Design of Software Systems
(Ontwerp van SoftwareSystemen)

1 Introduction

Roel Wuyts
2016-2017

People

Roel Wuyts

- URL: http://roelwuyts.be/

Philippe De Ryck

- philippe.deryck@cs.kuleuven.be

Mario Henrique Cruz Torres

- mariohenrique.cruztorres@cs.kuleuven.be

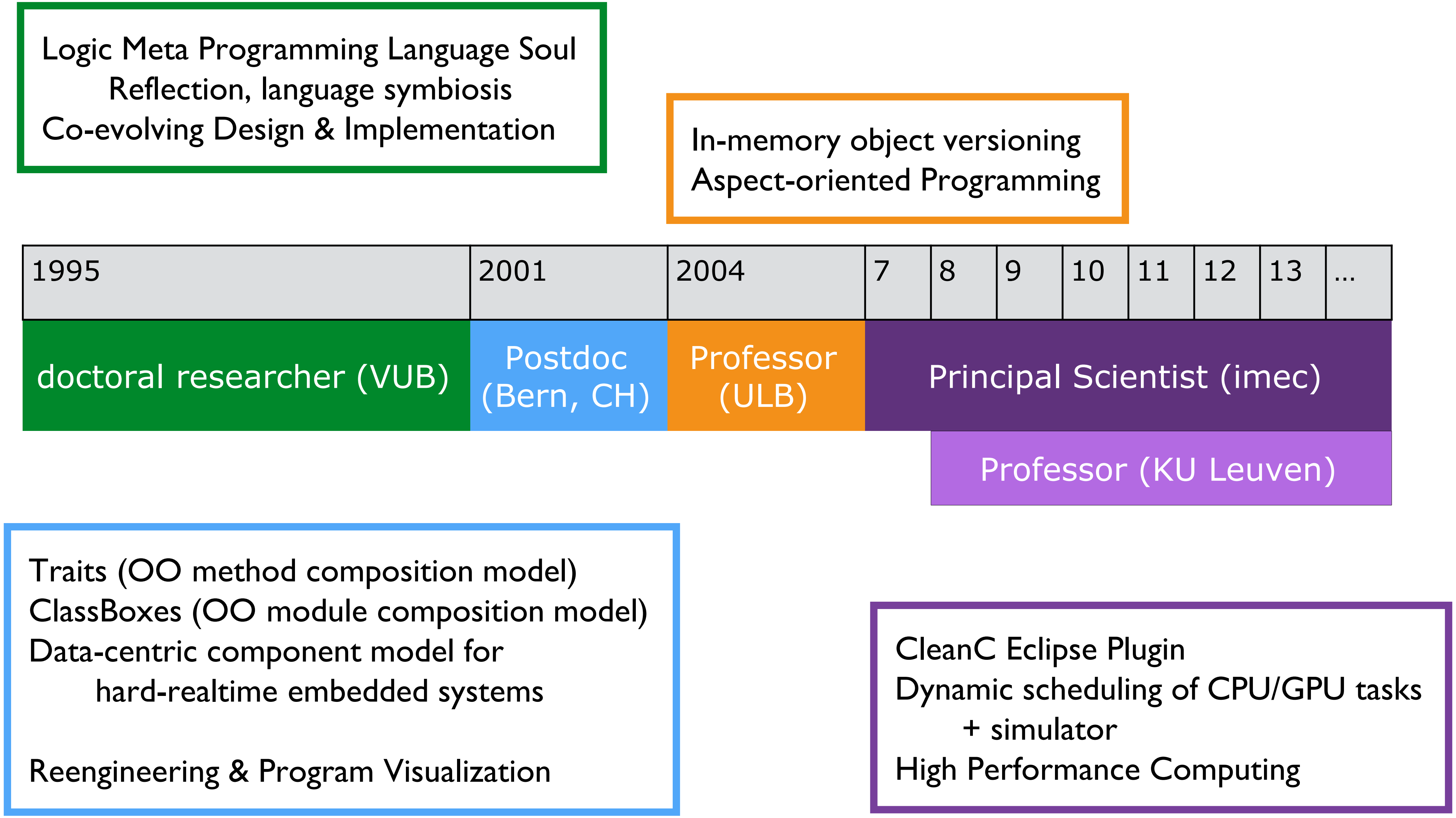
Neline Van Ginkel

- neline.vanginkel@cs.kuleuven.be

Jan Spooren

- Jan.spooren@cs.kuleuven.be

About me...



Logic Meta Programming Language Soul
Reflection, language symbiosis
Co-evolving Design & Implementation

In-memory object versioning
Aspect-oriented Programming

1995	2001	2004	7	8	9	10	11	12	13	...
doctoral researcher (VUB)	Postdoc (Bern, CH)	Professor (ULB)	Principal Scientist (imec)							
				Professor (KU Leuven)						

Traits (OO method composition model)
ClassBoxes (OO module composition model)
Data-centric component model for
hard-realtime embedded systems

Reengineering & Program Visualization

CleanC Eclipse Plugin
Dynamic scheduling of CPU/GPU tasks
+ simulator
High Performance Computing

About me...

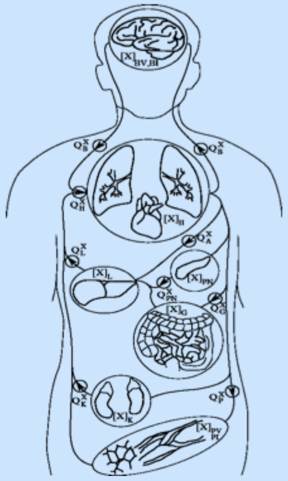
High Performance Computing for Life Sciences

ExaScience life lab

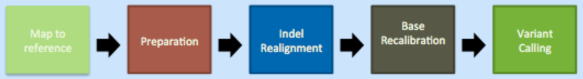
How to parallelize and distribute ?

How to deal with Big Data and Big Compute ?

How to let different stakeholder cooperate ?

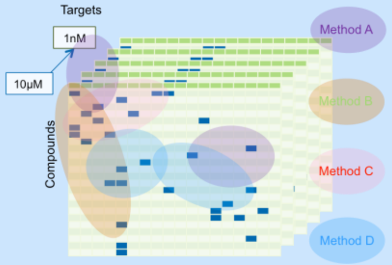


SAEM
Bayesian PK/PD



Map to reference → Preparation → Indel Realignment → Base Recalibration → Variant Calling

BWA-Cilk
elPrep
BWA-TBB-aln
BWA-TBB-mem
Scientific Workflow Languages



(initial results)

Course Goals

“This course is concerned with the design of software systems.

The focus lies on object-oriented methods.

The primary objective is learning how to take design decisions by comparing positive and negative aspects of possible design solutions with respect to analysis and requirements, design, implementation and organizational impact.

The theoretical aspects of the course are applied in a group project where a non-trivial, existing (but new to the students) application is extended with new functionality. “

Prerequisite knowledge

"Solid knowledge of object-oriented concepts and practical experience with at least one object-oriented programming language. Practical skills needed to develop software, such as the usage of an Integrated Development Environment like Eclipse or Netbeans and version control software (such as subversion)."

Main Topics

Overview of software development processes.

Object-oriented analysis and design using the UML modeling language.

Study, evaluation and usage of GRASP and design patterns.

Implementation techniques for realizing high quality object-oriented implementations.

Techniques for assessing the quality of the design and implementation of existing software systems.

Course Material

Slides and links on the website of the course

http://roelwuyts.be/OSS-1617/

Material

- Craig Larman, *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.), Prentice Hall, 2005.
- Design Patterns: Elements of Reusable Object-Oriented Software, *E. Gamma, R. Helm, R. Johnson, J. Vlissides*.
- Refactoring: Improving the Design of Existing Code, *M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts*.
- "No Silver Bullet: Essence and Accident in Software Engineering ", *F.P. Brooks*.

Project: applying the theory in practice

Group Project

- Number of persons in a group not yet known
  (depends on students following the course which is still subject to change)

Three iterations:

1. Investigate and evaluate an existing implementation
2. Analysis of an existing system
3. Extend it (Trade-offs!)
4. Decide what to modify to realize the extension
5. Refactor it
6. Clean up and maybe realize a smaller extension

Project Effort

Effort of 120 hours / student.

It is possible that you spend more or less !

- notify me in time of possible discrepancies

Escalation Policy

Groups do not always function smoothly

- But dealing with this is part of your education

In case of problems:

- discuss within group.
- if it cannot be resolved: mail to your assistant (with me in cc) to describe the problem.
- assistant may decide to involve me if necessary.

In case of problems with assistant: contact me.

In case of problems with me: contact MA1 responsible.

Course grading: First session

Score for Group Project defense (grade P)

Individual oral exam (grade I)

Grading Algorithm:

- If P <= 5 : final grade = P
- elif I <= 8: final grade = I
- else final grade = (P + I) / 2

If we find large work discrepancies within a group, specific grades for that group/person can be given

- 0 is possible when not collaborating !

Course grading: Second session

Second session

- – No possibility to redo project
- – New individual oral exam (I’)
  - Same grading algorithm as first session but:
    - – I replaced with I’

Project Defense

Each of you gets questions and answers. Then other group members can provide more information.

Questions originate from your report, design and implementation.

You get two kinds of feedback:

- during the defense:our questions and comments
- right after the defense:

🟢 ok

🟠 take care

🔴 not ok

Grades

1st session:

- Final grade ≥ 10 : done!
- Final grade < 10 : redo in second session

2nd session:

- Final grade ≥ 10 : done!
- Final grade < 10 : credit not obtained

Questions ?

About You

Let’s do an interactive “quiz”

- there is no right or wrong for most of the questions here; goal is for me to learn your reflexes when faced with questions related to programming language, design, or implementation.

Is the following correct ?

“A message sent to super is sent to the parent of the object”

What is the result of the following expression?

```
class A {
	public void m(A a) { System.out.println("1"); }
}

class B extends A {
	public void m(B b) { System.out.println("2"); }
	public void m(A a) { System.out.println("3"); }
}


B b = new B();
A a = b;
a.m(b);
```

What do you think of the following implementation?

```
// Return null to signify end of file
  protected IToken fetchToken() throws EndOfFileException {
      ++count;
      while (bufferStackPos >= 0) {
          // Tokens don't span buffers, stick to our current one
          char[] buffer = bufferStack[bufferStackPos];
          int limit = bufferLimit[bufferStackPos];
          int pos = bufferPos[bufferStackPos];

          switch (buffer[pos]) {

          case '_':
              t = scanIdentifier();
              if (t instanceof MacroExpansionToken)
                  continue;
              return t;

          case '#':
              if (pos + 1 < limit && buffer[pos + 1] == '#') {
                  ++bufferPos[bufferStackPos];
                  return newToken(IToken.tPOUNDPOUND);
              }

              // Should really check to make sure this is the first
              // non whitespace character on the line
              handlePPDirective(pos);
              continue;
```

…

(390 lines of code in total)

Reuse versus hack

Suppose you are responsible to add a new feature to an existing piece of software. The design of the existing software makes this hard. How do you decide whether to rewrite the existing software or whether to “hack in” the new feature ?

Understanding Existing Systems

Your boss wants you to quickly develop a new tool. You decide to start from a large existing open-source application you found on SourceForge.

How do you start ?

Object-Oriented Software Design Question

A restaurant menu consists of dishes, e.g. “Flemish stew”, “Black pudding with apples” and “Chicken Royale with Champagne”. Each dish consists of a number of ingredients and is either a starter, a main course or a dessert. The menu shows for each dish an authenticity score (1, 2 or 3), a calory score, as well as the price. Menus need to be printed in a variety of languages (dutch, french, english, japanese, arabic; some left-to-right and some right-to-left) and needs to be available on an interactive website (where a picture is shown of the dish). The menus change frequently with the seasons.

Why Software Engineering?

Problem Specification → Final Program

But ...

- Where did the specification come from?
- How do you know the specification corresponds to the user’s needs?
- How did you decide how to structure your program?
- How do you know the program actually meets the specification?
- How do you know your program will always work correctly?
- What do you do if the users’ needs change?
- How do you divide tasks up if you have more than a one-person team?

What is Software Engineering? (I)

Some Definitions and Issues

- “state of the art of developing quality software on time and within budget”

Trade-off between perfection and physical constraints

- Software engineering deals with real-world issues

State of the art!

- Community decides on “best practice” + life-long education

What is Software Engineering? (II)

“multi-person construction of multi-version software”

– Parnas

Team-work

- Scale issue (“program well” is not enough) + Communication Issue

Successful software systems must evolve or perish

- Change is the norm, not the exception

Communication and Modeling

Team-effort requires communication

Results have to be communicated externally

UML

Unified Modeling Language

De-facto standard that I expect everybody to know and follow

- working knowledge of at least the use case, class, sequence and communication diagrams
- use throughout course (theory, practice, project)

Self-study

- I give a short overview
- You study

General Goals of UML

Model systems using OO concepts

Establish an explicit coupling to conceptual as well as executable artifacts

To create a modeling language usable by both humans and machines

Models different types of systems (information systems, technical systems, embedded systems, real-time systems, distributed systems, system software, business systems, UML itself, ...)

11 diagrams in UML 2

Class diagram	Structural
Internal Structure Diagram	
Collaboration diagram	
Component diagram	
Use case diagram	
State machine diagram	Dynamic
Activity Diagram	
Sequence diagram	
Communication Diagram	
Deployment diagram	Physical
Package diagram	Model Management

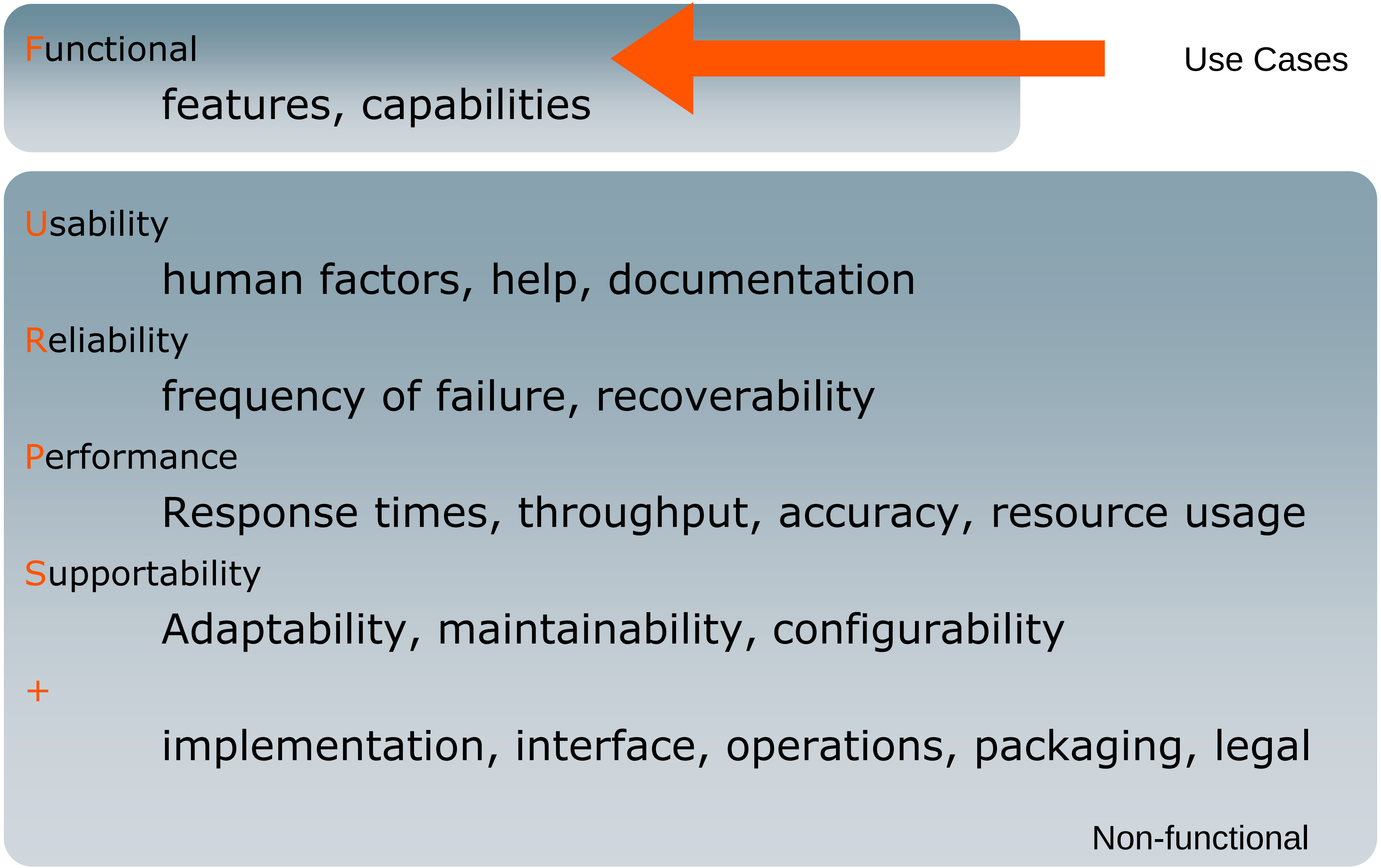
Requirements Engineering and Use Cases

Requirements: documented need for what a system or project should do

- 37% of problems with software projects have to do with requirements
- 25% of the requirements change during the project (and 35-50% in large projects)

Therefore: embrace change!

Types of Requirements: FURPS+ categorization



Functional

features, capabilities

Use Cases

Usability

human factors, help, documentation

Reliability

frequency of failure, recoverability

Performance

Response times, throughput, accuracy, resource usage

Supportability

Adaptability, maintainability, configurability

+

implementation, interface, operations, packaging, legal

Non-functional

Use Cases

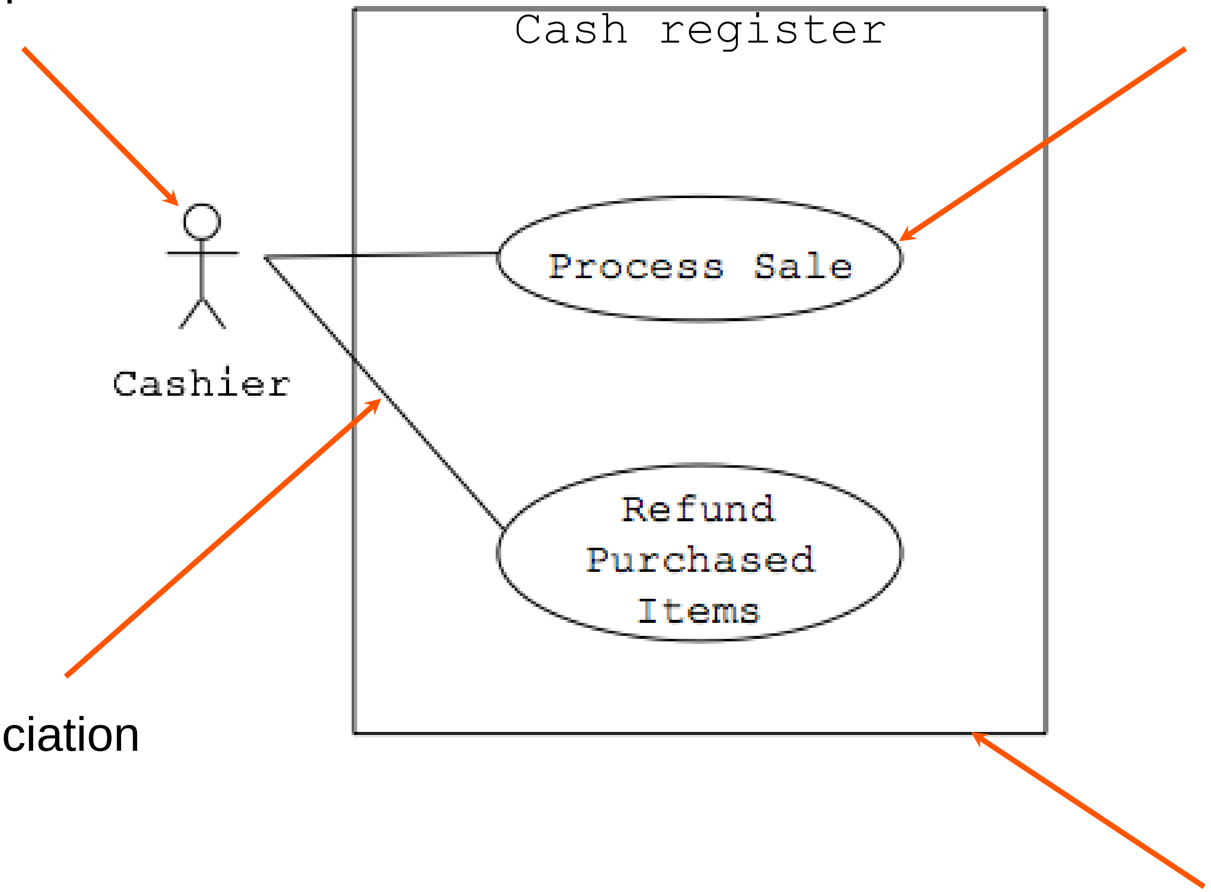
Stories that describe usage of the system

- describe sequence of actions with an observable result for a specific actor
- used by all kinds of stakeholders

It does not describe the internal working of the system

- What, not How
- Responsibilities of the system are described

Use Case Diagram



actor

use case

association

system border

Fully Dressed Use Case Description

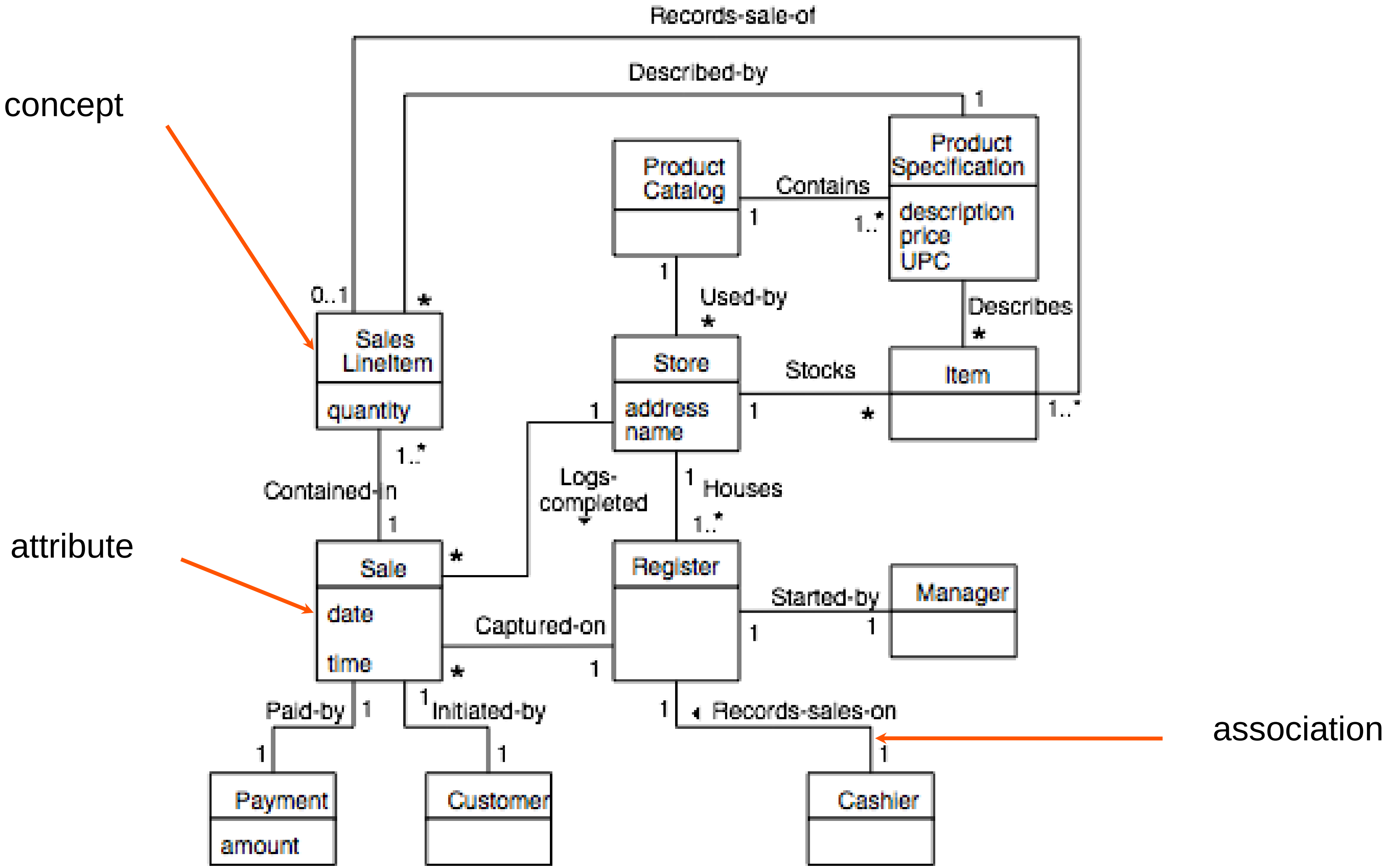
Use case:	**Process Sale**
Primary Actor:	Cashier
Stakeholders and interests:	• Cashier: wants accurate, fast entry, and no payment errors, as cash drawers shortages are deduced from his/her salary • Customer: wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns. • Manager, Government, Payment Company, ...
Precondition:	Cashier is identified and authenticated
Success Guarantee (postcondition)	Sale is saved. Tax is correctly calculated. Receipt is generated. Accounting and inventory are updated. Payment info is recorded.

Domain Modeling

A domain model describes meaningful concepts in the problem domain

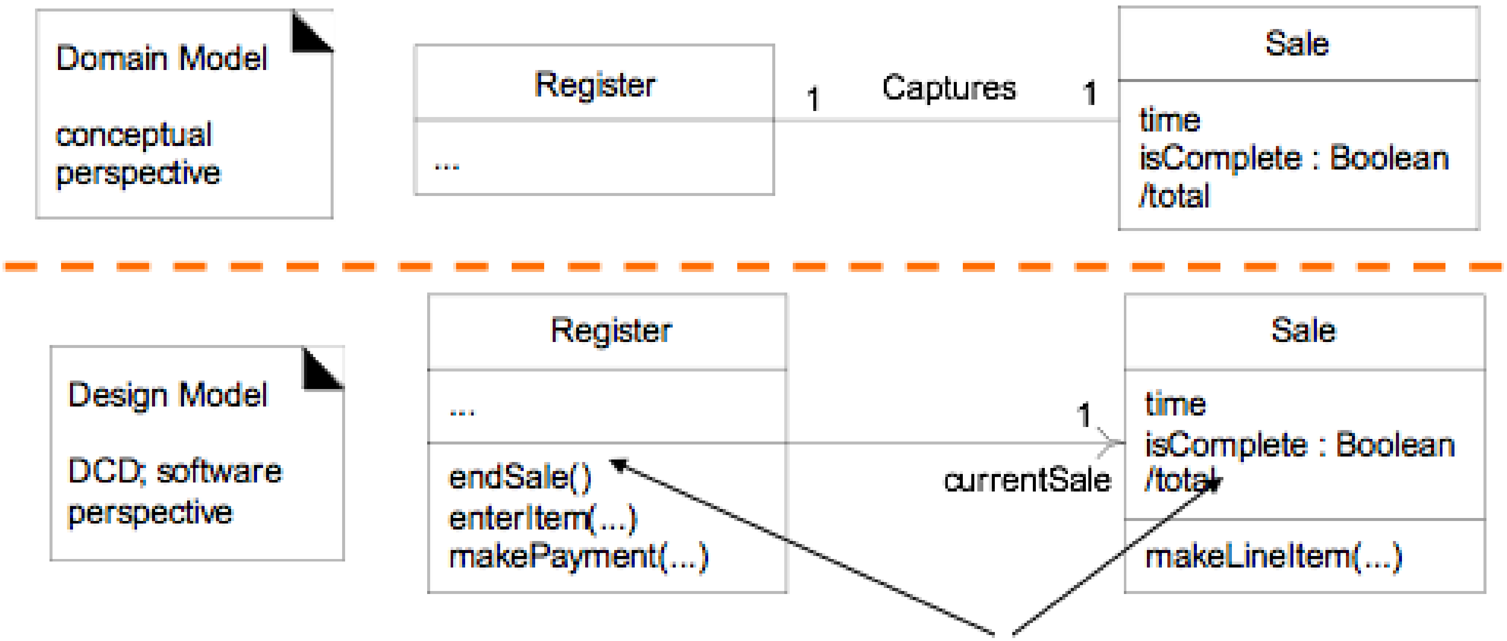
- again about the what, not the how
- does not model design artifacts (how), but models conceptual artifacts, real-world things

Domain model for the Cash Register example



Class Diagrams

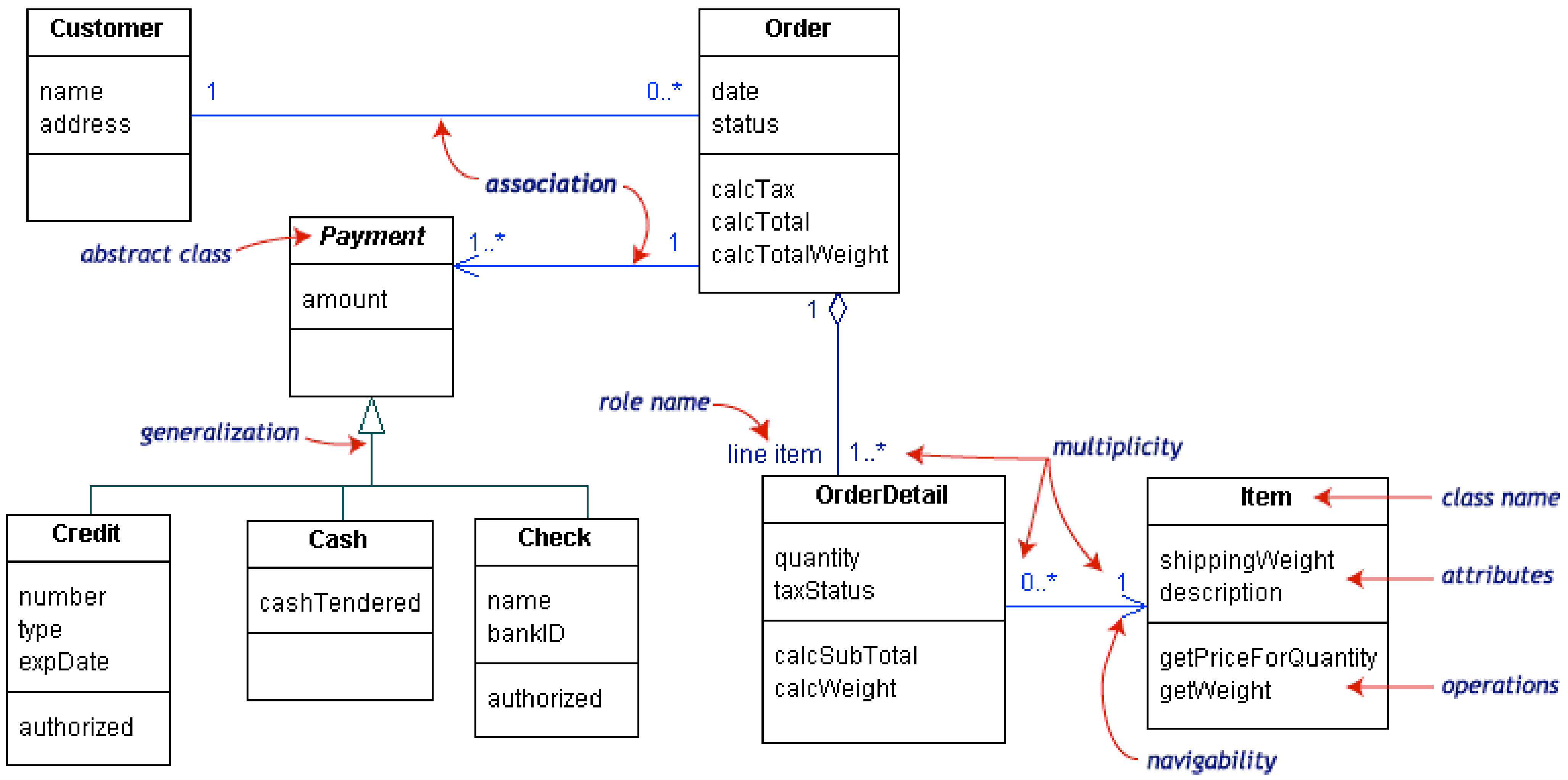
Domain model is the analysis *class diagram*
Don't show methods



Avoid showing no-argument constructors & getters/setters

Design model *class diagrams* shows methods and visibility (arrowhead on association)
Register has reference to Sale; Sale does not have reference to Register

Class Diagrams



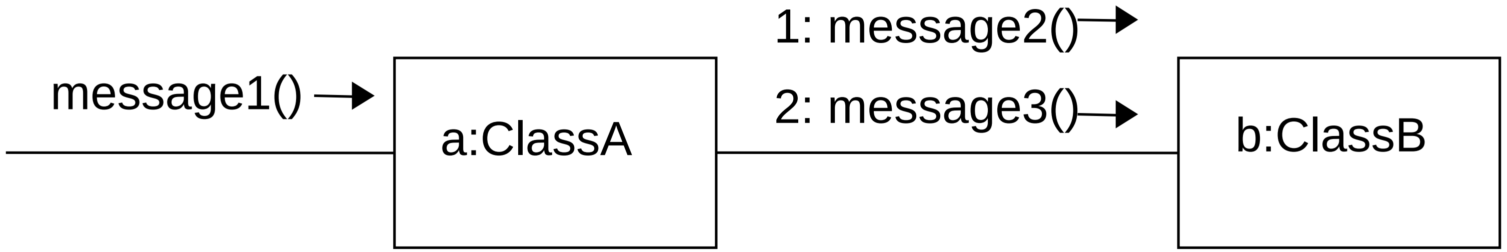
Interaction diagrams

UML Interaction diagrams

- model message-exchange between objects

2 kinds:

- Communication Diagrams – focus on interactions
- Sequence Diagrams – focus on time



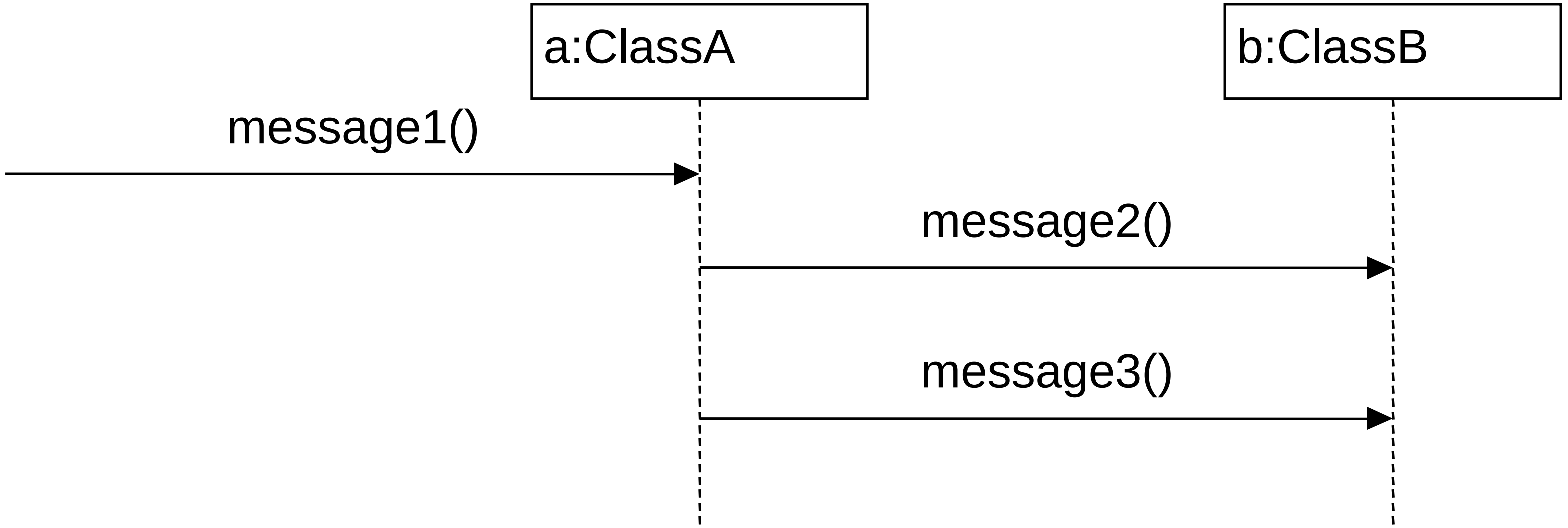
Interaction diagramma’s

UML Interaction diagrams

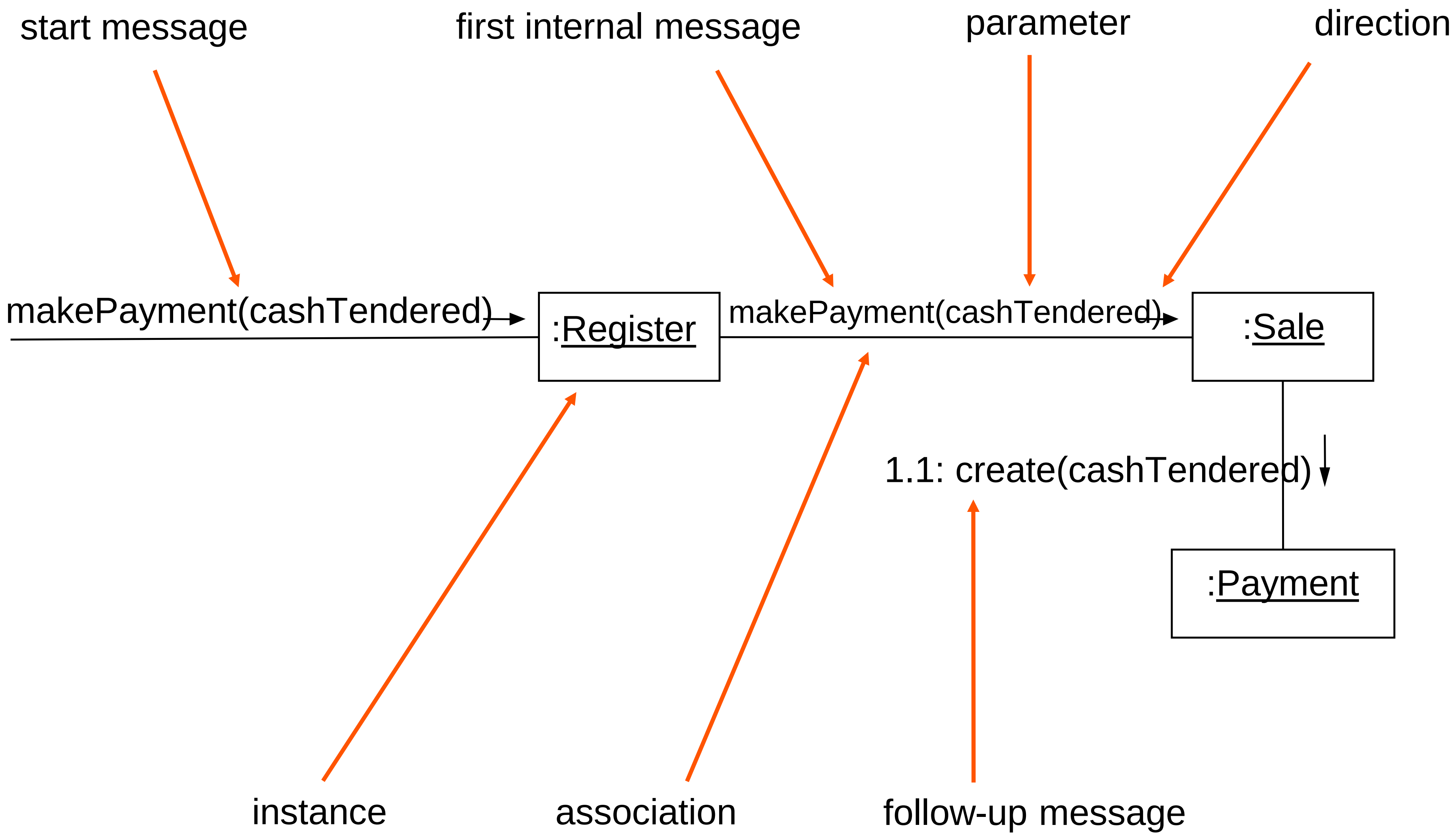
- model message-exchange between objects

2 kinds:

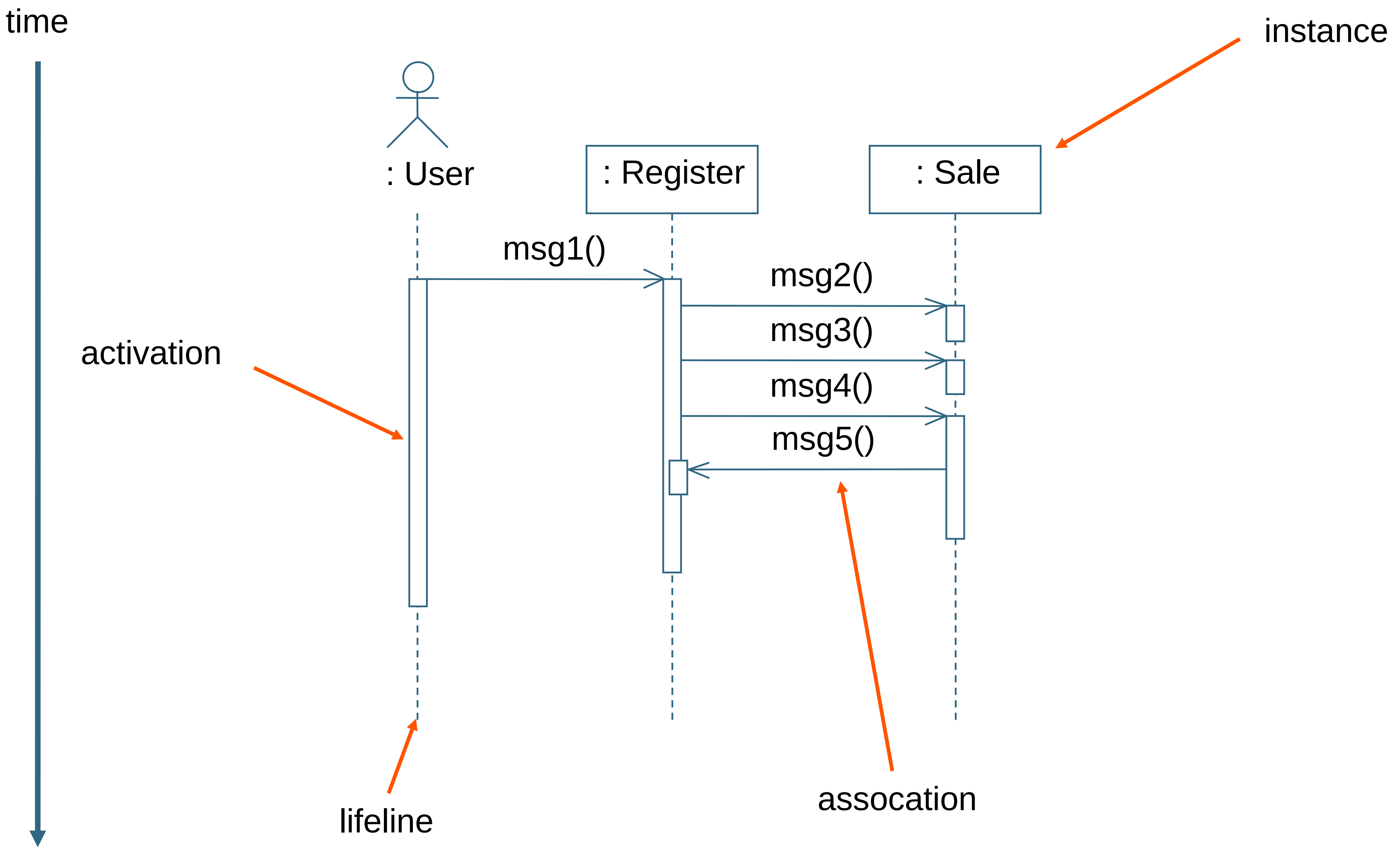
- Communication Diagrams – focus on interactions
- Sequence Diagrams – focus on time



Communication Diagram Example



Sequence Diagram Example



Conclusion

This course is about (OO) software design

- Understand quality design and implementation
- Make reasoned design decisions
- Make trade-offs that balance quality, effort, design, and implementation
- Be able to communicate your decision

http://roelwuyts.be/OSS-1617/

Design of Software Systems
(Ontwerp van SoftwareSystemen)

2 Basic OO Design

Roel Wuyts
2016-2017

Acknowledgments

Tom Holvoet (KUL, Leuven, BE) for the GRASP pattern slides

Oscar Nierstrasz (University of Bern, CH) for the Tic Tac Toe example

The whole course in one slide ?

Basic OO Design Principles:

- **Minimize Coupling**
- **Increase Cohesion**
- **Distribute Responsibilities**

Basic OO Design Principles

No matter whether you use forward engineering or re-engineering,
These basic OO Design Principles hold:

- Minimize Coupling
- Increase Cohesion
- Distribute Responsibilities

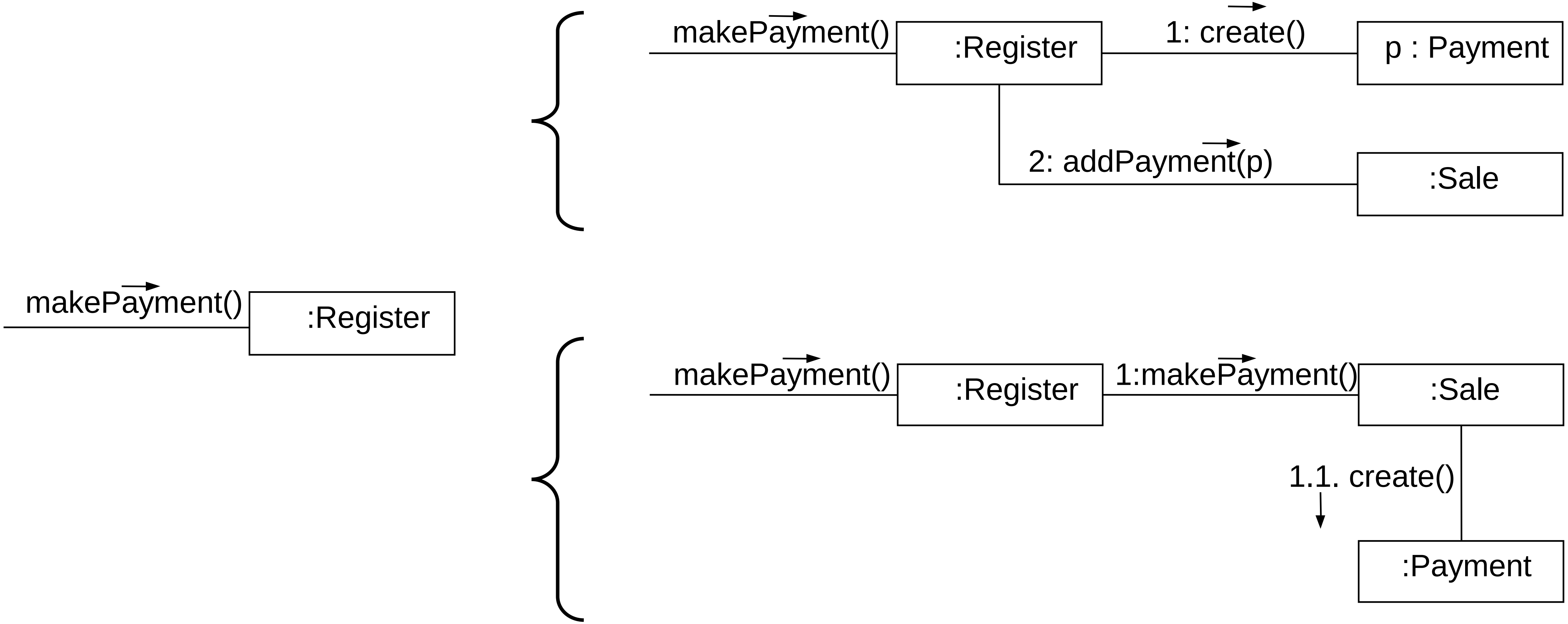
You should always <u>strife</u> to use and <u>balance</u> these principles

- they are fairly language- and technology independent
- processes, methodologies, patterns, idioms, ... all try to help to apply these principles in practice
  - It's still your job to determine the best balance

4. Low Coupling Pattern

Pattern	Low Coupling
Problem	How to stimulate low independance, reduce impact of change and increase reuse?
Solution	Assign responsibilities such that your design exhibits low coupling. Use this principle to evaluate and compare alternatives.

Low Coupling Pattern

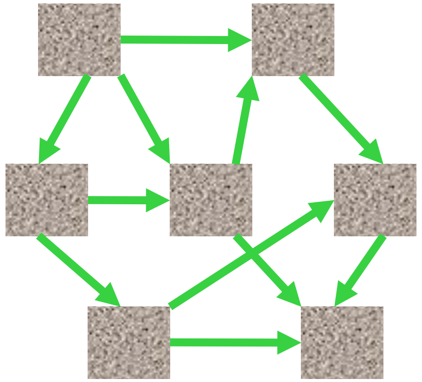


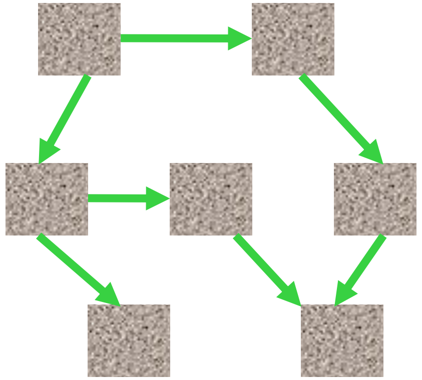
- Which design is better?
- Coupling to stable libraries/classes?
- Key principle for evaluating choices

Low Coupling Pattern

Coupling is a measure that shows how much a class is dependent on other classes

- X depends on Y (~ X does not compile without Y):
  - X has attribute of type Y
  - X uses a service of Y
  - X has method referencing Y (param, local variable)
  - X inherits from Y (direct or indirect)
  - X implements interface Y
- “evaluative” pattern:
  - use it to evaluate alternatives
  - try to reduce coupling





Low Coupling Pattern

Advantages of low coupling:

- reduce impact of changes (isolation)
- increase understandibility (more self-contained)
- enhance reuse (independance)

Is not an absolute criterium

- Coupling is always there
- Therefore you will need to make trade-offs !

Inheritance is strong coupling !!

Low Coupling Pattern: remarks

Aim for low coupling with all design decisions

Cannot be decoupled from other patterns

Learn to draw the line (experience)

- do not pursue low coupling in the extreme
  - Bloated and complex active objects doing all the work
  - lots of passive objects that act as simple data repositories
- OO Systems are built from connected collaborating objects

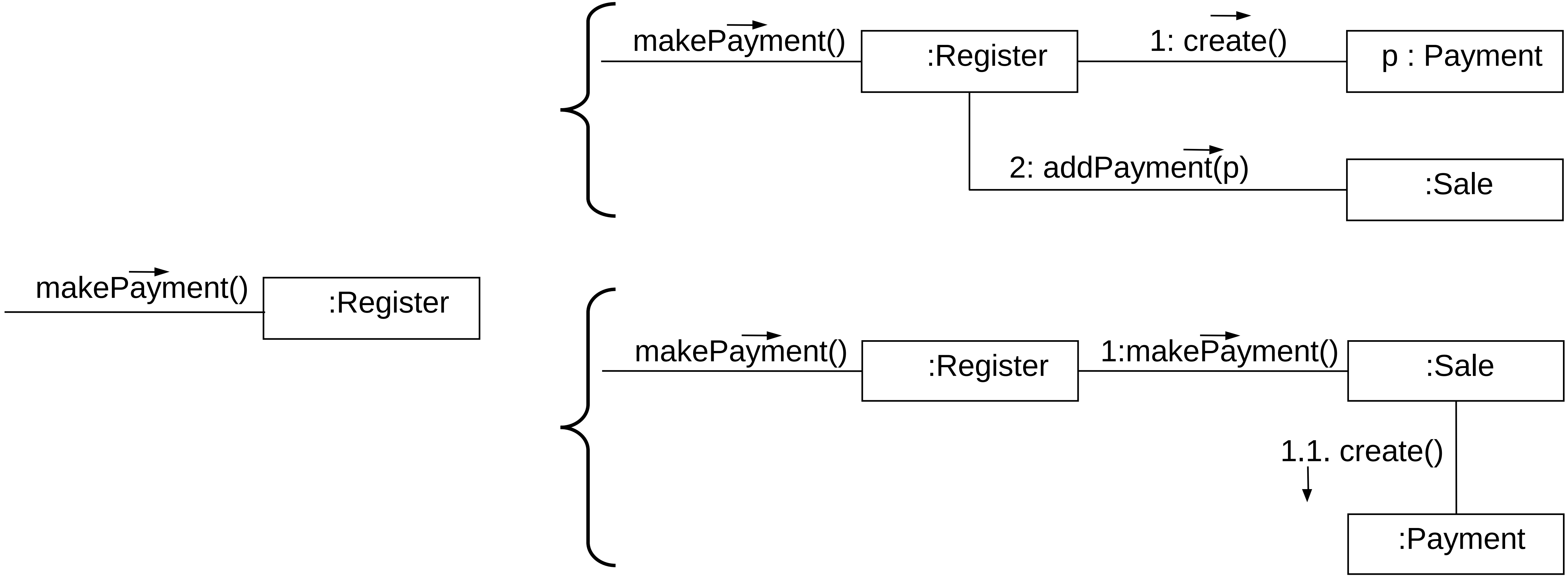
Coupling with standardized libraries is NOT a problem

Coupling with unstable elements IS a problem

5. High Cohesion Pattern

Pattern	High Cohesion
Problem	How to retain focus, understandability and control of objects, while obtaining low coupling?
Solution	Assign responsibilities such that the cohesion of an object remains high. Use this principle to evaluate and compare alternatives.

High Cohesion Pattern



- Cohesion: Object should have strongly related operations or responsibilities
- Reduce fragmentation of responsibilities (complete set of responsibility)
- To be considered in context => register cannot be responsible for all register-related tasks

High Cohesion Pattern

Cohesion is a measure that shows how strong responsibilities of a class are coupled.

Is an “evaluative” pattern:

- use it to evaluate alternatives
- aim for maximum cohesion
  - (well-bounded behavior)

Cohesion ↘

- number of methods ↗ (bloated classes)
- understandability ↘
- reuse ↘
- maintainability ↘

High Cohesion Pattern: remarks

Aim for high cohesion in each design decision

degree of collaboration

- Very low cohesion: a class has different responsibilities in widely varying functional domains
  - class RDB-RPC-Interface: handles Remote Procedure Calls as well as access to relational databases
- Low cohesion: a class has exclusive responsibility for a complex task in one functional domain.
  - class RDBInterface: completely responsible for accessing relational databases
  - methods are coupled, but lots and very complex methods

...

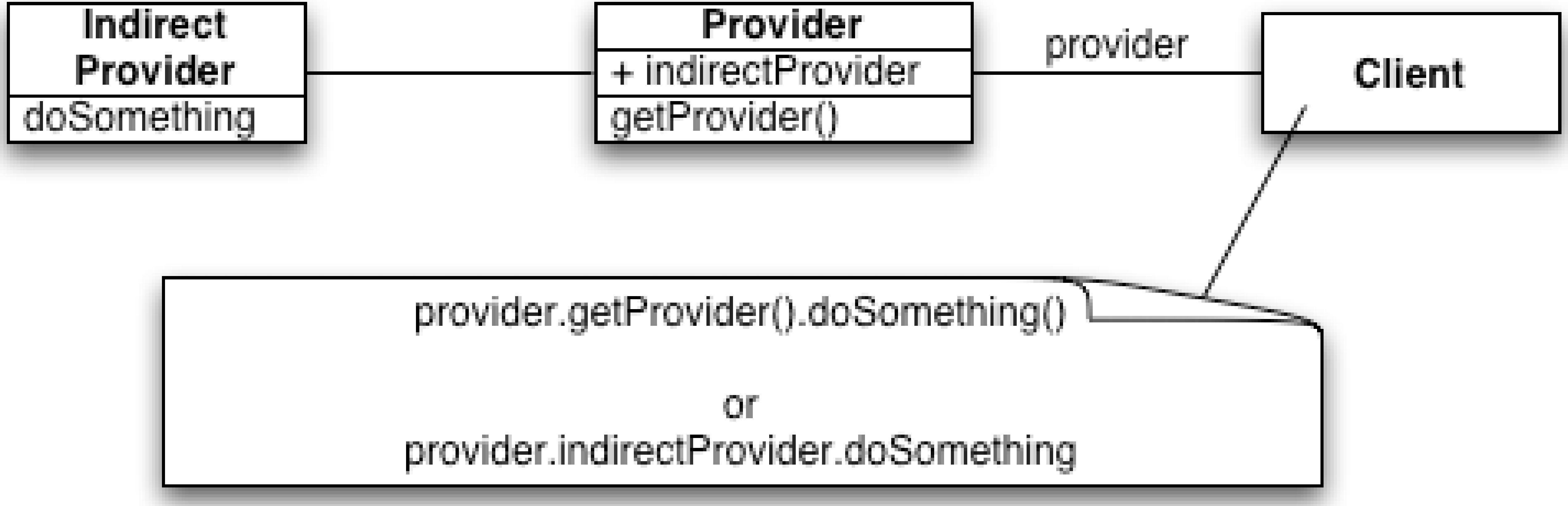
High Cohesion Pattern: remarks

Aim for high cohesion in each design decision

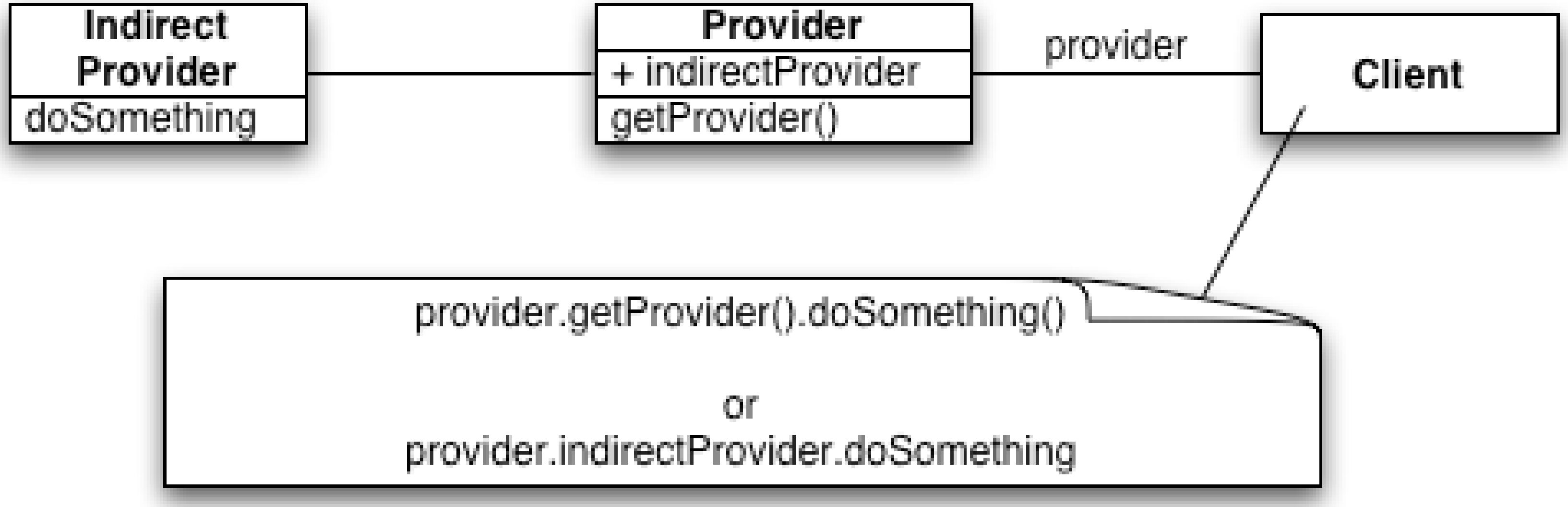
degree of collaboration (ctd)

- Average cohesion: a class has exclusive ‘lightweight’ responsibilities from several functional domains. The domains are logically connected to the class concept, but not which each other
  - a class Company that is responsible to manage employees of a company as well as the financials
  - occurs often in ‘global system’ classes !!
- High cohesion: a class has limited responsibilities in one functional domain, collaborating with other classes to fulfill tasks.
  - Class RDBInterface: partially responsible for interacting with relational databases

Example 1


Indirect Provider
doSomething
Provider
+ indirectProvider
getProvider()
provider
Client
provider.getProvider().doSomething()
or
provider.indirectProvider.doSomething

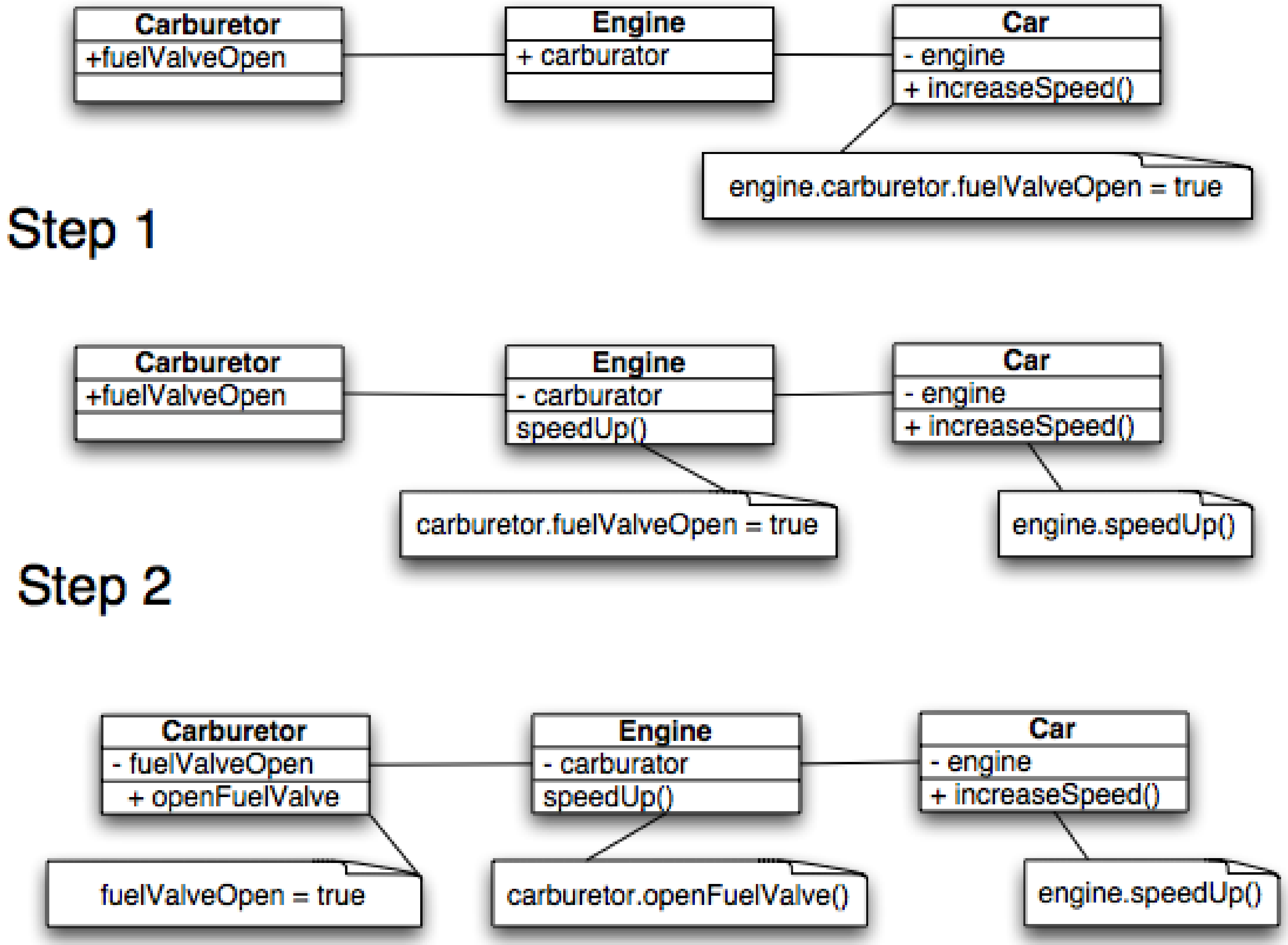
Why is this bad ?



Client knows how Provider is implemented

- has to know that it uses an IndirectProvider
  - uses the interface of Provider as well as of IndirectProvider
- Client and IndirectProvider are strongly coupled !
  - Client has to use them together
  - Changing either Provider or IndirectProvider impacts Client

Reducing the Coupling



Reducing Coupling impacts the design

The interfaces of the classes become more clear

- a method ‘speedUp()’ makes perfect sense: cohesion

Allows for more opportunity for reuse

- A subclass of Engine, “ElectricalEngine”, might not need a Carburetor at all
  - This is transparent for Car

“Law of Demeter” / Don’t talk to strangers

Each unit should only talk to its friends;
don't talk to strangers

or, more mechanically:

You are only allowed to send messages to:

- yourself (self/this, super)
- an argument passed to you
- an object you create

Lieberherr, Karl. J. and Holland, I., Assuring good style for object-oriented programs, IEEE Software, September 1989, pp 38-48

Example 2

procedural approach:
passing an int and using switch to decide which behavior to execute based on that int

```
void CVideoAppUi::HandleCommandL(TInt aCommand)
    {
    switch ( aCommand )
        {
              case EAknSoftkeyExit:
              case EAknSoftkeyBack:
              case EEikCmdExit:
                      { Exit();   break; }

              // Play command is selected
              case EVideoCmdAppPlay:
                     { DoPlayL(); break; }

               // Stop command is selected
              case EVideoCmdAppStop:
                     { DoStopL(); break; }

              // Pause command is selected
              case EVideoCmdAppPause:
                     { DoPauseL(); break; }

              // DocPlay command is selected
              case EVideoCmdAppDocPlay:
                     { DoDocPlayL(); break; }
              ......
```

Nokia S60 mobile video player 3gpp source code
http://www.codeforge.com/article/192637

Why is this bad ?

Case (switch) statements in OO code are a sign of a bad design

- lack of polymorphism: procedural way to implement a choice between alternatives
- hardcodes choices in switches, typically scattered in several places
  - when the system evolves these places have to be updated, but are easy to miss

See also: Replace Conditional with Polymorphism (http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism)

Solution: Replace case by Polymorphism

OO approach:
passing an object and using polymorphism to select behavior to execute

```
void CVideoAppUi::HandleCommandL(Command aCommand)
   {
               aCommand.execute();
              }
```

Create a Command class hierarchy, consisting of a (probably) abstract class AbstractCommand, and subclasses for every command supported. Implement execute on each of these classes:

- virtual void AbstractCommand::execute() = 0;
- virtual void PlayCommand::execute() { ... do play command ...};
- virtual void StopCommand::execute() { ... do stop command ...};
- virtual void PauseCommand::execute() { ... do pause command ...};
- virtual void DocPlayCommand::execute() { ... do docplay command ...};
- virtual void FileInfoCommand::execute() { ... do file info command ...};

Added advantage

These case statements occur wherever the command integer is used in the original implementation

- you will quickly assemble a whole set of useful methods for these commands
- Moreover, commands are then full-featured classes so they can share code, be extended easily without impacting the client, ...
- They can also be used when adding more advanced functionalities such as undo etc.

Have you noticed that the methods are shorter ?

Open question: can you think of disadvantages ?

Stepping Back

Showed concrete examples (and solutions) of breaches of basic OO design principles visible in code

- Fixing them improved the design!

Question: how can we avoid this ?

- be cautious ;-)
- get help by applying:
  - Design principles and methodologies
    - eg.: Responsibility Driven Design
  - GRASP patterns, Design Patterns
  - Idioms and Programming Practices
- use your head!

