CLEAN CODE

WER NICHT FRAGT BLEIBT DUMM

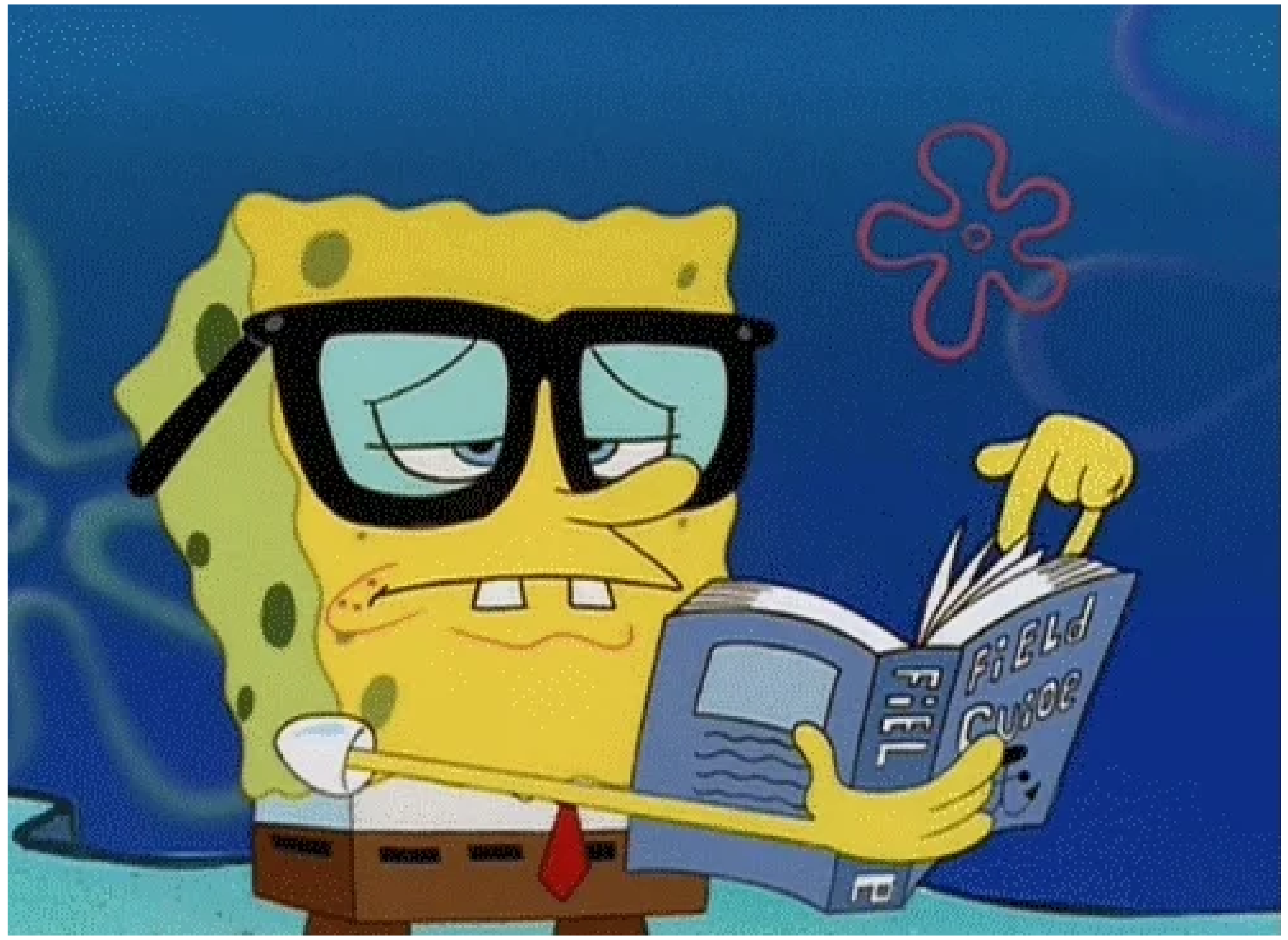
…bitte unterbrecht mich sofort, wenn ihr eine Frage habt!

WARUM IST CODE QUALITÄT WICHTIG?

