OO Reengineering Patterns

O.Univ.-Prof. DI Dr. Wolfgang Pree
Universität Salzburg
www.SoftwareResearch.net

Introduction

Goals

Convince you about the following:

- ❑ Yes, Virginia, there are *object-oriented legacy systems* too!
- ❑ Reverse engineering and reengineering are *essential activities* in the lifecycle of any successful software system. (And especially OO ones!)
- ❑ There is a large set of *lightweight tools* and techniques to help you with reengineering.

Lehman‘s laws

A classic study by Lehman and Belady [Lehm85a] identified several “laws” of system change.

Continuing change

- A program that is used in a real-world environment *must change, or become progressively less useful* in that environment.

Increasing complexity

- As a program evolves, it *becomes more complex, and extra resources are needed* to preserve and simplify its structure.

What is a legacy system?

A legacy system is a piece of software that:

- ❑ you have *inherited*, and
- ❑ is *valuable* to you.

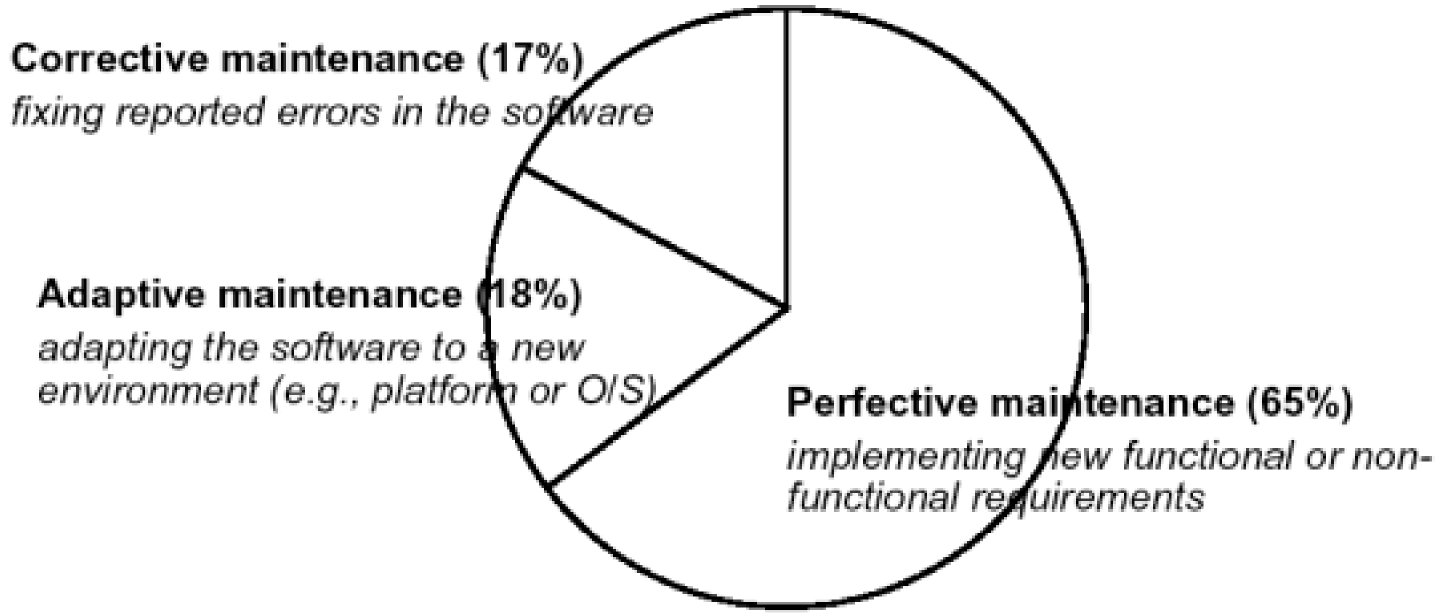
Typical problems with legacy systems are:

- ❑ original developers no longer available
- ❑ outdated development methods used
- ❑ extensive patches and modifications have been made
- ❑ missing or outdated documentation

so, *further evolution and development may be prohibitively expensive*

Software maintenance

Software Maintenance is the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" [ANSI/IEEE Std. 729-1983]



Various studies show 50% to 75% of available effort is spent on maintenance.

What about OO?

Any successful software system will suffer from the symptoms of legacy systems.

Object-oriented legacy systems are successful OO systems whose architecture and design no longer responds to changing requirements.

- ❑ The symptoms and the source of the problems are the same.
- ❑ The technical details and solutions may differ.

Although OO techniques promise better flexibility, reusability, maintainability etc. etc., *they do not come for free*

The claim:

A *culture of continuous reengineering* is a prerequisite for flexible and maintainable object-oriented systems.

Definitions

"Forward Engineering is the traditional *process of moving from* high-level abstractions and logical, implementation-independent *designs to the physical implementation* of a system."

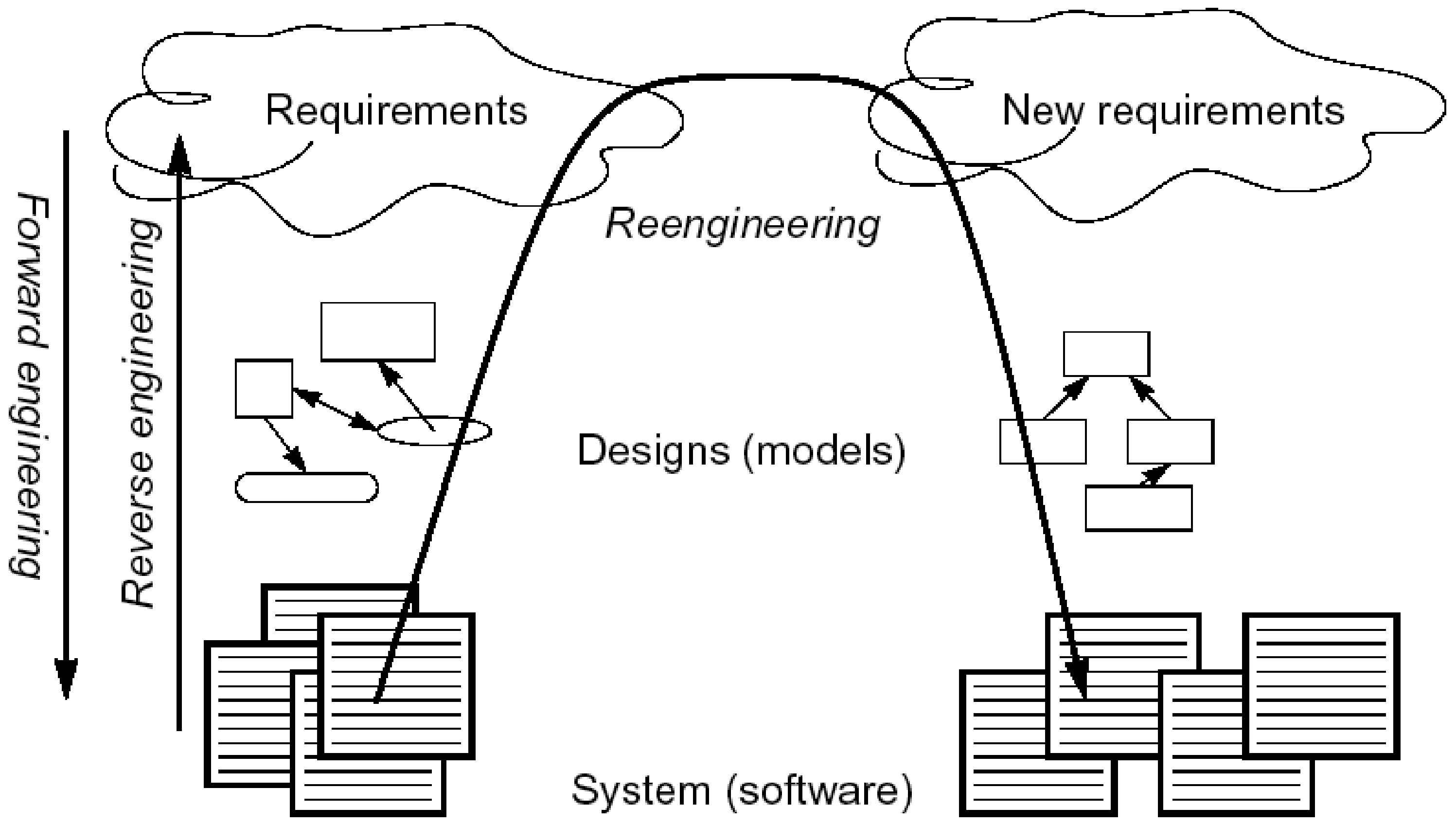
"Reverse Engineering is the process of *analyzing a subject system* to

- ❑ identify the system's components and their interrelationships and
- ❑ create representations of the system in another form or at a higher level of abstraction."

"Reengineering ... is the examination and *alteration of a subject system* to reconstitute it in a new form and the subsequent implementation of the new form."

— *[Chik90a] & [Chik90b]*

Reverse and reengineering



Goals of reverse engineering

Cope with complexity
- ❑ need techniques to understand large, complex systems

Generate alternative views
- ❑ automatically generate different ways to view systems

Recover lost information
- ❑ extract what changes have been made and why

Detect side effects
- ❑ help understand ramifications of changes

Synthesize higher abstractions
- ❑ identify latent abstractions in software

Facilitate reuse
- ❑ detect candidate reusable artifacts and components

Reverse engineering techniques

"Redocumentation is the *creation or revision of a semantically equivalent representation* within the same relative abstraction level."

- ❑ pretty printers
- ❑ diagram generators
- ❑ cross-reference listing generators

"Design recovery *recreates design abstractions* from a combination of code, existing documentation (if available), personal experience, and general knowledge about problem and application domains." [Bigg89c] & [Bigg89d]

- ❑ software metrics
- ❑ browsers, visualization tools
- ❑ static analyzers
- ❑ dynamic (trace) analyzers

Goals of reengineering

Unbundling

- ❑ split a monolithic system into parts that can be separately marketed

Performance

- ❑ "first do it, then do it right, then do it fast"

Port to other Platform

- ❑ the architecture must distinguish the platform dependent modules

Design extraction

- ❑ to improve maintainability, portability, etc.

Exploitation of New Technology

- ❑ i.e., new language features, standards, libraries, etc.

Reengineering techniques

"Restructuring is the *transformation from one representation form to another* at the same relative abstraction level, while preserving the system's external behaviour."

- automatic conversion from unstructured ("spaghetti") code to structured ("goto-less") code
- source code translation

"Data reengineering is the process of analyzing and *reorganizing the data structures* (and sometimes the data values) in a system to make it more understandable."

- integrating and centralizing multiple databases
- unifying multiple, inconsistent representations
- upgrading data models

Refactoring is restructuring within an object-oriented context

- renaming/moving methods/classes etc.

Architectural problems

Insufficient documentation

- most legacy systems suffer from inexistent or inconsistent documentation

Duplicated functionality

- "cut, paste and edit" is quick and easy, but leads to maintenance nightmares

Lack of modularity

- strong coupling between modules hampers evolution

Improper layering

- missing or improper layering hampers portability and adaptability

Refactoring opportunities

Misuse of inheritance

- ❑ for composition, code reuse rather than polymorphism

Missing inheritance

- ❑ duplicated code, and case statements to select behaviour

Misplaced operations

- ❑ unexploited cohesion — operations outside instead of inside classes

Violation of encapsulation

- ❑ explicit type-casting, C++ “friends” ...

Class misuse

- ❑ lack of cohesion — classes as namespaces

Tool integration

Tool integration—overview

- ❑ Why Integrate Tools?
- ❑ Which Tools to Integrate?
- ❑ Tool Integration Issues
- ❑ The “Help yourself” approach
  - How to Obtain Data?
  - API Examples (Java, SNiFF+, Rational/Rose)
- ❑ Exchange Standards
  - CDIF & MOF
  - UML shortcomings

Why integrate tools?

Tool Adage

Tools are necessary to improve productivity.

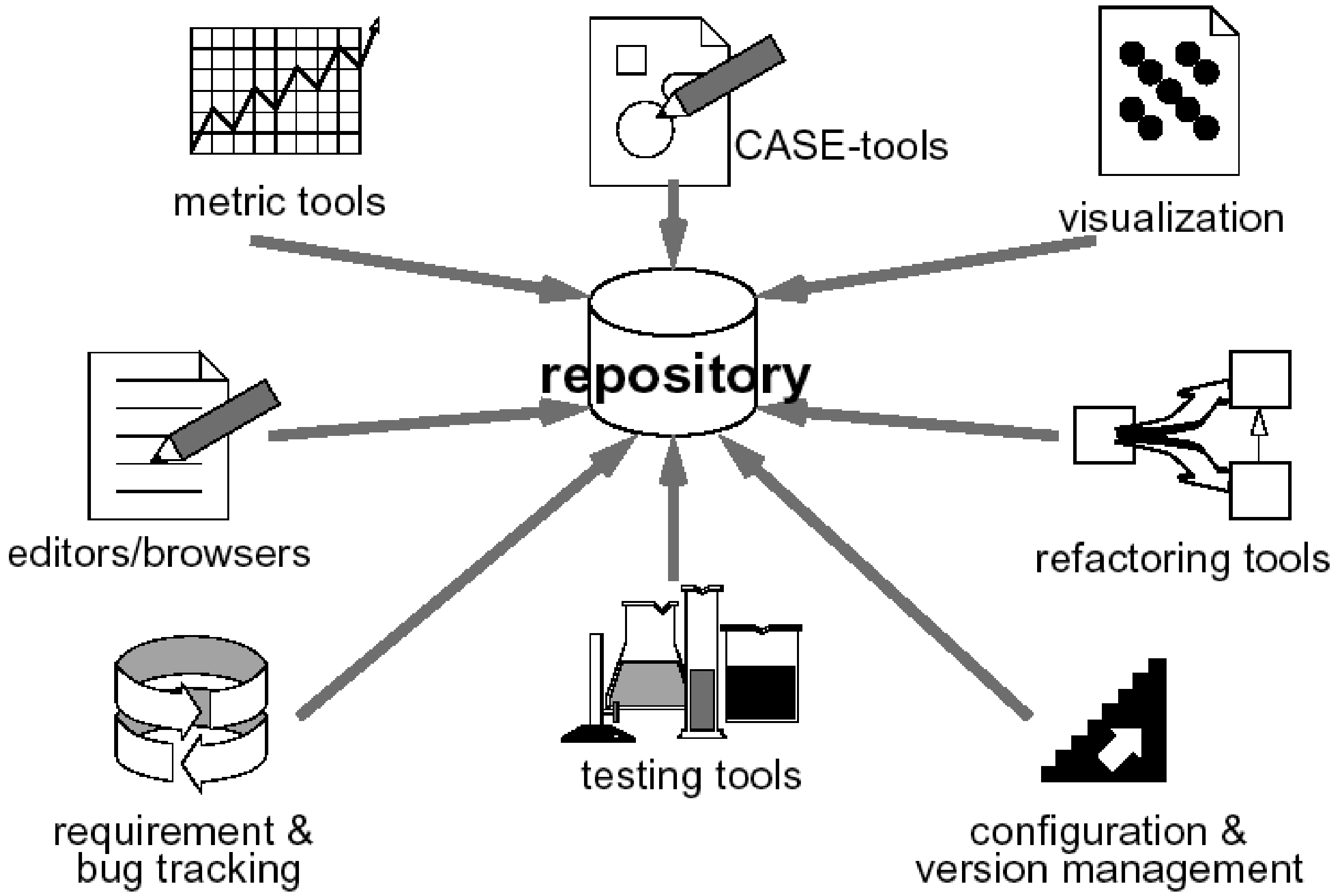
Tool Principle

Give Software Tools to Good Engineers. You want bad engineers to produce less, not more, poor-quality software [Davi95a].

Towards CARE

- ❑ **CAD/CAM** Computer Aided Design / Manufacturing - Late 70's
  Create and validate design diagrams & steer manufacturing processes
- ❑ **CASE** Computer Aided Software Engineering - Late 80's
  Support (parts of) the Software Engineering Process
- ❑ **CARE** Computer Aided Reengineering - Mid 90's
  Support Software Reengineering Activities
  - ☞ Y2K tools
  - ☞ Round-trip engineering

Which tools to integrate?



Tool integration issues

Reengineering vs. forward engineering

- ❑ Forward engineering tools are chosen deliberately.
- ❑ Reengineering tools must integrate with what's already in place.

