

Clean Code

Was erwartet Sie?

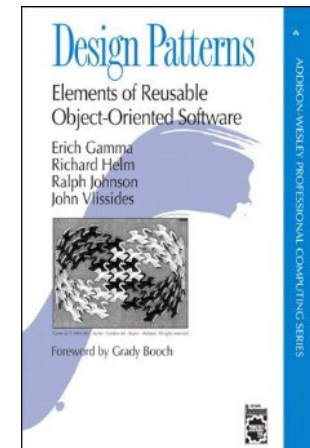
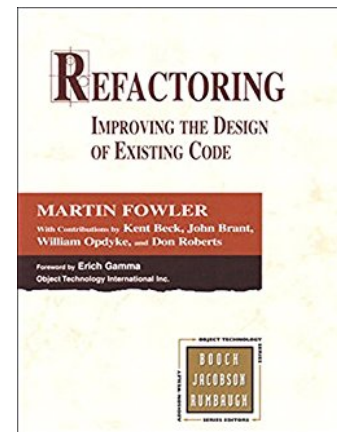
- Etwas über mich
- Prinzipien & Praktiken
- Einfache Beispiele

Wer ist das überhaupt?

- Sven Strittmatter (aka. Weltraumschaf)
- Mit 8 Jahren am Amiga 500 “programmiert”
- 2005 Studium abgeschlossen
- Software Architect bei [iteratec GmbH](#)

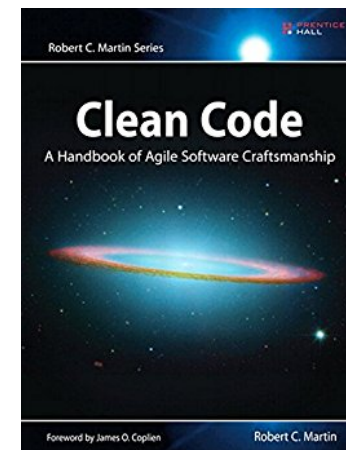
Und wie kommt der dazu?

- **Design Patterns** (Amazon)
 - natürlich überall Singleton benutzt
- **Martin Fowler**
 - **Refactoring** (Amazon)
 - Unit Testing
- **testbarer Code**
 - **Google Testing Blog**
 - **Miško Hevery**

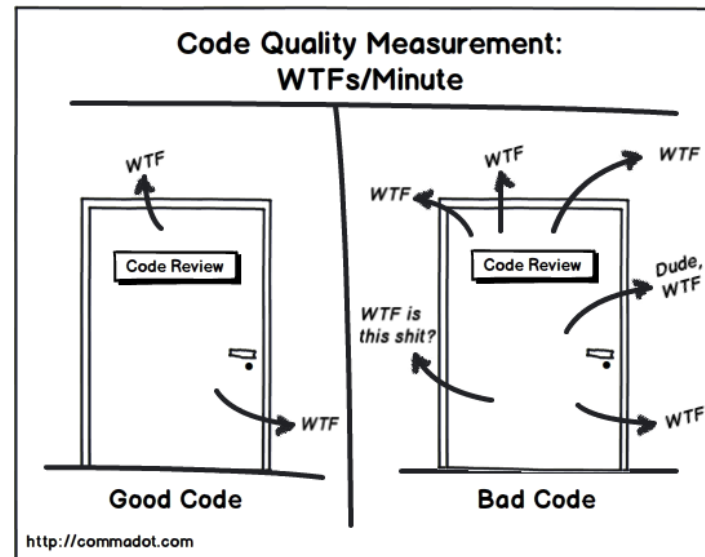


Die Softwerkskammer Karlsruhe

- Robert Martin (aka. Uncle Bob)
 - Software Craftsmanship
 - Clean Code (Amazon)
- Softwerkskammer Stuttgart



Und überall war Legacy



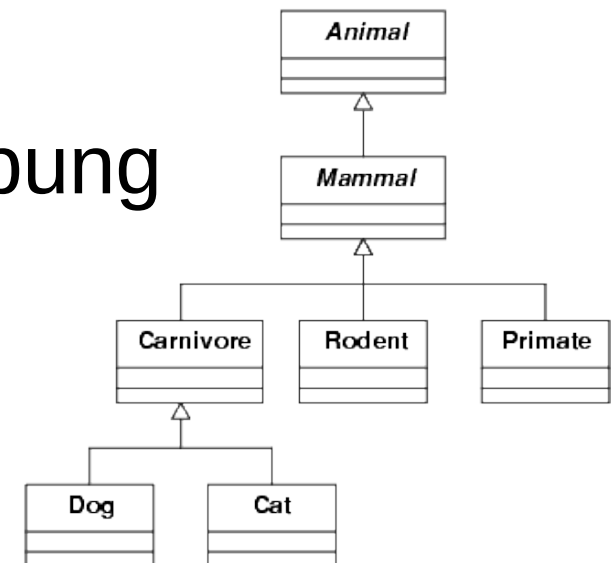
- Broken Window Theory
- Software Entropy (“software rot”)

Wie vermeidet man das?



