Seminararbeit

Thema:

The Extension Interface design pattern

6. Semester Diplominformatik
FH-Kempten

Autor: Kacper Bak
Matrikelnummer: XXXXXX
Betreuer: Prof. Dr. rer. nat. Georg Hagel

Unterschrift: _____

Inhaltsverzeichnis

Prolog

Die Software-Entwicklung ist in der Welt der Informatik eine der anspruchsvollsten Disziplinen. Für die Erstellung eines hochwertigen Anwendungsproduktes bedarf es einer genauen Erfassung der Zielanforderung, welche die Implementierung umsetzen soll. Nachdem die Funktionalität auf einer ausgesuchten Architektur erstellt wurde, folgt das Testen und die abschließende Integration beim Kunden. Dieser Weg ist lang, kostenintensiv und kann auf unterschiedliche Weisen begangen werden. Das Produkt ist aber immer eine spezifische Problemlösung.

Im Laufe der Zeit verändern sich meistens Anforderungen und/ oder es kommen welche dazu.

Unsere Problemlösung verliert dadurch an Effizienz und Genauigkeit und muss angepasst werden. Es existiert kaum eine Softwarelösung, die langfristig nicht einem Wandel unterliegt. Wird diese Behauptung in den Raum gestellt, liegt es meistens daran, dass das Produkt einfach nicht benutzt wird. Bei der Anpassung der Software spielt der Grad der Komplexität meist die entscheidende Rolle. Je vielschichtiger und anspruchvoller unsere Lösung ist, desto zeitintensiver und fehlerbehafteter wird die Umsetzung der Lösung sein. Durch das Hinzukommen neuer Funktionalität wächst der Informationsgehalt und die damit verbundene Komplexität. Folgt unsere Erweiterung keiner davor festgelegten/ m Regel/ Muster verliert das Anwendungsdesign, nach jeder Veränderung, immer mehr an Struktur. Diese Entwicklung beschrieben Prof. Meir M. Lehman und Laszlo Belady bereits 1974 in ihren acht Gesetzen der Softwareevolution.[1]
Eine weiteres Problem ensteht bei voneinander abhängigen Komponenten, die einseitig modifiziert werden. Der nicht veränderte Teil eines Systems wird nach dem Eingriff inkompatibel zum Modifizierten. Dieses Verhalten ist in der Praxis inakzeptabel.
Das Extension Interface Design Pattern befasst sich mit der Problematik der Veränderung und zeigt einen Weg auf, wie Software Designer/ -Architekten ihre Anwendung strukturieren können, damit diese resistent gegenüber Modifikation und Erweiterung wird. Aus diesem Grund kann man dieses Entwicklungsmuster den Struktur- bzw. Architekturmustern zuordnen.

Pattern Definition

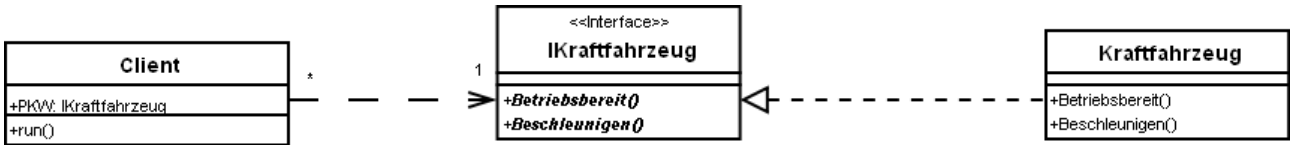
> “The Extension Interface design pattern1 prevents bloating of interfaces and breaking of client code when developers add or modify functionality to existing components. Multiple extensions can be attached to the same component, each defining a contract between the component and its clients.”[2]

Dieser Definition entsprechend widmet sich diese Arbeit. Im Folgenden werde ich die Problematik der Veränderung demonstrieren und einen Lösungsweg, anhand reflektierter Beispiele aus Douglas C. Schmidt's Werk, aufzeigen.

Das allgemeine Interface

Damit erstellter Anwendungscode größtmögliche Flexibilität erreicht, sollte beim Design immer darauf geachtet werden, niemals direkt zur Implementierung, sondern zu einem unbestimmten Datentyp wie einem Interface oder einer abstrakten Klasse hin zu programmieren.[3]
Das Verhalten der Anwendungsimplementierung mit unbestimmten Datentypen lässt sich einfacher beeinflussen, als mit konkreten Typen. Hierzu ein Beispiel:

1 Wikipedia, Lehman's laws of software evolution
2 Douglas C. Schmidt, Extension-Interface.doc, Seite 1.
3 Eric & Elisabeth Freeman - Kathy Sierra & Bert Bates, Head First Design Patterns, S. 11.

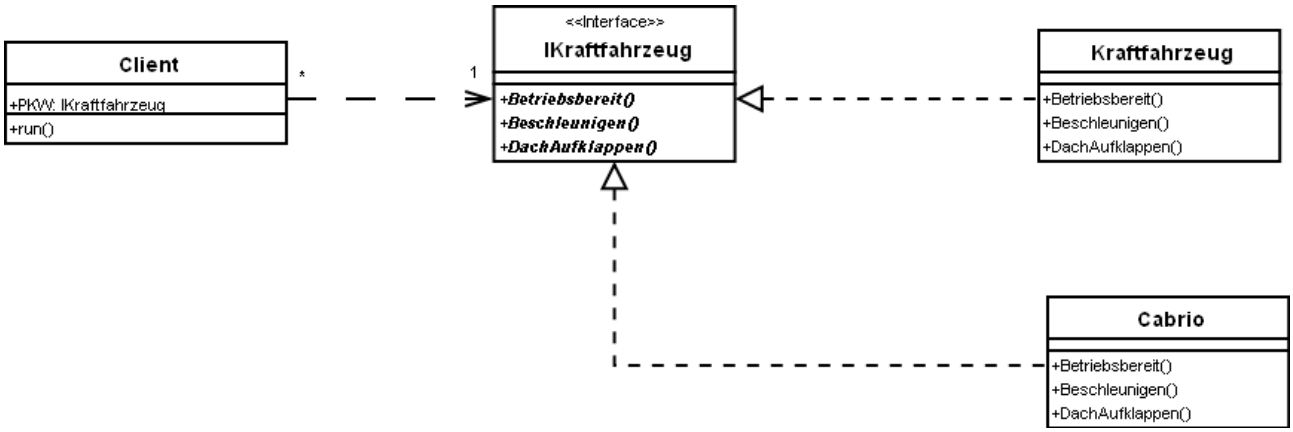


Das Interface “IKraftfahrzeug” stellt Dienste für den Client bereit. Die konkrete Implementierung dieser Services erfolgt in der Komponente “Kraftfahrzeug”. Hier wird das Verhalten des einzelnen Dienstes konkret definiert. Der Client greift also niemals direkt auf die Implementierung/ Komponente, sondern auf die ihm bereitgestellte Schnittstelle “IKraftfahrzeug” zu. Die Umsetzung der Anwendungsfunktionalität, sprich das Benutzen der angebotenen Methoden, ist in der “run” Methode des Clients realisiert.

Gegen diese Architektur ist zum aktuellen Kenntnisstand nichts einzuwenden. Die Nachteile entstehen erst nachdem sie integriert wurden und sich nach unbestimmter Zeit das Verhalten eines Dienstes verändert, bzw. weitere Funktionalität hinzukommen soll. Hauptvoraussetzung für eine erfolgreiche Modifikation ist das Bestehenbleiben der Clientfunktionalität. Der Clientcode darf unter keinen Umständen, durch Veränderungen an der Schnittstellenimplementierung, unbrauchbar werden. Diese Voraussetzung ist auch sehr praxisnahe, man stelle sich nur die Situation vor: Ein Update der Middleware wird vollzogen, jedoch wird nur bei einem Teil der Clients die Aktualisierung ausgeführt. Somit wäre der nicht aktualisierte Teil funktionsunfähig.