Responsibility Driven Design

http://www.wirfs-brock.com/PDFs/Responsibility-Driven.pdf

Metaphor – can compare to people

- Objects have responsibilities
- Objects collaborate

In RDD we ask questions like

- What are the responsibilities of this object ?
- Which roles does the object play ?
- Who does it collaborate with ?

Domain model

- classes do NOT have responsibilities!
- they merely represent concepts + relations
- design is about realizing the software ➔ someone has to do the work … who ??

Understanding Responsibilities

is key to good OO Design

RDD Process

Design = incremental journey of discovery and refinement

- build knowledge to take proper decisions
- start by looking for classes of key objects
    - can use the domain model for inspiration !
- then think about what actions must be accomplished, and who will accomplish them - how to accomplish them is for later !
    - Leads to responsibilities

Responsibilities

Two types of responsibilities

- Doing
  - Doing something itself (e.g. creating an object, doing a calculation)
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects
- Knowing
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things it can derive or calculate

Object Collaboration

Objects collaborate: one object will request something from another object

To find collaborations answer the following questions:

- What other objects need this result or knowledge?
- Is this object capable of fulfilling this responsibility itself?
- If not, from what other objects can or should it acquire what it needs?

Cfr: Coupling and Cohesion !

Example: Tic Tac Toe

Requirements:

> “A simple game in which one player marks down only crosses and another only ciphers [zeroes], each alternating in filling in marks in any of the nine compartments of a figure formed by two vertical lines crossed by two horizontal lines, the winner being the first to fill in three of his marks in any row or diagonal.”
>
> — Random House Dictionary

We should design a program that implements the rules of Tic Tac Toe.

Setting Scope

Questions:

- Should we support other games?
- Should there be a graphical UI?
- Should games run on a network? Through a browser?
- Can games be saved and restored?

A monolithic paper design is bound to be wrong!

Setting Scope

A monolithic paper design is bound to be wrong?

Let’s follow an *iterative development* strategy:

- limit initial scope to the minimal requirements that are interesting
- grow the system by adding features and test cases
- let the design emerge by refactoring roles and responsibilities

How much functionality should you deliver in the first version of a system?

- Select the minimal requirements that provide value to the client.

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Tic Tac Toe Objects

Some objects can be identified from the requirements:

Objects	***Responsibilities***
Game	Maintain game rules
Player	Make moves Mediate user interaction
Compartment	Record marks
Figure (State)	Maintain game state

Entities with clear responsibilities are more likely to end up as objects in our design.

Tic Tac Toe Objects ...

Others can be eliminated:

Non-Objects	***Justification***
Crosses, ciphers	Same as Marks
Marks	Value of Compartment
Vertical lines	Display of State
Horizontal lines	ditto
Winner	State of Player
Row	View of State
Diagonal	ditto

✎How can you tell when you have the “right” set of objects?

✓*Each object has a clear and natural set of responsibilities.*

Missing Objects

Now we check if there are unassigned responsibilities:

- Who starts the Game?
- Who is responsible for displaying the Game state?
- How do Players know when the Game is over?

Let us introduce a Driver that supervises the Game.

? How can you tell if there are objects missing in your design?

- When there are responsibilities left unassigned.

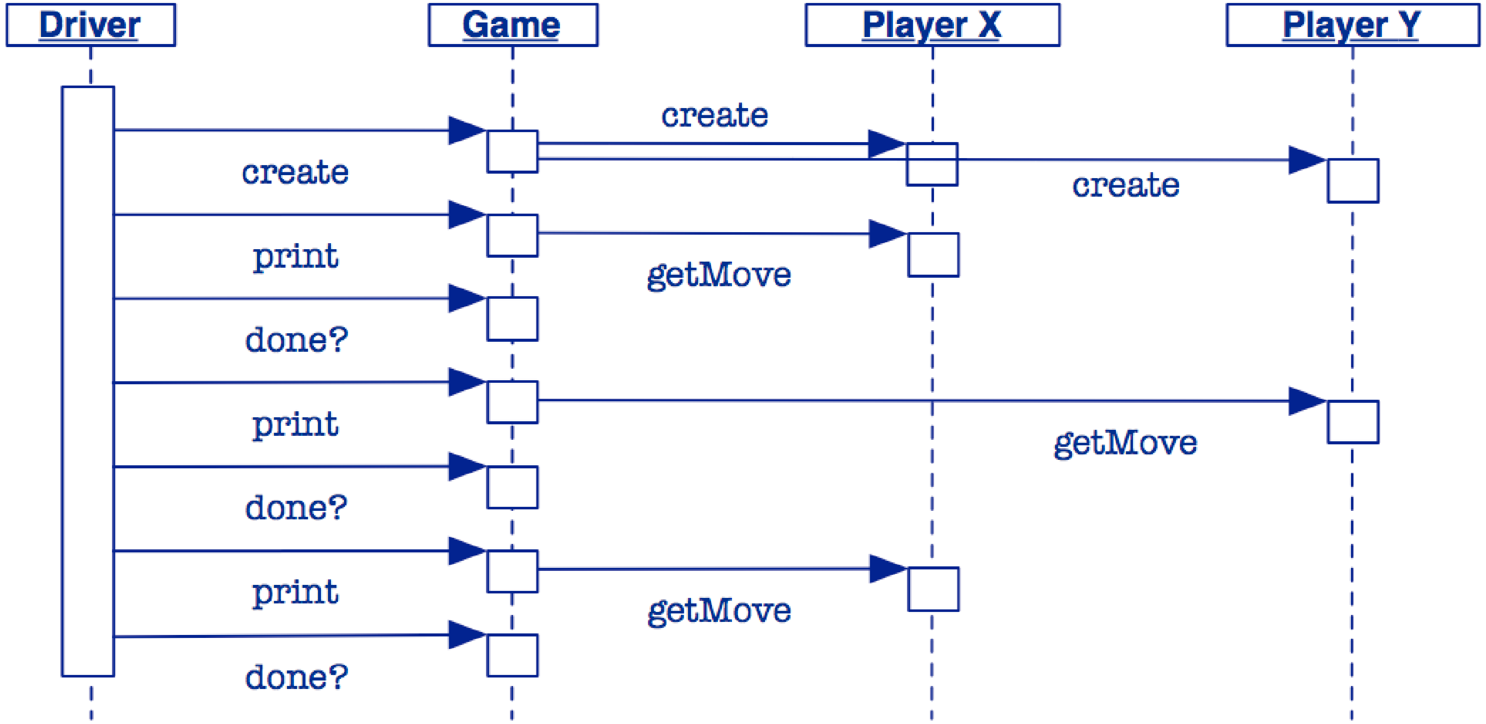
Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Scenarios

A scenario describes a typical sequence of interactions:



Are there other equally valid scenarios for this problem?

Version 0 — skeleton

Our first version does very little!

```
class GameDriver {
   static public void main(String args[]) {
      TicTacToe game = new TicTacToe();
      do { System.out.print(game); }
      while(game.notOver());
   }
public class TicTacToe {
   public boolean notOver() { return false; }
   public String toString() { return("TicTacToe\n");}
}
```

? How do you iteratively “grow” a program?

– Always have a running version of your program.

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Version 1 — game state

We will use chess notation to access the game state

- Columns ‘a’ through ‘c’
- Rows ‘1’ through ‘3’

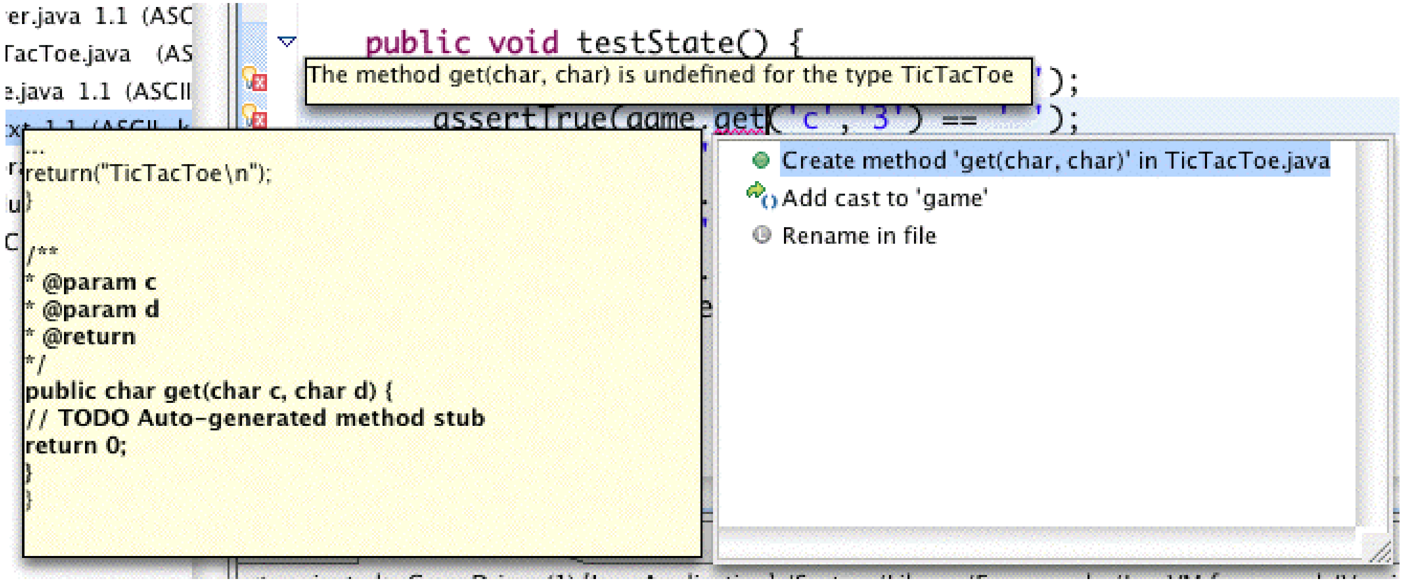
? How do we decide on the right interface?

- First write some tests!

Test-first development

```
public class TicTacToeTest {
    private TicTacToe game;

    @Before public void setUp() {
        super.setUp();
        game = new TicTacToe();
    }

    @Test public void testState() {
        assertTrue(game.get('a','1') == ' ');
        assertTrue(game.get('c','3') == ' ');
        game.set('c','3','X');
        assertTrue(game.get('c','3') == 'X');
        game.set('c','3',' ');
        assertTrue(game.get('c','3') == ' ');
        assertFalse(game.inRange('d','4'));
    }
}
```

Generating methods



```
public void testState() {
```

The method get(char, char) is undefined for the type TicTacToe

```
assertTrue(game.get('c','3') == ' ');
```

```
...
return("TicTacToe\n");
}

/**
* @param c
* @param d
* @return
*/
public char get(char c, char d) {
// TODO Auto-generated method stub
return 0;
}
}
```

Test-first programming can drive the development of the class interface …

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Representing game state

```
public class TicTacToe {
   private char[][] gameState;
   public TicTacToe() {
      gameState = new char[3][3];
      for (char col='a'; col <='c'; col++)
         for (char row='1'; row<='3'; row++)
            this.set(col,row,' ');
   }
...
```

Checking pre-conditions

set() and get() translate from chess notation to array indices.

```
public void set(char col, char row, char mark) {
   assert(inRange(col, row));  // NB: precondition
   gameState[col-'a'][row-'1'] = mark;
}
public char get(char col, char row) {
   assert(inRange(col, row));
   return gameState[col-'a'][row-'1'];
}
public boolean inRange(char col, char row) {
   return (('a'<=col) && (col<='c')
      && ('1'<=row) && (row<='3'));
}
```

Printing the State

By re-implementing TicTacToe.toString(), we can view the state of the game:

```
3    |   |
  ---+---+---
2    |   |
  ---+---+---
1    |   |
   a   b   c
```

? How do you make an object printable?

- Override Object.toString()

TicTacToe.toString()

Use a StringBuilder (not a String) to build up the representation:

```
public String toString() {
   StringBuffer rep = new StringBuilder();
   for (char row='3'; row>='1'; row--) {
      rep.append(row);
      rep.append("   ");
      for (char col='a'; col <='c'; col++) { ... }
      ...
   }
   rep.append("   a   b   c\n");
   return(rep.toString());
}
```

PS: newer version of Java have improved on using String directly

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Version 2 — adding game logic

We will:

Add test scenarios

Add Player class

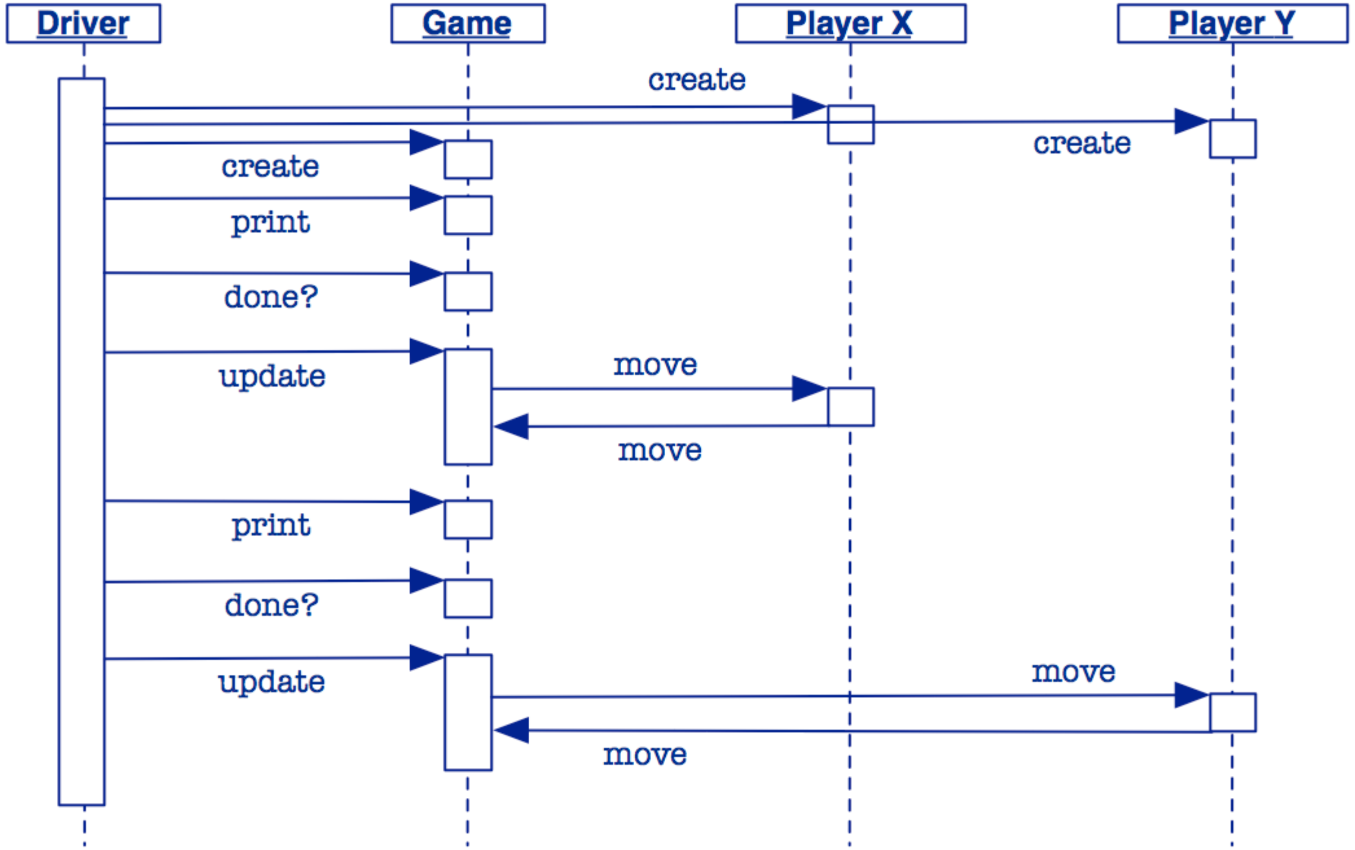
Add methods to make moves, test for winning

Refining the interactions

We will want both real and test Players, *so the Driver should create them.*

Updating the Game and printing it *should be separate operations.*

The Game should ask the Player to make a move, and *then the Player will attempt to do so.*



Testing scenarios

Our test scenarios will play and test scripted games

```
@Test public void testXWinDiagonal() {
   checkGame("a1\nb2\nc3\n", "b1\nc1\n", "X", 4);
}
// more tests …

public void checkGame(String Xmoves, String Omoves,
      String winner, int squaresLeft) {
   Player X = new Player('X', Xmoves);// a scripted player
   Player O = new Player('O', Omoves);
   TicTacToe game = new TicTacToe(X, O);
   GameDriver.playGame(game);
   assertTrue(game.winner().name().equals(winner));
   assertTrue(game.squaresLeft() == squaresLeft);
}
```

Running the test cases

```

3    |   |
  ---+---+---
2    |   |
  ---+---+---
1    |   |
   a   b   c
Player X moves: X at a1
3    |   |
  ---+---+---
2    |   |
  ---+---+---
1  X |   |
   a   b   c
...
```

```
Player O moves: O at c1
3    |   |
  ---+---+---
2    | X |
  ---+---+---
1  X | O | O
   a   b   c
Player X moves: X at c3
3    |   | X
  ---+---+---
2    | X |
  ---+---+---
1  X | O | O
   a   b   c
game over!
```

The Player

We use different constructors to make real or test Players:

```
public class Player {
    private final char mark;
    private final BufferedReader in;
```

A real player reads from the standard input stream:

```
    public Player(char mark) {
        this(mark, new BufferedReader(
              new InputStreamReader(System.in)
        ));
    }
```

This constructor just calls another one ...

...

Player constructors ...

But a Player can be constructed that reads its moves from any input buffer:

```
   protected Player(char initMark, BufferedReader initIn) {
          mark = initMark;
          in = initIn;
   }
```

This constructor is not intended to be called directly.

...

Player constructors ...

A test Player gets its input from a String buffer:

```
public Player(char mark, String moves) {
    this(mark, new BufferedReader(
            new StringReader(moves)
    ));
}
```

The default constructor returns a dummy Player representing “nobody”

```
public Player() { this(' '); }
```

Roadmap

TicTacToe example

- Identifying objects
- Scenarios
- Test-first development
- Printing object state
- Testing scenarios
- Representing responsibilities as contracts

Tic Tac Toe Contracts

Explicit invariants:

- turn (current player) is either X or O
- X and O swap turns (turn never equals previous turn)
- game state is 3×3 array marked X, O or blank
- winner is X or O iff winner has three in a row

Implicit invariants:

- initially winner is nobody; initially it is the turn of X
- game is over when all squares are occupied, or there is a winner
- a player cannot mark a square that is already marked

Contracts:

- the current player may make a move, if the invariants are respected

Encoding the contract

We must introduce state variables to implement the contracts

```
public class TicTacToe {
   static final int X = 0;                // constants
   static final int O = 1;
   private char[][] gameState;
   private Player winner = new Player();  // = nobody
   private Player[] player;
   private int turn = X;                  // initial turn
   private int squaresLeft = 9;
...
```

Supporting test Players

The Game no longer instantiates the Players, but accepts them as constructor arguments:

```
public TicTacToe(Player playerX, Player playerO)
{   // ...
   player = new Player[2];
   player[X] = playerX;
   player[O] = playerO;
}
```

Invariants

These conditions may seem obvious, which is exactly why they should be checked ...

```
private boolean invariant() {
   return (turn == X || turn == O)
      && ( this.notOver()
         || this.winner() == player[X]
         || this.winner() == player[O])
      && (squaresLeft < 9          // else, initially:
         || turn == X && this.winner().isNobody());
}
```

Assertions and tests often tell us what methods should be implemented, and whether they should be public or private.

Delegating Responsibilities

When Driver updates the Game, the Game just asks the Player to make a move:

```
  public void update() throws IOException {
     player[turn].move(this);
  }
```

Note that the Driver may not do this directly!

...

Delegating Responsibilities ...

The Player, in turn, calls the Game’s move() method:

```
public void move(char col, char row, char mark) {
    assert(notOver());
    assert(inRange(col, row));
    assert(get(col, row) == ' ');
    System.out.println(mark + " at " + col + row);
    this.set(col, row, mark);
    this.squaresLeft--;
    this.swapTurn();
    this.checkWinner();
    assert(invariant());
}
```

Small Methods

Introduce methods that make the intent of your code clear.

```
public boolean notOver() {
   return this.winner().isNobody()
              && this.squaresLeft() > 0;
}
private void swapTurn() {
   turn = (turn == X) ? O : X;
}
```

Well-named variables and methods typically eliminate the need for explanatory comments!

Use comments to explain non obvious design, algorithmic or implementation choices

Accessor Methods

Accessor methods protect clients from changes in implementation:

```
public Player winner() {
   return winner;
}
public int squaresLeft() {
   return this.squaresLeft;
}
```

? When should instance variables be public?

- Almost never! Declare public accessor methods instead.

getters and setters in Java

Accessors in Java are known as “getters” and “setters”.

- Accessors for a variable x should normally be called getx() and setx()

Frameworks such as EJB depend on this convention!

Code Smells — TicTacToe.checkWinner()

Duplicated code stinks!

How can we clean it up?

```
private void checkWinner()
   {
  char player;
  for (char row='3'; row>='1'; row--) {
    player = this.get('a',row);
    if (player == this.get('b',row)
      && player == this.get('c',row)) {
      this.setWinner(player);
      return;
    }
  }
```

```
  for (char col='a'; col <='c'; col++) {
    player = this.get(col,'1');
    if (player == this.get(col,'2')
      && player == this.get(col,'3')) {
      this.setWinner(player);
      return;
    }
  }
  player = this.get('b','2');
  if (player == this.get('a','1')
    && player == this.get('c','3')) {
      this.setWinner(player);
      return;
  }
  if (player == this.get('a','3')
    && player == this.get('c','1')) {
      this.setWinner(player);
      return;
  }
}
```

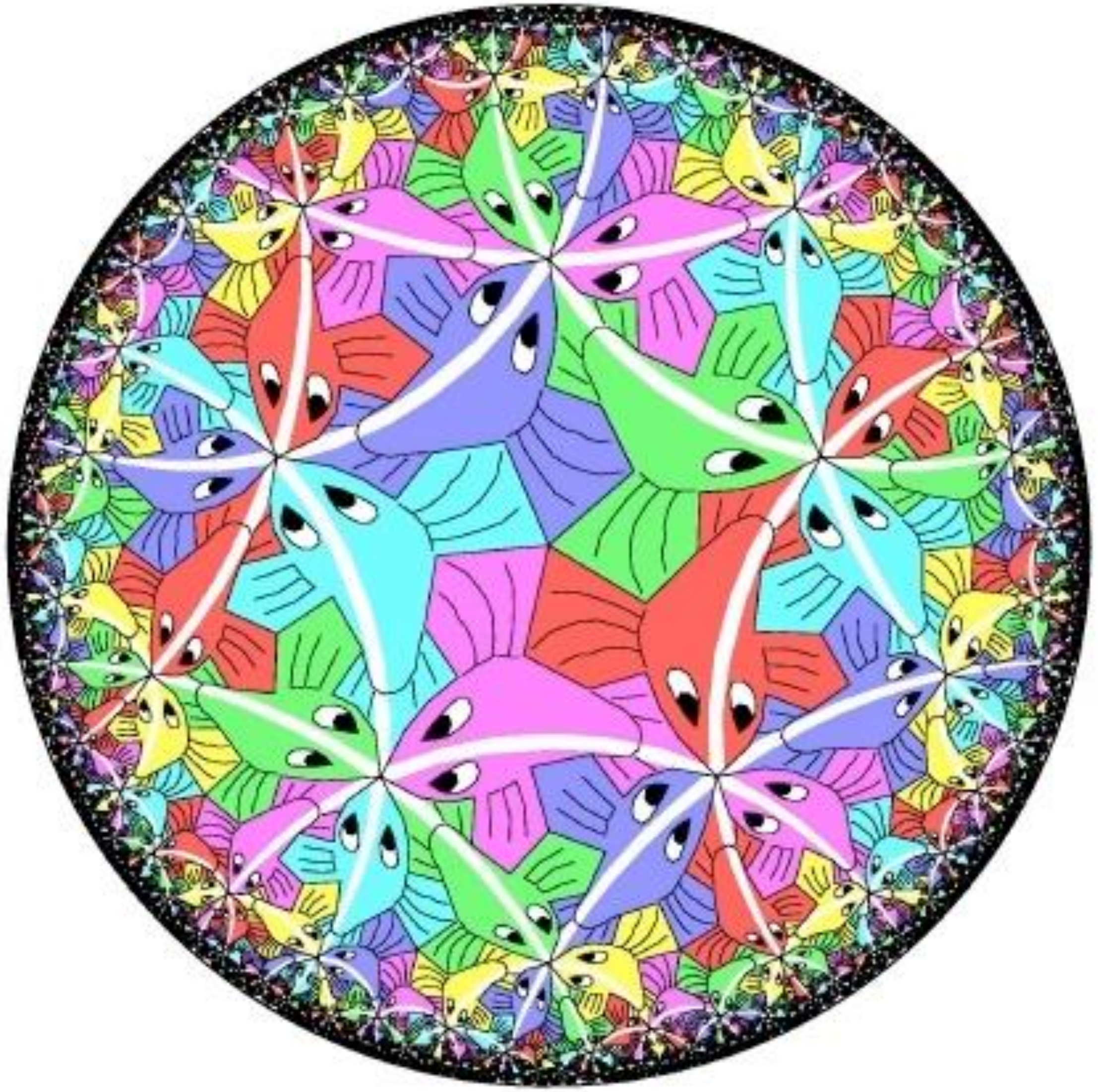
GameDriver

In order to run test games, we separated Player instantiation from Game playing:

```
public class GameDriver {
   public static void main(String args[]) {
      try {
         Player X = new Player('X');
         Player O = new Player('O');
         TicTacToe game = new TicTacToe(X, O);
         playGame(game);
      } catch (AssertionException err) {
         ...
      }
   }
```

How can we make test scenarios play silently?

Patterns



Bit of history...

Christoffer Alexander

- “The Timeless Way of Building”, Christoffer Alexander, Oxford University Press, 1979, ISBN 0195024028
- Structure of the book is magnificent (cfr writing your master dissertation…)
  - Christmass is close ;-)

More advanced than what computer science uses

- only the simple parts got mainstream

Alexander’s patterns

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without doing it the same way twice”

- Alexander uses this as part of the solution to capture the “quality without a name”

Illustrating Recurring Patterns...

Essential Elements in a Pattern

Pattern name

- Increase of design vocabulary

Problem description

- When to apply it, in what context to use it

Solution description (generic !)

- The elements that make up the design, their relationships, responsibilities, and collaborations

Consequences

- Results and trade-offs of applying the pattern

GRASP Patterns

guiding principles to help us assign responsibilities

GRASP “Patterns” – guidelines

- Controller
- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Hs 17

Hs 25

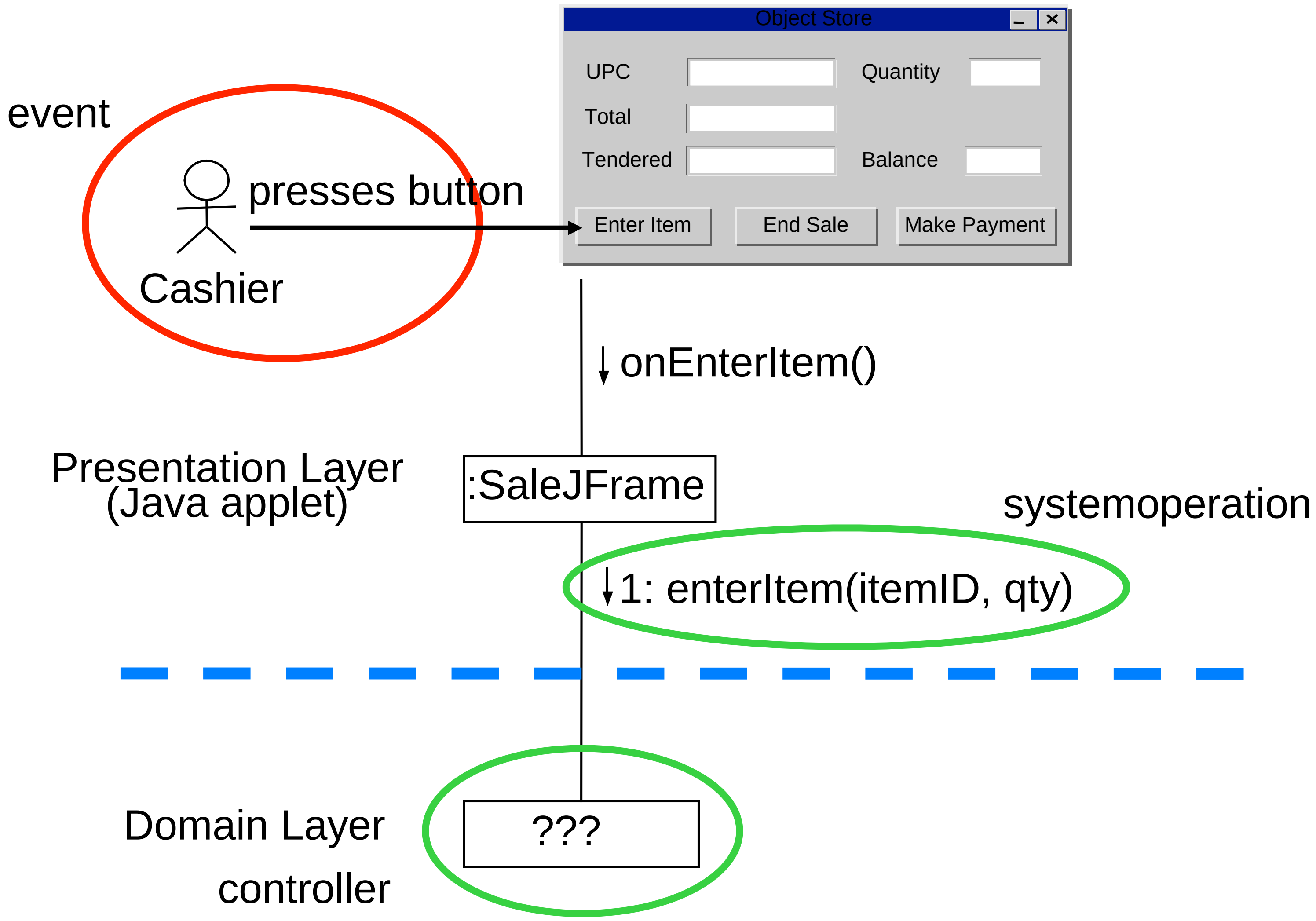
GRASP Patterns

We already saw High Coupling and Low Cohesion before

Let’s look at some more GRASP patterns now…

1. Controller Pattern

Who is responsible for handling Systemoperations ?



event

Cashier presses button

onEnterItem()

Presentation Layer (Java applet)

:SaleJFrame

systemoperation

1: enterItem(itemID, qty)

Domain Layer

controller

???

Controller Pattern

Pattern	Controller
Problem	Who is responsible for handling system events ?
Solution	Assign the responsibility to a class C representing one of the following choices: • C is a *facade controller*: it represents the overall system, a root object, the device that runs the software, or a major subsystem. • C is a *use case or session controller*: it represents an artificial objects (see *Pure Fabrication* pattern) that handles all events from a use case or session

System operations and System events

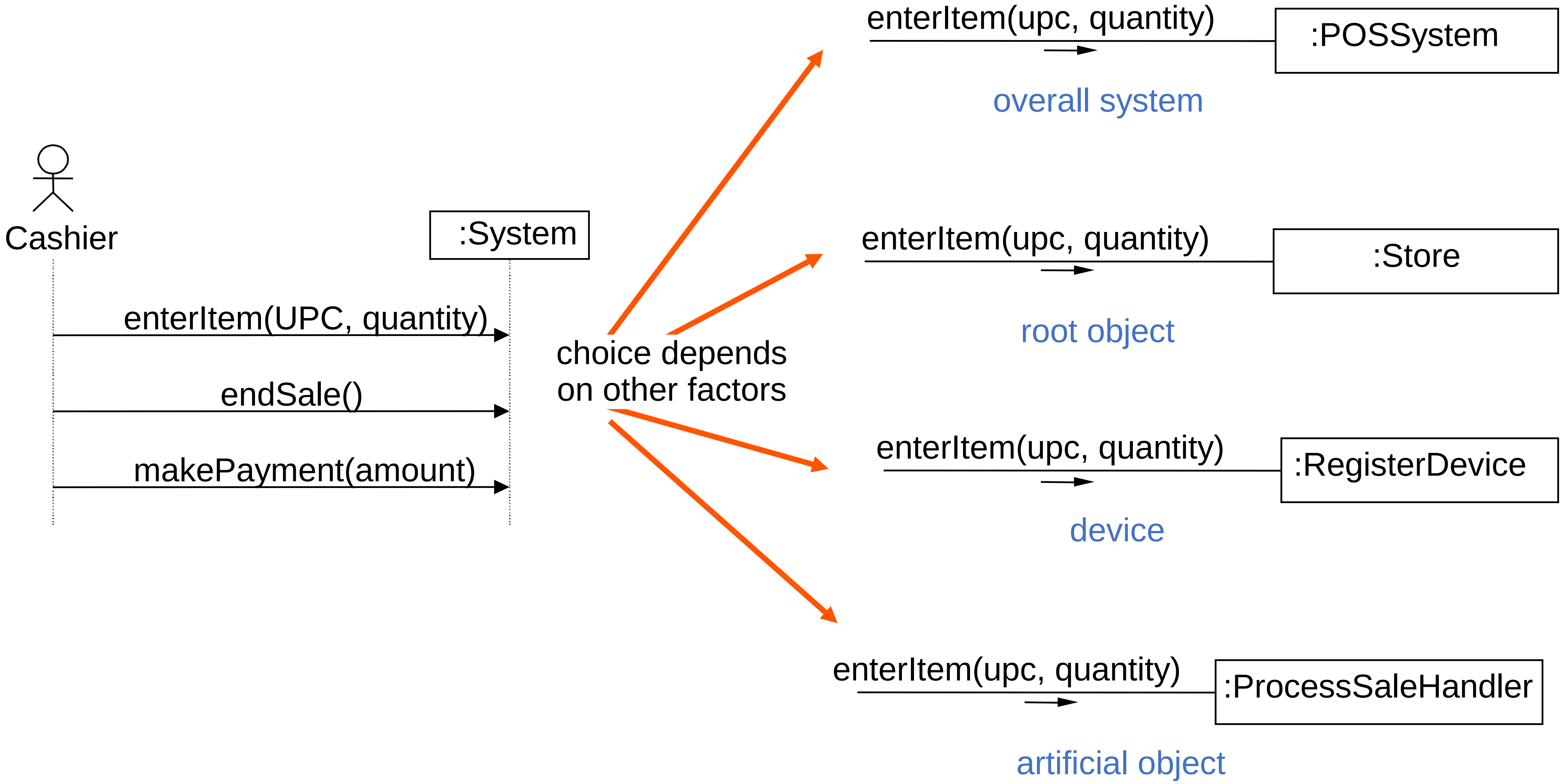
From analysis to design:

- Analysis: can group system operations in a conceptual “System” class
- Design: give responsibility for processing system operations to controller classes

Controller classes are not part of the User Interface

Model-View-Controller (MVC)

Who controls System events?



Controller Pattern: Guidelines

Limit the responsibility to “control and coordination”

- Controller = delegation pattern
- delegate real work to real objects
- Common mistake: fat controllers with too much behavior

Only support a limited number of events in Facade controllers

Controller Pattern: Use Case Controller Guidelines

Use Case (UC) controllers

- consider when too much coupling and not enough cohesion in other controllers (factor system events)
- Treat all UC events in the same controller class
- Allow control on the order of events
- Keep information on state of UC (statefull session)

Controller Pattern: Problems and Solutions

“Bloated” controllers

- symptoms
  - a single controller handling all system events
  - controller not delegating work
  - controller with many attributes, with system information, with duplicated information
- solutions
  - add Use Case controllers
  - design controllers that delegate tasks

Controller Pattern: Advantages

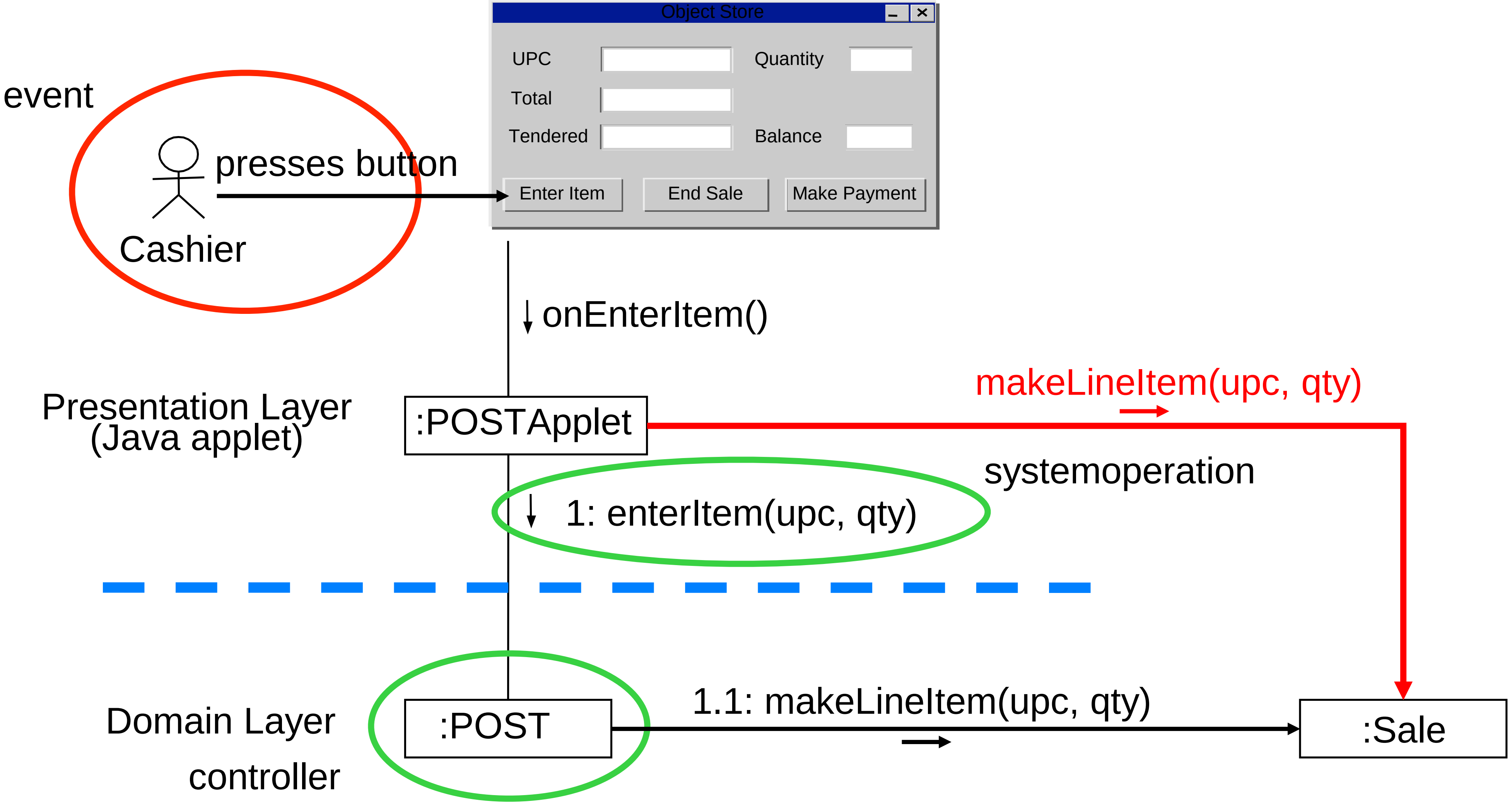
Increased potential for reuse

- domain-level processes handled by domain layer
- decouple GUI from domain level !
- Different GUI or different ways to access the domain level

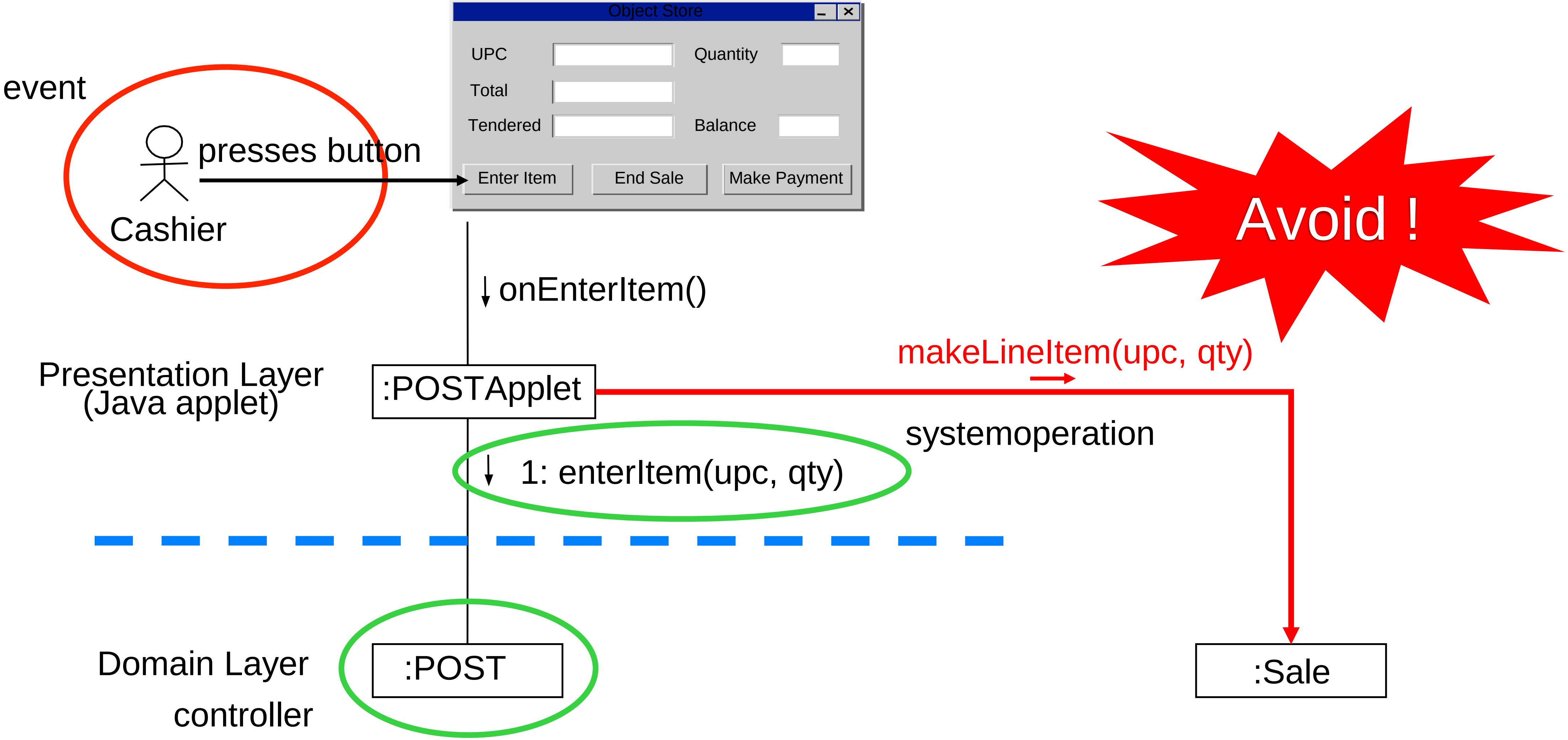
Reason about the state of the use case

- guarantee sequence of system operations

Example applying Controller Pattern



Example: **incorrect!**

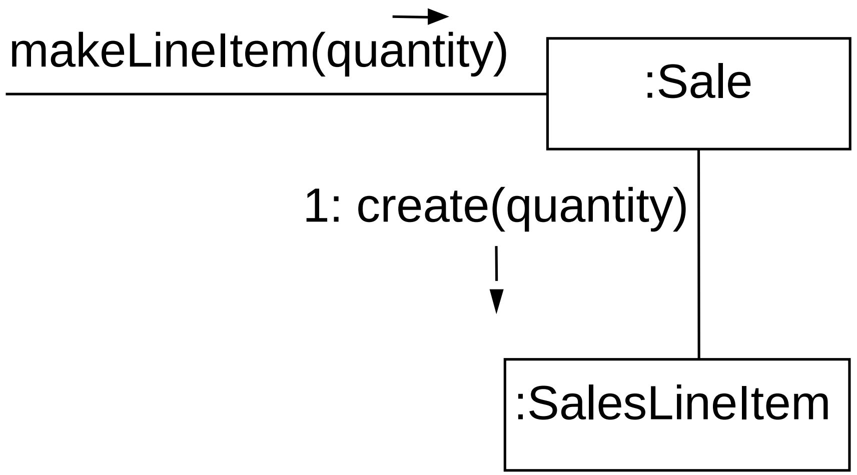


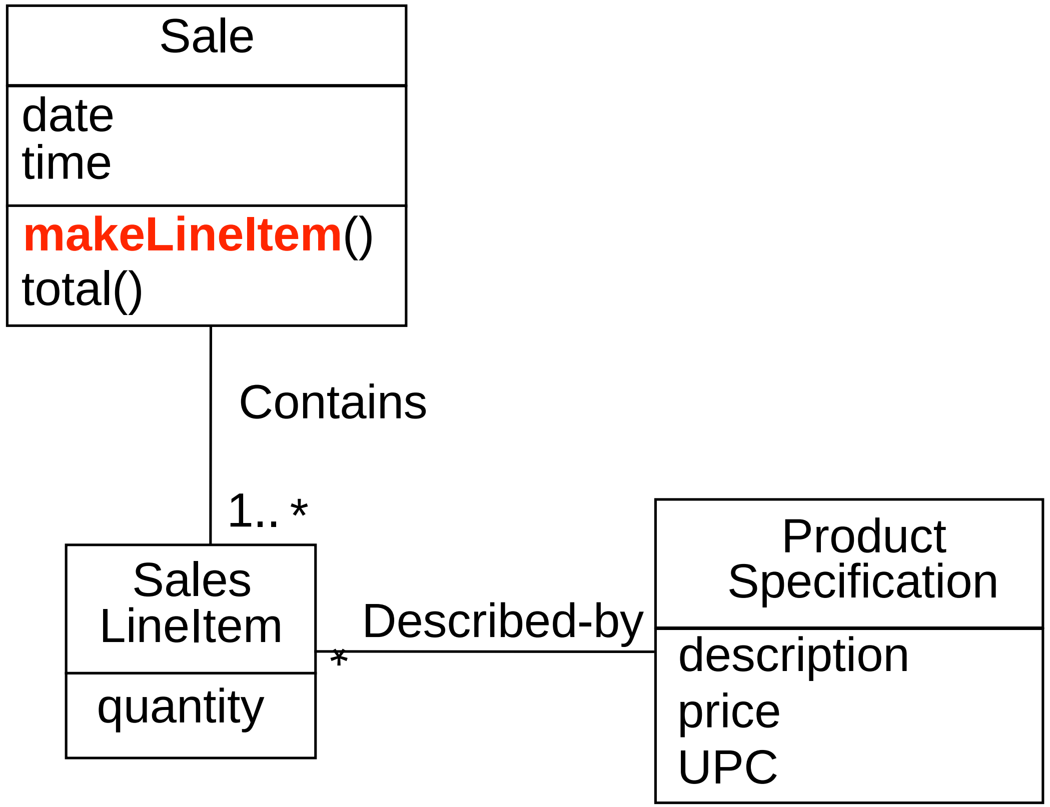
2. Creator Pattern

Pattern	Creator
Problem	Who is responsible for creating instances of classes ?
Solution	Assign a class B to create instances of a class A if: • B is a composite of A objects (*composition/aggregation*) • B contains A objects (*contains*) • B holds instances of A objects (*records*) • B closely collaborates with A objects • B has the information needed for creating A objects

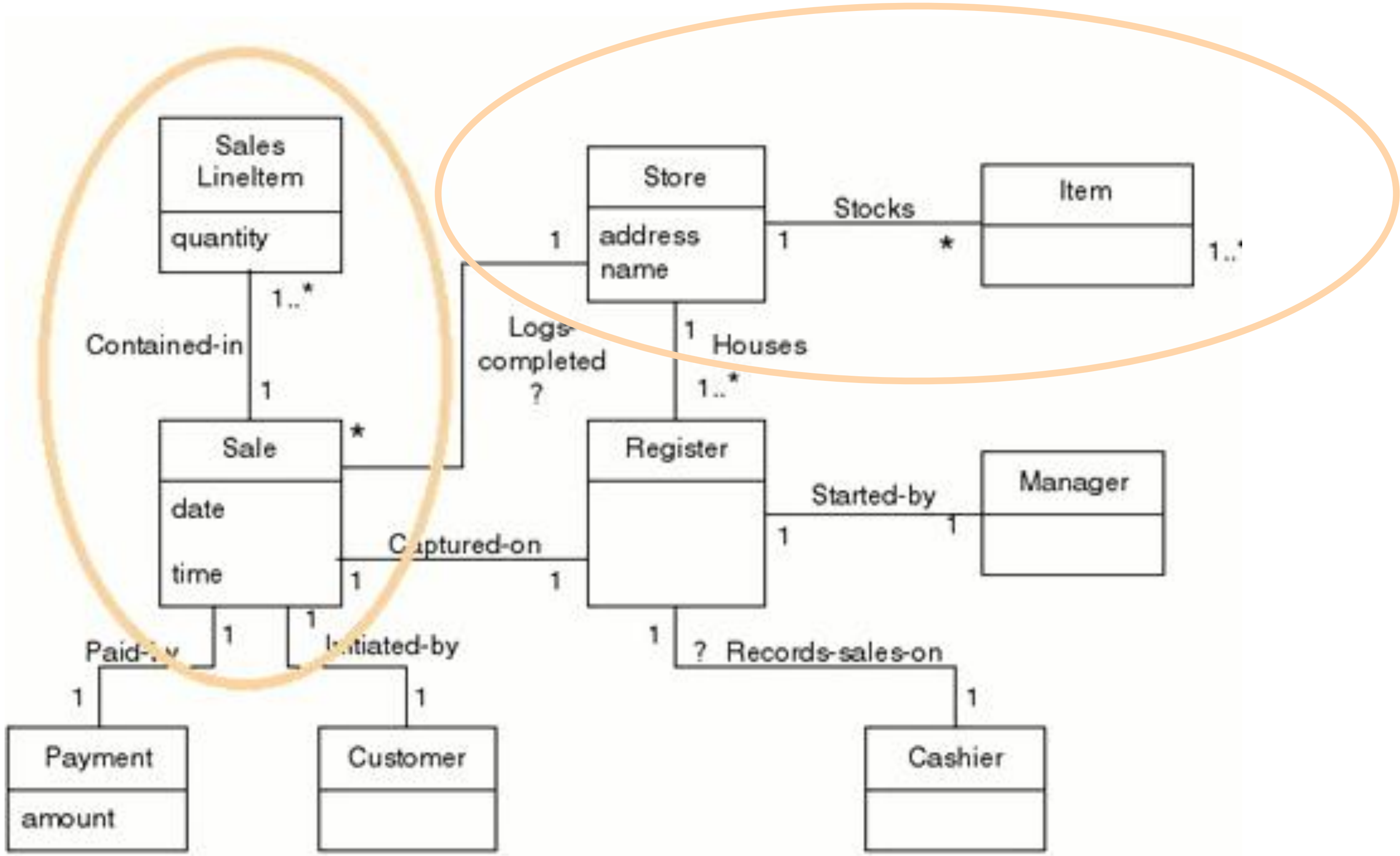
Creator Pattern: example

Creation of “SalesLineItem” instances





Creator Pattern: Inspiration from the Domain Model



3. Information Expert Pattern

A very basic principle of responsibility assignment

(cfr Responsibility Driven questions seen earlier)

Assign a responsibility to the object that has the information necessary to fulfill it -the information expert

“That which has the information, does the work”

Related to the principle of “low coupling”

⇨ Localize work

Expert Pattern

Pattern	(Information) Expert
Problem	What is the basic principle to assign responsibilities to objects ?
Solution	Assign responsibility to the class that has the information to fulfill it (the information expert)

Expert Pattern: remarks

Real-world analogy

- who predicts gains/losses in a company?
  - the person with access to the date (Chief Financial Officer)

Needed information to work out ‘responsibility’
=> spread over different objects

- “partial” experts that collaborate to obtain global information (interaction is required)

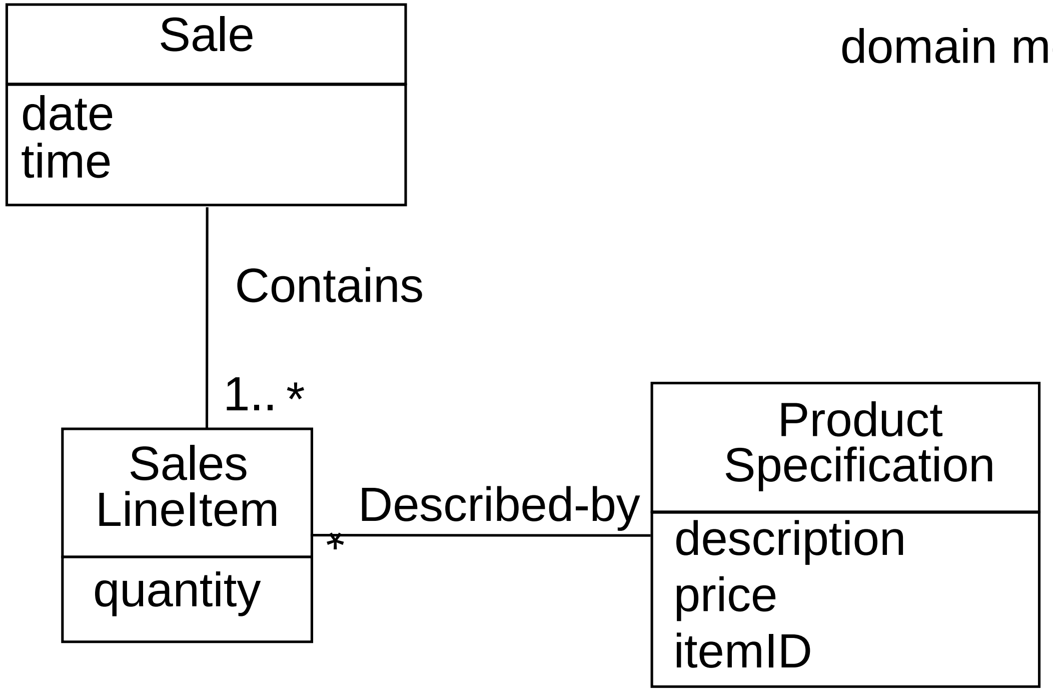
Not necessarily the best solution (e.g. database access)

- See low coupling & high cohesion

Expert Pattern: example 1

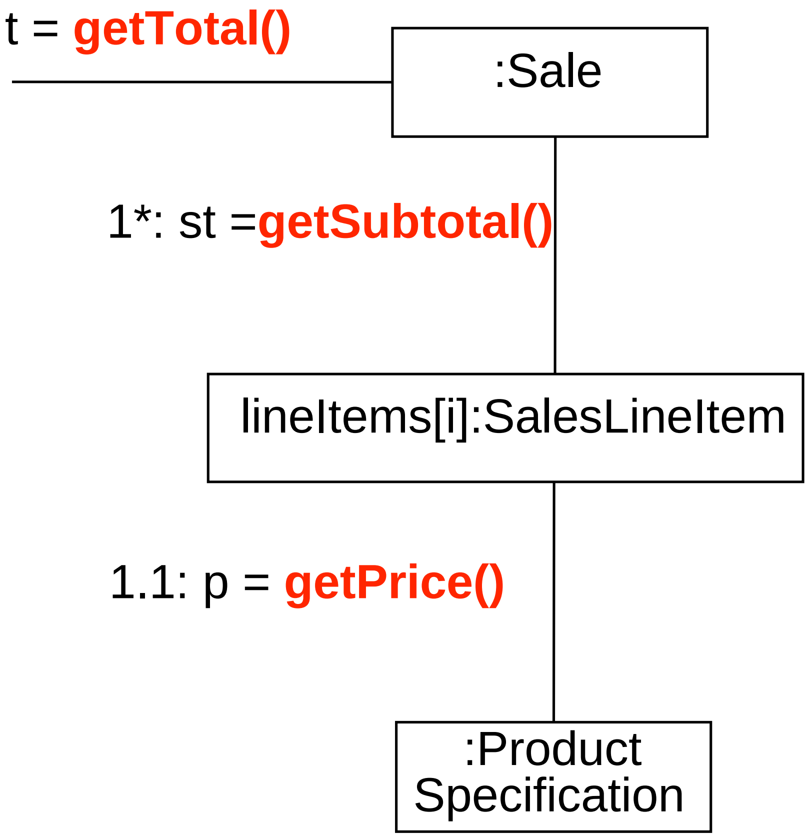
Example: Who is responsible for knowing the total of a “Sale”?

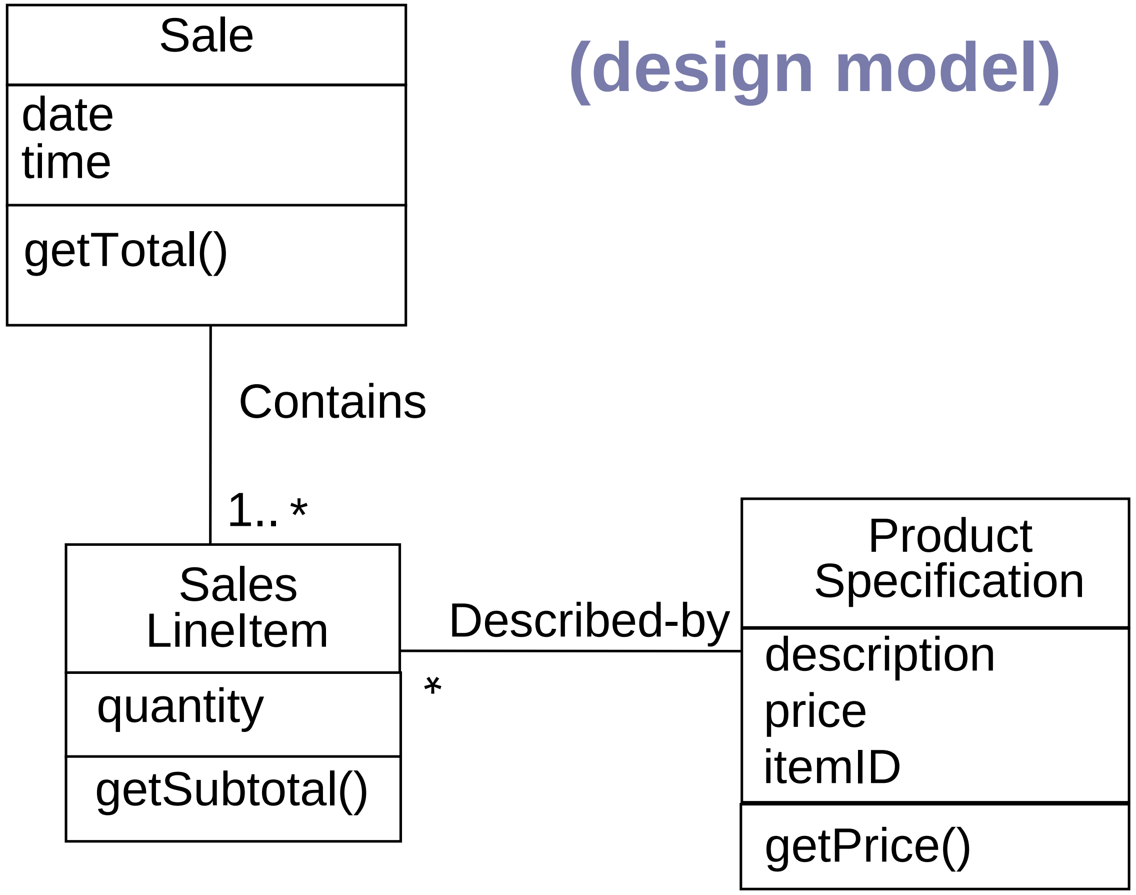
Who possesses the information?



domain model

Expert Pattern: example 1



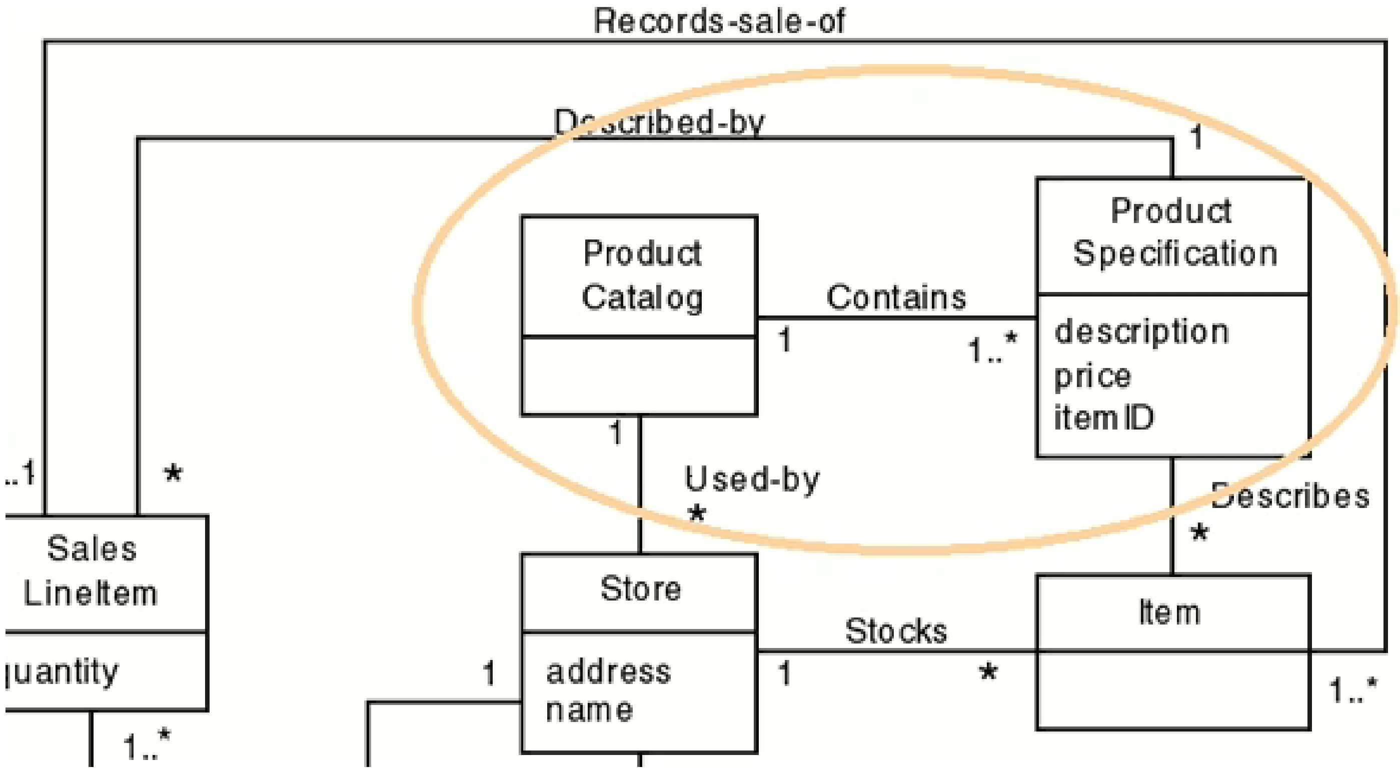


class diagram

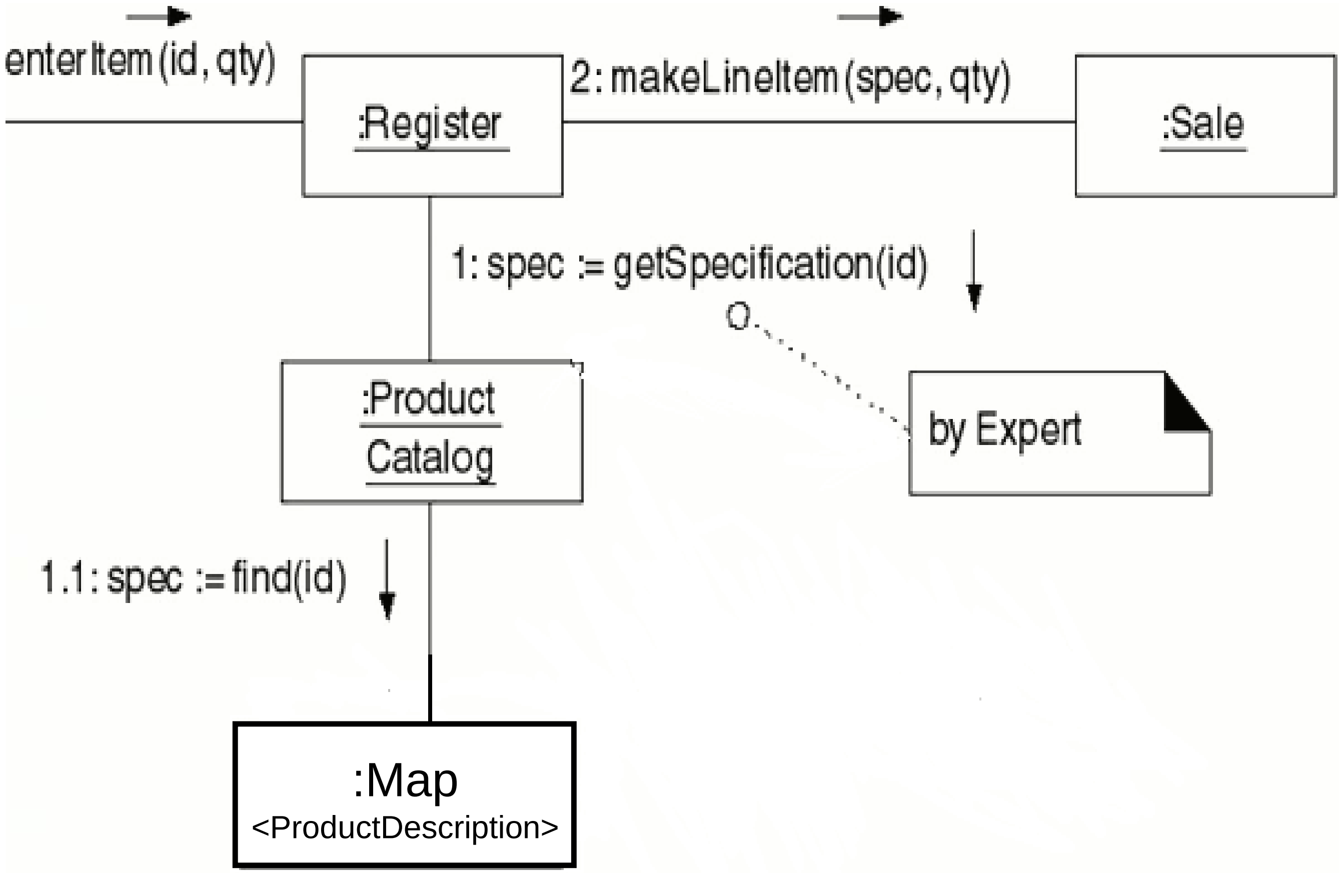
(design model)

Expert Pattern: Example 2

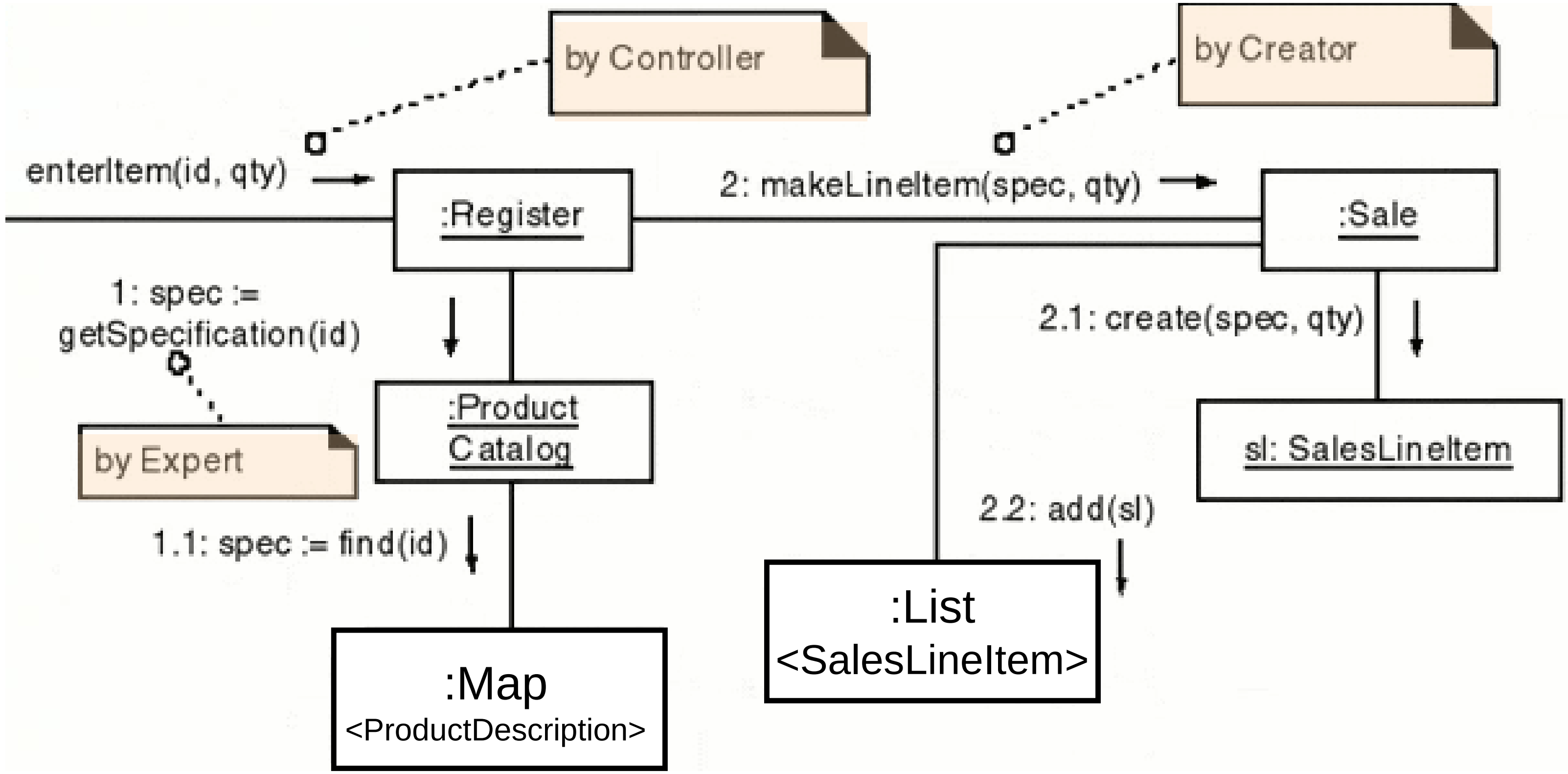
What object should be responsible for knowing ProductSpecifications, given a key?
Take inspiration from the domain model



Applying Information Expert



Design for “enterItem”: 3 patterns applied



GRASP Patterns

guiding principles to help us assign responsibilities

GRASP “Patterns” – guidelines

- Controller
- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Hs 17

Hs 25

6. Polymorphism

Pattern	Polymorphism
Problem	How handle alternatives based on type? How to create pluggable software components?
Solution	When related alternatives or behaviours vary by type (class), assign responsibility for the behavior -using polymorphic operations- to the types for which the behavior varies.

