GRASP

Maurice Müller

2023-11-20

Allgemein

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

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
- bessere Lesbarkeit, da weniger Kontext

Beispiele angeordnet von **starker Kopplung** zu **loser Kopplung**:

- Code in gleicher Methode *(starke Kopplung)*
- Statischer Methodenaufruf
- Polymorpher Methodenaufruf
- Polymorpher Aufruf an Interface
  - z.B. beim Listener-Pattern
- Versand eines Events auf Eventbus *(lose Kopplung)*
  - Sender und Empfänger kennen sich nicht mehr

Beispiel: Low Coupling

```java
public class Car {

    enum Type {MERCEDES, BMW, VW, AU
    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() {}
}
```

Lösung

```java
public class Car {

    private Engine engine;
    private Manufacturer manufacturer;

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

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

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

}
```

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

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

```java
public class SimpleEngine impl

    SimpleEngine(Manufacturer
        // set up engine based
    }

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

    private void checkOil() {}

    private void checkFanBelt(

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

Andere Kopplungsarten

- Kopplung an konkrete Klassen
  - Klassen ←→ Interfaces
- Kopplung durch Threads
  - bei gemeinsamen Locks
- Kopplung durch Ressourcen
  - z.B. Speicher oder CPU

Exkurs: Temporäre 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) {
        // skip implementation
    }

    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*

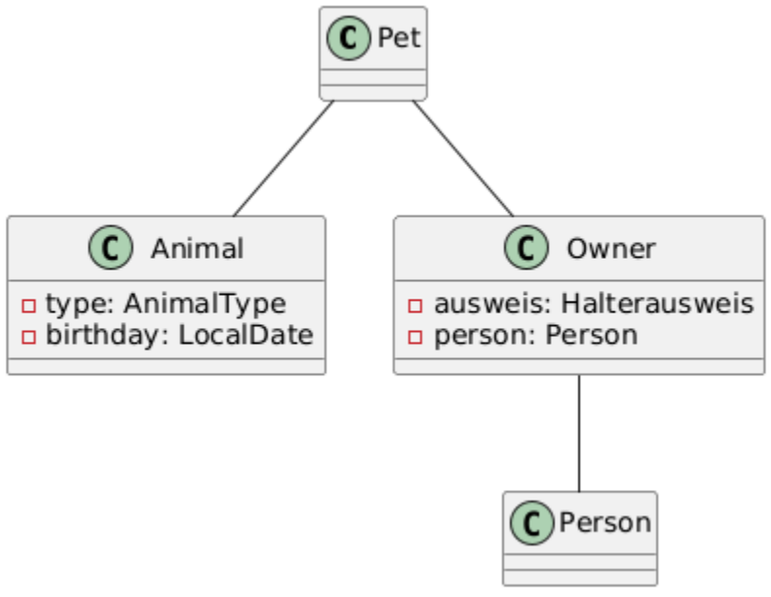
Beispiel: High Cohesion

Niedrige Kohäsion

```
class Animal {
   private String name;
   private Type type;
   private LocalDate birthday;
   private Person owner;

   public void rename(String newName) {
      name = newName;
   }

   public void changeOwner(Person newOwner) {
      owner = newOwner;
   }

   public String identity() {
      return name + " born at" + birthday.format(DateTimeFormatter.ISO_LOCA
   }
}
```

Kohäsion erhöhen

- Klasse Animal mit *type* und *birthday*
- Klasse Owner mit *Person* und ggf. weiteren Daten
  - z.B. Halterausweis
- Klasse Pet
  - verbindet *Owner* mit *Animal*



Hohe Kohäsion

```
public class Car {

    private boolean keyInserted;
    private Engine engine;

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

Kohäsionsmetriken

- Kohäsion ist ein semantisches Maß
  - menschliche Einschätzung entscheidend
- technische Metriken können Kohäsion (begrenzt) bestimmten
  - z.B. ein unbenutztes Feld
  - kann leicht falsch liegen
- Heuristiken helfen bei der Analyse
  - z.B. tendiert kohäsiver Code zur Kürze

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: Es soll die Grundfläche eines Kreises berechnet werden (die Klasse Kreis existiert bereits).

