GRASP

General Responsibility Assignment Software Patterns

Maurice Müller

Allgemein

- GRASP umfasst Prinzipien / Muster zur Zuständigkeit
  - wer sollte für was zuständig sein
- zusammengefasst von *Craig Larman*

Information Expert

oder *Expert* oder *Expert Principle*

dt: Informationsexperte oder Experte

Definition

- für eine neue Aufgabe ist derjenige zuständig, der schon das meiste Wissen für die Aufgabe mitbringt
- läuft in vielen Fällen auf "Do It Myself" hinaus

Beispiel (von Wikipedia):
gegeben ist die Klasse Kreis und es soll die Fläche berechnet werden

- (+): Kreis enthält schon den Radius und berechnet deshalb die Fläche selbst
- (-): eine Hilfsklasse, die geometrische Formen entgegen nimmt und die Fläche berechnet

Creator

dt.: Erzeuger oder Erzeuger-Prinzip

Definition

Das Erzeuger-Prinzip gibt vor, wer für die Erzeugung einer Instanz zuständig ist.

Eine Klasse A 'darf' eine Instanz von Klasse B erzeugen, wenn:

- A eine Aggregation von B ist oder Objekte von B enthält
- A Objekte von B verarbeitet
- A von B abhängt (starke Kopplung)
- A der Informationsexperte für die Erzeugung von B ist
  - z.B. hält A die Intialisierungsdaten oder ist eine Factory

Controller

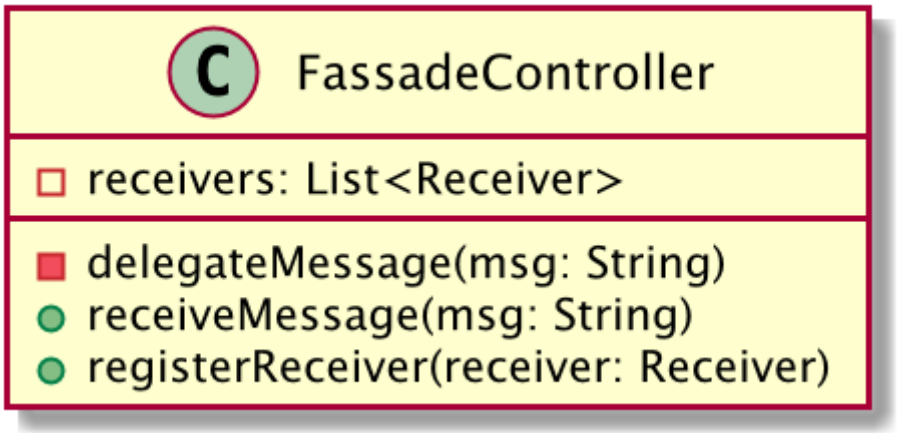
dt.: Steuereinheit

Definition

> Der Controller (Steuereinheit) beinhaltet das Domänenwissen und definiert, wer die für eine Nicht-Benutzeroberflächen-Klasse bestimmten Systemereignisse verarbeitet.
>
> — Wikipedia

- erste Schnittstelle nach der GUI
- macht wenig selbst
  - delegiert an andere Module
- Use Case Controller
  - verarbeitet alle Events eines spezifischen Use Cases
  - kann mehr als einen Mini Use Case beinhalten
    - z.B. Benutzer erzeugen *und* löschen
- Fassade Controller
  - hauptsächlich in Messaging-Systemen
  - hängt zwischen allen Nachrichten (da es normal auch nur einen Eintrittspunkt gibt)

Beispiel: Controller



Low Coupling

dt.: geringe Kopplung

Definition

> Der Begriff Kopplung bezeichnet den Grad der Abhängigkeiten zwischen zwei oder mehr 'Dingen'.
>
> — Gernot Starke in Effektive Software-Architekturen

> In der Informatik versteht man unter dem Begriff Kopplung die Verknüpfung von verschiedenen Systemen, Anwendungen, oder Softwaremodulen, sowie ein Maß, das die Stärke dieser Verknüpfung bzw. der daraus resultierenden Abhängigkeit beschreibt.
>
> — Wikipedia

