dieses System verwenden. Die Speicherung des Logs und der Konfiguration soll dabei dem XDG-Standard[1] entsprechen. In dieser Log-Datei werden Nachrichten des Systems gespeichert, um eventuelle Fehler leicht finden und beheben zu können. Es soll stets der aktuelle Zustand des MPD Servers widergespiegelt werden.

Dem Nutzer sollen viele verschiedene Informationen angezeigt werden, nicht nur Standardinformationen wie Titel, Album und Interpret, sondern auch Musik-Qualität, -Länge und Lautstärke. Es soll außerdem eine primitive Statistik implementiert werden, die anzeigt, wie viele Lieder, Alben und Interpreten in der Datenbank vorhanden sind, wie lange man schon mit dem Server verbunden ist und wie lange die gesamte Abspielzeit aller Lieder in der Datenbank dauert.

Eine Profilverwaltung muss nicht implementiert werden, da dies wird bereits über die Unix Benutzerverwaltung geregelt wird. Eine lokale Datenbank muss ebenfalls nicht vorhanden sein, dies wird durch den MPD-Server ermöglicht.

5.6. Qualitätsanforderungen

Die Software soll von hoher Qualität sein. Folgende Anforderungen sollen erfüllt werden:
Die Software soll korrekt sein, d.h. möglichst wenige Fehler enthalten. Sie soll aber auch, für den Fall, dass dennoch Fehler auftreten, robust und tolerant auf diese reagieren. Außerdem spielt die Wartbarkeit eine wichtige Rolle. Falls sich die Softwareumgebung des MPD-Clients ändert, muss dieser leicht angepasst werden können. Der Client soll intuitiv und schnell bedienbar sein. Das

[1] `http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html#variables`

Hauptaugenmerk liegt dabei aber auf Ressourceneffizienz. Vor allem soll dabei das Netzwerk nicht stark beansprucht werden, um auch bei langsamen Verbindungen den Client bedienen zu können. Speicher- und Recheneffizienz ist hierbei zwar von Belang, aber dennoch zweitrangig. Sollte es Funktionen geben, die nicht unter den Begriff SStandardffallen, sollte eine knappe und präzise Beschreibung der Funktion vorhanden sein. Das Design der Software-Oberfläche muss zwar ansprechend sein, ist im Endeffekt allerdings drittrangig.

5.7. Ergänzungen

5.7.1. Realisierung

Das System muss mit den Programmiersprachen C und/oder C++ realisiert werden. Dabei ist auf Objektorientierung zu achten, um Modularität und Wartbarkeit gewährleisten zu können. Es können beliebige Entwicklungsumgebungen verwendet werden, wobei auch ein Texteditor mit Syntaxhighlighting vollkommen ausreichend ist. Um einfaches und sicheres Arbeiten ermöglichen zu können, soll die Versionsverwaltungssoftware *git* benutzt werden, um die Entwicklungsdateien zu speichern und zu bearbeiten. Zu dem Projekt soll eine ausführliche Dokumentation erstellt werden, um dauerhafte Wartbarkeit und Anpassung des MPD-Clients gewährleisten zu können, dazu gehören auch entsprechende UML-Diagramme.

5.7.2. Die nächste Version

Aufgrund des modularen Aufbaus kann das System beliebig oft und in verschiedene Richtungen weiterentwickelt werden. Weiter oben sind bereits Möglichkeiten für konkrete Erweiterungen aufgelistet.

Teil IV.

Pflichtenheft

6. Pflichtenheft

Bei der Erstellung dieses Pflichtenheftes wurde sich an folgendem Beispiel von Stefan Baur orientiert.
`http://www.stefan-baur.de/cs.se.pflichtenheft.beispiel.html?glstyle=2010`

6.1. Zielbestimmungen

6.1.1. Projektbeteiligte

Wer soll an dem Projekt teilnehmen?

- Christopher Pahl
- Christoph Piechula
- Eduard Schneider

6.1.2. Muss-Kriterien

- Verbindungsaufbau
- Durchführung benutzerspezifischer Client-Einstellungen
- Musik-Steuerung
- Warteschlangenverwaltung
- Playlistverwaltung
- Datenbankverwaltung
- Frontend für die Settings
- Statistikanzeige
- Verwaltung der Ausgabegeräte

6.1.3. Wunsch-Kriterien

- Anzeige von Onlinecontent (Albencover, Lyrics etc.) unter Verwendung von libglyr

6.2. Produkteinsatz

Der MPD-Client ist nicht auf bestimmte Gewerbe beschränkt, ein jeder soll diesen Client verwenden können.
Die Software soll unter folgender Lizenz stehen: General Public License Version 3 vom 29 Juni 2007.
Definition der GPL:

`http://www.gnu.org/licenses/gpl.html`

6.2.1. Anwendungsbereiche

Einzelpersonen verwenden dieses System überall da, wo mit einem Unixoiden Betriebssystem Musik abgespielt werden soll. Das wären z.B. Personal Computer, Musikanlagen, Laptops und evtl. sogar diverse Smartphones möglich.

6.2.2. Zielgruppen

Personengruppen, die komfortabel von überall aus auf ihre Musik und Playlist zugreifen wollen, ohne diese jedes mal aufwändig synchronisieren zu müssen (z.B. Durch Abgleich von Datenträgern). Aufgrund der für das System vorgesehenen Betriebsumgebung sind ebenso Kenntnisse im Umgang mit Unix nötig. Der Benutzer muss die Systemsprache Englisch beherrschen.

6.2.3. Betriebsbedingungen

Das System soll sich bezüglich der Betriebsbedingungen nicht sonderlich von vergleichbaren Systemen bzw. Anwendungen unterscheiden und dementsprechend folgende Punkte erfüllen:

- Betriebsdauer: Täglich, 24 Stunden
- Keinerlei Wartung soll nötig sein
- Sicherungen der Konfiguration müssen vom Benutzer vorgenommen werden

6.3. Produktumgebung

6.3.1. Software

- Unixoides Betriebssystem

- MPD-Server
- Avahi Daemon
- Benötigte Bibliotheken können statisch einkompiliert werden

Ein MPD-Server muss nicht unbedingt lokal installiert, jedoch dann über das Netzwerk (Internet) erreichbar sein. Ohne MPD-Server soll der Client in einen definierten Zustand hochgefahren werden der das Verbinden ermöglicht. Ein laufender Avahi Daemon ist optional. Er ist nicht vonnöten, aber durchaus praktisch, wenn man sich nicht ständig den Host (bzw. die IP) vom Administrator geben lassen will und stattdessen in der eingebauten Serverliste nachschauen kann.

6.3.2. Orgware

- CMake (Buildsystem)
- g++ (C++ Compiler)
- Valgrind (Memorydebugger)
- git (Hosting auf Github)
- Glade (GUI-Designer)
- doxygen (Interne Dokumentationsgenerierung)
- Devhelp (Dokumentationsbrowser)

6.4. Produktfunktionen

Funktionen des MPD-Clients.
Beim ersten Start des Systems soll eine einkompilierte Standard-Konfiguration geladen und die Verbindungseinstellungen zu einem MPD-Server vorgenommen werden. Bei jedem weiteren Start soll die vom Benutzer erstellte Konfiguration geladen werden. Falls keine Konfiguration gefunden wurde oder diese beschädigt ist, wird auf eine vom Client bereitgestellte Standardkonfiguration zurückgegriffen. Der Benutzer soll sämtliche Einstellungen selbstverständlich zu jeder Zeit ändern können.

Mockup: Um ein Gefühl für die Funktionalität zu bekommen die der Client haben soll, wurde mit Glade ein nichtfunktionales Mockup erstellt, um daraus die finalen Features abzuleiten.

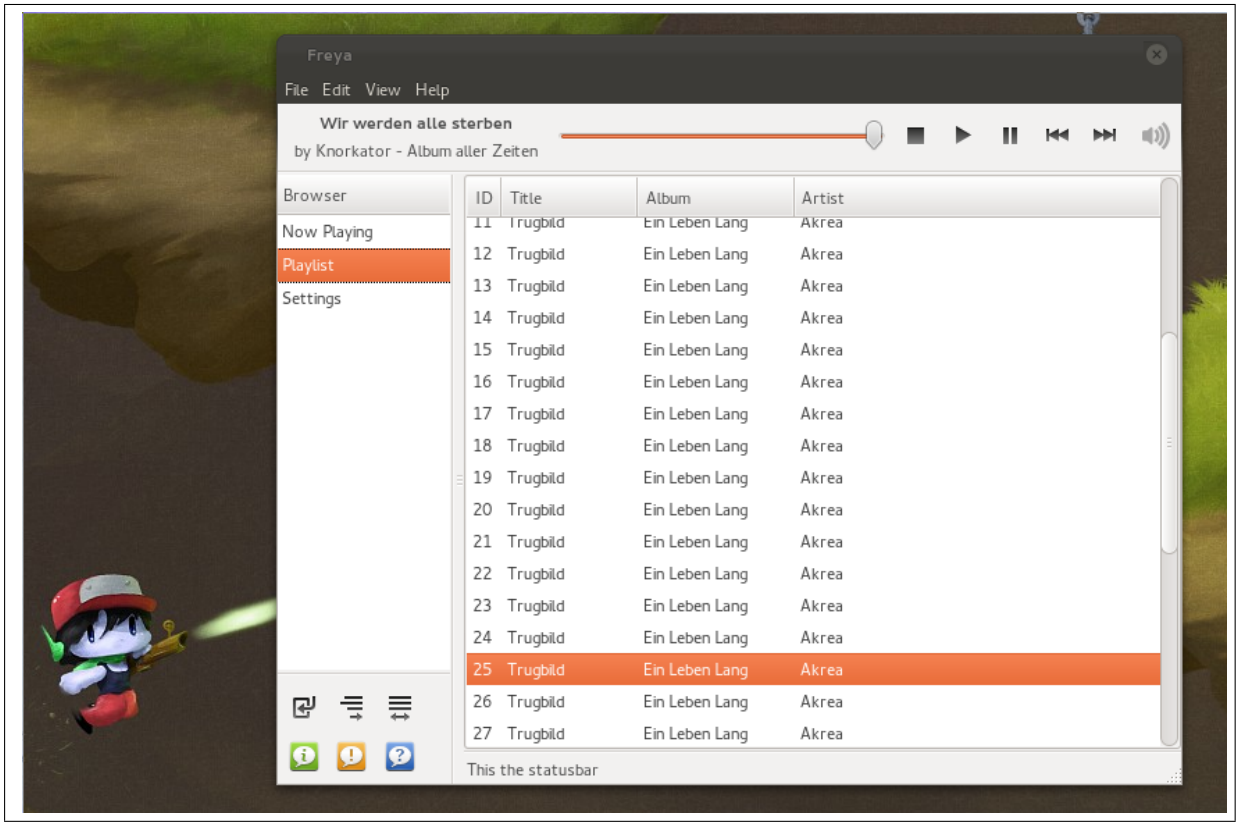


Abbildung 6.1.: Mockup des MPD Clients

Die Auflistung der Features erfolgt grob von links oben nach rechts unten. Auf eine Nummerierung der einzelnen Funktionen wurde verzichtet, da das Designdokument einen leicht anderen Aufbau hat, was das referenzieren erschwert.

6.4.1. Menü

Überall Anzeige der Keyshortcuts, für jedes relevante Element.

- File
  - ,,Connect"; Ausgegraut wenn bereits verbunden
  - ,,Disconnect", Ausgegraut wenn nicht verbunden
  - ,,Quit"
- ,,Playback" Anzeige mit Haken falls aktiviert
  - Next Song

  - Previous Song
  - Play
  - Stop
  - Consume, Single, Random, Repeat
- Help
  - About Dialog

6.4.2. Titelbar

- Anzeige des Musiktitels
  - Benutzerhinweis wenn Client nicht verbunden ist
- ,,Timeslider"
  - Zeigt die aktuelle Liedposition an.
  - Sprung an bestimmte Musikposition durch Ziehen möglich.
  - Bei Stop wird Slider auf 0 gesetzt, bei Pause bleibt dieser stehen.
- Steuerbuttons
  - Stopbutton - Stoppt Lied, setzt Timeslide zurück auf 0)
  - Pausebutton - zeigt ein ,,*Play Icon*" wenn pausiert und ein ,,*Pause Icon*" bei Wiedergabe
  - Previous - vorheriges Lied
  - Next - nächtes Lied
  - Volumebutton - Popupslider zum regeln der Lautstärke
- Untere Zeile
  - Anzeige von ,,by (*Artist*) on (*Album*) ((*Erscheinungsjahr - falls vorhanden*))"
  - Bei Musikstücken die nicht getaggt worden sind, und somit keine Artist oder Album-Informationen anbieten, soll der Name der Datei angezeigt werden.
  - Falls nicht connected oder nicht spielend soll ,,*Not Playing angezeigt*"

6.4.3. Sidebar

Die Sidebar befindet sich links und zeigt eine Liste verfügbarer Browser

- Bei Klick auf einem Browser wird er ausgewählt
- der Inhalt wird im Pane daneben angezeigt
- Sollte die Browserliste überfüllt sein wird ein Scrollbalken angezeigt

Nach einem Separator findet sich ein inaktives widget das den Song anzeigt der als nächstes spielen wird. Falls kein nächster Song wird eine entsprechende Nachricht angezeigt. Darunter findet sich 4 eindrückbare Buttons die eindrückbar sind:

- Repeat
- Consume
- Repeat
- Single

Falls diese Buttons beispielsweise in einem anderen Client aktiviert werden, so sollen die Änderungen automatisch durch die Eindrückung angezeigt werden.

6.4.4. Playlistmanager

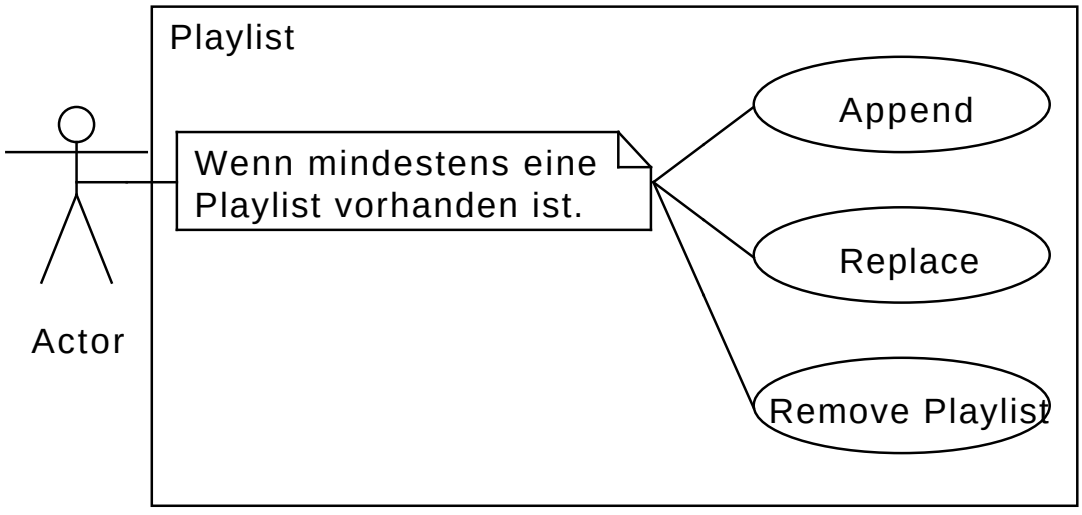


Abbildung 6.2.: Use Case Diagramm Playlist

Zeigt alle vorhandenen, auf dem Server gespeicherten Playlists in einer Liste. Dabei wird in der Liste der Playlistname und das letzte Änderungsdatum angezeigt. Die Namenszellen sind editierbar, so dass der Playlistname einfach geändert werden kann. Ein Rechtsklickmenü soll die folgenden Operationen bieten (siehe 6.2):

- **Append:** Fügt den Inhalt der Playlist der Queue am Ende hinzu
- **Replace:** Ersetzt den Inhalt der Queue mit dieser Playlist
- **Delete:** Entfernt die ausgewählten Playlists unwiderruflich

6.4.5. Databasebrowser

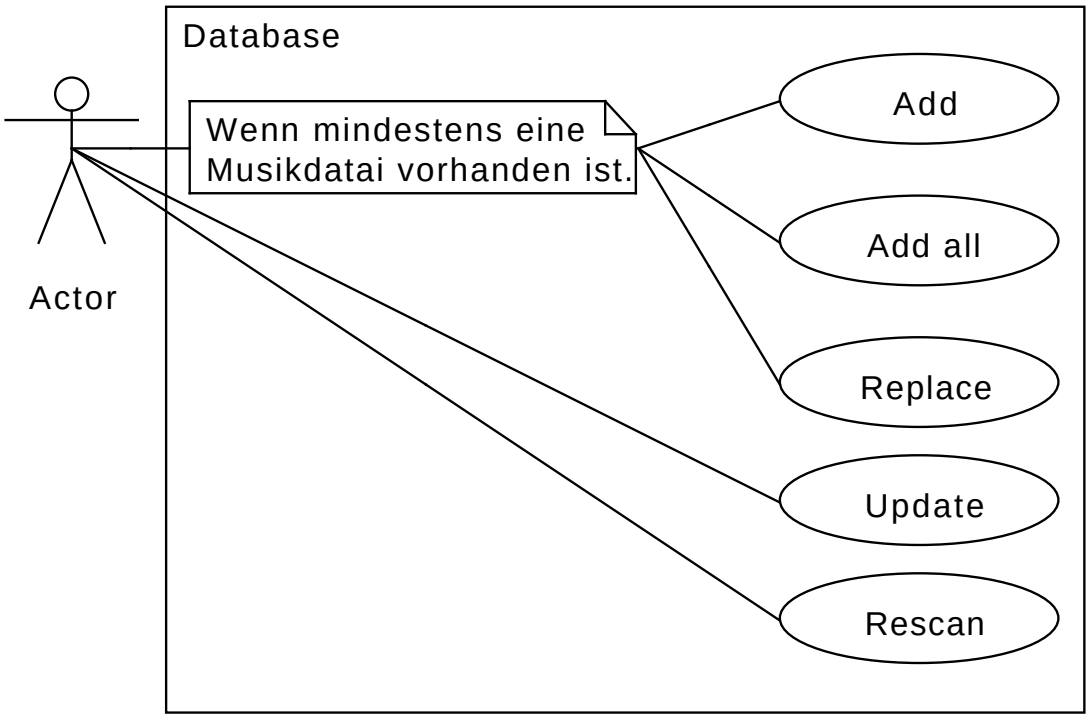


Abbildung 6.3.: Use Case Diagramm Database

Der Databasebrowser zeigt eine Visualisierung der Datenbank die an einen Filebrowser angelehnt ist. Es handelt sich hierbei nicht wirklich um ein Dateisystem, nur um eine Darstellung der Musikfiles die vom Server angeboten werden.

- Anzeige von Songs und Files durch unterschiedliche Icons
- Anzeige des "Dateinamenünter dem Icon

- Doppelklick oder Enter auf einen Ordner bewirkt ein Absteigen in diesen
- Doppelklick oder Enter auf ein Songfile bewirkt ein Hinzufügen dessen zur Ende der Queue
- ,,Backspace" geht ebenso ein Verzeichnis nach oben.
- Am unteren Rand befindet sich eine Steuerleiste:
  - **Homebutton:** Geht zum Wurzel-Verzeichnis zurück
  - **Zurückbutton:** Dasselbe wie Backspace
  - Das Suchfeld filtert die momentane Anzeige
  - ganze rechts zeigt ein Label den aktuellen Pfad an
- Ein Kontextmenü bietet folgende Optionen (siehe auch Abb.: 6.3):
  - **Add:** Fügt die ausgewählte Menge der Queue hinzu, rekursiv falls sich Verzeichnisse darunter befinden
  - **Add All:** Fügt alles, ungeachtet der auswahl der queue hinzu (performamtere Version von add)
  - **Replace:** Wie add, löscht aber vorher die Queue
  - **Update:** Weist den Server an die Datenbank zu aktualisieren, und neue/veränderte files zu aktualisieren
  - **Rescan:** Weist den Server die Datenbank zu aktualisieren; untersucht alle files von neuen (teuere Operation)

6.4.6. Queue

- Zeigt die aktuelle Warteschlange an
  - Künstler, Album und Songtitel
  - Spalten der Queue sind frei anordenbar
- Auswahl erfolgt durch linksklick (kombiniert mit shift/strg) oder durch Auswählen mit der Maus (,,Rubberbanding")
- Ein Rechtsklickmenü bietet die folgenden Möglichkeiten:
  - **Remove:** Entfernen der Songs aus der Queue
  - **Clear:** Entfernen aller Songs aus der Queue
  - **Save as playlist:** Die gesamte Queue wird als playlist abgespeichert, der Name der neuen Playlist wird durch einen Dialog abgefragt.