Wir bauen unsere Architekur aus und fordern von ihr eine neue Funktionalität. Unser Fahrzeug soll die Fähigkeit haben sein Dach auf zu klappen. Üblicherweise können dies nur Fahrzeuge, welche diesem Verhalten entsprechend angepasst sind. Da Cabrios nicht nur dem Kriterium des offenen Verdecks, sondern meistens einen sportlichen Fahrstil aufweisen müssen, wird die Implementierung der Methode “Beschleunigen” sich von der eines gewöhnlichen Kraftfahrzeuges unterscheiden. Komponenten sollten möglichst unabhängig von Ihrer Umgebung definiert werden, damit Abhängigkeiten reduziert und ihre Wiederverwendbarkeit ausgenutzt werden kann. Deshalb erstellen wir an diesem Punkt eine neue Komponente: ”Cabrio”[4].

4 Johannes Siedersleben, Moderne Softwarearchitektur, S. 43.



Damit unser Client weitherhin beide Komponenten nutzen kann, entschließen wir uns die neue Funktionelität "DachAuklappen" in das allgemeine Interface "IKraftfahrzeug" aufzunehmen und allgemeine Schnittstelle von der neuen Komponente "Carbio" implementieren zu lassen.
Das Resultat erfüllt zwar unsere Vorgaben, zerstört allerdings die Komponentenarchitektur mit jeder hinzukommenden Funktionalität und Komponente. Wir können zwar das neue Verhalten der Methoden "Beschleunigen" und "DachAufklappen" in der neuen Komponente "Cabrio" unabhängig von "Kraftfahrzeug" umsetzen, jedoch muss die neue Methode "DachAufklappen" in "Kraftfahrzeug" leer implementiert werden. Mit jeder hinzukommenden Funktionalität sind alle Komponenten dazu gezwungen, diese zu implementieren, auch wenn der Kontext in dem sie steht, zur neuen Funktionalität unpassend ist. Das allgemeine Interface bläht sich auf, die Komponentenarchitektur wird destabilisiert, die Pflege von Komponenten wird anstrengend und fehleranfällig[5]. Die Ursache dafür ist unsere unabhängige/ allgemeine Definition des "IKraftfahrzeug" Interfaces und die daraus entstehenden Redundanzen in unseren Komponentenimplementierungen. Das Problem vor dem der Softwarearchitekt nun steht, ist, das Design hin zu einer allgemeinen Schnittstelle, mit wenig Abhängigkeiten dafür aber redundanten Implementierungen in Kauf zu nehmen oder spezifische Interfaces zu definieren und bei einer Änderung die Anwendung fast neu zu schreiben. Das Extension Interface bietet an dieser Stelle einen Kompromiss, den jeder Softwarearchitekt zumindest kennen sollte.

Prinzip der Lösung

Auf der einen Seite ist es unser Ziel eine spezifische Schnittstelle bereitzustellen, auf der anderen Abhängigkeiten zu reduzieren. Damit widersprechen sich diese Ziel auf den ersten Blick. Kombinieren wir aber allgemeine und spezifische Funktionalität entsteht ein neuer Lösungsansatz.

1. Schritt

Wir gruppieren alle Methoden, die voneinander abhängig sind und semantisch dem gleichen

5 Douglas C. Schmidt, Extension-Interface.doc, Seite 2.

Kontext entsprechen zu einer Rolle. Wir versuchen dadurch gegenseitige Abhängigkeiten in einer Rolle zu vereinen.

2. Schritt

Funktionalität, welche keiner bereits bestehenden Rolle zugewiesen werden kann, definiert ihre eigene Rolle.

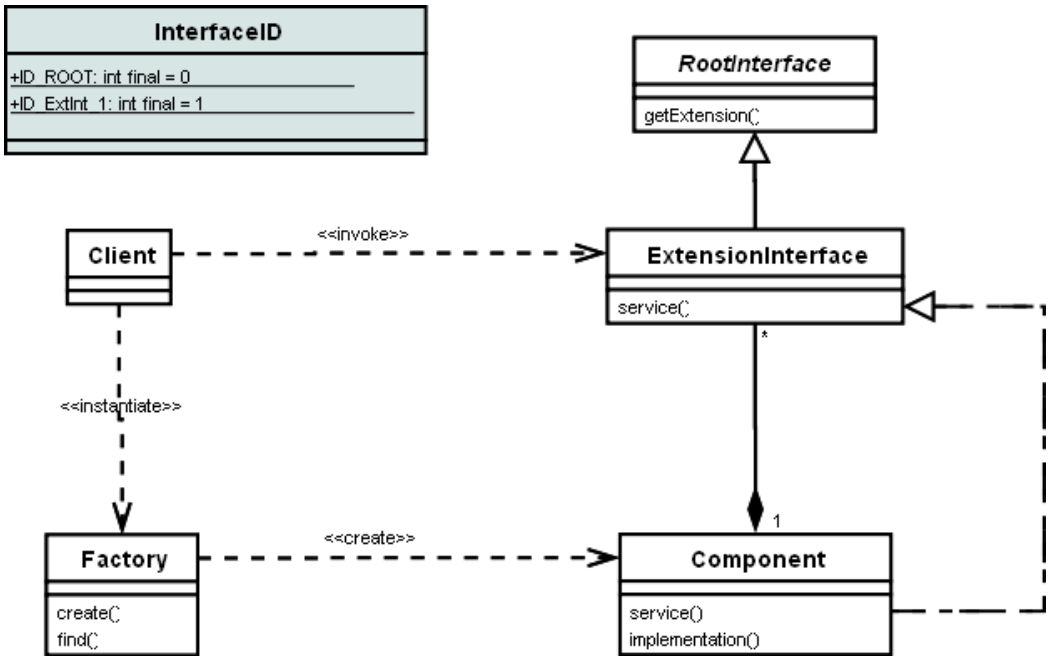
3. Schritt

Jede Rolle wird durch ein spezifisches Extension Interface referenziert.

4. Schritt

Damit wir zwischen den einzelnen Rollen navigieren/ wechseln können, muss jedes Extension Interface die Möglichkeit bieten zu einer anderen Rolle zu navigieren/ zu wechseln . Es handelt sich also um eine allgemeine Funktionalität, welches jedes Extension Interface bereitstellen muss.

Struktur des Extension Interface Patterns



6

Element:	Beschreibung der Aufgaben:
RootInterface	• Stellt allgemeine Funktionalität bereit, die jedes Extension Interface bereitstellen muss. • Die ”getExtension” Methode muss von jeder Komponente implementiert werden, damit jedes Extension Interface in der Lage ist, ein weiteres Extension Interface anzufordern und

6 Douglas C. Schmidt, Extension-Interface.doc, Seite 6.
Anmerkung: Die Komposition in diesem Diagramm bezieht sich auf eine Implementierung des Patterns mit nested Classes, siehe Kapitel Implementierung.

	damit die Rolle zu wechseln.
Extension Interface(s)	• Gruppiert semantische Funktionalität und reduziert damit Abhängigkeiten. • Erbt vom RootInterface die allgemeine Funktionalität. • Stellt die Rolle einer Kontextgruppe dem Client zur Verfügung. • Kapselt die angebotenen Dienste nach außen.
InterfaceID (Zusatz)	• Dient der eindeutigen Identifikation der Extension Interface's und hält damit die Rollen der Komponenten auseinander.
Factory	• Erzeugt die passende Komponente zum angeforderten Interface mit der “create” Methode.
Client	• Implementiert die Anwendungsfunktionalität, d.h.: Der Client verwendet die Dienste, welche durch die Extension Interface's bereit gestellt werden. • Der Zugriff des Clients auf den Komponenten-Service erfolgt immer über das passende Extension Interface, niemals direkt auf die Komponente selbst. • Lässt die Komponenten über zugehörige Factories erstellen
Komponente	• Implementiert die Funktionalität der zugehörigen Extension Interface's • Kann mehr als ein Extension Interface implementieren. • Gibt bei Aufruf der “getExtension” Methode die passende Interface Referenz zurück. • Wird von einer zugehörigen Factory erstellt.

