JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Arbeitsprotokoll

Design Patterns

Softwareentwicklung
5BHIT 2017/18

Martin Wölfer

Version 2.6
Begonnen am 17. November
Beendet am 28. November 2017

Note:
Betreuer: RAFM & DOLD

Inhaltsverzeichnis

1 Aufgabenstellung

1.1 UML-Klassendiagramm der verwendeten Architektur inkl. Beschreibung

1.2 Kurze allgemeine Ausarbeitung zu Design Patterns

- Wie können Design Patterns unterteilt werden
- Wozu Design Patterns
- Übersicht existierender Design Patterns

1.3 Ausarbeitung zum Decorator Pattern

- Allgemeines Klassendiagramm
- Grundzüge des Design Patterns (wichtige Operationen etc.) am Beispiel des implementierten Programms inkl. spezielles Klassendiagramm
- Vor- und Nachteile
- (Weitere) Anwendungsfälle

1.4 Ausarbeitung zu einem der folgenden Design Patterns: Observer, Abstract Factory, Strategy

- Allgemeines Klassendiagramm
- Grundzüge des Design Patterns (wichtige Operationen etc.) mit einem kurzen eigenen Beispiel inkl. spezielles Klassendiagramm
- Vor- und Nachteile
- (Weitere) Anwendungsfälle

2 Design Patterns

[1]

2.1 Unterteilung

Design Patterns können in folgende Kategorien geteilt werden: **Creational Pattern**, **Strctural Pattern** und **Behaviour Pattern**

2.1.1 Creational Pattern

Bei diesem Pattern geht es darum, die Instanziierung von Klassen passend für die Situation zu gestalten. Dabei geht es einerseits um Verkapselung von Klassen um bestimmte Inhalte sicherer zu gestalten aber auch um das kreieren und kombinieren von konkreten Objekten.

2.1.2 Structural Pattern

Dieses Pattern erleichtert den Entwurf von Software indem es Beziehung zwischen bestimmten Einheiten (`Entitäten`) herstellt bzw. vorgibt.

2.2 Behaviour Pattern

Dieses Pattern modelliert komplexes Verhalten in der Software. Besonders in der Hinsicht von verbreiteten Kommunikationsmethoden werden Behaviour Patterns sehr gerne verwendet um die Flexibilität des Programmes zu erhöhen.

2.3 Nutzen

Der Hauptnutzen von Design Patterns liegt darin, sehr oft vorkommende Probleme in der Programmierung systematisch und einfach lösen zu können. Vorteil ist auch, das (die meisten) Softwareentwickler bekannt sind mit dem Design Patterns, und somit Code geschrieben kann welcher leicht verständlich ist bzw. wenn es ein Problem gibt leicht zu lösen ist.

2.4 Übersicht Design Patterns

2.4.1 Creational Patterns

- Abstract factory
- Builder
- Dependency Injection
- Factory method

- Lazy initialization
- Multiton
- Object pool
- Prototype
- Resource acquisition is initialization (RAII)
- Singleton

2.4.2 Structural Patterns

- Adapter, Wrapper, or Translator
- Bridge
- Composite
- Decorator
- Extension object
- Facade
- Flyweight
- Front controller
- Marker
- Module
- Proxy
- Twin

2.4.3 Behavioural Patterns

- Active Object
- Balking
- Bind properties
- Blockchain
- Double-checked locking
- Event-based asynchronous

- Guarded suspension
- Join
- Lock
- Messaging design pattern
- Monitor object
- Reactor
- Read-write lock
- Scheduler
- Thread pool
- Thread-specific storage

3 Decorator Pattern

3.1 Allgemeines Klassendiagramm

[2]