Note: Not really a pattern but basic OO principle !

Example

procedural approach:
passing an int and using switch to decide which behavior to execute based on that int

```
void CVideoAppUi::HandleCommandL(TInt aCommand)
   {
   switch ( aCommand )
      {
           case EAknSoftkeyExit:
           case EAknSoftkeyBack:
           case EEikCmdExit:
                 { Exit();  break; }

           // Play command is selected
           case EVideoCmdAppPlay:
                { DoPlayL(); break; }

            // Stop command is selected
           case EVideoCmdAppStop:
                { DoStopL(); break; }

           // Pause command is selected
           case EVideoCmdAppPause:
                { DoPauseL(); break; }

           // DocPlay command is selected
           case EVideoCmdAppDocPlay:
                { DoDocPlayL(); break; }
           ......
```

Nokia S60 mobile video player 3gpp source code
http://www.codeforge.com/article/192637

Solution: Replace case by Polymorphism

OO approach:
passing an object and using polymorphism to select behavior to execute

```
void CVideoAppUi::HandleCommandL(Command aCommand)
   {
              aCommand.execute();
             }
```

Create a Command class hierarchy, consisting of a (probably) abstract class AbstractCommand, and subclasses for every command supported. Implement execute on each of these classes:

- virtual void AbstractCommand::execute() = 0;
- virtual void PlayCommand::execute() { ... do play command ...};
- virtual void StopCommand::execute() { ... do stop command ...};
- virtual void PauseCommand::execute() { ... do pause command ...};
- virtual void DocPlayCommand::execute() { ... do docplay command ...};
- virtual void FileInfoCommand::execute() { ... do file info command ...};

7. Pure Fabrication Pattern

Pattern	Pure Fabrication
Problem	What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?
Solution	Assign a cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept but is purely imaginary and fabricated to obtain a pure design with high cohesion and low coupling.

Pure Fabrication Pattern

Where no appropriate class is present: invent one

- Even if the class does not represent a problem domain concept
- “pure fabrication” = making something up: do when we’re desperate!

This is a compromise that often has to be made to preserve cohesion and low coupling

- Remember: the software is not designed to simulate the domain, but operate in it
- The software does not always have to be identical to the real world
  - Domain Model ≠ Design model

Pure Fabrication Example

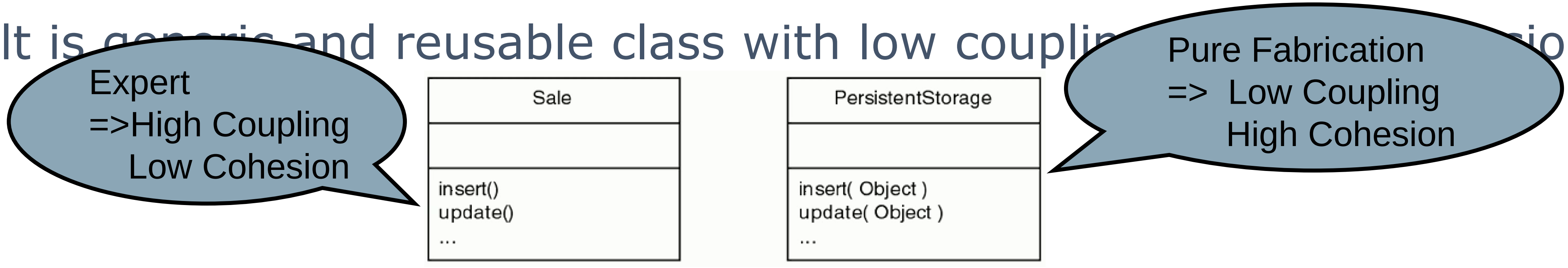
Suppose Sale instances need to be saved in a database

Option 1: assign this to the Sale class itself (Expert pattern)

- Implications of this solution:
  - auxiliary database-operations need to be added as well
  - coupling with particular database connection class
  - saving objects in a database is a general service

Option 2: create PersistentStorage class

- Result is generic and reusable class with low coupling and high cohesion



8. Indirection Pattern

Pattern	Indirection
Problem	Where to assign a responsibility to avoid direct coupling between two (or more) things? How to de-couple objects so that low coupling is supported and reuse potential remains higher?
Solution	Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. This intermediary creates an indirection between the other components.

Indirection Pattern

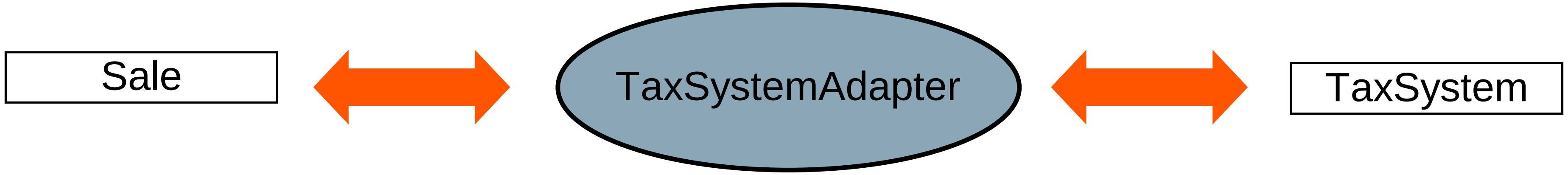
A common mechanism to reduce coupling

Assign responsibility to an intermediate object to decouple two components

- coupling between two classes of different subsystems can introduce maintenance problems

“most problems in computer science can be solved by another level of indirection”

- A large number of design patterns are special cases of indirection (Adapter, Facade, Observer)



9. Protected Variations Pattern

Pattern	Protected Variations
Problem	How to design objects, subsystems, and systems so that the variations or instability of these elements does not have an undesirable impact on other elements ?
Solution	Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

Protected Variations – example

Video game companies make money by creating a game engine

- many games use the same engine
- what if a game is to be ported to another console ???
  - a wrapper object will have to delegate 3D graphics drawing to different console-level commands
  - the wrapper is simpler to change than the entire game and all of its facets

Wrapping the component in a stable interface means that when variations occur, only the wrapper class need be changed

- In other words, changes remain localized
- The impact of changes is controlled

FUNDAMENTAL PRINCIPLE IN SW DESIGN

Protected Variations – Example

Open DataBase Connectivity (ODBC/JDBC)

- These are packages that allow applications to access databases in a DB-independent way
  - In spite of the fact that databases all use slightly different methods of communication
  - It is possible due to an implementation of Protected Variations
- Users write code to use a generic interface
  - An adapter converts generic method calls to DB and vice versa

Conclusion

Always try to apply and balance basic OO Design Principles

- Minimize Coupling
- Increase Cohesion
- Distribute Responsibilities

Use and learn from established sources of information

- Responsibility Driven Design
- GRASP patterns
  - Design Patterns: see later

References

Rebecca Wirfs-Brock, Alan McKean, Object Design — Roles, Responsibilities and Collaborations, Addison-Wesley, 2003.

http://www.wirfs-brock.com/PDFs/Responsibility-Driven.pdf

Craig Larman, Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.), Prentice Hall, 2005.

Design of Software Systems
(Ontwerp van SoftwareSystemen)

3 Design Patterns

Roel Wuyts
2016-2017

Warm-up Exercise

We need a design to be able to print different kinds of text (ASCII and PostScript) on different kinds of printers (ASCIIPrinter and PSPrinter).

ASCIIPrinters can only print ASCII text,
but PostscriptPrinters can print Postscript text as well as ASCIIText, after internally converting ASCIIText to Postscript text.

New types of printers and texts will be added in the near future.

Alexander’s patterns

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without doing it the same way twice”

- Alexander uses this as part of the solution to capture the “quality without a name”

Illustrating Recurring Patterns...

Beijing (China)

Zurich (Switzerland)

Osio (China)

Brussels (Belgium)

Singapore (Thailand)

Alert!

Do not overreact seeing all these patterns!

Do not apply too many patterns!

Look at the trade-offs!

Most patterns makes systems more complex!

- but address a certain need.

As always: do good modeling.

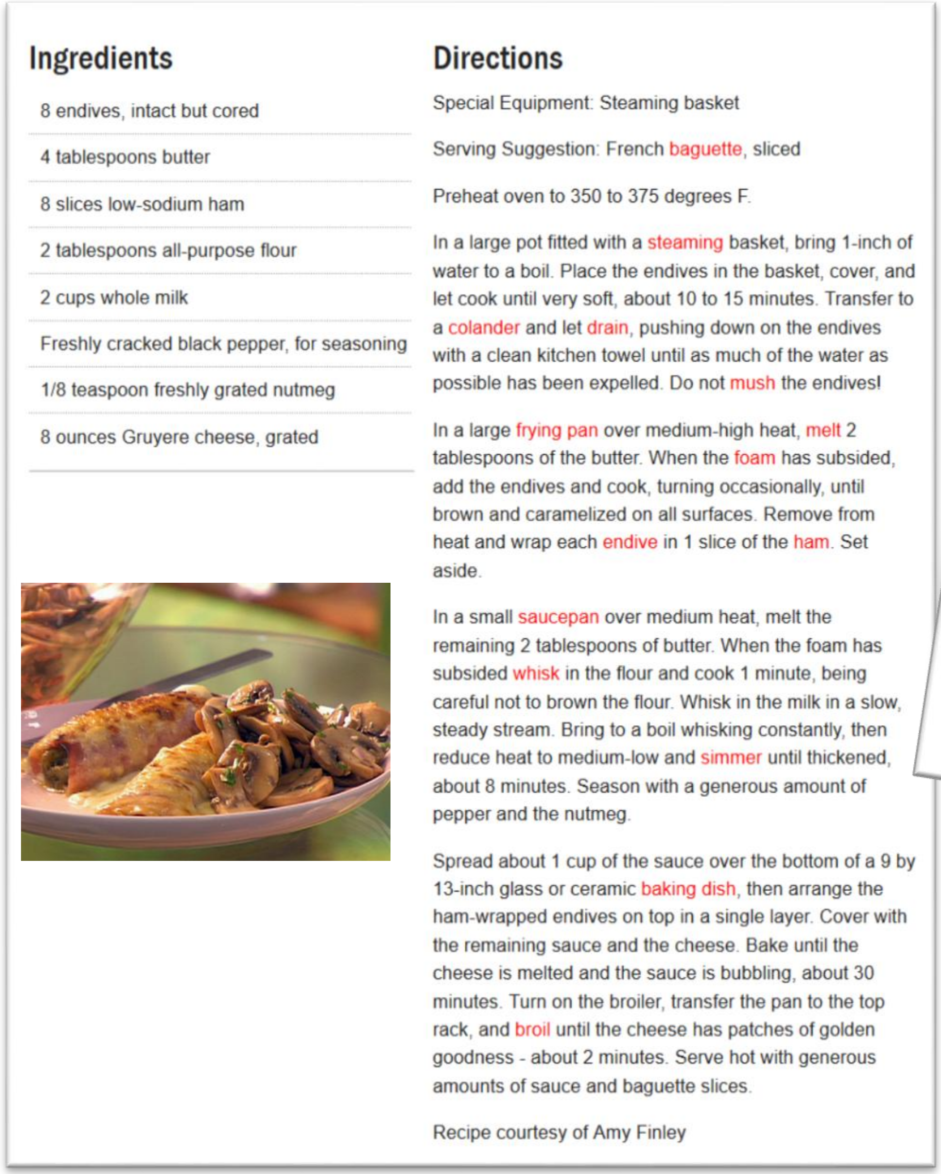
- First start your design and note problems or difficulties,
- then propose multiple potential solutions with different trade-offs,
  - potentially using patterns,
- then take motivated decision

Design Patterns

A Design Pattern is a *pattern* that captures a solution to a *recurring design problem*

- It is **not** a pattern for implementation problems
- It is **not** a ready-made solution that has to be applied
  - It's still up to you !
  - You can simply use the pattern for inspiration,
  - Or only apply part of the design pattern
  - Remember the basic OO design principles and use them to weigh your design

Adapt to suit taste, allergies, nr. of people, available ingredients, …



Ingredients

- 8 endives, intact but cored
- 4 tablespoons butter
- 8 slices low-sodium ham
- 2 tablespoons all-purpose flour
- 2 cups whole milk
- Freshly cracked black pepper, for seasoning
- 1/8 teaspoon freshly grated nutmeg
- 8 ounces Gruyere cheese, grated

Directions

Special Equipment: Steaming basket

Serving Suggestion: French baguette, sliced

Preheat oven to 350 to 375 degrees F.

In a large pot fitted with a steaming basket, bring 1-inch of water to a boil. Place the endives in the basket, cover, and let cook until very soft, about 10 to 15 minutes. Transfer to a colander and let drain, pushing down on the endives with a clean kitchen towel until as much of the water as possible has been expelled. Do not mush the endives!

In a large frying pan over medium-high heat, melt 2 tablespoons of the butter. When the foam has subsided, add the endives and cook, turning occasionally, until brown and caramelized on all surfaces. Remove from heat and wrap each endive in 1 slice of the ham. Set aside.

In a small saucepan over medium heat, melt the remaining 2 tablespoons of butter. When the foam has subsided whisk in the flour and cook 1 minute, being careful not to brown the flour. Whisk in the milk in a slow, steady stream. Bring to a boil whisking constantly, then reduce heat to medium-low and simmer until thickened, about 8 minutes. Season with a generous amount of pepper and the nutmeg.

Spread about 1 cup of the sauce over the bottom of a 9 by 13-inch glass or ceramic baking dish, then arrange the ham-wrapped endives on top in a single layer. Cover with the remaining sauce and the cheese. Bake until the cheese is melted and the sauce is bubbling, about 30 minutes. Turn on the broiler, transfer the pan to the top rack, and broil until the cheese has patches of golden goodness - about 2 minutes. Serve hot with generous amounts of sauce and baguette slices.

Recipe courtesy of Amy Finley

Design Patterns

Example:

- “We are implementing a drawing application. The application allows the user to draw several kinds of figures (circles, squares, lines, polymorphs, bezier splines). It also allows to group these figures (and ungroup them later). Groups can then be moved around and are treated like any other figure.”

➡ Look at *Composite* Design Pattern

Pattern structure

A design pattern is a kind of blueprint

Consists of different parts

- All of these parts make up the pattern!
- When we talk about the pattern we therefore mean all of these parts together
  - not only the class diagram...

Tip: remember this for the exam – know your complete pattern

Why Patterns?

Smart

- Elegant solutions that a novice would not think of

Generic

- Independent on specific system type, language
  - Although biased towards statically-typed class-based OO languages (C++, Java, …)

Well-proven

- Successfully tested in several systems

Simple

- Combine them for more complex solutions

GoF Design Pattern Book

Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang-of-Four (GoF))

Book is still very relevant today but (depending on edition):

- Original book uses OMT notation (analogous to UML)
- illustrations are in C++
  - Principles valid across OO languages!
  - Versions of book exists with illustrations in Java, …

GoF Design Pattern Book

23 Design Patterns

Classification

- according to purpose
- according to problems they solve (p. 24-25)
- according to degrees of freedom (table 1.2, p. 30)

Goal is to make it easy to find a pattern for your problem

Design Pattern Relationships

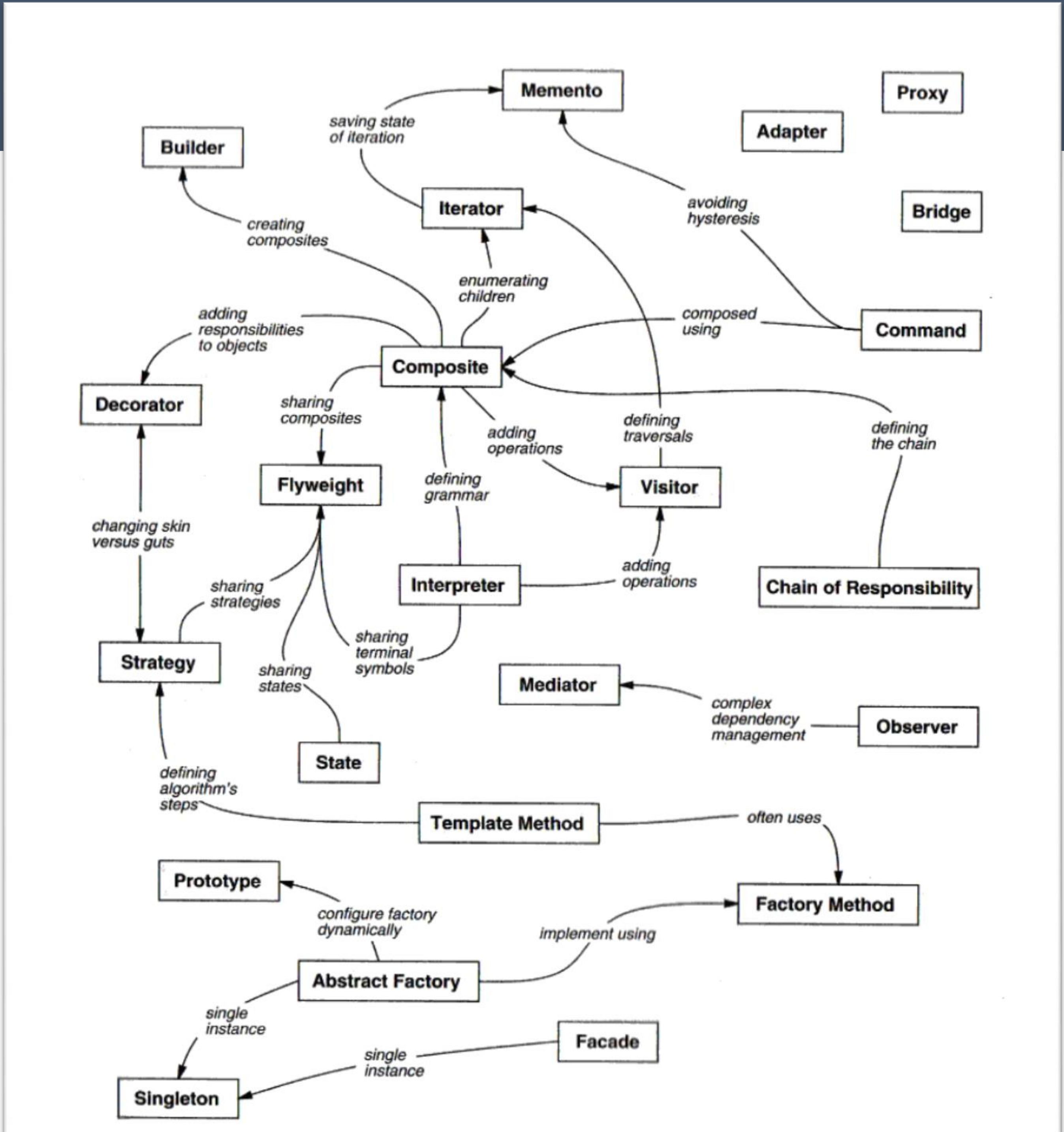


Figure 1.1: Design pattern relationships

Visitor

Visitor

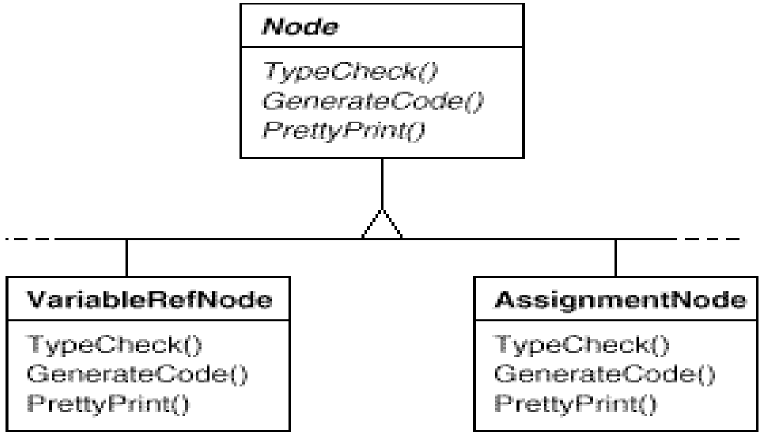
Category

- Behavioral

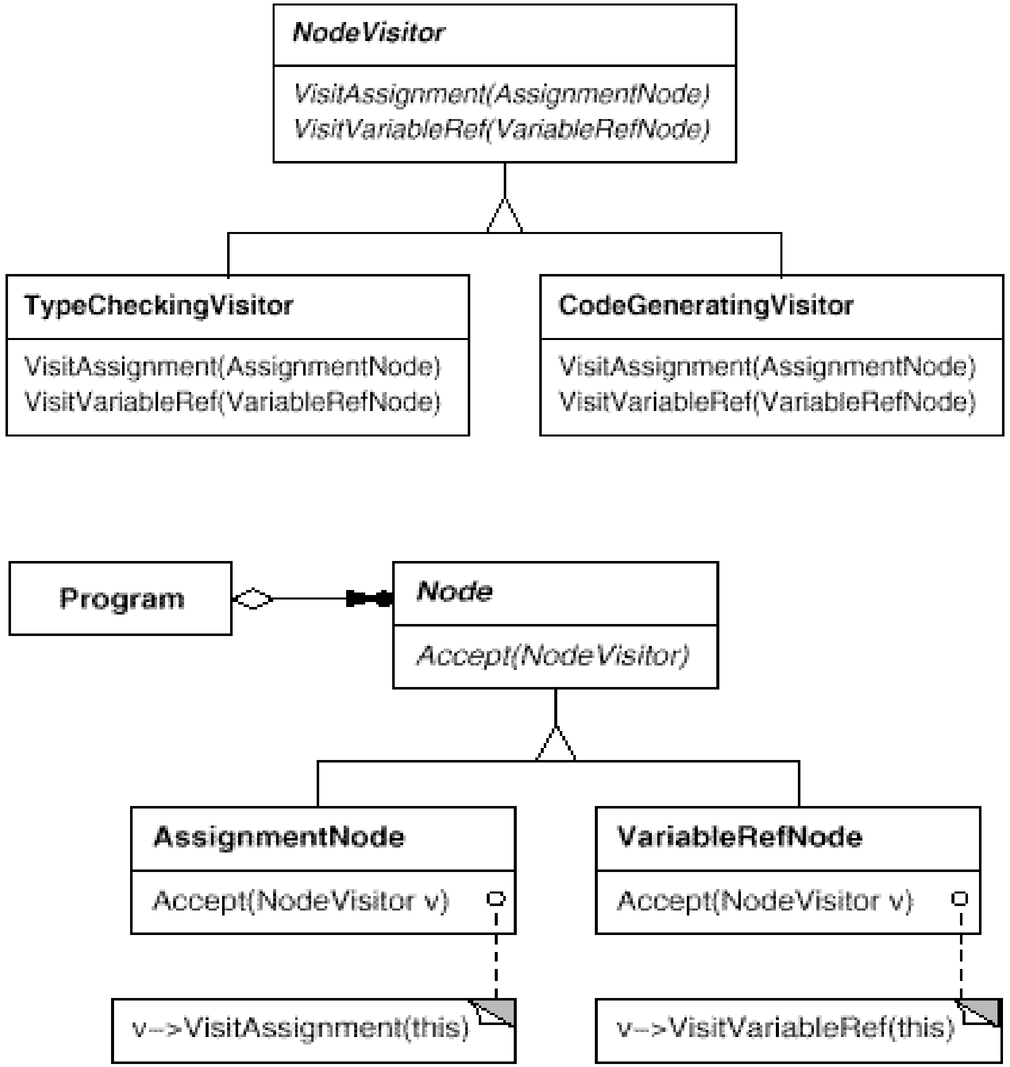
Intent

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation



Motivation (cont)


NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
Program
Node
Accept(NodeVisitor)
AssignmentNode
Accept(NodeVisitor v)
VariableRefNode
Accept(NodeVisitor v)
v->VisitAssignment(this)
v->VisitVariableRef(this)

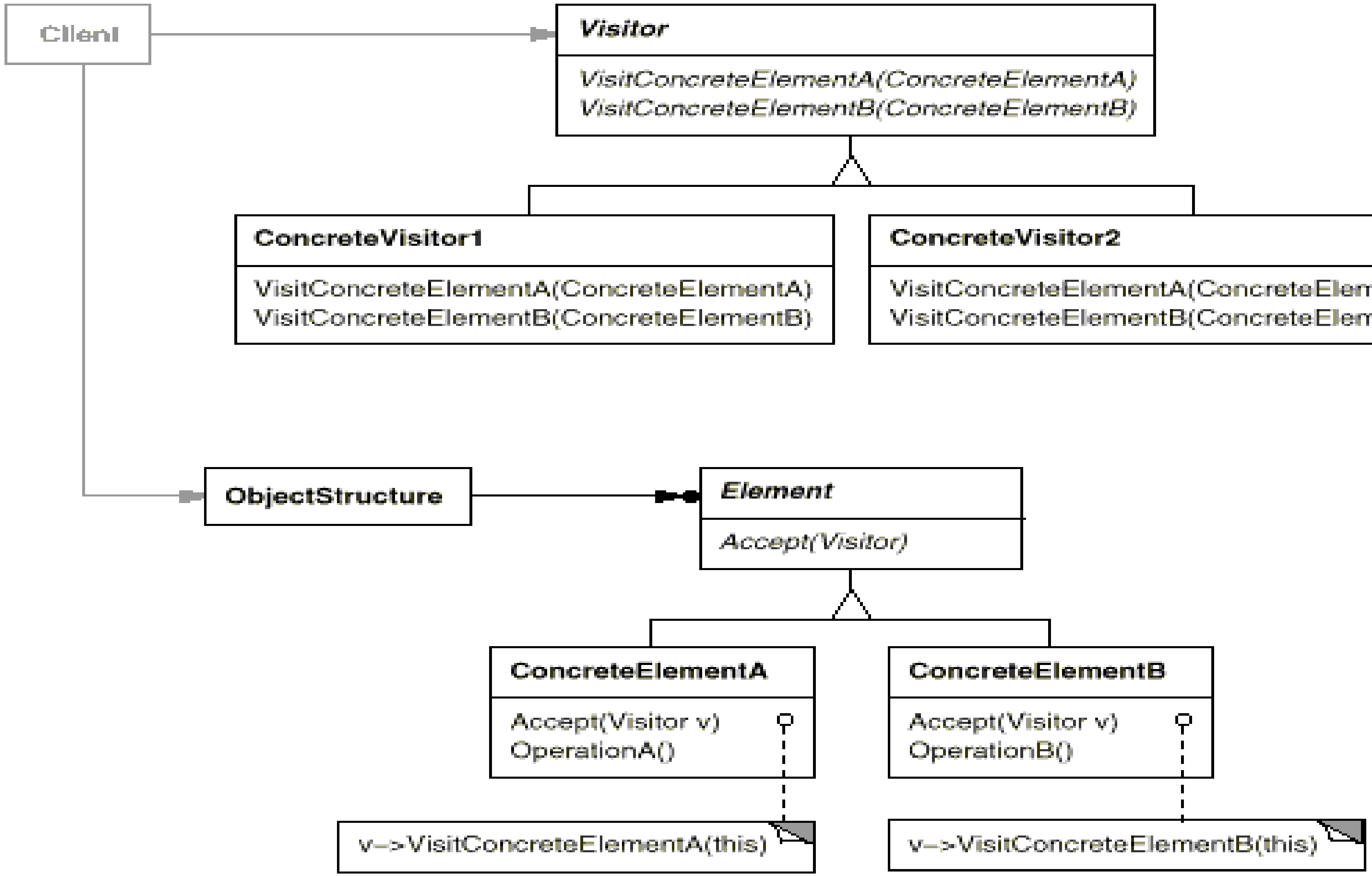
Applicability

An object structure contains many classes of objects with differing interfaces and you want to perform operations on these objects that depend on their concrete classes.

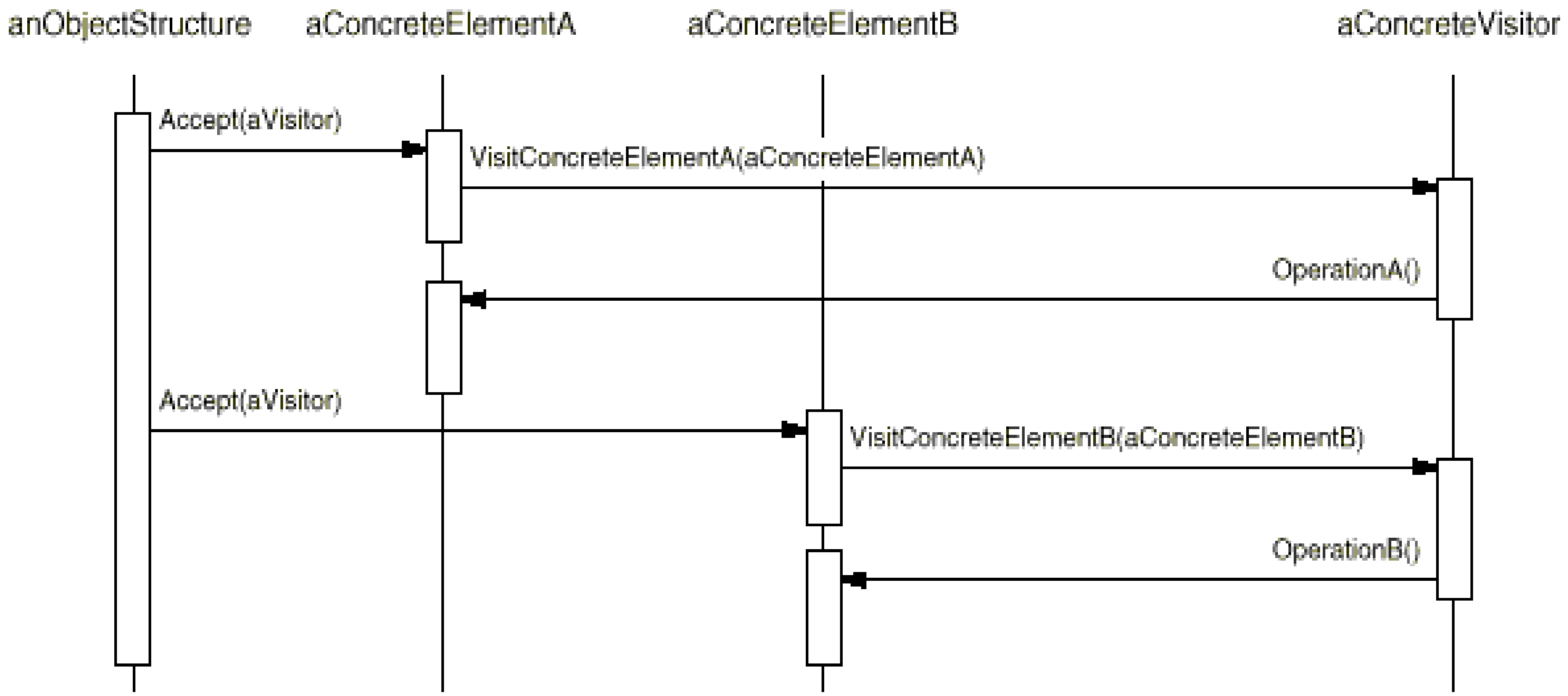
Many distinct and unrelated operations need to be performed on objects in an object structure an you want to avoid “polluting” their classes with these operations.

The classes defining the object structure rarely change but you often want to define new operations over the structure.

Structure


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v–>VisitConcreteElementA(this)
v–>VisitConcreteElementB(this)

Sequence



Cfr. *Double Dispatch* - this is key to this pattern in order to link concrete elements and concrete visitors !

Participants

Visitor

- Declares a Visit operation for each class of ConcreteElement in the object structure.
- The operations name and signature identifies the class that sends the Visit request.

ConcreteVisitor

- Implements each operation declared by Visitor.
- Each operation implements a fragment of the algorithm for the corresponding class of object in the object structure.
- Provides the context for the algorithm and stores its state (often accumulating results during traversal).

…

Participants (cont)

Element

- Defines an accept operation that takes a Visitor as an argument.

ConcreteElement

- Implements an accept operation that takes a visitor as an argument.

ObjectStructure

- Can enumerate its elements.
- May provide a high-level interface to allow the visitor to visit its elements.
- May either be a Composite or a collection such as a list or set.

Collaborations

A client that uses the visitor pattern must create a ConcreteVisitor object and then traverse the object structure visiting each element with the Visitor.

When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation.

Consequences

Makes adding new operations easy.

- a new operation is defined by adding a new visitor (in contrast, when you spread functionality over many classes each class must be changed to define the new operation).

Gathers related operations and separates unrelated ones.

- related behavior is localised in the visitor and not spread over the classes defining the object structure.

Consequences (cont)

Adding new ConcreteElement classes is hard.

- each new ConcreteElement gives rise to a new abstract operation in Visitor and a corresponding implementation in each ConcreteVisitor.

Allows visiting across class hierarchies.

- an iterator can also visit the elements of an object structure as it traverses them and calls operations on them but all elements of the object structure then need to have a common parent. Visitor does not have this restriction.

Consequences (cont)

Accumulating state

- Visitor can accumulate state as it proceeds with the traversal. Without a visitor this state must be passed as an extra parameter of handled in global variables.

Breaking encapsulation

- Visitor’s approach assumes that the ConcreteElement interface is powerful enough to allow the visitors to do their job. As a result the pattern ofthen forces to provide public operations that access an element’s internal state which may compromise its encapsulation.

Known Uses

In the Smalltalk-80 compiler.

In 3D-graphics: when three-dimensional scenes are represented as a hierarchy of nodes, the Visitor pattern can be used to perform different actions on those nodes.

Visitor Pattern

So, we’ve covered the visitor pattern as found in the book

  - Are we done?

Decisions, decisions …

visit(OperationA a)

visit(OperationB b)

vs

visitOperationA(OperationA a)

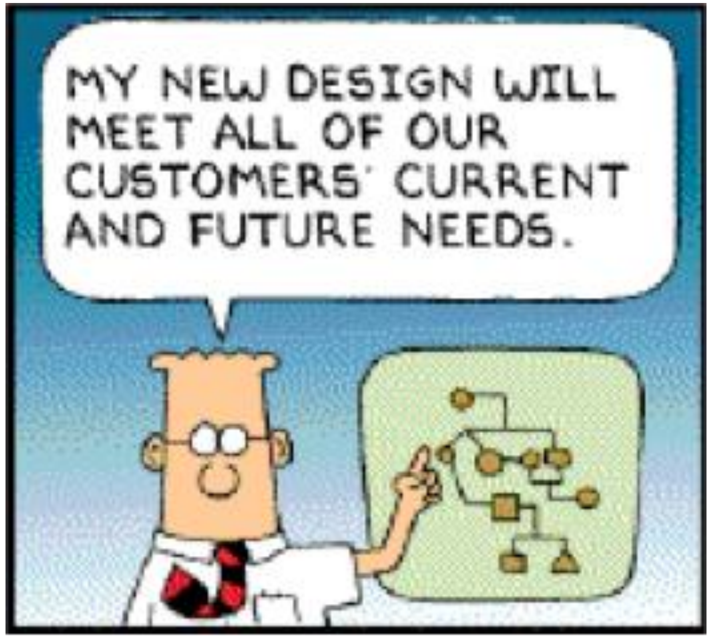
visitOperationB(OperationB b)

Short Feature...

What is the result of the following expression?

```
class A {
    public void m(A a) { System.out.println("1"); }
}

class B extends A {
    public void m(B b) { System.out.println("2"); }
    public void m(A a) { System.out.println("3"); }
}


B b = new B();
A a = b;
a.m(b);
```

Main Feature...


MY NEW DESIGN WILL MEET ALL OF OUR CUSTOMERS' CURRENT AND FUTURE NEEDS.

Visiting all Elements in the CDT Parsetree

```
public abstract class ASTVisitor {
    public int visit(IASTTranslationUnit tu)                        { return PROCESS_CONTINUE; }
    public int visit(IASTName name)                                 { return PROCESS_CONTINUE; }
    public int visit(IASTDeclaration declaration)                   { return PROCESS_CONTINUE; }
    public int visit(IASTInitializer initializer)                   { return PROCESS_CONTINUE; }
    public int visit(IASTParameterDeclaration parameterDeclaration) { return PROCESS_CONTINUE; }
    public int visit(IASTDeclarator declarator)                     { return PROCESS_CONTINUE; }
    public int visit(IASTDeclSpecifier declSpec)                    { return PROCESS_CONTINUE; }
    public int visit(IASTExpression expression)                     { return PROCESS_CONTINUE; }
    public int visit(IASTStatement statement)                       { return PROCESS_CONTINUE; }
    public int visit(IASTTypeId typeId)                             { return PROCESS_CONTINUE; }
    public int visit(IASTEnumerator enumerator)                     { return PROCESS_CONTINUE; }
    public int visit( IASTProblem problem )                         { return PROCESS_CONTINUE; }
}
```

To Arms! The Short Feature is Attacking the Main Feature

Y NEW WILL MEET ALL OF OUR CUSTOMERS' CURRENT AND FUTURE NEEDS.

Advanced Visitor Discussions

When looking more closely at the visitor and its implementation, we can discuss a number of things in more detail:

- Who controls the traversal?
- What is the granularity of the visit methods?
- Does there have to be a one-on-one correspondence between Element classes and visit methods ?
- …

Composite

Composite

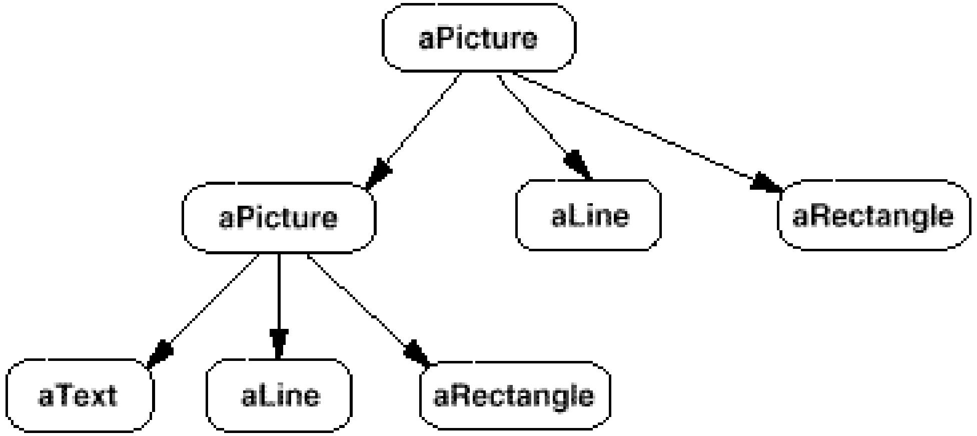
Category

- Structural

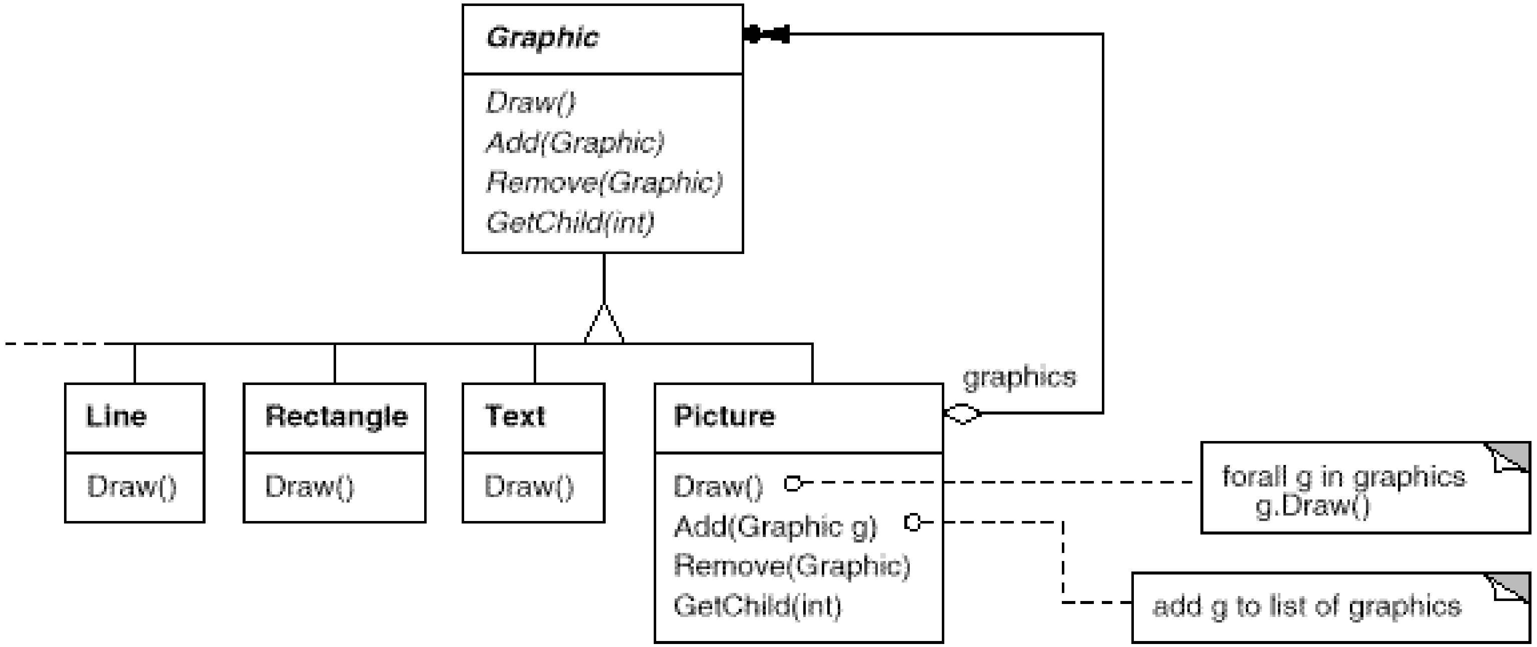
Intent

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation



Motivation (cont)

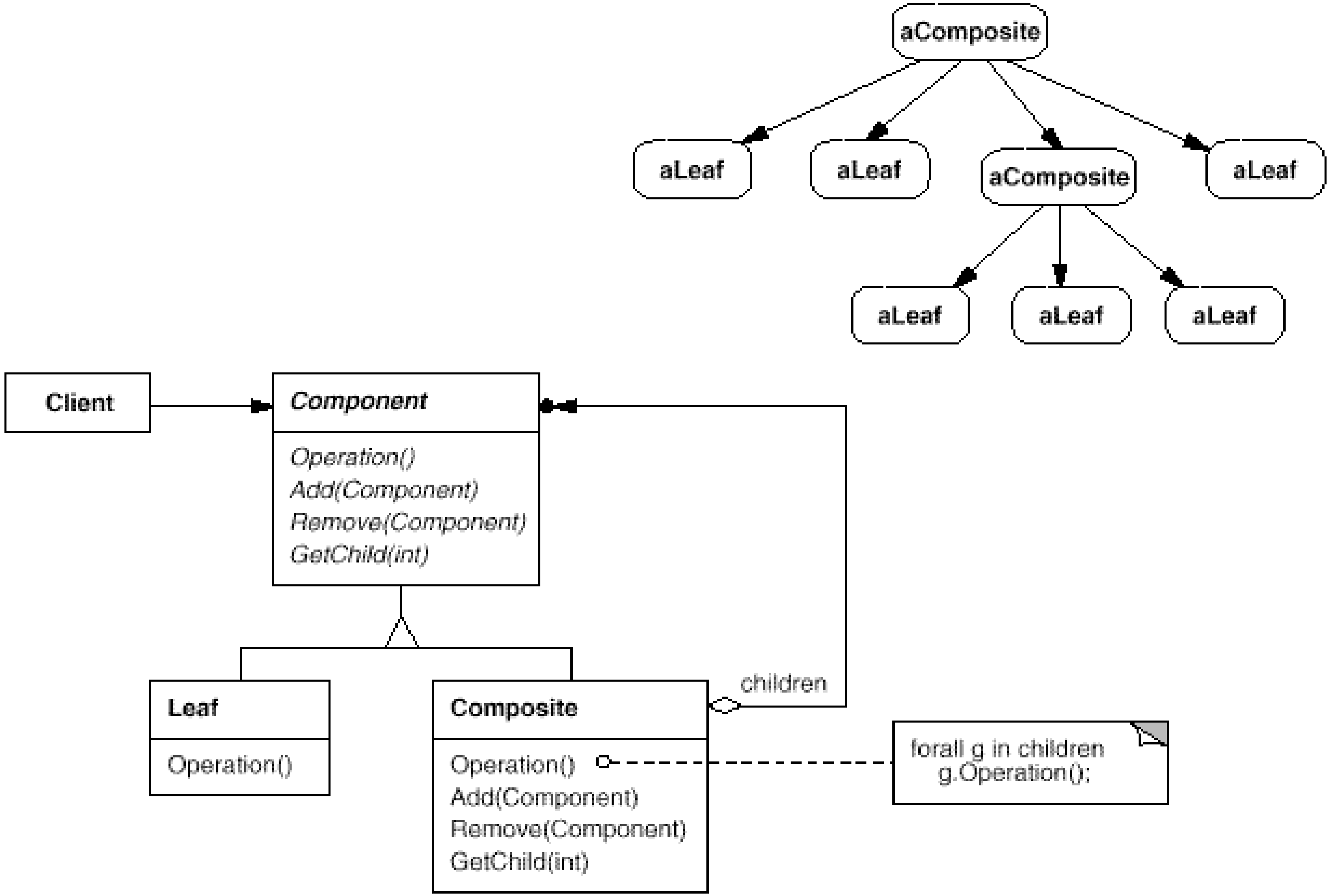


Applicability

Use the Composite Pattern when:

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure


aComposite
aLeaf
aLeaf
aComposite
aLeaf
aLeaf
aLeaf
aLeaf
Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
forall g in children
g.Operation();

Participants

Component

- Declares the interface for objects in the composition.
- Implements default behaviour for the interface common to all classes, as appropriate.
- Declares an interface for accessing and managing its child components.

Leaf

- Represents leaf objects in the composition. A leaf has no children.
- Defines behaviour for primitive objects in the composition.

Participants (cont)

Composite

- defines behaviour for components having children.
- stores child components.
- implements child-related operations in the Component interface.

Client

- manipulates objects in the composition through the Component interface.

Collaborations

Clients use the Component class interface to interact with objects in the composite structure. Leaves handle the requests directly. Composites forward requests to its child components.

Consequences

Defines class hierarchies consisting of primitive and composite objects.

Makes the client simple. Composite and primitive objects are treated uniformly (no cases).

Eases the creation of new kinds of components.

Can make your design overly general.

Known Uses

Can be found in almost all object oriented systems.

The original View class in Smalltalk Model / View / Controller was a composite.

Questions

How does the Composite pattern help to consolidate system-wide conditional logic?

Would you use the composite pattern if you did not have a part-whole hierarchy? In other words, if only a few objects have children and almost everything else in your collection is a leaf (a leaf that has no children), would you still use the composite pattern to model these objects?

Patterns Catalogue

- Command
- Decorator
- Strategy
- Factory Method
- Abstract Factory
- Singleton
- Proxy
- Adapter
- Observer
- Chain of Responsibility
- FlyWeight
- Facade

See Design pattern books,
Or the patterns reference on website

Wrap-up

Architectures

"can't be made, but only generated, indirectly, by the ordinary actions of the people, just as a flower cannot be made, but only generated from the seed." (Alexander)

- patterns describe such building blocks
- applying them implicitly changes the overall structure (architecture)
- whether it is on classes, components, or people

Conclusion

Can you answer this?

- How does Strategy improve coupling and cohesion?
- Does Abstract Factory says the same than the Creator GRASP Pattern?
- Can you give examples of patterns that can be used together ?
- When does it make sense to combine the Iterator and the Composite Pattern ?

Design of Software Systems
(Ontwerp van SoftwareSystemen)

4 Metrics and Software Visualization

Roel Wuyts
2016-2017

Acknowledgements

Slides adopted, with permission, from Prof. Dr. Michele Lanza

http://www.inf.unisi.ch/faculty/lanza/

You cannot control what you cannot measure

Tom de Marco

Metrics are functions that assign numbers to products, processes and resources

Software metrics are measurements which relate to software systems, processes or related documents

Metrics compress system properties and traits into numbers

Let’s see some examples..

Examples of size metrics

- NOM - Number of Methods
- NOA - Number of Attributes
- LOC - Number of Lines of Code
- NOS - Number of Statements
- NOC - Number of Children

Chidamber & Kemerer, 1994
Lorenz & Kidd, 1994

Cyclomatic Complexity (CYCLO)

- The McCabe cyclomatic complexity (CYCLO) counts the number of independent paths through the code of a function
  - Good: it reveals the minimum number of tests to write
  - Bad: its interpretation does not directly lead to improvement actions

McCabe, 1976

Weighted Method Count (WMC)

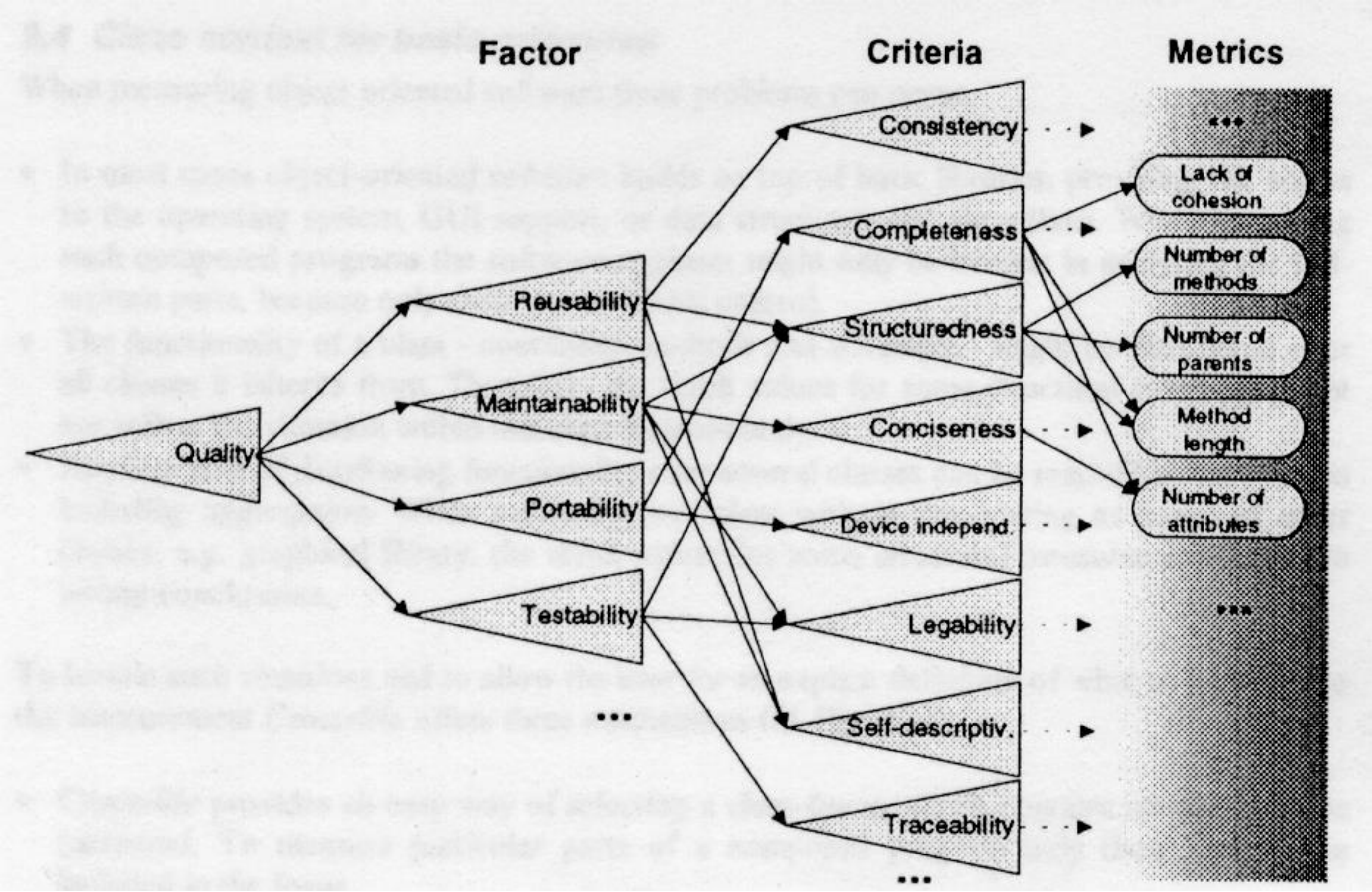
- WMC sums up the complexity of a class’ methods (measured by the metric of your choice, usually CYCLO)
  - Good: It is configurable, thus adaptable to our precise needs
  - Bad: Its interpretation does not directly lead to improvement actions

Chidamber & Kemerer, 1994

Coupling Between Objects (CBO)

- CBO shows the number of classes from which methods or attributes are used.
  - Good: CBO takes into account real dependencies, not just declared ones
  - Bad: No differentiation of types and/or intensity of coupling

Chidamber & Kemerer, 1994



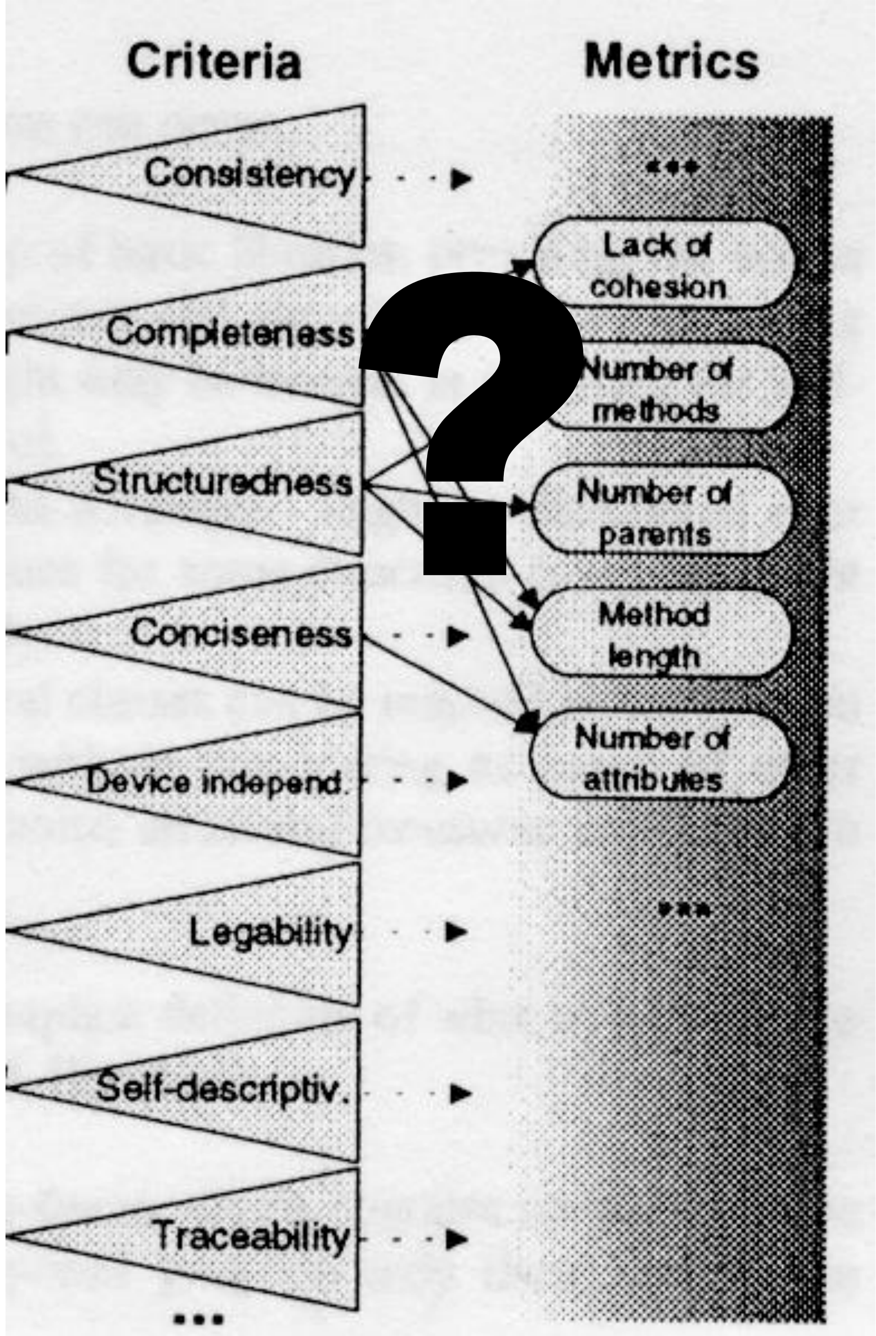
McCall, 1977
Boehm, 1978

Metrics help to assess and improve quality!

Do they?

Problems..

- Metrics granularity
  - metrics capture symptoms,not causes of problems
  - in isolation, metrics do not lead to improvement actions
- Implicit Mapping
  - we do not reason in terms of metrics, but in terms of design (principles)



McCall, 1977
Boehm, 1978

2 big obstacles in using metrics:

Thresholds make metrics hard to interpret

Granularity makes metrics hard to use in isolation

How do I get an
initial understanding of a system?

Metric	Value
LOC	35175
NOM	3618
NOC	384
CYCLO	5579
NOP	19
CALLS	15128
FANOUT	8590
AHH	0,12
ANDC	0,31

Metric	Value
LOC	35175
NOM	3618
NOC	384
CYCLO	5579
NOP	
CALLS	
FA	8590
A	0,12
ANDC	0,31

And now what?

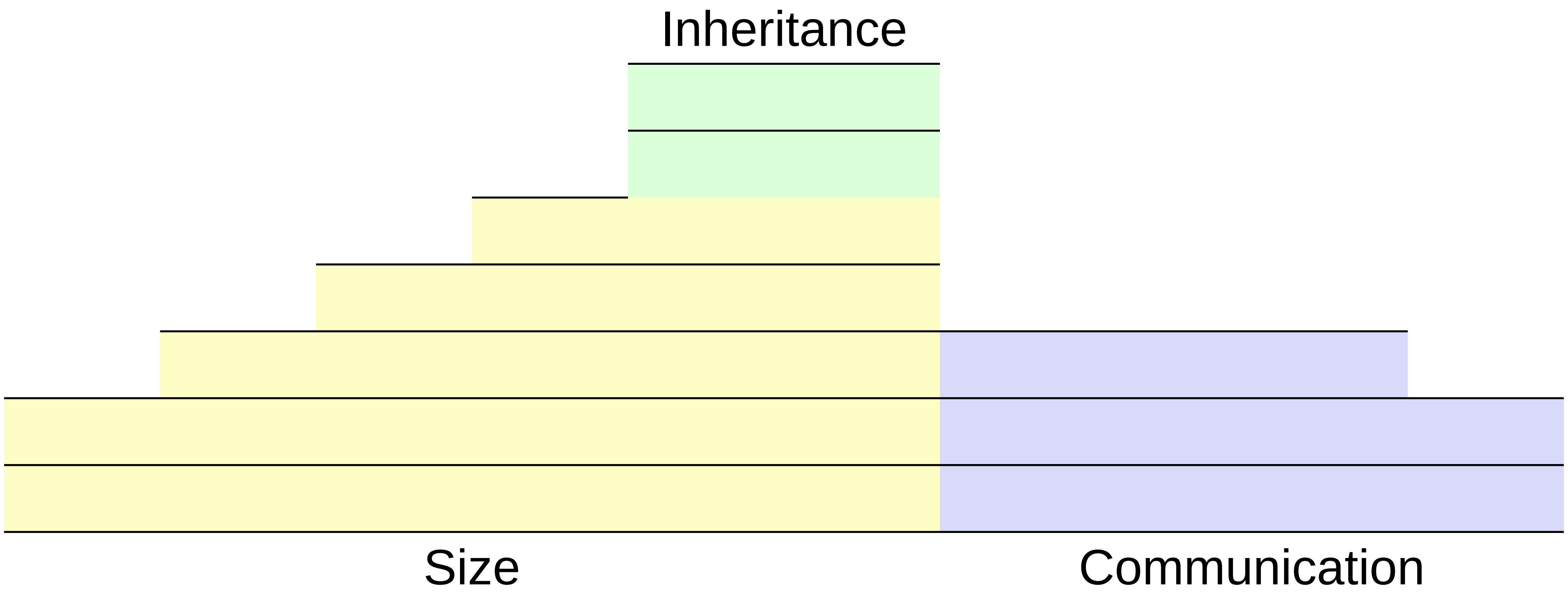
coupling?

We need means to compare

hierarchies?