- "Programmieren ist hauptsächlich Implementierung von neuen Features"
- "Der Code kompiliert, macht was er soll, somit ist alles in Ordnung"
- "Fachbereiche bezahlen für Features, nicht für "schönen" Code"

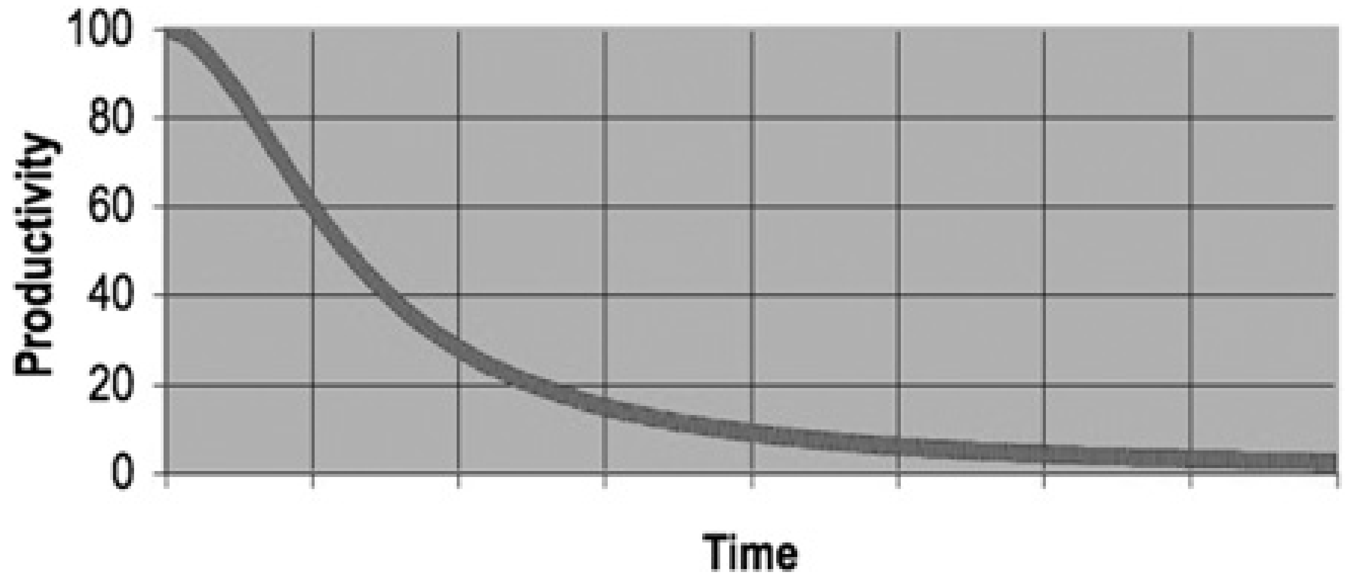
...what other people think we do...

…what we actually do…



- Das Verhältnis von Code lesen zu schreiben ist 10:1 (!)
- Die meiste Zeit wird existierender Code gelesen und evtl. erweitert
  - Wartung ist die längste Phase des Produktlebenszyklus
  - Trifft auch zu, wenn ein komplett neues Modul geschrieben wird
- Du bist nicht allein!
  - Entwickler arbeiten in Teams

