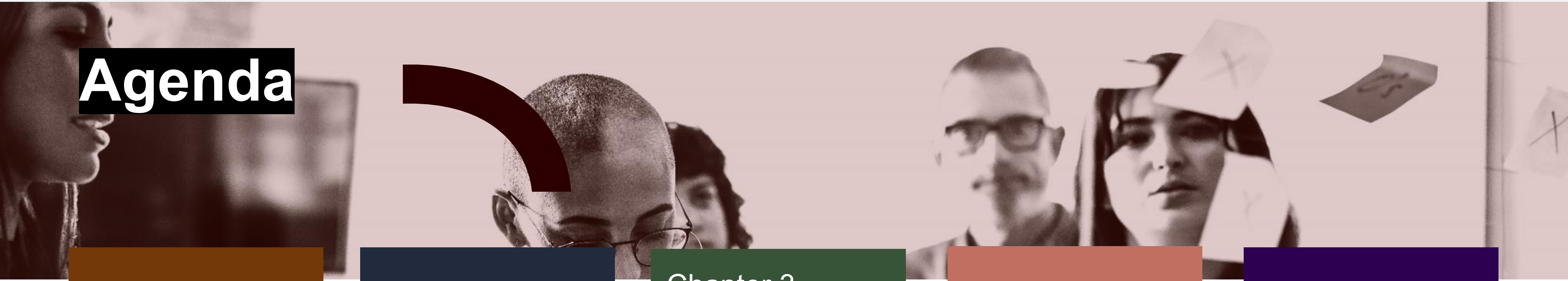
Clean Code - Professional code creation and maintenance

Agenda



Chapter 1
Clean Code – Introduction

Chapter 2
Object-oriented programming

Chapter 3
Professional classes and objects

Chapter 4
Names

Chapter 5
Methods

Chapter 6
Comments and documentation

Chapter 7
Code Formatting

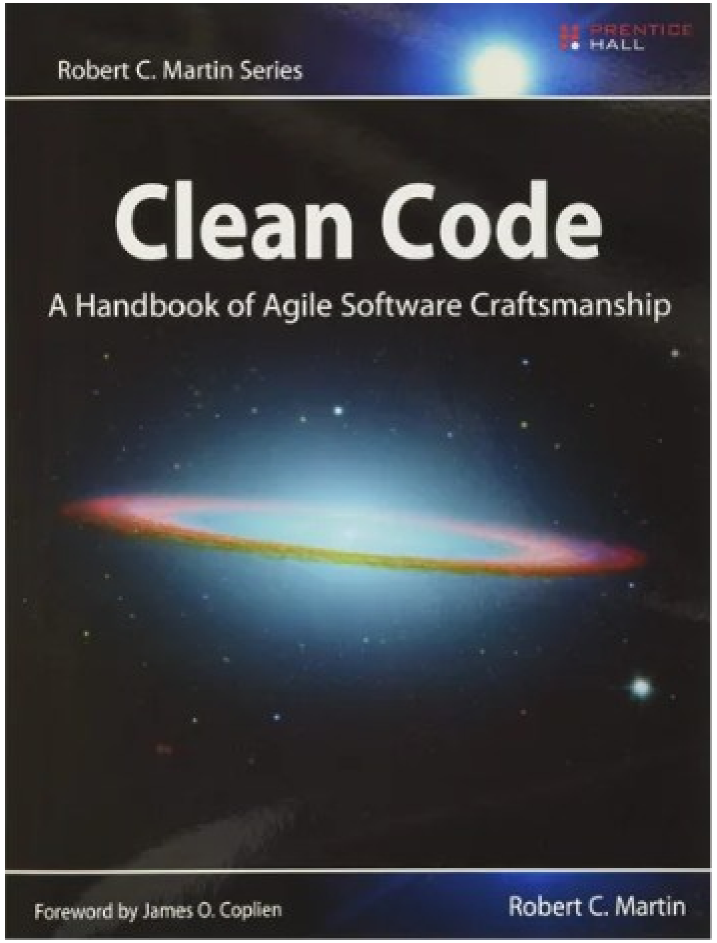
Chapter 8
Special topics

Chapter 9
Software Metrics

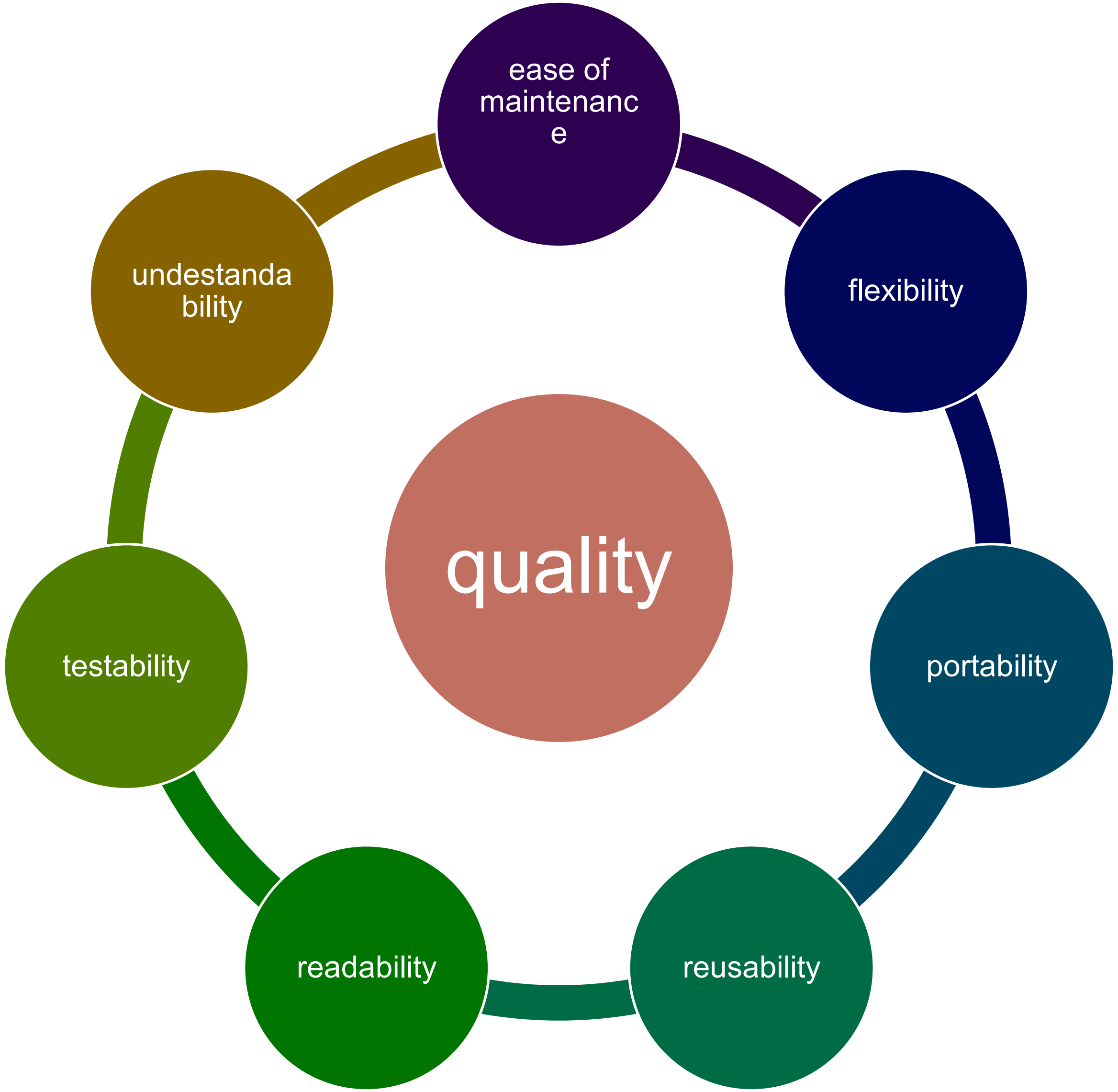
1. Clean Code – Introduction

Clean Code

- Term originates from a book by Robert C. Martin
  - ▸Concepts and ideas originate from software engineering
  - ▸Are actually well known
- Summary of concepts and ideas from software engineering
  - ▸Emerged with OO languages and object-oriented software development
  - ▸Essentially, it's about:
    - Clean structures (e.g. design patterns)
    - Naming conventions
    - Class and method sizes
  - ▸Can only be applied to scripting languages and non-OO languages to a limited extent

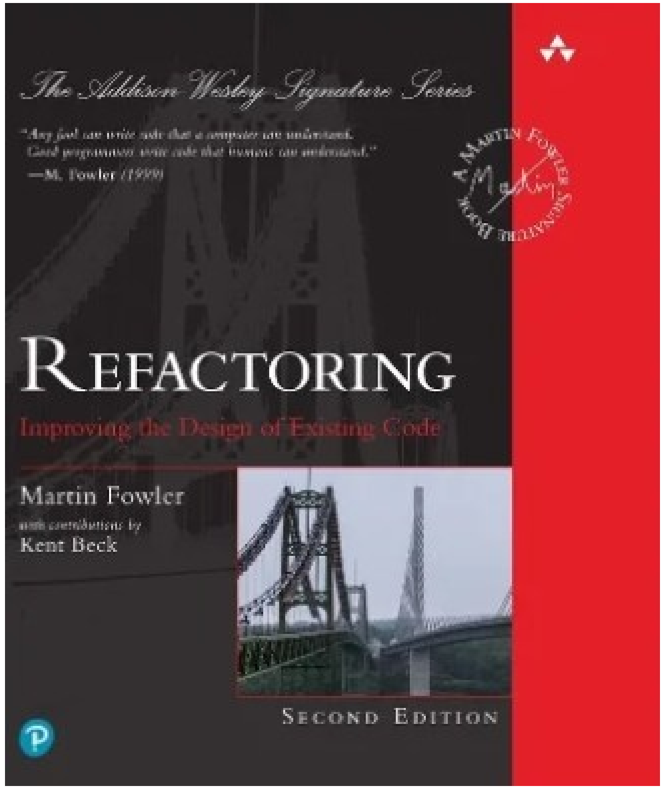

Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

Developers view on quality



Code Smell

- The term is intended to describe more concrete criteria for refactoring as a vague references to aesthetics.
- Code smells are not programming errors
  - ▶ but rather poorly structured
  - ▶difficult to understand
  - ▶When corrections and enhancements are made, new errors often creep in again.
- Code smells can be an indication of a deeper problem.
  - ▶Hidden poor structure that is only recognised through reworking.


The Addison Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Common code smells (1)

- Code duplication
  - Same code appears in multiple places
- Long method
- Big class
  - Too many variables, methods, etc in a class (also called „god object“)
- Long parameters list
  - Instead of passing an object to a method, object attributes are extracted and passed to the method as a long parameter list.
- Divergent changes
  - For a change, a class must be adapted in several places..
- Shotgun Surgery
  - This smell is even more serious than divergent changes: one change requires further changes to be made to many classes.
- Feature Envy
  - A method is more interested in the properties – especially the data – of another class than in those of its own class.

Common code smells (2)

- Data clumps
  - ▸A group of objects often occur together: as fields in some classes and as parameters of many methods..
- Primitive Obsession
  - ▸Elementary types are used, even though classes and objects are more meaningful for simple tasks.
- Case statements in object-oriented code
  - ▸Switch-case statements are used even though polymorphism makes them largely redundant and solves the associated problem of duplicate code.
- Parallel inheritance hierarchies
  - ▸For every subclass in one hierarchy, there is always a subclass in another hierarchy.
- Lazy class
  - ▸A class does too little to justify its existence.
- Speculative generality
  - ▸All possible special cases were provided for, even though they are not needed at all; such general code requires maintenance effort without being of any use..

Common code smells (3)

- Temporary fields
  - ▸An object only uses a variable under certain circumstances – the code is difficult to understand and debug because the field does not appear to be used.
- Message chains
- Middle Man (Proxy)
  - ▸A class delegates all method calls to another class.
- Inappropriate intimacy
  - ▸Two classes are too closely intertwined.
- Alternative classes with different interfaces
  - ▸Two classes do the same thing, but use different interfaces to do so.
- Incomplete library class
  - ▸A class in a programme library requires extensions in order to be used in an area suitable for it.

Common code smells (4)

- Data class
  - ▸Classes with fields and access methods without functionality.
- Refused Bequest
  - ▸Subclasses do not need the methods and data they inherit from superclasses (see also Liskov's substitution principle).
- Comments
  - ▸Comments generally make code easier to understand. However, comments often seem to be necessary precisely where the code is poor. Comments can therefore be an indication of poor code.

Further smells and programming anti-patterns (1)

- In addition to the smells mentioned by Fowler, there are a number of code smells that are often referred to as programming anti-patterns:
- Uncommunicative name
  - ▸A name that says nothing about the properties or use of the named item. Meaningful names are essential for understanding programme code.
- Redundant code
  - ▸A piece of code that is no longer used.
- Indecent exposure
  - ▸Internal details of a class are unnecessarily part of its external interface.
- Contrived complexity
  - ▸Forced use of design patterns where a simpler design would suffice.

Further smells and programming anti-patterns (2)

- Names that are too long
  - ▸In particular, the use of architectural or design components in the names of classes or methods.
- Names that are too short
  - ▸The use of 'x', 'i' or abbreviations. The name of a variable should describe its function.
- Over-callback
  - ▸A callback that attempts to do everything.
- Complex branches
  - ▸Branches that check a set of conditions that have nothing to do with the functionality of the code block.
- Deep nesting
  - ▸Nested if/else/for/do/while statements. They make the code unreadable.