Prinzipien & Praktiken

- DRY – Don't Repeat Yourself
- YAGNI -You Ain't Gonna Need It
- kleine Methoden/Klassen (mag auch der JIT)
- Dekomposition
- Vorsicht vor Optimierung
- Bevorzuge Komposition vor Vererbung
- ...



Prinzipien & Praktiken

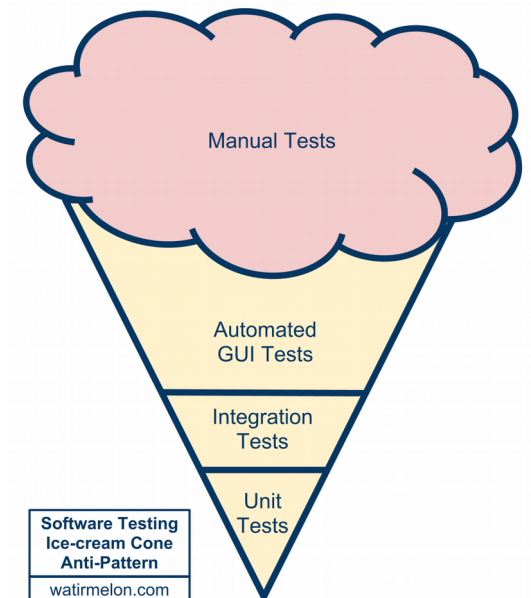
- **SOLID** (s. Uncle Bob)
 - Single Responsibility Principle
 - Mach nur ein Ding, aber das richtig.
 - Open Close Principle
 - Bestehenden Code nicht ändern, erweitern.
 - Liskov Substitution Principle
 - Sollte auch für Subklassen funktionieren.
 - Interface Segregation Principle
 - Kleine Interfaces, statt ein großes.
 - Dependency Inversion Principle
 - Bsp. `List<T>` statt `ArrayList<T>`.

Prinzipien & Praktiken

- weitere Heuristiken
 - Separation of Concern
 - Value vs. Service Objects
 - Mutable vs. Immutable
 - vermeide Seiteneffekte (pure Functions)
 - benutze **niemals** Singleton !!!1elf
 - vermeide Threads, und wenn doch
 - Synchronisation via Messages
 - kein Shared Memory!
 - Immutability

