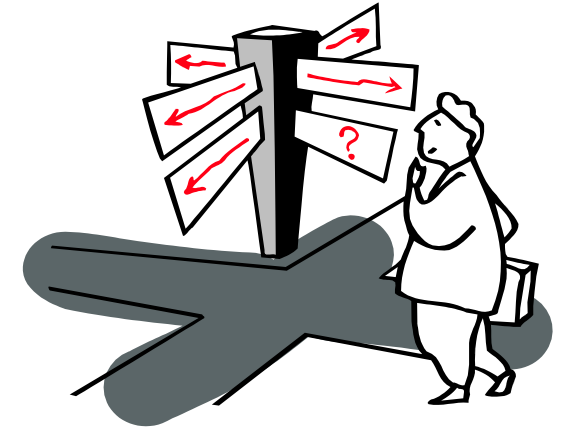


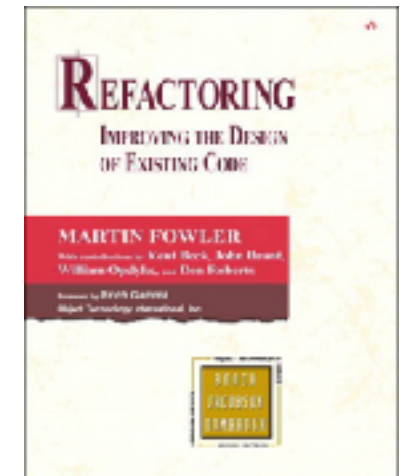
CHAPTER 11 – Refactoring

- Introduction
 - + When, Why, What?
 - + Which Refactoring Tools?
- Demonstration: Internet Banking
 - + Iterative Development Life-cycle
 - + Prototype
 - + Consolidation: design review
 - + Expansion: concurrent access
 - + Consolidation: more reuse
- Miscellaneous
 - + Tool Support
 - + Code Smells
 - + Refactoring God Class
 - An empirical study
 - + Scrum: Technical Debt
- Conclusion
 - + Correctness & Traceability



Literature

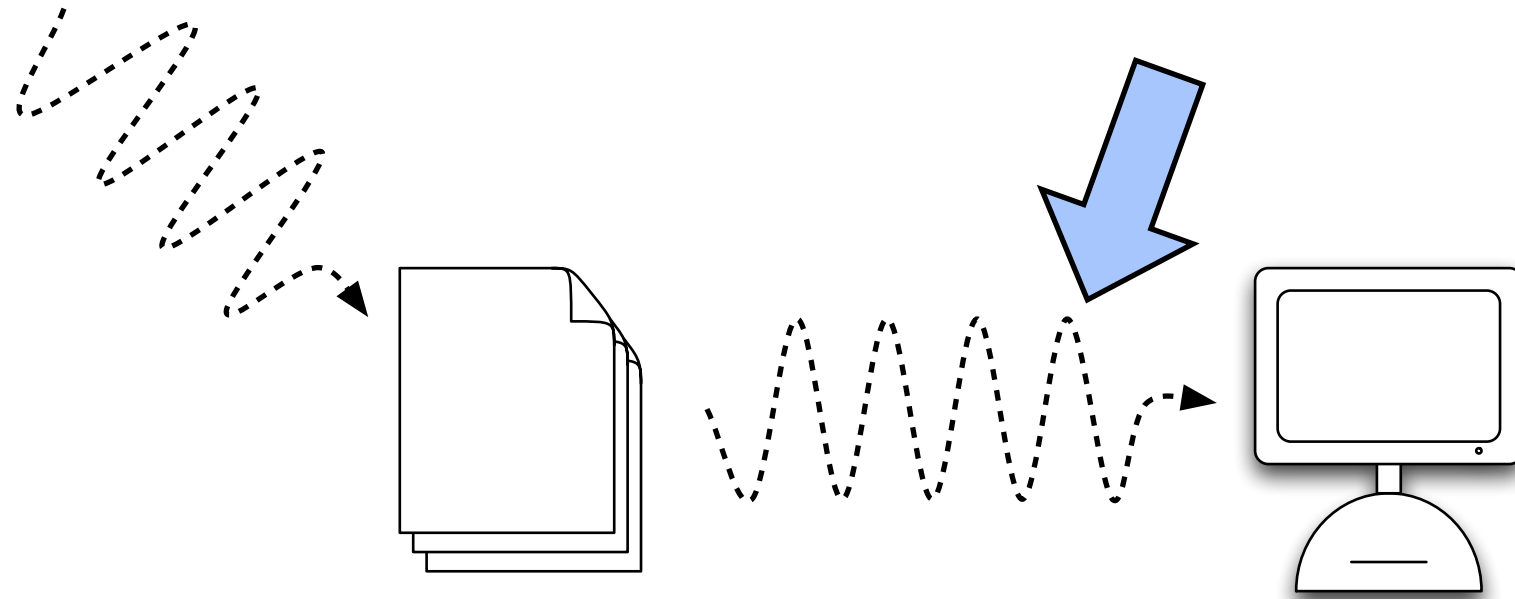
- [Somm05]: Chapter “Software Evolution”
- [Pres00], [Ghez02]: Chapters on Reengineering / Legacy Software
- [Fowl99] Refactoring, Improving the Design of Existing Code by Martin Fowler, Addison-Wesley, 1999.
 - + A practical book explaining when and how to use refactorings to cure typical code-smells.
- [Deme02] Object-Oriented Reengineering Patterns by Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, Morgan Kaufmann, 2002.
 - + A book describing how one can reengineer object-oriented legacy systems.



Web-Resources

- Following web-site lists a number of relevant code smells (= symptoms in code where refactoring is probably worthwhile)
<https://wiki.c2.com/?CodeSmell>

When Refactoring?



Any software system must be maintained

- The worst that can happen with a software system is that the people actually *use* it.
 - + >> Users will request changes ...
 - + >> Intangible nature of software
 - > ... makes it hard for users to understand the impact of changes

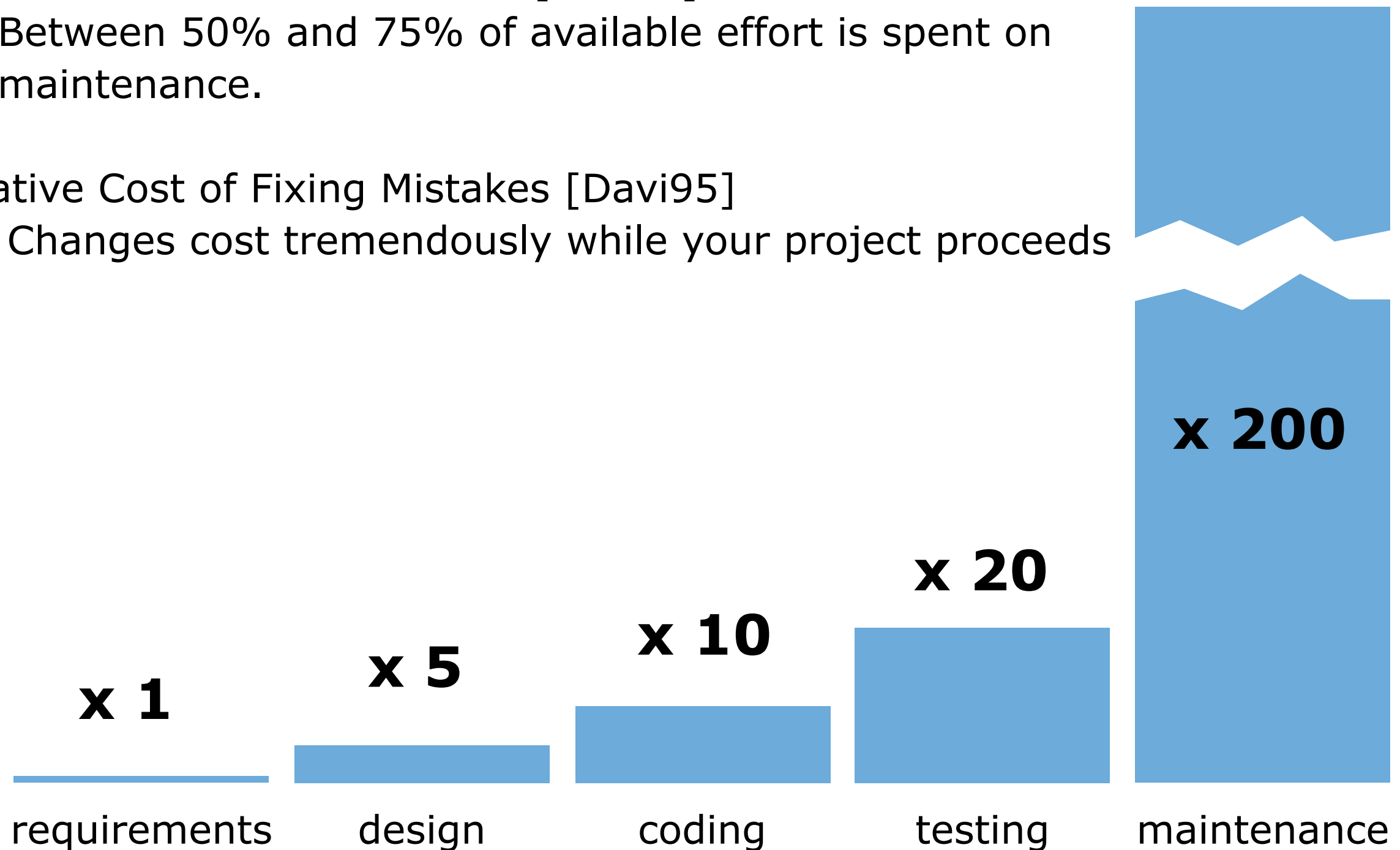
Why Refactoring? (1/2)