Architecture Smells – Code Smells "next level"

- Cyclical usage relationships between packages, layers and subsystems
- Size and division of packages or subsystems

- Solutions mostly with architecture patterns:
  - Layer
  - MVC, e.g. with the Observer

2. Object-oriented programming

Types of programming languages

- Procedural programming
  - Cobol
  - C
  - Fortran
- Functional programming
  - Haskell
  - LISP
- Logic programming
  - Prolog
- Object-oriented programming
  - Java
  - C++
  - Smalltalk

UML – Unified Modeling Language

- Originally comes from OO software development.
- UML is a graphical description language and can be customised.
- UML consists of various diagrams.
- UML diagrams can be used in different phases of a project with different intentions.
- Here only for reading (to understand concepts)
  - ▸Class and package diagrams are sufficient for this purpose
  - ▸Conceptual, to be able to describe initial structures

Abstraction

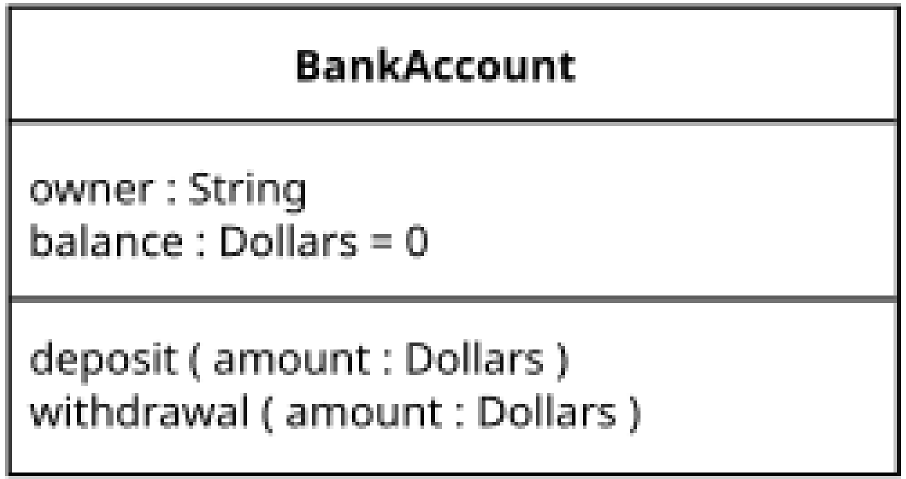
Each object in the system can be viewed as an abstract model of an actor that can complete tasks, report and change its status, and communicate with other objects in the system without having to disclose how these capabilities are implemented.

Such abstractions are either classes (in class-based object orientation) or prototypes (in prototype-based programming).

Classes and instances

- Classes
  - ▸ Grouping of similar objects (comparable behaviour and state)
  - ▸ Describe the behaviour and type of state
  - ▸ States are called **attributes, member** or **instance variables**
  - ▸ Abilities/behaviour are called **methods**

- Instances
  - ▸ Concrete manifestations of a class

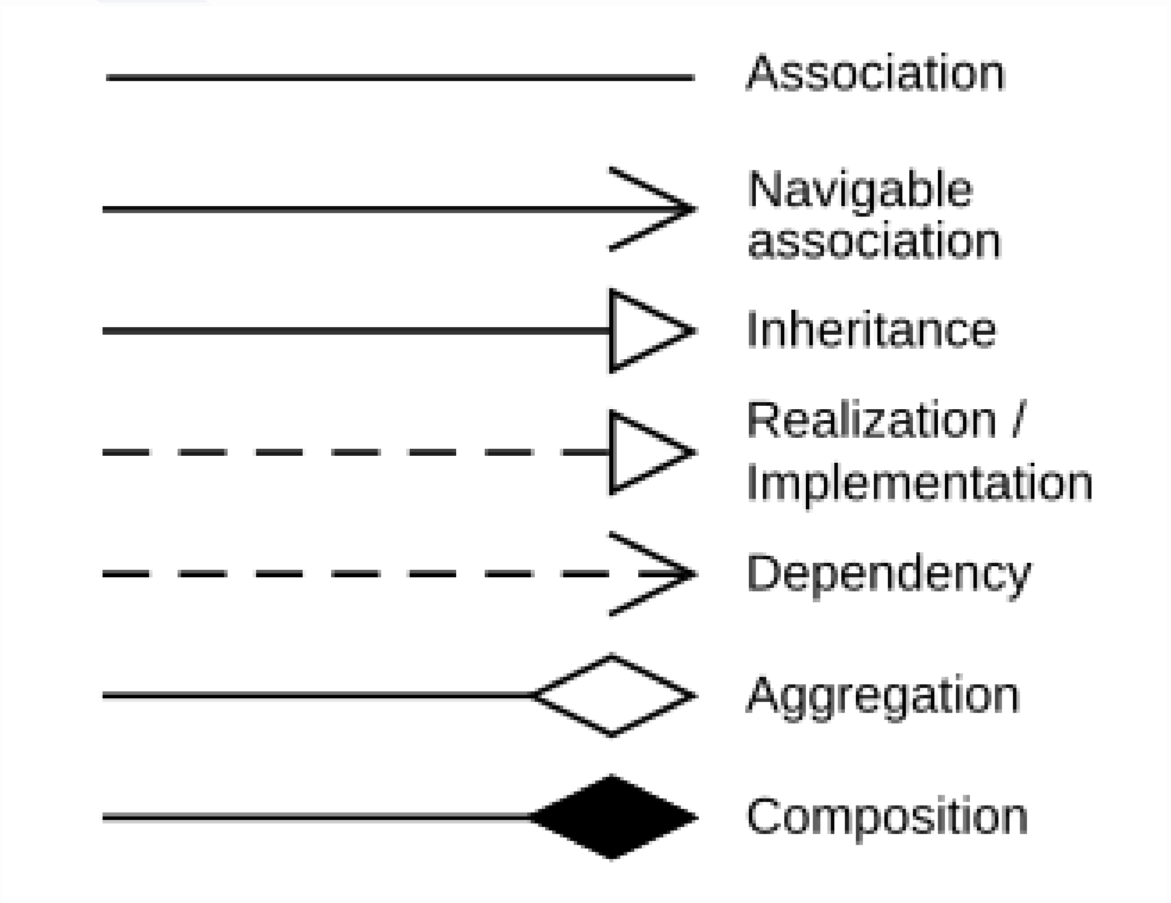


BankAccount

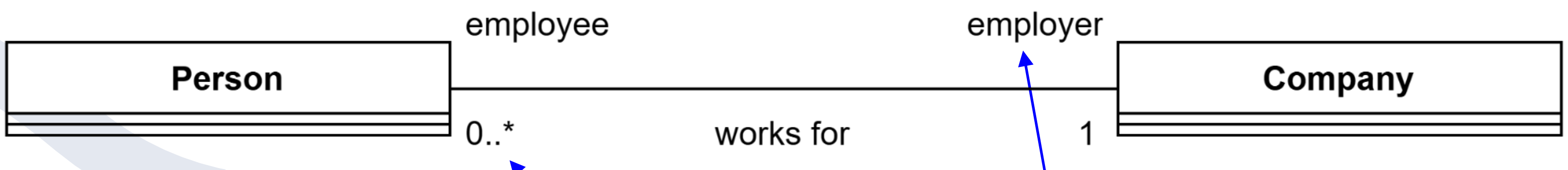
owner : String
balance : Dollars = 0

deposit (amount : Dollars)
withdrawal (amount : Dollars)

Associations and other relations



Associations with roles and multiplicities



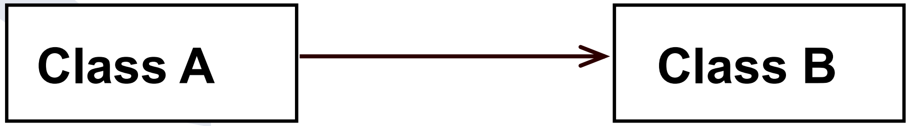
- **Role**
  - Describes the role of this object for the relation
  - Can be used for code generation
    `(Person.Arbeitgeber`
    `Firma.Angestellter)`

- **multiplicity**
  - describes the quantitative relationship between objects of the classes

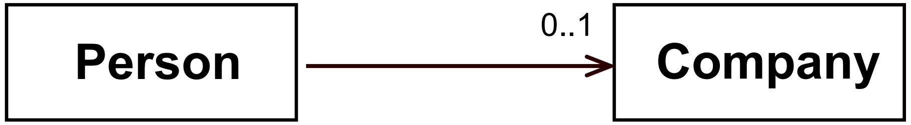
<u>Examples for multiplicities:</u>

1	Exactly one
0..1	zero or one
*	Any number (zero included)
2..8	two to eight (included)
1..*	at least one

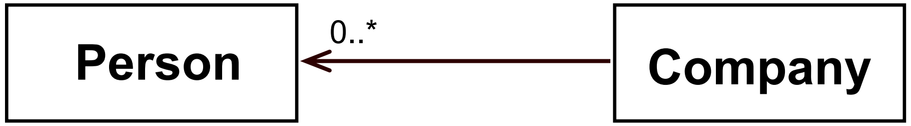
Associations with navigiation direction



The navigation direction is indicated by an arrowhead.



An object of the Person class references a Company object.



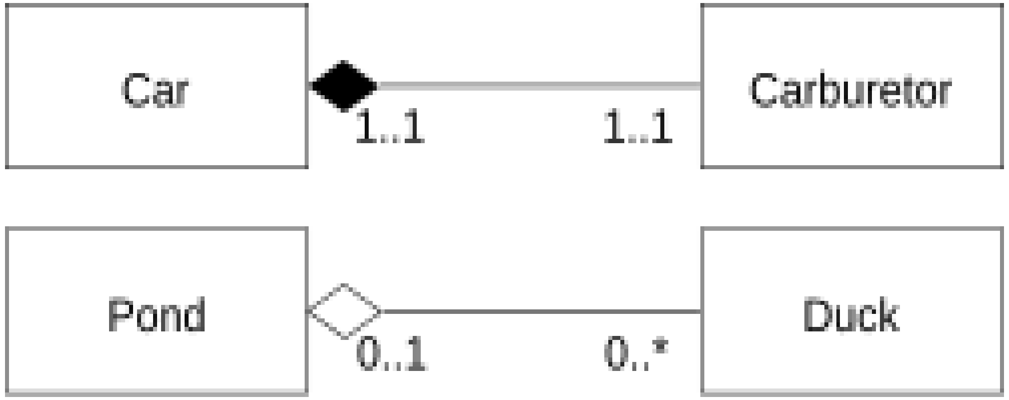
An object of the Company class references any number of Person objects.

CLEAN CODE

- Bidirectional associations should be avoided (coupling)
- Classes with bidirectional associations belong in the same package

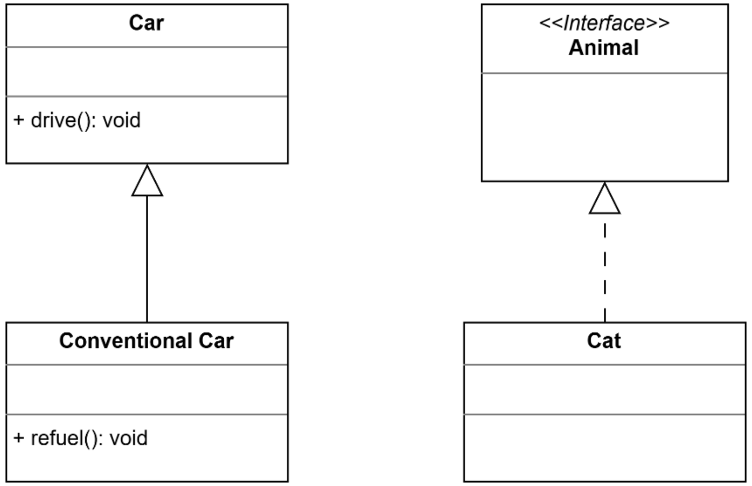
Composition and aggregation

- Usually identical in code (attributes)
- Composition:
  - The composite is not "viable" on its own
  - A car consists of a carburettor.
  - A carburettor is part of a car (and cannot exist without a car).
  - Also possible as aggregation in other contexts.
- Aggregation
  - Stronger association
  - "Contains" or "consists of" relationship
  - A pond contains ducks (or may not).
  - A duck belongs to no pond or to one pond.



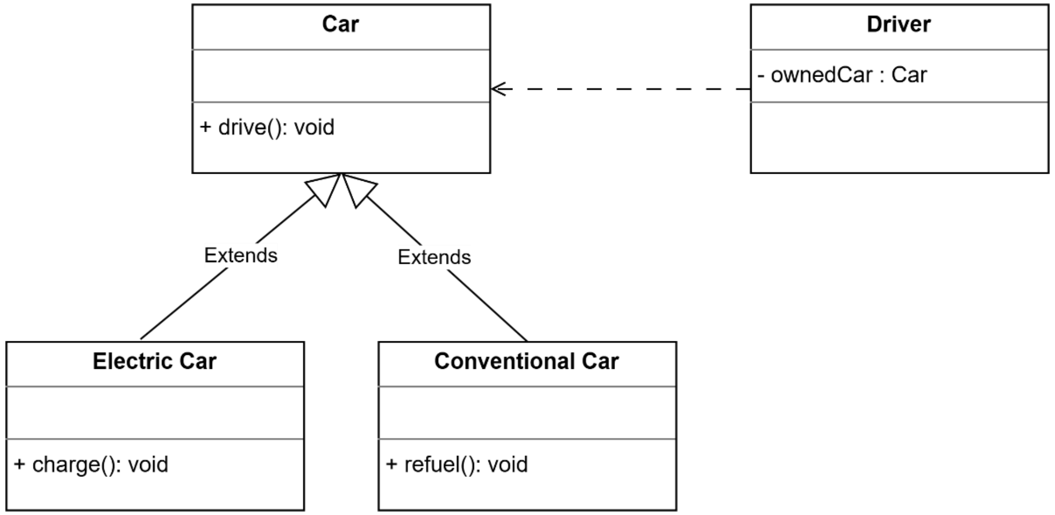
Inheritance

- Classes inherit from each other.
- A subclass is a specialisation of the superclass.
- For interfaces: realisation or implementation.
- The superclass is thereby extended or changed.
- Common features are summarised.
- Saving on paperwork is not the goal.
- A is a (special) B.
  - ▸An employee is a (special) person
  - ▸A dog is an animal.
- Abilities of the superclass cannot be "revoked"! (from an interface perspective)



