Patterns in Software Engineering

06 Antipatterns I

Stephan Krusche

05 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

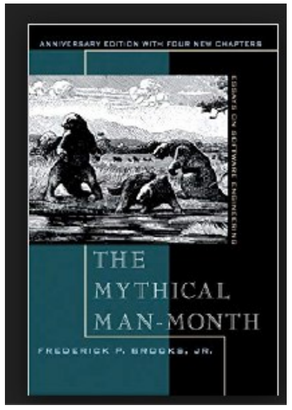
- **Context and assumptions**
  - You have understood the basic concepts of patterns
  - You have implemented many different design and architectural patterns
- **Learning goals: at the end of this lecture you are able to**
  - Identify an antipattern in existing code
  - Refactor an antipattern with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

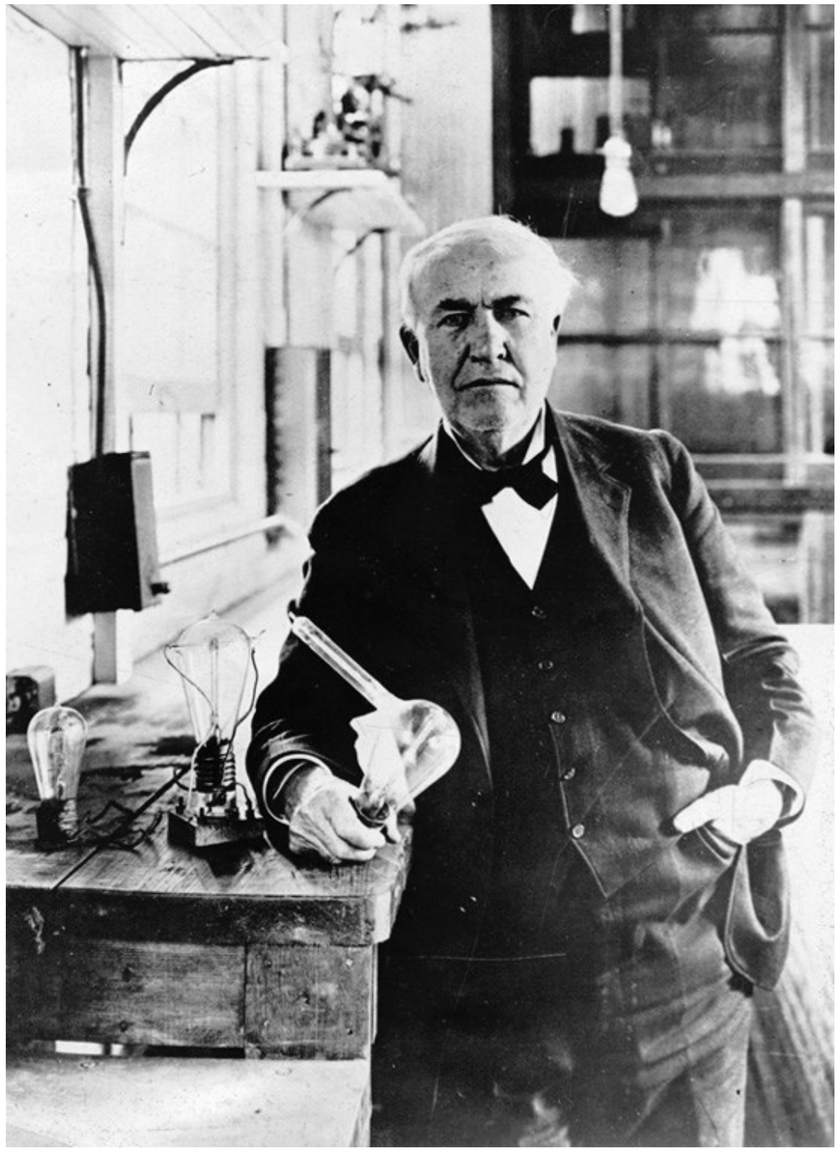
Outline

- **Antipattern definition**
- Golden hammer
- Functional decomposition
- Antipattern taxonomy
- Lava flow
- The blob

“Adding manpower to a late software project makes it later.”

— Frederick Brooks, *The Mythical Man-Month*



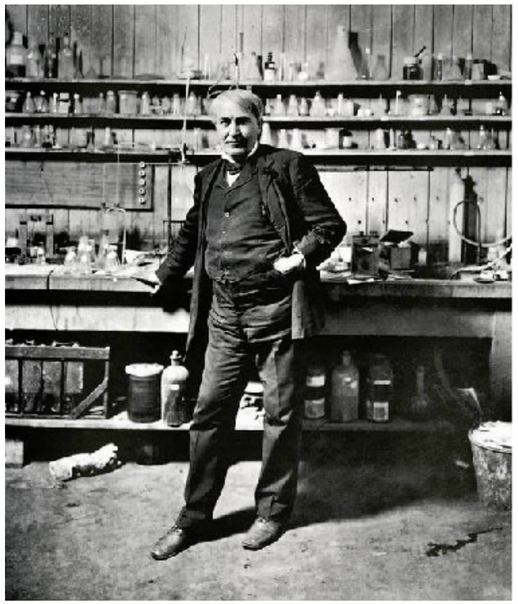


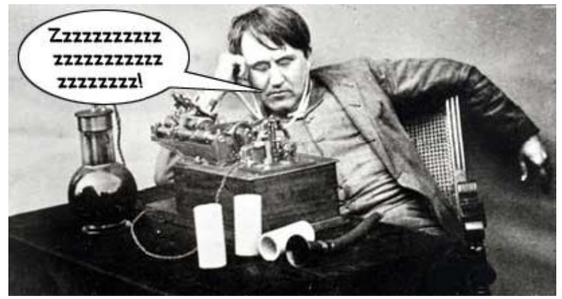
“I have not failed, I've just found 10,000 ways that won't work.”

— Thomas Edison

Thomas Edison (1847 – 1931)

- Inventor and businessman
  - Technologist & manager
  - CTO & CEO in one person
- 1093 patents: phonograph, first long-lasting light bulb, electric car battery, …
- He used teamwork for invention
  - Now called “Innovation management“
- Credited with the creation of the first industrial research laboratory
  - Later known as Bell Labs (13 Nobel prize winners)
- Known for crazy ideas & experiments
  - Inventor of the power nap
  - http://www.brainpickings.org/2013/02/11/thomas-edison-on-sleep-and-success





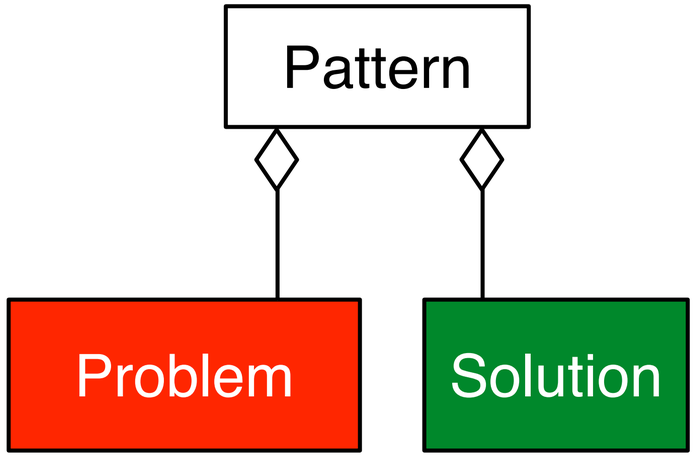
“If one does not know how to solve a problem, it may nevertheless be useful to know about likely blind alleys.

This is particularly true when something appears at first to be a solution but further analysis proves it is not…"

— Andrew Koenig
(coined the term *Antipattern*)

Pattern definition

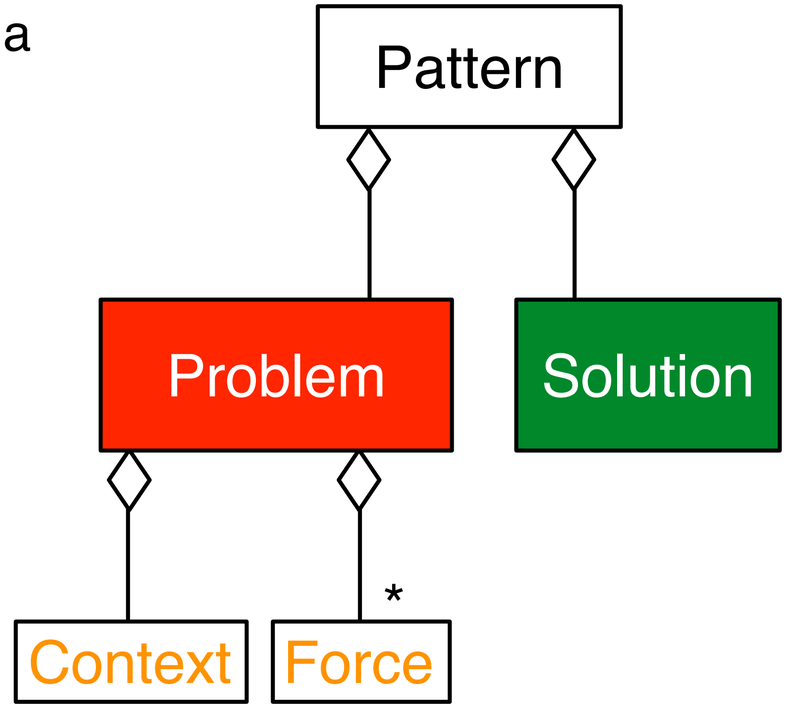
A pattern has two parts: **problem** and **solution**



Pattern definition

A pattern has two parts: **problem** and **solution**

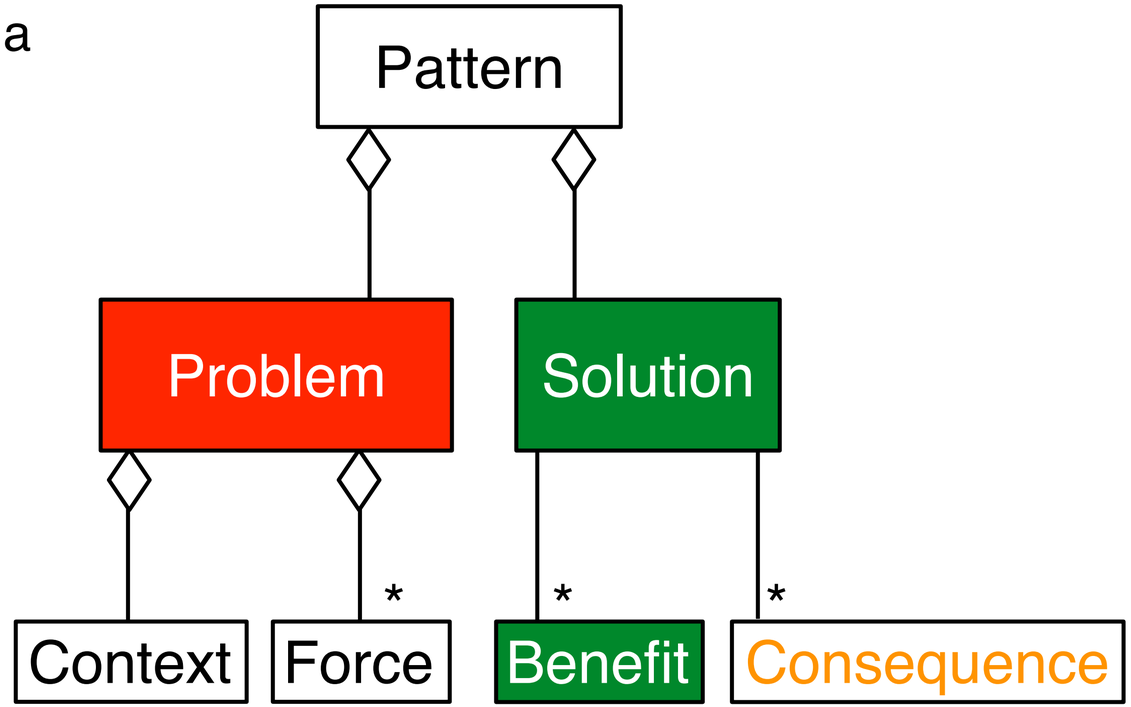
- The problem is elaborated in terms of a context and a set of forces



Pattern definition

A pattern has two parts: **problem** and **solution**

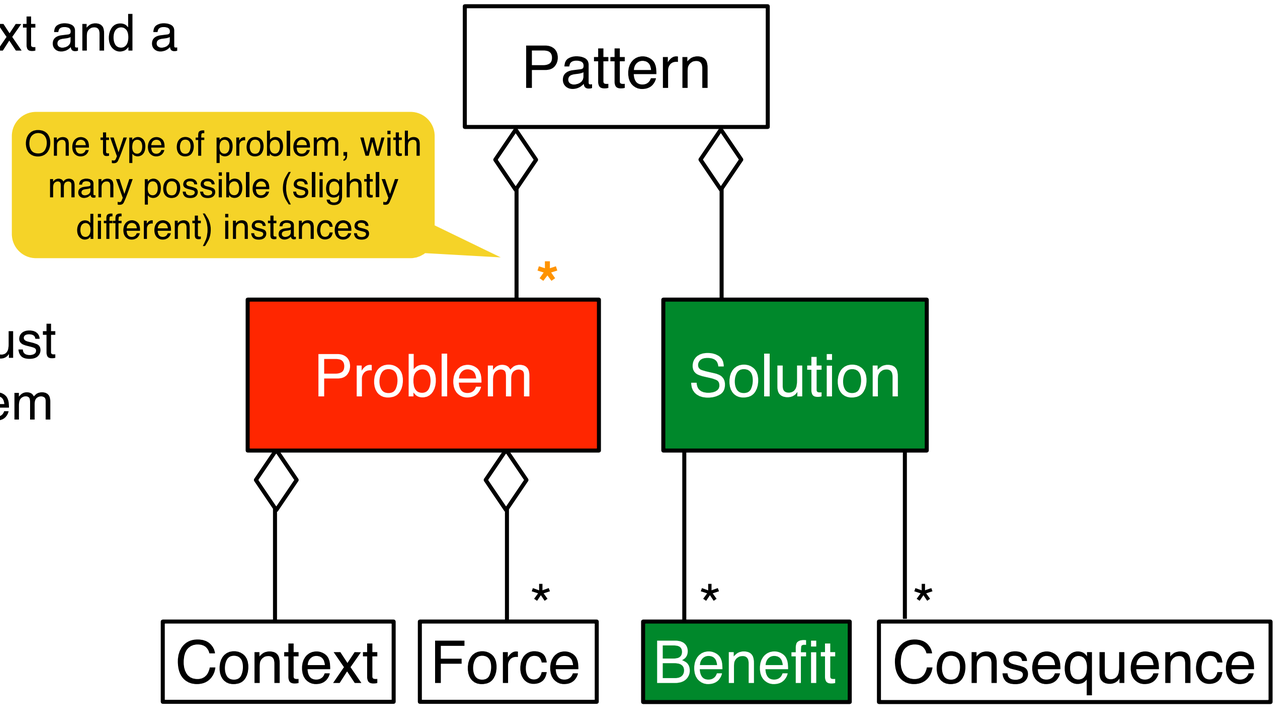
- The problem is elaborated in terms of a context and a set of forces
- The solution resolves these forces with benefits and consequences



Pattern definition

A pattern has two parts: **problem** and **solution**

- The problem is elaborated in terms of a context and a set of forces
- The solution resolves these forces with benefits and consequences
- To be considered as a pattern, the solution must be applicable to more than one specific problem



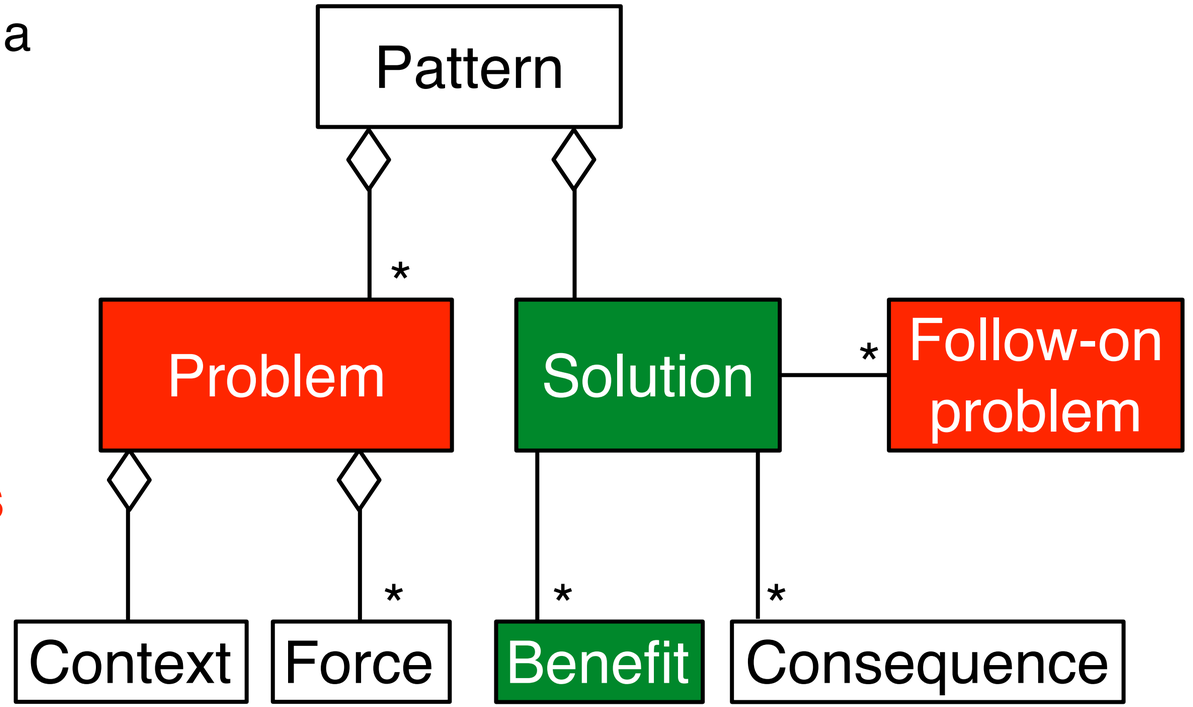
Pattern definition

A pattern has two parts: **problem** and **solution**

- The problem is elaborated in terms of a context and a set of forces
- The solution resolves these forces with benefits and consequences
- To be considered as a pattern, the solution must be applicable to more than one specific problem

Solutions usually generate **follow-on problems**

➡ Follow-on problems can again be elaborated in terms of context and forces which may lead to the applicability of other patterns



Patterns can become antipatterns

➡Patterns can evolve into antipatterns
when change occurs

Examples of changes:

- Change of requirements
  - Moving from a desktop system to a smart phone
  - Changing from centralized to decentralized control
- Change of project parameters
  - Increase of budget
  - Extending the project finish time
  - People leaving the project
- Change of methodology
  - Moving from functional decomposition to object-orientation
  - Moving from a defined process to an agile process

From [Gamma et al]:

“It‘s easiest to see a pattern as a solution, as a technique that can be adapted and reused.

