Karlsruhe Institute of Technology

Software Engineering II

Prof. Dr. Ralf H. Reussner

Topic 2

Clean Code

DSIS – DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
KASTEL – INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY

dsis.kastel.kit.edu

Overview on Today's Lecture

Content

- Motivation
- Good Coding Practices
- A brief look into Refactoring (if time permits)

Learning Goals

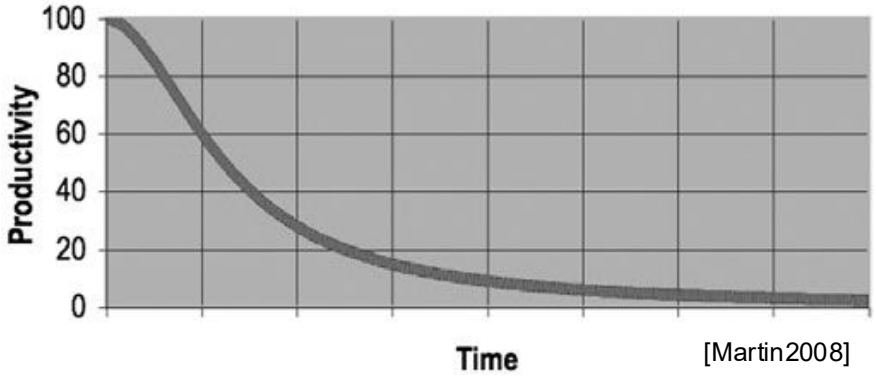
- get acquainted with best practices for coding
- reflect your personal coding habits

Motivation I

- There will always be code!

Bad code exponentially decreases productivity:

- „One broken window starts the process towards decay. “



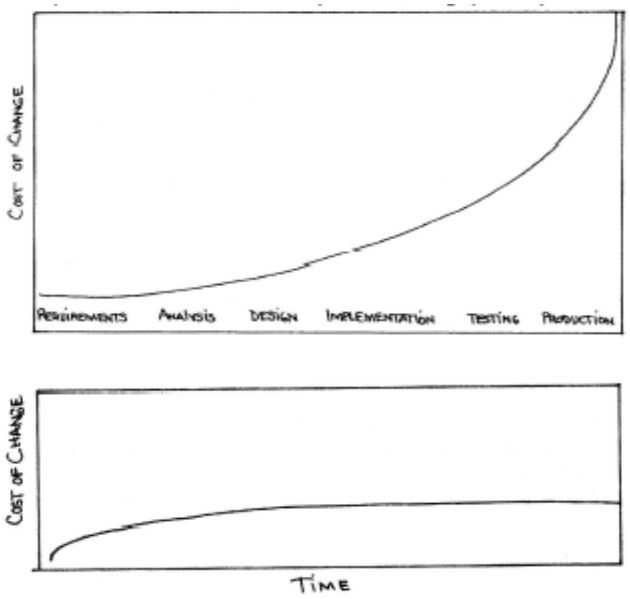
[Martin2008]

Readability of code is important:

- code is much more often read than written
- You write code for the next human to read, not for the compiler/interpreter/computer!

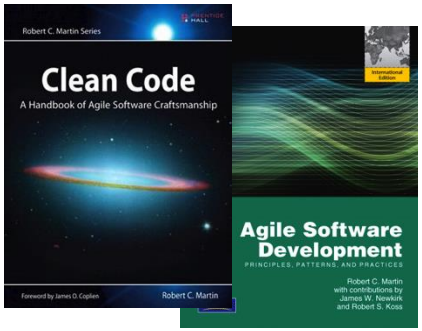
Motivation II

- Ever occurring changes in software requirements require frequent changes to the code base
  - Lehman's first law
    - *a system that is used will be changed*
  - Lehman's second law
    - *an evolving system increases its complexity unless work is done to reduce it*
- The effort required for a change increases the later it occurs
- Iterative and incremental (especially agile) development methods claim that the curve can be kept flat
  - by continuous *refactoring*
  - a term propagated especially in the agile community by –
    - Kent Beck, Ward Cunningham and Martin Fowler



Foundations for this Lecture

- *Clean Code: A Handbook Of Agile Software Craftsmanship*
  - *Recommendations for „clean code“*
  - Tools, techniques, and thought processes of good programmers
- *Agile Software Development: Principles, Patterns, and Practices*
  - Principles of object-oriented design
- *Clean Code Developer Initiative*: http://clean-code-developer.de



Robert Cecile „Uncle Bob“ Martin: Programmer since 1970, Founder and president of Object Mentor Inc.

What do you think makes code clean?

What is „Clean Code“?

- „elegant and efficient“
- „straightforward logic“
- „minimal dependencies“ *B. Stroustrup*
- „does one thing well“ *inventor of C++*

- „simple and direct“
- „reads like well-written prose“ *G. Booch*
- „never obscures the designer‘s intent“ *co-inventor of UML*

- „code that has been taken care of“ *M. Feathers*

What is „Clean Code“?

What we mainly present here is:

Robert Martin‘s
“School of Thought”

See it as field-tested

- **recommendations**
- Some recommendations are disputable and/or controversial

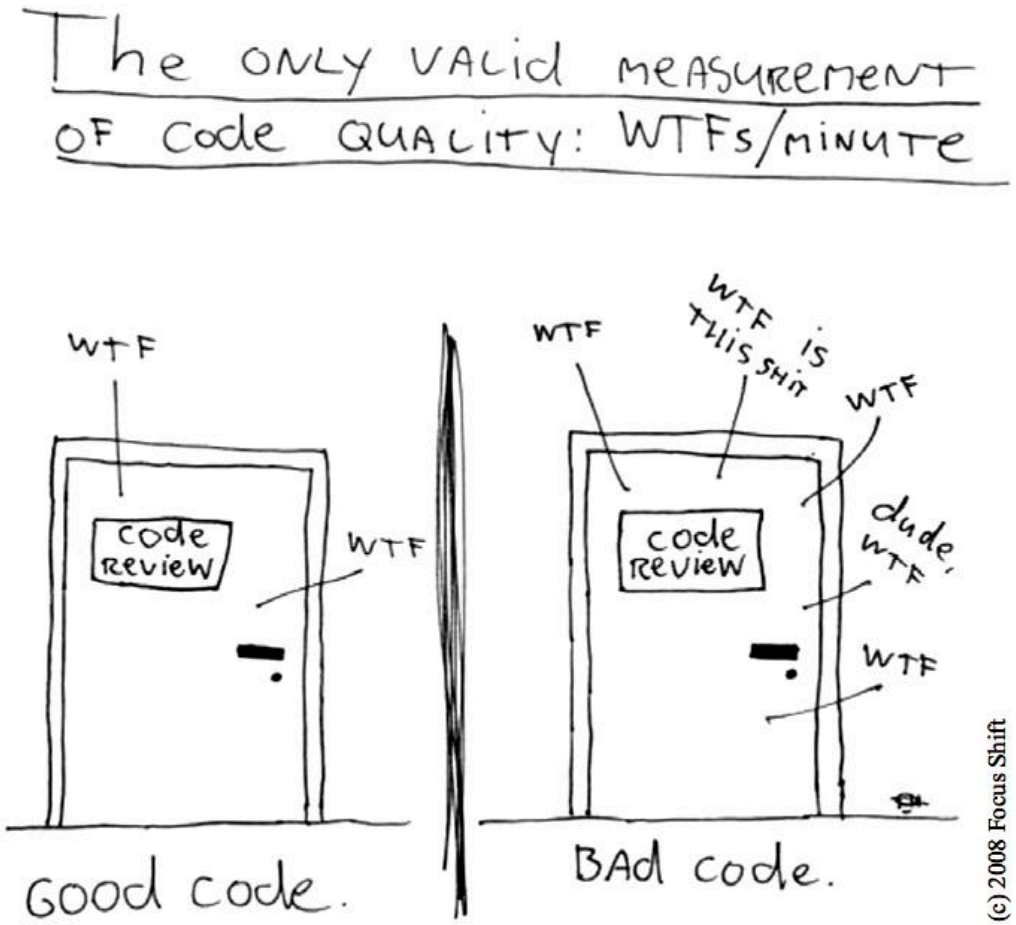


Image source: [Martin2008, p. xxv]

Object-Oriented Design (OOD)

- If you're thinking about clean code, you should also think about your code's underlying structure
- Object-Oriented Design (OOD)

  A design strategy to build a system "made up of interacting objects that maintain their own local state and provide operations on that state information." [Sommerville]

  - cf. GRASP in the upcoming lectures
- A more general set of guidelines / principles will be presented in the following
  - expected benefits: maintainability, understandability
- *There is often no clear separation between design and coding principles*

The Five SOLID Principles

- Five principles of good OO design
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DIP)
- Collected by Robert Martin
  - discussed in greater detail in the lecture “Software Evolution”
- More explanation of the principles and links to the original articles (freely available) at Wikipedia

[Martin2002]

SOLID1: Single Responsibility Principle (SRP)

"There should never be more than **one reason** for a class **to change**."
— *R. Martin*

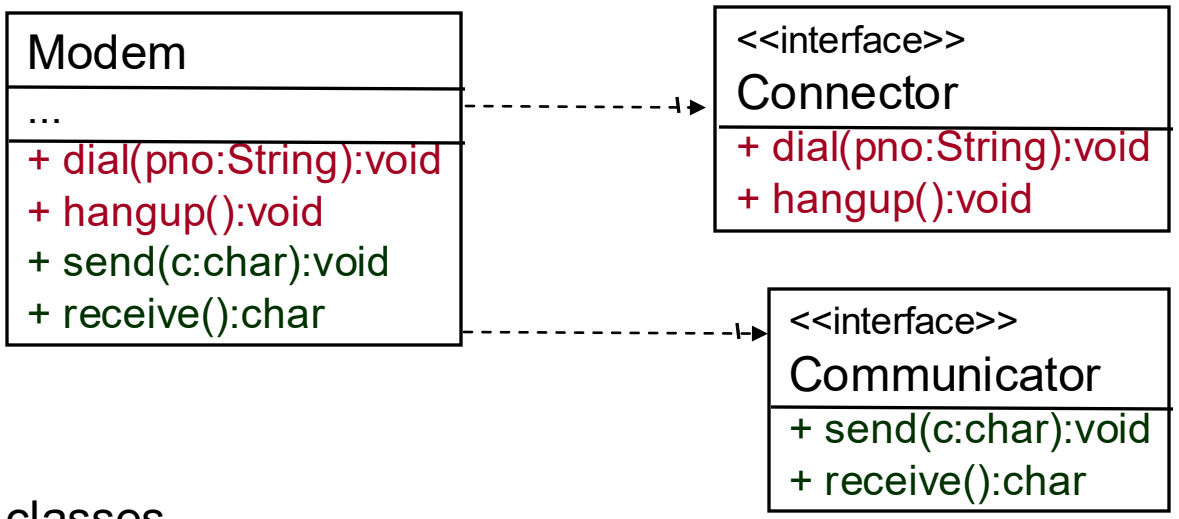
- Each responsibility deals with one core concern
  - It may also deal with further (cross-cutting) concerns
- Bad smell: Big class (~ >200 LOC, >15 methods/fields)
- Useful refactoring: Extract class
- Benefits:
  - Code is easier to understand
  - Adding/modifying functionality should affect few classes
  - Risk of breaking code is minimised