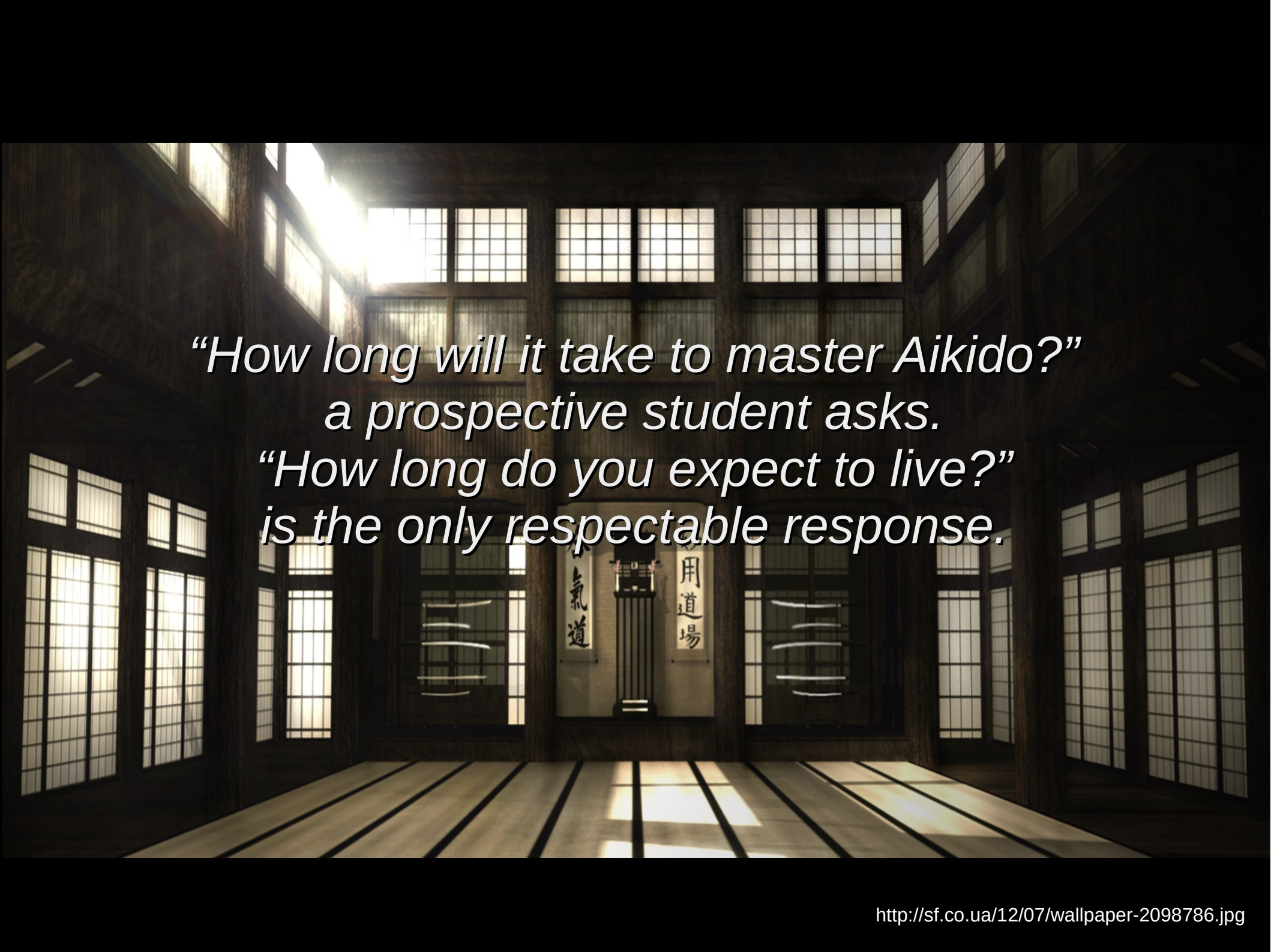
Prinzipien & Praktiken

- Reviews
 - Peer Reviews
 - Architektur Reviews
- Pair Programming
- automatisierte Tests (Unit, Integration, System, UI)
- Refactoring
 - horizontal vs. vertikal
 - commit early, commit fast
 - greppen: reflection
 - git bisect zur Fehlersuche



Tools die helfen

- Testframeworks
- IDE mit Refactoring-Tools
- Versionskontrollsystem
- Analyse-Tools (Sonar, Valgrind, ...)
- Profiler (nicht raten!)



*“How long will it take to master Aikido?”
a prospective student asks.
“How long do you expect to live?”
is the only respectable response.*