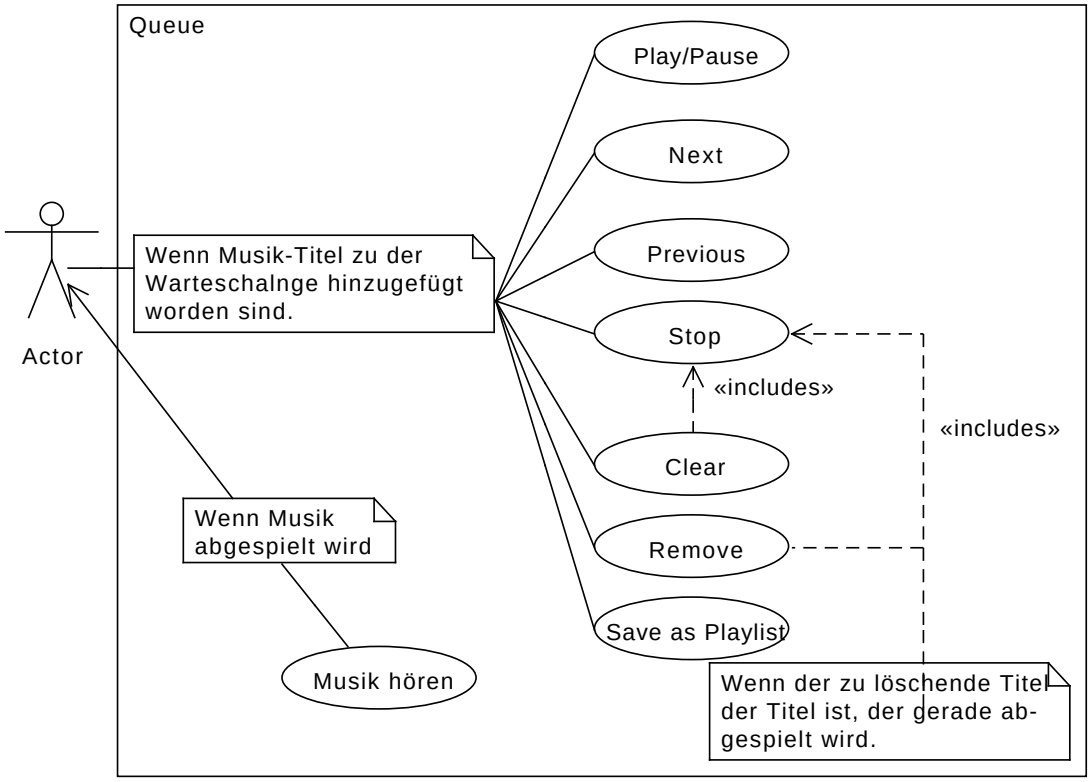


Abbildung 6.4.: Use Case Diagramm Queue

- Bei Änderungen des Server wird die Queue im Hintergrund geupdatet

6.4.7. Settingsbrowser

Die Einstellungen sollen in ein Tabbasiertes Layout eingebettet sein. Werden Änderungen vorgenommen, so werden sie nicht gleich gespeichert und übernommen. Es sollte daher eine Speicherfunktion geben und dazugehörig eine Undofunktion für die letzten Änderungen.
Falls der Client die Verbindung verliert, soll zum Settingsbrowser gesprungen werden, sodass der Benutzer entsprechende Änderungen machen kann. Aus diesem Grunde muss der Settingsbrowser auch ohne Verbindung voll funktionsfähig sein. Alle Settingstabs ändern die Werte nicht sofort:

- Sie werden erst persistent übernommen wenn der Benutzer die Konfiguration abspeichert
- Der ,,Undo"-Button setzt die Präsentation auf die letzen persistent gesetzten Werte
- ,,Reset" lädt überall die Defaultconfig

Innerhalb der Tabs sollen folgende Funktionen bereit gestellt werden:

- Der Benutzer kann Netzwerk-Einstellungen vornehmen
  - Server IP / Port
  - Avahi-Browser Anmerkung: Dieser soll eine Liste mit lokalen MPD Servern anzeigen, die sich über Zeroconf zu erkennen geben.
  - Autoconnect (Verbinde zum letzt eingestellten Server beim Start)
- Der Benutzer kann Playback-Einstellungen vornehmen
  - Crossfade in Sekunden (Weicher Übergang zwischen aktuellem und nächstem Lied)
  - Musik beim verlassen stoppen
- Der Benutzer kann Allgemein-Einstellungen vornehmen
  - Notifications(libnotify) nutzen, falls ja auch wie lange diese angezeigt werden
  - Tray-Icon anzeigen
- Der Benutzer soll eine Audioausgabeliste haben:
  - Innerhalb der Liste soll es Checkboxes geben um den Output entweder an oder auszuschalten
  - Dies soll nicht verfügbar sein wenn die Verbindung verloren geht

6.4.7.1. Fußleiste

Falls verbunden zeigt sie:

- Samplingrate in Khz
- Audiobitrate in Kbit
- Outputart (serverbedingt kann nur Stereo oder Mono angezeigt werden, 5.1 Audiofiles werden auch als Stereo dargestellt)
- Zeit aktuell von insgesamt
- Anzahl an Songs in der Datenbank
- Komplette Abspielzeit der Datenbank
- Lautstärke 0-100%

6.4.7.2. Sonstiges

- Nächster Song (Seitenleiste)

6.5. Produktdaten

6.5.1. Starten und Beenden

- Der Benutzer kann das System zu jedem Zeitpunkt starten.
- Der Benutzer kann das System zu jedem Zeitpunkt beenden.
- Beim ersten Start wird ein Standard-System-Zustand geladen.
- Beim Beenden wird der aktuelle System-Zustand gespeichert.
- Bei jedem weiteren Start wird der letzte System-Zustand geladen.

6.5.2. Abspielen von Musik (Buttons)

Der Benutzer kann

- Musik abspielen (Play)
- Musik stoppen (Stop)
- Musik pausieren (Pause)
- Musik vor und zurück schalten (Skip)
- Musik vor und zurück spulen (Seek)
- Musik zufällig abspielen (Random)
- Musik wiederholen (Repeat)
- Musik im Consume-Mode abspielen
- Musik im Single-Mode abspielen

6.5.3. Abspielen von Musik (Shortcuts)

Folgende Shortcut sollen in der finalen Version verfügbar sein:

Funktion	**Shortcut**
Play	Ctrl + G
Stop	Ctrl + S
Previous	Ctrl + P
Next	Ctrl + N
Random	Ctrl + Z
Single	Ctrl + Y
Repeat	Ctrl + R
Consume	Ctrl + T
Verbinden	Ctrl + C
Trennen	Ctrl + D
Beenden	Ctrl + Q

6.5.4. Administrator-Funktionen

Durch das Unix-artige System wird der Administrator-Zugriff geregelt. Sobald sich der Benutzer im Unix System als Administrator befindet, kann er auch den MPD-Client administrieren. Ein zusätzlicher Administrator-Modus wurde also nicht implementiert.

6.5.5. Suchen in der Queue

Eine einfache Textsuche zum finden von Titeln, Alben oder Interpreten innerhalb der Abspiellisten wurde implementiert. Dabei springt die Markierung des Textes beim eingeben von Zeichen in die Suche zu der ersten übereinstimmenden Artist in der Queue des Clients. Erst beim bestätigen der Eingabe im Suchfeld wird die Auswahl anhand einer Volltextsuche gefiltert. Eine Suche nach ,,Kno" würde so erst zum Artist ,,Knorkator" springen, und bei Bestätigung werden alle nicht zutreffenden Songs gefiltert.

- Der Benutzer kann seine Queue durchsuchen
- Der Benutzer kann sein Dateisystem durchsuchen

6.5.6. Statistik

- Der Benutzer kann eine gesamt Statistik einsehen
  - Anzahl der Interpreten
  - Anzahl der Alben

- Anzahl der Lieder
- Musiklänge der Datenbank
- Abspielzeit
- Zeit Online bzw. mit MPD verbunden
- Letztes Datenbank-Update

6.5.7. Persönliches Profil

Da die Software auf Unixoide Systeme beschränkt ist, wurde keine Profil-Verwaltung implementiert. Die verschiedenen Profile werden durch die verschiedenen User des gesamten Betriebssystems definiert und differenziert. Für spätere Versionen könnte ergänzend auch ein Serveraddressbuch implementiert werden.

6.5.8. Mehrfachstart des Clients

Gegen mehrfaches Starten ist der Client nicht abgesichert. Die einzige Schnittmengen die mehrere Instanzen des Clients sich teilen liegt in der persistenten Datenspeicherung des Logs. Werden die Clients zeitversetzt gestartet sollte hier allerdings kaum etwas passieren.

6.5.9. Persönliche Datenbank

Eine persönliche Datenbank ist lokal nicht vorhanden. Die Datenbank des Benutzers befindet sich auf dem MPD-Server. Einzig und alleine modulare Erweiterungen des MPD-Clients können lokale Datenbank-Implementierungen erfordern, um beispielsweise zusätzliche Informationen wie ein ,,Rating" zu speichern, oder die Suche zu beschleunigen.

6.5.10. Persönliche Einstellungen

Client Einstellungen werden lokal gespeichert, außerdem ist stets eine Defaultconfig vorhanden, falls die des Dateisystems defekt ist. Die Konfigurationsdatei wird nach dem XDG-Standard in */.config/freya/config.xml* gespeichert. Den selben Speicherplatz wählt auch die Logdatei (*/.config/freya/log.txt*). Sollten nur einzelne Werte in der Konfigurationsdatei nicht vorhanden sein so wird nachgeschaut ob die Defaultconfig diese Werte bereitstellt und es wird versucht sie von dort zu laden. So ist für valide Wert stets abgesichert, dass mindestens ein Wert vorliegt.

6.6. Qualitätsanforderungen

Die Software soll natürlich von hoher Qualität sein. Hierfür sollen folgende Anforderungen erfüllt werden:

6.6.1. Korrektheit

Die Software muss möglichst fehlerfrei und korrekt sein. Es wurden Testszenarien und Testfälle erstellt, um Fehler zu finden und auszubessern. Aber auch wenn nach Veröffentlichung der Software ein Fehler gefunden werden sollte, wird dieser sofort ausgebessert. Bei schwerwiegenden Fehlern werden die Nutzer direkt auf den Fehler aufmerksam gemacht.

6.6.2. Wartbarkeit

Der Wartungsaufwand der Software ist gering bis gar nicht vorhanden. Ändert sich die Umgebungssoftware (z.B. der MPD-Server), dann sind die Änderungen so geringfügig bzw. trivial, dass sie den MPD-Client nicht beeinflussen sollten. Fehler der Software (sollten Fehler auftreten) wären leicht analysiert bzw. prüfbar und natürlich auch leicht zu beheben. Zur Wartbarkeit gehört ebenso die Modularität, d.h. die Software ist technisch so realisiert, dass sie leicht erweitert werden kann - Stichwort ModelViewController (MVC).

6.6.3. Zuverlässigkeit

Das System funktioniert und reagiert tolerant auf fehlerhafte Eingaben bzw. fehlerhafte Benutzung. Das Programm funktioniert sieben Tage die Woche und 24 Stunden am Tag und muss nicht abgeschaltet werden.

6.6.4. Effizienz

Der MPD-Client funktioniert möglichst effizient, d.h. das Programm ist schnell geladen und Eingaben des Benutzers werden praktisch sofort ausgeführt. Es gibt so gut wie keine Wartezeiten, jedenfalls sind diese so genannten Reaktionszeiten für den Benutzer nicht merkbar. Selbst bei sehr großen Musik-Datenbanken und Playlists benötigt das Programm kaum Rechenzeit und sonstige Hardwareressourcen.

6.6.5. Benutzbarkeit

Die Software ist leicht verständlich und intuitiv bedienbar. Nötige Kenntnisse zur Nutzung des MPD-Clients sind leicht zu erlernen. Hier wird allerdings davon ausgegangen dass der MPD Server bereits fertig eingerichtet ist. Sollte dies nicht der Falls sein, so sind Fähigkeiten in der Kommandozeile durchaus hilfreich.

6.6.6. Design

Das Design soll ansprechend und modern sein, allerdings wenn es Konflikte zwischen technischer Umsetzung und Design oder Effizienz und Design geben sollte, ist stets im Interesse der

technischen Umsetzung bzw. der Effizienz zu entscheiden.

6.6.7. Hardware

Minimale Hardwareanforderungen: 500 Mhz, 512MB Ram, Festplattenspeicher ¡ 20MB Empfohlene Hardwareanforderungen: 1 Ghz, 512MB Ram, Festplattenspeicher ¡ 20MB

6.6.8. Orgware und Entwicklungsumgebung

- CMake (Buildsystem)
- g++ (C++ Compiler)
- Valgrind (Memorydebugger)
- git (Hosting auf Github) [1]
- Glade (GUI-Designer)
- doxygen (Interne Dokumentationsgenerierung)
- Devhelp (Dokumentationsbrowser)
- Unixodes Betriebssystem
- MPD-Server
- Avahi-Browser
- gcc (Compiler)
- libmpdclient
- gtkmm libraries

[1] https://github.com/studentkittens/Freya

6.7. Globale Testszenarien und Testfälle

6.7.1. Cxxtest

Als Testframework wurde CxxTest ausgewählt. Die Gründe für diese Entscheidung werden gut von der offiziellen Definition zusammengefasst:

> [2] CxxTest is a JUnit/CppUnit/xUnit-like framework for C/C++.
> It is focussed on being a lightweight framework that is well suited for integration into embedded systems development projects.
> CxxTest's advantages over existing alternatives are that it:
>
> - Doesn't require RTTI
> - Doesn't require member template functions
> - Doesn't require exception handling
> - Doesn't require any external libraries (including memory management, file/-console I/O, graphics libraries)
> - Is distributed entirely as a set of header files (and a python script).
> - Doesn't require the user to manually register tests and test suites
>
> This makes it extremely portable and usable.

6.7.2. Testfälle

Für Teile des Programmes die nicht vom Testprotokoll erfasst werden und automatisch gestestet werden, sollen Testfälle mit Cxxtest geschrieben werden.

6.7.3. Testprotokoll

Um Fehler aufzuspüren, die die grafische Oberfläche betreffen, wurde ein Testprotokoll erstellt, in dem zunächst alle möglichen Funktionen der grafischen Oberfläche aufgelistet werden. Außerdem müssen diese Funktionen mit anderen Funktionen kombiniert und mehrfach ausgeführt werden. Zu jedem dieser Fälle ist ein zu erwartendes Ergebnis festzulegen und anschließend zu überprüfen ob das erwartete Ergebnis eingetroffen ist. Das eingetroffene Ergebnis ist ebenfalls zu protokollieren. Es wurden jeweils die Buttons, sowie die Shortcuts geprüft.

[2] http://cxxtest.tigris.org/

6.7.3.1. Abspielfunktionen

Einfache Ausführung:

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Play	Musik spielt ab. Play wird zu Pause.	Ja
Pause	Musik pausiert. Pause wird zu Play.	Ja
Next	Nächstes Lied abspielen	Ja
Previous	Vorheriges Lied abspielen	Ja
Stop	Beende abspielen Pause wird zu Play.	Ja
Skipping	An Liedposition springen	Ja
Random	Musik der Queue zufällig abspielen	Ja
Repeat	Ein Lied wiederholen	Ja
Repeat all	Queue wiederholen	Ja
Consume Mode	Ein abgespieltes Lied entfernen	Ja
Single Mode	Ein Lied abspielen, dann Stoppen	Ja

Kombinierte Ausführung:

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Play, Stop, Play(1)	Musik spielt ab. Musik stoppt. Musik spielt ab.	Ja
Play, Pause, Play(2)	Musik spiel ab. Musik pausiert. Musik spiel ab.	Ja
Next, Next(3)	Skip weiter. Skip weiter.	Ja
Previous, Previous(4)	Skip zurück. Skip zurück	Ja
Next, Previous(5)	Skip weiter. Skip zurück	Ja
Previous, Next(6)	Skip zurück. Skip weiter	Ja
Random, Repeat all	Musik der Queue zufällig abspielen Queue wiederholen	Ja
Random, Consume Mode	Musik der queue zufällig abspielen Ein abgespieltes Lied entfernen	Ja
Random, Single Mode	Musik der Queue zufällig abspielen Ein Lied abspielen, dann stoppen	Ja
Consume Mode, Single Mode	Ein abgespieltes Lied entfernen Ein Lied abspielen, dann Stoppen	Ja
Consume Mode, Repeat all	Kann nur einmal durchlaufen	Ja
Random, 1	Musik der Queue zufällig abspielen 1	Ja
Random, 2	Musik der Queue zufällig abspielen 2	Ja
Random, 3	Musik der Queue zufällig abspielen 3	Ja
Random, 4	Musik der Queue zufällig abspielen 4	Ja
Random, 5	Musik der Queue zufällig abspielen 5	Ja
Random, 6	Musik der Queue zufällig abspielen 6	Ja
Repeat all, 1	Queue wiederholen 1	Ja
Repeat all, 2	Queue wiederholen 2	Ja
Repeat all, 3	Queue wiederholen 3	Ja
Repeat all, 4	Queue wiederholen 4	Ja

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Repeat all, 5	Queue wiederholen 5	Ja
Repeat all, 6	Queue wiederholen 6	Ja
Consume Mode, 1	Ein abgespieltes Lied entfernen 1	Ja
Consume Mode, 2	Ein abgespieltes Lied entfernen 2	Ja
Consume Mode, 3	Ein abgespieltes Lied entfernen 3	Ja
Consume Mode, 4	Ein abgespieltes Lied entfernen 4	Ja
Consume Mode, 5	Ein abgespieltes Lied entfernen 5	Ja
Consume Mode, 6	Ein abgespieltes Lied entfernen 6	Ja
Single Mode, 1	Ein Lied abspielen, dann Stoppen 1	Ja
Single Mode, 2	Ein Lied abspielen, dann Stoppen 2	Ja
Single Mode, 3	Ein Lied abspielen, dann Stoppen 3	Ja
Single Mode, 4	Ein Lied abspielen, dann Stoppen 4	Ja
Single Mode, 5	Ein Lied abspielen, dann Stoppen 5	Ja
Single Mode, 6	Ein Lied abspielen, dann Stoppen 6	Ja

Im folgenden wird auf das Protokoll der kombinierten Ausführung referenziert.

Mehrfache Ausführung:

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Fall 1 x 10	Fall 1 x 10	Ja
Fall 2 x 10	Fall 2 x 10	Ja
Fall 3 x 10	Fall 3 x 10	Ja
Fall 4 x 10	Fall 4 x 10	Ja
Fall 5 x 10	Fall 5 x 10	Ja
Fall 6 x 10	Fall 6 x 10	Ja
Fall 12 x 10	Fall 12 x 10	Ja
Fall 13 x 10	Fall 13 x 10	Ja
Fall 14 x 10	Fall 14 x 10	Ja
Fall 15 x 10	Fall 15 x 10	Ja
Fall 16 x 10	Fall 16 x 10	Ja
Fall 17 x 10	Fall 17 x 10	Ja
Fall 18 x 10	Fall 18 x 10	Ja
Fall 19 x 10	Fall 19 x 10	Ja
Fall 20 x 10	Fall 20 x 10	Ja
Fall 21 x 10	Fall 21 x 10	Ja
Fall 22 x 10	Fall 22 x 10	Ja
Fall 23 x 10	Fall 23 x 10	Ja
Fall 24 x 10	Fall 24 x 10	Ja
Fall 25 x 10	Fall 25 x 10	Ja
Fall 26 x 10	Fall 26 x 10	Ja
Fall 27 x 10	Fall 27 x 10	Ja
Fall 28 x 10	Fall 28 x 10	Ja
Fall 29 x 10	Fall 29 x 10	Ja
Fall 30 x 10	Fall 30 x 10	Ja
Fall 31 x 10	Fall 31 x 10	Ja
Fall 32 x 10	Fall 32 x 10	Ja
Fall 33 x 10	Fall 33 x 10	Ja
Fall 34 x 10	Fall 34 x 10	Ja
Fall 35 x 10	Fall 35 x 10	Ja

6.7.3.2. Queue-Funktionen

Einfache Ausführung

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Remove	Ein Lied aus Queue entfernen	Ja
Clear	Alle Lieder aus Queue entfernen	Ja
Save as Playlist	Queue als Playlist speichern	Ja
Suchen	Nach eingegebenem Wort suchen	Ja

Kombinierte Ausführung

Kombinierte Ausführung der Funktionen der Queue machen nicht wirklich viel Sinn da z.B. die Funktion Clear die Queue löscht. Die einzige Kombination die Sinn macht getestet zu werden sind die folgenden:

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Suchen, Remove	Nach eingegebenem Wort suchen Ein Lied aus der Queue entfernen	Ja
Suchen, Hinzufügen von gesuchten Songs aus DB	Doppelte Anzeige der Songs in Queue	Ja

Mehrfache Ausführung

Die mehrfache Ausführung ist ähnlich unsinnig wie die der kombinierten Ausführung. Mehrmals hintereinander die Queue löschen ist nicht möglich. So bleibt wieder nur ein Testfall zu prüfen:

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Suchen, Remove x 10	Nach eingegebenem Wort suchen Ein Lied aus der Queue entfernen x 10	Ja

6.7.3.3. Playlist-Funktionen

Einfache Ausführung

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Hinzufügen	Playlist hinzufügen	Ja
Ersetzen	Playlist ersetzen	Ja
Playlist entfernen	Playlist löschen	Ja

Kombinierte Ausführung

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Hinzufügen, Hinzufügen	Playlist hinzufügen Playlist hinzufügen	Ja
Ersetzen, Ersetzen	Playlist ersetzen Playlist ersetzen	Ja
Entfernen, Entfernen	Playlist entfernen Playlist entfernen	Ja
Hinzufügen, Entfernen	Playlist hinzufügen Playlist löschen	Ja
Ersetzen, Entfernen	Playlist ersetzen Playlist entfernen	Ja

Im folgenden wird auf das Protokoll der kombinierten Ausführung referenziert.

Mehrfache Ausführung:

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Fall 1 x 10	Fall 1 x 10	Ja
Fall 2 x 10	Fall 2 x 10	Ja
Fall 3 x 10	Fall 3 x 10	Ja
Fall 4 x 10	Fall 4 x 10	Ja
Fall 5 x 10	Fall 5 x 10	Ja

6.7.3.4. Dateibrowser-Funktionen

Einfache Ausführung

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Hinzufügen	Zur Queue hinzufügen	Ja
Alle Hinzufügen	Alle zur Queue hinzufügen	Ja
Ersetzen	Queue durch Auswahl ersetzen	Ja
Aktualisieren	Dateibrowser aktualisieren	Ja
Neu einlesen	Dateibrowser neu einlesen	Ja
Suchen	Nach eingegebenem Wort suchen	Ja

Kombinierte Ausführung:

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Hinzufügen Hinzufügen	Zur Queue hinzufügen Zur Queue hinzufügen	Ja
Alle Hinzufügen Alle hinzufügen	Alle zur Queue hinzufügen Alle zur Queue hinzufügen	Ja
Ersetzen Ersetzen	Queue durch Auswahl ersetzen Queue durch Auswahl ersetzen	Ja
Aktualisieren Aktualisieren	Dateibrowser aktualisieren Dateibrowser aktualisieren	Ja
Neu einlesen Neu einlesen	Dateibrowser neu einlesen Dateibrowser neu einlesen	Ja
Suchen Suchen	Nach eingegebenem Wort suchen Nach eingegebenem Wort suchen	Ja
Hinzufügen Alle Hinzufügen	Zur Queue hinzufügen Alle zur Queue hinzufügen	Ja
Hinzufügen Ersetzen	Zur Queue hinzufügen Queue durch Auswahl ersetzen	Ja
Hinzufügen Aktualisieren	Zur Queue hinzufügen Dateibrowser aktualisieren	Ja
Hinzufügen Neu einlesen	Zur Queue hinzufügen Dateibrowser neu einlesen	Ja

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Hinzufügen Suchen	Zur Queue hinzufügen Nach eingegebenem Wort suchen	Ja
Alle Hinzufügen Ersetzen	Alle zur Queue hinzufügen Queue durch Auswahl ersetzen	Ja
Alle Hinzufügen Aktualisieren	Alle zur Queue hinzufügen Dateibrowser aktualisieren	Ja
Alle Hinzufügen Neu einlesen	Alle zur Queue hinzufügen Dateibrowser neu einlesen	Ja
Alle Hinzufügen Suchen	Alle zur Queue hinzufügen Nach eingegebenem Wort suchen	Ja
Ersetzen Aktualisieren	Queue durch Auswahl ersetzen Dateibrowser aktualisieren	Ja
Ersetzen Neu einlesen	Queue durach Auswahl ersetzen Dateibrowser neu einlesen	Ja
Ersetzen Suchen	Queue durch Auswahl ersetzen Nach eingegebenem Wort suchen	Ja
Aktualisieren Neu einlesen	Dateibrowser aktualisieren Dateibrowser neu einlesen	Ja
Aktualisieren Suchen	Dateibrowser aktualisieren Nach eingegebenem Wort suchen	Ja
Neu einlesen Suchen	Dateibrowser neu einlesen Nach eingegebenem Wort suchen	Ja

Im folgenden wird auf das Protokoll der kombinierten Ausführung referenziert.