[Martin2002]

Single Responsibility & Interface Segregation

Violation Example

(inspired by SRP description linked at [http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod])



- Solution
  - divide into two classes
  - or at least derive two interfaces
    - so that clients do not have to depend on all methods of the class
- cf. SOLID4

Insertion: Command-Query-Separation

[Larman]

- Separate commands (actions) from simple queries (requests)

```
public class Dice {
   private int faceValue;
   ...
   public void roll() {
      faceValue = (int)(Math.random() * 6 + 1);
   }
   public int getFaceValue() {
      return faceValue;
   }
}
```

- Why?
- commands are expected to have
  - side effects on an object's state
  - queries should not change
  - the state of an object
  - *appropriate designs are simpler*
  - *to understand and easier to test*

[Bertrand Meyer, OOSC]

```
public class Missile {
   private String name;
   ...
   public String getName() {
      launch();
      return name;
   }
}
```

Insertion: Command-Query-Separation (2)

- Drawback: more method invocations
- Violations: the iterator-pattern implementations in Java and C#

Java

```
While (i.hasNext()) {
  Object ele = i.next();
}
```

- `hasNext()`
  query method
- `Object next()`
  query and command

C#

```
While (i.MoveNext()) {
  Object ele = i.Current();
}
```

- `Boolean MoveNext()`
  query and command
- `Object Current()`
  query

Note that iterator (`IEnumberator`) points to invalid element after creation

Insertion: Command-Query-Separation (3)

- An elegant iterator (?)

“New” Java

```
while (i.isCurrentValid()) {
  Object ele = i.current(); // no switch to next element
  i.moveNext();
}
```

- `boolean isCurrentValid()`
  query method
- `Object current()`
  query
- `void moveNext()`
  command

What happens if the last element is reached?

SOLID2: Open Closed Principle (OCP)

"Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**."

— *R. Martin, paraphrasing B. Meyer*

- Idea: Modify behaviour by adding new code, not by changing old code
- Strongly related to the "Information Hiding Principle"

- Example: Drawing a list of shapes using a switch statement

```
for (Shape shape : ShapeList)
    switch (shape.getType()) {
            case SQUARE: square.draw()
            case CIRCLE: circle.draw()
    }
```

- Needs to be modified for new shapes, better program against abstractions to keep the function open for extension:

```
for (Shape shape : ShapeList) shape.draw();
```

[Martin2002]

SOLID3: Liskov Substitution Principle (LSP)

"Functions that use pointers or references to base classes must be able to **use** objects of **derived classes without knowing** it." — *R. Martin*

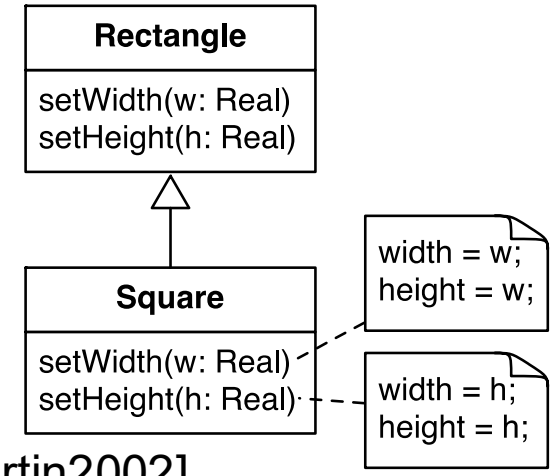
Barbara Liskov
(image from Wikipedia,
By Mirko Raner [CC BY-SA 3.0])

"Instances of sub-classes must be usable in the context of instances of the super-class." — *R. Reussner*

SOLID3: Liskov Substitution Principle (LSP)

"Functions that use pointers or references to base classes must be able to **use** objects of **derived classes without knowing** it." — *R. Martin*

- Square **is-a** Rectangle? Only in a mathematical sense!
- Square **can-NOT-substitute** Rectangle, because it offers limited behaviour (setWidth and setHeight are dependent)
- Shows that LSP goes beyond OO Analysis modelling



[Martin2002]

```
void f(Rectangle r) {
  r.setWidth(5);
  r.setHeight(4);
  // a Square would break such assumption
  assert(r.getWidth() * r.getHeight()) == 20);
}

...
f(new Square());
```

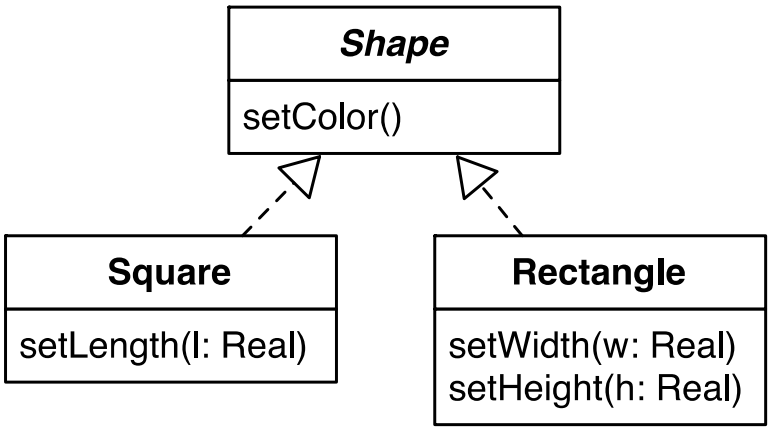
SOLID3: Liskov Substitution Principle (LSP)

- LSP is related to B. Meyer‘s **Design by Contract** (DbC):

*“When redefining a routine [in a derivative], you may only replace its **precondition** by a **weaker** one, and its **postcondition** by a **stronger** one.” — B. Meyer*

  - Rectangle’s setWidth postcondition: width = w and height = height
  - Square’s setWidth postcondition: width = w and height = w
  - Only weaker preconditions and stronger postconditions are allowed, as only they preserve substitutability
  - It is not allowed to change conditions to *arbitrarily different* ones

- Possible solution according to Liskov:
- Square/Rectangle **can-substitute**
- Shape, if Shape collects
  - less specific behaviour
  - Alternative: Drop “height = height”
- from Rectangle’s postcondition



SOLID4: Interface Segregation Principle (ISP)

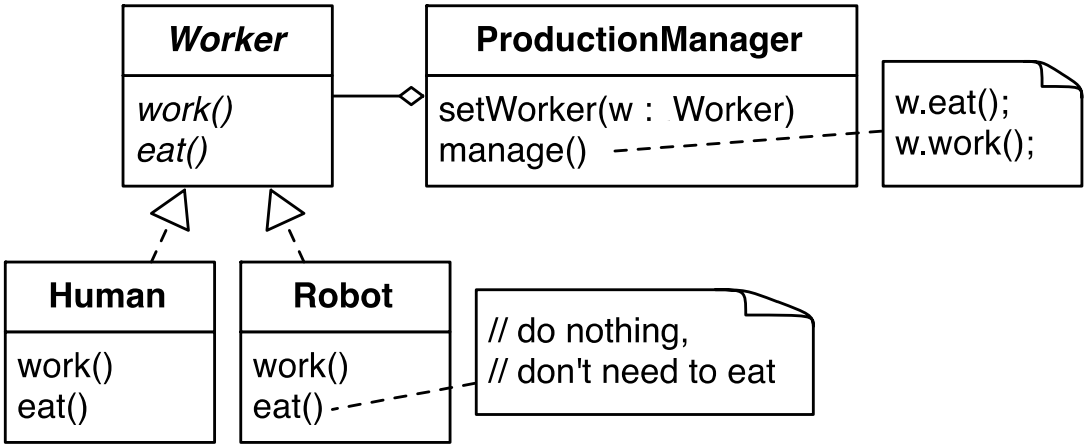
"Clients should not be forced to depend upon interfaces that they do not use." — *R. Martin*

- Power plug metaphor: only deals with one single concern
- Interfaces should be kept as lean as possible
  - **High cohesion**: Interfaces should only be concerned with single concepts
  - **Interface pollution**: Interfaces should not depend on other interfaces just because a subclass requires those
  - Interfaces should be separated if used by
- different clients
  - Refactorings: Extract interface/superclass

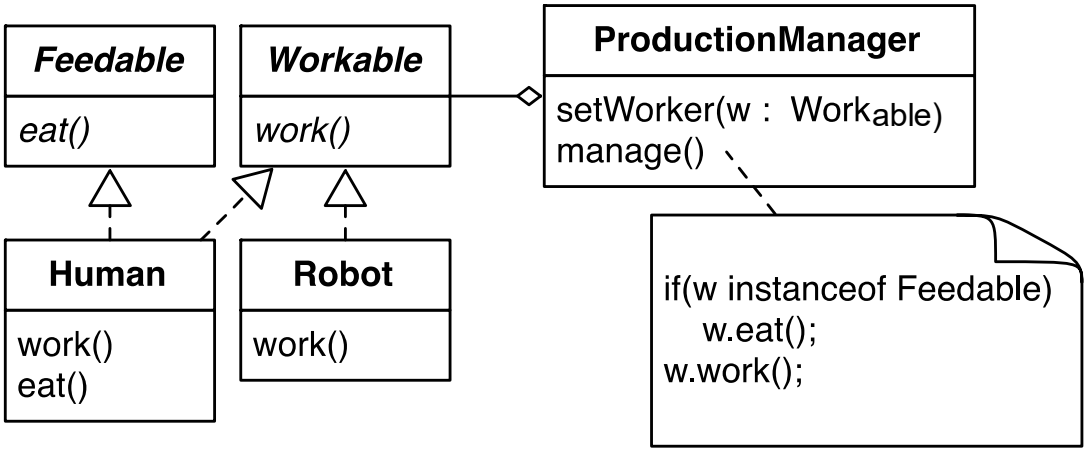
[Martin2002]

