

# 2IPC0 Programming Methods

## From Small to Large Programs

Loek Cleophas

Eindhoven University of Technology

Department of Mathematics & Computer Science

Software Engineering & Technology Group

<http://canvas.tue.nl/courses/473>

# Overview

---

- Recap Lecture 1
- Guidelines for (functional) decomposition
- Dealing with errors in engineering
- Unit testing
- Design patterns: Strategy Pattern

## Recap Manage Complexity

---

Technique: 'Divide and Conquer' (and Rule)

General problem solving technique:

1. Split problem into subproblems (Decomposition)
2. Solve subproblems independently (Recursion possible)
3. Combine subproblem solutions into total solution (Integration)

## Recap Divide, Conquer, and Rule with Java Methods

---

- Specify a subproblem

*Divide*

Syntax: method name, (formal) parameters, return type

Semantics: Contract with pre, post, return

- Create (design and implement) subproblem solution

*Conquer*

Method body

- Use subproblem solutions

*and Rule*

Method invocations with arguments (actual parameters)

## Recap Contract: concerns and benefits

Party	Two-sided Contract	
	Precondition	Postcondition
	User concern	benefit
	↓	↑
Provider	benefit	→ concern

**Contract** specifies *which* problem is solved, by **precondition** and **postcondition**.

**User** needs not know *how* the problem was solved.

**Provider** needs not know *what* solution is used for.

## Recap Specification by Contract as Interface

---



Contract as **interface** between **user** and **provider** of solution

- Relate invocation (call) to contract, implementation to contract.
  - *Never* relate invocation and implementation directly.
- ‘Divide’ fails → complexity and errors → no ‘conquering’.

## Recap Test-Driven Development (TDD)

---

1. Gather and analyze requirements
2. Select requirement to develop
3. Specify class/methods informally: javadoc summary sentences
4. Specify formally: model with invariants, headers and contracts  
Class has no data representation and empty method bodies
5. Implement rigorous tests, in unit test class
6. Choose data representation and implement class methods
7. Test the implementation, and fix defects
8. ... (see `test-driven-development.pdf`)

## Techniques (Methods)

---

- Product: Strive for modular structure in software.
- Process (way of working): Test-Driven Development (TDD)

This is also modular: distinct steps/phases



# Manage Complexity

---

Why?

# Manage Complexity

---

Why?

Intellectual limits of human beings, concerning

- productivity
- memory
- communication
- accuracy
- . . .

## Techniques to Manage Complexity

---

- Divide & Conquer
- Separation of Concerns
- Decomposition
- Modularization
- Encapsulation
- Abstraction

All related

# Abstraction

---

To abstract *from something*.

- Ignore/suppress distinctions that are considered irrelevant.
- Separate relevant from irrelevant details.

Example: Identify fractions  $a/b$  and  $c/d$  ( $b \neq 0 \neq d$ ) when  $ad = bc$

2IT60: equivalence relation, equivalence class, quotient set

Abstract from *how* a subproblem is solved, through a specification.

Abstract from *which* subproblem is solved, through parameters.

Abstract from *how* a solution is used, through a specification.

## Abstraction Example: Assignment Rectangle

---

On **input** are **6 integers**, representing three points in a Cartesian coordinate system, for each point first the x- and then the y-coordinate.

The first two points designate a **rectangle** with axes-parallel sides.

The first point is the **top left corner** of the rectangle, the second point is the **bottom right corner**.

The program has to **decide** whether the third **point** is **inside the rectangle** (including the edges) or outside.

The program should **output** an **error message** if the rectangle is **ill-defined**, i.e., if the second point is not to the right of and below the first point. The program may assume that no other errors occur in the input.

## Abstraction Example: Monolithic Solution, Variables

---

```
1 // Monolithic solution
2
3 import java.util.Scanner; // for input
4
5 class Rectangle1 {
6
7     Scanner scanner; // input source
8     int tlx; // top left x
9     int tly; // top left y
10    int brx; // bottom right x
11    int bry; // bottom right y
12    int px; // point x
13    int py; // point y
14
```

## Abstraction Example: Monolithic Solution, Method

---

```
15  void doRectangle() {
16      scanner = new Scanner(System.in);
17      tlx = scanner.nextInt();
18      tly = scanner.nextInt();
19      brx = scanner.nextInt();
20      bry = scanner.nextInt();
21      px = scanner.nextInt();
22      py = scanner.nextInt();
23
24      if (tlx > brx || tly > bry) {
25          System.out.println("error");
26          return;
27      }
28      // tlx <= brx && tly <= bry
```

## Abstraction Example: Monolithic Solution, Method

---

```
29     if (tlx <= px && px <= brx && tly <= py && py <= bry) {  
30         System.out.println("inside");  
31     } else {  
32         System.out.println("outside");  
33     }  
34 }
```



## Abstraction Example: Separate Methods, Top Level + Output

---

```
15     void doRectangle() {
16         scanner = new Scanner(System.in);
17         readInput();
18
19         if (! isValidRectangle()) {
20             System.out.println("error");
21             return;
22         }
23         // input rectangle is valid
24         if (isPointInsideRectangle()) {
25             System.out.println("inside");
26         } else {
27             System.out.println("outside");
28         }
29     }
```

## Abstraction Example: Separate Methods, Input

---

```
31      // Read input into tlx, ..., py via scanner
32      void readInput() {
33          tlx = scanner.nextInt();
34          tly = scanner.nextInt();
35          brx = scanner.nextInt();
36          bry = scanner.nextInt();
37          px = scanner.nextInt();
38          py = scanner.nextInt();
39      }
```

Note that `readInput()` operates on the global variables.