Relative Effort of Maintenance [Lien80]

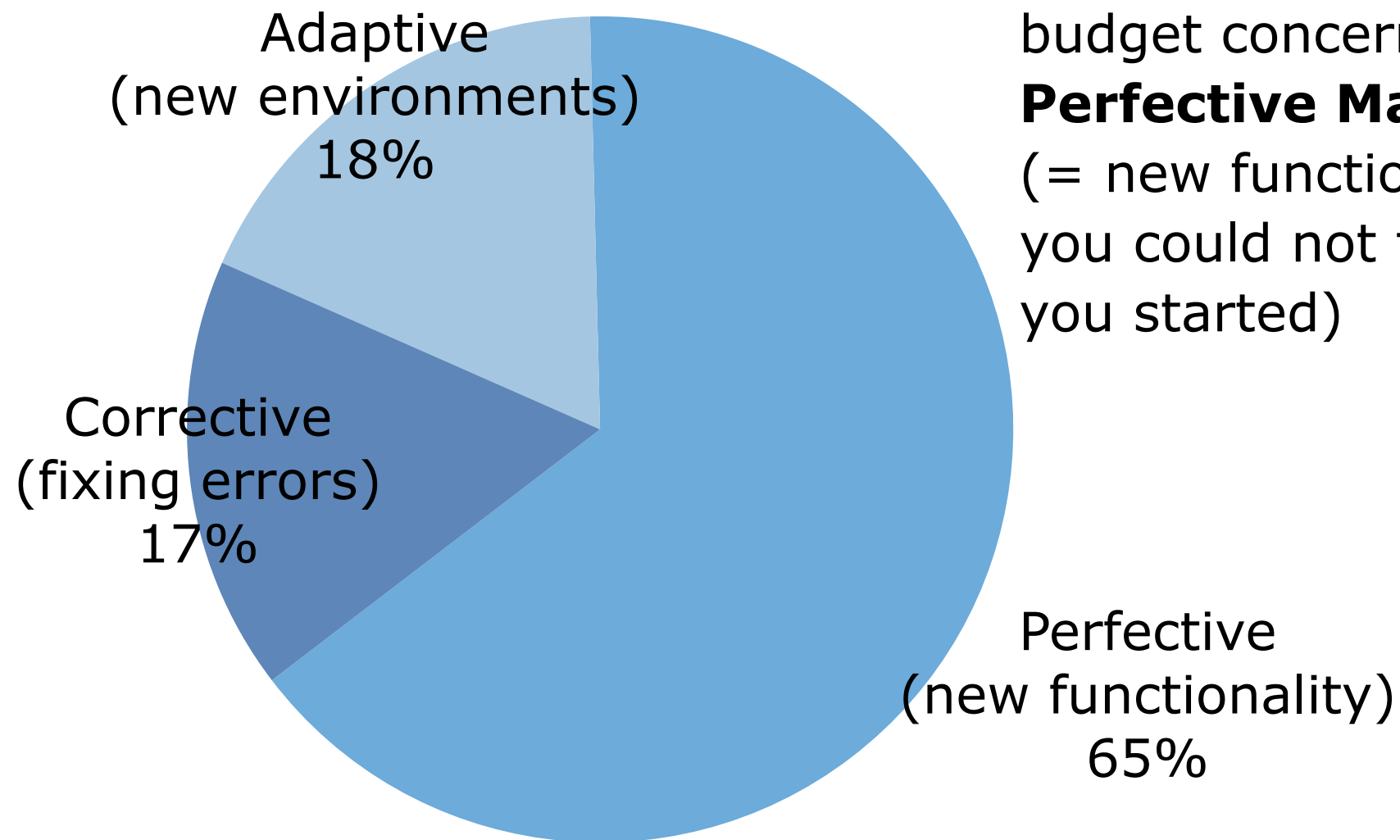
- Between 50% and 75% of available effort is spent on maintenance.

Relative Cost of Fixing Mistakes [Davi95]

⇒ Changes cost tremendously while your project proceeds



Why Refactoring? (2/2)



50-75% of maintenance budget concerns

Perfective Maintenance

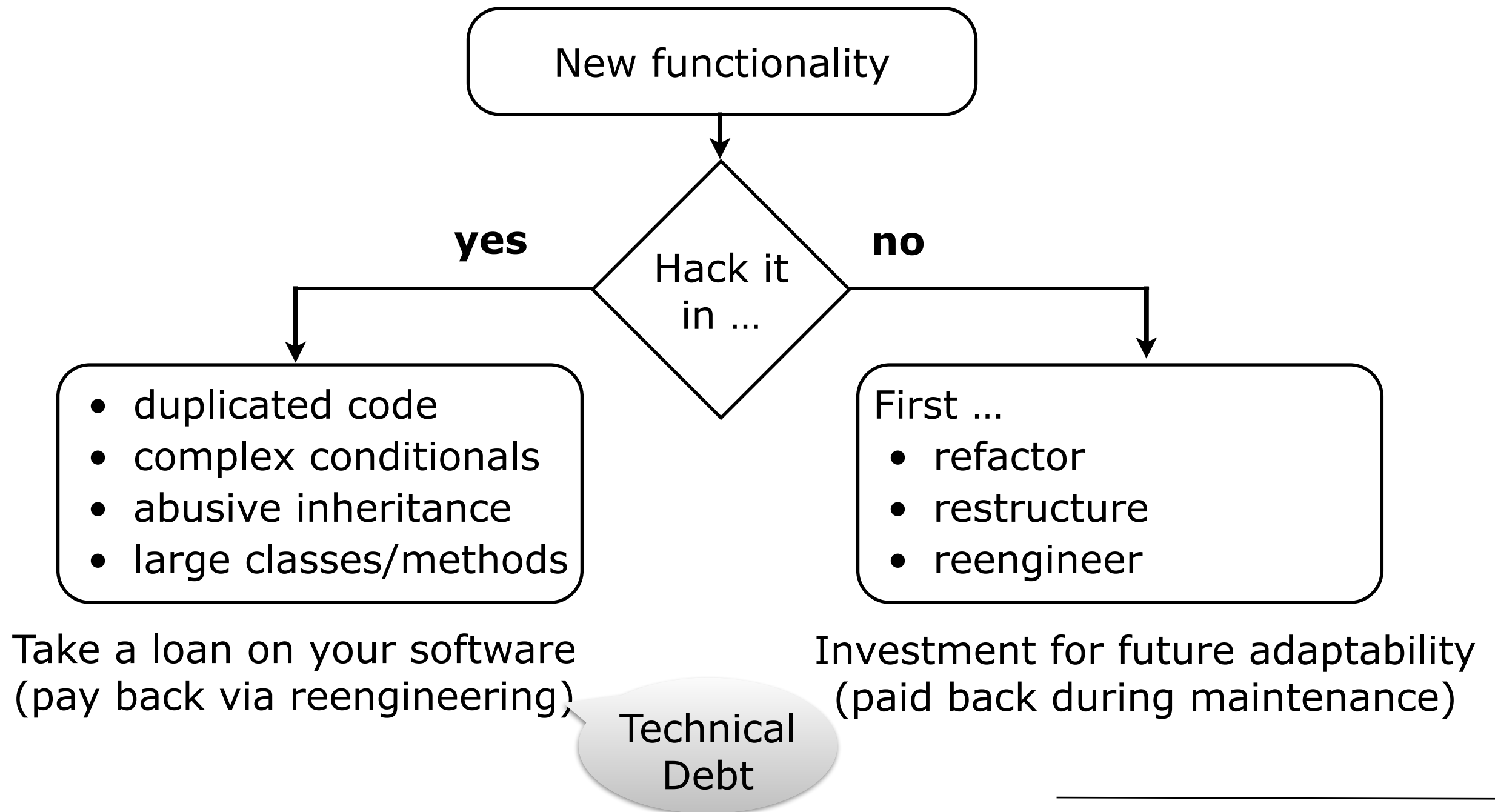
(= new functionality, which you could not foresee when you started)

⇒ New category of maintenance

Preventive Maintenance

Why Refactoring in OO?