It‘s harder to see when it is appropriate – to characterize the problems it solves and the context in which it is the best solution.”

➡**In such cases the solution proposed by the pattern can become a problematic solution**

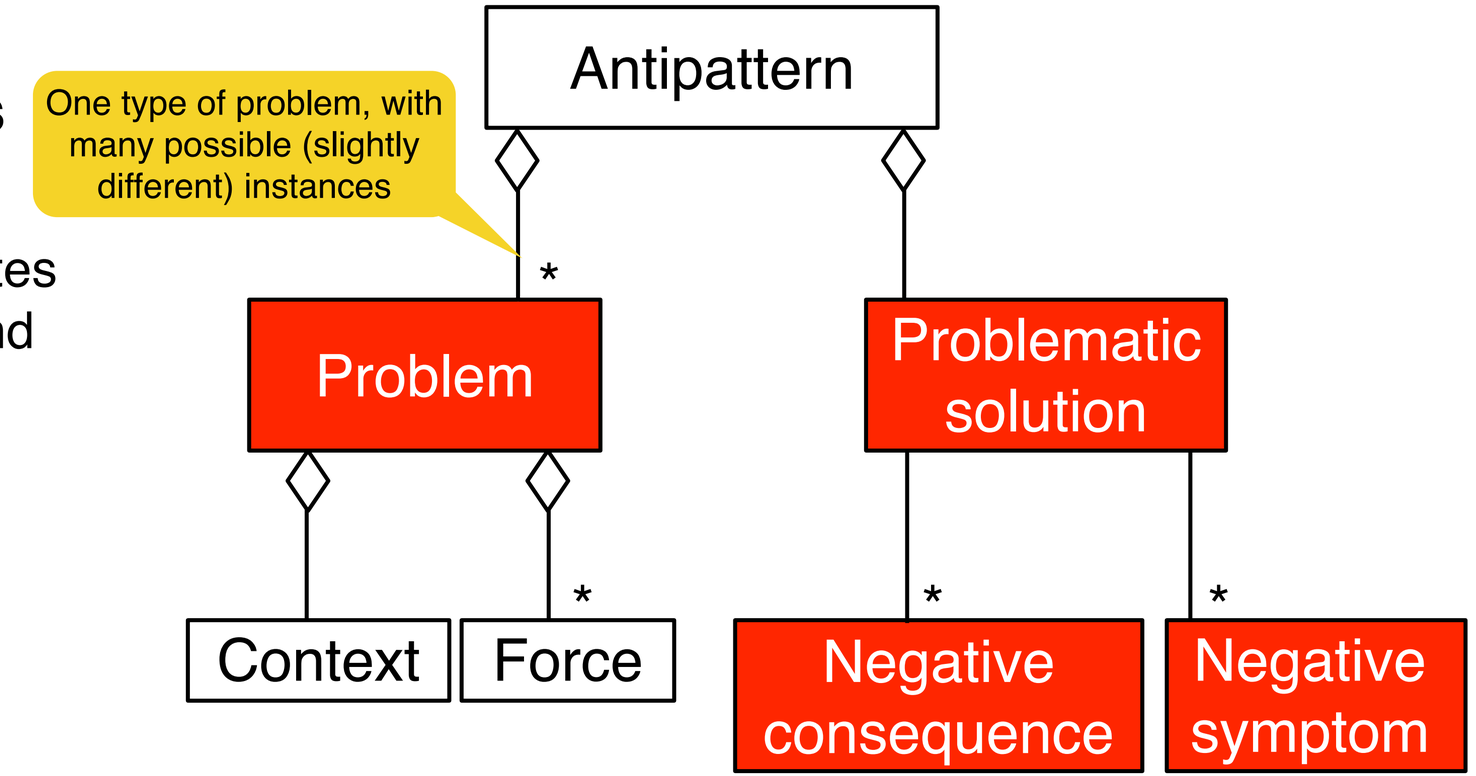
Informal antipattern definition

- **Antipatterns** identify and categorize common mistakes in software practice
- “An **antipattern** is something that looks like a good idea, but which backfires badly when applied“ (Jim Coplien)
- Synonyms
  - Bad practice
  - Bad idea or pitfall
  - Typical project management mistake
  - Typical design error
  - Bad use of an activity

Antipattern definition

An antipattern consists of
1 type of problem(s) and **2 solutions**

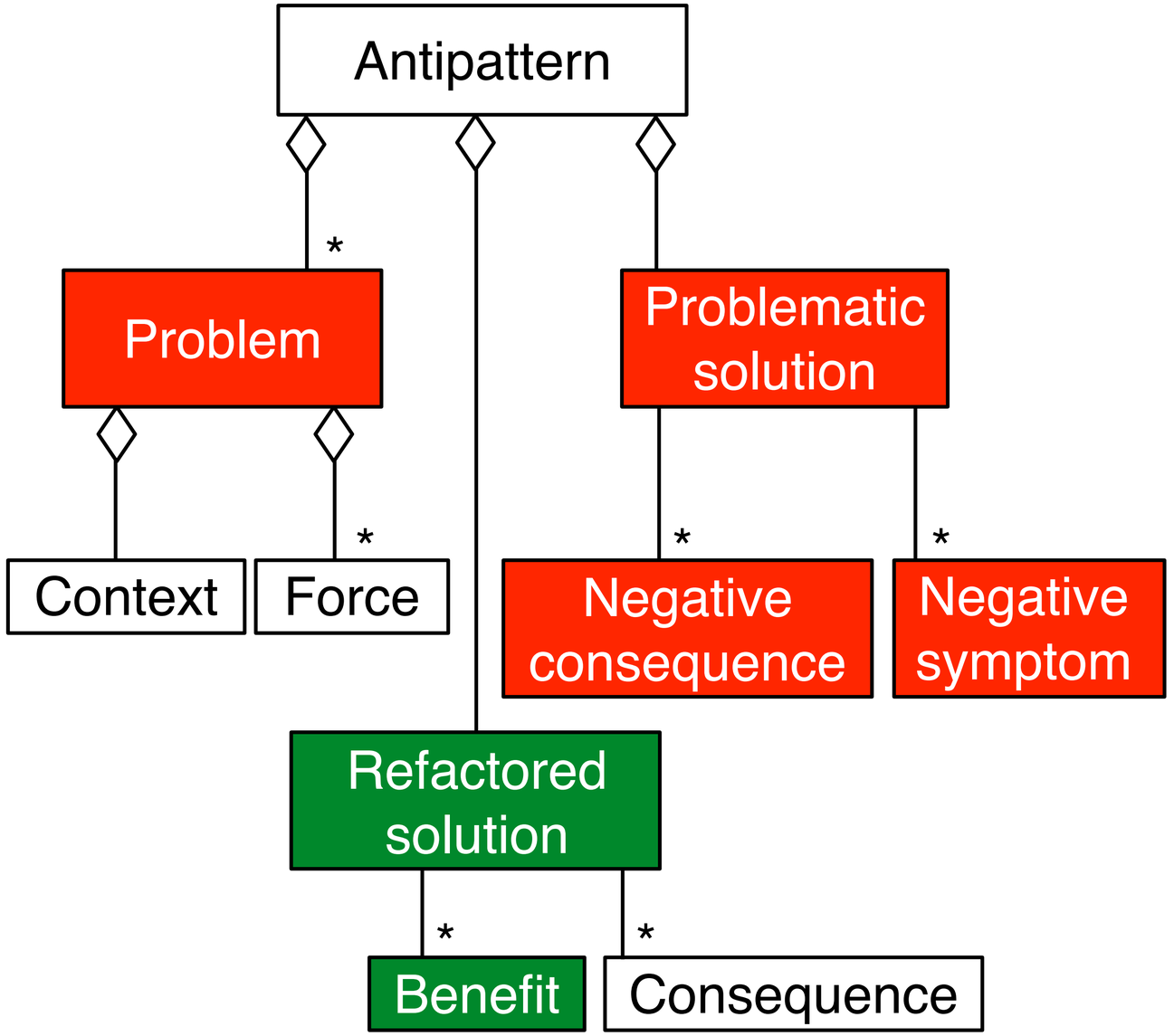
- The **problematic solution** describes a commonly occurring solution that generates overwhelming negative consequences and negative symptoms



Antipattern definition

An antipattern consists of
1 type of problem(s) and **2 solutions**

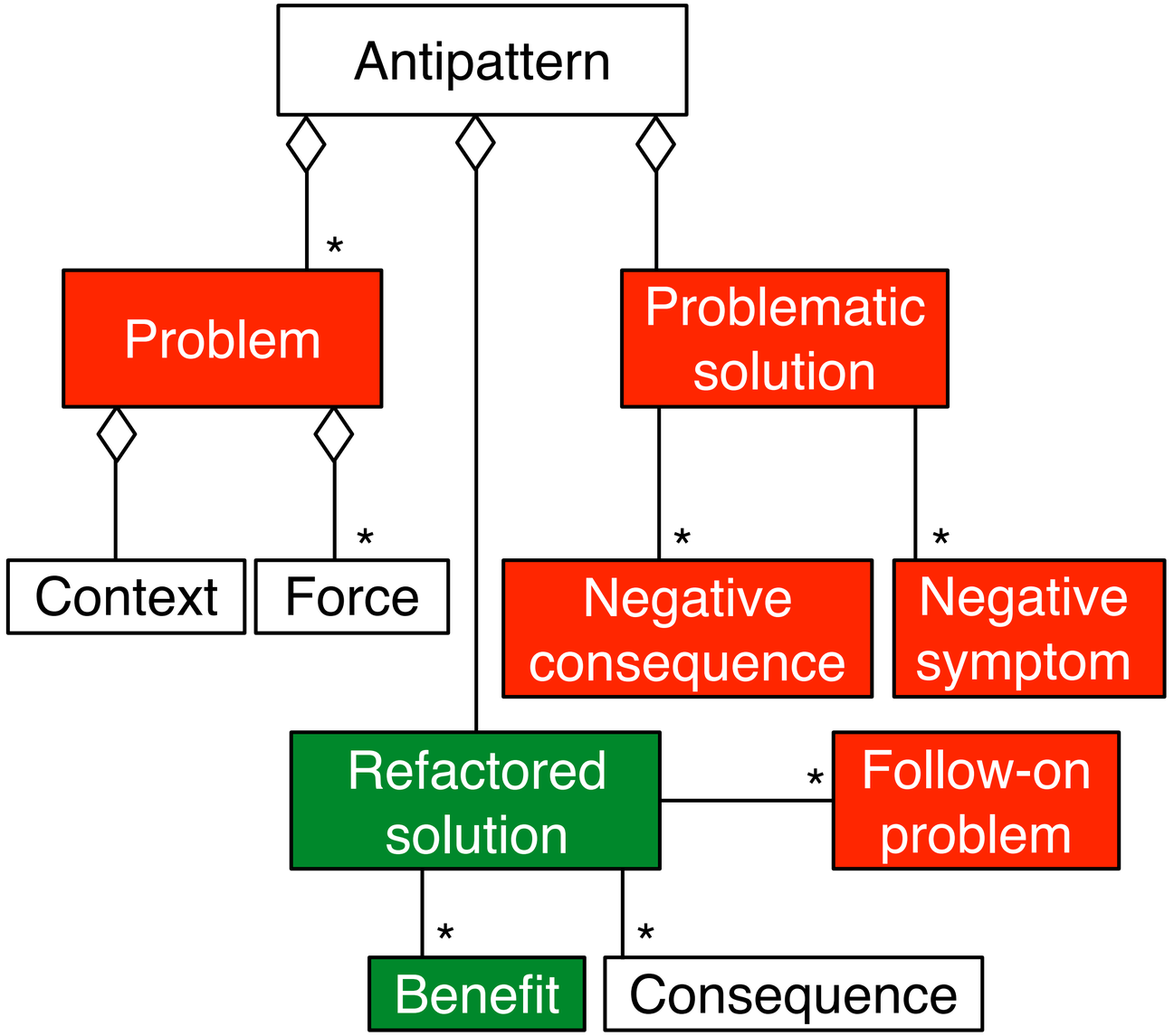
- The **problematic solution** describes a commonly occurring solution that generates overwhelming negative consequences and negative symptoms
- The **refactored solution** describes how the problematic solution can be reengineered to avoid these negative consequences and lead to benefits again



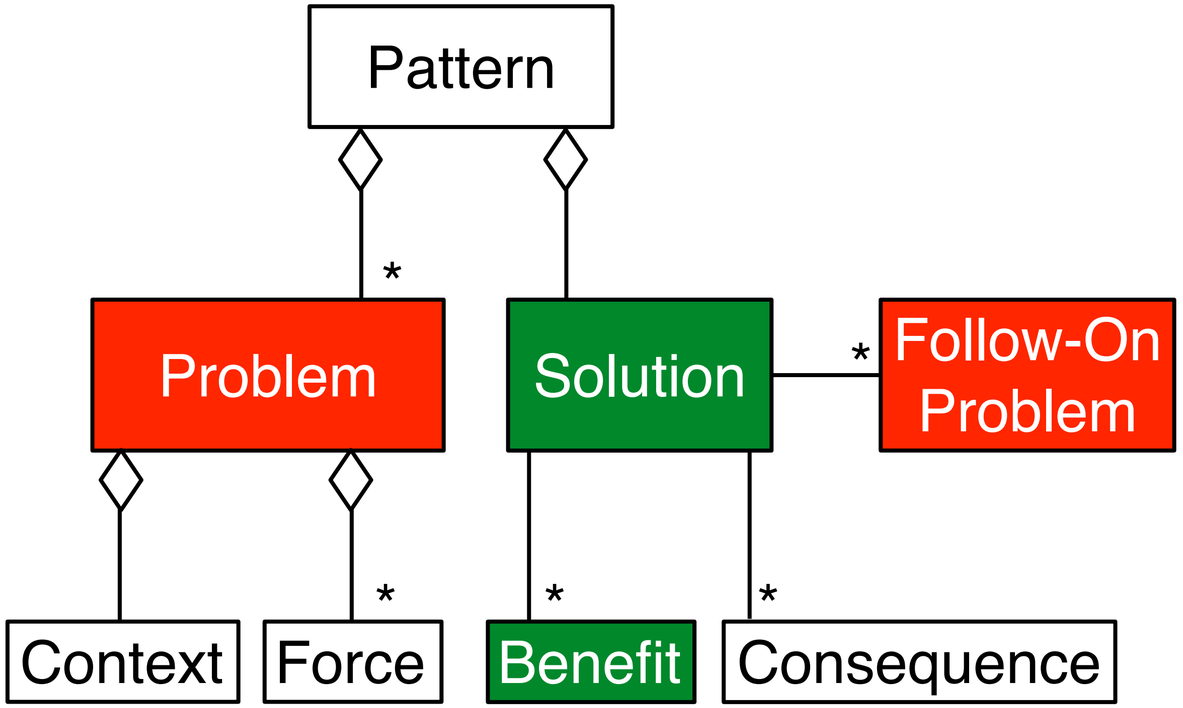
Antipattern definition

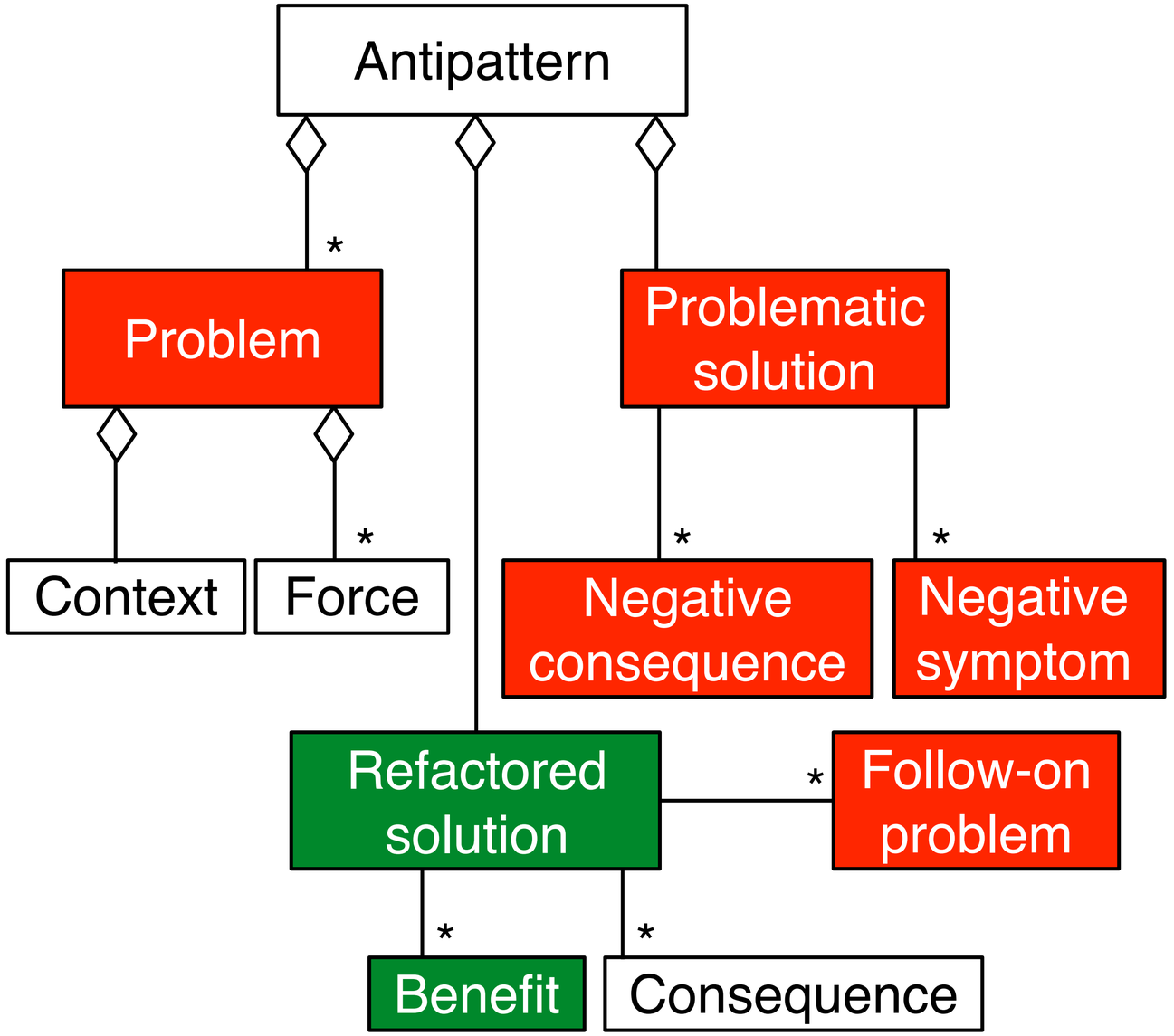
An antipattern consists of
1 type of problem(s) and **2 solutions**