Example: Human vs. Robot Workers

Adding robots as workers who don‘t need to eat



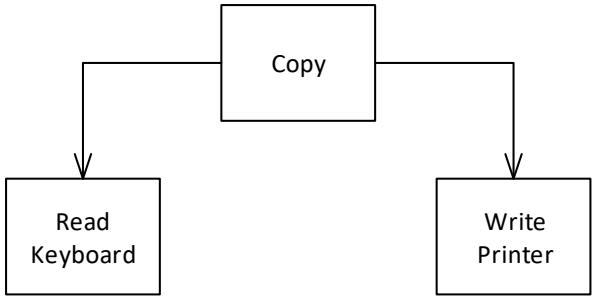
Using separate interfaces:



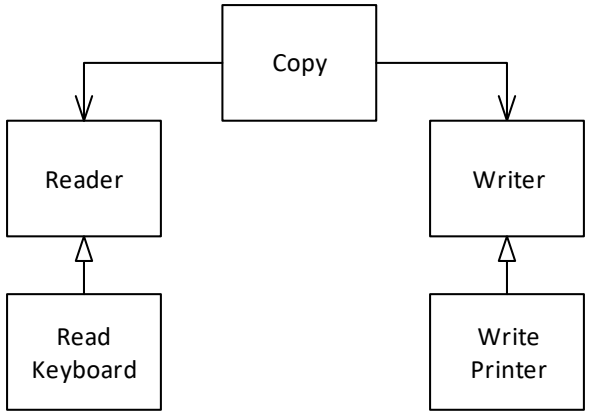
From: http://www.oodesign.com/interface-segregation-principle.html

SOLID5: Dependency Inversion Principle (DIP)

"**A.** High level modules should not depend upon low level modules. Both should **depend upon abstractions**.
B. Abstractions should not depend upon details. Details should depend upon abstractions." — *R. Martin*



Example: Copy
- knows low level modules,
- affected by change,
- cannot reuse Copy easily

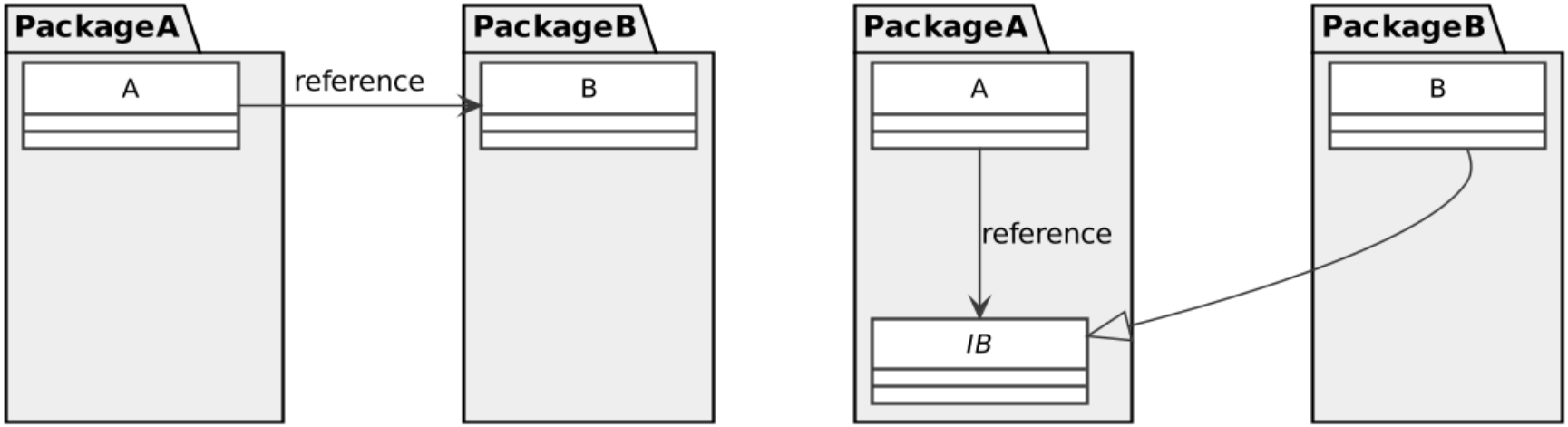


Copy Better Design:
- depends on abstraction only
- we can substitute other Readers and Writers

[Martin1996]

Example for B: .h files in C++ also define utility functions and private variables

Why “Inversion”?



Derived from https://en.wikipedia.org/wiki/Dependency_inversion_principle

- An interface has been used to *invert* the dependency between packages
- But in general:
  Add abstract concept that both classes A and B depend on

Break-Out Discussion: Find violations of SOLID principles:

```java
package model;
public abstract class User{
  private final String email;
  private String password;
  private boolean enabled = true;
  private UserManager manager;
  User(String email,String pw,UserManager mgr)[
    this.email=email; password=pw; manager=mgr;
    if (manager!=null) manager.save(this);
  }
  public String getEmail() { return email; }
  public String getPW() { return password; }
  public boolean isAdmin() { return false; }
  public boolean isEnabled() { return enabled; }
  public void setEnabled(boolean enabled) {
    this.enabled=enabled; }
}
public class Regular extends User {
  public Regular(String email,String pw,
                 UserManager mgr) {
    super(email, pw, mgr); } }
public class Admin extends User {
  public Admin(String email, String pw,
               UserManager mgr) {
    super(email, pw, mgr); }
  public void setEnabled(boolean enabled) {
    /* Can't disable admins */ }
  public boolean isAdmin() { return true; }
}
```

```java
package controller;
public class UserManager {
  private List<Regular> users=new ArrayList<>();
  private List<Admin>  admins=new ArrayList<>();
  public void save(User user) {
    if (user == null) return;
    if (user instanceof Admin) admins.add(user);
    if (user instanceof Regular) users.add(user);
  }
  public boolean auth(String email, String pw) {
    /* find user by email and check password */
  }
  public void messageToUser(User u, String t){
    Email mail = new Email();
    mail.setRecipient(u.getEmail());
    mail.setMessage(t);
    mail.send();
  }
}
```

Break-Out Discussion: Find violations of SOLID principles:

```java
package model;
public abstract class User{
  private final String email;
  private String password;
  private boolean enabled = true;
  private UserManager manager;
  User(String email,String pw,UserManager mgr){
    this.email=email; password=pw; manager=mgr;
    if (manager!=null) manager.save(this);
  }
  public String getEmail() { return email; }
  public String getPW() { return password; }
  public boolean isAdmin() { return false; }
  public boolean isEnabled() { return enabled; }
  public void setEnabled(boolean enabled) {
    this.enabled=enabled; }
}
public class Regular extends User {
  public Regular(String email,String pw,
                 UserManager mgr) {
    super(email, pw, mgr); } }
public class Admin extends User {
  public Admin(String email, String pw,
               UserManager mgr) {
    super(email, pw, mgr); }
  public void setEnabled(boolean enabled) {
    /* Can't disable admins */ }
  public boolean isAdmin() { return true; }
}
```

```java
package controller;
public class UserManager {
  private List<Regular> users=new ArrayList<>();
  private List<Admin>  admins=new ArrayList<>();
  public void save(User user) {
    if (user == null) return;
    if (user instanceof Admin) admins.add(user);
    if (user instanceof Regular) users.ad
  }
  public boolean auth(String email, String pw) {
    /* find user by email and check password */
  }
  public void messageToUser(User u, String t){
    Email mail = new Email();
    mail.setRecipient(u.getEmail());
    mail.setMessage(t);
    mail.send();
  }
}
```

DIP

ISP

LSP

OCP

SRP

More Principles

More principles of good OO design and clean code

- Law of Demeter (don’t talk to strangers)
- Boy Scout Rule
- Principle of Least Surprise
- Coding Conventions
- Don’t repeat yourself (DRY)
- Keep it simple, stupid (KISS)
- You ain’t gonna need it (YAGNI)
- Single Level of Abstraction

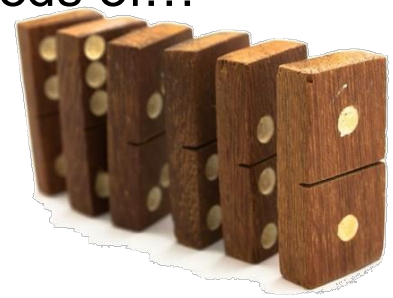
Law of Demeter ("Don't talk to strangers")

A module should not know about the innards of the objects it manipulates.

Corresponds to the bad smell “Message Chains”:

```
value = getClassA().getClassB(). … .getNeededValue();
```

- ➔ ties code to particular class structure, which is likely to break.
- Rule: A method m of a class C should only call the methods of…
  - C
  - An object created by m
  - An object passed as an argument to m
  - An object held in an instance variable of C



[Martin2008, p. 97]

Excursion: Law of Demeter

More formally, the Law of Demeter for functions requires that a method m of an object O may only invoke the methods of the following kinds of objects: [*]

- O itself
- m's parameters
- Any objects created/instantiated within m
- O's direct component objects
- A global variable, accessible by O, in the scope of m

(*) http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf

Excursion: Law of Demeter

```
// version with violation
class Engine {
  public void start() {
    // start the engine.
  }
}

class Car {
  public Engine engine;
  public Car() {
    engine = new Engine();
  }
}

class Driver {
  public void drive() {
    Car myCar = new Car();
    myCar.engine.start();
    // here the law is being violated
  }
}
```

```
class Car {
  private Engine engine;
  public Car() {
    engine = new Engine();
  }
  public void prepareForDriving() {
    engine.start();
  }
}


class Driver {
  public void drive() {
    Car myCar = new Car();
    myCar.prepareForDriving();
  }
}
```

Translated Example from German Wikipedia

Boy Scout Rule

„Leave the campground cleaner than you found it!“

— *The Boy Scouts of America*

- Code degrades as time passes
- We seldom start with a greenfield

- *Being honest:*
  - *To the code*
  - *To your colleagues*
  - *To yourself about the code*

- *Agile values (XP, Scrum)*
- *Cleaning up is part of the concept of „done“*

➔ *Refactor your code before checking it in*

[Martin2008]

Principle Of Least Surprise

Any function or class should implement the behaviours that another programmer could reasonably expect.