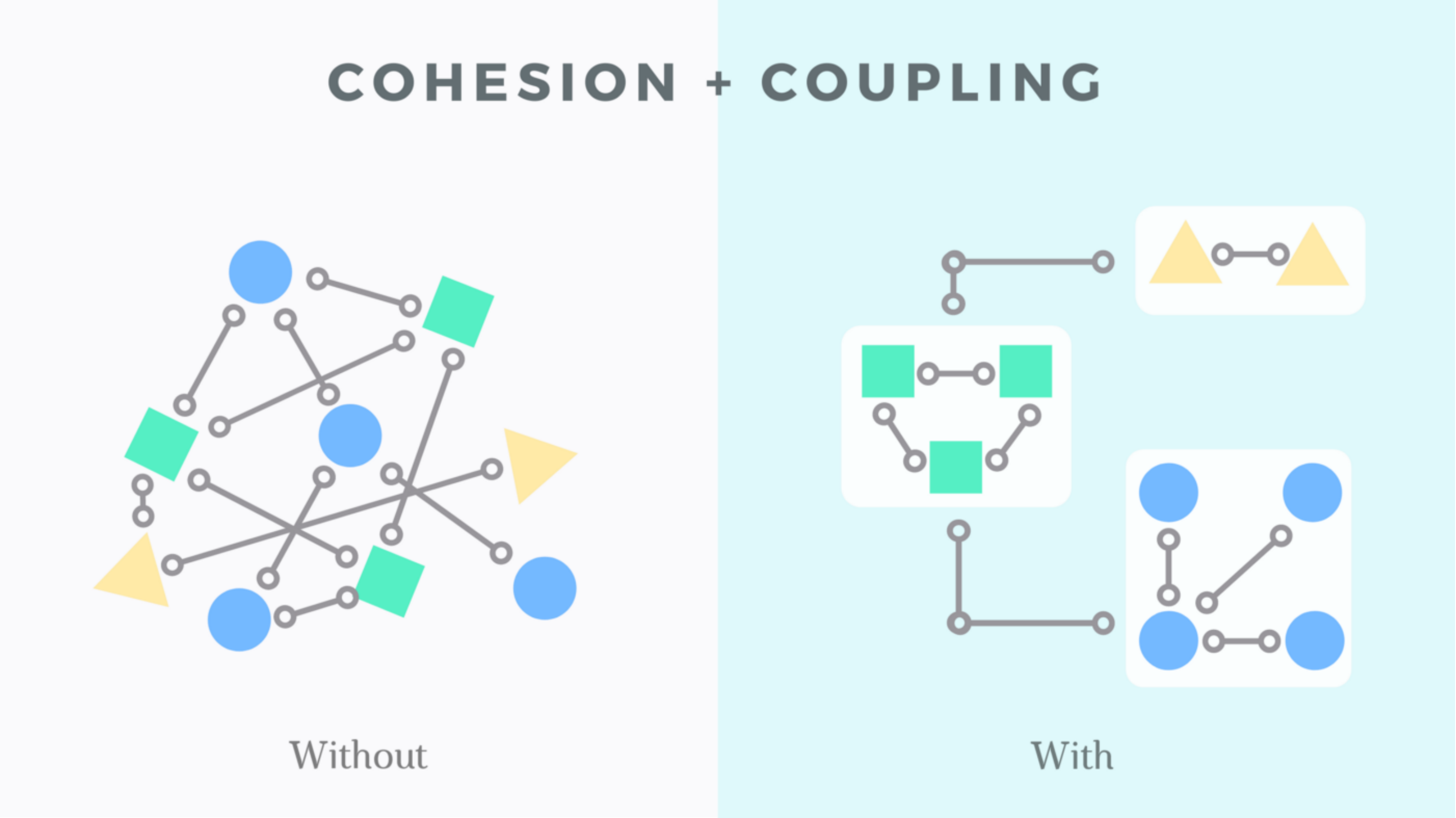
Polymorphism

- An instance can also be addressed as an instance of the superclass (interface).
- Which method (actual code) is executed at runtime is determined based on the actual class.
  - ▸ The compiler uses the "label" (variable type) to check whether the method may be called.
  - ▸ The system decides at runtime which code is executed.
- The calling code does not need to know the specific subclass whose method is being executed!



The three goals of clean object orientation

- Good encapsulation
- Loose coupling
- High cohesion



Encapsulation

- Classes only allow access to their internal components via well-defined interfaces.
- Implementation details are hidden.
- "Accidental" access is prevented.
- Implementation details can be changed without having to recompile the user code.
- Tools:
  - ►Visibilities: Distinction between public and protected interfaces
  - ►Interfaces should be kept stable.
- Access methods (getters and setters)
  - ►Also serve as documentation (the attribute becomes part of the public interface).
  - ►Allows checking of invariants and restrictions.
  - ►Creation of virtual attributes.

Visiblilities

For classes, methods, fields

- **Public (+)**
  - Can be called from everywhere
- **Private (-)**
  - May called only within the current class
- **Protected (#)**
  - May be called only by the current class and subclasses
- **Package (~)**
  - Visible within the same package
  - Not existent in C++
- **Instance-Private / Instance-Protected**
  - Like Private and Protected, but the call has to come from the same instance
  - Not part of UML
  - Not available in Ruby or Scala

Notes on Getters and Setters

- Some languages only allow access to attributes via getters and setters (e.g. Smalltalk).
- Depending on the language, access methods can result in a loss of performance,
  - ▸but not in Java.
- A decision must be made as to whether encapsulation should also apply to your own subclasses (private vs. protected attributes).

Rule 2-1

The signature and behaviour of interface methods should only be changed retrospectively in exceptional cases.

Rule 2-2

New methods should only be added to the interface if there is a specific use case for them.

Rule 2-3

Fields should be private by default, helper methods should be package-visible by default.

Coupling I

- Describes how closely two classes are related.
- How strongly does a change in one class affect the other (does it also need to be adjusted?)
- Content coupling
  - ▸One class accesses the internals of the other class.
  - ▸A change in one will probably also necessitate a change in the other class.
- Interface coupling
  - ▸Classes are only coupled to each other via their interface
  - ▸The classes do not "know" the concrete implementations
  - ▸Implementations can be changed or even replaced without affecting the calling class (as long as the contract of the methods is adhered to)

Coupling II

- Problems with interface coupling
  - At some point, an instance of the class to be used must be created.
  - This allows the calling class to "recognise" the class to be used.
  - Possible solutions:
    - FACTORY pattern
    - DEPENDENCY INJECTION pattern (INVERSION OF CONTROL)
    - Some languages offer implicit support for this.
- Data coupling
  - Capabilities are no longer described using methods, but using a general (usually text-based) language.
  - The executing class only has one method, "execute". What is to be executed is passed to this method as a parameter.
  - Minimal coupling, allows cross-language calls (e.g. web services)
  - Disadvantages:
    - No compiler checking
    - Difficult to read

Rule 2-4

Each class should be encapsulated with an appropriate interface. Client code should only access the class via the interface.

Cohesion

- Methods and attributes must "fit together“
- Indicator of weak cohesion
  - Certain attributes are only used by certain methods
  - Partitions (clusters) of attribute/method groups

Low cohesion ❌

```cpp
class BookManager {
private:
    std::string title;
    std::string author;
    int year;

public:
    BookManager(std::string t, std::string a, int y)
        : title(t), author(a), year(y) {}

    void printBookInfo() {
        std::cout << "Title: " << title
                  << ", Author: " << author
                  << ", Year: " << year << std::endl;
    }

    void saveToFile(const std::string& filename) {
        std::ofstream file(filename);
        if (file.is_open()) {
            file << title << "," << author << "," << year << std::endl;
        }
    }

    void loadFromFile(const std::string& filename) {
        std::ifstream file(filename);
        if (file.is_open()) {
            getline(file, title, ',');
            getline(file, author, ',');
            file >> year;
        }
    }

    void recommendBook() {
        if (year < 2000) {
            std::cout << title << " is a classic!" << std::endl;
        } else {
            std::cout << title << " is modern literature." << std::endl;
        }
    }
};
```

High cohesion ✅

```cpp
// Represents book data only
class Book {
public:
    std::string title;
    std::string author;
    int year;

    Book(std::string t, std::string a, int y)
        : title(t), author(a), year(y) {}
};

// Handles file persistence only
class BookRepository {
public:
    void save(const Book& book, const std::string& filename) {
        std::ofstream file(filename);
        if (file.is_open()) {
            file << book.title << "," << book.author << "," <<
book.year << std::endl;
        }
    }

    Book load(const std::string& filename) {
        std::ifstream file(filename);
        std::string title, author;
        int year = 0;
        if (file.is_open()) {
            getline(file, title, ',');
            getline(file, author, ',');
            file >> year;
        }
        return Book(title, author, year);
    }
};
```

```cpp
// Handles printing only
class BookPrinter {
public:
    void print(const Book& book) {
        std::cout << "Title: " << book.title
                  << ", Author: " << book.author
                  << ", Year: " << book.year << std::endl;
    }
};

// Handles business logic only
class BookRecommender {
public:
    void recommend(const Book& book) {
        if (book.year < 2000)
            std::cout << book.title << " is a classic!" << std::endl;
        else
            std::cout << book.title << " is modern literature." << std::endl;
    }
};
```

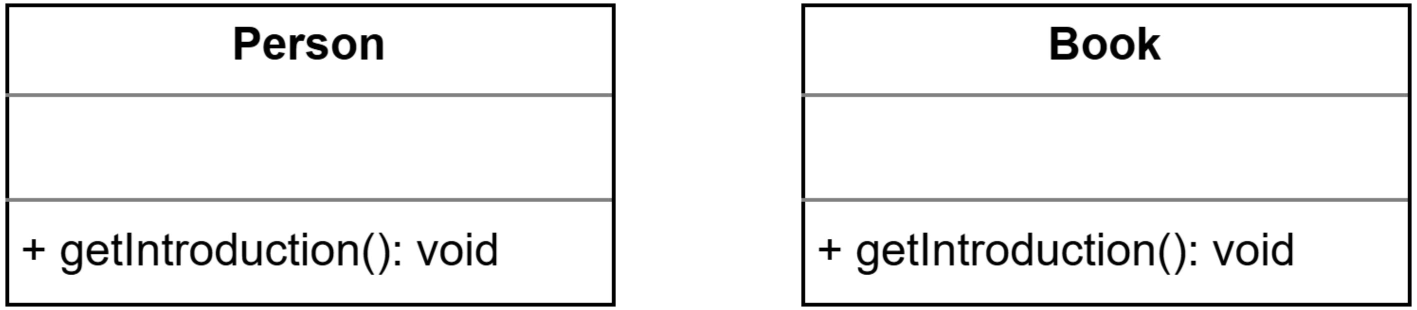
Rule 2-5

Classes should have strong cohesion.

3. Professional classes and objects

Inheritance is evil...

- What would be an appropriate, common parental class here?



Person

+ getIntroduction(): void

Book

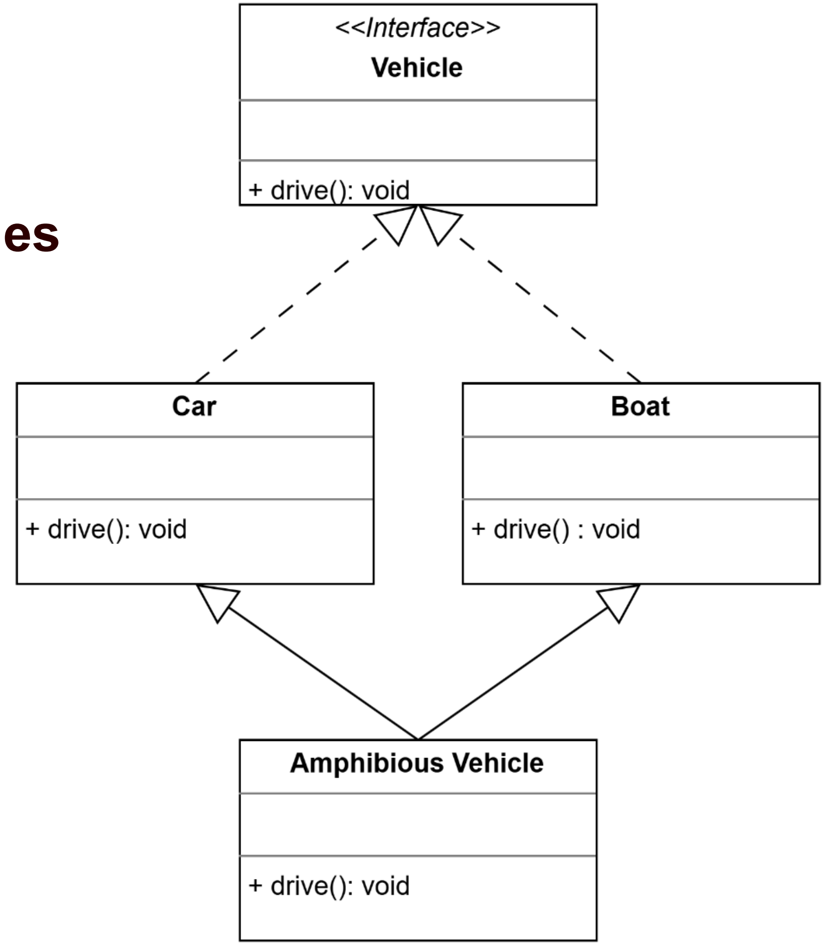
+ getIntroduction(): void

Rule 3-1

Inheritance should only be used to describe actual specialisations.

Multi-Inheritance

- **A class has two superclasses**
  - *"An amphibious vehicle is a land vehicle and a boat."*
- **Possible problems with overlaps in method names**
- **Diamond Problem:**
  - The drive() method is called polymorphically on a vehicle.
  - For Amphibious vehicles:
    Which method is executed?



Rule 3-4

Don‘t use multi-inheritance!