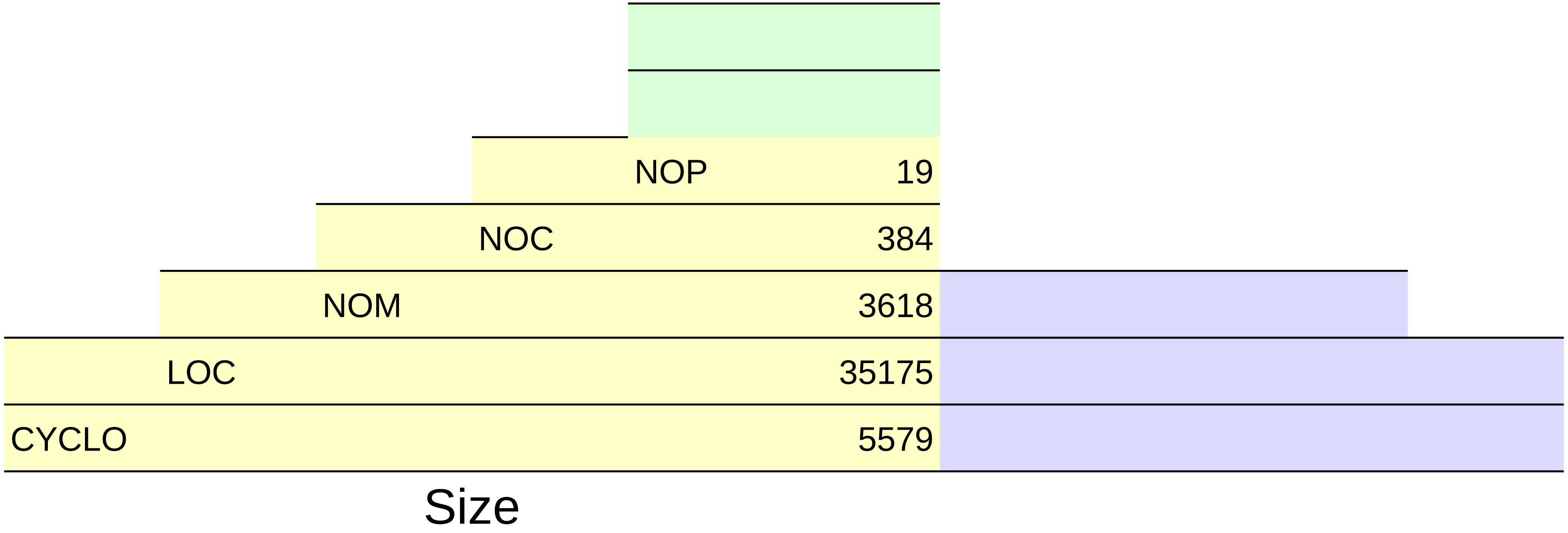
Characterizing Systems with Metrics

The Overview Pyramid provides a metrics overview



Lanza & Marinescu, 2006

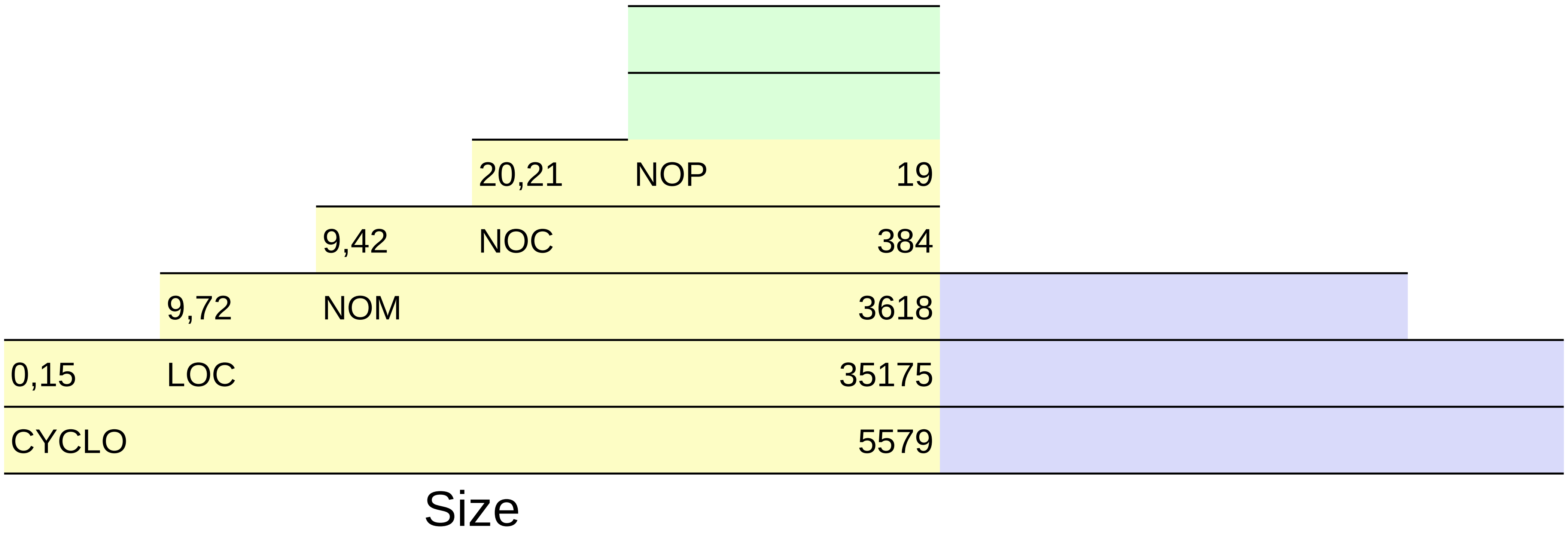
The Overview Pyramid provides a metrics overview



NOP	19
NOC	384
NOM	3618
LOC	35175
CYCLO	5579

Size

The Overview Pyramid provides a metrics overview



20,21 NOP 19

9,42 NOC 384

9,72 NOM 3618

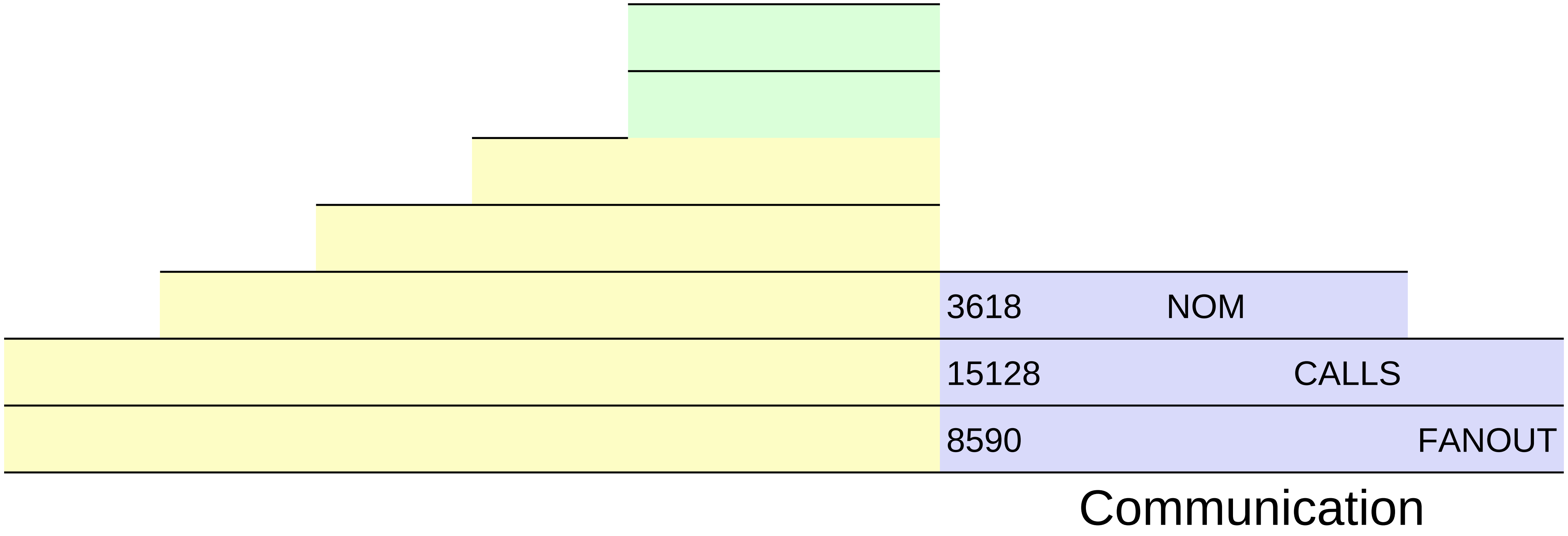
0,15 LOC 35175

CYCLO 5579

Size

Make relative: CYCLO / LOC = 0,15, etc.

The Overview Pyramid provides a metrics overview



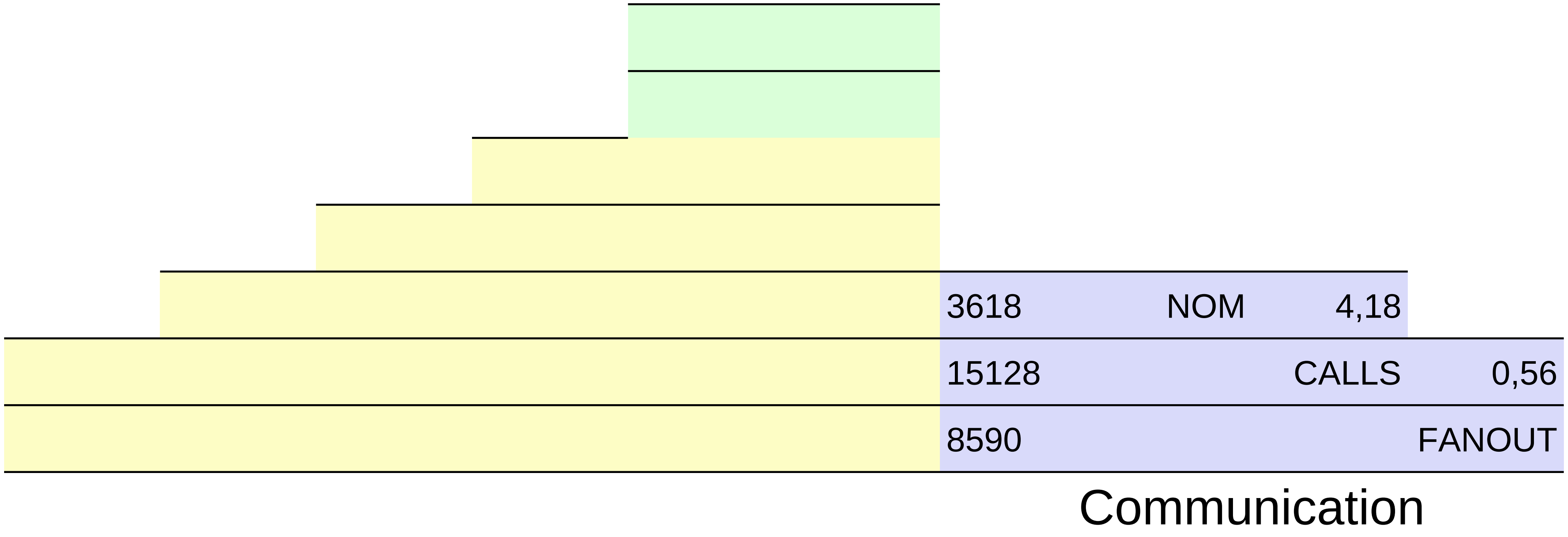
3618 NOM

15128 CALLS

8590 FANOUT

Communication

The Overview Pyramid provides a metrics overview



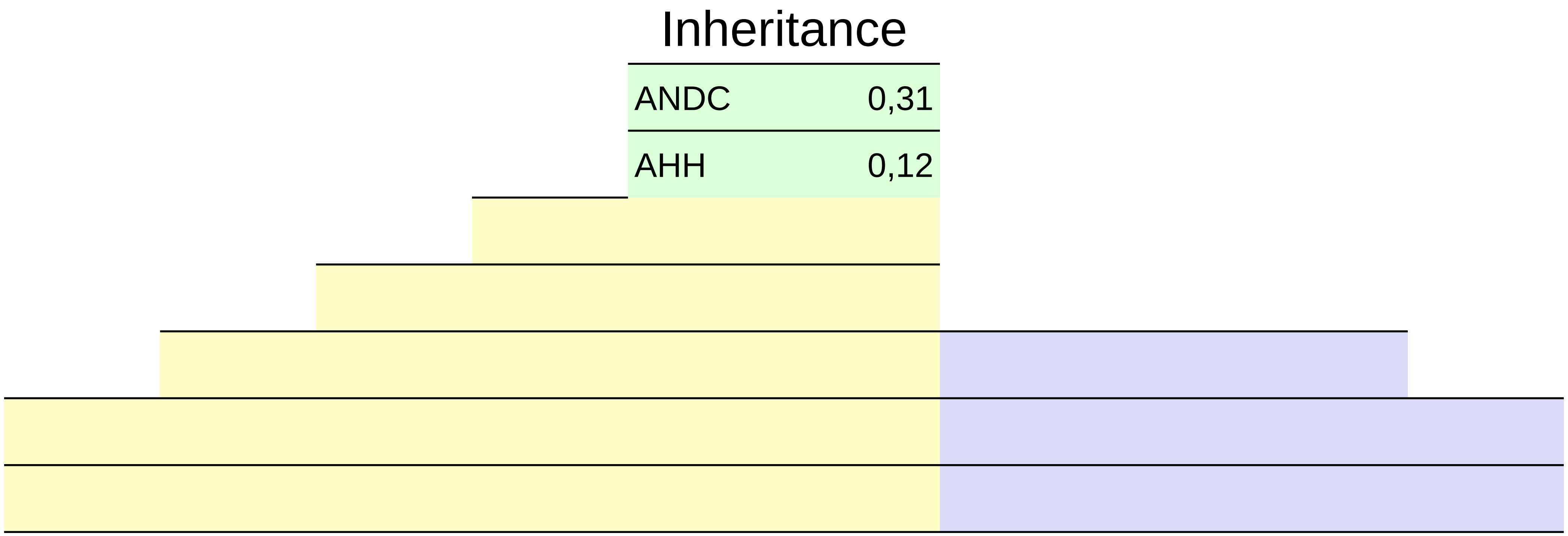
3618 NOM 4,18

15128 CALLS 0,56

8590 FANOUT

Communication

The Overview Pyramid provides a metrics overview

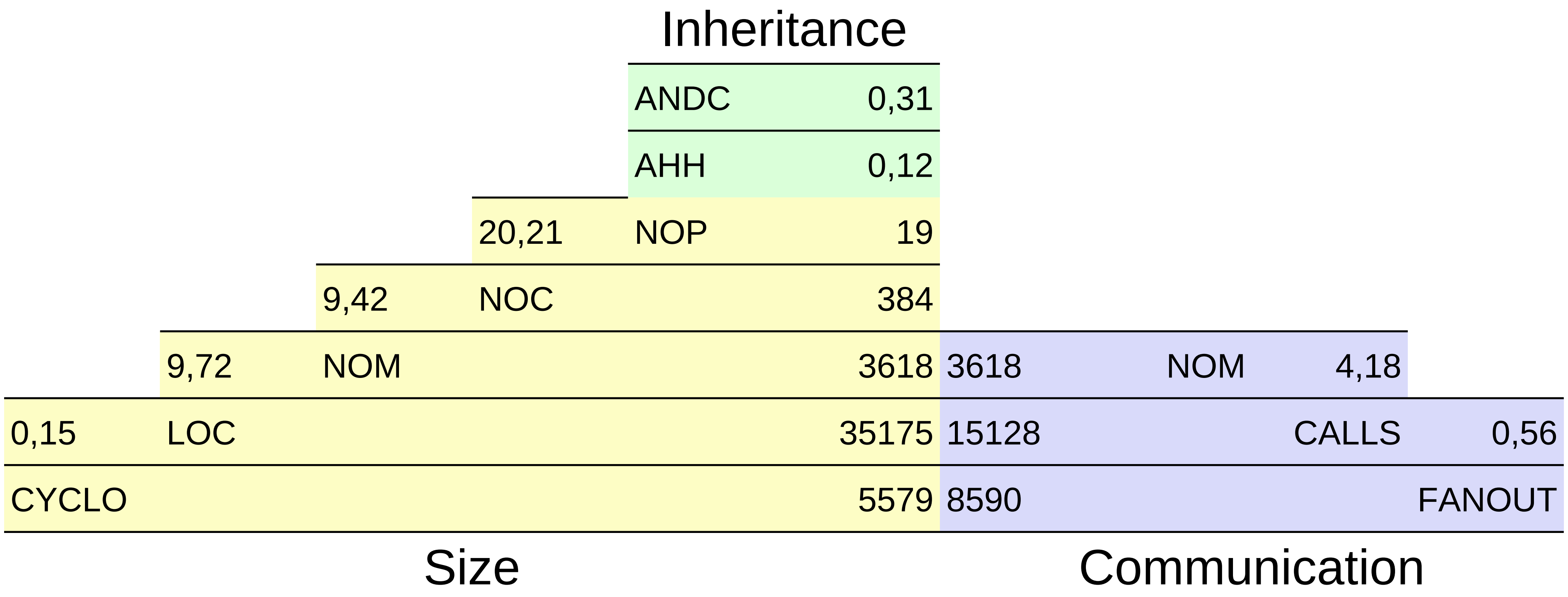


Inheritance

ANDC 0,31

AHH 0,12

The Overview Pyramid provides a metrics overview



Inheritance

		ANDC 0,31			
		AHH 0,12			
	20,21	NOP 19			
	9,42	NOC 384			
9,72	NOM	3618	3618	NOM	4,18
0,15	LOC	35175	15128	CALLS	0,56
CYCLO		5579	8590	FANOUT	

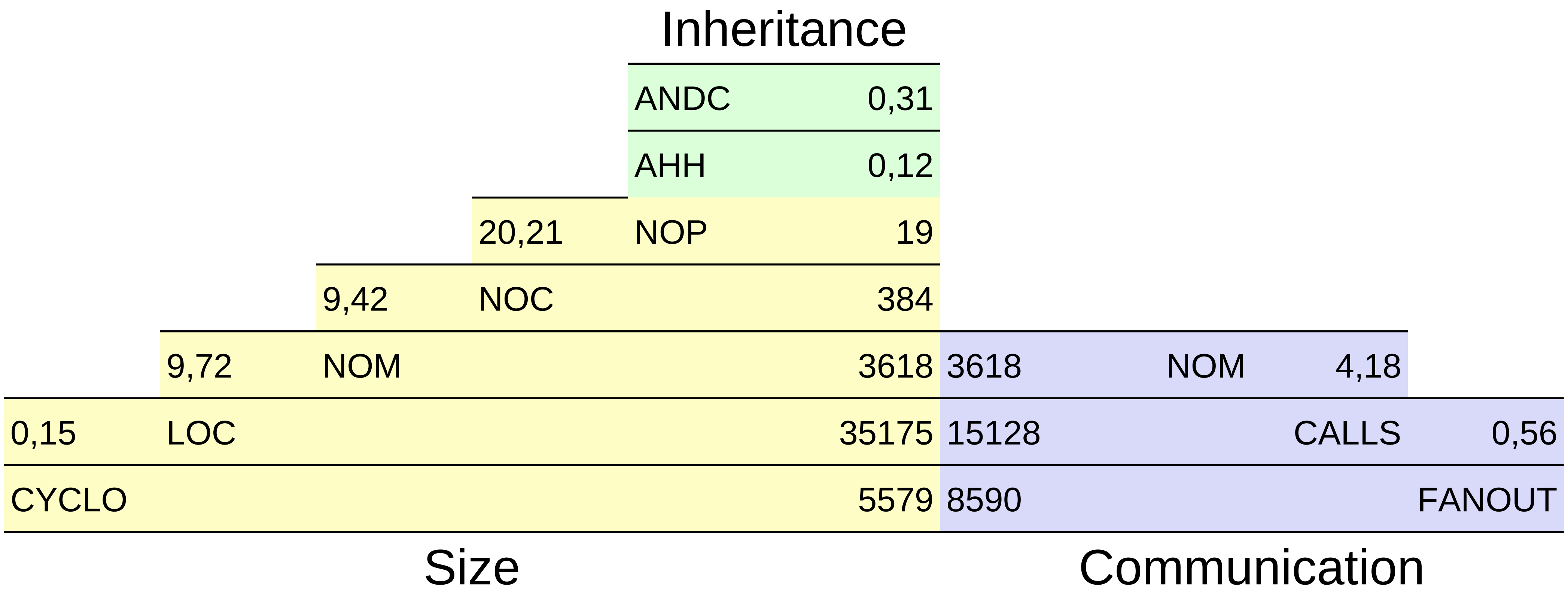
Size

Communication

Obtaining Thresholds

	Java			C++			php		
	LOW	AVG	HIGH	LOW	AVG	HIGH	LOW	AVG	HIGH
CYCLO/ LOC	0,16	0,2	0,24	0,2	0,25	0,3	0.16	0.20	0.24
LOC/N OM	7	10	13	5	10	16	7	10	13
NOM/N OC	4	7	10	4	9	15	4	7	10
...									

The Overview Pyramid provides a metrics overview



Inheritance

ANDC 0,31

AHH 0,12

20,21 NOP 19

9,42 NOC 384

9,72 NOM 3618 3618 NOM 4,18

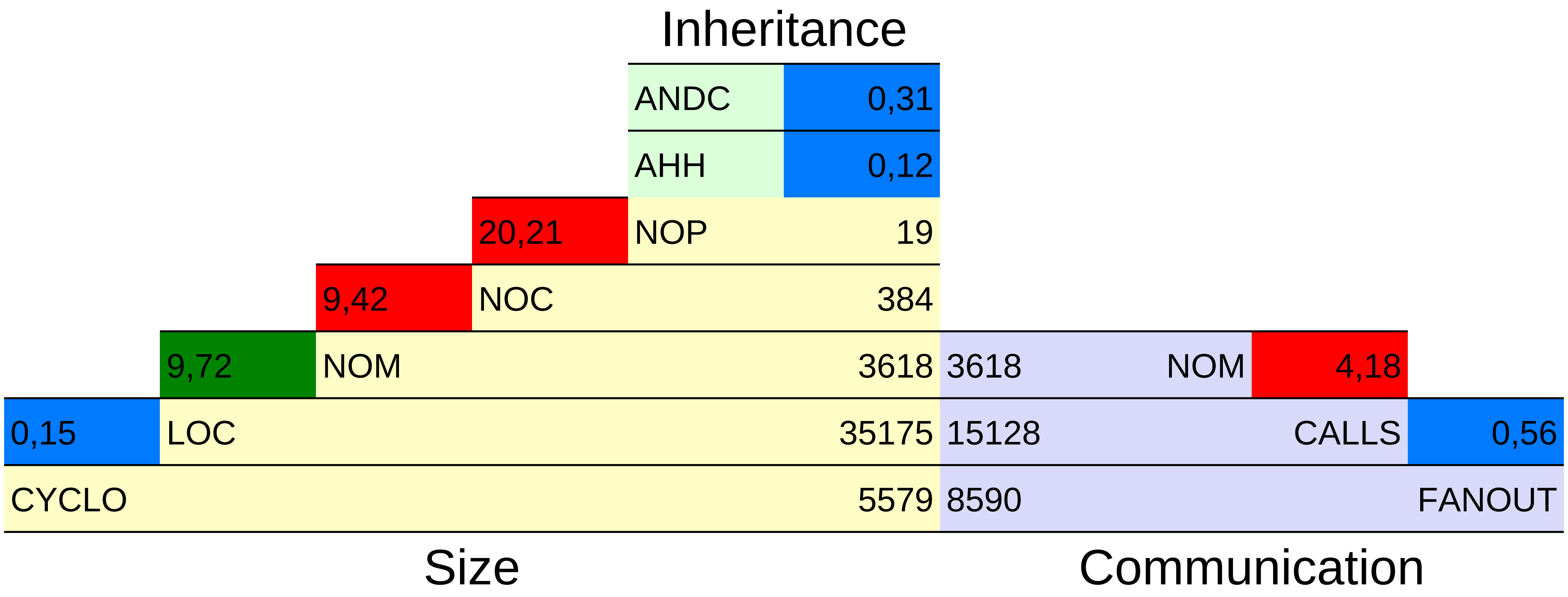
0,15 LOC 35175 15128 CALLS 0,56

CYCLO 5579 8590 FANOUT

Size

Communication

The Overview Pyramid provides a metrics overview



Inheritance

				ANDC	0,31				
				AHH	0,12				
			20,21	NOP	19				
		9,42	NOC		384				
	9,72	NOM			3618	3618	NOM	4,18	
0,15	LOC				35175	15128		CALLS	0,56
CYCLO					5579	8590			FANOUT

Size

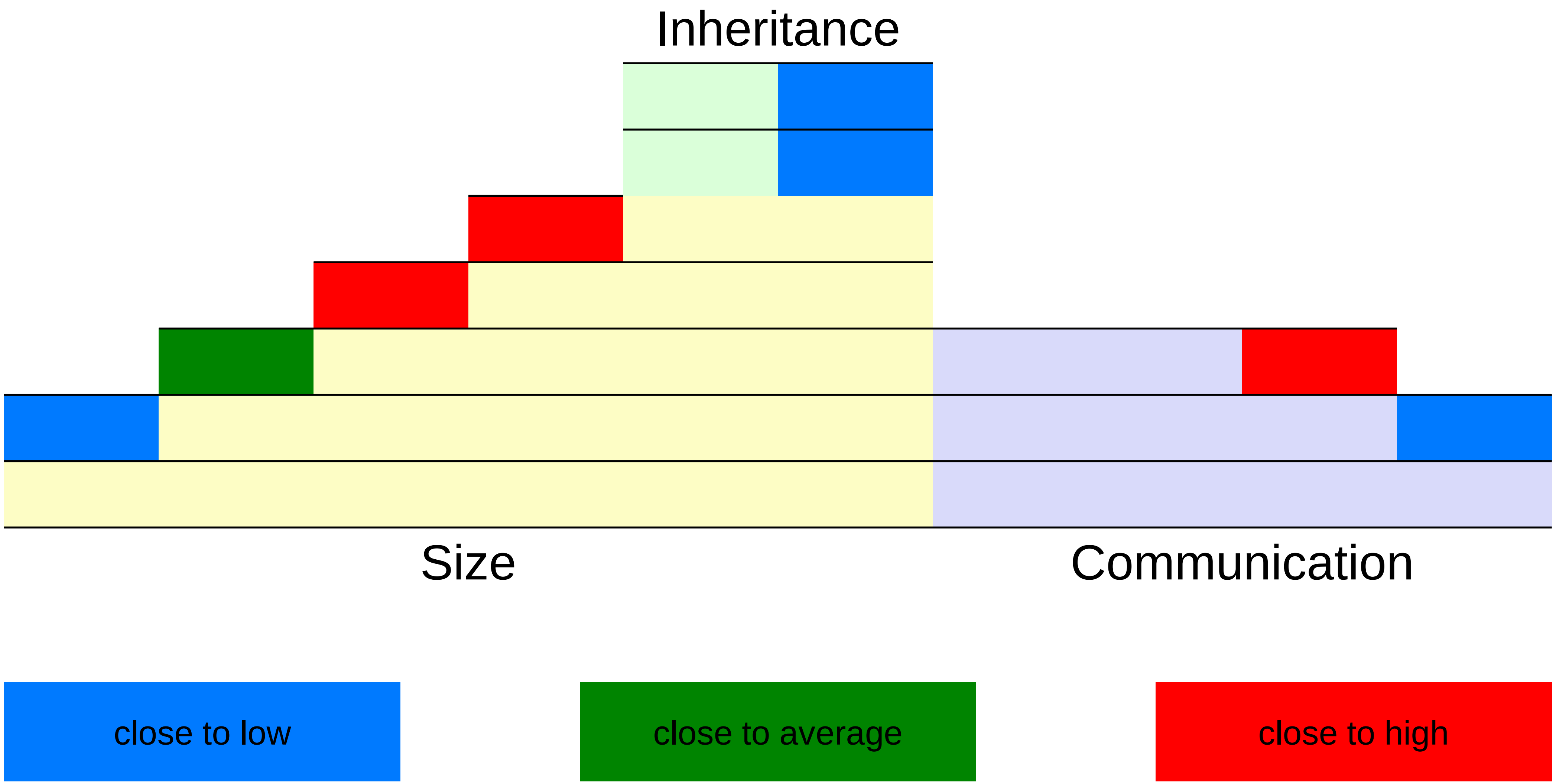
Communication

close to low

close to average

close to high

The Overview Pyramid provides a metrics overview



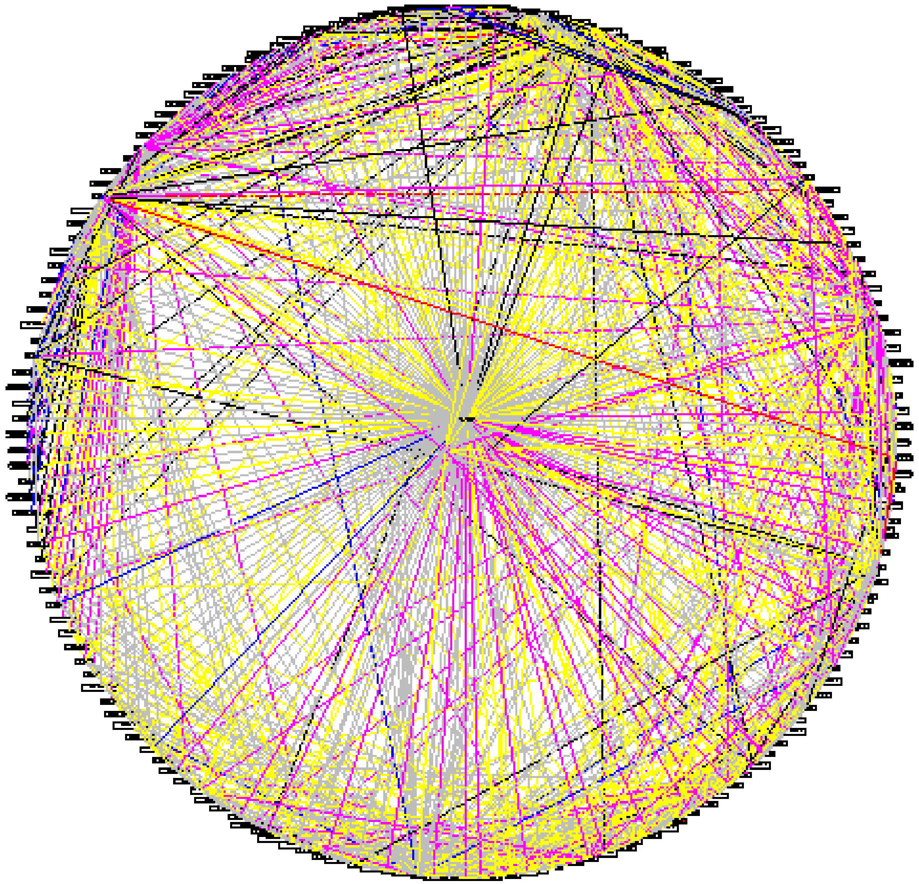
How do I improve my code?

- Quality is more than zero bugs
- Quality is about design principles, design heuristics, and best practices
- Breaking them leads to
  - Code deterioration
  - Design problems ~ Maintenance problems

Imagine...

You change a small design fragment...

...and one third of all classes require changes!



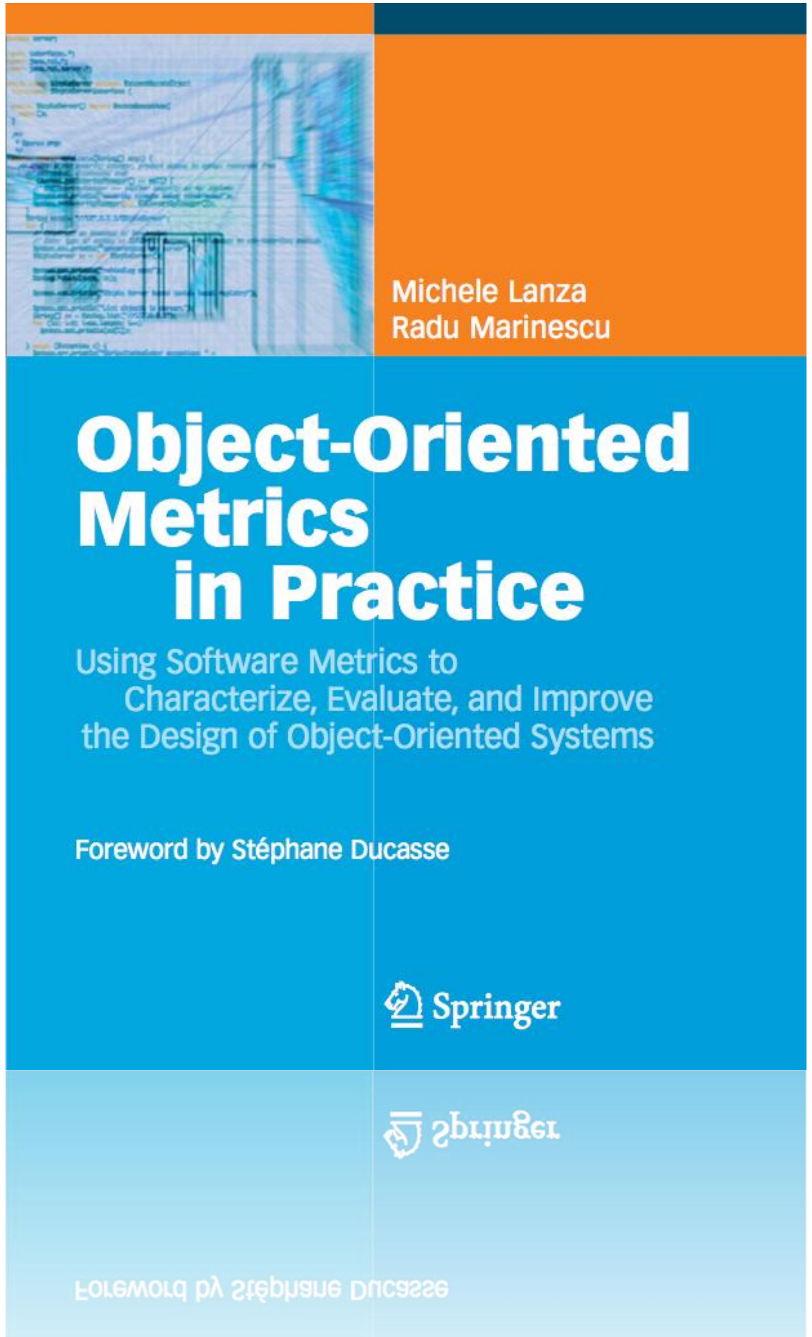
Design Problems

- Expensive
- Frequent
- Unavoidable
- How can we detect and eliminate them?

Reference

M. Lanza, R. Marinescu
“Object-Oriented Metrics in Practice”

Springer, 2006
ISBN 3-540-24429-8



Michele Lanza
Radu Marinescu

Object-Oriented Metrics in Practice

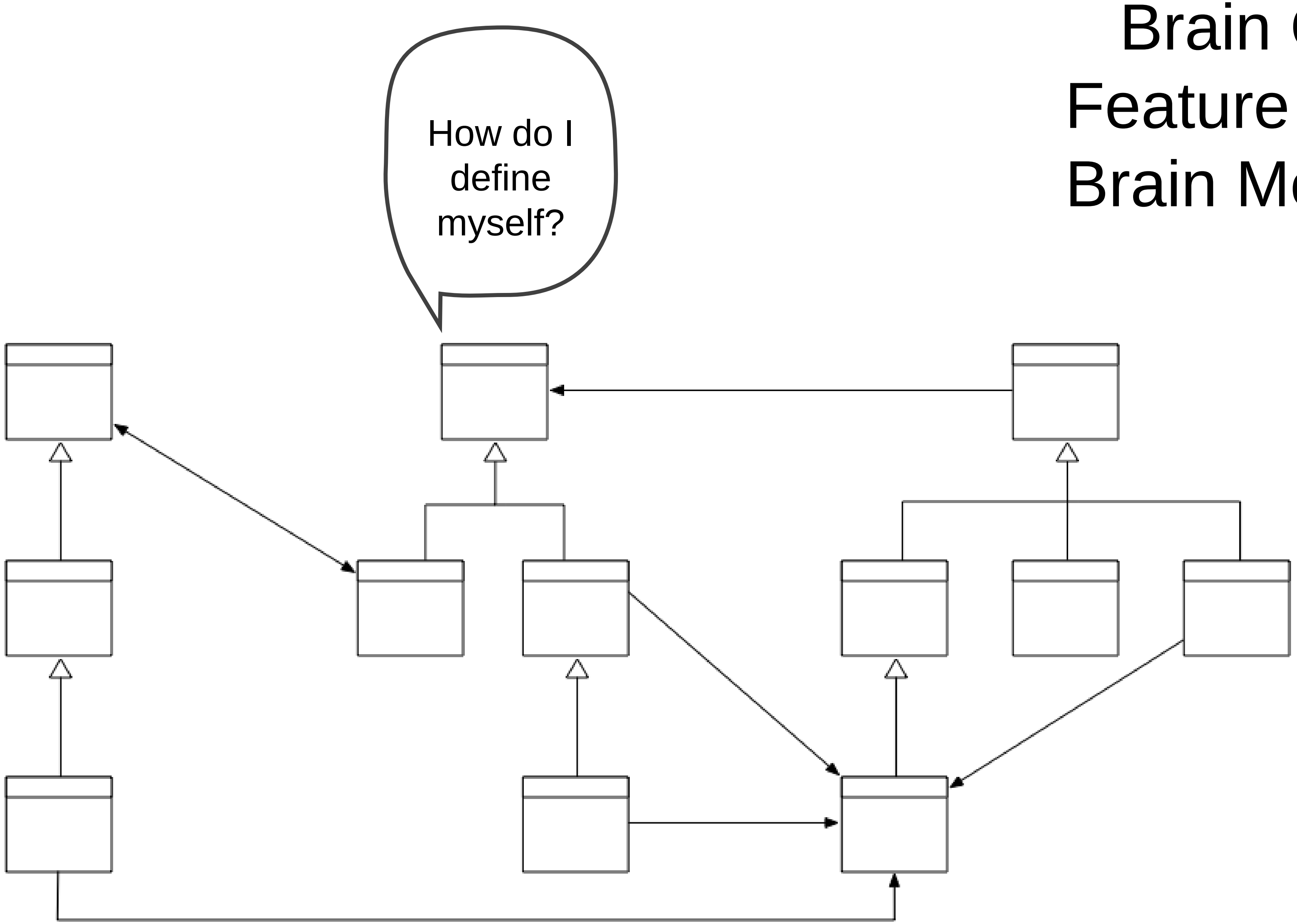
Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems

Foreword by Stéphane Ducasse

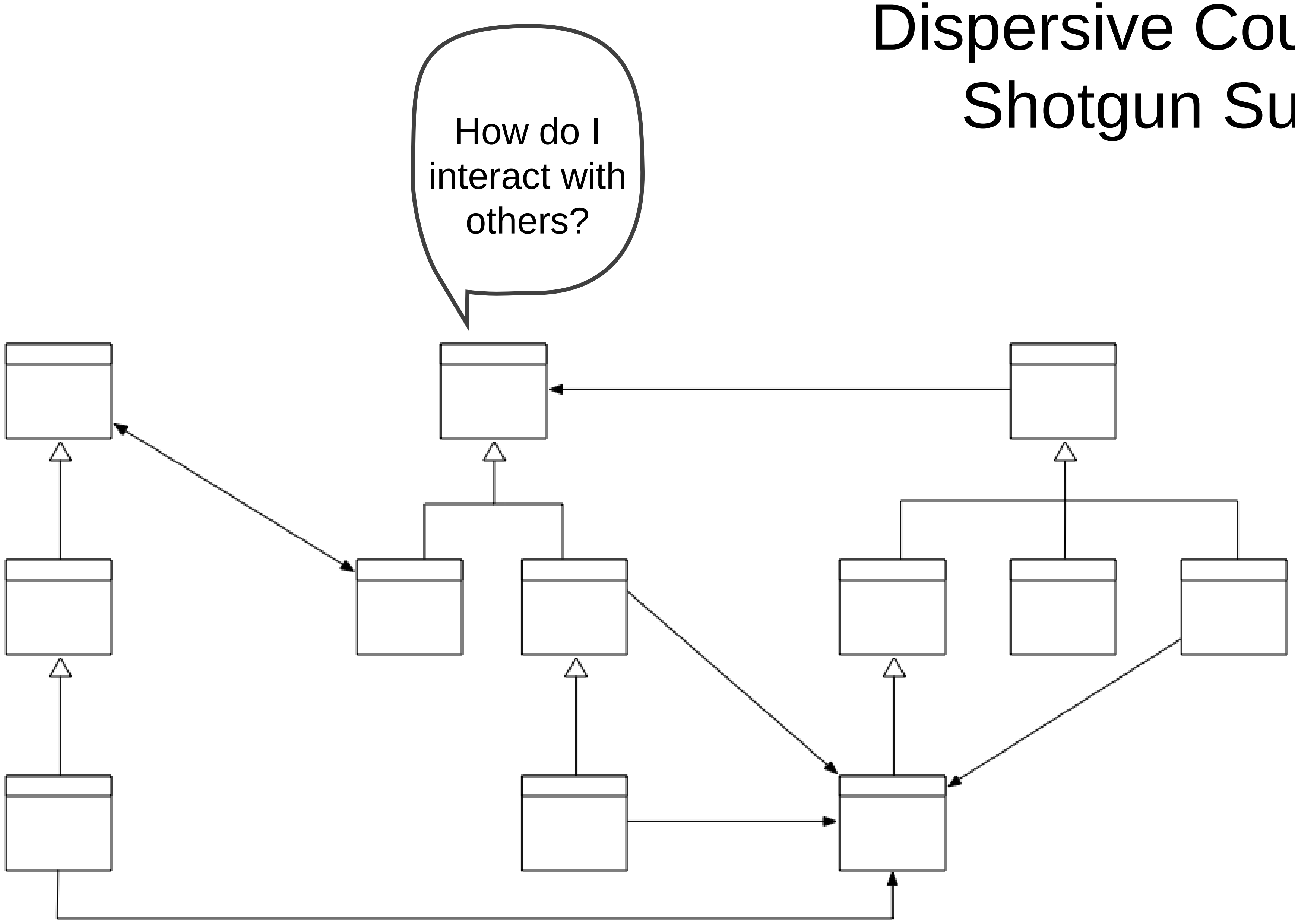
Springer

Identity Disharmony

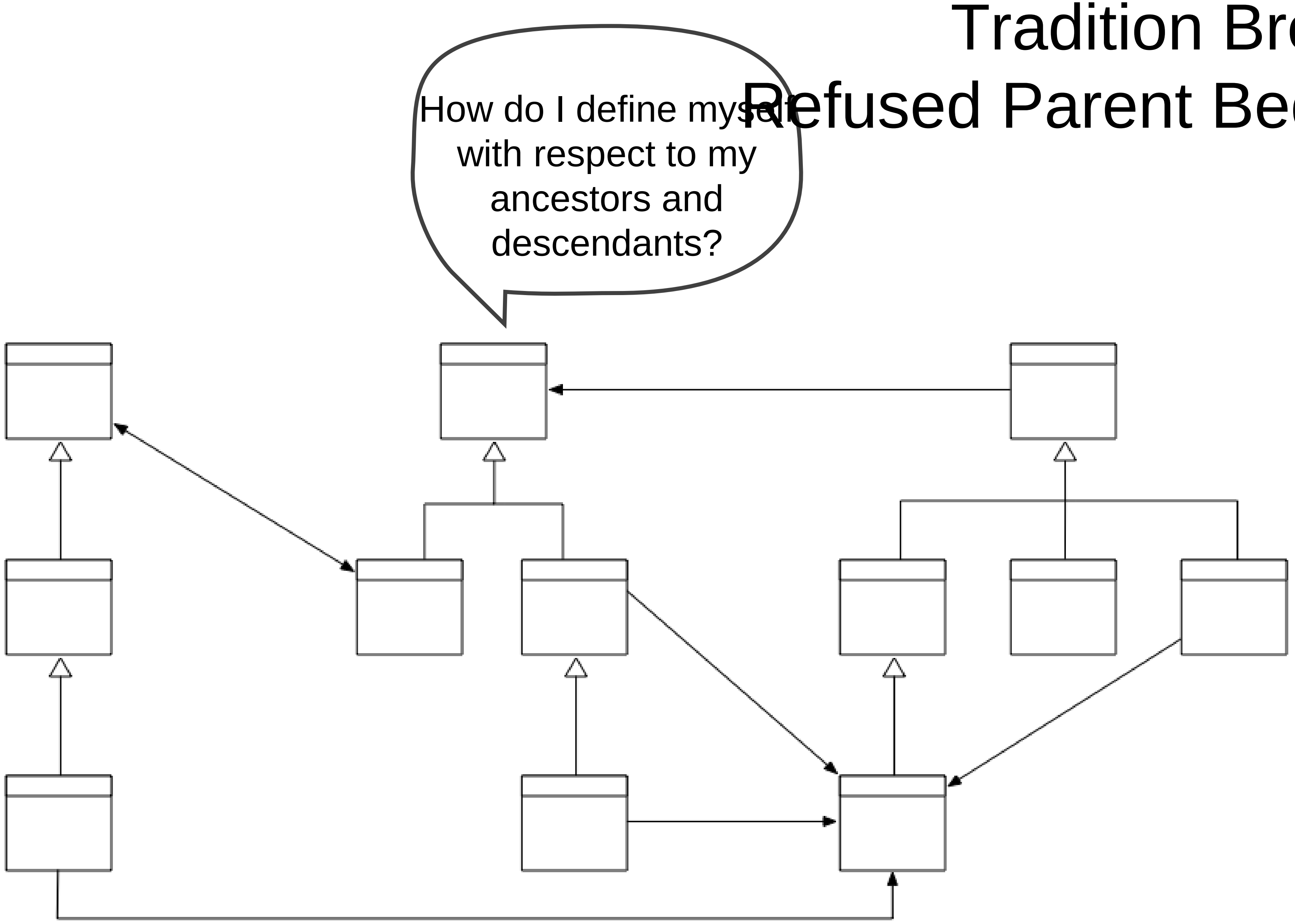
God Class
Data Class
Brain Class
Feature Envy
Brain Method



Intensive Coupling
Dispersive Coupling
Shotgun Surgery



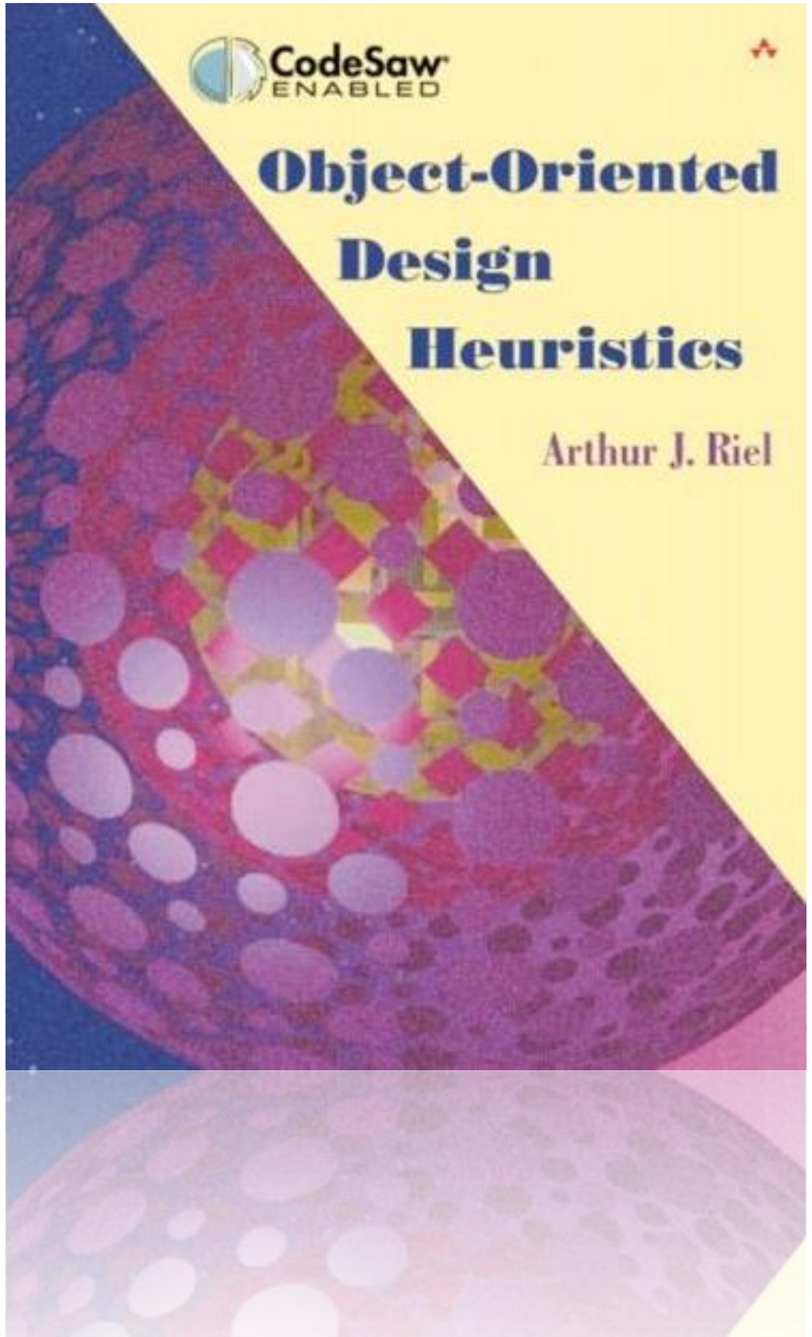
Futile Hierarchy
Tradition Breaker
Refused Parent Bequest


How do I define myself with respect to my ancestors and descendants?

God Class

“In a good object-oriented design
the intelligence of a system is uniformly distributed among the top-level classes.”

Arthur Riel, 1996



God Classes

- God Classes tend to centralize the intelligence of the system, to do everything and to use data from small data-classes
- God Classes tend
  - to centralize the intelligence of the system
  - to do everything and
  - to use data from small data-classes
- God Classes
  - centralize the intelligence of the system
  - do everything
  - use data from small data-classes

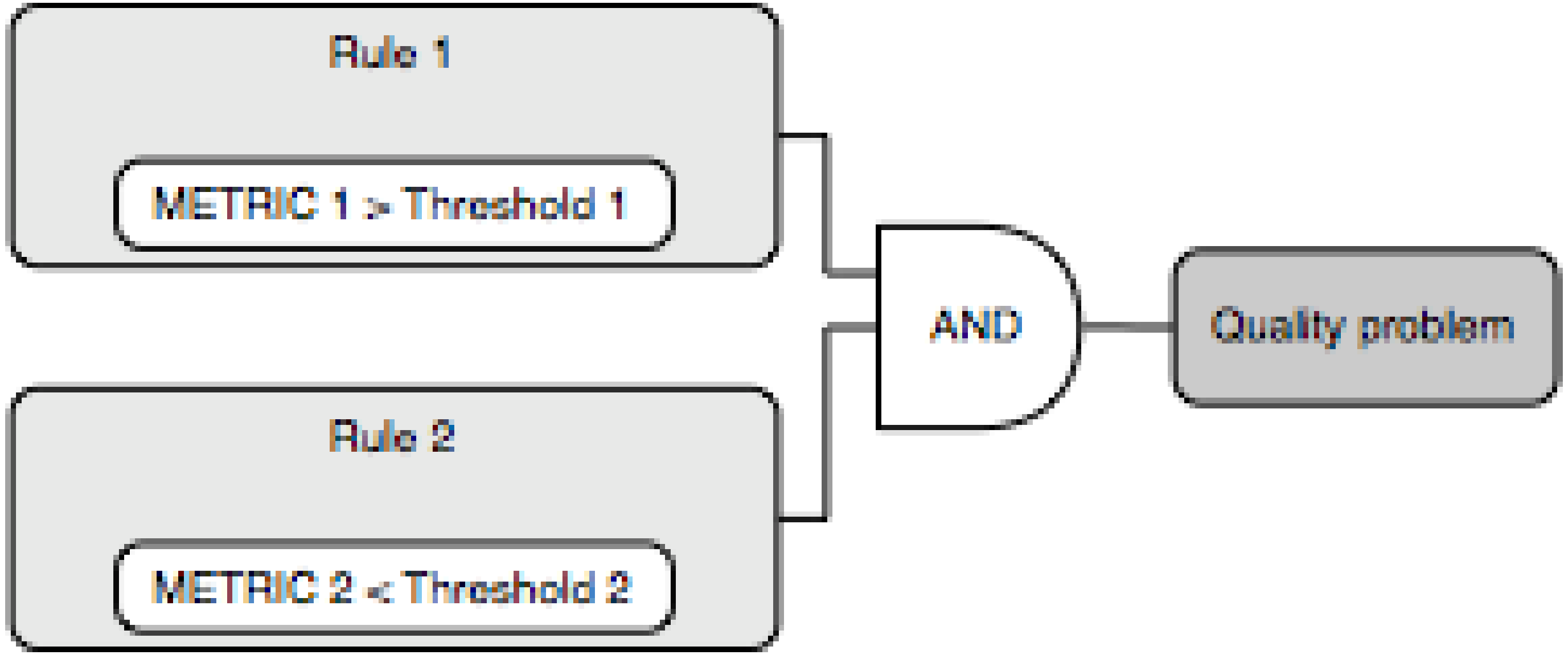
God Classes

- God Classes
  - centralize the intelligence of the system
  - do everything
  - use data from small data-classes
- God Classes
  - are complex: high WMC
  - are not cohesive: low TCC
  - access external data: ATFD

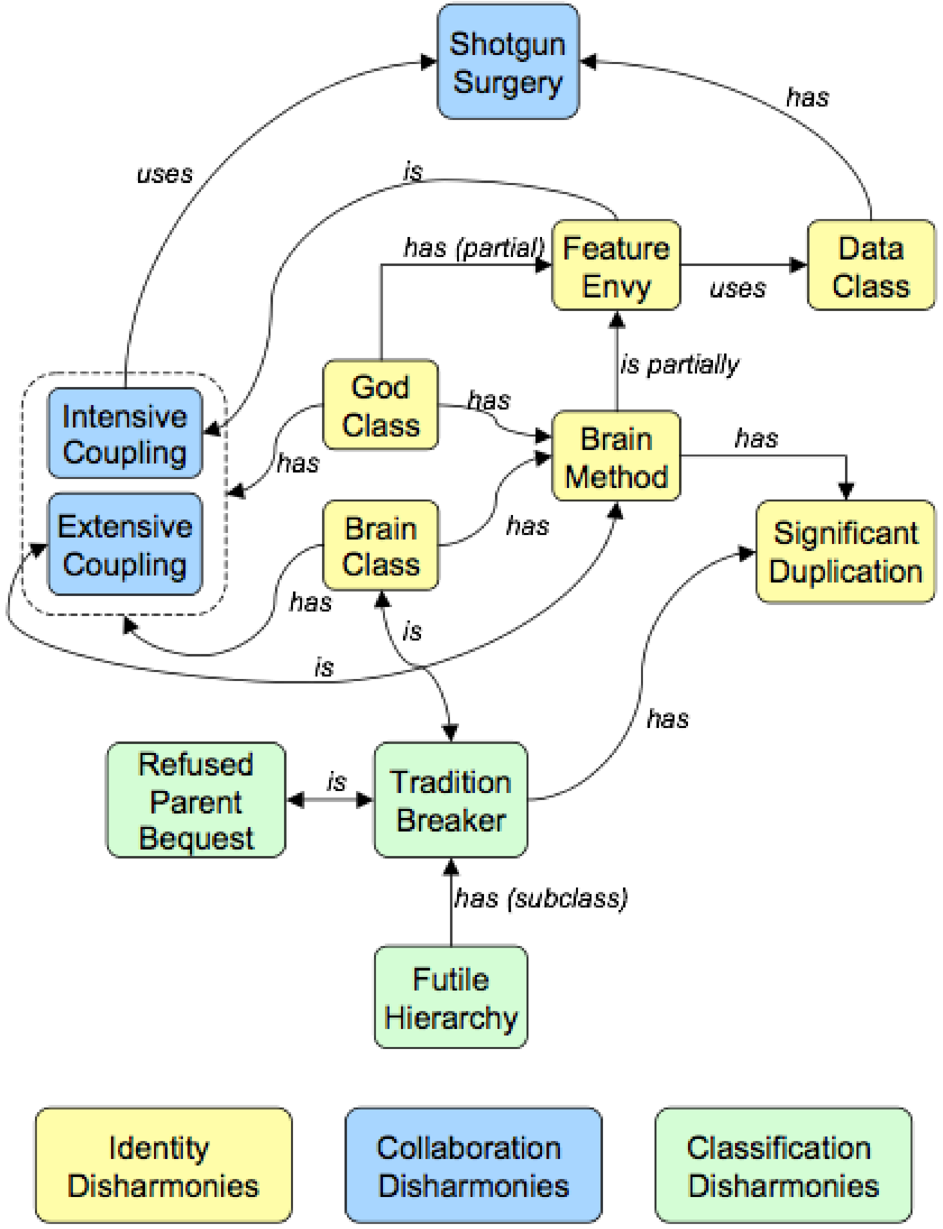
Compose metrics into queries
using logical operators

Detection Strategies

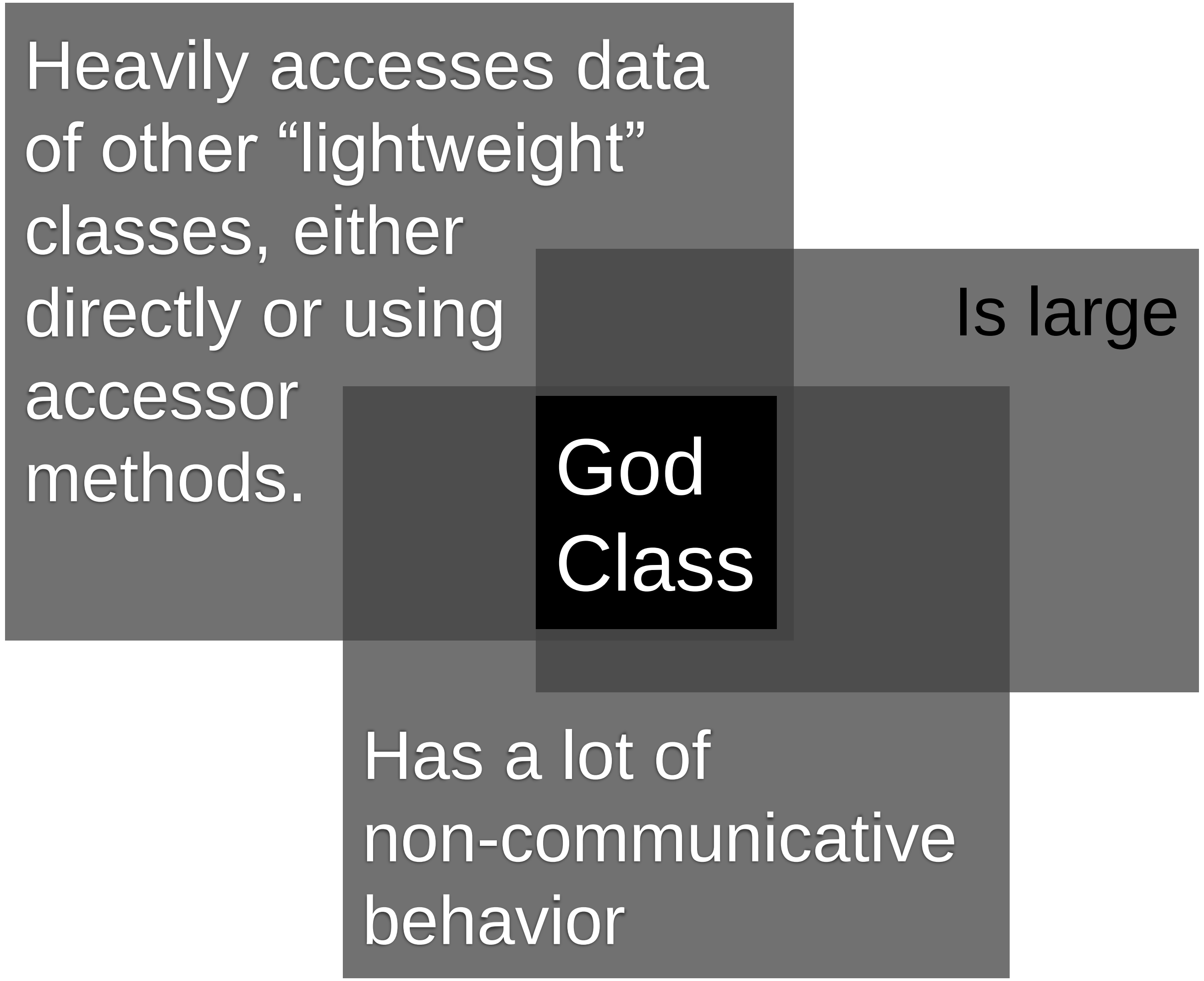
- Detection strategies are metric-based queries to detect design flaws



Design Flaws do not come alone



Characteristics of a God Class



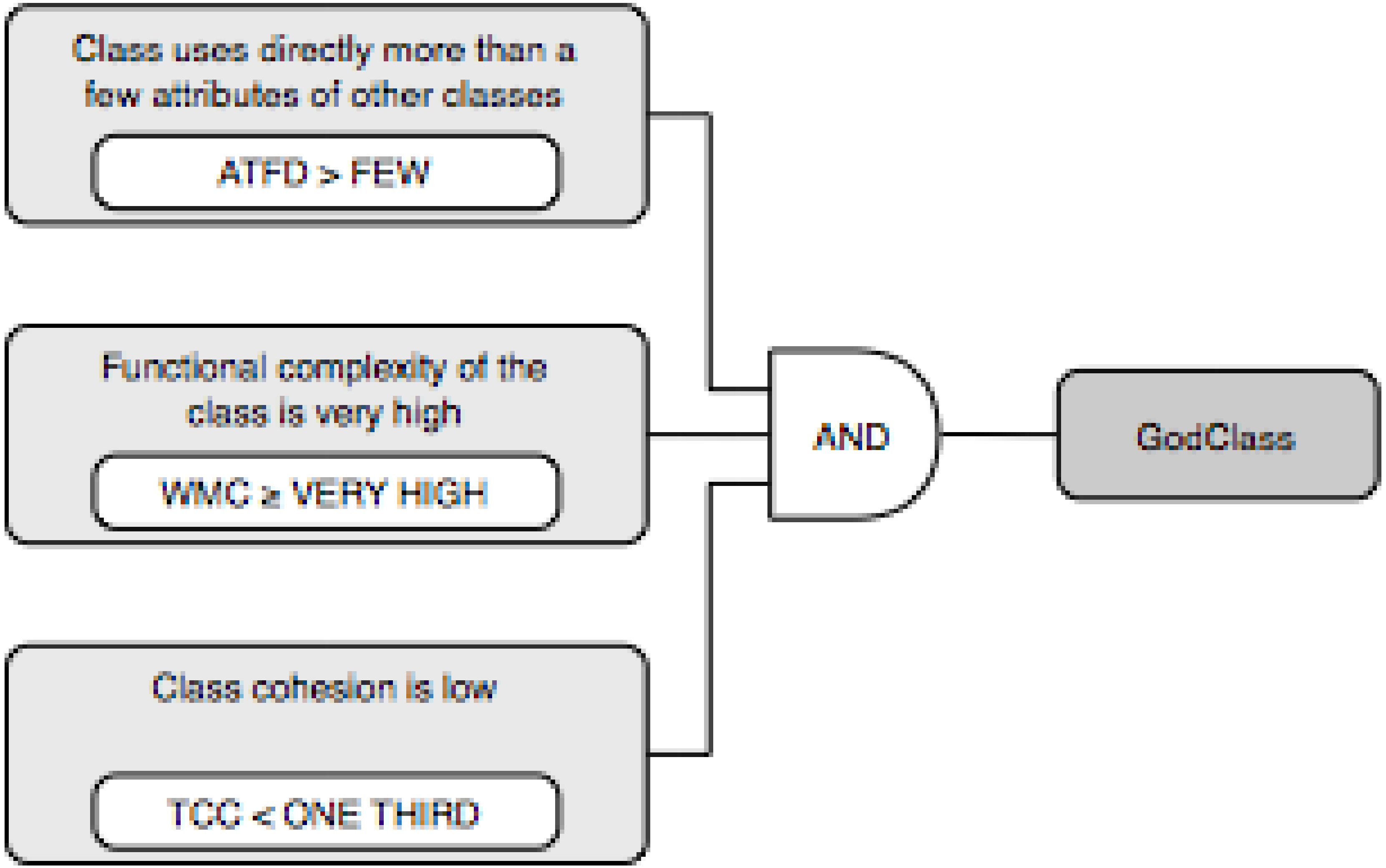
Heavily accesses data of other “lightweight” classes, either directly or using accessor methods.

Is large

God Class

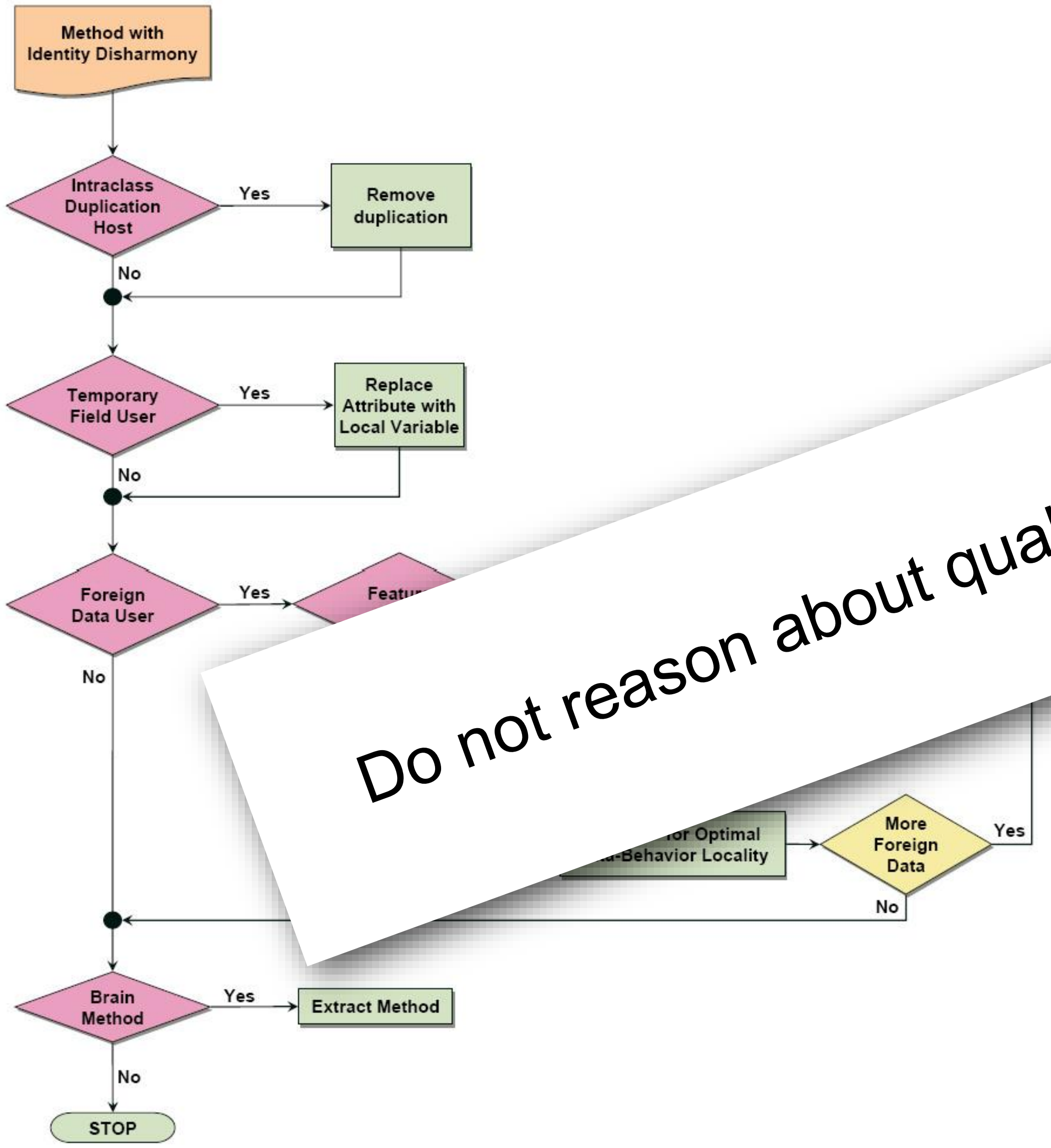
Has a lot of non-communicative behavior

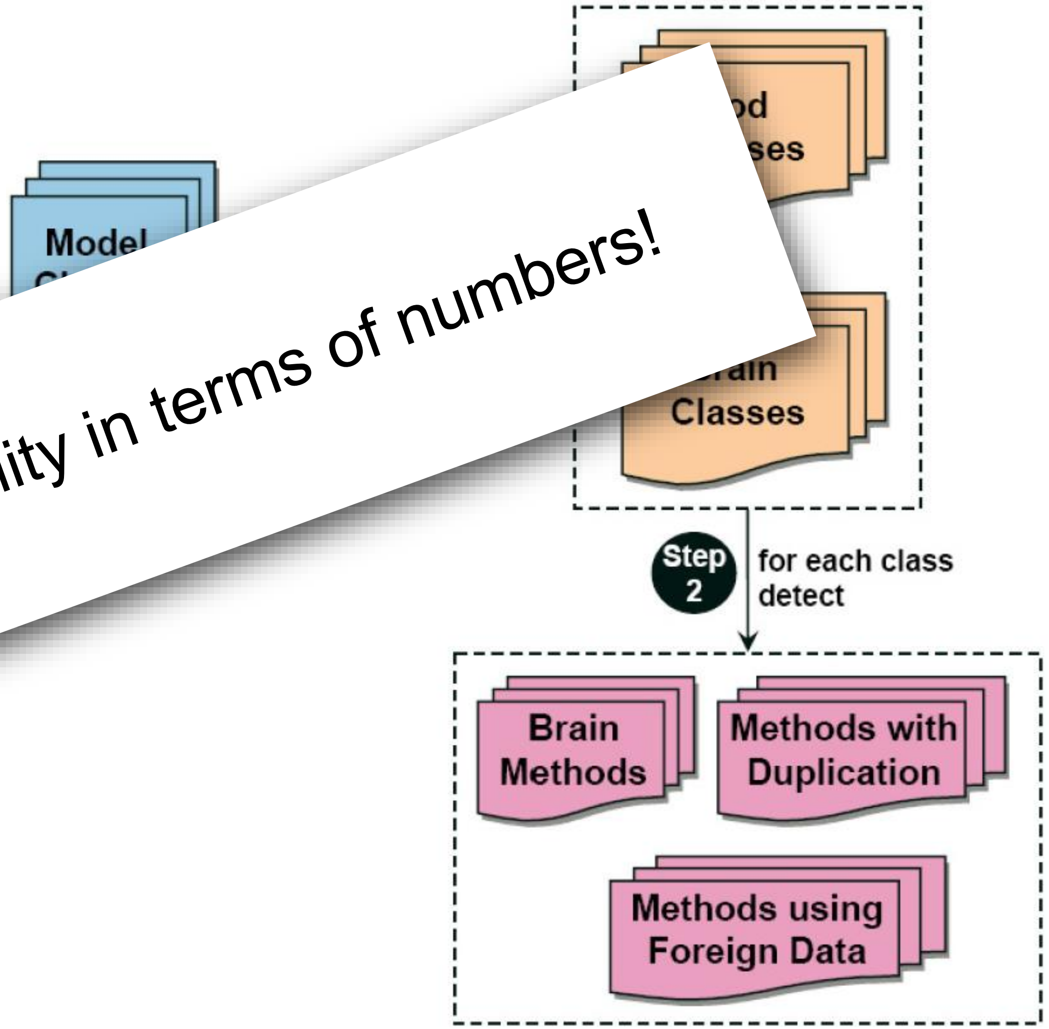
God Class Detection Strategy


Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY HIGH
Class cohesion is low
TCC < ONE THIRD
AND
GodClass

And Now?