If **obvious behaviour** remains unimplemented, readers and users…

- no longer depend on their **intuition** about function names
- fall back on reading internals

Example: What does a reader expect from the following code?

```
String dayName = "Monday";
Day day = DayDate.StringToDay(dayName);
```

- “Monday” should be translated to Day.MONDAY
- Should accept common abbreviations
- Should ignore case

[Martin2008, p. 288f]

Code Conventions: Naming

- Standardised (with respect to a project or team)
- Meaningful, i.e. clear for everyone
- **Intention-revealing**:

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList) if (x[0] == 4) list1.add(x);
  return list1;
}
```

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

[Martin2008, ch. 2]

Code Conventions: Naming

- Make meaningful distinction and avoid disinformation
  - Hints on **context** `AccountName` in class `Account`
    What is the distinction to just `Name`?
  - Hints on **types:** `nameString, accountList`
    superfluous or even disinformation
  - Certain **prefixes** `m_name`, `Iname`
    clutters the code
  - Rather rely on your IDE to provide such information
- Avoid **noninformation** `n1`
  - Except for well-accepted cases (`i` as a loop counter)

[Martin2008, ch. 2]

Code Conventions: Commenting

"Don't comment bad code—rewrite it."

— *B. W. Kernighan, P. J. Plaugher*

Good comments are **explaining** –

- Legal issues — `// Released under the terms of the…`
- Performance issues — `// Test case takes ages`
- Train of thought — `// Here, we are safe to assume…`
- Intent — `// RegEx matches date format DD-MM-YY`
- Algorithms — `// create 1k threads to provoke race cond.`

Good comments are **warning** –

- Of consequences — `// X used here isn't threadsafe, so…`
- Over importance — `// The trim here is important, because…`

Good comments are **informative** –

- Open issues, to-dos — `//TODO: [RM] Will take care of missing…`

[Martin2008, ch. 4]

Code Conventions: Commenting II

- Whenever possible, use **well-named code** to tell what is done
  - Intermediate variables explaining steps
  - Extra methods encapsulating expressions

✗
```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65)) ...
```

✓
```
if (employee.isEligibleForFullBenefits())  ...
```

- Good comments can get drowned among **redundant** comments
- Often, trivial Javadoc comments introduce noise:

```
/** The name. */
private String name;
```

- *Nobody has the courage to delete **code commented out** by others*
  - *Rather trust version control, and instantly delete code*

[Martin2008, ch. 4]

Code Conventions: Formatting

- Visually representing levels of cohesion
- **Vertical** openness between concepts, e.g. declarations
  - e.g. add blank lines after imports or after a method is finished
  - lines that are related should be written densely together
- **Horizontal** openness
  - to accentuate operators / operator precedence
  - to separate parameters
  - use spaces to emphasize elements and indent to make scopes visible
- Most IDEs support auto-formatting:
  - rules can be configured to team standards

[Martin2008, ch. 5]

Don´t Repeat Yourself (DRY)

Do not duplicate pieces of code!

- Copy & paste decreases…
  - **Understandability**
    - Code is less compact
    - An identical concept needs to be understood multiple times
  - **Maintainability / Evolvability**
    - Losing track of copies
    - **Horror: Undocumented dependencies** from one copy to the other(s)
    - Need to find and modify all copies when removing bugs or changing behaviour
- Duplicated code fosters errors and inconsistencies
- Create methods to factor our common code
- Introduce parameters for specific behaviour

Keep It Simple, Stupid (KISS)

"Make everything as **simple** as possible, but not simpler"
— *Albert Einstein*

- Good code is easy to understand by anybody
- Good code addresses the problem adequately

- For example, if an `IEnumerable` is suitable, do not use an `ICollection` or even an `IList`

- Techniques which help ensure that your code is understandable by others:
  - Code reviews
  - Pair programming

You Ain't Gonna Need It (YAGNI)

Only implement required features!

- Featurism is **costly**:
  - unrequested features need to be tested, documented
  - over-engineered systems sacrifice maintainability, as they are overly complex (KISS)
- Do not create a **platform**, unless you are explicitly asked and payed for this
- Beware of **optimisations**!
  - Often merely treat symptoms
  - Too costly to be done prematurely

Single Level of Abstraction (SLA)

- Newspaper metaphor:
  - Good newspaper articles are well-ordered
  - Navigation with details increasing:
  - headline (very high abstraction)
    → text with synopsis (high abstraction)
    → rest (details)

- **Statements within a function** should be at the same abstraction level
  - if not, extract expressions/statements of higher detail into an own method
  - *like one paragraph*

- **Functions in a class**: The abstraction level should decrease depth-first when reading from top to bottom
- i.e., all referenced functions are
  - listed directly afterwards
  - *like the newspaper article*

[Martin2008, p. 36f.]

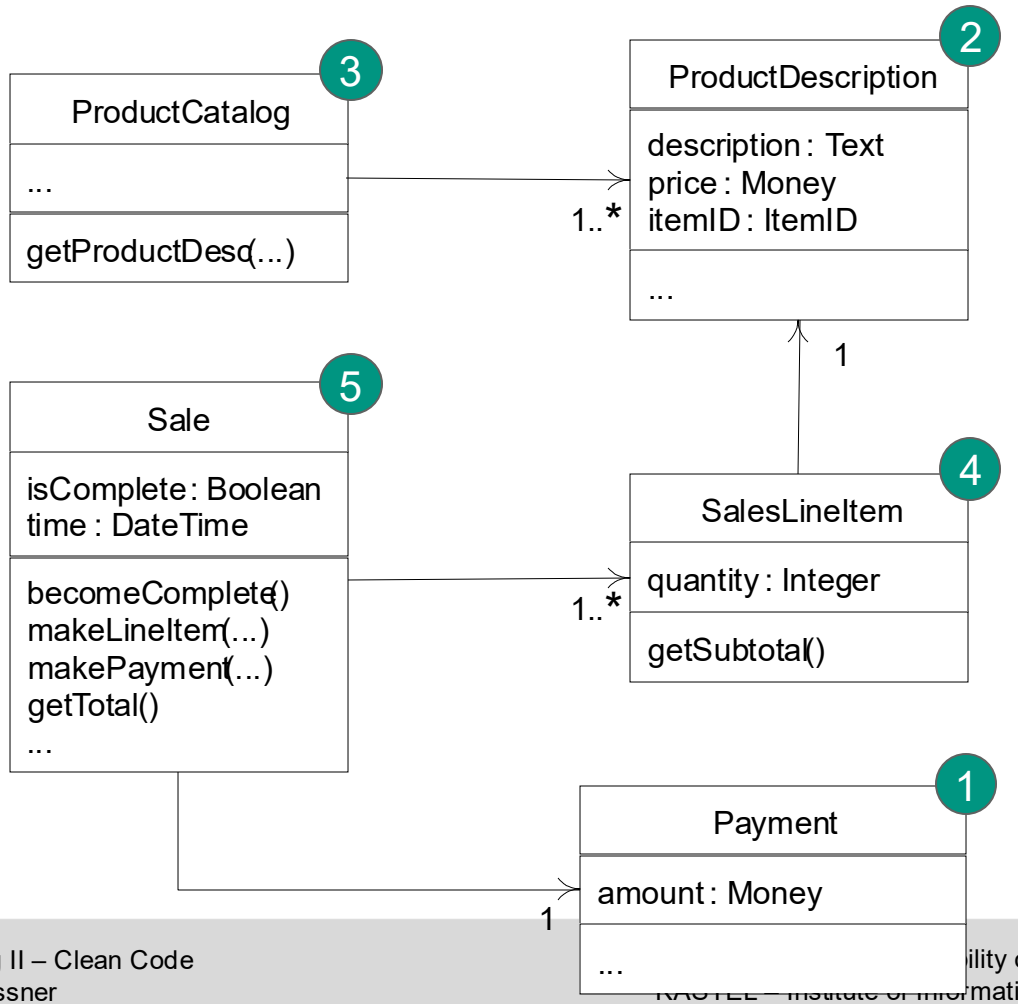
SLA Example

- Example from “Clean Code” [Martin2008]

> *“To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.*
>
> *To include the setups, we include the suite setup if this is a suite, then we include the regular setup.*
>
> *To include the suite setup, we search the parent hierarchy for the “SuiteSetUp” page and add an include statement with the path of that page.*
>
> *To search the parent…”*

Order of Implementation

- For the implementation (and unit testing later) always try to **move from the least-coupled to the most-coupled classes**
  - as it avoids unnecessary creation of “stubs”



[Larman]

Use a Version Control System

- **History** of commented changes
- Shared working in a team, even on same artefacts
- Branching and merging
- **Tagging** versions as pre-release etc.
- **Reverting** to previous revisions

➔ reduces fears of breaking code

- encourages a programmer‘s willingness to refactor code

- examples include –
  - CVS
  - Subversion
  - Mercurial
  - Git

Test First

An implementation is driven by its requirements

- (Test-Driven Development)

- Clean tests should follow the **F.I.R.S.T.** rules
  - **F**ast — to run them frequently
  - **I**ndependent — A failing test does not influence others
  - **R**epeatable — in any environment, so there is no excuse for failing tests
  - **S**elf-Validating — Tests either pass or fail automatically
  - **T**imely — Tests are written right before production code
- Tests should follow same standards as production code
  - and be executed in a continuous manner
    - so-called continuous integration
    - reduces fear of breaking code

[Martin2002, ch. 9]

Static Code Analyses

- Classes of metrics
  - Duplication (detection of DRY violation)
  - Unit tests (test coverage should be > 90
  - Complexity (avg. LoC per class)
  - Potential bugs
  - Coding rules
  - Comments
  - Architecture & design

- Static detection of error classes
  - possibly also with false-positives
- Tools
  - SISSy, Sonar, FindBugs, Understand …

Code Reviews

- **Explaining** your code to others helps…
  - **Detecting errors** and unclean passages
  - **Spreading knowledge** through a team,
  - esp. to less experienced colleagues
    - about design principles
    - about further aspects of the system under development

- **Refactoring** helps to **instantly apply** suggestions, so follow-up ideas can be given in one session
  - Works only in small groups with few opinions
  - In larger groups, **design reviews** are better suitable

[Martin2002, p. 50f.]

Refactoring

“If it stinks, change it.”

—Grandma Beck, discussing child-rearing philosophy

Perhaps he (Kent Beck) was under the influence of the odors of his newborn daughter at the time, but he had come up with the notion describing the “when” of refactoring in terms of smells. “Smells,” you say, “and that is supposed to be better than vague aesthetics?”

– Martin Fowler

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

– Martin Fowler

Refactoring Code: An Example

- Methods tend to grow during development
- Bad odour (smell) of a long method arises
- What to do? Extract cohesive parts into new methods