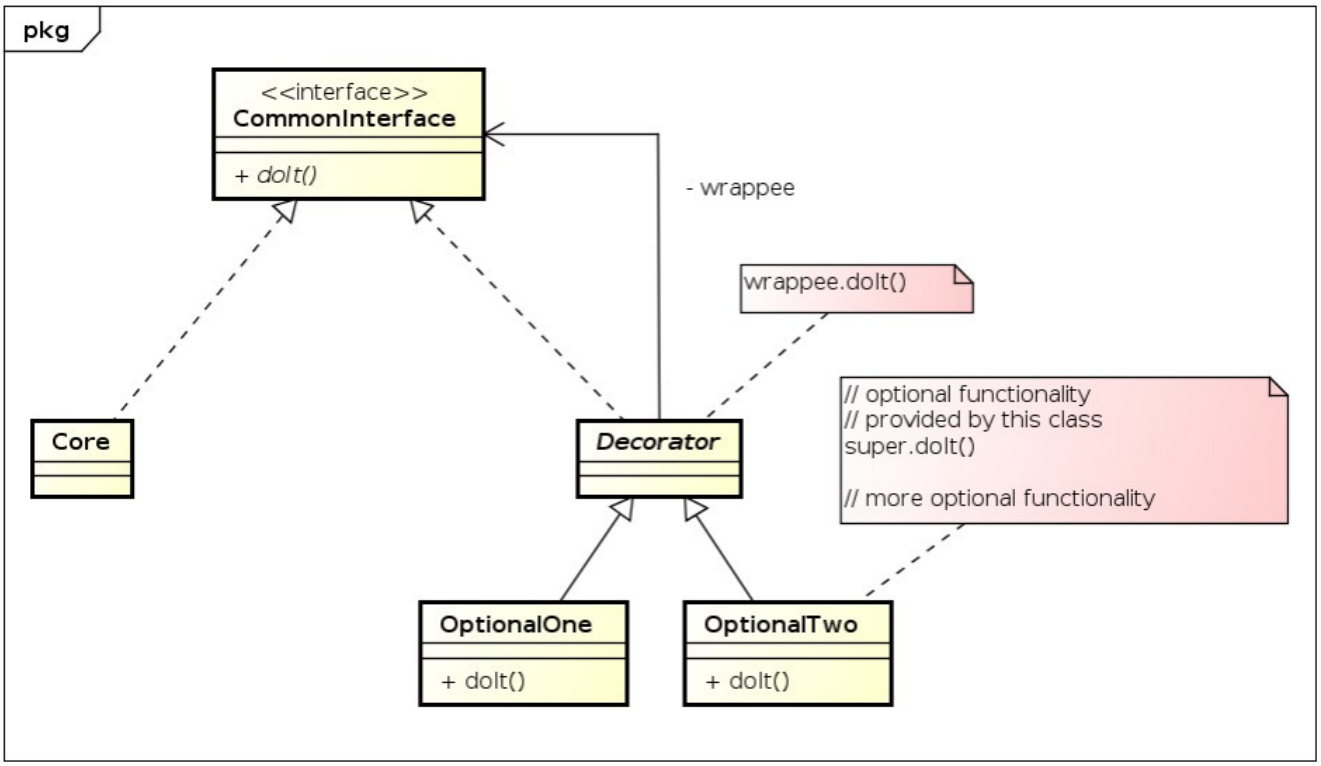

Abbildung 1: Generelles UML-Klassendiagramm zum Decorator Pattern

3.2 Eigenschaften

3.2.1 Anwendung

Auffallend beim Decorator Pattern ist die erstmals seltsam scheinende Anwendung:

```
# initialize a ServerChannel and decorate it with a StringChannel so a string can be received and sent
sc = StringChannel(ServerChannel())

# decorate with base64 channel
sc = BASE64Channel(sc)
# decorate further with AES channel
sc = AESChannel(sc)
```

Dies mag sonderbar wirken, ist aber sehr schlüssig. Der Sinn hinter dem Decorator Pattern ist es, ein bestimmtes Objekt vom Typen der **Core**-Klasse zu nehmen, und es mit einem oder mehrere **Decorator**-Klassen zu dekorieren, d.h. **erweitern**.

Dieses "erweitern" bedeutet in dem Beispiel, den `ServerChannel`(**Core**) mit dem `BASE64Channel` und dem `AESChannel` ($\rightarrow$ **Decorator**) zu erweitern, damit die Nachricht nicht im Format eines Byte-Arrays versendet wird, sondern **BASE64** und **AES** verschlüsselt, und dann versendet wird.