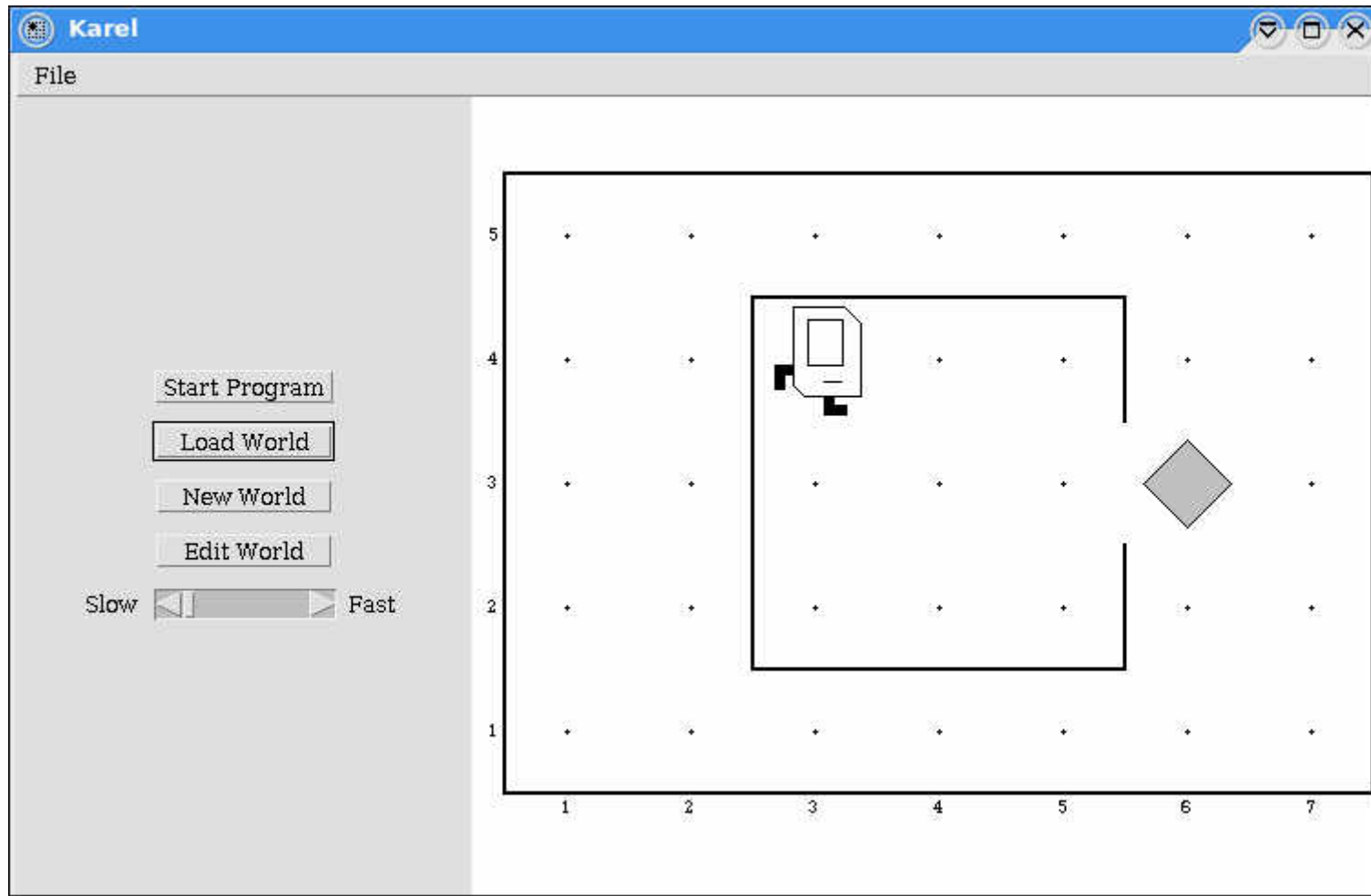
Clean Code Developer

- <http://clean-code-developer.de/>
- Verschiedene Grade wie im Kampfsport
- Disziplin & Selbstreflektion
- Übung macht den Meister
 - Coding Dojos
 - [Code Katas](#)



Beispiele

Dekomposition



Stanford University Course Programming Methodology CS106A (YouTube)

Dekomposition

```
abstract class SuperKarel {  
    abstract public function run();  
  
    public function beepersPresent() { /* ... */ }  
  
    public function pickBeeper() { /* ... */ }  
  
    public function putBeeper() { /* ... */ }  
  
    public function move() { /* ... */ }  
  
    public function turnArraound() { /* ... */ }  
}
```

(keine) Dekomposition

```
class DoYourThing extends SuperKarel {  
    public function run() {  
        $this->move();  
        while ($this->beepersPresent()) {  
            $this->pickBeeper();  
            $this->move();  
            $this->putBeeper();  
            $this->putBeeper();  
            $this->turnArraound();  
            $this->move();  
            $this->turnArraound();  
        }  
        $this->move();  
        while ($this->beepersPresent()) {  
            $this->pickBeeper();  
            $this->turnArraound();  
            $this->move();  
            $this->putBeeper();  
            $this->turnArraound();  
            $this->move();  
        }  
    }  
}
```

(mit) Dekomposition

```
class OurDoubleBeepers extends SuperKarel {  
    public function run() {  
        $this->move();  
        $this->doubleBeepersInPile();  
        $this->moveBackward();  
    }  
}
```

(mit) Dekomposition

```
class OurDoubleBeepers extends SuperKarel {  
    public function run() {  
        $this->move();  
        $this->doubleBeepersInPile();  
        $this->moveBackward();  
    }  
  
    public function doubleBeepersInPile() {  
        while ($this->beepersPresent()) {  
            $this->pickBeeper();  
            $this->putTwoBeepersNextDoor();  
        }  
  
        $this->movePileNextDoor();  
    }  
  
    public function moveBackward() {  
        $this->turnAround();  
        $this->move();  
        $this->turnAround();  
    }  
}
```

(mit) Dekomposition

```
class OurDoubleBeepers extends SuperKarel {  
    public function run() { /* ... */ }  
  
    public function doubleBeepersInPile() { /* ... */ }  
  
    public function moveBackward() { /* ... */ }  
  
    public function putTwoBeepersNextDoor() {  
        $this->move();  
  
        while ($this->beepersPresent()) {  
            $this->moveOneBeeperBack();  
        }  
  
        $this->moveBackward();  
    }  
  
    public function movePileNextDoor() {  
        $this->move();  
        $this->putBeeper();  
        $this->putBeeper();  
        $this->moveBackward();  
    }  
}
```

