Patterns in Software Engineering

07 Antipatterns II

Stephan Krusche

12 December 2022
Technical University of Munich

Course schedule

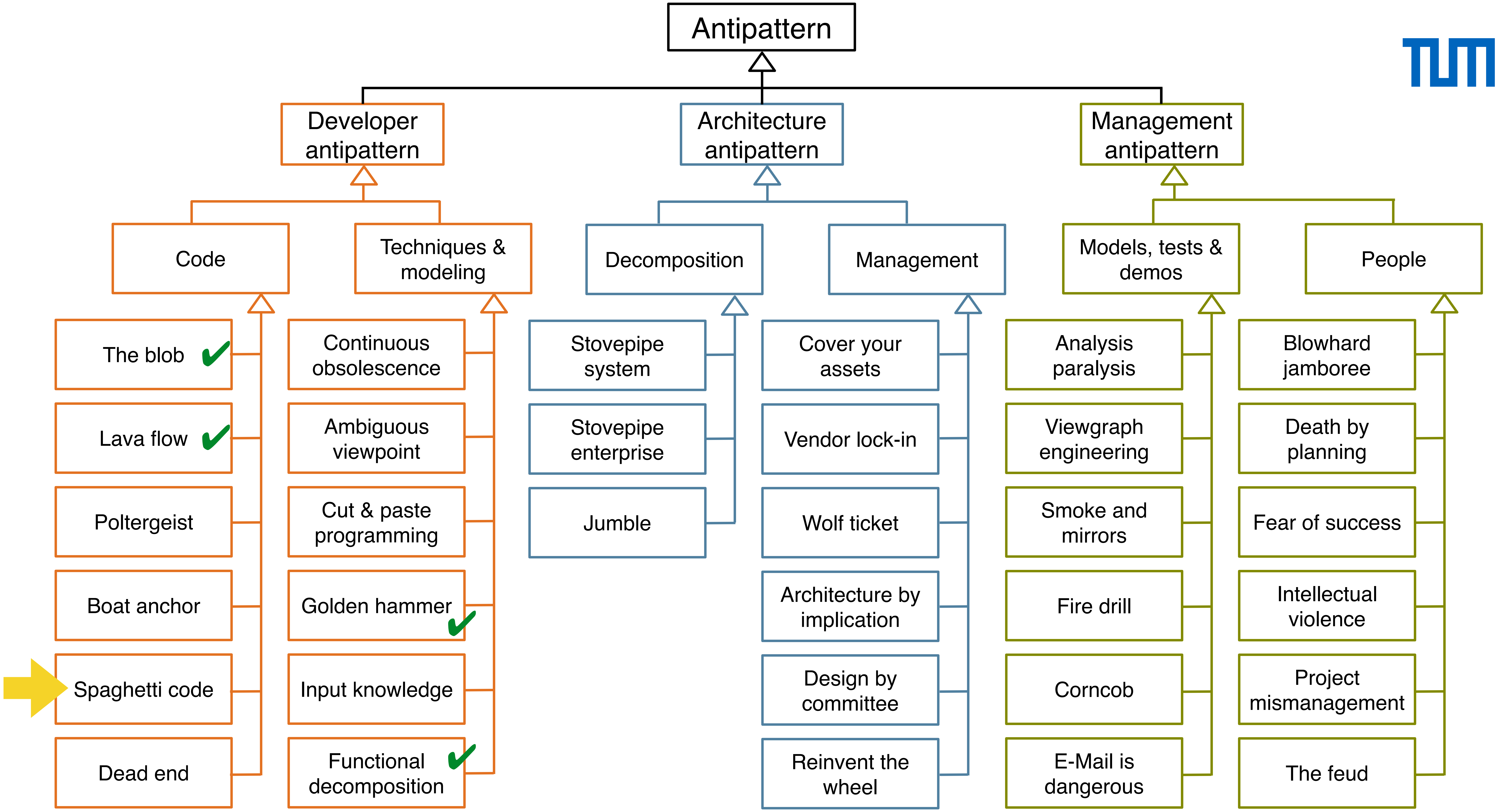
#	Date	Subject
	17.10.22	**No lecture, repetition week (self-study)**
1	24.10.22	Introduction
	31.10.22	**No lecture, repetition week (self-study)**
2	07.11.22	Design Patterns I
3	14.11.22	Design Patterns II
4	21.11.22	Architectural Patterns I
5	28.11.22	Architectural Patterns II
6	05.12.22	Antipatterns I
7	**12.12.22**	**Antipatterns II**
	19.12.22	**No lecture**
8	09.01.23	Testing Patterns I
9	16.01.23	Testing Patterns II
10	23.01.23	Microservice Patterns I
11	30.01.23	Microservice Patterns II
12	08.02.21	Course Review

Roadmap of the lecture

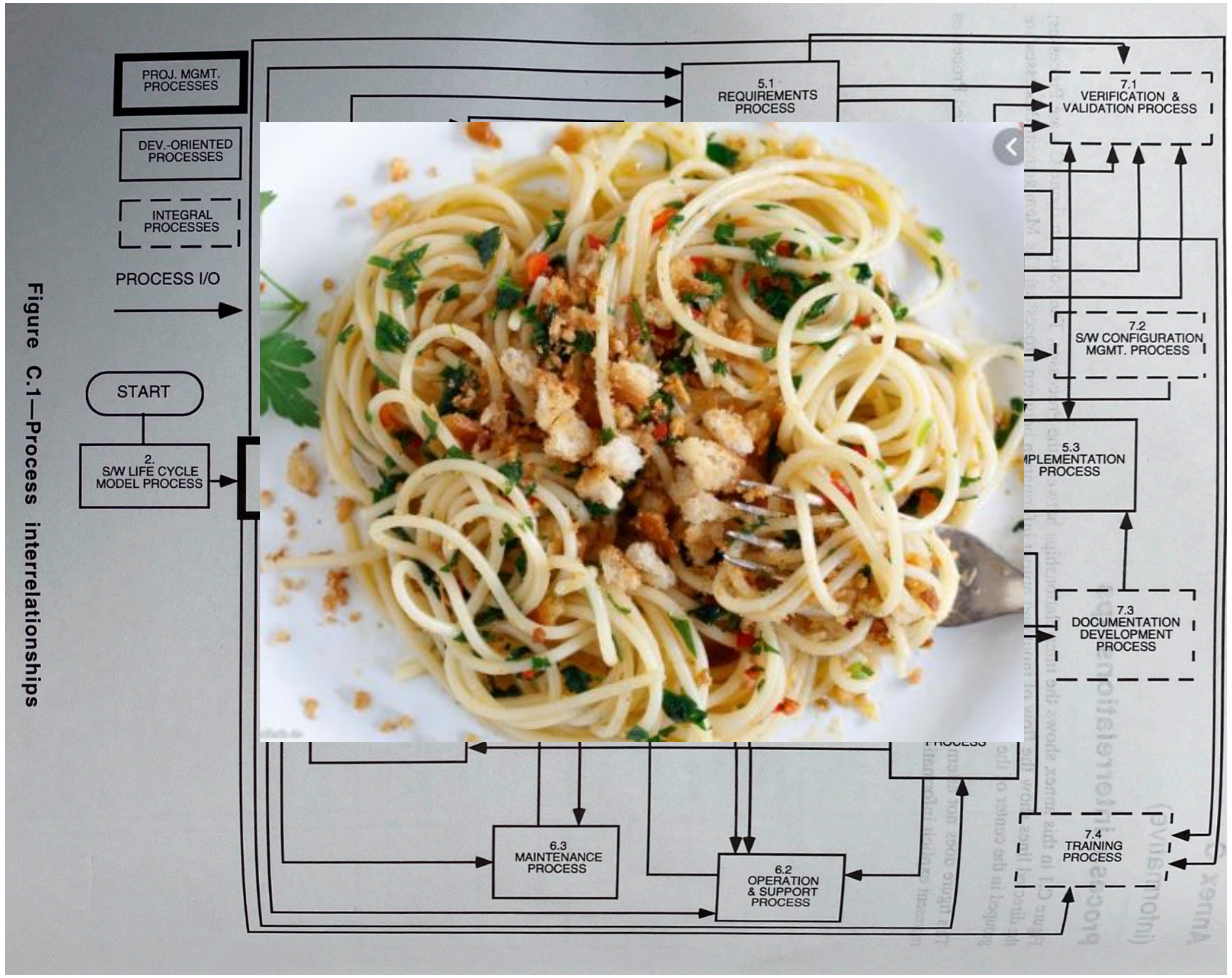
- **Context and assumptions**
  - You have understood the basic concepts of patterns
  - You have implemented many different design and architectural patterns
- **Learning goals: at the end of this lecture you are able to**
  - Identify an antipattern in existing code
  - Refactor an antipattern with an improved solution
  - Differentiate between the covered antipatterns

Outline

- **Antipatterns**
  - Spaghetti code
  - Cut and paste programming
  - Vendor lock-in
  - Analysis paralysis
- Code smells
  - Replace inheritance with delegation
  - Replace conditional with polymorphism
  - Replace error code with exception


Antipattern
Developer antipattern
Architecture antipattern
Management antipattern
Code
Techniques & modeling
Decomposition
Management
Models, tests & demos
People
The blob ✓
Lava flow ✓
Poltergeist
Boat anchor
Spaghetti code
Dead end
Continuous obsolescence
Ambiguous viewpoint
Cut & paste programming
Golden hammer ✓
Input knowledge
Functional decomposition ✓
Stovepipe system
Stovepipe enterprise
Jumble
Cover your assets
Vendor lock-in
Wolf ticket
Architecture by implication
Design by committee
Reinvent the wheel
Analysis paralysis
Viewgraph engineering
Smoke and mirrors
Fire drill
Corncob
E-Mail is dangerous
Blowhard jamboree
Death by planning
Fear of success
Intellectual violence
Project mismanagement
The feud

Software lifecycle model for software development


PROJ. MGMT. PROCESSES
DEV.-ORIENTED PROCESSES
INTEGRAL PROCESSES
PROCESS I/O
START
2. S/W LIFE CYCLE MODEL PROCESS
5.1 REQUIREMENTS PROCESS
7.1 VERIFICATION & VALIDATION PROCESS
7.2 S/W CONFIGURATION MGMT. PROCESS
5.3 IMPLEMENTATION PROCESS
7.3 DOCUMENTATION DEVELOPMENT PROCESS
7.4 TRAINING PROCESS
6.3 MAINTENANCE PROCESS
6.2 OPERATION & SUPPORT PROCESS
Figure C.1—Process interrelationships

Pasta theory of software

Spaghetti code: complicated, difficult to understand, and impossible to maintain software

Lasagna code: simple, understandable, and layered structure however: monolithic and not easy to modify

Ravioli code: ideal structure - small and loosely coupled components can be modified or replaced without significantly affecting other components

Raymond J. Rubey, Letter To The Editor
http://www.gnu.org/fun/jokes/pasta.code.html

Spaghetti code

Spaghetti code

- General form
  - The software has very little structure
  - Object methods are invoked in a single, multistage process flow
- Symptoms and consequences
  - Methods are process oriented, objects are named as processes
  - Execution flow is dictated by the class implementation of the objects, not by the class users
  - Inheritance is not used to provide for extension of the system; polymorphism is not used
  - The source code is difficult to reuse: **point of diminishing returns: the software maintenance effort is greater than a complete reengineering effort**
- Typical causes
  - No design prior to implementation
  - Inexperience with object oriented design technologies

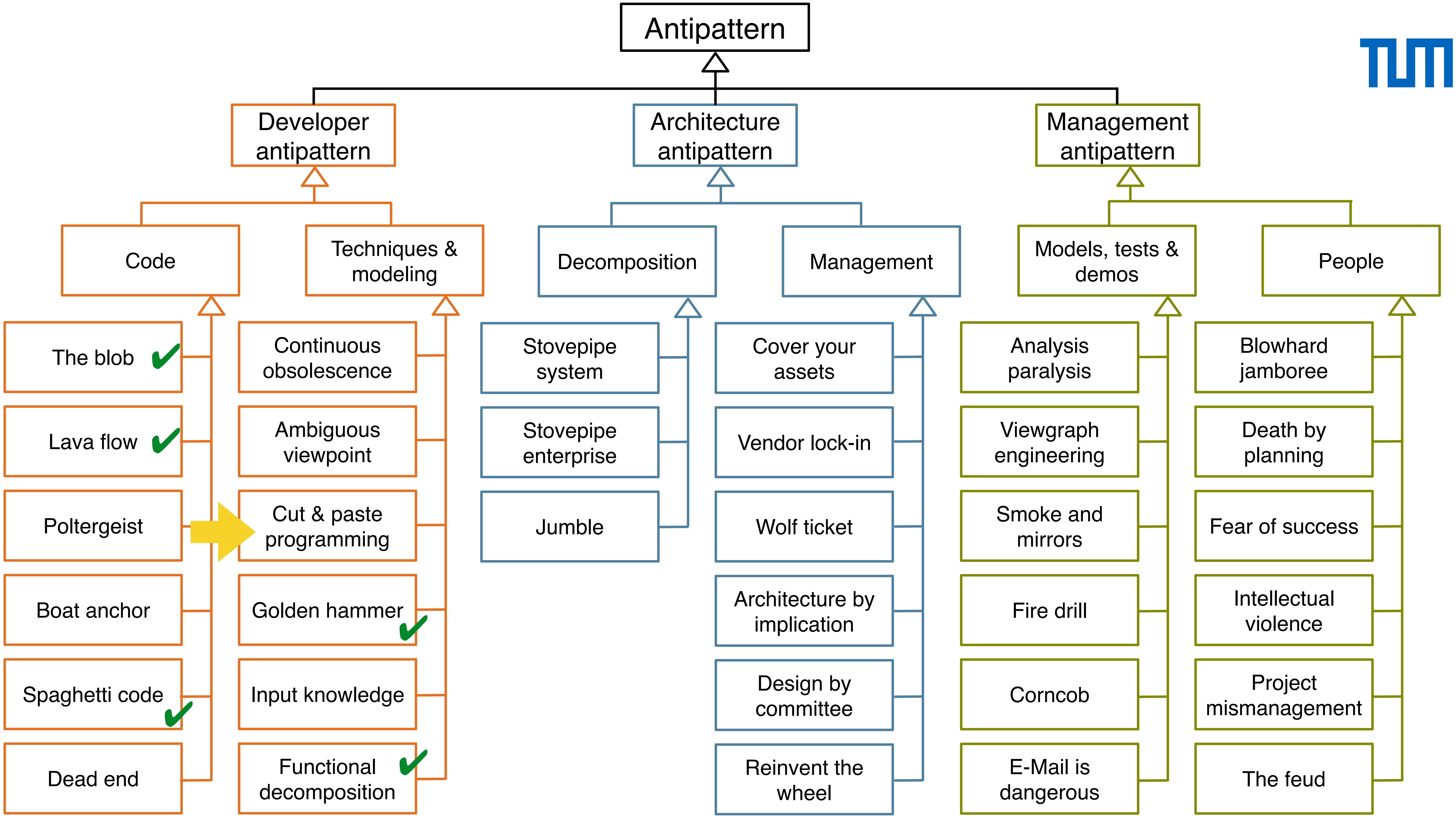
The best way to resolve the spaghetti antipattern is through prevention

“An ounce of prevention is worth a pound of cure.”

— Benjamin Franklin

Refactored solution

- Software refactoring
  - Nicer term for managers: maintenance of software investment
- Incremental (step by step) refactoring
  - Also called incremental reengineering
- Code cleanup should be a natural part of the development process
  - Continuous integration
  - Continuous software engineering
  - Advantage: well structured code provides a longer life cycle to support changes in the business and underlying technology


Antipattern
Developer antipattern
Architecture antipattern
Management antipattern
Code
Techniques & modeling
Decomposition
Management
Models, tests & demos
People
The blob ✓
Lava flow ✓
Poltergeist
Boat anchor
Spaghetti code ✓
Dead end
Continuous obsolescence
Ambiguous viewpoint
Cut & paste programming
Golden hammer ✓
Input knowledge
Functional decomposition ✓
Stovepipe system
Stovepipe enterprise
Jumble
Cover your assets
Vendor lock-in
Wolf ticket
Architecture by implication
Design by committee
Reinvent the wheel
Analysis paralysis
Viewgraph engineering
Smoke and mirrors
Fire drill
Corncob
E-Mail is dangerous
Blowhard jamboree
Death by planning
Fear of success
Intellectual violence
Project mismanagement
The feud

Cut and paste programming

- Also known as: clipboard coding, software cloning, software propagation
- Root causes: sloth
- Unbalanced forces: management of resources, technology transfer
- Anecdotal evidence
  - "Hey, I thought you fixed that bug already, so why is it doing this again?”
  - "You work fast. Over 400,000 lines of code in three weeks is outstanding progress!"

Background

- It's easier to modify existing software (e.g. small code snippets found on StackOverflow) than solve a problem from scratch
- This is usually correct and represents good software instincts
- However, the technique can be easily over used

Symptoms and consequences

- Software defects are replicated through the system
    - The same software bug reoccurs despite many local fixes
    - Developers create multiple unique fixes for bugs with no method of resolving the variations into a standard fix
- Lines of code increase without adding to overall productivity
- Code reviews and inspections are needlessly extended
- It becomes difficult to locate and fix all instances of a particular mistake
- Code can be reused with a minimum of effort
- **Excessive software maintenance costs**

Typical causes

- Short term payoff more important than long term investment
- **No reward of reusable components**: development speed overshadows all other evaluation factors (lines of code is the wrong metric!)
- **Lack of abstraction** among developers, often accompanied by a poor understanding of inheritance, composition, and other reuse possibilities
- Reusable components, once created, are not sufficiently documented or made readily available to developers
- “Not invented here” syndrome
- **Lack of forethought or forward thinking** among the development teams
- Inexperience with new technology or tools → use a working example and modify it, adapting it to specific needs

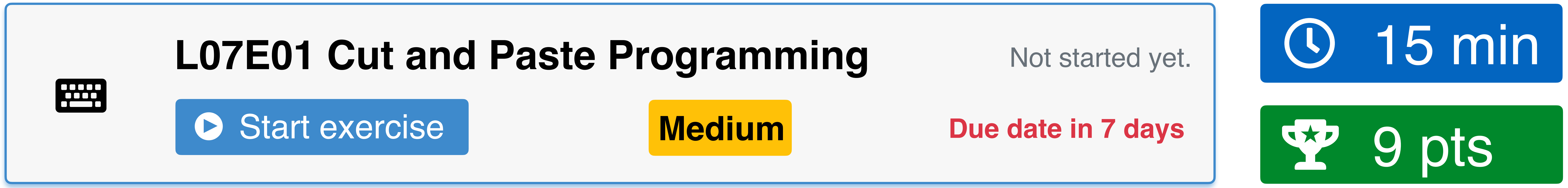
Refactored solutions

- Common method refactoring
  - Typically not an issue, at least for experienced developers
- **White box reuse** (inheritance)

  Modeled with △
  - A new subclass reuses the functionality of the superclass and may offer new functionality

  ➡ Can also be a symptom if used too often
- **Black box reuse** (delegation)

  Modeled with ◇ and ◆
  - A new class offers the aggregated functionality of the existing classes
  - Key benefits
    - The implementation of an object can be made independent of the object's interface
    - Take advantage of late binding by mapping an interface to a specific implementation at run time
    - Objects can use an interface (e.g. `List`), yet benefit over time from more advanced implementations (e.g. `ArrayList`, `LinkedList`) that support the same interface


L07E01 Cut and Paste Programming

Not started yet.