3.2.2 CommonInterface

Es wird üblicherweise dem Programmierer die Entscheidung überlassen ob er sich für eine **abstrakte Klasse** oder ein **Interface** entscheidet für das **CommonInterface**, aber da es in python keine Interfaces gibt wurden sogenannte **Abstract Base Classes** verwendet. Das **ABC** Package bietet auch die Möglichkeit Methoden mit der `@abstractmethod`-Annotiation zu versehen um sicherzustellen dass diese implementiert werden:

```
class Channel(ABC):
    # the main component
    def __init__(self):
        """
        Initialize a socket object
        """
        self.socket = socket.socket()

    @abstractmethod
    def printLine(self, message):
        """
        Define abstract printLine method
        :param message: The message to be printed
        :return: None
        """
        pass

    @abstractmethod
    def readLine(self):
        """
        Define abstract readLine method
        :return: None
        """
        pass
```

3.2.3 Core-Klassen

Die **Core**-Klassen erben vom **CommonInterface**, in dem Beispiel stellt die Klasse `Channel` das **CommonInterface** dar von welchem die **Core**- und **Decorator**-Klassen erben. In diesen Klassen werden die Methoden definiert vom Interface oder der abstrakten Klasse implementiert für eine bestimmte Funktionalität:

```
class ServerChannel(Channel):
    def __init__(self):
        """
        Bind the server channel for a client channel to connect
        """
        super().__init__()

        self.socket.bind(('localhost', 50000))
        self.socket.listen(5)

        (self.clientsocket, self.address) = self.socket.accept()
        print("A Client has successfully connected to the server!")

    def readLine(self):
        """
        Implement readLine method, receive data from client channel
        :return: the data received from client
        """
        data = self.clientsocket.recv(1024)
        if not data:
        self.clientsocket.close()
        else:
        return data.strip()

    def printLine(self, message):
        """
        Implement the printLine method, send response to the client
        :param message: the message to be sent to the client
        :return: None
        """
        self.clientsocket.send(message)
```

3.2.4 Decorator Base Class

Diese Klasse implementiert/erbt von der **Core**-Klasse und gibt bestimmte Funktionalitäten für die konkreten **Decorator**-Klassen vor bzw. gibt an dass diese implementiert werden müssen. Wichtig ist, dass diese Klasse einen Parameter vom Typen `ChannelDecorator` definiert, welcher als Attribut gesetzt wird:

```
class ChannelDecorator(Channel, ABC):
    # the channel decorator
    def __init__(self, channel):
        """
        Call super constructor of Channel
        :param channel: the channel which gets decaorated
        """
        super().__init__()
        self.channel = channel

    @abstractmethod
    def printLine(self, message):
        """
        Define abstract printLine method
        :param message: The message to be printed
        :return: None
        """
        pass

    @abstractmethod
    def readLine(self):
        """
        Define abstract readLine method
        :return: None
        """
        pass
```

3.2.5 Konkrete Decorator-Klassen

Diese Klassen implementieren die **Decorator** Base Class und dienen um das Objekt welches als Parameter übergeben wird auf eine bestimmte Art zu erweitern. In dem Beispiel wird der String welcher übergeben wird bei `printLine` mit BASE64 enkodiert bzw. der String welcher erhalten wird bei `readLine` mit BASE64 dekodiert:

```
class BASE64Channel(ChannelDecorator):
    # decrypt and encrypt string into a BASE64, format
    def __init__(self, channel):
        super().__init__(channel)

    def printLine(self, message):
        """
        implement printLine method with base64 encoding
        :param message: the message which is to be encoded
        :return: None
        """
        try:
        msg = base64.b64encode(message.encode())
        except AttributeError:
        msg = base64.b64encode(message)
        self.channel.printLine(msg)

    def readLine(self):
        """
        implement readLine method with base64 decoding
        :return: the bsae64 decoded string trimmed of its whitespaces
        """
        data = self.channel.readLine()
        try:
        data = base64.b64decode(data.decode())
        except AttributeError:
        data = base64.b64decode(data)
        # the trimming is necessary because the string has to be padded by the AES encoding
        return data.strip()
```

Sprich: Der String wird um eine **Kodierung** erweitert! Das `try-except` ist nötig, da die Mög-

lichkeit gegeben ist, dass der String welcher übergeben wird bereits ein Byte-Array ist, oder auch nicht.