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

Vorteile

- zusammengehörige Funktionalität sammelt sich an einem Ort (*high cohesion*)
- Internas müssen nicht nach außen gegeben werden (Geheimnisprinzip)
- es werden keine *Hilfsklassen* benötigt, die übersehen werden können
- Wiederverwendung wird wahrscheinlicher

Polymorphism

dt.: Polymorphie (aus dem Griechischen → Vielgestaltigkeit)

Idee

- Polymorphie (OO) = Methode wird bei Vererbung (neu) implementiert
- unterschiedliches Verhalten eines Typs soll durch Polymorphie ausgedrückt werden
  - d.h., die Verantwortlichkeit wird einer eigenen Ausprägung zugewiesen

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();
}
```

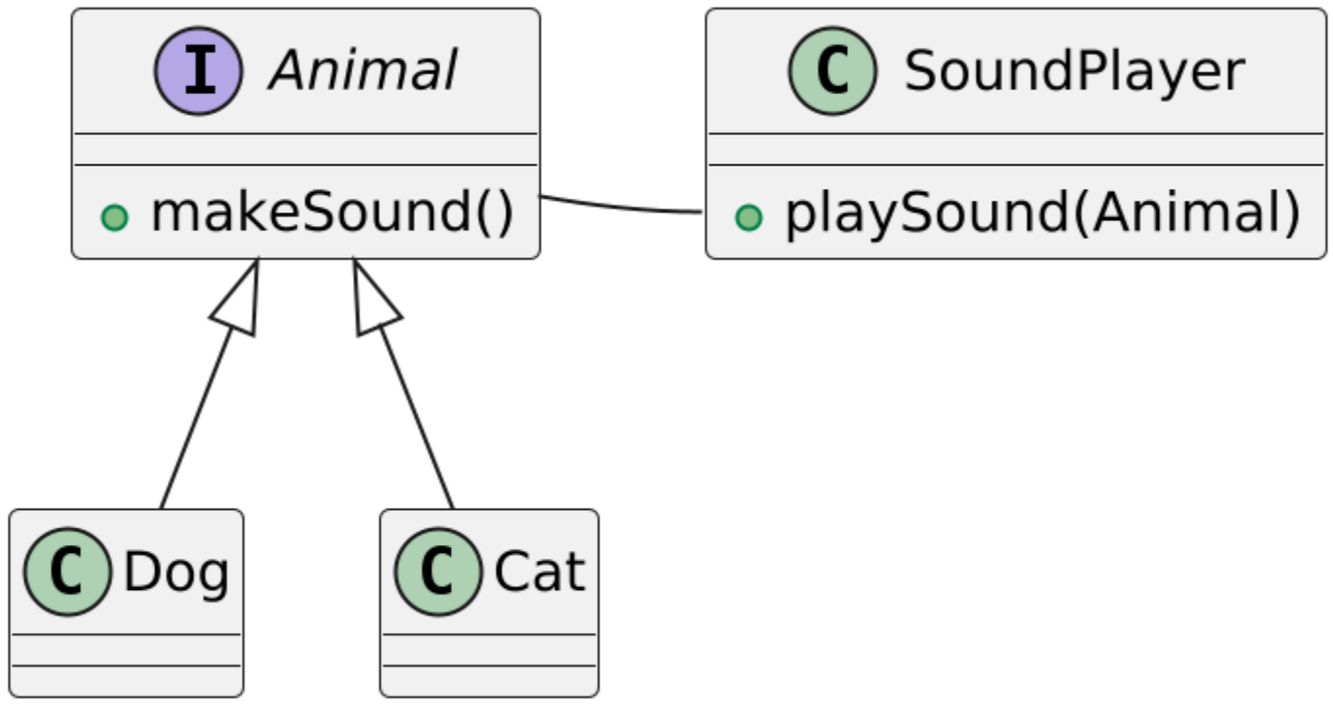
```
public class CarOrder {
    public double calculatePrice(Car car) {
        return car.price();
    }
}
```

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

Exkurs: Arten von Polymorphie

Polymorphie durch Vererbung

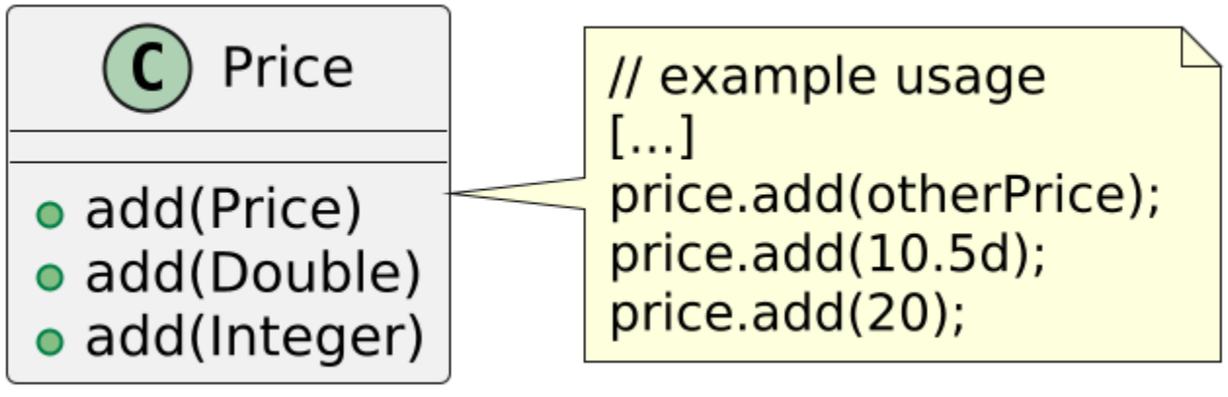
- Methode wird bei Vererbung (neu) implementiert
- gilt für Interfaces und Klassen
- zur Laufzeit entscheidet sich das konkrete Verhalten



Exkurs: Arten von Polymorphie

Polymorphie durch Überladung

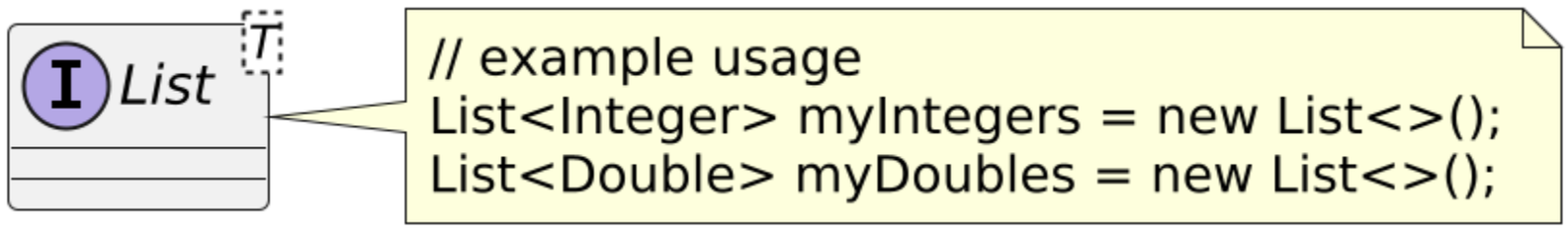