New or changing requirements will gradually degrade original design, ...
... unless extra development effort is spent to adapt the structure.



What is Refactoring?

Two Definitions

- *VERB*: 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 [Fowl99]
- *NOUN*: A behaviour-preserving source-to-source program transformation [Robe98]
 - > *Primitive* refactorings vs. *Composite* refactorings

Typical Primitive 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
	pull up	pull up
	push down	push down
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Quizz

Can you explain why
+ add class
+ add method
+ add attribute
... are behaviour preserving?



Tool support

Change Efficient

Refactoring

- Source-to-source program transformation
- Behaviour preserving

⇒ improve the program structure

Programming Environment

- Fast edit-compile-run cycles
- Support small-scale reverse engineering activities

⇒ convenient for “local” ameliorations

Failure Proof

Regression Testing

- Repeating past tests
- Tests require no user interaction
- Tests are deterministic
(Answer per test is yes / no)

⇒ improvements do not break anything

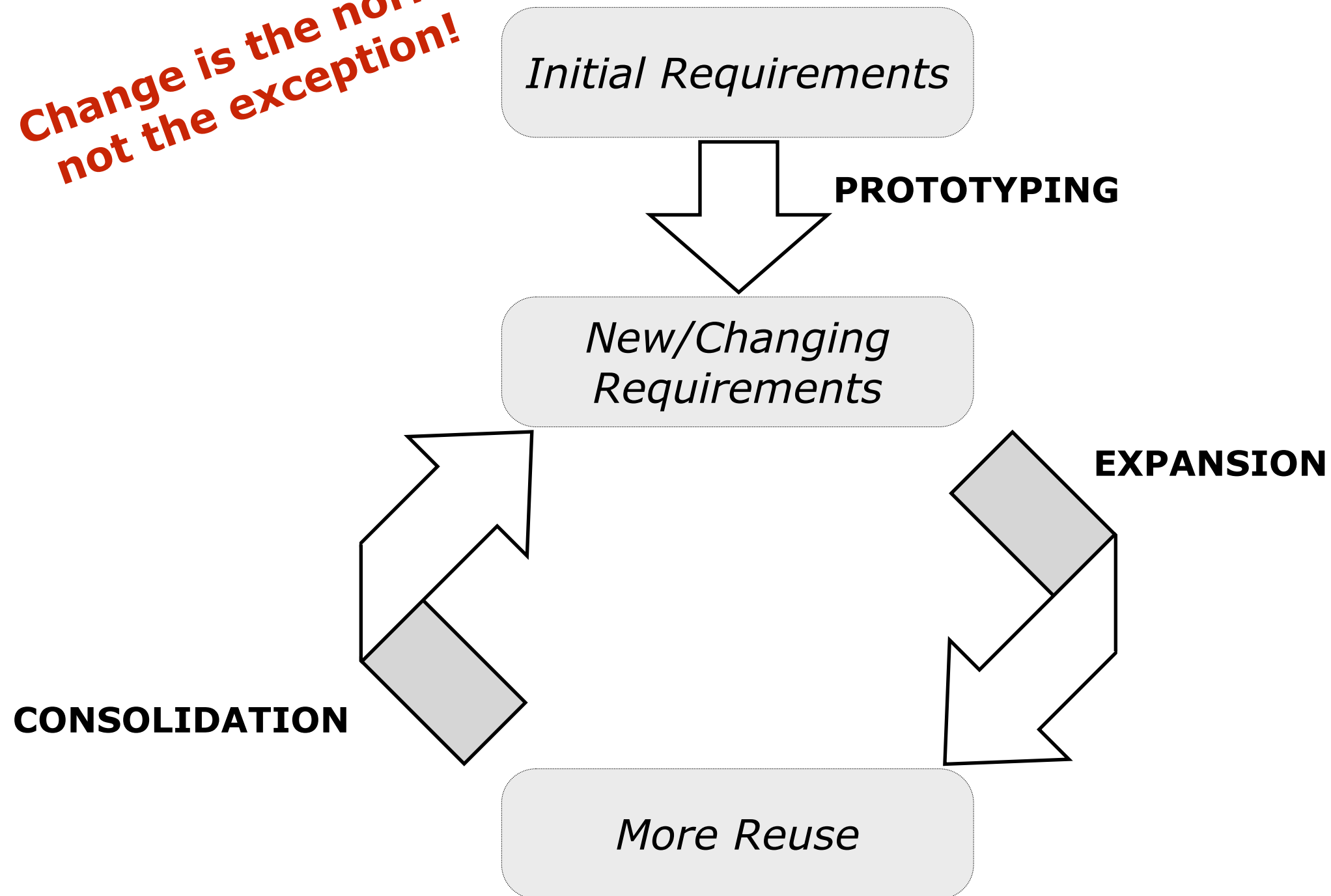
Configuration & Version Management

- keep track of versions that represent project milestones

⇒ go back to previous version

Iterative Development Life-cycle

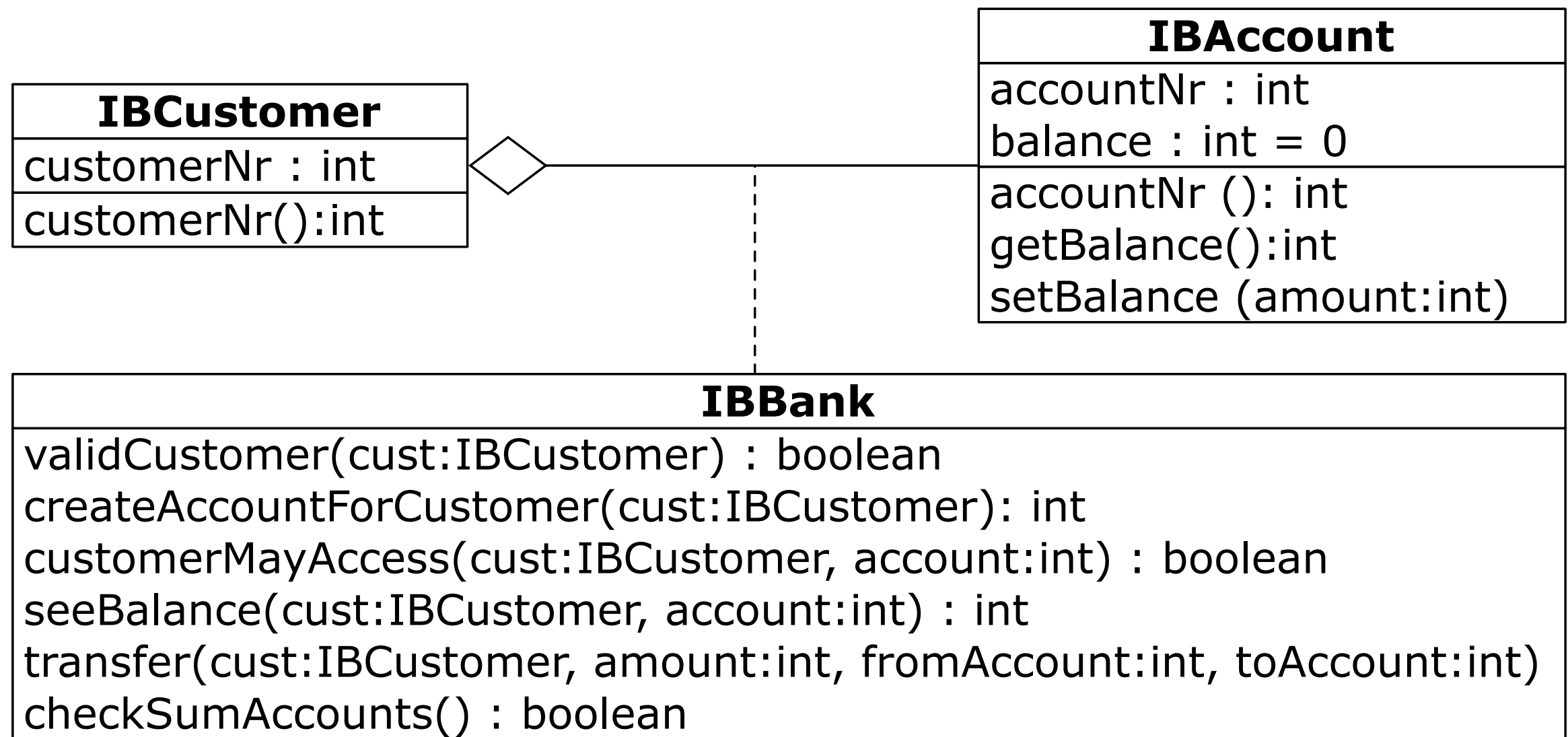
**Change is the norm,
not the exception!**



Example: Banking - Requirements

- + a bank has customers
- + customers own account(s) within a bank
- + with the accounts they own, customers may
 - deposit / withdraw money
 - transfer money
 - see the balance
- Non-functional requirements
 - + *secure*: only authorised users may access an account
 - + *reliable*: all transactions must maintain consistent state

Example: Banking - Class Diagram



Example: Banking - Contracts

IBank

invariant: checkSumAccounts()

Ensure the “secure” and “reliable” requirements.