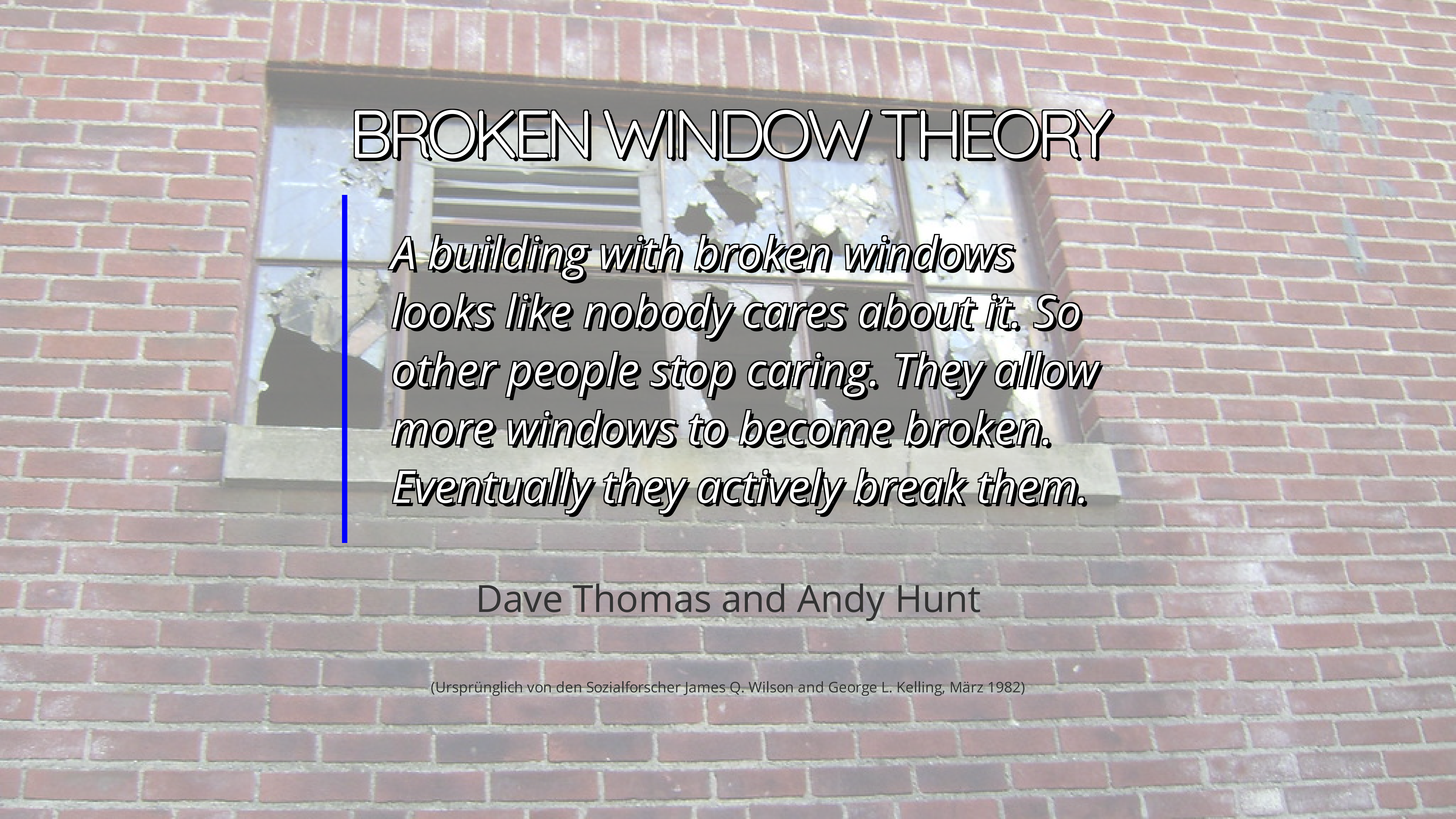
- Sinkende Produktivität über die Zeit bei unordentlichem Code



Quelle: Clean Code - A Handbook of Agile Software Craftsmanship (R.C. Martin)

PROBLEME MIT CHAOTISCHEM CODE

- Schwer zu verstehen
  - und noch viel schwerer zu erweitern
- Neue Fehler schleichen sich bei Änderungen leichter ein
- Sinkende Produktivität
- Das Chaos wächst, wenn man nicht mit Umsicht handelt!



BROKEN WINDOW THEORY

A building with broken windows looks like nobody cares about it. So other people stop caring. They allow more windows to become broken. Eventually they actively break them.

Dave Thomas and Andy Hunt

(Ursprünglich von den Sozialforscher James Q. Wilson and George L. Kelling, März 1982)

DEFINITIONEN VON CLEAN CODE

Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.

Grady Booch

> *Clean code can be read, and enhanced by a developer other than its original author.*

Dave Thomas

> *Clean code always looks like it was written by someone who cares.*

Michael Feathers

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

Martin Golding

WANN IST CODE CLEAN, SAUBER, AUFGERÄUMT?

- Einfach und direkt
- Aussagekräftige Namen
- Tut immer das, was man erwartet
- Einfach zu lesen und erweiterbar
- ...
- "Clean code does one thing well"

DIE WERTE DES CLEAN CODE ENTWICKLERS

1. Do Only What's Neccessary
2. Isolate Aspects
3. Minimize Dependencies
4. Honor Pledges

DO ONLY WHAT'S NECCESSARY

- Vorsicht vor Optimierungen:
  - You Ain't Gonna Need It (**YAGNI**)
  - Keep it simple, stupid (**KISS**)

ISOLATE ASPECTS

- Don't Repeat Yourself (DRY)
- Separation of Concerns (SoC)
- Single Level of Abstraction (**SLA**)
- Single Responsibility Principle (**SRP**)
- Interface Segregation Principle (**ISP**)

MINIMIZE DEPENDENCIES

- Information Hiding Principle
- **Law of Demeter**
- **Tell, don't ask**
- Dependency Inversion Principle (DIP)
- Interface Segregation Principle (**ISP**)
- Open Closed Principle (OCP)