- Methoden/Funktionen und Operatoren (z.B. *+*, -, …) werden überladen
- abhängig der übergebenen Parameter wird die entsprechende Methode ausgewählt
- zur Compile-Zeit entscheidet sich das Verhalten



Exkurs: Arten von Polymorphie

Polymorphie durch *Generics*

- bei der Instanziierung oder Deklaration wird ein Datentyp als Parameter mitgeben
- Verhalten zur Compile-Zeit oder Laufzeit ist abhängig von der Programmiersprache
  - *C#*: Prüfung zur Compile-Zeit, Erzeugung von dynamischen Typen zur Laufzeit
  - *Java*: Prüfung zur Compile-Zeit, Laufzeit hat keine bzw. kaum Informationen zu den Generics (type erasure)



Polymorphie: Vorteile

- Code ist dort, wo man in erwartet / wo es sinnvoll ist
  - unterstützt *High Cohesion* und *Information Expert*
- vermeidet *Switch*-Statements
  - objektorientierte Lösung
- vermeidet Fehler, wenn neue Typen hinzukommen

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*

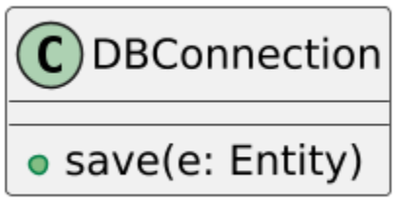
- häufig als Hilfsklassen vorzufinden
  - sollten nicht überwiegen, da sie dazu tendieren, *nicht* objektorientiert zu sein
- trennt Technologiewissen von Domänenwissen

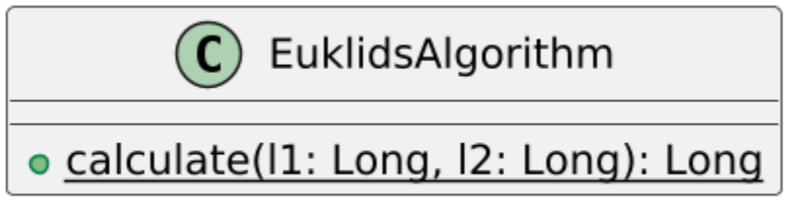
Beispiel: Pure Fabrication