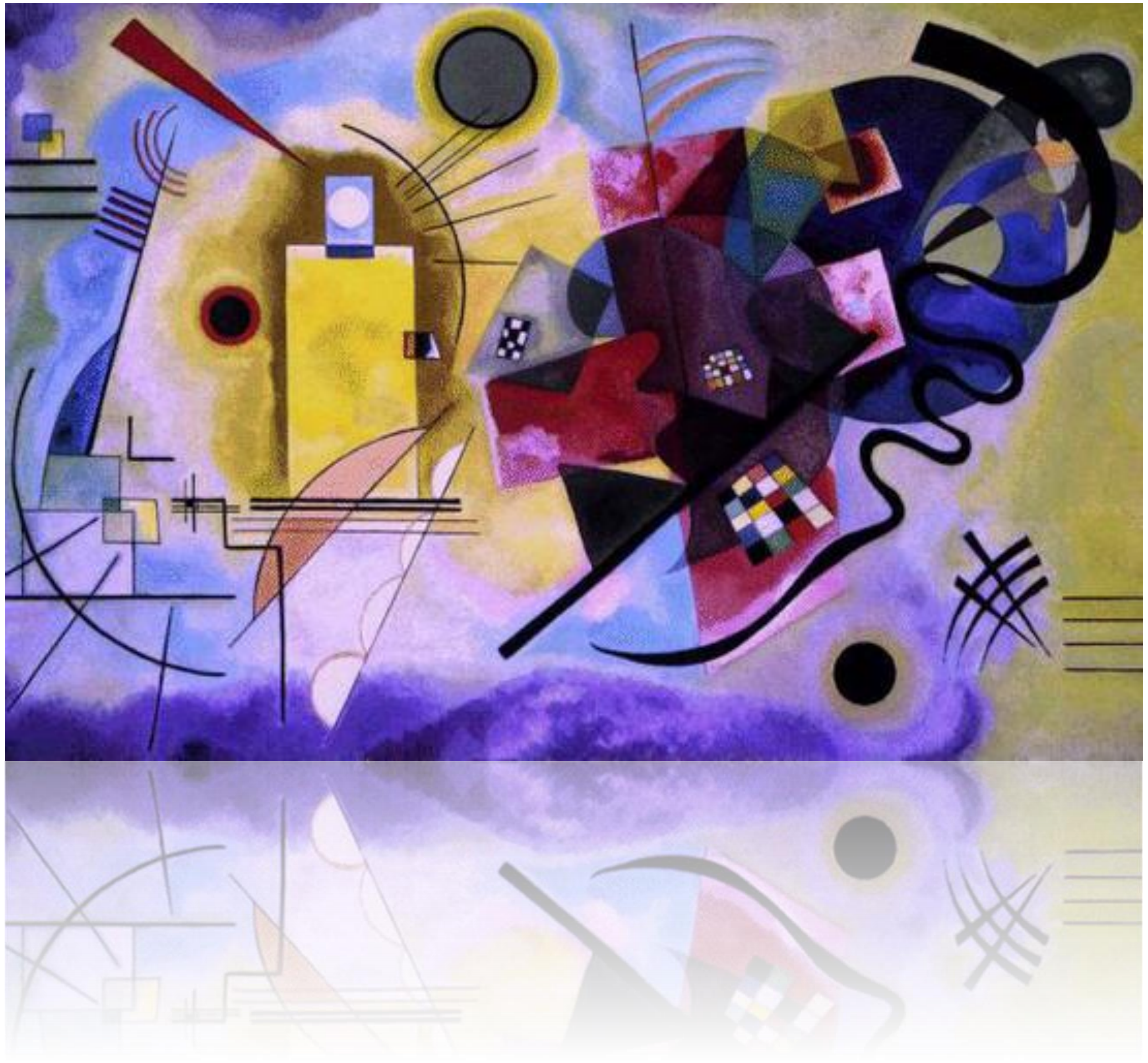
Follow A Clear and Repeatable Process





Do not reason about quality in terms of numbers!

Metrics are only half the truth



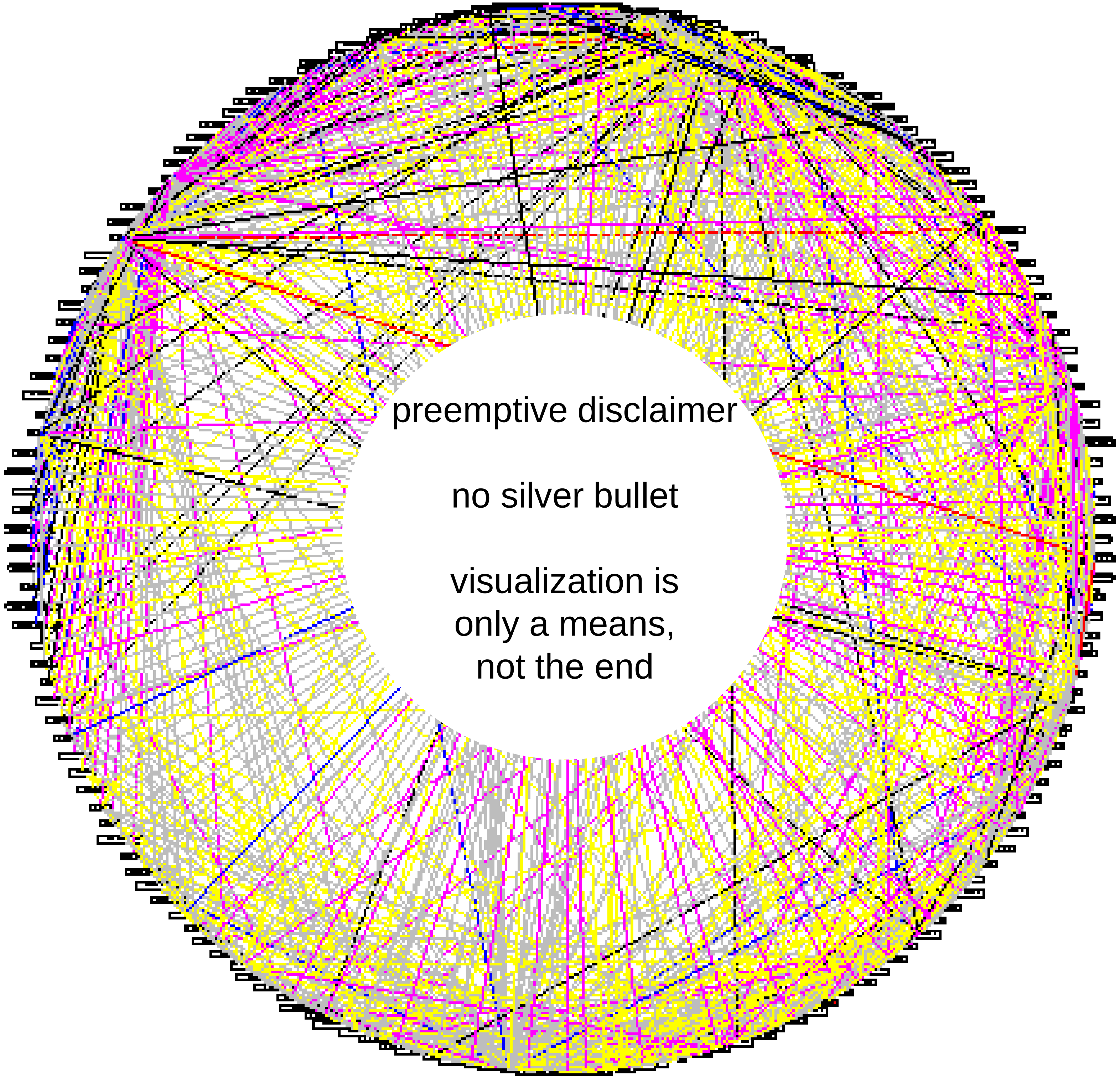
Can we understand the beauty of a painting by measuring its frame and counting its colors?

Source Code – Text

```
/**************************************************************************/
/*                              micro-Max,                               */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                             */
/* - recursive negamax search                                        */
/* - quiescence search with recaptures                               */
/* - recapture extensions                                          */
/* - (internal) iterative deepening                                 */
/* - best-move-first 'sorting'                                     */
/* - a hash table storing score and best move                         */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking     */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];          /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e4,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                      /* board: half of 16x8+dummy*/
T[1035],                                      /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                          /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                              /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                  /* first empty or match     */
 a+=i?j:0;                                 /* dummy A[0] if miss & full*/
 if(a->K)                                  /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                      /* examine stored data      */
  if(d>=n)                                 /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;            /* use if window compatible */
   d=n-1;                                  /* or use as iter. start    */
  }X&=~M;Y=a->Y;                             /*      with best-move hint */
  Y=d?Y:0;                                  /* don't try best at d=0    */
 }else d=X=Y=0;                               /* start iter., no best yet */
 N++;                                      /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                        /* iterative deepening loop */
 {x=B=X;                                    /* start scan at prev. best */
  Y|=8&Y>>4;                                  /* request try noncastl. 1st*/
  m=d>1?-I:e;                                 /* unconsidered:static eval */
  do{u=b[x];                                 /* scan board looking for   */
   if(u&k)                                 /*  own piece (inefficient!)*/
   {r=p=u&7;                                  /* p = piece type (set r>0) */
    j=o[p+16];                               /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                   /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                               /* y traverses ray          */
```

```
          if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;      /* K capt. or bad castling  */
          if(m>=l)goto C;                        /* abort on fail high       */




          if(h=d-(y!=z))                        /* remaining depth(-recapt.)*/
          {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
           b[G]=b[H]=b[x]=0;b[y]=u&31;              /* do move, strip virgin-bit*/
           if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
           if(p<3)                              /* pawns:                   */
           {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
                 ((x+2)&M||b[x+2]!=u)-1);         /*        squares plus bias */
            if(y+r+1&S){b[y]|=7;i+=C;}             /* promote p to Q, add score*/
           }
           v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
               J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
           v-=v>e;                              /* delayed-gain penalty     */
           if(z==9)                             /* called as move-legality  */
           {if(v!=-I&x==K&y==L)                    /*   checker: if move found */
            {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
            v=m;                                /* (prevent fail-lows on    */
           }                                 /*   K-capt. replies)       */
           b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;    /* undo move,G can be dummy */
           if(Y&8){m=v;Y&=~8;goto A;}              /* best=1st done,redo normal*/
           if(v>m){m=v;X=x;Y=y|S&G;}               /* update max, mark with S  */
          }                                  /*         if non castling */
          t+=p<5;                              /* fake capt. for nonsliding*/
          if(p<3&6*k+(y&V)==S                    /* pawn on 3rd/6th, or      */
             ||(u&~24)==36&j==7&&                  /* virgin K moving sideways,*/
             G&M&&b[G=(x|7)-(r>>1&7)]&32          /* 1st, virgin R in corner G*/
             &&!(b[G^1]|b[G^2])                   /* 2 empty sqrs. next to R  */
          ){F=y;t--;}                           /* unfake capt., enable e.p.*/
         }W(!t);                               /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                           /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                          /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,S,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                 /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;               /* empty, type (limit/exact)*/
  }                                      /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                             /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                          /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                      /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                        /* or think up one    */
```

re... Visualization?



preemptive disclaimer

no silver bullet

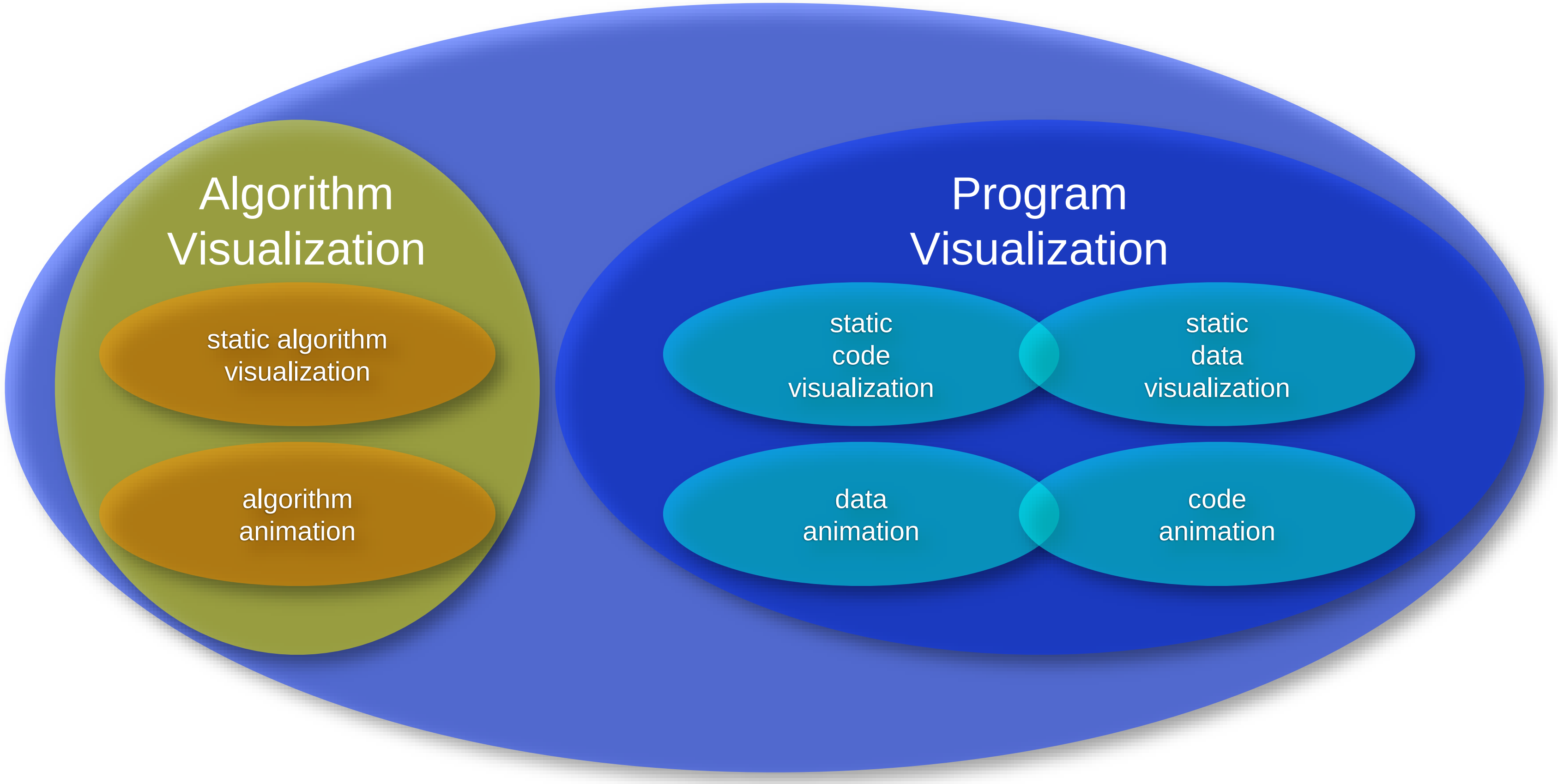
visualization is
only a means,
not the end

```
#include                                         <math.h>
#include                                       <sys/time.h>
#include                                       <X11/Xlib.h>
#include                                      <X11/keysym.h>
                                              double L ,o ,P
                                             ,_=dt,T,Z,D=1,d,
                                             s[999],E,h= 8,I,
                                             J,K,w[999],M,m,O
                                            ,n[999],j=33e-3,i=
                                            1E3,r,t, u,v ,W,S=
                                            74.5,l=221,X=7.26,
                                            a,B,A=32.2,c, F,H;
                                            int N,q, C, y,p,U;
                                           Window z; char f[52]
                                        ; GC k; main(){ Display*e=
 XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC                  ,0))
; scanf("%lf%lf%lf",y +n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCre                       400,
0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=si                          le6}
; K= cos(j); N=1e4; M+= H*_; Z=D*K; F+=_*P; r=E*K; W=cos( O                                 B=
sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T*E+ D*B*W; j                                   W+E
*T*B,E*d/K *B+v+B/K*F*D)*_; p<y; ){ T=p[s]+i; E=c-p                                        [s
]== 0|K <fabs(W=T*r-I*E +D*P) |fabs(D=t *D+Z *T                                       e2/ K
 *D; N-1E4&& XDrawLine(e ,z,k,N ,U,q,C); N=                                       l*l+M *M;
  XDrawString(e,z,k ,20,380,f,17); D=v/                                    u *=CS!=N){
                                                  F/l;
                                               M/ l,l*H
                                              *M+a*X)*_; H
                                             =A*r+v*X-F*l+(
                                             E=.1+X*4.9/l,t
                                             =T*m/32-I*T/24
                                              )/S; K=F*M+(
                                              h* 1e4/l-(T+
                                              E*5*T*E)/3e2
                                              )/S-X*d-B*A;
                                              a=2.63 /l*d;
                                              X+=( d*l-T/S
                                               *(.19*E +a
                                               *.64+J/1e3
                                               )-M* v +A*
                                               Z)*_; l +=
                                               K *_; W=d;
                                               sprintf(f,
                                               "%5d  %3d"
                                               "%7d",p =l
                                              /1.7,(C=9E3+
                               O*57.3)%0550,(int)i); d+=T*(.45-14/l*
                              X-a*130-J* .14)*_/125e2+F*_*v; P=(T*(47
                              *I-m* 52+E*94 *D-t*.38+u*.21*E) /1e2+W*
                              179*v)/2312; select(p=0,0,0,0,&G); v-=(
                               W*F-T*(.63*m-I*.086+m*E*19-D*25-.11*u
                                )/107e2)*_; D=cos(o); E=sin(o); } }
```

not software visualization

Software Visualization

- Program Visualization: “The visualization of the actual program code or data structures in static or dynamic form”
- Algorithm Visualization: “The visualization of the higher-level abstractions which describe software”



Software Visualization in Context

- There are many good-looking visualizations, but...
- When it comes to maintenance & evolution, there are several issues:
  - Scalability
  - Information Retrieval
  - What to visualize
  - How to visualize
  - Limited time
  - Limited resources

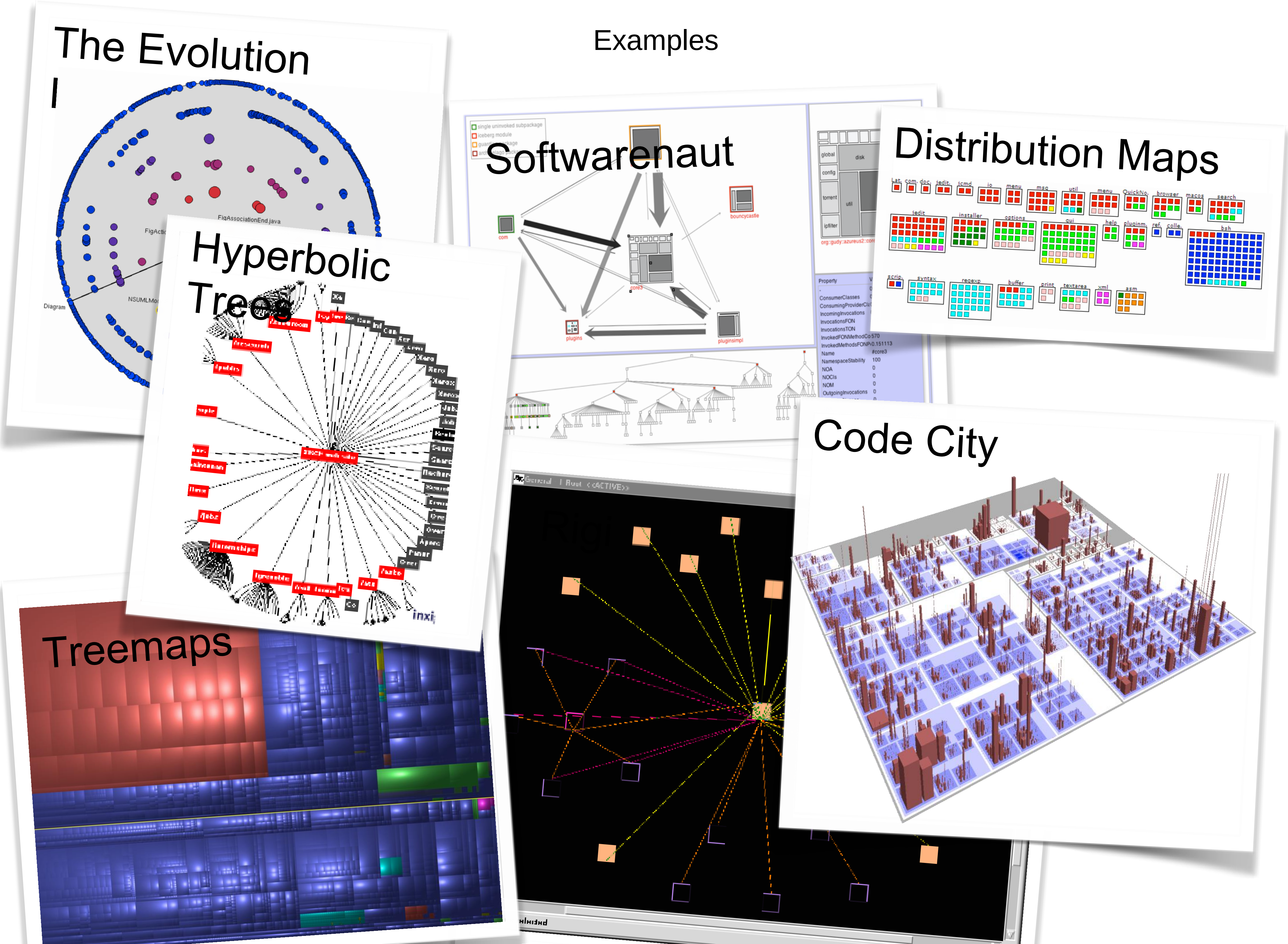
Program Visualization

- “The visualization of the actual program code or data structures in either static or dynamic form”
- Overall goal: generate views of a system to understand it
- Surprisingly complex problem domain/research area
  - Visual Aspects: Efficient use of space, overplotting problems, layout issues, HCI issues, GUI issues, lack of conventions (colors, shapes, etc.)
  - Software Aspects
    - Granularity (complete systems, subsystems, modules, classes, etc.)
    - When to apply (first contact, known/unknown parts, forward engineering?)
    - Methodology

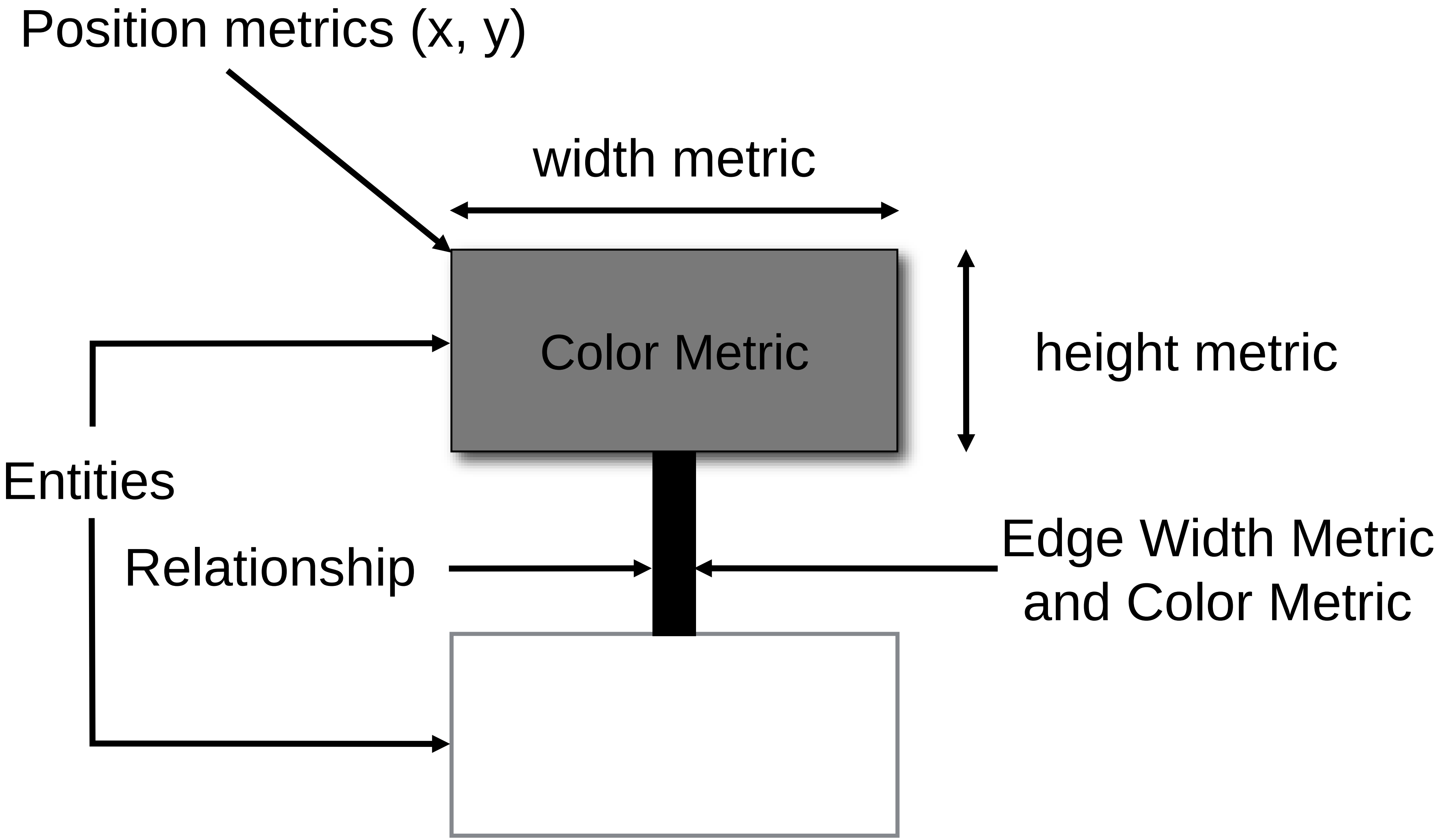
Static Code Visualization

- The visualization of information that can be extracted from a system at “compile-time”
- Directly influenced by programming languages and their paradigms
  - Object-Oriented: classes, methods, attributes, inheritance, ...
  - Procedural: procedures, invocations, imports, ...
  - Functional: functions, function calls, ...

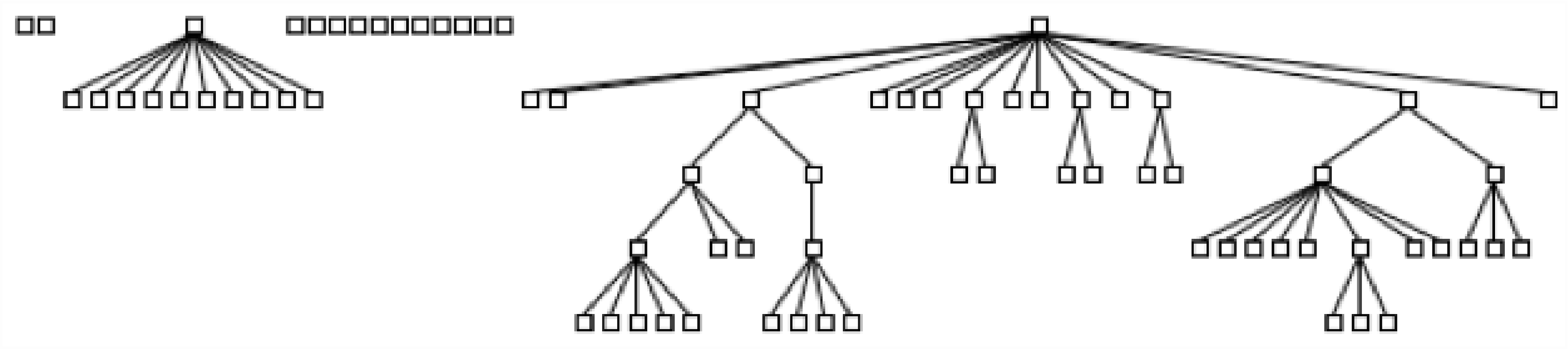
Examples


The Evolution
I
Softwarenaut
Distribution Maps
Hyperbolic Trees
Rigi
Code City
Treemaps

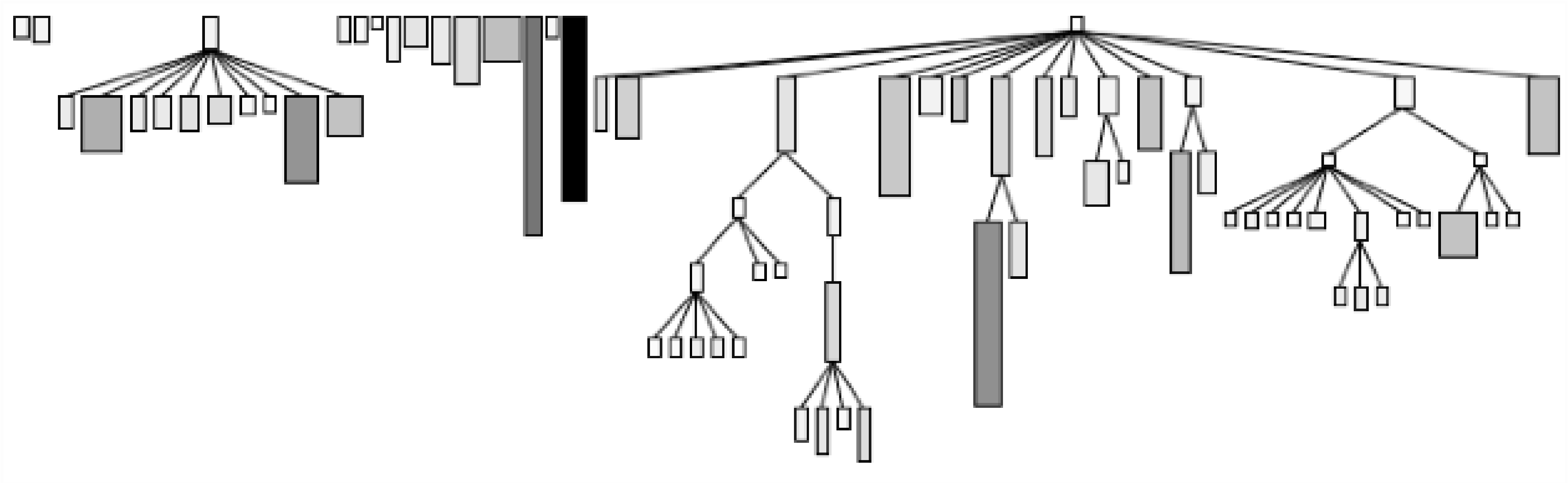
The Polymetric View Principle

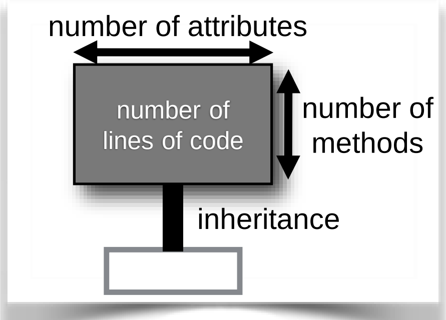


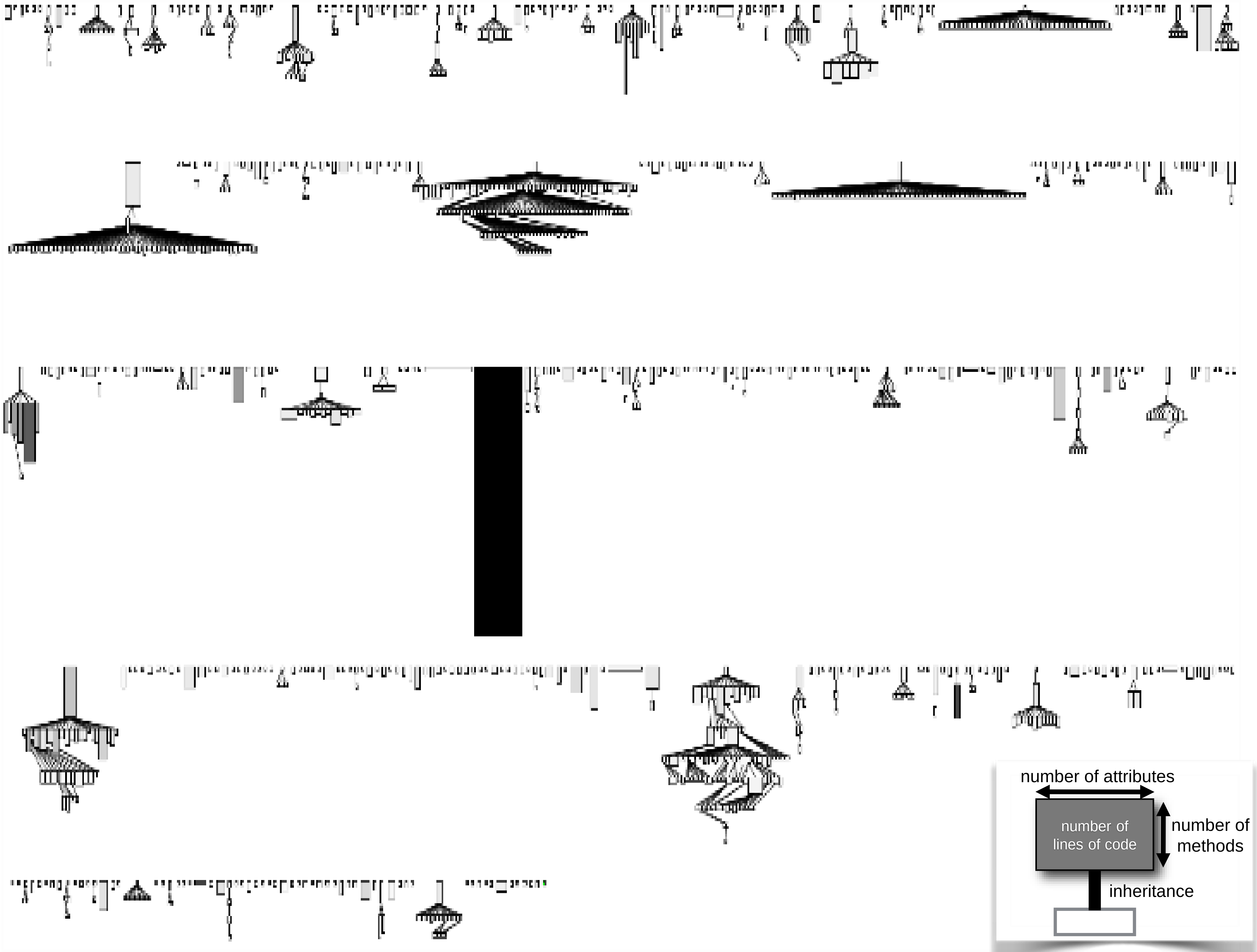
Class Hierarchy



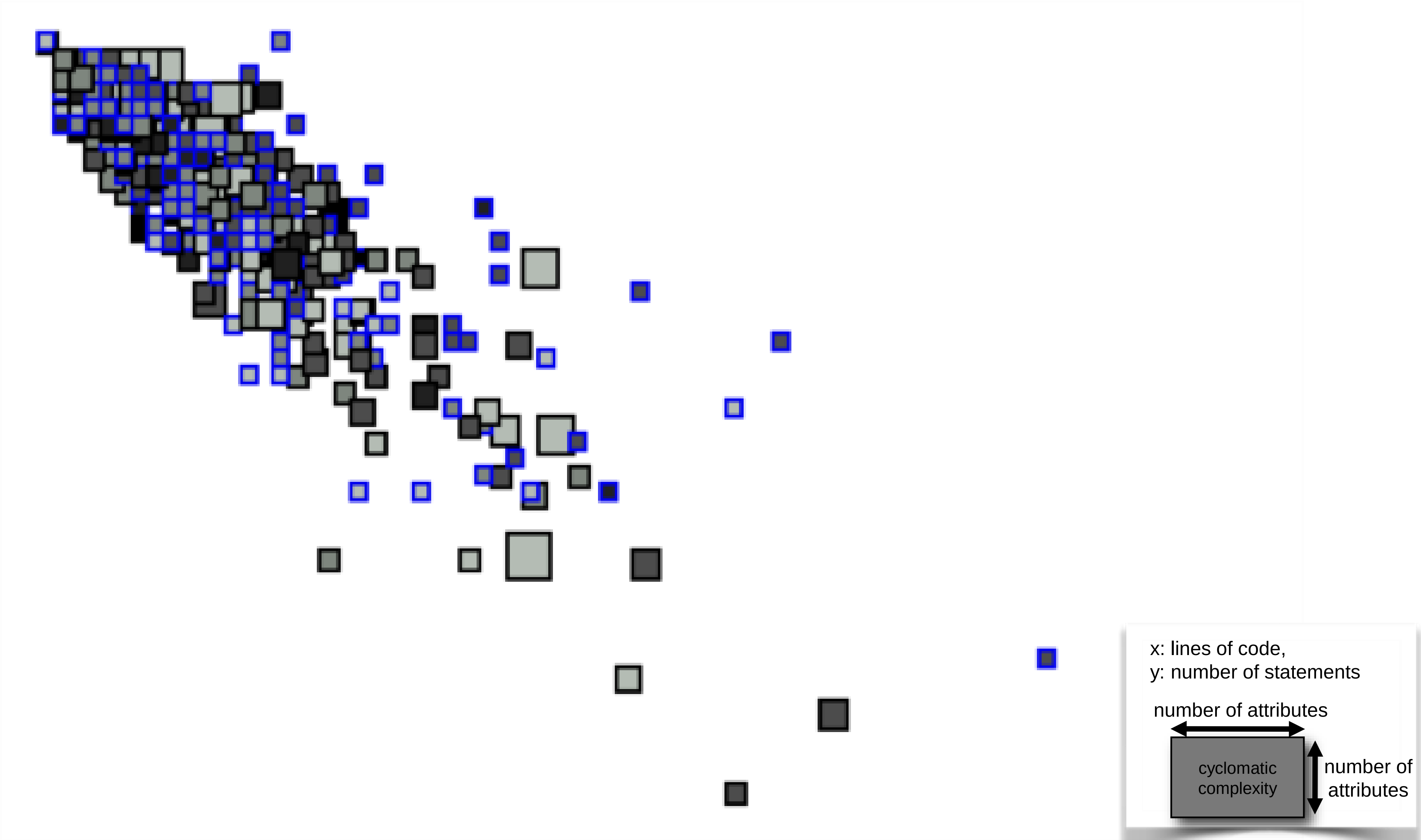
System Complexity View






number of attributes
number of lines of code
number of methods
inheritance

Method Structure Correlation View



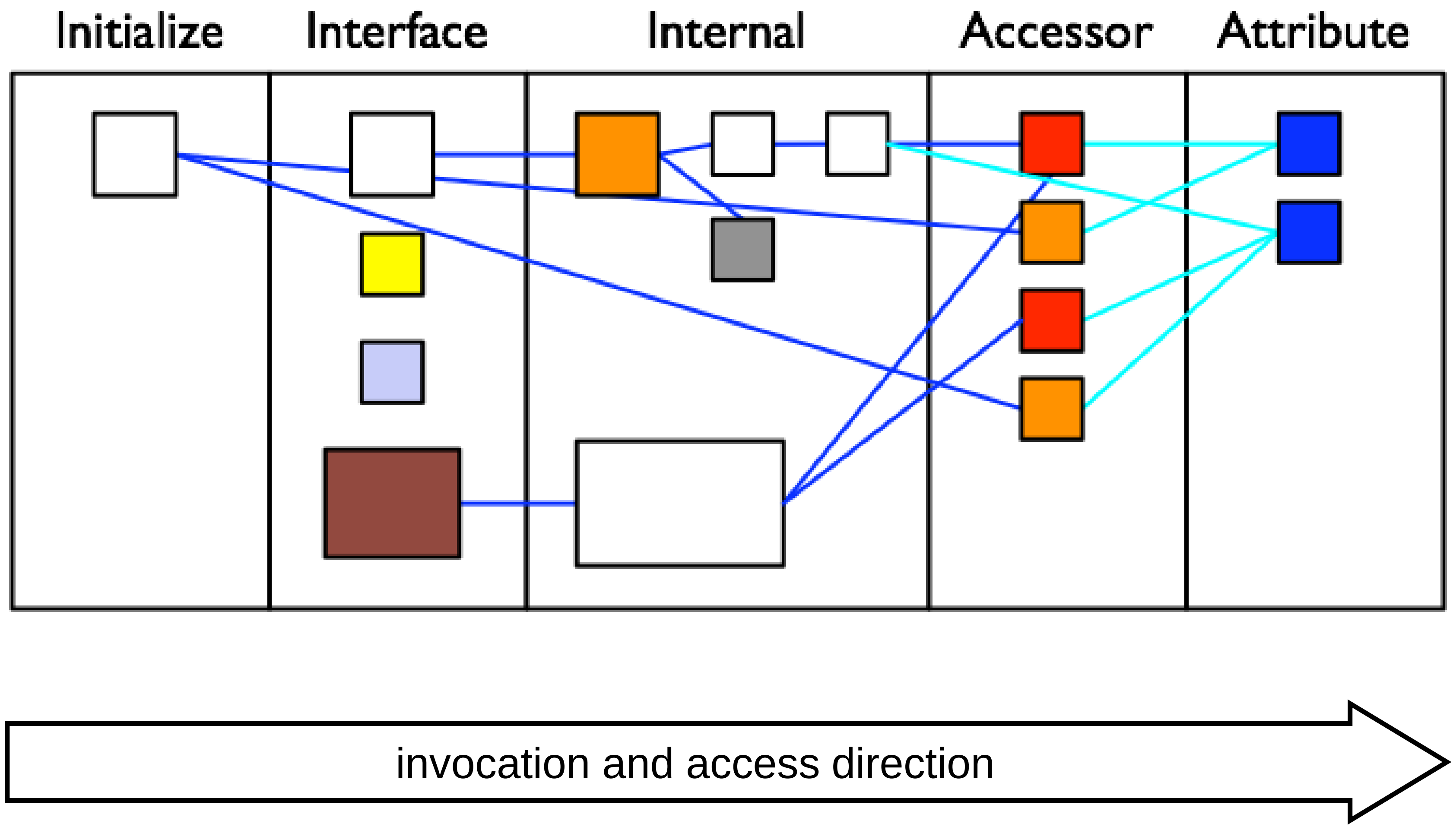
x: lines of code,
y: number of statements

number of attributes

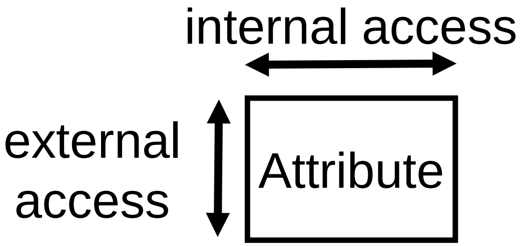
cyclomatic complexity

number of attributes

Increasing Information Granularity: The Class Blueprint

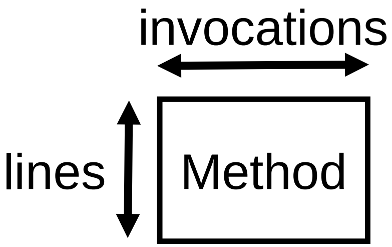


Detailing Class Blueprints

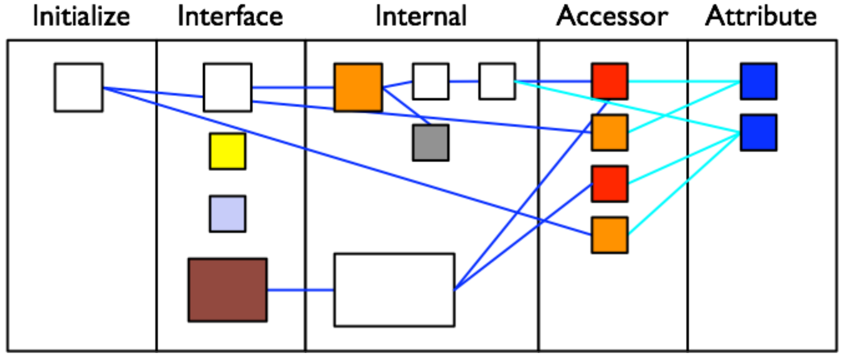


Access

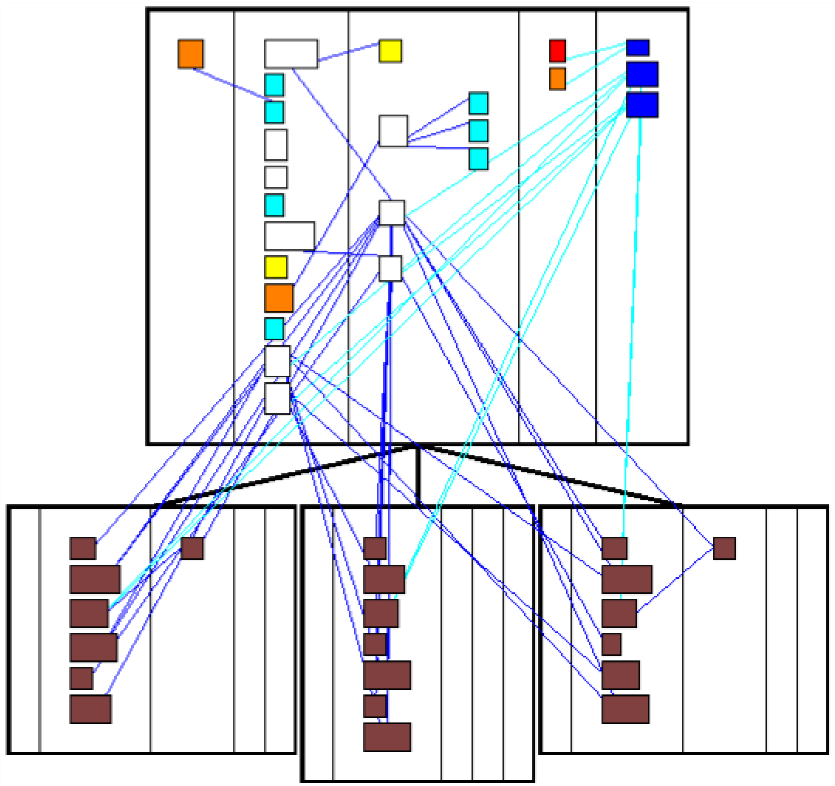
Invocation

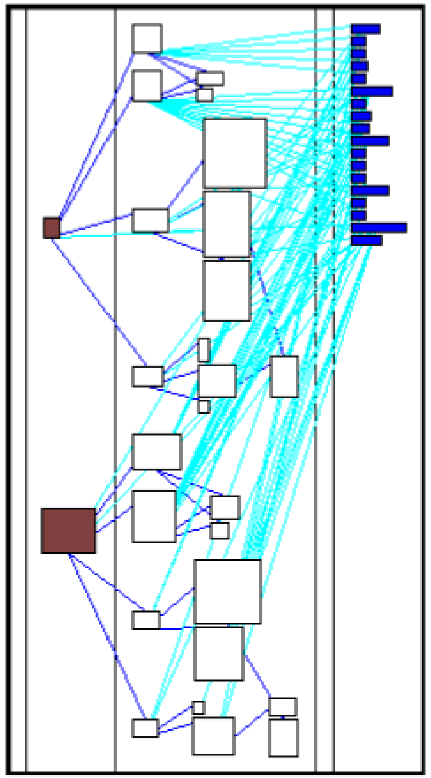


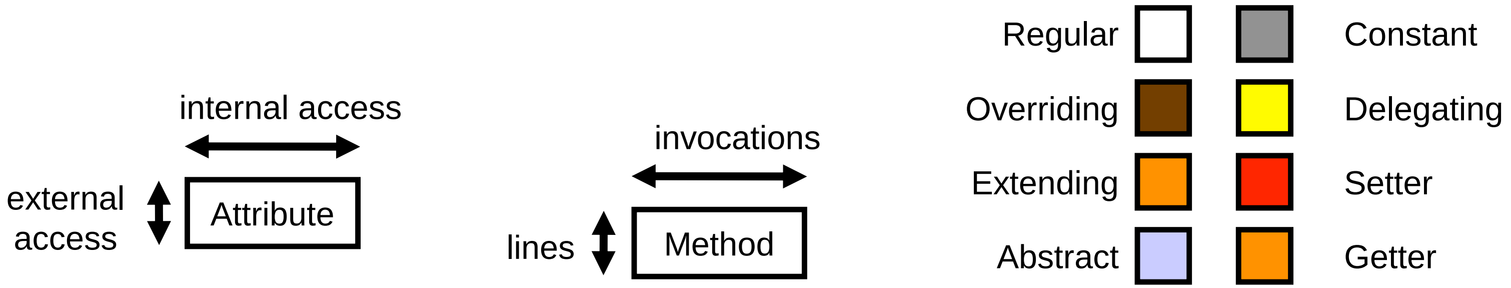
Regular	□	▩	Constant
Overriding	🟫	🟨	Delegating
Extending	🟧	🟥	Setter
Abstract	🟪	🟧	Getter



A Pattern Language based on Class Blueprints







internal access

external access

Attribute

invocations

lines

Method

Regular

Overriding

Extending

Abstract

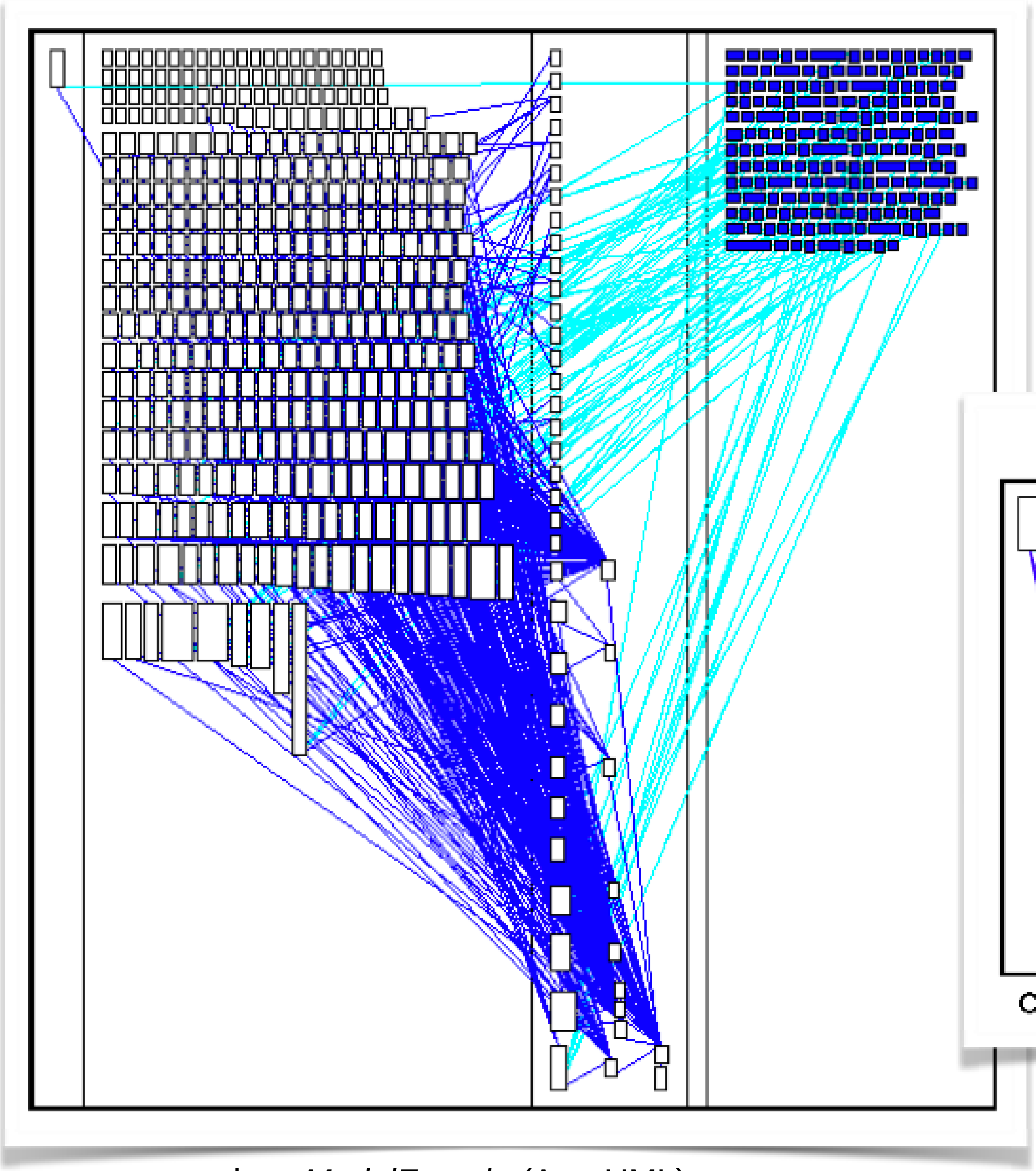
Constant

Delegating

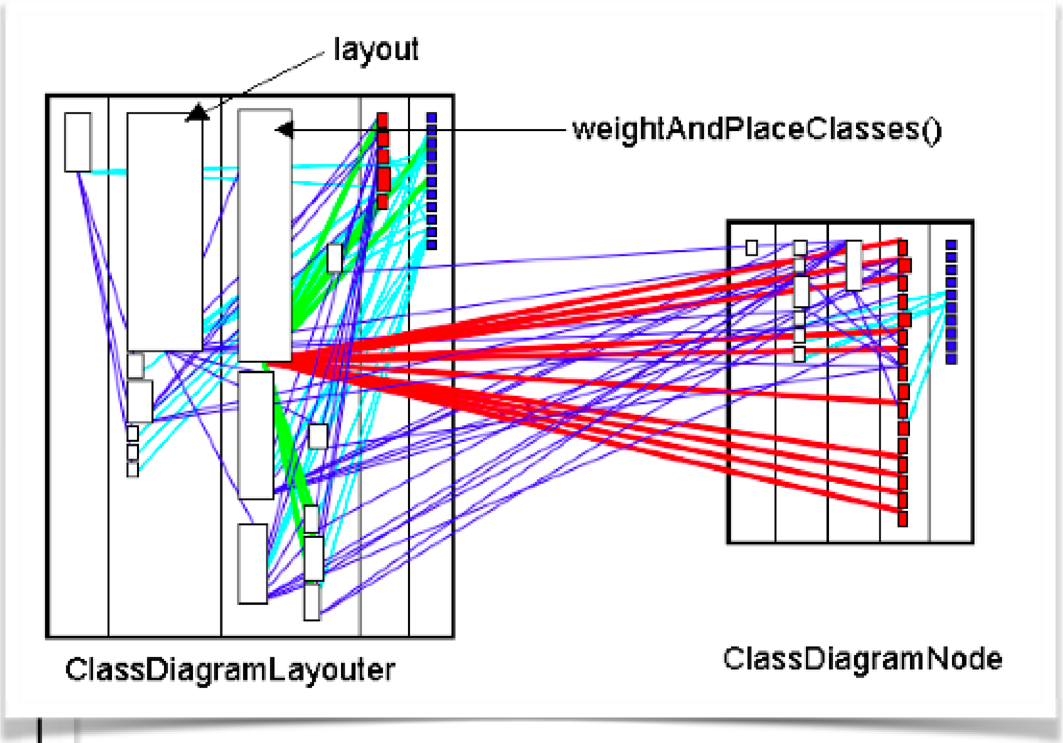
Setter

Getter

What do you see here ?



class *ModelFacade* (ArgoUML)



classes *ClassDiagramLayouter*
and *ClassDiagramNode* (ArgoUML)

Reflections on Static Visualization

- Pros
  - Intuitive
  - Aesthetically pleasing
- Cons
  - Several approaches are orthogonal to each other
  - No conventions
  - Too easy to produce meaningless results
  - Scaling up is possible at the expense of semantics
- Orthogonally
  - Without programming knowledge it’s only colored boxes and arrows..

References

M. Lanza, R. Marinescu, *Object-Oriented Metrics in Practice*, Springer, 2006.

M. Lanza, *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*, Ph.D. Thesis, University of Berne, Switzerland, 2003.
http://www.inf.usi.ch/faculty/lanza/Downloads/Lanz03b.pdf

Overview of the Overview Pyramid:
http://pdepend.org/documentation/handbook/reports/overview-pyramid.html

Design of Software Systems
(Ontwerp van SoftwareSystemen)

5 Software Development Processes

Roel Wuyts
2016-2017

Software Process

Set of activities that leads to the production of a software product

- lots of processes exist
- share some fundamental activities

Development Phases

Testing	Validate the solution against the requirements
Analysis	Model and specify the requirements (“what”)
Maintenance	Repair defects and adapt the solution to new requirements
Implementation	Construct a solution in software
Requirements Collection	Establish customer’s needs
Design	Model and specify a solution (“how”)

Software Development Process

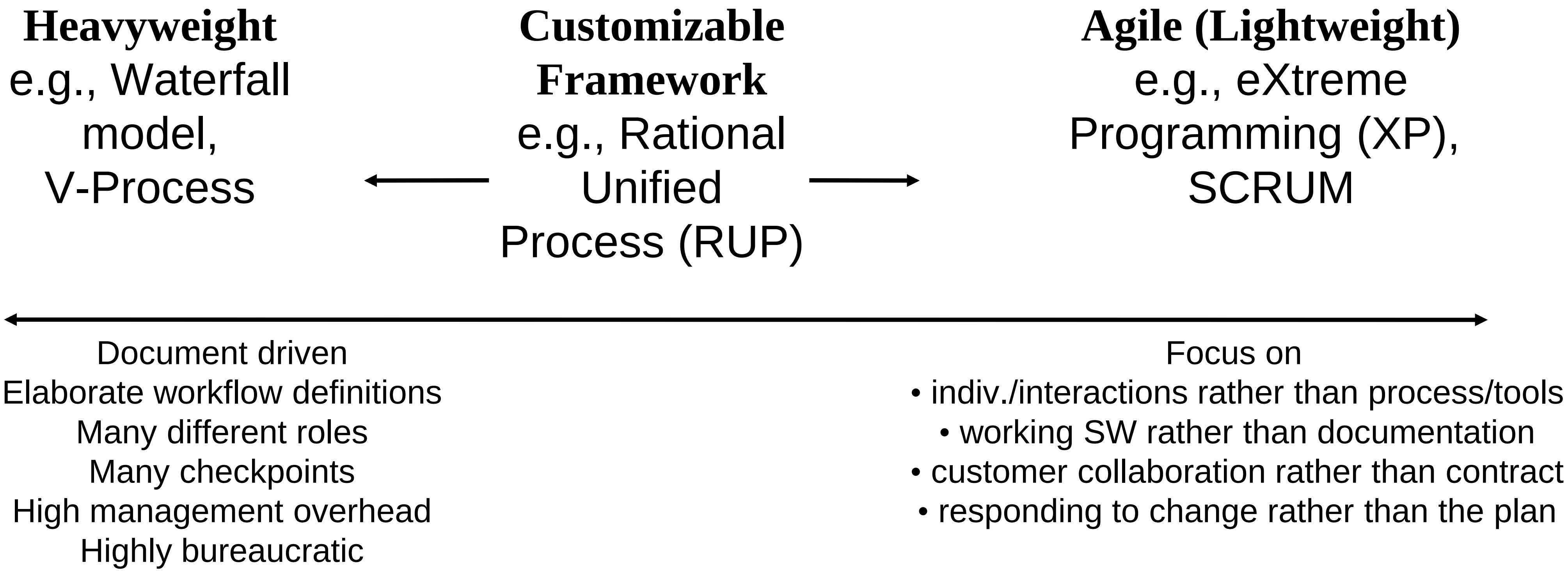
A software development methodology is

- a set of partially ordered steps
- to build, deploy, maintain, … software

Examples:

- Waterfall
- Spiral
- XP (eXtreme Programming)
- UP (Unified Process)
  - RUP (Rational Unified Process)
  - Agile UP

Lightweight vs. Heavyweight Processes



Heavyweight
e.g., Waterfall model,
V-Process

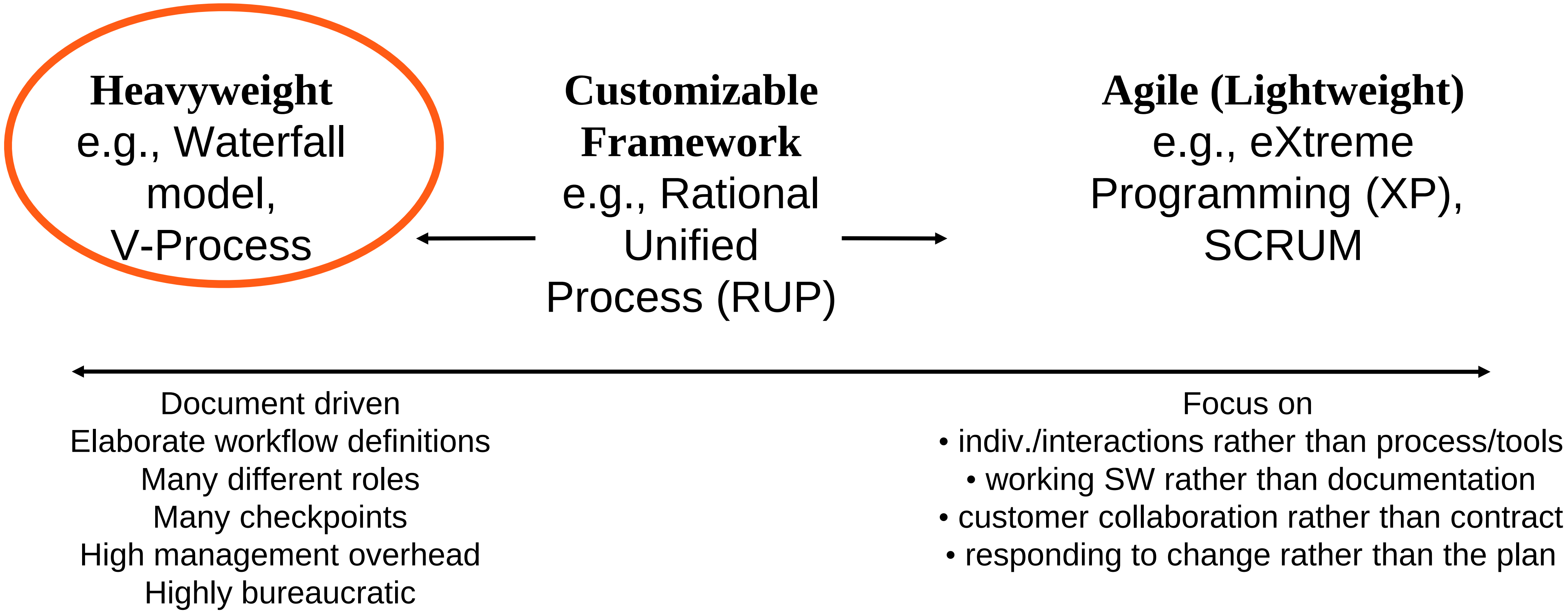
Customizable Framework
e.g., Rational Unified Process (RUP)

Agile (Lightweight)
e.g., eXtreme Programming (XP), SCRUM

Document driven
Elaborate workflow definitions
Many different roles
Many checkpoints
High management overhead
Highly bureaucratic

Focus on
- indiv./interactions rather than process/tools
- working SW rather than documentation
- customer collaboration rather than contract
- responding to change rather than the plan

Lightweight vs. Heavyweight Processes



Heavyweight
e.g., Waterfall model,
V-Process

Customizable Framework
e.g., Rational Unified Process (RUP)

Agile (Lightweight)
e.g., eXtreme Programming (XP),
SCRUM

Document driven
Elaborate workflow definitions
Many different roles
Many checkpoints
High management overhead
Highly bureaucratic

Focus on
- indiv./interactions rather than process/tools
- working SW rather than documentation
- customer collaboration rather than contract
- responding to change rather than the plan

Waterfall Model

Characterized by

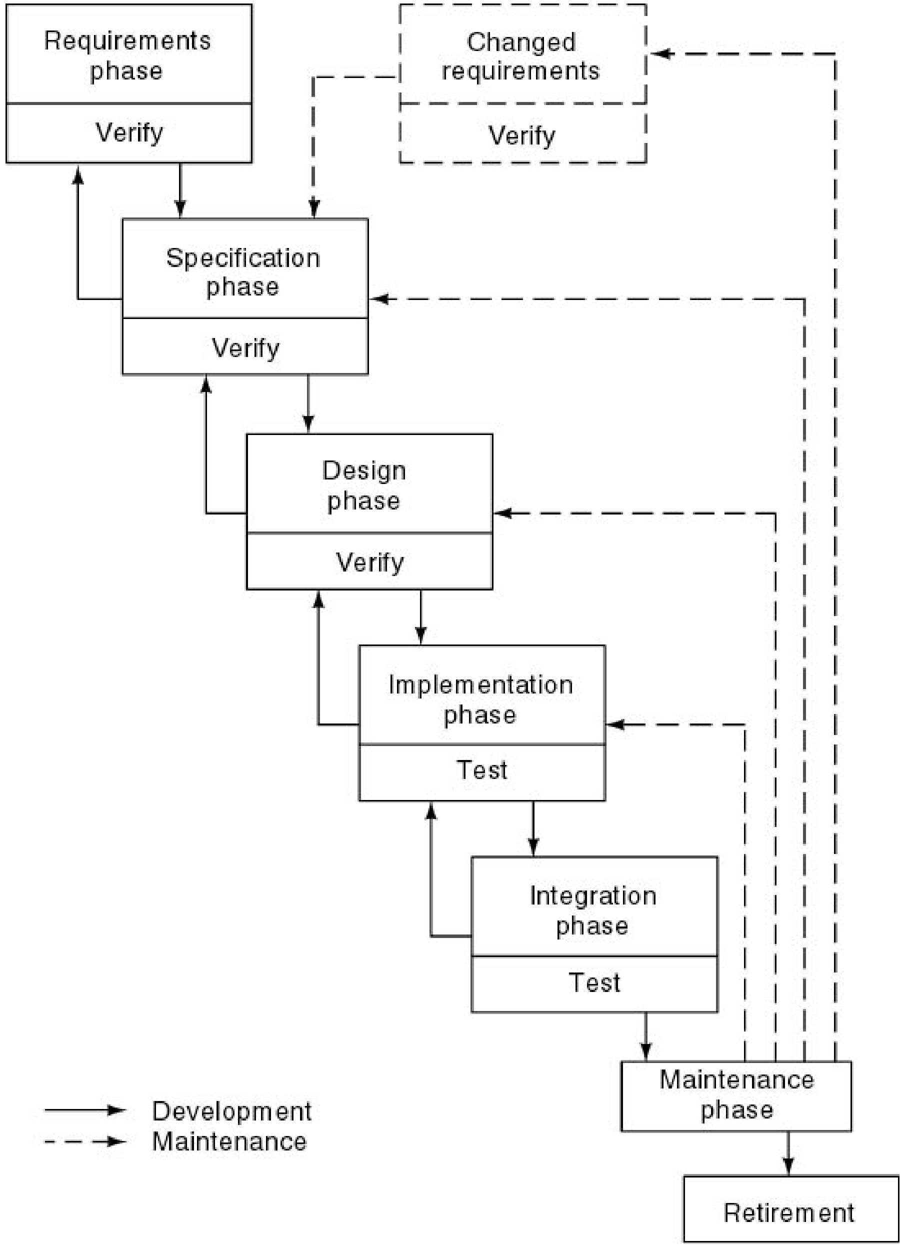
- Sequential steps (phases)
- Feedback loops (between two phases in development)
- Documentation-driven