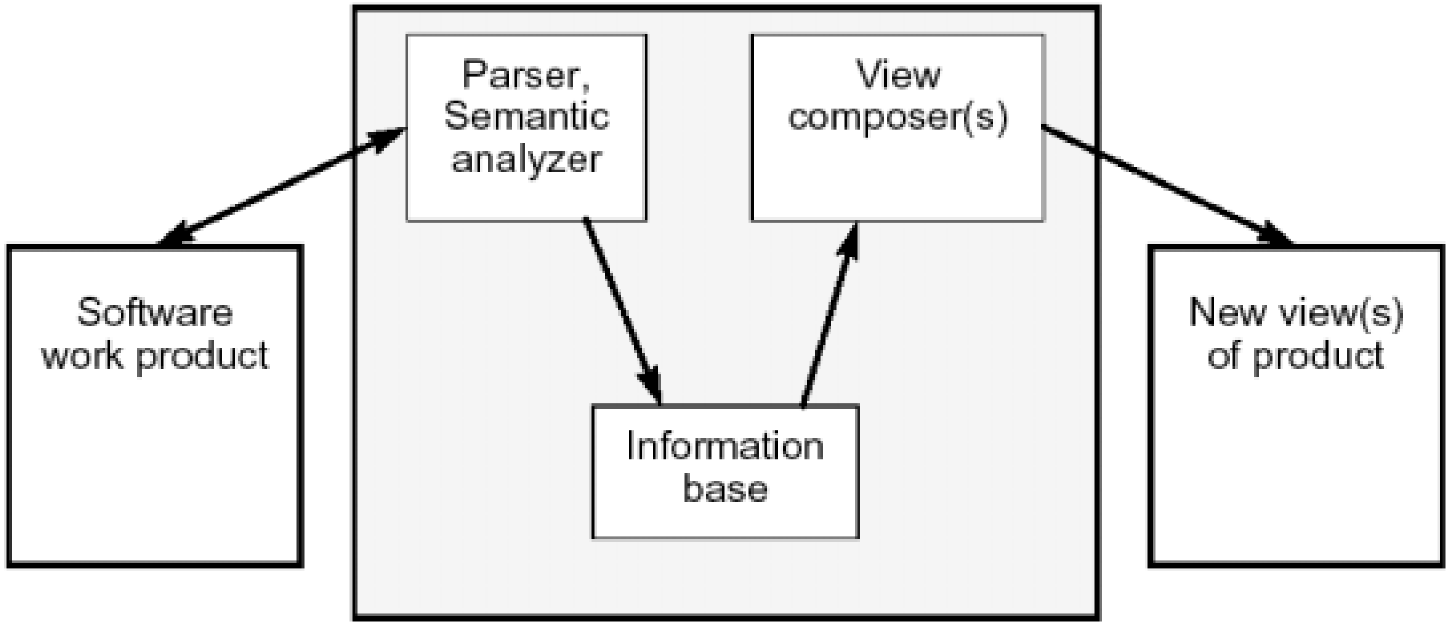
  - ☞ Tool integration in reengineering is harder
    ... but we can rely on forward engineering experience
  - ☞ "Help yourself" approach

Tools must work together

❑	share data	=> repository
❑	synchronize activities	=> API
❑	different vendors	=> interoperability standards

Basic tool architecture

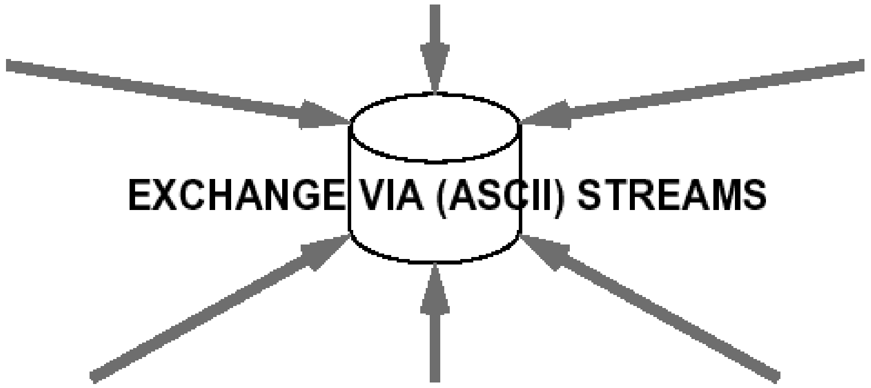
"Most tools for reverse engineering, restructuring and reengineering use the same basic architecture." [Chik90a], [Chik90b]



Help yourself approach

- build your own parser
- translate between file formats
- communicate via APIs
- collect execution traces

Exchange standards



Standardization Efforts

- ❑ CDIF (CASE data interchange format) - see http://www.eigroup.org/
  Mature standard (being approved by ISO)
  Little commitment from tool vendors
- ❑ MOF (Meta-Object Facility) from OMG - see http://www.omg.org/
  Currently immature (approved by OMG late 1997)
  Major commitment from tool vendors to be expected
  Builds on UML and CORBA/IDL

Reference format

- **Issue**

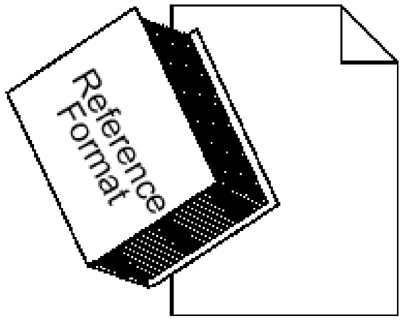
  How can tools exchange information without being aware of each other?
- **Answer**

  Tools agree on a single reference format
- **Analogy**

  How can French, German and Italian persons exchange documents? They agree to write their documents in Esperanto.
- **Advantage**

  Only need for one translation dictionary
- **Disadvantage**

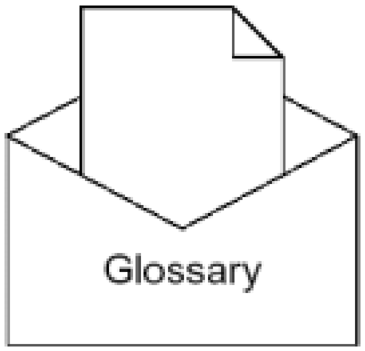
  Centralised reference models do not work in practice
  - Need for specialised constructs (i.e. jargon)
  - Cannot predict future needs



Openness

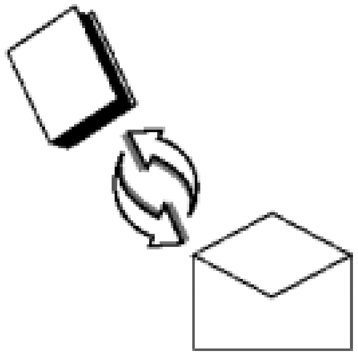
Specialised Constructs

- **Issue**
  How can tools extend the reference model with specialised constructs?
- **Answer**
  Each tool wraps the information with a glossary, explaining the specialised constructs in terms of a core reference model. *=> meta model*



Multiple Standards

- **Issue**
  How can tools deal with future extensions?
- **Answer**
  Define a small and generic core format. All glossaries (=meta models) define bidirectional mapping with the core model. *=> meta-meta model*



Meta models

Exchange standards community cultivated specialised terminology

☞ the Four Layer Metamodeling Architecture

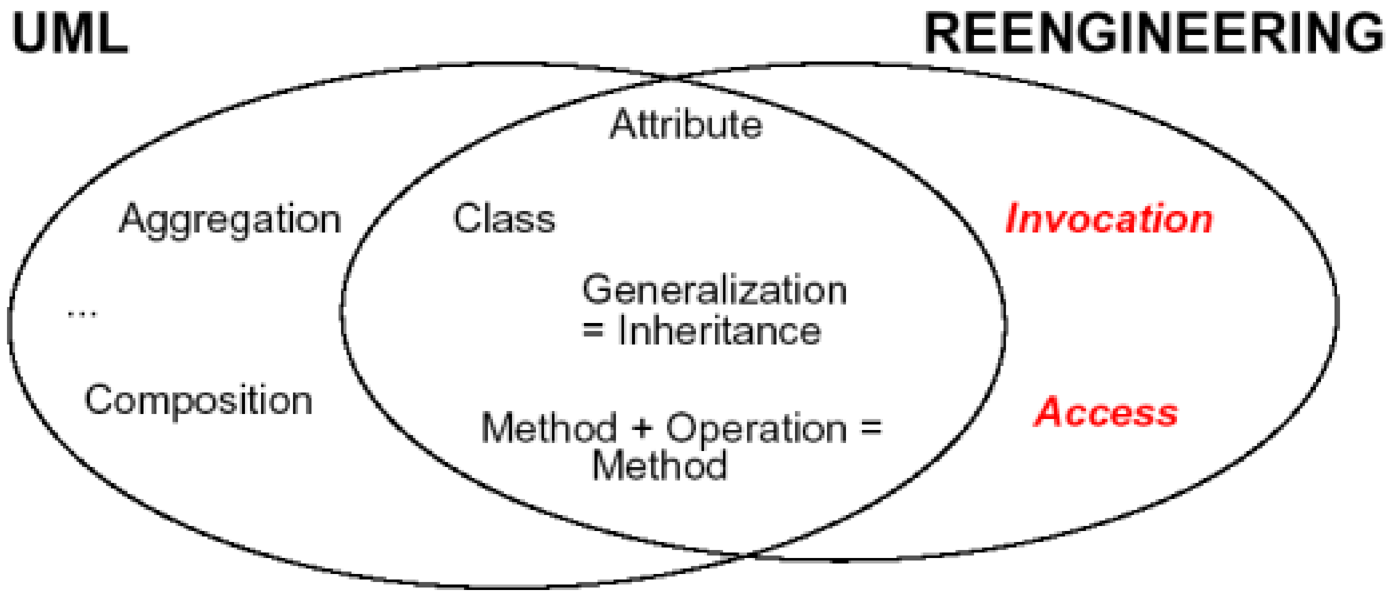
Layer	**Description**	**Example**
Meta Meta Model	Defines the core ingredients sufficient for defining languages for specifying meta-models	*(CDIF)* MetaEntity, MetaAttribute *(MOF)* Class, MofAttribute
Meta Model	Defines a language for specifying Models	(*UML*) Class, Attribute, Association (*Database*) Table, Column, Row
Model	Defines a language to describe an information domain.	Student, Course, enrolled_in
User Objects	Describes a specific situation in an information domain.	Student#3, Course#5, Student#3.enrolled_in.Course#5

SOFTWARE RESEARCH LAB

UML shortcomings

Current standardization efforts are geared towards UML.

- ☞ not enough for reengineering
- ☞ need “Invocation” & “Access”



- ❑ use extension mechanisms on the meta-model
  => how standard is standard?
- ❑ define a special reengineering standard (i.e., own meta-model)

Conclusion

- Reengineering requires Tools
  - Much in common with forward engineering
  - Must integrate with what's already in place
- "Help yourself" approach
  - Build your own parser
  - Translate between file-formats
  - Communicate via API's
  - Collect Execution Traces
- Standardization Efforts
  - CDIF is mature / MOF is safest bet for future
  - Extensibility via Meta models (4 layer architecture)
  - UML has shortcomings

Design extraction

extreme situation

"Company X is in trouble.
Their product is successful (they have 60% of the world market).
But:

- all the original developers left,
- there is no documentation at all,
- there is no comment in the code,
- the few comments are obsolete,
- there is no architectural description,...

And they must change the product to take into account new client requirements.
They asked a student to reconstruct the design."

Goals

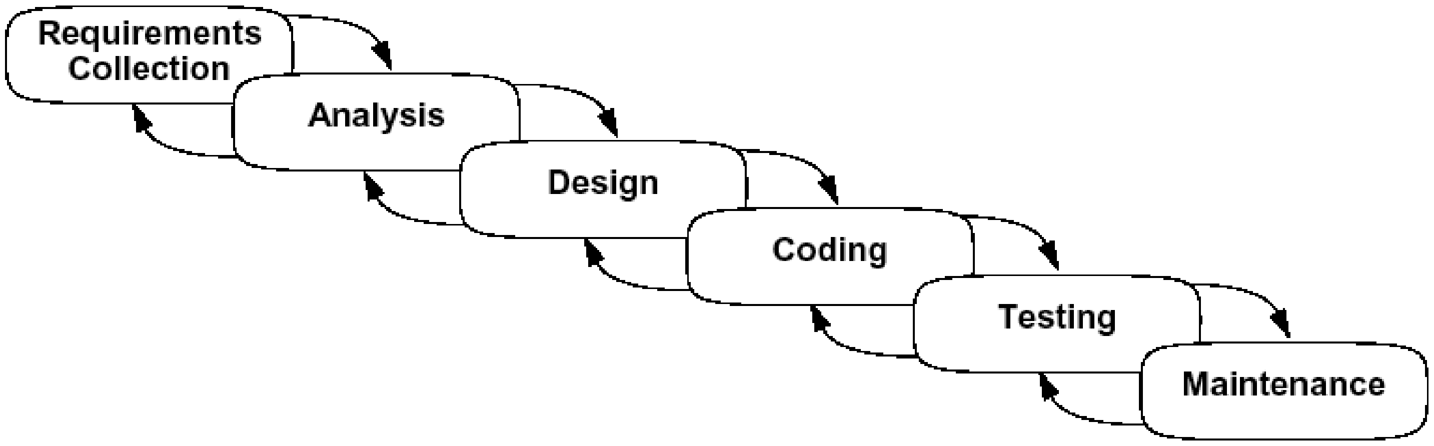
- Design is not code displayed with boxes and arrows
- Design extraction is not trivial
  - scalability
  - not fully automatized -> needs human intervention to filter out
- Give a critic view on hype: “we read your code and produce design”
- Show that UML is not that simple and clear
- Show that conventions for the interpretation are crucial
  - Language mapping
  - UML interpretation

What is ‘design‘?

”Design is really two activities: architectural design and detailed design.

Architectural design involves making strategic decisions about how system functionality is factored among independent system components, how components relate and how control transfers from one component to another. It often includes a specification of how users give and receive information, and how the system communicates with other systems.

Detailed design consists of tactical decisions, such as the choice of algorithms and data structures to meet performance and space objectives” [Gold95a]



Why design extraction is needed?

- ❑ Documentation inexistent, obsolete or too prolix
- ❑ Abstraction needed to understand applications (complexity)
- ❑ Original programmers left
- ❑ Only the code available

Why UML?

- ❑ Standard
- ❑ Communication based on a common language
- ❑ Can support documentation if we are precise about its interpretation
- ❑ Extensible
- ❑ Hype and market!

Small example—straighten UML reverse engineered diagrams (I)

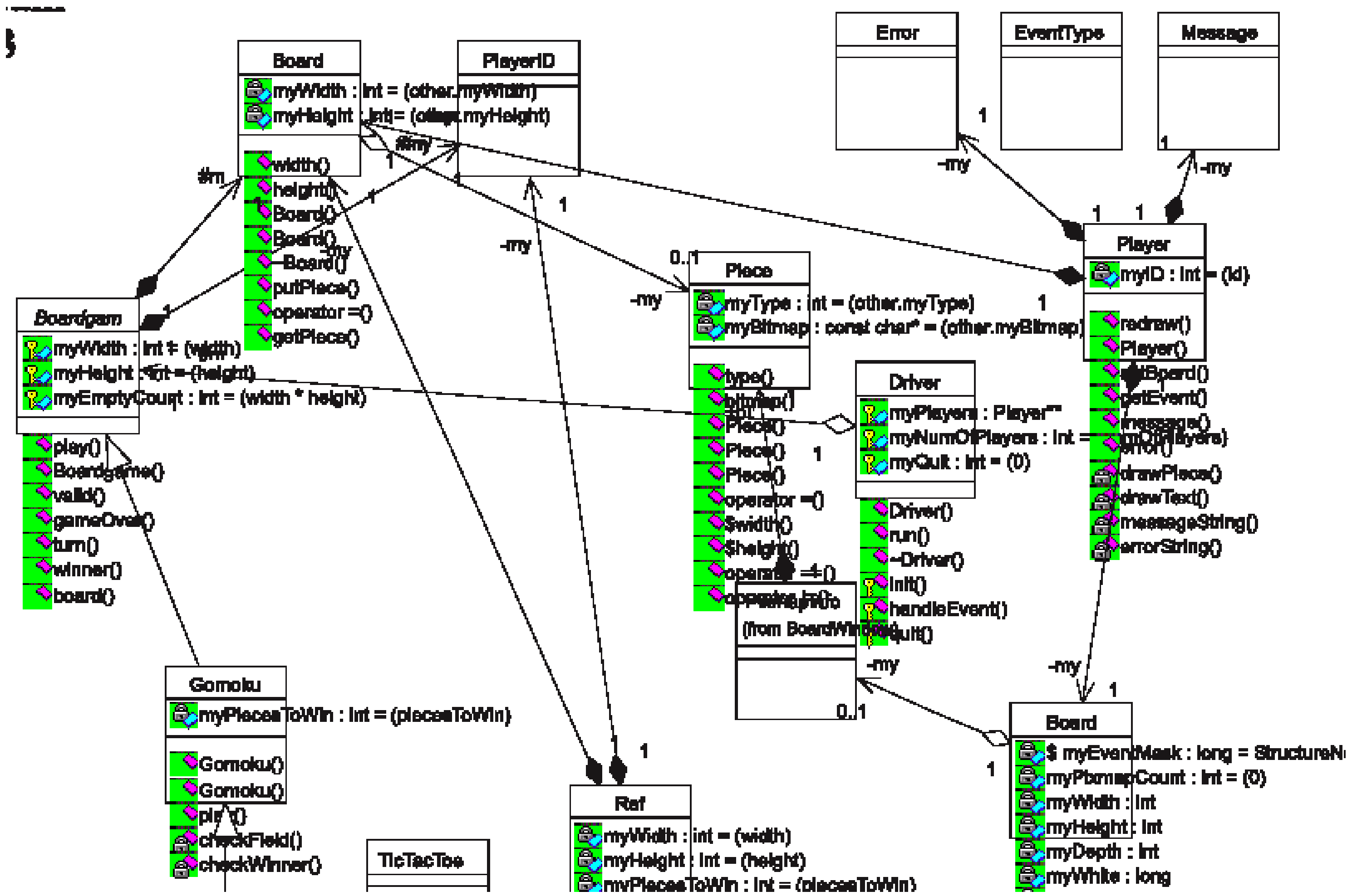
A small example in C++: A Tic-Tac-Toe Game!