IBank::createAccountForCustomer(cust:IBCustomer): int
precondition: validCustomer(cust)
postcondition: customerMayAccess(cust, <<result>>)

IBank::seeBalance(cust:IBCustomer, account:int) : int
precondition: (validCustomer(cust)) AND
 (customerMayAccess(cust, account))
postcondition: true

IBank::transfer(cust:IBCustomer, amount:int, fromAccount:int, toAccount:int)
precondition: (validCustomer(cust))
 AND (customerMayAccess(cust, fromAccount))
 AND (customerMayAccess(cust, toAccount))
postcondition: true

Example: Banking - CheckSum

- Bookkeeping systems always maintain two extra accounts, "incoming" and "outgoing"
- \Rightarrow the sum of the amounts of all transactions is always 0 \Rightarrow consistency check

Incoming

date	amount
1/1/2000	-100
1/2/2000	-200

MyAccount

date	amount
1/1/2000	+100
1/2/2000	+200
1/3/2000	-250

OutGoing

date	amount
1/3/2000	+250

Prototype Consolidation

Design Review (i.e., apply refactorings AND RUN THE TESTS!)

- Rename attribute
 - + rename `"_balance"` into `"_amountOfMoney"` (run test!)
 - + apply "rename attribute" refactoring to the above
 - > run test!
 - + check the effect on source code
 - comments + getter/setter methods
- Rename method
 - + rename `"get_balance"` into `"get_amountOfMoney"`
 - > run test!
- Change Method Signature
 - + change order of arguments for `"transfer"` (run test!)
- Rename class
 - + check all references to `"Customer"`
 - + apply "rename class" refactoring
 - > rename into `"Client"`
 - > run test!
 - + check the effect on source code
 - file name / makefiles / ...
 - `CustomerTest` >> `ClientTest`??

What is Refactoring?

Can you give the pre-conditions for a “rename method” refactoring?



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
	pull up	pull up
	push down	push down
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Expansion

Additional Requirement

- concurrent access of accounts

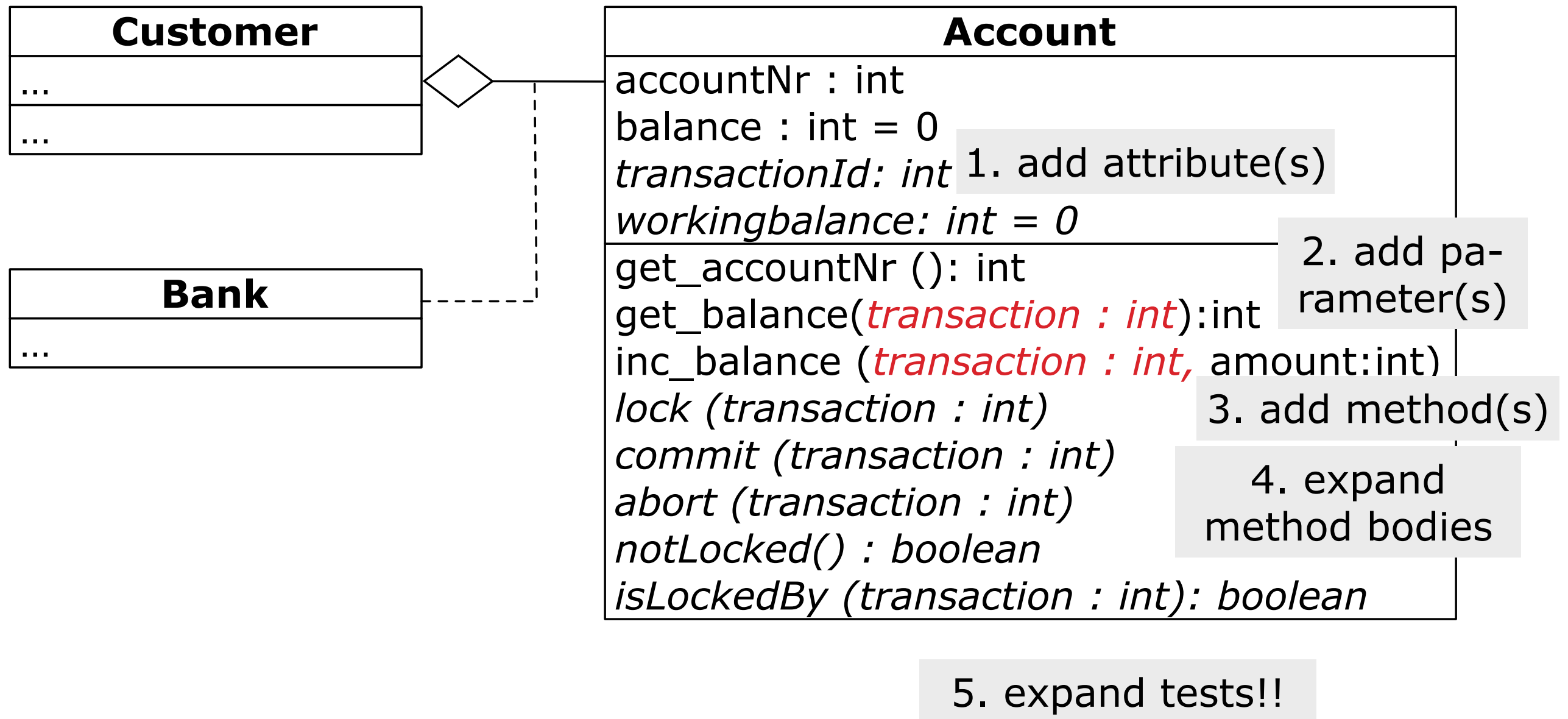
Add test case for

- Bank
 - + testConcurrent: Launches 10 processes that simultaneously transfer money between same accounts
 - > test fails!

**Can you explain why
the test fails?**



Expanded Design: Class Diagram



Expanded Design: Contracts

Account

invariant: (isLocked()) OR (NOT isLocked())

Account::get_balance(transaction:int): int

precondition: isLockedBy(transaction)

postcondition: true

Account::inc_balance(transaction:int, amount: int)

precondition: isLockedBy(transaction)

postcondition: peek_balance() = peek_balance() + amount

Account::lock(transaction:int)

precondition: notLocked()

postcondition: isLockedBy(transaction)

Account::commit(transaction:int)

precondition: isLockedBy(transaction)

postcondition: notLocked()

Account::abort(transaction:int)

precondition: isLockedBy(transaction)

postcondition: notLocked()

Expanded Implementation

Adapt implementation

- 1. Manually add attributes on Account
 - + "transactionId" and "workingBalance"
- 2. apply "change method signature"
 - + add "transaction"
 - + to "get_balance()" and "inc_balance()"
- 3. apply "add method"
 - + lock, commit, abort, isLocked, isLockedBy
- 4. expand method bodies (i.e. careful programming)
 - + of "seeBalance()" and "transfer()"
 - > load "Banking12"
- 5. expand Tests
 - + previous tests for "get_balance()" and "inc_balance()"
 - should now fail
 - * adapt tests
 - + new contracts, incl. commit and abort
 - * new tests

testConcurrent works!

> we can confidently ship a new release

Consolidation: Problem Detection

More Reuse

- A design review reveals that this “transaction” stuff is a good idea and is applied to Customer as well.