Vorteile Low Coupling

- leichte(re) Anpassbarkeit
- bessere Testbarkeit
- erhöhte Wiederverwendbarkeit

Beispiel: Low Coupling

```java
public class Car {

    enum Type {MERCEDES, BMW, VW, AUDI}
    private Type type;

    private Engine engine =
        new Engine(this);

    public Car(Type type) {
        this.type = type;
    }

    public void drive() {
        engine.start();
    }

    public void doService() {
        engine.checkFanBelt();
        engine.checkOil();
    }

    Type type() {
        return type;
    }
}
```

```java
public class Engine {

    public Engine(Car car) {
        switch (car.type()) {
            case MERCEDES:
                setupMercedesEngine();
                break;
            default:
                setupDefaultEngine();
        }
    }

    private void setupDefaultEngine() {}

    private void setupMercedesEngine()
{}

    void checkOil() {}

    void checkFanBelt() {}

    public void start() {}
}
```

NOTE

- *Car* und *Engine* hängen voneinander ab
- insbesondere *doService()* ist kritisch
  - Was passiert, wenn bei Engine *checkCoolingFluid()* dazu kommt? -> *Car* bekommt das erstmal nicht mit -> *doService()* würde nicht mehr alles prüfen
- Wie kann ein Audi einen Motor von VW bekommen (beide gehören zur Volkswagen Gruppe)?
  - da *Engine* von *Car* abhängt, ist dies so nicht möglich
- Lösung?
  - Dependency Injection (eine *Engine* reingeben)
  - Abstraktionen einführen (-> Dependency Inversion Principle)

Lösung

```
public class Car {

    private Engine engine;
    private Manufacturer manufacturer;

    Car(Manufacturer manufacturer,
Engine engine){
        this.manufacturer =
manufacturer;
        this.engine = engine;
    }

    void drive() {
        engine.start();
    }

    void doService() {
        engine.doService();
    }
}
```

```
public enum Manufacturer {
    MERCEDES, VW, AUDI, BMW;
}
```

```
public interface Engine {
    void doService();
    void start();
}
```

```
public class SimpleEngine implements
Engine {

    SimpleEngine(Manufacturer
manufacturer) {
        // set up engine based on
manufacturer
    }

    @Override
    public void doService() {
        checkOil();
        checkFanBelt();
    }

    private void checkOil() {}

    private void checkFanBelt() {}

    @Override
    public void start() {}
}
```

NOTE	*Car* hängt jetzt von einem Interface ab → Interfaces werden nur selten geändert und sind in der Regel durchdachter. Ändert sich nun die konkrete Implementierung von *Engine* hat das erstmal keine Auswirkung auf *Car*.

Exkurs: Versteckte Kopplung

- temporäre Kopplung sollte explizit sein
  - temporäre Kopplung = zeitlich abhängige Kopplung

```
class SelfDrivingCar {
    private Route route;

    void driveTo(Destination destination) {
        calculateRoute(destination);
        startDriving();
    }

    private void startDriving() {
        //using this.route to navigate to destination
    }

    private void calculateRoute(Destination destination) {
        this.route = Route.to(destination);
    }
}
```

- *startDriving()* hängt von *calculateRoute()* ab, kann aber unabhängig davon aufgerufen werden → schlecht

Lösung: Versteckte Kopplung

```
class SelfDrivingCar {

    void driveTo(Destination destination) {
        Route route = calculateRoute(destination);
        startDriving(route);
    }

    private void startDriving(Route route) {
        //using this.route to navigate to destination
    }

    private Route calculateRoute(Destination destination) {
        return Route.to(destination);
    }
}
```

- *startDriving()* benötigt nun explizit eine *Route* und kann nun nicht mehr "einfach so" aufgerufen werden

High Cohesion

dt.: hohe Kohäsion

Definition

> Kohäsion, zu Deutsch 'Zusammenhangskraft', ist ein Maß für den inneren Zusammenhalt von Elementen (Bausteinen, Funktionen). Sie zeigt, wie eng die Verantwortlichkeiten eines Bausteins inhaltlich zusammengehören.
>
> — Gernot Starke in Effektive Software Architekturen

> When Cohesion is high, it means that methods and variables of the class are co-dependent and hang together as a logical whole.
>
> — Robert C. Martin in Clean Code

Vorteile: High Cohesion

- übersichtlicherer und strukturierterer Code
  - Code ist dort, wo man ihn erwartet
- unterstützt *Low Coupling*

Beispiel: High Cohesion

```
public class Car {

    private boolean keyInserted;
    private Engine engine;

    public void drive() {
        if(keyInserted) {
            engine.start();
        }
    }
}
```

NOTE	Beide Instanzvariablen von *Car* werden in allen (in diesem Fall einer) Methode genutzt → maximale Kohäsion. **Aber:** maximale Kohäsion ist in der Regel kontraproduktiv.