- The **problematic solution** describes a commonly occurring solution that generates overwhelming negative consequences and negative symptoms
- The **refactored solution** describes how the problematic solution can be reengineered to avoid these negative consequences and lead to benefits again
- The **refactored solution** can lead again to follow-on problems



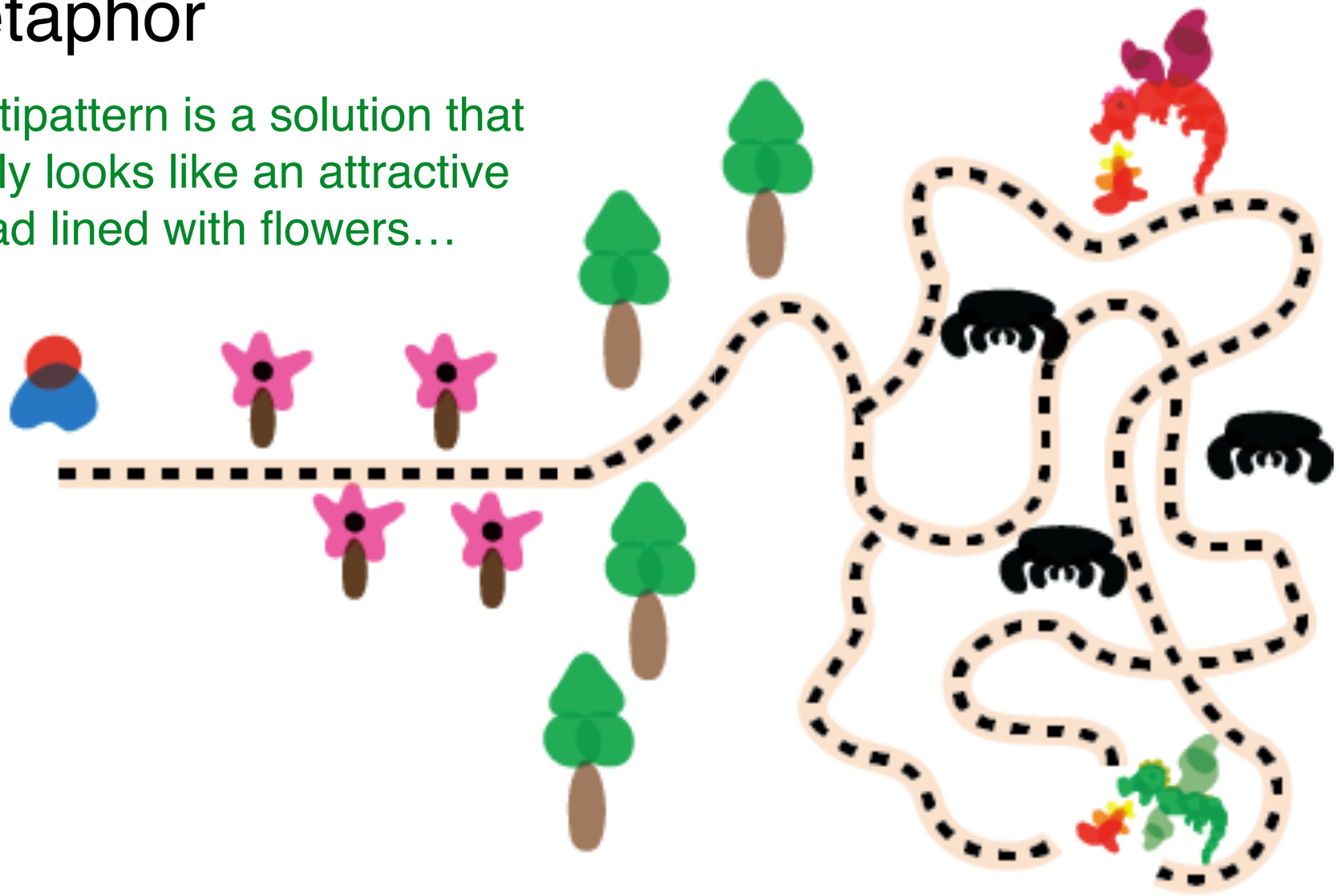
Pattern vs. antipattern





Antipattern metaphor

An antipattern is a solution that initially looks like an attractive road lined with flowers…



Source: Martin Fowler

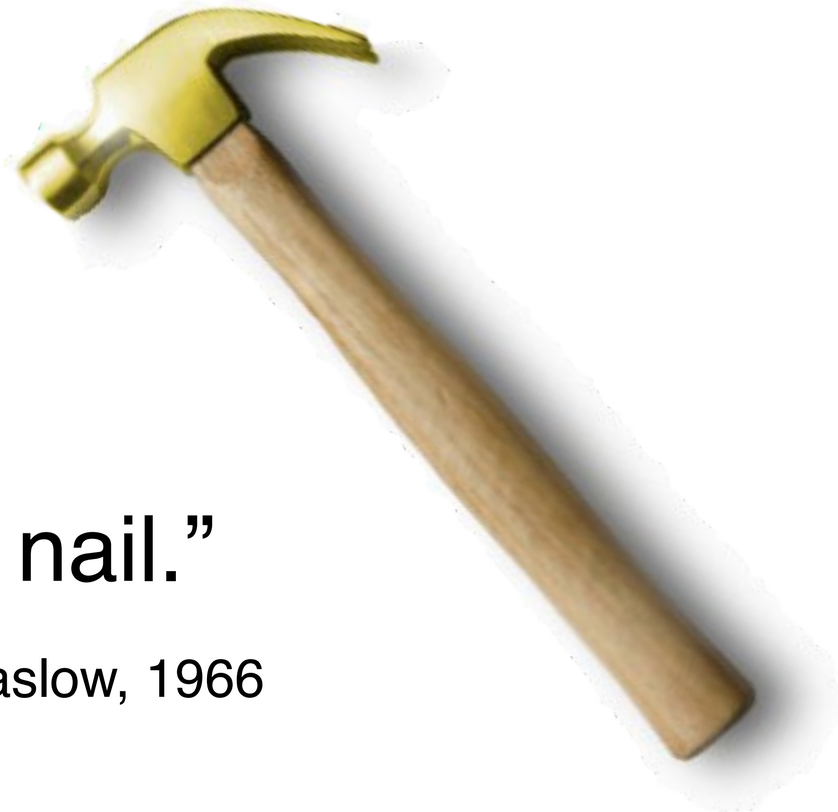
… but further on leads you into a maze filled with monsters

Outline

- Antipattern definition
- **Golden hammer**
- Functional decomposition
- Antipattern taxonomy
- Lava flow
- The blob

Golden hammer

“When the only tool you have is a hammer...

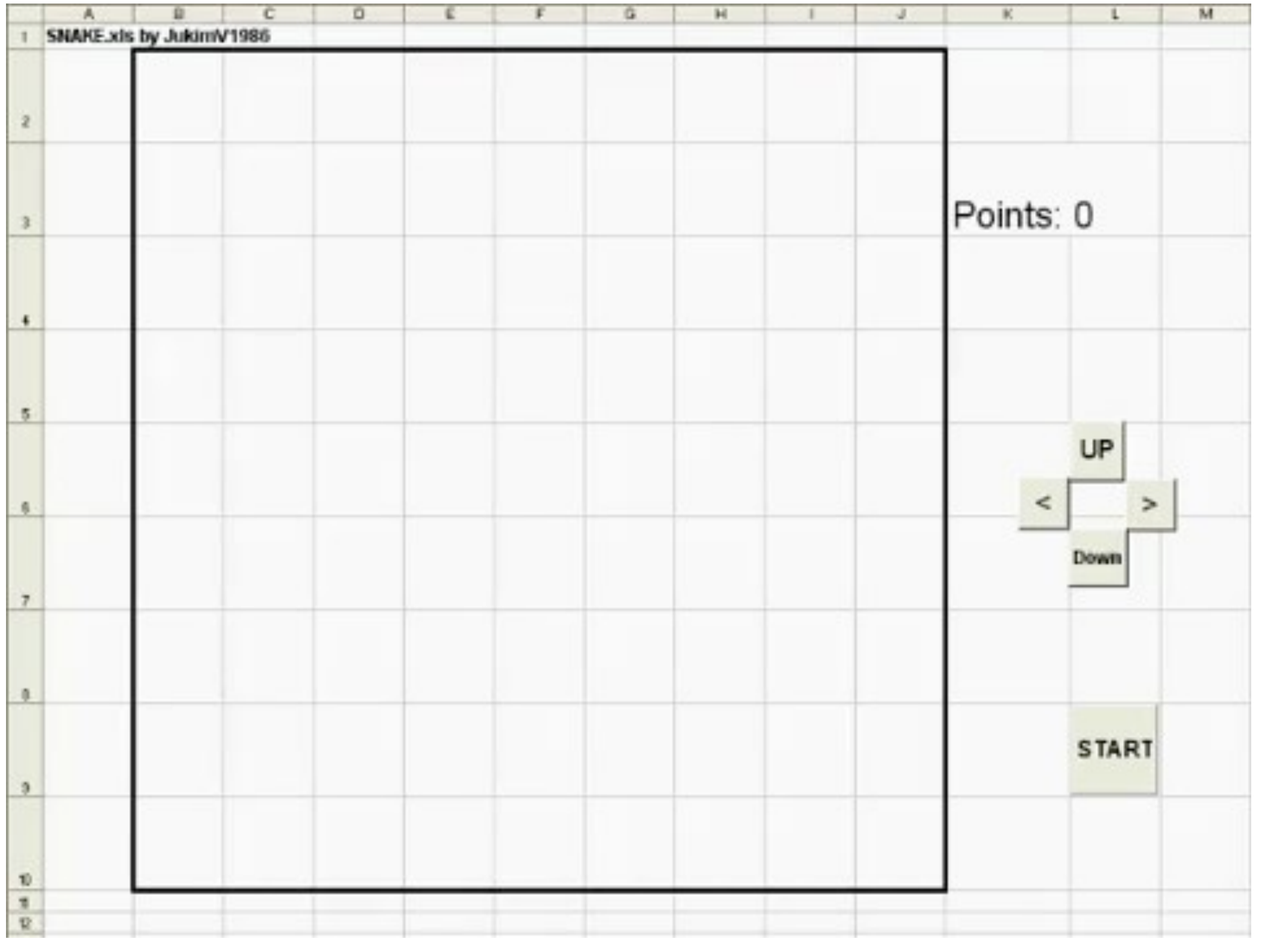


...it is tempting to treat everything as if it were a nail.”

— Abraham Maslow, 1966

A frequently used golden hammer: Excel

- Example: implementation of snake



https://codereview.stackexchange.com/questions/178135/drawing-a-snake-using-arrow-keys-in-excel-using-vba

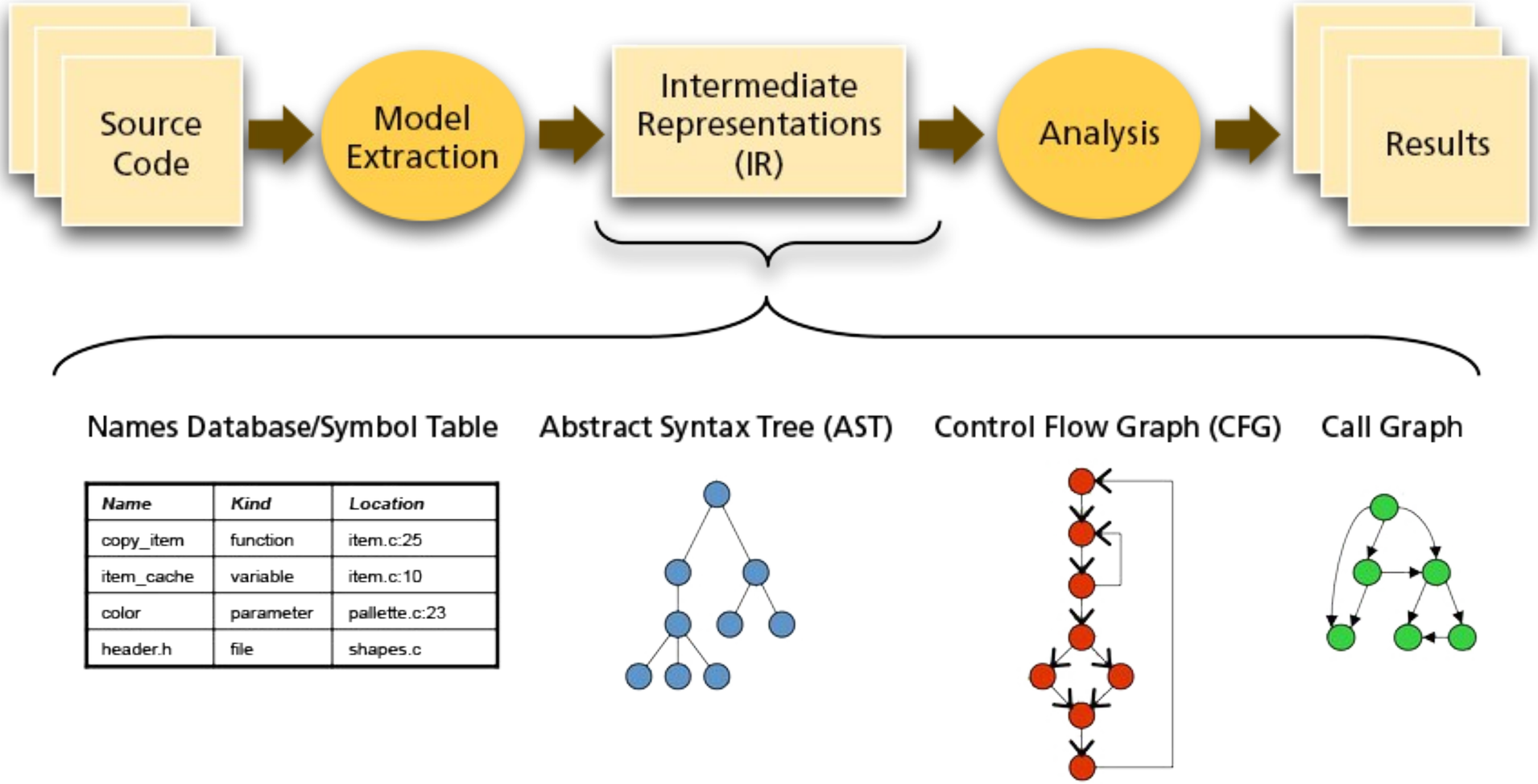
Golden hammer antipattern

- General form
  - Developer has a high level of competence in a particular solution
  - Every new development effort is solved with this solution
  - Developer is unwilling to learn and apply a new approach
- Symptoms and consequences
  - Identical tools are used for a wide array of diverse products
  - System architecture depends on a particular application suite and a specific vendor tool set
- Typical causes
  - Large investment has been made in products and specific technologies
  - Reliance on proprietary product features that are not available from other vendors or products
- Variations: obsessive use of a favorite software concept or design pattern
- Example: database centric environment with no additional architecture except that which is provided by the database vendor

Golden hammer antipattern

- Known exceptions: product is part of a vendor suite that provides for all the needs
- Refactored solution
  - Project organization develops commitment to explore new technologies
  - Software system is designed with well defined boundaries
  - Software developers keep up to date on technology trends
  - Management adopts a commitment to open systems and architectures
  - Management encourages hiring of people with different backgrounds
- Related solutions
  - Lava flow: when the golden hammer is applied over the course of several years and many projects
  - Vendor lock in: when the developer actively receives vendor support and encouragement in applying the golden hammer

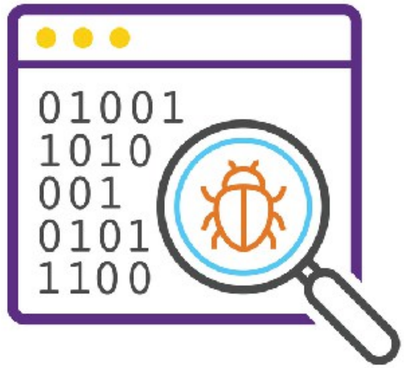
Excursion: static code analysis



Name	*Kind*	*Location*
copy_item	function	item.c:25
item_cache	variable	item.c:10
color	parameter	pallette.c:23
header.h	file	shapes.c

Excursion: static vs. dynamic analysis

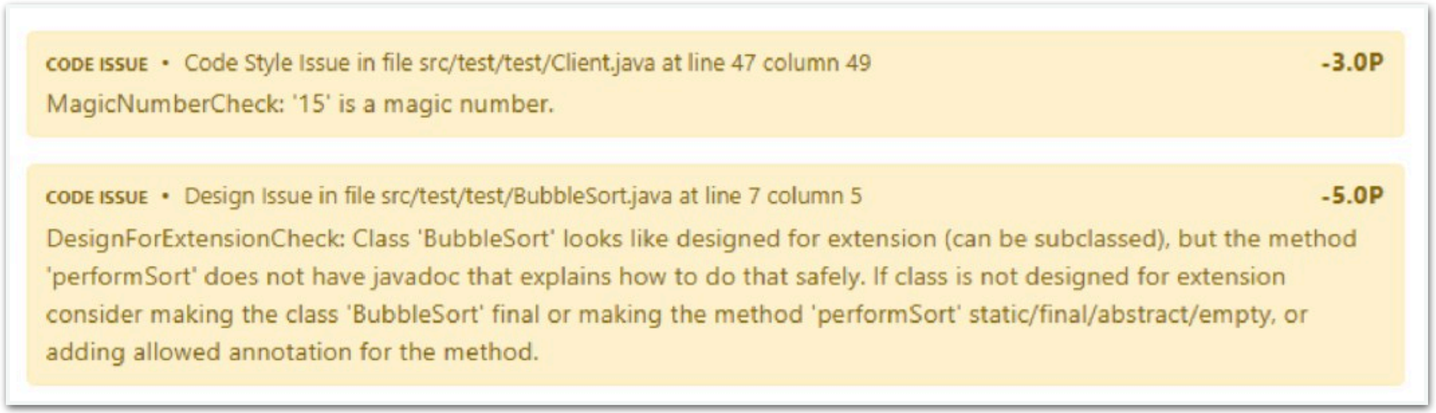
- Static analysis (no execution)
  - Manual execution by reading the source code
    - Walkthrough by informal presentation to others
    - Code inspection by formal presentation to others
  - Automated tools that check for
    - Syntactic and semantic errors
    - Deviation from coding standards

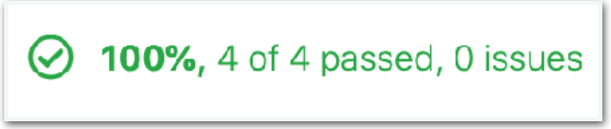


- Dynamic analysis (code execution)
  - Black box testing: test the input / output behavior
  - White box testing: test the implementation of the subsystem or class

Excursion: static code analysis on Artemis

- Penalty for each issue (minus points)
- Different categories of code issues





SpotBugs

https://spotbugs.github.io

PMD

https://pmd.github.io

checkstyle

https://checkstyle.sourceforge.io

L06E01 Golden Hammer

Not started yet.