Mehrfache Ausführung

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Fall 1 x 10	Fall 1 x 10	Ja
Fall 2 x 10	Fall 2 x 10	Ja
Fall 3 x 10	Fall 3 x 10	Ja
Fall 4 x 10	Fall 4 x 10	Ja
Fall 5 x 10	Fall 5 x 10	Ja
Fall 6 x 10	Fall 6 x 10	Ja
Fall 7 x 10	Fall 7 x 10	Ja
Fall 8 x 10	Fall 8 x 10	Ja
Fall 9 x 10	Fall 9 x 10	Ja
Fall 10 x 10	Fall 10 x 10	Ja
Fall 11 x 10	Fall 11 x 10	Ja
Fall 12 x 10	Fall 12 x 10	Ja
Fall 13 x 10	Fall 13 x 10	Ja
Fall 14 x 10	Fall 14 x 10	Ja
Fall 15 x 10	Fall 15 x 10	Ja
Fall 16 x 10	Fall 16 x 10	Ja
Fall 17 x 10	Fall 17 x 10	Ja
Fall 18 x 10	Fall 18 x 10	Ja
Fall 19 x 10	Fall 19 x 10	Ja
Fall 20 x 10	Fall 20 x 10	Ja
Fall 21 x 10	Fall 21 x 10	Ja

6.7.3.5. Statistik

Testfall	Erwartetes Ergebnis	Ergebnis eingetroffen?
Abgleich der Informationen mit MPD	Passend	Ja
Hinzufügen von Song zu DB	Entsprechende Anpassung der Information	Ja

6.7.3.6. Einstellungen

Einfache Ausführung

Testfall	Erwartetes Ergebnis	Ergebnis eingetroffen?
Zeige Liste	Zeige Avahi Liste	Ja

Kombinierte Ausführung

Testfall	Erwartetes Ergebnis	Ergebnis eingetroffen?
Fall 1 x 10	Fall 1 x 10	Ja

Es existieren keine Buttons oder Shortcuts die kombiniert werden könnten.

6.7.3.7. Lautstärke

Einfache Ausführung

Testfall	Erwartetes Ergebnis	Ergebnis eingetroffen?
Lautstärke erhöhen	Lautstärke erhöhen	Ja
Lautstärke verringern	Lautstärke verringern	Ja

Kombinierte Ausführung

Testfall	Erwartetes Ergebnis	Ergebnis eingetroffen?
Lautstärke erhöhen Lautstärke verringern	Lautstärke erhöhen Lautstärke verringern	Ja

Im folgenden wird auf das Protokoll der kombinierten Ausführung referenziert.

Mehrfache Ausführung

Testfall	Erwartetes Ergebnis	Ergebnis eingetroffen?
Fall 1 x 10	Fall 1 x 10	Ja

6.7.3.8. Sonstiges

Einfache Ausführung

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Verbinden	Verbindung zum MPD-Server	Ja
Trennen	Verbindung zum Server trennen	Ja
Beenden	MPD-Client beenden	Ja

Kombinierte Ausführung

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Verbinden Verbinden	Verbindung zum MPD-Server Verbindung zum MPD-Server	Ja
Verbinden Trennen	Verbindung zum MPD-Server Verbindung zum Server trennen	Ja
Verbinden Beenden	Verbindung zum MPD-Server MPD-Client beenden	Ja
Trennen Beenden	Verbindung zum Server trennen MPD-Client beenden	Ja

Im folgenden wird auf das Protokoll der kombinierten Ausführung referenziert.

Mehrfache Ausführung

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Fall 1 x 10	Fall 1 x 10	Ja
Fall 2 x 10	Fall 2 x 10	Ja
Fall 3 x 10	Nur 1 x ausführbar	Ja
Fall 4 x 10	Nur 1 x ausführbar	Ja

Teil V.

Designdokument

7. Software Design

7.1. Einführung

7.2. ,,Das Problem"

Da die grundlegende Funktionsweise des MPD Servers auf einer Client-Server-Architektur beruht, muss der MPD Client verschiedene Kommandos, wie zum Beispiel *play, pause, listplaylists* etc., an den Server schicken und zur gleichen Zeit aber auch auf Änderungen reagieren können, d.h. zum Beispiel wenn sich die Lautstärke ändert, da jederzeit auch andere Clients oder Server den internen Zustand des MPD ändern können. Diese Änderungen müssen auch anderen Programmteilen bekannt gemacht werden. Für die Realisierung eignet sich hier das Observer Pattern gut. [1]
Der Client sollte im ,,idle"-Mode möglichst keine Ressourcen verschwenden und auch beim trennen und verbinden die entsprechenden Änderungen anderen Teilen des Programms mitteilen können.

Das MPD Protokoll [2] bietet folgende Möglichkeiten das zu realisieren.

Periodisch (zB. alle 500ms) das ,,*status*" Kommando absetzen und nach Bedarf auch Kommandos wie ,,*currentsong*" senden.
Problem: Bei langsamen Netzwerkverbindungen erzeugt dies unnötige Netzwerklast. Prinzipiell würde sich auf diese Art jedoch z.B. die Musik Bitrate anzeigen lassen. Es ist jedoch ein ein wenig komfortablerer Weg, da man ansonsten das Rad neu erfinden und manuell heraus finden müsste, was genau sich eigentlich geändert hat.

Nutzung der ,,idle" und ,,noidle" commands: ,,*idle*" versetzt die Verbindung zum Server in einen Schlafzustand. Sobald Events wie ,,*player*" (wird beispielsweise durch pausieren getriggert) eintreten, wacht die Verbindung aus diesem Zustand auf und sendet an den Client eine Liste der Events die aufgetreten sind:

[1] http://de.wikipedia.org/wiki/Observer_(Entwurfsmuster)
[2] http://www.musicpd.org/doc/protocol/index.html

```
1 changed: player
2 changed: mixer
3 ...
4 OK
```

Abbildung 7.1.: Eine Beispielantwort des MPD Servers

Allerdings gibt es hier eine weitere Einschränkung: Während die Verbindung im idle mode ist, kann kein reguläres Kommando wie ,,*play*" gesendet werden! Sollte man es doch tun wird man augenblicklich vom Server getrennt. Die einzige Möglichkeit aus dem idle mode aufzuwachen ist das ,,*noidle*" Kommando, das gesendet werden kann, während die Verbindung schlafen gelegt wurde. Jedoch gibt es auch hier ein Problem, denn das ,,*idle*" Kommando blockiert, sprich es sendet kein ,,OK" an den Sender zurück. Ein Warten auf dieses ,,OK" würde mit dem Wunsch eine in der Zwischenzeit bedienbare Oberfläche zu haben kollidieren.

Prinzipiell gibt es 2 Möglichkeiten dieses Problem zu lösen:

- Man hält zwei Verbindungen zum Server: eine, die Kommandos sendet, und eine, die stets im ,,*idle*" mode liegt. Für die Realisierung müssten Threads herangezogen werden. Ein Thread würde dann im Hintergrund auf Events lauschen, der andere würde zum Abschicken der Kommandos benutzt werden. *Problem:* Es müssen 2 Verbindungen gehandelt werden, was wiederum ein Mehraufwand an Code bedeutet. Desweiteren werden Threads benötigt die auch in anderen Bereichen des Programms Lockingmechanismen bedeuten würden.

- Man hält eine asynchrone Verbindung zu dem Server. Diese kann das ,,*idle*" Kommando zum Server schicken, gibt aber sofort die Kontrolle dem Aufrufer zurück. Um nun eine Liste der events zu bekommen setzt man einen ,,*Watchdog*" auf die asynchrone Verbindung an (Vergleiche dazu den Systemaufruf ,,man 3 poll"). Da poll() ebenfalls den aufrufenden Prozess blockiert, wird die Glibfunktion[3] Glib::signal_io() benutzt, das sich in den laufenden MainLoop[4] einhängt und eine Callbackfunktion aufruft sobald auf der Verbindung etwas Interessantes passiert. Da während des Wartens der MainLoop weiterarbeitet, bleibt die GUI (und andere Module) aktiv und benutzbar.
  Problem: Vor dem Senden eines Kommandos wie ,,*play*", muss der idle mode verlassen werden. *Lösung:* Man kann das ,,*noidle*" Kommando zum Verlassen senden und nach dem Absenden des eigentlichen Kommandos wieder den idle-mode betreten.

[3] GLib ist eine Utility Bibliothek für C die von Gtk+ genutzt wird
[4] Siehe MainLoop im Glossar

Zum Verständnis der Problematik wird hier eine telnetsession gezeigt. Über telnet kann man das MPD Protokoll ,,interaktiv" benutzen und ausprobieren:

```
(master) $ telnet localhost 6600
Trying ::1...
Connected to localhost.
OK MPD 0.16.0      # Der Server antwortet bei verbindungsau
                   # stets mit einem OK und der Versionsnummer
pause              # Wir senden das 'pause' kommando zum
                   # pausieren des aktuellen liedes
OK                 # Der Server fuehrt es aus und antwortet
                   # mit einem OK
play               # Wir tun dasselbe mit dem 'play' command.
OK
idle               # Wir sagen dem server dass wir die
                   # Verbindung schlafen legen wollen...
changed: player    # Er returned aber sofort da seit dem
                   # Verbindungsaufbau etwas geschehen ist.
changed: mixer     # Und zwar wurde der Player pausiert,
                   # und das volume geaendert.
OK                 # Das Ende des idlemodes wird wieder
                   # mit OK angezeigt.
idle               # Probieren wir es noch einmal..
                   # Er antwortet nicht mit OK sondern
                   # schlaeft jetzt. Wuerden wir in
                   # einem anderen client pausieren
                   # So wuerde er hier aufwachen.
noidle             # Um aus den idlemode vorher aufzuwachen
                   # senden wir das noidle command
OK                 # OK sagt uns dass alles okay ist.
idle               # Probieren wir mal ein command zu senden
                   # waehrend die verbindung idlet:
play               # zum Beispiel das play command... als
                   # antwort wird die verbindung geschlossen:
Connection closed by foreign host.
$ Freya git:(master) $ echo 'ende.'
```

Die Idee zu dieser Implementierung (speziell das Benutzen einer asynchronen Verbindung), kommt von ,,*ncmpc*", der mehr oder minder offiziellen Referenzimplementierung des MPD Mit-Authors *Max Kellermann*. Vergleiche ncmpc quellcode: *src/gidle.c* und *src/mpdclient.c*

7.3. Namespace-Übersicht

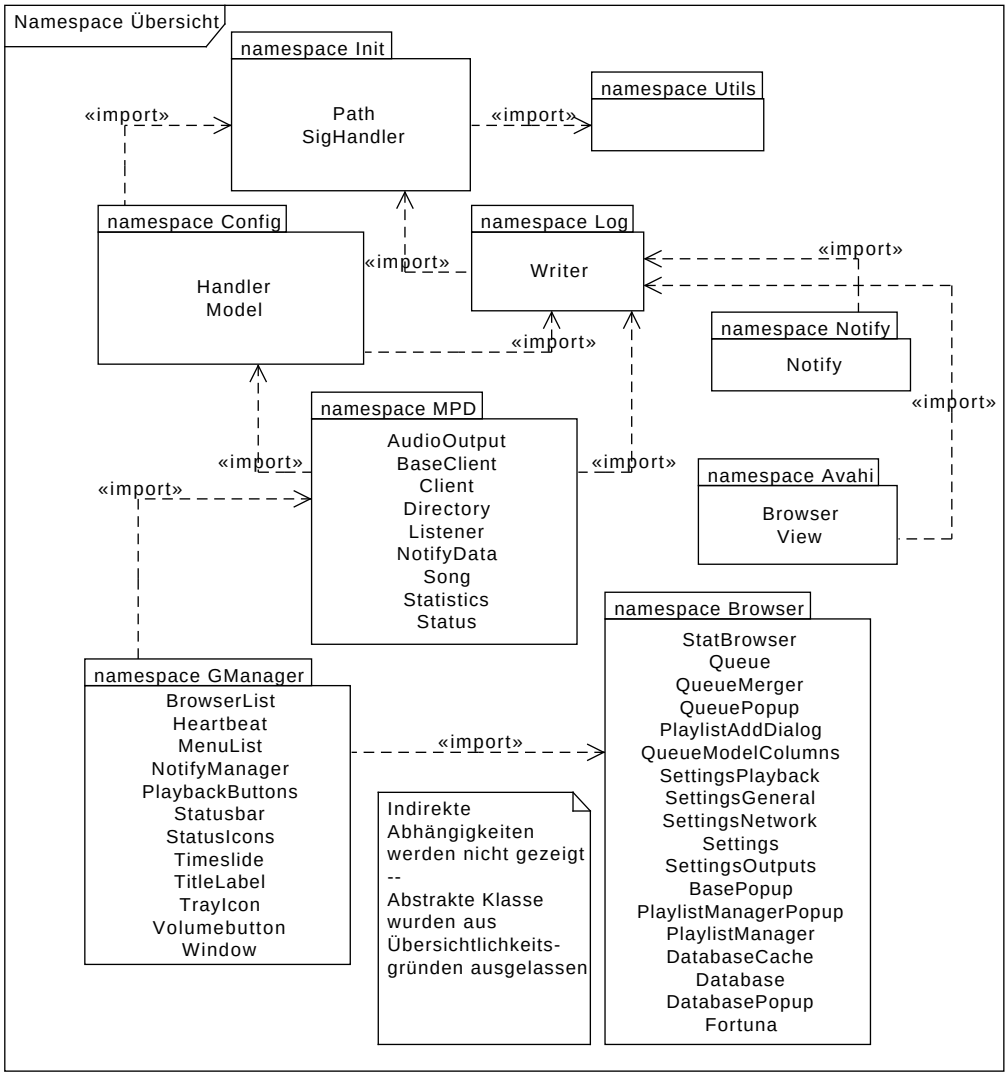


Abbildung 7.2.: Die Namespaces im Überblick

7.4. Aufbau der Anwendung

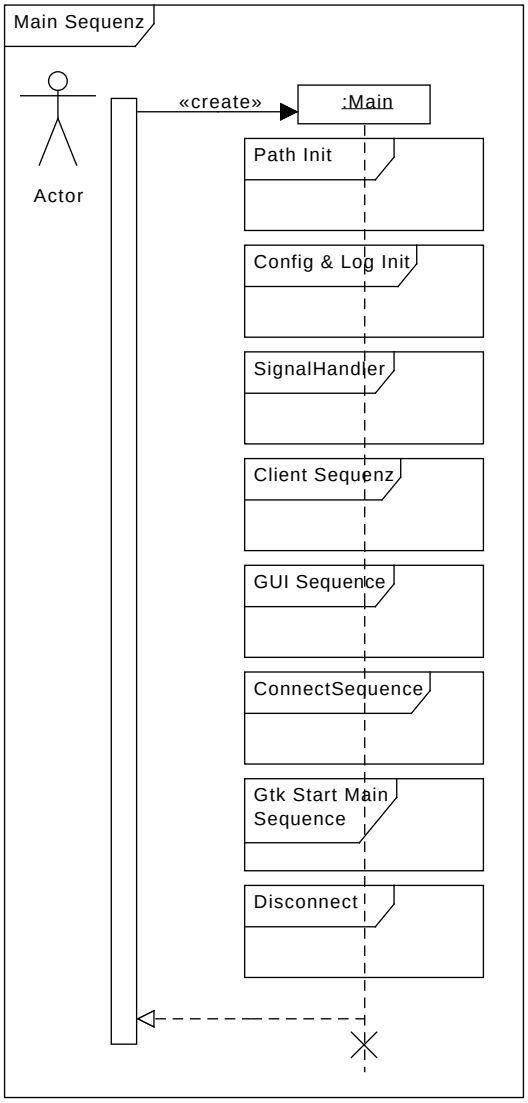


Abbildung 7.3.: Die Initialisierungsphase

7.5. Utils und Writer

7.5.1. Utils

Im *Utils Namespace* sollen sollen sich folgende Hilfsfunktionen zur Zeit/Datumsumrechnung befinden:

- Umrechnung von Sekunden in einen ,,Dauer-String Bsp" : ,,4 hours 2 minutes 0 seconds"

```
Glib::ustring seconds_to_duration(unsigned long);
```

- Umrechnung Sekunden in einen Timestamp, Bsp: ,,2011-04-02"

```
Glib::ustring seconds_to_timestamp(const long);
```

- Umwandlung eines Integer Wertes in einen String

```
std::string int_to_string(int num);
```

Diese grundlegenden Funktionen sollen ausgelagert werden damit, sie von mehreren Klassen verwendet werden können und um Redundanzen im Code zu vermeiden.

7.5.2. Logging

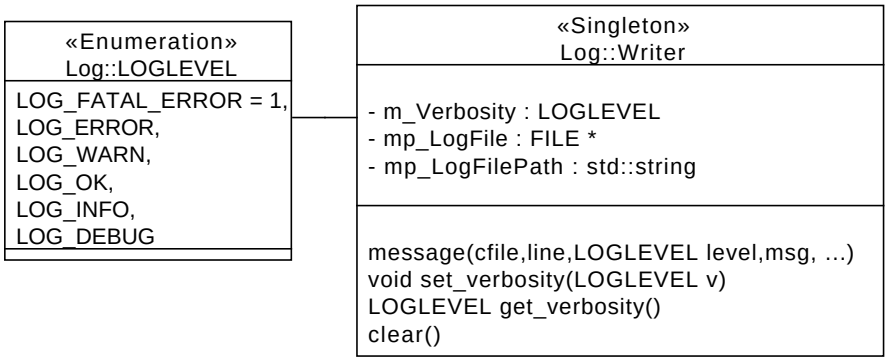


Abbildung 7.4.: Klassendiagramm zu Log::Writer

Im *Log Namespace* befindet sich die Writer Klasse, welche auch als sog. ,,*Utils-Klasse*" Funktionen für alle anderen Klassen in Namespace bereitstellt. Bei dieser Klasse wurde das Singleton

Pattern gewählt, damit sie leicht von ,,überall" aus erreichbar ist, ohne dass vorher explizit eine Instanzierung statt finden muss. Die Logdatei wird im Konfigurationsverzeichnis von Freya als log.txt abgespeichert. Im folgenden Dokument gibts dazu mehr Informationen. Die Logklasse dient zur Protokollierung von Fehlern und besonderen Ereignissen.

Folgende selbsterklärende Makros werden von der *Log::Writer* Klasse zur Protokollierung bereitgestellt:

```
_MSG(level, msg, ...) Log::Writer::instance().message(level, ...)

Fatal(msg, ...)     _MSG(Log::LOG_FATAL_ERROR, msg, ...)
Error(msg, ...)     _MSG(Log::LOG_ERROR, msg, ...)
Warning(msg, ...)   _MSG(Log::LOG_WARN, msg, ...)
Success(msg, ...)   _MSG(Log::LOG_OK, msg, ...)
Info(msg, ...)      _MSG(Log::LOG_INFO, msg, ...)
Debug(msg, ...)     _MSG(Log::LOG_DEBUG, msg, ...)
```

Als weitere Funktionalität soll beim Instanzieren geprüft werden ob die Größe der Logdatei 2MB überschreitet. In diesem Fall soll die Logdatei neu angelegt werden. Um die ,,Gesprächigkeit" (Verbosity) der Logklasse zu kontrollieren, sollen zudem folgende Makros geboten werden:

```
LogSetVerbosity(LV) Log::Writer::instance().set_verbosity((Log::LOGLEVEL)LV)
LogGetVerbosity     Log::Writer::instance().get_verbosity()
```

Die höchste Gesprächigkeit hat dabei *Log::FATAL_ERROR*, die niedrigste *Log::LOG_DEBUG*. Wird also beispielsweise LogSetVerbosity(*Log::LOG_WARN*) aufgerufen, so werden nur noch fatale Fehler, Fehler und Warnungen ausgegeben. Die Reihenfolge entspricht der Enumeration LOGLEVEL in 7.4. Soll die Ausgabe vollkommen abgeschalten werden so kann eine Zahl kleiner 1 an LogSetVerbosity() übergeben werden.

7.6. Config Hauptklassen

7.6.1. Path

Die *Init::Path* Klasse soll für die Initialisierung und das Management der Freya Config Pfade zuständig sein. Bei der Initialisierung soll überprüft werden, ob das Konfigurationsverzeichnis vorhanden ist. Wenn nicht wird ein Neues angelegt und anschließend eine default config.xml

geschrieben. Eine Standardkonfiguration ist im Quellcode als konstanter globaler String einkompiliert.

Schlägt das Erstellen der Konfigurationsdatei fehl, so soll versucht werden eine entsprechende Fehlermeldung in die Log Datei zu schreiben, falls diese zuvor erfolgreich angelegt wurde. Zusätzlich sollen DEBUG Ausgaben auf dem Bildschirm angezeigt werden, wenn das Programm über ein Terminal gestartet wird.

Instanzierung der Path Klasse kurz erläutert: Bei der Instanzierung werden die private Methoden get_config_dir() und get_config_path() aufgerufen, deren Rückgabewerte werden als Membervariablen der Init::Path gespeichert. Diese Methoden nutzen die g_get_user_config_dir() glib Methode welche den User Pfad nach XDG Standard zurückliefert.[5]

Je nach Funktion, wird an den Pfad ein ,,/freya" für das Freya Konfigurationsverzeichnis, ,,config.xml" für die Konfigurationsdatei oder ,,log.txt" für die Logdatei, gehängt.

Bei der Initialisierung wird die dir_is_available() Methode aufgerufen. Diese prüft ob die nötigen Verzeichnisse und Dateien existieren, wenn nicht wird versucht diese anzulegen. Diese Klasse schreibt DEBUG und ERROR Ausgaben auf die Konsole raus, da zum Zeitpunkt der Initialisierung nicht gewährleistet werden kann dass eine Logdatei existiert.

Die ,,dir is available()" Methode kurz erläutert: Diese Methode prüft zuerst über die glib Funktione g_file_test() ob ein ,,freya" Verzeichnis existiert, ist dies nicht der Fall, werden die privaten Methoden create_dir() und create_config() aufgerufen. Ist ein ,,freya" Verzeichnis vorhanden, so wird über die glib g_file_test() Funktion geprüft, ob die Konfigurationsdatei config.xml existiert. Existiert diese nicht, so wird die private create_config Methode() aufgerufen. Ansonsten wird mittels der glib g_access() Methode geschaut, ob diese lesbar und beschreibar ist. Bei Misserfolg wird über die glib Funktion g_warning eine entsprechende Fehlermeldung auf dem Bildschirm (Konsole) ausgegeben, da die Logklasse zu diesem Zeitpunkt noch nicht aufgebaut worden ist.