Advantages

- Documentation
- Maintenance easier

Disadvantages

- Complete and frozen specification document up-front often not feasible in practice
- Customer involvement in the first phase only
- Sequential and complete execution of phases often not desirable
- Process difficult to control
- The product becomes available very late in the process

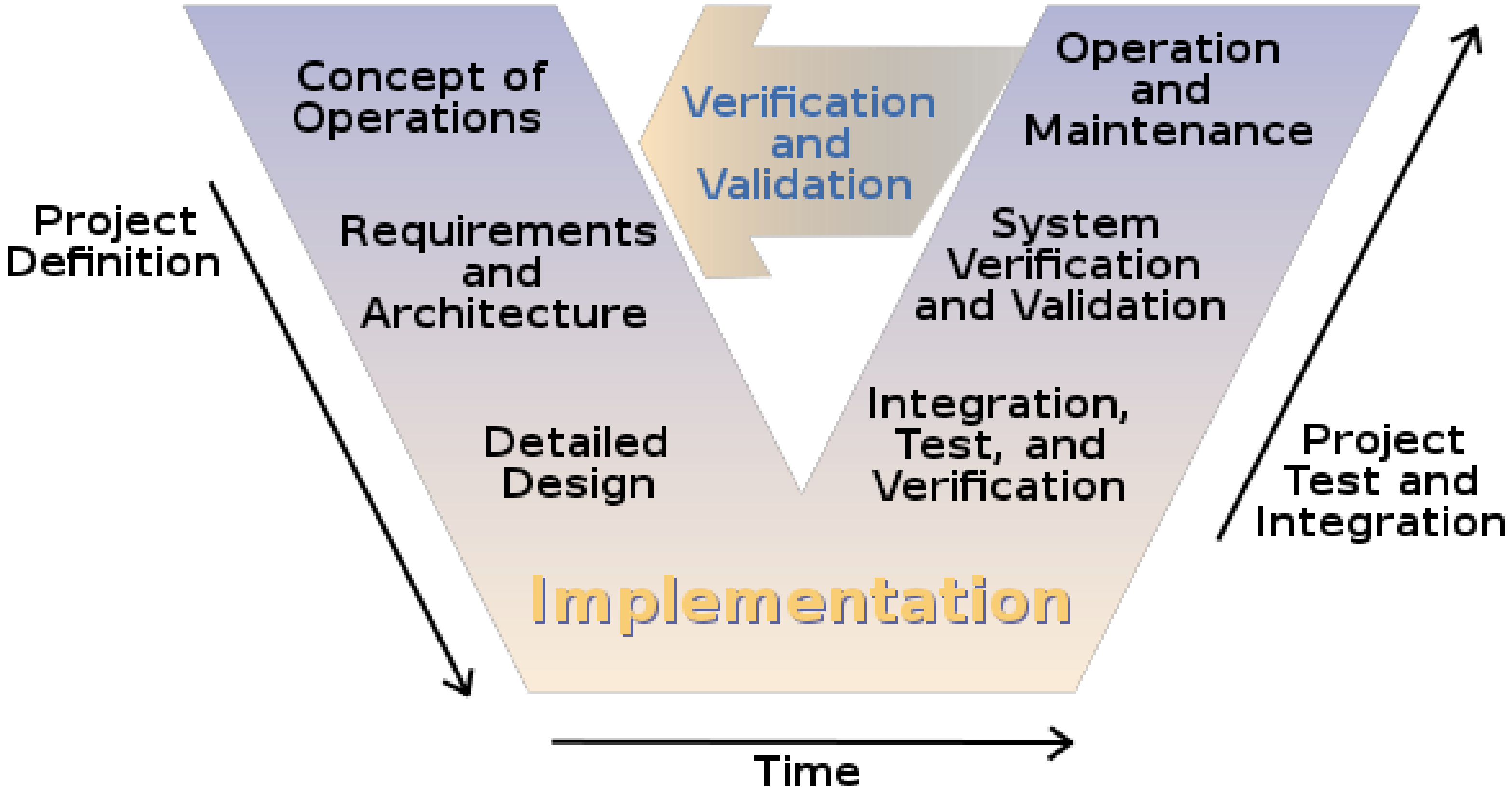


V-Model

Like the Waterfall model, it is a linear model that is very rigid

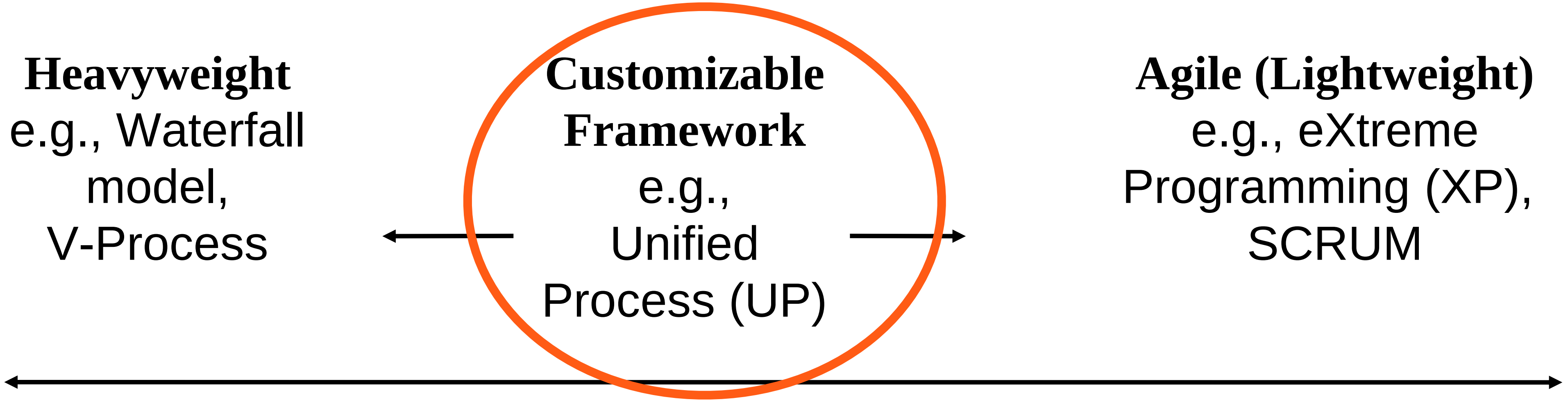
- Requirements are expected not to change
- Due to the V-shape, the first tests are the implementation tests

Unlike the waterfall model, every integration is tested



Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

Lightweight vs. Heavyweight Processes



Heavyweight
e.g., Waterfall model,
V-Process

Customizable Framework
e.g.,
Unified Process (UP)

Agile (Lightweight)
e.g., eXtreme Programming (XP),
SCRUM

Document driven
Elaborate workflow definitions
Many different roles
Many checkpoints
High management overhead
Highly bureaucratic

Focus on
- indiv./interactions rather than process/tools
- working SW rather than documentation
- customer collaboration rather than contract
- responding to change rather than the plan

UP: Iterative and Incremental development

iterative & incremental development : embracing change

- Essential for SW Development

  ("No Silver Bullet", Brooks, 1987)

iterative models: can be iterative w.r.t. value and/or requirements

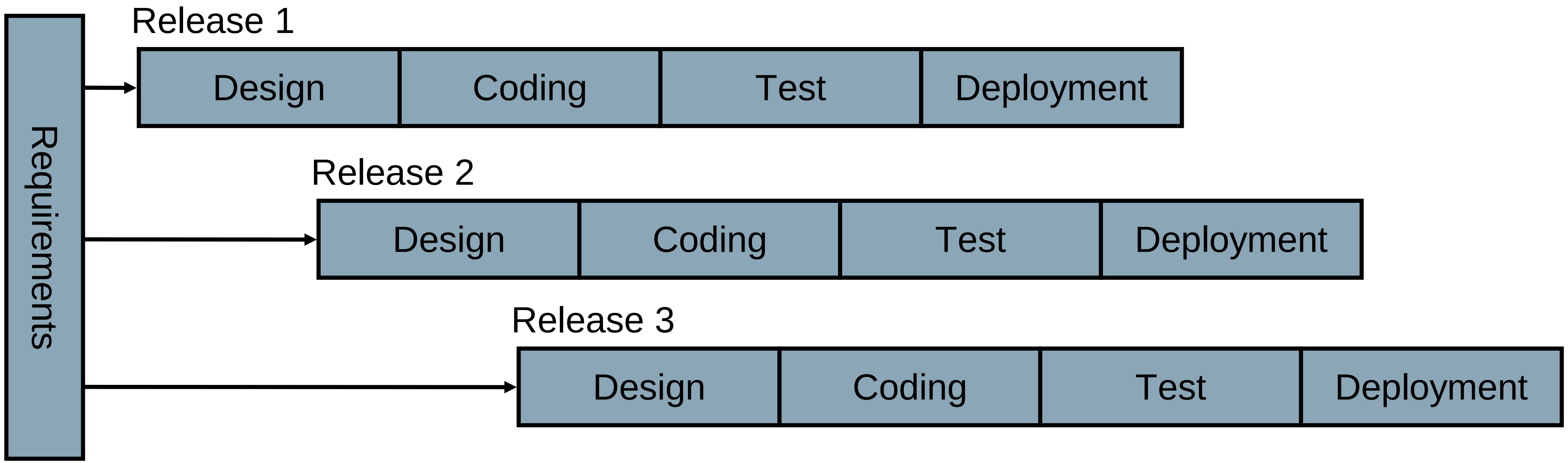
Iterative Development (a.k.a. incremental models)

More functionality with each release (new increment)

- Operational quality portion of product within weeks

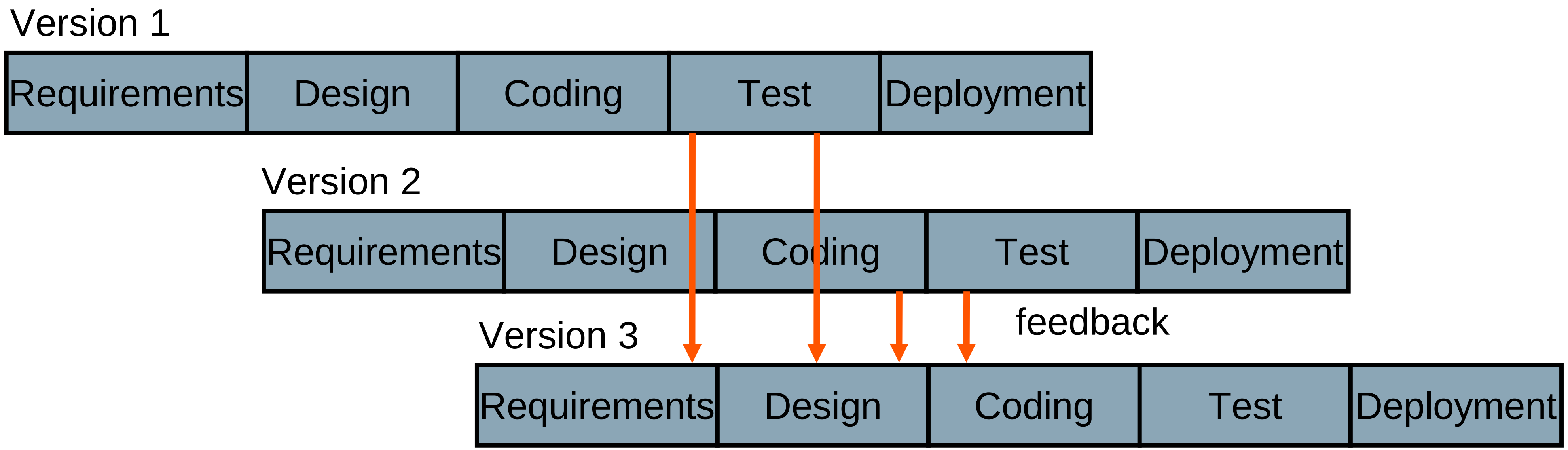
Non-incremental models (e.g. Waterfall)

- Operational quality complete product at end



Incremental development (a.k.a. Evolutionary Models)

New versions implement new and evolving requirements



UP is Use-Case-Driven

Use cases are concise, simple, and understandable by a wide range of stakeholders

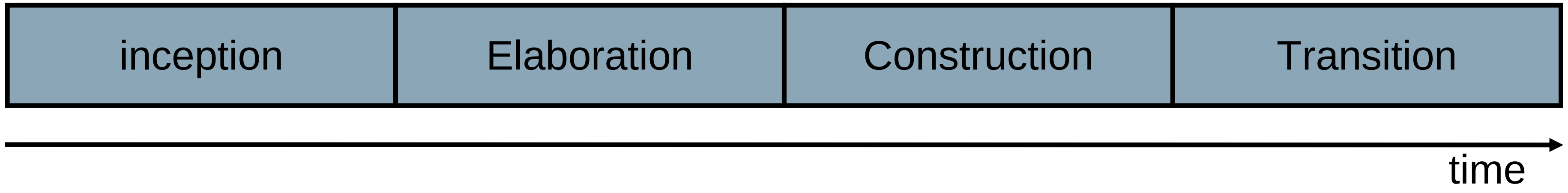
- End users, developers and acquirers understand functional requirements of the system

Use cases drive numerous activities in the process:

- Creation and validation of the design model
- Definition of test cases and procedures of the test model
- Planning of iterations
- Creation of user documentation
- System deployment

Use cases help synchronize the content of different models

UP’s 4 Project Life Cycle Phases



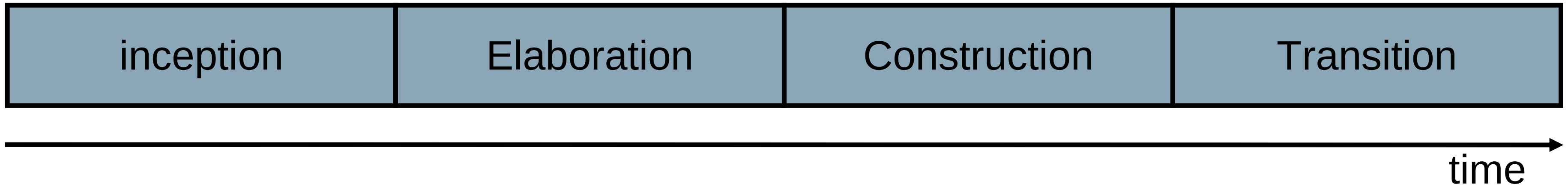
Inception

- Approximate vision
- Business case
- Scope
- Vague estimates
- Continue or stop?

Elaboration

- Identification of most requirements
- Iterative implementation of the core architecture
- resolution of high risks

UP’s 4 Project Life Cycle Phases (ctd)



inception | Elaboration | Construction | Transition

time

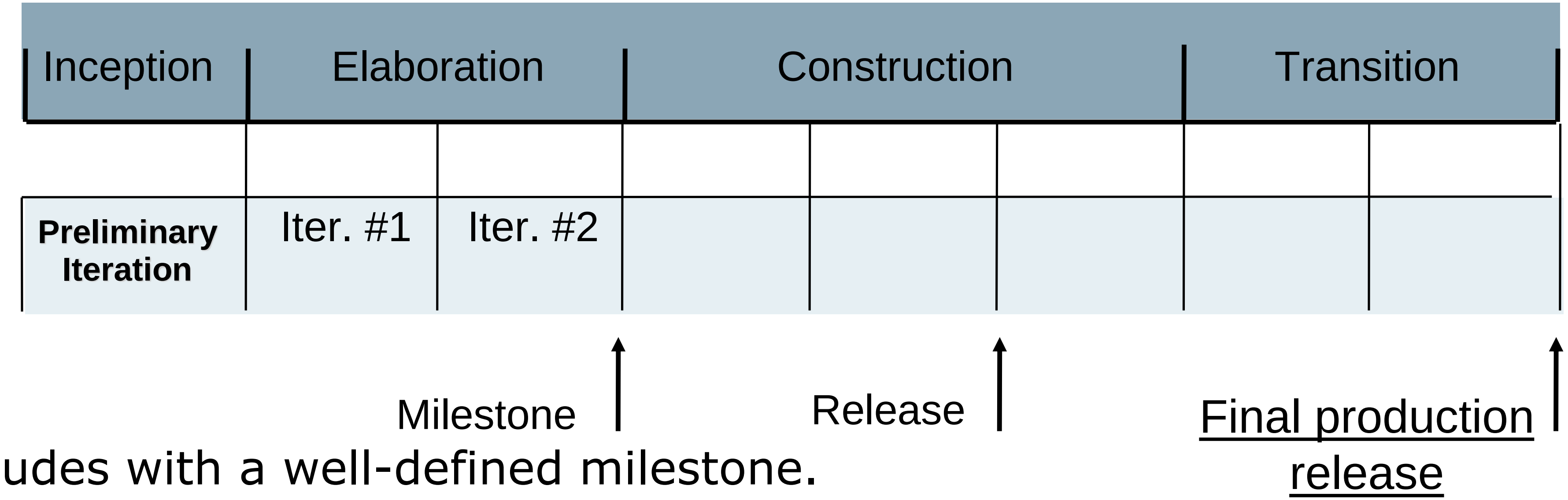
Construction

- Iterative implementation of lower risk elements
- Preparation for deployment

Transition

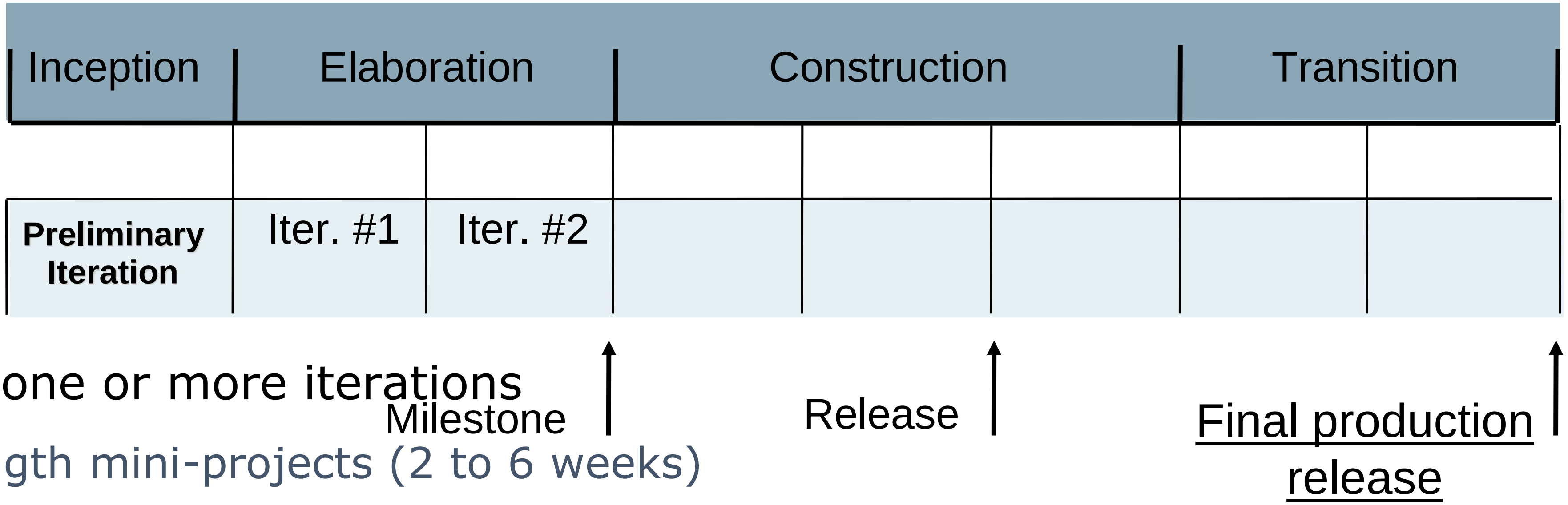
- Beta tests
- Deployment

Iterations and Milestones



Each phase concludes with a well-defined milestone.

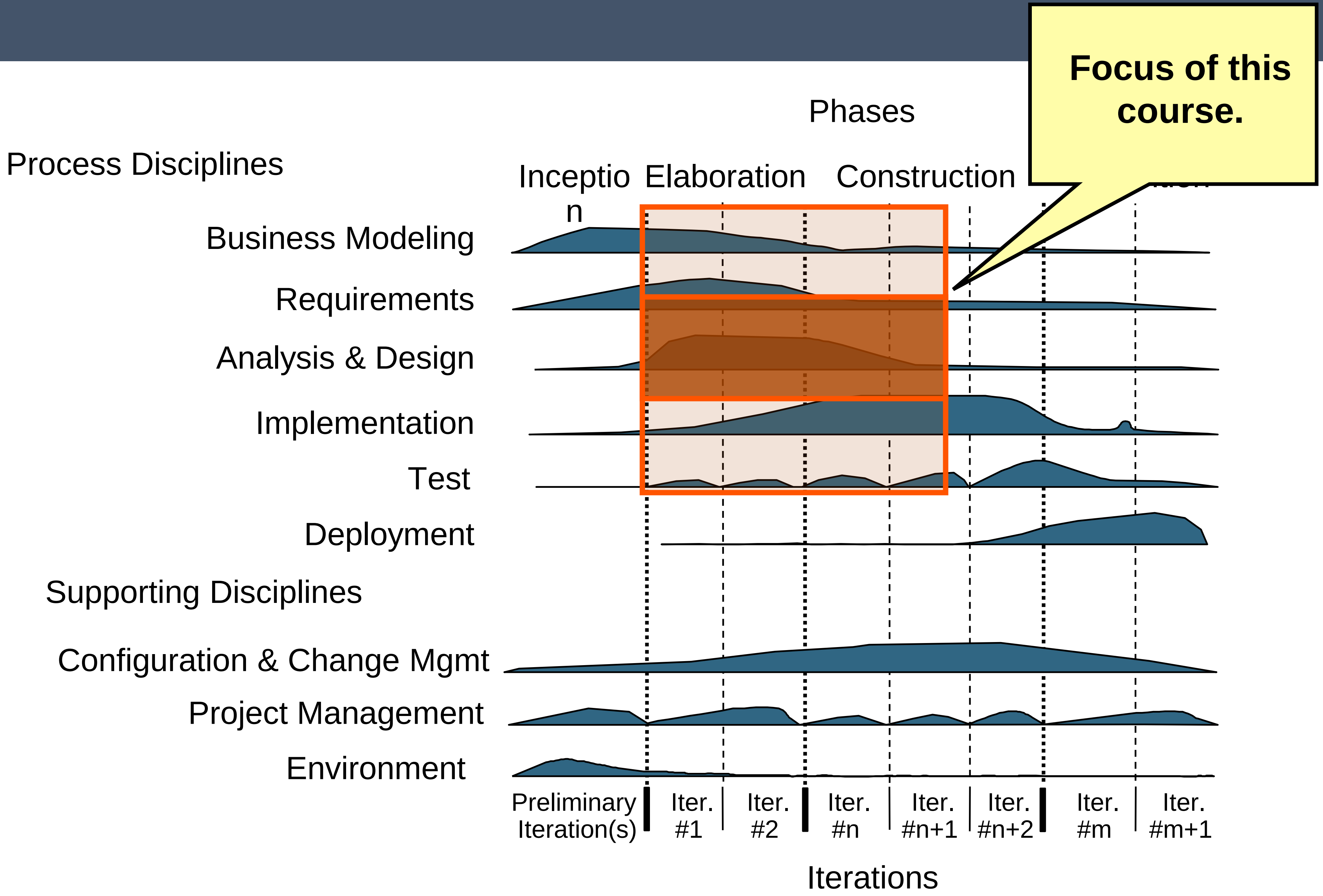
Iterations and Milestones



Phases consist of one or more iterations

- short fixed-length mini-projects (2 to 6 weeks)
- shift tasks to future iterations if necessary ...
- an iteration represents a complete development cycle
- the end of each iteration is a minor release, a stable, integrated executable subset of the final product

The UP Disciplines



Focus of this course.

UP

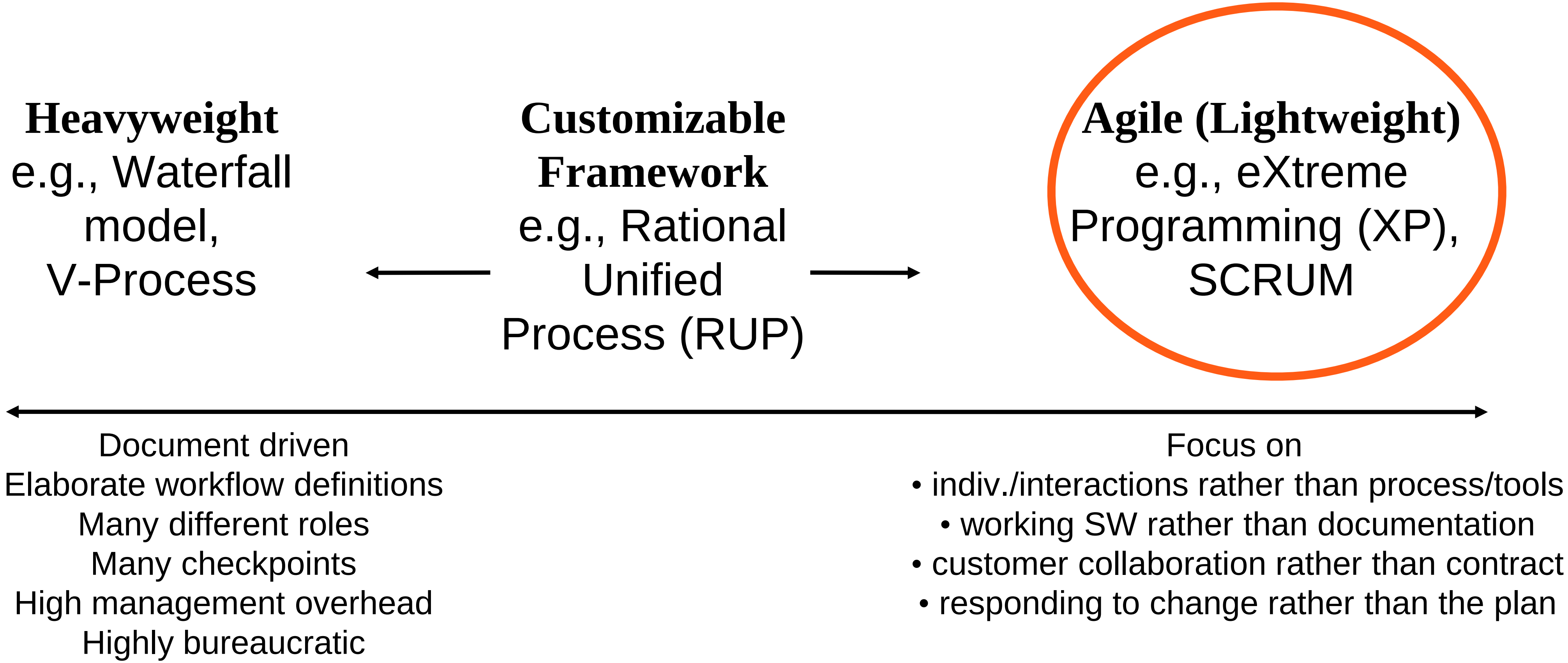
Advantages

- Incremental & Iterative
- Sits in between heavyweight and agile processes
  - best of both worlds ?
- Customizable

Potential pitfalls

- Use Cases do not model all requirements
- Hard to make really lightweight, even when customized
  - Quite some documentation and process remains

Lightweight vs. Heavyweight Processes



Heavyweight
e.g., Waterfall model,
V-Process

Customizable Framework
e.g., Rational Unified Process (RUP)

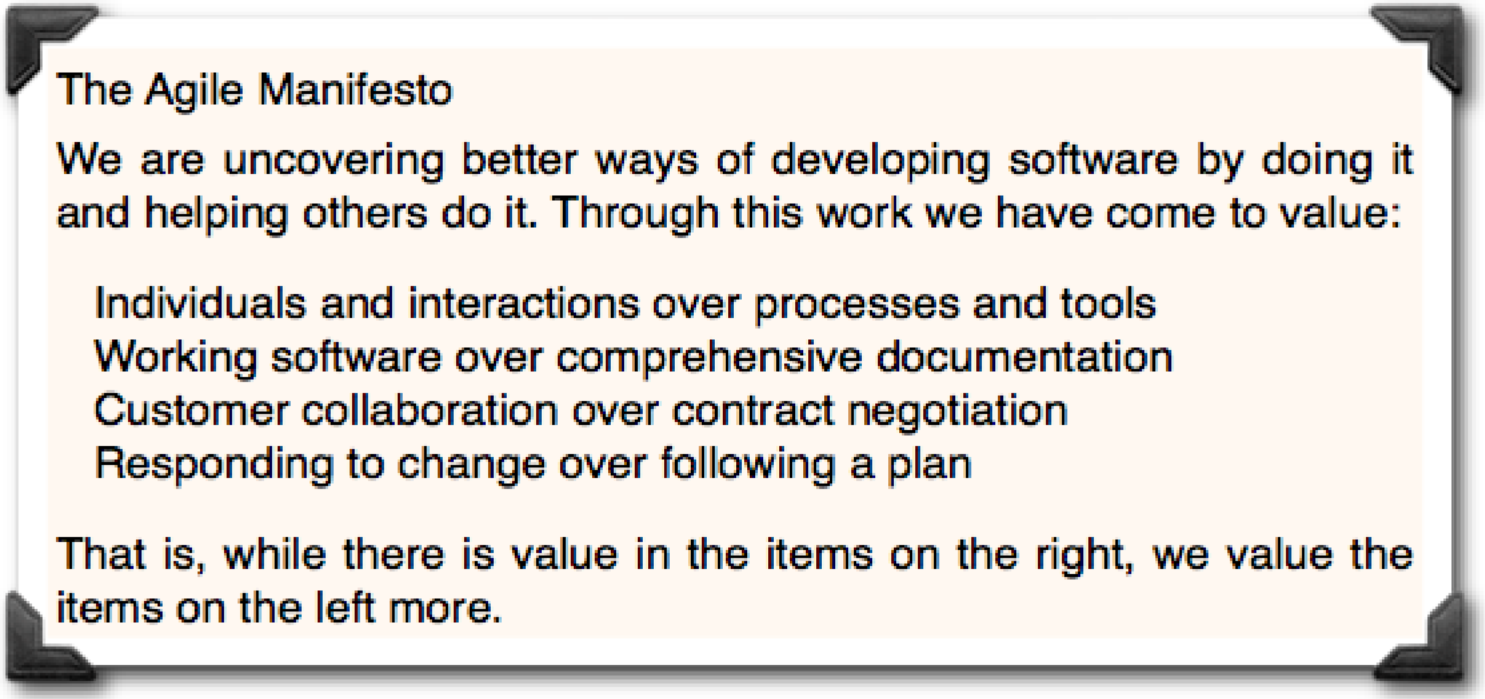
Agile (Lightweight)
e.g., eXtreme Programming (XP),
SCRUM

Document driven
Elaborate workflow definitions
Many different roles
Many checkpoints
High management overhead
Highly bureaucratic

Focus on
- indiv./interactions rather than process/tools
- working SW rather than documentation
- customer collaboration rather than contract
- responding to change rather than the plan

Agile Development

Group of iterative and incremental software methodologies



The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

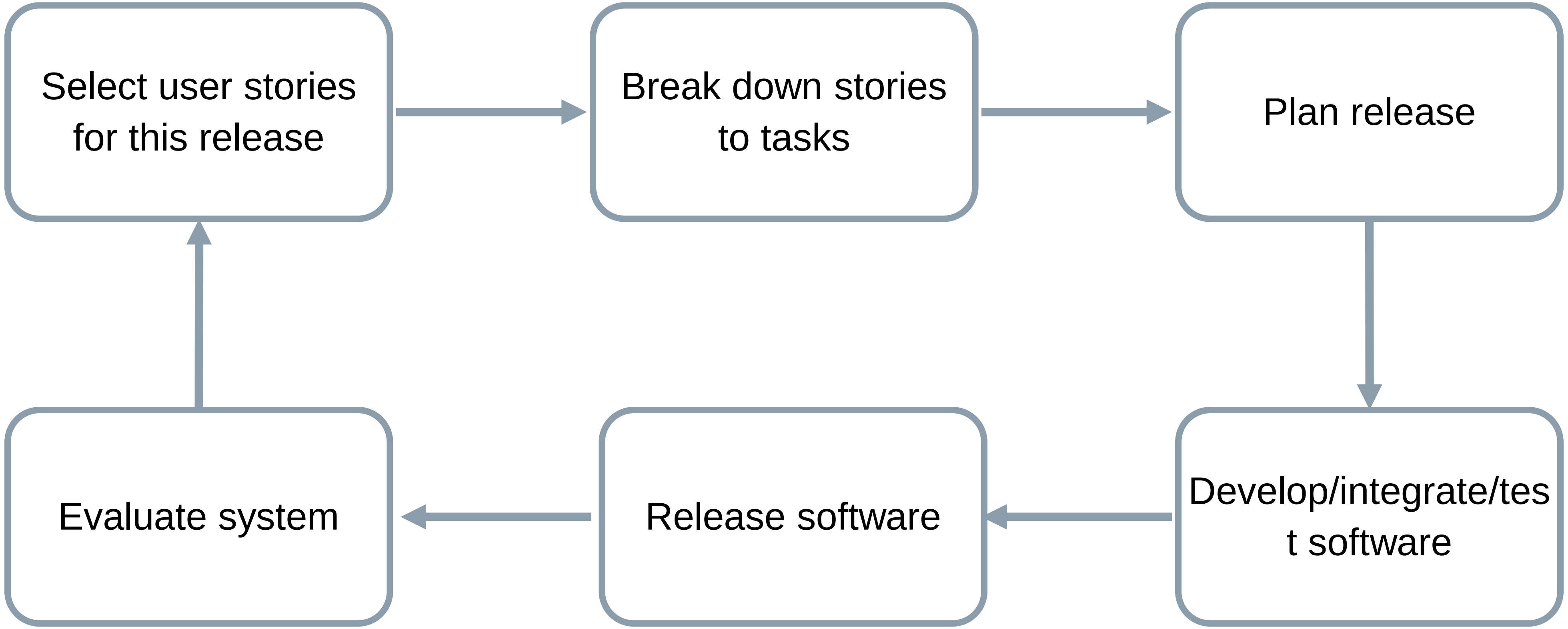
Extreme Programming (XP)

Point of XP: coping with change and uncertainty

Based on number of practices:

- small, frequent releases of the system
- full-time engagement of customer
- pair programming, collective ownership of the code, sustainable development
- regular system releases, test-first development, continuous integration
- constant refactoring, simplest thing that can work

The XP Release Cycle


Select user stories for this release
Break down stories to tasks
Plan release
Develop/integrate/tes t software
Release software
Evaluate system

Source: Sommerville: Software Engineering, 8th edition, 2007

Driving Metaphor

Driving a car is not about pointing the car in one direction and holding to it; driving is about making lots of little course corrections.

> “Do the simplest thing that could possibly work”

Customer-Developer Relationships

A well-known experience: The customer and the developer sit in a boat in the ocean and are afraid of each other

Customer fears	Developer fears
They won't get what they asked for	They won't be given clear definitions of what needs to be done
They must surrender the control of their careers to techies who don't care	They will be given responsibility without authority
They'll pay too much for too little	They will be told to do things that don't make sense
They won't know what is going on (the plans they see will be fairy tales)	They'll have to sacrifice quality for deadlines

Result: a lot of energy goes into protective measures and politics instead of success

The Customer Bill of Rights

You have the right to an overall plan	To steer a project, you need to know what can be accomplished within time and budget
You have the right to get the most possible value out of every programming week	The most valuable things are worked on first.
You have the right to see progress in a running system.	Only a running system can give exact information about project state
You have the right to change your mind, to substitute functionality and to change priorities without exorbitant costs.	Market and business requirements change. We have to allow change.
You have the right to be informed about schedule changes, in time to choose how to reduce the scope to restore the original date.	XP works to be sure everyone knows just what is really happening.

The Developer Bill of Rights

You have the right to know what is needed, with clear declarations of priority.	Tight communication with the customer. Customer directs by value.
You have the right to produce quality work all the time.	Unit Tests and Refactoring help to keep the code clean
You have the right to ask for and receive help from peers, managers, and customers	No one can ever refuse help to a team member
You have the right to make and update your own estimates.	Programmers know best how long it is going to take them
You have the right to accept your responsibilities instead having them assigned to you	We work most effectively when we have accepted our responsibilities instead of having them thrust upon us

Separation of Roles

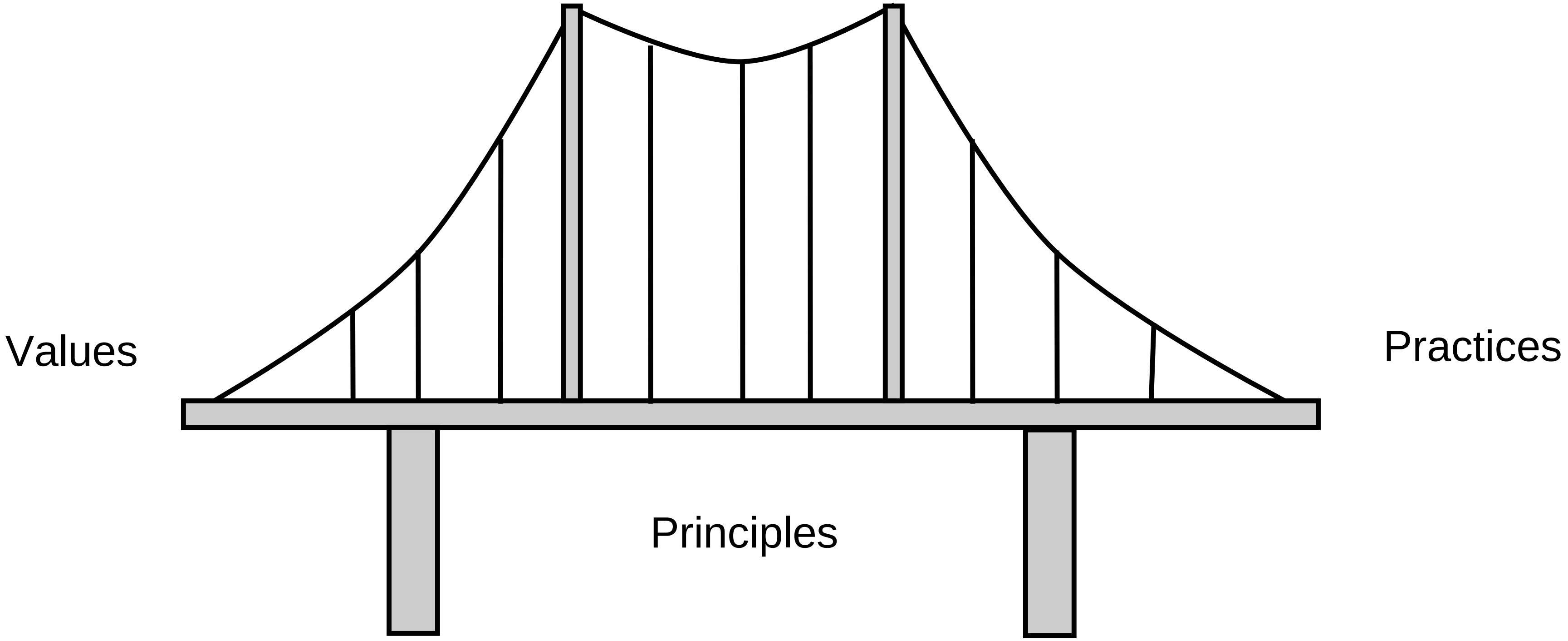
Customer makes business decisions

Developers make technical decisions

The Custome

Business Decisions	Technical Decisions
Scope	Estimates
Dates of the releases	Dates within an iteration
Priority	Team velocity
	Warnings about technical risks

Describing XP



Basic XP Values

Communication

- communicate problems&solutions, teamwork

Simplicity

- eliminate wasted complexity

Feedback

- change creates the need for feedback

Courage

- effective action in the face of fear

Respect

- care about you, the team, and the project

Principles

Humanity, Economics, Mutual Benefit, Self-Similarity, Improvement, Diversity, Reflection, Flow, Opportunity, Redudancy, Failure, Quality, Baby Steps, Accepted Responsibility

Will not detail them -- they govern what the practices tend to accomplish

So, on to the practices!

Primary Practices

Sit Together

Whole Team

Informative Workspace

Energized Work

Pair Programming

Stories

Weekly Cycle

Quarterly Cycle

Slack

Ten Minute Build

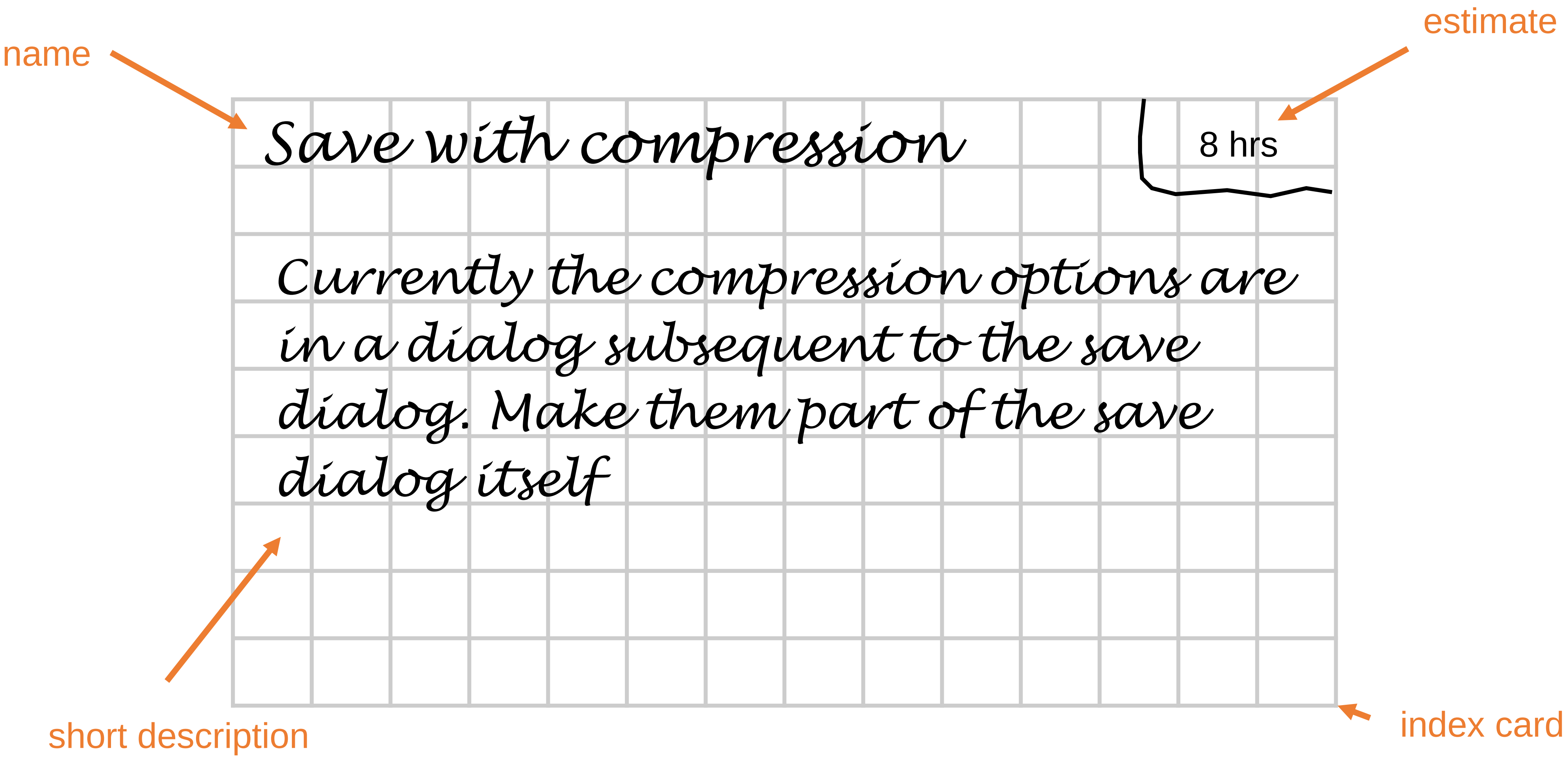
Continuous Integration

Test-First Programming

Incremental Design

Stories

plan using units of customer-visible functionality

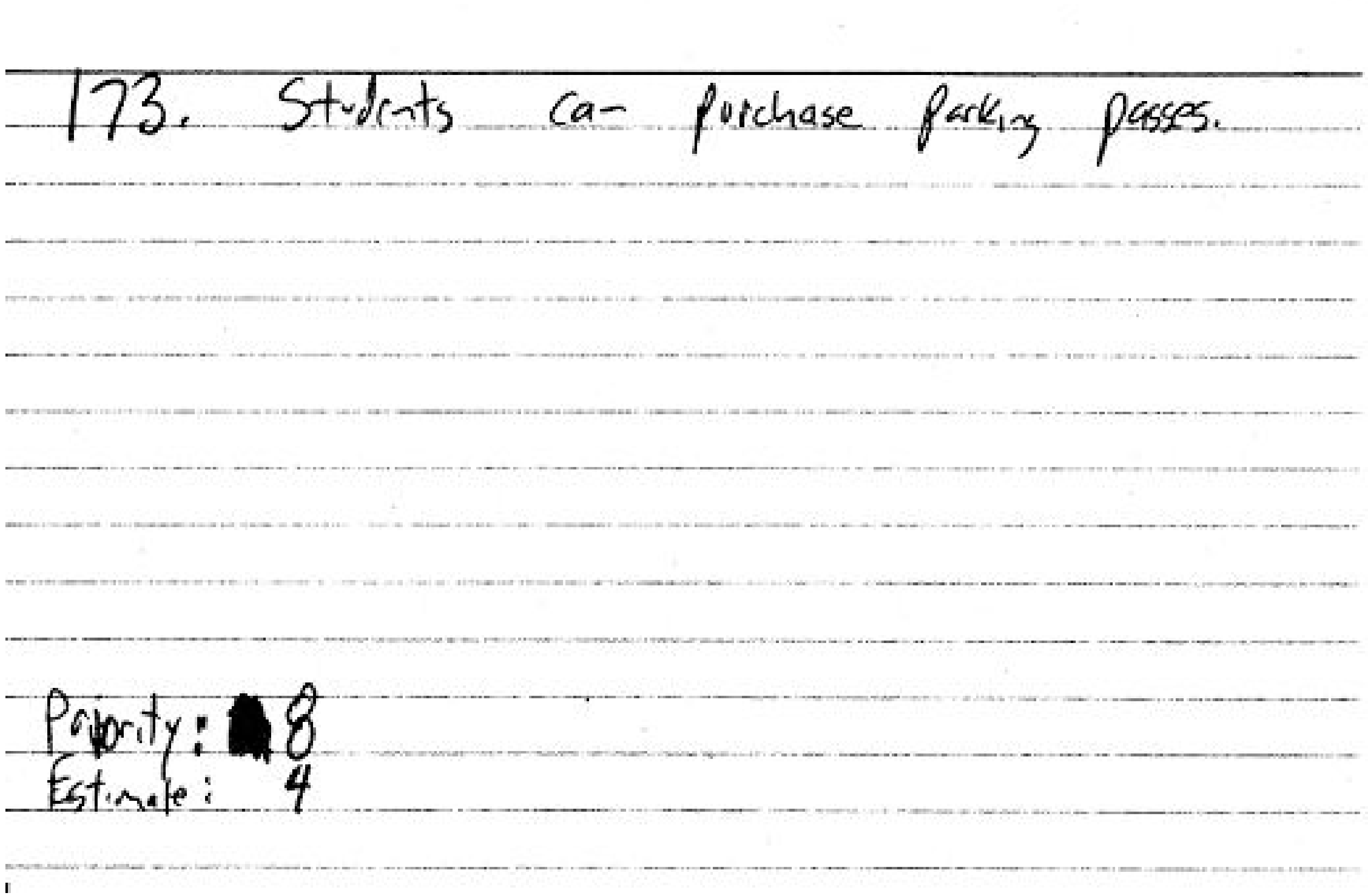


Save with compression

8 hrs

Currently the compression options are in a dialog subsequent to the save dialog. Make them part of the save dialog itself

Another example



173. Students ca- purchase parking passes.

Priority: 8
Estimate: 4

7 more User Stories

Students can purchase monthly parking passes online.

Parking passes can be paid via credit cards.

Parking passes can be paid via PayPal ™.

Professors can input student marks.

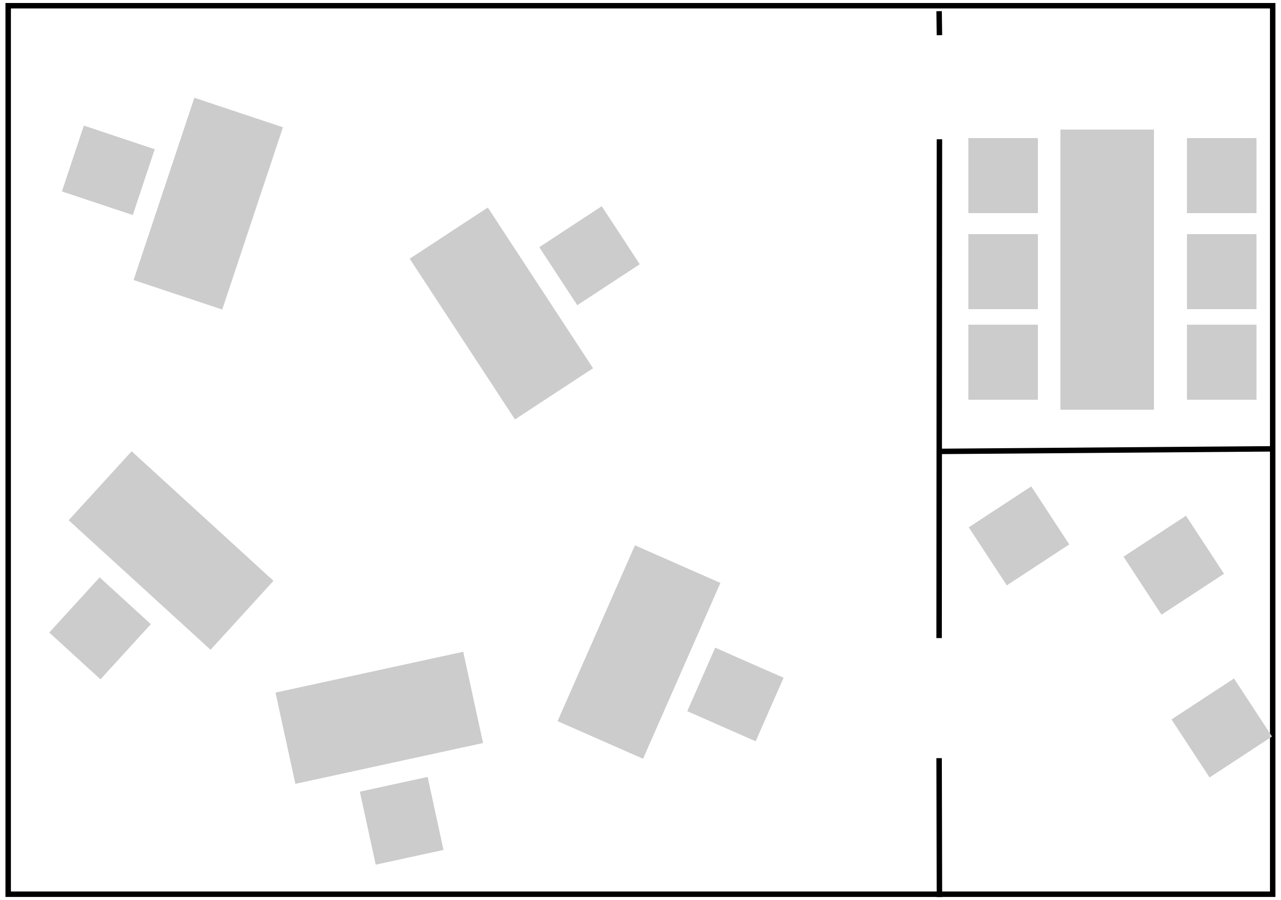
Students can obtain their current seminar schedule.

Students can only enroll in seminars for which they have prerequisites.

Transcripts will be available online via a standard browser.

Sit Together

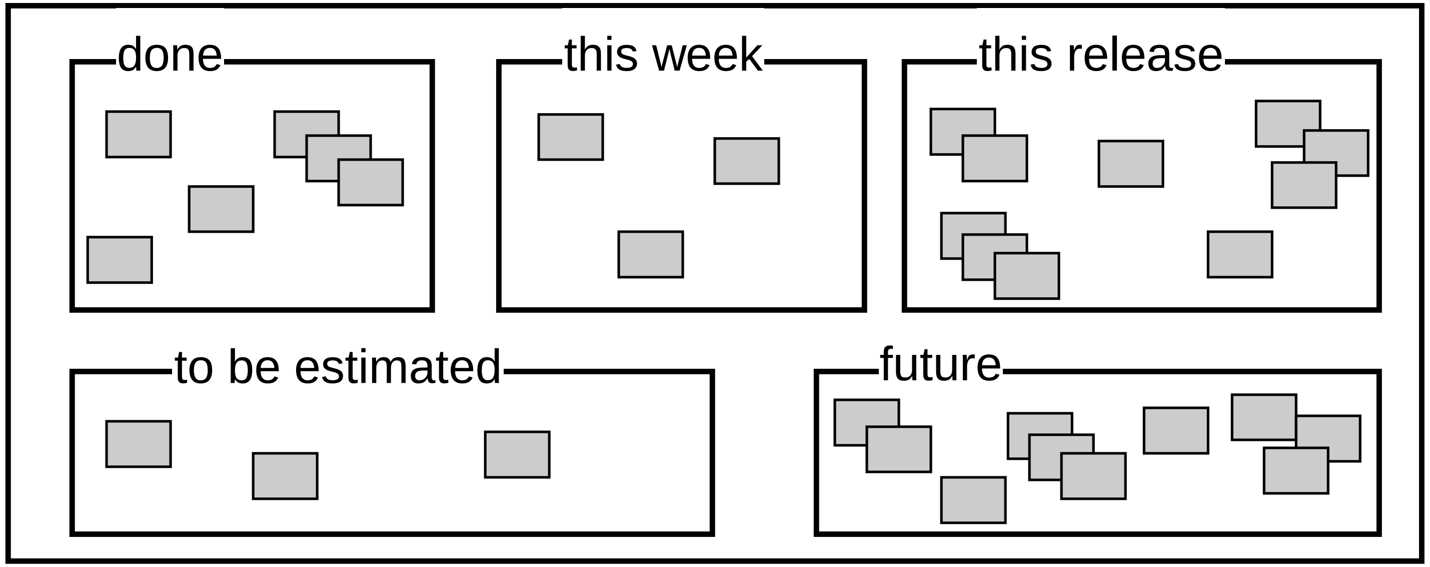
Develop in an open space big enough for the team



Informative Workspace

Workspace = about your work

- 15 seconds to convey how project is going
- shows important, active information
- drinks & snacks available, and clean



Pair Programming

Write all production programs with two people sitting at one machine

- make enough room, move keyboard and mouse

Pair programmers:

- keep each other on task
- brainstorm refinements to the system
- clarify ideas
- take initiative when partner is stuck (less frustration)
- hold each other accountable to practices

Pair programming and privacy

Sometimes you might need some privacy

- then go work alone
- come back with the idea (NOT the code)
    - quickly reimplemented with two
    - benefits the whole team, not you alone

Pair Programming

Rotate pairs frequently

- every couple of hours, at natural breaks in development
- with a timer, every 60 minutes (or 30 minutes for difficult problems)

Pair Programming and Personal Space

Not everybody likes to sit close!

Observe personal hygiene and health

Sexual feelings are not in best interest of the team

- – even when mutual

When uncomfortable pairing with somebody, talk about it with someone safe

- – chances are that you are not the only one
- – everybody needs to feel comfortable

Weekly Cycle

Plan work one week at a time.

Do this on a meeting at the begin of each week

- Review progress.
- Let customers pick a week’s worth of stories to implement this week.
- Break the stories into tasks. Team members sign up for tasks and estimate them.

Start writing tests that will run when the stories are completed

Ten-Minute Build

Automatically build the whole system and run all of the tests in ten minutes

- longer: will not be used (and errors result)
- shorter: not enough time to drink coffee

Note: if it takes longer than 10 minutes:

- maybe only rebuild changed part or test changes
- But: introduces errors. Only do this when necessary

Lowers stress: “Did we make a mistake? Let’s see.”

Continuous Integration

Team Programming = Divide, Conquer, Integrate

Integrate and test changes after no more than a couple of hours

- integration typically takes long
- when done at the end, risks the whole project when integration problems are discovered
- the longer you wait, the more it costs and the more unpredictable it becomes

Using Continuous Integration

Synchronous

- After a task is finished, you integrate and run the tests
- Immediate feedback for you and your partner

Asynchronous

- After submitting changes, the build system notices something new, builds and tests the system, and gives feedback by mail, notification, etc.
- Feedback typically comes when a new task is started
- Pair programmers might have been switched already

Test-first Programming

Write a failing automated test before changing code

Addresses many problems:

- Scope creep: focus coding by what the code should do, not on the “just in case” code
- Coupling and cohesion: If it’s hard to write a test, there is a design problem (not a testing problem)
- Trust: clean working code + automated tests
- Rhythm: gives focus on what to do next
  - efficient rhythm: test, code, refactor, test, ...

Incremental Design

Invest in the design of the system every day. Strive to the design of the system an excellent fit for the needs of the system that day

- Completely opposite to lots of other practices
  - Waterfall and similar approaches

Can work with XP because of the other practices

- Automated tests, continuous integration, ...

Note: you need to invest in design!

- not just implement story after story after story...

Corollary Practices

Real Customer Involvement

Incremental Deployment

Team Continuity

Shrinking teams

Root-Cause Analysis

Shared Code

Code and Tests

Single Code Base

Daily Deployment

Negotiated Scope Contract

Pay-Per-Use

Stages in XP Project

Initiation

- User Stories

Release Planning

Release (each Release is typically 1 -6 months)

- Iteration 1 (typically 1 -3 weeks)
- Iteration 2
- :
- Iteration n

XP

Advantages

- works well for small teams
- low process overhead, lean & mean

Potential pitfalls

- no documented compromises of user conflicts
- lack of an overall design specification or document
- can be hard to fit in organizations/workflows

Conclusion

A software development methodology is

- a set of partially ordered steps
- to build, deploy, maintain, … software

Many methodologies exist

- each with trade-offs
- pick the one according to your needs
  - project size
  - project partners
  - development team(s)
  - outside constraints (legislation, domain constraints, ...)

References

Craig Larman, Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.), Prentice Hall, 2005.

Kent Beck, Extreme Programming Explained: Embrace Change (2nd ed.), 2004.

http://c2.com/cgi/wiki?ExtremeProgramming

Design of Software Systems
(Ontwerp van SoftwareSystemen)

5 Unit Testing, Refactoring and Profiling

Roel Wuyts
2016-2017

A golden rule...

Make it Work

Make it Right

Make it Fast

How does this work?

First make sure the software does what you want

- use unit tests

Then rework the code until it speaks for itself

- use refactorings

Then optimize the performance, if needed

- use profiling

Testing

Unit Testing	test individual components
Module Testing	test a collection of related components
Sub-System Testing	test sub-system interface mismatches
System Testing	• test interactions between sub-systems • tests that the complete system fulfils requirements
Acceptance Testing	test system with real rather than simulated data

Unit Testing

How can I trust that changes did not destroy something?

What is my confidence in the system ?

How do I write tests?

What is unit testing?

Tests

Tests represent your trust in the system

Build them incrementally

- Do not need to focus on everything
- When a new bug shows up: write a test

Even better: test first!

- Act as your first client
- Helps finding proper interfaces

Tests are active documentation: they are always in sync

Testing Style

“The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.”

- write unit tests that thoroughly test a single class
- write tests as you develop (even before you implement)
- write tests for every new piece of functionality

“Developers should spend 25-50% of their time developing tests.”

But I can’t cover anything!

Sure! Nobody can but:

- When someone discovers a defect in your code, first write a test that demonstrates the defect.
- Then debug until the test succeeds.

> “Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.”
>
> Martin Fowler

Unit Testing

Ensure that you get the specified behaviour of the public interface of a class

- Normally tests a single class

General setup of a test:

- Create a context,
- Send a stimulus,
- Check the results

Example

```
public class SaleTest extends TestCase
{
  // …
  public void testMakeLineItem() {
    Sale fixture = new Sale();
    Money total = new Money(7.5);
    Money price = new Money(2.5);
    ItemID id = new ItemID(1);
    ProductDescription desc = new ProductDescription(id, price, “product 1”);

    sale.makeLineItem(desc, 1);
    sale.makeLineItem(desc, 2);

    assertTrue(sale.getTotal().equals(total));
}
```

About Failures and Errors

A failure is a failed assertion

- i.e., an anticipated problem that you test.
    - assertEquals(2, myContainer.nrOfElements())

An error is a condition you didn’t check for.

- e.g. an exception being thrown you did expect

```
boolean isExceptionThrown = false;
try {
     myContainer.get(3);
} catch(IndexOutOfBoundsException e) {
     isExceptionThrown = true;
}
assertTrue(isExceptionThrown);
```

Good Unit Tests

Are repeatable

- have to be deterministic to be useful

Require no human intervention

- so that they can be automated

Are “self-described” and tell a story

- to serve as documentation

Change less often than the system

- they encode stable functionality

Designing tests

Build simple tests

Check that failures are caught

Run tests frequently (every couple of minutes)

Test Infrastructure code first, then application-specific code

Reuse as much test code as you can (tests are code!)

Write small tests that test one particular aspect

Make sure the tests are deterministic

Why spending time testing?

Find problems soon.

- in context of what you were doing!

Serve as documentation.

Ease maintenance and evolution.

- new developers jump in anytime..

Have something to show all the time.

Testing Frameworks

Tests have to be repeatable

Unit Testing Frameworks implement necessary infrastructure so that you can set up your tests, run them frequently, and see the results

SUnit is “the mother of all unit test frameworks”

- started in Smalltalk
- fanned out to all kinds of other languages
  - JUnit, NUnit, CppUnit, ...

JUnit overview

Junit (inspired by Sunit) is a simple “testing framework” that provides:

- classes for writing Test Cases and Test Suites
- methods for setting up and cleaning up test data (“fixtures”)
- methods for making assertions
- textual and graphical tools for running tests

Testing Frameworks

Key parts

- TestCase: bundles test methods
- Some mechanism to execute test code
- (methods, macroes, ...)
- Fixture (≈ Resource): known set of objects that serves as a base for a set of test cases
- TestSuite: bundles testcases so that they can be run together
- TestRunner: runs a testsuite, outputting results

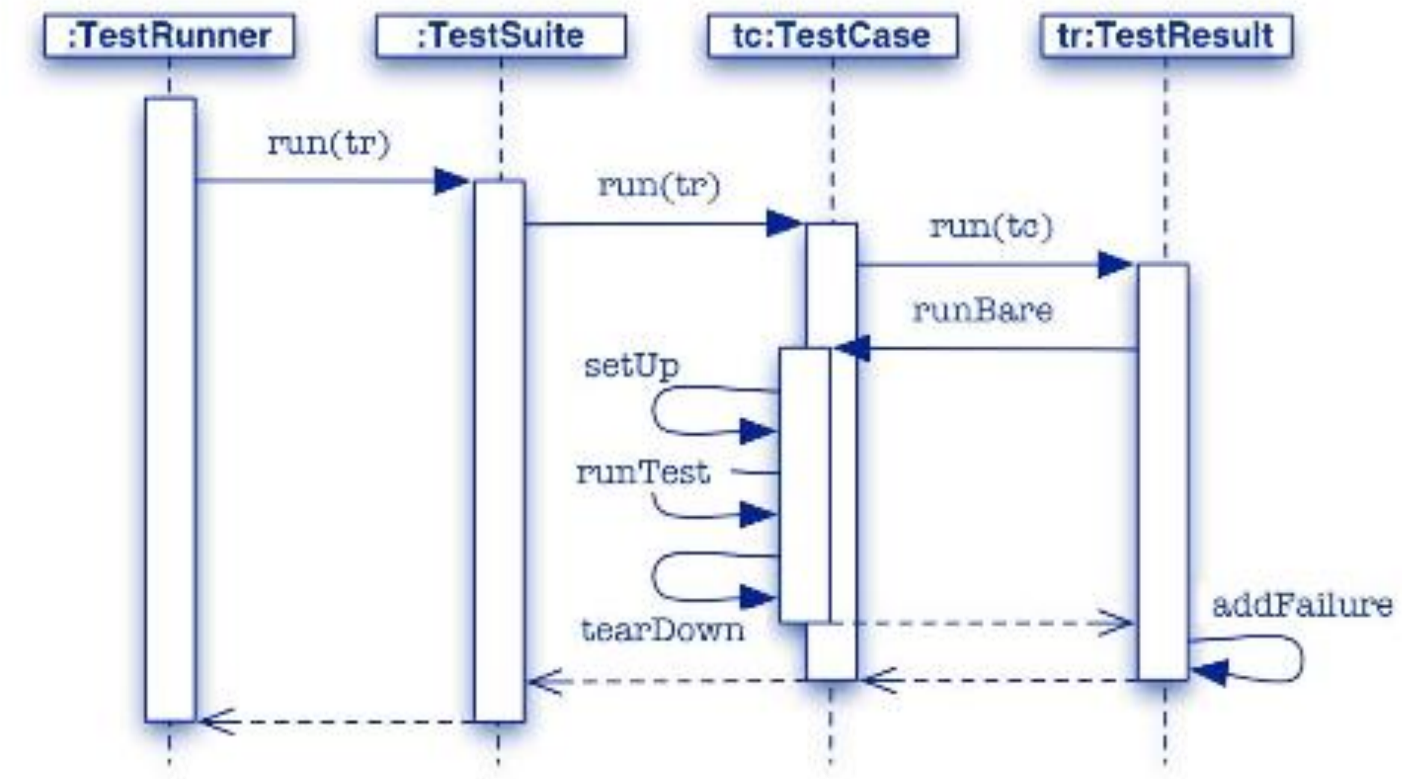
A testing scenario

The framework calls the test methods that you define for your test cases

- You need to declare a TestRunner
- You specify who will gather the results
- You add the needed tests to the runner
- You run the TestRunner
  - this automatically runs all tests, collecting the results
- You pass the results to an Outputter

A testing scenario

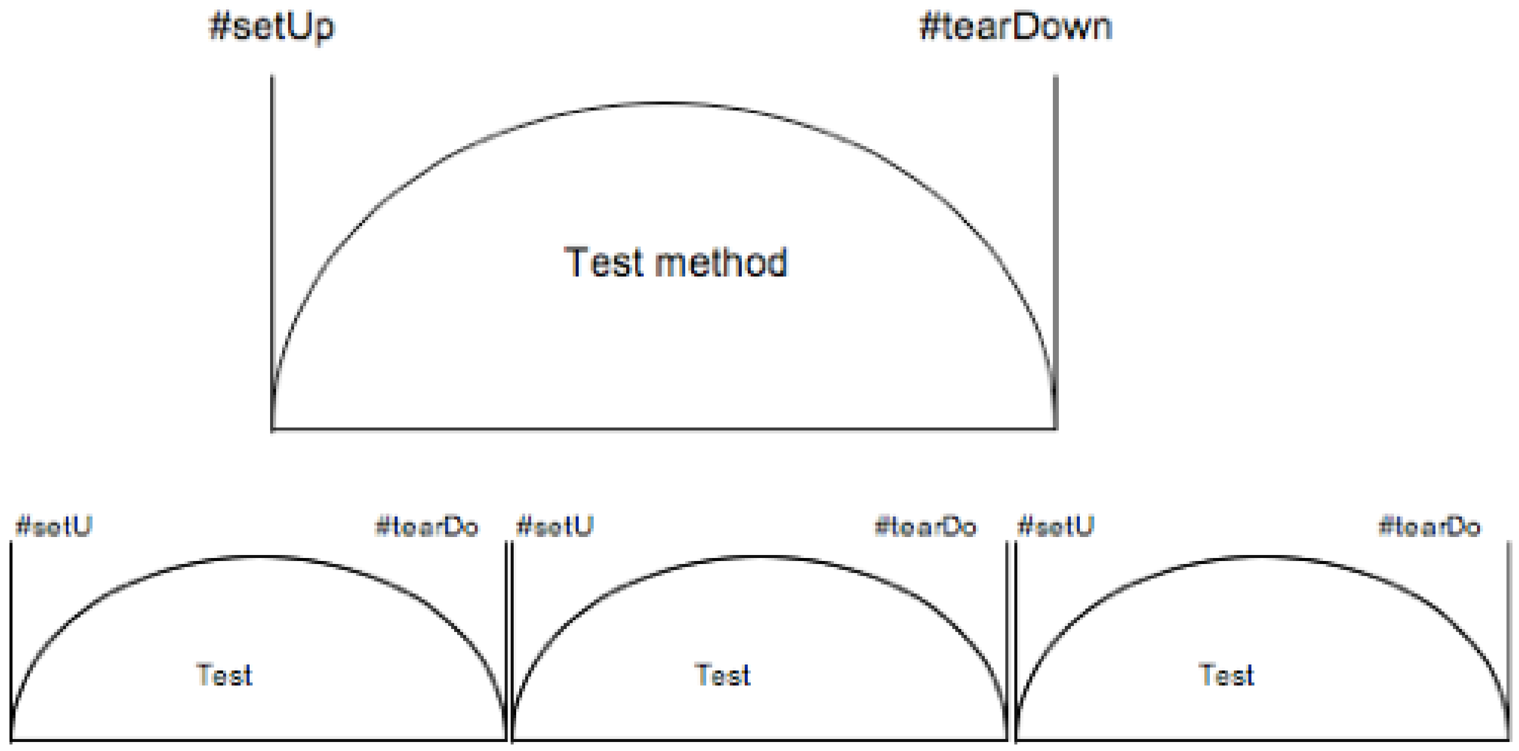
The framework calls the test methods that you define for your test cases



Setup and TearDown

Executed before and after each test

- – setUp allows us to specify and reuse the context
- – tearDown makes us clean-up afterwards



Example unit test for an online ordering system

Mocking & Stubbing

Example unit test for an online ordering system

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
```

Mocking & Stubbing

Example unit test for an online ordering system

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
```

Collaborator (wharehouse)

tested object

“system under test” (SUT)

state verification

Mocking & Stubbing

Using mocking (jMock library example)

```
public class OrderInteractionTester extends MockObjectTestCase {

  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    Order order = new Order(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    order.fill((Warehouse) warehouseMock.proxy());

    warehouseMock.verify();
    assertTrue(order.isFilled());
  }

}
```

setup - data

setup - expectations

exercise

verify

More info: http://martinfowler.com/articles/mocksArentStubs.html

Refactorings

Refactoring

- What is it?
- Why is it necessary?
- Examples
- Tool support
- Obstacles to refactoring

What is Refactoring?

