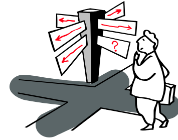
CHAPTER 11 – Refactoring



Introduction
- When, Why, What?
- Which Refactoring Tools?

Demonstration: Internet Banking
- Iterative Development Life-cycle
- Prototype
- Consolidation: design review
- Expansion: concurrent access
- Consolidation: more reuse

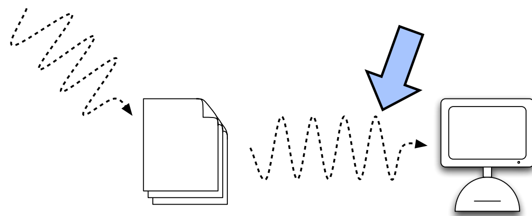
Conclusion
- Tool Support
- Code Smells
- Refactoring God Class
  + An empirical study
- Correctness & Traceability

Literature

- [Somm04a]: Chapter "Software Evolution"
- [Pres01a], [Ghez02a]: Chapters on Reengineering / Legacy Software
- [Fowl99a] 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 some typical code-smells.
- [Deme02a] 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 refectoring is probably worthwhile) http://c2.com/cgi/wiki?CodeSmells
  + Wiki-web with discussion on code smells

When Refactoring?

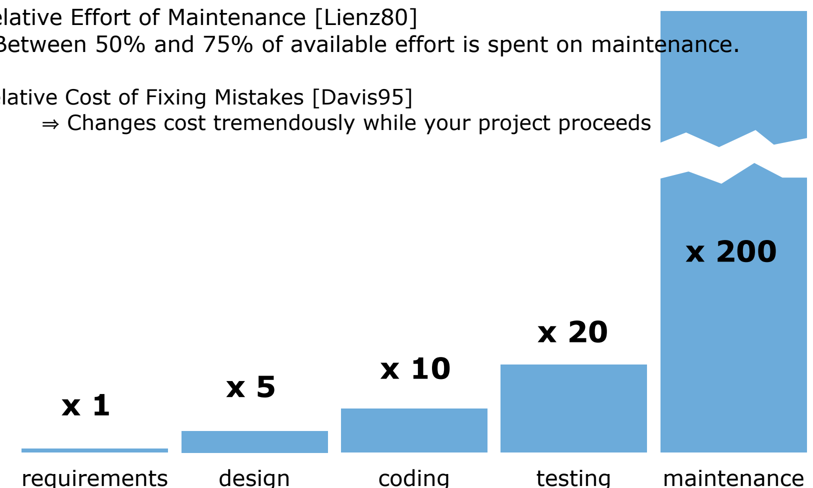


Any software system must be maintained
- The worst that can happen with a software system is that the people actually *use* it.
  >> Users will request changes ...
  >> Intangible nature of software
  … makes it hard for users to understand the impact of changes

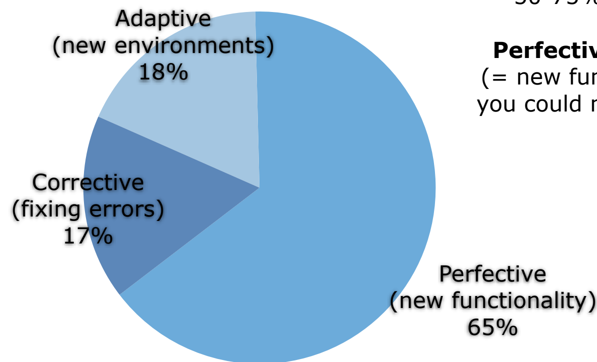
Why Refactoring ? (1/2)

Relative Effort of Maintenance [Lienz80]
- Between 50% and 75% of available effort is spent on maintenance.