You will do it now........

But:

- ❑ do not interpret the code
- ❑ do not make any assumption about it
- ❑ do not filter out anything

Small example—straighten UML reverse engineered diagrams (II)



slides are based on OO Reengineering (Demeyer, Ducasse, Nierstrasz)

Small example—straighten UML reverse engineered diagrams (III)

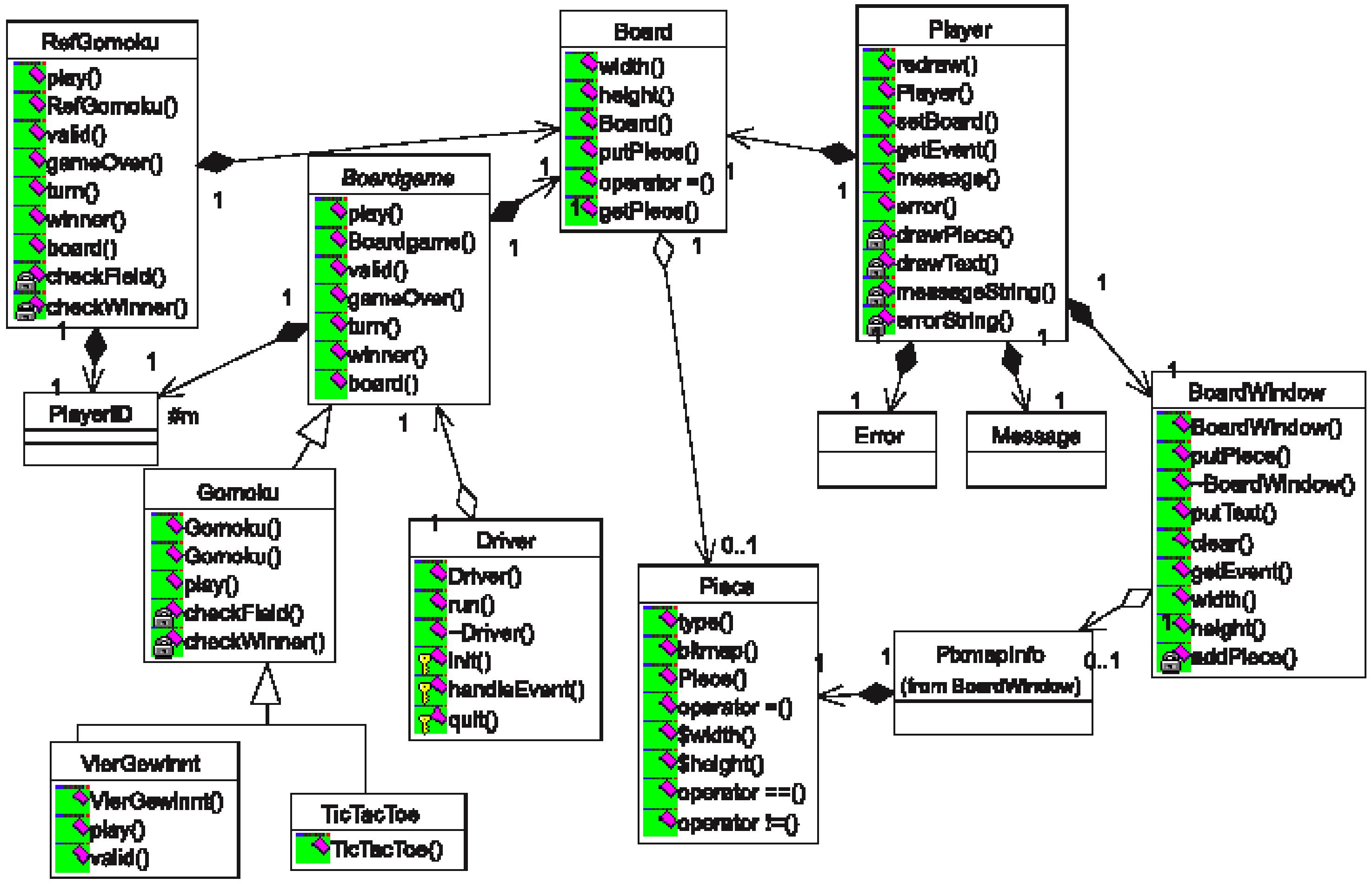
We should have heuristics to extract the design.

Try to clean the previous solution you found

Try some heuristics like removing:

- ❑ private information,
- ❑ remove association with non domain entities,
- ❑ simple constructors,
- ❑ destructors, operators

Small example—straighten UML reverse engineered diagrams (IV)



slides are based on OO Reengineering (Demeyer, Ducasse, Nierstrasz)

Essential questions when interpreting UML

When we extract design we should be precise about:

- ❑ What are we talking about? Design or implementation?
- ❑ What are the conventions of interpretation that we are applying?
- ❑ What is our goal: documentation programmers, high level views....

UML purists do not propose different levels of interpretation, they refer to the UML semantics!

- ❑ Levels of interpretations are not of UML but there are necessary!

  What is the sense of representing subclassing based inheritance between two classes using generalization?

    Dictionary is a subclass of Set in Smalltalk (subclassing)

    but a Dictionary is not a subtype nor generalization of Set

So at the minimum we should have:

- ☞ Clear level of interpretation + Clear conventions + Clear goal + UML extensions: stereotypes

Levels of interpretation: perspectives

Fowler proposed 3 levels of interpretations called perspectives [Fowl97a]:

- conceptual
- specification
- implementation

Three Levels:

- ❑ Conception: we draw a diagram that represents the concepts that are somehow related to the classes but there is often no direct mapping.
- ❑ Specification: we are looking at interfaces of object not implementation, types rather than classes. Types represent interfaces that may have many implementations
- ❑ Implementation: implementation classes

Attributes in perspectives

Syntax:

  visibility attributeName: attributeType = defaultValue

  + name: String

Conceptual:

  Customer name = Customer has a name

Specification:

  Customer class is responsible to propose some way to query and set the name

Implementation:

  Customer has an attribute that represents its name

Possible Refinements	
Attribute Qualification	- Immutable: Value never change
	- Read-only: Client cannot change it

Operations in perspectives

Syntax: visibility name (parameter-list):return-type
+ public, # protected, - private

- Conceptual: principal functionality of the object. It is often described as a sentence
- Specification: public methods on a type
- Implementation: methods

Can be approximate to methods but operations are more abstract methods
methods represent how such operations are defined.

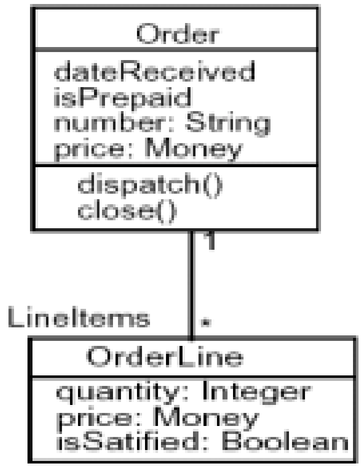
Possible Refinements:

-Method qualification: Query (does not change the state of an object)
Cache (does cache the result of a computation), Derived Value (depends on the value of other values), Getter, Setter

Associations

- Represent relationships between instances

- Each association has two roles: each role is a direction on the association.
  - a role can be explicitly named, labelled near the target class
    if not named from the target class and goes from a source class to a target class
  - a role has a multiplicity: 1, 0, 1..*, 4



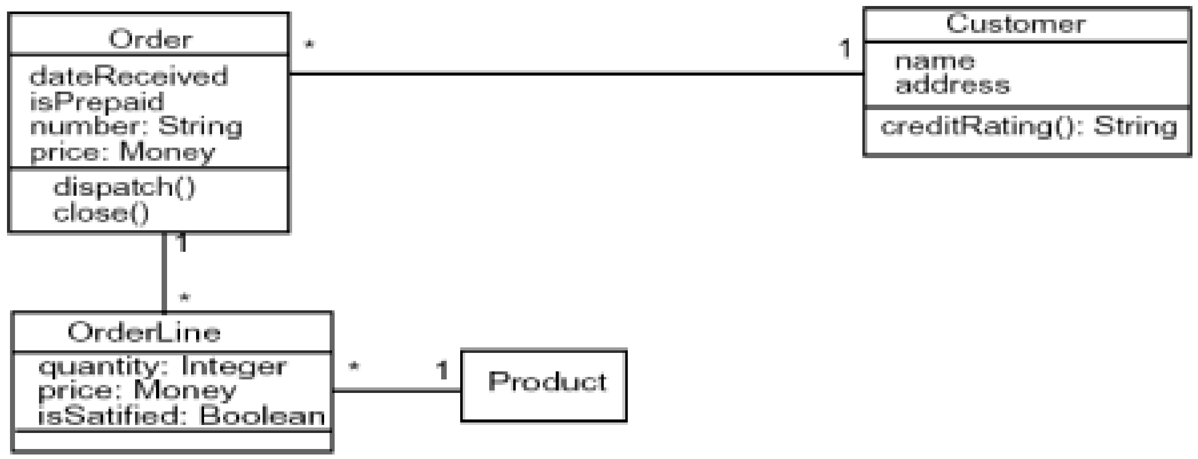
LineItems = role of direction Order to OrderLines

LineItems role = OrderLine role

One Order has several OrderLines

Associations—conceptual perspective

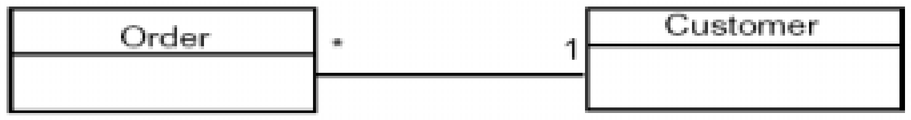
Conceptual Perspective: associations represent conceptual relationships between classes



An Order has to come from a single Customer.
A Customer may make several Orders.
Each Order has several OrderLines that refers to a single Product.
A single Product may be referred to by several OrderLines.

Associations—specification perspective

Specification Perspective: Associations represent responsibilities



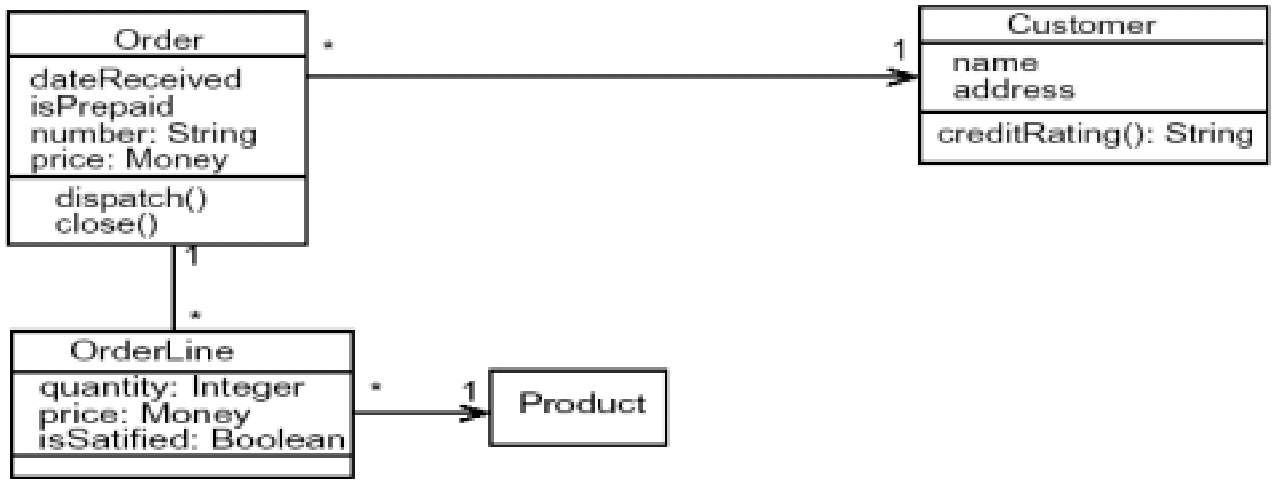
Implications:

- One or more methods of Customer should tell what Orders a given Customer has made.
- Methods within Order will let me know which Customer placed a given Order and what Line Items compose an Order

Associations also implies responsibilities for updating the relationship, like:

- specifying the Customer in the constructor for the Order
- add/removeOrder methods associated with Customer

Arrows—navigability



No arrow = navigability in both sides or unknown

☞ conventions needed!!

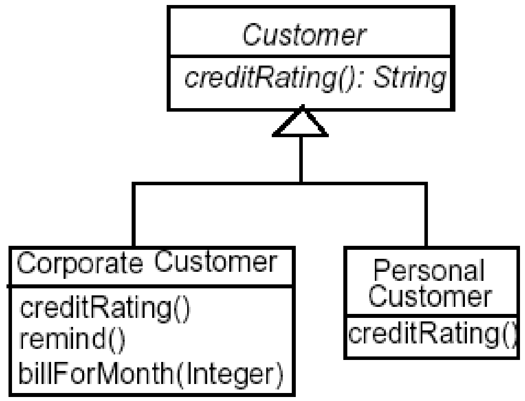
- Conceptual perspective: no real sense
- Specification perspective: responsibility

  an Order has the responsibility to tell which Customer it is for but Customer don't
- Implementation perspective:

  an Order points to a Customer, an Customer doesn't

SOFTWARE RESEARCH LAB

Generalization



Conceptual: What is true for an instance of a superclass is true for a subclass (associations, attributes, operations).

Corporate Customer is a Customer

Specifications: interface of a subtype must include all elements from the interface of a superclass (conformance).

Substituability principle: if that’s works for superclass that should works for a subclass.

Implementation: Generalization semantics is not inheritance. But we should interpret it this way for representing extracted code.

A subclass inherits all the methods and fields of its superclass(es). It may override some of them.

Need for a clearer mapping

UML

- ❑ language independent
- ❑ fuzzy (navigability, package...)
  - ☞ We should define how we interpret it:
  - ☞ define some conventions

In C++, examples show that:

```
Board& board()
Board& operator =(const Board& other) throw (const char*);
```

board(): Board

```
Piece* myMap;
```

myMap: Piece

```
class Gomoku: public Boardgame {
```

«public inherits»

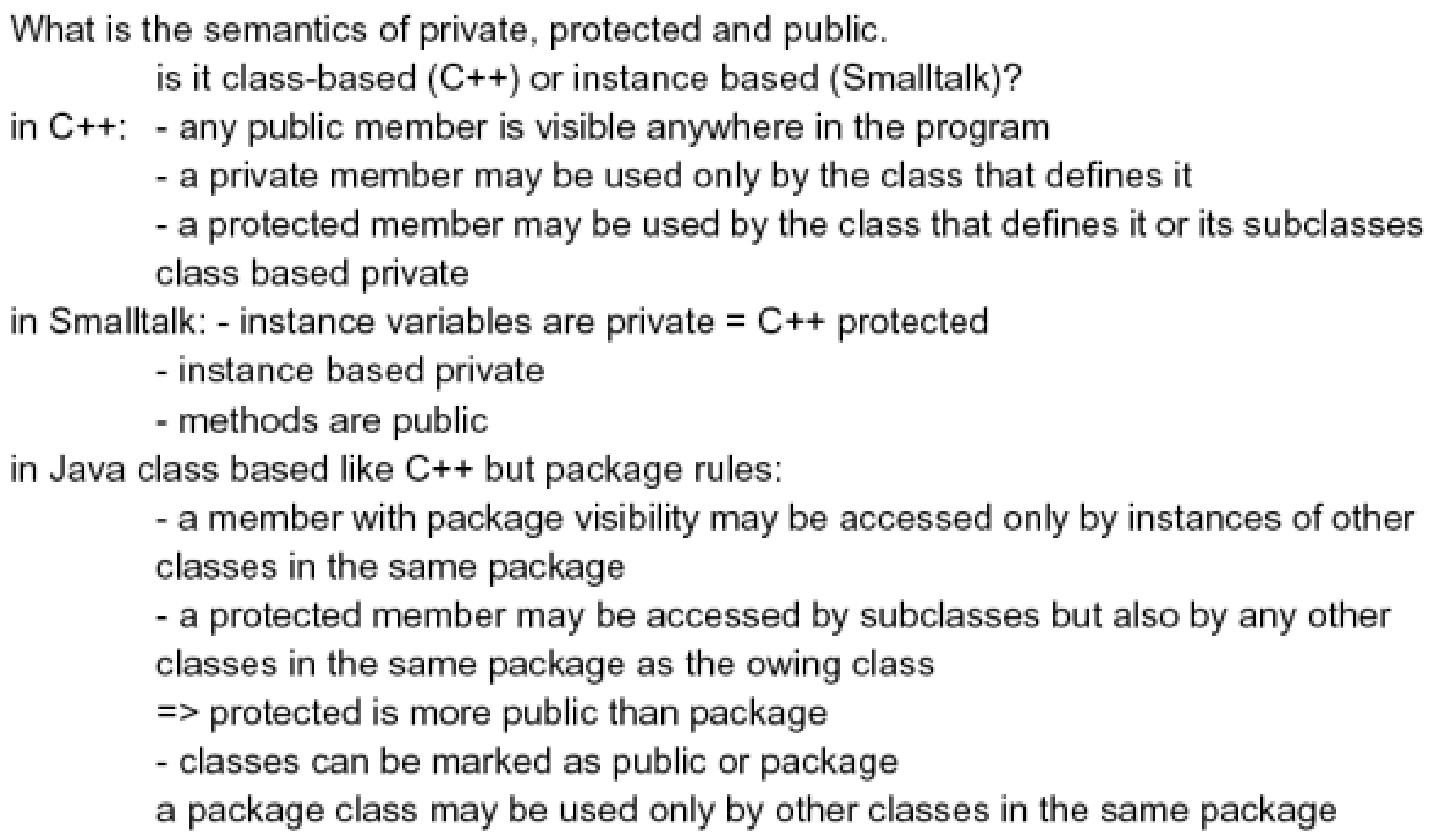
```
virtual void checkWinner(int x, int y);
```

checkWinner

```
static int width();
```

<u>width:Integer</u>

Meanings of ‘ private‘



What is the semantics of private, protected and public.

is it class-based (C++) or instance based (Smalltalk)?

in C++: - any public member is visible anywhere in the program

- a private member may be used only by the class that defines it
- a protected member may be used by the class that defines it or its subclasses

class based private

in Smalltalk: - instance variables are private = C++ protected

- instance based private
- methods are public

in Java class based like C++ but package rules:

- a member with package visibility may be accessed only by instances of other classes in the same package
- a protected member may be accessed by subclasses but also by any other classes in the same package as the owing class

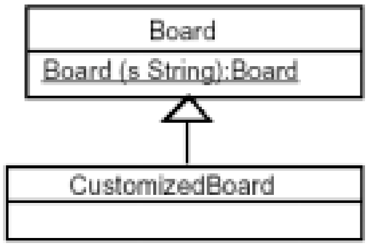
=> protected is more public than package

- classes can be marked as public or package

a package class may be used only by other classes in the same package

class method inheritance

Does it mean that CustomizedBoard can be instantiated by calling Board("Player 1")?