The process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure [Fowl99a]

A behaviour-preserving source-to-source program transformation [Robe98a]

A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck99a]

Typical Refactorings

Class Refactorings	Method Refactorings	Attribute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Why Refactoring?

“Grow, don’t build software” (Fred Brooks)

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” (Fowler)

Some argue that good design does not lead to code needing refactoring ...

Why Refactoring?

In reality

- Extremely difficult to get the design right the first time
- You cannot fully understand the problem domain
- You cannot fully understand user requirements
- You cannot really plan how the system will evolve
- Original design is often inadequate
- System becomes brittle, difficult to change

Why Refactoring?

Refactoring helps you to

- Manipulate code in a safe environment
  - Behaviour preserving
- Recreate a situation where evolution is possible
- Understand existing code

Remember: software needs to be maintained

- This is one way to do it safely

Examples of Refactoring Analysis

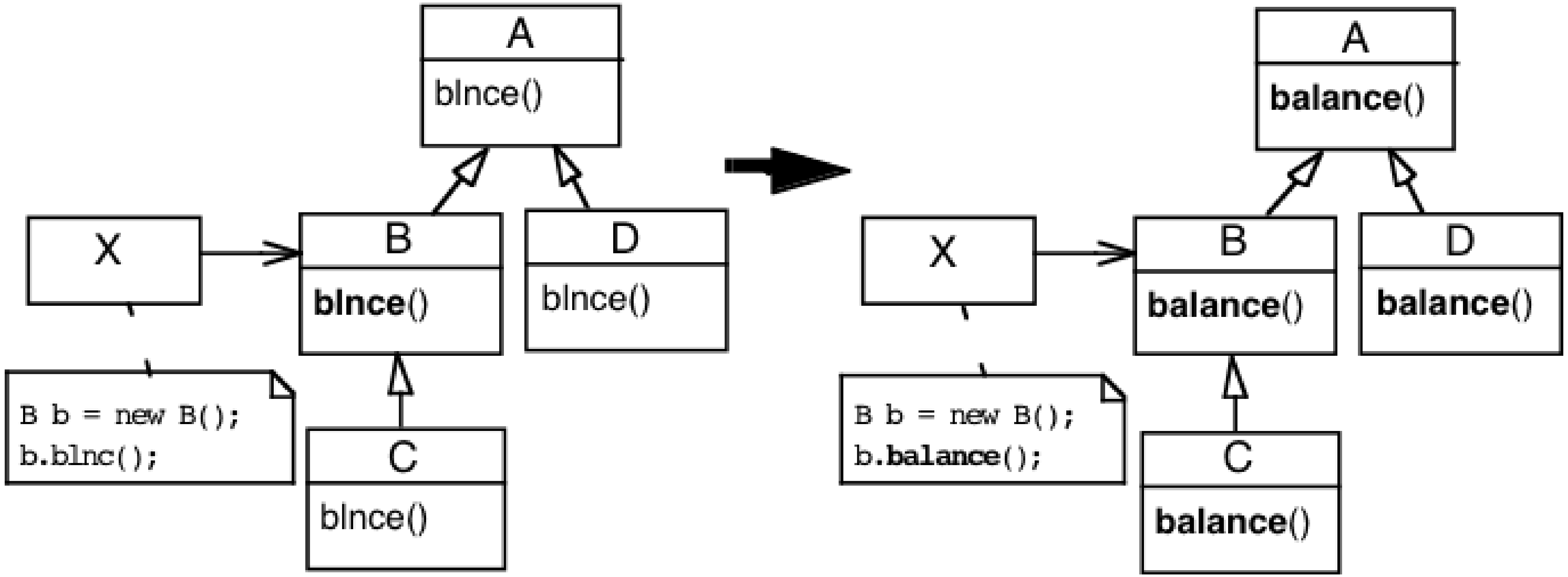
Rename Method

- – existence of similar methods
- – references of method definitions
- – references of calls

AddClass

- – simple
- – namespace use and static references between class structure

Rename Method



Rename Method: Do It Yourself

Check if a method does not exist in the class and superclass/subclasses with the same “name”

Browse all the implementers (method definitions)

Browse all the senders (method invocations)

Edit and rename all implementers

Edit and rename all senders

Remove all implementers

Test

Rename Method

Rename Method (method, new name)

Preconditions

- no method exists with the signature implied by new name in the inheritance hierarchy that contains method
- [Smalltalk] no methods with same signature as method outside the inheritance hierarchy of method
- [Java] method is not a constructor

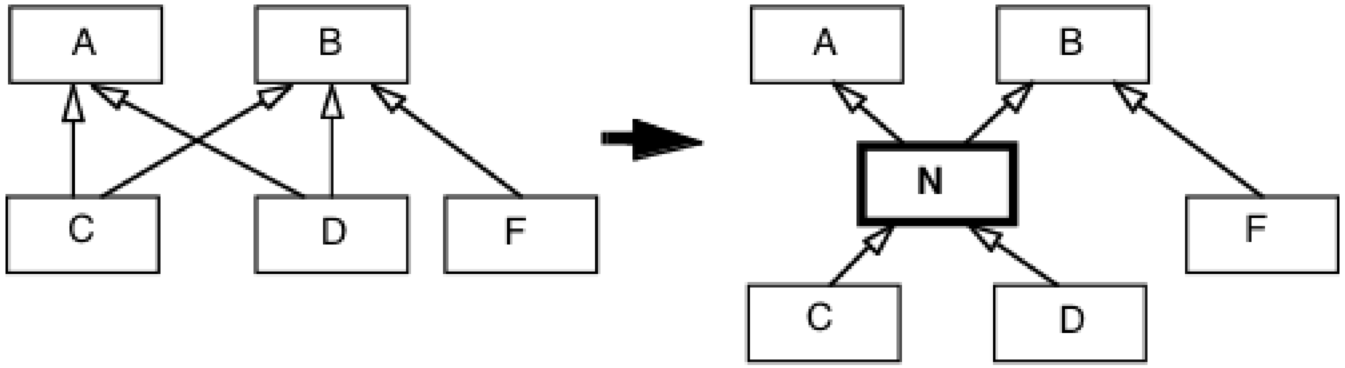
PostConditions

- method has new name
- relevant methods in the inheritance hierarchy have new name
- invocations of changed method are updated to new name

Other Considerations

- Typed/Dynamically Typed Languages => Scope of the renaming

Add class



Add Class

Preconditions

- no class and global variable exists with classname in the same scope
- subclasses are all subclasses of all superclasses
- [Smalltalk] superclasses must contain one class
- [Smalltalk] superclasses and subclasses cannot be metaclasses

Postconditions

- new class is added into the hierarchy with superclasses as superclasses and subclasses as subclasses
- new class has name classname
- subclasses inherit from new class and not anymore from superclasses

Considerations: Abstractness

Tool Support

Could do refactoring by hand

- see Rename Method example

But much better if automated

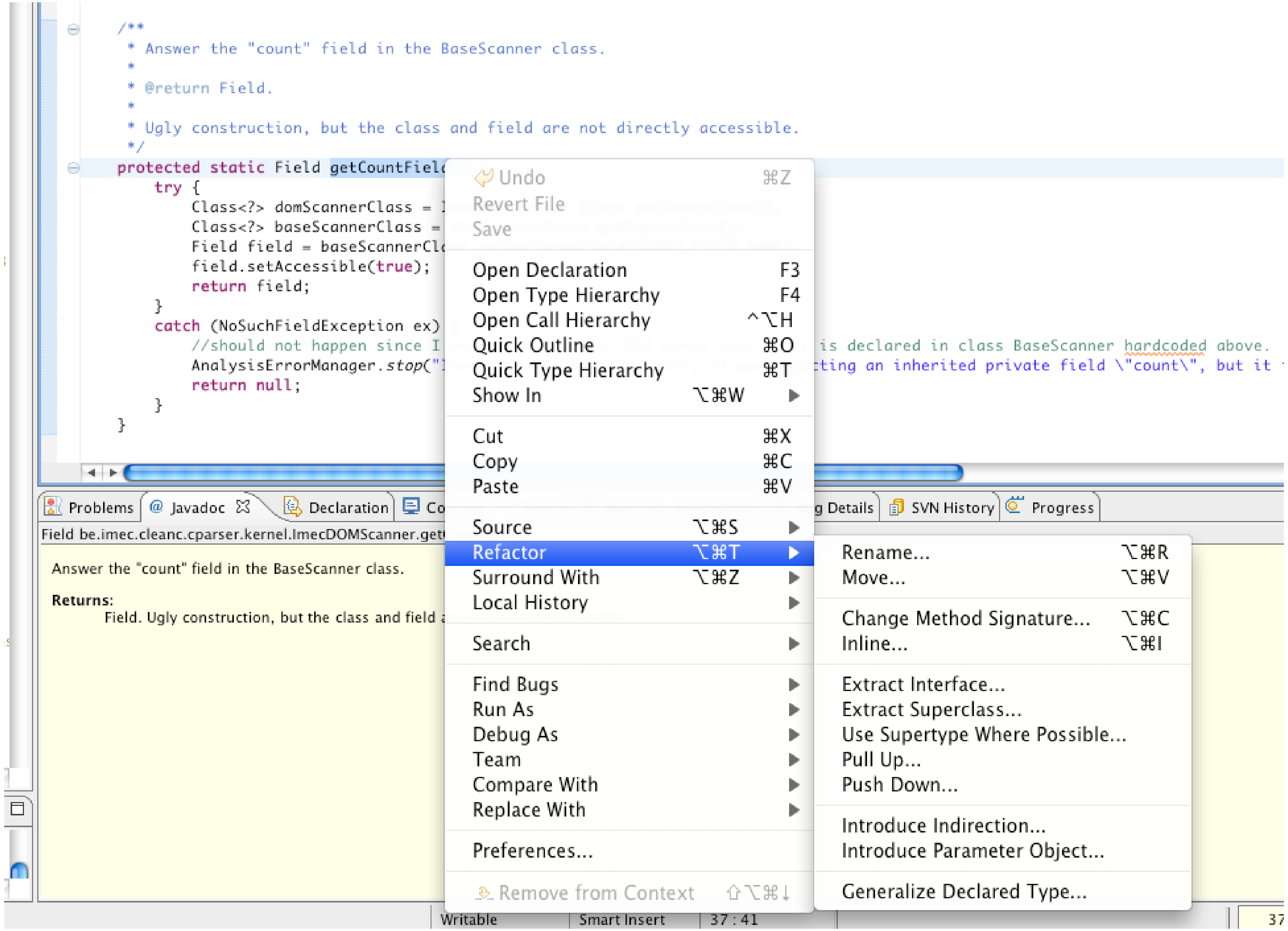
- easier
- safer

Which tools are needed to support refactoring?

Tool support for refactoring activities

Change Efficiently	Failure Proof
Refactoring Tools - source-to-source program transformation - behaviour preserving ⇒ Improve Structure	Regression Testing - Repeating past tests - requires no user interaction - is deterministic ⇒ Verify damage to previous work
Development Environment - Fast edit-compile-run - Integrated in environment ⇒ Convenient	Configuration&Version Management - track different versions - track who did what ⇒ can revert to earlier versions

Refactoring in Eclipse



```
/**
 * Answer the "count" field in the BaseScanner class.
 *
 * @return Field.
 *
 * Ugly construction, but the class and field are not directly accessible.
 */
protected static Field getCountFiel
    try {
        Class<?> domScannerClass = I
        Class<?> baseScannerClass =
        Field field = baseScannerCl
        field.setAccessible(true);
        return field;
    }
    catch (NoSuchFieldException ex)
        //should not happen since I                              is declared in class BaseScanner hardcoded above.
        AnalysisErrorManager.stop("I                             ting an inherited private field \"count\", but it
        return null;
    }
}
```

Field be.imec.cleanc.cparser.kernel.ImecDOMScanner.get

Answer the "count" field in the BaseScanner class.

Returns:
Field. Ugly construction, but the class and field a

When to Refacctor ?

When you add functionality

- Helps you to understand the code you are modifying.
- Sometimes the existing design does not allow you to easily add the feature.

When you need to fix a bug

- If you get a bug report, it’s a sign the code needs refactoring
- because the code was not clear enough for you to see the bug in the first place

When you do a code review

- Code reviews help spread knowledge through the development team.
- Works best with small review groups

When to Refactor

You should refactor:

- Any time that you see a better way of doing things
  - “Better” means making the code easier to understand and to modify in the future
- You can do so without breaking the code
  - Unit tests are essential for this (remember: do not refactor in isolation)

You should NOT refactor:

- Stable code (code that won’t ever need to change, code library)
- Someone else’s code
  - Unless you’ve inherited it (and now it’s yours) ← ≉ XP practice!

Rule of Thumb: ‘Three strikes and you refactor’

- 1st time: Write from scratch
- 2nd time: Duplication eventually admissible
- 3rd time: Refactor !!!

Example: Switch Statements

Switch statements are very rare in properly designed object-oriented code

- Therefore, a switch statement is a simple and easily detected “bad smell”
- Of course, not all uses of switch are bad
- A switch statement should NOT be used to distinguish between various kinds of object

There are several well-defined refactorings for this case

- The simplest is the creation of subclasses

Example: Bad Smell

```
class Animal {
   final int MAMMAL = 0, BIRD = 1, REPTILE = 2;
   int myKind;  // set in constructor
   ...

   String getSkin() {
      switch (myKind) {
         case MAMMAL: return "hair";
         case BIRD: return "feathers";
         case REPTILE: return "scales";
         default: return "integument";
      }
   }
}
```

Example: Improved

```
class Animal {
        String getSkin() {
            return "integument";
        }
}

class Mammal extends Animal {
        String getSkin() {
            return "hair"; }
        }

class Bird extends Animal {
        String getSkin() {
            return "feathers";
        }
}

class Reptile extends Animal {
        String getSkin() {
            return "scales";
        }
}
```

JUnit Tests

As we refactor, we need to run (JUnit) tests to ensure that we haven’t introduced errors

```
public void testGetSkin() {
        assertEquals("hair", myMammal.getSkin());
        assertEquals("feathers", myBird.getSkin());
        assertEquals("scales", myReptile.getSkin());
        assertEquals("integument", myAnimal.getSkin());
}
```

This should work equally well with either implementation

The setUp() method of the test fixture may need to be modified

Re-running unit tests proves that the refactoring succeeded
(= external behavior remained unchanged)

Refactoring Examples

Add Parameter

Change Association

Change Reference to Value

Change Value to Reference

Collapse Hierarchy

Consolidate Conditional

Convert Procedures to Objects

Decompose Conditional

Encapsulate Collection

Encapsulate Downcast

Encapsulate Field

Extract Class

Extract Interface

Extract Method

Extract Subclass

Extract Superclass

Form Template Method

Hide Delegate

Hide Method

Inline Class

Inline Temp

Introduce Assertion

Introduce Explain Variable

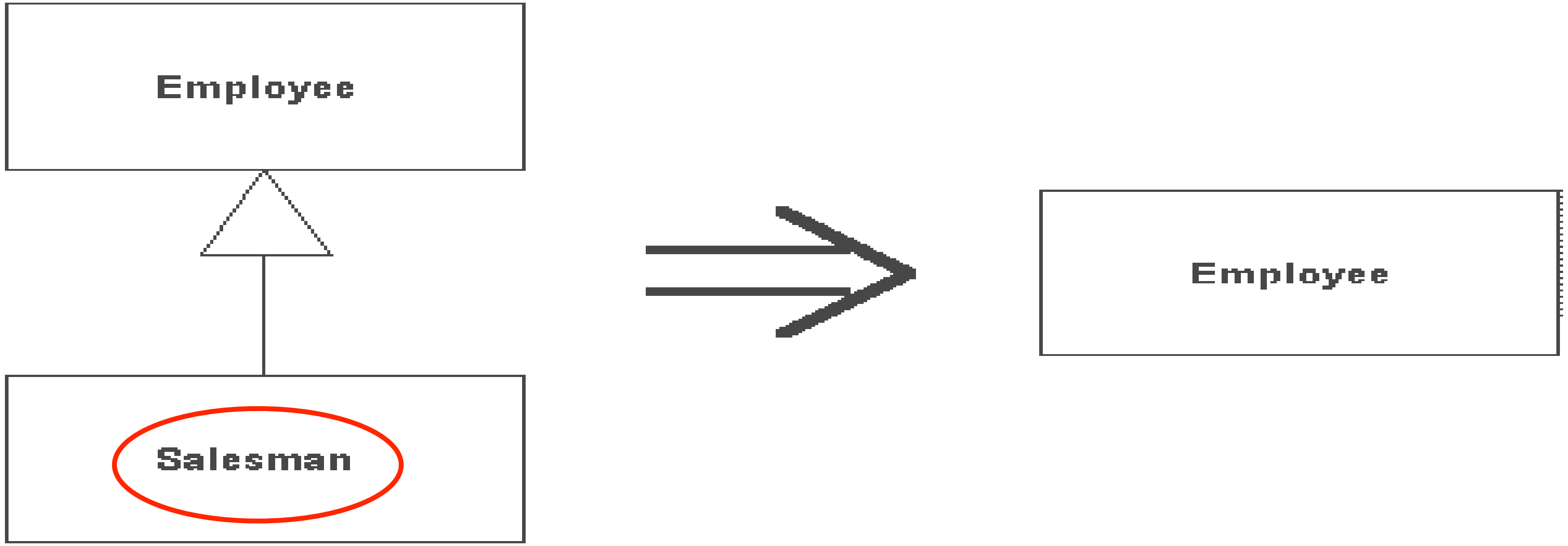
Introduce Foreign Method

…

72 Refactorings identified by Fowler

Refactoring Example: Collapse Hierarchy

When superclass and subclass are not very different: Merge them



Refactoring Example: Consolidate Conditional

When the same fragment of code is in all branches: Move it out

```
double disabilityAmount()
 {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12)
       return 0;
    if (_isPartTime) return 0;
   // compute the disability amount
}
```

```
double disabilityAmount()
 {
    if (isNotEligableForDisability())
       return 0;
     // compute the disability amount
 }
```

Refactoring Example: Decompose Conditional

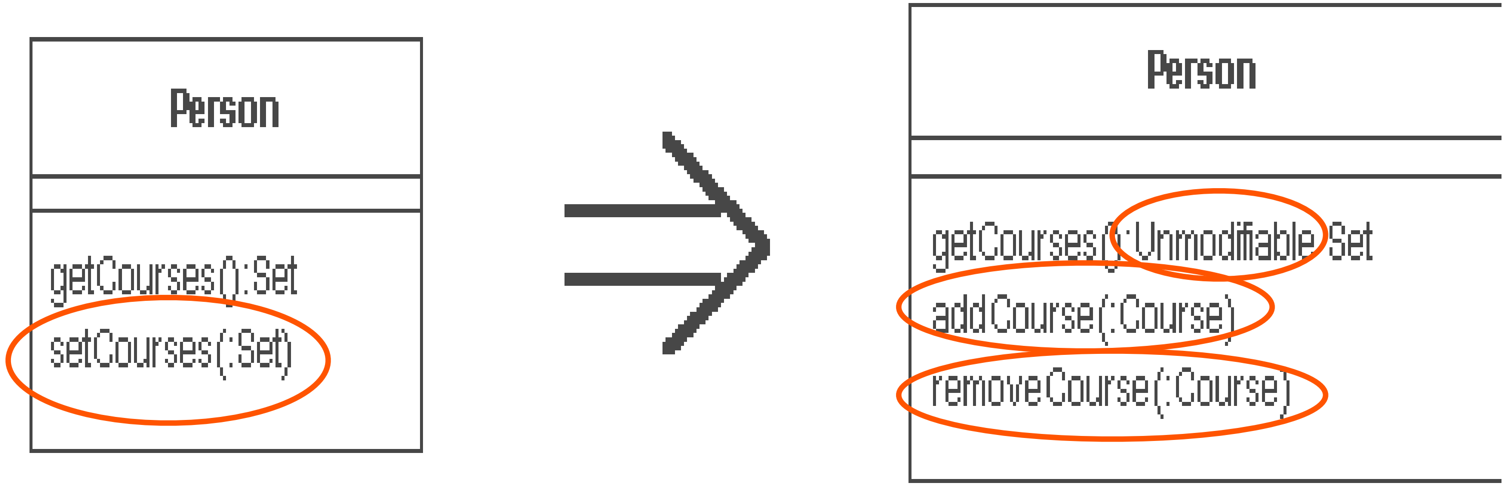
When having a complicated conditional statement: Extract if/then/else parts

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
        charge = quantity * _winterRate + _winterServiceCharge;
else
        charge = quantity * _summerRate;
```

```
if ( notSummer(date) )
        charge = winterCharge (quantity);
else charge = summerCharge (quantity);
```

Refactoring Example: Encapsulate Collection

When a method returns a collection: Provide Read-only view & add/remove methods

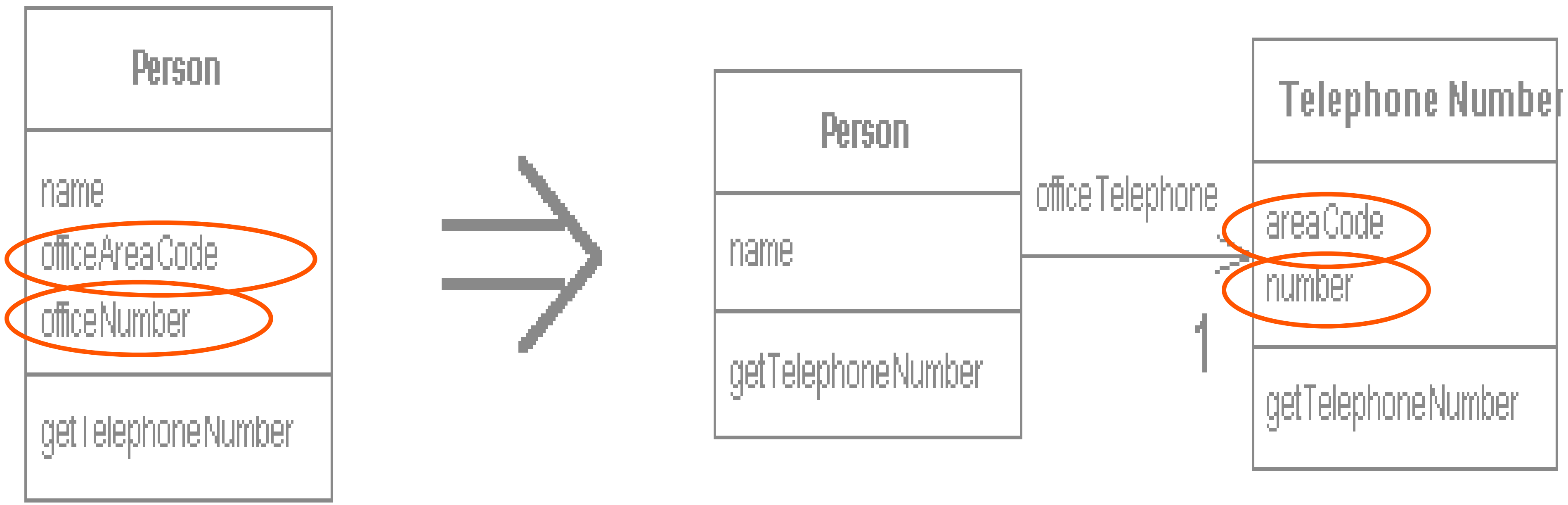


Refactoring Example: Extract Class

When we have 1 class doing the work that should be done by 2:

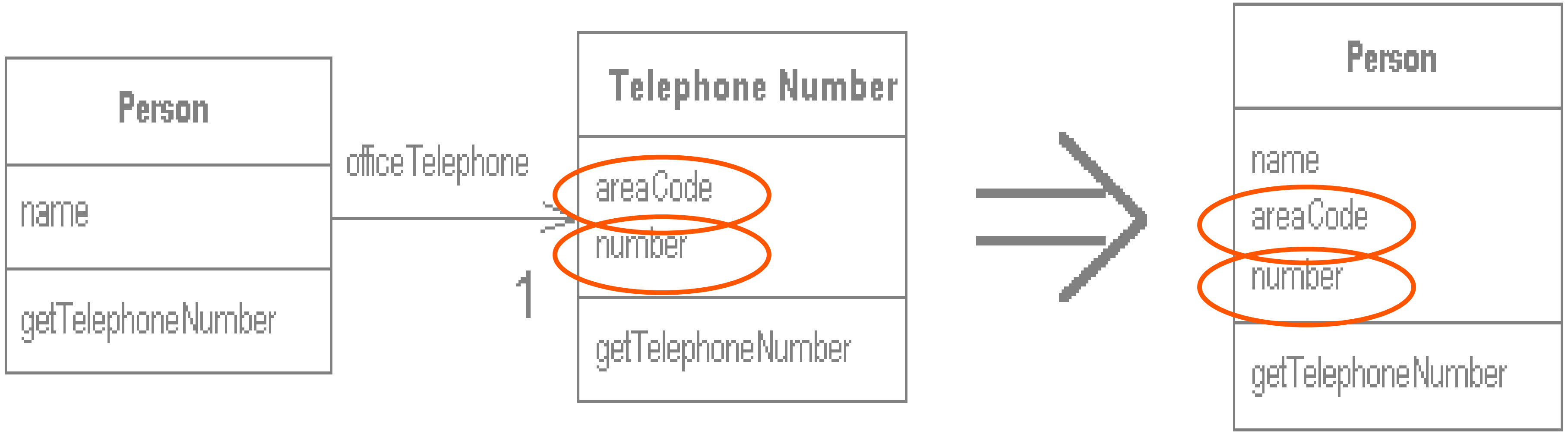
Create new class, move fields & methods

=> GRASP High Cohesion



Refactoring Example: Inline Class

When a class isn't doing very much: Merge with other class



Refactoring Example: Encapsulate Downcast

When a method returns an object that needs to be downcasted by its callers:

- – Move the downcast to within the method.
- – happens often when a class uses a collection or iterator

```
Object lastReading()  {
  return readings.lastElement();
}

Reading lastReading =
  (Reading) theSite.lastReading();
```

```
Reading lastReading()  {
  return (Reading) readings.lastElement();
}

Reading lastReading = theSite.lastReading();
```

Refactoring Example 9: Extract Method

When we have a code fragment that can be grouped together: turn the fragment into a method with an explanative name

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

```
void printOwing() {
   printBanner();
   printDetails(getOutstanding());
}
```

Bad Smells in Code

Duplicated Code

Long Method

Large Class

Long Parameter List

Divergent Change

Shotgun Surgery

Feature Envy

Data Clumps

Primitive Obsession

Switch Statements

Comments

Parallel Inheritance/Interface Hierarchies

Lazy Class

Speculative Generality

Temporary Field

Message Chains

Middle Man

Inappropriate Intimacy

Incomplete Library Class

Data Class

Refused Bequest

Alternative Classes with Different Interfaces

Bad Smells

Where did this term come from?

> "If it stinks, change it."
> --Grandma Beck

The basic idea is that there are things in code that cause problems

- Duplicated code, Long methods, …

But any time you change working code, you run the risk of breaking it

- A good test suite makes refactoring much easier and safer

Bad smells gives inspiration, but are not designed as metrics

- You have to decide yourself when something is "too much", …

Example: Duplicated Code

If you see the same code structure in more than one place, find a way to unify them

“Number one in the stink parade” !!!

The usual solution is to perform

- ExtractMethod: create a single method from the duplicated code
- Invoke from all places: Use it wherever needed
- You sometimes need additional refactorings (Add Parameter, …)

This adds the overhead of method calls, thus the code could get a bit slower

Other Bad Smells

Long Method

- The longer a procedure is, the more difficult it is to understand.
- Solution: perform EXTRACT METHOD or Decompose Conditional or Replace Temp with Query.

Large class

- When a class is trying to do too much, it often shows up as too many instance variables.
- Solution: perform EXTRACT CLASS or EXTRACT SUBCLASS

Feature Envy

- A method that seems more interested in a class other than the one it is in.
- Solution: perform MOVE METHOD or EXTRACT METHOD on the jealous bit and get it home.

Other Bad Smells

Shotgun Surgery

- This situation occurs when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
- Solution: perform MOVE METHOD/FIELD or INLINE CLASS bring a whole bunch of behavior together.

Long Parameter List

- In OO, you don't need to pass in everything the method needs.
  Instead, you pass enough so the method can get to everything it needs
- Solution: Use REPLACE PARAMETER WITH METHOD when you can get the data in one parameter by making a request of an object you already know about.

Bad Smell/Sweet Smell: Comments

Fowler says “comments often are used as a deodorant”

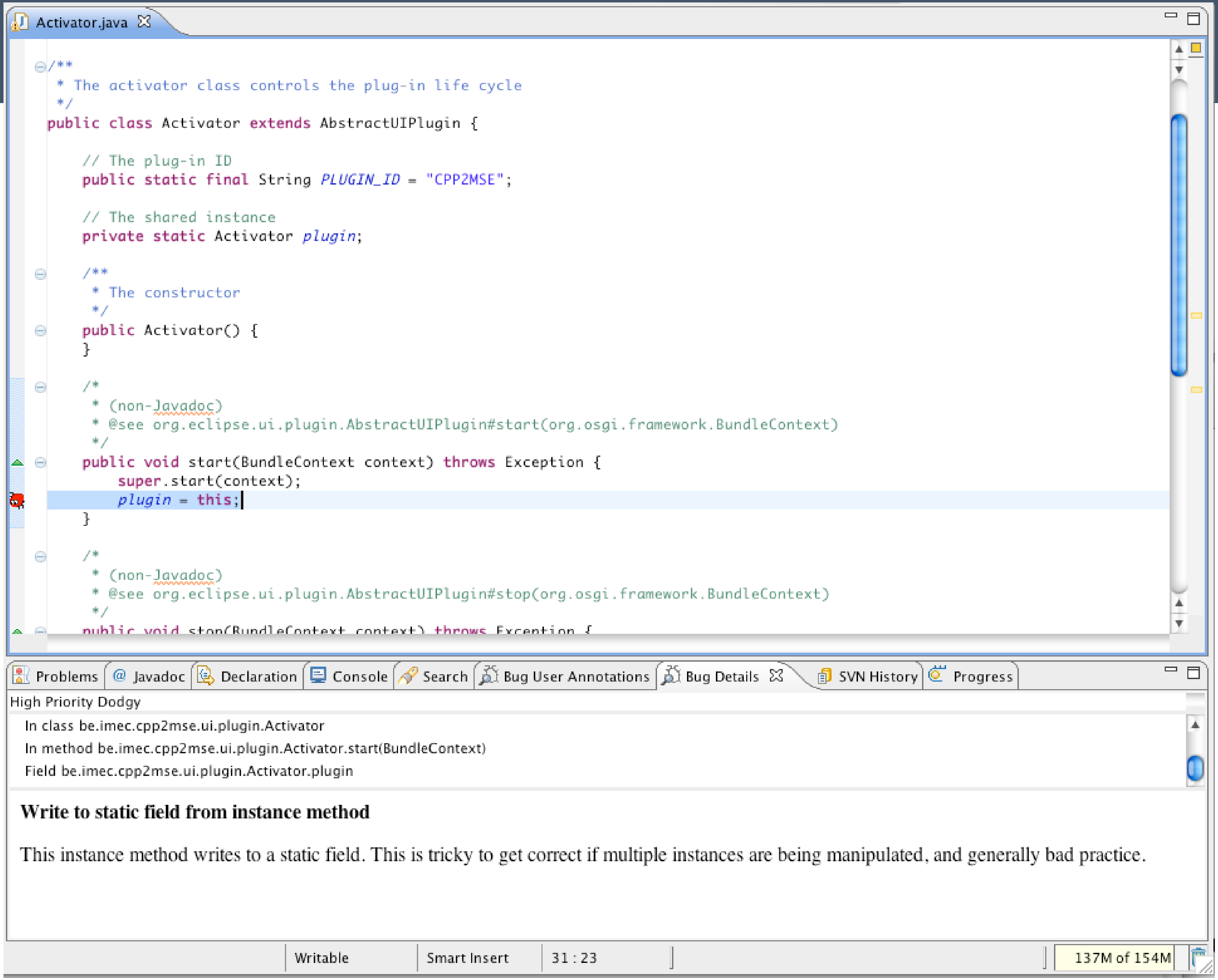
- If you need a comment to explain what a block of code does, use Extract Method
- If you need a comment to explain what a method does, use Rename Method
- If you need to describe the required state of the system, use Introduce Assertion

When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous

The point is that code should be self-explanatory, so that comments are not necessary.

A comment is a good place to say *why* you did something

Java FindBugs



```
/**
 * The activator class controls the plug-in life cycle
 */
public class Activator extends AbstractUIPlugin {

    // The plug-in ID
    public static final String PLUGIN_ID = "CPP2MSE";

    // The shared instance
    private static Activator plugin;

    /**
     * The constructor
     */
    public Activator() {
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        super.start(context);
        plugin = this;
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
```

High Priority Dodgy

In class be.imec.cpp2mse.ui.plugin.Activator

In method be.imec.cpp2mse.ui.plugin.Activator.start(BundleContext)

Field be.imec.cpp2mse.ui.plugin.Activator.plugin

Write to static field from instance method

This instance method writes to a static field. This is tricky to get correct if multiple instances are being manipulated, and generally bad practice.

Obstacles to Refactoring

Performance issue

- “Refactoring will slow down the execution”

Cultural Issues

- “We pay you to add new features, not to improve the code!”

If it doesn’t break, do not fix it

- “We do not have a problem, this is our software!“

Development is always under time pressure

- Refactoring takes time
- Refactoring better after delivery
- Process should take it into account, like testing

Conclusion

Refactoring is just a way of rearranging code

- Refactorings are used to solve problems
- If there’s no problem, you shouldn’t refactor

The notion of “bad smells” is a way of helping us recognize when we have a problem

- Familiarity with bad smells helps us avoid them in the first place

Refactorings are mostly pretty obvious

- Most of the value in discussing them is just to bring them into our “conscious toolbox”
- Refactorings have names in order to crystalize the idea and help us remember it

Profiling

What and how

Performance Myth

Don’t think that clean software is slow!

Normally only 10% of your system consumes 90% of the resources so just focus on 10 %.

- Refactorings help to localise the part that need change
- Refactorings help to concentrate the optimisations

Always use a profiler on your “slow” system to guide your optimisation effort

- Never optimise first!

Profiling

“Measure the behaviour of a program as it runs”

Note: can profile different things

- execution speed
- memory usage
- …

Profiling concepts

How does it work?

- Sampling: gather information from time to time
  - Less accurate
  - Less performance overhead
- Code instrumentation: modify program to analyze itself
  - Full instrumentation is very exact
  - Slower
  - Risc for Heisenbugs
  - Can be manual, static, dynamic, ...

Profiler Tools

Can be integrated in Development Environment

- linked with code: can highlight slow methods, …
- make profile data understandable and usable

Can be stand-alone

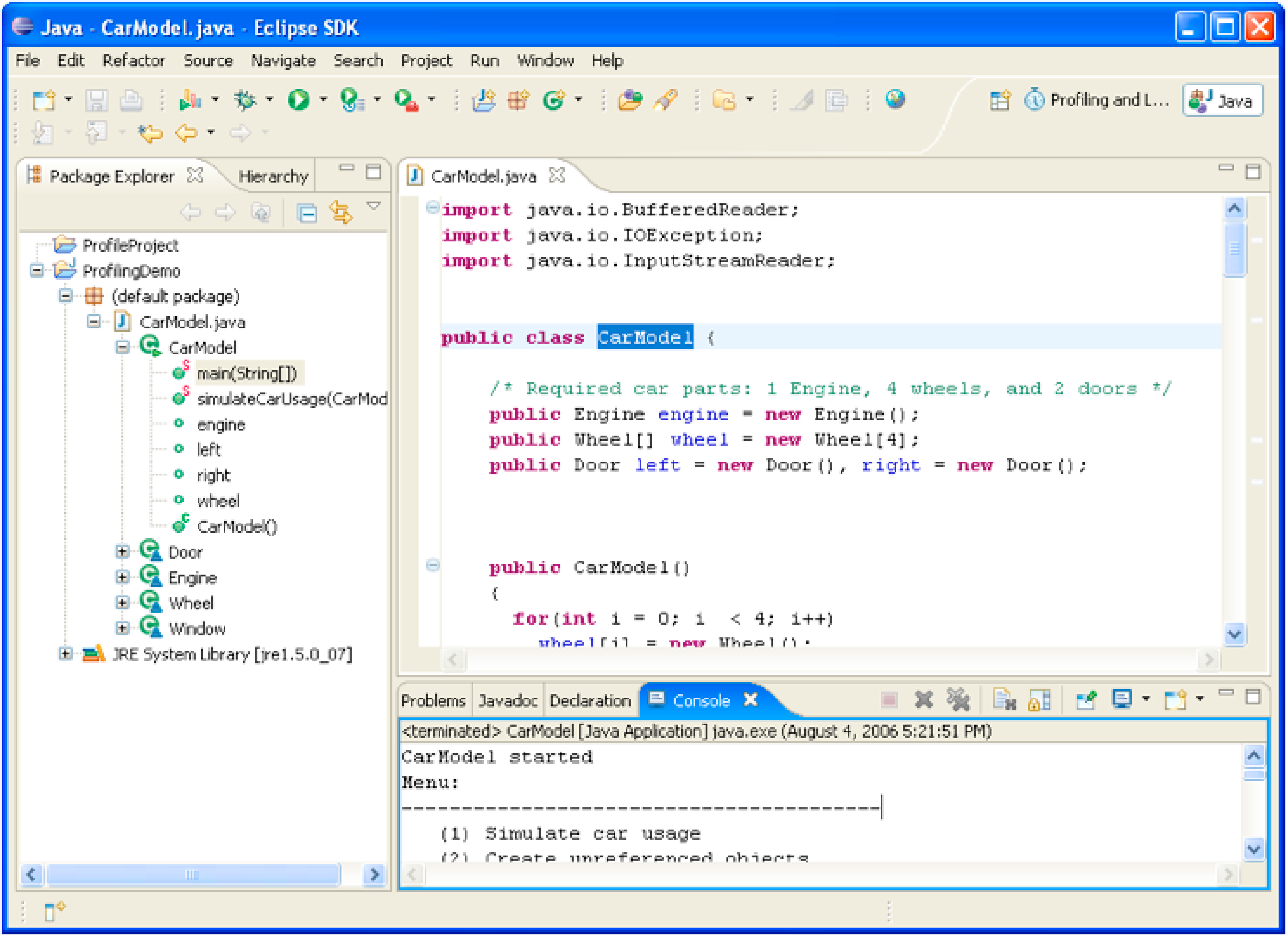
- no need to get project in IDE just to profile

Example: Java Profiling in Eclipse

Java profiling can be installed in Eclipse

- Does Memory and Execution Time profiling
  - local or remote

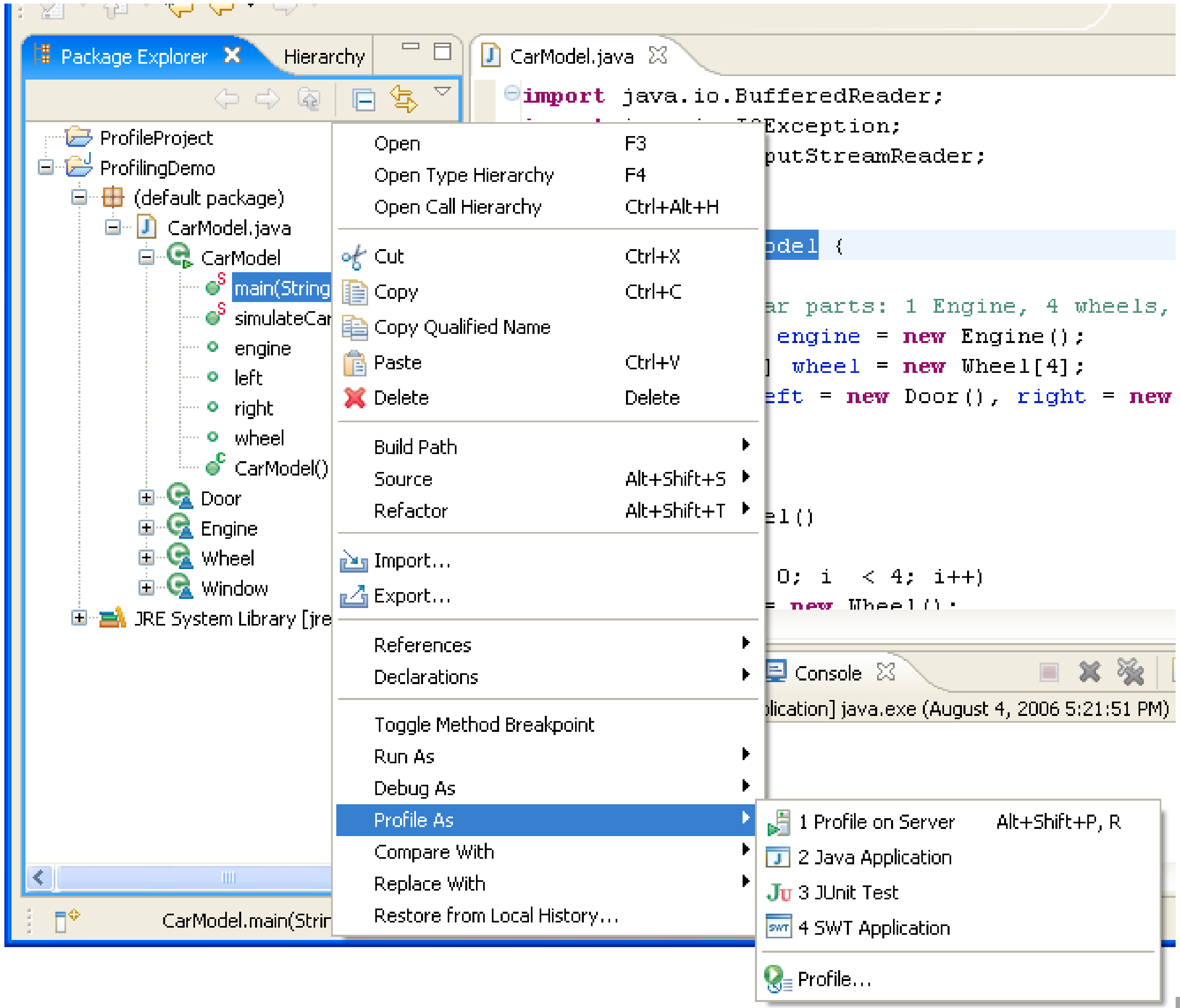
We have a Java project to profile...



```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;


public class CarModel {

    /* Required car parts: 1 Engine, 4 wheels, and 2 doors */
    public Engine engine = new Engine();
    public Wheel[] wheel = new Wheel[4];
    public Door left = new Door(), right = new Door();



    public CarModel()
    {
      for(int i = 0; i  < 4; i++)
        wheel[i] = new Wheel();
```

```
<terminated> CarModel [Java Application] java.exe (August 4, 2006 5:21:51 PM)
CarModel started
Menu:
----------------------------------------
   (1) Simulate car usage
   (2) Create unreferenced objects
```

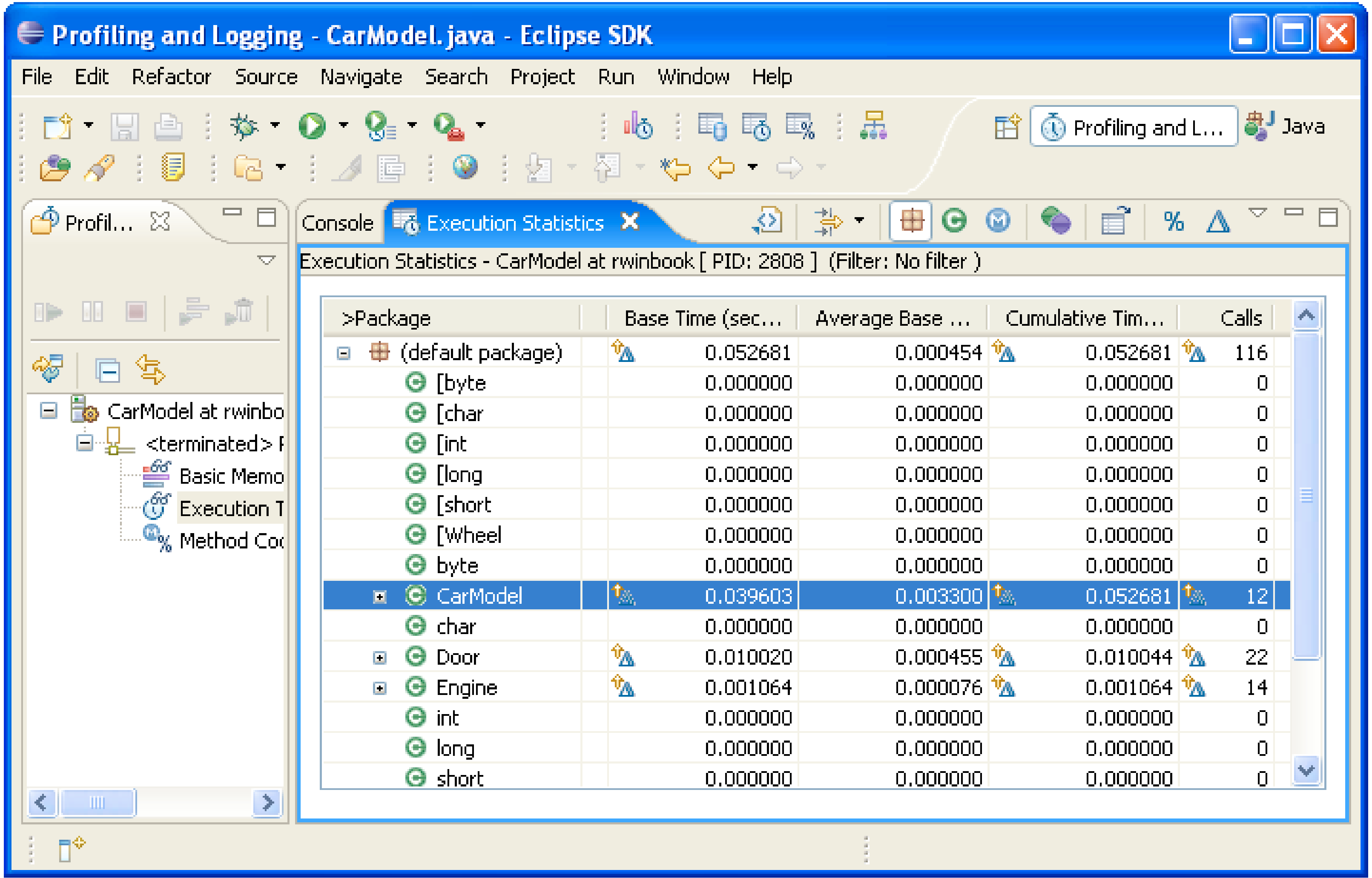
Profile the main function



```
import java.io.BufferedReader;
         IOException;
         putStreamReader;

         odel {

         ar parts: 1 Engine, 4 wheels,
          engine = new Engine();
         ] wheel = new Wheel[4];
         eft = new Door(), right = new

         el()

          0; i  < 4; i++)
         = new Wheel();
```

plication] java.exe (August 4, 2006 5:21:51 PM)

View results in Profiling perspective



Execution Statistics - CarModel at rwinbook [PID: 2808] (Filter: No filter)

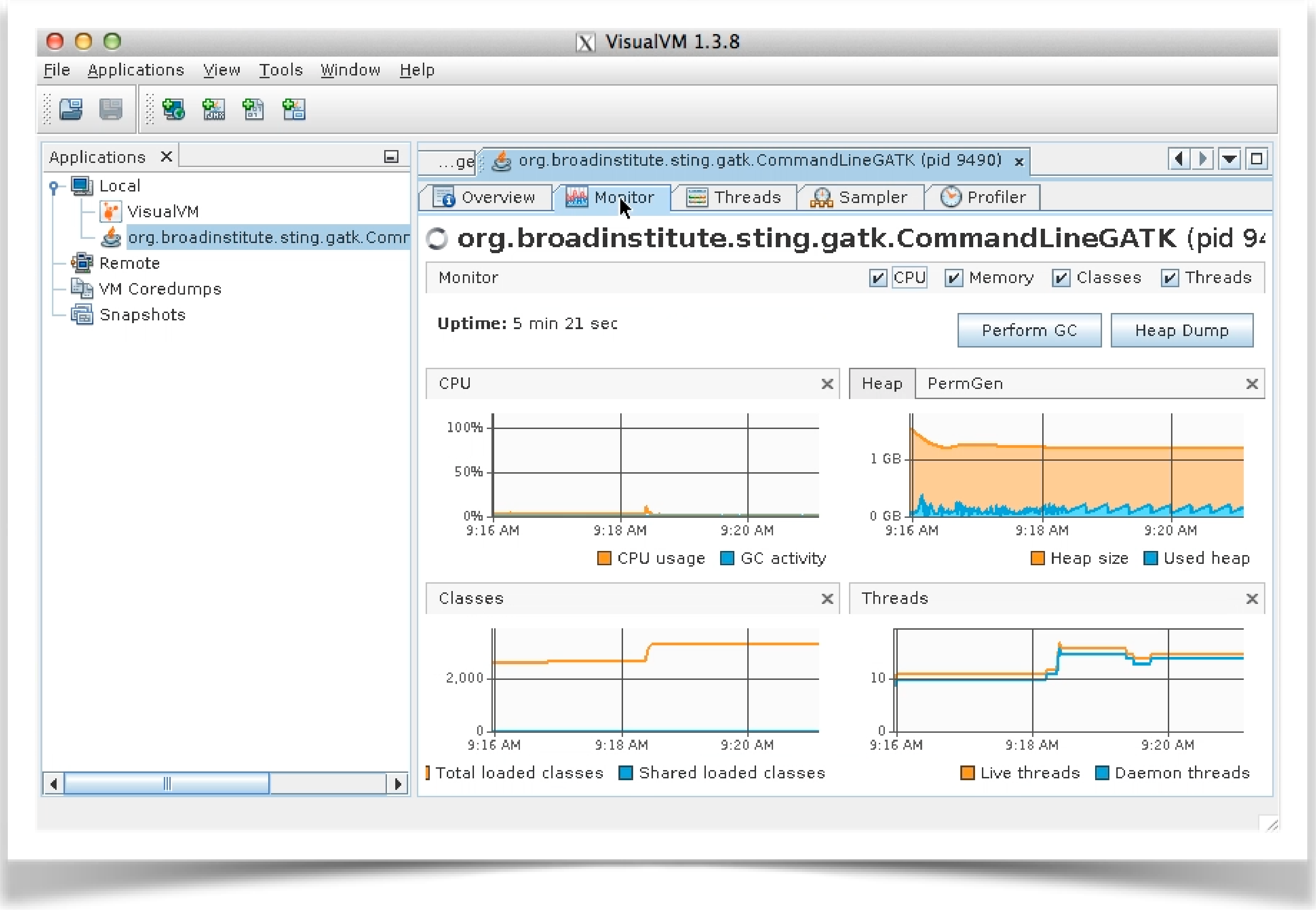
>Package	Base Time (sec...	Average Base ...	Cumulative Tim...	Calls
(default package)	0.052681	0.000454	0.052681	116
[byte	0.000000	0.000000	0.000000	0
[char	0.000000	0.000000	0.000000	0
[int	0.000000	0.000000	0.000000	0
[long	0.000000	0.000000	0.000000	0
[short	0.000000	0.000000	0.000000	0
[Wheel	0.000000	0.000000	0.000000	0
byte	0.000000	0.000000	0.000000	0
CarModel	0.039603	0.003300	0.052681	12
char	0.000000	0.000000	0.000000	0
Door	0.010020	0.000455	0.010044	22
Engine	0.001064	0.000076	0.001064	14
int	0.000000	0.000000	0.000000	0
long	0.000000	0.000000	0.000000	0
short	0.000000	0.000000	0.000000	0

Example: VisualVM (http://visualvm.java.net/)

monitor and/or sample CPU time and memory

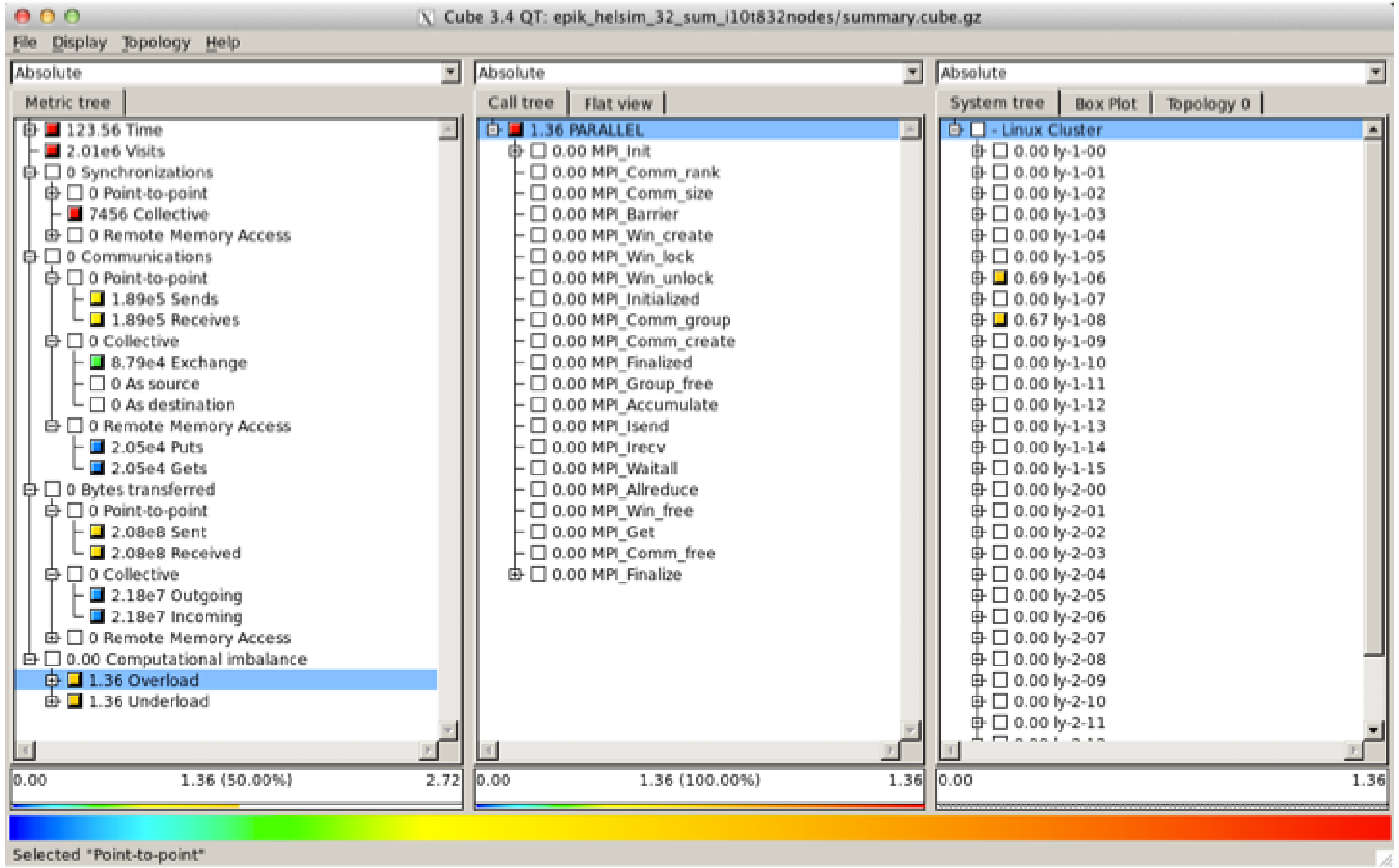
Easy to use, stand-alone

See video



Other useful tools exist for profiling...

“Scalasca” : spot communication&synchronization imbalances in MPI programs (http://scalasca.org)



```
123.56 Time
2.01e6 Visits
0 Synchronizations
  0 Point-to-point
  7456 Collective
  0 Remote Memory Access
0 Communications
  0 Point-to-point
    1.89e5 Sends
    1.89e5 Receives
  0 Collective
    8.79e4 Exchange
    0 As source
    0 As destination
  0 Remote Memory Access
    2.05e4 Puts
    2.05e4 Gets
0 Bytes transferred
  0 Point-to-point
    2.08e8 Sent
    2.08e8 Received
  0 Collective
    2.18e7 Outgoing
    2.18e7 Incoming
  0 Remote Memory Access
0.00 Computational imbalance
  1.36 Overload
  1.36 Underload
```

```
1.36 PARALLEL
  0.00 MPI_Init
  0.00 MPI_Comm_rank
  0.00 MPI_Comm_size
  0.00 MPI_Barrier
  0.00 MPI_Win_create
  0.00 MPI_Win_lock
  0.00 MPI_Win_unlock
  0.00 MPI_Initialized
  0.00 MPI_Comm_group
  0.00 MPI_Comm_create
  0.00 MPI_Finalized
  0.00 MPI_Group_free
  0.00 MPI_Accumulate
  0.00 MPI_Isend
  0.00 MPI_Irecv
  0.00 MPI_Waitall
  0.00 MPI_Allreduce
  0.00 MPI_Win_free
  0.00 MPI_Get
  0.00 MPI_Comm_free
  0.00 MPI_Finalize
```

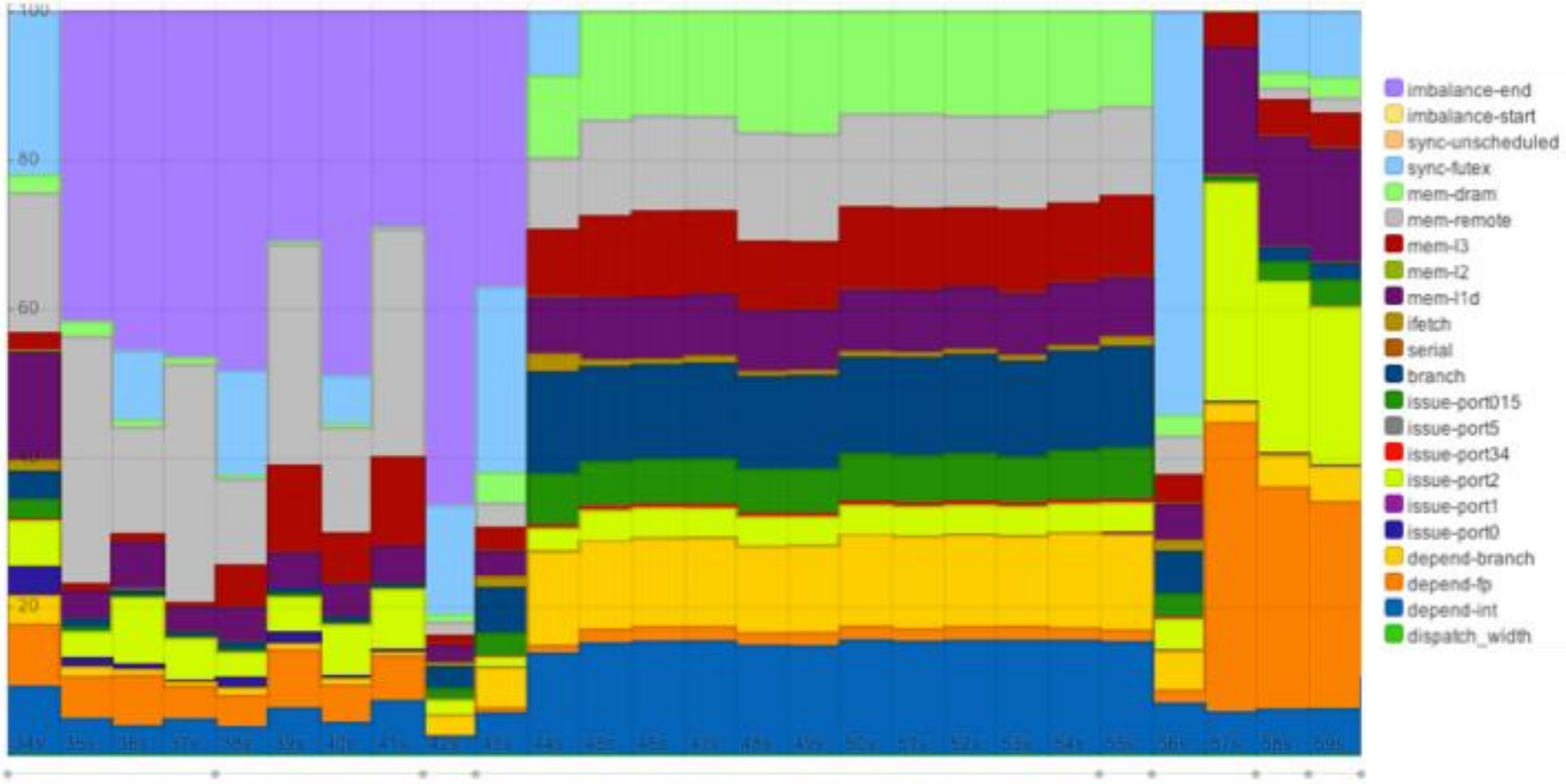
```
- Linux Cluster
  0.00 ly-1-00
  0.00 ly-1-01
  0.00 ly-1-02
  0.00 ly-1-03
  0.00 ly-1-04
  0.00 ly-1-05
  0.69 ly-1-06
  0.00 ly-1-07
  0.67 ly-1-08
  0.00 ly-1-09
  0.00 ly-1-10
  0.00 ly-1-11
  0.00 ly-1-12
  0.00 ly-1-13
  0.00 ly-1-14
  0.00 ly-1-15
  0.00 ly-2-00
  0.00 ly-2-01
  0.00 ly-2-02
  0.00 ly-2-03
  0.00 ly-2-04
  0.00 ly-2-05
  0.00 ly-2-06
  0.00 ly-2-07
  0.00 ly-2-08
  0.00 ly-2-09
  0.00 ly-2-10
  0.00 ly-2-11
```