⇒ Code Smells

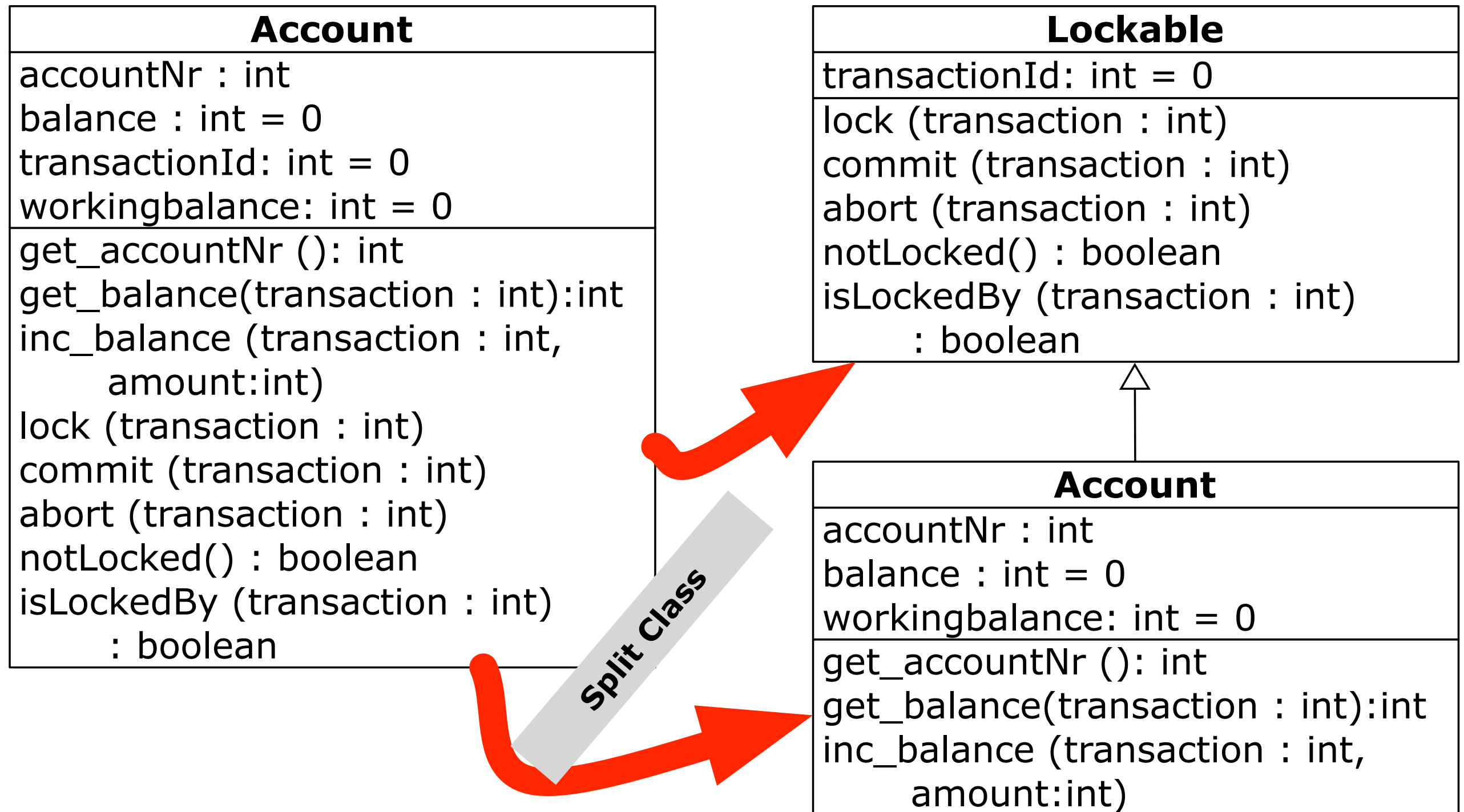
- duplicated code
 - + lock, commit, abort
 - + transactionId
- large classes
 - + extra methods
 - + extra attributes

⇒ Refactor

- “Lockable” should become a separate component, to be reused in Customer and Account

Customer
customerNr : int name: String address: String password: String transactionId: int workingName: String ...
get_customerNr():int getName(transaction : int):String setName (transaction : int, name:String) ... lock (transaction : int) commit (transaction : int) abort (transaction : int) isLocked() : boolean isLockedBy (transaction : int) : boolean

Consolidation: Refactored Class Diagram



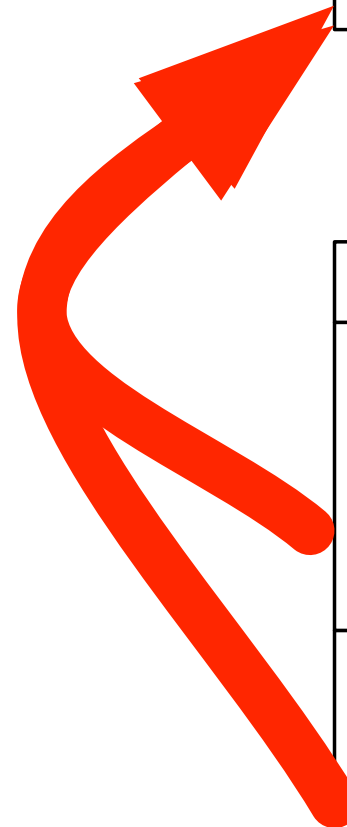
Refactoring Sequence: 1/4

Refactoring: Extract Superclass

- Position on Account
 - + superclass name = Lockable
 - + members: transactionId + notLocked + isLockedBy
 - action = extract
- verify effect on code
- run the tests!

Lockable

Account
accountNr : int balance : int = 0 transactionId: int = 0 workingbalance: int = 0
... notLocked() : boolean isLockedBy (transaction : int) : boolean ...

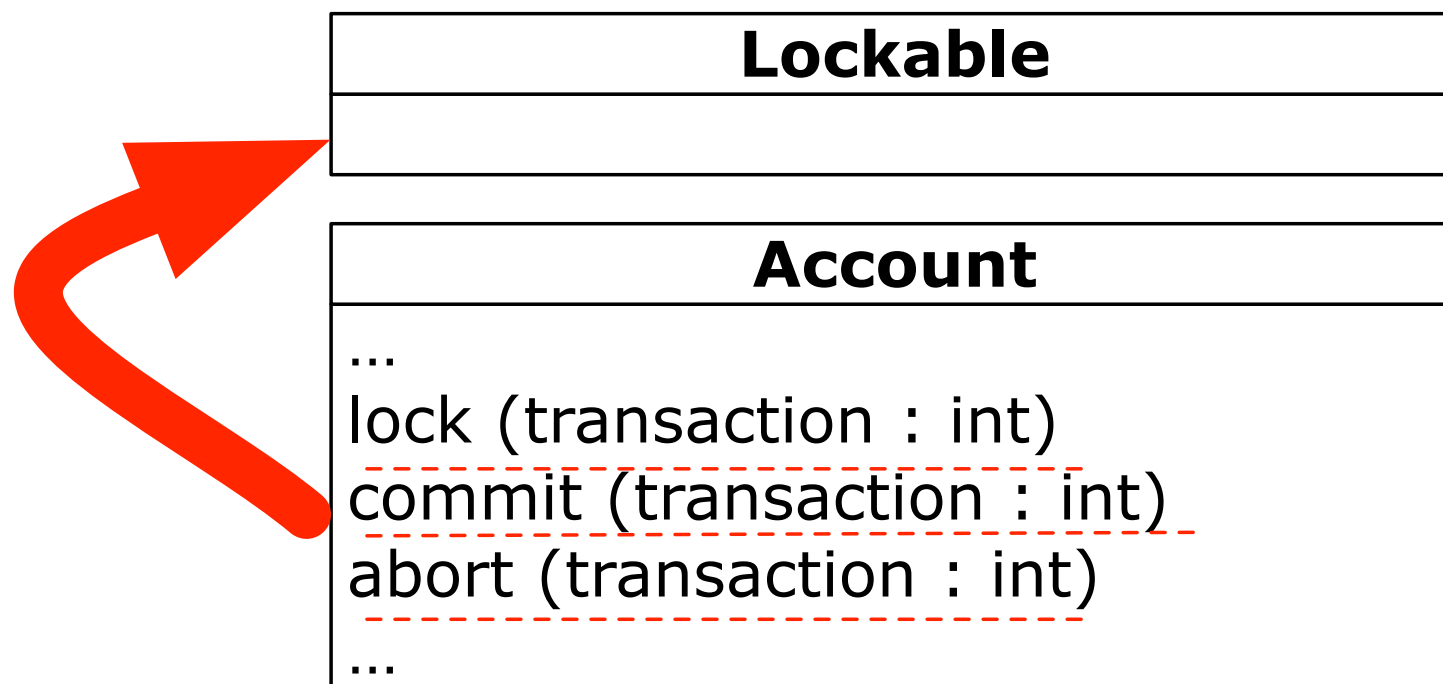


Refactoring Sequence: 2/4

Refactoring: Pull Up

- apply “pull up ...” on “Account”
 - + to move “lock / commit / transaction” onto lockable
 - + apply “pull up” to “abort:”, “commit:”, “lock:”

> failure: why???