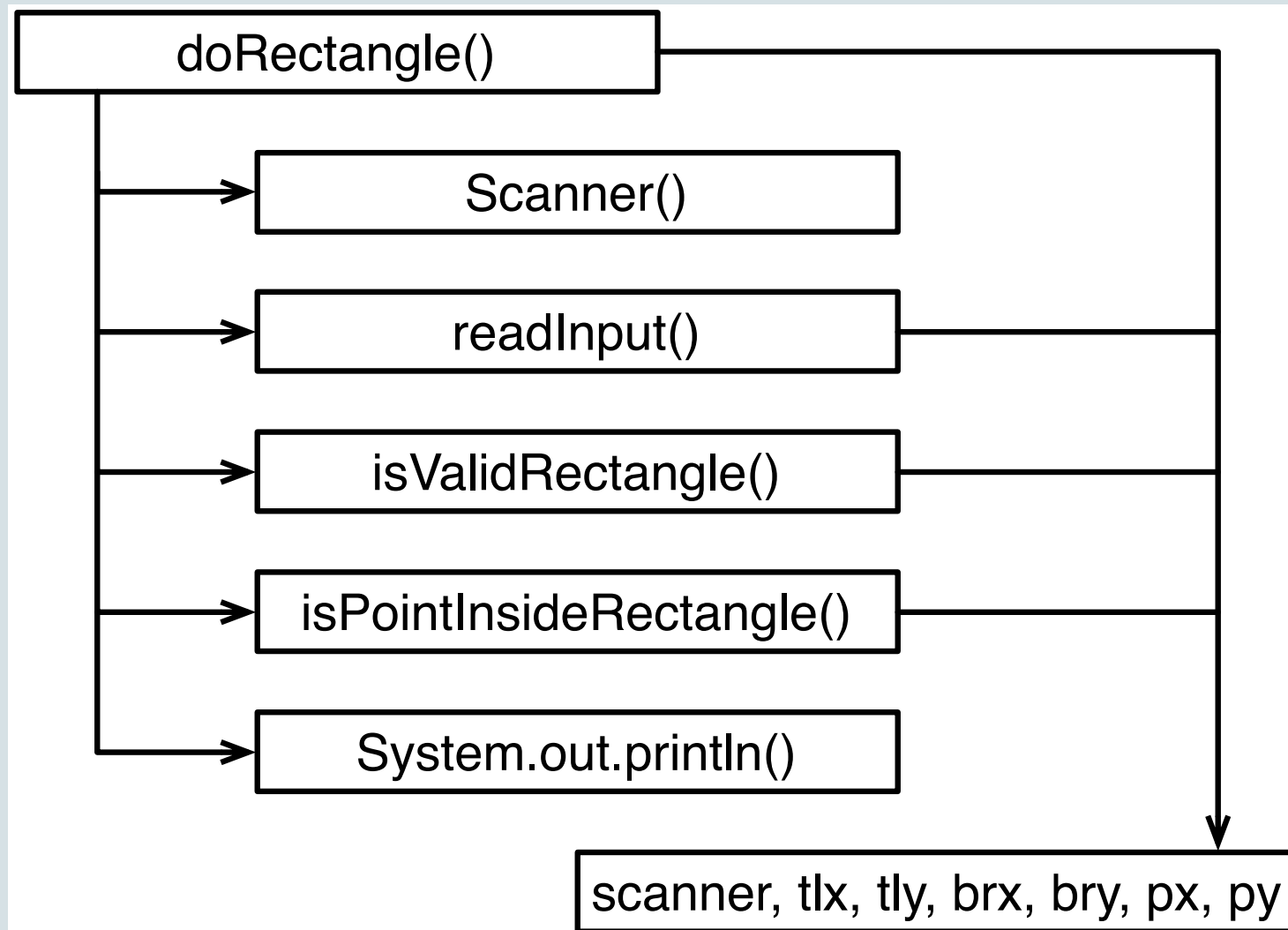
## Abstraction Example: Separate Methods, Calculations

---

```
1  // Returns whether tlx, ..., bry is a rectangle
2  boolean isValidRectangle() {
3      return tlx <= brx && tly <= bry;
4  }
5
6  // Returns whether px, py is inside rectangle tlx, ..., bry
7  boolean isPointInsideRectangle() {
8      return tlx <= px && px <= brx && tly <= py && py <= bry;
9  }
```

Note that these methods also operate on the global variables.

## Abstraction Example: Dependence Diagram 1



## Dependence Diagrams

---

Thinking in terms of dependencies is key to good design.

- $\boxed{A} \longrightarrow \boxed{B}$  means “ $A$  depends on  $B$ ”.
- When code of  $B$  changes, code of  $A$  may also have to change.
- Code of  $A$  can change, without affecting code of  $B$ .
- When reusing code of  $A$ , also code of/like  $B$  is needed.
- When reusing code of  $B$ , code of  $A$  is not needed.

## Abstraction Example: Parameterized Methods, Top Level

---

```
14    void doRectangle() {
15        readInput(new Scanner(System.in));
16
17        if (! isValidRectangle(tlx, tly, brx, bry)) {
18            System.out.println("error");
19            return;
20        }
21        // input rectangle is valid
22        if (isPointInsideRectangle(tlx, tly, brx, bry,
23                                px, py)) {
24            System.out.println("inside");
25        } else {
26            System.out.println("outside");
27        }
28    }
```

## Abstraction Example: Parameterized Methods, Input

---

```
29      // Read input into global tlx, ..., py via scanner
30      void readInput(Scanner scanner) {
31          tlx = scanner.nextInt();
32          tly = scanner.nextInt();
33          brx = scanner.nextInt();
34          bry = scanner.nextInt();
35          px = scanner.nextInt();
36          py = scanner.nextInt();
37      }
38
```