LAW OF DEMETER

```
class Street { string Name { get; set; } }

class Address { Street Street { get; set; } }

class Person {
    int Id { get; set; }
    Address Address { get; set; }
}

class PersonService {

    void DoSomething() {
        var person = repo.GetById(id)
        var street = person.Address.Street; // ← train wreck
    }

    void DoSomethingBetter() {
        var person = repo.GetById(id)
        var street = person.GetStreet();
    }
}
```

HONOR PLEDGES

- Überraschungen vermeiden
- AKA: **Principle of Least Astonishment**
- Implementation mirrors design
- **Favour Composition over Inheritance (FCoI)**
- Liskov Substitution Principle (LSP)

THE BOY SCOUT RULE

Leave the campground cleaner than you found it!

PRAKTIKEN DES CLEAN CODE ENTWICKLERS

1. Embrace Uncertainty
2. Focus
3. Value Quality
4. Get Things Done
5. Stay Clean
6. Keep Moving

EMBRACE UNCERTAINTY

- Versionsverwaltung verwenden
- Automatisierte Unit- und Integrationstests
- Mit Mockups testen
- Continuous Integration einsetzen

FOCUS

- Modular arbeiten
- Test first
- Limit WIP

VALUE QUALITY

- Nur hohe Qualität akzeptieren
- Unittests automatisieren
- Code Reviews durchführen

GET THINGS DONE

- Iterative Entwicklung
- Continuous Delivery
- Obergrenze für Work in Progress (WiP)

STAY CLEAN

- Boy Scout Rule einhalten
- Regelmäßiges Refactoring
- Statische Code Analyse verwenden
- Code Coverage Analyse verwenden
- Coding Conventions einhalten

KEEP MOVING

- Man lernt das ganze Leben
- Wissen verteilen
- Selbstreflexion
- Grundursachen bekämpfen, nicht die Symptome
- Ergebnisse messen
- Regelmäßige Retrospektiven im Team

SOLID

"P" steht immer für "Principle"

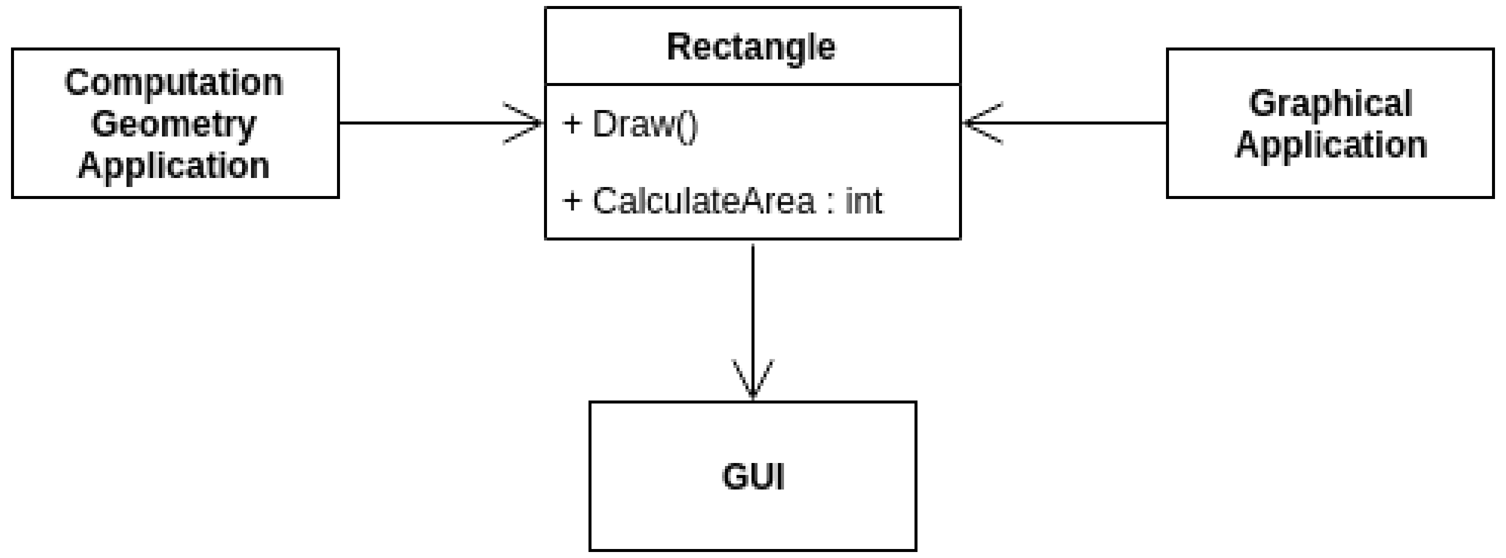
- SRP: Single Responsibility
- OCP: Open/Close
- LSP: Liskov Substitution
- ISP: Interface Segregation
- DIP: Dependency Inversion

Code Beispiele folgen...