Professional class design

- Building on the foundations of the previous chapter
  - ▸Details that distinguish a good class from a professional class
  - ▸Especially with the interaction of several classes
  - ▸Whether through inheritance or associations.
- Principles:
  - ▸SOLID
  - ▸KISS
  - ▸Design Patterns

SOLID

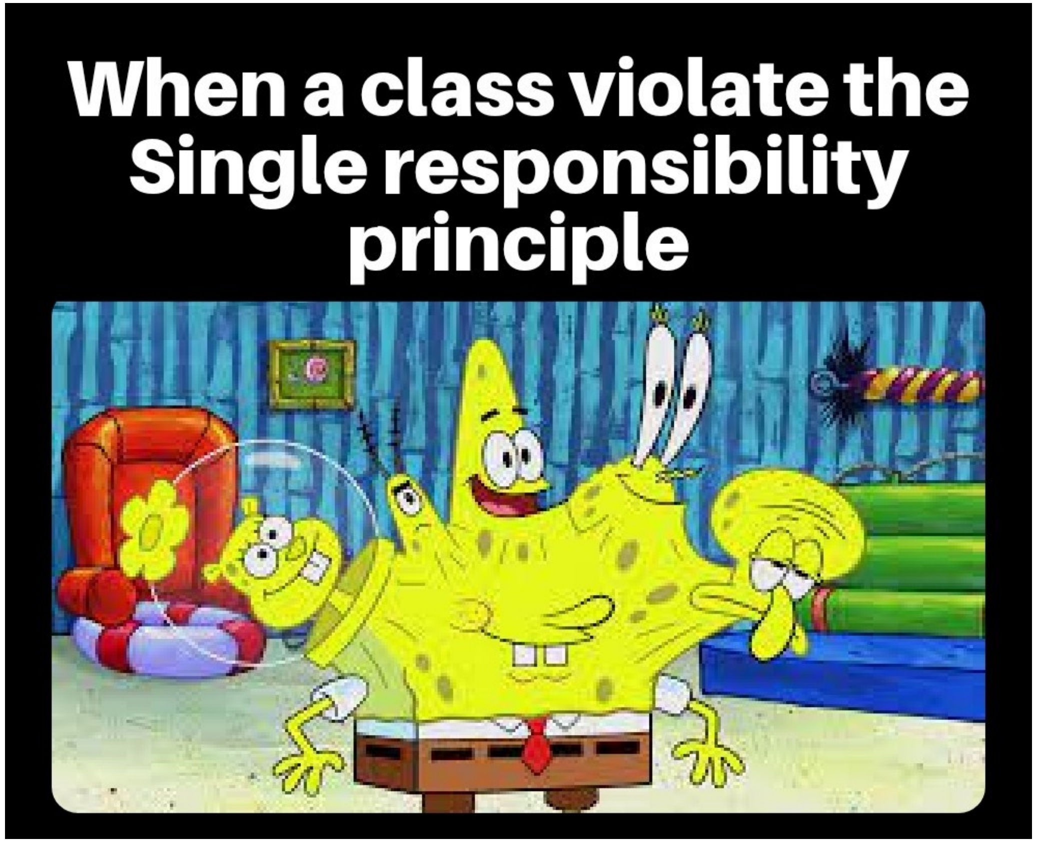
SRP	**Single-Responsibility-Principle** There should only be one reason to change each class.
OCP	**Open-Closed-Principle** Classes should be open for extension but closed for modification.
LSP	**Liskov-Substitution-Principle** Lower classes must be able to take the place of their upper classes.
ISP	**Interface-Segregation-Principle** Clients should not be forced to rely on interfaces they do not use.
DIP	**Dependency-Inversion-Principle** High-level modules should not depend on low-level modules. Both should only depend on abstractions.Abstractions should not depend on details. Details should depend on abstractions.

The vision principle

Every concept (packages, classes, methods) must be describable in an understandable way in one main sentence (the vision).

Single-Responsibility-Principle (SRP)

- A class should only be responsible for one thing! (= a single aspect of a requirement)
- There should never be more than one reason to change a class.
- Consequence:
  - ▸Many small classes

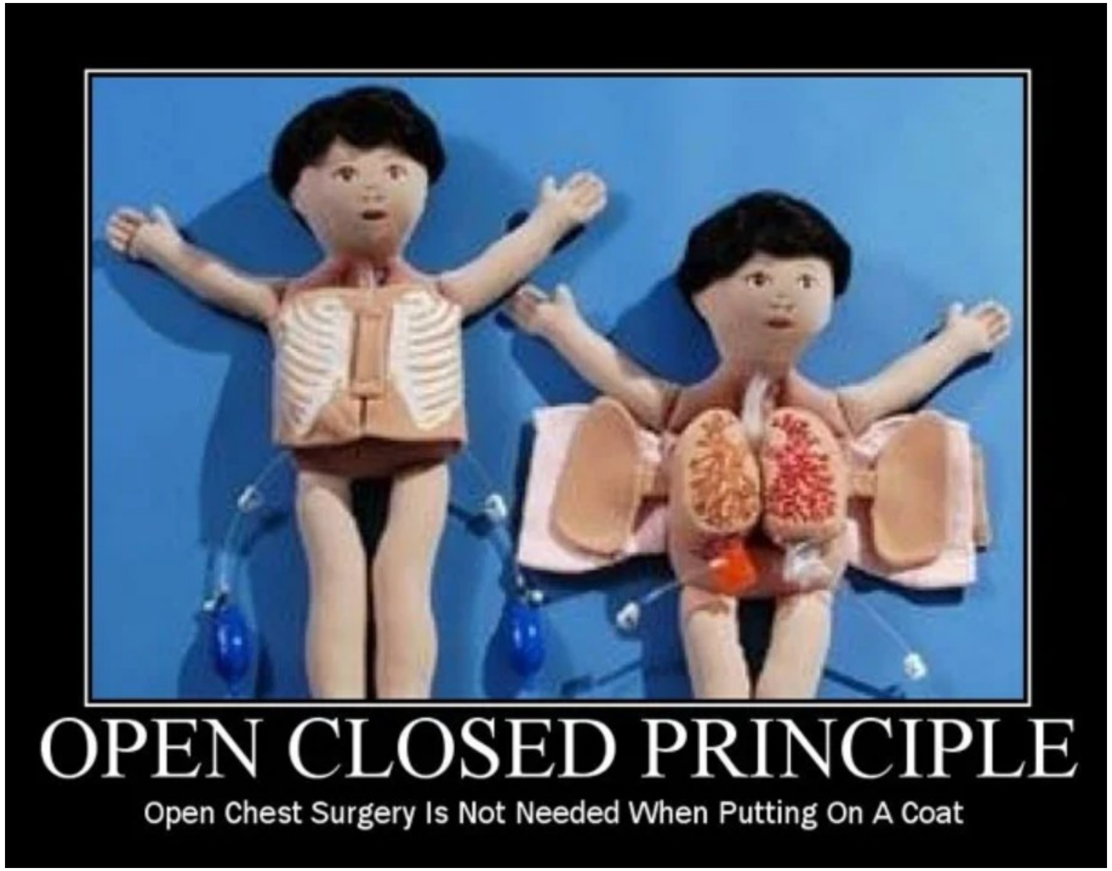


Goal of SRP

Our system consists of a large number of small classes (rather than a few large ones). Each class encapsulates a single responsibility, has only one potential reason for change, and works with a few other classes to map the desired behaviour.

Open-Closed-Principle (OCP)

- Open for extensions
  - ▸Functionality must be able to be added/supplemented by writing subclasses.
- Locked for changes
  - ▸Changes should not result in the behaviour or source code of the class itself being altered.
- Consequence:
  - ▸Create sufficiently abstract superclasses



Rule 3-15

(OCP)

Classes should be open for extension but closed for modification.

Liskov's substitution principle

NimblePros
LISKOV SUBSTITUTION
If it looks like a duck, quacks like a duck, but needs batteries —
you probably have the wrong abstraction.

The circle-ellipse problem

There is a circle class and an ellipse class. From a geometric point of view, the following statement is certainly correct:

A circle is a special type of ellipse (in which both semi-axes are of equal length).

- **Problem:**
  - The methods scaleX() and scaleY() cannot be adequately implemented in the subclass (circle).
  - Polymorphism may therefore not function as expected.

Liskov's substitution principle (LSP)

Wherever the superclass is used, it must also be possible to use an instance of the subclass without hesitation.

or:

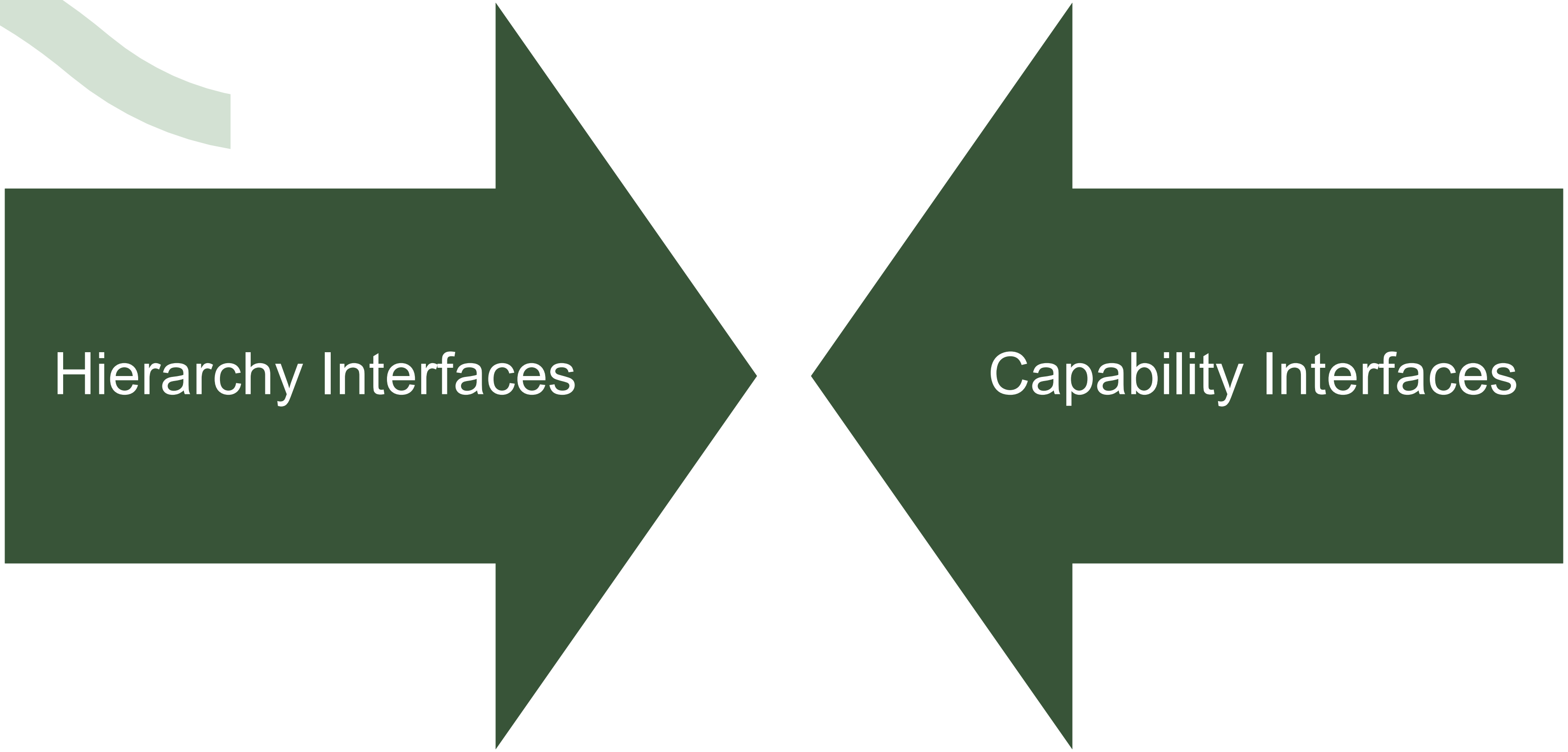
All tests that run correctly on an instance of the superclass must also be able to run correctly on instances of the subclass.

Rule 3-2

(LSP)

Subclasses must be able to take the place of their parent classes.

Interfaces

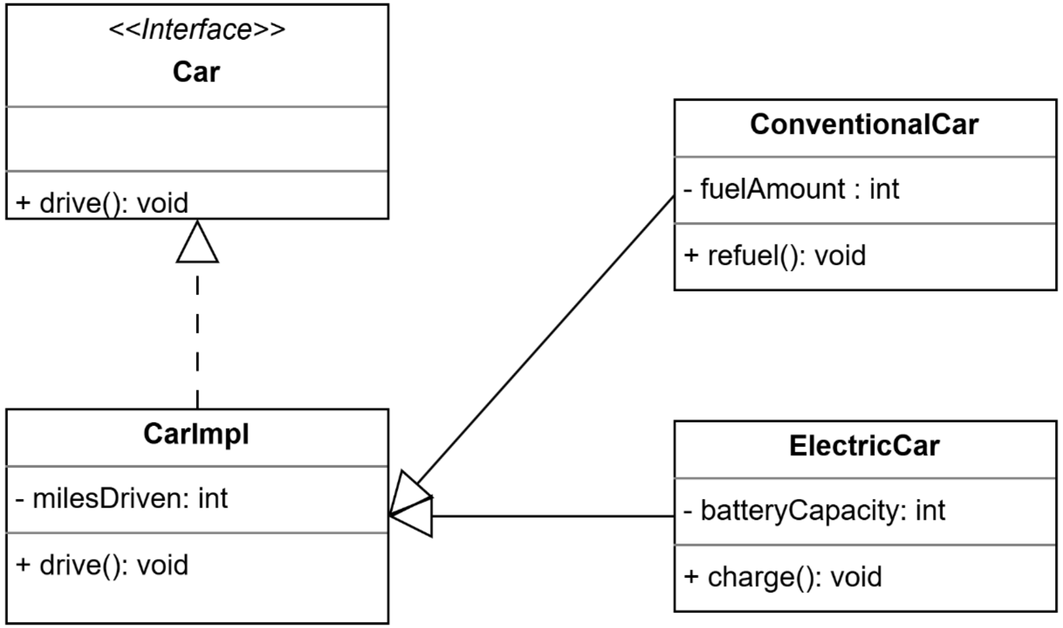


Hierarchy Interfaces

Capability Interfaces

Hierarchy Interfaces

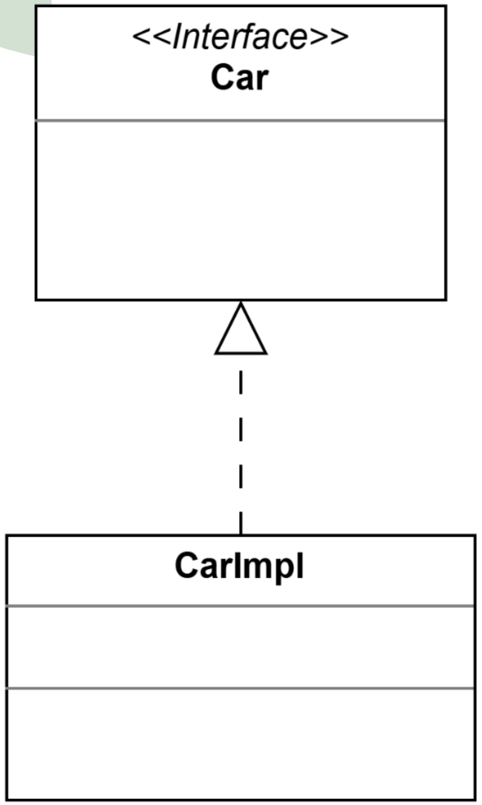
- Interfaces that are only used as "super-abstract" classes also follow the rules for classes.
  - ▸Terminology
  - ▸Naming rules
- Some languages do not have their own interface construct.