## Abstraction Example: Parameterized Methods, Calculations

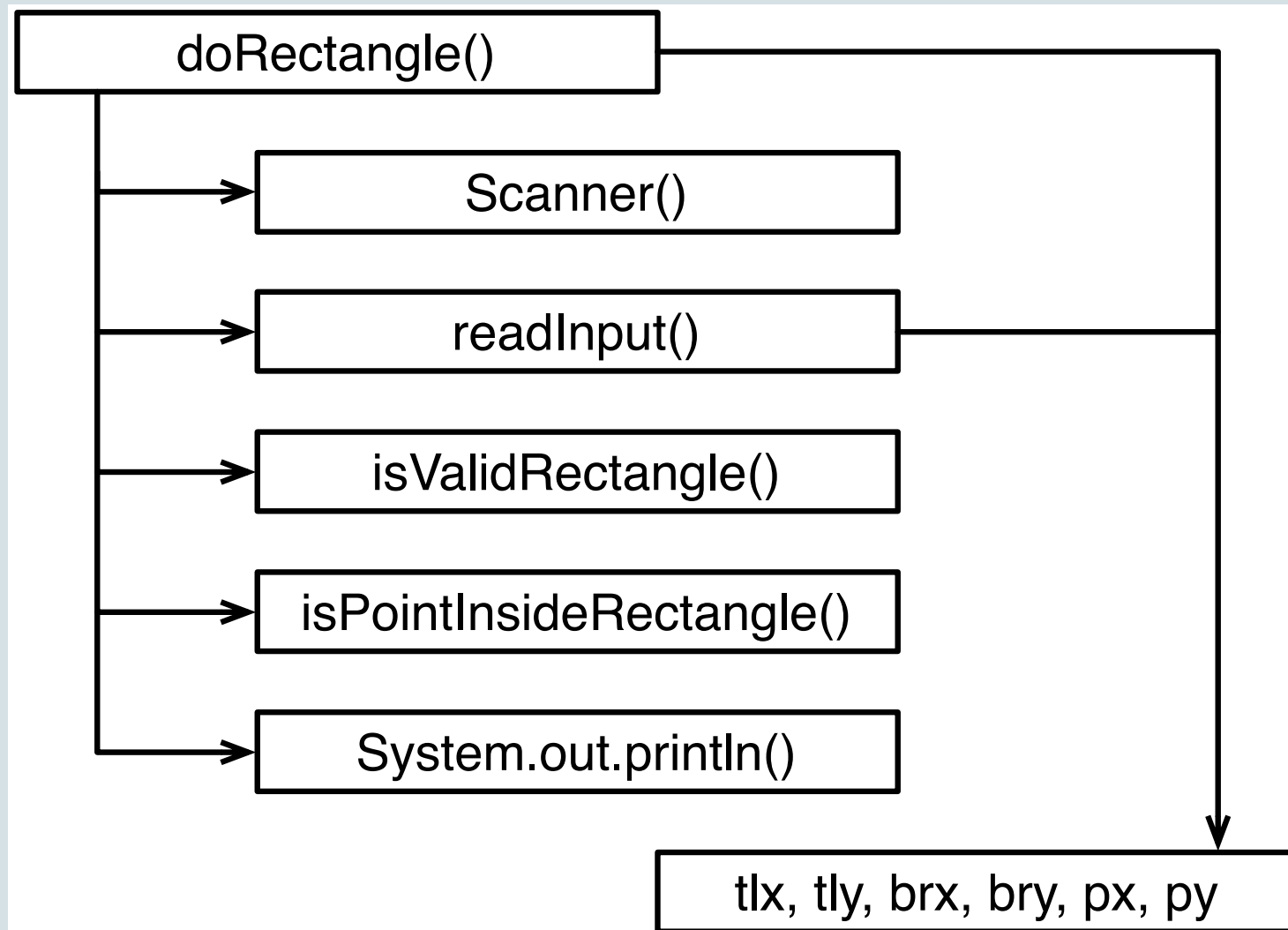
---

```
39      // Returns whether tlx, ..., bry is a rectangle
40      boolean isValidRectangle(int tlx, int tly,
41                               int brx, int bry) {
42          return tlx <= brx && tly <= bry;
43      }
44
45      // Returns whether x, y is inside rectangle tlx, ..., bry
46      boolean isPointInsideRectangle(int tlx, int tly,
47                                    int brx, int bry,
48                                    int x, int y) {
49          return tlx <= x && x <= brx && tly <= y && y <= bry;
50      }
```

Note that these methods now do *not* involve the global variables. Instead, they operate on parameters only.



## Abstraction Example: Dependence Diagram 2



## Abstraction Example: Reduce Code Duplication

---

- Design principle: DRY = Don't Repeat Yourself
- Avoid code duplication
- Also avoid near duplication
- Compare

```
return tlx <= brx && tly <= bry;
```

to

```
return tlx <= x && x <= brx && tly <= y && y <= bry;
```

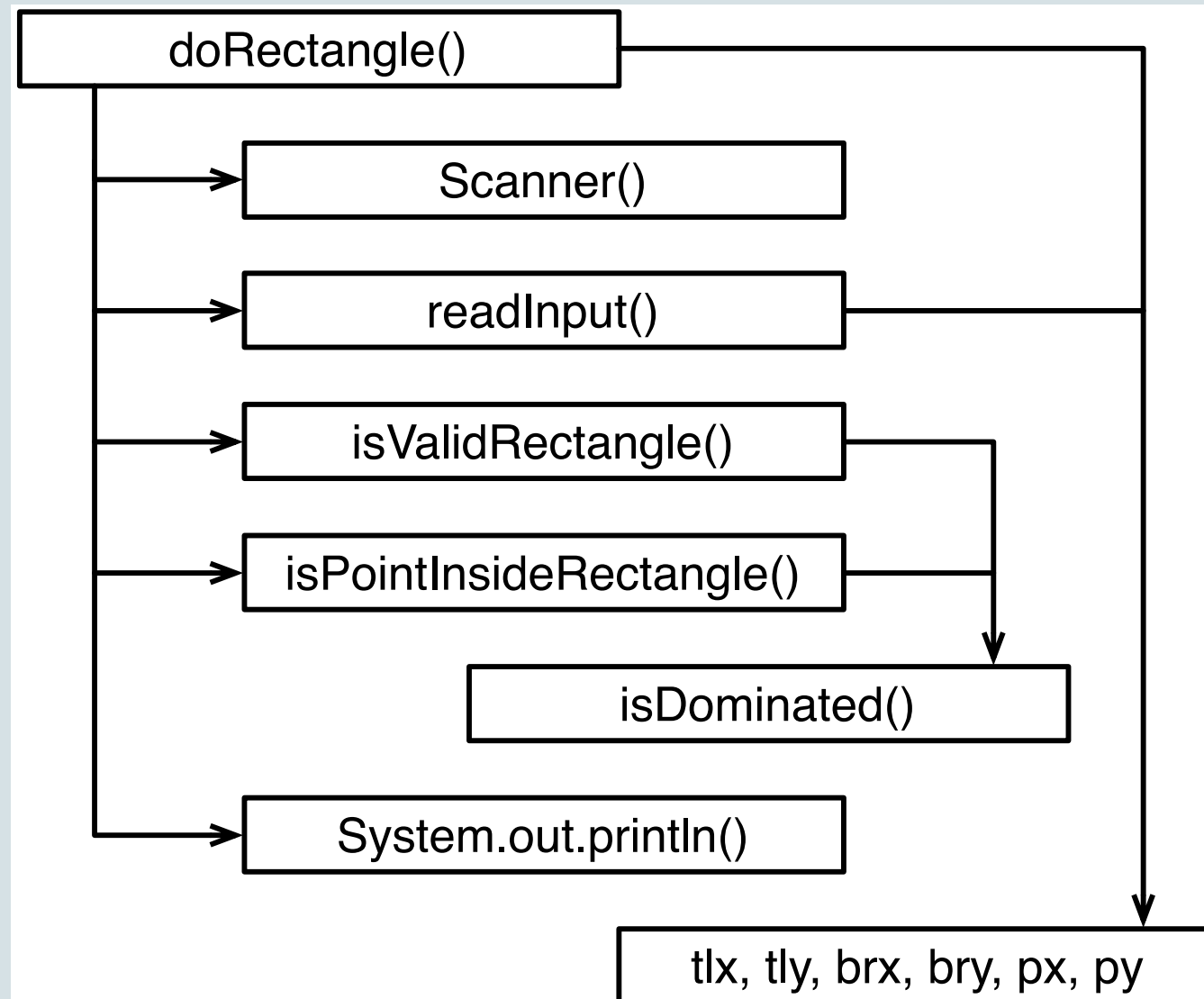
- Use parameterized auxiliary methods to avoid duplication.