Die ,,create config()" Methode kurz erläutert: Die Mothode soll zum Anlegen eines Konfigurationsverzeichnisses verwendet werden. Im Fehlerfall wird eine entsprechende Warnung über glib Funktion g_warning() auf dem Bildschirm ausgegeben.

[5] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

Die ,,create dir()" Methode erläutert: Diese Methode bietet die Funktionalität über die glib Funktion g_mkdir_with_parents(configdir,0755) ein Verzeichnis mit den rechten 755 (drwxr-r-xr-x) anzulegen. Bei Erfolg wird mit g_message() eine Erfolgsmeldung auf dem Bildschirm (Konsole) ausgegeben, andernfalls eine Warnung mit g_warning().

7.6.2. Konfigurationsdatei

Die Freya Konfigurationsdatei soll im simplen XML Format realisiert werden. XML wird gewählt um das Parsen zu vereinfachen und um ein standardisiertes Format nach außen bereitzustellen. Die Konfigurations- und Logdatei soll nach XDG Standard (`$XDG_CONFIG_HOME`) unter `$HOME/.config/freya/<config.xml,log.txt>` gespeichert werden.

- `http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html#variables`

Die Optionen in der Konfigurationsdatei sind baumartig nach ,,Domainprinzip" aufgebaut. Die Konfigurationsdatei unter 7.5 zeigt exemplarisch den gewünschten Aufbau.

```
<?xml version=\1.0\ encoding=\utf-8\?>
<freya>
    <settings>
        <connection>
            <port>6600</port>
            <musicroot>~/chris/Musik</musicroot>
            <host>localhost</host>
            <!-- Connect on startup? -->
            <autoconnect>1</autoconnect>
            <!-- In seconds -->
            <timeout>20</timeout>
             <!-- Autoreconnect interval in seconds -->
            <reconnectinterval>2</reconnectinterval>
        </connection>
        <libnotify>
           <!-- Show notifications? -->
            <signal>0</signal>
           <!-- How long? -->
            <timeout>-1</timeout>
        </libnotify>
        <trayicon>
            <!-- Show trayicon? -->
            <tray>0</tray>
            <!-- To tray when closing? -->
            <totrayonclose>0</totrayonclose>
        </trayicon>
        <playback>
            <!-- Stop music when closing Freya? -->
            <stoponexit>0</stoponexit>
        </playback>
    </settings>
    <plugins>
    </plugins>
</freya>
```

Abbildung 7.5.: Config.xml Konfigurationsdatei im Überblick

7.6.3. Model

Die *Config::Model* Klasse gehört nach dem MVC Paradigma zur Model Schicht. Diese Klasse soll die nötigen Daten (Konfigurationsdatei), die zum Betrieb von Freya nötig sind, im Speicher vorhalten und Methoden zum Lesen und Speichern der Konfigurationsdatei auf die Festplatte bereit stellen (siehe 7.6).

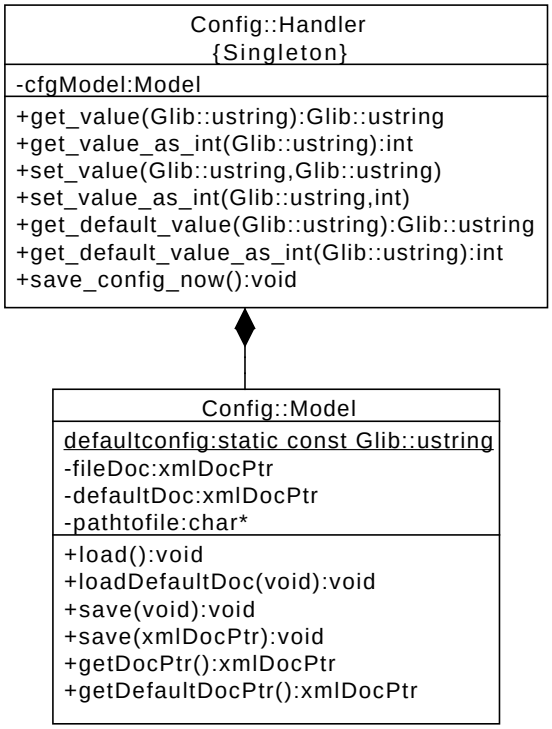


Abbildung 7.6.: Config-Model Beziehung

Zum Parsen der XML Datei soll hier die C Programmbibliothek libxml2 verwendet werden. Diese Library wurde gewählt, weil sie alle benötigten Funktionen enthält, nach dem ANSI-C Standard implementiert ist und bereits seit über einem Jahrzehnt Quasi-Standard im C Umfeld ist.

Informationen zur zu verwendenden Libaray:

- `http://xmlsoft.org/`
- `http://en.wikipedia.org/wiki/Libxml2`

7.6.3.1. Instanzierung des Models

Über die *Init::Path* Klasse holt sich das Model bei seiner Instanziierung über die path_to_config() Methode den Pfad zur Konfigurationsdatei, parst diese sowie die einkompilierte default Config und initialisiert zwei XML Document Pointer die, auf ein DOM Objekt, welches einen Dokumentenbaum enthält, zeigen. Hierzu werden die privaten load(pathtofile) und die loadDefaulDoc() Methoden der Config::Model Klasse verwendet.

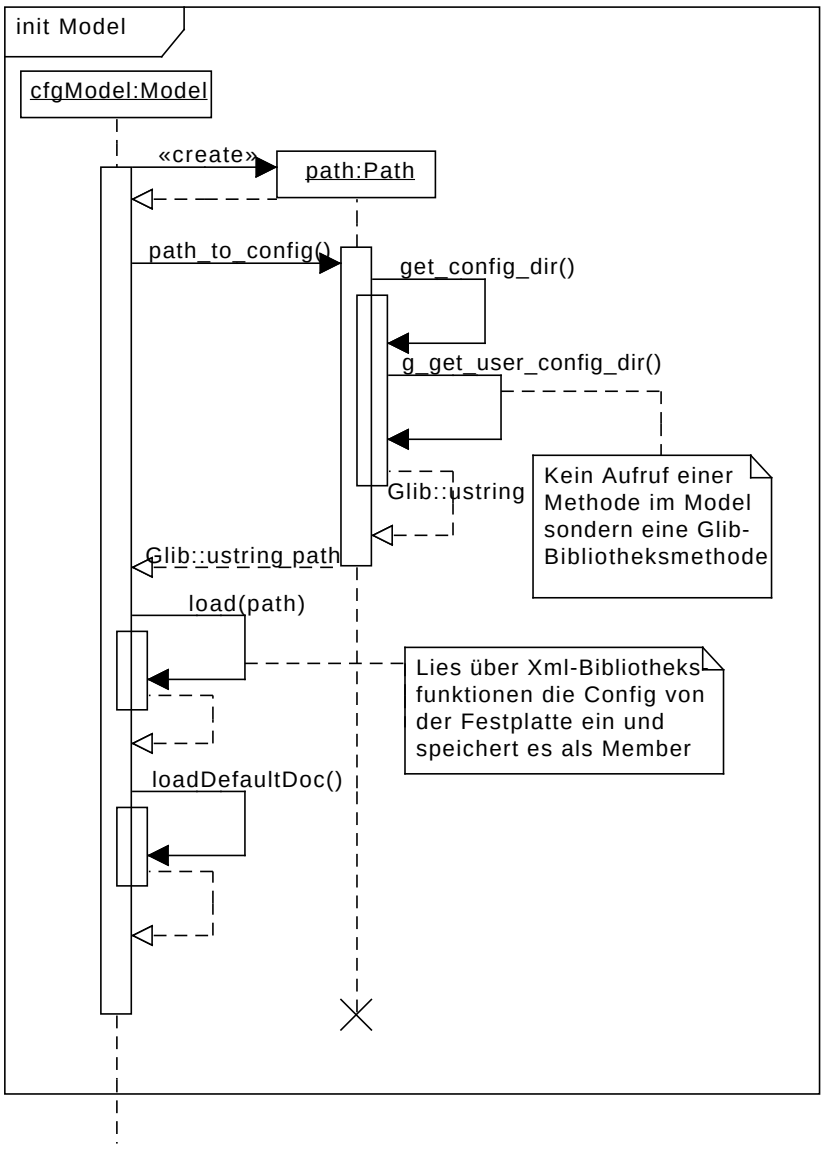


Abbildung 7.7.: Initialisierung des Models

Anschließend kann man über diese DOM Objekte traversieren und Werte der Konfigurationsdatei lesen oder setzen. Die default config wurde implementiert um fehlerhaften Werten oder einer kaputten Konfiguration vorzubeugen. Ist ein benötigter Wert nicht in der User config vorhanden oder ist diese beschädigt so wird auf die default config zurückgegriffen. Bei Beendigung des Models wird das aktuelle Objekt als XML Konfigurationsdatei auf die Festplatte geschrieben. Wie andere Objekte auch, nutzt das Model die Log-Klasse um zu Informationen und Fehler zu protokollieren.

7.6.3.2. Prinzipieller Ablauf der load() Methode

Beim Instanzieren ruft das Model seine load() Methode mit dem aktuellem Pfad auf. In dieser wird als Erstes die libxml2 Methode xmlParseFile(pathtofile) aufgerufen. Diese bekommt den Pfad zur Konfigurationsdatei übergeben und versucht über den übergebenen Pfad das File zu laden.
Wenn die Operation erfolgareich war, wird ein Dokument Pointer von der xmlParseFile() zurückgegben, im Fehlerfall wird *NULL* zurückgegeben.
Anschließend wird das geladene Dokument auf Gültigkeit geprüft. Zeigt dieses auf *NULL*, so wird eine entsprechende Fehlermeldung über den Log::Writer in die Logdatei geschrieben. Wurde ein gültiger xmlDocPtr zurückgegeben so soll folgendes passieren:

- Ein Node Pointer wird auf das root Element über xmlDocGetRootElement() gesetzt
- Überprüfung, ob Node Pointer *NULL* ist, trifft das zu, so wird ein Error in die Logdatei geschrieben, allokierter Speicher vom Dokument Pointer mittels xmlFreeDoc() freigegeben und dieser auf *NULL* gesetzet.
- Ist der Node Pointer gültig, so wird mittels der libxml2 Methode xmlStrcmp(DocPtr, ,,freya") geprüft, ob das root Element ,,freya" entspricht. Ist dies der Fall, wird eine Erfolgsmeldung in die Logdatei geschrieben. Im Fehlerfall wird eine Fehlermeldung rausgeschrieben, allokierter Speicher freigegeben und der Dokument sowie der Node Pointer auf *NULL* gesetzt.

7.6.3.3. loadDefaultDoc() Methode

Diese Methode holt sich die default Konfigurationsdatei aus einem einkompilierten String. Dieser String wird anschließend mittels der libxml2 xmlParseMemory() geparst und ein xmlDocPtr zurückgegeben, welcher als defaultDoc Membervariable gespeichert wird.

7.6.3.4. Ablauf der save() Methode

Die save() Methode ist eine Wrapper Methode für save(char*, xmlDocPtr). Sie ruft lediglich diese mit dem aktuellen Dokument Pointer und dem Pfad zur config.xml auf.

Quellen zur Implementierung:

- `http://xmlsoft.org/tutorial/index.html`
- `http://student.santarosa.edu/~dturover/?node=libxml2`

7.6.4. Handler

Die Config::Handler Klasse gehört nach dem MVC Paradigma zur Controller Schicht. Diese Klasse soll für das Management bzw. für den Zugriff auf das Model und somit die Konfigurationsdatei zuständig sein. Sie enthält Methoden zum Lesen und Setzen der einzelnen Optionen.

Der Config::Handler wird als Singleton implementiert um einen zentralen Zugriff für alle anderen Programmteile über eine einzelne Schnittstelle zu ermöglichen.

Der Handler soll einen Pointer als Membervariable auf das aktuelle Model Objekt bekommen, um direkten Zugriff auf die Dokument Pointer zu haben. Desweiteren sollen Wrapper für die get und set value Methoden geschrieben werden, um verschiedene Datentypen lesen und setzen zu können, so kann gleich eine ,,Teilvalidierung" erfolgen.
Der Config::Handler stellt folgende Makros bereit:

```
CONFIG_SET(x,y)
CONFIG_GET(x)
CONFIG_SET_AS_INT(x,y)
CONFIG_GET_AS_INT(x)
CONFIG_SAVE_NOW()
CONFIG_GET_DEFAULT(x)
CONFIG_GET_DEFAULT_AS_INT(x)
```

Über die *save_now()* Abb. 7.8 Methode soll die aktuelle Konfiguration direkt über das Model gespeichert werden können. Alle Methoden nutzen nach Möglichkeit Log::Writer um Informationen und Fehler in der Logdatei zu protokollieren.

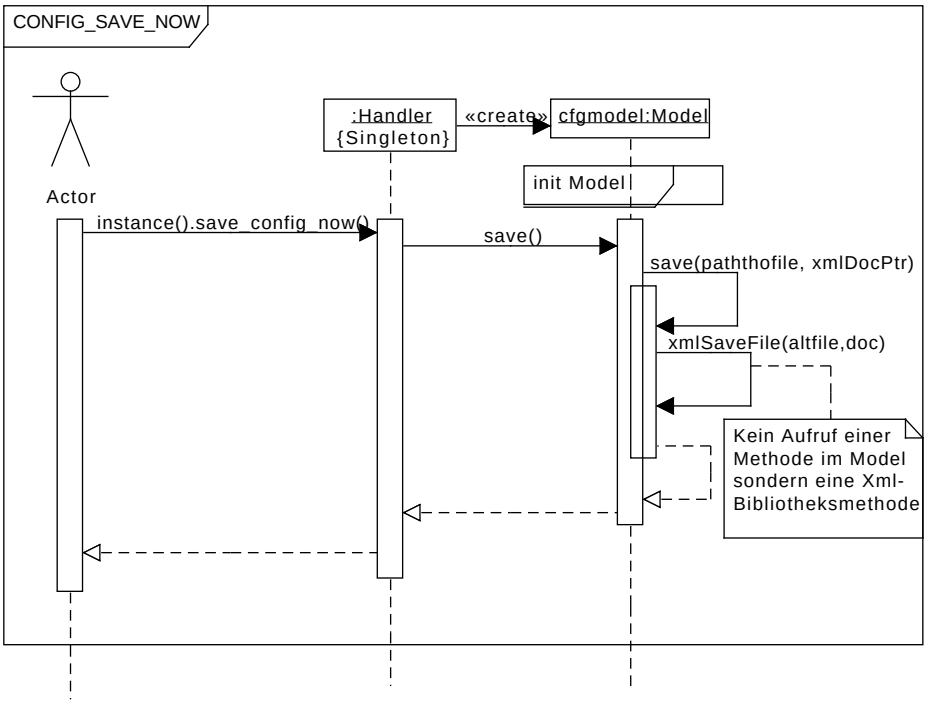


Abbildung 7.8.: Ablauf der Config Save Methode

Grober Ablauf beim setzen einer Integer Wertes:

- Aufruf des `CONFIG_SET_AS_INT("settings.connection.port",6667)` Makros
- Über das Makro wird die Wrapper Methode `set_value_as_int()` aufgerufen
- Diese Methode wandelt den Integer Wert in einen String um und ruft die eigentliche set Methode set_value(url, wert) auf
- Die set_value(url, wert) Methode holt sich mit cfgmodel.getDocPtr() einen aktuellen Dokument Pointer über das Model
- Ist der Dokument Pointer *NULL*, so kann kein Wert geschrieben werden, also wird über den Log:Writer eine entsprechende Warnung in die Logdatei geschreiben.
- Bei einem gültigem Dokument Pointer wird mittels xmlDocGetRootElement() auf das root Element zugegriffen
- Anschließend wird mit xmlChildrenNode der Node Pointer auf den folgenden Kinderknoden gesetzt und die traverse() Methode aufgerufen

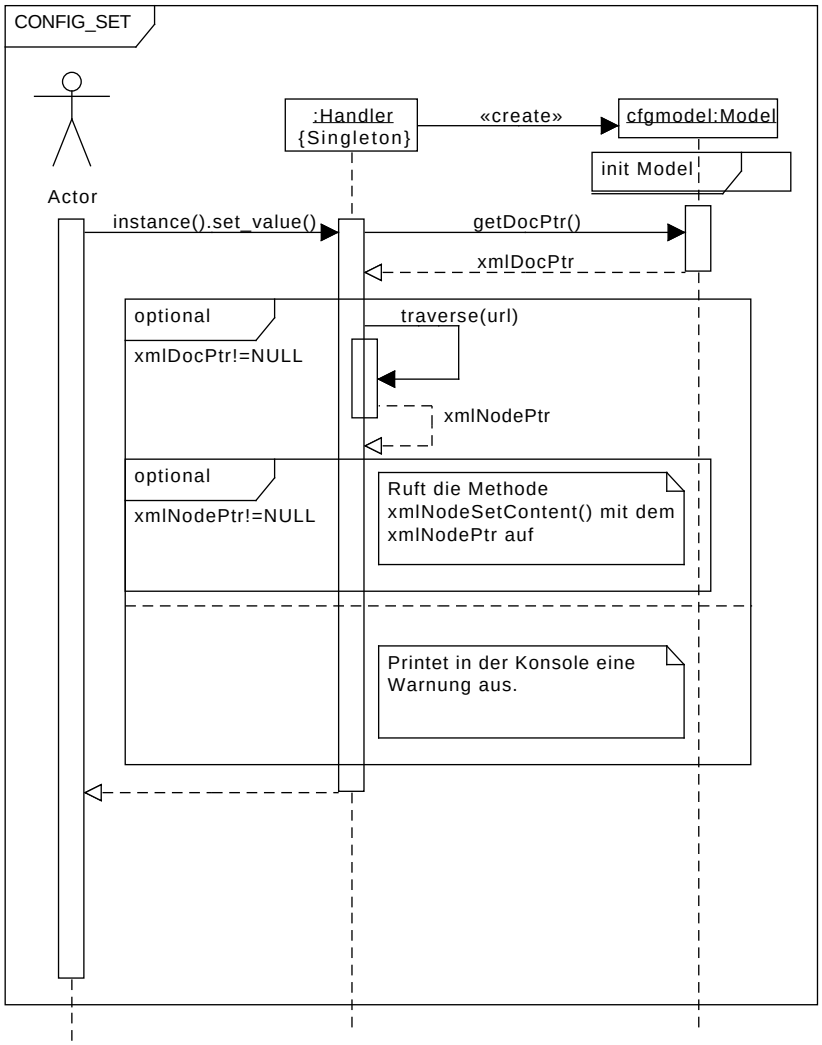


Abbildung 7.9.: Config set vlaue() Ablauf

- Nun werden diverse Vorbereitungen getätigt und anschließend rekursiv im Baum nach der übergebenen Url gesucht. Hierbei wird jeweils der rekursive Teilstring (eins.zwei.drei) gemäß dem Url Aufbau untersucht
- Wird die Url nicht gefunden oder sind andere Fehler aufgetreten wird ein *NULL Pointer* zurückgegeben und eine entsprechende Fehlermeldung in die Logdatei geschrieben
- Wird die Url gefunden, so wird ein NodePointer auf den Optionswert über die Aufruferkette an die set_value() Methode *returned*. In dieser wird dann der Wert an der entsprechende Stelle gesetzt.

Grober Ablauf beim Lesen eines Wertes:

- Das Makro `CONFIG_GET("settings.connection.host")` wird analog dem Setzen aufgerufen, dieses ruft die entsprechende Wrapper Methode get_value() welche die eigentliche private _get_value() Methode aufruft.
  Der zweite Parameter dieser Methode ist ein *boolean flag*, dieser dient dazu der _get_value() Methode mitzuteilen, ob der entsprechende Default Wert Pointer, welcher auf die einkompilierte Konfigurationsdatei zeigt, oder der Custom User Config Pointer geladen werden soll.
- Die _get_value() Methode holt entsprechend dem flag, den ,,richtigen" Dokument Pointer über das Model (analog Setzen eines Integer Wertes)
- Bei einen gültigen Pointer wird analog zum Setzen der Node Pointer auf das erste Element gesetzt und *traverse()* aufgerufen (siehe Setzen eines Wertes).
- Kann kein Node ermittelt werden (d.h. cur Pointer zeigte auf NULL nach dem traversieren), so wird eine entsprechende Warnung in die Logdatei geschrieben.
- Anschließend wird die _get_value(url,true) Methode rekursiv mit einem *true flag* aufgerufen. Aufgrund des true flags wird nun der Dokument Pointer mit den Default Werten geladen.
- Analog zum bisherigen Verlauf beim ,,Lesen eines Wertes" erfolgt die Suche des Default Wertes. Kann am Ende kein Default Wert ermittelt werden so wird an den Aufrufer eine leerer String zurückgegeben.

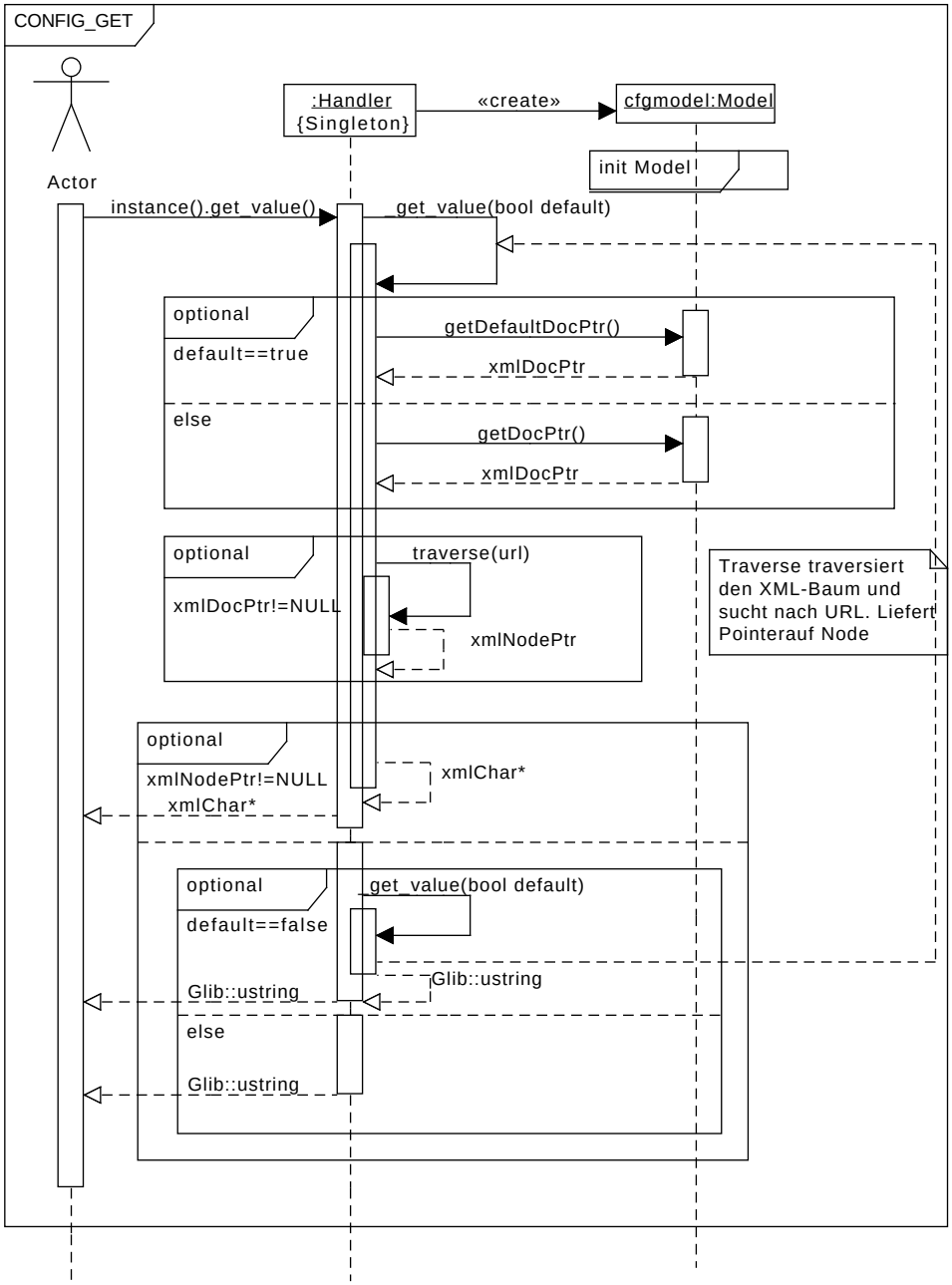


Abbildung 7.10.: Config get_value() Ablauf

7.7. Aufbau des Clients

Aus den oben genannten Anforderungen kann eine grobe Architektur abgeleitet werden. Zur Realisierung des Observerpatterns wird die sigc++ library benutzt, die mit Gtkmm geliefert wird. Es wird empfohlen, die Einführung bis einschließlich Kapitel 2 zu lesen:

- `http://developer.gnome.org/libsigc++-tutorial/stable/ch01.html`

Die ersten Einsätze werden auch noch tiefer erklärt.

