Refactorings

Refactoring

- What is it?
- Why is it necessary?
- Examples
- Tool support

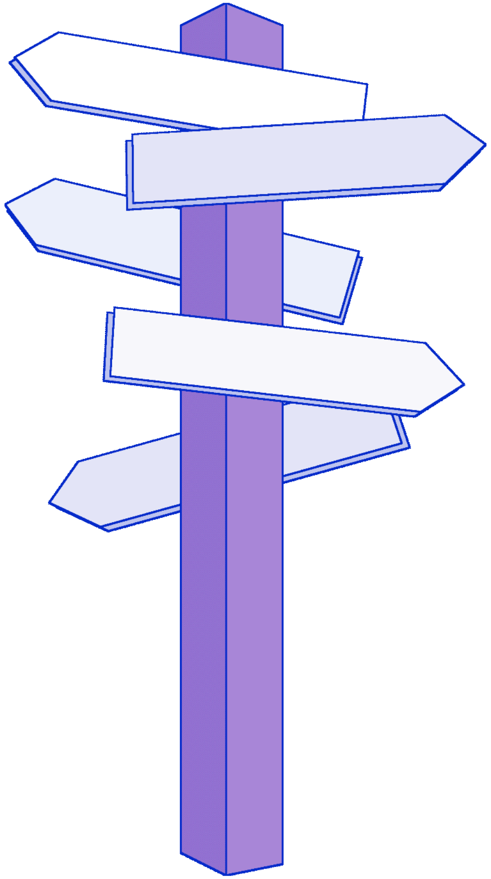
Refactoring Strategy

- Code Smells
- Examples of Cure

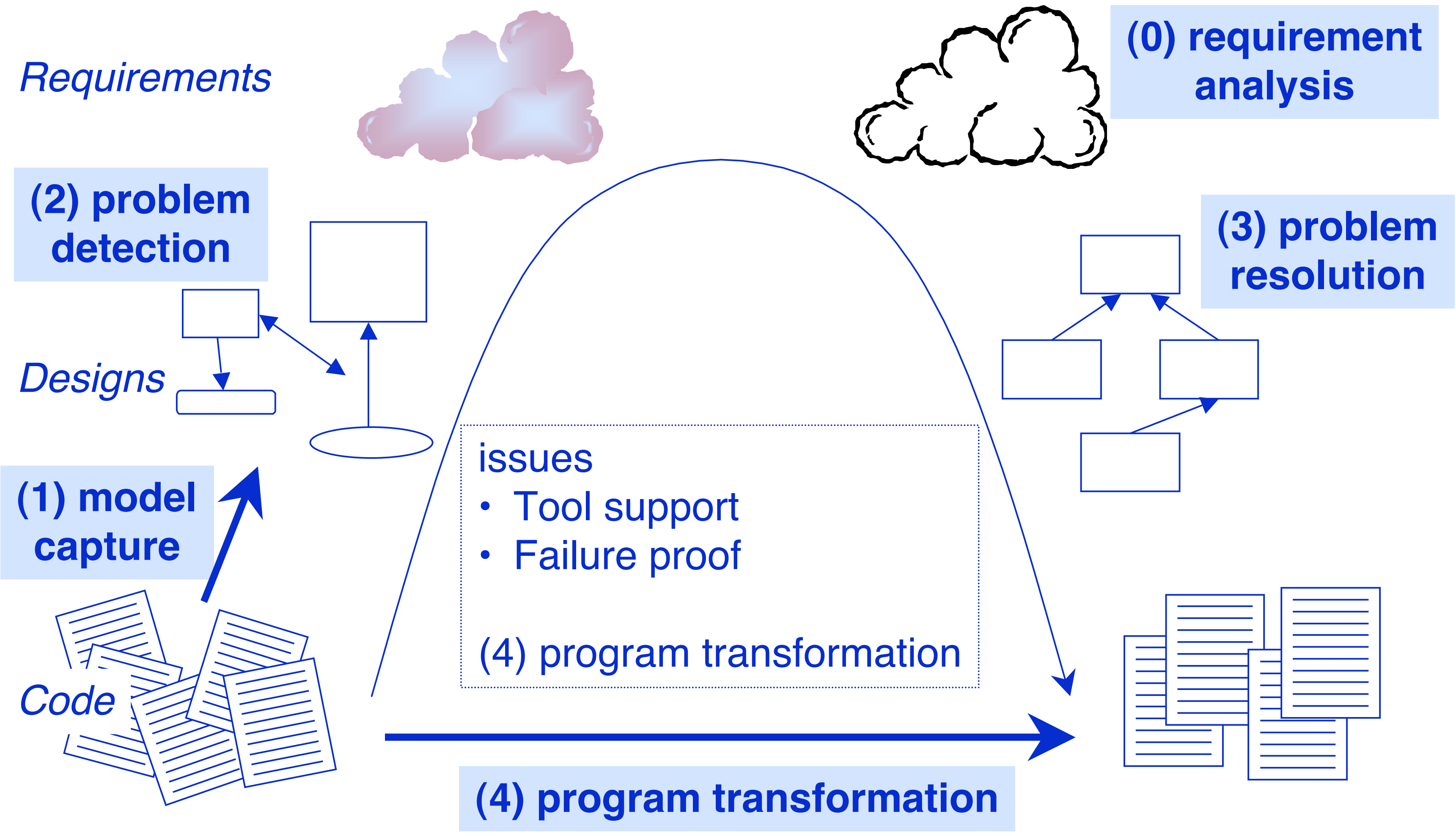
Demonstration: Refactoring and Reverse Engineering