In Smalltalk: Yes this is normal inheritance between (meta) classes.

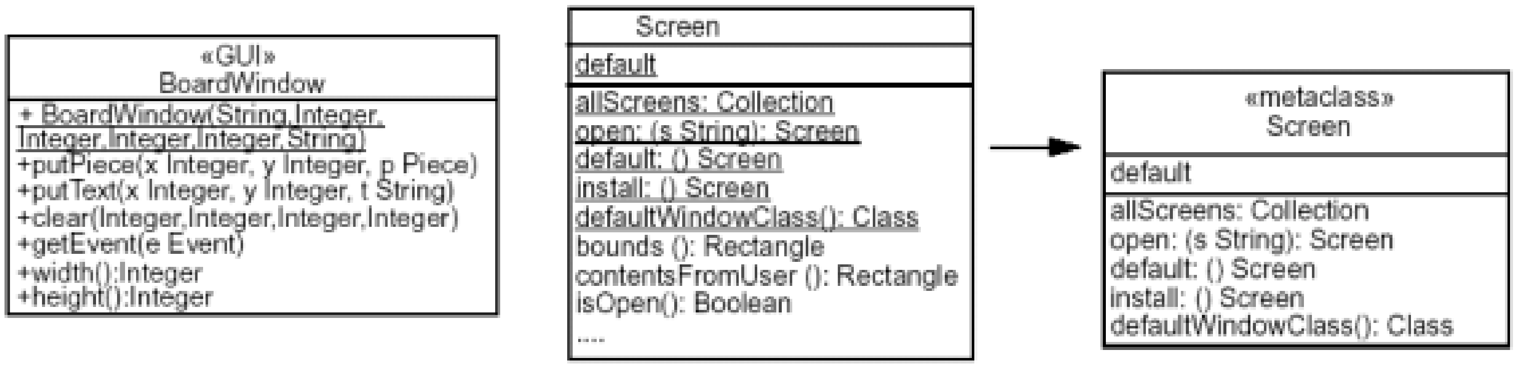
In Java: No there is no inheritance between non-default class constructor.

```
CustomizedBoard instance = new CustomizedBoard() -> Board() is called
CustomizedBoard instance = new Board("player 1") -> does not work
```

☞ Conventions needed

Stereotypes to extend UML

- Mechanism to specialize the semantics of the UML elements
- New properties are added to an element
- When a concept is missing or does not fit your needs select a close element and extend it.

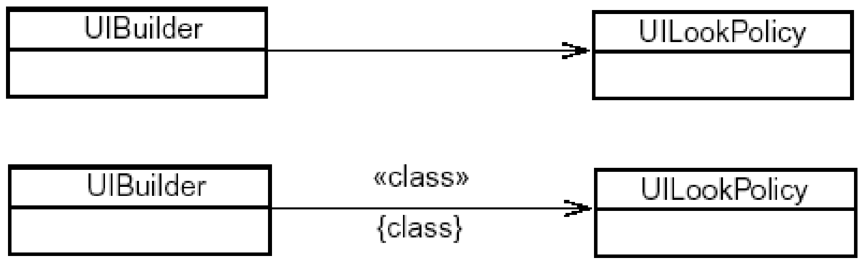


- 40 predefined stereotypes (c = class, r = relation, o = operation, a = attribute, d = dependency, g = generalization): metaclass (c), instance (r), implementation class (c) constructor (o), destructor(o), friend (d), inherits (g), interface (c), private (g), query (o), subclass (g), subtype (g), utility (classifier) (only class scope operations and attributes)
- Do not push stereotypes to the limits else you lose standard

Instance/class associations

How to distinguish between associations between classes and association between instances?

In VisualWorks, UIBuilder class is related to the UILookPolicy class



But an instance of UIBuilder is also related to an instance of UILookPolicy

☞ Use a stereotype or a constraint

Association extractions (I)

Goal: Explicit references to domain classes

- ❏ Domain Objects
  Qualifying as attributes only implementation attributes that are not related to domain objects.
  Value objects -> attributes and not associations,
  Object by references -> associations
  Ex: name: String -> an attribute
  order: Order -> an association
  myDisplay: Display -> not an association

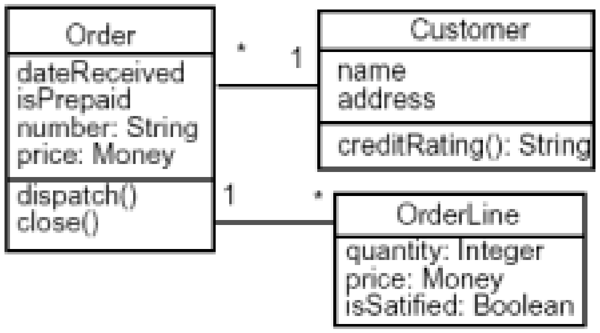
- ❏ Define your own conventions
  Ex: integer x integer -> point attribute

- ❏ Two classes possessing attributes on each other
  -> an association with navigability at both side

Association extractions (II)

- Filtering based coding conventions or visibility
  In Java, C++ filter out private attributes
  _*
- In Smalltalk depending on code practices you may filter out attributes
  - attributes
  - that have accessors and are not accessed into subclasses.
  - with name: *Cache.
  - attributes that are only used by private methods.
- If there are some coding conventions

```
class Order {
    public Customer customer(); (single value)
    public Enumerator orderLines(); (multi-values)}
```



Operation extractions (I)

You may not extract

- accessors,
- operators,
- simple instance creation methods
  (new in Smalltalk, constructor with no parameters)
- non-public methods,
- methods already defined in superclass,
- methods already defined in superclass that are not abstract, recursively
- methods that are responsible for the initialization, printing of the objects

Use company conventions to filter

- Access to database
- Calls for the UI
- Naming patterns

Operation extractions (II)

If there are several methods with more or less the same intent

- select the method with the smallest prefix

    if you want to know that the functionality exists not all the details

- select the method with the more parameters

    if you want to know all the possibilities but not all the ways you can invoke them

- categorize methods according to the number of time they are reference into clients

    but a method can be a hook method that is often called but still important

In Smalltalk, do not show

- methods that belongs to categories: ‘printing’, ‘accessing’, ‘initialize-release’, ‘private’...
- methods with name: #printOn:, #storeOn:,
- methods with the name of an attribute

What is important to show.

- Smalltalk class methods in ‘instance creation’ category,
- Constructors in Java or C++=> represent the creation interface of an object

Design patterns as documentation elements?

- Design Patterns reveal the intent so they are definitively appealing for supporting documentation [John92a] [Oden97a]

But.

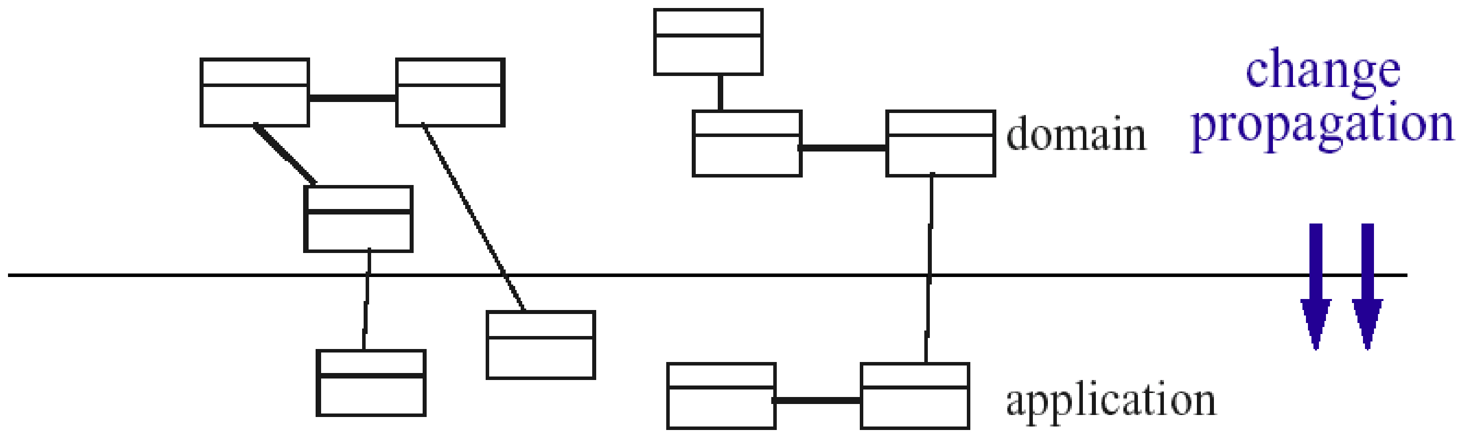
- Difficult to identify design patterns from the code
  [Brow96c, Wuyt98a, Prec98a]

  What is the difference between a facade and a mediator from the code point of view?

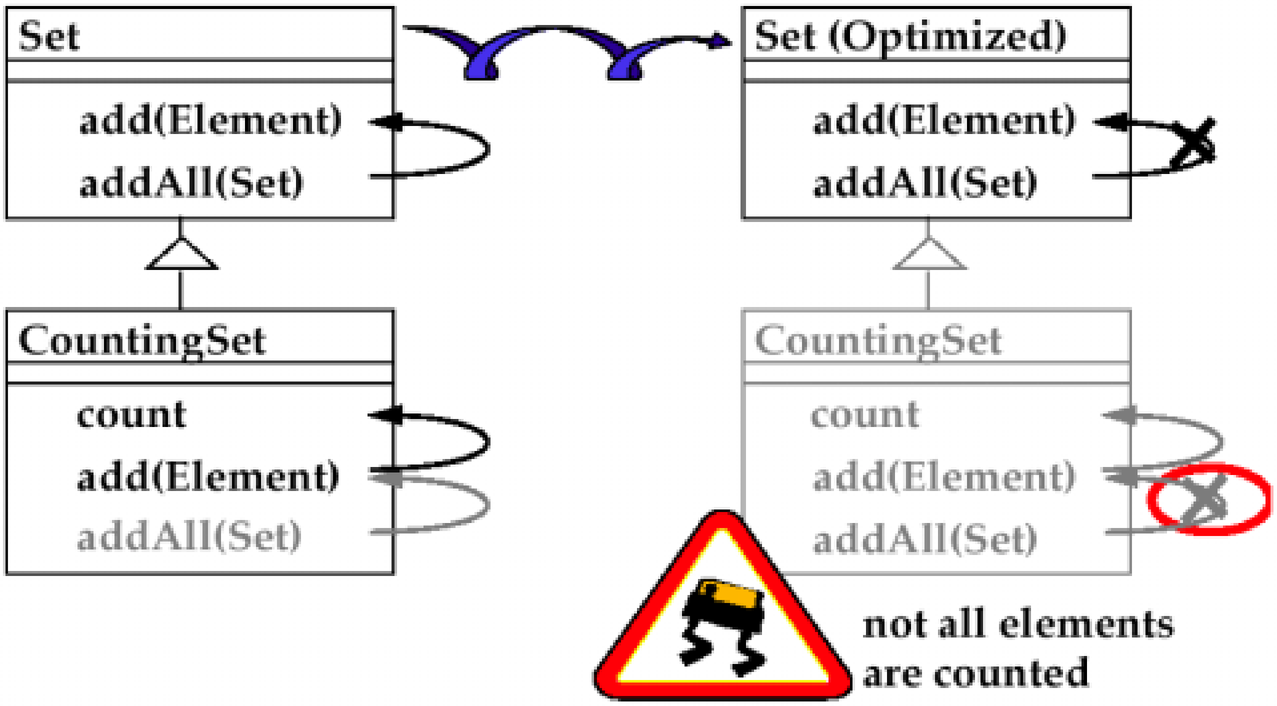
- Need somebody that knows
- Lack of support for code annotation so difficult to keep the use of patterns and the code evolution [Flor97a]

Evolution impact analysis: reuse contract

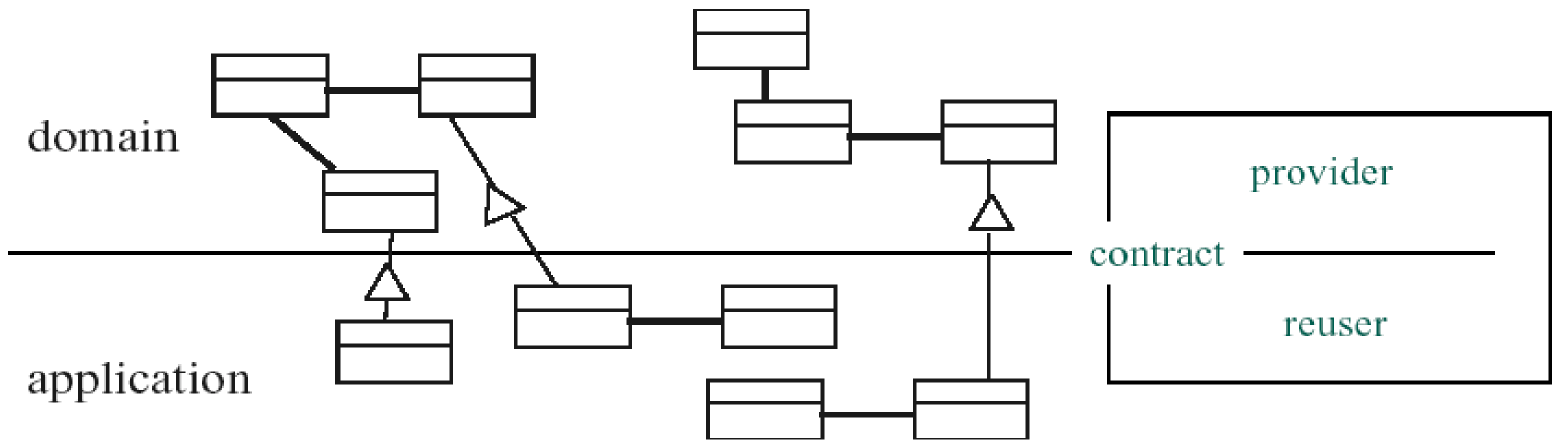
How to identify the impact of changes?