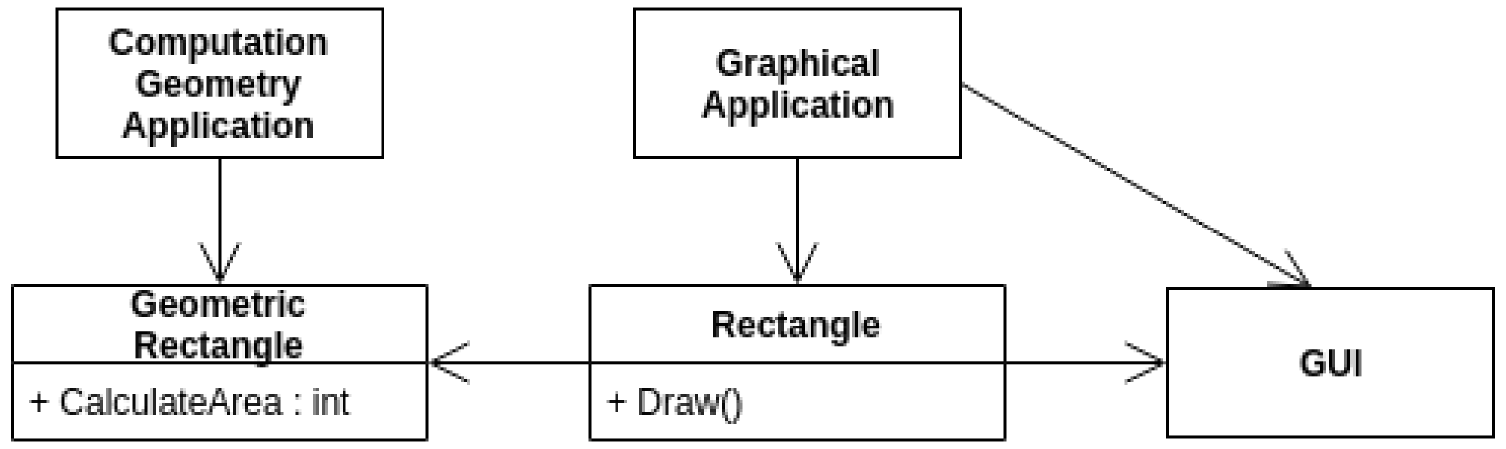
SINGLE RESPONSIBILITY PRINCIPLE

> *A module should have only one reason to change*

Robert C. Martin



From: Agile Principles, Patterns and Practices in C#, Robert C. Martin



From: Agile Principles, Patterns and Practices in C#, Robert C. Martin

```
class SomeService {
    void DoMagic(string message) {

        var emailService = new EmailService { Credentials = "bar" };

        try { emailService.Send(message); }
        catch (Exception e) { /*  ...  */ }

        _smsService.Send(message);
    }
}
```

```
class SomeServiceBetter {
    private readonly IEmailService _emailService;
    private readonly ISmsService smsService;

    SomeServiceBetter(IEmailService emailSrv, ISmsService smsService) {
        _emailService = emailSrv;
        _smsService = smsService;
    }

    void DoMagic(string message) {
        _emailService.Send(message);
        _smsService.Send(message);
    }
}
```

```
class Rectangle {
    // ...
    int width;
    int height;

    void Draw() {
        // draw to output device
    }

    int CalculateArea() ⇒ width * height;
}
```

```
class GeometricRectangle {
    // ...
    int width;
    int height;

    int CalculateArea() ⇒ width * height;
}

class Rectangle {
    void Draw() {
        // draw to output device
    }
}
```

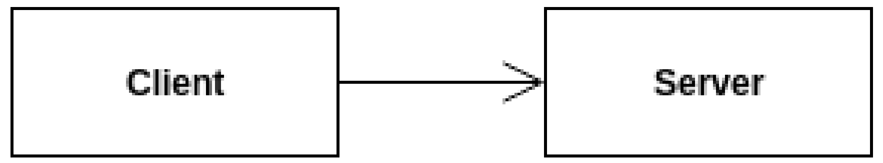
LIVE CODING

Wenn was unklar ist: Fragen!