Relative Cost of Fixing Mistakes [Davis95]
⇒ Changes cost tremendously while your project proceeds



Why Refactoring ? (2/2)



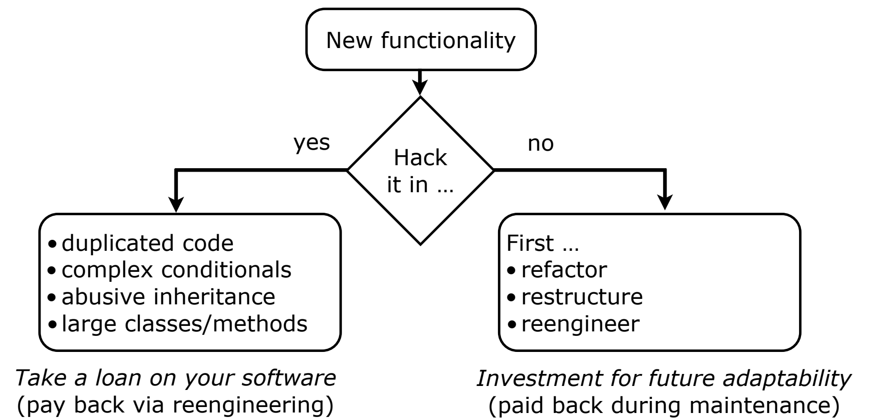
50-75% of maintenance budget concerns **Perfective Maintenance** (= new functionality, which you could not foresee when you started)

⇒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.



Take a loan on your software
(pay back via reengineering)

Investment for future adaptability
(paid back during maintenance)

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 [Fowler1999]
- *NOUN*: A behaviour-preserving source-to-source program transformation [Robertson1998]

➡ *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
	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	

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 deterministicAnswer per test is yes / no