Start exercise

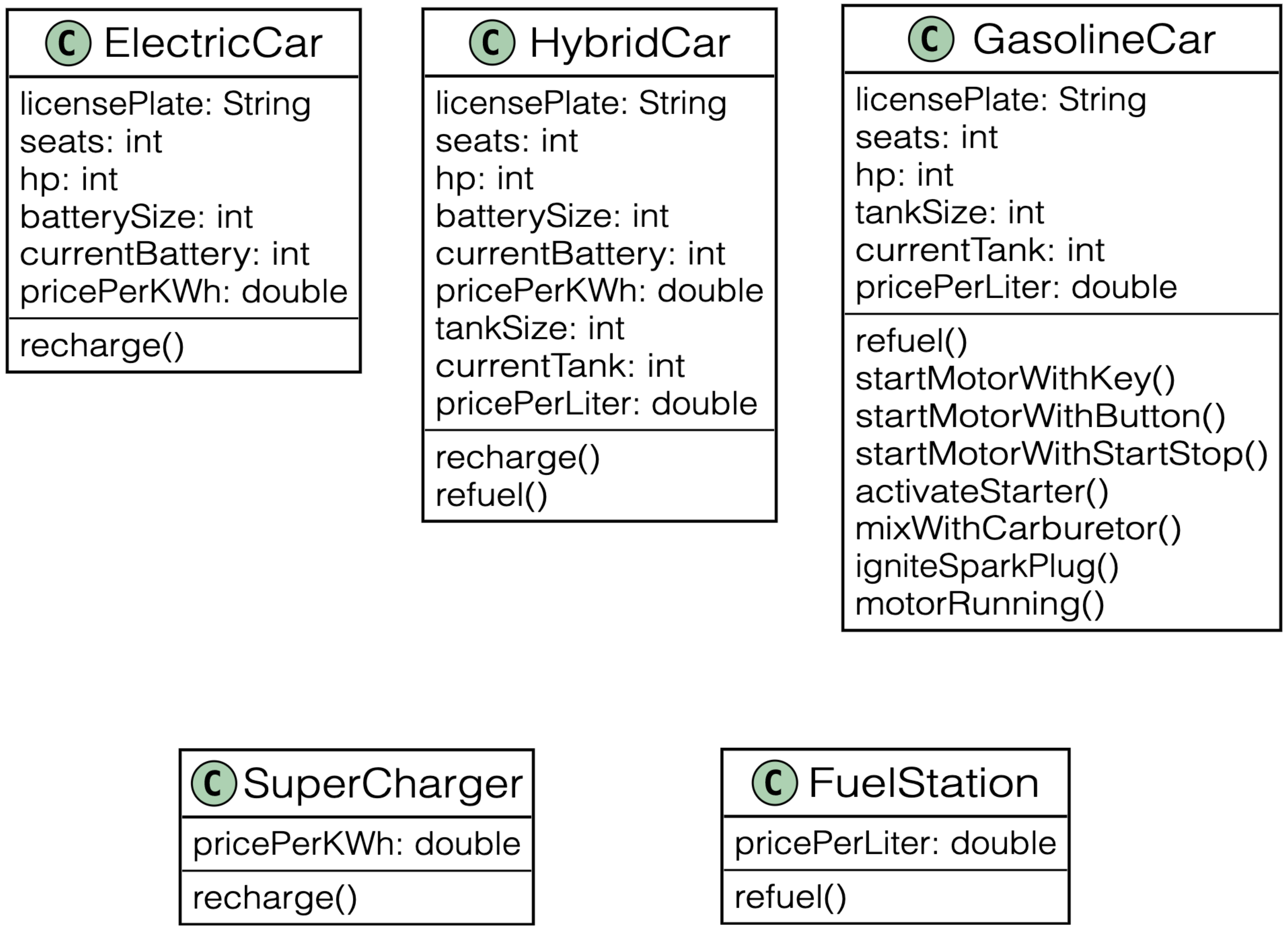
Medium

Due date in 7 days

15 min

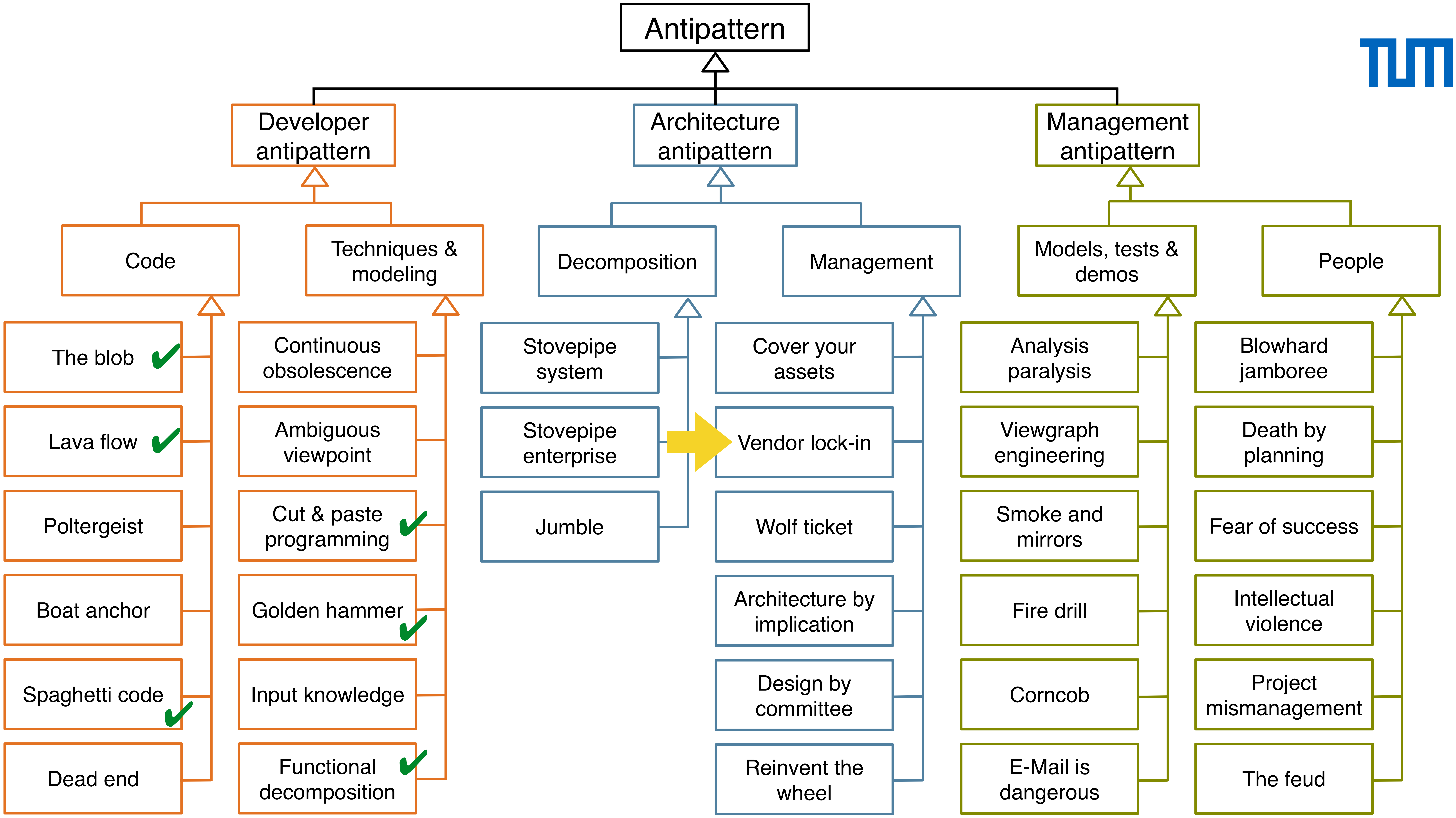
9 pts

- Problem statement: **fueling cars**
  - Common method refactoring
  - White box reuse
  - Black box reuse



Outline

- Antipatterns
  - Spaghetti code
  - Cut and paste programming
  - ➡ **Vendor lock-in**
  - Analysis paralysis
- Code smells
  - Replace inheritance with delegation
  - Replace conditional with polymorphism
  - Replace error code with exception


Antipattern
Developer antipattern
Architecture antipattern
Management antipattern
Code
Techniques & modeling
Decomposition
Management
Models, tests & demos
People
The blob ✓
Lava flow ✓
Poltergeist
Boat anchor
Spaghetti code ✓
Dead end
Continuous obsolescence
Ambiguous viewpoint
Cut & paste programming ✓
Golden hammer ✓
Input knowledge
Functional decomposition ✓
Stovepipe system
Stovepipe enterprise
Jumble
Cover your assets
Vendor lock-in
Wolf ticket
Architecture by implication
Design by committee
Reinvent the wheel
Analysis paralysis
Viewgraph engineering
Smoke and mirrors
Fire drill
Corncob
E-Mail is dangerous
Blowhard jamboree
Death by planning
Fear of success
Intellectual violence
Project mismanagement
The feud

Vendor lock-in

Vendor lock-in

- General form: a software project adopts a product technology and becomes completely dependent upon the vendor's implementation
- Symptoms and consequences
  - Commercial product upgrades drive the application software maintenance cycle
  - Promised product features are delayed or never delivered, subsequently, causing failure to deliver application updates
  - Application programming requires in-depth product knowledge
- Typical causes
  - The product is selected based entirely upon marketing and sales information, and not upon more detailed technical inspection
  - The product varies from published open system standards because there is no effective conformance process for the standard

Vendor lock-in

- Known exceptions: a single vendor's code makes up the majority of code needed in an application
- **Refactored solution**: isolation layer to the vendor software (closed architecture)
  - Level of abstraction between application software and lower-level infrastructure
  - Provides software portability from underlying middleware and platform-specific interfaces
  - Separation of infrastructure knowledge from application knowledge
  - Reduction of the risks and costs of infrastructure changes
- Related solutions: adapter design pattern

L07E02 Vendor Lock-in

Start exercise

Medium

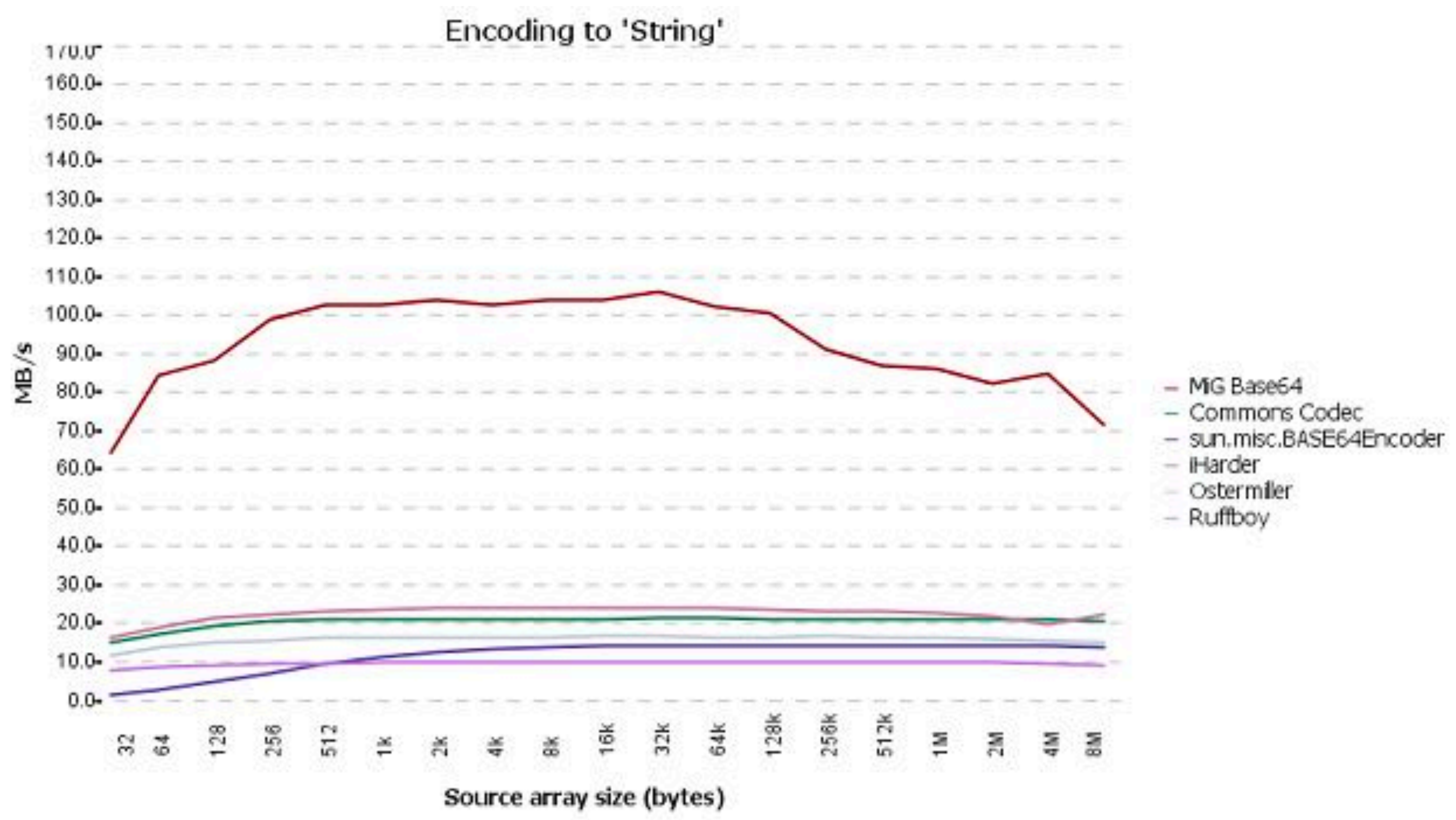
Not started yet.

Due Date: in 7 days

10 min

6 pts

- Problem statement: **Apache, Apache, Apache**
    - Apache Common Codec's Base64 —> MiGBase64
    - Apache's Crypto Streams —> jasypt



Analysis paralysis

Analysis paralysis

- General form
  - Goal to achieve perfection and completeness of the analysis phase
  - Generation of very detailed models
- Assumptions
  - Everything about the problem can be known a priori
  - Detailed analysis can be successfully completed prior to coding
  - Analysis model will not be extended nor revisited during development
- Symptoms and consequences
  - Cost of analysis exceeds expectation without a predictable end point
  - Analysis documents no longer make sense to the domain experts
- Typical causes
  - Management (and often the customer) assumes a **waterfall** progression of phases
  - Goals in the analysis phase are **not well defined**

Everything about the problem **can** be known a priori

Can we always hit the target?

What if the target is far away?

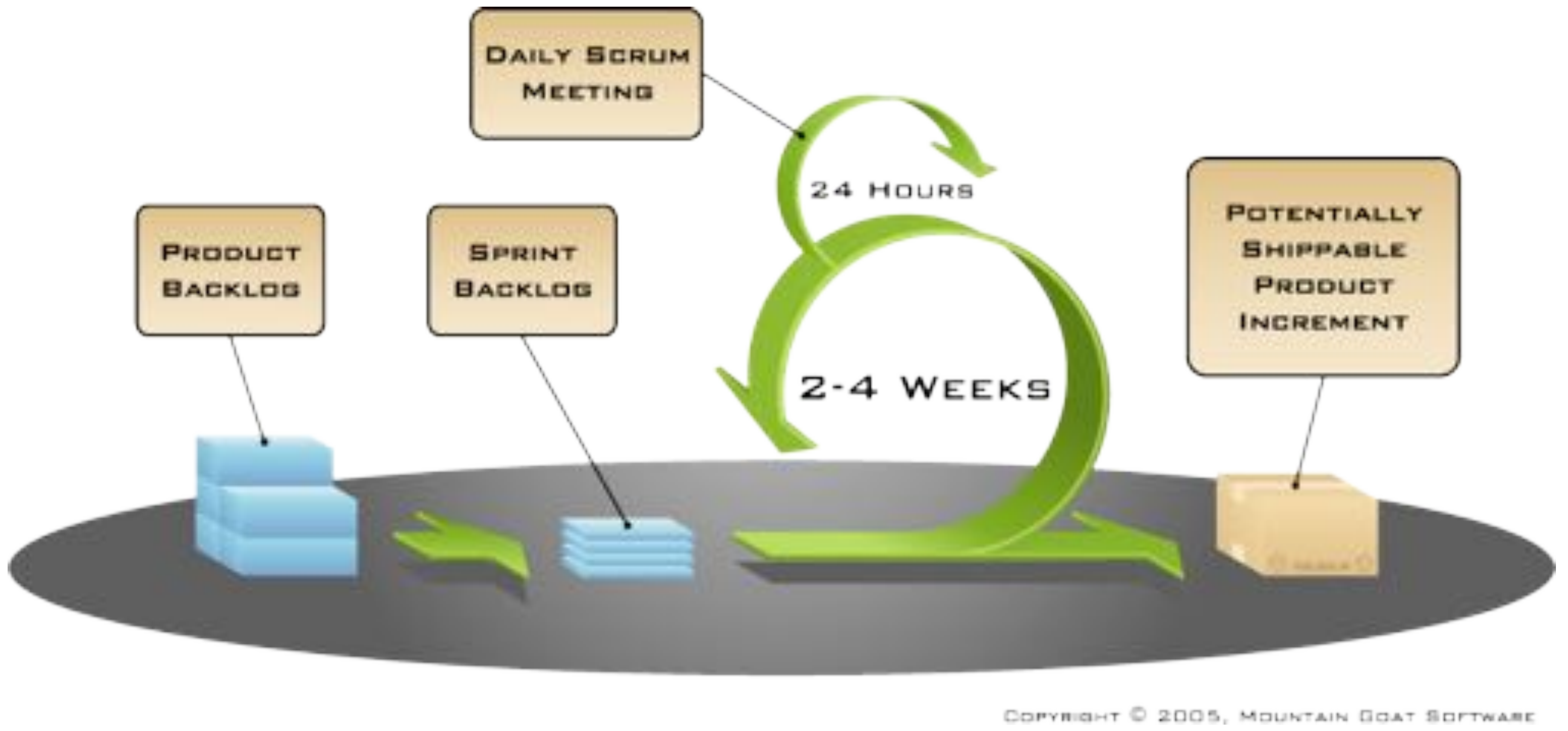
Everything about the problem **cannot** be known a priori

Can we always hit the target?

What about impediments?