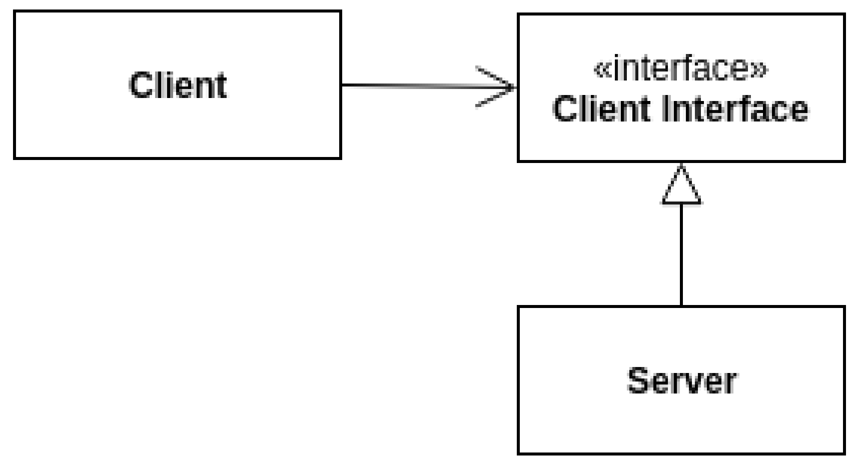
OPEN/CLOSE PRINCIPLE

> *Software entities ... should be open for extension, but closed for modification.*

Bertran Meyer / Robert C. Martin



From: Agile Principles, Patterns and Practices in C#, Robert C. Martin



From: Agile Principles, Patterns and Practices in C#, Robert C. Martin

```
class SomeService {

    void DoMagic(string message) {
        _emailService.Send(message);
        _smsService.Send(message);
    }
}
```

```
class SomeServiceBetter {

    private readonly List<IService> _services;

    SomeServiceBetter(List<IService> services) {
        _services = services;
    }

    void DoMagic(string message) {
        for (var service in _services) {
            service.Send(message);
        }
    }
}
```

LIVE CODING

Wenn was unklar ist: Fragen!

LISKOV SUBSTITUTION PRINCIPLE

> *Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.*

Barbara Liskov

```
class Base {
    virtual int DoSomethingWithNumber(int i) => i;
}

class OtherNotOk : Base {
    override int DoSomethingWithNumber(int i)
        => i == 42
           ? i
           : throw new Exception(); // <- NOT OK !!
}

class OtherOk : Base {
    override int DoSomethingWithNumber(int i) => i * 100;
}
```

LIVE CODING

Wenn was unklar ist: Fragen!

INTERFACE SEGREGATION PRINCIPLE

> *Many client-specific interfaces are better than one general-purpose interface.*

Robert C. Martin

```
interface IPerson {
    Guid Id { get; set; }
    string FirstName  { get; set; }
    string LastName  { get; set; }
    Address Address  { get; set; }
    List<PersonalDetail> PersonalDetails  { get; set; }
}

class Person : IPerson {
    Guid Id { get; set; }
    string FirstName  { get; set; }
    string LastName  { get; set; }
    Address Address  { get; set; }
    List<PersonalDetail> PersonalDetails  { get; set; }
}
```

- Randnotiz:
    - this is a **Java Bean** -> pointless
        - (JEE violates most aspects of OO, even more than .NET)
- violates ISP

Typische Anforderungen:

- Listenansicht (wenig Information pro Eintrag)
- Detailansicht (viele Informationen)

```csharp
interface IPersonDetailViewModel {
    Guid Id { get; set; }
    string FirstName { get; set; }
    string LastName { get; set; }
    string Address { get; set; }
}

class PersonDetailViewModel : IPersonDetailViewModel {
    Guid Id { get; set; }
    string FirstName { get; set; }
    string LastName { get; set; }
    Address Address  { get; set; }
}
```

```csharp
interface IPersonListViewModel {
    Guid Id { get; set; }
    string Name { get; set; }
}

class PersonListViewModel : IPersonListViewModel {
    Guid Id { get; set; }
    string Name { get; set; }
}
```

```
IPersonListViewModel ConvertToListViewModel(Person person)
    ⇒ new PersonListViewModel(person);

IPersonDetailViewModel ConvertToDetailViewModel(Person person)
    ⇒ new PersonDetailViewModel(person);
```

```
class PersonListViewModel : IPersonListViewModel {
    // ctor
    PersonListViewModel(Person person) {
        Id = person.Id;
        Name = $"{person.LastName}, {person.FirstName}";
    }
    // ...
}
```

```
class PersonDetailViewModel : IPersonDetailViewModel {
    // ctor
    PersonDetailViewModel(Person person) {
        Id = person.Id;
        FirstName = person.FirstName;
        LastName = person.LastName;
        Address = person.Address;
    }
    // ...
}
```