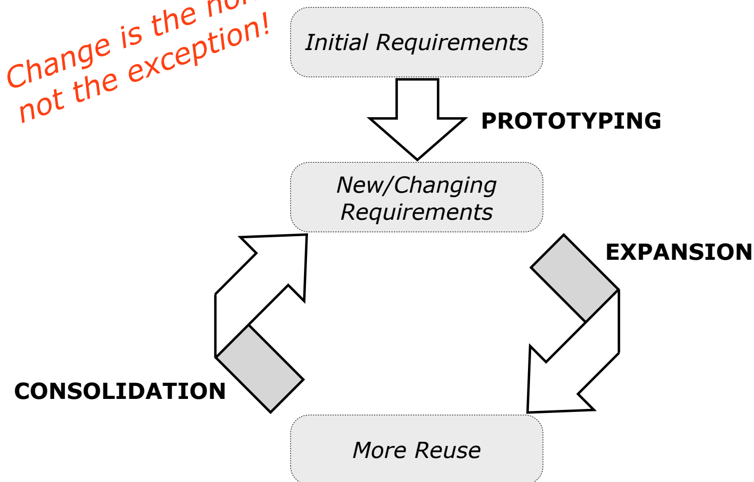
⇒ *improvements do not break anything*

Configuration & Version Management

- keep track of versions that represent project milestones

⇒ *go back to previous version*

Iterative Development Life-cycle



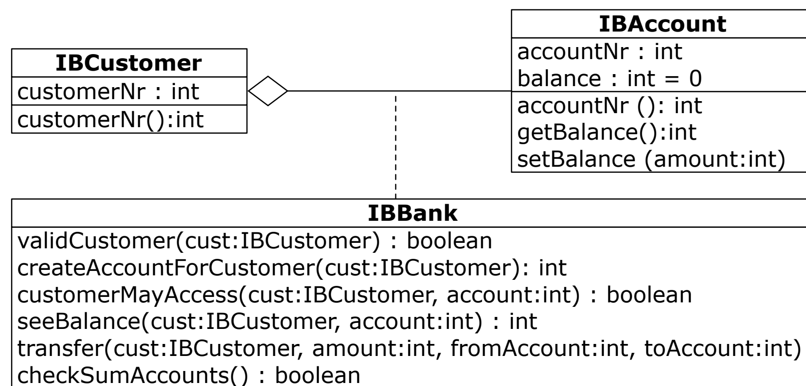
Example: Banking - Requirements

- 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

Ensure the "secure" and "reliable" requirements.

```
IBBank
   invariant: checkSumAccounts()


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


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


IBBank::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"
⇒ the sum of the amounts of all transactions is always 0
⇒ consistency check

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

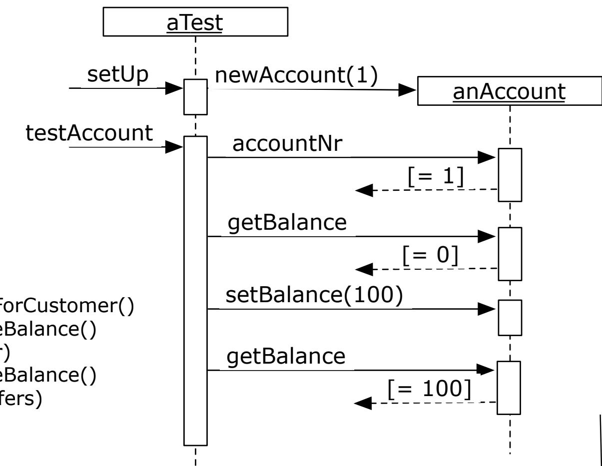
MyAccount	
date	amount
1/1/2000	+100
1/2/2000	+200
1/3/2000	-250

OutGoing	
date	amount
1/3/2000	+250

Initial Prototype — Unit Tests

➡ see demo "IBanking1"

Include test cases for
- IBCustomer
  + customerNr()
- IBAccount
  + getBalance()
  + setBalance()
- IBBank
  + createAccountForCustomer()