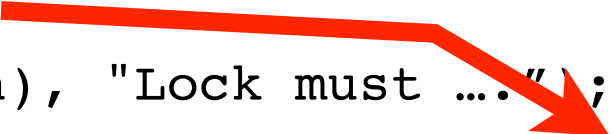


Refactoring Sequence: 3/4

Refactoring: Extract Method

- apply “extract method” on
+ groups of accesses to “balance” and “WorkingBalance”

```
public synchronized void lock(int transaction) {  
    this.require(this.notLocked(), "No other transaction ...");  
    this.transactionId = transaction;  
    this._workingBalance = this._balance;  
    this.ensure(this.isLockedBy(transaction), "Lock must ...");  
}
```



```
protected void copyToWorkingState() {  
    this._workingBalance = this._balance;  
}
```

```
public synchronized void lock(int transaction) {  
    this.require(this.notLocked(), "No other transaction ...");  
    this.transactionId = transaction;  
    copyToWorkingState();  
    this.ensure(this.isLockedBy(transaction), "Lock must ...");  
}
```

- similar for
+ “abort” (⇒ clearWorkingState) & “commit” (⇒ commitWorkingState)

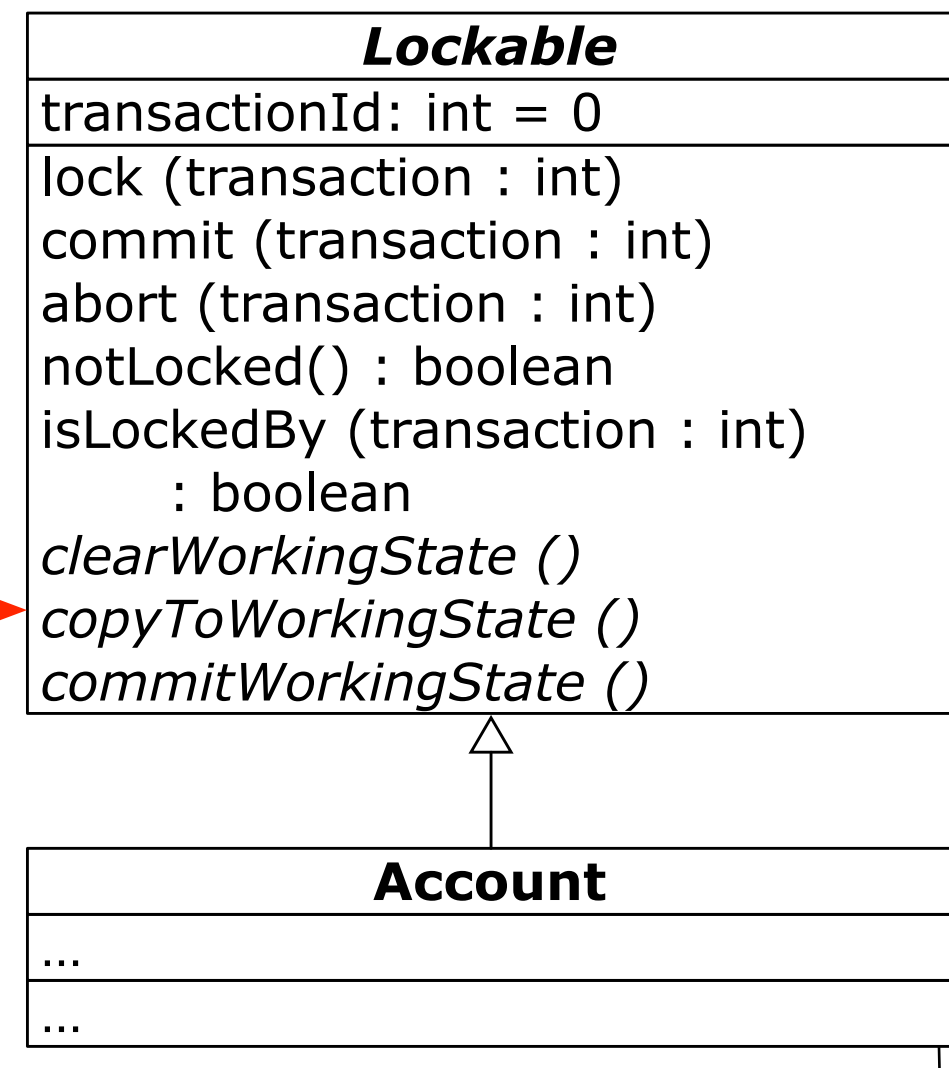
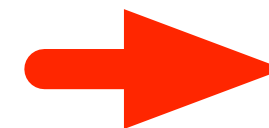
Refactoring Sequence: 4/4

Refactoring “Pull up...” revisited

- apply “pull up ...” on “Account”
 - + members `clearWorkingState` / `copyToWorkingState` / `commitWorkingState`
 - action = declare abstract in destination
 - + members “abort”, “commit”, “lock”
 - action = pull up

Are we done?

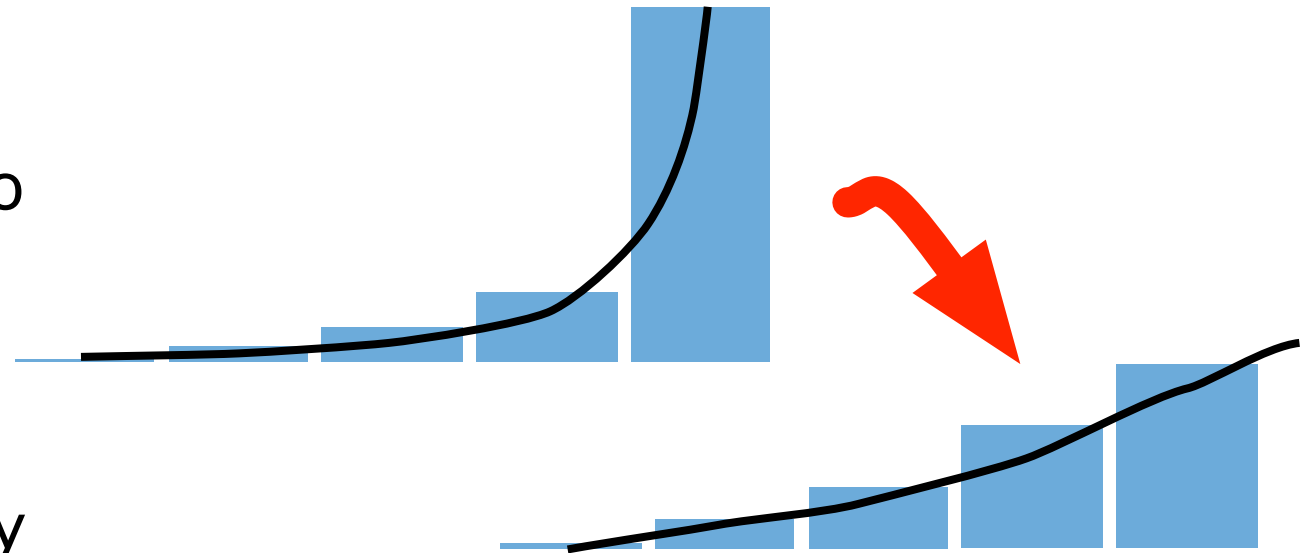
- Run the tests ...
- Customer subclass of `Lockable`
 - + expand functionality to incorporate locking protocol



Tool Support

Refactoring Philosophy

- combine simple refactorings into larger restructuring (and eventually reengineering)
 - > improved design
 - > ready to add functionality
- Do not apply refactoring tools in isolation



	Smalltalk	C++	Java
refactoring	+	- (?)	+
rapid edit-compile-run cycles	+	-	+-
reverse engineering facilities	+-	+-	+-
regression testing	+	+	+
version & configuration management	+	+	+