7.7.1. Hauptklassen

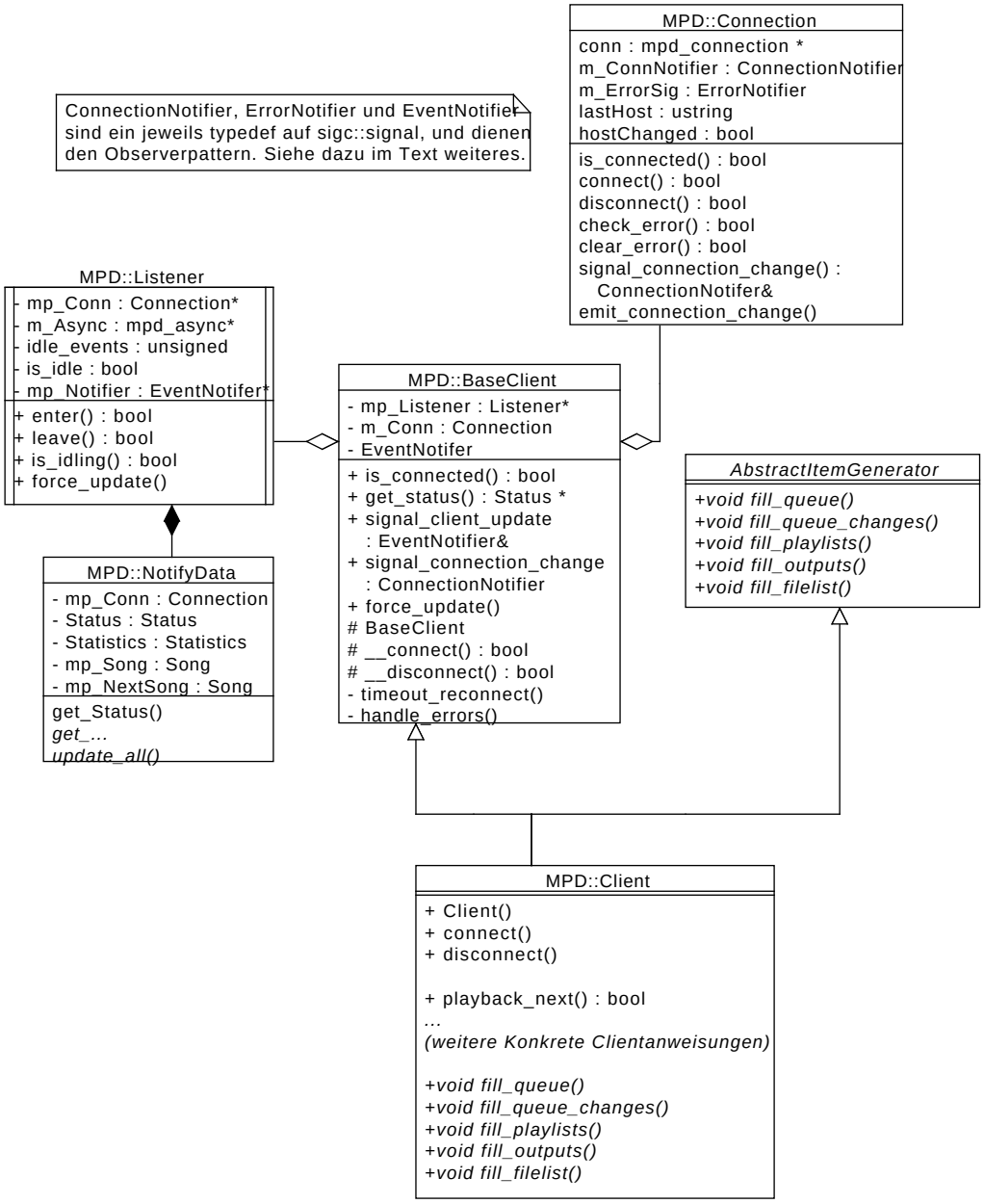


Abbildung 7.11.: Übersichtsklassendiagramm zur Clientstruktur

7.7.1.1. Listener

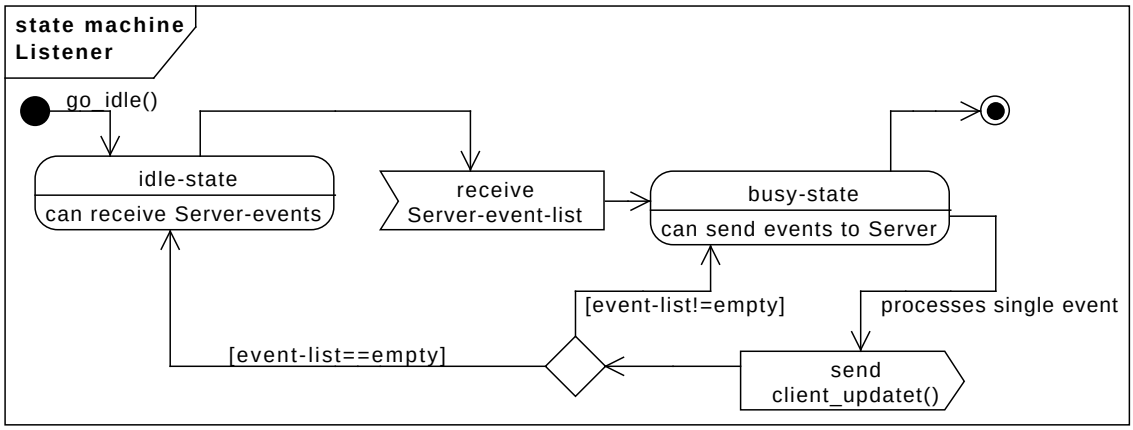


Abbildung 7.12.: Zustandsdiagramm Listener ,,idle/busy"

Der Listener verwaltet alles was mit dem Betreten und Verlassen des ,,Idlemodes" und den damit verbundenen Events zu tun hat. Er setzt den eingangs beschriebenen ,,Watchdog" auf die asynchrone Verbindung an, und ,,parst" die entsprechenden Antworten des Servers von Hand (siehe 7.12).

Es folgt eine Liste aller möglichen Events die von libmpdclient definiert werden. Die Aktionen die passieren müssen, damit diese eintreten, stehen in Klammern.

- MPD_IDLE_DATABASE *(Datenbank hat sich geändert)*
- MPD_IDLE_STORED_PLAYLIST *(Änderung an den Playlisten)*
- MPD_IDLE_QUEUE *(Änderung an der Queue: Clear, Add, Remove)*
- MPD_IDLE_PLAYER *(Pause,Stop,Play,Next,Stop,Previous,Seek)*
- MPD_IDLE_MIXER *(Änderung an der Lautstärke)*
- MPD_IDLE_OUTPUT *(An/Ausschalten eines Outputs)*
- MPD_IDLE_OPTIONS *(Random,Repeat,Single,Consume)*
- MPD_IDLE_UPDATE *(Datenbankupdate/rescan wurde gestartet)*

Bei der Instanzierung des Listeners soll das *sigc::signal*, welches das Clientupdate darstellt, übergeben werden. Zudem benötigt der Listener eine Referenz auf MPD::Connection, um an die darunter liegende asynchrone Verbindung zu kommen.

```
Listener(EventNotifier& Notifier, Connection& sync_conn);
```

Eine weitere Aufgabe des Listeners ist es, das Model MPD::NotifyData aktuell zu halten. Siehe dazu auch MPD::NotifyData. Bemerkt der Listener Events, so ruft er emit() auf dem übergebenen *sigc::signal* auf, und übergibt als Parameter das aufgetretene Event, sowie eine Instanz von MPD::NotifyData. [6]
Es folgt eine Liste von Funktionen, die der Listener mindestens haben soll.
enter() tritt in den ,,Idlemode" ein. Es ist ab diesem Punkt nicht mehr erlaubt Kommandos zu senden. leave() ist das genaue Gegenteil von enter() und verlässt den ,,Idlemode" sodass Kommandos gesendet werden können. is_idling() sollte selbsterklärend sein.

```
bool enter(void);
void leave(void);
bool is_idling(void);
```

Es soll zudem eine force_update() Funktion geben die ,,künstlich" alle Events auslöst. Dies ist nützlich bei der Initialisierung, bzw. bei ,,Reconnectvorgängen", wenn die GUI gezwungen werden soll sich zu updaten.

```
void force_update(void);
```

7.7.1.2. Connection

Diese Klasse stellt eine Art Wrapper um die *mpd_connection* (siehe `http://www.musicpd.org/doc/libmpdclient/connection_8h.html`) Struktur von libmpdclient da. Sie bietet die ,,eigentlichen" connect() und disconnect() Methoden die letzlich *mpd_connection_new()* bzw. *mpd_connectio_free()* aufrufen. Siehe 7.13 für den detaillierten Verbindungsablauf. Die benötigten Verbindungsdaten (Host, Port, Timeout in Sekunden) holt sich MPD::Connection aus der Config.
Sie hält zudem den letzten Host als Membervariable um feststellen zu können ob sich dieser zwischen zwei Verbindungsvorgängen geändert hat. Desweiteren bietet sie eine Schnittstelle um andere Klassen über Fehler in der Verbindung informieren zu lassen (signal_error()), bzw. um sie zu ,,reparieren" (clear_error()).
Es folgt eine Liste von Funktionen die mindestens vorhanden sein sollten. Ein boolean-Rückgabewert von true zeigt stets Erfolg an.
connect() soll die eigentliche Verbindung herstellen, disconnect() löscht die Verbindung wieder. get_connection() liefert einen Pointer auf die darunter liegende C-Struktur. Alle 3 Funktionen prüfen zudem intern bereits auf Fehler.

[6] Als spätere Optimierung könnte das aufgetretene Event übergeben werden, um selektiv Daten ,,upzudaten"; Sprich bei einer Lautstärkeänderung muss kein neues Statistikobjekt geholt werden.

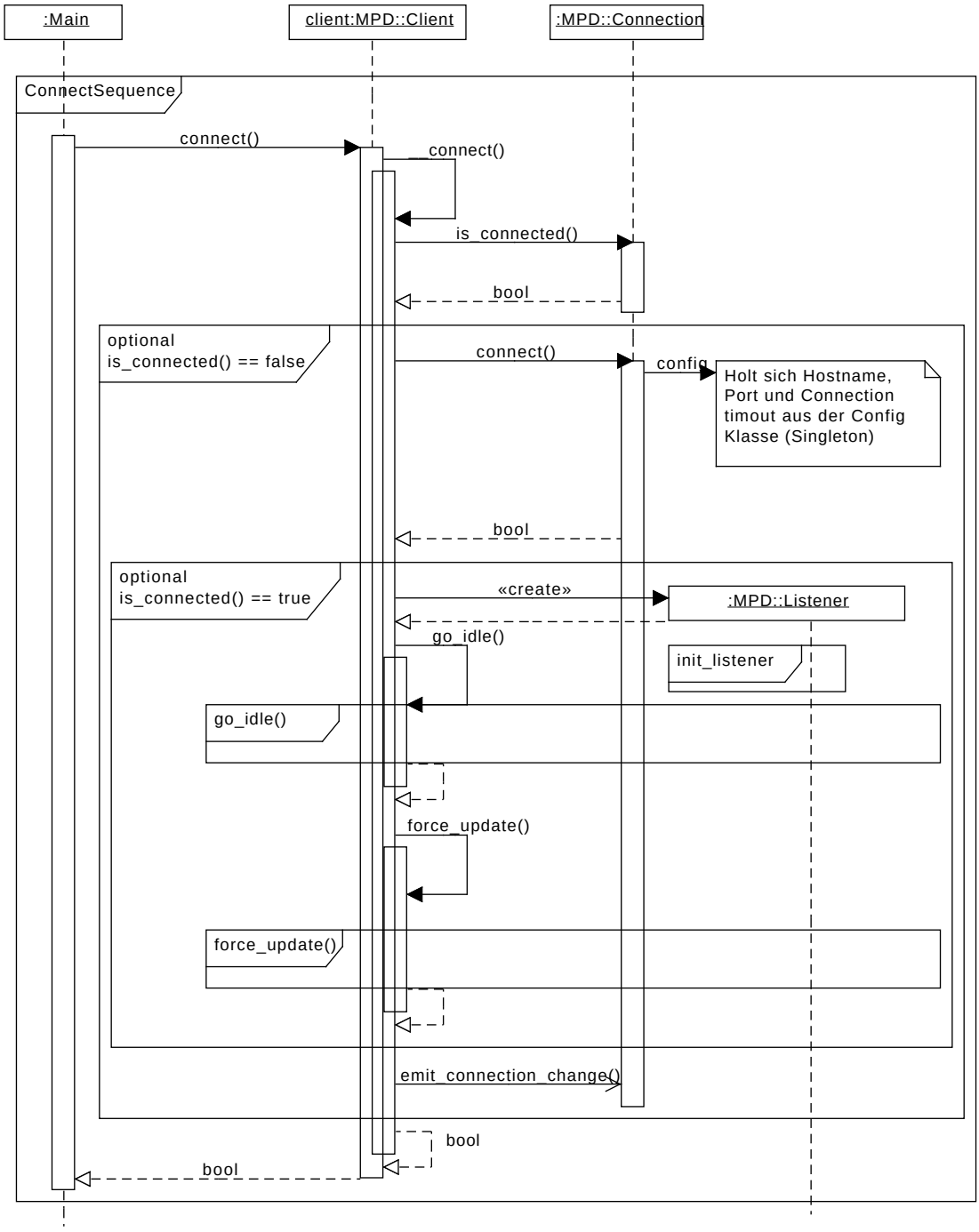


Abbildung 7.13.: Sequenzdiagramm zum Verbindungsaufbau

```
bool is_connected(void);
bool connect(void);
bool disconnect(void);
```

Zur Implementierung konkreter Kommandos wird die darunterliegende C-Struktur benötigt.
Siehe auch: AbstractClientExtension

```
mpd_connection * get_connection(void);
```

Die Connectionklasse soll zudem Schnittstellen bieten, um sich für Fehler- und Verbindungsänderungen zu registrieren. Auf den Rückgabewert der folgenden Funktionen kann sigc::signal::connect() aufgerufen werden, um einen Funktionspointer (bzw. ein ,,Funktor" um den libsigc++ Begriff zu gebrauchen) zu registrieren, der aufgerufen wird sobald ein Fehler eintritt, bzw. sich die Verbindung ändert. Die Schnittstellen sollen wie folgt aussehen:

```
typedef sigc::signal<void, bool,mpd_error> ErrorNotify;
typedef sigc::signal<void,bool,bool> ConnectionNotifier;

ErrorNotify& signal_error(void);
ConnectionNotifier& signal_connection_change(void)
```

Die Prototypen entsprechen stets den Templateargumenten in den typedefs:

```
void error_handler(bool is_fatal, mpd_error err_code);
void conn_change_handler(bool server_changed, bool is_connected);
```

Siehe MPD::BaseClient weiter unten für ein Beispiel wie eine Callbackfunktion registriert wird.
libmpdclient verbietet es weitere Kommandos an den Server zu senden wenn vorher ein Fehler passiert ist. Fehler müssen zuerst mit *mpd_connection_clear_error()* ,,bereinigt" werden, dies tut check_error(). Die Funktion wird normal nicht selbst aufgerufen, da sie von allen anderen Funktionen der Klasse implizit aufgerufen wird. Ist ein Fehler passiert so werden alle Klienten die sich zuvor mit signal_error() registriert haben benachrichtigt.

```
bool check_error(void);
```

7.7.1.3. BaseClient

Diese Klasse bildet die Basis zum eigentlichen Client. Sie kann nicht direkt instanziert werden, da der Konstruktor protected sein soll. Sie verwaltet administrative Tätigkeiten wie den eigentlichen Verbindungsaufbau an sich (siehe 7.13 und 7.14). Desweiteren bietet die Klasse einfache Methoden zum Eintreten (go_idle()) und Verlassen (go_busy()) des ,,Idlemodes" an. Geht die Verbindung verloren, ohne dass *__disconnect()* explizit aufgerufen wurde, so wird versucht sich periodisch zu reconnecten. Das Intervall in dem diese Versuche geschehen sollen, soll von ,,settings.connection.reconnectinterval" gelesen werden.

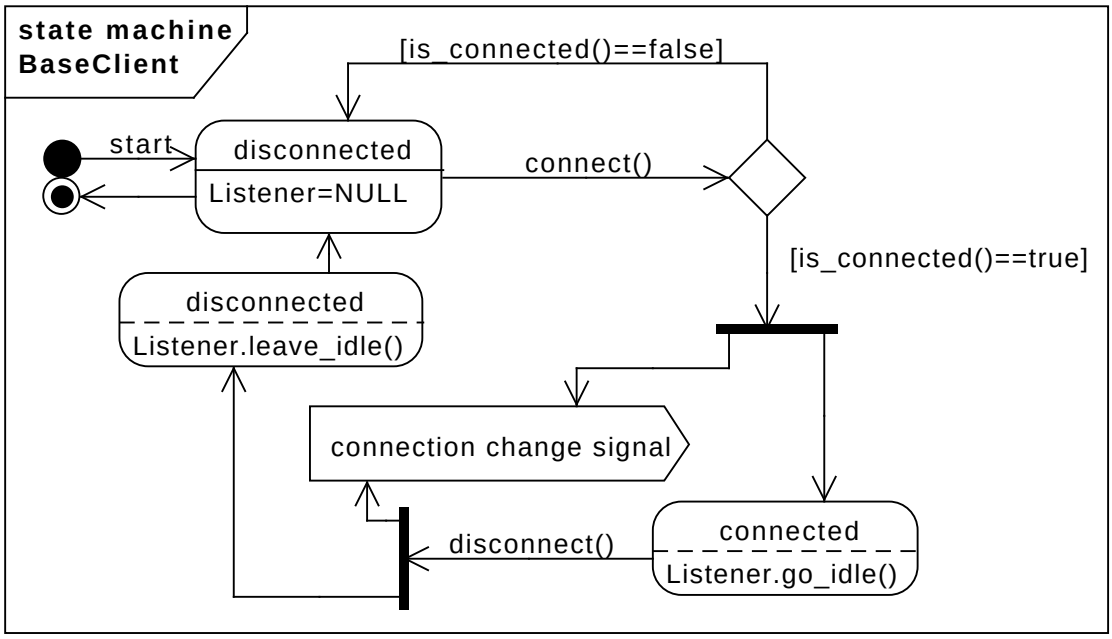


Abbildung 7.14.: Zustandsdiagramm BaseClient Connection

Siehe auch: AbstractClientExtension
MPD::BaseClient soll mindestens folgende public Methoden bieten:
Gibt eine Referenz auf das zugrunde liegende MPD::Connection Objekt zurück. Siehe Abstract-ClientExtension für eine detailliertere Erklärung.

```
Connection& get_connection(void);
```

Gibt *true* zurück wenn eine Verbindung besteht.

```
bool is_connected(void);
```

Die get_status() Funktion soll den letzten aktuellen MPD::Status zurückliefern, oder *NULL* falls nicht verbunden. Es soll garantiert sein, dass stets ein MPD::Status vorliegt, wenn is_connected() wahr ergibt.

```
Status * get_status(void);
```

Die folgenden Methoden bieten eine Möglichkeit sich für Clientevents (signal_client_update()), bzw. für Verbindungsänderungen (signal_connection_change()) zu registrieren.

```
typedef sigc::signal<void,mpd_idle,MPD::NotifyData&> EventNotifier;
EventNotifier& signal_client_update(void);

typedef sigc::signal<void,bool,bool> ConnectionNotifier;
ConnectionNotifier& signal_connection_change(void);
```

Registrieren kann man sich über die sigc::signal::connect() Methode:

```
// Callback Funktion wird bei jedem eingetretenen Event aufgerufen
void on_client_update(enum mpd_idle event, MPD::NotifyData& data)
{
   // Tue etwas bei einem 'player' event
   if(event == MPD_IDLE_PLAYER)
   {
       // Gib den Namen des aktuellen Songs aus
       cerr << data.get_song().get_path() << endl;
   }
}

// Registrieren der Callbackfunktion
// - Ableiten von AbstractClientUser macht dies automatisch
m_Client.signal_client_update().connect(sigc::ptr_fun(on_client_update));
```

Folgende 3 Funktionen funktionieren genau wie enter(), leave() und force_update() des Listeners, allerdings prüfen sie mit Connection::check_error() vorher stets auf Fehler.

```
void go_idle(void);
void go_busy(void);
void force_update(void);
```

7.7.1.4. Client

Der Client erbt von BaseClient und implementiert konkrete Kommandos wie ,,play", ,,random" etc. Er bietet zudem Schnittstellen zur Befüllung der Datenbank, der Queue und des Playlistmanagers, indem er die abstrakte Klasse *AbstractClientExtension* ausimplementiert. Er bietet die Methoden connect() und disconnect(), die letztendlich von Anwendern der Clientklasse zum Verbinden und Trennen genutzt werden. Ist in der config ,,settings.connection.autoconnect" gesetzt, so connected er sich automatisch.

connect() und disconnect() stellen die öffentliche Schnittstelle zum Verbinden dar. Sie rufen intern lediglich __connect() bzw. __disconnect() von MPD::BaseClient auf.

```
void connect(void);
void disconnect(void);
```

Der Client soll eine Reihe von Kommandos bereitstellen um das Playback zu kontrollieren. Die ersten 5 der folgenden Funktionen sollten relativ klar sein. Zu playback_pause() sei angemerkt, dass es bei Wiedergabe anhält und bei keiner Wiedergabe wie playback_play() funktioniert. playback_seek() springt in den Song mit der ID song_id an die Stelle abs_time in Sekunden. Die ID des momentan spielenden Songs kann durch get_status() gefunden werden. Alle Funktionen, mit Ausnahme von playback_crossfade), lösen ein ,,player" Event aus. playback_crossfade löst hingegen ein ,,options" Event aus.

```
void playback_next(void);
void playback_prev(void);
void playback_stop(void);
void playback_play(void);
void playback_crossfade(unsigned seconds);
void playback_pause(void);
void playback_seek(unsigned song_id, unsigned abs_time);
void playback_song_at_id(unsigned song_id);
```

Die folgenden Funktionen dienen dazu jeweils die *random, consume, repeat*und *single*-Modi umzuschalten. Alle lösen ein ,,options" Event aus.

```
void toggle_random(void);
void toggle_consume(void);
void toggle_repeat(void);
void toggle_single(void);
```

Desweiteren sollten Methoden zum Bearbeiten der Queue vorhanden sein.