```
void printOwing(double amount) {
  printBanner();
  //print details
  System.out.println("name:" + _name);
  System.out.println("amount" + amount);
}
```

move out into separate method

```
void printOwing(double amount) {
  printBanner();
  printDetails(amount);
}
```

```
void printDetails(double amount) {
  System.out.println("name:" + _name);
  System.out.println("amount" + amount);
}
```

[Fowler1999]

Refactoring

A „**disciplined technique for restructuring** an existing body of code, altering its internal structure **without changing its external behavior**.“ — *M. Fowler*

- Example refactoring steps:
  - Extract method/class
  - Move method/field
  - Pull up/down field
  - Inline class
  - …
- Meanwhile even supported by some IDEs
  as, for example, Eclipse, IntelliJ

[Fowler1999]

The First Rule in Refactoring…

- …is to build a **solid set of tests**
  - good tests help to prevent introducing bugs into the program through refactoring
- **Refactor with tests only!**

> *"Here are the Two Rules of Miyagi-Ryu Karate.*
>
> *Rule Number One: 'Karate for defense only.'*
>
> *Rule Number Two: 'First learn rule number one.'"*

Mr. Miyagi in *Karate Kid II*

Refactoring: Bad Smells…

- Bad code smells: symptoms for deeper problems
  - **Long method**: having code blocks lead by comments
  - **Duplicated code**
  - **Feature envy**: class excessively calls another class's methods
  - **Data class**: class merely holds data
  - **Large/God class**: class tries to do too much
  - **Inappropriate intimacy**: class has dependencies on implementation details of another class
  - …

For even more refactorings that cannot be covered here, have a look at Fowler's catalogue of refactorings:

cf. http://www.refactoring.com/catalog/index.html

[Fowler1999]

Refactoring: …and Their Cure

Suggested treatments for some smells:

Long method:

1. Extract Method: extract block

Feature envy: class A excessively calls class B's methods

➔ parts of A's methods want to be in class B

1. Extract Method: extract code block calling class B
2. Move Method: move extracted part to class B

Data class: class is a „dumb data holder"

➔ enforce information hiding principle, collect functionality

1. Encapsulate field: getter/setter instead of public access
2. Remove setting method: only for read-only value
3. Move method: collect functionality implemented elsewhere

think about responsibilities of the class

[Fowler1999]

Catalog of Refactorings

- The **generic format** of a refactoring in Fowler's book is as follows –
  - **name** of the refactoring
  - short **summary** of what the refactoring does
  - the **motivation** describes when a refactoring should and should not be done
  - the **mechanics** are a step-by-step description how to carry out the refactoring
  - the **examples** show how it works
    - may be illustrated with code snippets or UML diagrams
- *However, as the book was published in 1999 it is largely based upon Java 1.1 and UML 1.1*

When to Refactor?

- It is rather easy to carry out refactorings
  - however, it is not that simple to find out **when to refactor**
    - as there are no precise criteria for this
    - Nevertheless, so-called "***bad smells***" in code may give a good indication when refactoring is worthwhile

- More general guidelines are –
  - when you find yourself looking up details frequently
    - *what was the order of the method parameters again?*
    - *where was this method again and what does it do?*
  - when you feel the need to write a **comment**
    - **try to refactor the code so that the comment becomes superfluous**

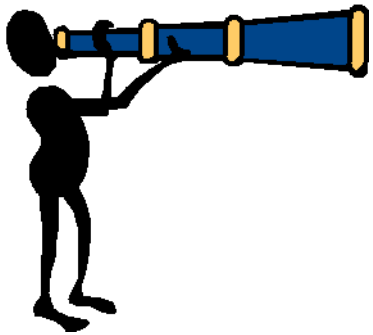
Limitations

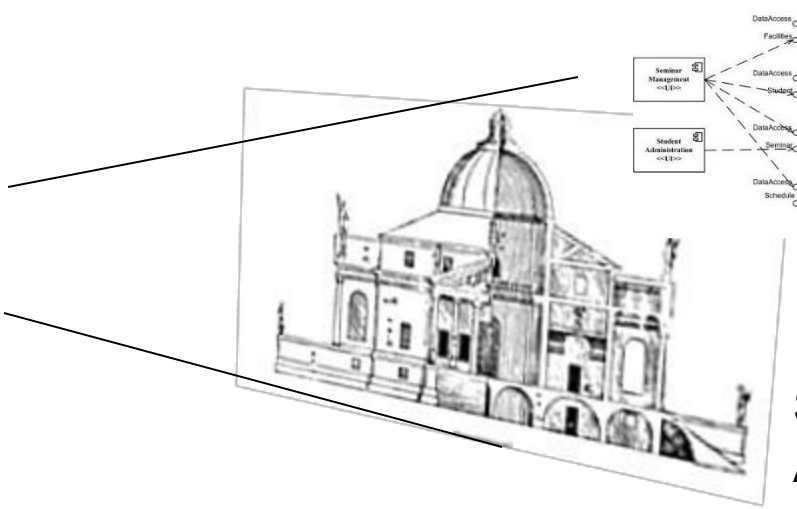
- Refactoring may influence performance negatively
  - however, Fowler recommends to do the refactoring first
    - and the performance tuning on the cleaner code afterwards
- Are there things that are difficult to refactor?
  - what if the elements of a domain layer need to be **persisted**?
    - it is often hard to change the database in the persistence layer
      - however, automated O/R mappers such as Hibernate may mitigate this problem
  - what about **published interfaces**?
    - as long as you have access to all callers of an interface, you can easily change it
    - however, once it is published to the outside this becomes harder
      - don't publish interfaces prematurely (cf. "deprecated" in Java)
      - perhaps you have to forward the calls of an old interface to the new one for some time

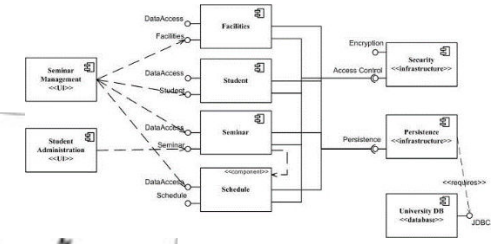
Conclusion

- What we learned –
  - recommendations on how to structure code
  - refactoring as a technique to clean up code
  - design philosophies and principles

- This lecture does only cover some aspects
  - Martin Fowler's (refactoring) and Robert Martin's (clean code) books are worth reading for every serious programmer
  - further design concepts presented in our lecture "Software-Evolution"







Software Architectures

Literature

- [Martin2008] Martin, R.C.: *Clean Code: A Handbook Of Agile Software Craftsmanship*, Prentice Hall, 2009.
- [Martin2002] Martin, R.C.: *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.