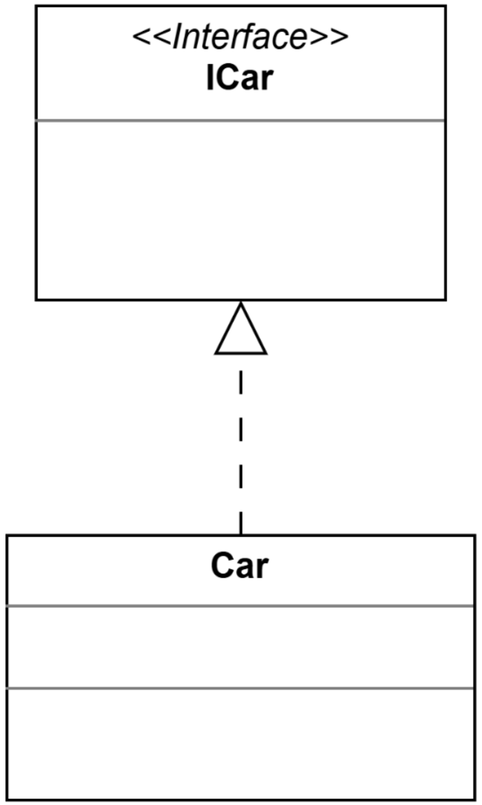
Rule 3-5

A class should always either derive from a superclass or implement a hierarchy interface.

Naming of Hierarchy Interfaces



oder



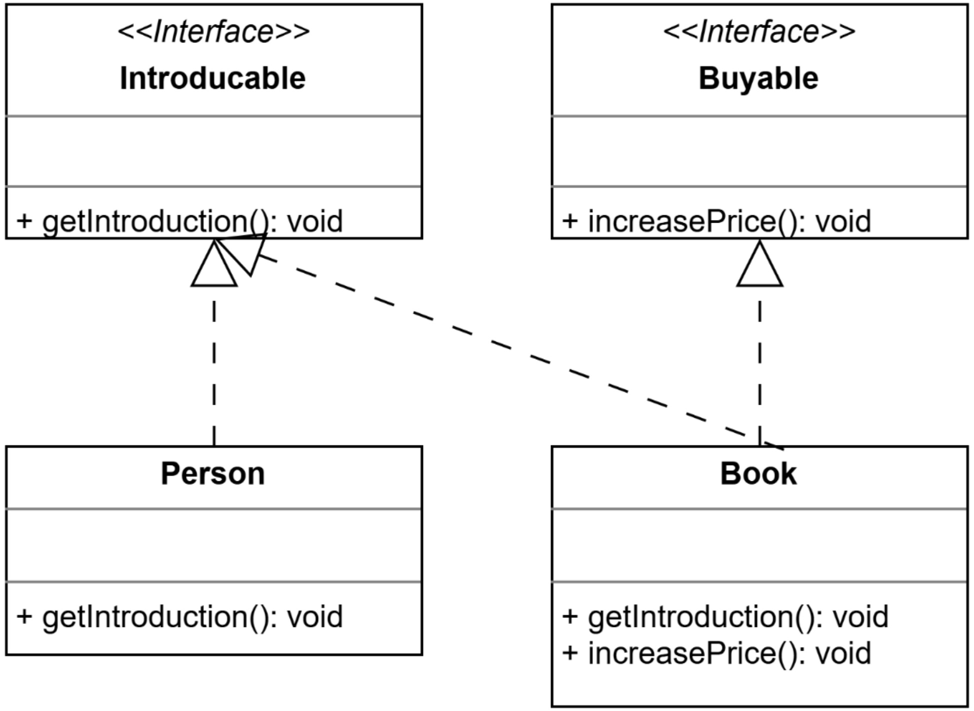
Rule 3-6

The name of the construct (class/interface) that is used most frequently in the code should be the "most beautiful" one.

Capability Interfaces

- Solves the problem with the getIntroduction() method
  - ▸ *A person is introducable.*
  - ▸ *A book is introducable.*

- Usually describes cross-sectional tasks.
- Can also be used polymorphically (the "label" is a capability interface).



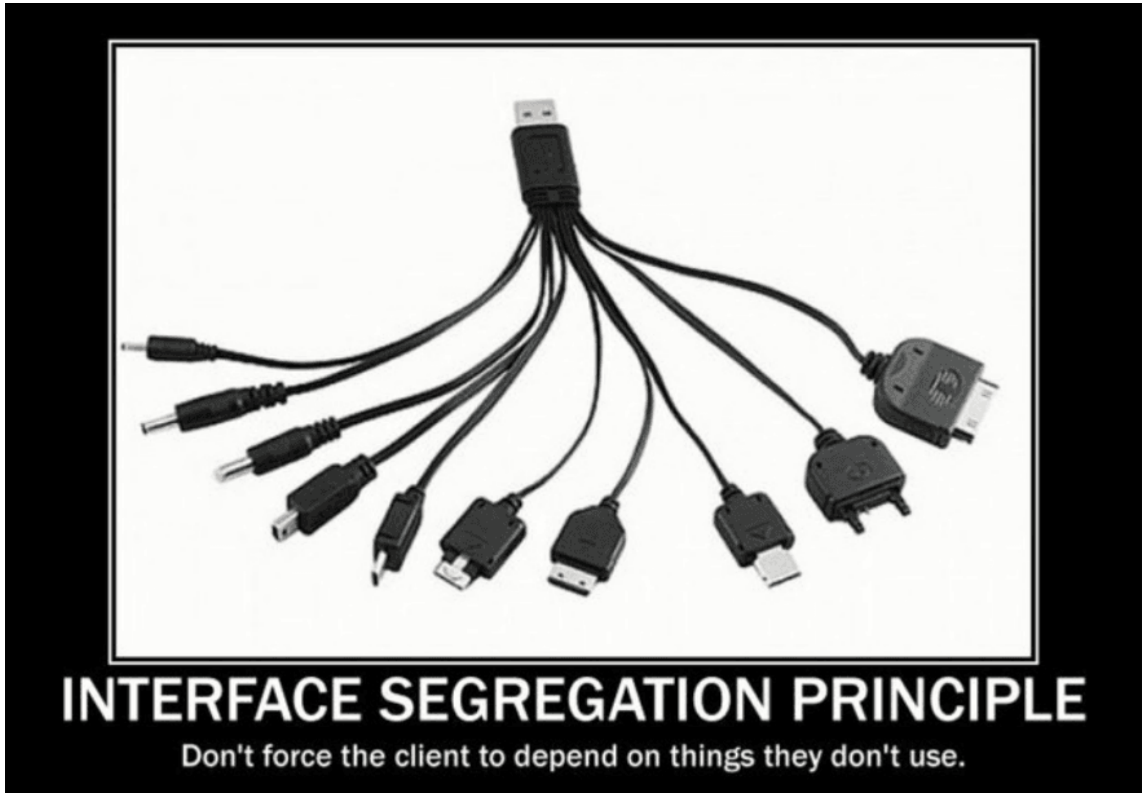
Rule 3-7

Cross-sectional capabilities are implemented via capability interfaces.

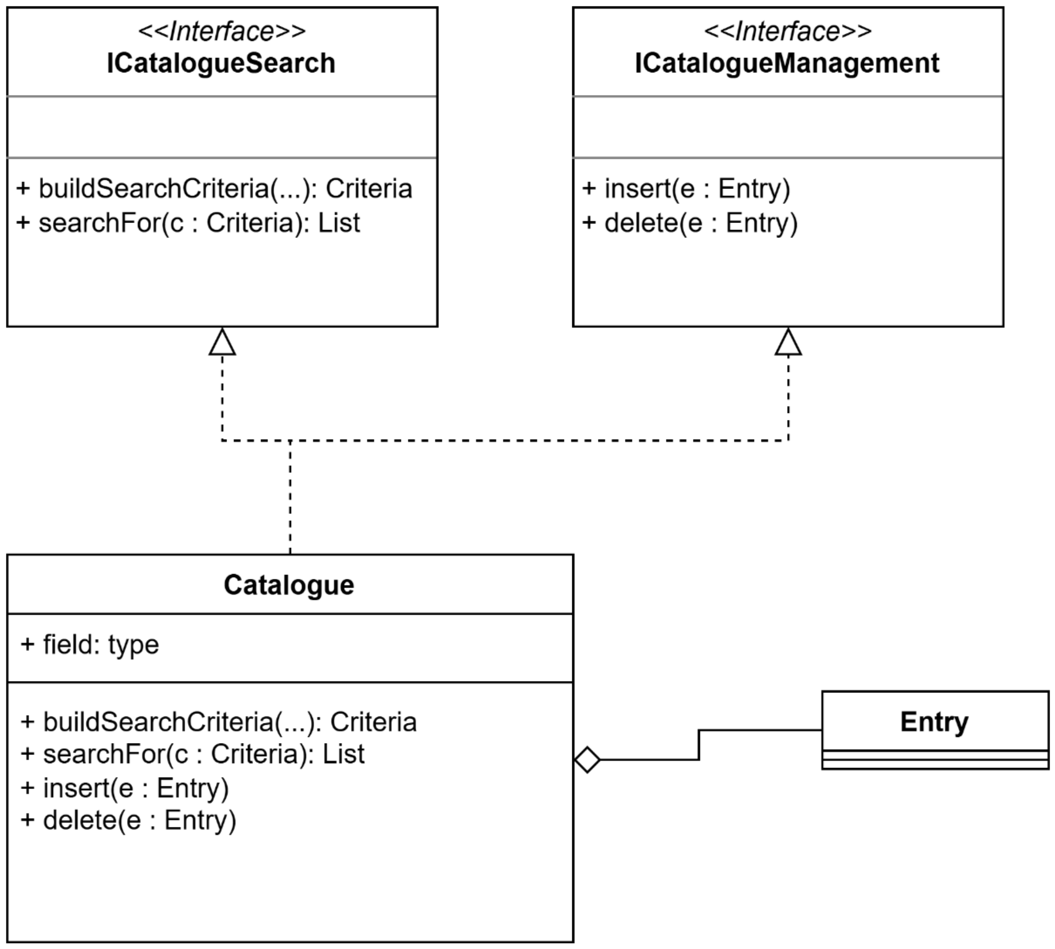
Interface Segregation Principle (ISP)

Clients should not be forced to rely on interfaces they do not use.

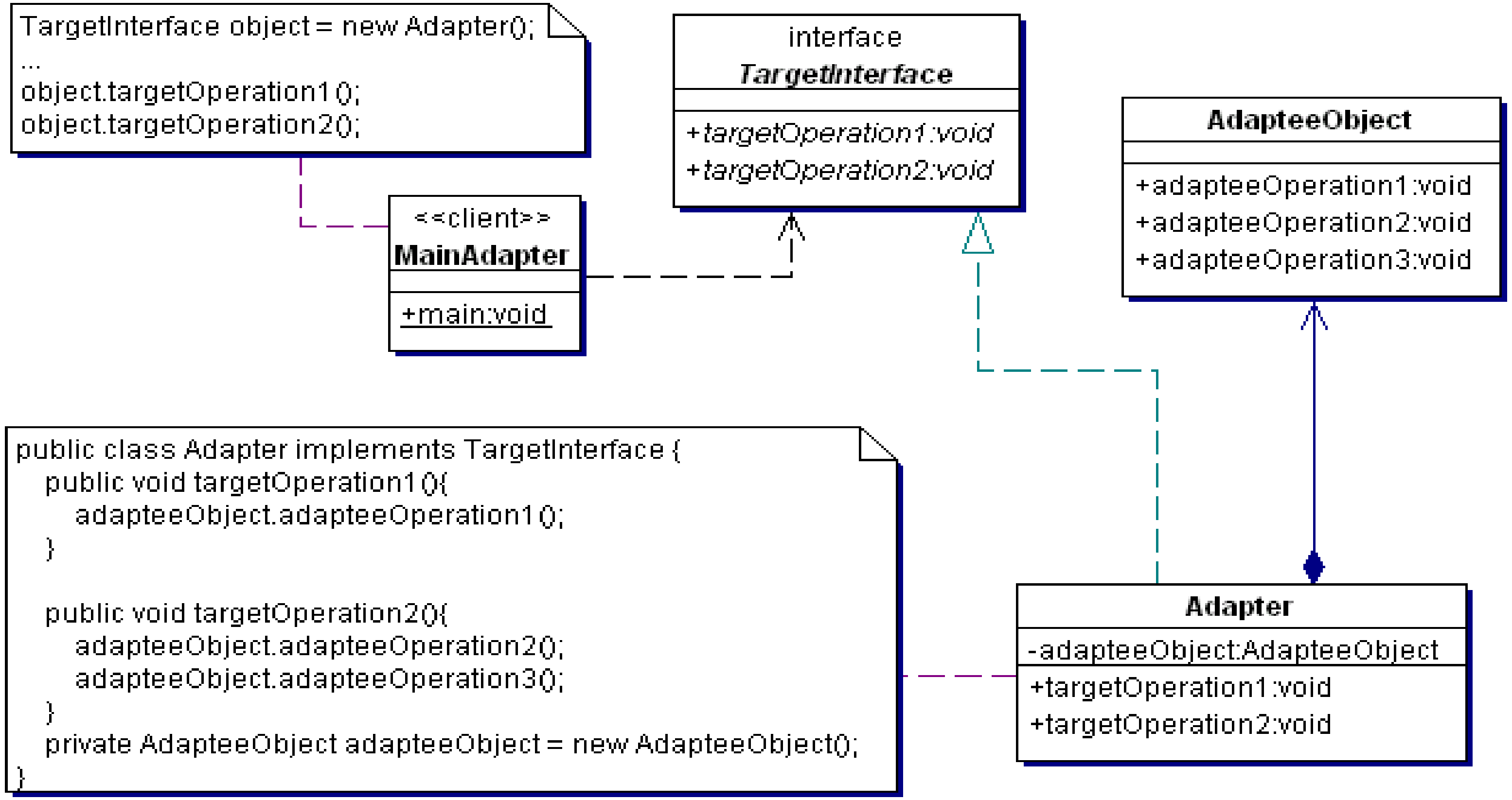
- Solution:
  - ▸various Interfaces
  - ▸Adapter Pattern


INTERFACE SEGREGATION PRINCIPLE
Don't force the client to depend on things they don't use.