3.3 Konkretes Klassendiagramm

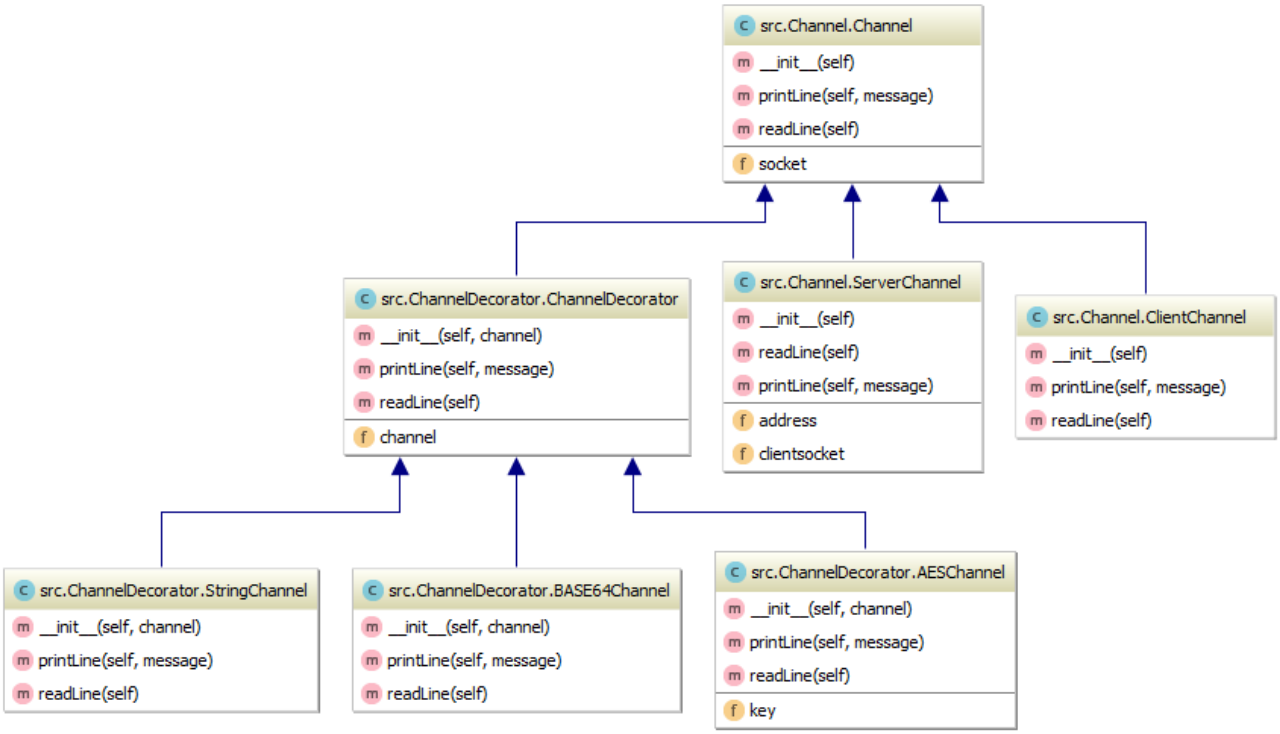

Abbildung 2: Spezifisches UML-Klassendiagramm des Beispiels zum Decorator Pattern

3.4 Vor- und Nachteile

[3]

3.4.1 Vorteile

- **Decorator**-Klassen bieten flexible Alternative für Sub-Klassen um Funktionalitäten zu erweitern
- **Decorator**-Klassen ermöglichen Veränderung während der Laufzeit anstatt den Code ”per Hand” zu ändern
- Sie lösen **Permutation**-Probleme indem man die **Core**-Klassen mit beliebig vielen **Decorator**-Klassen ”einwickeln” kann
- Folgt dem Prinzip, dass Klassen offen für Erweiterung
  aber nicht für Veränderung sein sollen