Refactored solution

- Assume that details of the problem are not all known a priori and its solution will be learned in the course of the development process
- Use vertical prototyping → agile methods (e.g. Scrum: sprint based development)
- Allow incremental, iterative as well as adaptive development


Daily Scrum Meeting
24 Hours
Product Backlog
Sprint Backlog
2-4 Weeks
Potentially Shippable Product Increment
Copyright © 2005, Mountain Goat Software

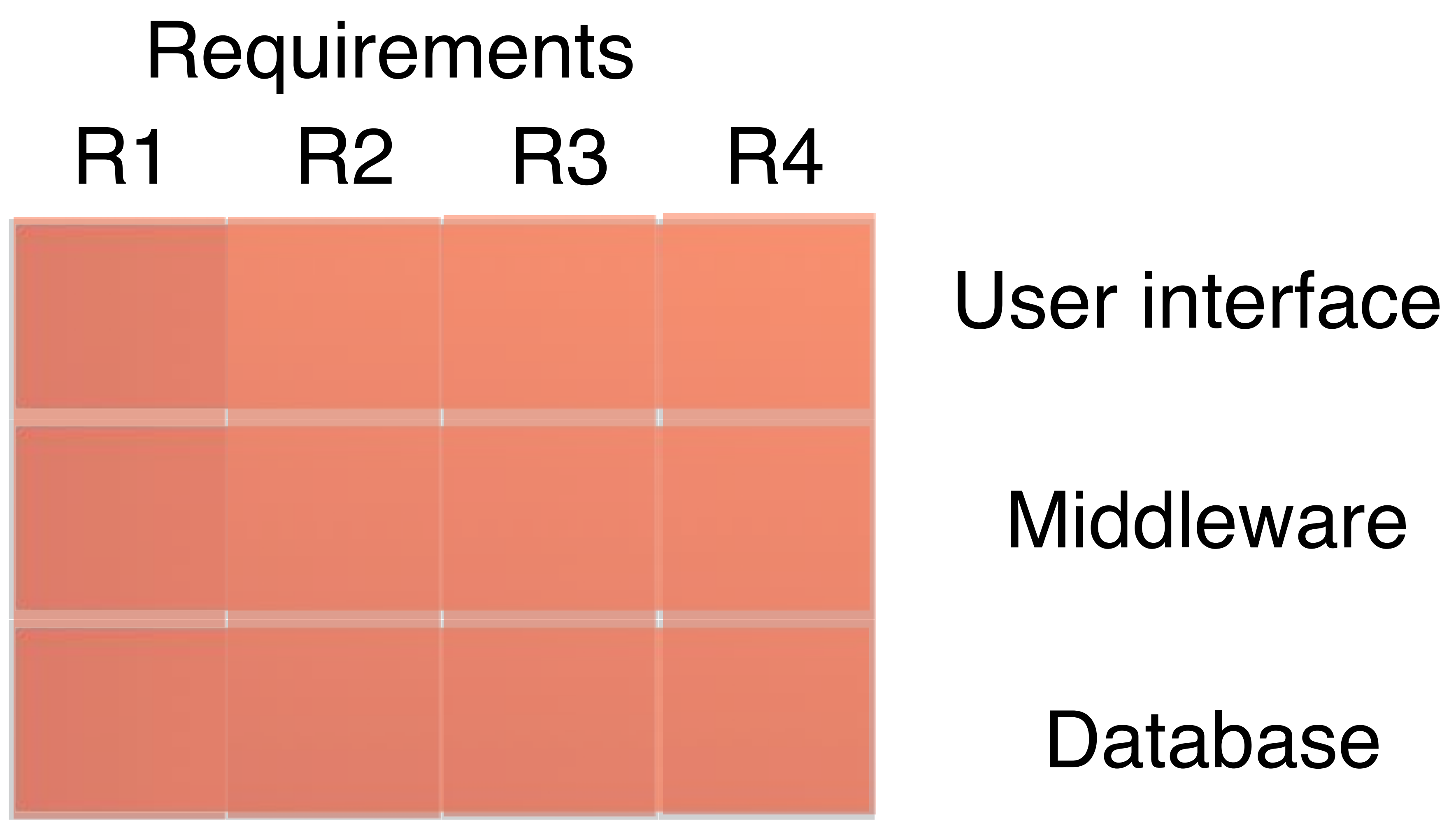
Incremental, iterative and adaptive development

- Incremental means to “**add onto something**”
  - Incremental development helps to improve the process

Bottom-up integration

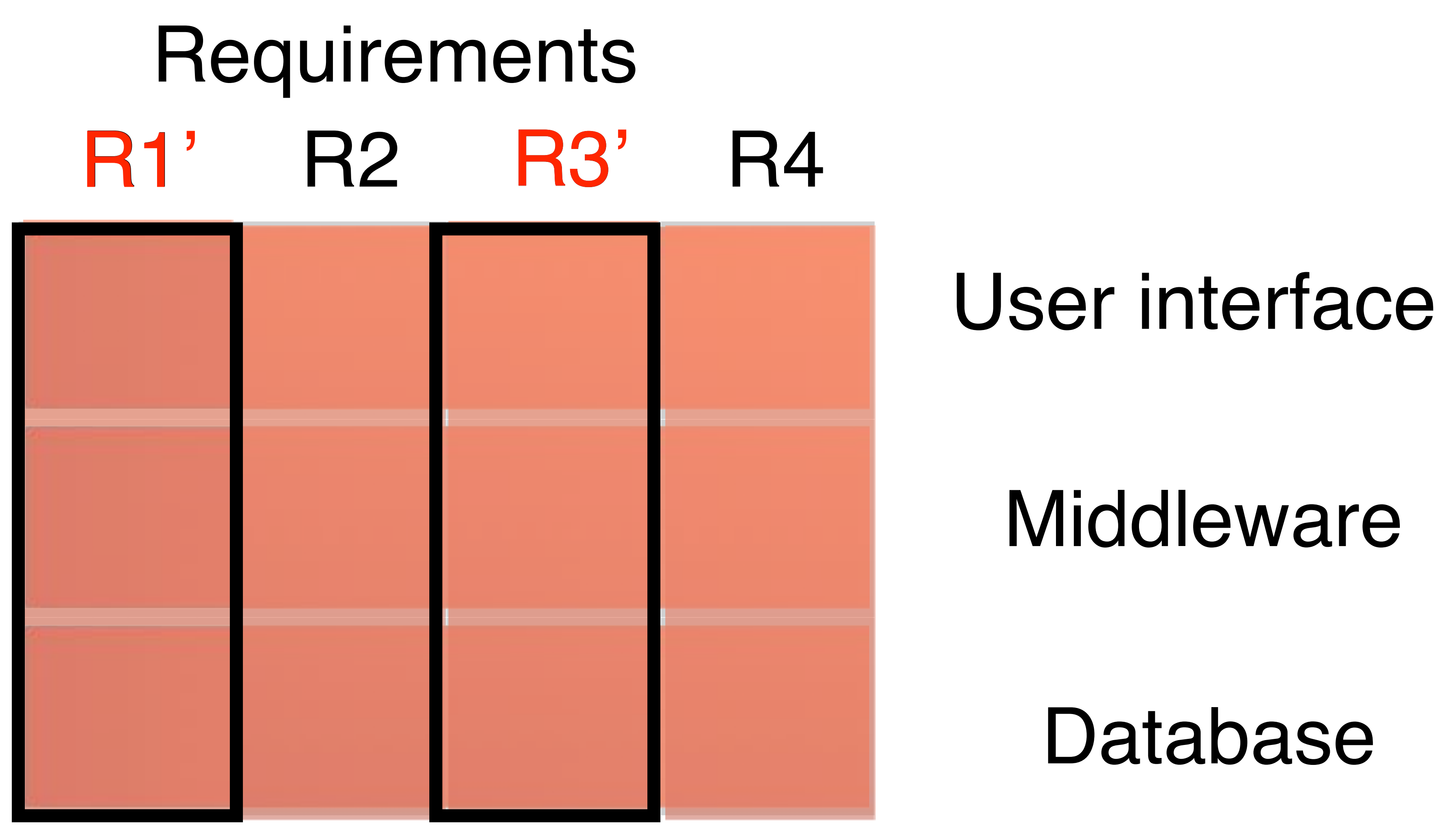
Top-down integration

Vertical integration



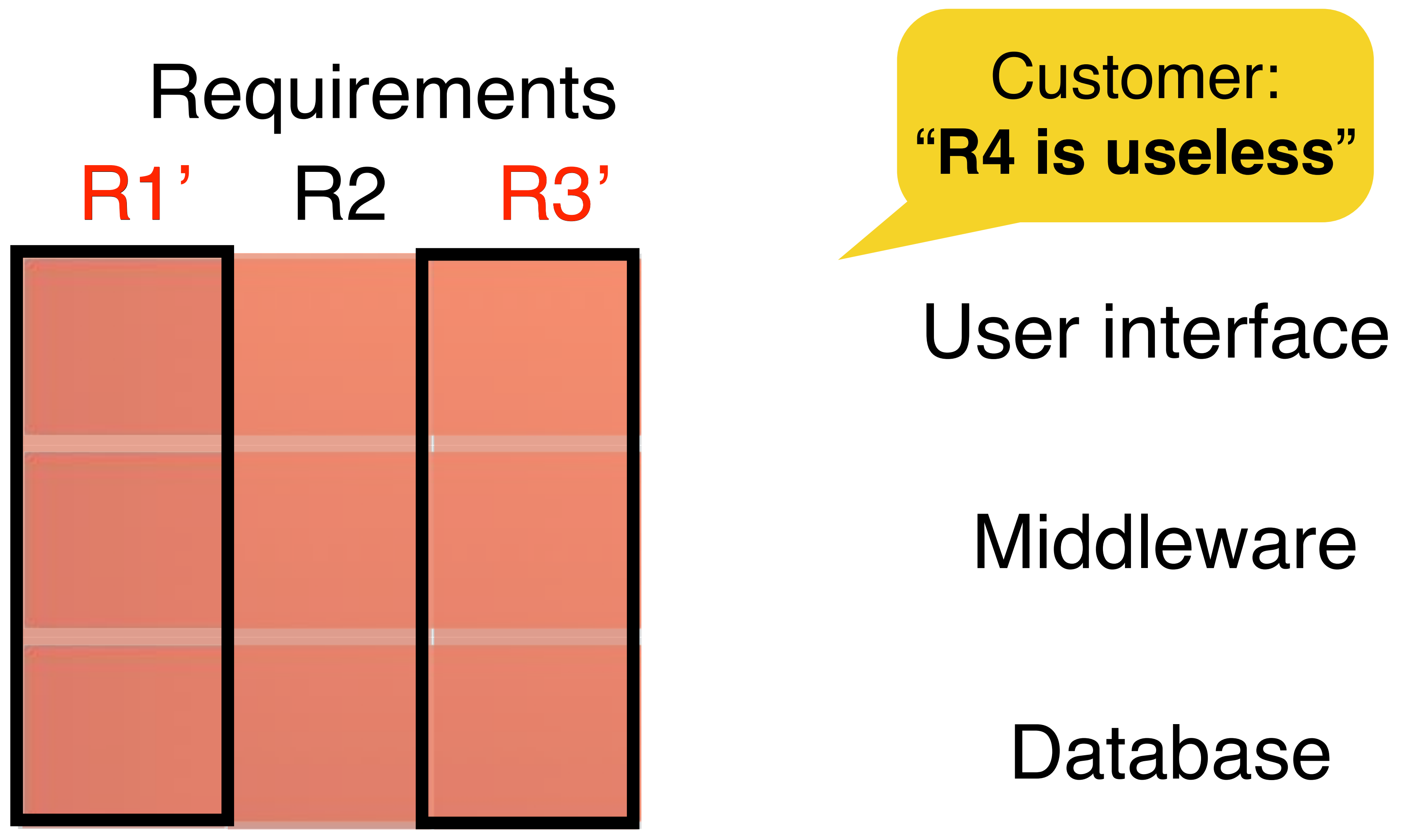
Incremental, iterative and adaptive development

- Incremental means to “**add onto something**”
  - Incremental development helps to improve the process
- Iterative means to “**re-do something**”
  - Iterative development debugs and improves your product

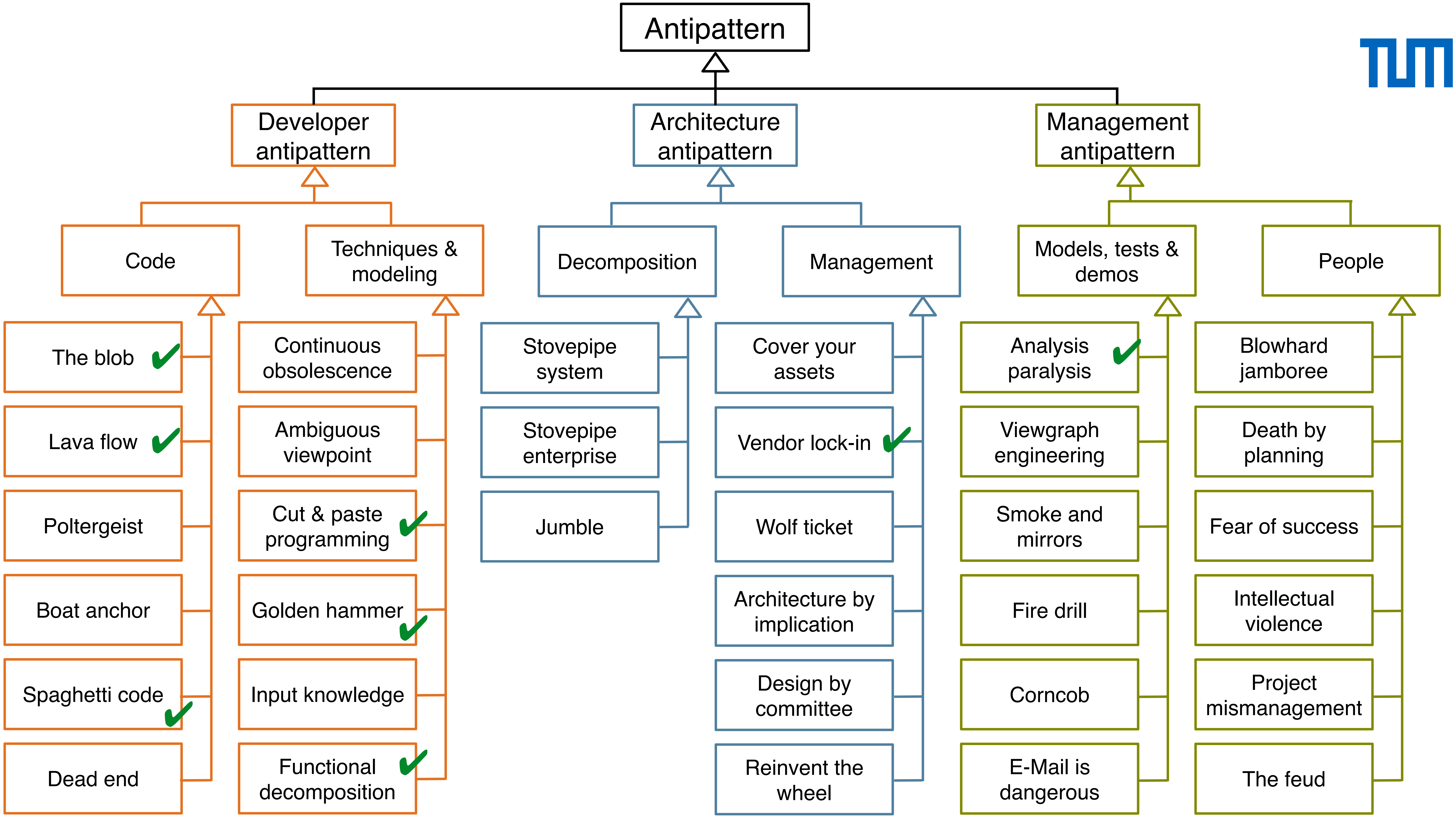


Incremental, iterative and adaptive development

- Incremental means to "**add onto something**"
  - Incremental development helps to improve the process
- Iterative means to "**re-do something**"
  - Iterative development debugs and improves your product
- Adaptive means to "**react to changing requirements**"
  - Adaptive development improves the reaction to changing customer needs

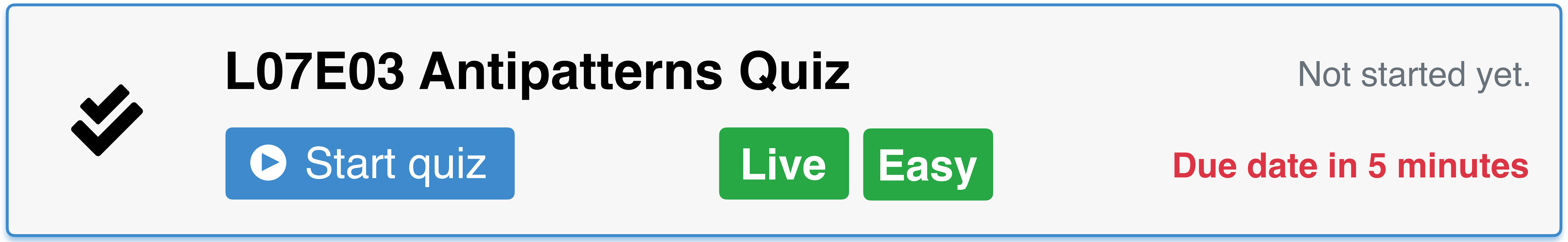


"People don't know what they want until you show it to them." - Steve Jobs.


Antipattern
Developer antipattern
Architecture antipattern
Management antipattern
Code
Techniques & modeling
Decomposition
Management
Models, tests & demos
People
The blob ✓
Lava flow ✓
Poltergeist
Boat anchor
Spaghetti code ✓
Dead end
Continuous obsolescence
Ambiguous viewpoint
Cut & paste programming ✓
Golden hammer ✓
Input knowledge
Functional decomposition ✓
Stovepipe system
Stovepipe enterprise
Jumble
Cover your assets
Vendor lock-in ✓
Wolf ticket
Architecture by implication
Design by committee
Reinvent the wheel
Analysis paralysis ✓
Viewgraph engineering
Smoke and mirrors
Fire drill
Corncob
E-Mail is dangerous
Blowhard jamboree
Death by planning
Fear of success
Intellectual violence
Project mismanagement
The feud

Intermediate summary: covered antipatterns

- **Functional decomposition**: design and implementation style from people, who got their training in structured analysis methods and imperative languages, with no or little experience in object-oriented methods or languages
- **Lava flow**: dead code and forgotten design information in a constantly changing design/implementation
- **Blob**: one object has the lion's share of the responsibilities, other objects do only simple things
- **Golden hammer**: one specific technology is obsessively applied to all problems
- **Spaghetti code**: ad hoc structure which makes it hard to extend or optimize a model or code
- **Cut and paste programming**: excessive code duplication (clones) make the software expensive and difficult to maintain
- **Vendor lock-in**: dependence on a proprietary architecture or tool set, making it hard to switch to another vendor
- **Analysis paralysis**: spending excessive time in requirements elicitation and analysis


L07E03 Antipatterns Quiz

Not started yet.

Due date in 5 minutes

5 min

2 pts

- Answer two questions about antipatterns
- Select all correct answer choices

Outline

- Antipatterns
  - Spaghetti code
  - Cut and paste programming
  - Vendor lock-in
  - Analysis paralysis
- **Code smells**
  - Replace inheritance with delegation
  - Replace conditional with polymorphism
  - Replace error code with exception

Code smells and source code refactoring

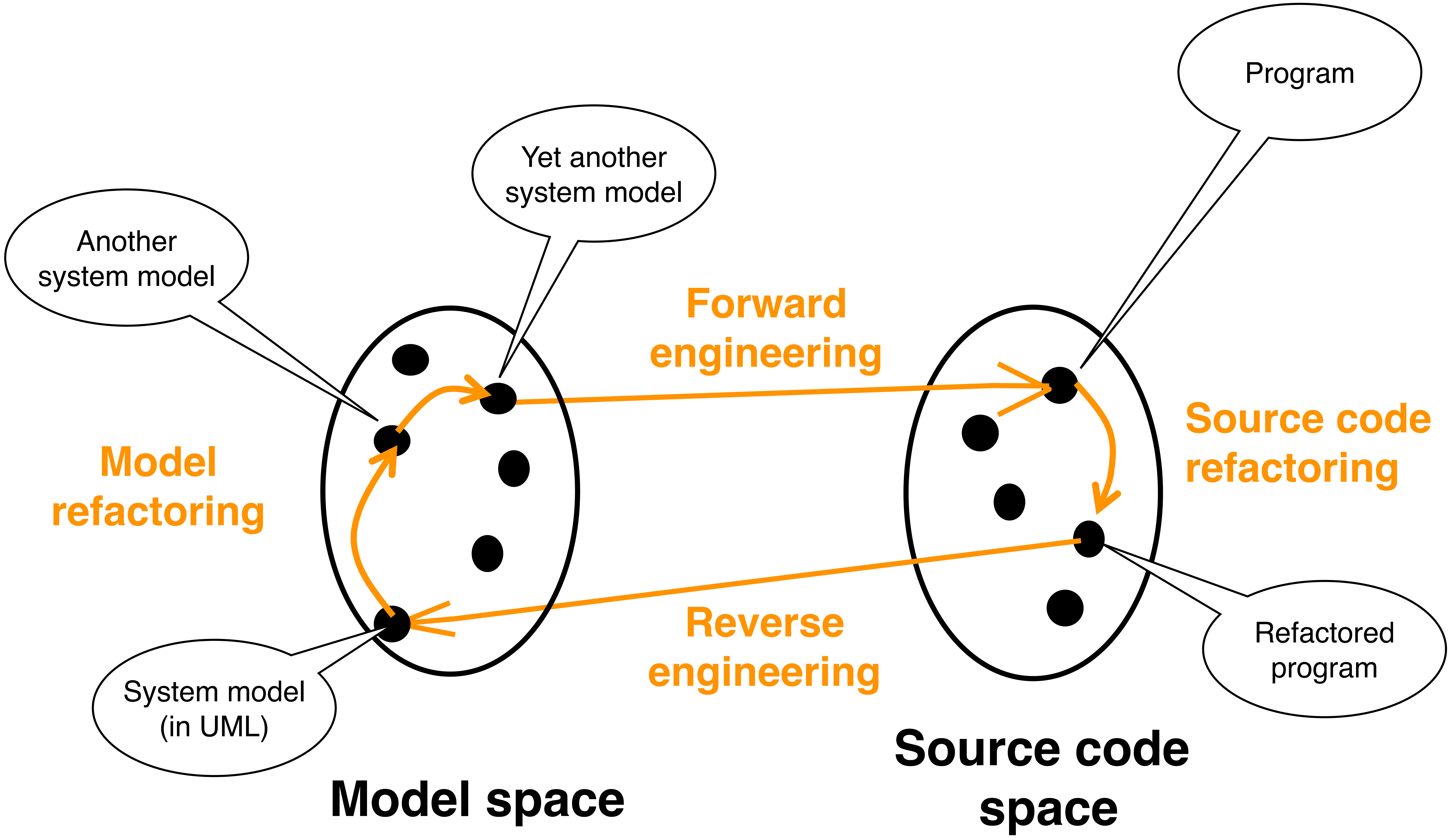
“If it stinks, change the diaper!”