Start exercise

Easy

Due Date: in 7 days

10 min

4 pts

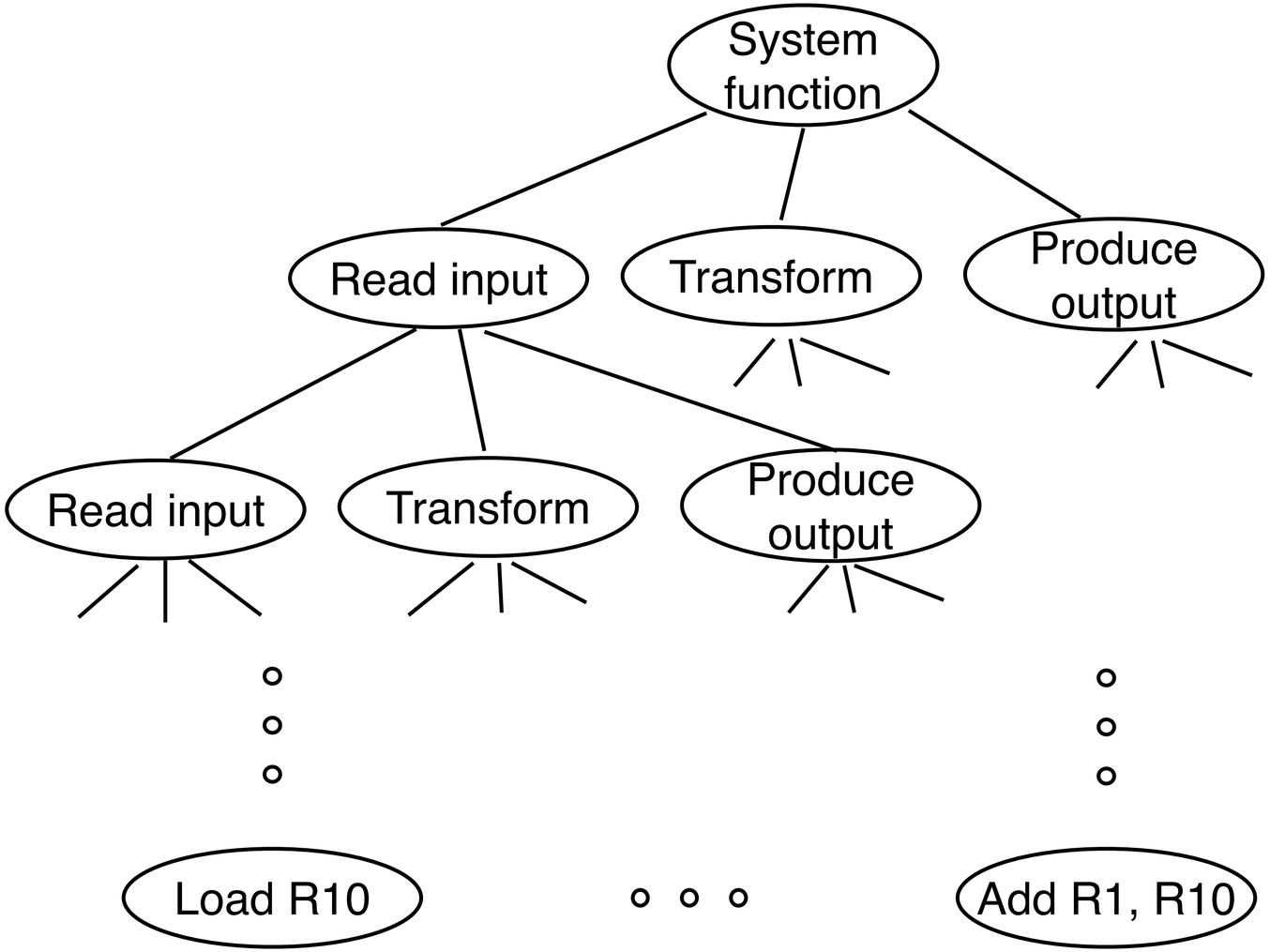
- Problem statement:

"Lists are the only data structure I know"

Outline

- Antipattern definition
- Golden hammer
- **Functional decomposition**
- Antipattern taxonomy
- Lava flow
- The blob

Functional decomposition: example



Top level functions

Level 1 functions

Level 2 functions

Machine instructions

Source: Tom DeMarco, Structured Analysis and System Specification, 1979

Functional decomposition

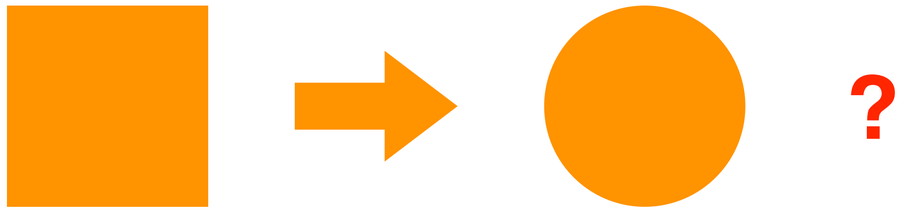
- Also known as **No OO**
- **General form:** everything is a function, lots of files called misc, util, util1, aux1, aux2…
- **Unbalanced forces:** management of complexity, change management is difficult
- Symptoms
  - The functionality is spread all over the system
  - Maintainer must understand the whole system to make a single change to the system
- Consequences
  - Source code is hard to understand
  - Source code is complex and impossible to maintain
  - User interface is often awkward and non-intuitive

Functional decomposition

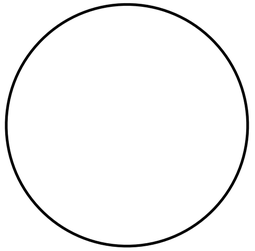
- **Typical causes**
  - Programmers have been trained only in an imperative / functional language
  - Designers have been trained with a functional decomposition method (DeMarco)
- **Refactored solutions**
  - Object oriented analysis
  - Object oriented reengineering (process)
- Refactored solution type: process

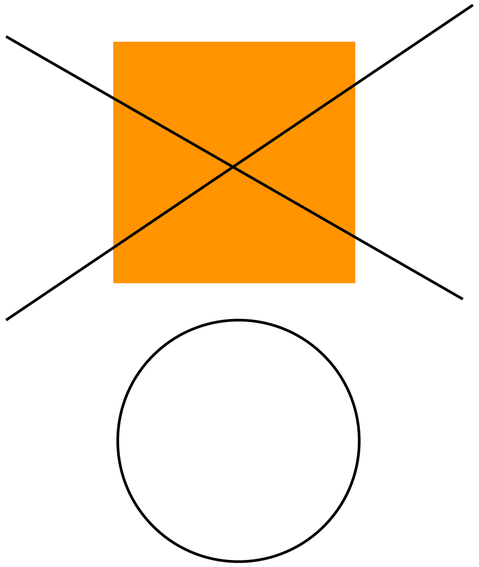
Problems with functional decomposition

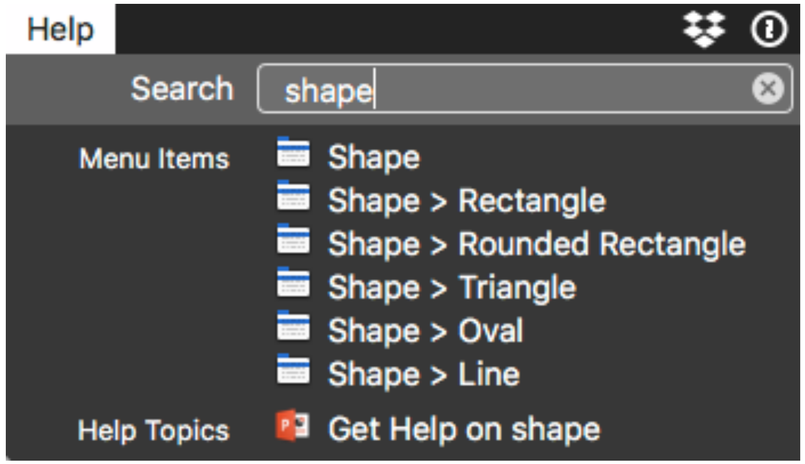
- Example: many drawing and presentation programs work with the concept of shapes
  - Types of shapes: rectangles, ovals, circles, etc.
  - Operations on shapes: draw, change, zoom, move
- Specific example: shapes in Microsoft Powerpoint
- How do I change a rectangle into a circle?

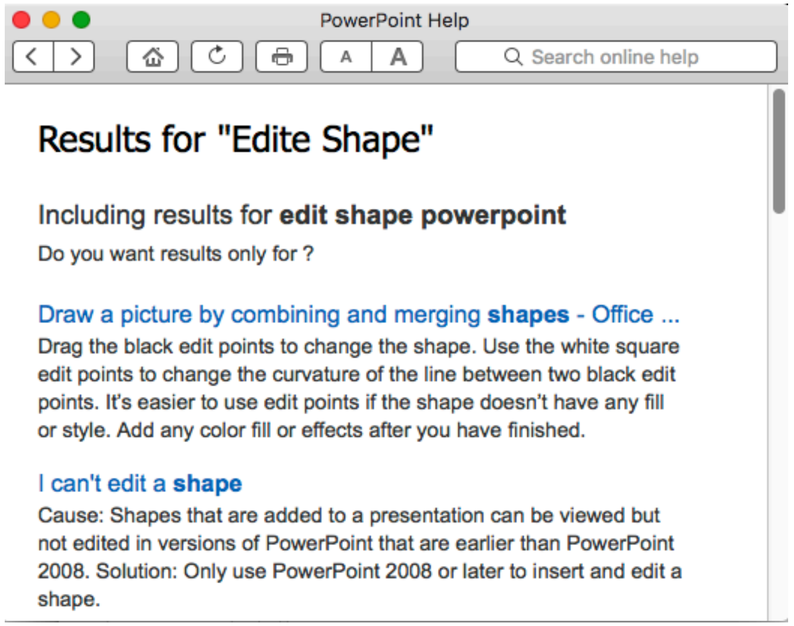


How do I change a rectangle into a circle?







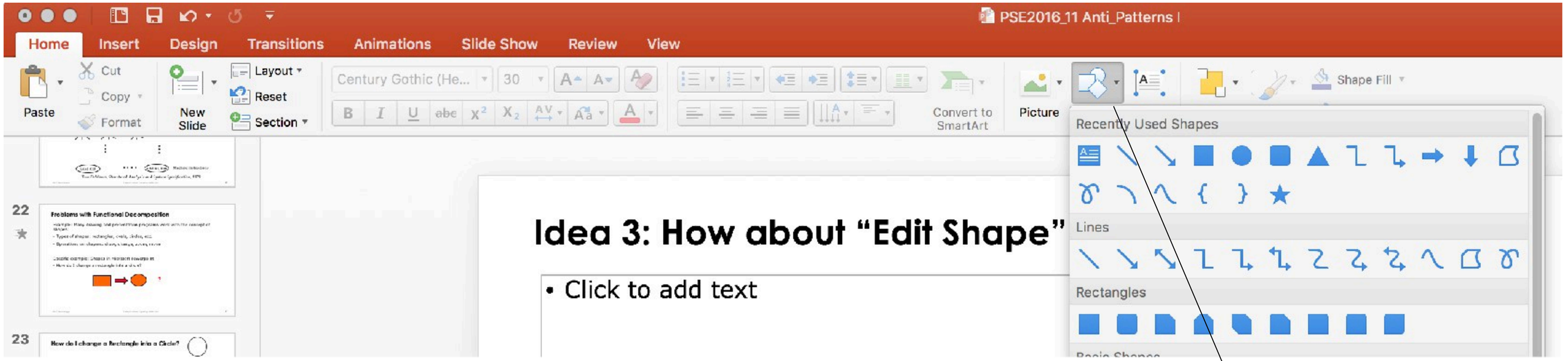


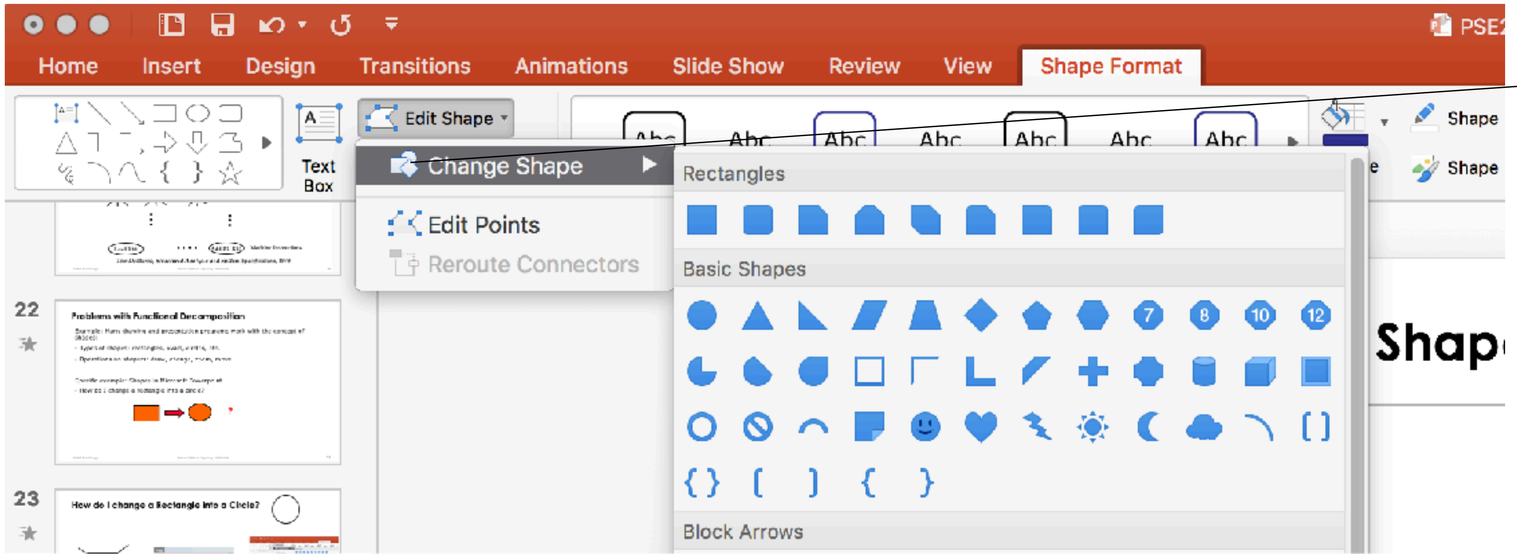
Idea 1: delete and redraw

Idea 2: invoking help…

Idea 3: how about “Edit Shape?”

“Edit shape” – Powerpoint Office 365

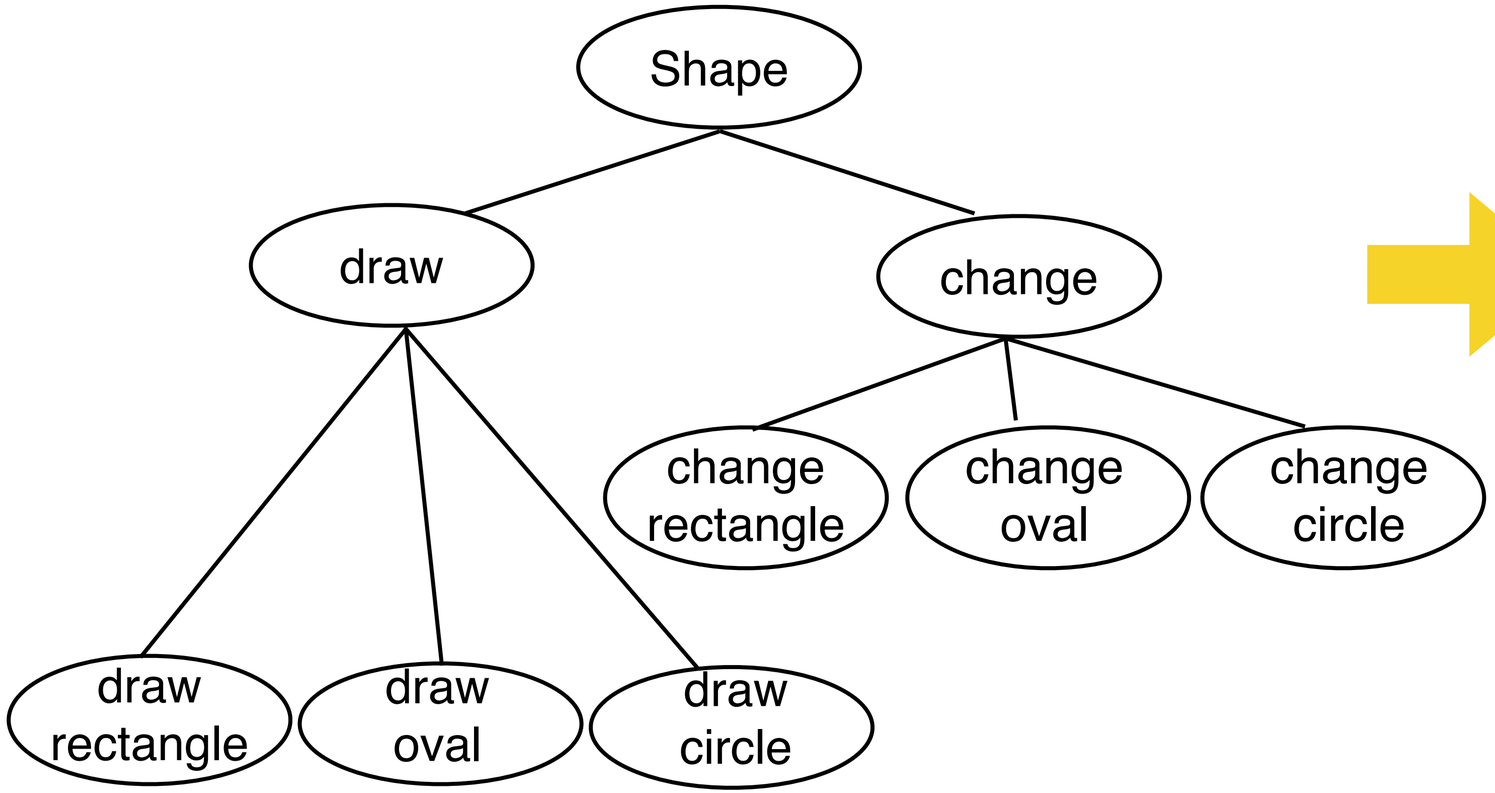




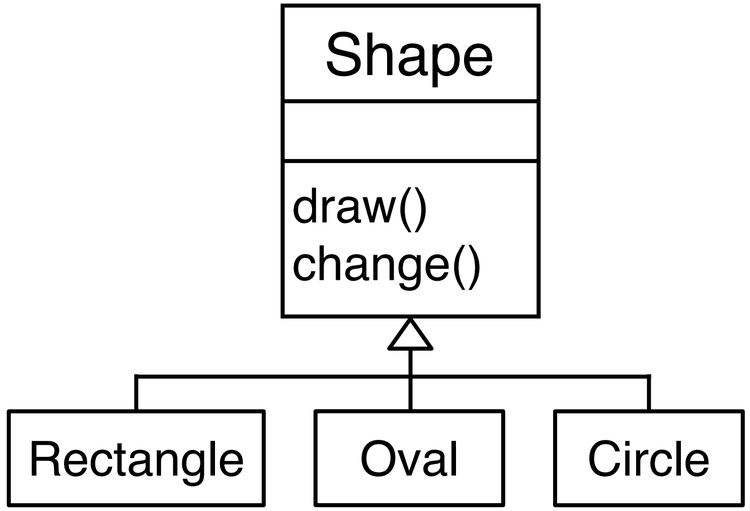
Same icon, different semantics!
Insert vs. Edit Form