Typische Anforderung:

- Schnittstelle: **GetPeople**
- Schnittstelle: **GetPersonById**

```
class Person : IPerson  { /* ... */ }

class PersonService {
    List<IPerson> GetPeople() { /* ... */ }
    IPerson GetPersonById(Guid id) { /* ... */ }
}

class PersonController {
    // ...
    ActionResult PeopleList() {
        var people = _personService.GetPeople();
        return people; // View has unused infos in model!
    }
}
```

```
class Person : IPersonListEntry, IPersonDetail  { /* ... */ }

class PersonService {
    List<IPersonListEntry> GetPeople() { /* ... */ }
    IPersonDetail GetPersonById(Guid id) { /* ... */ }
}

class PersonController {
    // ...
    ActionResult PeopleList() {
        var people = _personService.GetPeople();
        return people; // View model is optimized
    }
}
```

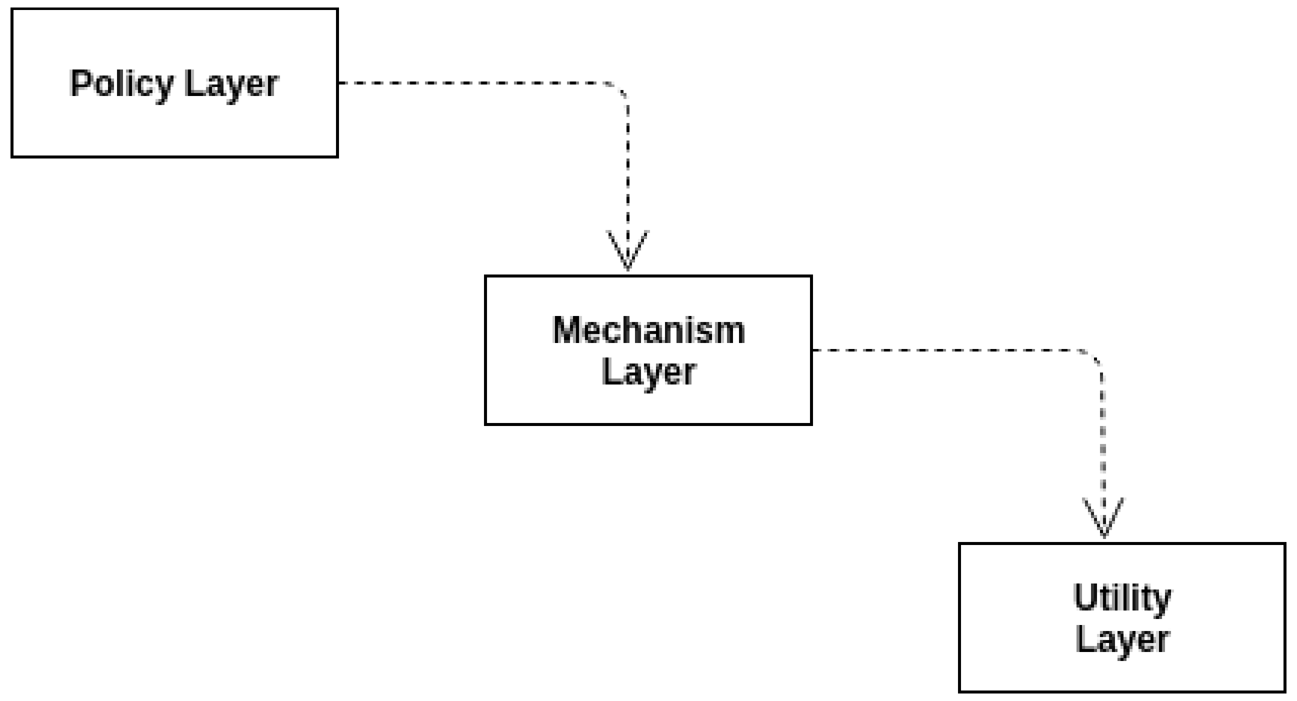
LIVE CODING

Wenn was unklar ist: Fragen!