  + transfer() / seeBalance() (single transfer)
  + transfer() / seeBalance() (multiple transfers)



Prototype Consolidation

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

- Rename attribute
  + manually rename "balance" into "amountOfMoney" (run test!)
  + apply "rename attribute" refactoring to reverse the above
    ➡ run test!
  + check the effect on source code
- Rename class
  + check all references to "IBCustomer"
  + apply "rename class" refactoring to rename into IBClient
    ➡ run test!
  + check the effect on source code
- Rename method
  + rename "init()" into "initialize()"
    ➡ run test!
    - see what happens if we rename "initialize()" into "init"
  + change order of arguments for "transfer" (run test!)

pre-conditions for renaming ?

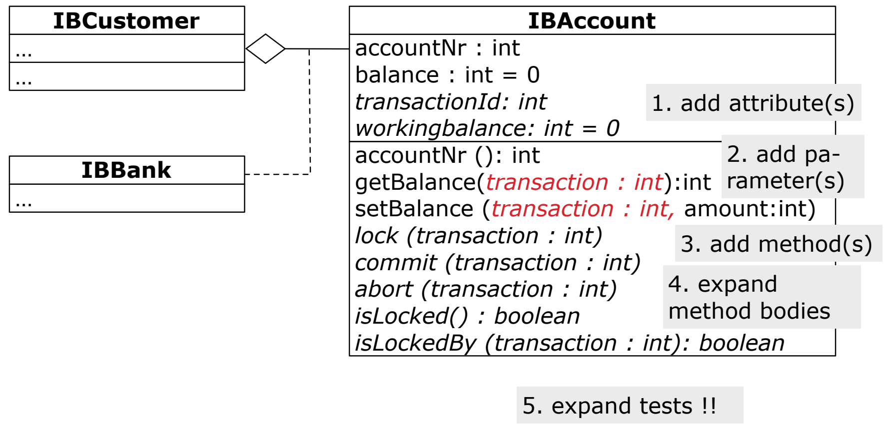
Expansion

Additional Requirement
- concurrent access of accounts

Add test case for
- IBBank
  + testConcurrent: Launches 2 processes that simultaneously transfer money between same accounts
    ➡ test fails!

Can you explain why the test fails?

Expanded Design: Class Diagram



5. expand tests !!

Expanded Design: Contracts