Example


Set
add(Element)
addAll(Set)
CountingSet
count
add(Element)
addAll(Set)
Set (Optimized)
add(Element)
addAll(Set)
CountingSet
count
add(Element)
addAll(Set)
not all elements
are counted

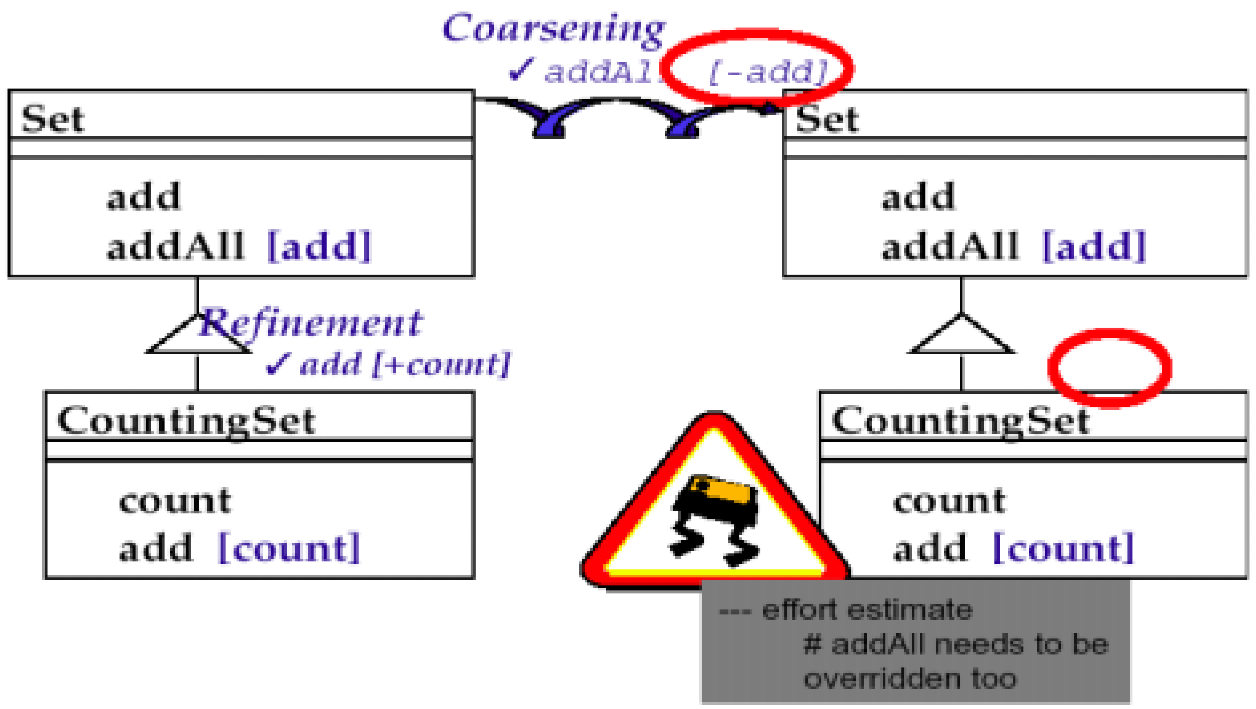
Reuse contracts—general idea



Reuse Contracts [Stey96a] propose a methodology to:

- specify and qualify extensions
- specify evolution
- detect conflicts
- Classification Browser support Reuse Contract extraction

Example



Extend UML to specify which other methods a method invokes (reuse contracts)

In class Set

+ addAll: (c Collection): Collection {invokes add}

Documenting dynamic behavior

- ❑ Focusing only at static element structural elements (class, attribute, method) is limited, does not support:
  - protocols description (message A call message B)
  - describe the role that a class may play e.g. a mediator
- ❑ Calling relationships is well suited for
  - method interrelationships
  - class interrelationships

UML proposes Interaction Diagrams = Sequence Diagram or Collaboration Diagram

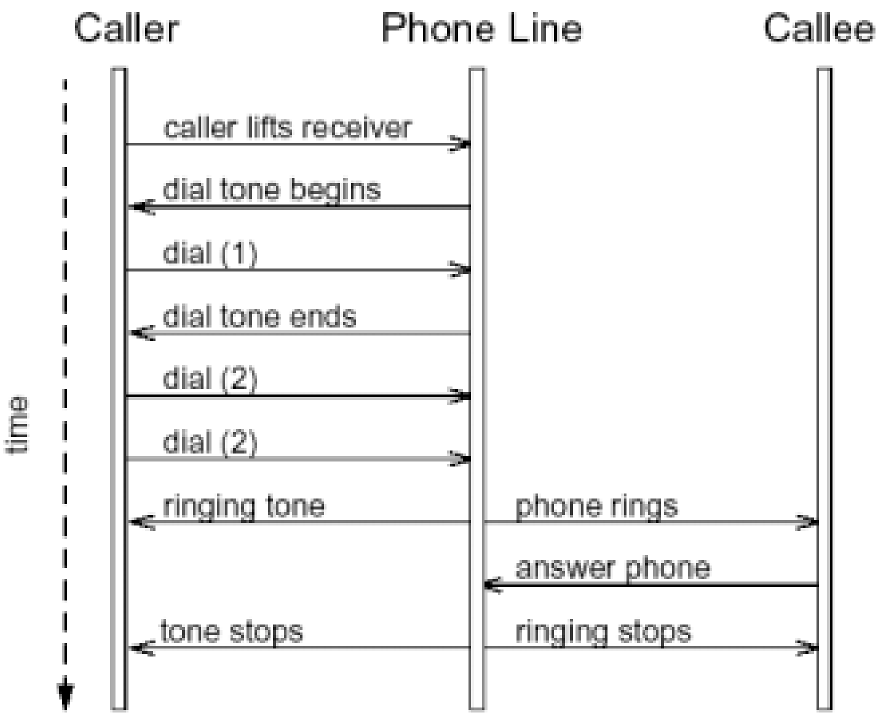
UML sequence diagrams

A *sequence diagram* depicts a scenario by showing the interactions among a set of objects in temporal order.

Objects (not classes!) are shown as vertical bars.

Events or message dispatches are shown as horizontal (or slanted) arrows from the send to the receiver.

Recall that a scenario describes a typical *example* of a use case, so conditionality is not expressed!



Implications

Statically extracting methods.

- potential not the real behaviour
- blur important effective scenario

But extracting runtime information needs.

- reflective language support (MOP, message passing control)
- code instrumentation (heavy)
- storing retrieved information (may be huge)

Amount of generated data is HUGE.

- selection of the parts of the system that should be extracted
- selection of the functionality
- selection of the use cases
- filters should be defined

(several classes as the same, several instance as the same...

☞ A simple approach could be to open a special debugger that generates specific traces

Conclusions

What we did not talk about

- ❑ Abstract Classes
- ❑ Aggregations and composition extraction [Wins87a]
- ❑ Qualified associations Lessons Learnt

You should be clear about:

- ❑ Your goal (detailed or architectural design)
- ❑ Conventions like navigability,
- ❑ Language mapping based on stereotypes
- ❑ Level of interpretations

For Future Development

- ❑ Emphasize literate programming approach
- ❑ Extract design to keep it synchronized

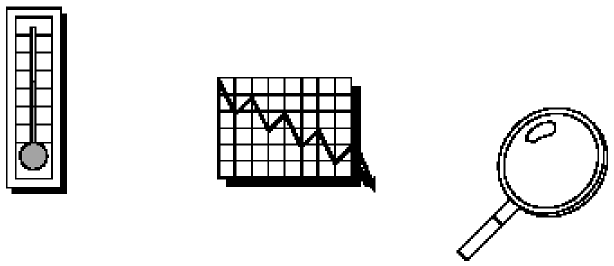
UML as Support for Design Extraction

- ❑ Often fuzzy
- ❑ Composition aggregation limited
- ❑ Do not support well reflexive models
- ❑ But UML is extensible, define your own stereotype

Metrics for OO reengineering

Outline

- ❑ Why Metrics in OO Reengineering?
- ❑ Which Metrics to Collect?
  - Goal-Question-Metric paradigm
  - Metric Definitions
- ❑ Applicability for...
  - Problem Detection
  - Stability Assessment
  - Reverse Engineering
- ❑ Conclusion



Why metrics in OO reengineering?

Cost Estimation

- ❑ What's the effect of reuse?
- ❑ Is it worthwhile to reengineer, or is it better to start from scratch?

=> Not covered ☹

Software Quality Evaluation

- ❑ Which parts have bad quality? (Hence, should be reengineered first)
- ❑ Which parts have good quality?

=> Metrics as a project management tool

Iterative Development

- ❑ Can I use metrics to measure changes?
- ❑ Can I use change metrics to reverse engineer design?

=> Metrics as a reverse engineering tool

SCALABILITY

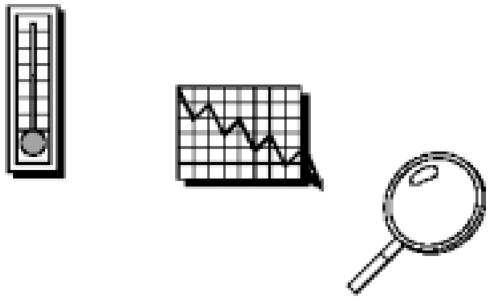
Which metrics to collect?

Goal

- ❑ Support reverse and reengineering of object-oriented programs

Question

1. Which parts of the design will cause problems with future extensions?
2. Which parts of the design are unstable?
3. Which parts of the design have been refactored?



Metric

- ❑ Low overhead for developer
- ❑ Take advantage of OO structure
- ❑ Exploit presence of different releases

=> Collect from source code

Assumptions

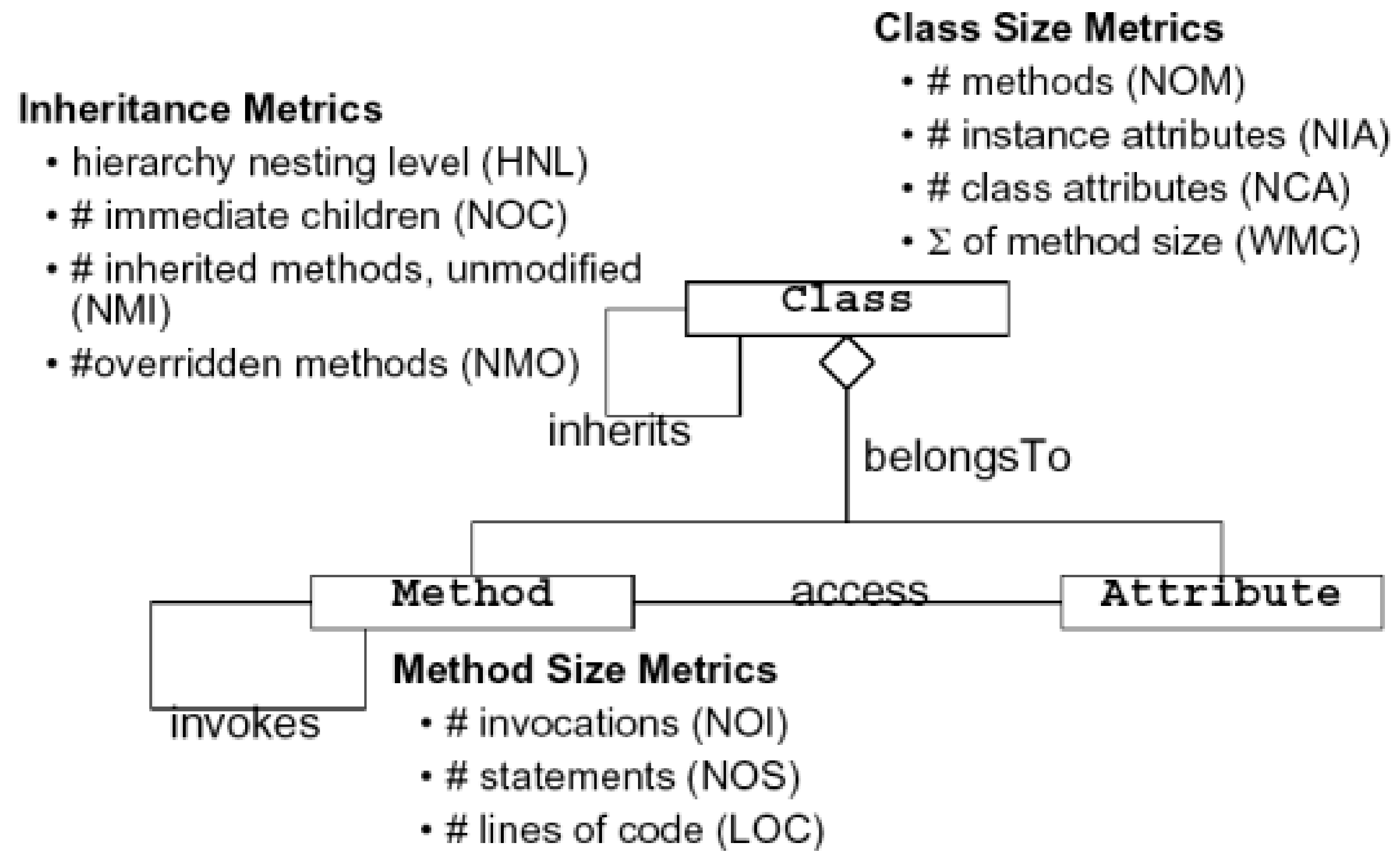
Question - Assumptions

1. Which parts of the design will cause problems with future extensions?
   - ❑ Large methods & classes
   - ❑ Classes with big impact on the inheritance hierarchy
   - ❑ Classes influenced a lot via the inheritance hierarchy

2. Which parts of the design are unstable?
   - ❑ Methods and classes that change in size
   - ❑ Places where the inheritance hierarchy is changed

3. Which parts of the design have been refactored?
   - ❑ Methods that decrease in size have been *split*
   - ❑ Classes that change in size have their attributes and methods *redistributed*
   - ❑ Changes in the inheritance relationship are symptoms for *optimization of the class hierarchy*

Definitions



Class Size Metrics

- # methods (NOM)
- # instance attributes (NIA)
- # class attributes (NCA)
- Σ of method size (WMC)

Inheritance Metrics

- hierarchy nesting level (HNL)
- # immediate children (NOC)
- # inherited methods, unmodified (NMI)
- #overridden methods (NMO)

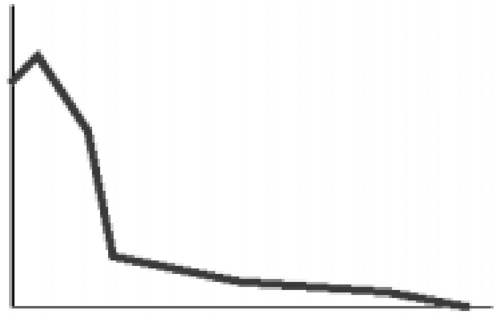
Method Size Metrics

- # invocations (NOI)
- # statements (NOS)
- # lines of code (LOC)

Results: problem detection

- between 2/3 and 1/2 of detected problems are left unchanged in subsequent release
- considerable amount of detected problems measure worse in subsequent release

=> unreliable as problem detection tool



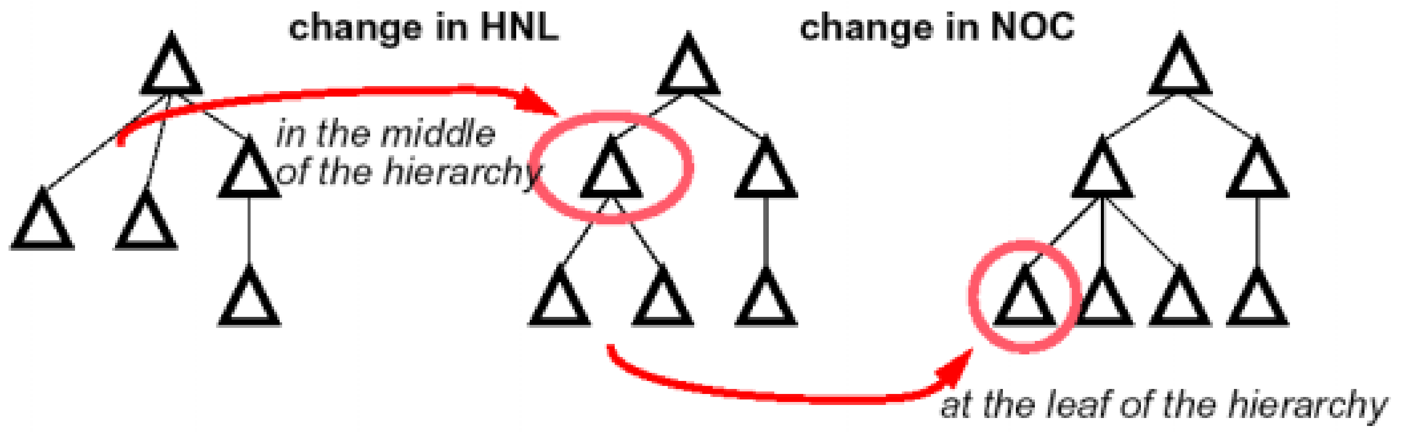
=> 80%-20% distribution as a litmus test

Results: stability assessment

- changes may go unnoticed
  => false negatives are possible

- all detected changes are real
  => no false positives (but lot of noise)

Sometimes the kind of instability is revealing!