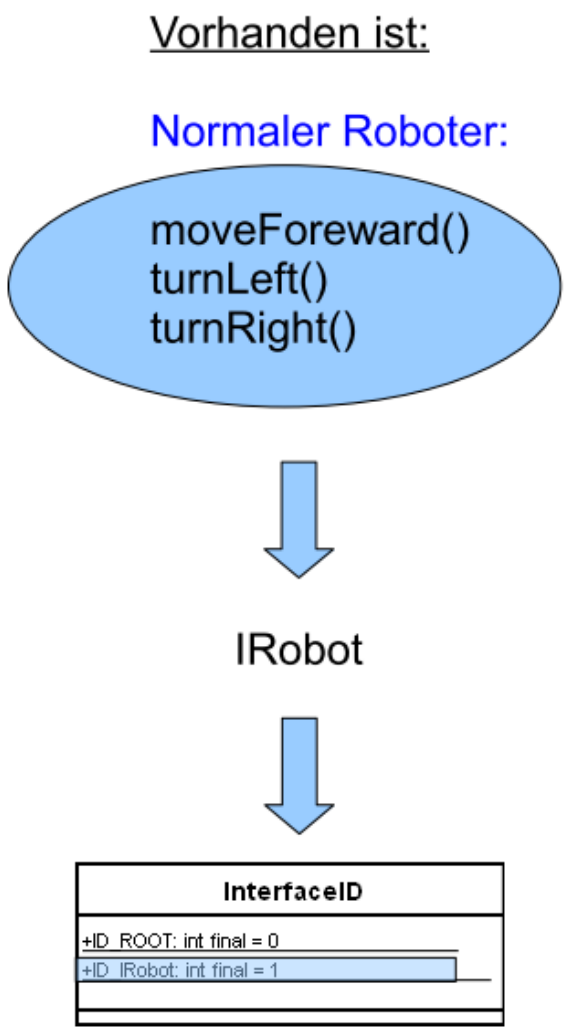
Beispiel 1. Hinzufügen eines Extension Interface

Im Folgenden werde ich das Verhalten des Patterns anhand eines selbst erstellten Beispieles demonstrieren. Die Implementierung der Verlinkung erfolgt hier in der Variante “Mehrfachvererbung”[7] indem eine Komponente mehrere Extension Interfaces implementiert und alle Extension Interface's vom RootInterface erben. Eine weitere Möglichkeit der Implementierung wird im Kapitel: **Implementierung** angerissen.

Nehmen wir an das 30. Jahrhundert wurde überschritten und die Menschheit hat es endlich geschafft ein maschinelles Ebenbild ihres Gleichen herzustellen. Diese Entwicklung geschah in Friedenszeiten und Roboter sind in der Lage motorische Funktionen, wie die der Fortbewegung auf zwei Beinen, umzusetzen. Entsprechende Roboter wurden bereits hergestellt und fristen ihr Dasein unter uns.

Hier eine abstrakte Darstellung mit den wichtigsten Merkmalen:

7 Douglas C. Schmidt, Extension-Interface.doc, Seite 6.



Rolle:

Die Grundfunktionalität des Bewegungsapparates wurde zu einer semantischen Funktionsgruppe = Rolle zusammengefasst.

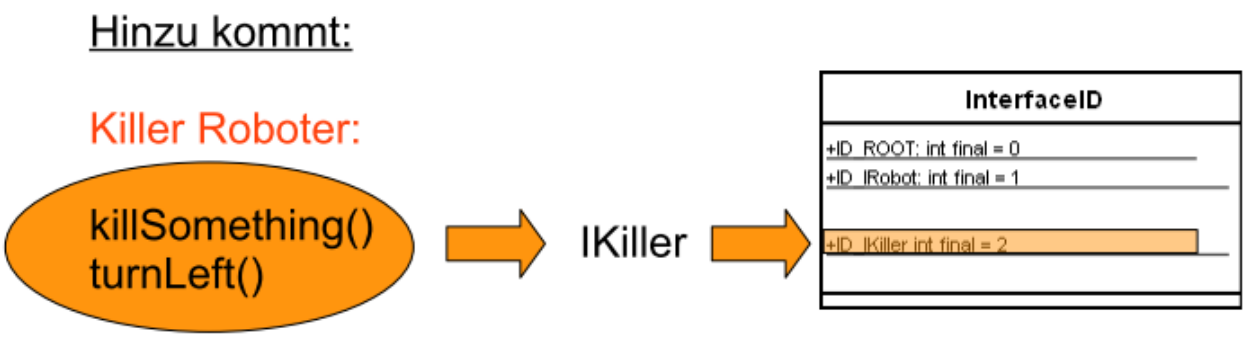
Extension Interface: IRobot

Der definierten Rolle wurde ein Extension Interface zugeordnet.

ID

Jedes Extension Interface muss über eine eindeutige ID identifiziert werden können. In unserem Fall wird eine einfache Integer Konstante definiert, hier kann aber auch ein intelligenter Algorithmus eine UUID generieren.

Da Geschichte sich in einem unbestimmten Zeitintervall wiederholt, wird lange nachdem die ersten Roboter laufen können, die Entscheidung getroffen, diese auch gegen ihre Schöpfer einzusetzen.



Wir bilden, wie beim ersten Vorgehen, eine neue Rolle, da das Verhalten unseres Roboters

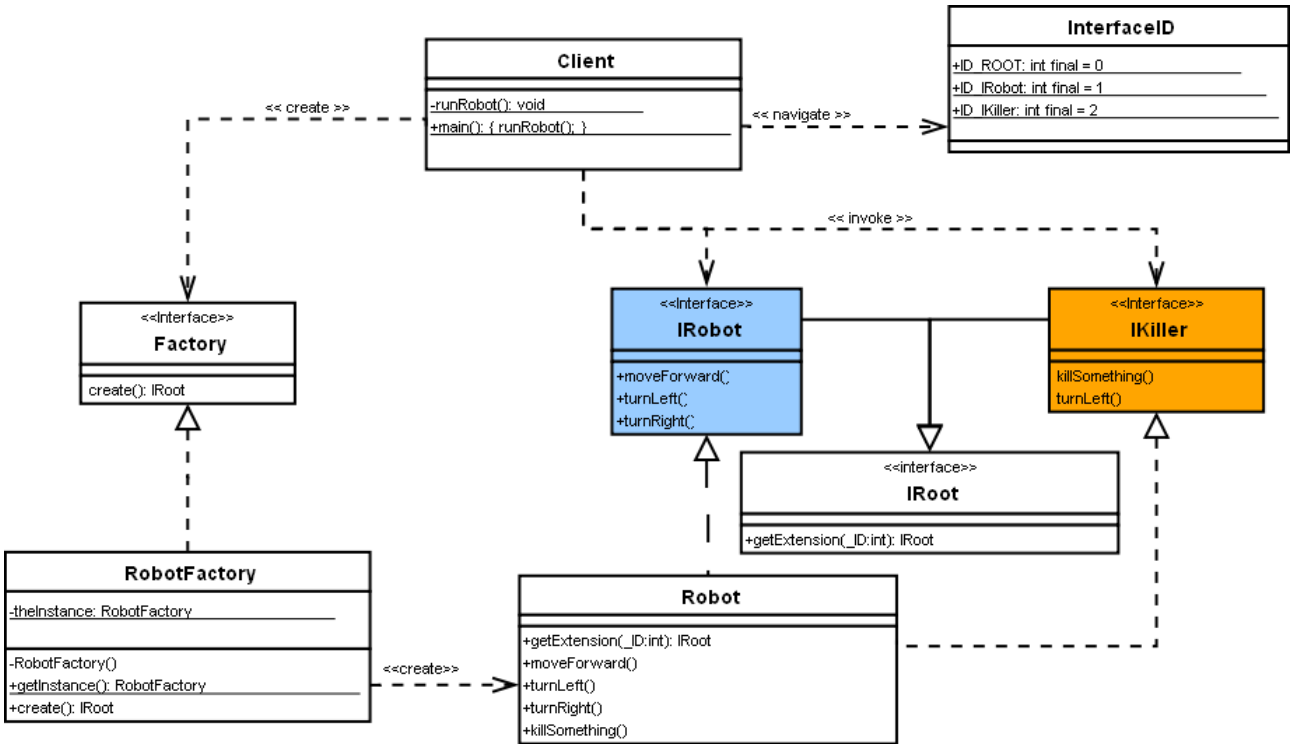
erweitert werden soll.

Wichtige Anmerkung:

An dieser Stelle ist anzumerken, dass bei der Erweiterung einer Funktionalität, alle bisherigen Funktionen (moveForeward, turnLeft, turnRight) in das neue Extension Interface übernommen und die neu Funktionalität (killSomething) hinzugefügt wird. Damit ist dann gewährleistet, dass der ClientCode nicht gebrochen wird.

Zweck dieses Beispieles ist es aber, nicht nur die Form der Veränderung darzustellen, sondern explizit darauf hinzuweisen, dass ausschließlich die Funktionalität an den Client bereitgestellt wird, welche im übergebenen Extension Interface auch definiert ist.

Klassendiagramm



Damit wir dem Prinzip: ”Design to an Interface” gerecht werden, lassen wir den Client nicht direkt auf eine “RobotFactory” zugreifen, sondern realisieren auch diesen Zugang über ein allgemeines Interface “Factory”. Da ich mich in dieser Arbeit auf das Extension Interface Pattern konzentriere, behandle ich keine vorkommenden Pattern's ausführlich, sondern wende diese im Kontext meines Beispieles an. Der Zweck der Factory ist es, mit dem Aufruf der “create” Methode, eine Komponente des entsprechenden Factorytypes zu erstellen. In der Robotfactory wird das “Singleton” Pattern verwendet, welches sicherstellt, dass jeweils nur eine Instanz eines Fabriktypes

zur Laufzeit vorhanden ist[8].

Erstellung und Initialisierung neuer Komponent-Instanzen

Bevor der Client die vom Extension Interface bereitgestellten Service's nutzen kann, muss eine Instanz der Komponente erstellt werden, deren Dienst das Extension Interface anbietet. Dies wird verständlicher, wenn wir es anhand unseres Beispieles betrachten.

1. Schritt

Wir erstellen ein Factory des Types unserer Komponente:

```
// create factory //
Factory robotfactory = RobotFactory.getInstance();
```

2. Schritt

Wir haben nun eine Factory die Roboter Komponenten herstellt, also erstellen wir einen Roboter.

```
// create robot COMPONENT //
IRoot robot = robotfactory.create();
```

3. Schritt

Um den Service des Extension Interface's zu nutzen, benötigt der Client eine Instanz davon. Dabei muss er der entsprechenden Komponente mitteilen, welche Rolle der Komponente verwendet werden soll. Diese Information ist in der Assoziation: Extension Interface → ID zugeordnet worden und kann über die Klasse InterfaceID abgefragt werden.

```
// get ExtensionInterface of service //
IRobot r1 = (IRobot) robot.getExtension(InterfaceID.ROBOT_ID);
```

4. Schritt

```
// use service from EXT_1 //
r1.moveForeward();
```

Nun kann der Client alle Services, die in IRobot deklariert wurden, über r1 benutzen.

Unterschiede

Das Sequenzdiagramm von Douglas C. Schmidt unterscheidet sich von unserer Implementierung. Dies liegt an der Wahl der Verlinkung, die wir zu Beginn dieses Beispieles bereits erwähnten.

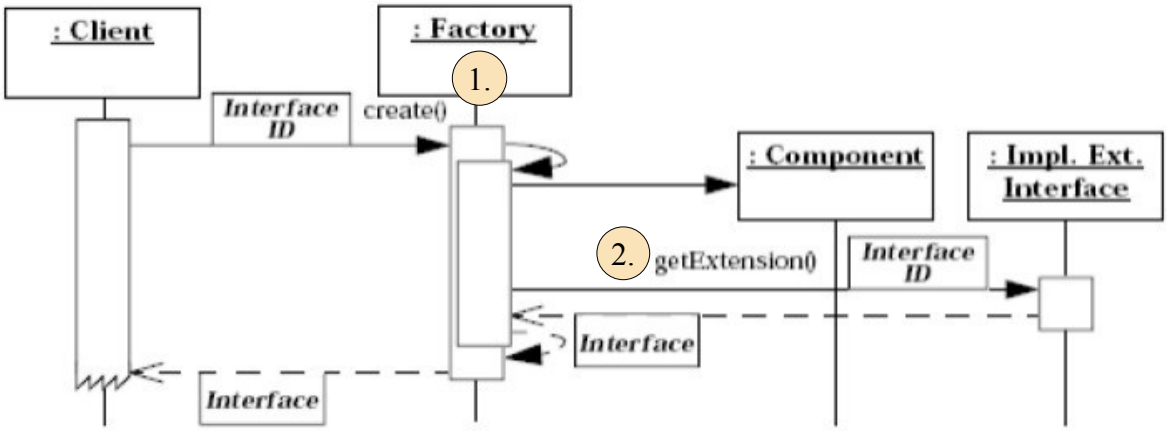
1. Punkt

8 Eric & Elisabeth Freeman - Kathy Sierra & Bert Bates, Head First Design Patterns, S. 169.

Der Client greift zuerst auf das Factory Interface zu und erstellt einen FactoryTyp passend zur Komponente, die erstellt werden soll.

2. Punkt

Die Komponente, nicht die Factory, ruft die “getExtension” Methode auf, über die das erste Extension Interface geladen wird.



9

Navigation

Darunter versteht der Client den Rollenwechsel der Komponente. Der gleiche Roboter, welcher erstellt wurde, implementiert nicht nur das gutartige Verhalten der Fortbewegung, sondern auch eine destruktive Methode.

1. Schritt

Um das neue Verhalten der Komponente zu nutzen, benötigen wir den bereits erstellten Roboter, eine Instanz des Extension Interface's “IKiller” und die ID, hinter welcher sich unser ExtensionInterface verbirgt.

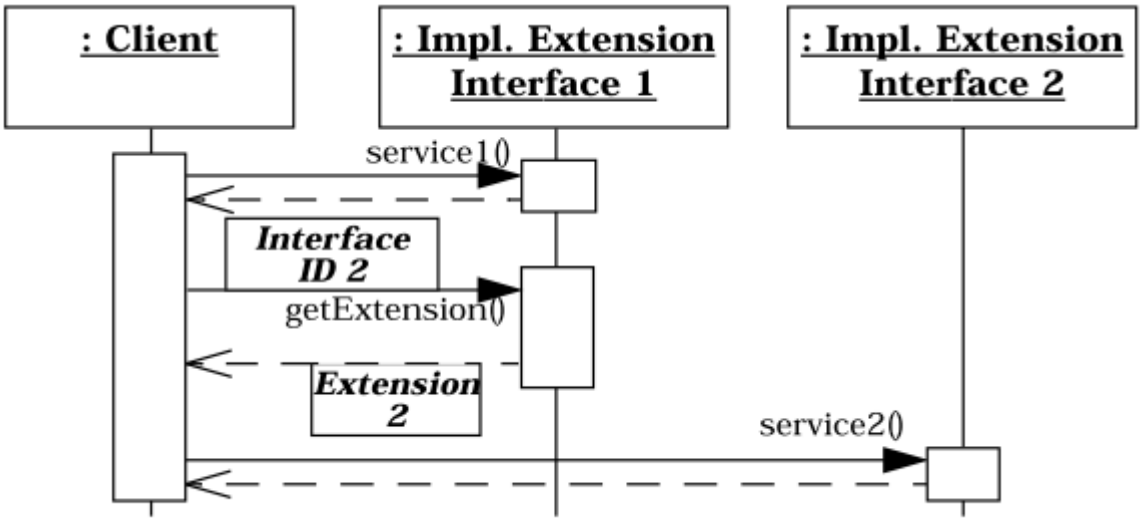
```
//get ExtensionInterface of service //
IKiller meetYOURmaker = (IKiller) robot.getExtension(InterfaceID.KILLER_ID);
```

2. Schritt

Nun kann der Client die neuen Dienste von “**meetYOURmaker**” nutzen.

```
//use services from EXT_2 //
meetYOURmaker.killSomething();
```

9 Douglas C. Schmidt, Extension-Interface.doc, Seite 7.



10

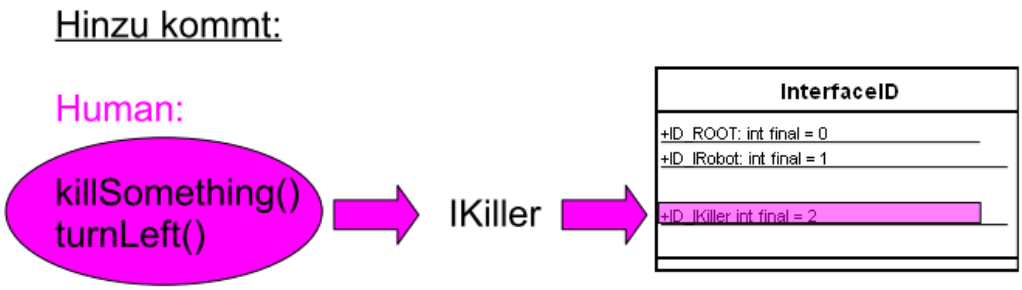
In unserem Beispiel ist das “IKiller” Interface jedoch so deklariert, dass die Methoden des gutartigen Roboters eingeschränkt genutzt werden können. Das Extension Interface “IKiller” kann den Roboter immer nur nach links drehen lassen oder etwas terminieren. In der Hoffnung das niemand dem Roboter zu nahe kommt, wird sich dieser immer nur im Kreis drehen.
Folgender Aufruf auf der Client Seite würde nicht funktionieren.

```
meetYOURmaker.moveForeward();
```

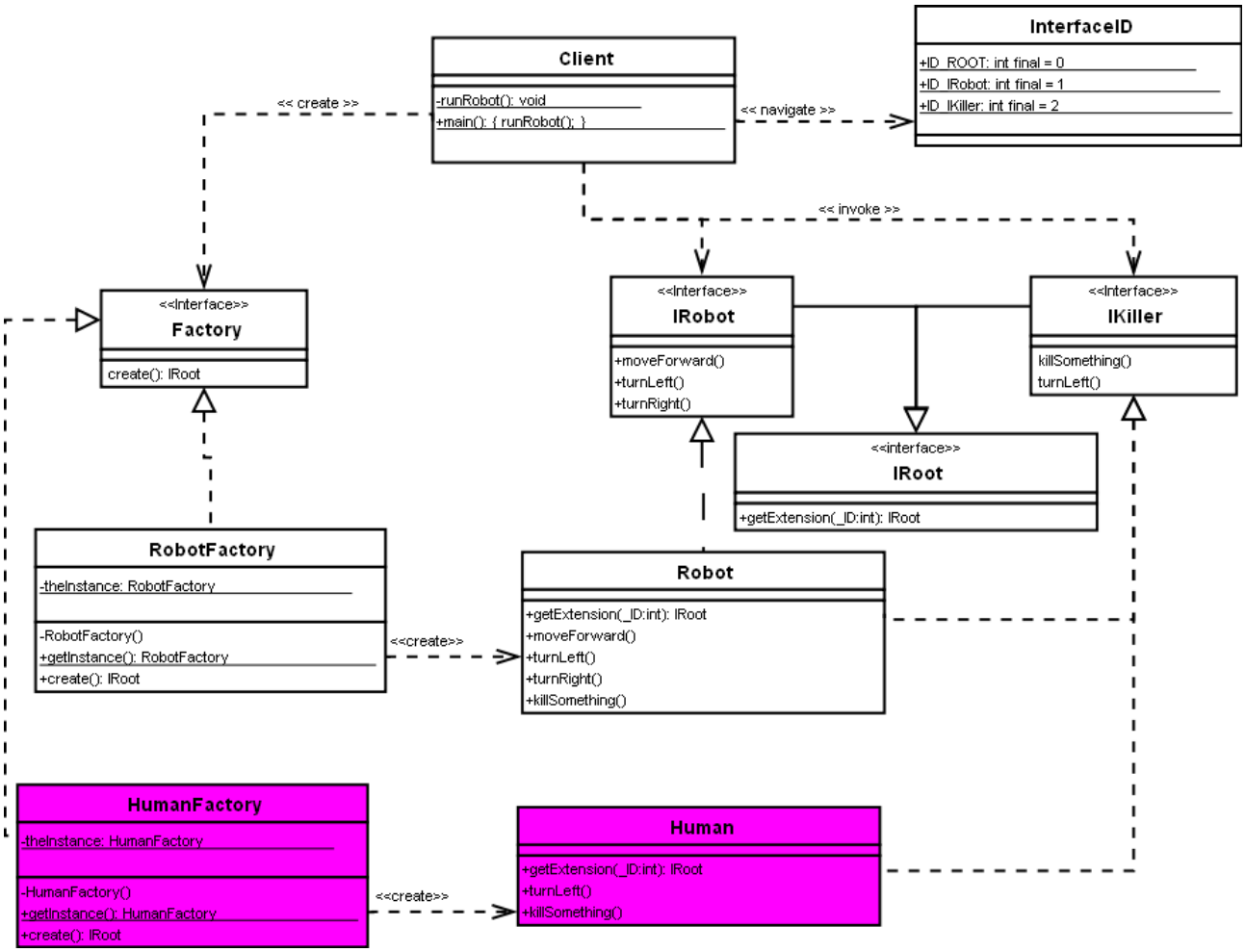
Beispiel 2. Hinzufügen einer Komponente

Die Fortschritt nimmt seinen Lauf und der nächste Schritt der Evolution, besteht darin echte Menschen mit vorherbestimmten Fähigkeiten herzustellen. Noch immer ist das dunkle Regime an der Macht und erhofft sich durch künstlich erschaffene Menschen, die Untergrundbewegung des Widerstandes zu infiltrieren.
Dafür wird die “Human” Komponente hinzugefügt und implementiert die Methoden aus dem “IKiller” Interface. Der Mensch soll sich nach links drehen und Dinge terminieren können.
Die Implementierung der Methoden kann von der Roboter-Komponente übernommen werden oder sie weicht ab. Das ist der Hauptvorteil der neuen Komponente, wir benutzen das gleiche Extension Interface zur Verhaltensdeklaration, ändern aber die Verhaltensdefinition in der Komponente.

10 Douglas C. Schmidt, Extension-Interface.doc, Seite 7.



Da die neue Komponente das gleiche “IKiller” Extension Interface benutzt, wie unsere Rolle des bösen Roboters, bedarf es keiner neuen InterfaceID Zuweisung. Hinzukommen die Komponente “Human” und die entsprechende Factory “HumanFactory”, welche die Komponente instanziert.



Der Widerstand hat es geschafft, die Implementierung der Komponente “Human” zu sabotieren. Der künstliche Mensch kaut jetzt lieber Kaugummi, als etwas zu zerstören.

Vergleichen wir dazu die zwei Komponenten:

```
public class Robot implements IRobot, IKiller{
...
// IRobot //
public void moveForward(){
          System.out.println("Robot.moveForward(): one step forward...");
}
public void turnLeft(){
          System.out.println("Robot.turnLeft(): one step left...");
}
public void turnRight(){
          System.out.println("Robot.turnRight(): one step right...");
}
// IKiller //
public void killSomething(){
          System.out.println("Robot.killSomething():destroy human mankind...");
}
}
public class Human implements IKiller{
...
// IKiller //
public void turnLeft(){
          System.out.println("Human.turnLeft(): one step left...");
}

public void killSomething(){
          System.out.println
          ("Human.killSomething():
                    no time for killing... I'm chewing bubblegum!");
          }
}
```

Das Beispiel demonstriert:

Die Implementierung der Methode “turnLeft” übernehmen wir 1:1 aus der Komponente ”Robot”, während wir für die Methode “killSomething” eine differenziertes Verhalten definieren.

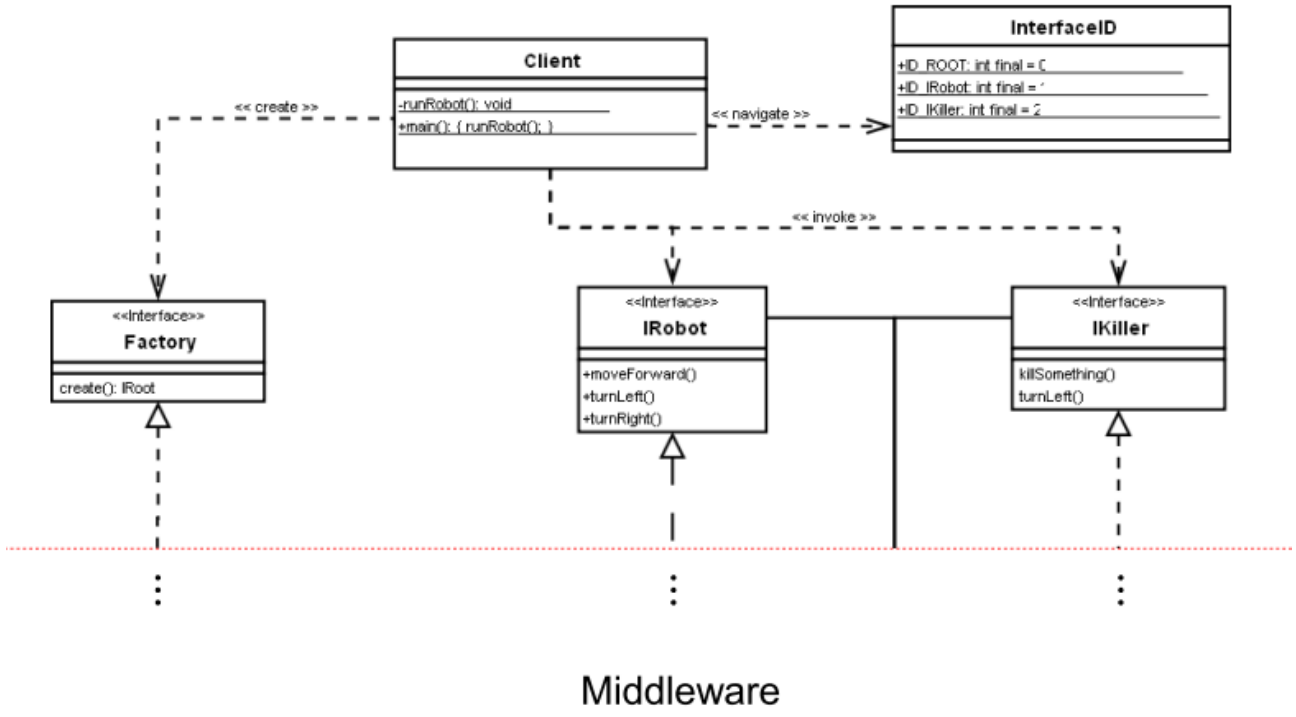
Die Methode “**getExtension**” muss von **jeder Komponente** implementiert werden, damit die Navigation zwischen den Rollen funktioniert. Diese gibt bei Aufruf die Referenz des angeforderten Extension Interface's zurück (siehe Bsp.1. Navigation).

```
...
// IRoot //
public IRoot getExtension(int _ID){

switch(_ID){

          case InterfaceID.ROOT_ID: return this;
          case InterfaceID.ROBOT_ID: return this;
          case InterfaceID.KILLER_ID: return this;

          default: return null;
          }
}
...
```

Neue Probleme & Lösungen

Factory Verwaltung

Indem neue Komponenten hinzugefügt werden, steigt die Anzahl der Komponenten selbst, als auch die dazugehörigen Factories. Dies stellt den Client vor das Problem: Welche Komponente implementiert das passende Extension Interface? Da die Factory die Komponente davor instanzieren muss, lautet die Frage: Welche ID passt zur entsprechenden Factory?[11]
Die Ursache dafür liegt in der gewollten Kapselung unserer Anwendung. Der Client greift ausschließlich auf Interface's zu, nämlich auf die Extension Interface's, um Dienste aufzurufen. Sowie auf das “Factory” Interface, um Instanzen von Komponenten zu erstellen.

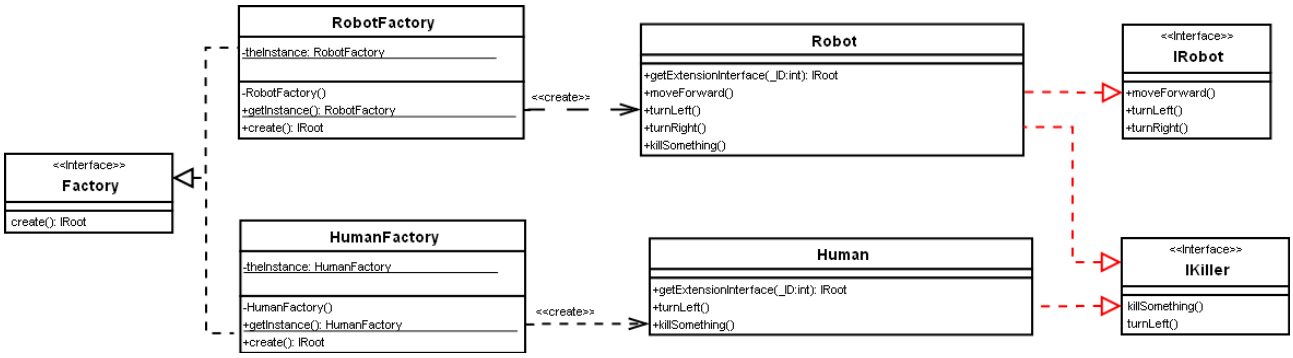


Die dem Client fehlende Information besteht aus der Assoziation zwischen Komponente und Extension Interface. Indem die Kompenente ein Extension Interface implementiert, setzt es diese Verbindung, die für den Client nicht ersichtlich ist. Die Lösung besteht darin, dem Client diese Assoziation erkenntlich zu machen, indem wir eine Tabelle einfügen. Jede ID repräsentiert genau ein Extension Interface, jede Komponente implementiert mindestens ein Extension Interface, eine Komponente wird von einer Factory gleichen Types erzeugt. Wir ordnen deswegen jeder ID mindestens eine Factory zu.

11 Douglas C. Schmidt, Extension-Interface.doc, Seite 17.

Interface_ID:	Factory:
IRobot: 1	robotFactory
IKiller: 2	robotFactory
IKiller: 2	humanFactory

Jeder Eintrag in der Tabelle, steht für eine Assoziation Extension Interface → Komponente, aus dem Klassendiagramm (Rote Pfeile).



Die Verwaltung dieser Tabelle übernimmt eine neu hinzukommende Klasse: **FactoryFinder**, welche eine HashTable zur Zuordnung verwendet. Dabei ist der Key, die InterfaceID und die Factoryreferenz der Value <Key,Value>.
Die zwei wichtigsten Methoden sind:

- findFactory(ID), welche zur übergebenen ID die passende Factory aus der Liste heraussucht.
- registerFactory(ID, factory), welche die Assoziation bei der Initialisierung des Systems übernimmt und für jede Factory ausgeführt werden muss.

Der FactoryFinder darf selbst nur einmal zur Laufzeit vorkommen und muss global zu Verfügung stehen. Er implementiert das Singleton Pattern. Eine weitere Möglichkeit mit einer Vielzahl an Factories umzugehen, wäre, die Erstellung und Zuordnung der Factories, einem Abstract Factory Pattern zu überlassen.[12] Der Client muss nicht alle Factories kennen, es werden nur diejenigen in die Tabelle aufgenommen/ registriert, welche auch für die Anwendungsfunktionalität des Clients benötigt werden.[13]

Initialisierung

In unserem Fall erstellen wir eine **ComponentInstaller** Klasse, die zu Beginn des Systems aufgerufen wird. Ihre Aufgaben sind:

12 Douglas C. Schmidt, Extension-Interface.doc, Seite 16.
13 Ebd., S. 18.

- Eine Instanz des FactoryFinder's zu erstellen.
- Benötigte Factories zu instanzieren.
- Jeder Factory mindestens eine ID, mit registerFactory(ID, factory), zuzuweisen.

Implementierung

Bei der Umsetzung dieses Pattern's in ein funktionsfähiges System sind im Vorfeld einige Punkte festzulegen, da es sich um fundamentale Fakten handelt, welche das Design maßgeblich bestimmen.

1. Domain Analyse

Begrenzt sich die Anwendung auf einen bzw. sehr wenige Namensräume kann das Pattern umgesetzt werden. Handelt es sich um eine verteilte Anwendung, ist ein bereits entwickeltes "Component Object Modell", wie COM+ oder CORBA, einer eigenen Entwicklung vorzuziehen.[14]

2. Verteilung der Funktionalität

Alle semantisch verwandten Methoden werden gruppiert und bekommen ein Extension Interface zugewiesen, welches wiederum über eine eindeutige ID identifiziert werden muss.
Alle Methoden, die sich keiner Gruppe zuordnen lassen, werden einem eigenen Extension Interface zugeordnet.
Allgemeine Funktionalität, wie die "getExtension" Methode, wird im RootInterface deklariert. Steht zum Zeitpunkt des Entwurfes noch nicht konkret fest, ob die gewünschte Funktion wirklich von allen Komponenten implementiert werden soll, z.B.: Eine Persistenzlösung, so wird ein eigenes Extension Interface definiert, welches zur Not von allen Komponenten implementiert werden kann.
Ein wichtiger Punkt, der in dieser Arbeit aus Übersichtsgründen ausgelassen wurde, ist die Fehlerbehandlung. Jede Komponente sollte eine selbst definierte Exception werfen, falls ein Interfacetyp angefordert wird, der nicht existiert.[15]

3. Verlinkung der Extension Interface's

Das in den zurückliegenden Kapiteln behandelte Beispiel realisiert die Verlinkung mit Schnittstellenvererbung und Komponentenimplementierung.
Programmiersprachen, die das Schnittstellenkonzept nicht unterstützen, können aber auch das Extension Interface als Innereklasse, in der Komponentenklasse, implementieren. Die Komponentenklasse instanziert dann genau ein Objekt pro Extension Interface. Die Extension Interface Klassen erben dann von der *abstrakten Klasse* "RootInterface".

14 Douglas C. Schmidt, Extension-Interface.doc, S. 8.
15 Ebd., Seite 9.

Für den Client besteht kaum ein Unterschied zwischen den beiden Ansätzen, da dieser immer auf ein Extension Interface zugreift (siehe Client).

4. Navigation

Damit die Rolle einer Komponente gewechselt werden kann, wird die "getExtension" Methode aufgerufen. Diese Methode wird von allen Komponenten implementiert. Dazu muss die entsprechende ID an die "getExtension" Methode übergeben werden. Dieser Zugriffsmechanismus muss nicht nur global verfügbar sein, sondern die Extension Interface's müssen auch eindeutig identifiziert werden können. Für die Navigation zwischen den Schnittstellen müssen folgende Regeln gelten:

reflexiv: aRa

ExtInt(A) → getExtension(A): ExtInt(A)

Das Extension Interface A ruft die "getExtension" Methode auf und übergibt die ID für sich selbst. Der Client muss dabei wieder das Extension Interface A bekommen.

symetrisch: aRb → bRa

ExtInt(A) → getExtension(B): ExtInt(B)
ExtInt(B) → getExtension(A): ExtInt(A)

Das Extension Interface A ruft die "getExtension" Methode auf und übergibt die ID für das Extension Interface B. Der Client greift auf Extension Interface B zu.
Das Extension Interface B ruft die "getExtension" Methode auf und übergibt die ID für das Extension Interface A. Der Client hat Zugriff auf Extension Interface A.

transitiv: aRb ^ bRc → aRc

ExtInt(A) → getExtension(B): ExtInt(B)
ExtInt(B) → getExtension(C): ExtInt(C)
==
ExtInt(A) → getExtension(C): ExtInt(C)

Das Extension Interface A ruft die "getExtension" Methode auf und übergibt die ID für das Extension Interface B. Der Client greift auf Extension Interface B zu.
Das Extension Interface B ruft die "getExtension" Methode auf und übergibt die ID für das Extension Interface C. Der Client hat Zugriff auf Extension Interface C.
Demnach sollte der direkte Weg von Extension Interface A nach Extension Interface C möglich sein.

5. Factory

Einen einheitlichen Umgang mit verschiedenen Factorytypen garantiert ein allgemeines Factory Interface. Dieses deklariert die Methode “create”, welche eine differenzierte Erzeugung über unterschiedliche “create” Methoden der Factories unterbindet. Zur Verwaltung der unterschiedlichen Factorytypen, kann an dieser Stelle auch das Abstract Factory Pattern angewendet werden.

6. Client

Bei der Implementierung des Clients sollten ähnliche Bedenken erfolgen, wie bei der Verteilung der Funktionalität. Die bestehende Komponentenfunktionalität sollte genutzt und nicht vom Client neu definiert werden. Komponenten können wieder aus weiteren Komponenten bestehen oder welche nutzen. Durch den Einsatz von vererbten Inteface's entsteht polymorphes Verhalten. In unserem Beispiel ist die Methode “killSomething” im “Roboter” anders implementiert, als in der “Human” Komponente.