## Abstraction Example: Parameterized Methods, Auxiliaries

---

```
47  // Returns whether tlx, ..., bry is a rectangle
48  boolean isValidRectangle(int tlx, int tly,
49                          int brx, int bry) {
50      return isDominated(tlx, tly, brx, bry);
51  }
52
53  // Returns whether x, y is inside rectangle tlx, ..., bry
54  boolean isPointInsideRectangle(int tlx, int tly,
55                               int brx, int bry,
56                               int x, int y) {
57      return isDominated(tlx, tly, x, y) &&
58             isDominated(x, y, brx, bry);
59  }
60
61  // Returns whether ax, ay is dominated by bx, by
62  boolean isDominated(int ax, int ay, int bx, int by) {
63      return ax <= bx && ay <= by;
64  }
```

## Abstraction Example: Dependence Diagram 3



## Abstraction Example (cont'd)

---

- Separate input, calculations, and output
- Parameterize methods

N.B. In this style, it is harder to parameterize `readInput` further, because Java has no output parameters, only return values.

(No Java method with parameters  $v$ ,  $E$  equivalent to  $v := E$  exists.)

`readInput()` could return an array, but then the indexing would be awkward. Separate classes for rectangles and points is the Java way.

Doing output could be separated from the top-level *coordination*, using a string result.

# Advantages and Disadvantages of Divide & Conquer

---

- Advantages?
- Disadvantages?

## Advantages of Divide & Conquer

---

- Enables **solution** of more complex problems.
- Organizes **communication** about *solution* domain.
- Facilitates **parallel construction** by a team.
- Improves ability to **plan work and track progress**.
- Improves **verifiability** (facilitates “getting it to work”):
  - Allows early **review** of design.
  - Allows **unit testing** of separate components.
  - Allows **stepwise integration** (avoiding a “big bang”).
- Improves **maintainability**: changes affect few components.
- Improves possibilities for **reuse**.

## Disdvantages of Divide & Conquer

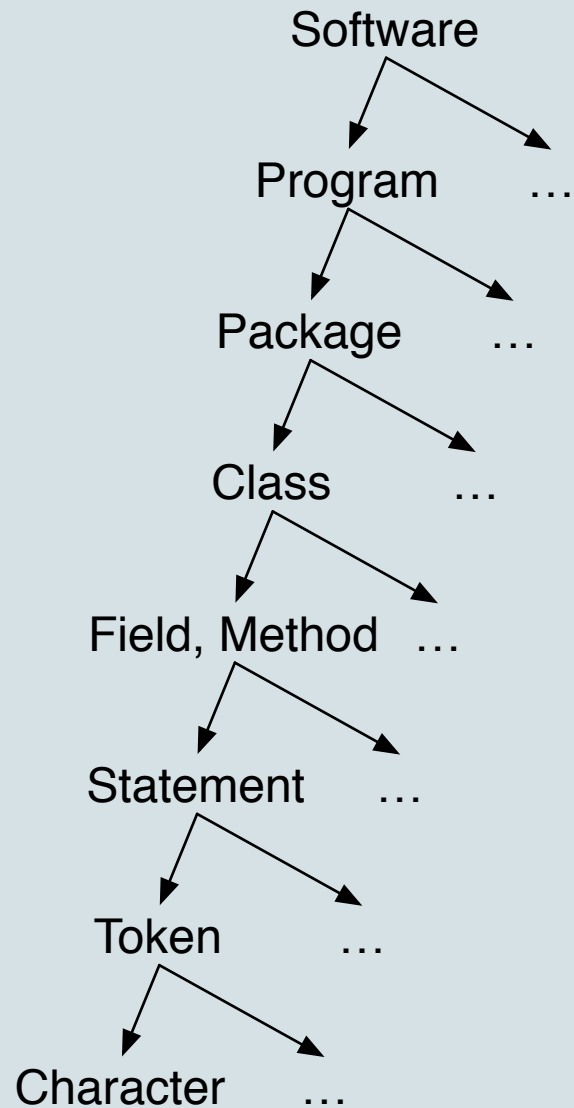
---

- Top-level divisions are hard to make: can have big consequences.  
Need experience and need to experiment (try alternatives).
- Separation often brings **overhead** (to cross boundary/interface).  
Also depends on implementation facilities (programming language).
- The result looks bigger (overhead, again).  
But it has **explicit structure** (there is a price to that).
- Functions cannot be specified/designed/implemented in *isolation*.  
They need to be considered in related groups (cf. class).



# Modularization Applies to Multiple Levels

---



## Guidelines for Functional Decomposition

---

- Maximize cohesion

Cohesion = measure of relatedness of 'things' *inside* a 'module'

Keep related things together; separate unrelated things.

- Minimize coupling, also stated as minimize dependency

Coupling = measure of connectedness *between* 'modules'

*Interface of a 'module'* concerns all its external connections.

Consider both *number and nature of dependencies*:

++ via parameter

□ use constant, type, (variable,) method defined elsewhere

-- implicit common 'knowledge' (file name in string literal)

## Guidelines for Functional Decomposition (cont'd)

---

- Each 'function' should solve a single well-defined problem.

Single Responsibility Principle: SRP

- If specification is difficult to provide, then something is wrong.
- Specify on the right level of data abstraction!
- Limit the size of the interface, i.e., number of parameters.

Do not pass unnecessary data.

Possibly combine parameters (in a new class, representing a new concept, raising the level of abstraction).

## Guidelines for Functional Decomposition (cont'd)

---

- Each problem should be solved in one place only.

Don't Repeat Yourself: DRY

- Avoid code duplication, a.k.a. code clones.
- Copy-Paste-Edit is a bad way to reuse code. (Why?)
- Use parameters to unify similar code.

## Guidelines for Functional Decomposition (cont'd)

---

- Each 'function' should have a small/simple/clear implementation.
- No more than a (few) dozen lines, with limited nesting depth.
- Nested control structures should invite further subdivision, especially nested loops.
- Cyclomatic complexity is a good metric.