```
IBAccount
   invariant: (isLocked()) OR (NOT isLocked())

IBAccount::getBalance(transaction:int): int
   precondition: isLockedBy(transaction)
   postcondition: true

IBAccount::setBalance(transaction:int, amount: int)
   precondition: isLockedBy(transaction)
   postcondition: getBalance(transaction) = amount

IBAccount::lock(transaction:int)
   precondition: true
   postcondition: isLockedBy(transaction)

IBAccount::commit(transaction:int)
   precondition: isLockedBy(transaction)
   postcondition: NOT isLocked()

IBAccount::abort(transaction:int)
   precondition: isLockedBy(transaction)
   postcondition: NOT isLocked()
```


Expanded Implementation

Adapt implementation

- 1. apply “add attribute” on IBAccount
  - + “transactionId” and “workingBalance”
- 2. apply “add parameter”
  - + “transaction”
  - + to “getBalance()” and “setBalance()”
- 3. apply “add method”
  - + lock, commit, abort, isLocked, isLockedBy
- 4. expand method bodies (i.e. careful programming)
  - + of “seeBalance()” and “transfer()”
    - ➡ load “IBanking2”
- 5. expand Tests
  - + previous tests for “getBalance()” and “setBalance()”
    - - 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

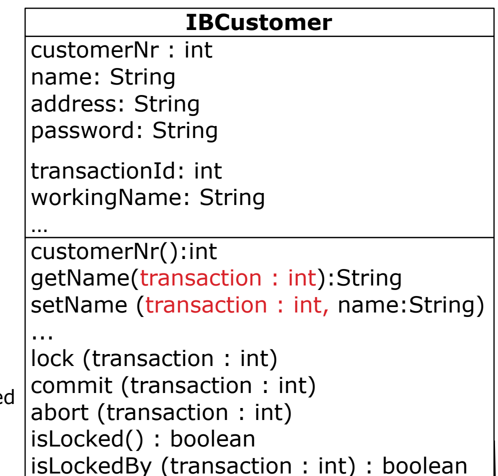
- A design review reveals that this “transaction” stuff is a good idea and is applied to IBCustomer as well.

⇒ Code Smells

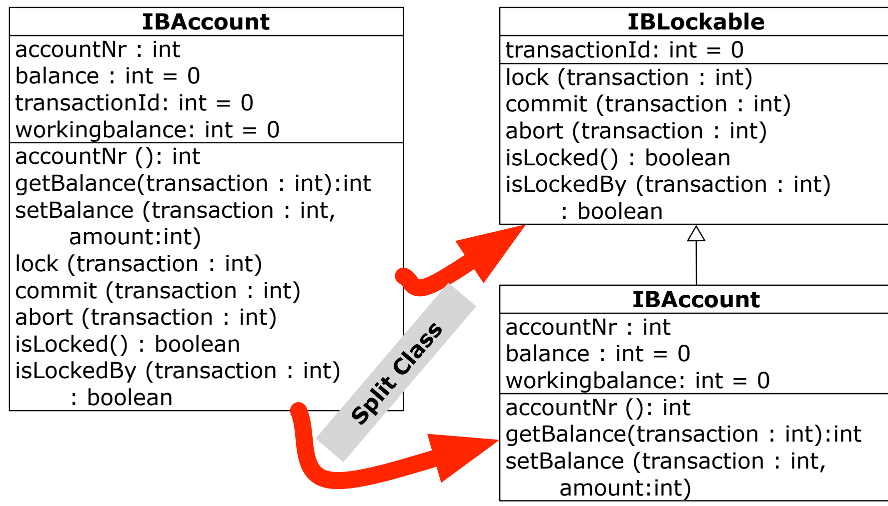
- duplicated code
  - + lock, commit, abort
  - + transactionId
- large classes
  - + extra methods
  - + extra attributes)

⇒ Refactor

- “Lockable” should become a separate component, to be reused in IBCustomer and IBAccount



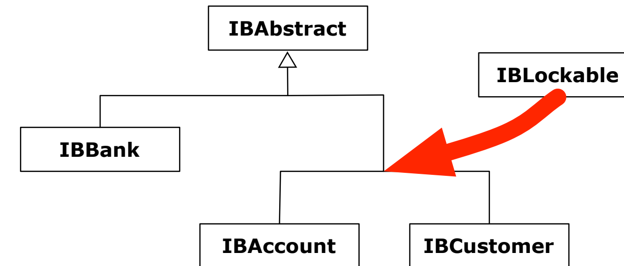
Consolidation: Refactored Class Diagram



Refactoring Sequence: 1/5

Refactoring step: Create subclass

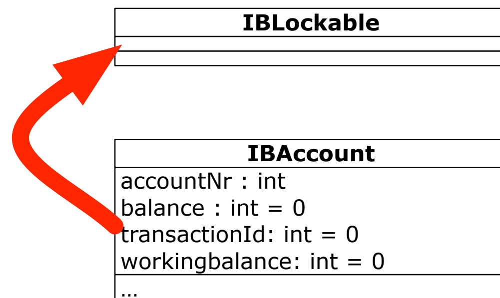
- apply "Create Subclass" on "IBAbstract"
  - + to create an empty "IBLockable"
  - + with subclass(es) "IBAccount" & "IBCustomer"



Refactoring Sequence: 2/5

Refactoring: Move Attribute

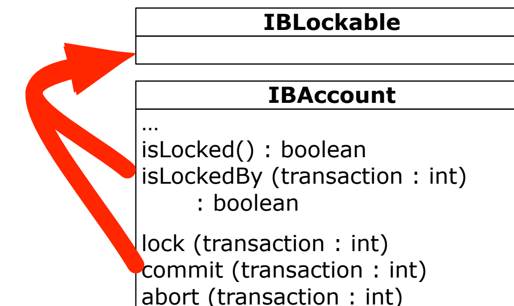
- apply "pull up attribute" on "IBLockable"
  - + to move "transactionId" up



Refactoring Sequence: 3/5

Refactoring: Move Method

- apply "push up method" on "IBAccount"
  - + to move "isLocked", "isLockedBy" up



+ apply "push up" to "abort:", "commit:", "lock:"
  - ➡ failure: why ???

Refactoring Sequence: 4/5

Refactoring: Extract Method

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