3.4.2 Nachteile

- Viele **Decorator**-Klassen führen zu vielen kleinen Objekten welche bei großen Programm zu Verwirrungen führen können
- **Decorator**-Klassen können Probleme verursachen wenn der Client stark abhängig von konkreten Komponenten ist
- Das Decorator-Pattern kann die Instanziierung von Objekten verkomplizieren, weil es nicht nur instantiiert wird, sondern auch von mehreren **Decorator**-Klassen umwickelt werden muss
- Es kann kompliziert werden, den Prozess von den miteinander agierenden **Decorator**-Klassen einen Überblick zu behalten, da diese sich gegenseitig aufrufen. Man kann es sich wie ein Schichtenmodell vorstellen, welches immer eine Stufe tiefer geht, und dann wieder alle "Stufen" hinauf geht

3.5 Weitere Anwendungsfälle

[4]

- Eine Textkomponente kann mit einer Scollbar oder einem Rahmen dekoriert werden
- Zu einer Anfrage setzbare Filter können nach dem Decorator Pattern modelliert und damit beliebig hinzugefügt oder entfernt werden
- Tuning verschiedener Autotypen mit verschiedenen Features (Tieferlegung, Spoiler, Chip)
- Verschiedene Telefontypen (Handy, klassisches Telefon) mit variablen Verhalten (Vibration, Klingeln, Lautlos, Leuchten)

4 Abstract Factory

4.1 Allgemeines Klassendiagramm

[5]