(mit) Dekomposition

```
class OurDoubleBeepers extends SuperKarel {  
    public function run() { /* ... */ }  
  
    public function doubleBeepersInPile() { /* ... */ }  
  
    public function moveBackward() { /* ... */ }  
  
    public function putTwoBeepersNextDoor() { /* ... */ }  
  
    public function movePileNextDoor() { /* ... */ }  
  
    public function moveOneBeeperBack() {  
        $this->pickBeeper();  
        $this->moveBackward();  
        $this->putBeeper();  
        $this->move()  
    }  
}
```

Dependency Injection

Goldene Regel des `new`-Operators:

- Ok für Domänen-Klassen, nicht für Services
- Ok in Tests und spezialisierten Erzeuger-Klassen, nicht in der Business-Logik

Dependency Injection

```
class FriendFinder {  
  
    public function __construct() {  
        $this->search = new Search();  
        $this->strategy = Strategy::create();  
    }  
  
}
```

- schwer zu testen (2 Abhängigkeiten)
- Don't do work in constructor!
- keine Test-Doubles möglich
- wird bei jedem Test ausgeführt
- Verkompliziert Test-Setup

Dependency Injection

```
class FriendFinder {  
  
    public function __construct(Search $search, Strategy $strategy) {  
        $this->search = $search;  
        $this->strategy = $strategy;  
    }  
  
}
```

- ermöglicht Test-Doubles
- nicht jeder Test braucht das volle Brett
- „Ask for things, don't look for things.“

Dependency Injection

- DI by Constructor (voriges Bsp.)
- DI by Setter
 - `$search->setSearch(Search $s)`
 - `$search->setStrategy(Strategy $s)`
- DI by Interface
 - <http://martinfowler.com/articles/injection.html>

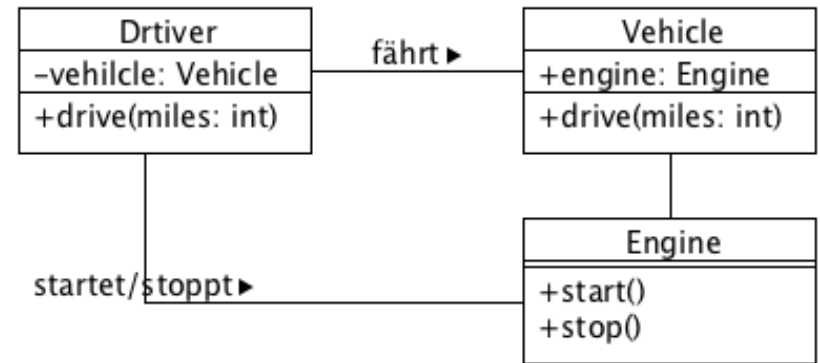
Gesetz von Demeter

*Objekte sollten nur mit Objekten
in ihrer unmittelbaren Umgebung
kommunizieren.*

Wikipedia

Gesetz von Demeter

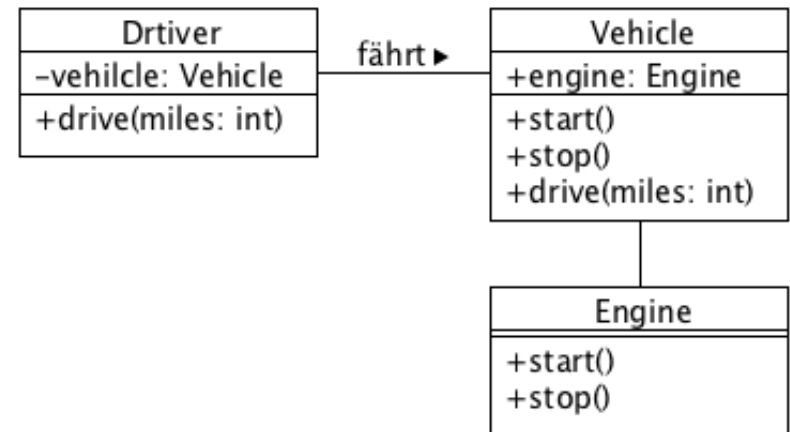
```
class Driver {  
    public function drive($miles) {  
        $this->vehicle->engine->start();  
        $this->vehicle->drive($miles);  
        $this->vehicle->engine->stop();  
    }  
}
```



- schwer testbar, braucht immer Engine-Objekt
- Driver eng an Engine gekoppelt
- interner Status von Vehicle offen gelegt

Gesetz von Demeter

```
class Driver {  
    public function drive($miles) {  
        $this->vehicle->start();  
        $this->vehicle->drive($miles);  
        $this->vehicle->stop();  
    }  
}
```



- leichter testbar
- Driver und Engine entkoppelt → leichter wartbar
- weniger fehleranfällig

Don't Repeat Yourself (DRY)

Jede Doppelung von Code oder auch nur Handgriffen leistet Inkonsistenzen und Fehlern Vorschub.