```
commit: transactionID
 self require: [self isLockedBy: transactionID]
     usingException: #lockFailureSignal.
 balance := workingBalance.
 workingBalance := nil.
 transactionIdentifier := nil.
 self ensure: [self notLocked].
```

```
commitWorkingState
    balance := workingBalance
    workingBalance := nil.
```

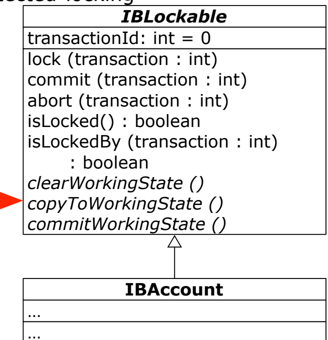
```
commit: transactionID
 self require: [self isLockedBy: transactionID]
         usingException: #lockFailureSignal.
 self commitWorkingState.
 transactionIdentifier := nil.
 self ensure: [self notLocked]
```

- similar for
  - + "abort:" (-> clearWorkingState)
  - + "lock:" (-> copyToWorkingState)

Refactoring Sequence: 5/5

Clean-up

- apply "push up method" on "IBAccount"
  - + to move "abort:", "commit:", "lock:" up
- make the extracted methods (e.g. commitWorkingState) protected
  - + … and define them as new abstract methods in the IBLocking class
    - - apply "rename protocol" on "IBAccount"
    - - to rename "public-locking" into "protected-locking"



Refactoring: Copy Method

- Apply "move method" on "IBAccount"
  - + to copy "clearWorkingState", "copyToWorkingState",
  - + "commitWorkingState"
  - + to "IBLockable>protected- locking"
- Make "IBLockable::clearWorkingState", … abstract
  - + Destructive editing; not refactoring

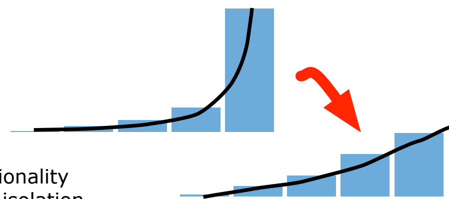
Are we done?

- Run the tests …
- Expand functionality of the IBCustomer

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

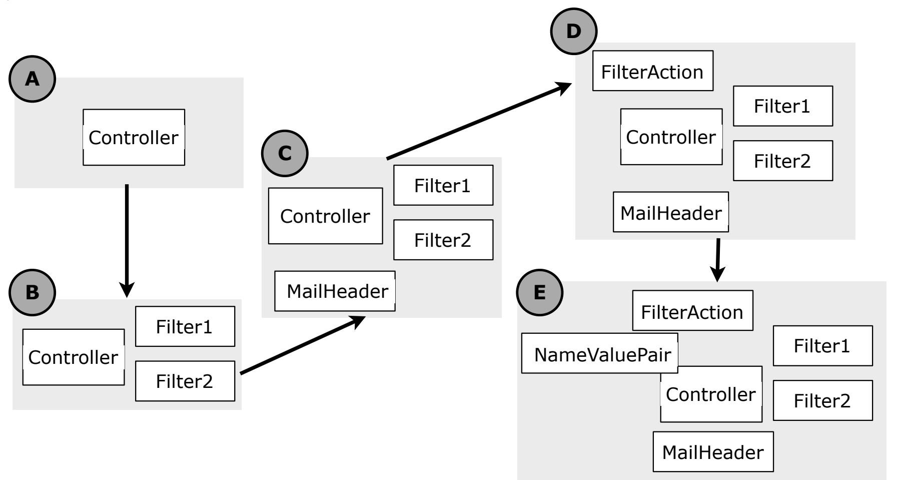
- Refactored designs are more complex
  - ➡ Introduce a lot of extra small classes/methods
- Use "code smells" as symptoms for refactoring opportunities
  - + Duplicated code
  - + Nested conditionals
  - + Large classes/methods
  - + Abusive inheritance

- 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

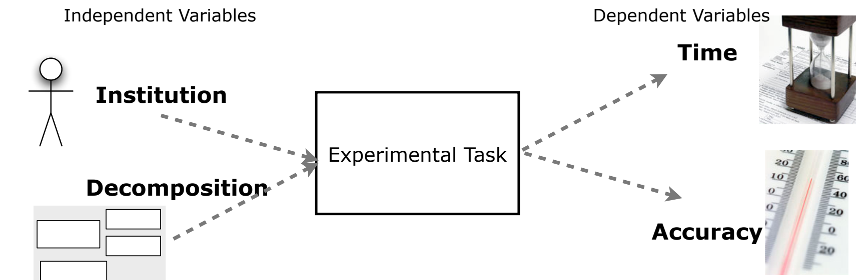
- Wiki-web with discussion on code smells
  - + http://c2.com/cgi/wiki?CodeSmells

Refactoring God Class: Optimal Decomposition?



Empirical Validation

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



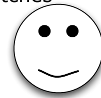
"Optimal decomposition" differs with respect to education
- Computer science: preference towards decentralized designs (C-E)
- ICT-electronics: preference towards centralized designs (A-C)

Advanced OO training can induce preference
- Consistent with [Arisholm et al. 2004]

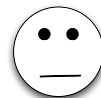
Correctness & Traceability

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 <-> 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/ass attribute are behaviour preserving ?
- Can you give the pre-conditions for a "rename method" 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 (3/5) on page 24)?

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 refectorings 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?

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?