Checkstyle measures this and warns when too high.

- If it is above 8, consider subdivision, a.k.a. Refactoring.

NetBeans can assist:

Select code, right-click, Refactor > Introduce Method...

## Further Abstraction Example: Data Abstraction

---

Problem *Rectangle*

```
doRectangle(int tlx, int tly, int brx, int bry, int x, int y)
```

## Further Abstraction Example: Data Abstraction

---

Problem *Rectangle*

```
doRectangle(int tlx, int tly, int brx, int bry, int x, int y)
```

Group parameters to form Point and Rectangle:

```
boolean contains(Rectangle r, Point p) { ... }
```

or in **class** Rectangle:

```
/** @return whether this contains p (incl. boundary) */  
boolean contains(Point p) { ... }
```

applied as

```
Rectangle r; Point p; ... r.contains(p) ...; ...
```

(We treat data abstraction in more detail later.)

## Generalization

---

- Consider possibilities for **generalization**: solving a more general problem than (initially) required.

Generalization improves (re)usability and insight.  
It can also simplify solution!

- **Generalize by means of parameters**, not via global variables.

Parameters abstract from the identity of the data operated on.

**Code duplication** should invite **refactoring** by parameterization.

- Decomposing a problem into simpler versions of itself gives rise to **recursion**.
- Weigh benefits against costs: too general makes use costly.



## Generalization Example

---

How many regions arise when space is cut by five arbitrary planes?

‘Arbitrary’ means: No two planes are parallel, and no three planes have a point in common.

## Generalization Example

---

How many regions arise when space is cut by five arbitrary planes?

‘Arbitrary’ means: No two planes are parallel, and no three planes have a point in common.

Parameterize on number of planes  $N$

## Generalization Example

---

How many regions arise when space is cut by five arbitrary planes?

‘Arbitrary’ means: No two planes are parallel, and no three planes have a point in common.

Parameterize on number of planes  $N$

Parameterize on dimension  $D$  of ‘space’ to be divided :

$D = 1 \rightarrow$  line,  $D = 2 \rightarrow$  plane,  $D = 3 \rightarrow$  space

## Generalization Example: Solution

---

$R(D, N)$  = # regions in  $D$ -space when cut by  $N$  arbitrary  $D - 1$ -spaces

## Generalization Example: Solution

---

$R(D, N)$  = # regions in  $D$ -space when cut by  $N$  arbitrary  $D - 1$ -spaces

$R(D, 0) = 1$  (when not divided)

$R(1, N) = N + 1$  (a line divided by  $N$  points)

$R(D, N) = R(D, N - 1) + R(D - 1, N - 1)$  if  $2 \leq D$  and  $1 \leq N$

## Generalization Example: Solution

---

$R(D, N)$  = # regions in  $D$ -space when cut by  $N$  arbitrary  $D - 1$ -spaces

$R(D, 0) = 1$  (when not divided)

$R(1, N) = N + 1$  (a line divided by  $N$  points)

$R(D, N) = R(D, N - 1) + R(D - 1, N - 1)$  if  $2 \leq D$  and  $1 \leq N$

$R(D, N)$		$N \rightarrow$					
		0	1	2	3	4	5
$D$ $\downarrow$	1	1	2	3	4		
	2	1	2	4	7	11	
	3	1	2	4	8	15	26

## Typical Purposes of 'Functions'

---

- Input, calculate, output
- Parse, convert, format
- De-nest (disentangle) control structures
- Initialize, query, update, finalize
- Check condition
- Create, delete
- Coordinate activities, handle events
- Abstract from the way data is represented

## 'Function' Names, Parameter Order

---

- Use verb-plus-noun, but avoid overly general words (`processInput`).

Do not include the object name that **this** refers to.

In **class** `Document` do not name a method `printDocument`.

- Function name should reflect returned value.
- Procedure name should reflect postcondition.
- Short parameter names facilitate writing of specifications.
- Parameter order: 1 input-only, 2 input-and-output, 3 output-only (status and error information last).



## Engineering gone wrong

---

Do you know examples of engineering 'inconveniences' and disasters?

Were you ever the victim of an engineering mistake?

Did you ever make an engineering mistake? How about BIG ones?

What is the most memorable engineering mistake you made?

## Terminology: IEEE Classification

---

**Failure**: product deviates from requirements during use/operation

**Defect, fault**: anomaly in a product that can somehow (eventually) lead to a failure; product can be any artifact (e.g. a design document)

**Mistake**: human action (“slip”) causing a fault

**Error**: difference between actual and specified/expected result

These notions assume existence of requirements/specification/contract as a frame of reference, which needs to be established in advance

N.B. **Beautiful versus ugly**: an opinion, not an error

## Economy of Defects

---

- The longer a defect is undiscovered, the higher its cost: grows exponentially in distance between injection and elimination.
- Defects decrease the predictability of a project: cost (time) of defect localization and repair are extremely variable.
- Defects concern risks, i.e. uncertainty; product could be defect-free at once, but defects are likely.
- The likelihood of defects increases rapidly with higher system complexity.

## Kinds of Mistakes

---

There are many different kinds of mistakes, such as

## Kinds of Mistakes

---

There are many different kinds of mistakes, such as

- Misunderstanding a customer
- Misinterpreting a written requirement
- Overlooking a case in analyzing the requirements
- Adopting an incorrect algorithm
- Making an unwarranted assumption when implementing a method
- Making a typing error (typo) when entering code

## Kinds of Mistakes

---

There are many different kinds of mistakes, such as

- Misunderstanding a customer
- Misinterpreting a written requirement
- Overlooking a case in analyzing the requirements
- Adopting an incorrect algorithm
- Making an unwarranted assumption when implementing a method
- Making a typing error (typo) when entering code

No single method can cure all this.

## Dealing with Mistakes

---

1. **Admit** that people make mistakes and inject defects: awareness
2. **Prevent** them as much as possible
3. **Minimize** consequences: fault tolerance, defensive programming
4. **Detect** presence of defects as early as possible
5. **Localize** defects
6. **Repair** defects
7. **Trace** them: root causes and possible other consequences
8. **Learn** from them: improve the process and tools

## Detecting Defects

---

- Reviewing :
  - Examine an artifact *with the intent of finding defects*.
  - Can be done early in the development process.
  - Often localizes the defects as well.
  - Can and should also be applied to code.
- Testing :
  - Use a product systematically *with the intent of finding defects*.
  - Works through *failures*; does not localize underlying defects.
  - Requires a working product (part).