STACKOVERFLOW

Can You See it?

```
try {
    executeComponent(execute == null ? master : execute);
} catch (final ReplayException ex) {
    finishDebugging(obj.getDomainKey());
    resetDebugger();
    throw ex;
} catch (final KernelException ex) {
    LogManager.acquireLM().dumpStackTrace(this, VER, MN, IRecordType.TYPE_ERROR_EXC, ex);
    if (this.rules != null) {
        this.rules.setStatus(new LocaleStatus(ex.getMessage()));
    }
    finishDebugging(obj.getDomainKey());
    throw ex;
} catch (final ParseException ex) {
    LogManager.acquireLM().dumpStackTrace(this, VER, MN, IRecordType.TYPE_ERROR_EXC, ex);
    if (this.rules != null) {
        this.rules.setStatus(new LocaleStatus(ex.getMessage()));
    }
    handleParseException(ex);
} catch (final Exception ex) {
    LogManager.acquireLM().dumpStackTrace(this, VER, MN, IRecordType.TYPE_ERROR_EXC, ex);
    if (this.rules != null) {
        this.rules.setStatus(new LocaleStatus(ex.getMessage()));
    }
    finishDebugging(obj.getDomainKey());
    throw new Exception(ex.getMessage(), ex);
}
```


Can You See it?

```
try {
    executeComponent(execute == null ? master : execute);
} catch (final ReplayException ex) {
    finishDebugging(obj.getDomainKey());
    resetDebugger();
    throw ex;
} catch (final KernelException ex) {
    LogManager.acquireLM().dumpStackTrace(this, VER, MN, IRecordType.TYPE_ERROR_EXC, ex);
    if (this.rules != null) {
        this.rules.setStatus(new LocaleStatus(ex.getMessage()));
    }
    finishDebugging(obj.getDomainKey());
    throw ex;
} catch (final ParserException ex) {
    LogManager.acquireLM().dumpStackTrace(this, VER, MN, IRecordType.TYPE_ERROR_EXC, ex);
    if (this.rules != null) {
        this.rules.setStatus(new LocaleStatus(ex.getMessage()));
    }
    handleParserException(ex);
} catch (final Exception ex) {
    LogManager.acquireLM().dumpStackTrace(this, VER, MN, IRecordType.TYPE_ERROR_EXC, ex);
    if (this.rules != null) {
        this.rules.setStatus(new LocaleStatus(ex.getMessage()));
    }
    finishDebugging(obj.getDomainKey());
    throw new Exception(ex.getMessage(), ex);
}
```

Can You See it?

```
try {
    executeComponent(execute == null ? master : execute);
} catch (final ReplayException ex) {
    finishDebugging(obj.getDomainKey());
    resetDebugger();
    throw ex;
} catch (final KernelException ex) {
    LogManager.acquireLM().dumpStackTrace(this, VER, MN, IRecordType.TYPE_ERROR_EXC, ex);
    if (this.rules != null) {
        this.rules.setStatus(new LocaleStatus(ex.getMessage()));
    }
    finishDebugging(obj.getDomainKey());
    throw ex;
} catch (final ParserException ex) {
    LogManager.acquireLM().dumpStackTrace(this, VER, MN, IRecordType.TYPE_ERROR_EXC, ex);
    if (this.rules != null) {
        this.rules.setStatus(new LocaleStatus(ex.getMessage()));
    }
    handleParserException(ex);
} catch (final Exception ex) {
    LogManager.acquireLM().dumpStackTrace(this, VER, MN, IRecordType.TYPE_ERROR_EXC, ex);
    if (this.rules != null) {
        this.rules.setStatus(new LocaleStatus(ex.getMessage()));
    }
    finishDebugging(obj.getDomainKey());
    throw new Exception(ex.getMessage(), ex);
}
```

Komposition vor Vererbung

“Because inheritance exposes a subclass to details of its parent's implementation, it's often said that inheritance breaks encapsulation.”

Gang of Four

Komposition vor Vererbung

*Komposition fördert die lose
Kopplung und die Testbarkeit
eines Systems und ist oft flexibler.*