- **queue_add():** Fügt rekursiv den Pfad in der Datenbank hinzu. queue_add("/") entspricht der ganzen Datenbank.
- **queue_clear():** Leert die gesamte Queue.
- **queue_delete():** Leert den Song an der Position 'pos'
- **queue_save_as_playlist():** Speichert die aktuelle Queue als Playlist mit dem Namen 'name'

```
void queue_add(const char * url);
void queue_clear(void);
void queue_delete(unsigned pos);
void queue_save_as_playlist(const char * name);
```

database_update() sendet MPD Server Hinweis um DB zu aktualisieren. database_rescan() sendet MPD Server Hinweis um DB neu einzulesen (teuer).

```
void database_update(const char * path);
void database_rescan(const char * path);
```

Setzen des ,,volumes" von 0 bis 100%. Die Abfrage des Volumes kann über get_status() erfolgen.

```
void set_volume(unsigned vol);
```

Folgende Funktionen sollen von *AbstractItemGenerator* voll implementiert werden. Siehe daher *AbstractItemGenerator* für eine genaue Erklärung.

```
void fill_queue(AbstractItemlist& data_model);
void fill_queue_changes(AbstractItemlist& data_model,
                        unsigned last_version,
                        unsigned& first_pos);
void fill_playlists(AbstractItemlist& data_model);
void fill_outputs(AbstractItemlist& data_model);
void fill_filelist(AbstractItemlist& data_model, const char * path);
```

7.7.1.5. NotifyData

Speichert den aktuellen Status, den aktuellen Song und die aktuelle Datenbankstatistik. Bietet zudem eine Funktion um entsprechende Daten bei Aufruf zu ,,updaten". Die Klasse gehört nach dem MVC Pattern somit der Model-Ebene an. Der Listener instanziert NotifyData im Konstruktor und gibt an wann sich dieser ,,updaten" soll (über update_all()).

Die folgenden 2 Funktionen garantieren einen validen Rückgabewert, solange eine Verbindung besteht:

```
Status& get_status(void);
Statistics& get_statistics(void);
```

Anmerkung: Die folgenden 2 Funktionen sollen *NULL* zurückgeben können, falls beispielsweise nichts wiedergegeben wird, oder man im „*Singlemode*" ist. get_song() liefert den aktuell spielenden MPD::Song, oder *NULL*. get_next_song() liefert den als nächstes spielenden MPD::Song oder *NULL*.

```
Song * get_song(void);
Song * get_next_song(void);
```

Die update_all() sollte nur vom Listener aufgerufen werden. Sie aktualisiert den internen Zustand von NotifyData.

```
void update_all(unsigned event);
```

7.7.2. Weitere Klassen

Desweiteren gibt es einige weitere Klassen die am Rande eine Rolle spielen, und meist objektorientierte Wrapperklassen für die C-Strukturen von libmpdclient bereitstellen, oder im Falle von MPD::AudioOutput und MPD::Playlist eigene Clientkommandos implementieren.

7.7.2.1. Song

Die Song Klasse ist ein Wrapper für mpd_song Struktur und die dazugehörigen Funktion von libmpdclient. MPD::Song soll alle Funktionen von libmpdclient (siehe `http://www.musicpd.org/doc/libmpdclient/song_8h.html`) anbieten. Diese werden hier nur aufgelistet aber nicht erklärt da sie genau wie ihre Vorbilder funktionieren sollen:

```
const char * get_path(void);
const char * get_tag(enum mpd_tag_type type, unsigned idx);
unsigned get_duration(void);
time_t get_last_modified(void);
void set_pos(unsigned pos);
unsigned get_pos(void);
unsigned get_id(void);
```

MPD::Song soll zudem eine Funktion bieten um die Metadaten des Songs in einer „sprintf" ähnlichen Art als String zurückzuliefern:

```
Glib::ustring song_format(const char* format, bool markup=true);
```

Ein beispielhafter Aufruf:

```
SomeSong.song_format("Artist is by ${artist}")
```

Die Escapestrings die dabei unterstützt werden sollen in etwa der *mpd_tag_type* Enumeration vom libmpdclient entsprechen. [7] Die unterstützten Typen sind somit: *artist, title, album, track, name, data, album_artist, genre, composer, performer, comment, disc*. Ist ein Escapestring nicht bekannt, so wird er nicht escaped. Ist der ,,tag" nicht vorhanden, soll mit ,,unknown", bzw. im Fall "artist" mit dem Dateinamen escaped werden.
Diese Klasse gehört nach dem MVC Paradigma zur Modelschicht.

7.7.2.2. Directory

Die Directory Klasse ist ein Wrapper für mpd_directory C-Strukutr. Diese wird als Anzeige für ein Verzeichniss benutzt, jedoch nicht als Container für andere Elemente.

Da MPD::Directory die abstrakte Klasse AbstractComposite erweitert muss als einzige öffentliche Funktion get_path() implementiert werden:

```
void get_path(void);
```

Diese Klasse gehört nach dem MVC Paradigma zur Modelschicht.

[7] http://www.musicpd.org/doc/libmpdclient/tag_8h.html#a3e0e0c332f17c6570ffdf788a685adbf

7.7.2.3. Statistics

Die Statistics Klasse ist ein Wrapper für mpd_stats und implementiert somit gemäß `http://www.musicpd.org/doc/libmpdclient/stats_8h.html` folgende Funktionen:

```
unsigned get_number_of_artists(void);
unsigned get_number_of_albums(void);
unsigned get_number_of_songs(void);
unsigned long get_uptime(void);
unsigned long get_db_update_time(void);
unsigned long get_play_time(void);
unsigned long get_db_play_time(void);
```

Diese Klasse gehört nach dem MVC Paradigma zur Model-Schicht.

7.7.2.4. Playlist

Die Playlist Klasse ist Wrapper für die mpd_playlist Struktur und implementiert von `http://www.musicpd.org/doc/libmpdclient/playlist_8h.html` folgende Funktionen:

```
const char * get_path(void);
time_t get_last_modified(void);
```

Die Klasse bietet desweiteren Funktionen zum:

- Entfernen der Playlist vom Server (Das Playlistobjekt ist danach invalid):

  ```
  void remove(void);
  ```

- Laden der Playlist in die Queue:

  ```
  void load(void);
  ```

- Umbennen der Playlist:

  ```
  void rename(const char * new\_name);
  ```

- Hinzufügen von Songs zur Playlist:

  ```
  void add_song(const char * uri);
  void add_song(MPD::Song& song);
  ```

Die genannten Funktionen müssen den idlemode verlassen können, daher leitet MPD::Playlist von AbstractClientExtension ab.

7.7.2.5. AudioOutput

Die AudioOutput Klasse ist ein Wrapper für mpd_output und implementiert von `http://www.musicpd.org/doc/libmpdclient/output_8h.html` folgende Funktionen:

```
unsigned get_id(void);
const char * get_name(void);
bool get_enabled(void);
```

Die Klasse bietet desweiteren Funktionen zum:

- ,,Enablen" des Ausgabegerätes:

  ```
  bool enable(void);
  ```

- ,,Disablen" des Ausgabegerätes:

  ```
  bool disable(void);
  ```

enable() und disable() müssen den ,,idlemode" verlassen können, daher leitet MPD::AudioOutput von *AbstractClientExtension* ab. Diese Klasse gehört nach dem MVC Paradigma zur Modelschicht.

7.7.3. Abstrakte Klassen

7.7.3.1. AbstractClientUser

- Verwaltet einen Pointer auf die MPD::Client Klasse, sodass der Anwender der Klasse dies nicht selbst tun muss. Im Konstruktor der Klasse muss eine Referenz auf den Client übergeben werden:

```
AbstractClientUser(MPD::Client& client)
```

- Leitet man von dieser Klasse ab so müssen folgenden Methoden implementiert werden:

```
void on_client_update(enum mpd_idle event, MPD::NotifyData& data);
```

on_client_update() aufgerufen sobald der Listener eine Änderung feststellt, siehe weiter unten „Interaktion des Clients mit anderen Modulen" für eine genauere Erklärung.

```
void on_connection_change(bool server_changed, bool is_connected);
```

on_connection_change() wird aufgerufen sobald sich der Client verbunden oder getrennt hat. Im ersten Fall ist is_connected true, im anderen false. Sollte sich der Client verbunden haben und der neue Server entspricht nicht mehr dem alten, so ist auch server_changed true. server_changed soll beim Start des Clients automatisch wahr sein.
Beide Signale werden automatisch durch den Konstruktor von AbstractClientUser registriert. Weiterhin können alle Klassen über den mp_Client Pointer auf den Client zugreifen.

7.7.3.2. AbstractItemlist

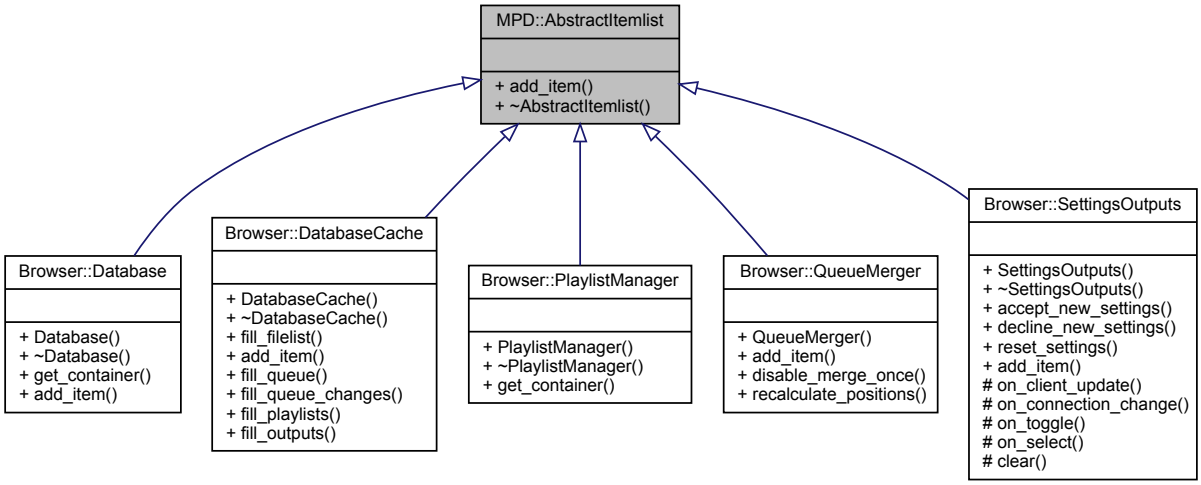


Abbildung 7.15.: Die AbstractItemList

Für bestimmte Client Funktionen muss eine Nutzerklasse von *AbstractItemlist* ableiten (siehe 7.15). Leitet man ab, so muss die Methode add_item(AbstractComposite * data) implementiert werden. Je nach Bedarf kann über `static_cast<Zieltyp*>(data)` der entsprechende Datentyp ,,rausgecasted" werden. Beim Aufruf von MPD::Client::fill_queue ruft der Client die add_item Methode für jeden Song den er vom Server bekommt auf. Die ableitende Klasse kann diese dann verarbeiten.

Dadurch werden alle Methoden von AbstractItemGenerator (bzw. die Klassen die davon ableiten) benutzbar.

7.7.3.3. AbstractItemGenerator

Lässt ableitende Klasse folgende Methoden implementieren: Jede dieser Methoden ruft MPD::Playlist add_item() von *AbstractItemlist* auf um ihre Resultate weiterzugeben. Sie erlaubt den Einsatz des *Proxy-Patterns*.[8] Andere Klassen können sich so als Client „ausgeben". Dies fand Anwendung bei der Klasse „DatabaseCache" weiter unten.
Holt alle Songs der aktuellen Queue.

```
void fill_queue(AbstractItemlist& data_model);
```

Holt alle geänderten Songs in der Queue seit der Version last_version. Die Position des ersten geänderten Songs wird in first_pos gespeichert.

```
void fill_queue_changes(AbstractItemlist& data_model,
                        unsigned last_version,
                        unsigned& first_pos);
```

Holt alle gespeicherten Playlisten vom Server.

```
void fill_playlists(AbstractItemlist& data_model);
```

Holt alle Audio Outputs vom Server.

```
void fill_outputs(AbstractItemlist& data_model);
```

Holt ein Listing aller Songs und Directories aus der Datenbank im Pfad 'path' (nicht rekursiv!)

```
void fill_filelist(AbstractItemlist& data_model, const char * path);
```

[8] http://en.wikipedia.org/wiki/Proxy_pattern

7.7.3.4. AbstractComposite

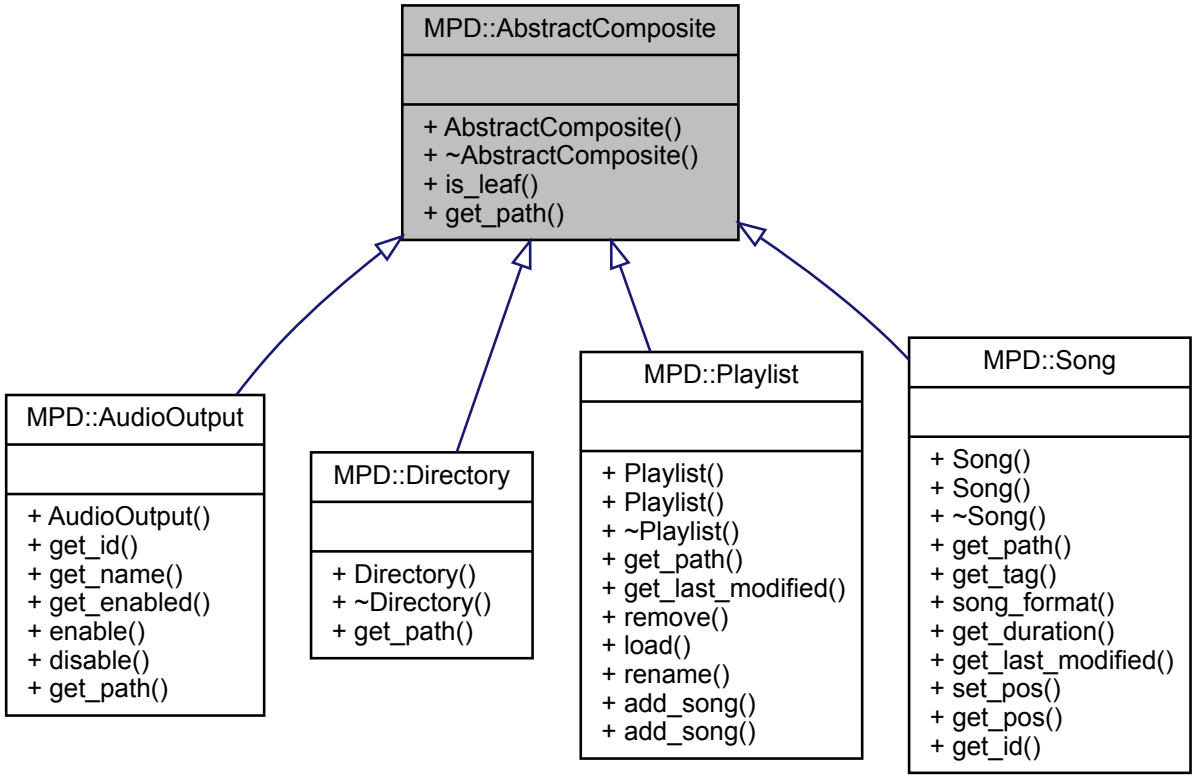


Abbildung 7.16.: Klassendiagramm zu AbstractComposite

Vereinheitlicht Zugriff auf Komponenten verschiedenen Typs. Die abstrakte Klasse (siehe Abb.: 7.16) zwingt seine Kinder dazu eine *get_path()* Funktion zu implementieren, die die Lage im virtuellen Filesystem des Servers angibt. Der Hauptanwender dieser Klasse ist der Databasebrowser, bzw. der dahinter gelagerte Cache, da AbstractComposite es erlaubt Songs und Verzeichnisse gleich zu behandeln (vgl. Composite Pattern).
Die erbende Klasse muss im Konstruktor angeben, ob es sich bei der Klasse um ein ,,File" (*true* für MPD::Song) oder um einen ,,Container" (*false* für MPD::Directory) handelt. Diese ,,is_leaf" Eigenschaft kann später mit der Funktion *is_leaf()* abgefragt werden.

7.7.3.5. AbstractClientExtension

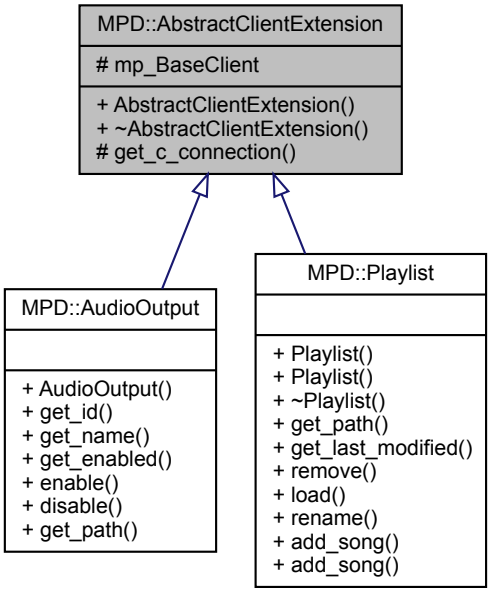


Abbildung 7.17.: Klassendiagramm zu AbstractClientExtension

Diese abstrakte Klasse erlaubt abgeleiteten Klassen ähnlich zum BaseClient eigene Kommandos zu implementieren (siehe 7.17). Dies geschieht indem die abstrakte Klasse eine Referenz auf den BaseClient im Konstruktor erwartet und speichert. Ableitende Klassen können die *protected* get_c_connection() Methode benutzen um eigene Kommandos zu implementieren.

```
AbstractClientExtension(MPD::BaseClient& base_client)
```

AbstractClientExtension wird in diesem Entwurf von MPD::Playlist und MPD::AudioOutput benutzt. Man kann allerdings darüber diskutieren, ob diese abstrakte Klasse der Modelschicht die Möglichkeit gibt, zu ausgefeilte Logik zu implementieren, was nach dem MVC Paradigma nicht sein sollte. Da die Logik meist darin besteht einfache Kommandos an den Server zu schicken, wurde dieser Weg gewählt um den Entwurf zu vereinfachen.

7.8. GUI Elementklassen

7.8.1. Interaktion des Clients mit anderen Modulen

- Die meisten GUI Klassen leiten von *AbstractClientUser* ab und speichern daher eine Referenz auf eine Instanz von *MPD::Client* Sie können daher Funktionen wie queue_add() direkt aufrufen. AbstractClientUser zwingt die abgeleitenden Klassen folgende Funktionen zu implementieren:

```
1) void on_client_update(mpd_idle event, MPD::NotifyData& data)
2) void on_connection_change(bool server_changed, bool is_connected)
```

  1. Wird aufgerufen sobald der Listener ein Event festgestellt hat. Für jedes eingetretene Event wird on_client_update() einmal aufgerufen. ,,event" ist dabei eine Enumeration aller möglichen Events, die von libmpdclient vorgegeben werden. (Siehe auch "www.musicpd.org") ,,data" ist eine Referenz auf eine Instanz von MPD::NotifyData. Die benutzenden Klassen können mit folgenden Funktionen bei Events sofort die aktuellen Daten auslesen:
     - get_status() gibt den aktuellen MPD::Status
     - get_song() gibt den aktuellen MPD::Song
     - get_statistics() gibt die aktuellen MPD::Statistics
  2. Wird vom Client aufgerufen sobald die Verbindung verloren geht. Dabei zeigt der übergebene boolean Wert ,,is_connected" an, ob man connected oder disconnected wurde. ,,server_changed" soll dann anzeigen, ob der Server derselbe ist, wie beim zuvor geschehenen Connectvorgang. Dies ist beim ersten Start stets wahr. ,,server_changed" kann nicht wahr sein, wenn ,,is_connected" falsch ist.

- Ableitung von den oben beschriebenen abstrakten Klassen AbstractItemlist und AbstractFilebrowser, um alle Funktionen von AbstractItemGenerator nutzen zu können. Beispiele dazu folgen weiter unten.

7.8.2. Hauptklassen

Der GManager Namespace enthält Klassen, die der Verwaltung und Kontrolle des Hauptfensters von Freya dienen, jedoch nicht den eigentlichen Inhalt des Hauptfensters bereitstellen (dies soll von den Browserklassen getan werden). Von der Aufteilung her wurden für alle zusammengehörigen Elemente jeweils eine Klasse angelegt. Alle Klassen gehören nach dem MVC Paradigma der Controllerschicht an.
Der Begriff ,,Browser" wird im folgenden für die einzelnen Tabs benutzt die Links in der Sidebar zu finden sind. Beispiele dafür sind ,,Queue", ,,Database" und ,,Settings".
Viele Klassen sind nur stichpunktartig erklärt, da sie oft einander ähnlich sind und ein gewisser Freiraum bei der Implementierung der GUI gelassen werden soll.