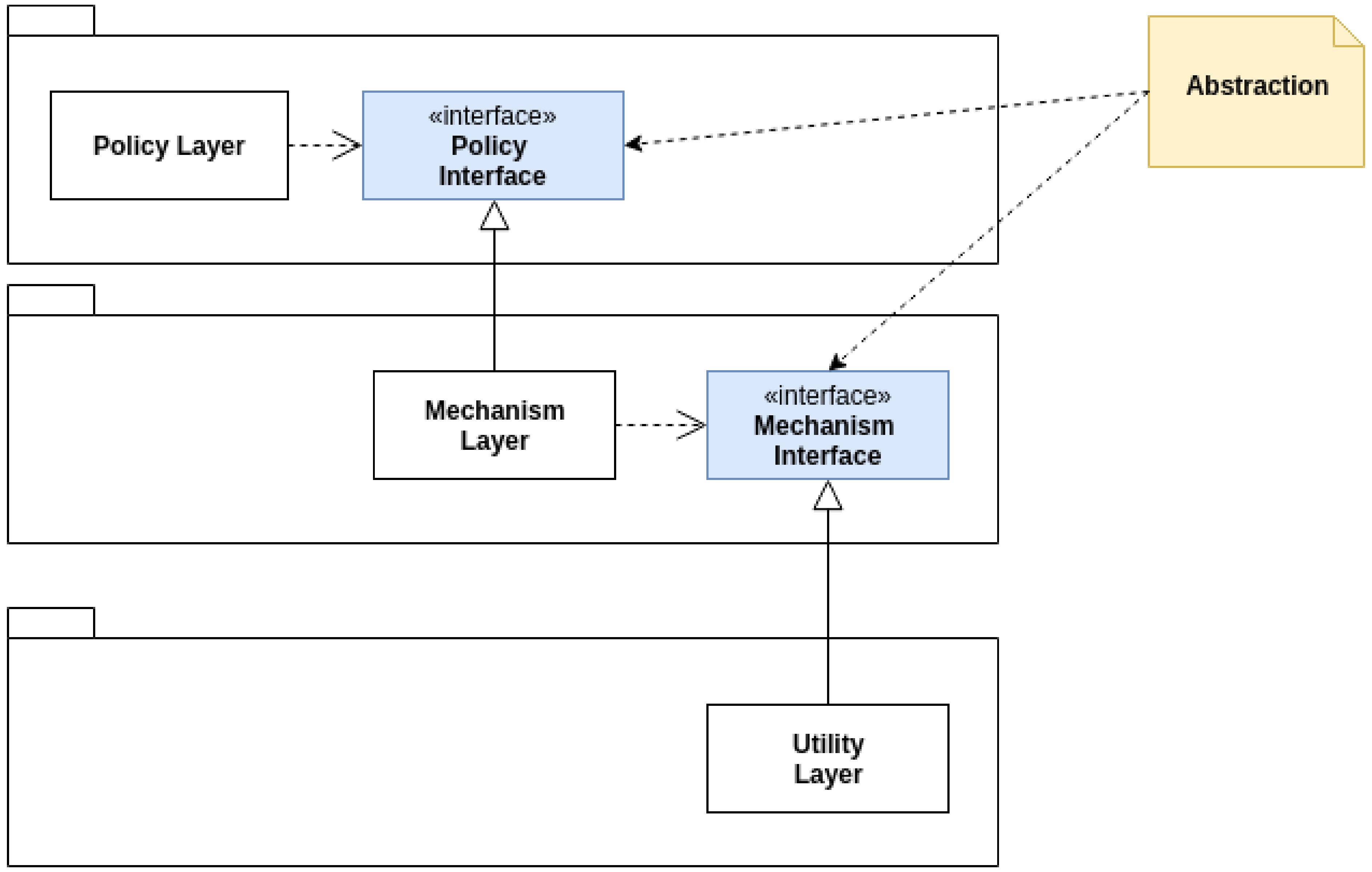
DEPENDENCY INVERSION PRINCIPLE

> *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> *Abstractions should not depend upon details. Details should depend upon abstractions.*

Robert C. Martin



From: Agile Principles, Patterns and Practices in C#, Robert C. Martin



From: Agile Principles, Patterns and Practices in C#, Robert C. Martin

LIVE CODING

Wenn was unklar ist: Fragen!

CODE SMELLS

see Wikipedia: Code smells

LITERATUR

- **The Pragmatic Programmer**. From Journeyman to Master. Andrew Hunt und David Thomas
- **Clean Code**: A Handbook of Agile Software Craftsmanship. Robert C. Martin
- **Working Effectively with Legacy Code**. Micheal Feathers
- **The Clean Coder**: A Code of Conduct for Professional Programmers, Robert C. Martin
- **Clean Architecture**: A Craftsman's Guide to Software Structure and Design. Robert C. Martin