```
0.00    1.36 (50.00%)    2.72
0.00    1.36 (100.00%)   1.36
0.00                     1.36
```

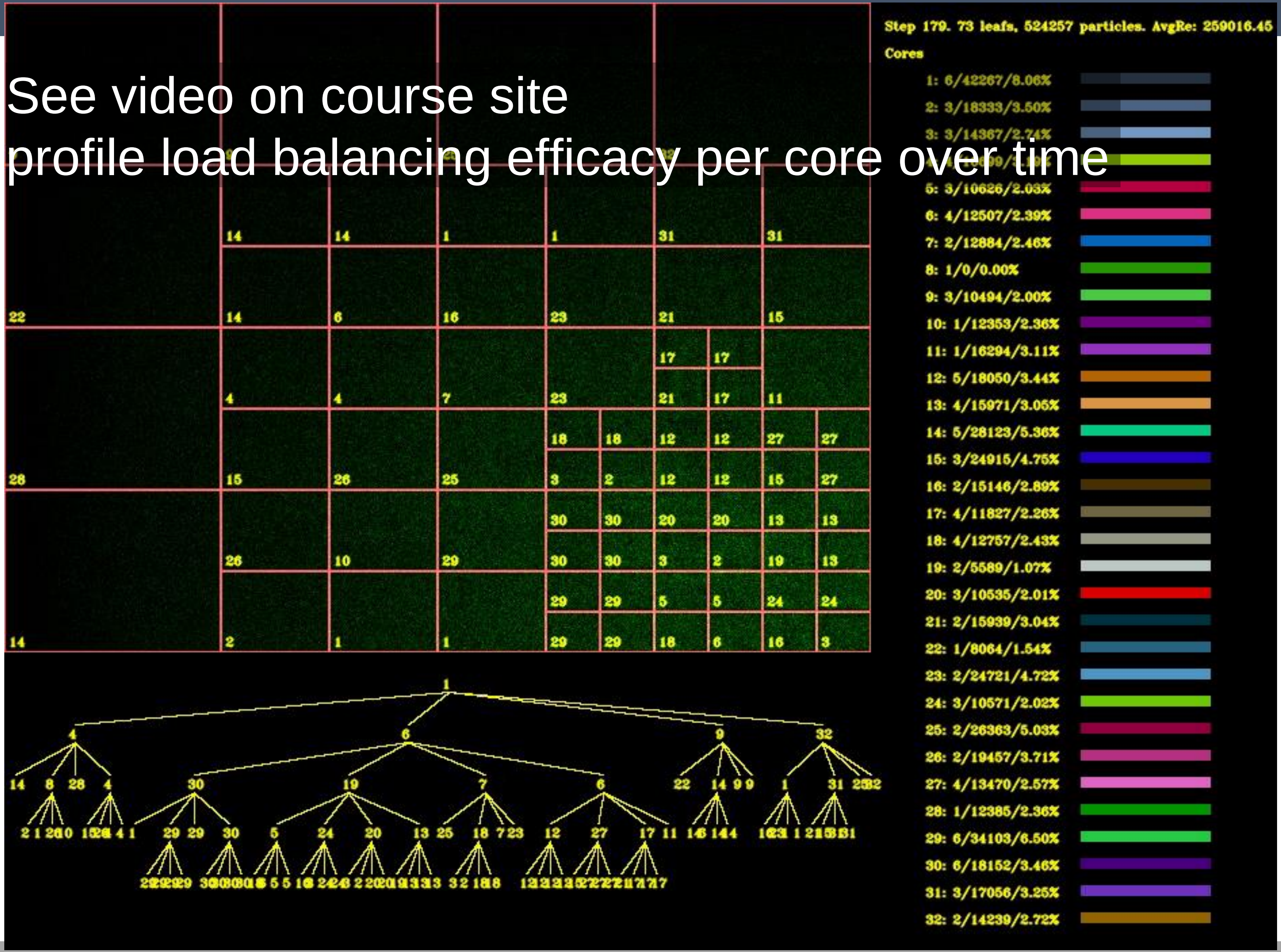
Selected "Point-to-point"

Other useful tools exist for profiling...

“Sniper” : fast hardware simulator for detailed analysis (http://snipersim.org)



Sometimes you have to *roll your own*



See video on course site

profile load balancing efficacy per core over time

Step 179. 73 leafs, 524257 particles. AvgRe: 259016.45

Cores

1: 6/42267/8.06%	▬
2: 3/18333/3.50%	▬
3: 3/14367/2.74%	▬
4: [illegible]	▬
5: 3/10626/2.03%	▬
6: 4/12507/2.39%	▬
7: 2/12884/2.46%	▬
8: 1/0/0.00%	▬
9: 3/10494/2.00%	▬
10: 1/12353/2.36%	▬
11: 1/16294/3.11%	▬
12: 5/18050/3.44%	▬
13: 4/15971/3.05%	▬
14: 5/28123/5.36%	▬
15: 3/24915/4.75%	▬
16: 2/15146/2.89%	▬
17: 4/11827/2.26%	▬
18: 4/12757/2.43%	▬
19: 2/5589/1.07%	▬
20: 3/10535/2.01%	▬
21: 2/15939/3.04%	▬
22: 1/8064/1.54%	▬
23: 2/24721/4.72%	▬
24: 3/10571/2.02%	▬
25: 2/26363/5.03%	▬
26: 2/19457/3.71%	▬
27: 4/13470/2.57%	▬
28: 1/12385/2.36%	▬
29: 6/34103/6.50%	▬
30: 6/18152/3.46%	▬
31: 3/17056/3.25%	▬
32: 2/14239/2.72%	▬

Conclusion

Make it Work, Make it Right, Make it Fast

Unit testing remove fear of making changes

Refactoring remove fear of making changes

Profiling tells you where to make performance-related changes

- focus your effort

Design of Software Systems
(Ontwerp van SoftwareSystemen)

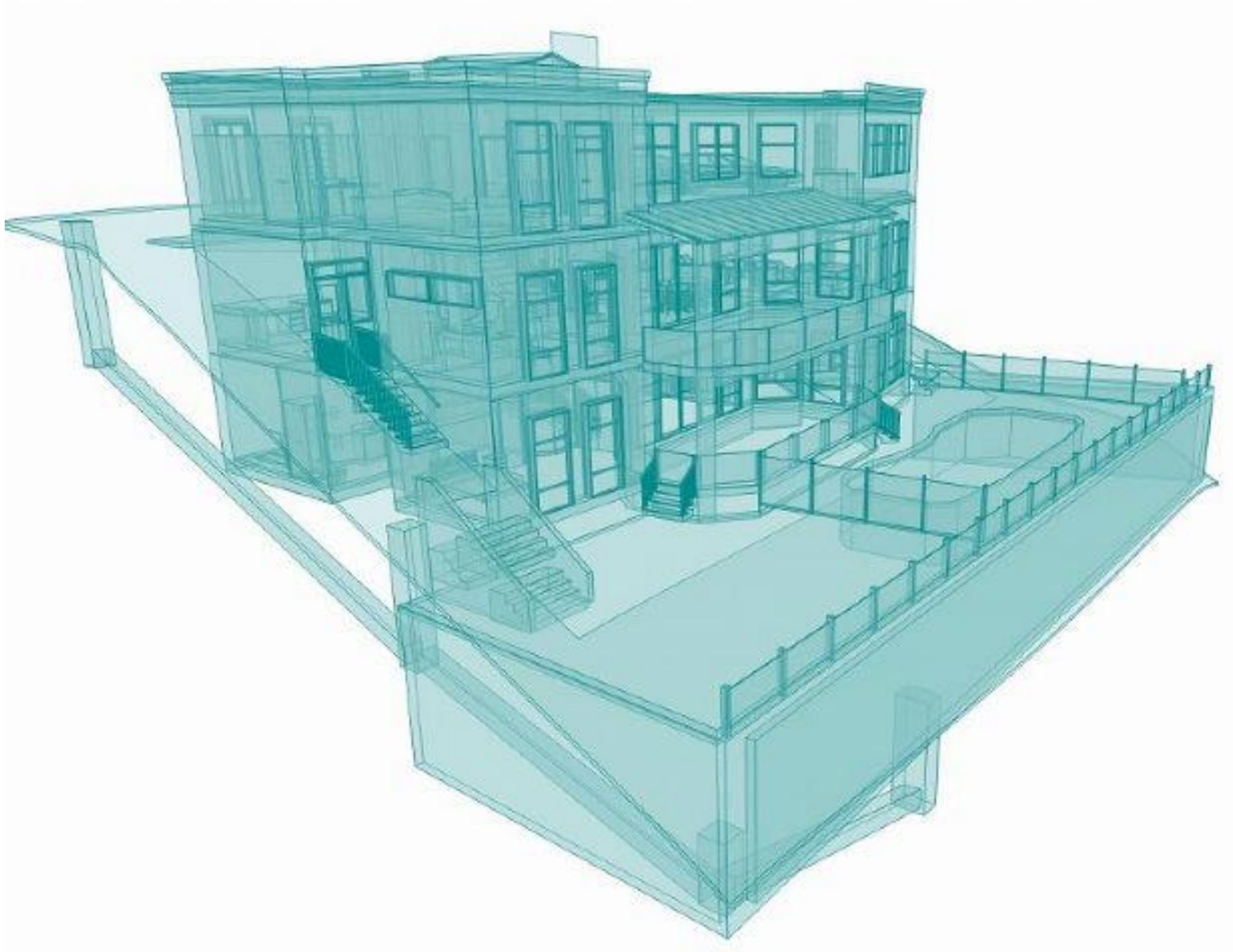
7 On Multi-user Development Tools,Versioning and Packaging of code, their Relations, the Universe and Everything.

Roel Wuyts
2016-2017

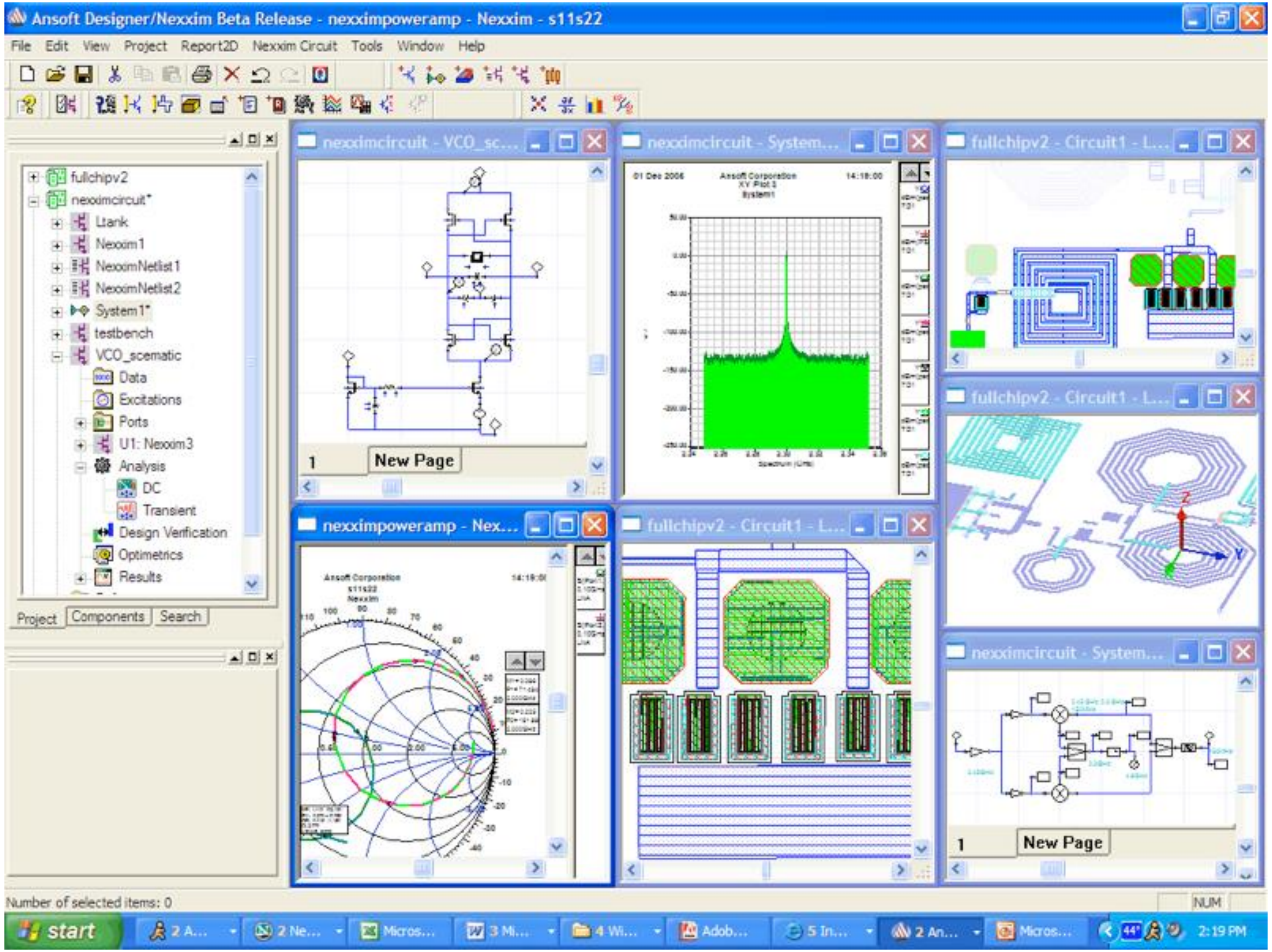
Developing Complex Systems

How do scientific disciplines construct complex systems ?

Architectural Software



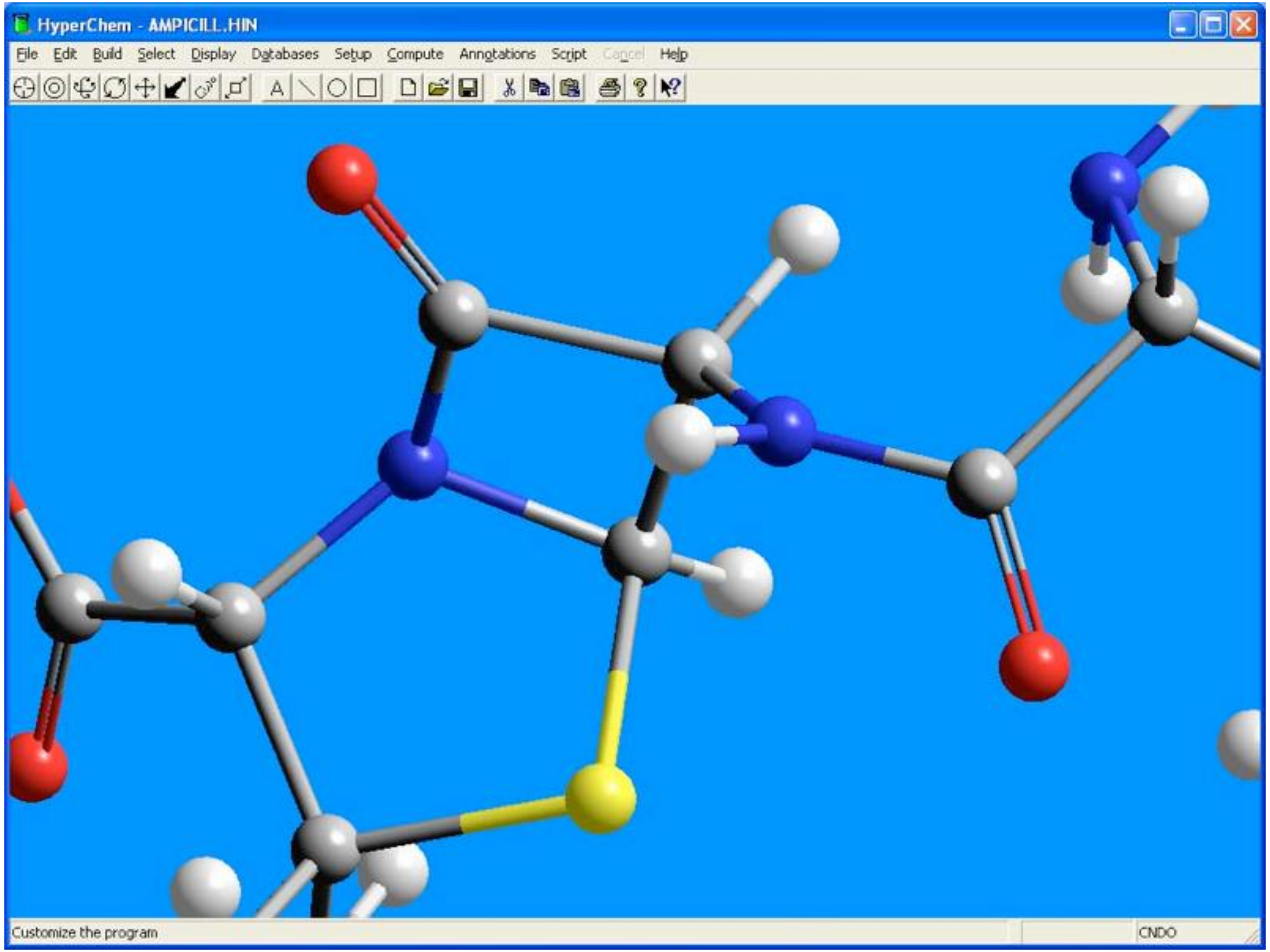
RF/mW Design & Analog/RFIC Verification



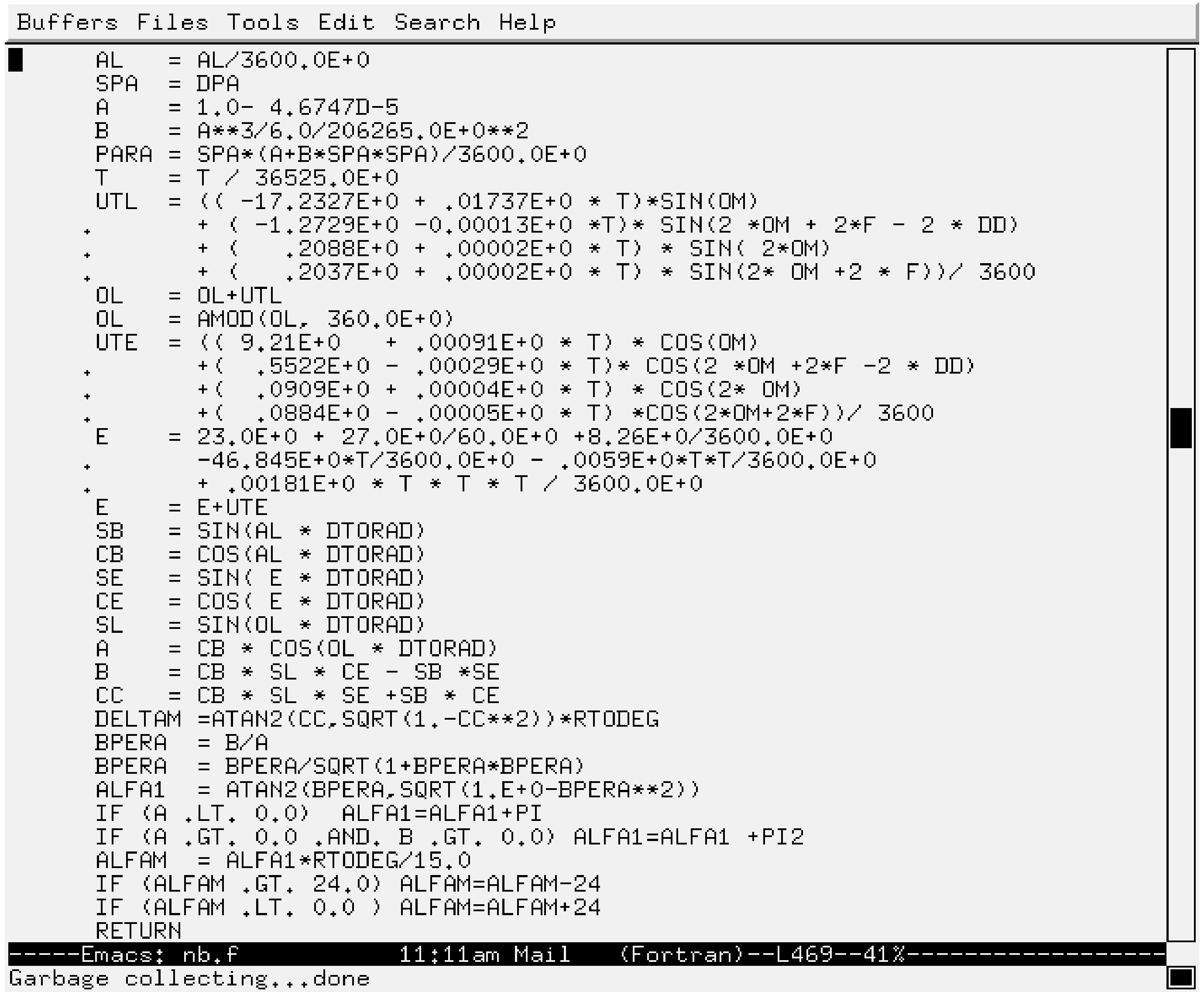
Ansoft Designer/Nexxim Beta Release - nexximpoweramp - Nexxim - s11s22

Number of selected items: 0

Visualization & Manipulation of molecules


HyperChem - AMPICILL.HIN
File Edit Build Select Display Databases Setup Compute Annotations Script Cancel Help
Customize the program
CNDO

Computer Science/Engineering…



```
      AL   = AL/3600.0E+0
      SPA  = DPA
      A    = 1.0- 4.6747D-5
      B    = A**3/6.0/206265.0E+0**2
      PARA = SPA*(A+B*SPA*SPA)/3600.0E+0
      T    = T / 36525.0E+0
      UTL  = (( -17.2327E+0 + .01737E+0 * T)*SIN(OM)
     .       + ( -1.2729E+0 -0.00013E+0 *T)* SIN(2 *OM + 2*F - 2 * DD)
     .       + (   .2088E+0 + .00002E+0 * T) * SIN( 2*OM)
     .       + (   .2037E+0 + .00002E+0 * T) * SIN(2* OM +2 * F))/ 3600
      OL   = OL+UTL
      OL   = AMOD(OL, 360.0E+0)
      UTE  = (( 9.21E+0    + .00091E+0 * T) * COS(OM)
     .       +(  .5522E+0 - .00029E+0 * T)* COS(2 *OM +2*F -2 * DD)
     .       +(  .0909E+0 + .00004E+0 * T) * COS(2* OM)
     .       +(  .0884E+0 - .00005E+0 * T) *COS(2*OM+2*F))/ 3600
      E    = 23.0E+0 + 27.0E+0/60.0E+0 +8.26E+0/3600.0E+0
     .       -46.845E+0*T/3600.0E+0 - .0059E+0*T*T/3600.0E+0
     .       + .00181E+0 * T * T * T / 3600.0E+0
      E    = E+UTE
      SB   = SIN(AL * DTORAD)
      CB   = COS(AL * DTORAD)
      SE   = SIN( E * DTORAD)
      CE   = COS( E * DTORAD)
      SL   = SIN(OL * DTORAD)
      A    = CB * COS(OL * DTORAD)
      B    = CB * SL * CE - SB *SE
      CC   = CB * SL * SE +SB * CE
      DELTAM =ATAN2(CC,SQRT(1.-CC**2))*RTODEG
      BPERA  = B/A
      BPERA  = BPERA/SQRT(1+BPERA*BPERA)
      ALFA1  = ATAN2(BPERA,SQRT(1.E+0-BPERA**2))
      IF (A .LT. 0.0)  ALFA1=ALFA1+PI
      IF (A .GT. 0.0 .AND. B .GT. 0.0) ALFA1=ALFA1 +PI2
      ALFAM  = ALFA1*RTODEG/15.0
      IF (ALFAM .GT. 24.0) ALFAM=ALFAM-24
      IF (ALFAM .LT. 0.0 ) ALFAM=ALFAM+24
      RETURN
-----Emacs: nb.f            11:11am Mail    (Fortran)--L469--41%------------------
Garbage collecting...done
```

Corollary

We need to construct systems that are typically more complex than in other disciplines

- for several reasons

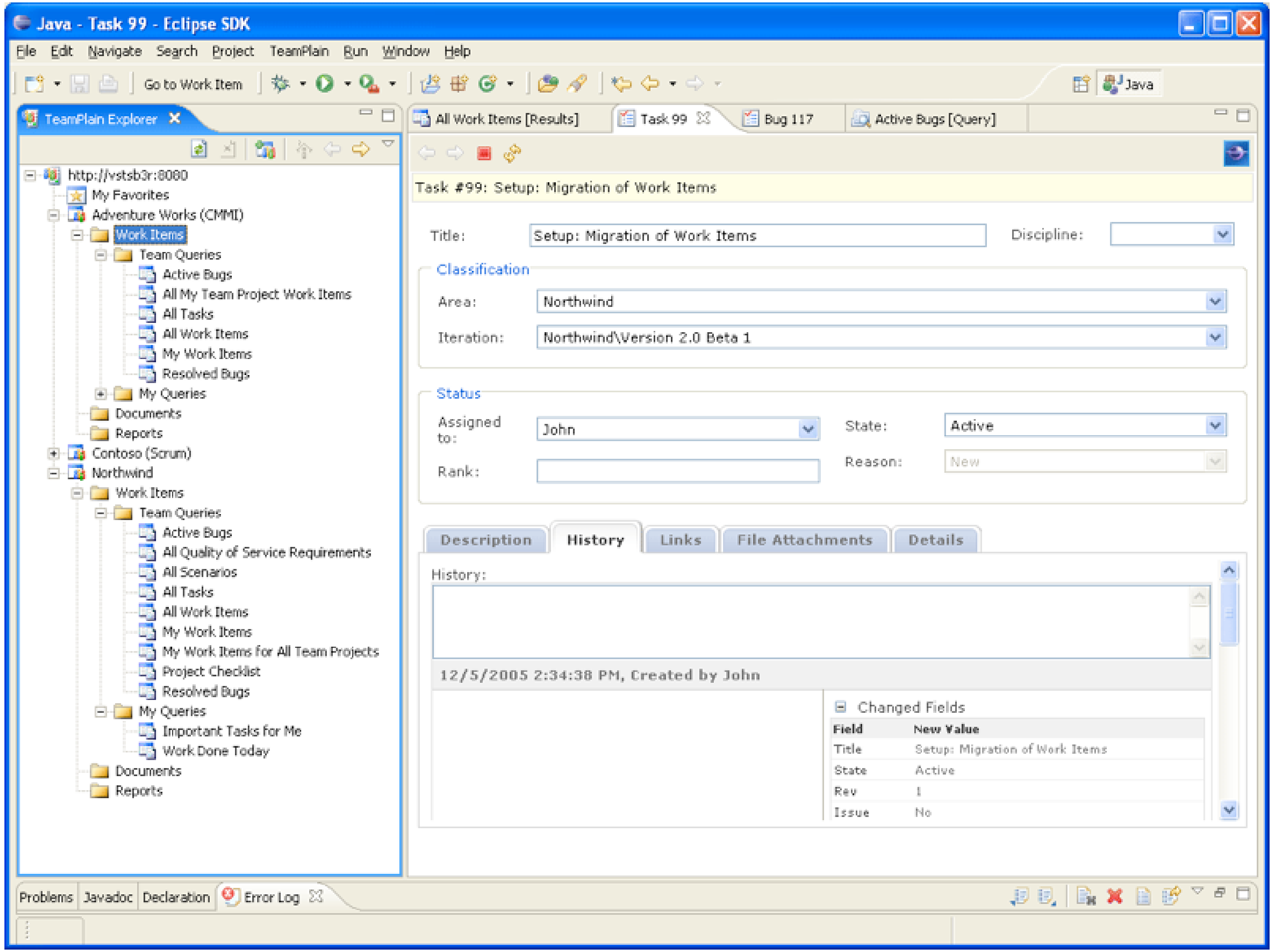
We have tangible elements to manipulate

- Buildings, circuits and molecules need a representation that is different than their physical one

Yet lots of developers still seem to prefer basic tools

- yes, emacs is a basic tool...

Eclipse/Netbeans/IntelliJ/... ?



Task #99: Setup: Migration of Work Items

Title: Setup: Migration of Work Items

Discipline:

Classification

Area: Northwind

Iteration: Northwind\Version 2.0 Beta 1

Status

Assigned to: John

State: Active

Rank:

Reason: New

History:

12/5/2005 2:34:38 PM, Created by John

Changed Fields

Field	New Value
Title	Setup: Migration of Work Items
State	Active
Rev	1
Issue	No

Eclipse...

Eclipse is a decent integrated development environment

- integrates navigation, editing, unit tests, refactoring, ...
- was developed by a lot of former Smalltalk people :-)

But at its core it is file-based (and so are most others)

- So ? Why don’t I like this ?

Files versus Objects

Non computer science disciplines:

- Architects work with construction materials&buildings
  - So do their tools
- Molecular biologists work with modules
  - Environment manipulates molecules
- ...

We work with objects

- Most tools deal with files ?!

Smalltalk image approach

The Smalltalk image is a live environment

- consists entirely of objects
- objects are manipulated

Files are one way of storing objects

- code too, since code are objects
- Databases are another mechanism, or network sockets or ...

Sidenote on Environments

- Good developers tailor their environment
  - So they need to be easily extensible
    - emacs: easy
    - Smalltalk environments: easy
    - Eclipse: possible
    - Most environments: hard or not possible
- Always favour an extensible one
  - control your tools!

Multi-user Development

Software engineering is a teamsport ;-)

Needed

- a code repository that allows multiple users
- integrated versioning
- configuration management

The language also has packaging mechanisms

- with or without namespaces

These concepts cross-cut

Code repositories and multiple users

Need to store code (obviously)

- but preferable also binaries, documentation, tests, ...

Locking vs. concurrent

- Lock: one user has (part of) code, unlocks when done
- Concurrent (lazy locking): several users can work simultaneously on the same system

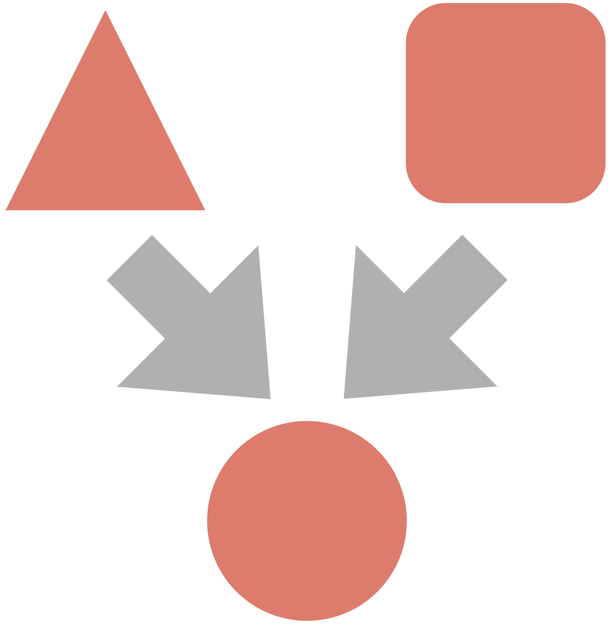
Centralized vs. Distributed

- Centralized: Only the master repository contains complete version history
- Distributed: all repositories have complete history

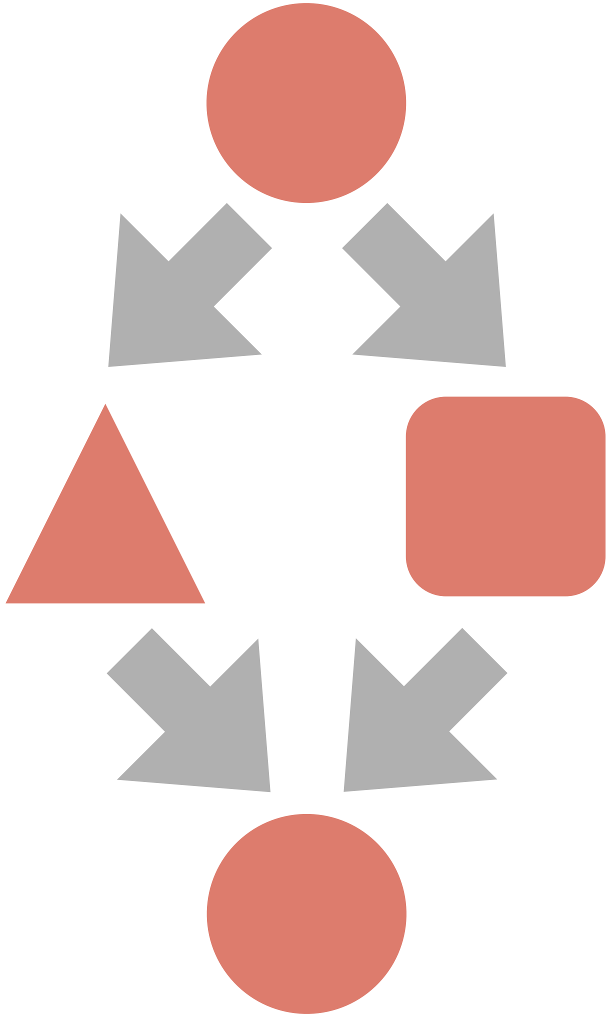
Support for merging

Merges

Two-way merge



Three-way merge



Example

One framework,

instantiated for two different clients,

each with their own customizations,

Where there is a stable version,

and two development branches

- a new version and a brand new one
- one dependent on the customization of the framework for one particular client

Let’s view two systems

Concurrent, centralized systems:

- Subversion (svn)
- Envy

Concurrent, distributed system:

- git

(Many more exist, but svn is archetypical for most popular tools like cvs/git, and Envy is a contrast)

svn : Subversion

descendant of cvs (concurrent versioning system)

Granularity: file

Users work detached from the repository:

- Load local copy of files from svn server (repository)
- Work on local copy (working directory)
- Commit changed files back to repository

Loading local copy can be done from the network

svn Workflow

Check-out code from repository in local environment

Work on code.

- can at all time see the difference between the current change and the state when checked-out

When finished, commit changes back to repository

- can trigger (3-way) merge when repository was updated in the meantime

Semantics

svn (like cvs, git, …) versions text

- has no semantics about what text it stores
- works with latex files, C++ files, ...

Therefore its operations have no semantics

- e.g. looking at changes after doing a renaming a method refactoring result in a list of textual changes to potentially many files
  - can be hard to know it was a refactoring, especially when combined when several other changes
  - commit often, and add comments !

Envy

It’s a versioning system, but not as you know it ;-)

Users are meant to be always connected to the repository

- can work separately but that is the exception

Works with methods, classes, ...

- Versioning knows about your language concepts
  - e.g. have all versions for a particular class, automatically includes all methods for that version of the class
- Smallest granularity is a method

Envy Workflow

Load code from repository in local environment

Work on code.

- Every change of a method or a class automatically (!) creates an edition
- These editions can be compared with, restored, …

Editions can be versioned

- the edition then gets a name and version number
- once versioned everybody in the repository can see and load versions
    - easier and earlier integration and conflict detection

Envy: Configuration Management features

From the ground up Envy has support for configuration management

- Applications group classes and methods
    - can have editions and versions themselves
    - have prerequisite versions (!)
- Configurations group applications
    - (e.g. Manifests in Microsoft .Net)
- Support for conditional loading and prerequisites
    - Platform-specific code, for example
    - Can be at application or configuration level

Removes need for external build systems like cmake, Maven, …

Git

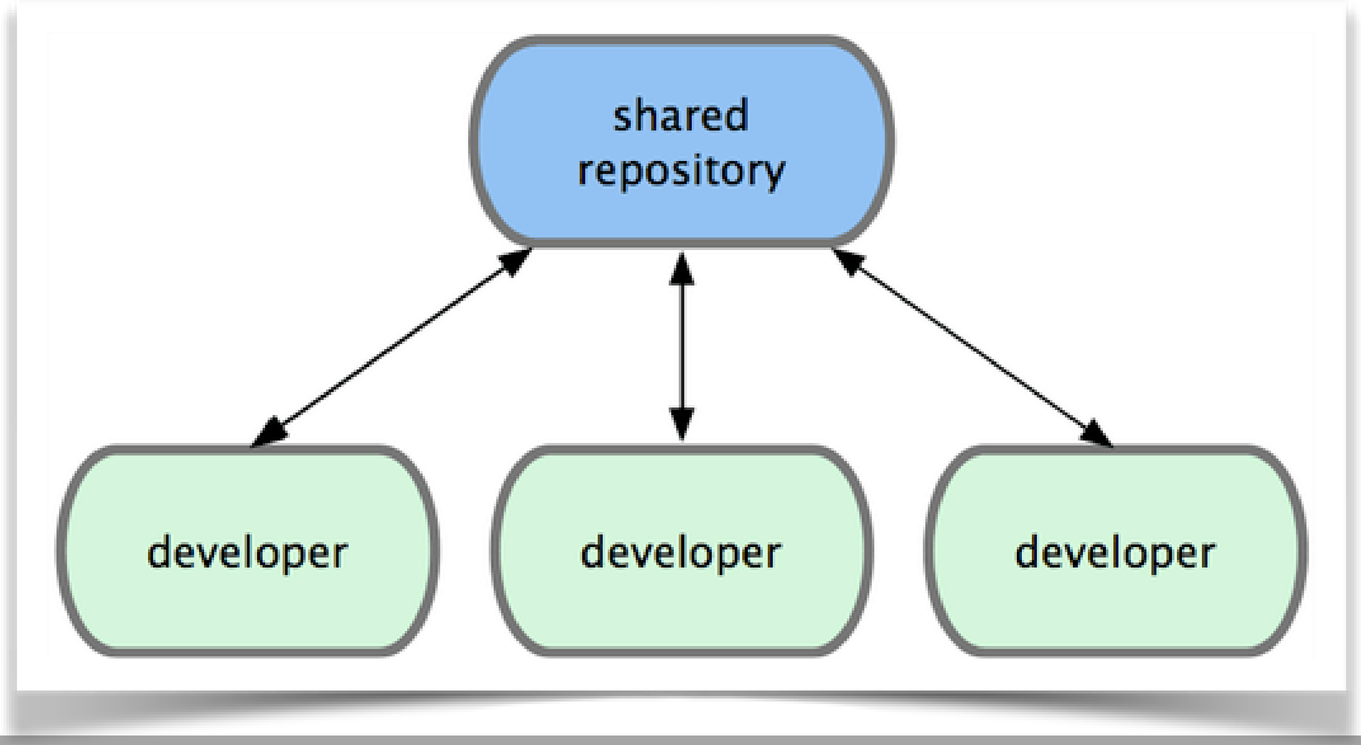
Distributed version control system

- Breaks the master/slave relationship prevalent in cvs/svn
  - every repository has the complete history
  - repositories sync with each other
- Good support for branching and advanced forms of version management (cherry picking, reverting changes, …)
- Like svn/cvs/…: stores text

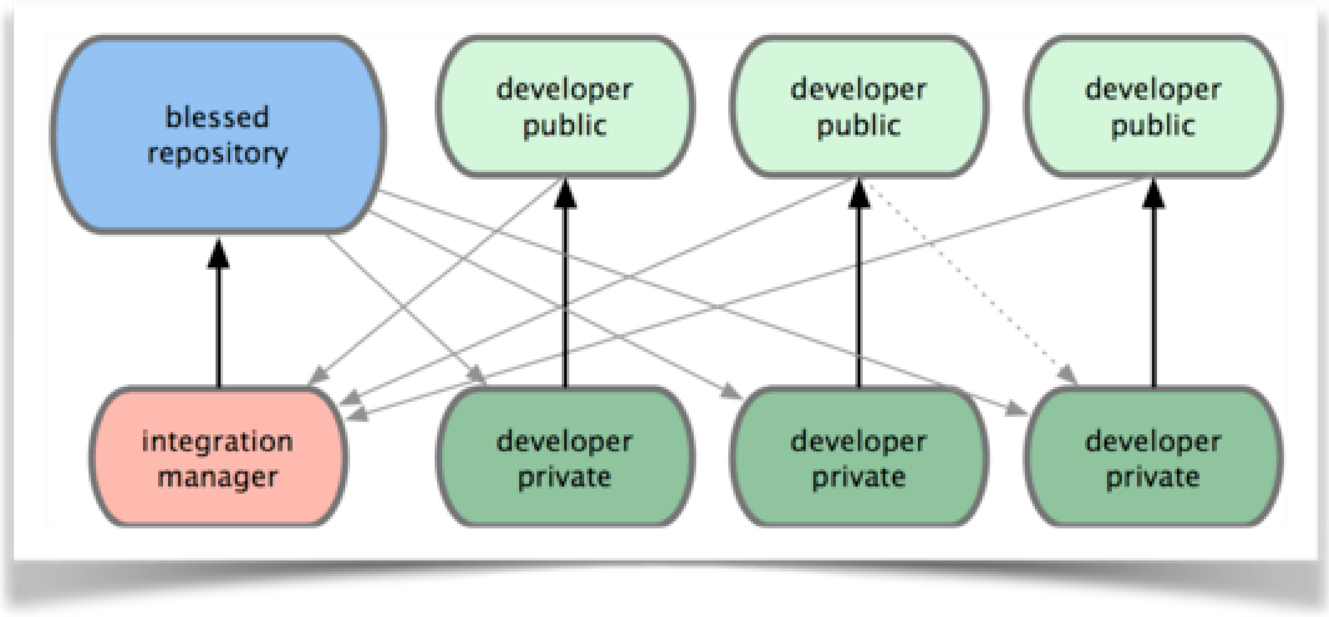
Git workflow

Many possibilities

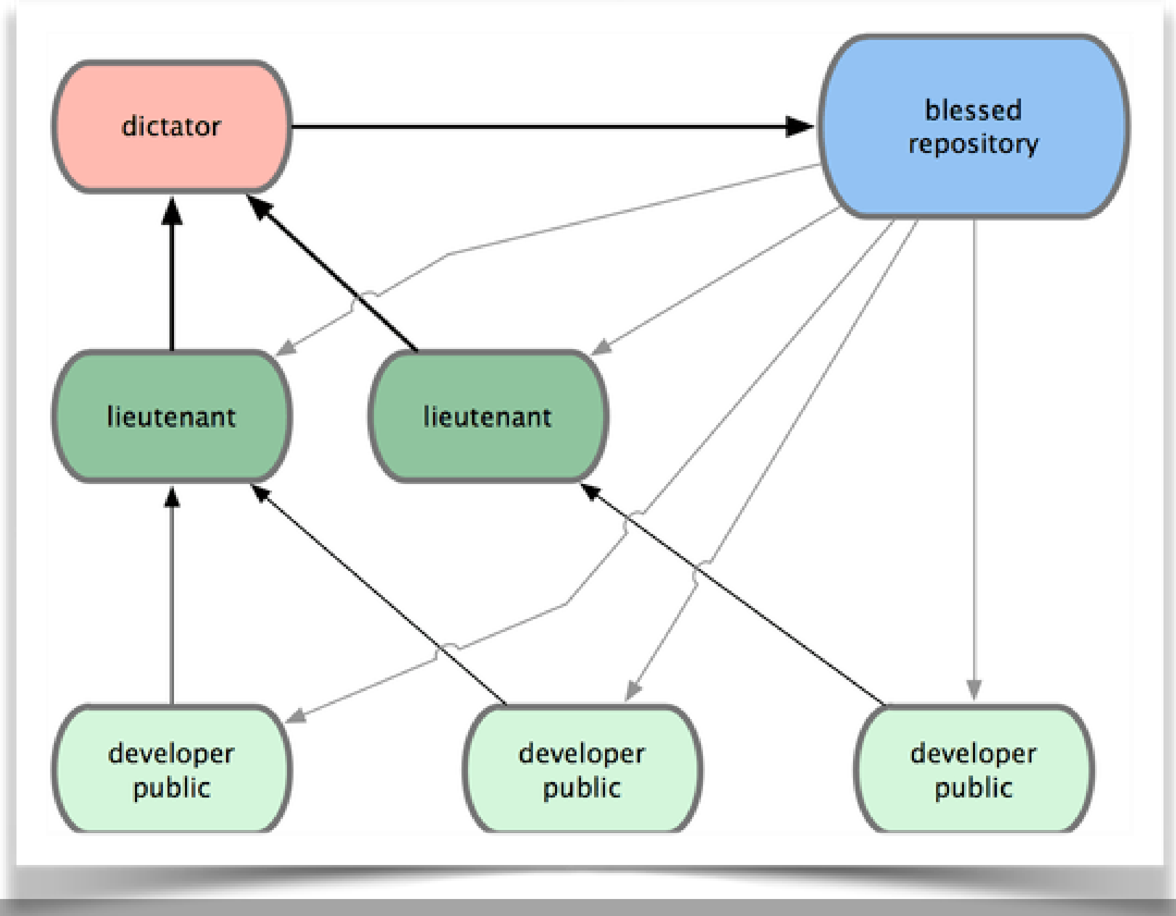
Can of course do the centralised workflow
(as in centralised approaches like svn/Envy/…)



Git workflow



Integration-Manager Workflow



Dictator and Lieutenants Workflow

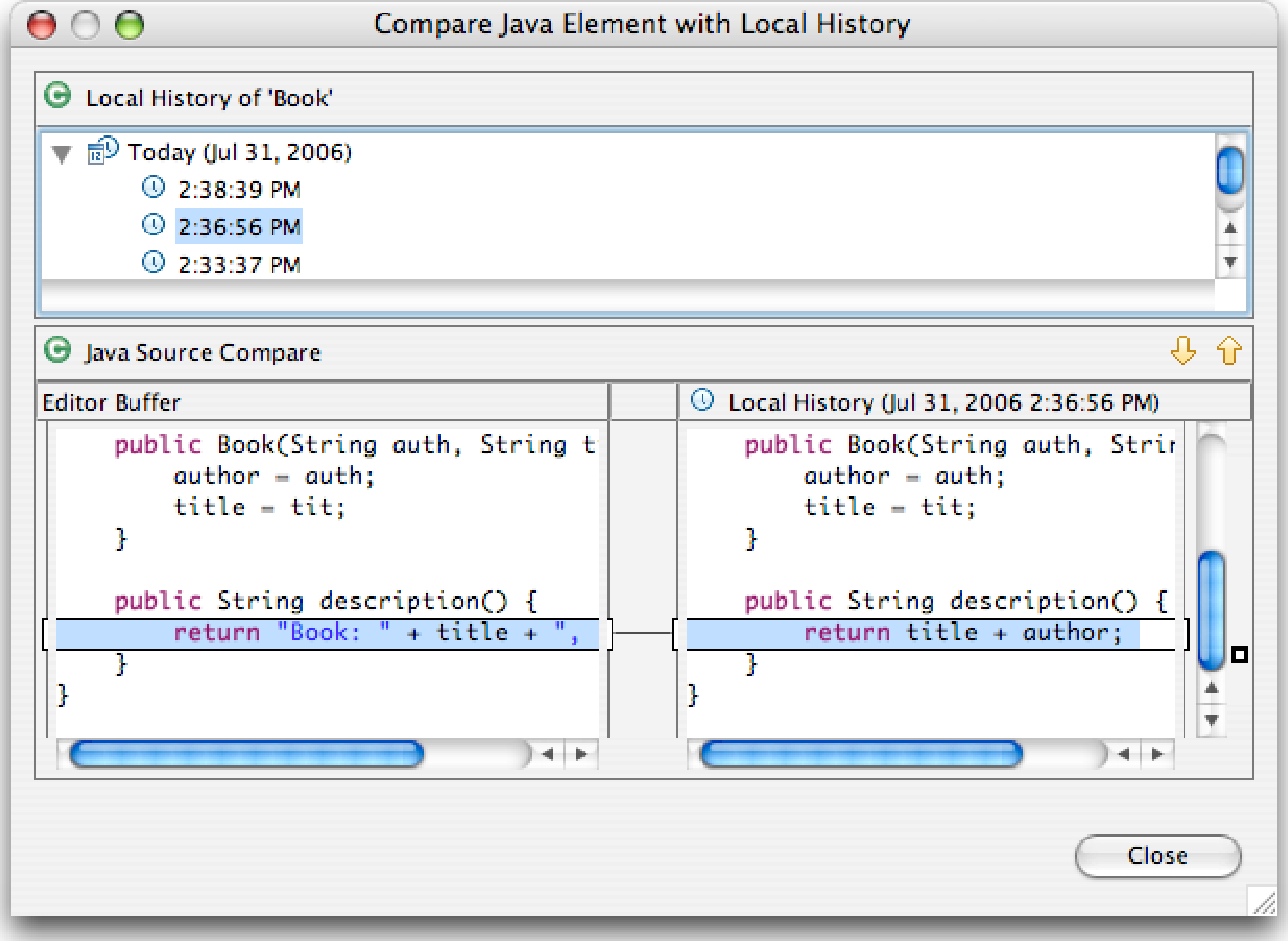
On granularity...

With svn/git/…, you have a *history of the files* you've checked in

With Envy, you have a *history of the development* you did

This is fundamentally different !

What is Envy doing in Eclipse ?!



```
    public Book(String auth, String t
        author = auth;
        title = tit;
    }

    public String description() {
        return "Book: " + title + ",
    }
}
```

```
    public Book(String auth, Strin
        author = auth;
        title = tit;
    }

    public String description() {
        return title + author;
    }
}
```

Concepts in Code Repositories

Code

Package

Configuration

Packages and Namespaces should be orthogonal

- package contains definitions
- namespaces is a visibility mechanism

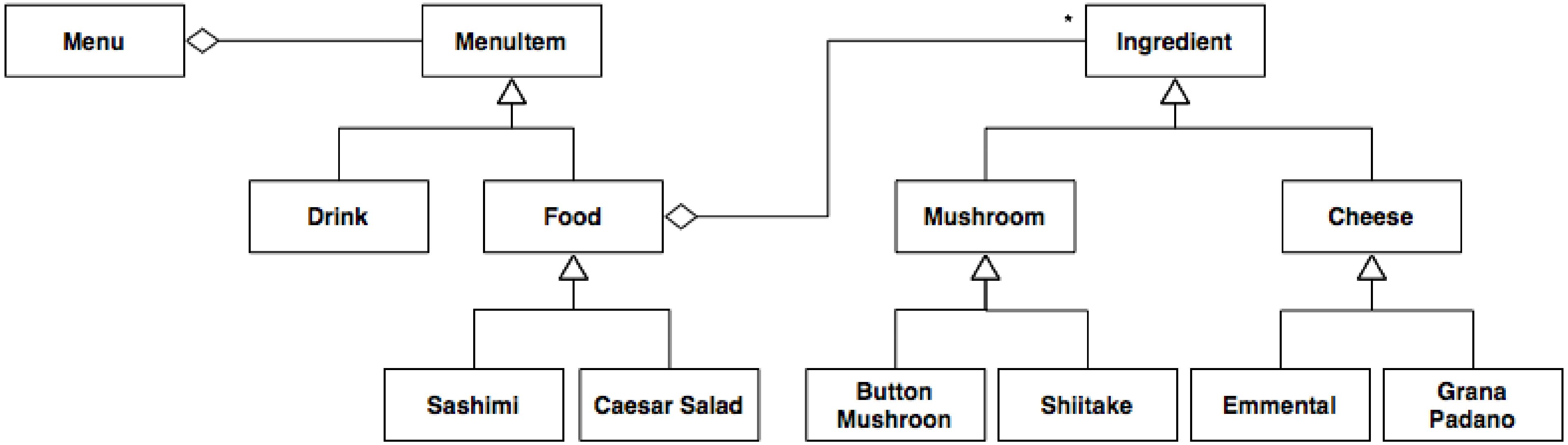
Versioning

All the elements need to be versionable

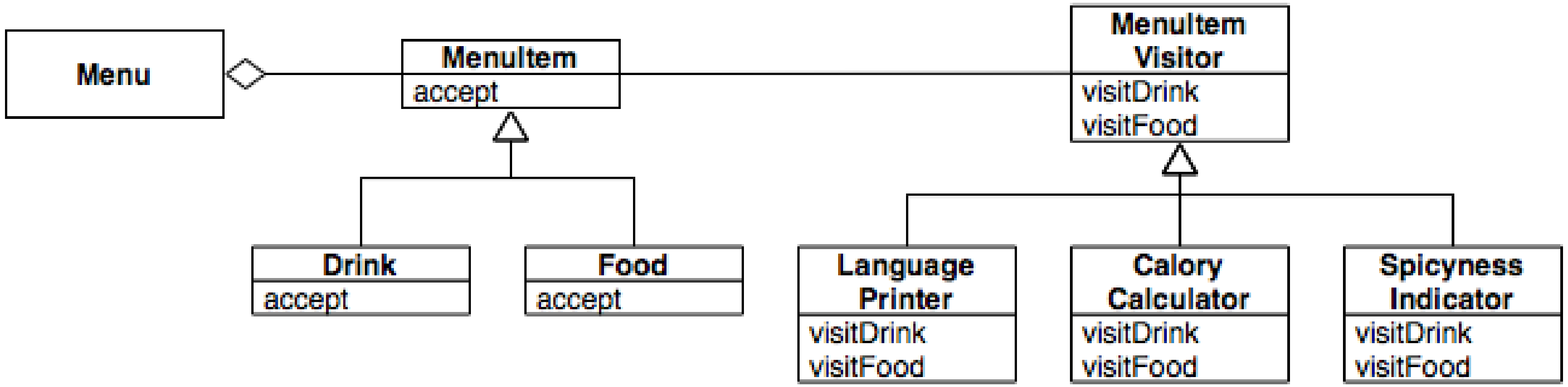
Decisions, decisions:

- granularity of version
  - line of code, method, class+methods, package, ...
- forms of version numbers
  - single number, composed number, alphanumeric
- version numbers versus release numbers
  - and their relationships

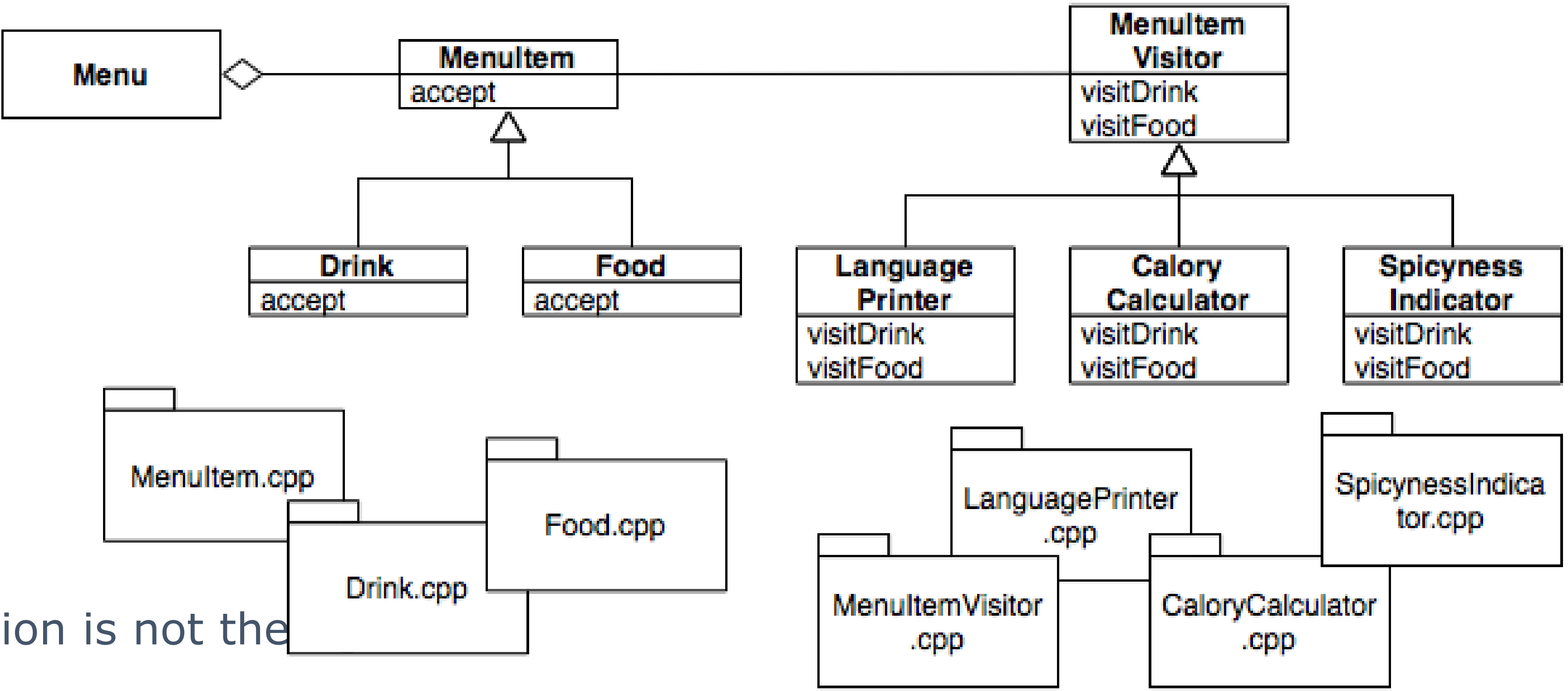
Concrete example : Menu Framework



Menu Framework with Visitor



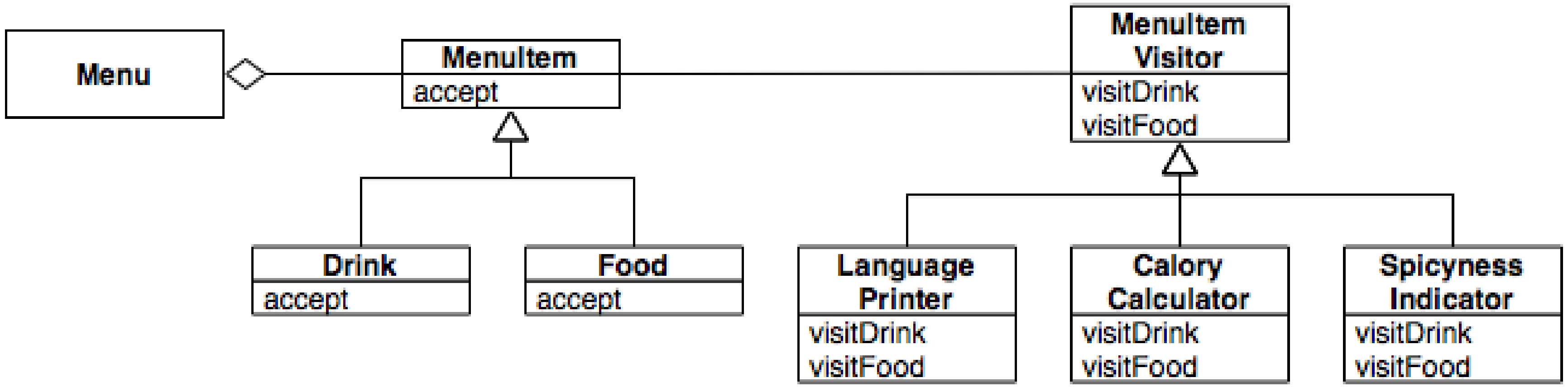
C++ Files



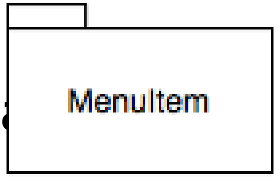
Files can go in cvs

- But decomposition is not the
- What if the visitor traversal needs to be changed?

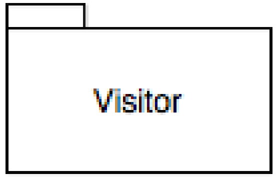
Java Packages



Packages to regroup classes, stor



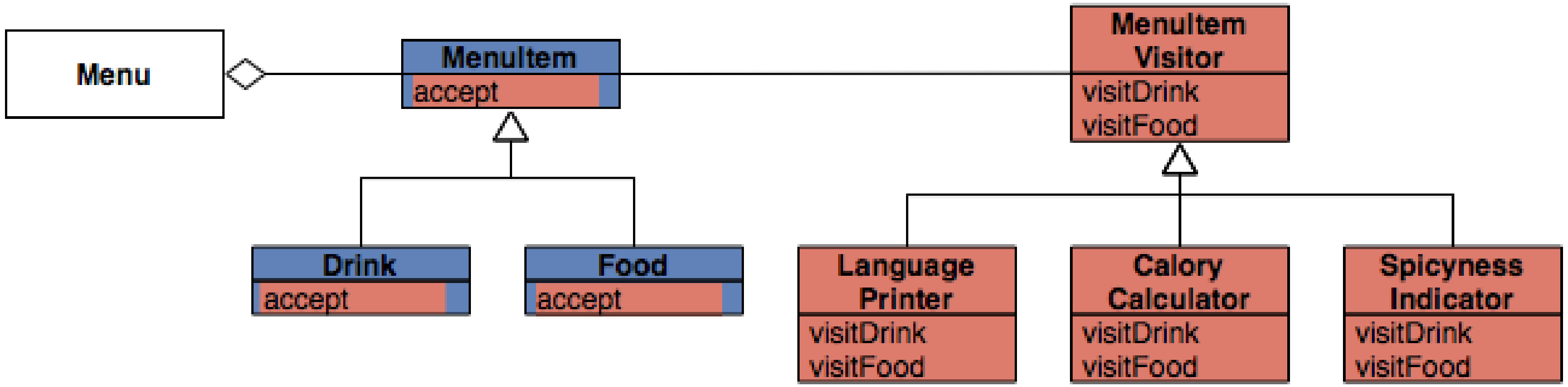
files



Decomposition still not the right one

- What would be the right decomposition?

Smalltalk class extensions



MenuItem
Visitor

- Packages defines classes and/ s
  - Can be different versions, under control of different people/project/companies

Note: declarative packages

Package systems should support software engineering and design principles

- e.g. packaging Visitor pattern

Approaches exist but should become mainstream

- Smalltalk’s class extensions
- C# Partial classes and extension methods
- Java Open Classes (for example in MultiJava)
- …

PS: or multi-methods in Lisp (and from there other languages)

Software-engineering wise

Important to be able to separate development into logical, manageable pieces

- e.g. Visitor design pattern

Each piece should have:

- owners & responsibles
- versions
- dependencies
- post-load and pre-unload statements

Corollary

Good packages support evolution

- Company can sell parsetree
- Other company can sell visitor for parsetree

Code repositories and packages should support flexible forms of packaging code

Code repositories, packaging & storage are linked

Sidenote

Design question:

- why is the plug-in mechanism in Eclipse so difficult?

Last but not least

We discussed granularity

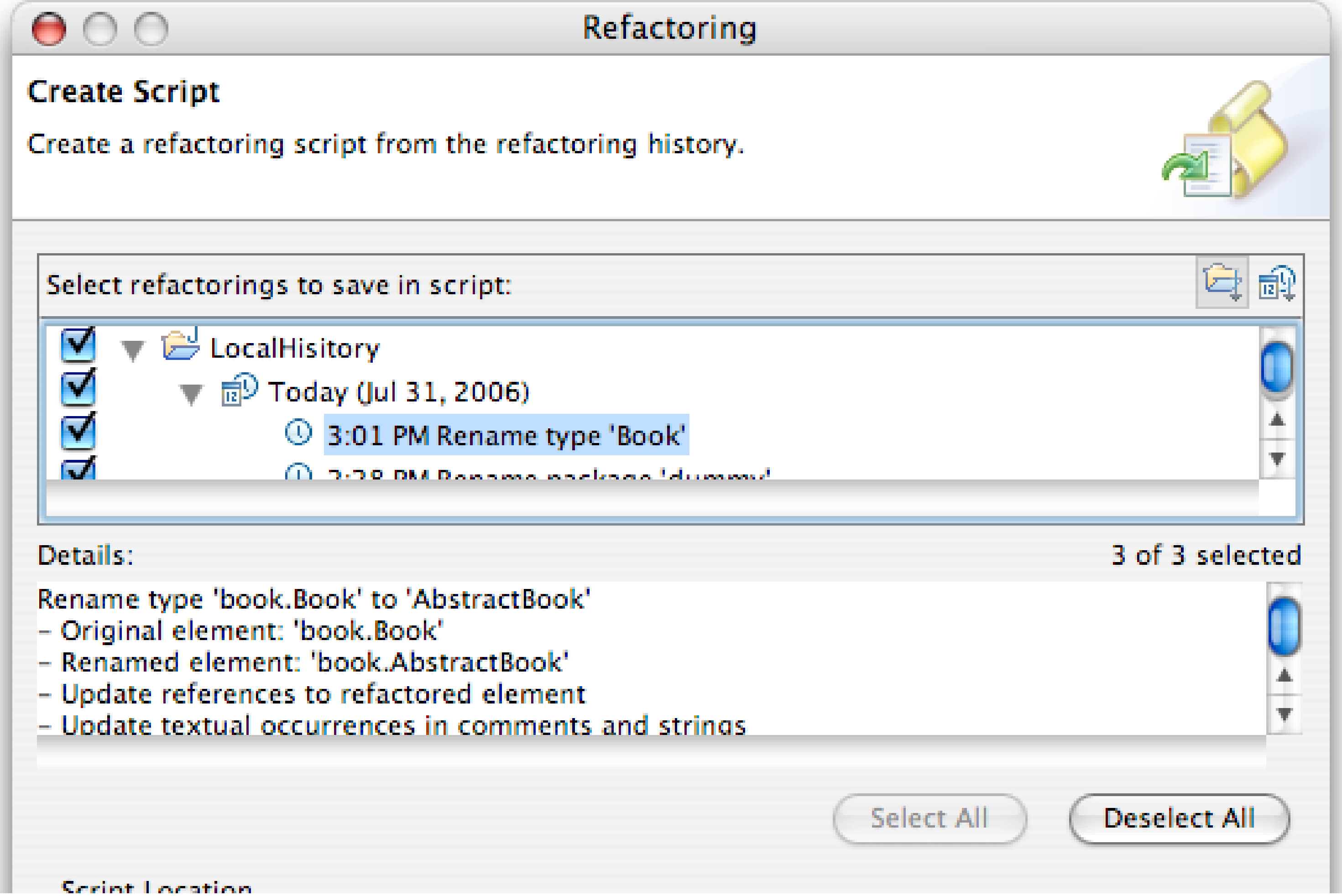
- want to see the development you really did, not the changes you made

Nice example: Refactoring Scripts in Eclipse

- Record and replay the refactorings you did

Why is this practical ?

Saving & Replaying Refactoring Scripts



Refactoring

Create Script

Create a refactoring script from the refactoring history.

Select refactorings to save in script:

- ☑ LocalHisitory
  - ☑ Today (Jul 31, 2006)
    - ☑ 3:01 PM Rename type 'Book'
    - ☑ 2:28 PM Rename package 'dummy'

Details:

3 of 3 selected

Rename type 'book.Book' to 'AbstractBook'
– Original element: 'book.Book'
– Renamed element: 'book.AbstractBook'
– Update references to refactored element
– Update textual occurrences in comments and strings

Conclusion

Need for supporting Multi-user development

- code repositories with concurrent access
- version support
- (automatic) merge support
- configuration management

Current systems are quite weak

- svn/git/… & files
- proper packaging mechanisms
- watch out for newer offerings

References

Subversion: https://subversion.apache.org/

git: http://git-scm.com/book

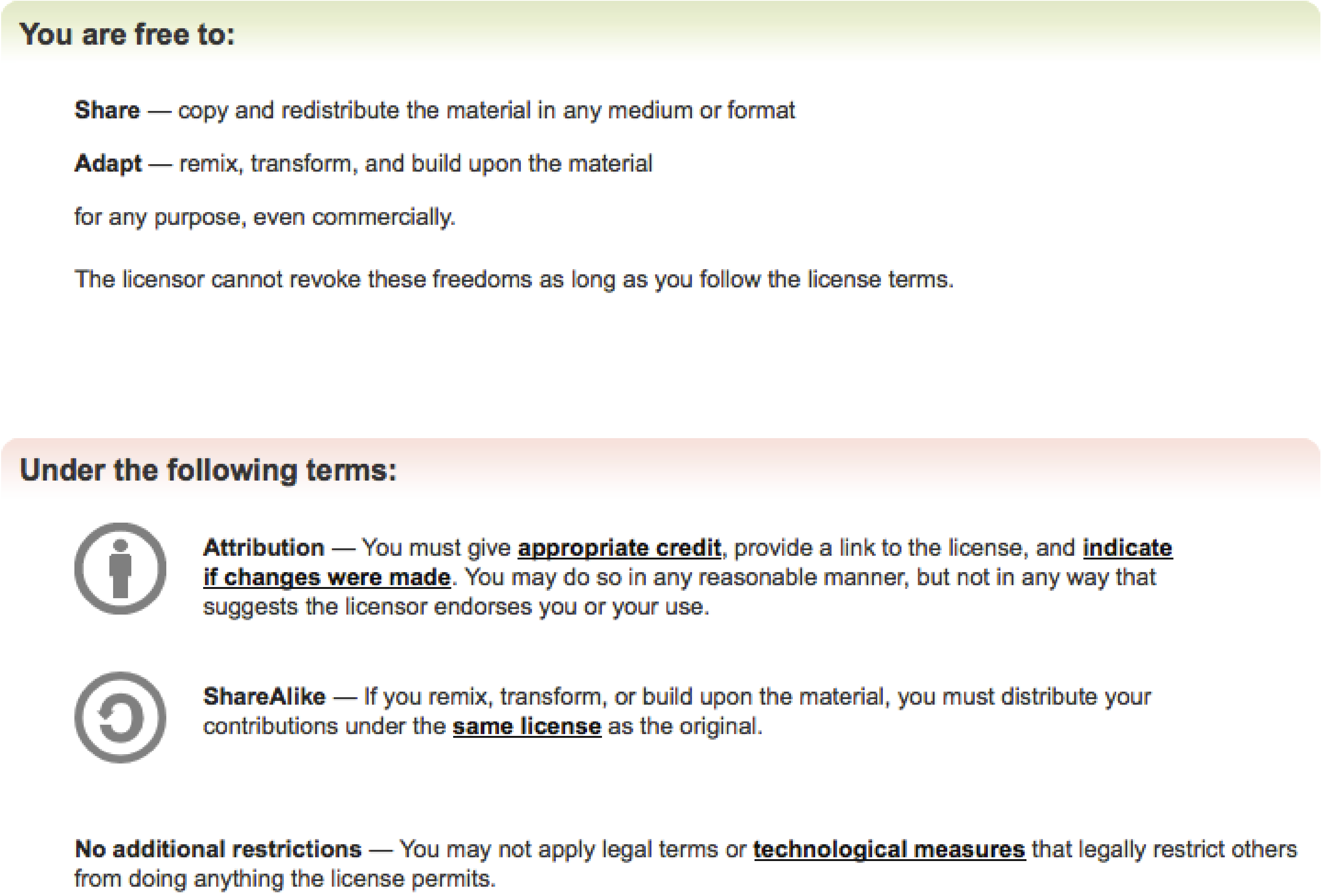
Envy overview: http://stephane.ducasse.free.fr/FreeBooks/ByExample/36%20-%20Chapter%2034%20-%20ENVY.pdf

Envy: Joseph Pelrine, Alan Knight, Adrian Cho, Mastering ENVY/Developer, Cambridge University Press, 2001.

Smalltalk Class Extensions: https://www.youtube.com/watch?v=VNi_VQMosXQ

License: Creative Commons 4.0

http://creativecommons.org/licenses/by-sa/4.0/



You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give **appropriate credit**, provide a link to the license, and **indicate if changes were made**. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the **same license** as the original.

No additional restrictions — You may not apply legal terms or **technological measures** that legally restrict others from doing anything the license permits.