7.8.2.1. AbstractBrowser

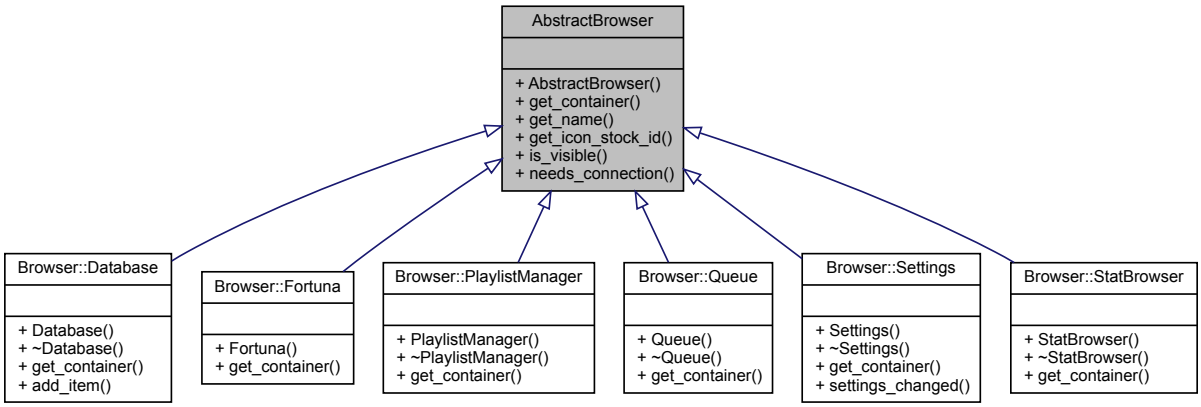


Abbildung 7.18.: Klassendiagramm zu AbstractBrowser

Eine abstrakte Basisklasse 7.18, durch die...

```
Gtk::Widget * get_container(void)
```

..implementiert werden muss. Diese sollte den umliegenden Container des Browser als Pointer zurückgeben, so dass *GManager::BrowserList* diesen (und damit seine Kinder) im Hauptbereich anzeigen kann. Siehe auch GManager::BrowserList für die nähere Erklärung zu den anderen nicht-abstrakten Methoden dieser Klasse.

7.8.2.2. BrowserList

Zeigt eine Liste von Browsern in der Sidebar.

- Bietet eine add() Methode, die eine Referenz auf AbstractBrowser erwartet und fügt diesen Browser in der Sidebar hinzu.

- set() zeigt den Browser im Hauptbereich, ohne ihn hinzuzufügen.

Die Klasse benutzt alle Methoden von AbstractBrowser, um diesen entsprechend anzuzeigen: Der Container, der im Hauptbereich beim Wechseln angezeigt wird

```
Gtk::Widget * get_container();
```

Welcher Name soll in der Sidebar angezeigt werden?

```
Glib::ustring get_name();
```

Welche Gtk::Stock::ID (eine ID die ein Icon repräsentiert) soll in der Liste angezeigt werden?

```
Gtk::Stock::ID get_icon_stock_id();
```

Ist sichtbar in der Leiste?

```
bool is_visible();
```

Benötigt dieser Browser eine Verbindung zum funktionieren?

```
bool needs_connection();
```

Als *View* wird ein Gtk::TreeView benutzt. Die Browserreferenzen werden in einem Gtk::ListStore gespeichert, womit das Model dargestellt wird.

7.8.2.3. Heartbeat

Diese Klasse soll alle 500ms ein Signal aussenden, die bisher vergangene Zeit summieren und sich regelmäßig mit dem Client synchronisieren, so dass die summierte Zeit der aktuellen Zeit innerhalb des aktuellen Songs entspricht. Über signal_client_update() können sich andere Klassen als Klienten registrieren:

```
Heartbeat.signal_client_update().connect(<funktionspointer>);
```

Der angegebene Funktionspointer wird dann aufgerufen und muss folgender Signatur entsprechen:

```
void func(double time)
{
    ...
}
```

Der übergebene Parameter ist die Zeit, die seit dem Instanzieren vergangen ist. Sie kann durch folgende Funktionen verändert werden:

```
void pause(void)  - Setzt das Zählen aus
void play(void)   - Fängt damit wieder an
void reset(void)  - Fängt von 0 wieder an
void get(void)    - Bekommt die jetzige Zeit
void set(void)    - Setzt die jetzige Zeit absolut und zählt von dort weiter
```

Zusätzlich stoppt die Heartbeat Klasse das Zählen wenn der Client das Playback pausiert. Wird es fortgesetzt, so wird play() aufgerufen. Zusätzlich wird bei jedem ,,Client Event" der Zähler an der vergangen Zeit im gerade spielenden Song justiert.
Neben dem Listener ist Heartbeat somit die einzige ,,aktive" Klasse.

7.8.2.4. MenuList

Kontrolliert und verwaltet die Anzeige (Sensitivität) und Steuerung der Menüleiste.

7.8.2.5. NotifyManager

Kontrolliert und verwaltet die Anzeige von Notifications bei entsprechenden ,,events". Greift dabei auf die Notifylib zurück.

7.8.2.6. PlaybackButtons

Kontrolliert die Anzeige der oberen rechten ,,Playbackbuttons" Stop, Play/Pause, Next, Previous Das Icon des Playbuttons wird entsprechend geändert, falls das Playback pausiert ist, bzw. fortgesetzt wird. Es sollen die Buttons *Stop, Play/Pause, Previous und Next* angezeigt wird.

7.8.2.7. Statusbar

Kontrolliert die Anzeige der Statusbar (was den Text mit einfasst). Benutzt GManager::Heartbeat um die Zeitanzeige zu aktualisieren. Ansonsten bekommt es alle Informationen rein vom Client update.

7.8.2.8. StatusIcons

Kontrolliert und verwaltet Anzeige der Icons unter der Sidebar. Bei Aktivierung sollen die Icons eingedrückt bleiben. Folgende Icons sollen dargestellt werden:

- Repeat-Mode (Wiederholt Queue)
- Consume-Mode (Entfernt Song nach Abspielen aus der Queue)

- Random-Mode (Zufälliges Abspielen innerhalb der Queue)
- Single-Mode (Hält nach Abspielen eines Songs an)

7.8.2.9. Timeslider

Zeigt und kontrolliert die aktuelle Zeit innerhalb des momentan spielenden Liedes. Beim Klicken innerhalb der Timeline wird zur entsprechenden Stelle im Song gesprungen. Benutzt GManager::Heartbeat um die Zeitanzeige zu aktualisieren.

7.8.2.10. TitleLabel

Verwaltet und kontrolliert Anzeige des Titels bzw. Künstlers und Albums in der Titelleiste, sowie der ,,Next Song" Anzeige in der Sidebar.

7.8.2.11. Trayicon

Verwaltet und kontrolliert Anzeige und Interaktion des Trayicons, das optional angezeigt werden kann. Dazu gehört auch die Definition und Anzeige des Popupmenüs, weshalb die Klasse von *Browser::BasePopup* ableitet. (Siehe dazu die Erklärung zu Browser::BasePopup weiter unten)

7.8.2.12. Volumebutton

Verwaltet und kontrolliert die Anzeige des Volumebuttons. Aus Performance Gründen sollen nur alle 0.05 Sekunden Volumeänderungen erlaubt werden.

7.8.2.13. Window

Verwaltet das Hauptfenster von Freya. Falls das Verstecken des Fensters beim Schließen gewünscht ist (*,,settings.trayicon.totrayonclose"* ist gesetzt), so wird *Gtk::Window::hide()* aufgerufen. Andernfalls wird einfach der Mainloop beendet wodurch die Kontrolle zur main() Methode zurückkehrt. Zudem wird eine get_window() Methode bereitgestellt die das darunterliegende Fenster (ein Gtk::Window) zurückgibt. Der Mainloop z.B. benötigt dies als Startargument.

7.9. Avahi Serverliste

7.9.0.14. Brower

Die Avahi Klassen *Avahi::Browser* und *Avahi::View* gehören nach dem MVC Paradigma zur Controller und View Schicht. Bei dieser Komponente wurde die Trennung zwischen Controller und View manuell durchgeführt.

Die Controller Klasse realisiert die technische Implementierung[9] des ,,Avahi Browsers", um MPD Server im Netzwerk zu finden.

Der View Part ist vom Controller völlig abgetrennt und dient lediglich zur Visualisierung der im Netzwerk gefundenen MPD Server mit der Möglichkeit diese direkt auszuwählen.

Nach dem Start des Bowsers über den ,,Show List" Button in den Freya Settings bekommt der Benutzer eine Liste mit sich im gleichen Netzwerk befindenden MPD Servern, aus welchen der Benutzer direkt einen auswählen kann. Wurden keine MPD Server gefunden oder läuft der Avahi Daemon nicht, so bekommt der User einen entsprechenden Hinweis.

Um den Avahi-Dienst überhaupt betreiben zu können, muss ein Avahi Daemon auf den jeweiligen Systemen installiert sein und auch laufen. Außerdem müssen in der MPD Konfiguration die beiden Einträge ,,zeroconf_enabled" und ,,zeroconf_name" eingepflegt sein damit sich der MPD Server am Avahi Daemon registriert. Avahi muss dabei vor dem MPD Server gestartet werden.

Informationen zu Avahi selbst:

- `http://avahi.org/`

Informationen zur Programmierschnittstelle von Avahi:

- `http://avahi.org/download/doxygen/`

Browser: Er sollte mindestens folgende Schnittstellen bieten:

```
Gtk::Window& get_window(void);
bool is_connected(void);

typedef sigc::signal<void,ustring,ustring,ustring,int> SelectNotify;
SelectNotify& signal_selection_done(void);
```

- get_window() gibt das Dialogfenster der View zurück. (Zur Anzeige nötig)
- is_connected() zeigt durch 'true' an ob eine Verbindung zum Avahidaemon besteht
- signal_selection_done() wird ausgelöst sobald der User in der View einen Server auswählt. Da es sich hier wieder um ein sigc::signal handelt kann der Anwender sigc::signal::connect() anwenden. Der Prototyp der dabei übergeben wurden muss sieht wie folgt aus:

[9] http://de.wikipedia.org/wiki/Zeroconf

```
void (ustring ip, ustring hostname, ustring name, int port)
{
  ...
}
```

View: Die View stellt lediglich die Daten dar und bietet daher auch nur Möglichkeiten um Server zur Liste hinzuzufügen und daraus zu löschen. Sie leitet von Gtk::Window ab und wird daher nur über die Methoden von Gtk::Window angesprochen.

```
void server_append(ip, hostname, name, port);
void server_delete(name);
```

7.10. Browserimplementierungen

Der Browser Namespace implementiert die einzelnen Browser die in der Sidebar angezeigt werden. Alle Klassen in diesem Namespace gehören nach dem MVC Paradigma der Controllerebene an - sie sind gewissermaßen der Kleber zwischen der Präsentation und der Logik. Wie die meisten anderen peripheren Klassen erben diese von *AbstractClientUser*, um Änderungen von diesem empfangen zu können. Dies wird im Folgenden nicht mehr explizit erwähnt.

7.10.1. Hauptklassen

7.10.1.1. BasePopup

Alle Klassen, die ein „Kontextmenü" anzeigen wollen, leiten von dieser Klasse ab. Die ableitende Klasse muss dem Konstruktor von BasePopup das Widget übergeben, welches das Kontextmenü anzeigen soll, und erwartet eine UI Definition des Menüs, dessen Struktur von Gtk+ vorgegeben wird. Eine beispielhafte UI Definition sieht folgendermaßen aus:

```
<ui>
  <popup name='QueuePopupMenu'>
    <menuitem action='q_remove'/>
    <menuitem action='q_clear'/>
    <separator />
    <menuitem action='q_add_as_pl'/>
  </popup>
</ui>
```

Die ableitende Klassen definieren dann das Aussehen des Menüs über die vererbte Funktion menu_add_item():

```
// Durch BasePopup definiert
void menu_add_item(Glib::RefPtr<Gtk::Action>& action,
                   Glib::ustring item_name,
                   Glib::ustring item_label,
                   Glib::ustring item_tooltip,
                   Gtk::StockID icon);

// Angewendet in einer abgeleiteten Klasse:
menu_add_item(m_ActionClear,"q_clear","Clear","Clear Queue",Gtk::Stock::CLEAR)
```

Zudem bietet die Klasse eine get_action() Methode, um die eigentliche Implementierung der Aktionen nicht in die abgeleitete Klasse machen zu müssen.

Im Code könnte das so aussehen:

```
/* mp_Popup ist die Instanz einer von BasePopup abgeleiteten Klasse */
mp_Popup->get_action("q_clear").connect(<funktionspointer>);

...

void Queue::clear_queue_action(void)
{
    ...
}
```

7.10.1.2. Database

Database Diese Klasse kontrolliert die Anzeige des Datenbankbrowsers. Sie leitet sich daher von *AbstractBrowser* ab, um sich bei der Browserliste registrieren zu können. Um die Methoden des *AbstractItemGenerator* Interface zu benutzen, leitet es zudem von *AbstractItemlist* ab und implementiert daher eine add_item() Methode. Diese fügt letzlich die gewonnenen Items seinem Model (einem Gtk::ListStore) hinzu. Siehe auch 7.19.

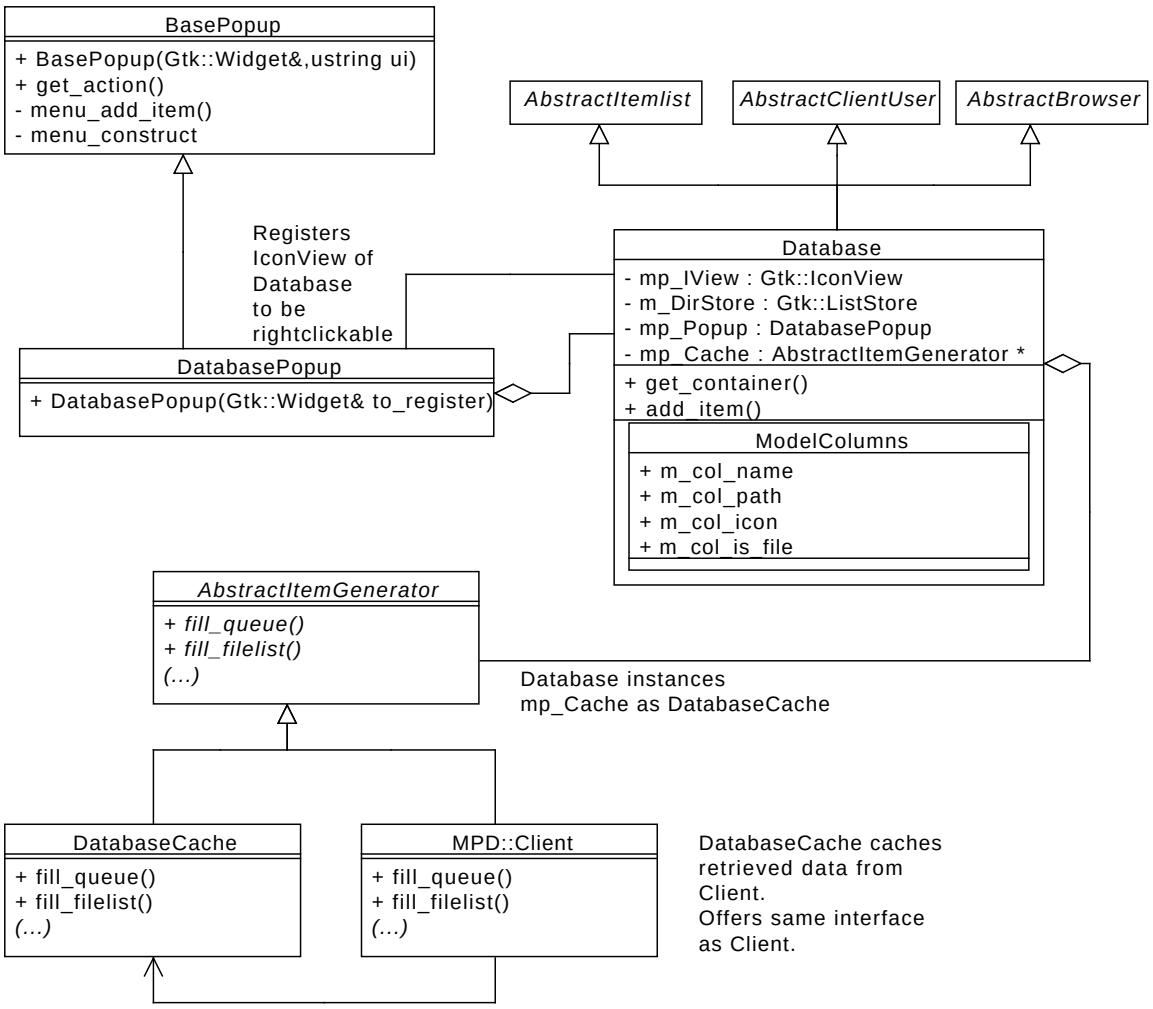


Abbildung 7.19.: Klassenübersicht zum Database Browser

DatabasePopup Eine Klasse die von BasePopup ableitet und das Popup definiert, das auftaucht wenn man im Databasebrowser ,,rechtsklickt". Sie bietet die folgenden Aktionen an, die man über die Methode get_action() abgefragt werden kann. Dadurch kann auf folgende Aktionen reagieren kann:

- ,,db_add" (Fügt Auswahl zum Ende der Queue hinzu)
- ,,db_add_all" (Fügt alles zum Ende der Queue hinzu)
- ,,db_replace" (Dasselbe wie db_add, leert aber Queue vorher)
- ,,db_update" (Sendet Server einen Updatehinweis)
- ,,db_rescan" (Sendet Server einen Rescanhinweis)

DatabaseCache Ein Zwischenspeicher für die im Databasebrowser angezeigten Ordner und Dateien. Sie setzt das Proxy-Pattern für MPD::Client um und erbt daher von der *AbstractItemGenerator* um sich als Client ausgeben zu können. Sie implementiert daher die fill_filelist() Methode vor, lässt aber die anderen Methoden ohne Implementierung. Da sie auch selbst Daten dem Cache hinzufügen muss leitet sich auch von *AbstractItemlist* ab und implementiert daher auch eine add_item() Methode.
Das zugrunde liegende Model ist dabei eine std::map (also eine Art Hashmap) die als Key den Pfad der zu ladenden Seite benutzt und als Wert ein Vector von *AbstractComposites* speichert. Wird eine Seite vom ,,Cache" über die fill_filelist() Methode verlangt, so wird nachgeschaut, ob im angegeben Pfad bereits eine Seite gespeichert ist, falls nicht wird sie vom Server geholt und gespeichert. Anschließend wird über die Elemente iteriert welche durch die add_item() Methode des Aufrufers weitergegeben werden. Sollte sich der Server wechseln bzw. sich die Datenbank aktualisieren, so wird der Cache geleert damit die Anzeige stets aktuell ist. Das MPD Protokoll bietet hier leider keine Möglichkeit herauszufinden, was genau sich geändert hat.
Anmerkung: Alle anderen Funktionen von AbstractItemGenerator werden nicht ausimplementiert.

7.10.1.3. PlaylistManager

PlaylistManager Diese Klasse kontrolliert die Anzeige des ,,Playlists" Browsers. Er verwaltet eine Liste der auf dem Server gespeicherten Playlisten. Es soll dabei eine Spalte mit dem Namen und eine Spalte mit dem letzten Änderungsdatum angezeigt werden. Die Namensspalte soll editierbar sein und nicht prüfen ob der neue Name bereits vorhanden ist. In diesem Falle soll der Editiervorgang nochmal gestartet werden. Zudem werden die Aktionen des Popupmenüs implementiert.

PlaylistManagerPopup Eine Klasse, die von BasePopup ableitet und das Popup definiert, das auftaucht, wenn man im PlaylistManager ,,rechtsklickt". Sie bietet die folgenden Aktionen an, die man über die Methode get_action() abfragen kann und dadurch auf diese Aktionen reagieren kann:

- pl_append (Fügt Inhalt der ausgewählten Playlists zum Ende der Queue hinzu)
- pl_replace (Dasselbe wie pl_append, aber leert vorher Queue)
- pl_delete (Löscht Playliste aus der Liste und vom Server)

7.10.1.4. Queue

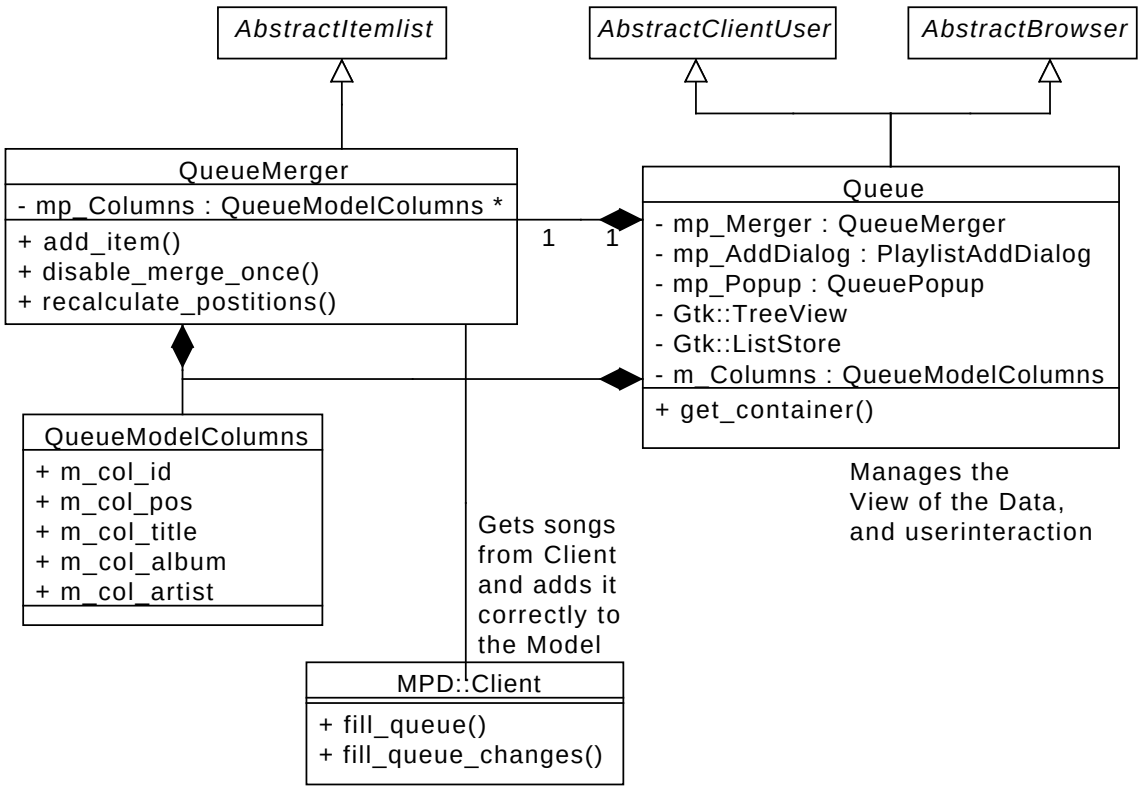


Abbildung 7.20.: Klassenübersicht zur Queue

Queue Diese Klasse kontrolliert die Anzeige der Queue (der aktuellen Playlist) und auch die Verwaltung des darunter liegenden Suchfelds. (Siehe auch 7.20)

Suchfunktion: Beim Tippen soll zum Künstler in der Liste gesprungen werden der mit den getippten Buchstaben beginnt. ,,Kno" sollte also zu ,,Knorkator" springen. Bei Aktivierung des Suchfelds muss die Auswahl entsprechend einer Volltextsuche gefiltert werden. Durch Aktivieren der Tastenkombination *Ctrl-F* soll zudem der Fokus auf das Suchfeld gelegt werden. Zudem werden folgende Aktionen des Popupmenüs implementiert:

- Remove - Entfernt ausgewählte Elemente aus der Queue und benachrichtigt Server.
- Clear - Leert alle Daten aus dem Model und benachrichtigt den Server entsprechend
- Save as Playlist - Speichert aktuellen Inhalt als Playliste; Namensabfrage durch PlaylistAddDialog

Das zugrundeliegende Model ist ein Gtk::ListStore, dessen Spaltenlayout durch QueueModelColumns festgelegt wird. Als View wird ein Gtk::TreeView verwendet.