— Grandma Beck, discussing child-rearing philosophy

Code smell

- First coined by Kent Beck and Martin Fowler
- Any symptom in a program that possibly indicates a bigger problem
- A heuristic that indicates when to refactor, and what specific technique to use
- **Code smell pattern**
  - Problem in the source code (solution domain)
  - Solution: source code refactoring
- **Smell pattern**
  - Problem in the system model (application domain)
  - Solution: model refactoring
- Source code refactoring and modeling refactoring are examples of transformations

Types of transformations

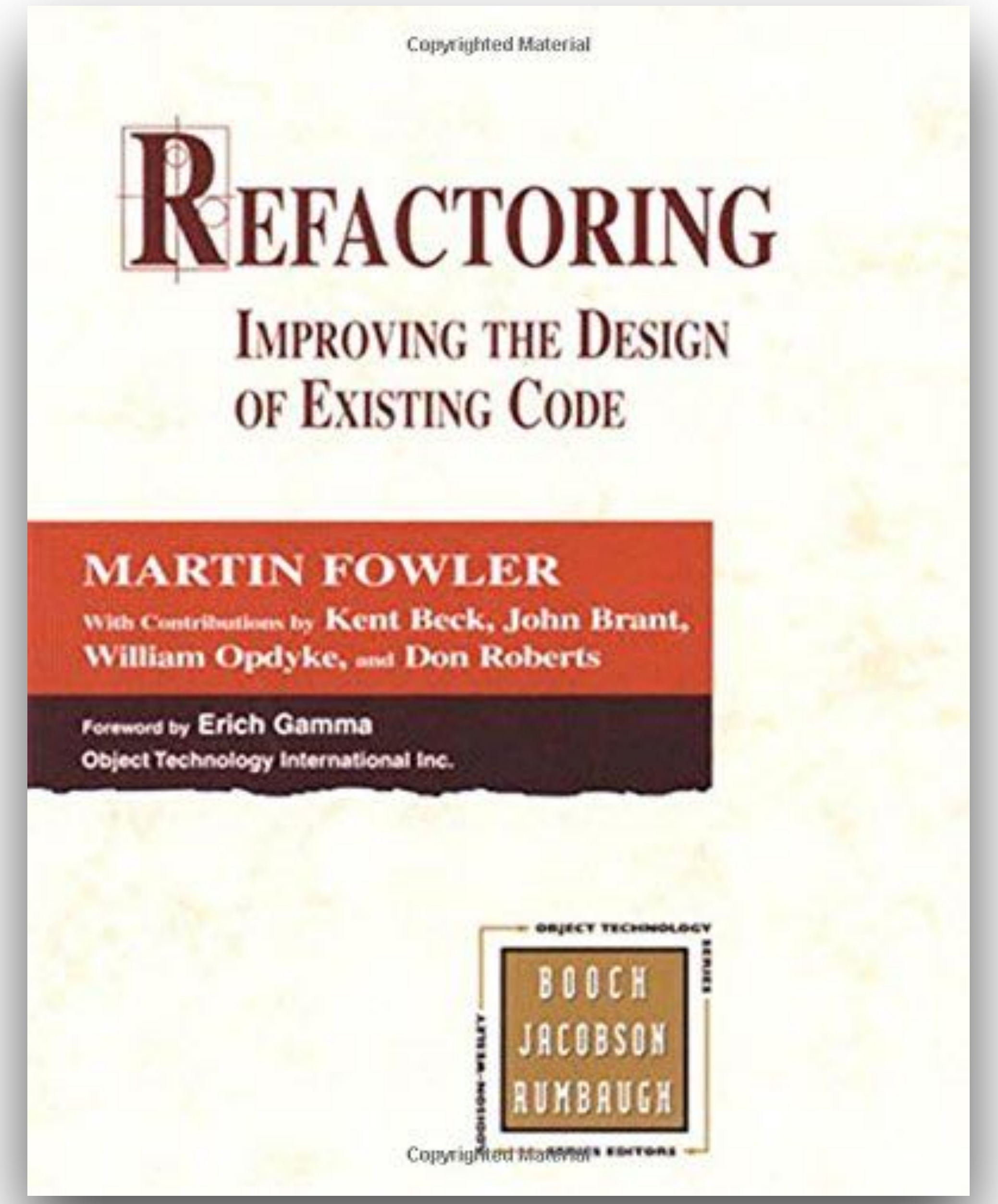


Examples of code smells

- **Smell:** method too long
  - Problem: method is hard to read
  - Refactored solution: extract method
- **Smell:** duplicated code
  - Problem: programmer was lazy
  - Refactored solutions: extract method, pull up method, extract class
- **Smell:** class too large
  - Problem: class is hard to understand
  - Refactored solution: extract superclass
- **Smell:** parameter list too long
  - Problem: signature hard to understand
  - Refactored solution: replace parameter with explicit methods, introduce parameter object
- **Smell:** feature envy
  - Problem: class uses methods of another class excessively
  - Refactored solution: move method
- **Smell:** lazy class
  - Problem: class has no interesting behavior
  - Refactored solution: turn the class into an attribute
- **Smell:** speculative generality
  - Problem: excessive use of inheritance, usually due to analysis paralysis
  - Refactored solution: collapse the inheritance hierarchy
- **Smell:** refused bequest
  - Problem: subclass is reusing behavior of the superclass, but does not want to support the superclass interface for the class user
  - Refactored solution: replace inheritance with delegation

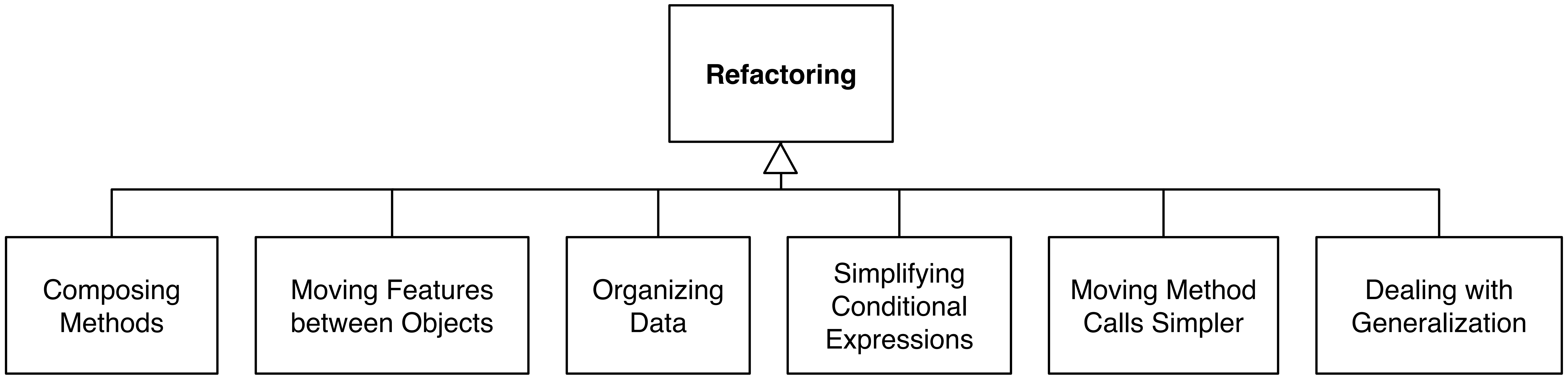
Code smells references

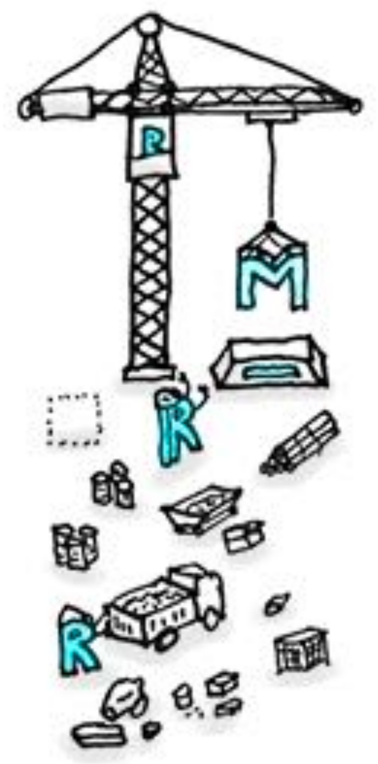
- Book

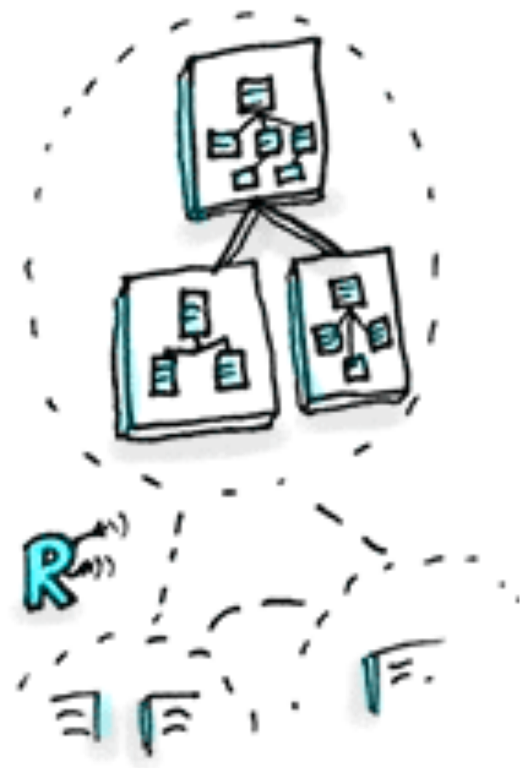

Copyrighted Material
REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH
JACOBSON
RUMBAUGH

- Online version of Fowlers book
  - http://martinfowler.com/refactoring/catalog/index.html
  - German version: http://www.tutego.de/java/refactoring/catalog/
- Source code refactoring taxonomy
  - http://sourcemaking.com/refactoring

Refactoring techniques

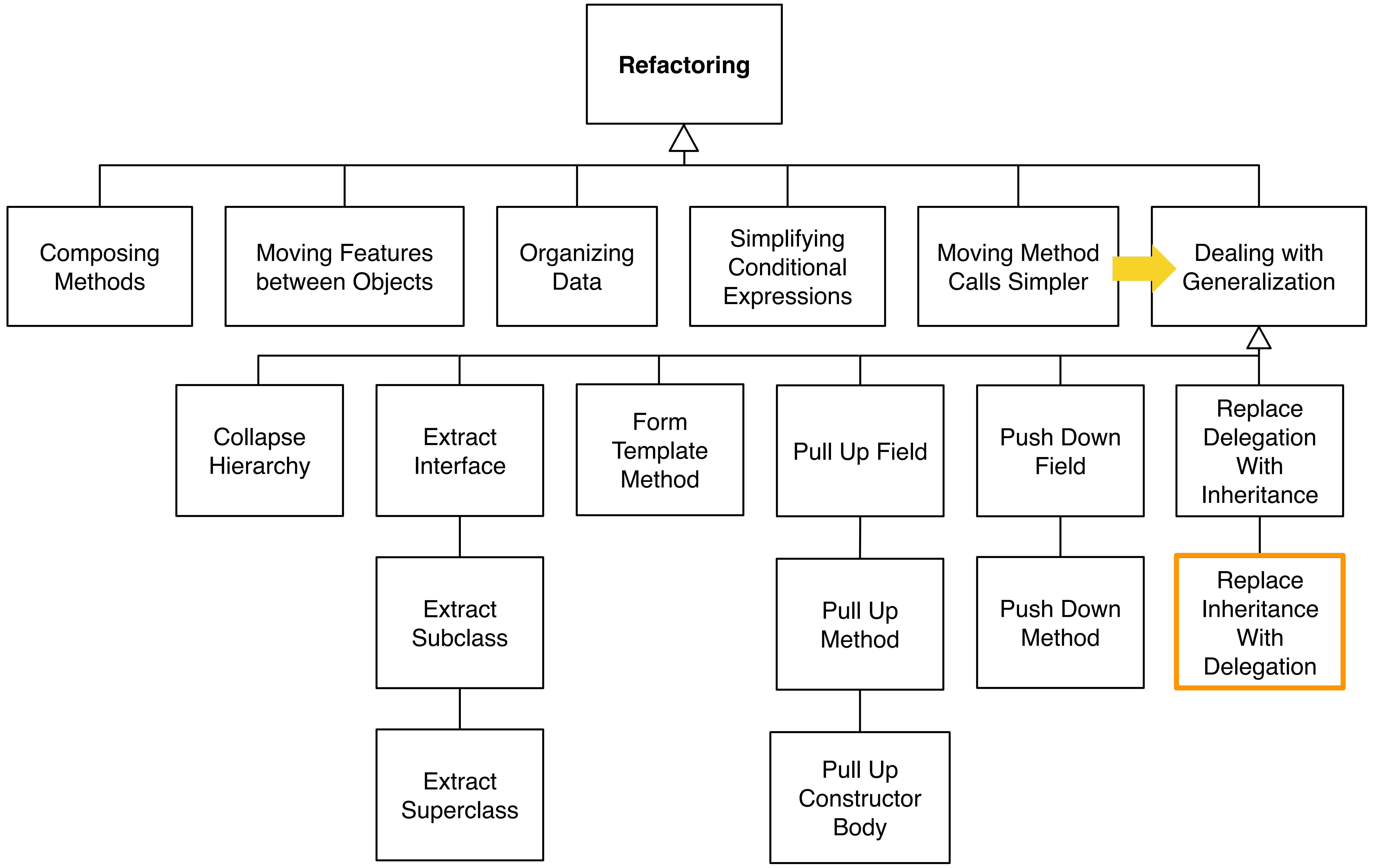

Refactoring
Composing Methods
Moving Features between Objects
Organizing Data
Simplifying Conditional Expressions
Moving Method Calls Simpler
Dealing with Generalization

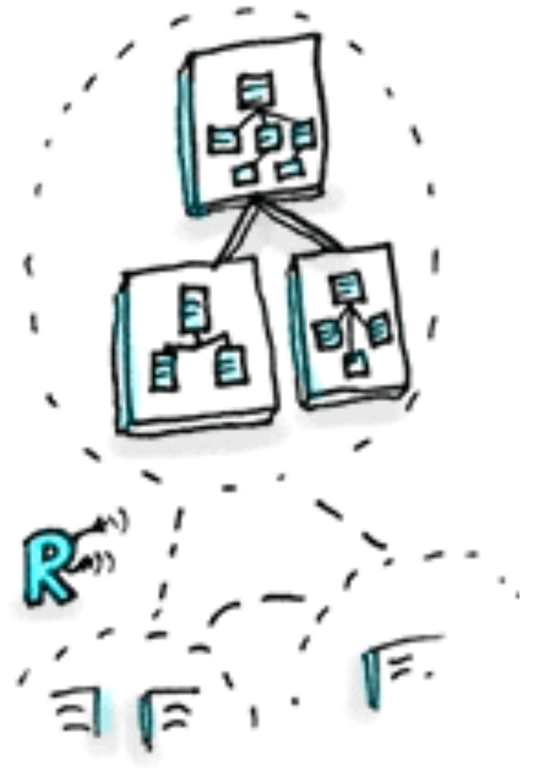




http://sourcemaking.com/refactoring

Dealing with generalization

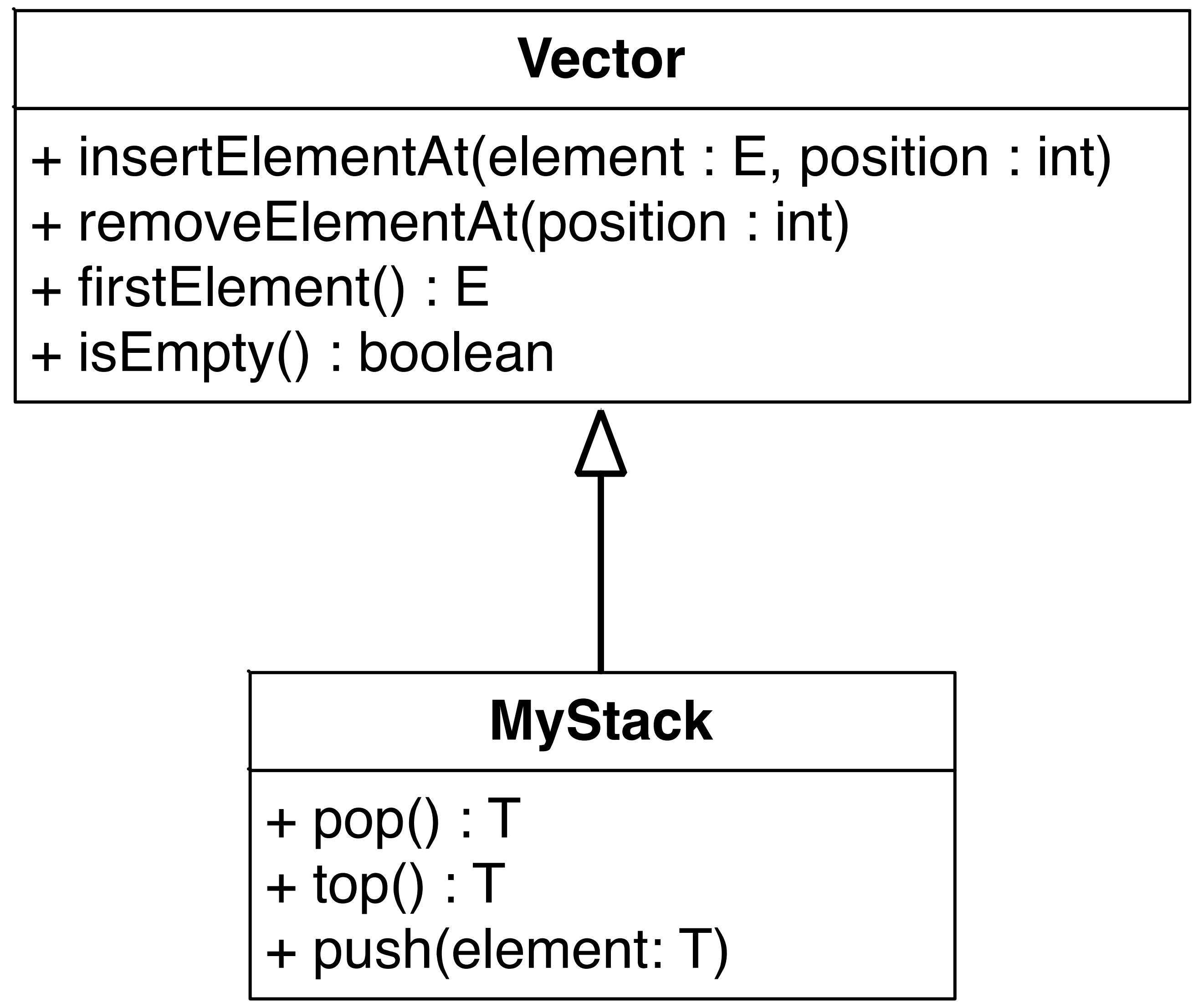




Replace inheritance with delegation: motivation

- Problem: a subclass uses only part of a superclass interface or does not want to inherit data
  - This results in source code that says one thing when your intention is something else—a confusion you should remove
- Refactored solution: replace inheritance with delegation
  - Make it clear that you are making only partial use of the delegated class