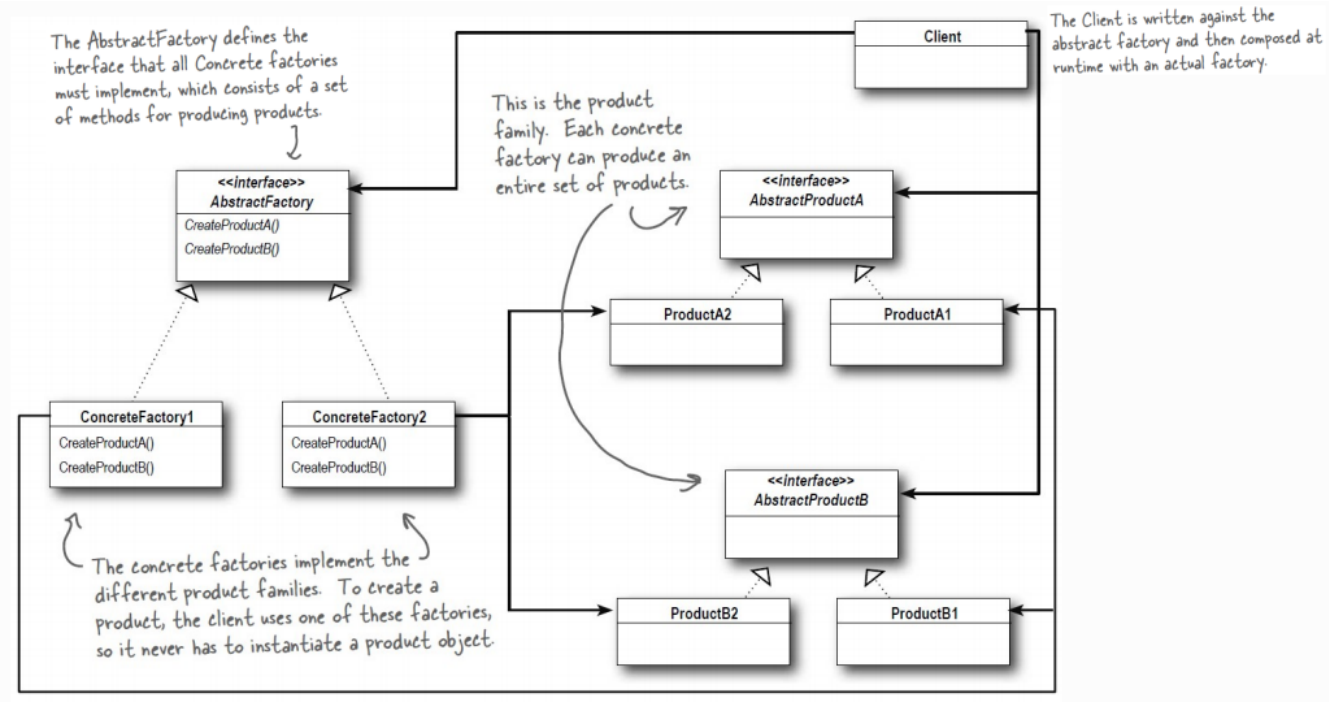

Abbildung 3: Generelles UML-Klassendiagramm zur Abstract Factory

4.2 Eigenschaften

Ich hab das Beispiel des Musik Players gewählt um das Prinzip des Abstract Factory Patterns zu erklären.

4.2.1 Anwendung

Im Gegensatz zum Decorator Pattern, unterscheidet sich die Anwendung nicht sehr stark von einer normalen Klassen welche instantiiert wird:

```
1 fabrik = MusikdatenbankFileFabrik(self.dir,self.update_function)
  fabrik.abspielen()
```

4.2.2 AbstractFactory

Diese Abstrakte Klasse gibt die Funktionalitäten für die konkreten Fabriken vor bzw. implementiert schon einige. In dem Beispiel wurde die `abspielen()`-Methode bereits implementiert:

```
class MusikdatenbankFabrik(metaclass=ABCMeta):
    """ Die Basisklasse fuer Fabriken
    """

    def __init__(self):
        self.geladen = False

    @abstractmethod
    def lade_musik(self):
        pass

    def abspielen(self):
        if not self.geladen:
            self.lade_musik()
        for song in self.playlist:
            song.abspielen()
            time.sleep(1)
        print('Wir sind am Ende der Playliste angelangt. Auf Wiedersehen!')
```

4.2.3 ConcreteFactory

In der konkreten Factory werden nun die vordefinierten Methoden implementiert. In dem Music-Player Beispiel, wir die `lade_musik()`-Methode implementiert. Es ist wichtig zu sehen, dass am Ende ein `MusikStueckFile`-Objekt erzeugt wird, die ist das sogenannte `Product`:

```
class MusikdatenbankFileFabrik(MusikdatenbankFabrik):
    # Konkrete FileFactory implementiert abstrake MusikdatenFactory
    def __init__(self, dir, update_function):
        """
        Call super Constructor and assign class attributes
        :param dir: the directory where songs get searched to be added to the playlist
        :param set_data: callback function which can get called in order to update the data in the gui
        """
        MusikdatenbankFabrik.__init__(self)
        self.playlist = []
        self.dir = dir
        self.update_function = update_function

    def lade_musik(self):
        """
        Fuegt der Playlistt MockupMusikstuecke hinzu welche automatisch ausgelesen werden

        :return: None
        """
        # iterate through given directory
        for dirname, subdirs, files in os.walk(self.dir):
            for filename in files:
                # get the file extension of each file
                extension = os.path.splitext(filename)[1][1:]
                # check if the extension is a music file
                if extension.lower() in ["mp3","wma", "wav","ra", "ram", "rm", "mid", "flac", "ogg"]:
                    # get each mp3 file in this directory
                    file = pyglet.media.load(dirname + os.path.sep + filename)

                    # get each specific info needed
                    song_laenge = file.duration
                    song_titel = file.info.title.decode()
                    song_interpret = file.info.author.decode()
                    song_album = file.info.album.decode()
                    # append each song with the informations to the playlist and the gui object
```

```
                    self.playlist.append(MusikstueckFile(file, song_laenge, song_titel, song_interpret,
                        song_album, self.update_function))
                else:
                    print("%s is not a music file!" % filename)
        if len(self.playlist) == 0:
            # if no music file was found, print out error message and exit program
            print("No suitable Files found. Please restart Program and choose another Folder")
            sys.exit(0)
```