Reviewing and testing complement each other.



## Limits of Testing

---

Testing in itself does not create quality; it is a *measure* of quality.

Edsger W. Dijkstra (CACM, 1972; winner of Turing Award 1972):

“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”

Dijkstra’s advice: Prove mathematically that an artifact has required properties. Ideally: let proof development drive the design, leading to **Correctness by Construction**.

## Removing Defects

---

Testing and debugging are two very different activities

**Testing**: The process of systematically executing software with the intent of *detecting the presence of defects*.

**Debugging**: The process of localizing, diagnosing, and correcting *detected defects*.

Debugging is a time consuming, unpredictable, frustrating process.

Good testing effort prevents/simplifies debugging

## Testing Terminology

---

Test case :

- has a purpose/intent (motivation);
- controls activation and input;
- observes response and output;
- decides on pass/fail based on expected (specified) result.

What input(s) to offer?

What result(s) to inspect?

How to decide on pass/fail?

## JUnit: Automated Unit Testing Framework

---

JUnit: organizes code for test cases, runs them, reports results

N.B. Use JUnit 4.x, *not* JUnit 3.8.x

Each test case is coded in a method: `@Test public void ...()`

Motivate each test case in (javadoc) comment

Checking and reporting: `assertTrue`, `assertEquals`, `fail`, ...

Can also test for required exceptions: no/wrong exception → failure

`@Test(expected = NullPointerException.class)`

Can set time limit (in ms)

`@Test(timeout = 1000)`

## JUnit: Execution

---

```
1 import static org.junit.Assert.*;
2 import org.junit.After;
3 import org.junit.Before;
4 import org.junit.Test;
5
6 public class MyClassTest { // Test cases for class MyClass
7     ... // global variables defining a 'test fixture'
8     @Before protected void setUp() { /* initialize fixture */ }
9     @After  protected void tearDown() { /* finalize fixture */ }
10    @Test public void testCase1() { /* code for test case 1 */ }
11    @Test public void testCase2() { /* code for test case 2 */ }
12 }

setUp(); testCase1(); tearDown();
setUp(); testCase2(); tearDown();
```

N.B. The execution order of test cases is not predetermined!  
Test cases should be *independently* executable.

## JUnit and NetBeans

---

See NetBeans IDE Sample Java Project *Anagrams* (via New Project).

Help > Javadoc References > JUnit 4.10 (or via 'More Javadoc...')

Right-click Java file in NetBeans project: Tools > Create JUnit Tests

File > New File... > JUnit > ...

Run > Test Project

Right-click project: Test

## Approaches to Selecting Input for Test Cases

---

### **Black-box, or test-to-specifications, or functional :**

Checks the functionality of the software.

Consider specification/requirements only. Ignore code.

### **Glass-box, or test-to-code, or structural :**

Checks the internal logic of the software.

Consider code only. Ignore specification/requirements.

## Techniques for Constructing Test Cases

---

- Boundary analysis: pick values on and near boundaries
- Equivalence classes: pick a case from every equivalence class
- Statement, branch, and path coverage
- Random input (should be reproducible through seeding )



## Coverage: Example

```

if (C) { v = 1; }
if (D) { w = 2; }
else { w = 3; }
    
```

5 (!) statements, 2 + 2 branches, 2 \* 2 paths

Test Cases				Coverage		
1	2	3	4	Statement	Branch	Path
$\neg C, \neg D$				60%	50%	25%
$C, D$				80%	50%	25%
$C, D$	$C, \neg D$			100%	75%	50%
$C, D$	$\neg C, \neg D$			100%	100%	50%
$C, D$	$C, \neg D$	$\neg C, D$	$\neg C, \neg D$	100%	100%	100%

## How to Decide on Pass/Fail

---

- Select trivial or highly structured input with known output
- ‘Manually’ determine expected output
- Check correctness of output via ‘inverse’ process;  
e.g. check whether a list is a sorted permutation of the input,  
rather than comparing it to a sorted version of the input;  
this is a must when input does not uniquely determine output
- Use alternate implementation to compute expected output;  
e.g. using a slower but simpler algorithm
- Cross check consistency of implementation;  
e.g. check whether sorting a list and sorting a permutation of that  
same list yield the same output

## Testing Advice

---

- Develop test cases before coding ( **Test-Driven Development** ): Forces you to consider contracts and to use interfaces *before* implementing them; can test immediately *after* implementing.
- Code & test incrementally (not everything together at once).
- Code & test simple parts first.
- Use assertions (built-in tests; **“fail early”** ): Test pre- and post-conditions, and ‘can’t-happen’ cases.
- Automate testing.
- Keep test software, data, motivation, and results.
- Re-test after making changes ( **regression testing** ).

## Design Patterns

---

- A **Design Pattern** is an *outline* of a *general* solution to a design problem, *reusable* in *multiple, diverse, specific* contexts.
- It is *not* a solution in the form of source code that can be reused “as is” .
- Examples: Strategy, Iterator, Observer
- The term *Design Pattern* comes from building architecture and was made popular for software by the book

*Design Patterns: Elements of Reusable Object-Oriented Software*

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995.

The authors are commonly referred to as the **Gang of Four** (GoF).