Results: reverse engineering

- vulnerable to renaming
- imprecise for many changes
- requires experience
- considerable resources

=> inherent to source code extraction

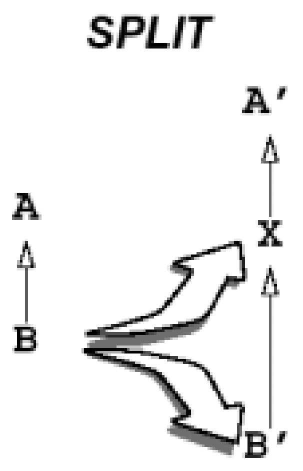
- good focus (scaleability)
- reliable
- provides road map (best focus first)
- reveals class interaction
- unbiased

=> good in the early stages

Split into superclass/merge with superclass

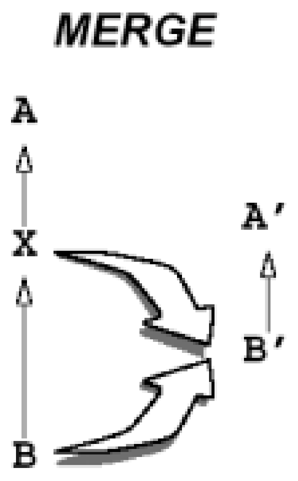
Recipe

- Use change in "Hierarchy Nesting Level" (HNL) as main indicator
- Complement with changes in "# methods" (NOM), "# instance attributes" (NIA) and "# class attributes" (NCA) to look for push-up, push down of functionality
- Include changes in "# inherited methods" (NMI) and "# overridden methods" (NMI) to assess overall protocol



Split B into X and B'
(delta_HNL(B') > 0) and
    ((delta_NOM(B') < 0)
    or   (delta_NIA(B') < 0)
    or   (delta_NCA(B') < 0))

Merge X and B into B'
(delta_HNL(B') < 0) and
    ((delta_NOM(B') >0)
    or   (delta_NIA(B') > 0)
    or   (delta_NCA(B') > 0))

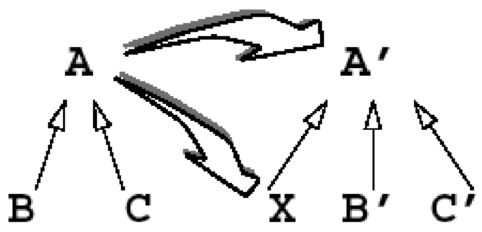


Split into subclass/merge with subclass

Recipe

- ❑ Use change in “# immediate children” (NOC) as main indicator
- ❑ Complement with changes in “# methods” (NOM), “# instance attributes” (NIA) and “# class attributes” (NCA) to look for push-up, push down of functionality

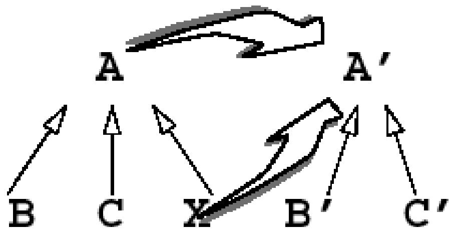
SPLIT

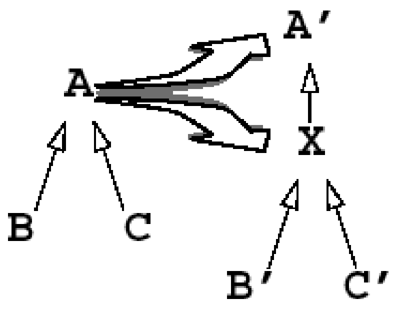


Split A into X and A’

(delta_NOC(A’) <> 0) and
    ((delta_NOM(A’) < 0)
    or   (delta_NIA(A’) < 0)
    or   (delta_NCA(A’) < 0))

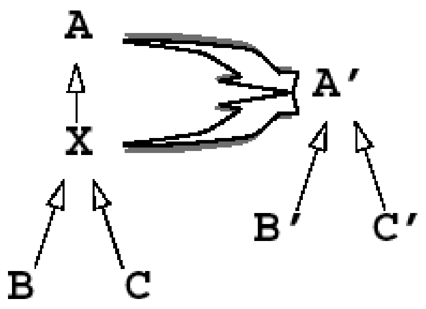
MERGE





Merge X and A into A’

(delta_NOC(A’) <> 0) and
    ((delta_NOM(A’) >0)
    or   (delta_NIA(A’) > 0)
    or   (delta_NCA(A’) > 0))

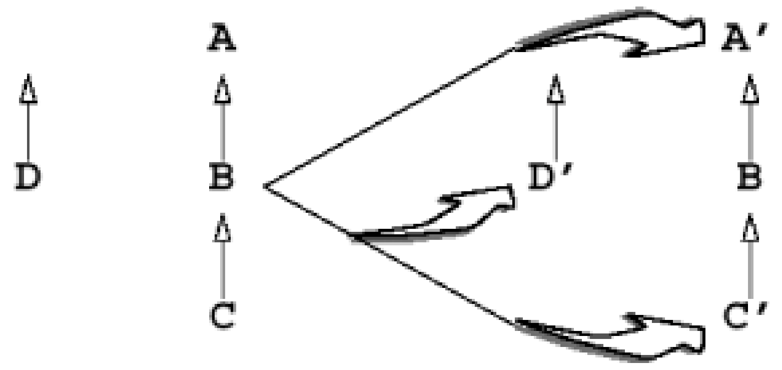


Move to superclass/subclass or sibling class

Recipe

- ❑ Use decreases in "# methods" (NOM), "# instance attributes" (NIA) and "# class attributes" (NCA) as main indicator
- ❑ Select only the cases where "# immediate children" (NOC) and "Hierarchy Nesting Level" (HNL) remains equal

MOVE



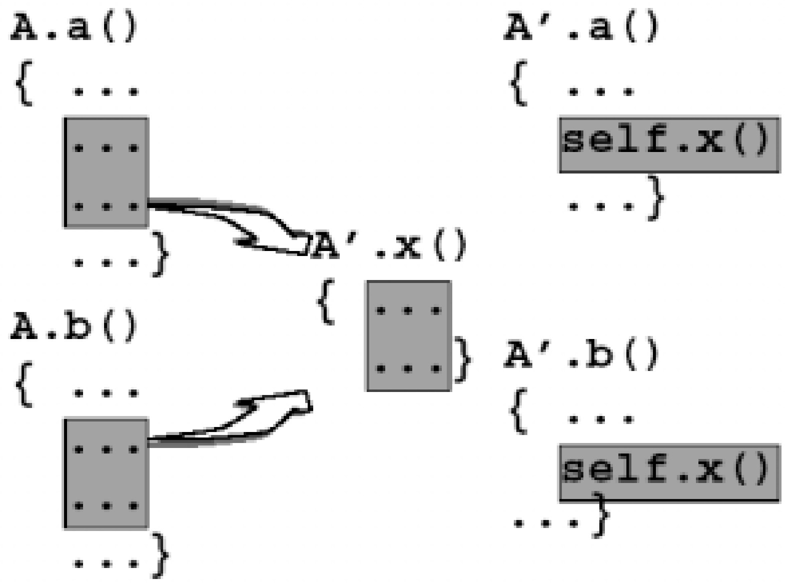
Move from B to A', C' or D'

((delta_NOM(B') < 0)
or (delta_NIA(B') < 0)
or (delta_NCA(B') < 0))
and (delta_HNL(B') = 0)
and (delta_NOC(B') = 0)

Split method/factor common functionality

Recipe

- ❑ Use decreases in "# invocations" (NOI) as main indicator
- ❑ Combine with "# statements" (NOS) and "# Lines of Code" (LOC)
- ❑ Check similar decreases in other methods defined on the same class



Split part of A.a() in A'.x()

(delta_NOI(A'.a()) < 0)

Factor out part of A.a() and A.b() into A'.x()

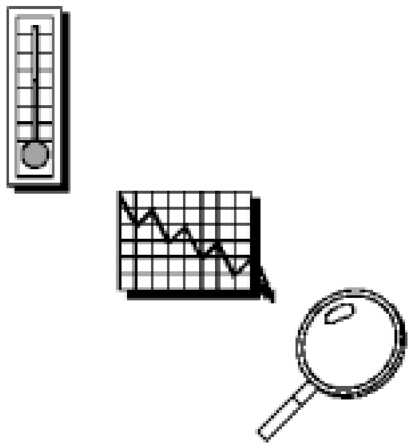
(delta_NOI(A'.a()) < 0)
and (delta_NOI(A'.b()) < 0)
and (delta_NOI(A'.a())
= delta_NOI(A'.b()))

Conclusions

Question

Can metrics help to answer the following questions?

1.	Which parts of the design will cause problems with future extensions?	Not reliably
2.	Which parts of the design are unstable?	Yes
3.	Which parts of the design have been refactored?	Yes



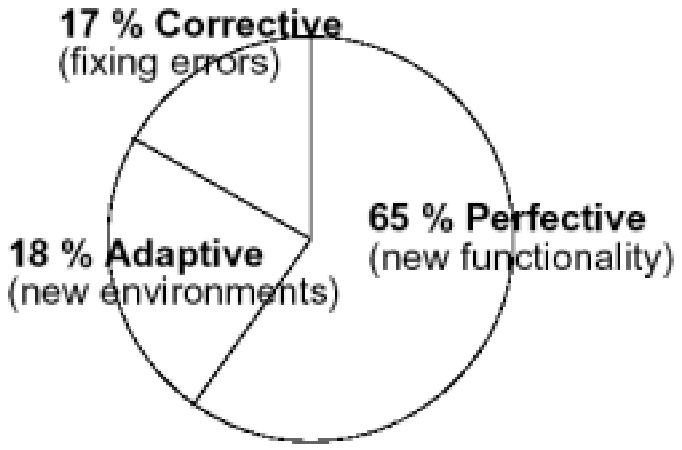
Refactoring

Outline

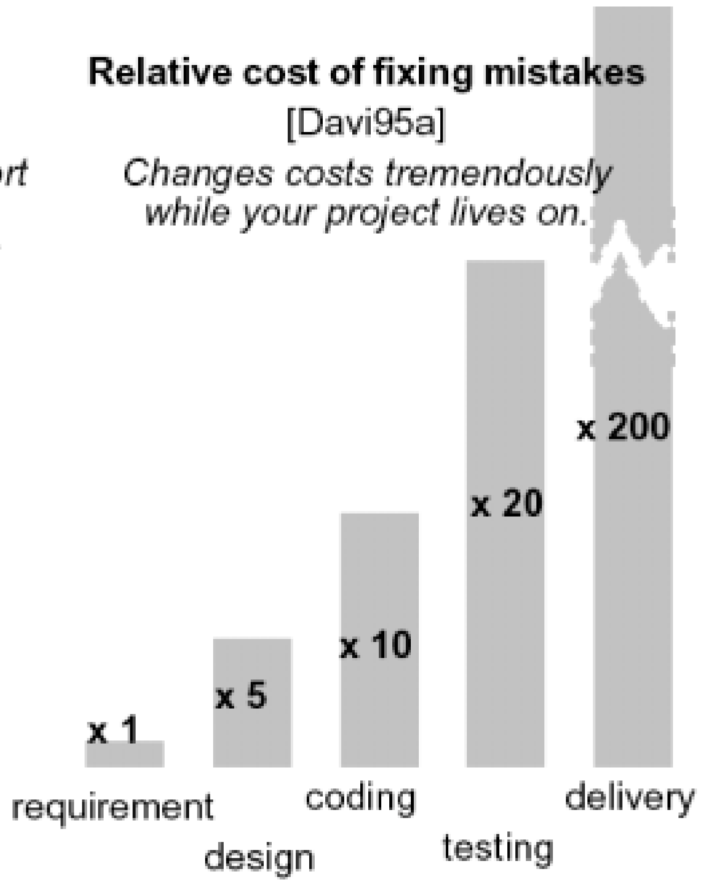
- ❑ Why Refactoring?
- ❑ Iterative Development Life-cycle
- ❑ What is Refactoring?
- ❑ Which Refactoring Tools?
- ❑ Case-study: Internet Banking
  - prototype
  - consolidation: design review
  - expansion: concurrent access
  - consolidation: more reuse
- ❑ Conclusion

Why refactoring?

Relative Effort of Maintenance
[Somm96a]
Between 50% and 75% of available effort is spent on maintenance. 65% of that concerns new functionality, which you could not foresee when you started.

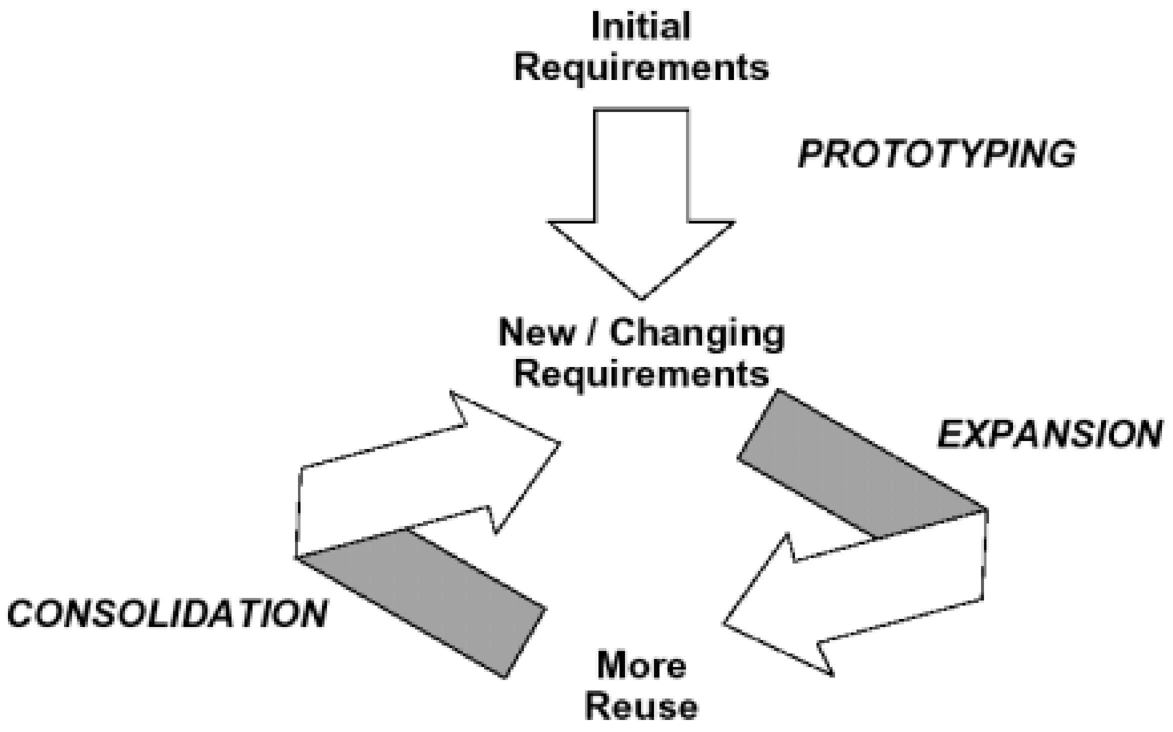


Relative cost of fixing mistakes
[Davi95a]
Changes costs tremendously while your project lives on.

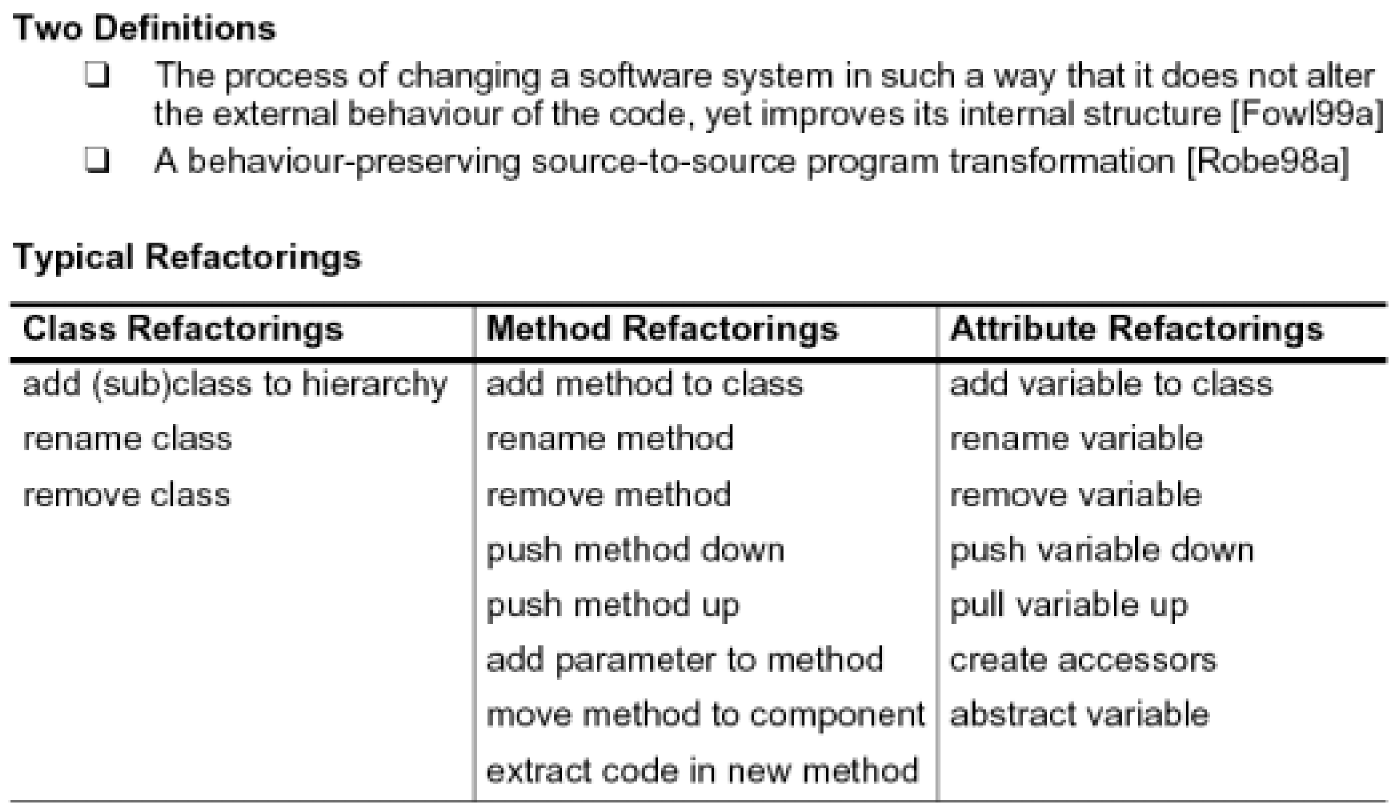


Iterative development life cycle

Change is the norm, not the exception !



What is refactoring?



Two Definitions

- ❑ 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]

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	

Which refactoring tools?