- [Martin1996] Martin, R.C.: The Dependency Inversion Principle, C++ Report, May 1996 http://www.objectmentor.com/resources/articles/dip.pdf
- [Fowler1999] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
- [Johnson1990] Johnson, R., Opdyke W.: *Refactoring: An aid in designing application frameworks and evolving object-oriented systems*. In: Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA), 1990.
- [GOF1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Element of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [Freeman2006] Freeman, E., Freeman, E.: *Head First – Design Patterns*, O'Reilly. 2004.

Also Interesting…

- Be passionate about what you are doing!
  - how to become a better programmer
    - in mainly a non-technical sense

When did you learn a new programming
- language the last time..?

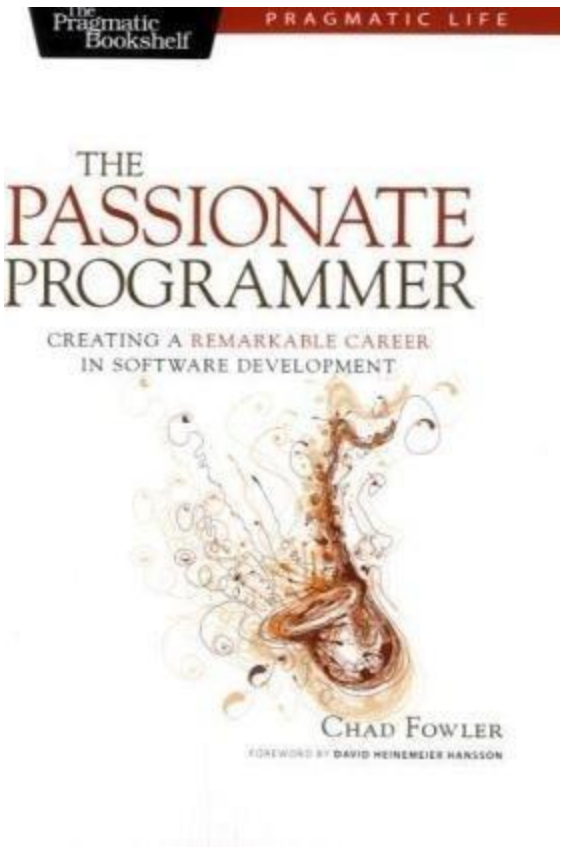


Image Credits

http://www.sxc.hu/photo/901054 abandoned house
[Martin2008, p. xxv] the only valid measurement of code quality
http://duncandavidson.photoshelter.com Robert Martin on stage at RailsConf 2009, © J. D. Davidson
http://www.sxc.hu/photo/889094 Neuschwanstein castle
http://www.sxc.hu/photo/1125736 businessman
http://www.sxc.hu/photo/673687 gift surprise
http://www.sxc.hu/photo/1047008 hammer
http://www.sxc.hu/photo/201768 trash bin
http://www.sxc.hu/photo/1208775 mailbox
http://www.sxc.hu/photo/1302426 feet
http://www.sxc.hu/photo/418651 newspapers
http://www.sxc.hu/photo/246239 caliper
http://www.sxc.hu/photo/1096878 dominoes
http://www.sxc.hu/photo/866529 feedback form
http://www.sxc.hu/photo/951744 USB plug
http://www.sxc.hu/photo/1189035 malloy duck
http://www.sxc.hu/photo/592872 rubber duck

Appendix

Separation of Concerns (SoC)

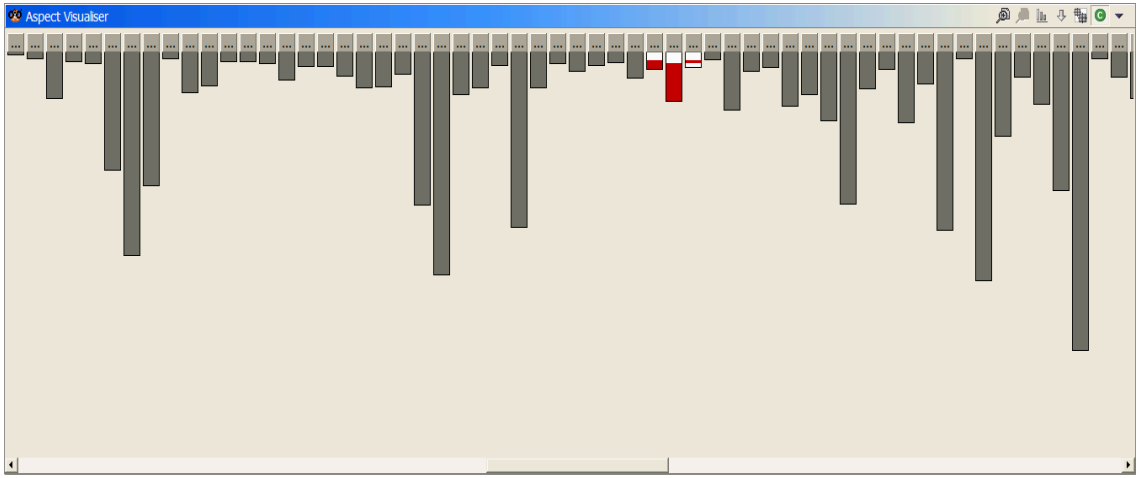
Each module should be focused on a single concern.

- Benefits
  - Loose coupling, high cohesion
  - Better testability: each test stays focused on one module
- Some concerns may crosscut a system‘s core concerns
  - Typical crosscutting concerns:
    - Tracing/Logging
    - Security
    - Transactionality
    - Caching
  - Aspect Oriented Programming (AOP) provides adequate concepts
    - through weaving concerns into the business logic

SoC Example

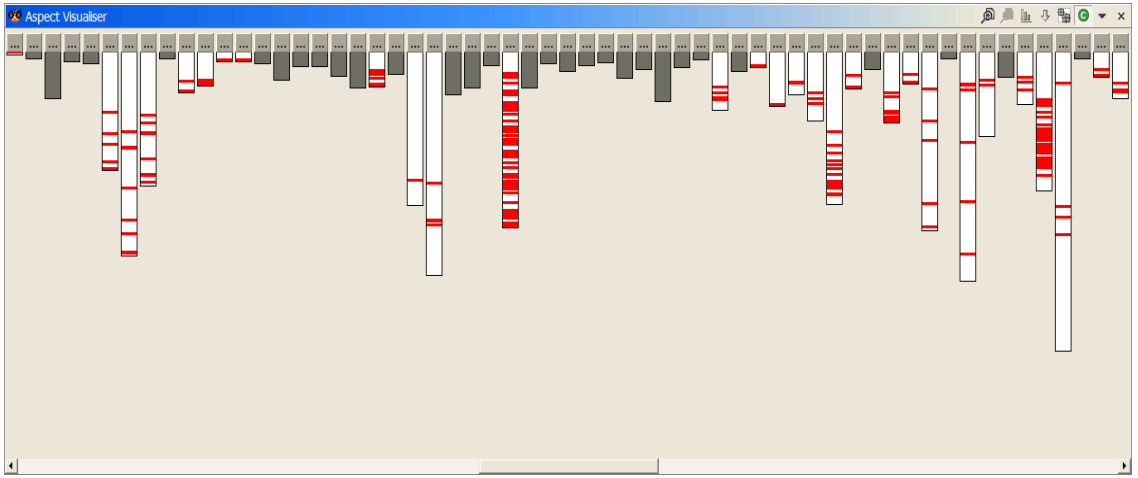
[AspectJ]

- Socket creation in Tomcat
- good separation
  - of concerns



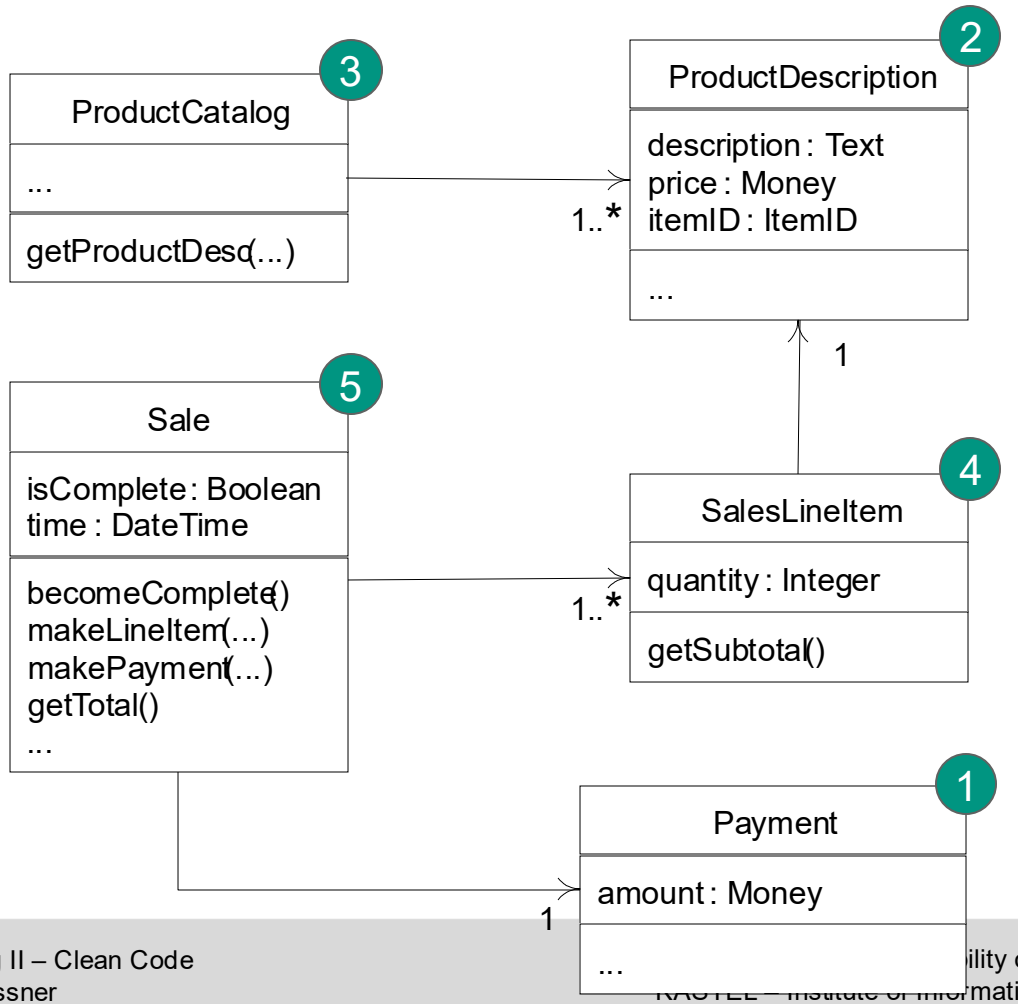
vs.

- Logging in Tomcat
- concerns are
- scattered all over
  - the code



Order of Implementation

- For the implementation (and unit testing later) always try to **move from the least-coupled to the most-coupled classes**
  - as it avoids unnecessary creation of “stubs”



[Larman]

Use a Version Control System

- **History** of commented changes
- Shared working in a team, even on same artefacts
- Branching and merging
- **Tagging** versions as pre-release etc.
- **Reverting** to previous revisions

➔ reduces fears of breaking code

- encourages a programmer‘s willingness to refactor code

- examples include –
  - CVS
  - Subversion
  - Mercurial
  - Git

Test First

An implementation is driven by its requirements

- (Test-Driven Development)

- Clean tests should follow the **F.I.R.S.T.** rules
  - **F**ast — to run them frequently
  - **I**ndependent — A failing test does not influence others
  - **R**epeatable — in any environment, so there is no excuse for failing tests
  - **S**elf-Validating — Tests either pass or fail automatically
  - **T**imely — Tests are written right before production code
- Tests should follow same standards as production code
  - and be executed in a continuous manner
    - so-called continuous integration
    - reduces fear of breaking code

[Martin2002, ch. 9]

Static Code Analyses

- Classes of metrics
  - Duplication (detection of DRY violation)
  - Unit tests (test coverage should be > 90%)
  - Complexity (avg. LoC per class)
  - Potential bugs
  - Coding rules
  - Comments
  - Architecture & design
- Tools
  - SISSy, Sonar, FindBugs, Understand …

Code Reviews

- **Explaining** your code to others helps…
  - **Detecting errors** and unclean passages
  - **Spreading knowledge** through a team,
  - esp. to less experienced colleagues
    - about design principles
    - about further aspects of the system under development

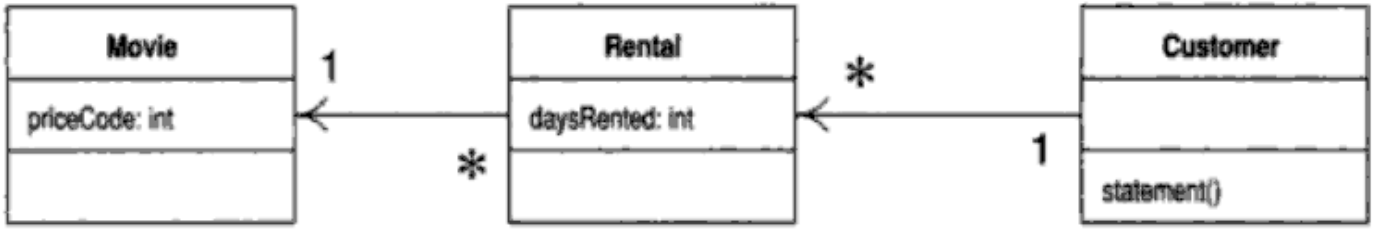
- **Refactoring** helps to **instantly apply** suggestions, so follow-up ideas can be given in one session
  - Works only in small groups with few opinions
  - In larger groups, **design reviews** are better suitable

[Martin2002, p. 50f.]

Appendix: Example from Fowler Book

- Imagine the following code being part of a larger system
  - it is a program to **calculate and print a statement of a customer's charges** at a **video store**
    - taken from [Fowler]
  - it is also supposed to **calculate frequent renter points**

- The program is told which movies a customer rented and for how long
  - the initial version of the program comprises just three classes
    - shown in the UML diagram below
      - with their most important elements only



Movie and Rental Classes

- … before refactoring begins
  - just two simple data classes with constructor, getters and setters