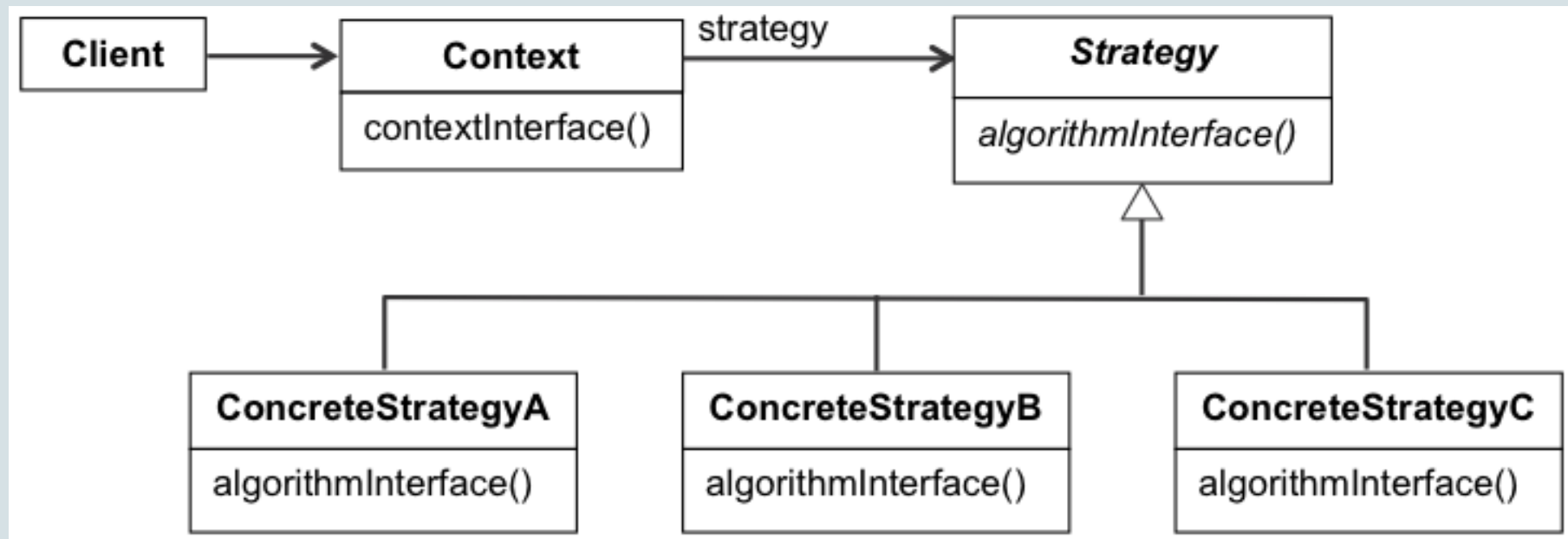
## Strategy Design Pattern (Burris, Chapter 7)

---

- Problem: Accommodate multiple implementations of same method
- Problem: Allow selection of implementation at runtime
- Solution:
  - Put specification of method in (abstract) class or interface.
  - Put implementations in subclasses of specification.
  - Declare variable with specification type.
  - Assign to variable a class of an implementation type.

## Strategy Design Pattern Diagram (Burris, Chapter 7)

UML diagram with *roles* and *relationships*:



## Strategy Design Pattern: Example (See JUnit4Examples)

---

```
1 public abstract class AbstractPowerClass {
2     /** ... Specification ... */
3     public abstract long power(int a, int b) throws ...;
4 }
5
6 public class PowerByRepeatedMultiplication extends AbstractPowerClass {
7     @Override
8     public long power(int a, int b) {
9         /* ... implementation by repeated multiplication */
10    }
11 }
12
13 public class PowerByRepeatedSquaring extends AbstractPowerClass {
14     @Override
15     public long power(int a, int b) {
16         /* ... implementation by repeated squaring/multiplication */
17    }
18 }
```

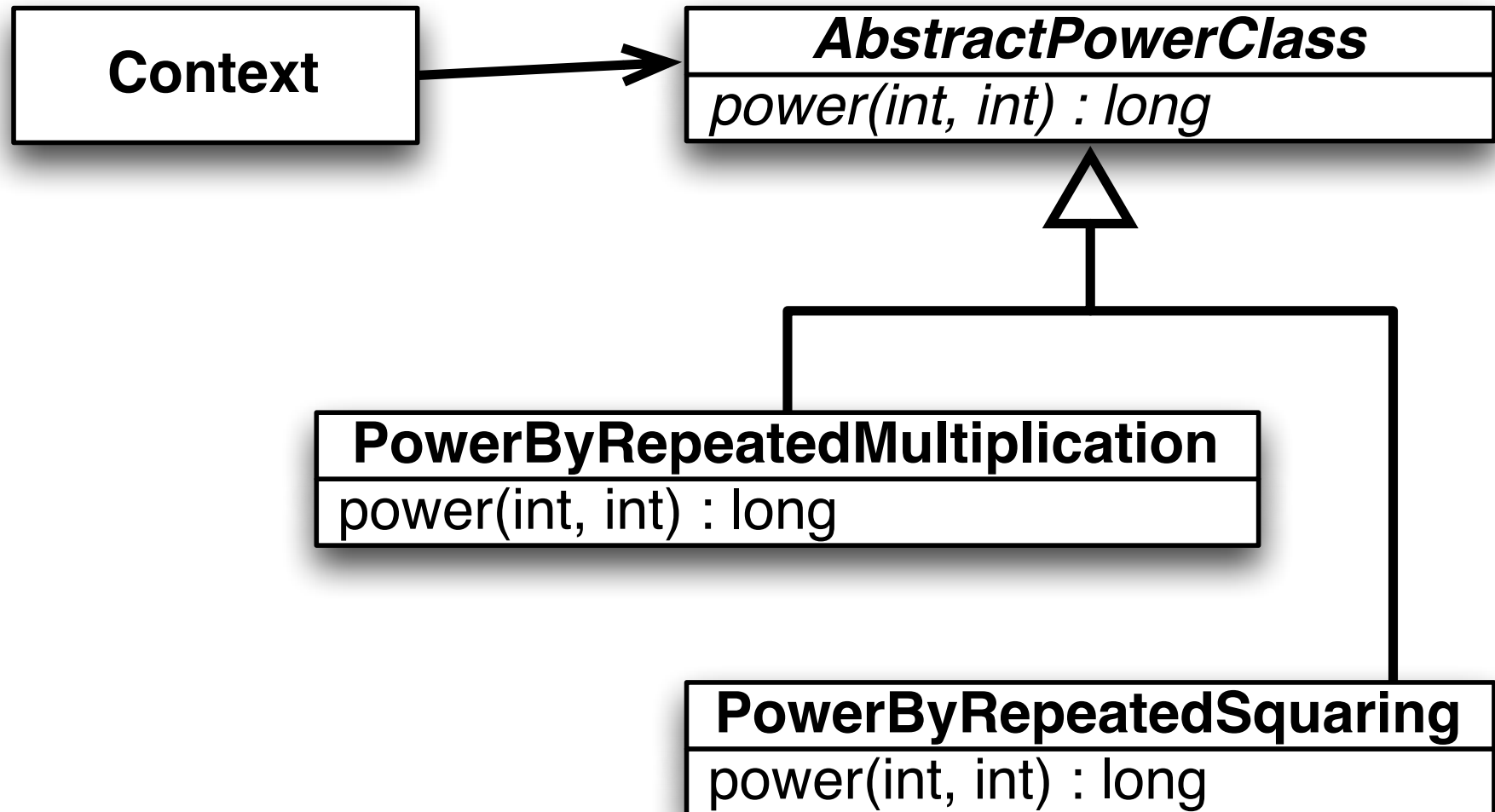
## Strategy Design Pattern: Example (continued)