Polymorphism

dt.: Polymorphie (aus dem Griechischen → Vielgestaltigkeit)

Definition

- unterschiedliches Verhalten eines Typs soll durch Polymorphie ausgedrückt werden
  - d.h., die Verantwortlichkeit wird einer eigenen Ausprägung zugewiesen

Vorteile:

- vermeidet *Switch*-Statements
  - objektorientierte Lösung
- vermeidet Fehler, wenn neue Typen hinzukommen

Beispiel: Polymorphie

```
public enum Manufacturer {
    BMW, AUDI, MERCEDES, VW;
}
```

```
public class Car {
    private Manufacturer type;

    public Car(Manufacturer type) {
        this.type = type;
    }

    public Manufacturer type() {
        return this.type;
    }
}
```

- abhängig des Herstellertyps soll nun ein Preis berechnet werden

Klassisches Switch (schlecht)

```
public class CarOrder {

    public double calculatePrice(Car car) {
        switch (car.type()) {
            case BMW:
                return 77777.99;
            case AUDI:
                return 66666.99;
            case MERCEDES:
                return 99999.99;
            case VW:
                return 88888.99;
            default:
                //<this will never happen>
                return 0.99;
        }
    }
}
```

- *Switch*es können nur größer werden
- *default*: wird er wirklich nie ausgelöst?
    - wenn ein Tesla mit ins Programm aufgenommen würde, würde er für 0.99$ verkauft werden

mit Polymorphie

```
public abstract class Car {
    private Manufacturer type;

    public Car(Manufacturer type) {
        this.type = type;
    }

    public Manufacturer type() {
        return this.type;
    }

    public abstract double price();
}
```

```
public class CarOrder {
    public double calculatePrice(Car car) {
        return car.price();
    }
}
```

- Preis vergessen zu berechnen ist nicht mehr (so einfach) möglich
- Sondermodelle sind jetzt möglich

Pure Fabrication

dt.: reine Erfindung

Definition

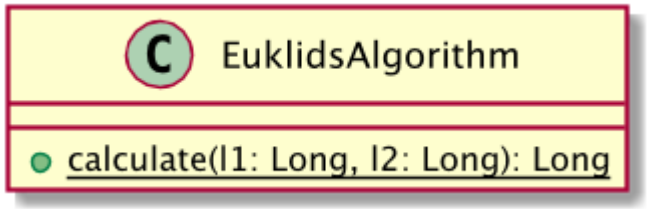
> Eine Pure Fabrication (reine Erfindung), stellt eine Klasse dar, die so nicht in der Problem Domain existiert. Sie stellt eine Methode zur Verfügung, für die sie nicht Experte ist.
>
> — Wikipedia

- häufig als Hilfsklassen vorzufinden
  - sollten nicht überwiegen, da sie dazu tendieren, *nicht* objektorientiert zu sein

Vorteile:

- trennt Technologiewissen von Domänenwissen

Beispiel: Pure Fabrication



Indirection / Delegation

dt.: Indirektion / Delegation

Definition

> Delegation is a way to make composition as powerful for reuse as inheritance [Lie86, JZ91]. In delegation, two objects are involved in handling a request: a receiving object delegates operations to its delegate. This is analogous to subclasses deferring requests to parent classes.
>
> — aus 'Design Patterns' von Erich Gamma, et al.