4.2.4 AbstractProduct

Von den jeweiligen konkreten Factorys, werden Produkte erstellt. Diese Produkte besitzen auch eine **abstrakte** Klasse, von denen alle **konkreten** Produkte erben, das **AbstractProduct**. Im Beispiel, werden im **AbstractProduct** lediglich Attribute festgelegt und Methoden zum implementieren definiert:

```
class Musikstueck(metaclass=ABCMeta):
    """ Die basisklasse fuer alle Musikstuecke
    """

    def __init__(self, titel, interpret, album):
        self.titel = titel
        self.interpret = interpret
        self.album = album

    @abstractmethod
    def abspielen(self):
        pass
        class Musikstueck(metaclass=ABCMeta):
        """ Die basisklasse fuer alle Musikstuecke
        """

    def __init__(self, titel, interpret, album):
        self.titel = titel
        self.interpret = interpret
        self.album = album

    @abstractmethod
    def abspielen(self):
        pass
```

4.2.5 ConcreteProduct

Die **konkreten** Produkte implementieren die abstrakte Produkt-Klasse. In dem Beispiel wird die `abspielen()`-Methode implementiert, welche das File abpsielt und das GUI mittels der übergeben `update_function()` updated:

```
class MusikstueckFile(Musikstueck):
# File Produkt Klasse implementiert abstrakte Produkte Klasse durch tatsaechliche Ausgabe via pyglet

def __init__(self, file, laenge, titel, interpret, album, update_function):
    """
    Adds 2 additional parameters to the super constructor
    :param file: Thile file object which gets played
    :param laenge: The length of the mp3 file in order to pause the playlist for this amount
    :param titel: Title of the song, read out of the mp3 file
    :param interpret: Artist of the song, read out of the mp3 file
    :param album: Album of the song, read out of the mp3 file
    :param update_function: function of the gui object to update the data
    """
    # call super constructor
    Musikstueck.__init__(self, titel, interpret, album)
```

```
        # set the 3 additional parameters to class attributes
17      self.file = file
        self.laenge = laenge
19      self.update_function = update_function

21  def abspielen(self):
        """
23      Play the file object given and update information
        In order for the playlist not to play every song at the same time, everytime a song is played, time.
            sleep for the length of the song
25      :return: None
        """
27      self.file.play()
        self.update_function(str(self.interpret), str(self.titel), str(self.album))
29      time.sleep(self.laenge)
```