- Refactor to Understand

Conclusion



The Reengineering Life-Cycle



issues
- Tool support
- Failure proof

(4) program transformation

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

List of refactorings provided by the refactoring browser

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

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

But in reality

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

Refactoring helps you to

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

Refactoring and OO

Object-Oriented Programming

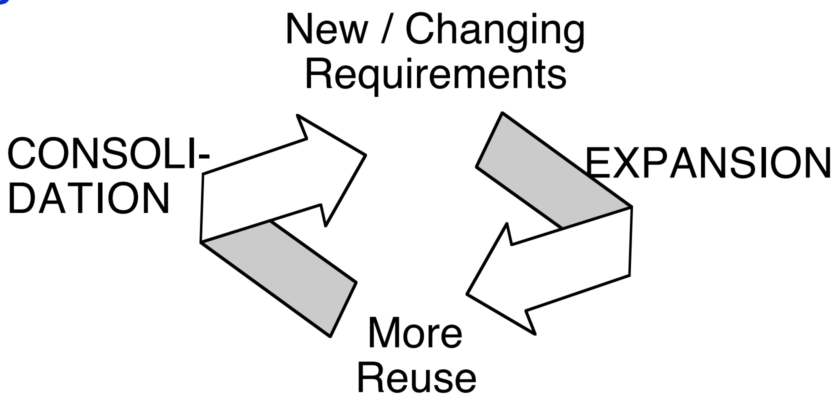
- emphasize the possibility of changes
- rapid development cycle
- incremental definition