ISP: various Interfaces



ISP: object adapter



```
TargetInterface object = new Adapter();
...
object.targetOperation1();
object.targetOperation2();
```

<<client>>
MainAdapter

+main:void

interface
TargetInterface

+*targetOperation1:void*
+*targetOperation2:void*

AdapteeObject

+adapteeOperation1:void
+adapteeOperation2:void
+adapteeOperation3:void

Adapter

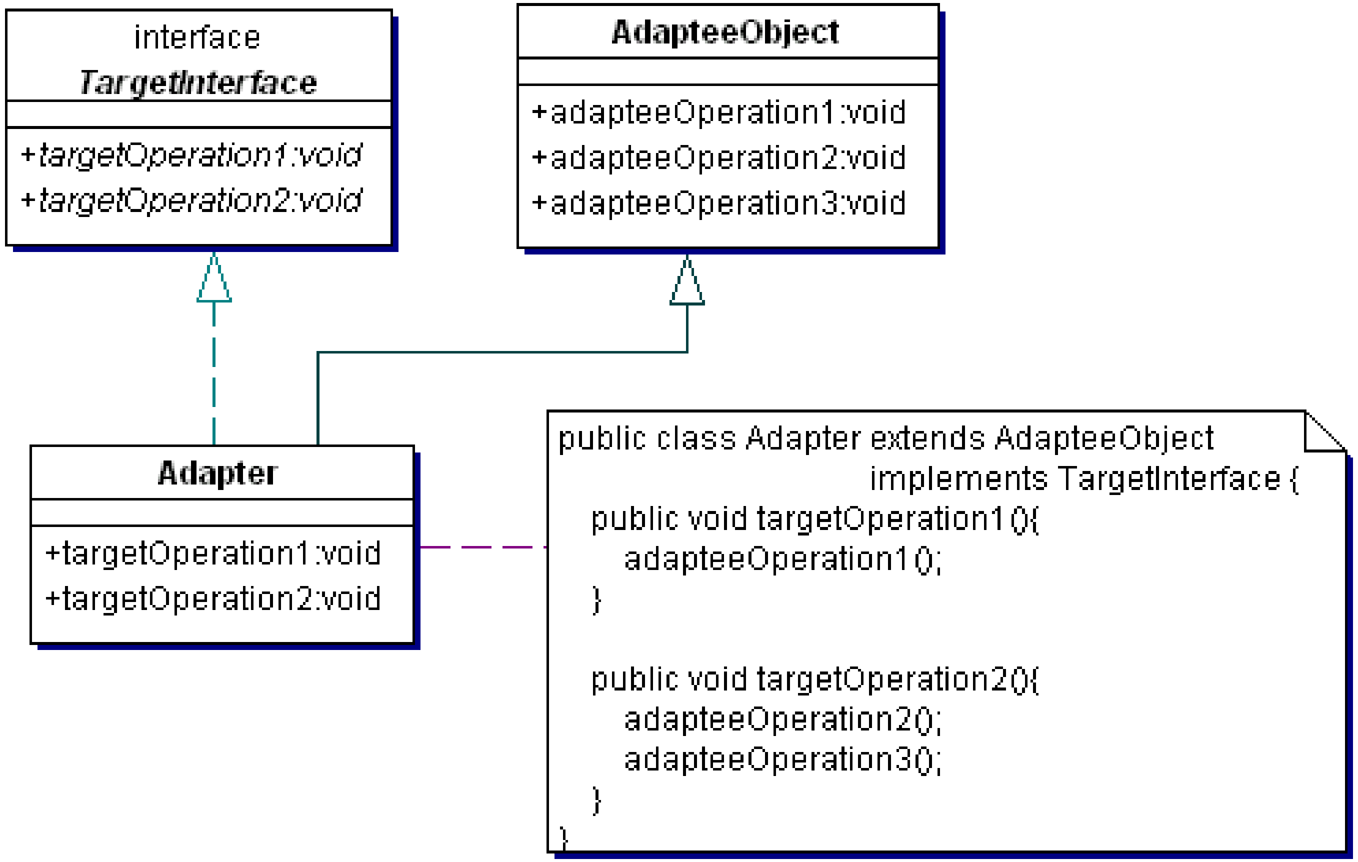
-adapteeObject:AdapteeObject

+targetOperation1:void
+targetOperation2:void

```
public class Adapter implements TargetInterface {
   public void targetOperation1(){
      adapteeObject.adapteeOperation1();
   }

   public void targetOperation2(){
      adapteeObject.adapteeOperation2();
      adapteeObject.adapteeOperation3();
   }
   private AdapteeObject adapteeObject = new AdapteeObject();
}
```

ISP: class adapter (variation)



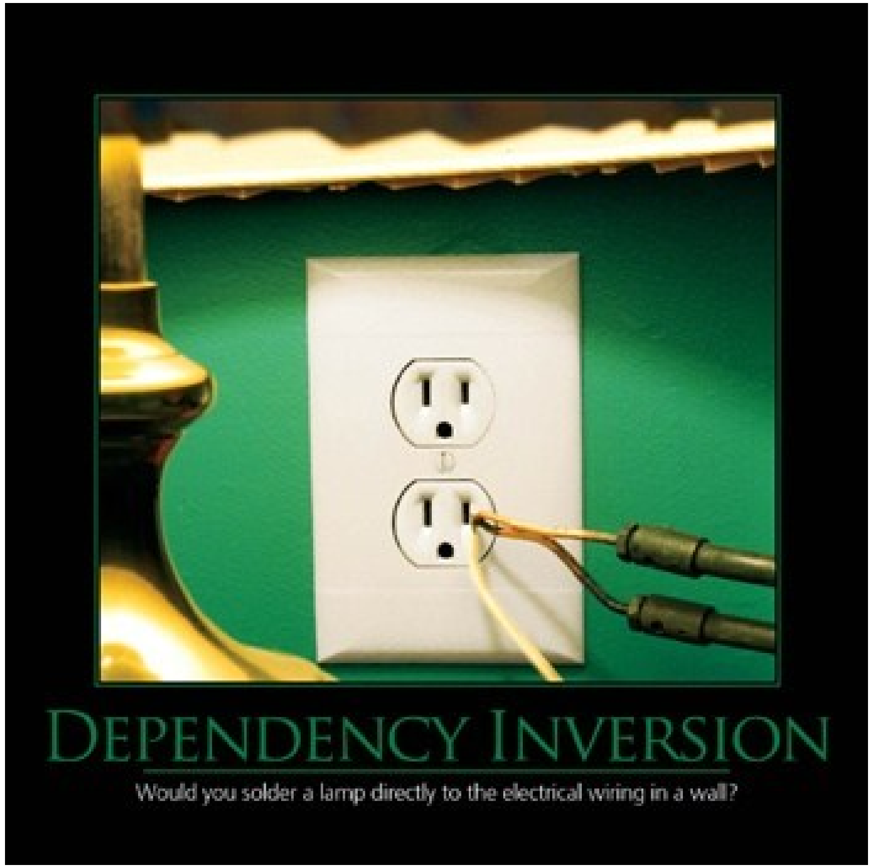
```
public class Adapter extends AdapteeObject
                          implements TargetInterface {
   public void targetOperation1(){
      adapteeOperation1();
   }

   public void targetOperation2(){
      adapteeOperation2();
      adapteeOperation3();
   }
}
```

Dependency Inversion Principle (DIP)

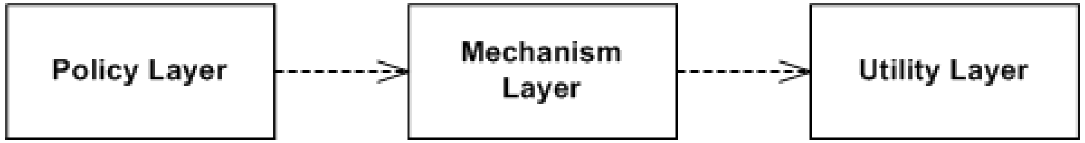
High-level modules should not depend on low-level modules. Both should only depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

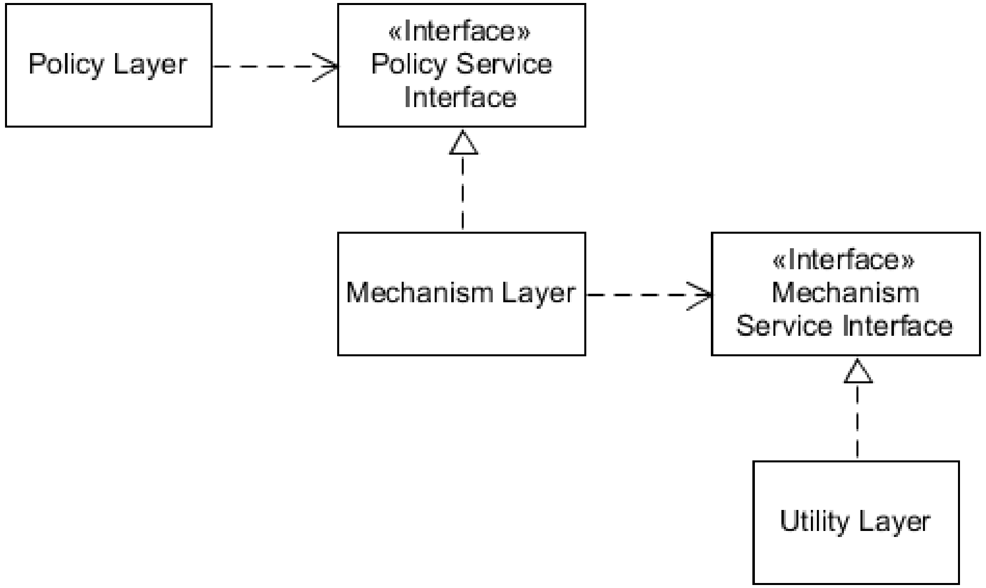

DEPENDENCY INVERSION
Would you solder a lamp directly to the electrical wiring in a wall?

DIP - Solutions

- Traditional layers pattern



- Dependency inversion pattern


Policy Layer
«Interface» Policy Service Interface
Mechanism Layer
«Interface» Mechanism Service Interface
Utility Layer

4. Naming

Which language?

Options for teams in Germany

- German (or the team's native language)
  - Terms closer to the specialist department
  - Less translation work for those who do not know English
  - Difficulties with possible outsourcing
- English
  - Keywords in programming languages are usually in English (if, for, while, get... set...)
  - Risk: Better no comments at all than poor ones.
- Mixed
  - Not everything is easy to read
  - Subsequent translation is unrealistic.

Meaningful names

```cpp
for (int i = 1; i < m; i++) {
    bool b = true;
    for (int j = 2; j < i; j++) {
        if (i % j == 0) {
            b = false;
            break;
        }
    }
    if (b && i > 1) {
        std::cout << i << std::endl;
    }
}
```

Meaningful names

```
for (int possiblePrime = 1; possiblePrime < maxNumber; possiblePrime++) {
    bool isPrime = true;
    for (int possibleDivider = 2; possibleDivider < possiblePrime; possibleDivider++) {
        if (possiblePrime % possibleDivider == 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime && possiblePrime > 1) {
        std::cout << possiblePrime << std::endl;
    }
}
```

Rule 4-1

Variables should have names that reflect their meaning.

Rules for naming

- Classes
  - Nouns (Employee, Person, Document)
  - No "softeners" (Manager, Service, Processor, Data, Info)
- Abstract classes
  - True abstractions (to be used polymorphically)
  - Player is an abstract superclass of Golfer and VideoGamer.
  - Car is an abstract superclass of ConventionalCar and ElectroCar
  - Same rules as for normal classes
  - Technical helper classes (not visible in the client)
    - Should begin with the word Abstract
- Hierarchy interfaces are named like classes
- Capability interfaces are named with adjectives

Rules 4-2,3,4

Classes and abstractions bear the names of (possibly compound) nouns.

Abstract classes used as implementation aids should be prefixed with "Abstract".

Capability interfaces have adjectives as names.

Rules 4-5,6

Methods should have names that are verbs or terms derived from verbs.

Access methods should begin with get, set, or is. Other methods should not use these prefixes.