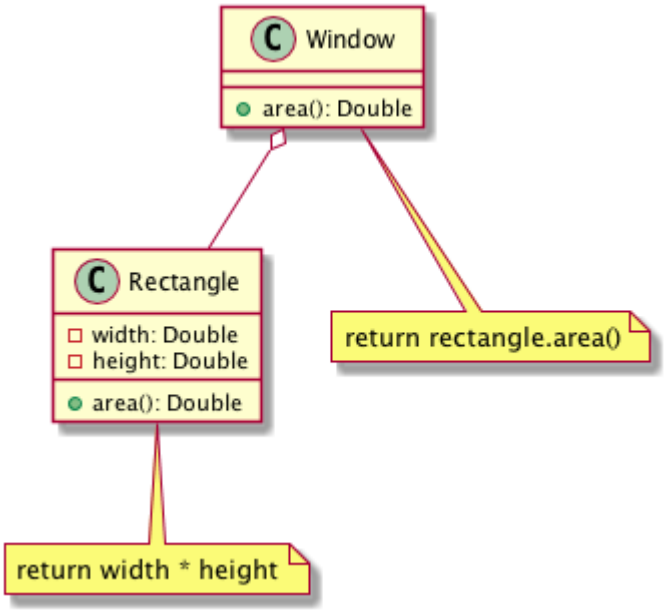
- zwei Einheiten kommunizieren über einen Vermittler, anstatt direkt miteinander
- kann Vererbung ersetzen

Vorteile: Indirection

- geringere Kopplung
- flexibler als Vererbung
  - aber benötigt mehr Code

Beispiel: Indirektion

Quelle: 'Design Patterns' von Erich Gamma, et al.



Protected Variations

dt.: geschützte Veränderungen

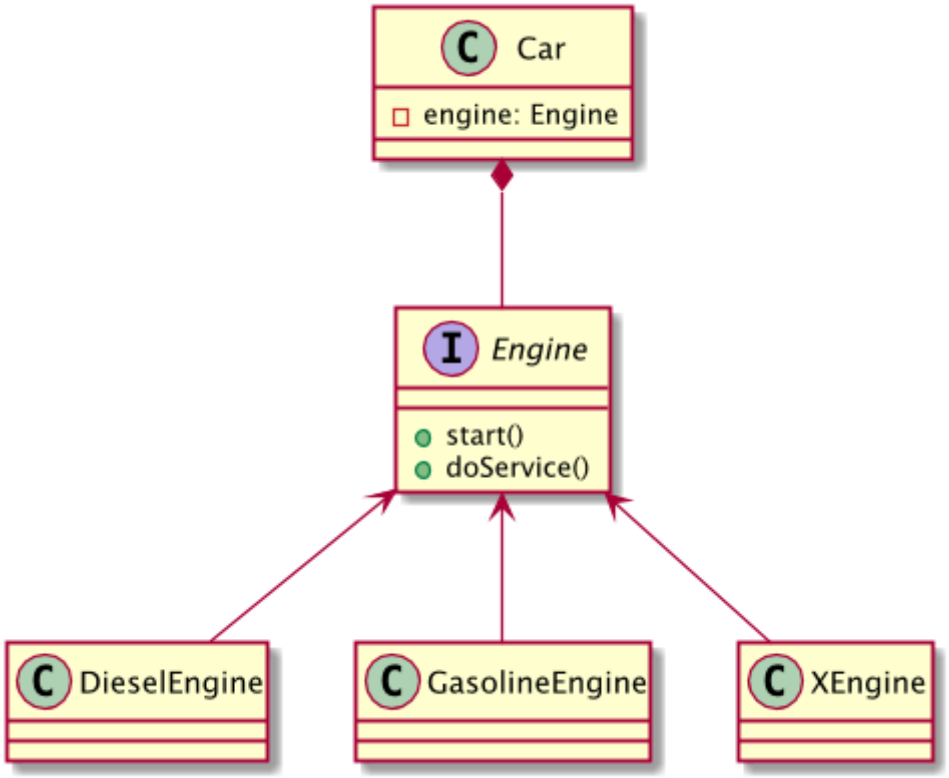
Definition

> Interfaces sollen immer verschiedene konkrete Implementierung verstecken. Man nutzt also Polymorphismus und Delegation, um zwischen den Implementierungen zu wechseln. Dadurch kann das restliche System vor den Auswirkungen eines Wechsels der Implementierung geschützt werden.
>
> — Wikipedia

Vorteile:

- System ist geschützt vor den Auswirkungen eines Wechsels

Beispiel: Protected Variations



- *Car* kann nun eine andere *Engine* bekommen, ohne, dass es zu ungewollten Nebenwirkungen kommt