Functional decomposition: example

Problematic solution: functional decomposition



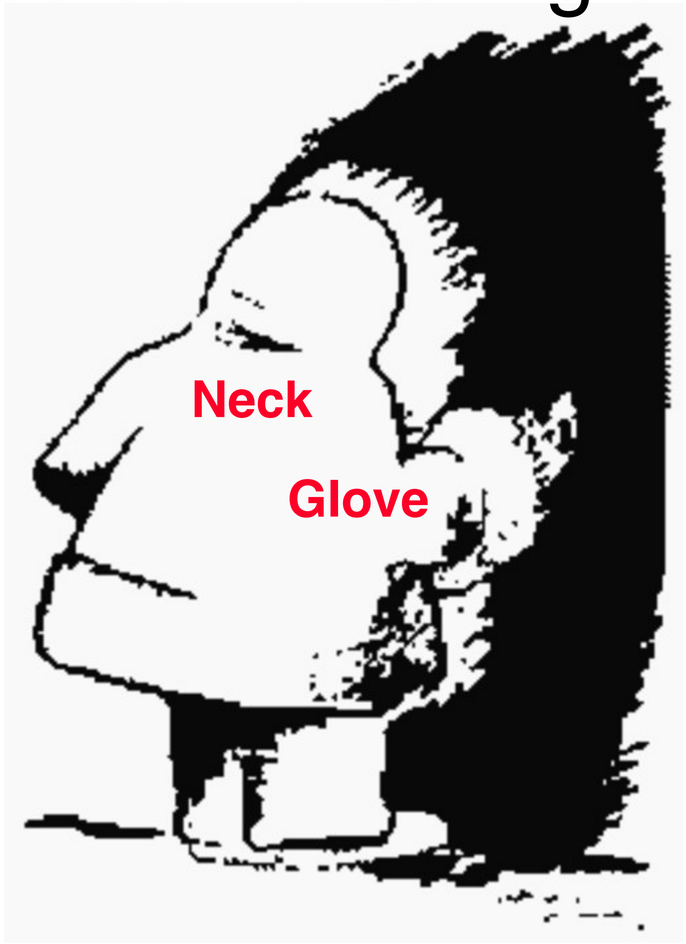
Refactored solution: object oriented analysis



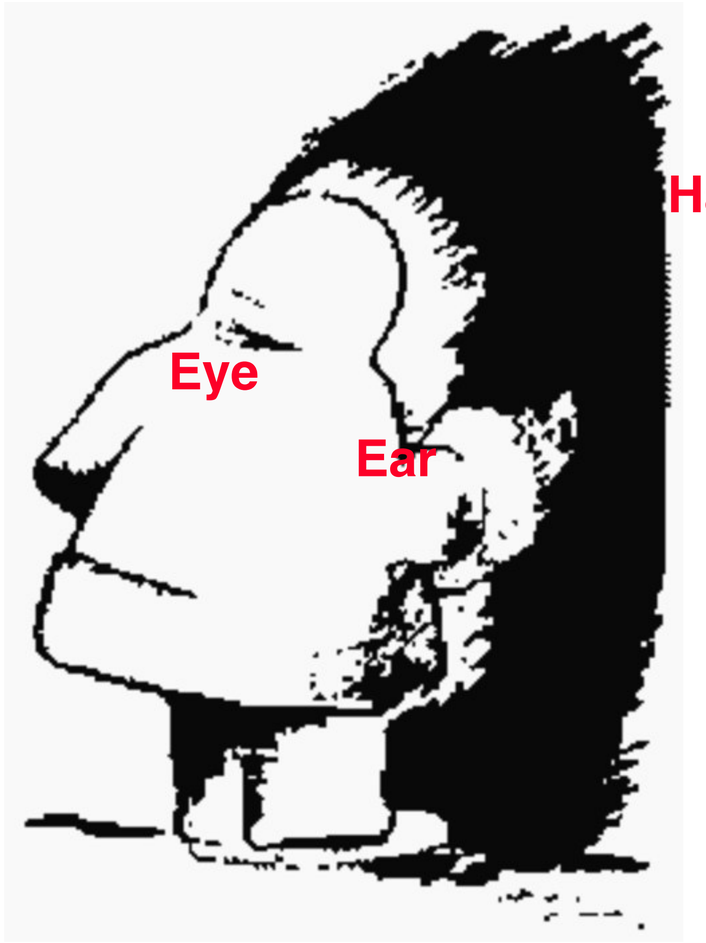
➡ However, even object orientation is not always the best solution (**follow-on problem**)

Object oriented analysis also has problems

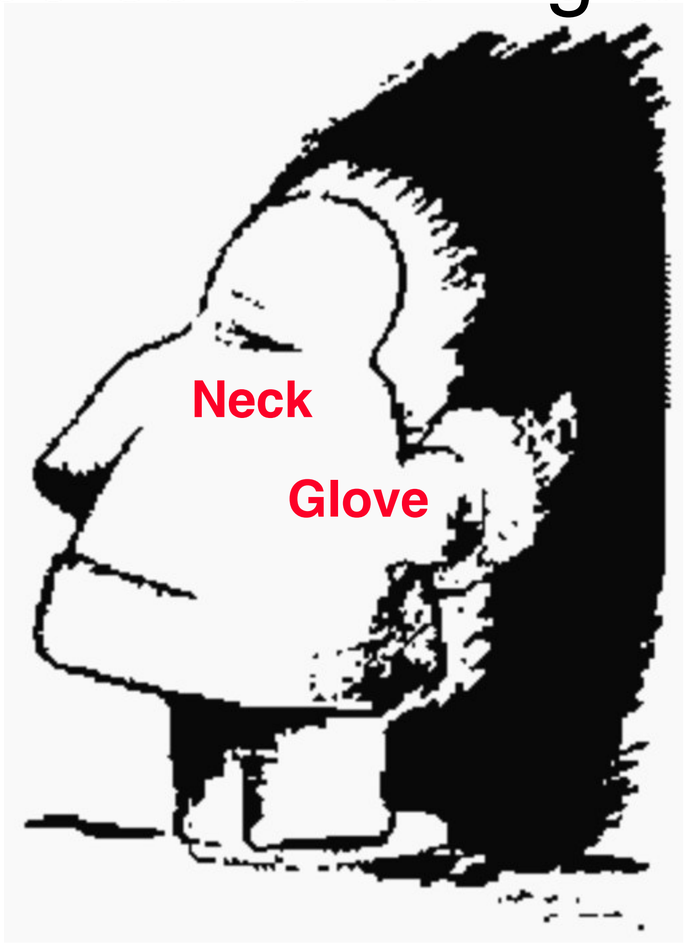
A person entering a cave!



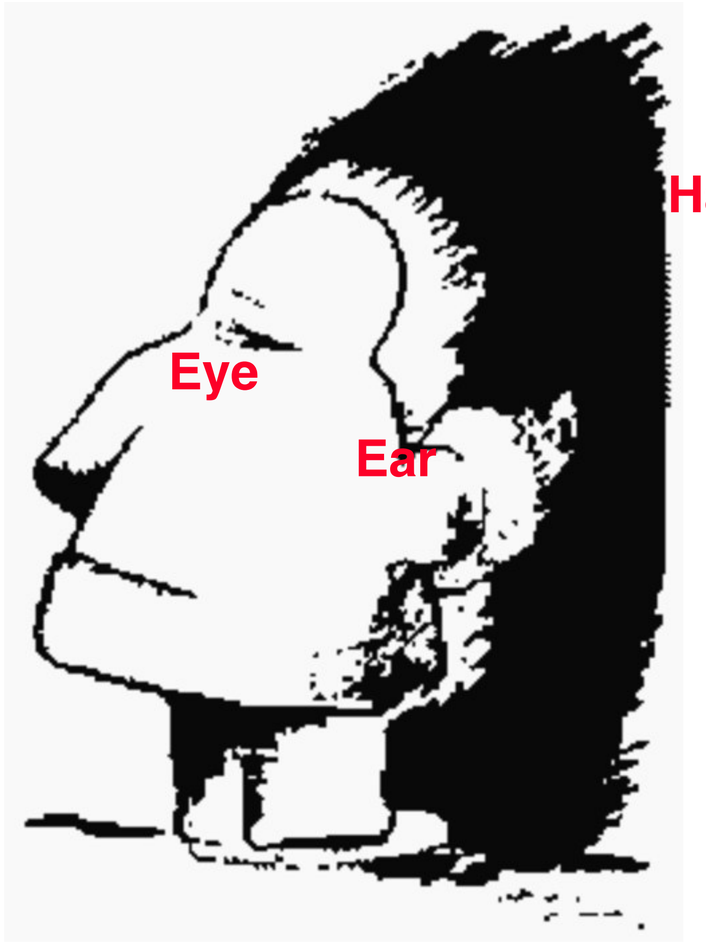
A face!



A person entering a cave!



A face!



Which object decomposition is correct?

An inuit!

Cave

Neck

Elbow

Glove

Pocket

Coat

A face!

Hair

Eye

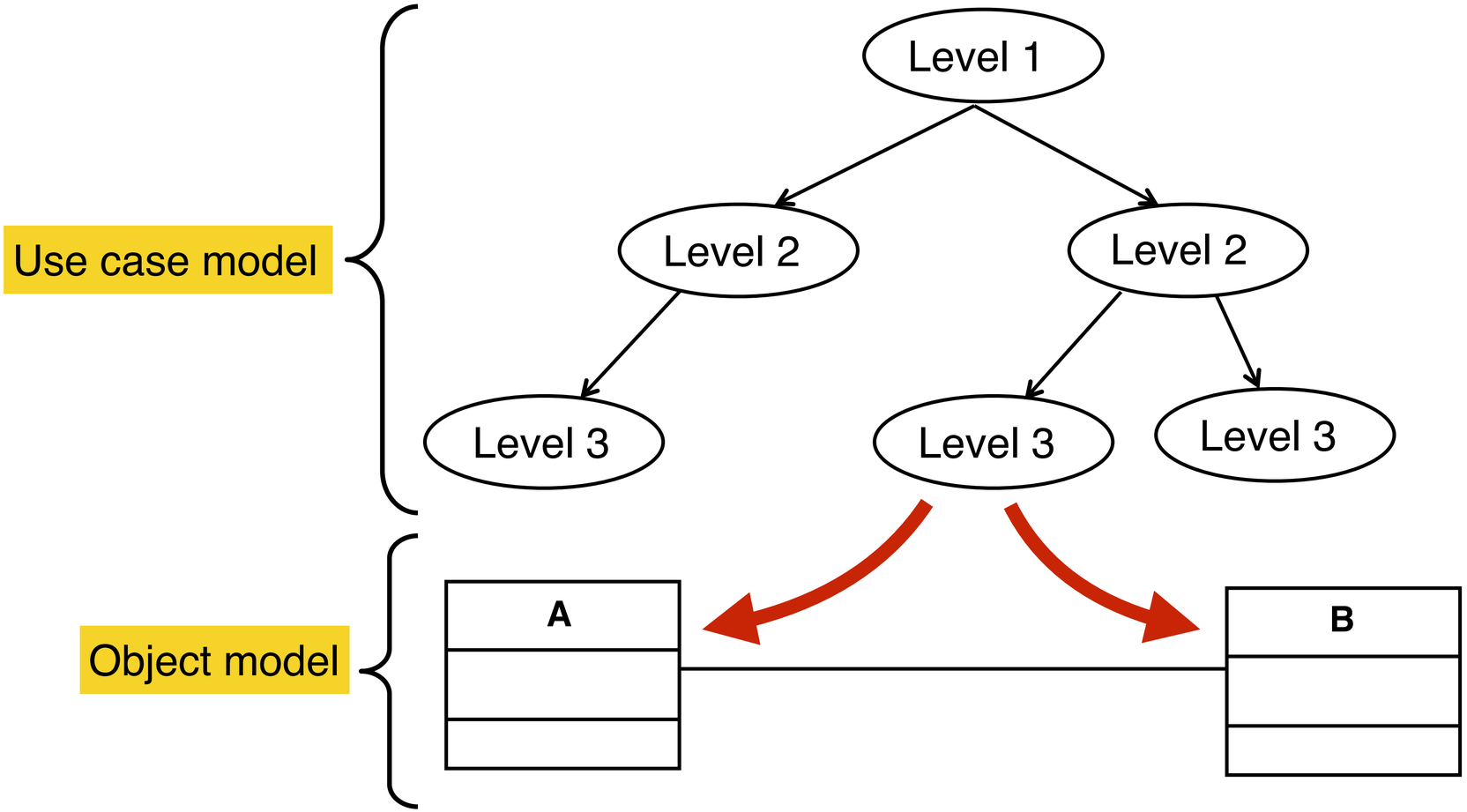
Nose

Ear

Mouth

Chin

Best practice: first identify use cases, then identify objects



Top level use cases
(“business processes”)

Level 2 use cases

Level 3 use cases

A and B are called
participating objects

L06E02 Functional Decomposition

Not started yet.

Start exercise

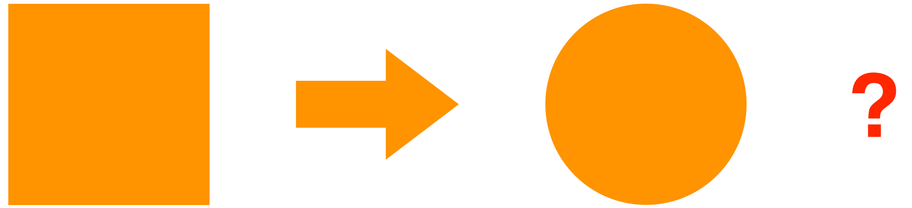
Medium

Due Date: in 7 days

15 min

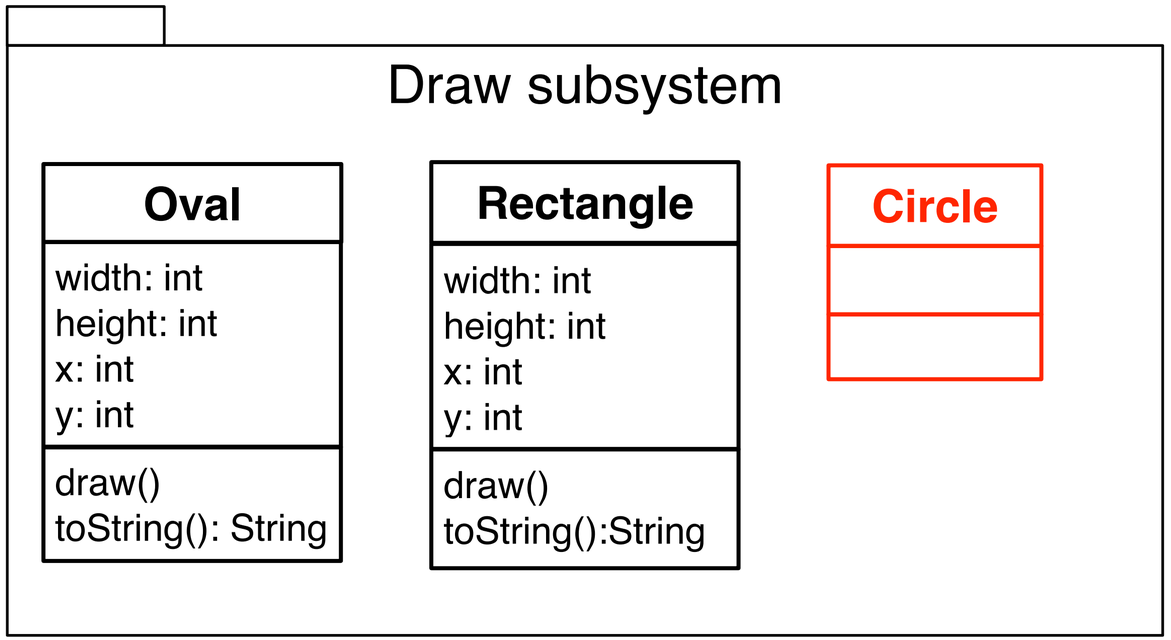
8 pts

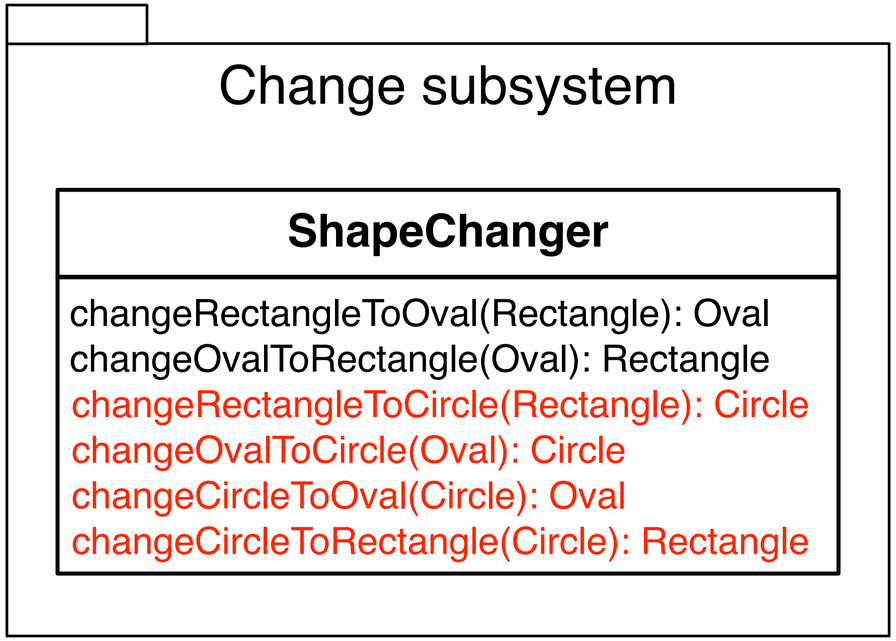
- Problem statement: changing shapes
  - Introduce an abstract `Shape` class
  - Make `Oval` and `Rectangle` extend the new `Shape`
  - Implement the `changeForm` method
  - Add a new shape `Circle`
  - Deprecate `ShapeChanger`
- Benefits
  - Shape instance can be changed at runtime
  - Adding a new shape subclass should be possible without changing the code for the existing shapes



Hint: problem with functional decomposition

- The legacy system contains a subsystem for drawing shapes
  - New requirement: support also changing of shapes (example: change an oval into a rectangle)
  - The manager says: “do not touch the existing code!”
- The functional decomposition leads to the code implementing change functionality being in a separate change subsystem
  - For each pair of shapes two functions would have to be implemented to perform the changes
  - Adding new shapes is expensive





Outline

- Antipattern definition
- Golden hammer
- Functional decomposition
- **Antipattern taxonomy**
- Lava flow
- The blob

Applicability of patterns and antipatterns

- Patterns are good for problems which have no solution yet
  - Innovation projects (green field engineering projects)
- Patterns emphasize the use of proven good design principles
  - Information hiding, high cohesion, low coupling, ...
- Antipatterns are good for emphasizing the recognition of mistakes
- Examples of antipatterns
  - Bad design
  - Too much time spent in analysis
  - Constantly missing deadlines
  - **Typical excuse:** “I knew about this design shortcut, but I had no choice because of {budget, deadline, missing resources,…}”

Changing bad solutions into better ones

- When a pattern becomes an antipattern, it is useful to have an approach to change the problematic (bad) solution into a better one
- Incremental Reengineering (refactoring)
  - The process of incrementally changing the bad structure of a system, project or organization into a better structure with the use of antipatterns

Why antipatterns?