Frameworks

- family of products from the same skeletons or kernel
- reuse of functionality

However software evolves, grows and...
dies if not taken care of
=> refactoring

Iterative development



Consolidation is necessary to ensure next expansion success

Examples of Refactoring Analysis

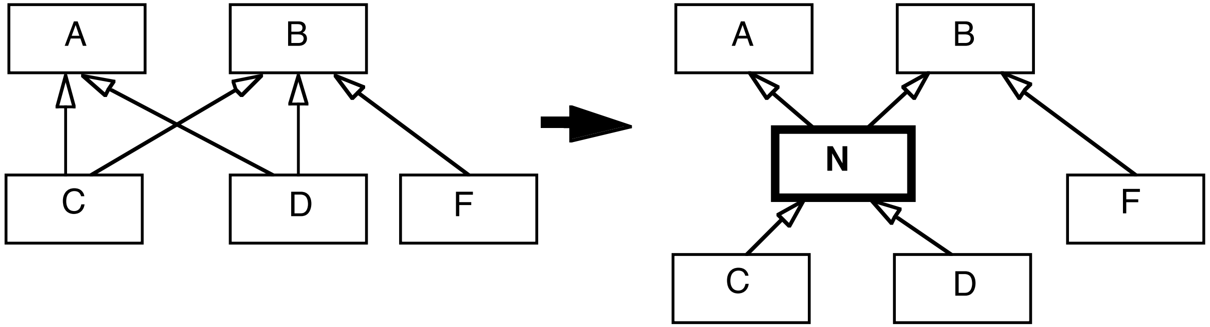
AddClass

- simple
- namespace use and static references between class structure

Rename Method

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

Add Class



Preconditions

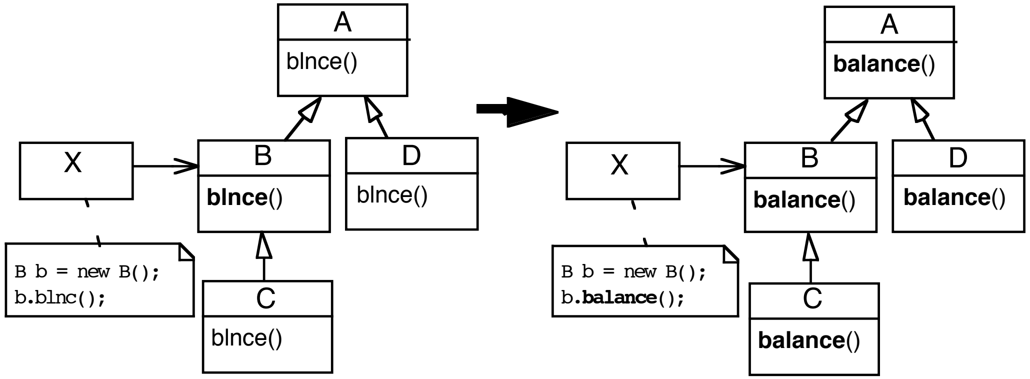
- no class and global variable exists with classname in the same scope
- subclasses are all subclasses of all superclasses
- [Smalltalk] superclasses must contain one class
- [Smalltalk] superclasses and subclasses cannot be metaclasses

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

Rename Method: Do It Yourself



```
B b = new B();
b.blnc();
```

```
B b = new B();
b.balance();
```

- Do it yourself approach
- 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
- Automated refactoring is better !

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

Which Refactoring Tools?

Change Efficient

Refactoring

- Source-to-source program transformation
- Behaviour preserving

=> improve the program structure

Programming Environment

- Fast edit-compile-run cycles
- Integrated into your environment
- 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

=> verify if improved structure does not damage previous work

Configuration & Version Management

- keep track of versions that represent project milestones

=> possibility to go back to previous version

Top Ten of Code Bad Smells (i)

"If it stinks, change it" Grandma Beck

- Duplicated Code
- Long Method
- Large Class (Too many responsibilities)
- Long Parameter List (Object is missing)
- Case Statement (Missing polymorphism)
- Divergent Change (Same class changes differently depending on addition)
- Shotgun Surgery (Little changes distributed over too much objects)

Top Ten of Code Bad Smells (ii)

- Feature Envy (Method needing too much information from another object)
- Data Clumps (Data always use together (x,y -> point))
- Parallel Inheritance Hierarchies (Changes in one hierarchy require change in another hierarchy)
- Lazy Class (Do not do too much)
- Middle Man (Class with too much delegating methods)
- Temporary Field (Attributes only used partially under certain circumstances)
- Message Chains (Coupled classes, internal representation dependencies)
- Data Classes (Only accessors)

Two Low-Level Cures

Long methods

- A method is the smallest unit of overriding
- Extract pieces as smaller method
- Comments are good delimiters

Not Intention Revealing Methods

- Rename Method

```
setType: aVal
   "compute and store the variable type"
   self addTypeList: (ArrayType with: aVal).
   currentType :=  (currentType computeTypes: (ArrayType with: aVal))
=>
computeAndStoreType: aVal
   self addTypeList: (ArrayType with: aVal).
   currentType :=  (currentType computeTypes: (ArrayType with: aVal))
```

One High-Level Cure: Duplicated Code

"Say everything exactly once" Kent Beck

Makes the system harder to understand and to maintain

- In the same class
- Extract Method

Between two sibling subclasses

- Extract Method
- Push identical methods up to common superclass
- Form Template Method

Between unrelated class

- Create common superclass
- Move to Component
- Extract Component (e.g., Strategy)

Other High-Level Cures

God Class

- Find logical sub-components (set of working methods/instance variables)
- Move methods and instance variables into components
- Extract component
- If not using all the instance variables
- Extract Subclass