Code Smells

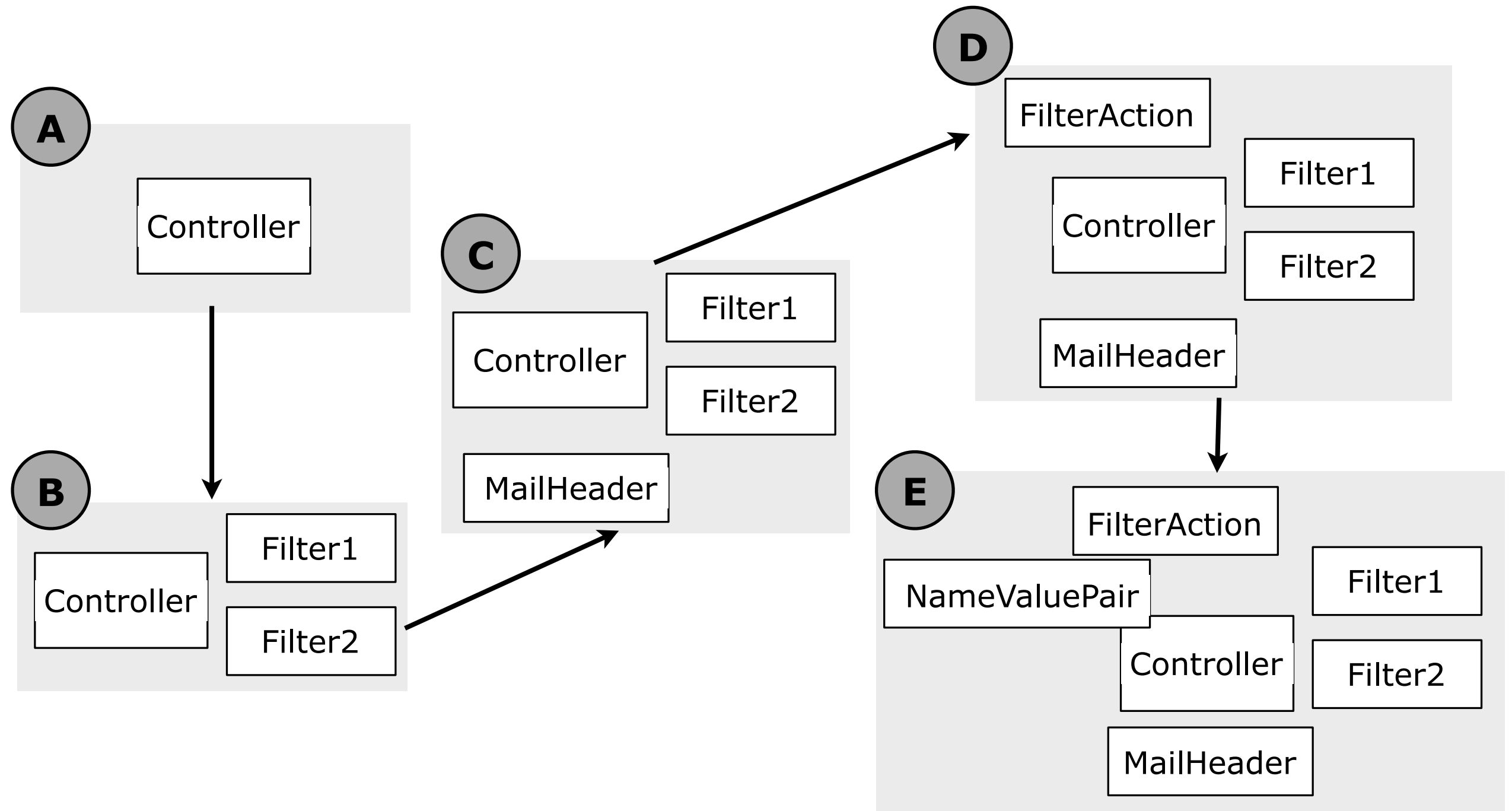
Know when is as important as know-how

- Refactored designs are more complex
 - > Introduce a lot of extra small classes/methods
 - Use “code smells” as symptoms for refactoring opportunities
 - + Duplicated code
 - + Nested conditionals
 - + Large classes/methods
 - + Abusive inheritance
- Rule of the thumb:
 - + All system logic must be stated *Once and Only Once*
 - > a piece of logic stated more than once implies refactoring

More about code smells and refactoring

- Wiki-web with discussion on code smells
 - + <https://wiki.c2.com/?CodeSmell>

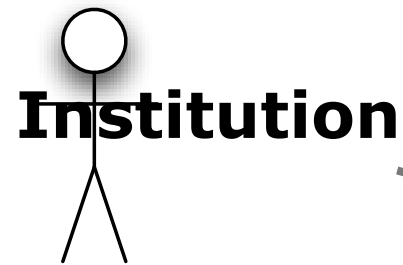
Refactoring God Class: Optimal Decomposition?



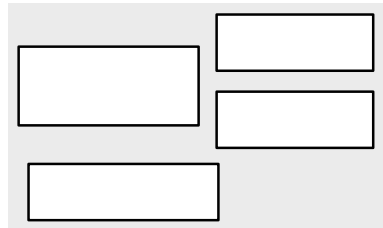
Empirical Validation

Controlled experiment with 63 last-year master-level students (CS and ICT)