Example

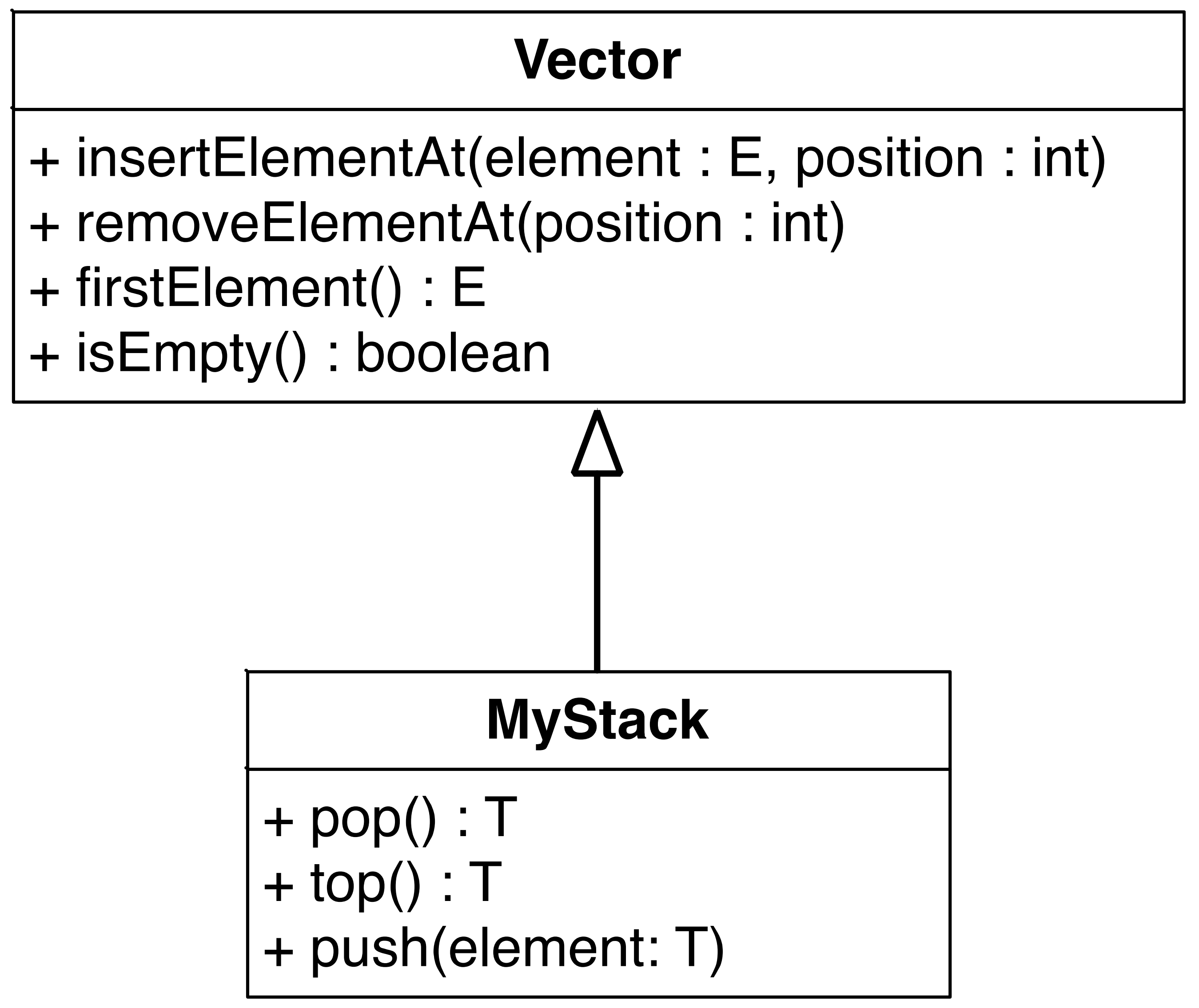


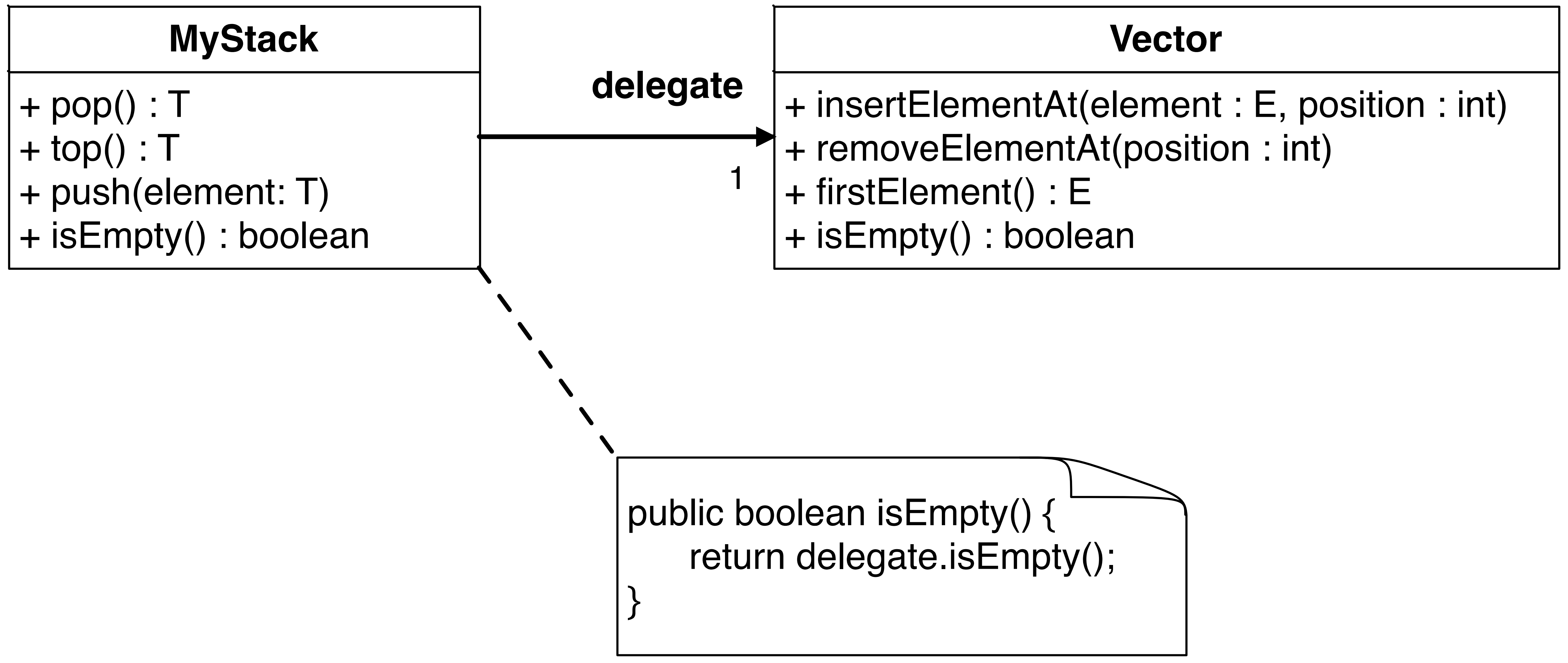
```
public class MyStack<T> extends Vector<T> {


}
```

Inheritance in this case is a bad choice because:

- The methods of **Vector** can be overridden by **MyStack,** which leads to unexpected behavior
- The methods of the super class **Vector** are public, so they are visible to a caller using **MyStack**, so the caller could remove elements in the middle of the stack

Refactored solution



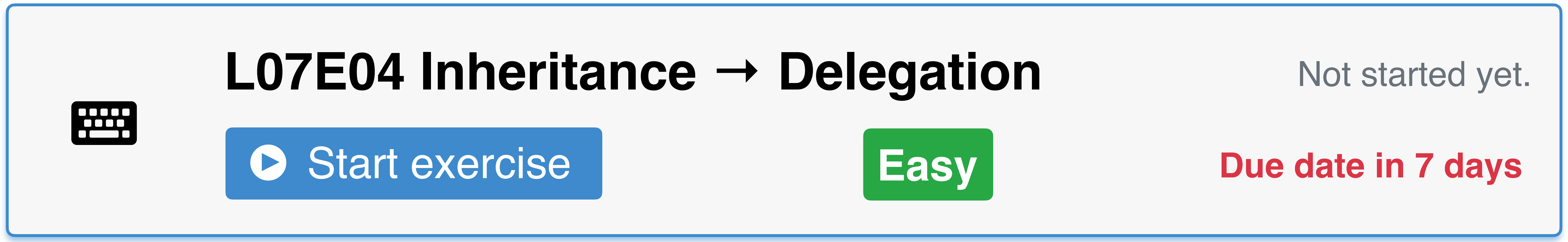


```
public boolean isEmpty() {
     return delegate.isEmpty();
}
```

```
public class MyStack<T> extends Vector<T> {
    . . .
}
```

```
public class MyStack<T> {
    private Vector<T> delegate = new Vector<T>();
    . . .
}
```

Delegation is a better solution because the methods of class **`Vector`** have been used to implement **`MyStack`**, but this fact is invisible to clients calling the **`MyStack`** methods



L07E04 Inheritance → Delegation

Not started yet.

Start exercise

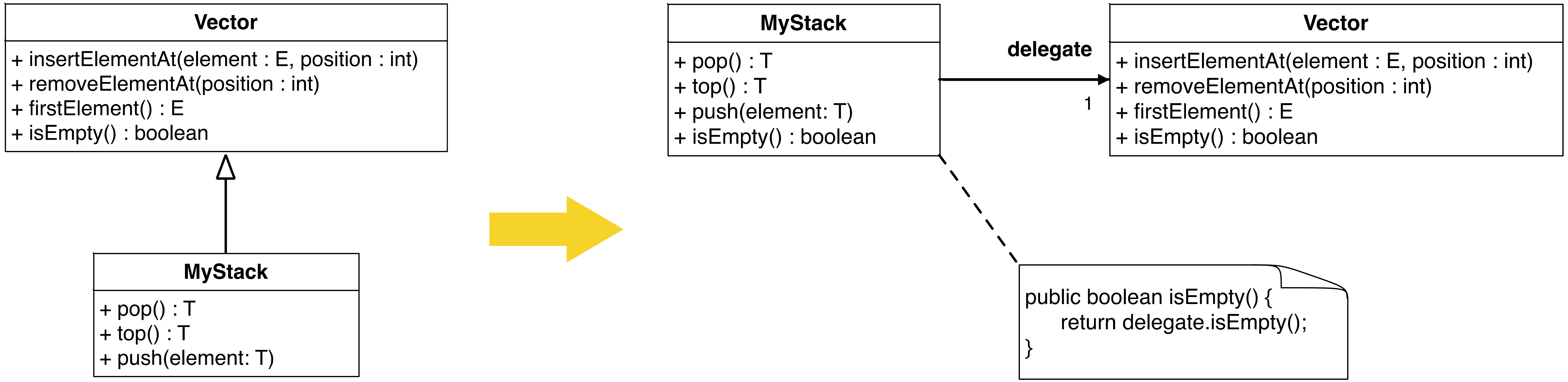
Easy

Due date in 7 days

5 min

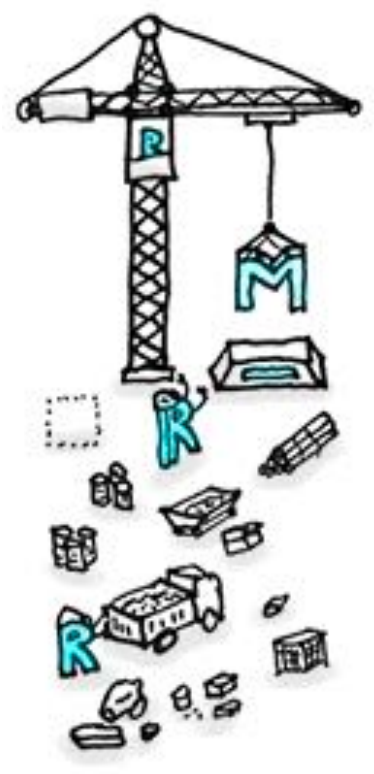
3 pts

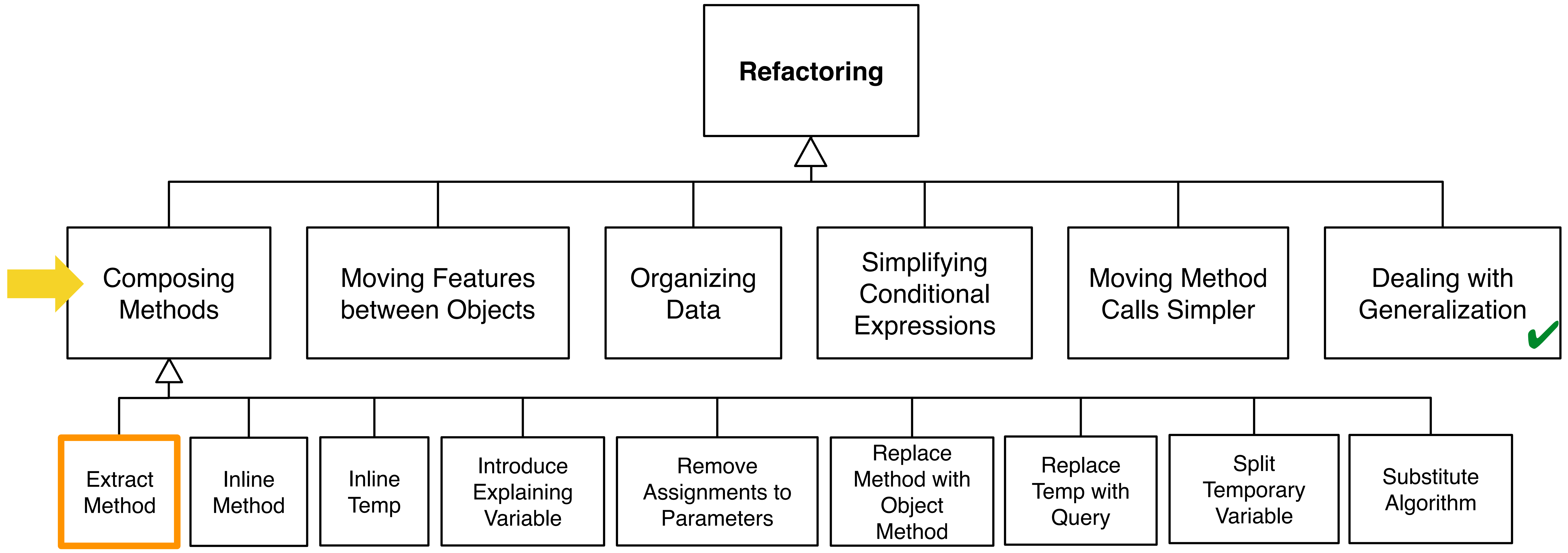
- Problem statement: refactor **MyStack** to use delegation instead of inheritance



```
public boolean isEmpty() {
    return delegate.isEmpty();
}
```

Composing methods





Refactoring
Composing Methods
Moving Features between Objects
Organizing Data
Simplifying Conditional Expressions
Moving Method Calls Simpler
Dealing with Generalization
✓
Extract Method
Inline Method
Inline Temp
Introduce Explaining Variable
Remove Assignments to Parameters
Replace Method with Object Method
Replace Temp with Query
Split Temporary Variable
Substitute Algorithm

Extract method

- One of the most common refactoring
- Symptom
  - The method is too long
  - The code needs a comment to understand its purpose

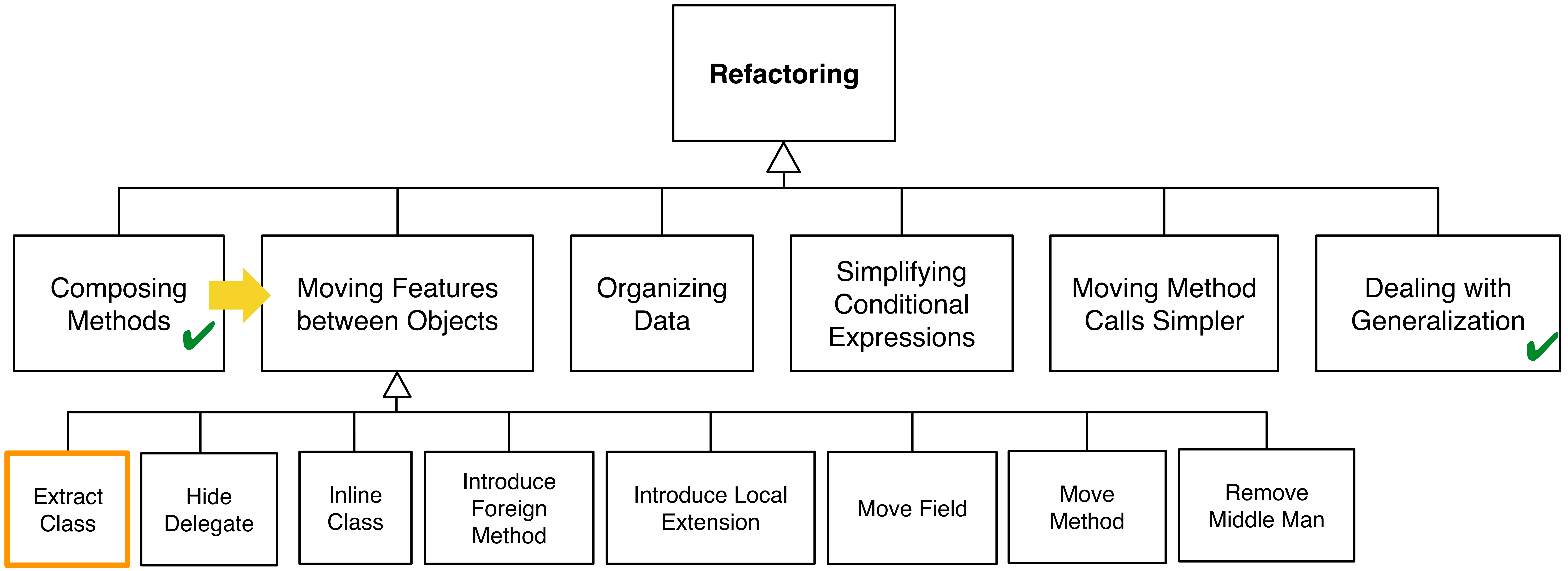
Original code:

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

Refactored code:

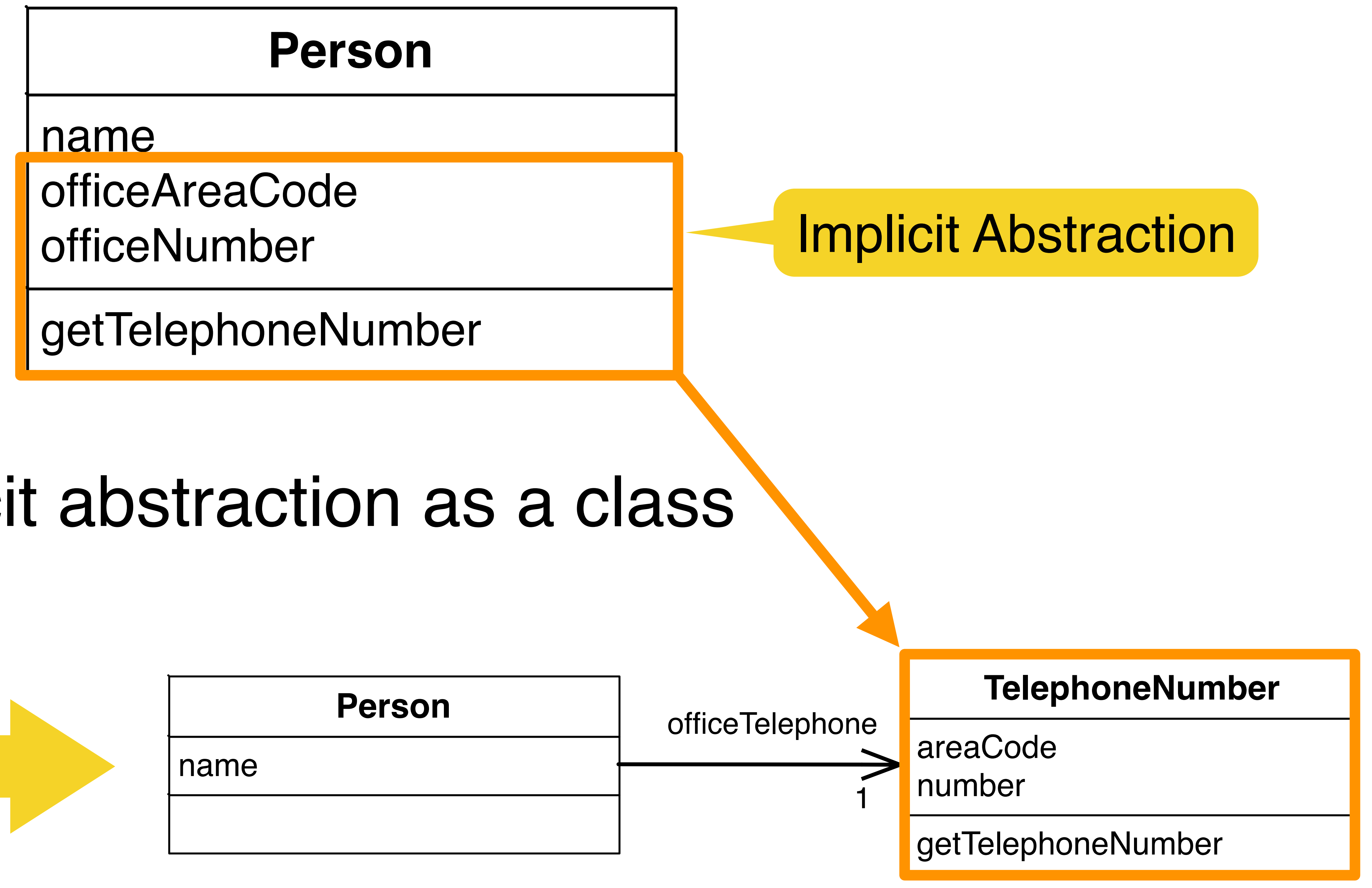
```java
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

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

Moving features between objects

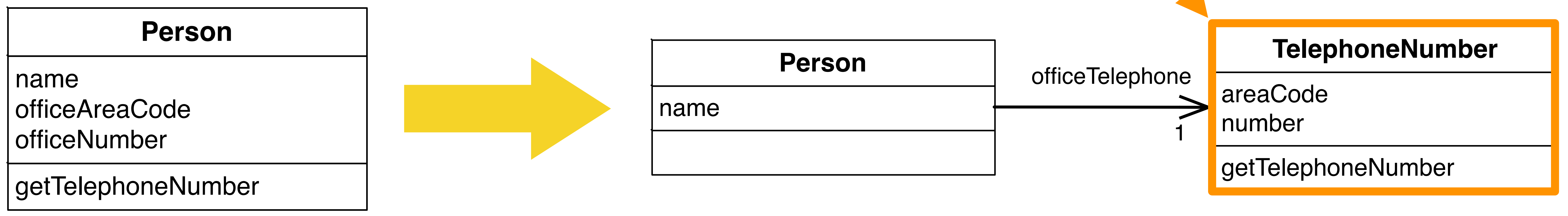

Refactoring
Composing Methods ✓
Moving Features between Objects
Organizing Data
Simplifying Conditional Expressions
Moving Method Calls Simpler
Dealing with Generalization ✓
Extract Class
Hide Delegate
Inline Class
Introduce Foreign Method
Introduce Local Extension
Move Field
Move Method
Remove Middle Man

Extract class

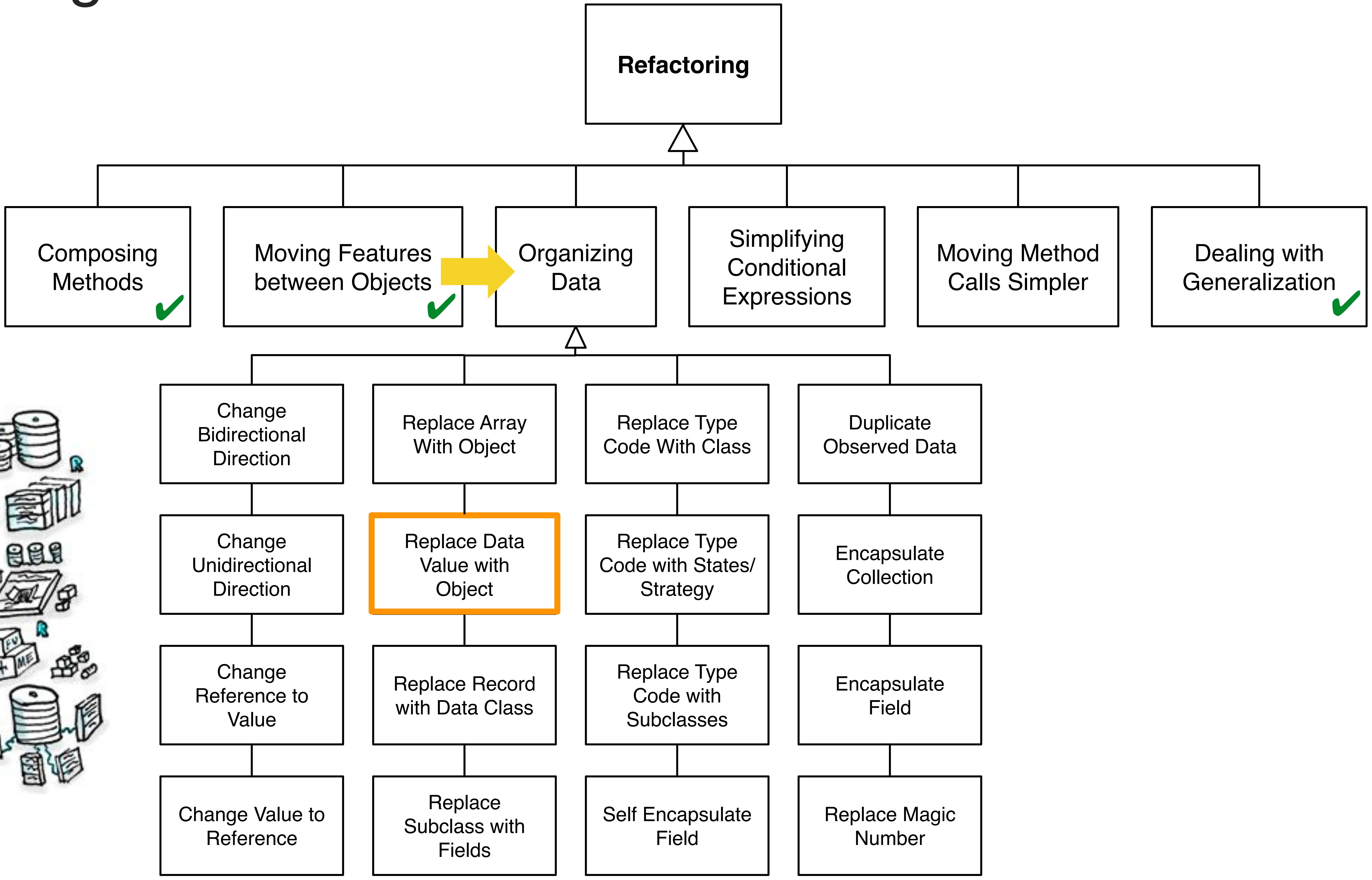
- Motivation
  - A class is an abstraction that handles a few clear responsibilities
  - In practice, classes grow by adding attributes and methods and the class may become too complicated
- Smell: a class contains an implicit abstraction that is not explicitly modeled
- Example



Model refactoring: model the implicit abstraction as a class



Organizing data



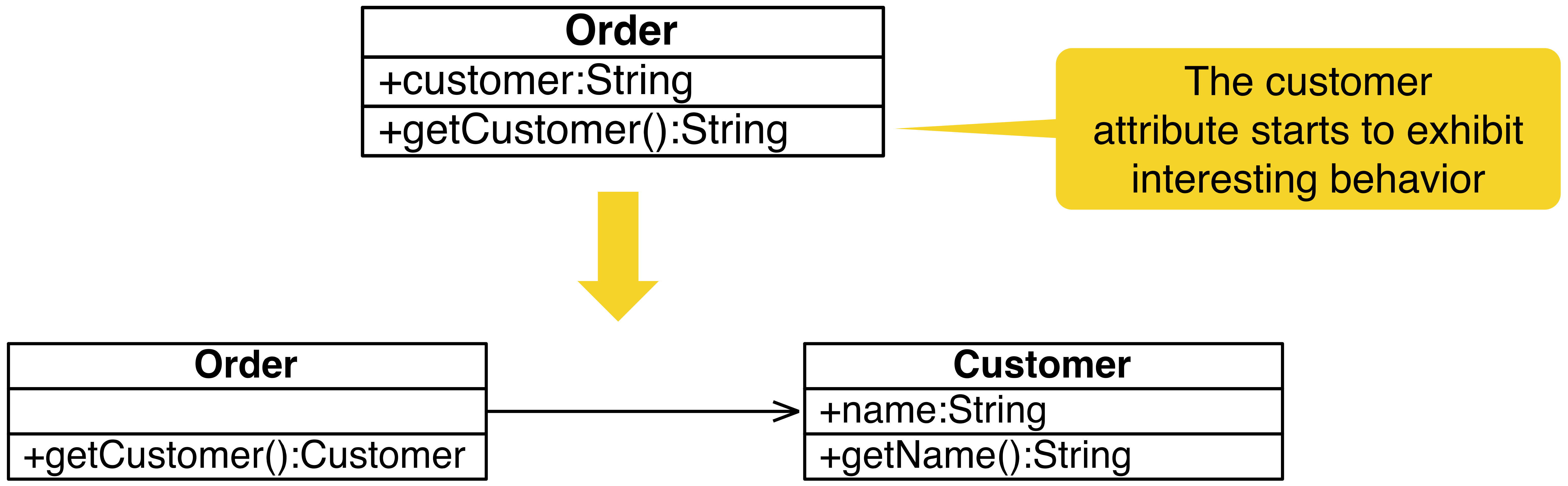
Refactoring
Composing Methods ✓
Moving Features between Objects ✓
Organizing Data
Simplifying Conditional Expressions
Moving Method Calls Simpler
Dealing with Generalization ✓
Change Bidirectional Direction
Change Unidirectional Direction
Change Reference to Value
Change Value to Reference
Replace Array With Object
Replace Data Value with Object
Replace Record with Data Class
Replace Subclass with Fields
Replace Type Code With Class
Replace Type Code with States/ Strategy
Replace Type Code with Subclasses
Self Encapsulate Field
Duplicate Observed Data
Encapsulate Collection
Encapsulate Field
Replace Magic Number

Replace data value with object

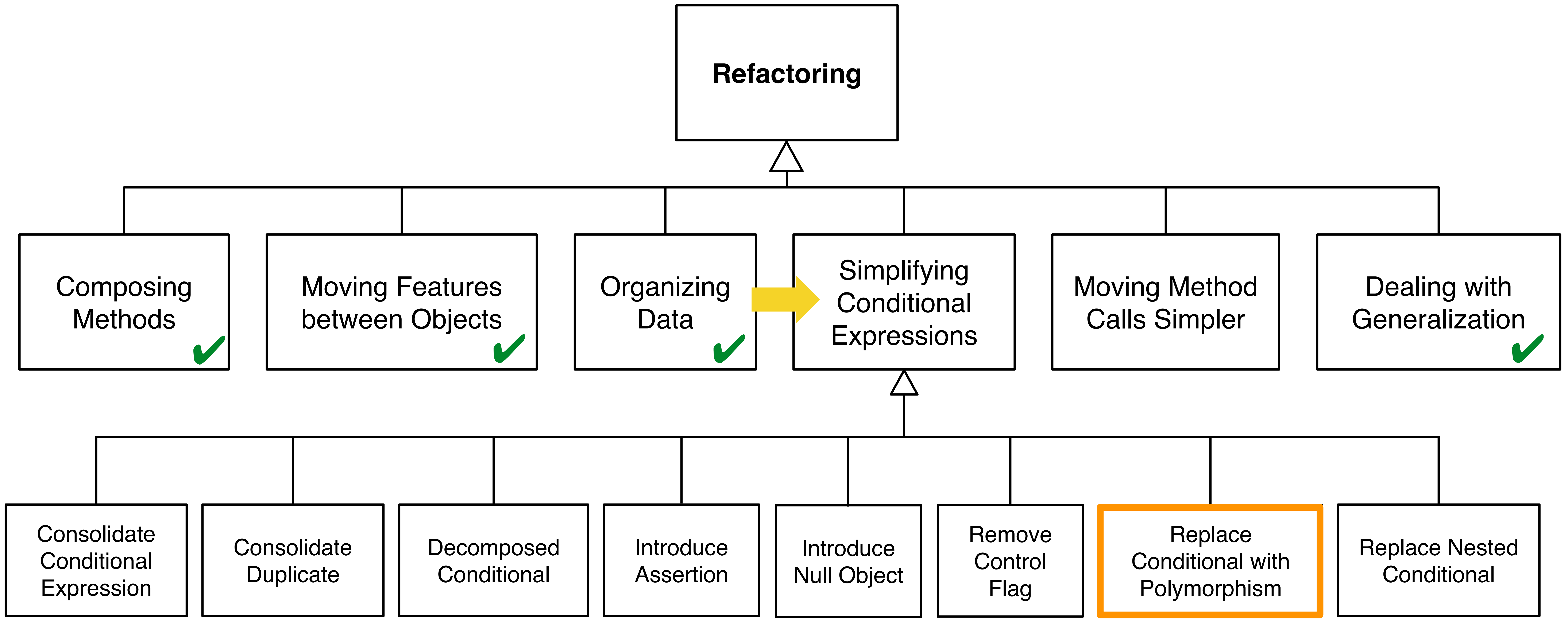
- Motivation
  - An attribute in a class exhibits interesting behavior
  - The developer starts to add more and more methods to the class to deal with this behavior
- Source code refactoring steps
  1. Create a new class with an attribute of the same type as the attribute in the source class
  2. Add a getter and a constructor that takes the attribute as an argument
  3. Change the type of the attribute in the source class to the new class
  4. Change the getter in the source class to call the getter in the new class
  5. If the attribute is mentioned in the source class constructor, assign the field using the constructor of the new class
  6. Change the getter method to create a new instance of the new class
  7. You may need to use the refactoring Change Value to Reference on the new class

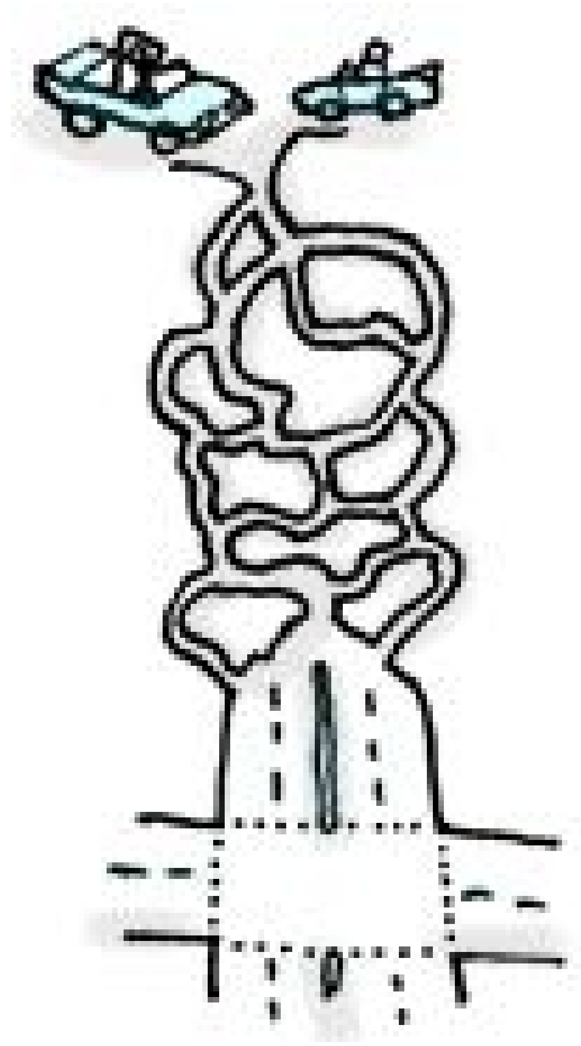
Replace data value with object

- Motivation
  - An attribute in a class exhibits interesting behavior
  - The developer starts to add more and more methods to the class to deal with this behavior
- Example



Simplifying conditional expressions


Refactoring
Composing Methods ✓
Moving Features between Objects ✓
Organizing Data ✓
Simplifying Conditional Expressions
Moving Method Calls Simpler
Dealing with Generalization ✓
Consolidate Conditional Expression
Consolidate Duplicate
Decomposed Conditional
Introduce Assertion
Introduce Null Object
Remove Control Flag
Replace Conditional with Polymorphism
Replace Nested Conditional



Replace conditional with polymorphism

Motivation: assume you are modeling the different speeds of a bird
Your source code solution might look like this

```
public class Bird {
    public static final int SMALL = 1;
    public static final int MIDDLE = 2;
    public static final int LARGE = 3;
    double baseSpeed = 10;
    double factor = 2.0;
    public double getSpeed(int type) {
        switch(type) {
        case SMALL:
            return baseSpeed;
        case MIDDLE:
            return baseSpeed – factor;
        case LARGE:
            return baseSpeed – factor * 2;
        }
        return -1;
    }
}
```

Now assume that you need to add a new bird that flies at supersonic speed ☺ (change of requirements)

—> You propose: "Add another case statement"

Smell ☹

Replace conditional with polymorphism: steps

1. Turn the class (`Bird`) into an abstract class
2. Create a subclass for each case label
   - e.g. small —> `SmallBird`, medium —> `MediumBird`, large —> `LargeBird`
3. Add an abstract method (`getSpeed`) to the super class for the method that contains the case statement
4. Create subclass methods that override the super class method (`getSpeed`)
5. Repeat for each branch of the case statement: move its body into the appropriate subclass method