Nested Conditionals

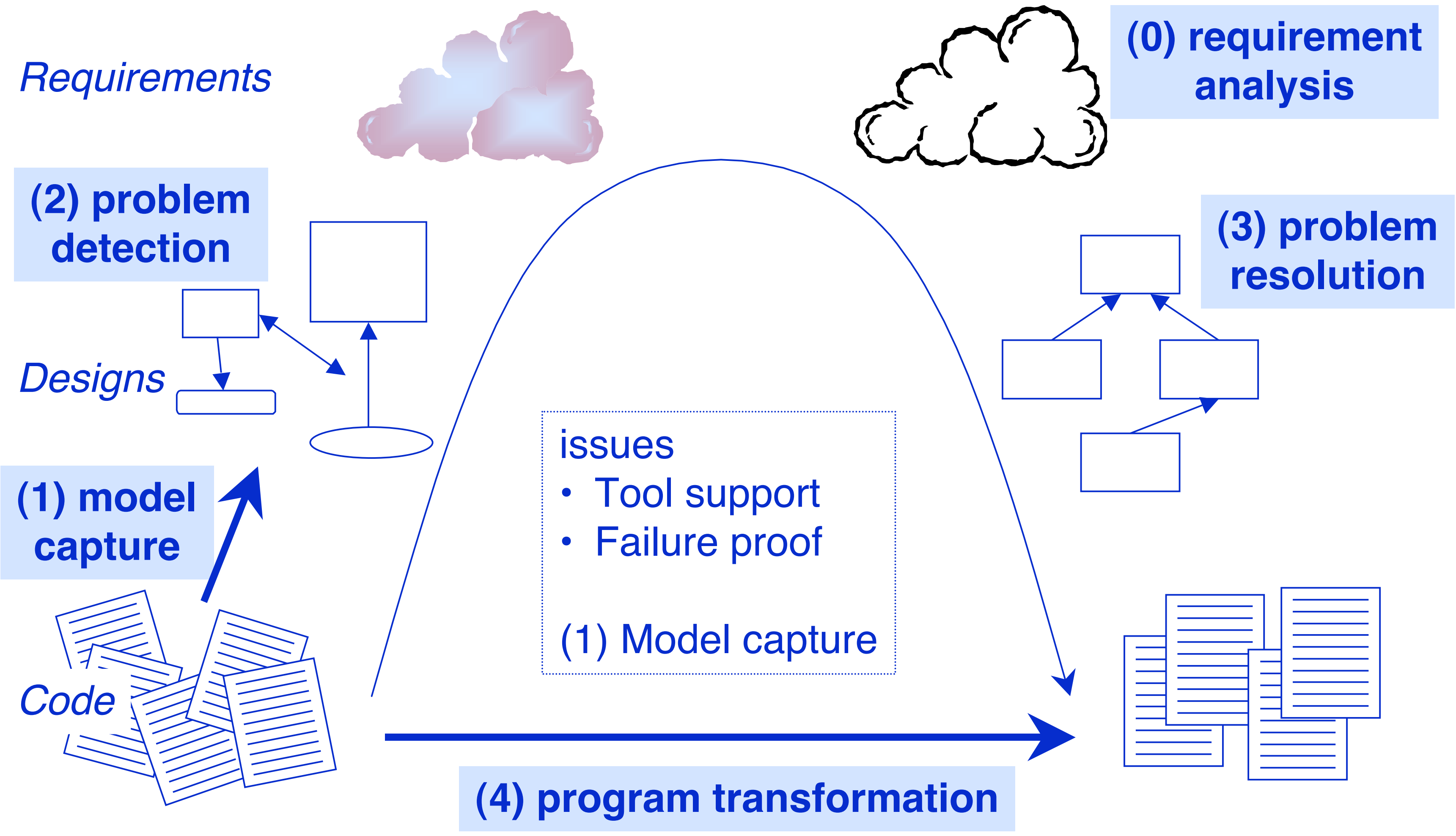
New cases should ideally not require changing existing code

May apply the State / Strategy / NullObject pattern

Use dynamic dispatch

- Define classes if not created
- Define abstract method in superclass
- Define makeCall methods
- Extract Methods

Refactor and Reverse Engineerging



issues

- Tool support
- Failure proof

(1) Model capture

Refactor To Understand

The Obvious:

- Programs hard to read => Programs hard to understand => Programs hard to modify
- Programs with duplicated logic are hard to understand
- Programs with complex conditionals are hard to understand
- Programs hard to modify

Refactoring code creates and supports the understanding

- Renaming instance variables helps understanding methods
- Renaming methods helps understanding responsibility
- Iterations are necessary

The refactored code does not have to be used!

Obstacles to Refactoring

Complexity

- Changing design is hard
- Understanding code is hard

Possibility to introduce errors

- Run tests if possible
- Build tests

Clean first **Then** add new functionality

Cultural Issues

- Producing negative lines of code, what an idea!
- *"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!"*

Conclusion: Tool Support

Refactoring Philosophy

combine simple refactorings into larger restructuring

=> improved design

=> ready to add functionality

Do not apply refactoring tools in isolation

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

Obstacles to Refactoring

- Performance issue
  - Refactoring may slow down the execution
  - The secret to write fast software:

    "Write tunable software first then tune it"
- Normally only 10% of your system consumes 90% of the resources so just focus on 10 %.
  - Refactorings help to localize the part that need change
  - Refactorings help to concentrate the optimizations
- Development is always under time pressure
  - Refactoring takes time
  - Refactoring better after delivery

Conclusion: Know-when & Know-how

- Know when is as important as know-how
  - Refactored designs are more complex
  - Use "*code smells*" as symptoms
  - Rule of the thumb: "*Once and Only Once*" (Kent Beck)

    => a thing stated more than once implies refactoring

- More about code smells and refactoring
  - Book on refactorings [Fowl99a].
  - http://www2.awl.com/cseng/titles/0-201-89542-0/refactor/

Wiki-web with discussion on code smells

- http://c2.com/cgi/wiki?CodeSmells

Refactoring Browser

- http://wiki.cs.uiuc.edu/RefactoringBrowser
- http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser/