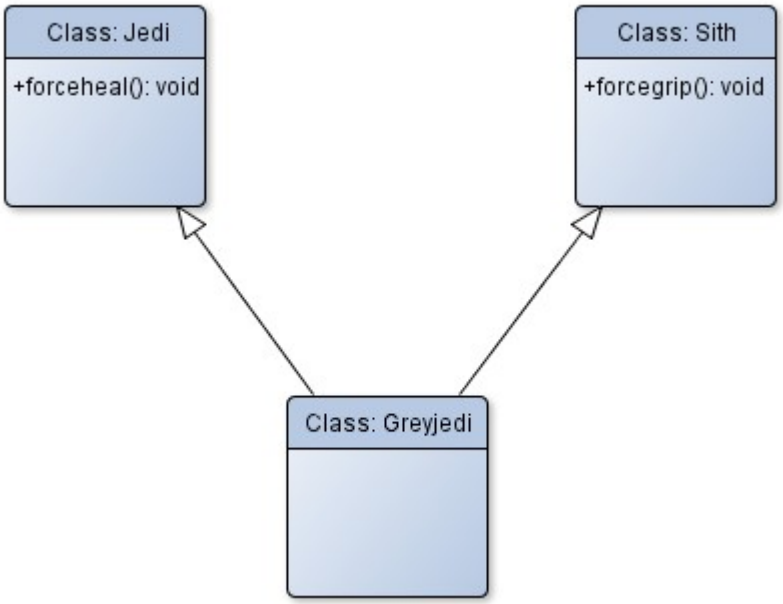
1. Prinzip und Anwendung der Mehrfachvererbung

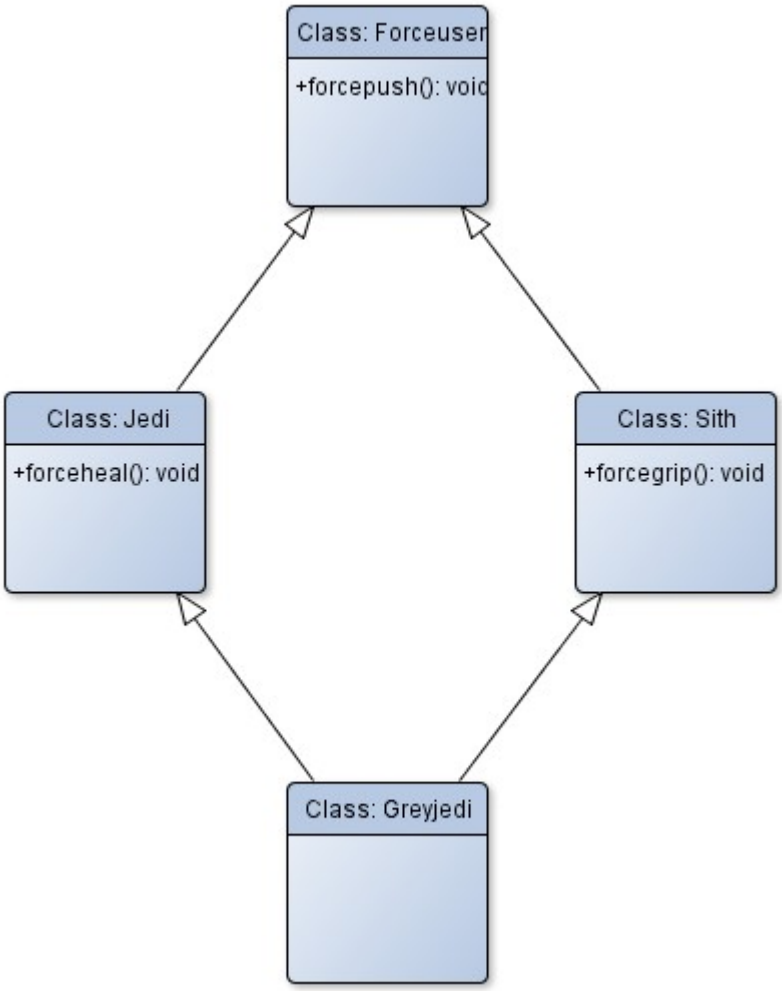


Die Klasse ***Greyjedi*** erbt von der Klasse ***Sith*** und ***Jedi***, somit kann diese Klasse sowohl die Methoden ***forcegrip()*** und ***forceheal()*** benutzen.

Eine sinnvolle Anwendung von Mehrfachvererbung ist abhängig von der benutzten Programmiersprache und der vorliegenden Architektur.

z.B Sinnvoll wäre es von der Klasse ***Customer*** und ***FrontendUser*** zu erben um die Klasse ***TypoGentoUser*** zu erstellen.

2. Problematiken die dabei auftreten können



Von der Klasse ***Forceuser*** erben die Klassen ***Sith*** und ***Jedi***, von denen wieder die Klasse ***Greyjedi*** erbt.

Nun ensteht ein Problem da beide Klassen die Methode ***forcepush()*** bereitstellen…

Das Diamant-Problem ist geboren.

3. Ursprünglicher Lösungansatz via Interfaces

```
interface Forceuser {
        public function forcepush();
}

interface Sith extends Forceuser{
        public function forcegrip();
}

interface Jedi extends Forceuser{
        public function forceheal();
}
class GreyJedi implement Jedi, Sith {
        public function forceheal() {
        }
        public function forcepush() {
        }
        public function forcegrip() {
        }
}
```

In einem Interface wird eine Methode samt ihren Argumenten deklariert,die eigentliche Logik wird weggelassen.

Sobald eine Klasse nun das Interface implementiert, muss die Klasse gleichzeitig auch die Methoden bereitstellen.

4. Neuer Ansatz mit Traits ab PHP 5.4

In PHP 5.4 gibt es ein neues Konzept namens Traits, welches im Prinzip eine lose Klasse bzw. ein Fragment einer Klasse repräsentiert.
Dabei kann, im Gegensatz zu Interfaces, die Logik schon gegeben sein.

```
trait  Forceuser {
        public function forcepush()    {
        }
}

trait Sith {
        public function forcegrip()   {
        }
}

trait Jedi {
        public function forceheal()   {
        }
}
class GreyJedi {
        Use Sith, Jedi;
}
```

5. Traits in Kombination mit Interfaces

```
interface ForceuserInterface {
	public function forcepush();
}

interface SithInterface extends ForceuserInterface {
	public function forcegrip();
}

interface JediInterface extends ForceuserInterface {
	public function forceheal();
}

trait Forceuser {
	public function forcepush() {
		echo __METHOD__;
	}
}
trait Sith {
	public function forcegrip() {
		echo __METHOD__;
	}
}
trait Jedi {
	public function forceheal()  {
		echo __METHOD__;
	}
}
class GreyJedi implements JediInterface, SithInterface {
	use Sith, Jedi, Forceuser;
}

(new GreyJedi)->forcepush();
```

6. Konfliktauflösung bei Traits

   Falls zwei Traits eine Methode mit gleichen Namen einfügen, so wird ein Fatal Error geworfen. Um dies zu umgehen kann Methoden eines Traits ausschließen.

```
trait Jedi {
        public function forcepush() {
        }
        public function forcepull() {
        }
}

trait Sith {
        public function forcepush() {
        }
        public function forcepull() {
        }
}

class GreyJedi
{
        use Jedi, Sith {
                Sith::forcepush insteadof Jedi;
                Jedi::forcepull insteadof Sith;
        }
}
```

Oder man kann einen Alias vergeben:

```
class GreyJedi
{
        use Jedi, Sith {
                Jedi::forcepush insteadof Sith;
                Sith::forcepull insteadof Jedi;
                Sith::forcepush as evilpush;
        }
}
```