```
boolean verifyAge(Person person) {
   if(person.age >= 18) {
      if(person.city == City.Karlsruhe) {
         _kaLogger.info(person.name + " was successfully verified.");
      }
      return true;
   } else {
      if(person.city == City.Karlsruhe) {
         _kaLogger.info("Failed to verify " + person.name);
      }
      return false;
   }
}
```

```java
// DOMAIN CLASS
boolean verifyAge(Person person) {
   if(person.age >= 18) {
      onSuccessfullVerification(person);
      return true;
   } else {
      onFailedVerification(person);
      return false;
   }
}

void onSuccessfullVerification(Person person) {
   verificationListener.forEach(listener -> listener.informSuccess(person))
}

void onFailedVerification(Person person) {
   verificationListener.forEach(listener -> listener.informFailure(person))
}

// KA LOGGER
void informSuccess(Person person) {
    if(person.city != City.Karlsruhe) {
        return;
    }
    info(person.name + " was successfully verified.");
}
```

Pure Fabrication: Weitere Beispiele





Indirection / Delegation

dt.: Indirektion / Delegation

oder: "Andere für sich arbeiten lassen"

Definition

> *Delegation is a way to make composition as powerful for reuse as inheritance. 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.*
>
> *et al.*
> *— aus 'Design Patterns' von Erich Gamma*

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

Beispiel: Indirektion

Was macht folgender Code?

```
// [...]
Message[] messages = getUnreadMessages();
Message m1 = messages[0];
Message[] updated = new Message[messages.length - 1];
for (int i = 0; i < updated.length; i++) {
  updated[i] = messages[i + 1];
}
setUnreadMessages(updated);
// [...]
```

- fehleranfällig
- schwer zu verstehen
- Lösung?

Beispiel: Indirektion

Lösung: Delegation an eine Liste

```
// [...]
List<Message> messages = getUnreadMessages();
Message m1 = messages.remove(0);
// [...]
```

- deutlich verständlicher
- Fachlichkeit immer noch unklar
- Lösung?

Beispiel: Indirektion

Lösung: Ableitung einer spezialisierten Liste

```
class UnreadMessages extends ArrayList<Message> {
    public Message oldest() {
      return messages.remove(0);
   }
}
```

- (+) fachliche Logik kann sprechend implementiert werden
- (-) viele technische Methoden sind aufrufbar
  - z.B. *sort* könnte hier zu Problemen führen
- Lösung?

Beispiel: Indirektion

Lösung: Wrapper mit Delegation