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

=> *verify if improved structure does not damage previous work*

Configuration & Version Management

- ❑ keep track of versions that represent project milestones

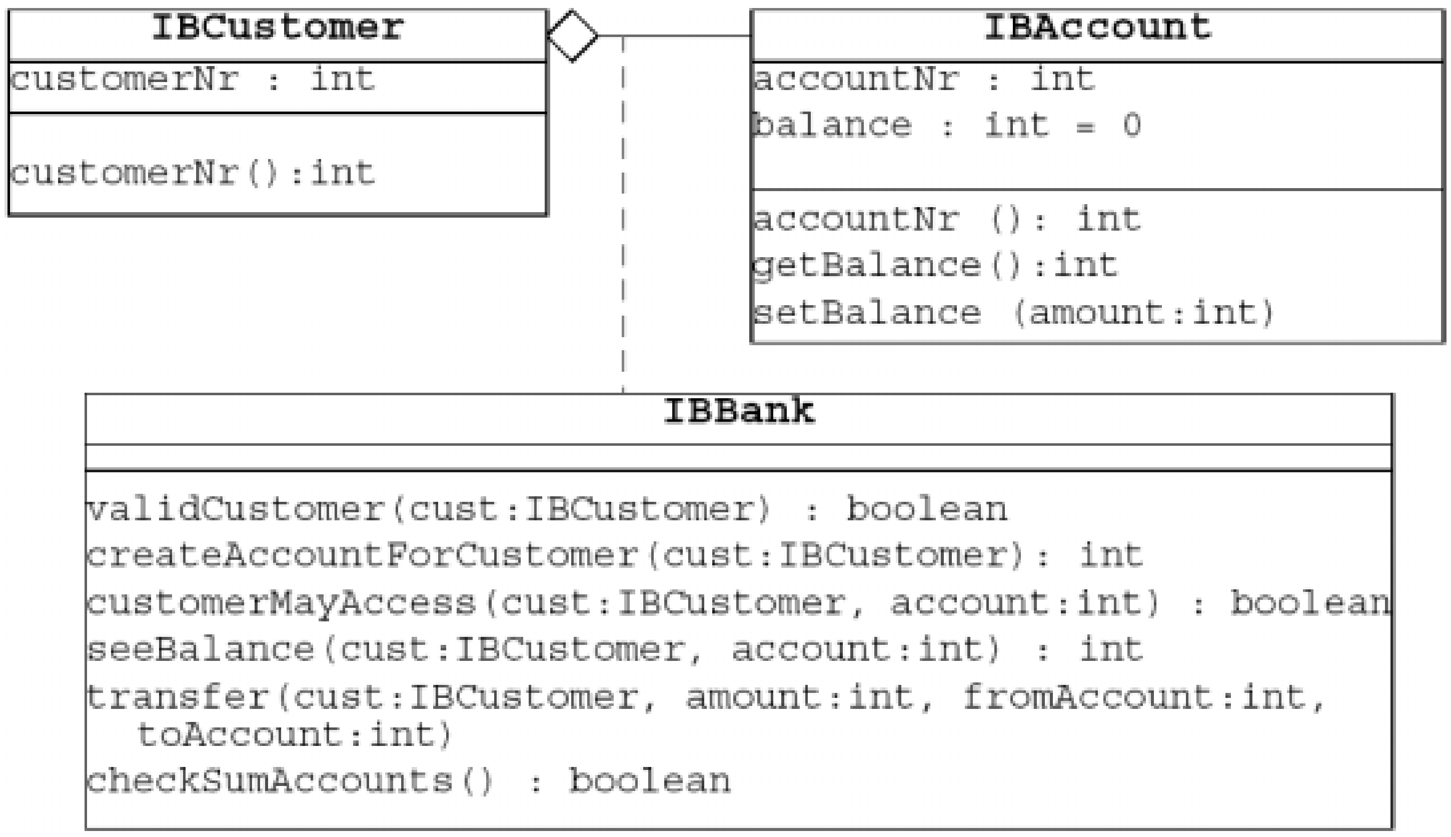
=> *possibility to go back to previous version*

Case study: Internet banking initial 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

- *secure:* only authorised users may access an account
- *reliable:* all transactions must maintain consistent state

Prototype design


IBCustomer
customerNr : int
customerNr():int
IBAccount
accountNr : int
balance : int = 0
accountNr (): int
getBalance():int
setBalance (amount:int)
IBBank
validCustomer(cust:IBCustomer) : boolean
createAccountForCustomer(cust:IBCustomer): int
customerMayAccess(cust:IBCustomer, account:int) : boolean
seeBalance(cust:IBCustomer, account:int) : int
transfer(cust:IBCustomer, amount:int, fromAccount:int, toAccount:int)
checkSumAccounts() : boolean

Prototype design of contracts

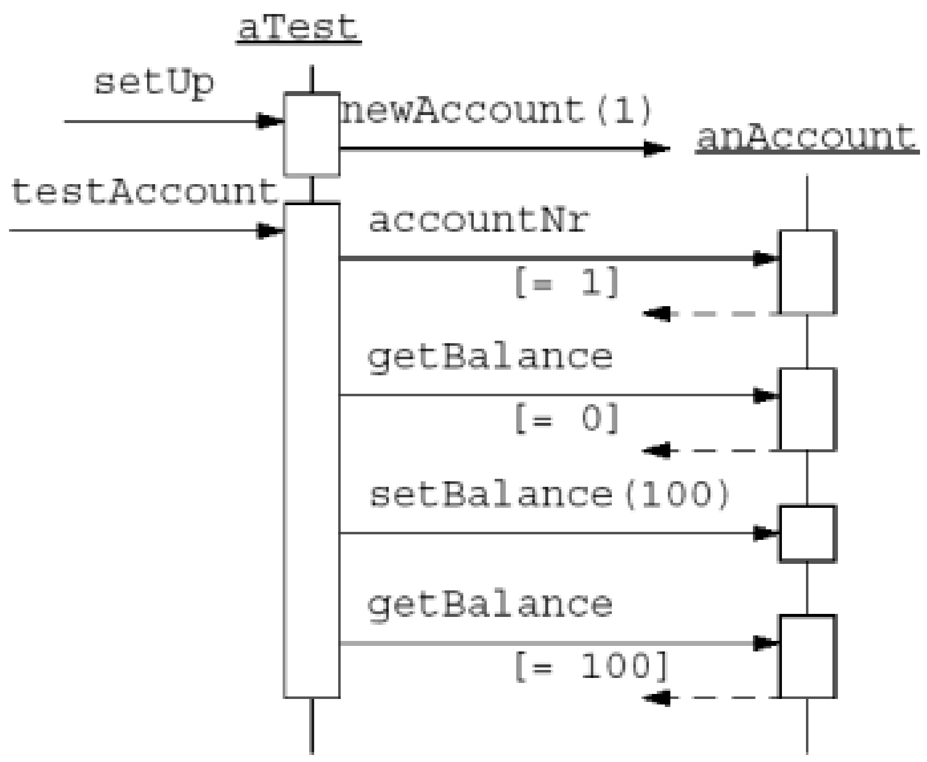
Ensure the “*secure*” and “*reliable*” requirements.

```
IBBank::createAccountForCustomer(cust:IBCustomer): int
      require: validCustomer(cust)
      ensure: customerMayAccess(cust, <<result>>)

IBBank::seeBalance(cust:IBCustomer, account:int) : int
      require: (validCustomer(cust)) AND
            (customerMayAccess(cust, account))
      ensure: checkSumAccounts()

IBBank::transfer(cust:IBCustomer, amount:int, fromAccount:int,
  toAccount:int)
      require: (validCustomer(cust))
            AND (customerMayAccess(cust, fromAccount))
            AND (customerMayAccess(cust, toAccount))
      ensure: checkSumAccounts()
```

Prototype implementation

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!)

Expansion

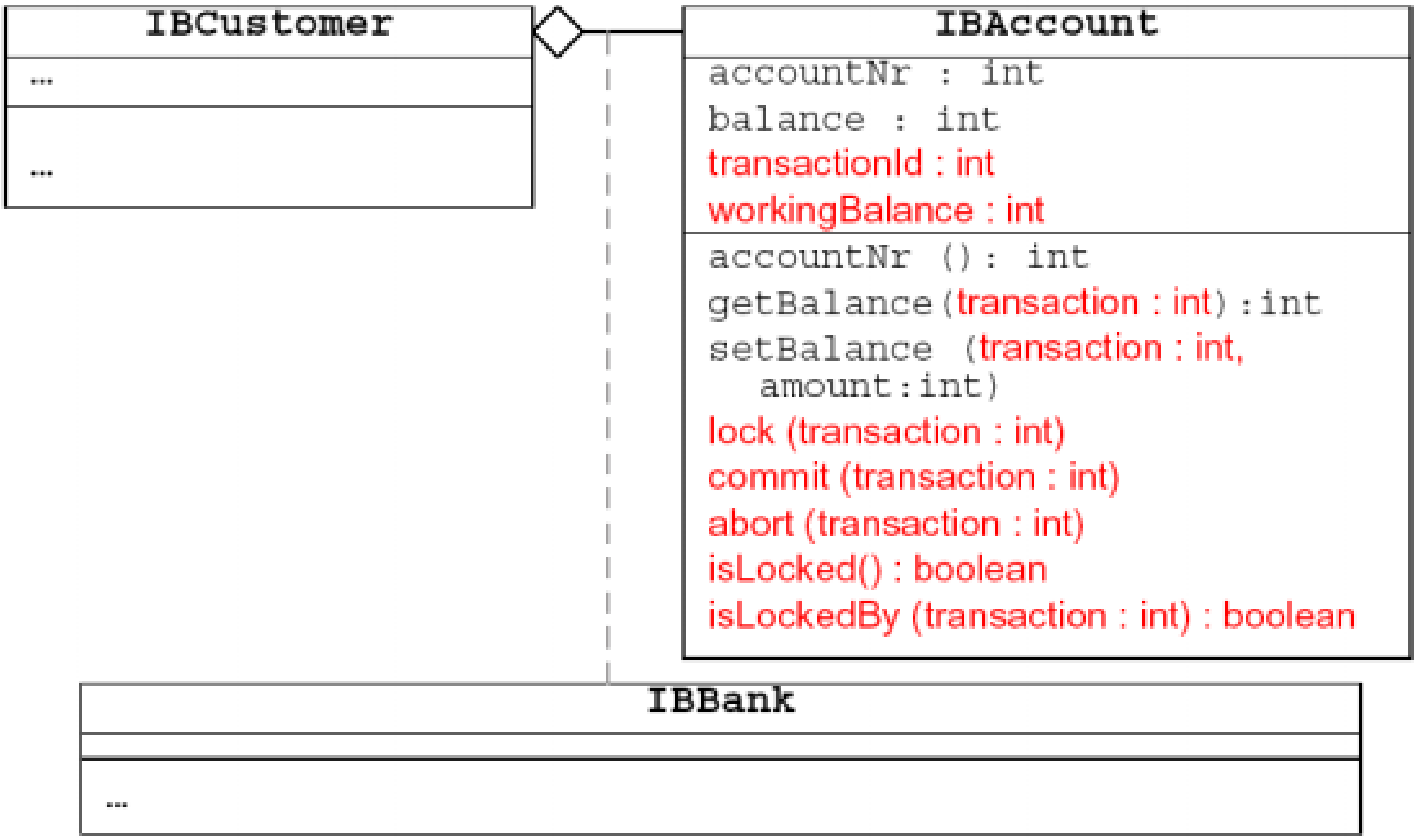
Additional Requirement

- ❑ *concurrent access* of accounts

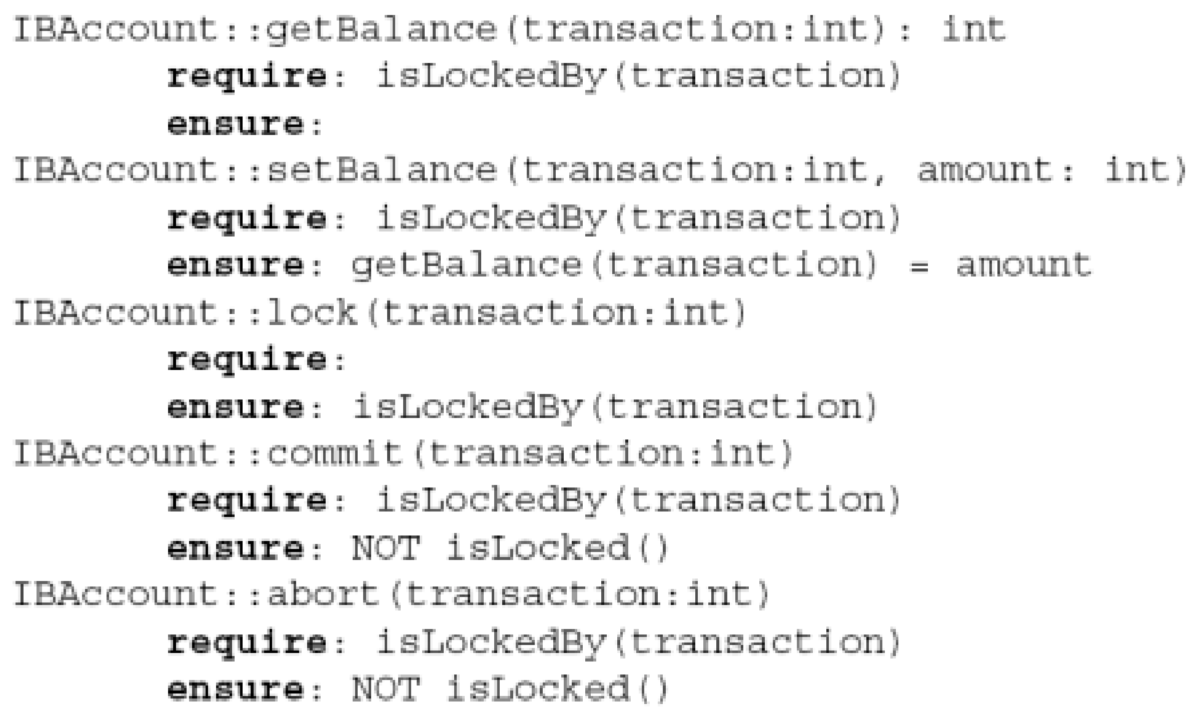
Add test case for

- ❑ IBBank
  - testConcurrent: Launches 2 processes that simultaneously transfer money between same accounts
    => test fails!

Expanded design


IBCustomer
...
...
IBAccount
accountNr : int
balance : int
transactionId : int
workingBalance : int
accountNr (): int
getBalance(transaction : int):int
setBalance (transaction : int, amount:int)
lock (transaction : int)
commit (transaction : int)
abort (transaction : int)
isLocked() : boolean
isLockedBy (transaction : int) : boolean
IBBank
...

Expanded design: contracts



```
IBAccount::getBalance(transaction:int): int
        require: isLockedBy(transaction)
        ensure:
IBAccount::setBalance(transaction:int, amount: int)
        require: isLockedBy(transaction)
        ensure: getBalance(transaction) = amount
IBAccount::lock(transaction:int)
        require:
        ensure: isLockedBy(transaction)
IBAccount::commit(transaction:int)
        require: isLockedBy(transaction)
        ensure: NOT isLocked()
IBAccount::abort(transaction:int)
        require: isLockedBy(transaction)
        ensure: NOT isLocked()
```

Expanded implementation

Adapt implementation

- ❑ apply “add attribute” on IBAccount with “transactionId” and “workingBalance”
- ❑ apply “add parameter” to “getBalance()” and “setBalance()” with “transaction”
- ❑ use normal editing to expand functionality of “seeBalance()” and “transfer()”
  => load “IBanking2”

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

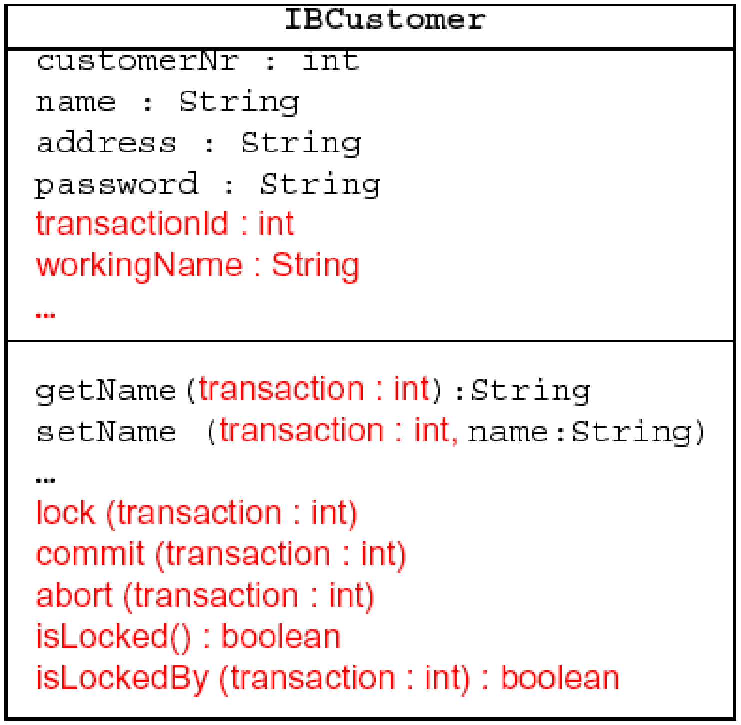
- ❑ A design review reveals that this “transaction” stuff is a good idea and should be applied to IBCustomer as well.