Komposition vor Vererbung

2 Konzepte in der OOP

- 1. Whitebox-Reuse (Vererbung)*
- 2. Blackbox-Reuse (Komposition)*

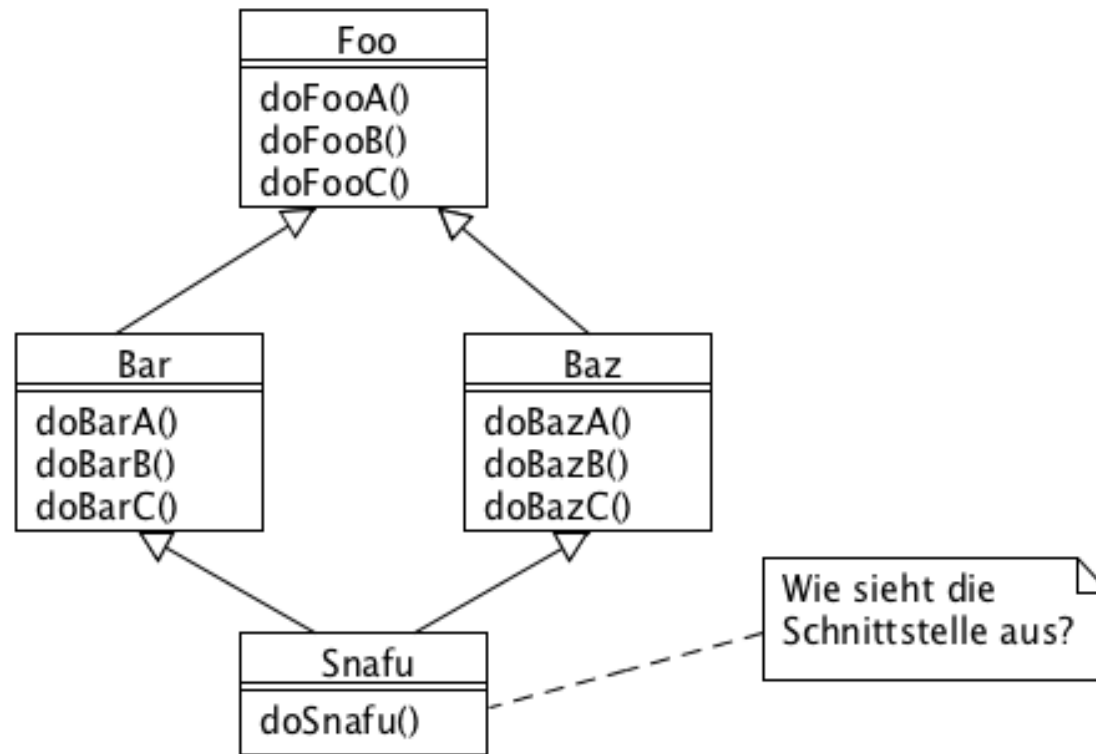
Komposition vor Vererbung

Whitebox-Reuse (Vererbung):