```
class UnreadMessages {
   private final List<Message> messages = new ArrayList<>();

   public void add(Message message) {
      messages.add(message);
   }

   public Message oldest() {
      return messages.remove(0);
   }

   public int size() {
      return messages.size();
   }
}
```

- Implementierung kann leicht gewechselt werden
  - z.B. auf ein Set
  - das ist bei Vererbung deutlich schwieriger, da überall eine ArrayListe erwartet wird

Vorteile: Indirection

- kann zu geringerer Kopplung führen (Low Coupling)
- flexibler als Vererbung
  - aber benötigt mehr Code und ist aufwendiger
- "Wir können jedes Problem lösen, indem wir eine zusätzliche Ebene der Indirektion einführen." (- David Wheeler)
  - *…außer das Problem von zu vielen Indirektionsebenen.*

Protected Variations

dt.: geschützte Veränderungen

Definition

- Implementierungen können wechseln
- ein Wechsel der Implementierung soll das System nicht beeinträchtigen
- das System soll vor den Auswirkungen des Wechsels *geschützt* werden

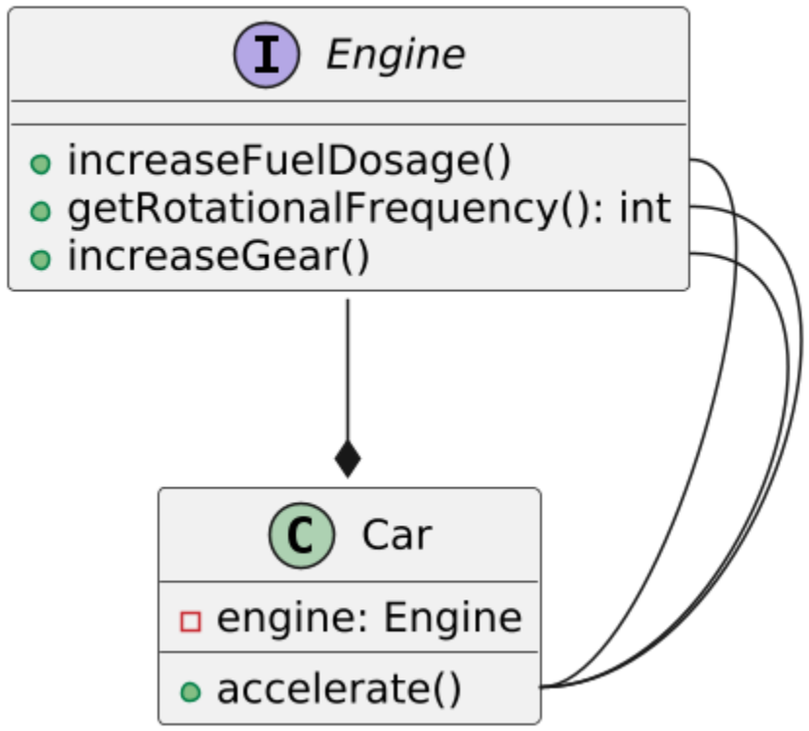
Protected Variations: Beispiel

```
class Car {

    GasolineEngine engine = new GasolineEngine();