---

```
1 public class Context {
2
3     // Declare variable using the specification class
4     private AbstractPowerClass myPower;
5
6     public Context(boolean slow) {
7         // choose concrete implementation
8         if (slow) {
9             myPower = new PowerByRepeatedMultiplication();
10        } else {
11            myPower = new PowerByRepeatedSquaring();
12        }
13
14        // use the chosen implementation
15        ... myPower.power(..., ...) ...
16    }
17 }
```



## Strategy Pattern Class Diagram



## Strategy Design Pattern: Example 2

---

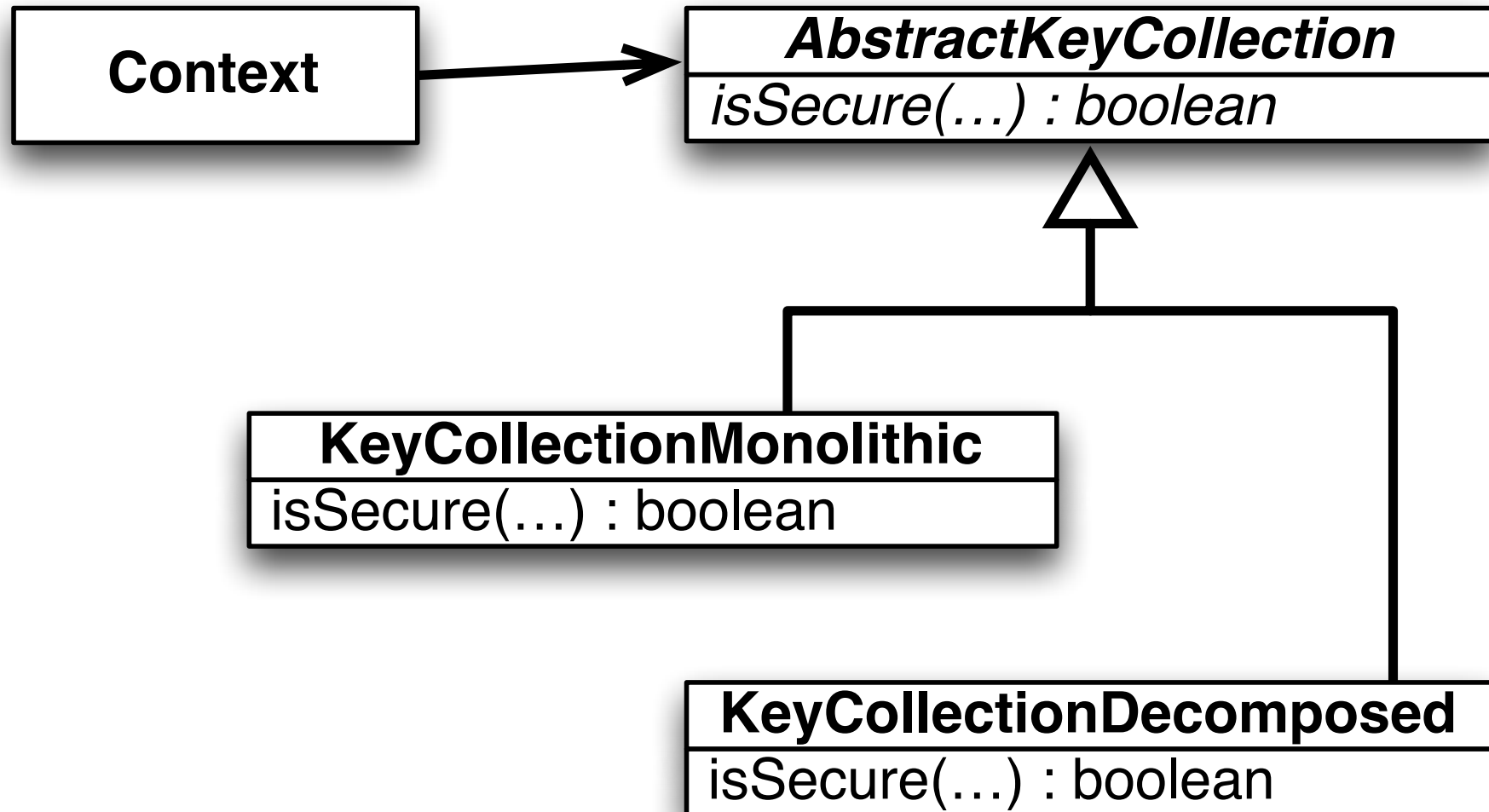
```
1 public abstract class AbstractKeyCollection {
2     /** ... Specification ... */
3     public abstract boolean isSecure(...);
4 }
5
6 public class KeyCollectionMonolithic extends AbstractKeyCollection {
7     @Override
8     public boolean isSecure(...) {
9         /* ... Monolithic implementation */
10    }
11 }
12
13 public class KeyCollectionDecomposed extends AbstractKeyCollection {
14     @Override
15     public boolean isSecure(...) {
16         /* ... Decomposed implementation */
17    }
18 }
```

## Strategy Design Pattern: Example 2 (continued)

---

```
1 public class Context {
2
3     // Declare variable using the specification class
4     private AbstractKeyCollection myKeyCollection;
5
6     public void useKeyCollection(boolean mono) {
7         // choose concrete implementation
8         if (mono) {
9             myKeyCollection = new KeyCollectionMonolithic();
10        } else {
11            myKeyCollection = new KeyCollectionDecomposed();
12        }
13
14        // use the chosen implementation
15        myKeyCollection.isSecure(...)
16    }
17 }
```

## Strategy Pattern Class Diagram 2



## Java `class` mechanism

---

The **`class`** concept plays central role in Java programming language.

It is very general and “powerful”, and can easily be abused.

So far, we use a **`class`** to collect *static variables and (static) methods*.

## Abstract Classes (no details, just the concept)

---

- A method can be declared **abstract**.
- An **abstract** method does not have an implementation:  
Instance variables and/or method bodies can be missing.
- A class containing **abstract** methods must be declared **abstract**.
- An **abstract class** cannot be instantiated.
- A subclass can define (additional) instance variables and implementations for inherited **abstract** methods, by *overriding*.

It is recommended to *annotate* such methods with `@Override`:

```
@Override  
public String stringOf() { ... }
```

## Submitting Work for Assignments in Momotor

---

- Name
- ID number
- Date
- Cut line --8<--

## Summary

---

- Guidelines for (functional) decomposition
- Dealing with errors in engineering
- Unit testing
- Design patterns: Strategy Pattern