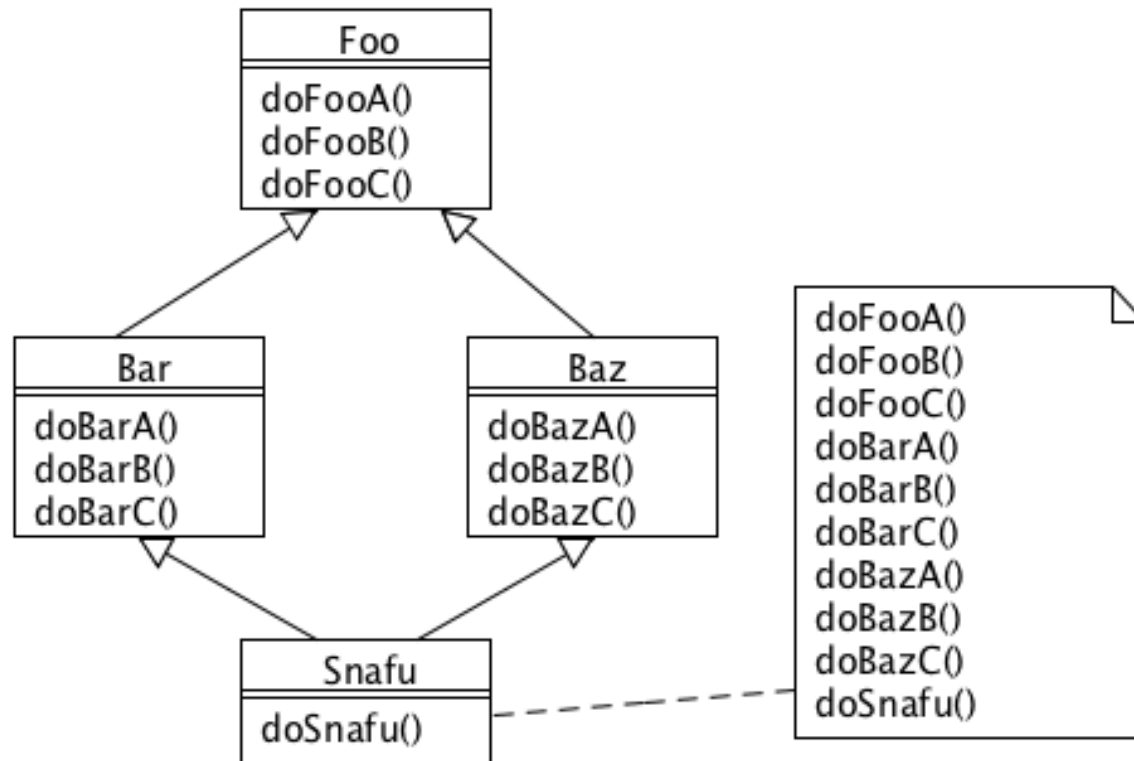
Subklasse abhängig von Elternklasse

- unnötige Komplexität (große Hierarchien, Mehrfachvererbung)
- schlecht testbar (großer Scope, Dependencies)
- statisch, Implementierung nicht zur Laufzeit tauschbar.

Komposition vor Vererbung



Komposition vor Vererbung

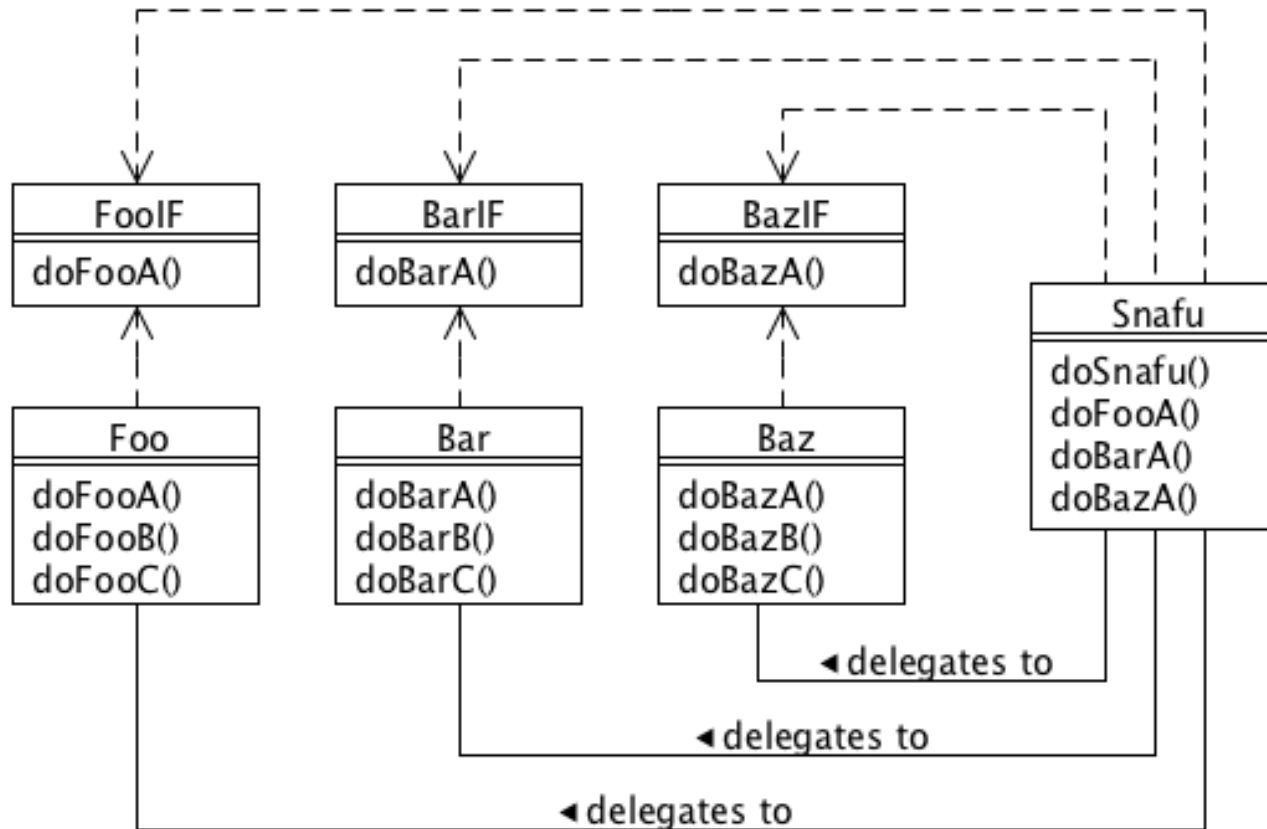


Komposition vor Vererbung

Blackbox-Reuse (Komposition):

Fördert die Entkopplung, wenn man geeignete Interfaces benutzt.

Komposition vor Vererbung



Fragen & Diskussion



https://c1.staticflickr.com/4/3177/2556117468_e821acd407_b.jpg

Slides: <https://github.com/Weltraumschaf/Slides/tree/master/CleanCode>