- “The study of antipatterns is an important research activity.
- The presence of good patterns in a successful system is not enough.
- You also must show that those patterns are absent in unsuccessful systems.
- Likewise, it is useful to show the presence of certain patterns in **unsuccessful systems**, and their absence in **successful systems**.”

Jim Coplien [Antipattern Wiki]

➡ It is also useful to show the presence of certain patterns in **unsuccessful projects**, and their absence in **successful projects**

Typical mistakes in software development

What do you have to do to kill a project?

- Show the same demo twice to the same audience
- Focus on technologies, not on problems and scenarios
- Don’t maintain consistency between releases
- Isolate team efforts from other teams within an organization
- Rewrite existing clients, servers and applications
- Change the purpose of the system so that the models describe the wrong objects

Source: How to kill a software project [Brown 1998]

Root causes for antipatterns

- Most common mistakes in software project management and development:
  - Insufficient communication with the client
  - Unfulfilled requirements
  - Insufficient testing
  - Cost overruns and schedule slips
- Reason for these mistakes: “The 7 deadly sins”
  - **Cardinal sins:** pride, greed, lust, envy, gluttony, wrath and sloth
    - SALIGIA (Latin): superbia, avaritia, luxuria, invidia, gula, ira, acedia
  - Now a popular analogy used to identify ineffective practices in software project management
    - Haste, pride, apathy, sloth, ignorance, avarice, narrow-mindedness

Pieter Bruegel: The Seven Deadly Sins

The seven deadly sins in software practice

1) Haste
Solutions based on hasty decisions (“time is most important”) lead to compromises in software quality

2) Pride (hubris)
Not invented here (NIH): Not willing to adopt anything from the outside

3) Apathy
Not caring about a problem, followed by unwillingness to attempt a solution

4) Sloth
Making poor decisions based on “easy” answers

5) Ignorance
Failure to seek understanding

6) Avarice (excessive complexity)
No use of abstractions, excessive modeling of details

7) Narrow-mindedness
The refusal to use solutions that are widely known (“Why reuse? I only have to solve one problem”)

Antipattern template ([Brown])

Name

- Select a word that expresses contempt

Also Known As

- Additional popular phrase

Most Frequent Scale

- Keywords from the reference model

Refactored Solution Name

- Name of refactored solution patterns

Refactored Solution Type

- Software, Technology, Process or Role

Root Causes

- Keywords from the reference model

Unbalanced Forces

- Identifies the forces that have been ignored or misused in this pattern

Anecdotal Evidence

- Stories, urban legends

Background

- Additional useful material

General Form

- Diagram and textual description

Symptoms and Consequences

- Symptoms resulting from the pattern

Typical Causes

- More causes in addition to root causes

Known Exceptions

- Situations where this pattern is ok

Refactored Solution

Variation

Example

Related Solutions

Applicability to Other Viewpoints and Scales

Antipattern template ([Brown])

Name

Also Known As

Most Frequent Scale

Refactored Solution Name

Refactored Solution Type

Root Causes

Unbalanced Forces

Anecdotal Evidence

Background

General Form

Symptoms and Consequences

Typical Causes

Known Exceptions

Refactored Solution

- Explains a single new solution that resolves the forces identifier in the unbalanced forces section

Variation

- Optional section that describes major variations of the Antipattern and one or more alternative Solutions

Example

- Demonstrates how the refactored solution can be applied to the problem

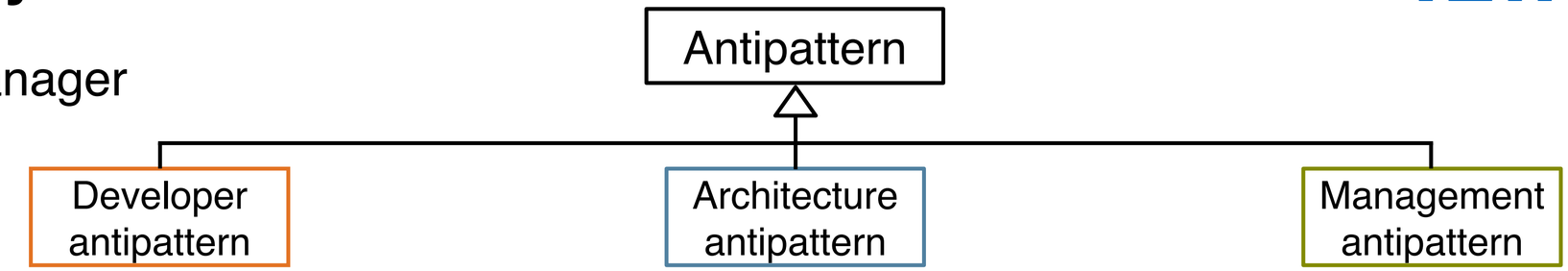
Related Solutions

- Related pattern and antipatterns

Applicability to Other Viewpoints and Scales

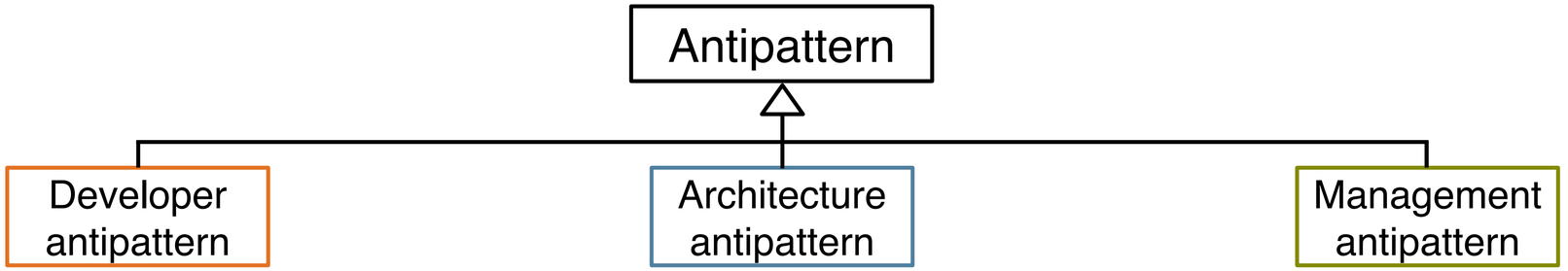
- Describes how the antipattern impacts other roles and patterns in other scales

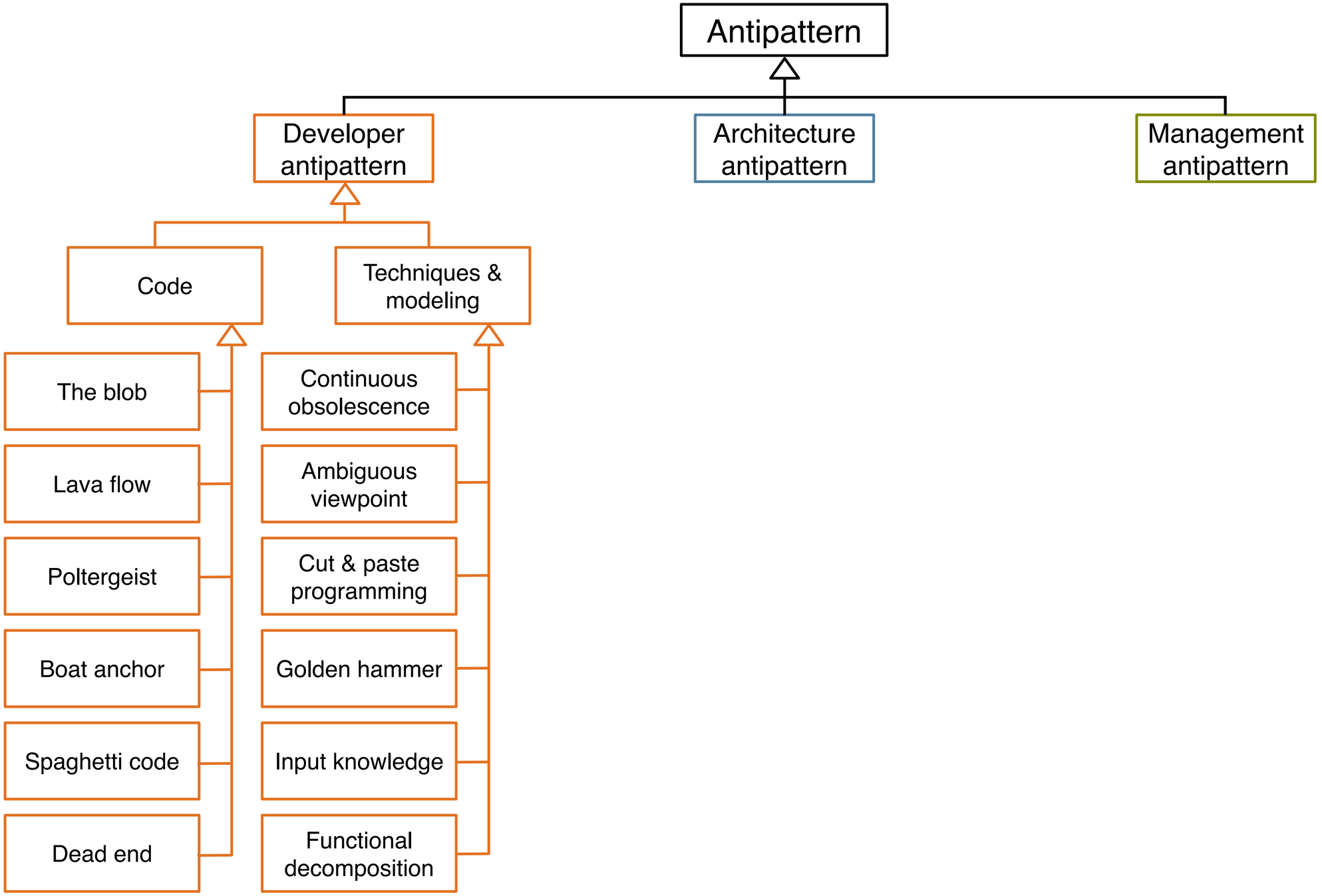
Antipattern taxonomy

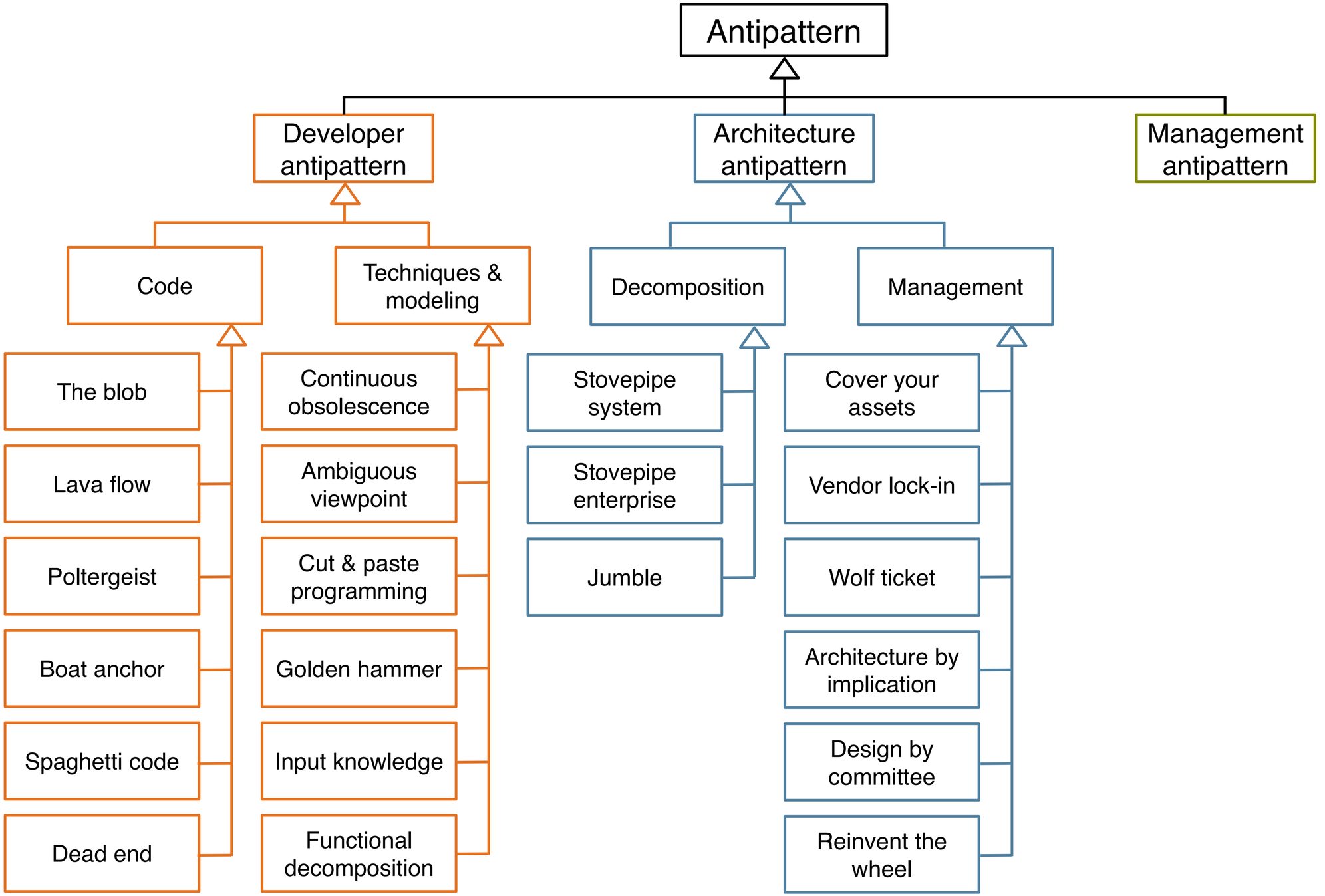


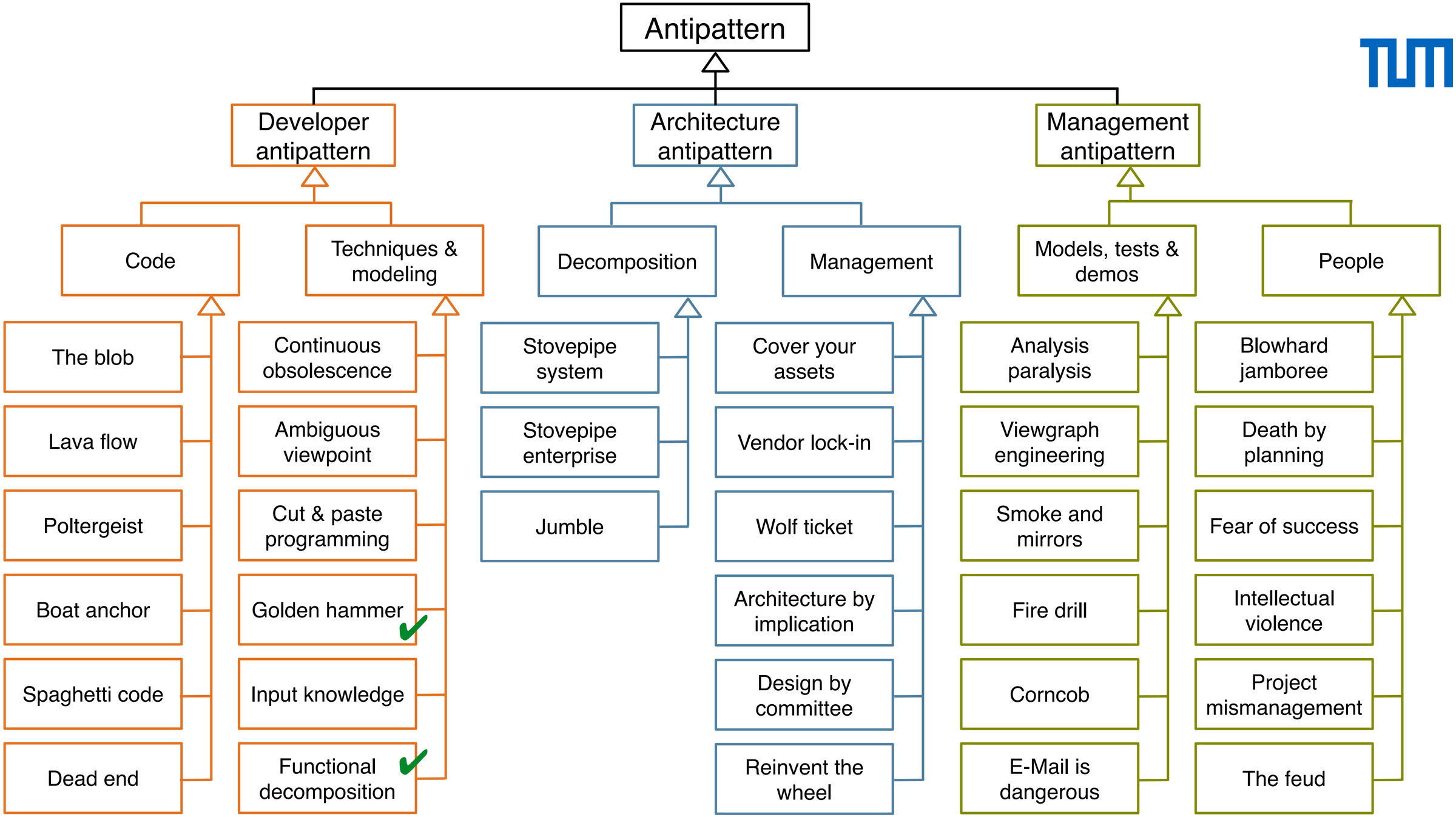
- **3 roles:** developer, architect, manager
  - ➡ 3 types of antipatterns
- **Development antipatterns**
  - Focus on the viewpoint of the software developer
  - **Issues:** software refactoring, modification of source code to improve the software structure with respect to long-term maintainability
- **Architecture antipatterns**
  - Focus on the viewpoint of the software architect
  - **Issues:** partitioning of subsystems and components, platform independent definition of interfaces, and connectivity of components
- **Management antipatterns**
  - Focus on the viewpoint of the software project manager
  - **Issues:** software project organization, software project management, software process model, human communication, rationale management and resolution of issues