```java
public class SmallBird extends Bird {

    public double getSpeed() {
        return baseSpeed;
    }
}
```

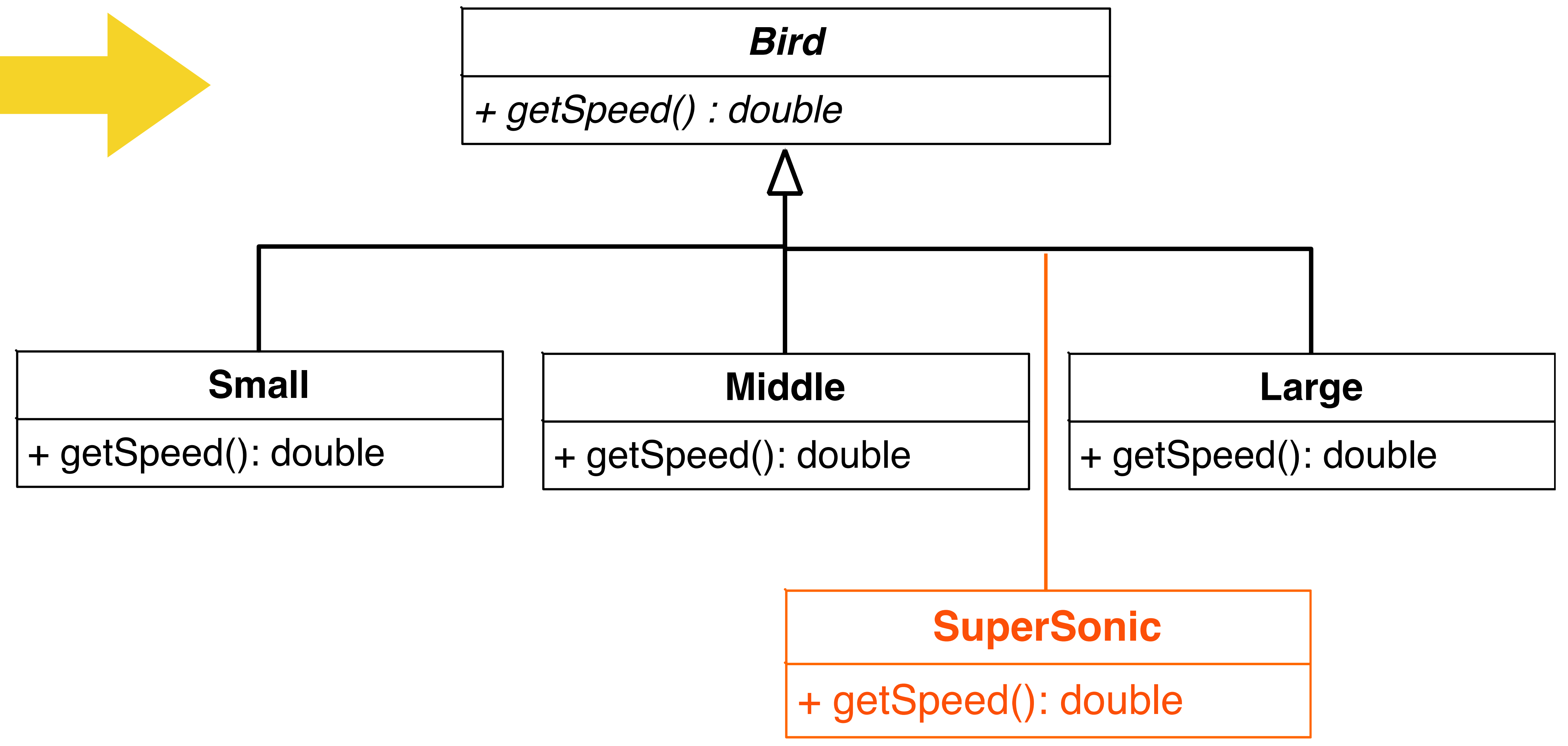
```java
public class MiddleBird extends Bird {

    public double getSpeed() {
        return baseSpeed - factor;
    }
}
```

```java
public class LargeBird extends Bird {

    public double getSpeed() {
        return baseSpeed - factor * 2;
    }
}
```

Replace conditional with polymorphism

Source code refactoring + model refactoring: model the different kinds of birds in a taxonomy

```java
public class Bird {
   public static final int SMALL = 1;
   public static final int MIDDLE = 2;
   public static final int LARGE = 3;
   double baseSpeed = 10;
   double factor = 2.0;

   public double getSpeed(int type) {
      switch(type) {
      case SMALL:
         return baseSpeed;
      case MIDDLE:
         return baseSpeed-factor;
      case LARGE:
         return baseSpeed-factor*2;
      }
      return -1;
   }
}
```



- Advantages of the refactored solution
  - Easier to add new subclass
  - No need to understand the other subclasses
  - Easier to understand the code

L07E05 Conditional → Polymorphism

Not started yet.

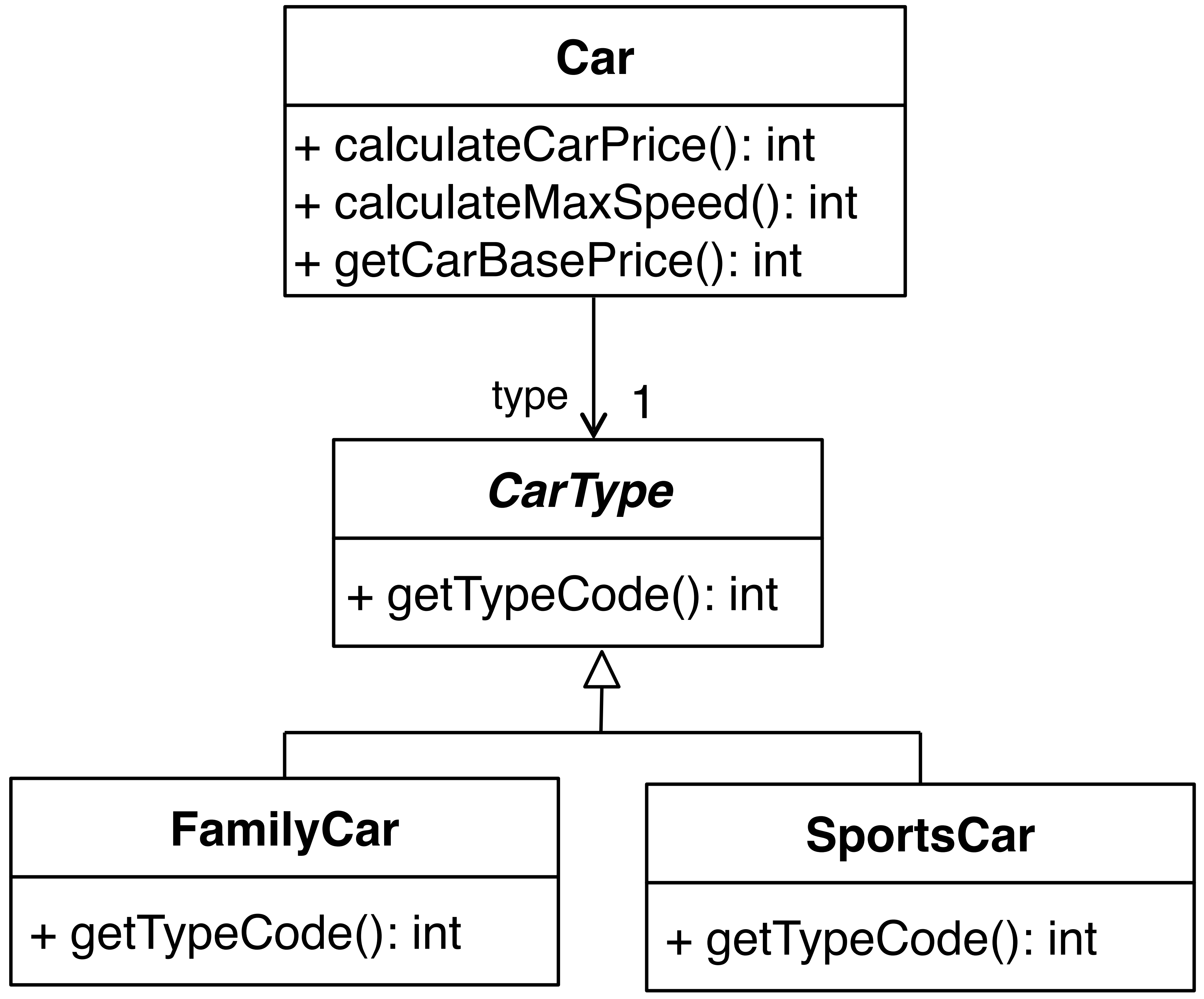
Start exercise

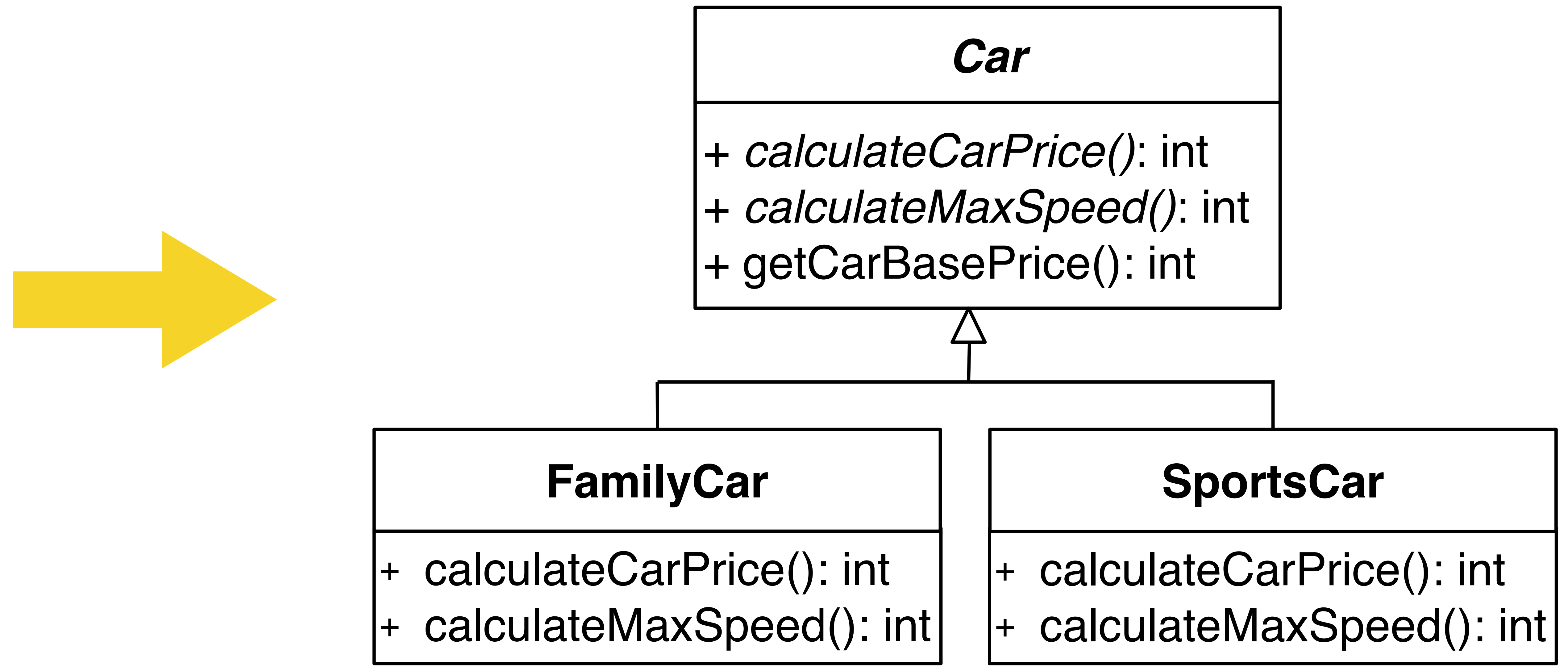
Easy Bonus

Due date in 7 days

3 pts

- Problem statement




Car
+ calculateCarPrice(): int
+ calculateMaxSpeed(): int
+ getCarBasePrice(): int
FamilyCar
+ calculateCarPrice(): int
+ calculateMaxSpeed(): int
SportsCar
+ calculateCarPrice(): int
+ calculateMaxSpeed(): int

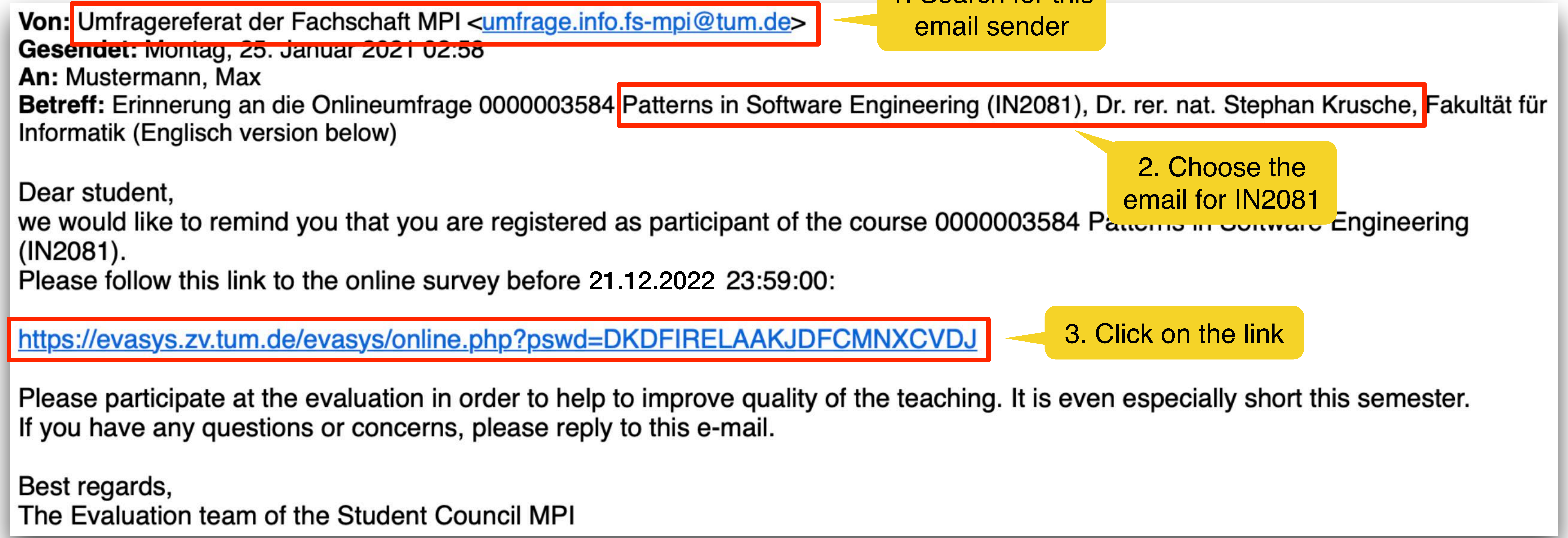
University course evaluation

- We put a lot of effort and passion into creating a great learning atmosphere in PSE and to provide you with the latest concepts, tools, and workflows
- We hope you appreciate our effort with a positive evaluation 😊
  - … and with comments on issues, that we can improve in the future semesters

Your feedback is valuable to us and to the university!

University course evaluation (20 min)

- Find the email with a link to https://evasys.zv.tum.de/... for IN2081



Von: Umfragereferat der Fachschaft MPI <umfrage.info.fs-mpi@tum.de>
Gesendet: Montag, 25. Januar 2021 02:58
An: Mustermann, Max
Betreff: Erinnerung an die Onlineumfrage 0000003584 Patterns in Software Engineering (IN2081), Dr. rer. nat. Stephan Krusche, Fakultät für Informatik (Englisch version below)

Dear student,
we would like to remind you that you are registered as participant of the course 0000003584 Patterns in Software Engineering (IN2081).
Please follow this link to the online survey before 21.12.2022 23:59:00:

https://evasys.zv.tum.de/evasys/online.php?pswd=DKDFIRELAAKJDFCMNXCVDJ

Please participate at the evaluation in order to help to improve quality of the teaching. It is even especially short this semester.
If you have any questions or concerns, please reply to this e-mail.

Best regards,
The Evaluation team of the Student Council MPI

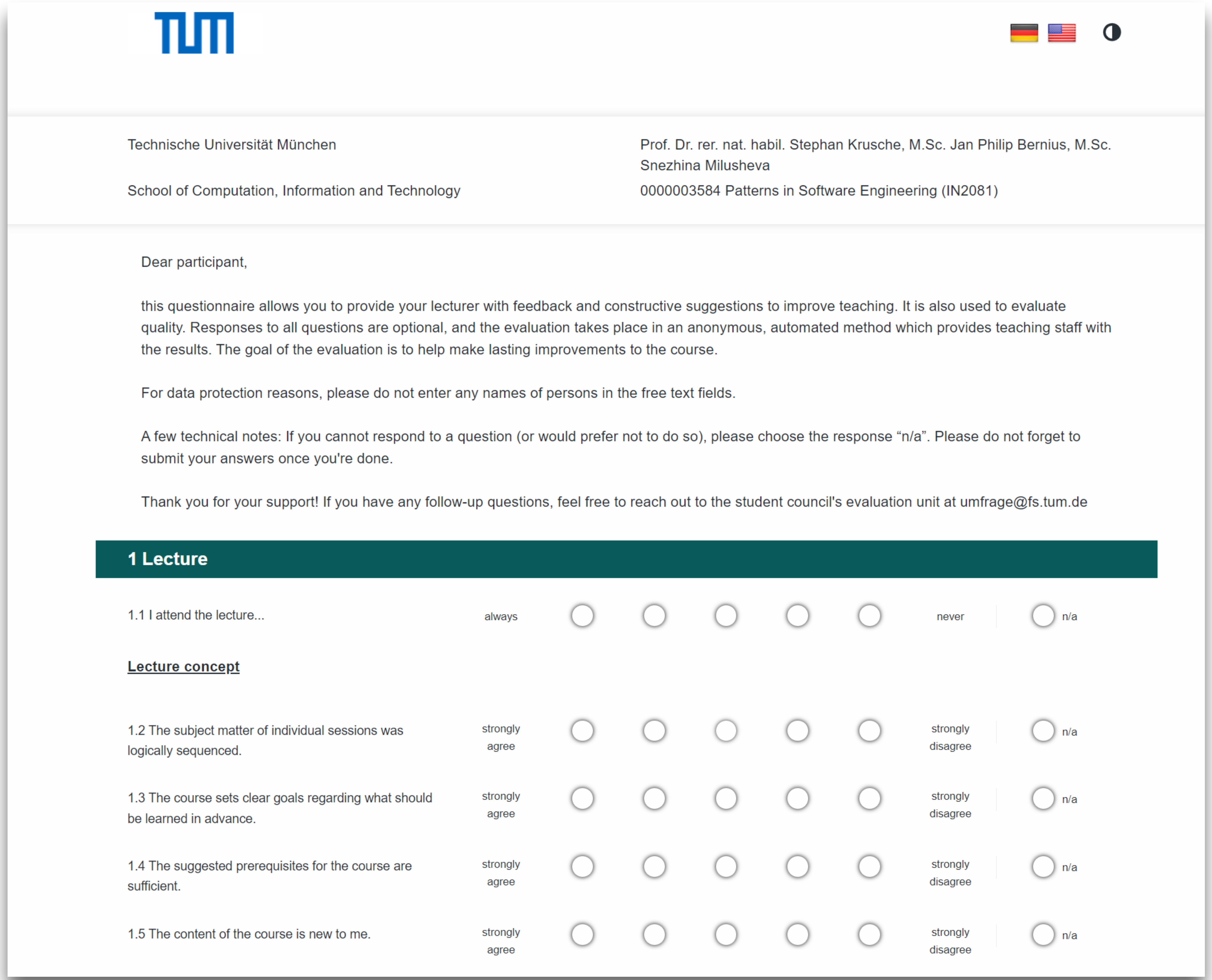
1. Search for this email sender

2. Choose the email for IN2081

3. Click on the link

University course evaluation (20 min)

- Fill out the following form



Technische Universität München

School of Computation, Information and Technology

Prof. Dr. rer. nat. habil. Stephan Krusche, M.Sc. Jan Philip Bernius, M.Sc. Snezhina Milusheva

0000003584 Patterns in Software Engineering (IN2081)

Dear participant,

this questionnaire allows you to provide your lecturer with feedback and constructive suggestions to improve teaching. It is also used to evaluate quality. Responses to all questions are optional, and the evaluation takes place in an anonymous, automated method which provides teaching staff with the results. The goal of the evaluation is to help make lasting improvements to the course.

For data protection reasons, please do not enter any names of persons in the free text fields.

A few technical notes: If you cannot respond to a question (or would prefer not to do so), please choose the response “n/a”. Please do not forget to submit your answers once you're done.

Thank you for your support! If you have any follow-up questions, feel free to reach out to the student council's evaluation unit at umfrage@fs.tum.de

1 Lecture

1.1 I attend the lecture...	always	○	○	○	○	○	never	○ n/a

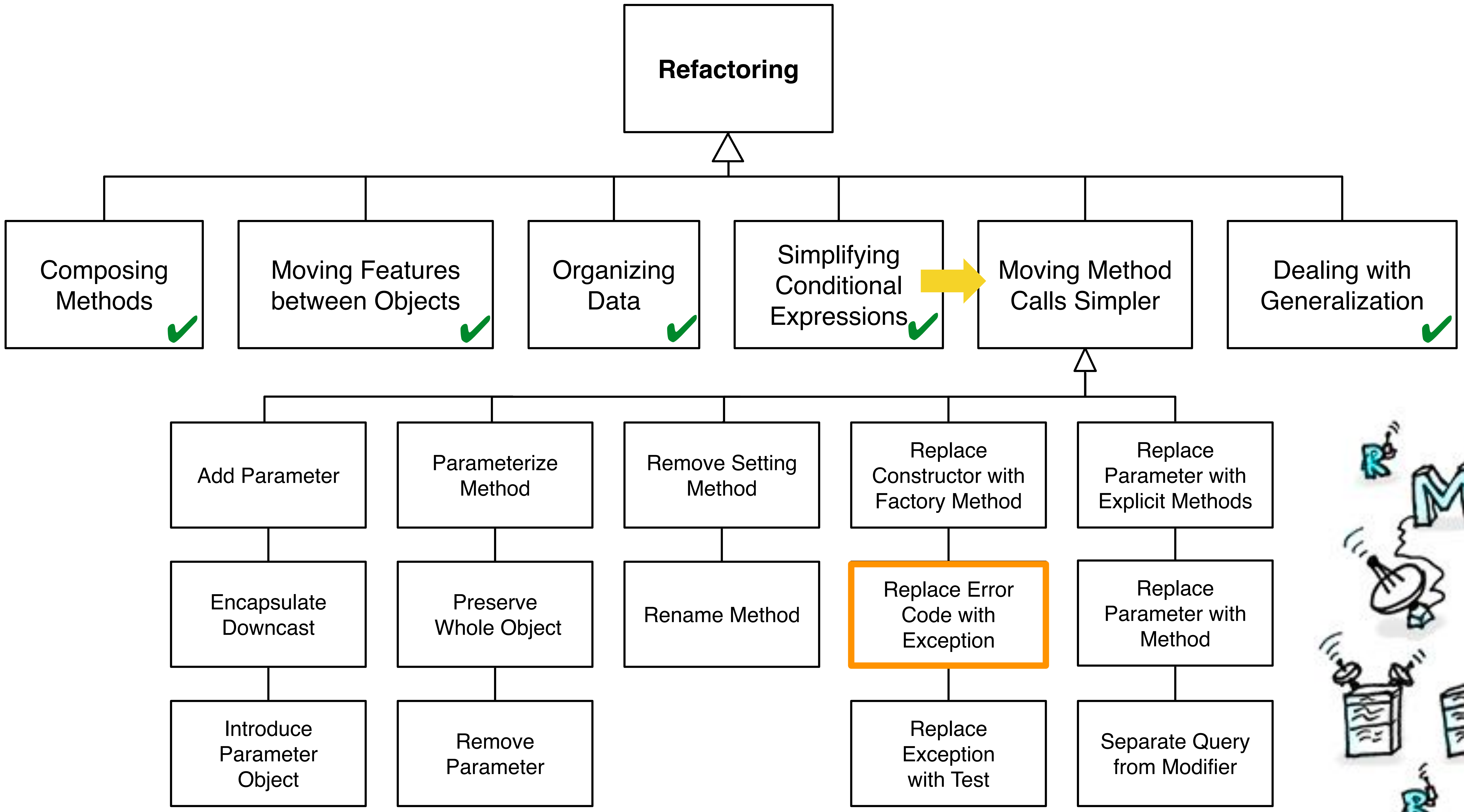
Lecture concept

1.2 The subject matter of individual sessions was logically sequenced.	strongly agree	○	○	○	○	○	strongly disagree	○ n/a
1.3 The course sets clear goals regarding what should be learned in advance.	strongly agree	○	○	○	○	○	strongly disagree	○ n/a
1.4 The suggested prerequisites for the course are sufficient.	strongly agree	○	○	○	○	○	strongly disagree	○ n/a
1.5 The content of the course is new to me.	strongly agree	○	○	○	○	○	strongly disagree	○ n/a

Outline

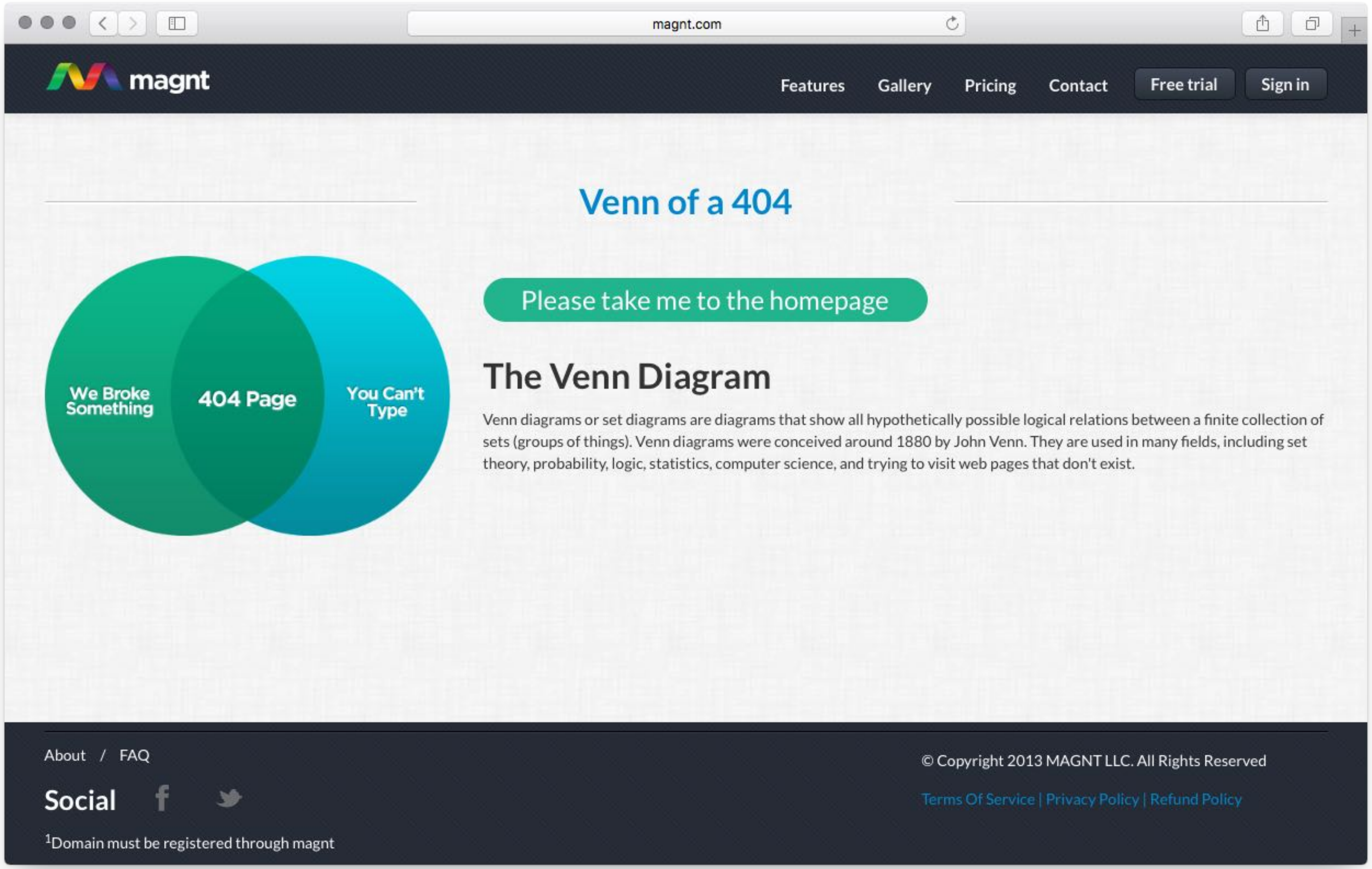
- Antipatterns
  - Spaghetti code
  - Cut and paste programming
  - Vendor lock-in
  - Analysis paralysis
- Code smells
  - Replace inheritance with delegation
  - Replace conditional with polymorphism
  - ➡ **Replace error code with exception**

Making method calls simpler



Refactoring
Composing Methods ✓
Moving Features between Objects ✓
Organizing Data ✓
Simplifying Conditional Expressions ✓
Moving Method Calls Simpler
Dealing with Generalization ✓
Add Parameter
Encapsulate Downcast
Introduce Parameter Object
Parameterize Method
Preserve Whole Object
Remove Parameter
Remove Setting Method
Rename Method
Replace Constructor with Factory Method
Replace Error Code with Exception
Replace Exception with Test
Replace Parameter with Explicit Methods
Replace Parameter with Method
Separate Query from Modifier

An error code you might know



Venn of a 404

The Venn Diagram

Venn diagrams or set diagrams are diagrams that show all hypothetically possible logical relations between a finite collection of sets (groups of things). Venn diagrams were conceived around 1880 by John Venn. They are used in many fields, including set theory, probability, logic, statistics, computer science, and trying to visit web pages that don't exist.

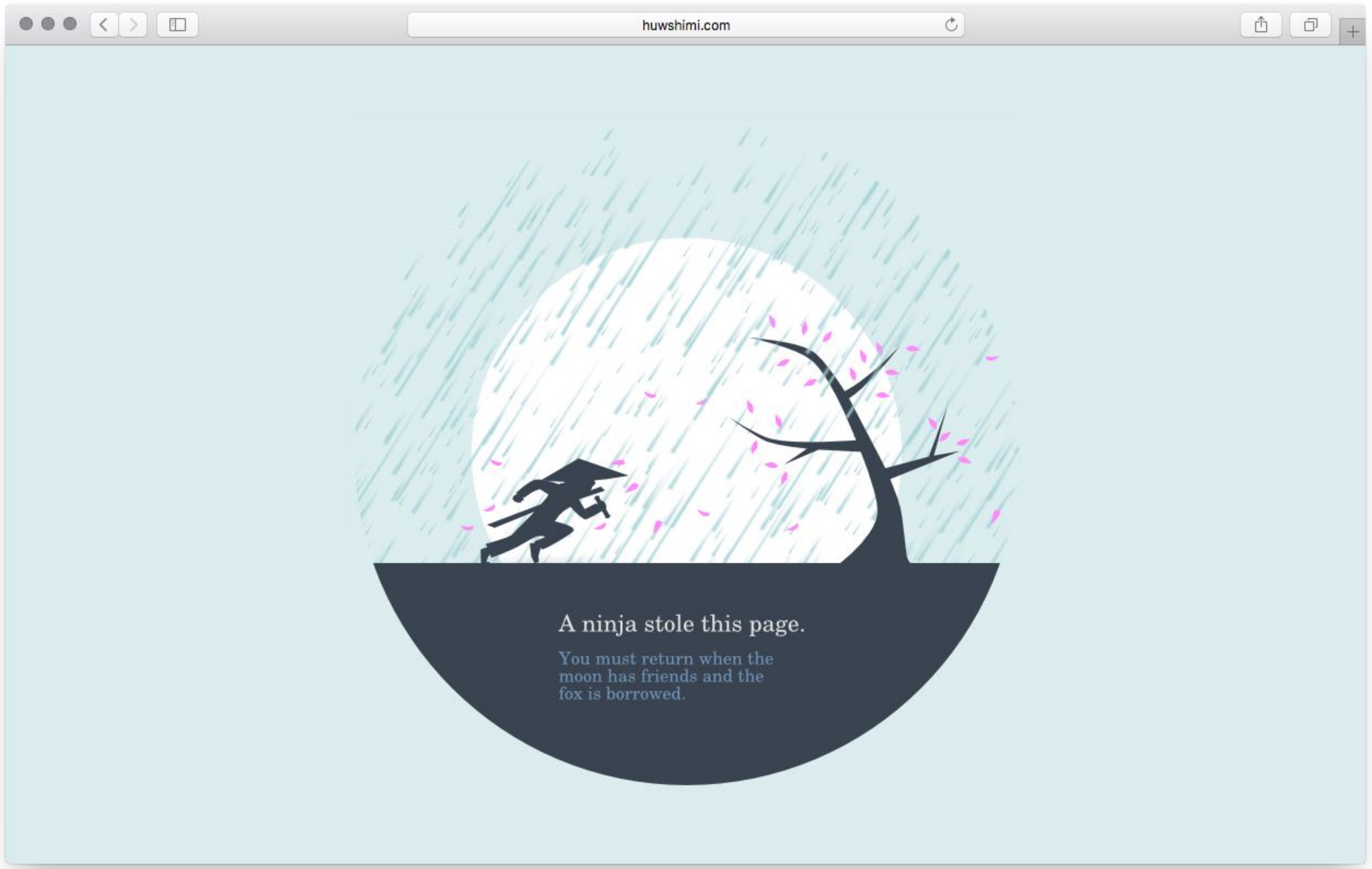
[1]Domain must be registered through magnt

© Copyright 2013 MAGNT LLC. All Rights Reserved

Terms Of Service | Privacy Policy | Refund Policy

Motivation

- Exceptions are better ways to deal with errors than error codes
- Make the problem explicit instead of using a magic number
- Using exceptions makes the error condition and exception handling easier to understand



A ninja stole this page.

You must return when the moon has friends and the fox is borrowed.

Replace error code with exception