Allerdings ist beim Anfordern des Extension Interface's ein Casting auf den Interfacetyp notwendig.

```
IRobot r1 = (IRobot) robot.getExtension(InterfaceID.ROBOT_ID);
```

Daraus entsteht eine schwache Bindung zwischen der Clientimplementierung und der Komponentenimplementierung. Wird im späteren Verlauf die Komponentenimplementierung auf die Verlinkung mit InnerClasses umgestellt, wird der Clientcode unbrauchbar![16]

Erweiterbarkeit

Anwendungsfall:	Vorgehen:
Hinzufügen eines Extension Interface's	1. Neue Extension Interface Schnittstelle definieren. 2. Neue Interface ID generieren und zuweisen. 3. Mindestens eine Komponente muss das neue Extension Interface implementieren. 4. Die Assoziation: ID → Factory muss der Initialisierung hinzugefügt werden.
Hinzufügen einer Komponente	1. Neue Komponente definieren. 2. Neue Factory zuweisen. 3. Neue Komponente muss mindestens ein neues/ bestehendes Extension Interface implementieren. 4. Die Assoziation: ID → Factory muss der Initialisierung

16 Douglas C. Schmidt, Extension-Interface.doc, Seite 21.

	hinzugefügt werden.
Modifikation einer Komponente	Da die Wiederverwendbarkeit des Clientcodes eine Grundvoraussetzung darstellt, wird jede Art von Modifikation an bestehender Implementierung vermieden. Eine Veränderung stellt damit einen neuen Service dar, der in einem neuen Extension Interface definiert wird.[17]

Vorteile

- Durch das Gruppieren semantisch verwandter Funktionalität haben wir Abhängigkeiten reduziert.
- Mit dem Einsatz von Extension Interface's lassen wir den Client nur über spezifische Schnittstellen auf die Komponente zugreifen. Das Erstellen der Komponenten wird ebenfalls über ein dafür vorgesehenes Interface realisiert. Damit haben wir die Bindung zwischen Client und Komponentenimplementierung auf ein Minimum reduziert und Kohäsion sichergestellt.
- Durch den modularen Aufbau ist diese Architektur sowohl gegen funkionale Addition, wie auch Modifikation resistent, der Clientcode wird niemals gebrochen.
- Wir haben Polymorphie ohne den Einsatz von Subklassen ermöglicht und bleiben dadurch für weitere Veränderungen höchst flexibel.

Nachteile

- Es ergibt sich eine schlechtere Laufzeiteffizienz, da niemals direkt auf die Implementierung zugegriffen wird.
- Das Muster erhöht die Komplexität der Entwicklung, Implementierung und Integration. In einer OOP-Umgebung folgt die Umsetzung dem hier beschriebenen Schema. Sind aber bereits bestehende Kompenenten in nicht OOP-Sprachen umgesetzt, kann von keiner trivialen Aufgabe mehr die Rede sein.
- Ein Wechsel der Verlinkungsstrategie der Extension Interface's bricht den Clientcode und ist unter allen Umständen zu vermeiden.

Variante: Distributed Extension Interface

Das Distributed Extension Interface behandelt, wie der Name schon sagt, die Problematik verteilter Anwendungen. In verteilten Systemen teilen sich Client und Server unterschiedliche Adressräume.

17 Douglas C. Schmidt, Extension-Interface.doc, Seite 8.

Es muss also ein Dienst bereitgestellt werden, welcher verfügbare Kompenenten registriert/un-registriert. Die physikalische Trennung zwischen Schnittstellen und Implementierung wird mit Hilfe von Proxies realisiert.[18] Handelt es sich um eine heterogene Laufzeitumgebung, in der Client und Server in unterschiedlichen Programmiersprachen umgesetzt sind, wird ein gemeinsamer Nenner zur einheitlichen Verständigung benötigt. An dieser Stelle kommt CORBA ins Spiel.
Teil der Common Object Request Broker Architecture ist die Interface Definition Language, kurz IDL. Damit erstellt der Entwickler eine formale Spezifikation der Schnittstelle, welche der IDL-Compiler in die Zielarchitektur überführt.[19] Es stellt sich die Frage, wer das Erstellen der Proxies und das Übersetzen in die IDL verwaltet. Diese Aufgabenstellung lässt sich am besten mit dem Broker Patttern lösen, bei dem der Broker die Kommunikation in einer verteilten heterogenen Client – Server Anwendung übernimmt.

Verwendung

COM / COM+

Ist ein binärerer Interfacestandard zur Interprozesskommunikation und dynamischer Objekterzeugung für einen großen Bereich der Programmiersprachen.[20] Jeder COM Komponente wird ein eigenes Factory Interface bereitgestellt, das für die Instanzierung verantwortlich ist. Jede COM Klasse implementiert ein oder mehrere (Extension) Interface's, welche von der Basis (Root) Schnittstelle erben.[21] COM/COM+ implementieren die Distributed Extension Interface Variante, sind aber mittlerweile als “deprecated” klassifiziert. **.**NET ist in der Lage weiterhin COM/COM+ über Wrapper zu benutzen.

Opendoc

Diese Softwarearchitektur wurde für die Erschaffung komponentenorientierter und kooperativer Programme entwickelt. Der Zusammenschluss vieler namhafter Unternehmen wie Apple Computer, IBM, Novell, Adobe, Xerox, sollte einen Ansatz schaffen, der Verbunddokumente unterstützt und den Fokus auf das Dokument und nicht auf die Anwendung legt, in der es erstellt wird. Die benutzerspezifische Anpassbarkeit, Erweiterbarkeit sowie die Plattformunabhängigkeit waren weitere Ziele.[22]
Unter einem Verbunddokument ist eine Komposition aus unterschiedlichen Formaten (Text, Bild, usw...) in einem Dokument zu vertstehen. Eine HTML Datei wäre ein Beispiel für solch ein

18 Douglas C. Schmidt, Extension-Interface.doc, Seite 22.
19 Wikipedia, Common Object Request Broker Architecture
20 Wikipedia, Component Object Model
21 Douglas C. Schmidt, Extension-Interface.doc, Seite 24.
22 Wikipedia, OpenDoc

Dokument. Soll eine Komponente bearbeitet werden, geschieht dies in dem Editor, in welchem das Dateiformat erstellt wurde.

OLE

Die Object Linking and Embedding Technologie stellt dabei eine von Microsoft entwickelte Lösung dar und steht unter Windows als Betriebssystemkomponente allen Applikationen zur Verfügung. Damit wird das Verschieben einer in Excel erstellten Tabelle in die Word Anwendungen ermöglicht. Die Steuerung erfolgt dabei über die bereits in COM erwähnten Interface's. Dieses Verhalten kann sowohl bei der Microsoft Office Produktfamilie, als auch von OpenOffice betrachtet werden. [23]

Epilog

Das Extension Interface Design Pattern ist **eine** Möglichkeit der Veränderung strukturiert entgegenzuwirken. Neben seiner hohen Flexibilität folgt gleich der Nachteil der aufwendigen Umsetzung. Diesen Preis wird der Entwickler immer mehr oder weniger zahlen müssen, je nachdem, wie hoch der Grad dieser Flexibilität sein soll. Deshalb ist immer vor Projektbeginn abzuschätzen, wie oft das Programm verändert werden soll und vor allem, wie hoch dabei der entstehende Aufwand ist. Stehen beide Punkte nicht zur Debatte an, kann dieses Pattern als “good to know” aufgefasst werden. Andernfalls sollte der Designer anfangen, sich Gedanken über die Rahmenbedingungen, Kapitel: **Implementierung**, zu machen.

23 Wikipedia, Object Linking and Embedding

Literaturverzeichnis

Literaturquellen

Eric & Elisabeth Freeman - Kathy Sierra & Bert Bates, Head First Design Patterns

Johannes Siedersleben, Moderne Softwarearchitektur

Internetquellen

Douglas C. Schmidt, Extension-Interface.doc,
http://www.laputan.org/pub/sag/extension-interface.pdf

Wikipedia, Component Object Model,
http://en.wikipedia.org/wiki/COM%2B

Wikipedia, Common Object Request Broker Architecture,
http://de.wikipedia.org/wiki/CORBA

Wikipedia, Lehman's laws of software evolution,
http://en.wikipedia.org/wiki/Lehman%27s_laws_of_software_evolution

Wikipedia, Object Linking and Embedding,
http://de.wikipedia.org/wiki/Object_Linking_and_Embedding

Wikipedia, OpenDoc,
http://de.wikipedia.org/wiki/OpenDoc