Independent Variables



Decomposition



Experimental Task

Dependent Variables

Time

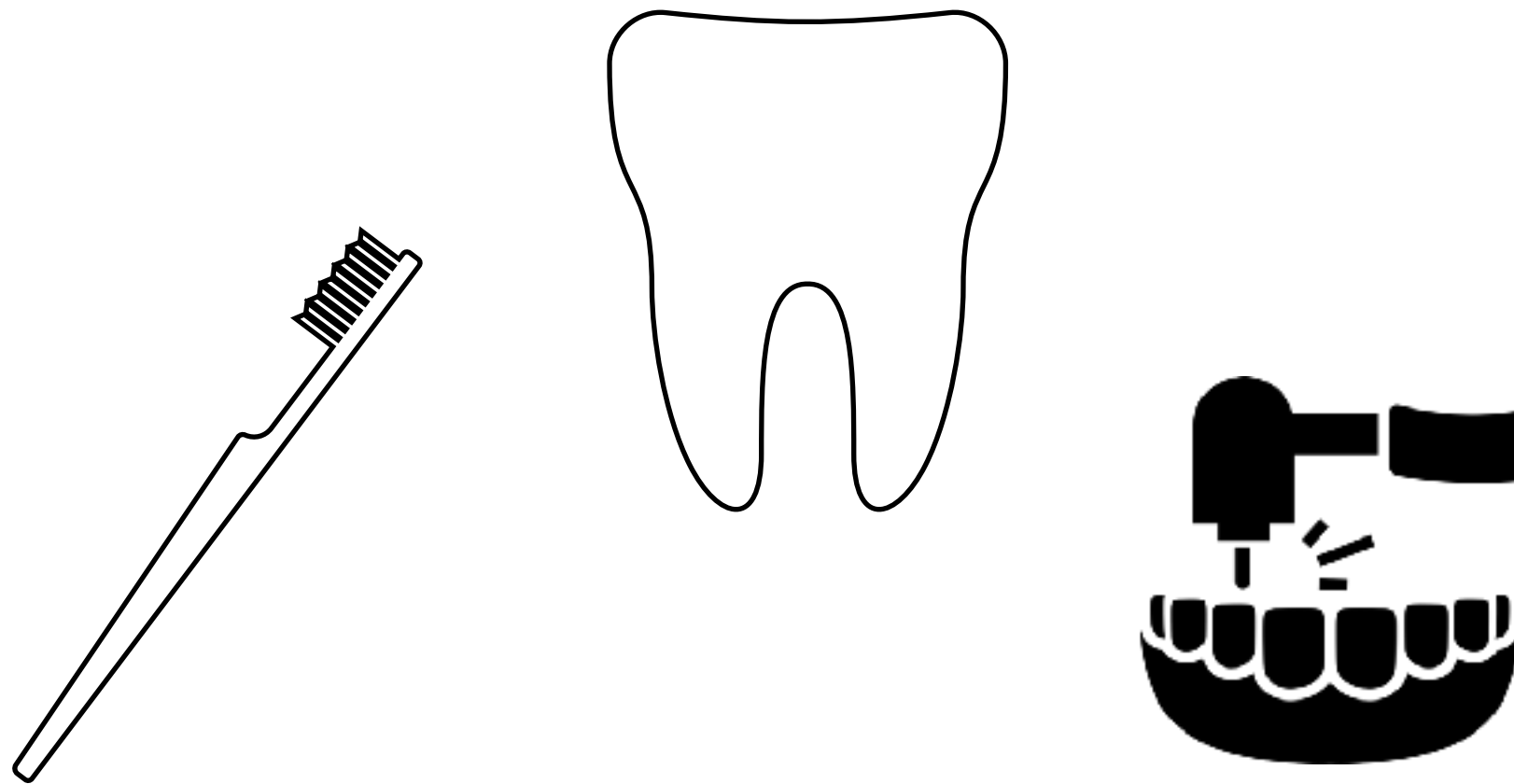


Accuracy



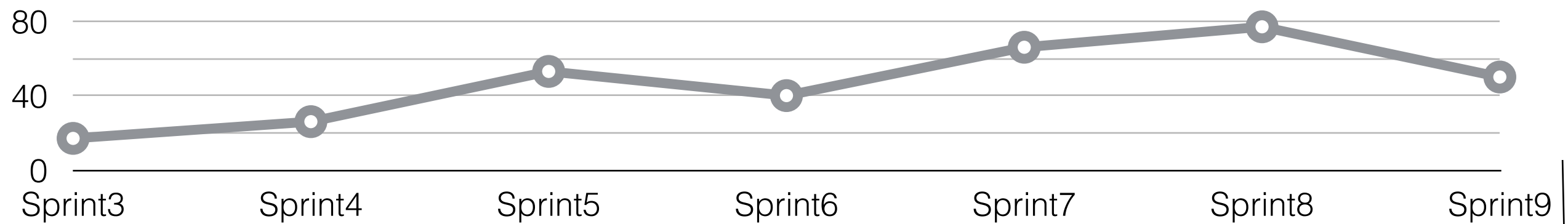
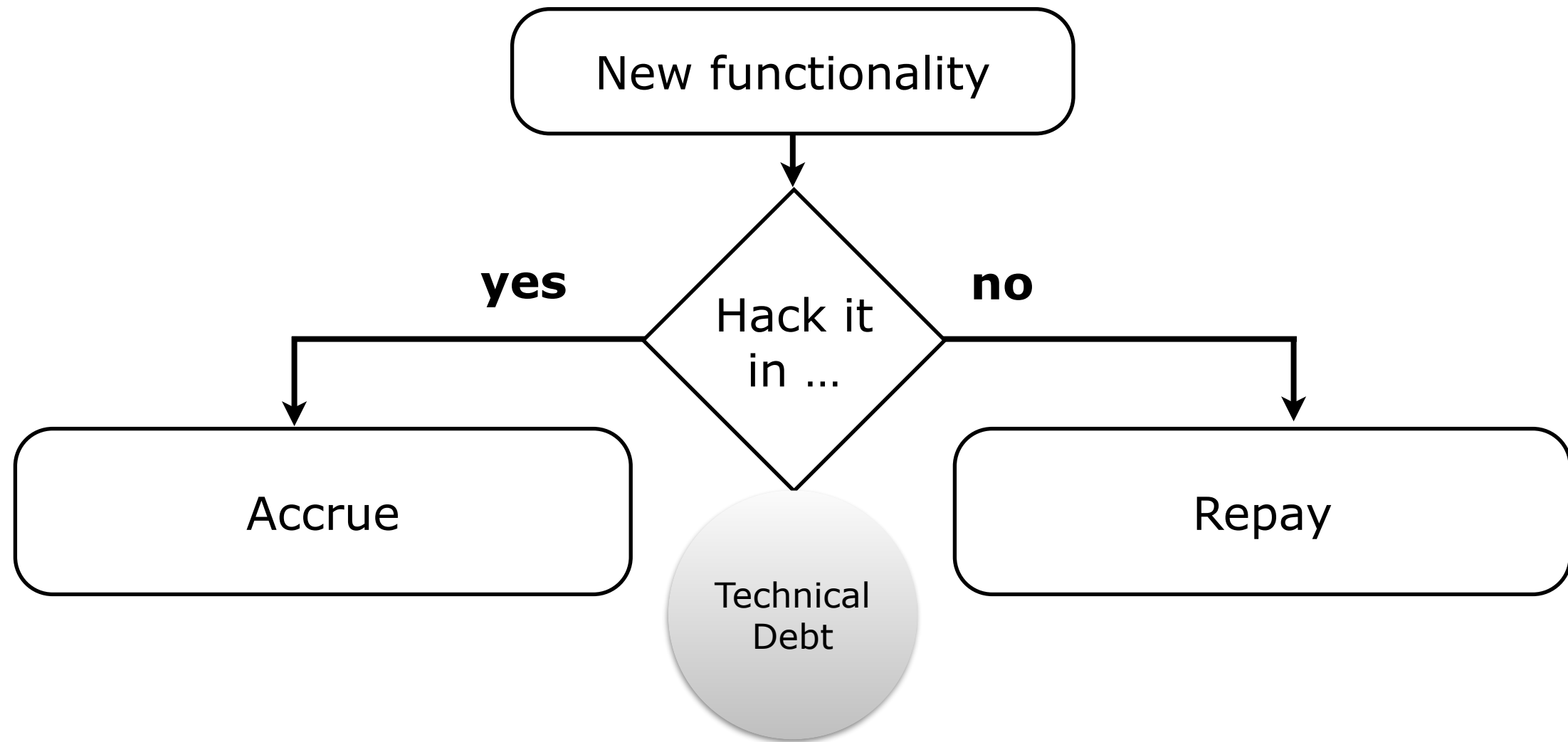
- “Optimal decomposition” differs with respect to education
- Computer science: preference towards decentralized designs (C-E)
 - ICT-electronics: preference towards centralized designs (A-C)
- Advanced OO training can induce preference
- Consistent with [Arisholm et al. 2004]

Floss Refactoring vs. Root-Canal Refactoring



E. Murphy-Hill and A. P. Black, "Refactoring Tools: Fitness for Purpose," in IEEE Software, vol. 25, no. 5, pp. 38-44, Sept.-Oct. 2008, doi: 10.1109/MS.2008.123.

Technical Debt



DevOps: Monitor Technical Debt



I WANT YOU
CAPSTONE PROJECT



Quality Gate

Passed

1 **B**
Bugs

0 **A**
Vulnerabilities

43 **A**
Code Smells

86.0%
Coverage

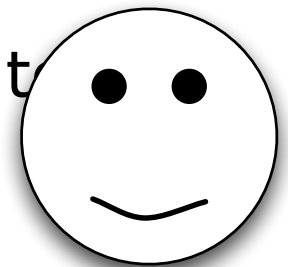
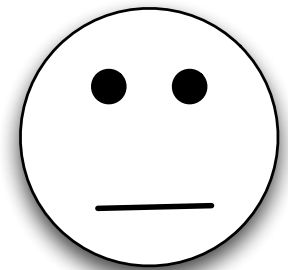
0.5%
Duplications

sonarqube

Correctness & Traceability

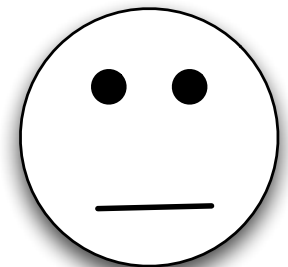
Correctness

- Are we building the system right?
- Assured via “behaviour preserving” nature & regression testing
 - > We are sure the system remains as “correct” as it was before
- Are we building the right system?
 - + By improving the internal design we can cope with mismatches
 - > First refactor (= consolidate) ...
 - > then new requirements (= expand)



Traceability

- Requirements \leftrightarrow System?
 - + Requires a lot of discipline ... thus extra effort!
 - + But renaming is refactoring too
 - > Adjust code to adhere to naming conventions



Summary (i)

You should know the answers to these questions:

- Can you explain how refactoring differs from plain coding?
- Can you tell the difference between Corrective, Adaptive and Perfective maintenance? And how about preventive maintenance?
- Can you name the three phases of the iterative development life-cycle? Which of the three does refactoring support the best? Why do you say so?
- Can you give 4 symptoms for code that can be “cured” via refactoring?
- Can you explain why add class/add method/add attribute are behaviour preserving?
- Can you give the pre-conditions for a “rename method” refactoring?
- Which 4 activities should be supported by tools when refactoring?
- Why can’t we apply a “push up” to a method “x()” which accesses an attribute in the class the method is defined upon (see Refactoring Sequence on page 27–31)?

You should be able to complete the following tasks

- Two classes A & B have a common parent class X. Class A defines a method a() and class B a method b() and there is a large portion of duplicated code between the two methods. Give a sequence of refactorings that moves the duplicated code in a separate method x() defined on the common superclass X.
- What would you do in the above situation if the duplicated code in the methods a() and b() are the same except for the name and type of a third object which they delegate responsibilities too?
- Monitor the technical debt of you bachelor capstone project.



I WANT YOU
CAPSTONE PROJECT

Summary (ii)

Can you answer the following questions?

- Why would you use refactoring in combination with Design by Contract and Regression Testing?
- Can you give an example of a sequence of refactorings that would improve a piece of code with deeply nested conditionals?
- How would you refactor a large method? And a large class?
- Consider an inheritance relationship between a superclass "Square" and a subclass "Rectangle". How would you refactor these classes to end up with a true "is-a" relationship? Can you generalise this procedure to any abusive inheritance relationship?