Find more info on https://sourcemaking.com/antipatterns


Antipattern
Developer antipattern
Architecture antipattern
Management antipattern

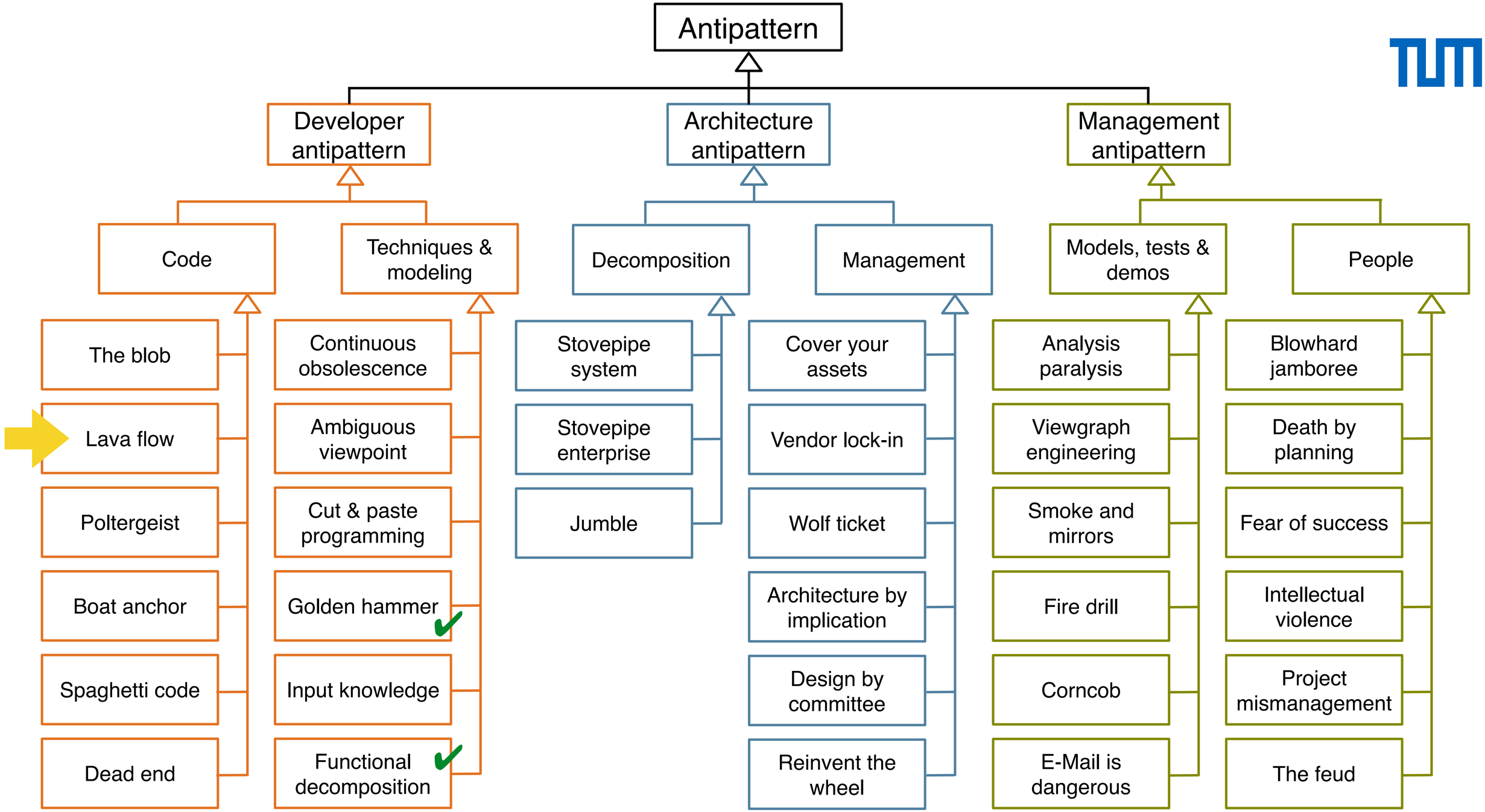






Outline

- Antipattern definition
- Golden hammer
- Functional decomposition
- Antipattern taxonomy
- **Lava flow**
- The blob



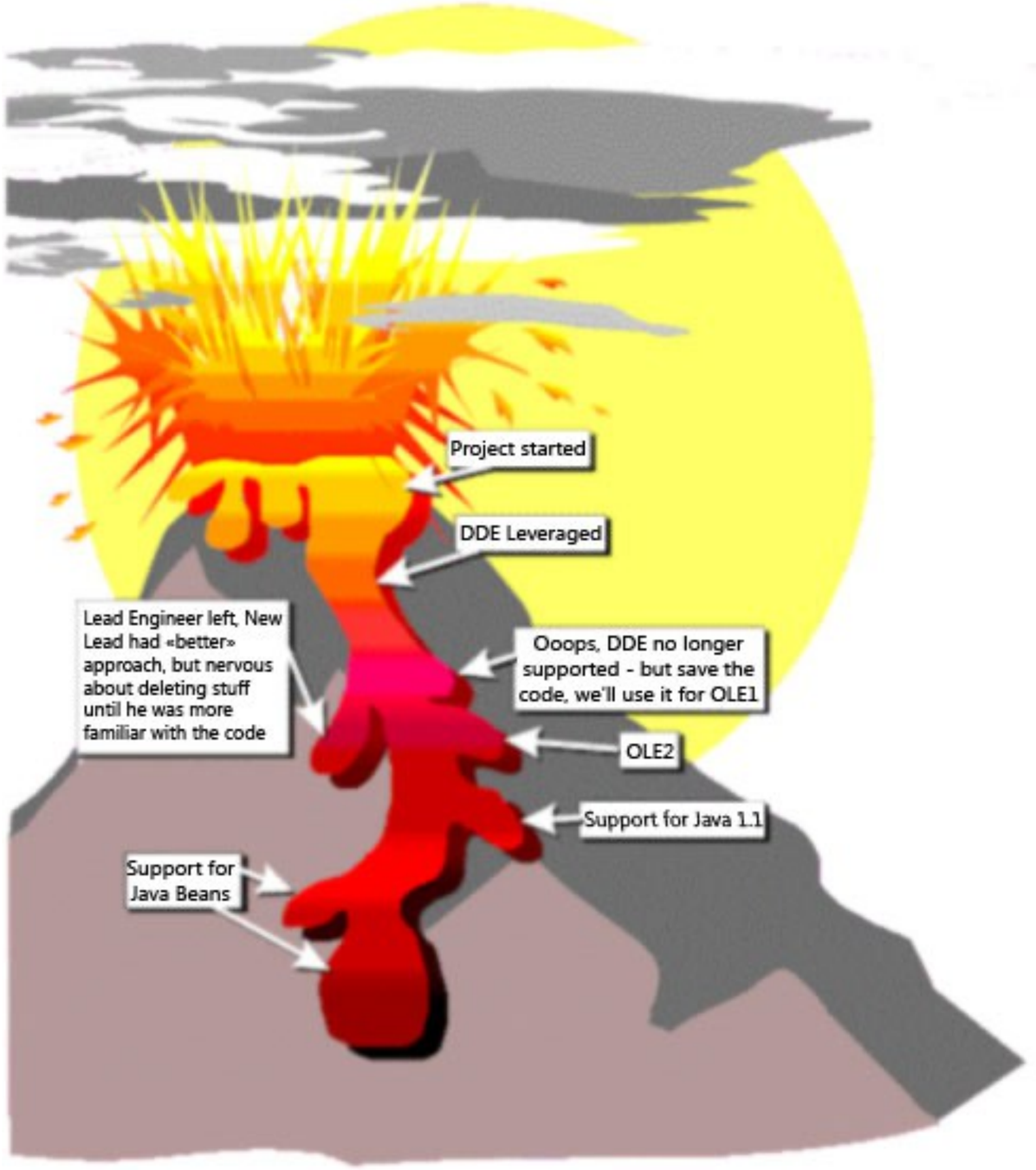

Lava Flow

Lava flow antipattern

- Also known as: **dead code**
- General form: lava like “flows” of previous development hardened into a basalt like mass of code, difficult to remove once it has solidified
- Symptoms and consequences
  - Unused or commented out code; undocumented complex, important looking code
  - Functions or classes that don’t relate to the system architecture
  - “Evolving” architecture
- Typical causes
  - R&D code placed into production
  - Implementation of several trial approaches toward implementing some functionality
  - High programmer turnover rate
  - Fear of breaking something and not knowing how to fix it
  - Architectural scars; unclear, repeatedly changing project goals

Lava flow antipattern

- Known exceptions: small-scale, rapidly developed throwaway prototypes
- **Refactored solution**
  - Architecture centric management
  - Ensure that sound architecture precedes production code development
  - Architecture must be backed up by configuration management process
  - Avoid architecture changes during active development
- Applicability to other viewpoints and scales
  - Architect plays a key role in preventing lava flows
  - Managers must postpone further development when lava flow is first identified until a clear architecture is defined


Project started
DDE Leveraged
Lead Engineer left, New Lead had «better» approach, but nervous about deleting stuff until he was more familiar with the code
Ooops, DDE no longer supported - but save the code, we'll use it for OLE1
OLE2
Support for Java 1.1
Support for Java Beans

L06E03 Lava Flow

Start exercise

Easy

Not started yet.

Due Date: in 7 days

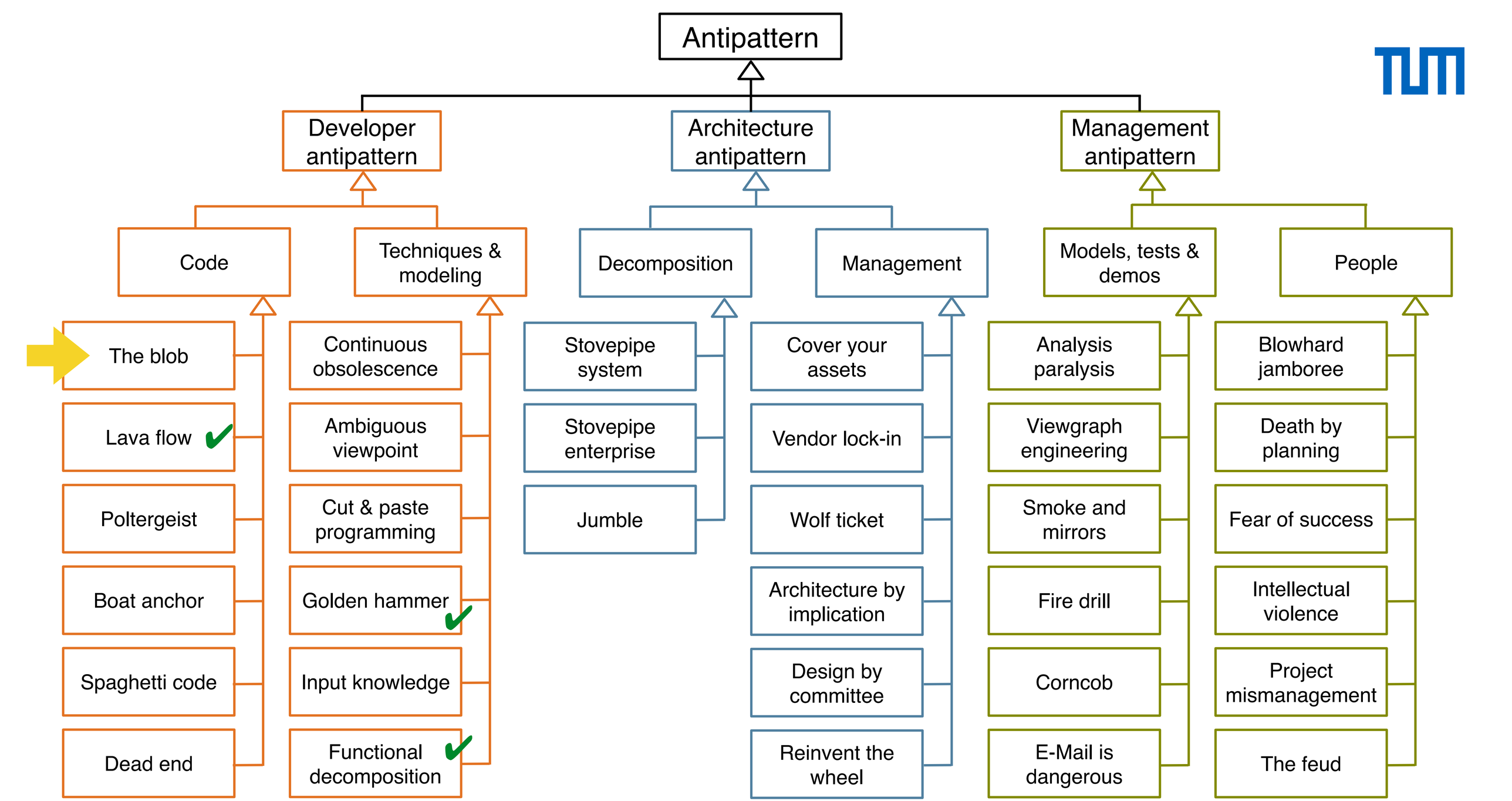
10 min

4 pts

- Problem statement:
  - Light switch application (reused from command pattern)
  - Examine the code: Identify the lava flow antipattern in code
  - Check the warnings
  - Look for unused methods
  - Refactor the code: remove the lava flow

Outline

- Antipattern definition
- Golden hammer
- Functional decomposition
- Antipattern taxonomy
- Lava flow
- **The blob**




INDESCRIBABLE...
INDESTRUCTIBLE!
NOTHING CAN STOP IT!
THE BLOB
STEVEN McQUEEN
ANETA CORSEAUT · EARL ROWE
PRODUCED BY
JACK H. HARRIS · IRVIN S. YEAWORTH, JR. · THEODORE SIMONSON AND KATE PHILLIPS
FROM AN IDEA BY IRVINE H. MILLGATE
A TONYLYN PRODUCTION · COLOR BY DE LUXE

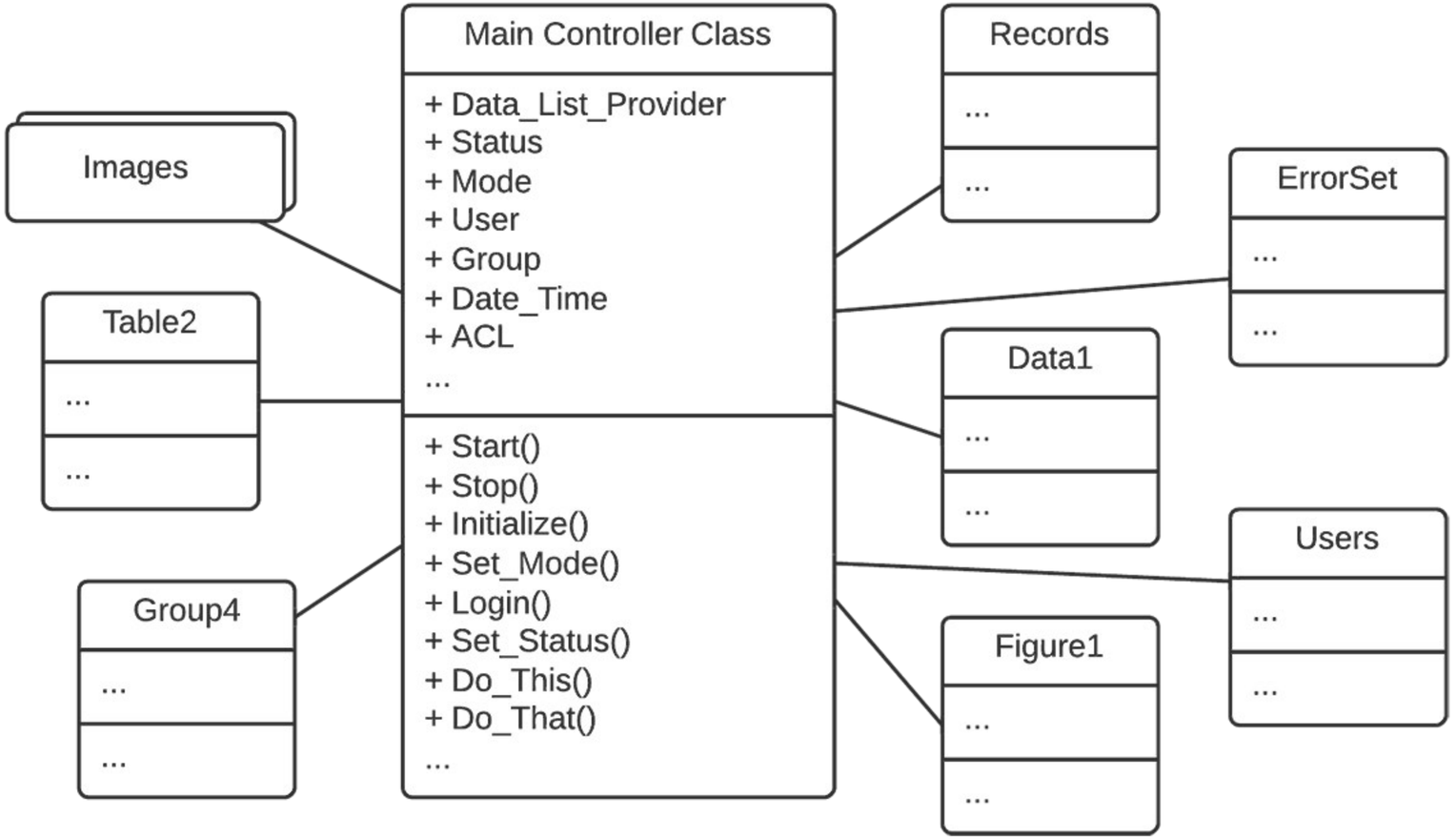
The blob

- Also known as: god class
- General form
  - Majority of responsibilities are allocated to single complex controller
  - Associated with simple data classes
- Symptoms and consequences
  - A single class with a huge number of unrelated attributes and operations encapsulated the class
  - Single controller encapsulating the functionality.
    - Example: a procedural main program
  - The blob class is typically too complex for reuse and testing

The blob

- Typical causes
  - Lack of an object oriented architecture
  - Lack of (any) architecture
  - Lack of architecture enforcement
  - Too limited intervention in iterative projects
- Known exceptions: wrapping of legacy systems
- **Refactored solution**
  - Distribution of responsibilities into smaller classes
  - Identify or categorize related attributes and operations
  - Move them into the classes they belong to (→ source code refactoring)
  - Remove redundant, indirect associations

Example


Main Controller Class
+ Data_List_Provider
+ Status
+ Mode
+ User
+ Group
+ Date_Time
+ ACL
...
+ Start()
+ Stop()
+ Initialize()
+ Set_Mode()
+ Login()
+ Set_Status()
+ Do_This()
+ Do_That()
...
Images
Table2
...
...
Group4
...
...
Records
...
...
ErrorSet
...
...
Data1
...
...
Users
...
...
Figure1
...
...