```
class Account {
   private int balance;

   public int withdraw(int amount) {
      if (amount > balance) {
         return -1;
      }
      else {
         balance -= amount;
         return 0;
      }
   }
}
```

Example: using the Java exception mechanism

```
class Account {
   private int balance;

   public void withdraw(int amount) throws BalanceException {
      if (amount > balance) throw new BalanceException();
      balance -= amount;
   }
}
```

To replace the error code with an exception, first decide if it is the responsibility of the caller to test the balance before withdrawing or whether it is the responsibility of the withdraw routine to make the test

Terminology: checked vs. unchecked exceptions

- Unchecked exception
  - Unchecked exceptions do not need to be declared
  - The class RuntimeException in Java and its subclasses are unchecked exceptions
  - **It is the responsibility of the caller to do any testing**
- Checked exception
  - Checked exceptions need to be declared
  - The class Exception in Java and any subclasses that are not subclasses of RuntimeException are checked exceptions
  - **It is the responsibility of the callee to do any testing**
- Good design
  - If you can choose, use unchecked exceptions because your code is more robust (often you cannot expect that the called method does any checking)
  - Robustness is a good design goal

Steps to replace error codes with exceptions

1. Decide if you want to place the responsibility of the exception handling with the caller (ensure robustness!) or the callee?
2. Find all the callers of the method and adjust the method calls to deal with the exception
    - For **unchecked** exception:
        - Make the appropriate check before calling the method
        - In the method, throw an exception instead of returning an error code
    - For **checked** exception:
        - Call the method in a try block and handle the exception in a catch block
        - Define an exception class and throw the exception instead of returning the error code in the method
3. Change the signature of the method to reflect the new usage, in particular remove the return code, as the error is now handled via the exception mechanism

Example: unchecked exception

Caller:

```
if (account.withdraw(amount) == -1) {
    handleOverdrawn();
}
else {
    doTheUsualThing();
}
```

Callee:

```
public int withdraw(int amount) {
    if (amount > balance) {
        return -1;
    }
    else {
        balance -= amount;
        return 0;
    }
}
```

Before: using error codes

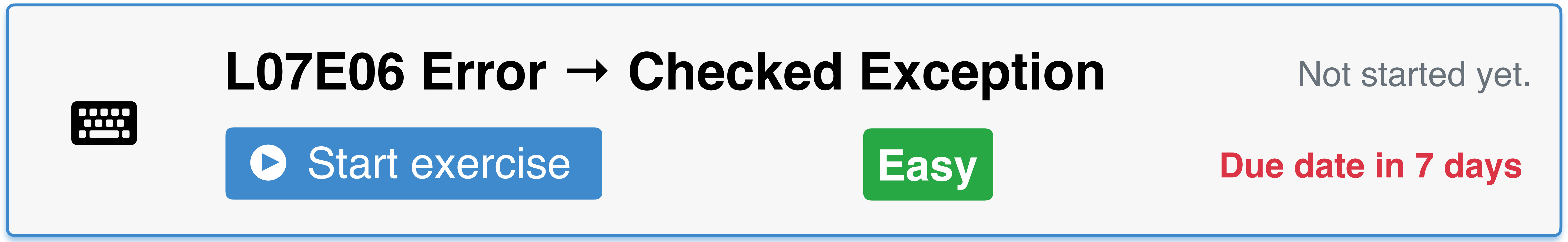
Caller: Test condition before call

```
if (!account.canWithdraw(amount)) {
    handleOverdrawn();
}
else {
    account.withdraw(amount);
    doTheUsualThing();
}
```

Callee: Throws exception instead of returning error code

```
public void withdraw(int amount) {
    if (amount > balance) {
        throw new IllegalArgumentException
        ("Amount too large”);
    }
    balance -= amount;
}
```

After: using exceptions


L07E06 Error → Checked Exception

Not started yet.

Start exercise

Easy

Due date in 7 days

5 min

3 pts

- Problem statement: refactor the withdraw() method using a **checked exception**
  - Introduce a new checked exception called **`BalanceException`**
  - Add a try-catch block to the caller to handle this exception

Summary

- **Spaghetti code**: ad hoc structure which makes it hard to extend or optimize a model or code
- **Cut and paste programming**: excessive code duplication (clones) make the software expensive and difficult to maintain
- **Vendor lock-in**: dependence on a proprietary architecture or tool set, making it hard to switch to another vendor
- **Analysis paralysis**: spending excessive time in requirements elicitation and analysis
- **Code smells** are heuristics that indicates when to refactor, what specific technique to use and how the **refactored solution** can improve the problem

Literature

- Martin Fowler: Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
- Wikipedia: http://en.wikipedia.org/wiki/Code_smell
- Sourcemaking: http://sourcemaking.com/refactoring
- Amr Elssamadisy: Agile Adoption Patterns, Addison Wesley, 2008