```
public class Movie {

        public static final int CHILDRENS = 2;
        public static final int NEW_RELEASE = 1;
        public static final int REGULAR = 0;

        private String _title;
        private int _priceCode;

        public Movie(String title, int priceCode) {
                _title = title;
                _priceCode = priceCode;
        }

        public int getPriceCode() {
                return _priceCode;
        }

        public void setPriceCode(int arg) {
                _priceCode = arg;
        }
        public String getTitle() {
                return _title;
        }

}
```

```
public class Rental {

        private Movie _movie;
        private int _daysRented;

        public Rental(Movie movie, int daysRented) {
                _movie = movie;
                _daysRented = daysRented;
        }

        public int getDaysRented() {
                return _daysRented;
        }

        public Movie getMovie() {
                return _movie;
        }
}
```

The Customer Class

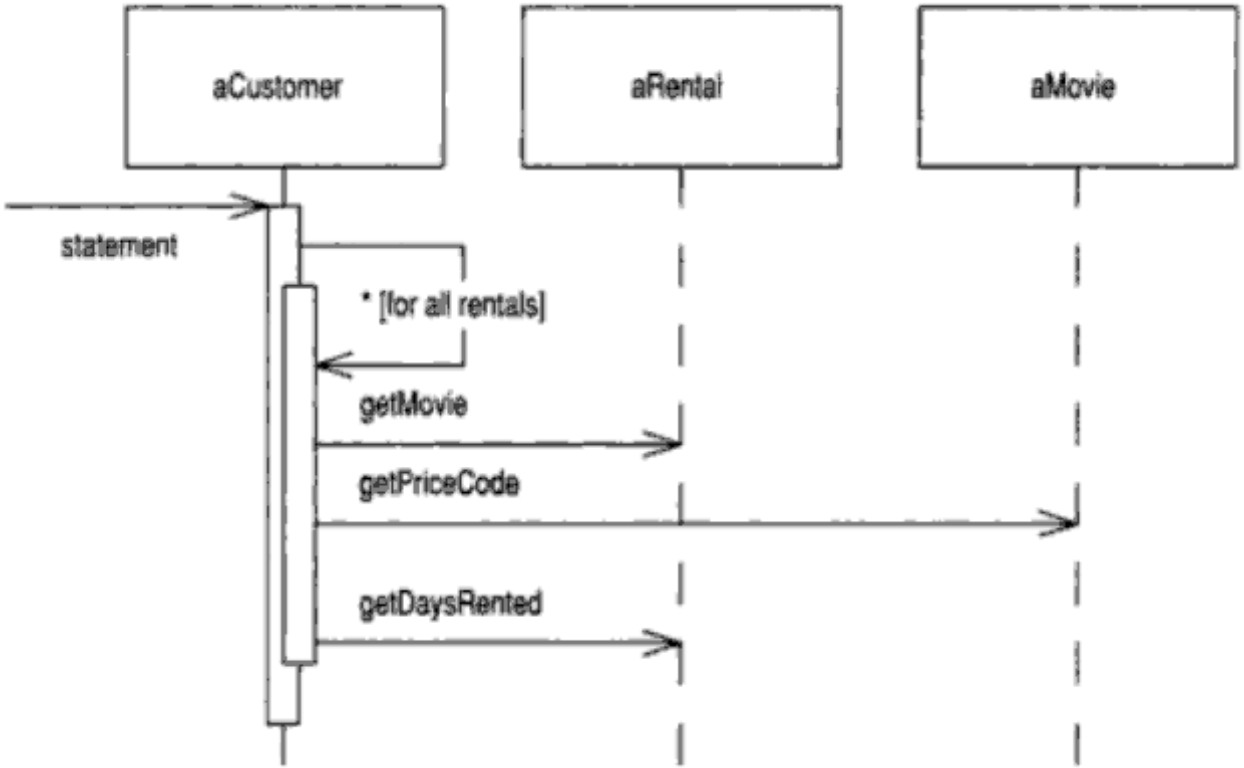
- … is also quite simple

```
public class Customer {

        private String _name;
        private Vector _rentals = new Vector();

        public Customer(String name) {
                _name = name;
        }

        public void addRental(Rental arg) {
                _rentals.addElement(arg);
        }

        public String getName() {
                return _name;
        }
}
```

- if there wasn‘t this overly complex statement method…

Interactions…

- … required for the statement method



Source Code of statement() I

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";

        while (rentals.hasMoreElements()) {
                double thisAmount = 0;
                Rental each = (Rental) rentals.nextElement();

                //determine amounts for each line
                switch (each.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                        thisAmount += 2;
                        if (each.getDaysRented() > 2)
                                thisAmount += (each.getDaysRented() - 2) * 1.
                        break;
                case Movie.NEW_RELEASE:
                        thisAmount += each.getDaysRented() * 3;
                        break;
                case Movie.CHILDRENS:
                        thisAmount += 1.5;
                        if (each.getDaysRented() > 3)
                                thisAmount += (each.getDaysRented() - 3) * 1.
                        break;
                }

...
```

Source Code of statement() II

```
...
                        // add frequent renter points
                        frequentRenterPoints++;
                        // add bonus for a two day new release rental
                        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
                                                        && each.getDaysRented() > 1)
                                frequentRenterPoints++;

                        // show figures for this rental
                        result += "\t" + each.getMovie().getTitle() + "\t"
                                                + String.valueOf(thisAmount) + "\n";
                        totalAmount += thisAmount;
                }

                // add footer lines
                result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
                result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";

                return result;
}
```

- This program works fine…
  - but it is quite quick and dirty
  - and rather procedural instead of object-oriented (GRASP!)
- In other words: ***the code smells***

Challenges

- The compiler does not care whether code is ugly or clean
  - but for a human that massive statement method is hard to understand
    - and thus hard to change
- Now imagine the customer would like to have an HTML version of the statement
  - duplicating the statement with an htmlStatement method does not seem too difficult for now
  - but what happens if the charging rules change?
    - would you like to do the update twice?
  - or when the movie classification should be changed?
    - the only guarantee you have is that it will be changed in a few months again ☺
- When you have to add a feature to a program and find the program not conveniently structured to add it
  - refactor the program before adding the feature

Remember: The First Rule in Refactoring…

- …is to build a **solid set of tests**
  - good tests help to prevent introducing bugs into the program through refactoring
  - **Refactor with tests only!**

- JUnit tests for the system may look as follows

```
@Test
public void testStatementForRegularMovie() {
        Customer customer2 = new Customer("Sallie");
        Movie movie1 = new Movie("Gone with the Wind", Movie.REGULAR);
        Rental rental1 = new Rental(movie1, 3); // 3 day rental
        customer2.addRental(rental1);
        String expected = "Rental Record for Sallie\n" +
                          "\tGone with the Wind\t3.5\n" +
                          "Amount owed is 3.5\n" +
                          "You earned 1 frequent renter points";
        String statement = customer2.statement();
        assertEquals(expected, statement);
}
```

Let‘s Refactor…

- The first target for a refactoring is clearly the **overly complex statement method**
  - try to decompose it into smaller pieces
    - as they make code more understandable and manageable
  - the target is to make the creation of an HTML statement simpler
    - with less duplicated code
- The **switch statement** seems to be a good starting point
  - it looks as it could be extracted into its own method
    - following the *Extract Method* refactoring that suggests –
      - to investigate required parameters and local variables of the parent method
        - *each* and *thisAmount* in our case
      - non modified variables can become a parameter of the new method
      - modified variables need to be returned by the new method

Refactoring Example – Step I

```
… OLD code
//determine amounts for each line
switch (each.getMovie().getPriceCode()) {
case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                                thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3
                                thisAmount +=
                break;
}
```

```
// NEW code
thisAmount = amountFor(each);
```

```
private double amountFor(Rental each) {                    // … NEW code
    double thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                                thisAmount += (each.getDaysRented() - 2) * 1.
                break;
    case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
    case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                                thisAmount += (each.getDaysRented() - 3) * 1.
                break;
    }
    return thisAmount;
}
```

avoid redundancy!

Step II

- Variable names are a bit strange and hard to understand for humans
  - let‘s change them

meaningful names!

```
private double amountFor(Rental each) {                    // … OLD code
    double thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2
                                thisAmount +=
                break;
    case Movie.NEW_RELEASE:
                thisAmount += each.getDays
                break;
    case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3
                                thisAmount +=
                break;
    }
    return thisAmount;
}
```

```
class Customer…
private double amountFor(Rental aRental) {                 // … NEW code
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
    case Movie.REGULAR:
                result += 2;
                if (aRental.getDaysRented() > 2)
                                result += (aRental.getDaysRented() - 2) * 1.5;
                break;
    case Movie.NEW_RELEASE:
                result += aRental.getDaysRented() * 3;
                break;
    case Movie.CHILDRENS:
                result += 1.5;
                if (aRental.getDaysRented() > 3)
                                result += (aRental.getDaysRented() - 3) * 1.5;
                break;
    }
    return result;
}
```

Step III

- The amountFor method uses information from Rental, but none from Customer
  - thus it is better, to move it over

information expert!

```
class Customer…
private double amountFor(Rental aRental) {              // … OLD code
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
    case Movie.REGULAR:
                result += 2;
                if (aRental.getDaysRented() >
                            result += (aRenta
                break;
    case Movie.NEW_RELEASE:
                result += aRental.getDaysRen
                break;
    case Movie.CHILDRENS:
                result += 1.5;
                if (aRental.getDaysRented() >
                            result += (aRenta
                break;
    }
    return result;
}
```

```
class Rental…
private double getCharge(/* parameter removed */) {      // … NEW code
    double result = 0;
    switch (getMovie().getPriceCode()) {
    case Movie.REGULAR:
                result += 2;
                if (getDaysRented() > 2)
                            result+= (getDaysRented() - 2) * 1.5;
                break;
    case Movie.NEW_RELEASE:
                result += getDaysRented() * 3;
                break;
    case Movie.CHILDRENS:
                result += 1.5;
                if (getDaysRented() > 3)
                            result += (getDaysRented() - 3) * 1.5;
                break;
    }
    return result;
}
```