QueueModelColumns Definiert die Spalten für die Queue, und erbt daher von Gtk::TreeModel::ColumnRecord, sodass ein Gtk::ListStore etwas damit anfangen kann. Die Definition ist nicht wie bei anderen Klassen als ,,Nested Class" realisiert, da sowohl Queue als auch QueueMerger darauf zugreifen müssen. Sie definiert die folgenden Spalten:

- m_col_id: Speichert die Songid eines Songs (nicht sichtbar)
- m_col_pos: Speichert die Position eines Songs (beginnend bei 0) (nicht sichtbar)
- m_col_title: Der Songtitel
- m_col_album: Der Albumtitel
- m_col_artist: Der Artisttitel

QueueMerger Diese Klasse verwaltet die eigentlichen Daten die die Queue anzeigt. Sie soll ihre Daten vom Client beziehen und erbt daher von *AbstractItemlist*, wodurch eine add_item() Methode implementiert werden muss. Da sie die Änderungen auch in die Queue einpflegen muss, erwartet die ,,Merger" Klasse eine Referenz auf das der Queue zugrunde liegende Gtk::ListStore Model, sowie deren Spaltendefinition, die als drittes Argument übergeben werden muss:

```
QueueMerger(MPD::Client& client,
            Glib::RefPtr<Gtk::ListStore>& queue_model,
            QueueModelColumns& queue_columns);
```

QueueMerger soll die folgenden *public* Funktionen bieten:
disable_merge_once() lässt das ,,Zusammenführen" einmal ausfallen. Dies ist nützlich bei der Implementierung der ,,Remove" Funktionalität, da man weiß wo ein Element gelöscht wurde, und es so aus Performancegründen explizit aus View und Model entfernen kann.

```
void disable_merge_once(void);
```

recalulate_positions() kann nützlich im Zusammenhang mit disable_merge_once() sein. Löscht man etwas explizit, so ist die Spalte mit den Positionsangaben korrumpiert. Diese Funktion berechnet die Positionen angefangen bei ,,pos" neu.

```
void recalculate_positions(unsigned pos = 0);
```

Bei einem Clientupdate sollen Änderungen über die Clientmethode fill_queue_changes() vom Server geholt und in die Queue eingepflegt werden.

QueuePopup Eine Klasse, die von BasePopup ableitet und das Popup definiert das auftaucht, wenn man in der Queue ,,rechtsklickt". Sie bietet die folgenden Aktionen an, die man über die Methode get_action() abfragen und dadurch auf diese Aktionen reagieren kann:

- q_remove (Entfernt ausgewählte Elemente aus der Queue)
- q_clear (Leert Queue völlig)
- q_add_as_pl (Zeigt den PlaylistAddDialog)

PlaylistAddDialog Zeigt einem Dialog zum Speichern der aktuellen Queue als Playlist mit einem bestimmten Namen. Der Name wird durch den Dialog abgefragt. Es wird keine Validierung durchgeführt, außer dass der Name länger als ein 0 Zeichen sein muss. Der eingegebene Name soll an den Aufrufer zurückgegeben werden.

7.10.1.5. Settings

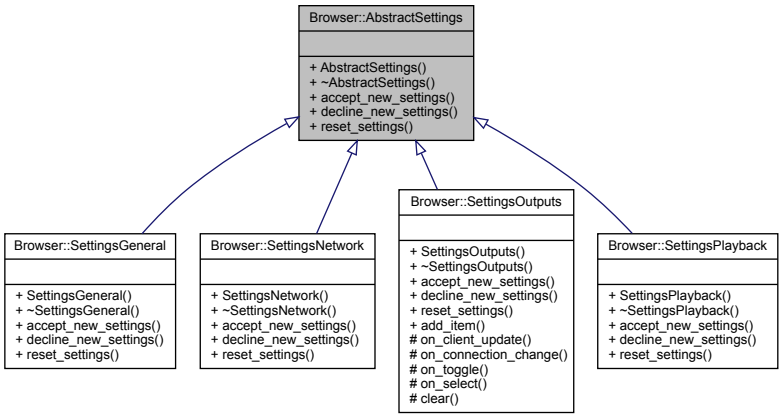


Abbildung 7.21.: Nutzer von AbstractSettings

AbstractSettings Eine abstrakte Klasse die einen Reiter im Settingsbrowser repräsentiert. Sie soll die folgenden ,,pure virtual" Methoden definieren:

```
virtual void accept_new_settings(void)
```

Weist Reiter an, alle Werte in die Config zu speichern

```
virtual void decline_new_settings(void)
```

Weist Reiter an, die letzten validen Werte aus der Config zu laden

```
virtual void reset_settings(void)
```

Weist Reiter an, die Defaultwerte aus der einkompilierten Config zu laden.

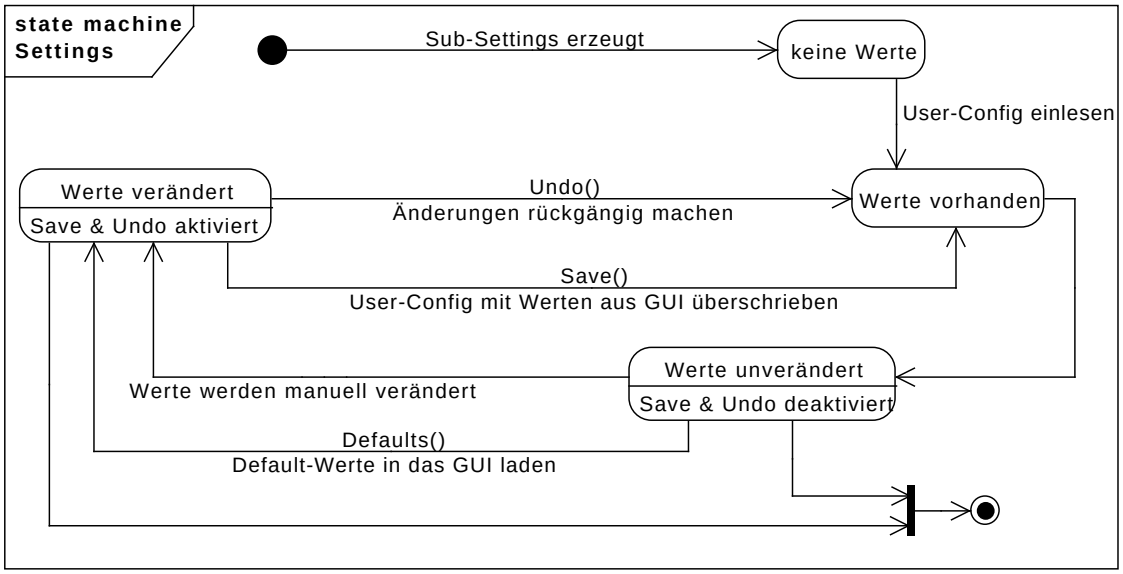


Abbildung 7.22.: Zustandsdiagramm Settings

Settings Die Settings Klasse repräsentiert den Settingsbrowser. Wie jeder andere Browser implementiert diese Klasse *AbstractBrowser* und eine get_container() Methode. Sobald etwas in der Präsentation geändert wird, soll es nicht gleich in die Config übernommen werden. Dies soll erst durch den Speicherbutton geschehen (siehe 7.23).

- Zurücksetzen - Setzt alle Einstellungen auf Fabrikstandards zurück
- Rückgängig - Setzt Änderungen auf letzten Stand zurück
- Speichern - Speichert aktuelle Änderungen

Die Klasse soll zudem eine Methode bieten, um anzuzeigen, dass die ,,Settings" geändert wurden z.B. durch Ausgrauen des Speicherbuttons:

```
void settings_changed(void)
```

Um in jedem Tab die Settings zurückzusetzen (auf letzten validen Wert oder Standardwert) speichert die Settingsklasse für jeden Reiter eine AbstractSettings Instanz in eine Liste. Sie kann so später darüber iterieren um die werte zu speichern, rückgängig zu machen oder zurück zu setzen.

SettingsGeneral Die konkrete Klasse, die den ,,General" Tab implementiert. Folgende Einstellungen sollen geändert werden können:

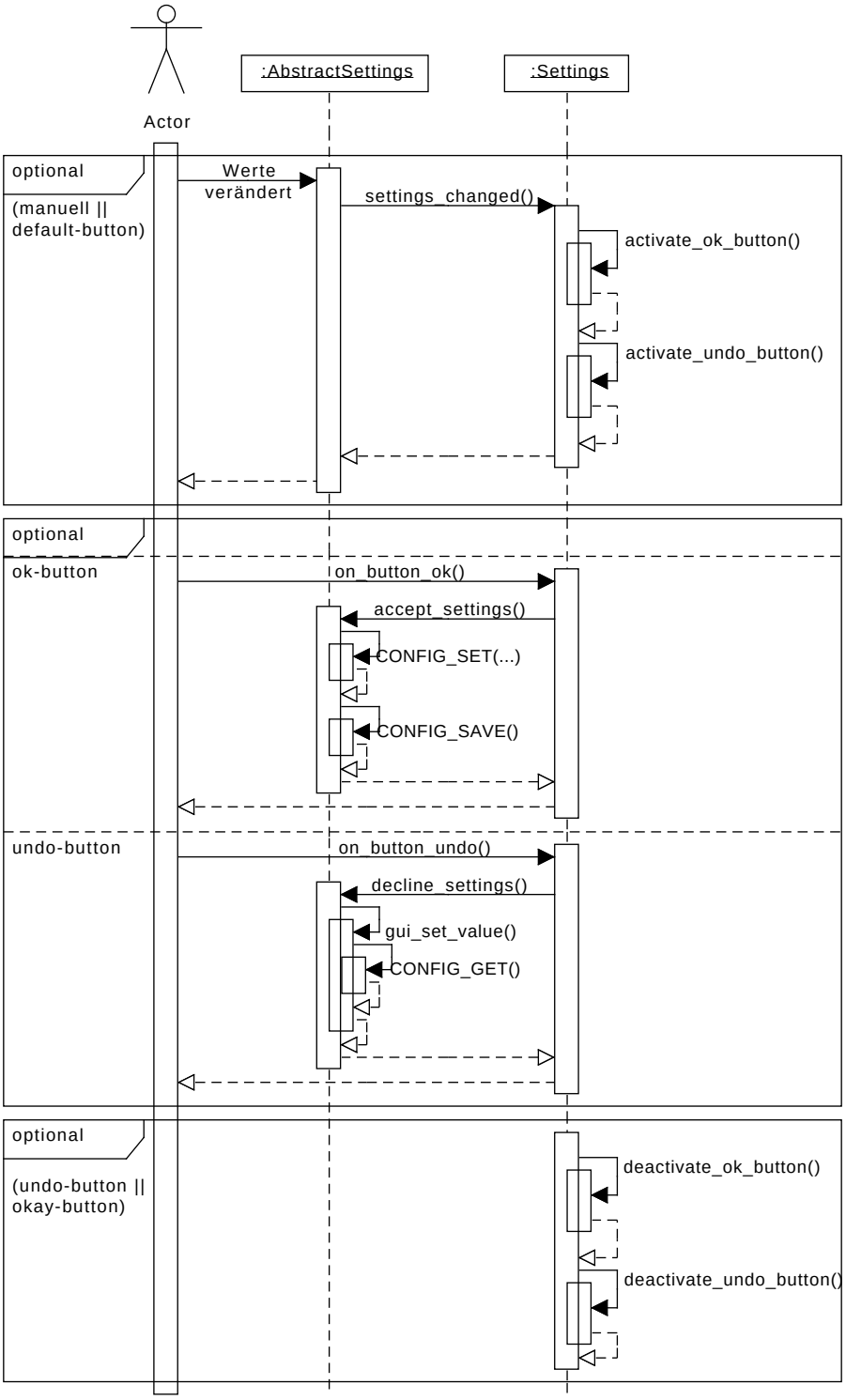


Abbildung 7.23.: Settings Sequenzdiagramm

- ,,settings.libnotify.signal" (checkbox)
- ,,settings.libnotify.timeout" (numberslider) (ausgeblendet wenn 'signal' nicht aktiviert)
- ,,settings.trayicon.tray" (checkbox)
- ,,settings.trayicon.totrayonclose" (checkbox) (ausgeblendet wenn 'tray' nicht aktiviert)

SettingsNetwork Die konkrete Klasse die den ,,Network" Tab implementiert. Folgende Einstellungen sollen geändert werden können:

- ,,settings.connection.port" (numberslider)
- ,settings.connection.host" (stringentry)
- ,,settings.connection.autoconnect" (checkbox)
- ,,settings.connection.timeout" (numberslider)
- ,,settings.connection.reconnectinterval" (numberslider)

Zusätzlich soll ein Button zum Zeigen der Avahi-Serverliste angezeigt werden.

SettingsPlayback Die konkrete Klasse die den ,,Playback" Tab implementiert. Folgende Einstellungen sollen geändert werden können:

- Eine Einstellung zum ,,Crossfade" (Überblendzeit). Diese wird vom Server gespeichert.
- ,,settings.playback.stoponexit" (checkbox)

SettingsOutputs Zeigt und verwaltet eine Liste von Outputs. Die Klasse benutzt die Funktion *fill_outputs()* von *AbstractItemGenerator* und muss daher von AbstractItemlist erben.

Wenn Änderungen übernommen werden, so wird über die Liste iteriert und für jeden Output entsprechend *enable()* oder *disable()* aufgerufen, falls der Output vorher aus-, respektive eingeschalten war.

OutputsModelColumns Die Spaltendefinition für die Outputliste. Die Liste besteht aus dem Outputnamen (einem String), einer Anzeige ob der Aktiv ist (boolean), und einen Pointer auf die AudioOutput Instanz um den entsprechenden Output en/disablen zu können.

7.10.1.6. Statistics

Statistics Eine Browserklasse, die lediglich eine Reihe von Labels verwaltet und sie bei einem Clientupdate mit den aktuellen Server Statistiken.

Folgende ,,Statistics" soll die Klasse darstellen:

- ,,Number of artists" - Anzahl der Künstler in der Datenbank
- ,,Number of albums" - Anzahl der Alben in der Datenbank
- ,,Number of songs" - Anzahl der Songs in der Datenbank
- ,,DB Playtime" - Gibt die gesamte Spielzeit der Datenbank an
- ,,Playtime" - Seit wann abgespielt wird
- ,,Uptime" - Gibt an wie lange der Server läuft
- ,,Most recent db update" - Aktuellstes Datenbankupdate

Diese Informationen werden beim Start des Freya Clients und bei jedem Datenbankupdate abgerufen.

7.11. Testfälle

7.11.1. Testen der GUI

Zum Testen der GUI des Freya Clients wurde ein Protokoll erstellt, welches den jeweiligen Testfall, das erwartete Ereignis sowie das Resultat auflistet. Hierzu wird eine Liste aller möglichen GUI Testfälle erstellt. Diese soll während der Implementierung und nach ,,Fertigstellung" der Anwendung durchgangen werden.

Die Testfälle sollen desweiteren in einer ,,einfachen-", ,,kombinierten-", einer ,,mehrfach-" Ausführung durchlaufen werden. Durch dieses Verfahren können Programm-und Anwenderfehler erkannt und beseitigt werden. Das Testen der GUI über ein Testframework wäre prinzipiell auch möglich, ist jedoch aus zeitlichen Gründen, die man für die Einarbeitung und Konfiguration für so ein Framework benötigt, nicht realisierbar.

Beispielauszug Testprotokoll ,,Einfache Ausführung":

Testfall	**Erwartetes Ergebnis**	**Ergebnis eingetroffen?**
Remove	Ein Lied aus Queue entfernen	Ja/Nein?
Clear	Alle Lieder aus Queue entfernen	Ja/Nein?
Save as Playlist	Queue als Playlist speichern	Ja/Nein?
Suchen	Nach eingegebenem Wort suchen	Ja/Nein?

7.11.2. Testen des ,,Backends"

Um die eigentliche Anwendung zu Testen, soll das ,,Cxxtest Testframework" verwendet werden. Durch den Einsatz eines Frameworks soll eine möglichst effiziente Methode des Testens bereitgestellt und das Fehlerrisiko beim Testen auf ein Minimum reduziert werden. Jedes Modul soll eine eigene ,,Testsuite" bekommen und die Ausführung soll über CMake automatisiert werden. Abarbeitung durch ein ,,test" Target, sodass einfach 'make test' gestartet werden kann. Die TestSuite Dateien bestehen aus einem Header der sich nach folgenden Konventionen richten soll:

- File- bzw Headername der jeweiligen TestSuite soll nach dem Muster `check_<modulname>.hh` aufgebaut werden
- Klassenname der jeweiligen Testsuite `<Name des Namespaces>TestSuite`
- Die TestSuite muss die TestSuite Cxxtest sowie das zu testende Modul einbinden

- Die TestSuite muss von CxxTest::TestSuite abgeleitet werden
- Der Name der zu testenden Funktion soll nach dem Muster `void test<func_name(void)` aufgebaut werden

Eine TestSuite soll wie folgt aussehen:

```
1  #include <cxxtest/TestSuite.h>
2  #include "Notify.hh"
3
4  class NotifyTestSuite : public CxxTest::TestSuite
5  {
6
7  public:
8
9       void testsend_big( void )
10      {
11          /* insert testcode here */
12
13     ...
14
15          /* check if result is valid */
16          TS_ASSERT( ... );
17      }
18
19 private:
20      /* data */
21 };
```

Listing 7.1: TestSuite Template check_notify.hh Beispielcodefragment für die Klasse ,,Notify"

7.12. Doxygen

Als interne Entwicklerdokumentation soll das Tool Doxygen[10] verwendet werden. Durch den Einsatz eines Dokumentationstools wird für die Entwicklern innerhalb des Teams eine zentrale interne Dokumentation geschaffen, auf die jederzeit zugegriffen werden kann. *Literate programming*[11] dient laut Prof. Donald E. Knuth[12] es erstklassige Dokumentation weil sie den Entwickler dazu zwingt genau über sein Software Design nachzudenken, wodurch Fehlentscheidungen automatisch minimiert und die Code Qualität verbessert werden.

[10] http://www.stack.nl/ dimitri/doxygen/
[11] http://en.wikipedia.org/wiki/Literate_programming
[12] http://de.wikipedia.org/wiki/Donald_Ervin_Knuth

7.13. Ordnerstruktur

```
Freya
    |-- bin
        |-- test_client (..)
        |-- check_log (..)
        |-- Freya
    |-- doc
    |   |-- architecture
    |   |   |-- gfx
    |   |   |-- tex
    |   |   |-- src
    |   |   |-- dia
    |   |   |-- doku.pdf
    |   |-- doxygen
    |         |-- html
    |             |-- index.html
    |-- ui
    |   |-- Freya.glade (..)
    |-- src
        |-- Avahi
        |-- AvahiBrowser
        |-- Browser
        |   |-- Database
        |   |-- Fortuna
        |   |-- PlaylistManager
        |   |-- Queue
        |   |-- Settings
        |   |-- Statistics
        |-- GManager
        |-- Init
        |   |-- Main.cc
        |-- Notify
        |-- Config
        |   |-- check_config.hh
        |-- Log
        |   |-- check_log.hh
        |-- MPD
        |   |-- check_mpd.hh
        |-- Utils
            |-- check_utils.hh
```

Abbildung 7.24.: Übersicht über die Ordnerstruktur und wichtiger Dateien

7.14. Glossar

Mainloop: In Zusammenhang mit Freya handelt sich stets um Glib Implementation eines Mainloops. Zum Verständniss sei daher die Lektüre der Glib Dokumentation empfohlen.

Browser: Mit Browser ist ein Punkt in der Sidebar von Freya gemeint. Beispiele sind somit ,,Queue", ,,Settings" oder ,,Database".

Teil VI.

Epilog

8. Probleme bei der Entwicklung

8.1. Probleme durch das Wasserfallmodell

Im Laufe der Entwicklung mussten wir leider feststellen, dass das Wasserfallmodell mit Rücksprung als Vorgehensweise für unser Projekt bei weitem nicht so gut geeignet war wie erwartet.

Aufgrund der hohen Komplexität der MPD Libraries konnten keine brauchbaren Entwürfe der Software gemacht werden, da man sich viele Funktionen des MPD völlig anders vorgestellt hatte, als sie in Wirklichkeit funktionierten.

Aufgrund dieser Tatsachen mussten wir parallel zu den eigentlichen Planungen auf horizontales Prototyping zurückgreifen[1] und erste ,,Proof-of-Concepts" schrieben um sich mit der Materie vertraut zu machen. Als Beispiel sei hier der Testclient (bin/test_client) genannt, der letzlich auch die Grundlage für die heutige Architektur darstellt, bzw. die Grundlage aus der sie entstanden ist. Auch die Config sowie erste Oberflächenfunktionen wurden seperat erstellt um deren Machbarkeit zu untersuchen.

Die Entscheidung auf Prototyping umzuschwänken war unabdingbar, da ansonsten das Projekt höchster Wahrscheinlichkeit nach gescheitert wäre. Da die Dokumentation allerdings bereits den Regeln des Wasserfallmodells nach entworfen wurde, wurde die strategische Entscheidung getroffen diese Form beizubehalten.

Desweiteren gab es auch diesbezüglich Probleme innerhalb des Gruppe, was gegen Ende des Projektzeitraums zu zeitlichen Problemen führte und aufgrund dieser nur wenige Cxxtest Testfällen exemplarisch implementiert werden konnten. Es war uns wichtig diese Punkte am Ende nochmal zu erwähnen, da Sie den Projektverlauf nicht unwesentlich beeinflusst haben.

Nichtsdestotrotz ist es gelungen das Projekt auch mit wenig Erfahrung *erfolgreich* über die Bühne zu bringen.

[1] zu diesem Zeitpunkt war das Lasten und Pflichtenheft bereits fertiggestellt

Abbildung 8.1.: Das fertige Resultat

9. Softwarerepository

Die Freya Quelltexte finden sich online auf Github:
`https://github.com/studentkittens/Freya`