Names of methods

- If the meaning of an argument is not clear, the meaning can be encoded in the name.
- Clarity in the call to action is crucial
  - `printPrimes(int max)` is understandable
  - `printPrimes(15)` on the client side not really
  - `printPrimesUpTo(15)` should do the job

Constructors

- How can naming be done for constructors?
  - ▸Example: Point class with Cartesian and polar constructors

```
Point upperLeft = Point(10, 20);
Point lowerRight = Point(10f, 20f);
```

- Better (FACTORY-Pattern):

```
Point upperLeft = Point.FromCartesian(10, 20);
Point lowerRight = Point.FromPolar(10f, 20f);
```

- Even better (but with additional effort: BUILDER-Pattern)

```
Point lowerRight =
  Point.buildWith().angle(10f).distance(20f).build();
```

- Or:

```
Point lowerRight =
Point.buildWith().angleOf(10f).and().distanceOf(20f).andReturnIt();
```

Rule 4-7

Unclear constructors should be "named" using factory methods. The constructor itself should then no longer be visible.

Naming (continued)

Naming rules

- Names should be understandable in their context.
- If a variable leaves its context, the context (the "origin") should be encoded in the variable name.

```
Person user = ...;
std::string userName = user.getName();
```

- The longer the context, the more detailed the name should be.

Special names

- Loop variable „i“
  - But only as long as the loop counter has no technical significance.
- „result“ variable as return value for methods
- „it“ and „next“ for iterators

Rule 4-8

A handful of defined standard names make it easier to maintain clarity if all developers are familiar with them.

Result variavles

- How could the avg variable be named better?

  ```
  int avg = averageSalary(employees);
  ```

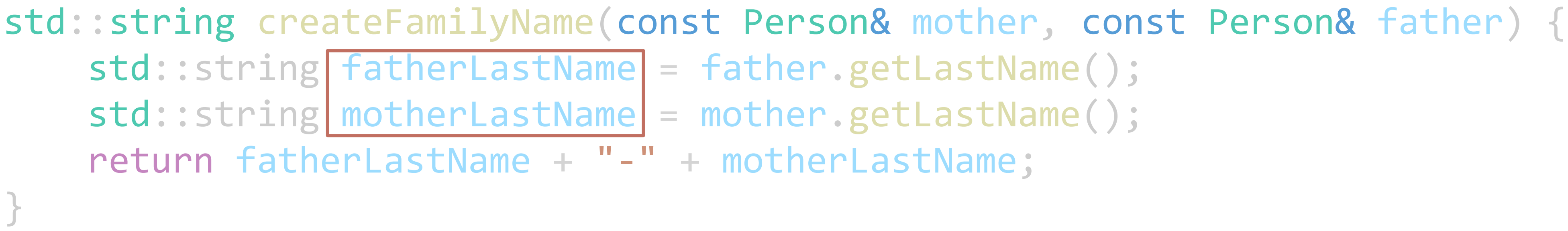
- Ideas:
  - Prefix: averageSalaryOfEmployees
  - Suffix: employeesAverageSalary
  - short: averageSalary
  - short, suffix: salaryAverage
- Extended prefix: `int averageSalaryOfPartTimeWorkers = averageSalary(partTimeWorkers);`

Rule 4-9

Variables that contain the result of a method should bear the name of that method.

Result variables with origin

- It may be necessary to also encode the origin in the variable names. :



```
std::string createFamilyName(const Person& mother, const Person& father) {
    std::string fatherLastName = father.getLastName();
    std::string motherLastName = mother.getLastName();
    return fatherLastName + "-" + motherLastName;
}
```

Bad names

- Misleading names
  - Names that indicate an incorrect concept (e.g. a method setXY that has side effects)

- text noise
  - To satisfy the compiler

```
void addEvenValues(const std::vector<int>& list1, std::vector<int>& list2) {
    for (int next : list1) {
        if (next % 2 == 0) {
            list2.push_back(next);
        }
    }
}
```

  - better: source und destination

Bad names

- Text noise continued:
  - Pointer and Pointr
  - aPoint and thePoint
  - Person and PersonInfo
  - Payment and PaymentObject
- Domains language vs. solution language
  - Domain language is the language of the specialist area:
    - Booking, invoice, item, discount
  - Solution language is the technical language of the programmer:
    - Pattern, list, controller, view

Rules 4-10,11

The differences between two chosen names must be selected in such a way that the reader understands them in terms of content.

Technical concepts should be formulated in domain language, technical details in solution language.

Naming concepts

- Identical concepts should always be described with the same word.
  - Get, retrieve, and fetch should have consistent, distinguishable behaviour.
- Different concepts should also use different words.
  - retrieve should not return an object in one method and delete and return it in another.
- Related concepts usually have word pairs

add/remove	insert/delete	begin/end	lock/unlock
show/hide	create/destroy	source/target	start/stop
min/max	next/previous	open/close	old/new
first/last	up/down	get/set	get/put

Rules 4-12,13

The same concepts should be described by the same word, different concepts by different words.

Related concepts should also be described using related terms.

Naming continued…

- Names should follow the conventions of the programming language.
- Optical confusion due to interchangeable characters (lowercase L, uppercase O)

```
int a = l;
if (O == 1)
    a = O1;
else
    l = 01;
```

- Slightly different names are misleading:
  - XYZControllerForEfficientHandlingOfStrings
  - XYZControllerForEfficientStorageOfStrings

Pronounceable names

- The following code is relatively easy to understand:

```
int mxNoPts = …;
while (ptList.size() > mxNoPts) {
    Point rmvd = ptList.back();
    sprList.push_back(rmvd);
}
```

- But better:

```
int maxNumberOfPoints = …;
while (pointList.size() > maxNumberOfPoints) {
    Point removed = pointList.back();
    spareList.push_back(removed);
}
```

- Negativ „worst“ example (really seen in production!):

```
gaSuspSvcWOregLstnr()
```

Bad naming

- Encodings:
  - Hungary notation:
    - iLength
    - sName
  - Hungary notation detailled:
    - nameString
    - sizeInt
  - Context encodings:
    - pNumber (for parameters)
    - fSize (for fields)
- Puns and slang
  - killThemAll()
  - bigBang()
  - call911()
  - insertB4() (instead of insertBefore())

Rules 4-14,15,16

Names should be visually different enough that they can be distinguished at a glance.

Names should be pronounceable. Abbreviations should only be used in exceptional cases, and even then only those that can be spoken.

Encodings for types and contexts should not be used.

If you need to fight for the time to refactor names:

- Names are easy and simple to change
- Memory requirements for longer names are generally negligible.
- Names that no longer fit should be changed immediately.
  - But of course, caution is advised when it comes to interfaces.
- Thanks to code completion in modern IDEs, a long name only needs to be written once – after that, the IDE completes it for you.

A project style guide is crucial for readable code!!!

5. Methods

Terms

- Operation: The interface/signature/capability
- Method: The actual implementation
- Function: A method that returns a value
- Procedure: A method that does not return a value (and thus has a side effect)
- Routine: Term from the pre-OO era

Types of methods

- Interface methods
  - Part of the public / protected interface of a class
- Private helper methods

Rule 5-1

Methods shall be small

What is „small“?

- How big shall a method be?
  - Upper limit: One full size screen
  - at a resultion of 1920x1200 and font size 8?
  - Rule of thumb: 20 lines

- The Hrair-Limit
  - Psychological concept
  - A person can only process a maximum of 7+/-2 concepts at the same time.
  - Anything above that is grouped subconsciously.
  - The exact number is a predisposition.
  - The term "Hrair" is an homage by Grady Booch to the novel "Watership Down" by Richard Adams.

Rule 5-3

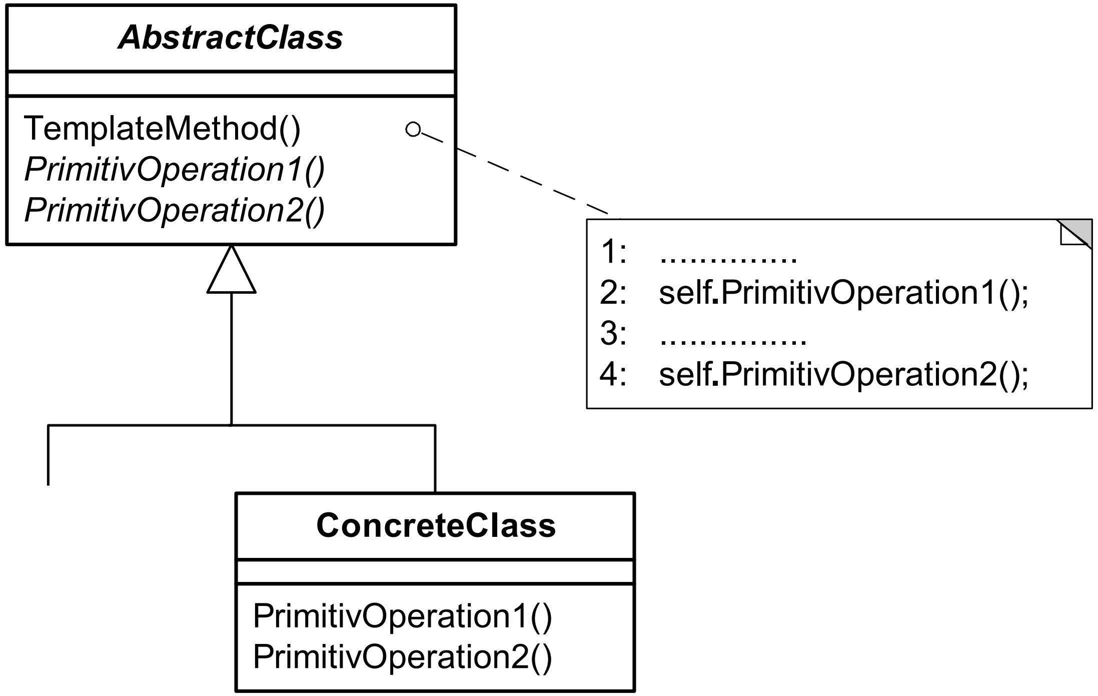
Methods should comply with the Hrair limit (no more than 7 lines)

Extreme cases

- Oneliners
  - Translation between technical and specialist methods
    - registerUser(user) statt userList.add(user)
  - Clarification of a confusing calculation
- Zero-liners
  - Only make sense in inheritance
  - The superclass should contain the empty method, the subclasses should then contain code
  - The reverse would be a violation of LSP
  - Zero-line methods should contain a comment explaining why they are empty

TEMPLATE-Pattern

A template method is a method that does not represent a complete implementation, but rather a "template-like" implementation, i.e. it specifies a sequence of steps (flow control), but leaves the content of the individual steps to the subclasses.



Rule 5-4

Blocks should be single-line.

Single-line blocks / Mikro-Methoden

```cpp
for (int possiblePrime = 1; possiblePrime < maxNumber; possiblePrime++) {
    bool isPrime = true;
    for (int possibleDivider = 2; possibleDivider < possiblePrime; possibleDivider++) {
        if (possiblePrime % possibleDivider == 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime && possiblePrime > 1) {
        std::cout << possiblePrime << std::endl;
    }
}
```

Single-line blocks / Mikro-Methoden

```
bool isDividerOf(int divider, int number) {
    return number % divider == 0;
}

bool isPrime(int number) {
    if (number < 2) return false; // 1 und kleinere Zahlen!
    for (int i = 2; i < number; i++) {
        if (isDividerOf(i, number)) return false;
    }
    return true;
}

void printIfPrime(int possiblePrime) {
    if (isPrime(possiblePrime)) {
        std::cout << possiblePrime << std::endl;
    }
}

void printPrimesUpTo(int maxNumber) {
    for (int i = 1; i < maxNumber; i++) {
        printIfPrime(i);
    }
}
```

Names of methods

- A verb or a verb with nouns
  - void store(Order order) is understandable.

- On Client side?
  - backend.store(priority);
- besser:
  - backend.storeOrder(priority);

Rule 5-5

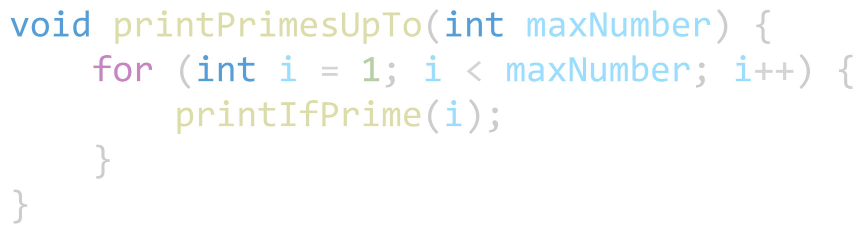
The name of a method must be understandable on the client side, together with its arguments.

Rule 5-6

A method should do one thing. It should do it well. It should do only that one thing.

One task per method

- How many things is this method doing?



```
void printPrimesUpTo(int maxNumber) {
    for (int i = 1; i < maxNumber; i++) {
        printIfPrime(i);
    }
}
```

- Option 1: 2 things
  - ▸Go through all numbers from 1 to maxNumber
  - ▸Output each number if it is a prime number
- Option 2: 1 thing
  - ▸Output all prime numbers up to maxNumber.
- Option 2 describes what the method does, option 1 how it does it.
- Both variants together:
  - ▸ *To **printPrimesUpTo maxNumber**, count from 1 to **maxNumber** and for each Number, **printIfPrime**.*

The Stepdown-rule

- By nesting the levels of abstraction, you can break down the public method's sequence in detail:
  - To printPrimesUpTo maxNumber, count from 1 to maxNumber and for each Number, printIfPrime.
    - *To printIfPrime, we check if the number isPrime and if so, print it on the console.*
      - *To check if a number isPrime, we check for all numbers between 2 and the number it isDividerOf number. If so, it is no Prime (false). Else, it isPrime.*
- When reading, you can decide at any time to stop at a certain level.