Step III.b

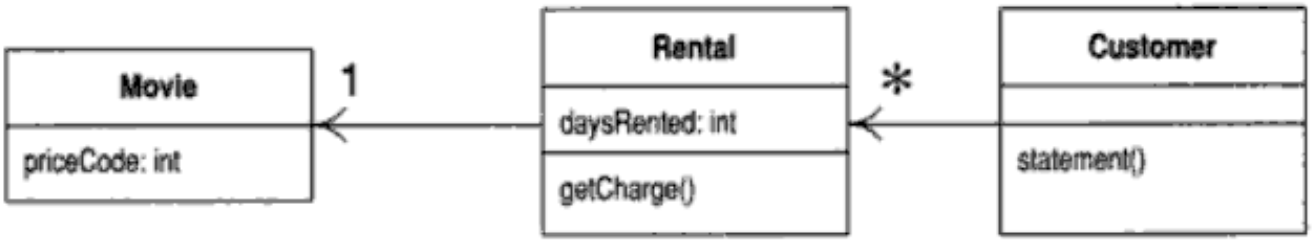
- Don’t forget to adjust the references to the old code accordingly

```
// OLD code
thisAmount = amountFor(each);
```

```
// NEW code
thisAmount = each.getCharge();
```

  - and to remove the old method afterwards
    - the compiler should tell you, if you forgot to adapt a call

- Afterwards, the class diagram looks as follows



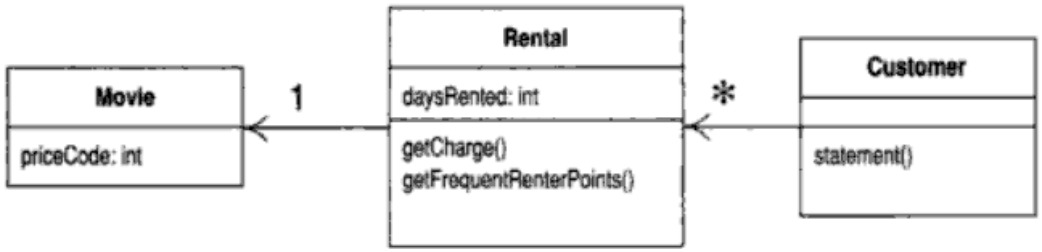
Step IV

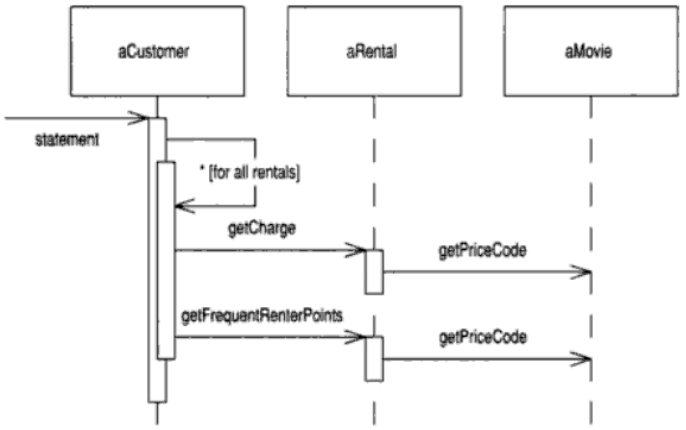
- Let's turn to the statement method again
  - and do the same with the calculation of the frequent renter points
    - this is better placed as a method in Rental as well

```
... // OLD code
                // add frequent renter points
                frequentRenterPoints++;
                // add bonus for a two day new release rental
                if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
                                                && each.getDaysRented() > 1)
                        fre

                // show figures fo
                result += "\t" + ea

                totalAmount += t
}
```

```
... // NEW code
                frequentRenterPoints += each.getFrequentRenterPoints();

                // show figures for this rental
                result += "\t" + each.getMovie().getTitle() + "\t"
                                                + String.valueOf(thisAmount) + "\n";
                totalAmount += thisAmount;
}
```

high cohesion!
information expert





Step V

- Using a switch statement for an attribute of another object is a bad idea
  - this calls for polymorphism for this object
    - thus, as a first step, getCharge should be moved to Movie

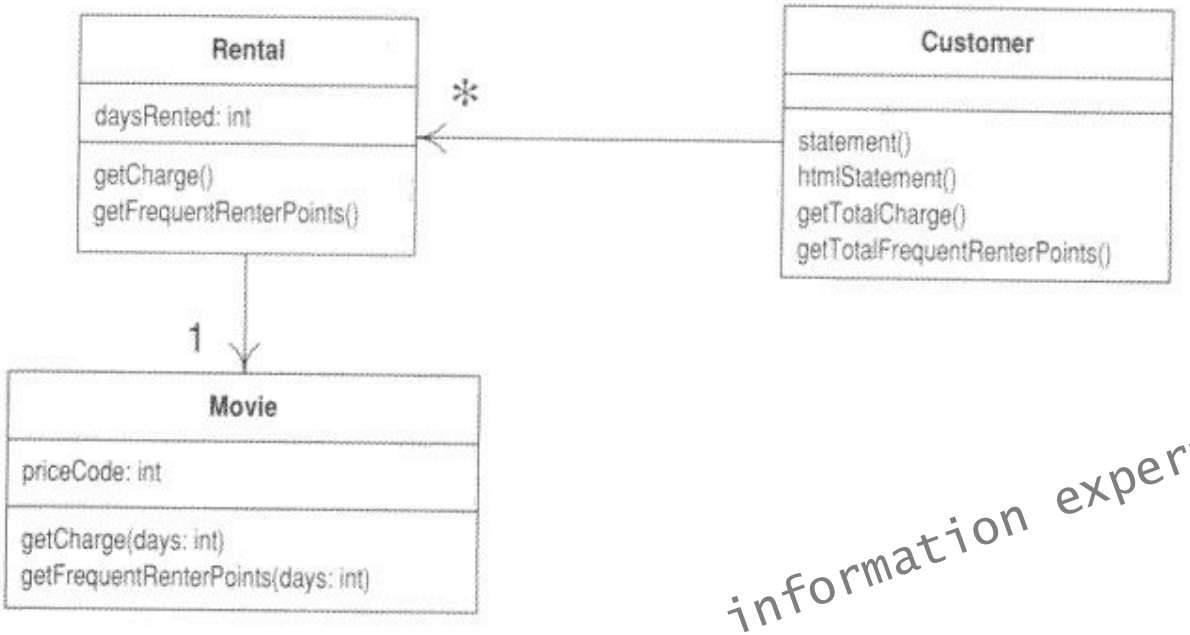
information expert

```
class Rental…
private double getCharge() {      // … OLD code
    double result = 0;
    switch (aRental.getMovie().getPriceCode())
    case Movie.REGULAR:
                result += 2;
                if (aRental.getDaysRented() > 2)
                            result+= (aRental.g
                break;
    case Movie.NEW_RELEASE:
                result += aRental.getDaysRente
                break;
    case Movie.CHILDRENS:
                result += 1.5;
                if (aRental.getDaysRented() > 3)
                            result += (aRental.
                break;
    }
    return result;
}
```

```
class Movie…
private double getCharge(int daysRented) {      // … NEW code
    double result = 0;
    switch (getPriceCode()) {
    case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                            result+= (daysRented - 2) * 1.5;
                break;
    case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
    case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                            result += (daysRented - 3) * 1.5;
                break;
    }
    return result;
}
```

Step V.b

- As a second step, we also move the calculation of the frequent renter points to Movie
  - we preserve both methods in Rental as a forward
  - this results in the following class diagram

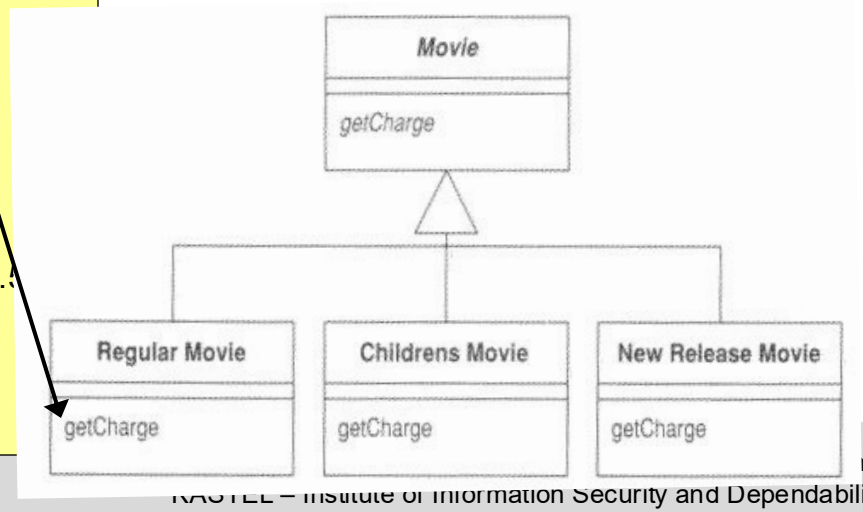


Step VI

- Introduce the inheritance for the Movie class
  - i.e. make it abstract and create a subclass for each case in the switch statement
  - and move the getCharge cases to the method of the appropriate subclass

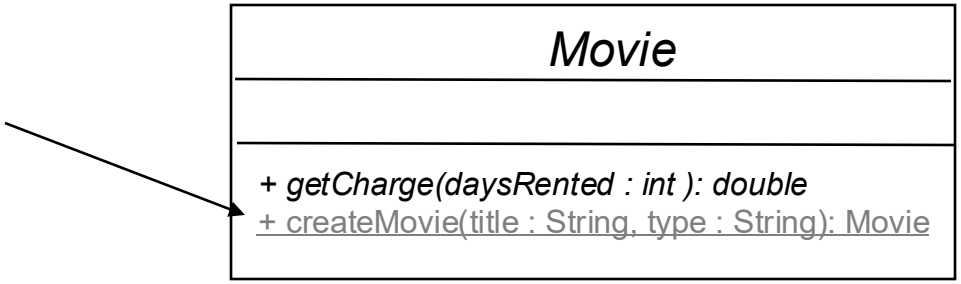
```
class Movie…
private double getCharge(int daysRented) {      // … OLD code
    double result = 0;
    switch (getPriceCode()) {
    case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                                result += (daysRented - 2) * 1.5;
                break;
    case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
    case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                                result += (daysRented - 3) * 1.5;
                break;
    }
    return result;
}
```

polymorphism!



Does it work? ☺

- As far as we are concerned right now: YES
- However, a more detailed look on Movie reveals the need for some more modification
- it makes sense to introduce a
  - static factory method
  - deciding which
    - subclass to instantiate
    - this is another well-known
    - design pattern
    - you might want to use an *enum* for this…
      - see e.g. http://whiteboxcomputing.com/java/enum_factory



- But, what if a Movie changes its type during its lifetime?
  - e.g. a new release becomes a regular movie after some time
  - objects cannot change their types
  - this calls for a more complex refactoring that separates the price calculation from the Movie and makes it changeable
    - can be solved via introducing *strategies* for this purpose