4.3 Konkretes Klassendiagramm

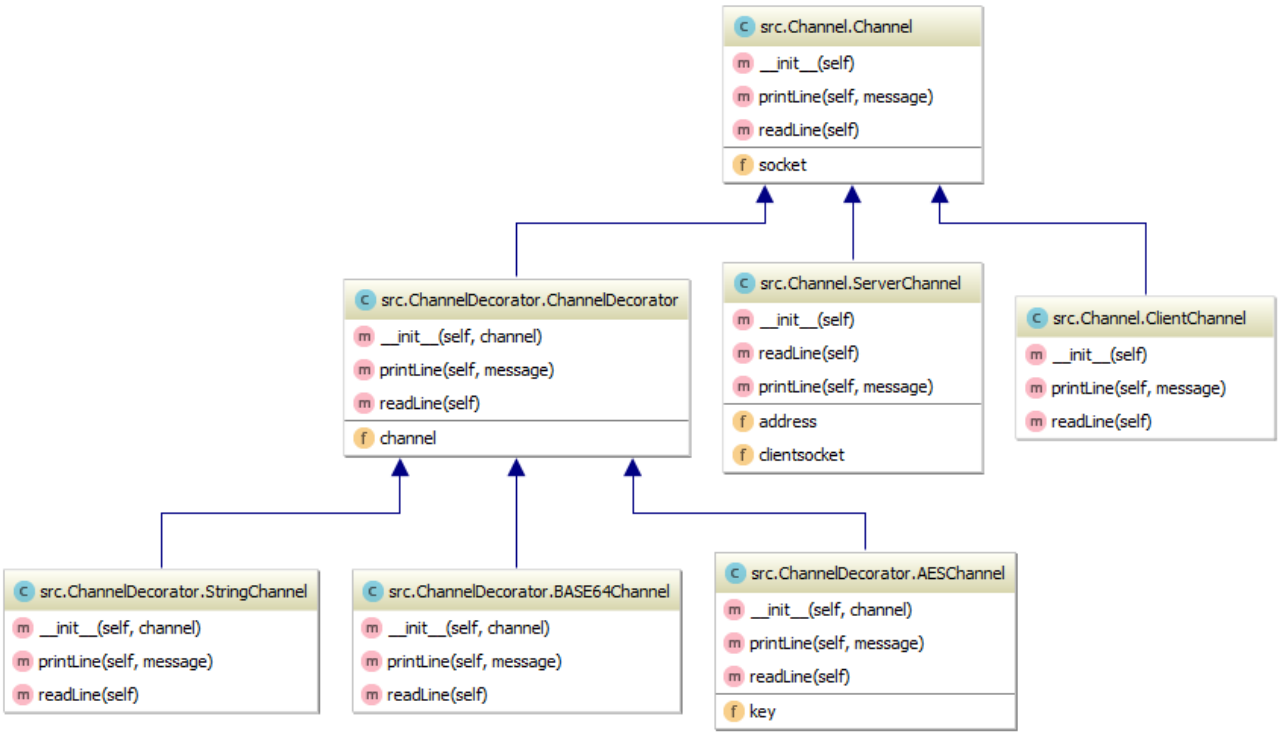

Abbildung 4: Spezifisches UML-Klassendiagramm des Beispiels zur Abstract Factory

4.4 Vor- und Nachteile

[6]

4.4.1 Vorteile

- Es ist möglich bestimmte Implementationsteile zu verstecken hinter einer factory
- Das Pattern macht Testing und Mocking sehr einfach
- Folgt dem Prinzip "loose Kopplung"

4.4.2 Nachteile

- Code ist manchmal schwerer zur verstehen da er hinter noch einer Schicht von Abstraktion steckt
- Generell Code-Komplexität ist höher da viele neue, und vorallem kleine, Klassen erstellt werden
- Verletzt theoretisch gesehen dass Prinzip, welches besagt dass zu Interfaces programmiert werden soll

4.5 Weitere Anwendungsfälle

[7]

- In einem System müssen Dateien verschiedenen Formats eingelesen und unterschiedlich weiterverarbeitet werden. Dafür wird für jedes Format eine Reader-Klasse erstellt und für jede mögliche Weiterverarbeitung eine Transformer-Klasse. Dabei ist es unbedingt erforderlich, dass zu einem Reader auch der passende Transformer gewählt wird. Um dieser Anforderung zu genügen, kann das Abstract Factory verwendet werden.
- Die Persistenzlogik einer Applikation kapselt den Zugriff auf die Datenbank. Dabei soll diese natürlich nicht nur auf einer MySQL-Datenbank ihre Daten ablegen können, sondern auch mit anderen Datenbanken (Oracle etc.) interagieren können. Dazu sind allerdings verschiedene DBConnection- und jeweils korrespondierende DBCommand-Objekte von Nöten. Nun könnte nach dem Abstract Factory Pattern eine OracleDBClientFactory zwei Methoden zum Erhalten von solchen Objektpaaren bereitstellen. Die Persistenzlogik arbeitet auf Interfaces und hat keine Kenntnis von den verwendeten datenbankspezifischen Objekten

Literatur

[1] Wikipedia. Wikipedia Design Patterns.

[2] elearning. DesignPatterns Decorator.

[3] Morgan Neill. Decorator Pattern Pros and Cons.

[4] Philipp Hauer. No Title.

[5] elearning. DesignPatterns Abstract Factory.

[6] Pete Smith-Kline. Factory Design Pattern Pros and Cons.

[7] Philipp Hauer. Abstract Factory.

Tabellenverzeichnis

Listings

Abbildungsverzeichnis