L06E04 The Blob

Start exercise

Hard

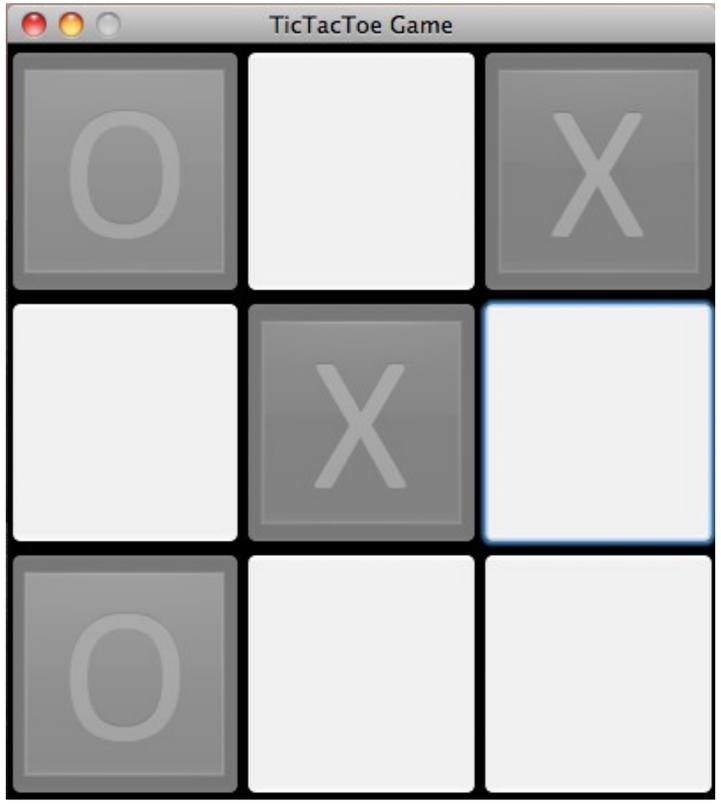
Not started yet.

Due Date: in 7 days

25 min

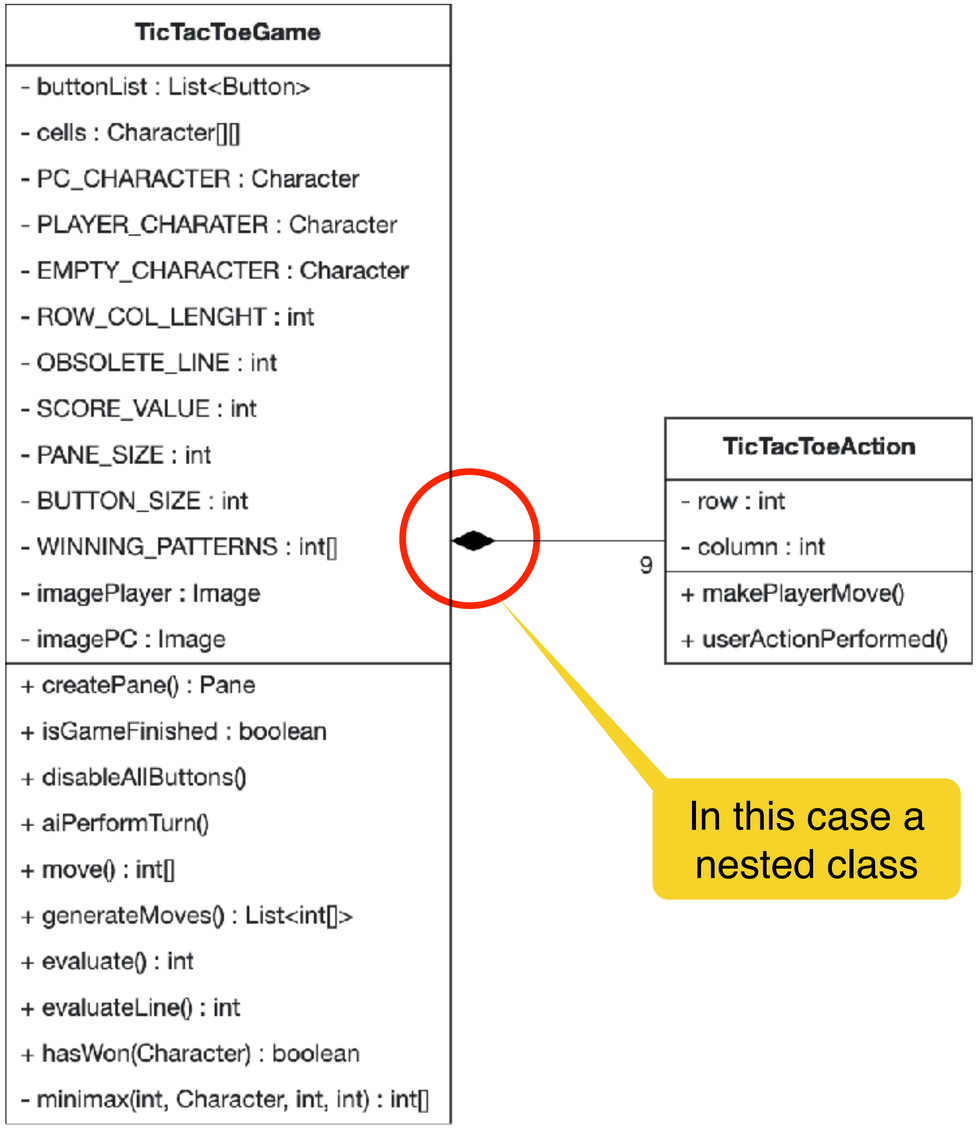
12 pts

- Problem statement: **TicTacToe Game**
  - Created by a programmer with little experience in object oriented programming
  - All code located within one class **`TicTacToeGame.java`**
  - **Blob alarm!**



Simple user interface designed for two players: human and computer player

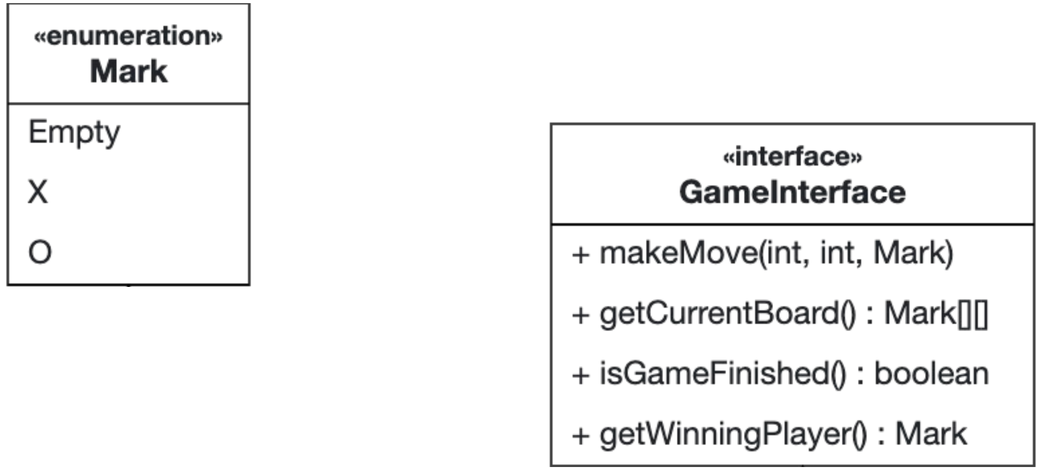
Hint: existing code structure



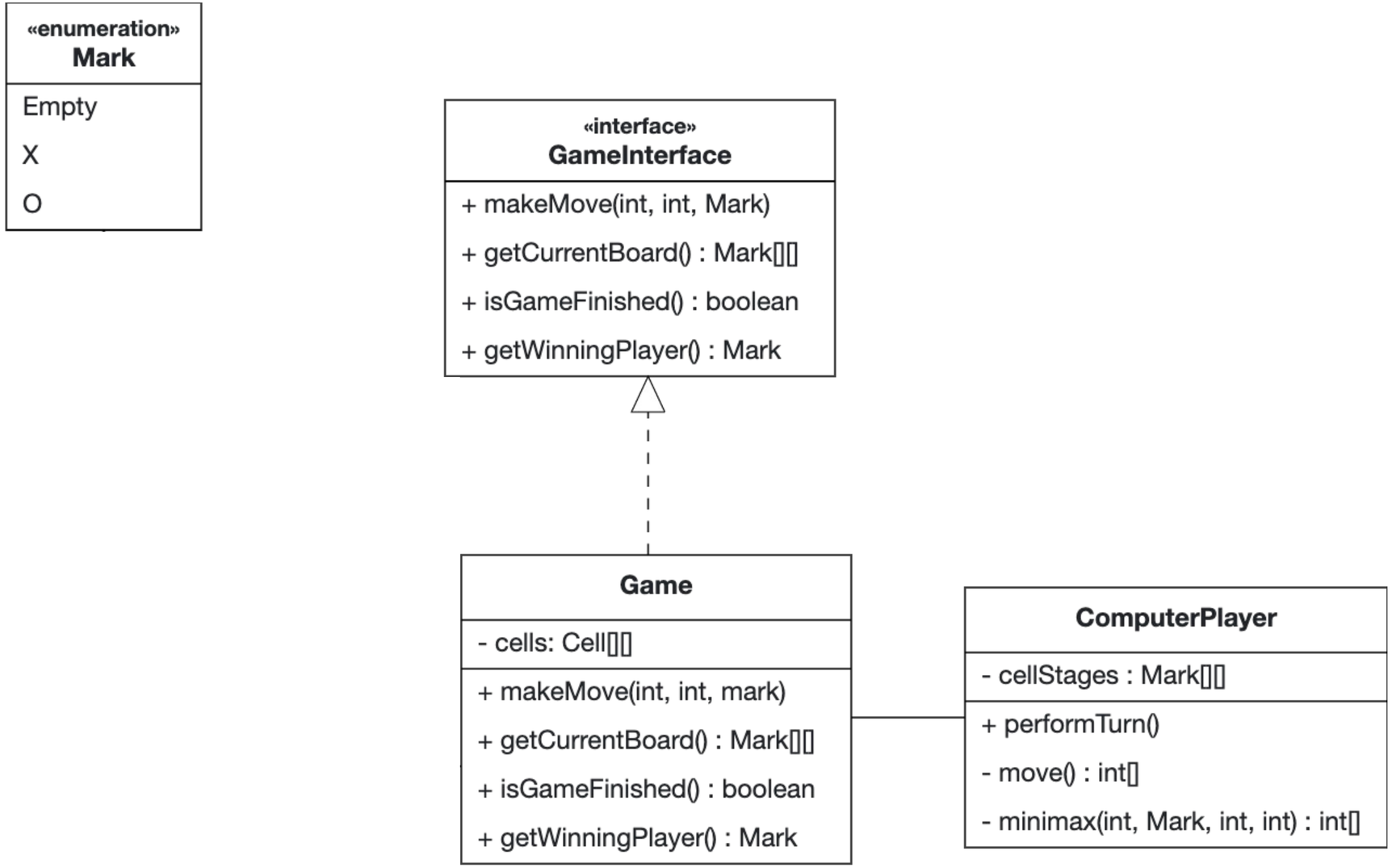
Hint: source code refactoring tasks

- Create an interface **`GameInterface`**
- Extract classes to represent the game and the computer player
- Extract a class to represent the user interface (button)
- Extract a class (e.g. `Cell`) to store the model and an enum `Mark` to store the state of a cell
- Remove unused (“dead”) code
- Perform a regression test after refactoring

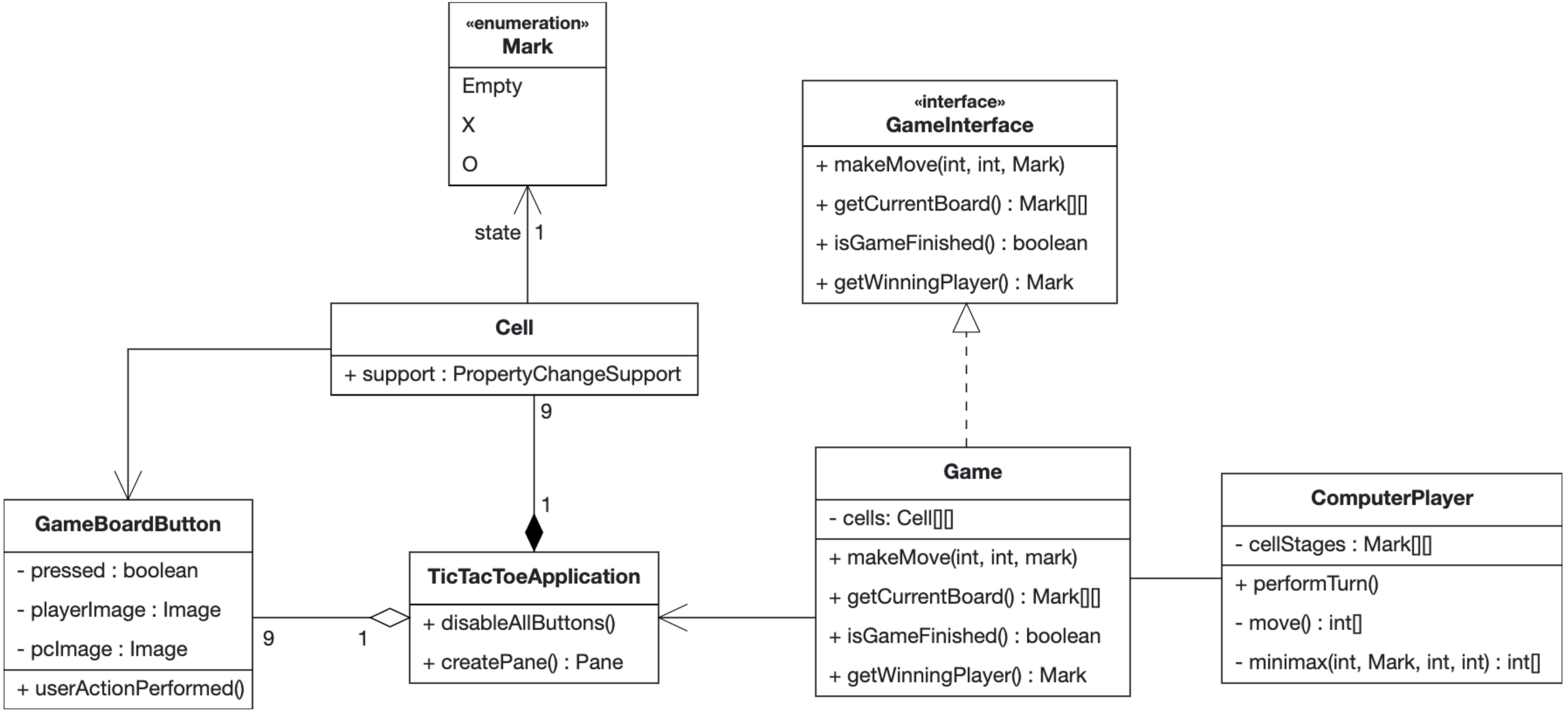
Hint: possible model refactoring



Hint: possible model refactoring



Hint: possible model refactoring



Summary

- **Antipatterns** identify and categorize common mistakes in software practice and provide **refactored solution** to improve development, architecture and management
- **Golden hammer:** identical tools are used for a wide of diverse problems
  - **Refactored solution**: explore new technologies
- **Functional decomposition:** changing and adding new functionality is difficult, because the functionality is spread all over the system
  - **Refactored solution**: object oriented programming
- **Lava flow:** lots of unused or commented out code, undocumented complex, important looking code, evolving architecture
  - **Refactored solution**: ensure that system and object design (architecture) are implemented properly
- **The blob:** a single (god) class with a huge number of unrelated attributes and operations
  - **Refactored solution**: distribution of responsibilities into smaller classes based on object oriented analysis and design

Literature

- W. Brown, R. Malevau, H.McCormick, T. Mowbray: Antipatterns: Refactoring Software, Architectures and Projects in Crisis, Wiley, 1998
- Frederick P. Brooks: The Mythical Man-Month. Essays on Software Engineering, Addison-Wesley, 1995
- Thomas Edsion Wikipedia Article - https://en.wikipedia.org/wiki/Thomas_Edison
- Source Making Antipatterns - https://sourcemaking.com/antipatterns
- W. Brown, H. McCormick, S. Thomas: AntiPatterns and Patterns in Software Configuration Management, Wiley, 1999
- A. Koenig, Patterns and Antipatterns, Journal of Object Oriented Programming, Vol 8 Nr 1, pp. 46-48, 1995