Number of arguments

- Niladic Methods(= 0 parameters)
  - Super easy to read
    - `before:  printResultInto(writer);`
    - `after: printResult();`
  - This is achieved by converting the parameter into a field that is set beforehand.
- Monadic Methods(= 1 parameter)
  - The individual argument takes on the role of the object in the "sentence" formed by the name of the method.
    - `printIfPrime(myNumber)`
    - `openFile("data.txt")`

Main types of monads

- queries
  - ▸provide information on the argument
    - `List.contains(Object o)`
    - `String.indexOf("Hallo")`
- transformers
  - ▸convert the argument into another one
    - `InputStream openFile("data.txt")`
    - `List.get(15)`
- Events
  - ▸change the state of the object/system
  - ▸Most prominent example: setter
- Hybrid forms (a setter that also returns something) are more difficult to read.

Rule 5-9

A monadic method should always be a query, a transformer, or an event.

Dyadic methods

- Dyadic methods (= 2 parameters)
  - Are more difficult to grasp than monads
  - Require care with the order of arguments (example: assertEquals())
  - Easily lead to arguments being ignored
  - Are more difficult to test, as there are more possible combinations that need to be tested

„ The places we ignore are the places where bugs hide.“

Robert C. Martin

Triadic methods

- Are difficult to overview
- Are even more difficult to test in detail
- Require prior knowledge or time investment on the part of the reader
- Should be reduced where possible by using argument objects or instance variables

More than 3 arguments

- Extreme example:

```
GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double
weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady)
```

- The call is even worse

```
new GridBagConstraint(5, 10, 1, 1, 1.0, 2.0, CENTER, BOTH, emptyInsets, 1, 1)
```

- Only chance to use that: Comments!

```
new GridBagConstraint(
  5, 10, // grid position
  1, 1,  // cell size
  1.0, 2.0, // weight
  CENTER, // anchor
  BOTH, // fill
  emptyInsets,
  1, 1) padding
```

Flag-Methods

- Flag methods are methods with a Boolean argument. :

```
void paint(boolean isSelected)
```

▶It may be understandable, but in the call?

```
paint(true)
```

▶Better in two separate methods:

```
void paintAsSelected()

void paintAsUnselected()
```

▶Or with an enumeration:

```
paint(SELECTED);

paint(UNSELCTED);
```

Return parameters

- Output parameters, i.e. parameters that are changed by the method, significantly impair understanding.

```
…
DataBasket basket = …
basket.addToList(orders);
…
```

- Who is writing in which list here?
- Even worse if the parameters is (unnecessarily) returned

```
…
DataBasket basket = …
orders = basket.addToList(orders);
…
```

- Of course, this is not possible if more than one output parameter is used.

Rule 5-10,11

Flag methods should be avoided, especially with monads.

Methods should have at most one output argument, which should be duplicated as a return value.

Objects as arguments

```
public Rectangle(int left, int top, int right, int bottom);
public Rectangle(Point upperLeft, Point lowerRight);
```

- On the callers side:

```
new Rectangle(0, 0, 10, 10)
new Rectangle(new Point(0, 0), new Point(10, 10))
```

- When the point-parameters are created before this gets readable a lot better:

```
new Rectangle(origin, lowerRight)
```

Style

- Side effects:
  - Side effects are effects of a method that change the state or trigger actions in the system.
  - The method name should always clearly indicate whether a method has side effects or not.
- Command Query Separation
  - A method should either return something or do something (change something).
  - Bad example: `if (knownNames.add("Peter")) ...`
- Multiple Exit points
  - Acceptable in short methods
  - Can be highlighted by the IDE
- Recursion
  - Short and smart
  - but only understandable with considerable effort

6. Comments and documentation

The big problem here…

```
/**
 * Add a new Action to the manager. Returns true if the action
 * is already existant. If the action is already registered,
 * it is NOT replaced.
 * @param action the action to add
 * @return True if an action with the same name has
 * already been added, false otherwise.
 */
public void addAction(JaspiraAction action)
```

- Kommentare veralten schnell und werden nicht immer angepasst.

Readable code

- It is better to make the code readable than to add comments.

```
// is order eligible for free shipping
if (order.getValue() > FREE_SHIPPING_LIMIT
     || isPremiumMember(order.getCustomer()))
```

- Or even better:

```
if (isEligibleForFreeShipping(order))
```

- Than the following should not be happening:

```
// is order eligible for free shipping
Customer customer = order.getCustomer();
customer.addToOrderHistory(order);

if (order.getValue() > FREE_SHIPPING_LIMIT
     || isPremiumMember(order.getCustomer()))
```

Rule 6-1

Before adding a comment to explain code, you should always try to make the code itself more understandable first.

Good comments

- Legal notices

```
/*
 * (c) 2009, 2010 by Integrata AG, all rights reserved
 * Release under Apache License 1.0
 */
```

- Clarifications

```
Pattern dateTimePattern = Pattern.compile(
    //   31  .   08   .  2004              16     :  28   +1
    //    1  .    4   .  04                4      :  15
    "\\d{1,2}.\\d{1,2}.((\\d{4})|(\\d{2}) \\d{1,2}:\\d{2} (+\\d)?")
```

- Intentions: Why was that code written in a certain way?

Good comments

- Design Patterns: (Unless the name of the pattern is already part of the class/method name.)

```
/**
 * Provides Singleton access to the DataBase Layer.
 * …
 */
public class DataBaseControl {
```

- Rule violations: Indication that a rule/convention has been violated, which one and why

- Underlining: To highlight an essential step in the code that might otherwise be overlooked

```
members.clear() // necessary for Garbage Collection to reclaim members
```

Good comments

– Formale Kommentare

- ▸JavaDoc, Doxygen, POD, i.e. formal formats from which documentation can be generated automatically
- ▸However, JavaDoc is a poor example with its HTML syntax:

```
/**
 * Returns an XML-Representation of this MemberList.
 * Generated Code has the following format:
 *
 * &lt;members&gt;<br/>
 * &nbsp;&lt;person&gt;<br/>
 * &nbsp;&nbsp;&lt;firstname&gt;Dieter&lt;/firstname&gt;<br/>
 * &nbsp;&nbsp;&lt;lastname&gt;Maier&lt;/lastname&gt;<br/>
 * &nbsp;&nbsp;&lt;birthday&gt;1973-15-21&lt;/birthday&gt;<br/>
 * &nbsp;&lt;/person&gt;<br/>
 * &lt;/members&gt;<br/>
 */
public void toXML() {
```

Rule 6-2,3,4

Rule violations must be marked with a comment and justified.

Formal comments should follow the vision principle.

Formal comments should be readable in the source code.

Bad comments

- Incomprehensible comments:
  - sentence structure that distorts the meaning or a half-finished sentence.

- Redundancies:

```
// Register implementations in the service registry
registerServices(services);

// Initialize the persistence layer
initPersistence();

// Reads all models
readModels();

// Initializes advanced of the system
initServices(services2);

// Register a shutdown hook that allows correct database shutdown
registerShutdownHook();

// Initializes the remote services (if necessary).
initRemoting();
```

Bad comments

- Forced comments
  - Comments that are only written because a guideline requires it ("Every method must be commented").
  - Requirement to comment on getters and setters.

- History comments (Git is doing that job!)

```
/*
 * Demo.java
 *
 * 18.03.03 sp Initial Version
 * 15.04.03 sp added Lifecyclemethods
 * 18.04.03 jf implemented Comparable
 * 30.05.03 sp general Refactoring
 */
```

Bad comments

- Bracket comments

```
for (int i = 0; i < max; i++) {
    try {
    …
    } // try
    catch (IOException e) {
    …
    } // catch
} // for
```

- Old code

- Unnecessary information
  - Historical information about an algorithm is out of place.

Bad comments

- TODOs
  - Comments that a programmer uses to indicate that something still needs to be done here.
  - They are not bad per se if they are quickly removed.
  - However, reality shows that these comments are rarely ever revisited.
  - They should not be used as a "second ticket system".

- Formal private comments
  - Adding formal JavaDoc to private methods takes time and space.

Test cases as documentation

- Test cases are an alternative to conventional documentation.
- They demonstrate an API in use and can therefore also serve as a tutorial.
- Unlike normal documentation, they are regularly checked for accuracy.

Conclusion

- Comments are not a justification for poor code.
- A comment that is only written because convention requires it is unnecessary – in this case, the convention should be changed.
- When it comes to comments, less is definitely more.

7. Code formatting

Why formatting?

- Code appears "seamless“
- It is easier to navigate code that is formatted in the usual way
- If formatting varies (especially if it is automatic) among developers, a commit to source code management can display countless changes that are not actually changes
- Source code is communication!

The newspaper metaphor

A lesson should be structured like a newspaper article.

- Headline
  - ▸Provides the topic of the article.
  - ▸Is the first deciding factor in whether it is worth reading further.
  - ▸Corresponds to the lesson name.
- Subtitle
  - ▸A single sentence that specifies the content more precisely.
  - ▸Corresponds to the class vision.
- The introduction/lead
  - ▸In newspaper articles, the first part in bold.
  - ▸Provides a more detailed overview and summarises the article roughly.
  - ▸Corresponds to the formal class commentary.

The newspaper metaphor

- Paragraphs
  - A series of logically connected sentences
  - Separated from each other by blank lines (or half lines, if necessary)
  - Corresponds to methods
- Sequences
  - The sequence of methods is also based on a newspaper article
  - From a general overview, you slowly move into detail
  - The eye should only have to flow from top to bottom
  - Related items should be close together

Rules 7-1..5

Methods should be separated from each other by a blank line.

More abstract methods come before more specific methods.

Dependent methods should be placed close together, with the caller (the more abstract one) above the called method.

Conceptually related methods should be placed close together.

Fields should be defined before methods, constants before fields

Class template:

- Constants
- Fields
- Interface methods
- Abstraction methods
- Getters/setters

Sizes

- Classes should generally stay within 100-200 lines.
  - This is not a hard limit.
  - More than 500 lines should be an exception.
- The width should be based on 80 characters.
  - This allows for reasonable printing and convenient viewing of two files side by side on a monitor.

Further considerations

- Indentation is important for readability.
  - Whether tabs or spaces are used should be irrelevant.
- Exceptions to code formatting should only be used if the code formatter can handle them.
  - It is pointless to argue about the position of curly brackets.

Rules 7-6..7

The only correct formatting style is the one specified by the team.

What is important is not which formatting rules are used in detail, but that these rules exist and are used by everyone.

8. Clean Code – Special Topics

Potential topics

- Testing
- Git best Practices
- Hands on: Code-Review 🤓

9. Software metrics

Why metrics?

- Code Entropy
  - ▸Code degenerates over time
  - ▸it deviates from the ideal, intrinsic quality characteristics
  - ▸Metrics attempt to make entropy measurable
- Timeline
  - ▸Plotting metrics on a timeline allows quality issues to be identified at an early stage.
- Automation
  - ▸Metrics must be generated automatically, otherwise they are of little use.

Cyclomatic Complexity (CC)

- Provides an assessment of how complex a method/class is
- A method has a CC of 1 + 1 for each decision point.
  - Decision points are if, for, while, and case branches in a switch statement.
  - For the extended CC (CC'), Boolean short-circuit operators (&& and ||) are also evaluated as decision points.

```
public void countCC() {
    if ( c1() ) // +1
        f1();
    else
        f2();

    if ( c2() ) // +1
        f3();
    else
        f4();
}
```

  - => CC is 3

Cyclomatic Complexity

- The CC provides a good approach to how well a method needs to be tested.
- In order for it to be fully tested, there must be at least as many test cases as there are CCs.
- The CC should not exceed 10, or 20 in exceptional cases.
- With the techniques of method splitting (Hair Limit), the CC will rarely exceed 2 or 3.
- CC provides a good indicator for refactoring rounds.

Metrics for software size

- Lines of Code (LOC)
  - Unsuitable as a metric
  - Only serves to provide an estimate in the timeline of how quickly the code is growing or how successful refactoring has been
- Non Commenting Source Statements (NCSS)
  - Counts statements rather than lines
  - Provides a better result than LOC

Object-oriented metrics

- Weighted Methods per Class (WMC)
  - A summary of the complexity of all methods in a class (usually the CC of the entire class)
  - Provides conclusions about maintainability
  - A useful consideration over time is the average value

- Depth of Inheritence Tree (DIT)
  - Number of ancestors of a class
    - In Java at least 1
    - In C++ a DIT of 0 is possible
  - Provides an assessment of the reusability of the class.
    - The more ancestors, the more specialised the class.
    - This means it is less reusable.

Object-oriented metrics

- Number of Children (NOC)
  - Number of direct and indirect subclasses of this class.
  - Direct measure of the importance of the class.
- Coupling between Object Classes (CBO)
  - How many other classes is this class coupled with?
    - In this case, coupling means that the class accesses methods or instance variables of the other class.
  - Provides an indicator of a class's reusability.
    - The higher it is, the more additional dependencies this class has.
- Response for a Class (RFC)
  - Indicates how many methods can be accessed from a class.
    - Own methods.
    - Methods called from the own method
  - Provides information about the complexity of a class.

evaluation

- Trends (timeline) are more important than absolute values.
- If strict limits are to be set, they should be fairly generous.
- A sensible way of setting limits is to define a series of criteria and only consider a breach of several criteria to be a problem.
- Example:
  - WMC > 100 (weighted methods per class)
  - CBO > 5 (coupling between class objects)
  - RFC > 100 (response for a class)
  - NOM > 40 (number of methods)
  - RFC > 5 x NOM

Static analysis tools

- Static analysis tools are tools that find violations of conventions and best practices.
- The number of violations is again a meaningful metric (Number of Rule Violations – NRV).
- Examples: FindBugs, PMD, Checkstyle, CppCheck

Runtime metrics

- Runtime metrics are metrics for which the code must be executed.
- Test coverage
  - ▸determines the percentage of code reached by tests.
  - ▸Absolute numbers are rarely useful here.
- Build-duration
  - ▸The duration of the build process itself is an important metric for identifying problems in the development environment at an early stage.

Rule 9-1

The metrics used must be understood and evaluated regularly.

Copyright und Impressum

© Cegos Integrata GmbH

Cegos Integrata GmbH
Zettachring 4
70567 Stuttgart

Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.

Gender Disclaimer