    void accelerate() {
        engine.increaseFuelDosage();
        int rf = engine.getRotationalFrequency();
        if(rf > 5000) {
            engine.increaseGear();
        }
    }
}
```

- *Car* hängt direkt von *GasolineEngine* ab
- Änderungen in *GasolineEngine* können zu Änderungen in *Car* führen
- Wechsel der *Engine* z.B. zu Dieselmotor nicht möglich
- Lösung?

Protected Variations: Beispiel

Lösung: Abstraktion einführen

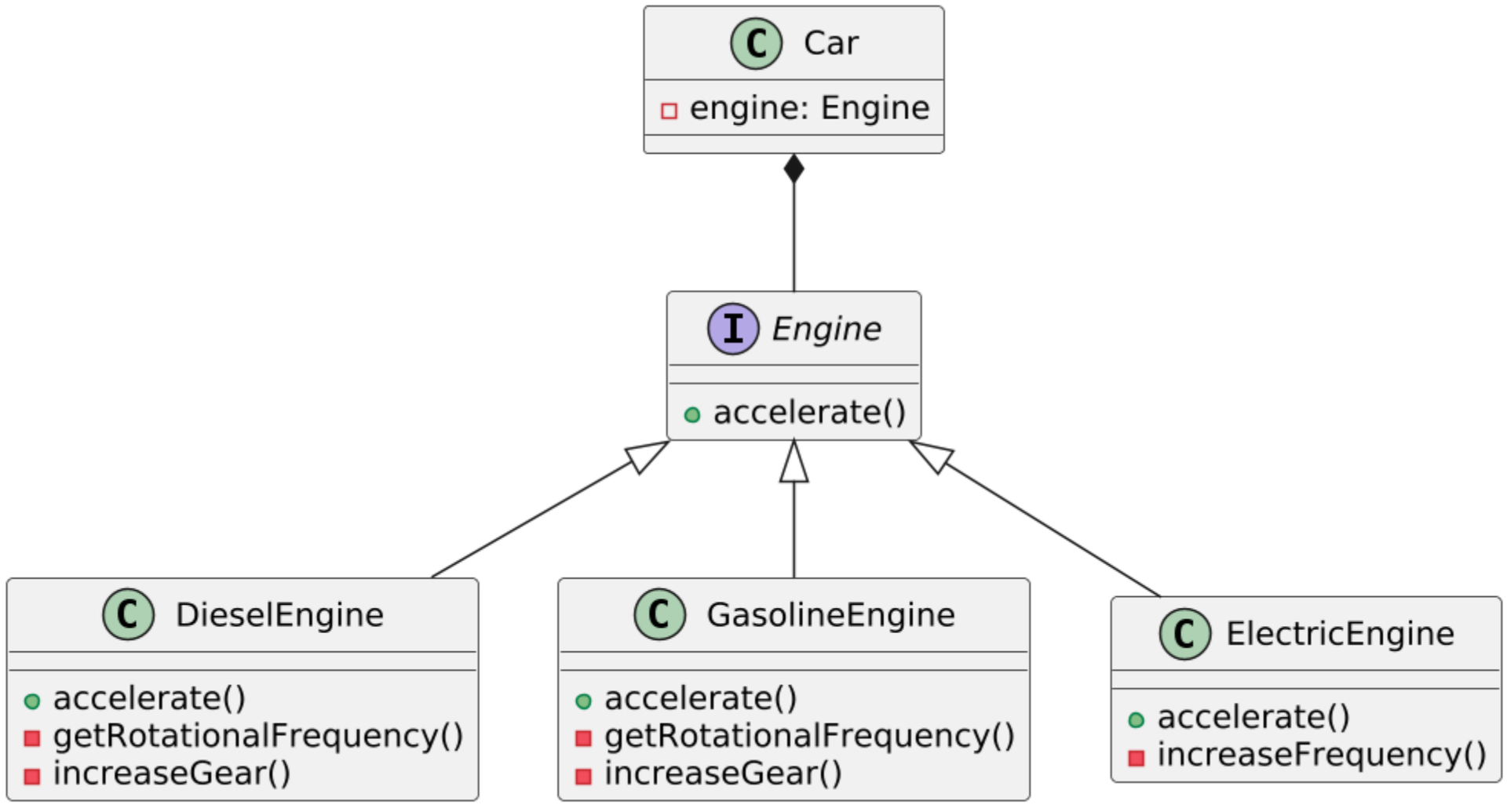


- (+) *Car* hängt nur noch von *Engine* ab
- (+) Implementierungen können nun wechseln ohne negative Auswirkungen
- (o) Abstraktion ist sehr technisch
- Lösung?

Protected Variations: Beispiel

Lösung: Information Expert etablieren

die Engine weiß selbst am Besten, wie sie beschleunigt



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

Protected Variations: Vorteile

- schützt vor Änderungen
- unterstützt Low Coupling
- führt zu (besseren) Abstraktionen

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

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