=> Code Smells

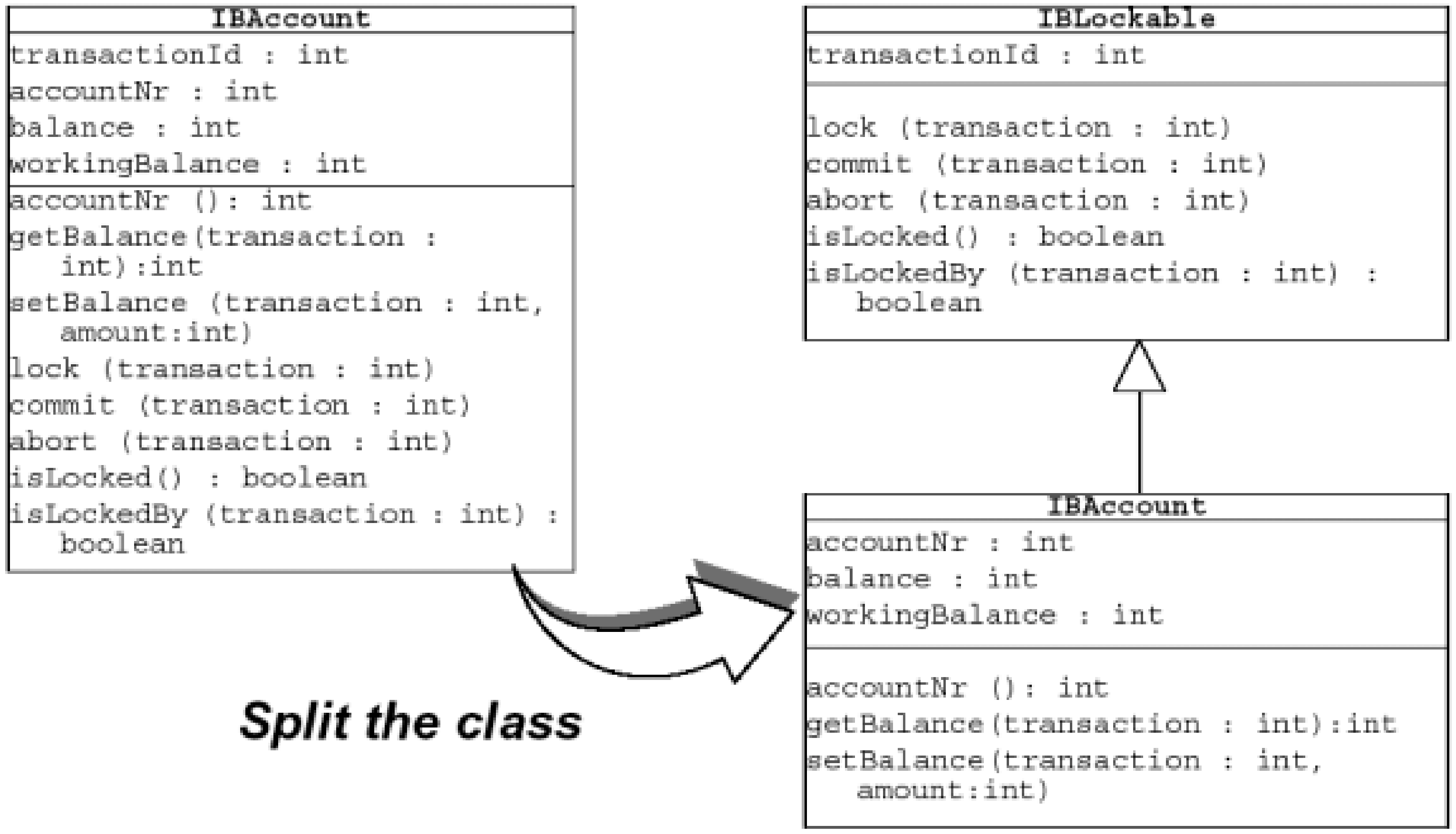
- ❑ duplicated code (lock, commit, abort + transactionId)
- ❑ large classes (extra methods, extra attributes)

=> Refactor

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


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

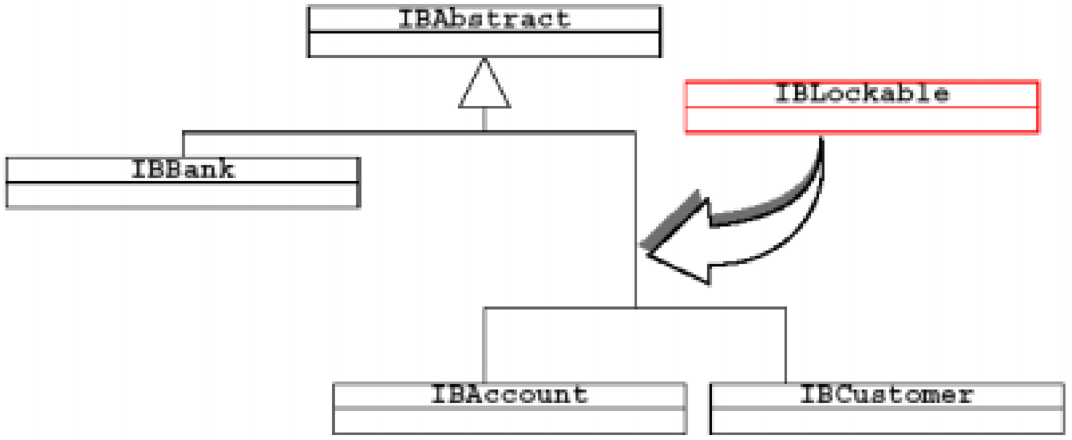
Consolidation: refactored class diagram


IBAccount
transactionId : int
accountNr : int
balance : int
workingBalance : int
accountNr (): int
getBalance(transaction : int):int
setBalance (transaction : int, amount:int)
lock (transaction : int)
commit (transaction : int)
abort (transaction : int)
isLocked() : boolean
isLockedBy (transaction : int) : boolean
IBLockable
transactionId : int
lock (transaction : int)
commit (transaction : int)
abort (transaction : int)
isLocked() : boolean
isLockedBy (transaction : int) : boolean
IBAccount
accountNr : int
balance : int
workingBalance : int
accountNr (): int
getBalance(transaction : int):int
setBalance(transaction : int, amount:int)
Split the class

Refactoring sequence (I)

Refactoring: Create Subclass

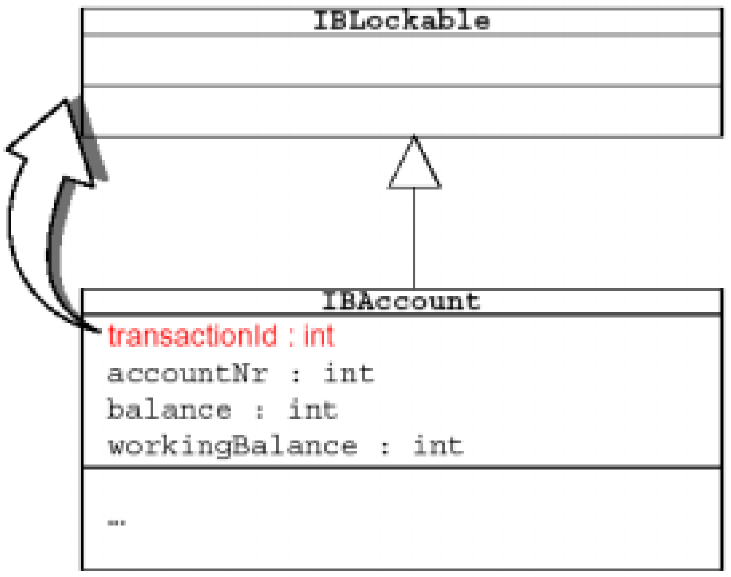
- apply "Create Subclass" on "IBAbstract" to create an empty "IBLockable" with subclass(es) "IBAccount" & "IBCustomer"



Refactoring sequence (II)

Refactoring: Move Attribute

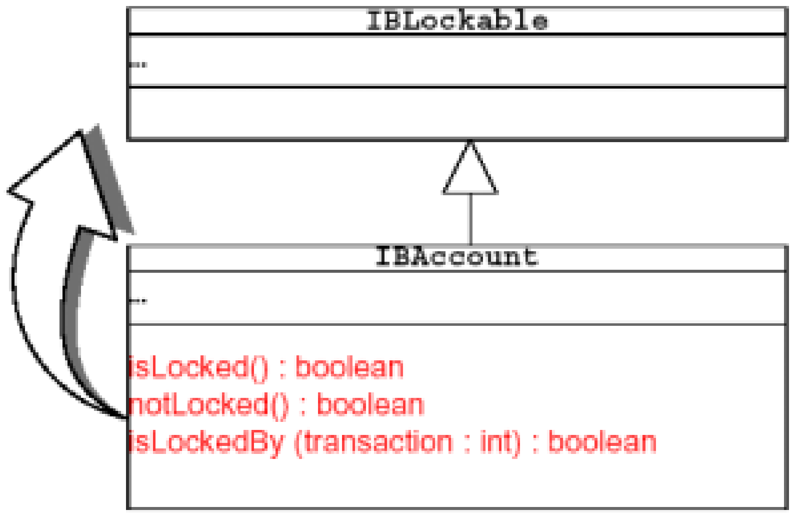
- apply "pull up attribute" on "IBLockable" to move "transactionId" up



Refactoring sequence (III)

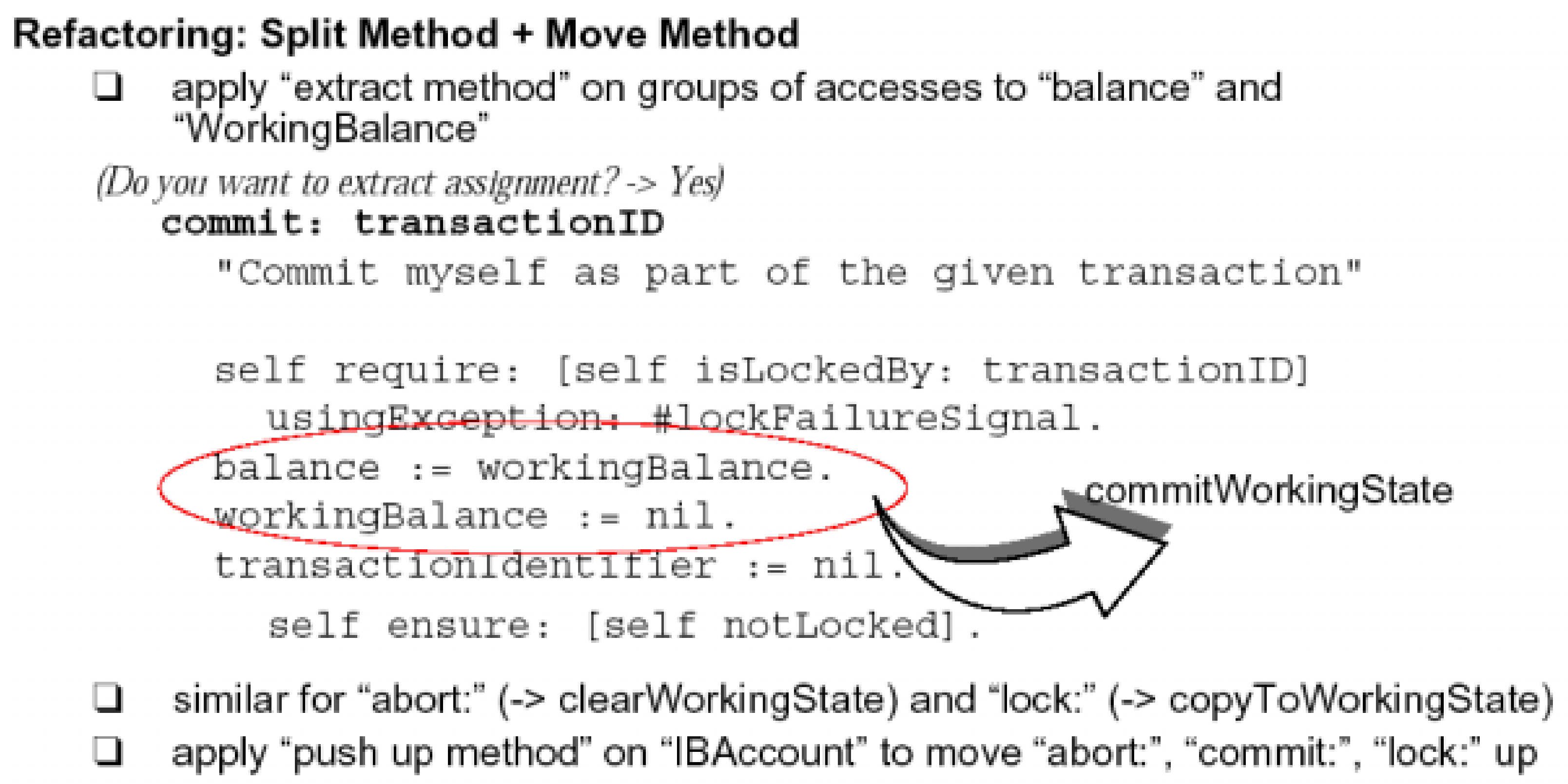
Refactoring: Move Method

- apply “push up method” on “IBAccount” to move “isLocked”, “isLockedBy”, “notLocked” up



- apply “push up” to “abort:”, “commit:”, “lock:”
  => failure: accesses to “balance” and “workingBalance” attributes

Refactoring sequence (IV)



Refactoring: Split Method + Move Method

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

(Do you want to extract assignment? -> Yes)

```
commit: transactionID
  "Commit myself as part of the given transaction"

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

commitWorkingState

- ❑ similar for "abort:" (-> clearWorkingState) and "lock:" (-> copyToWorkingState)
- ❑ apply "push up method" on "IBAccount" to move "abort:", "commit:", "lock:" up

Refactoring sequence (V)

Clean-up: make the extracted methods protected and define them as new abstract methods in the IBLocking class

- ❑ Apply "rename protocol" on "IBAccount" to rename "public-locking" into "protected-locking"

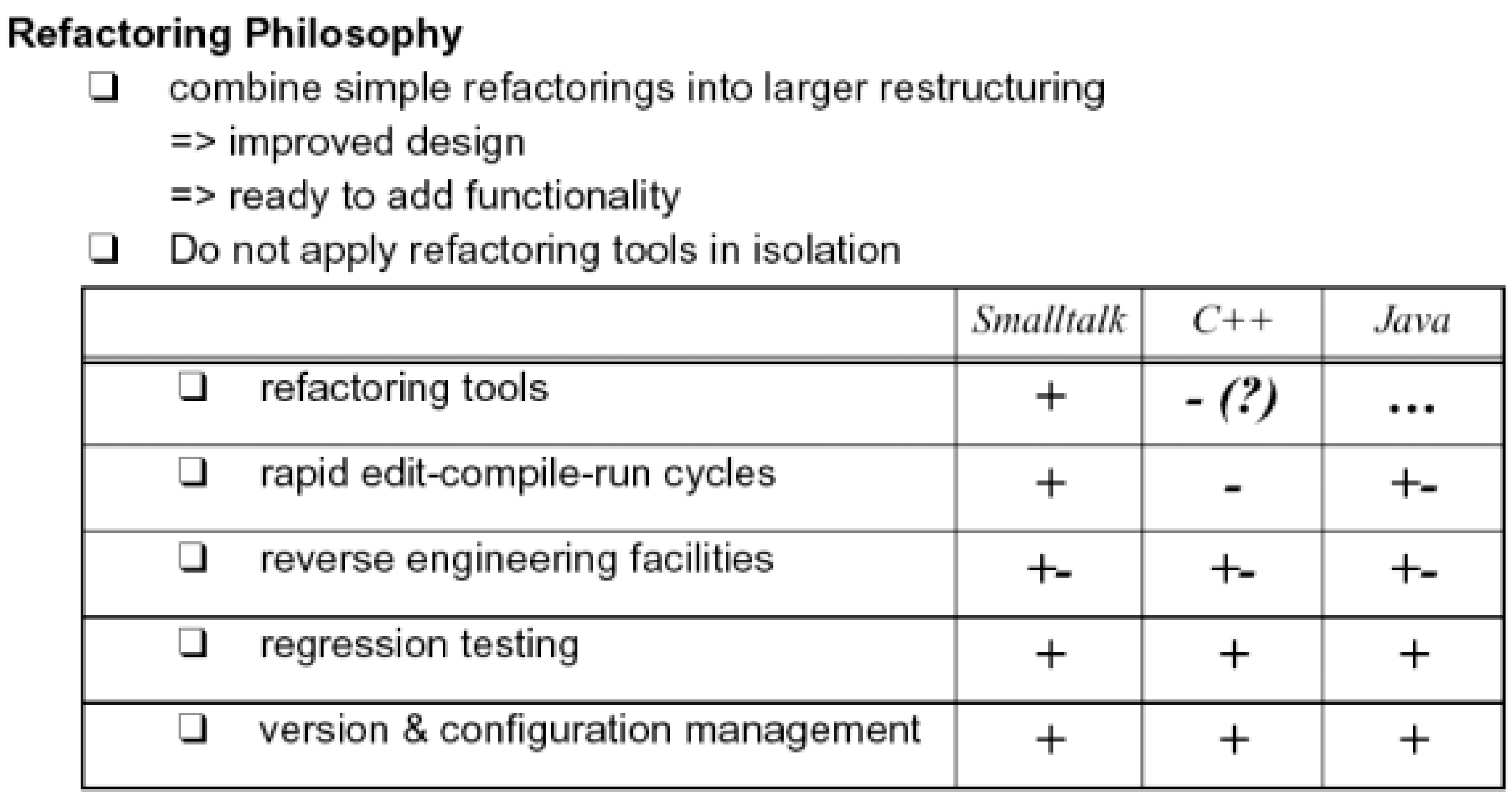
Refactoring: Copy Method

- ❑ Apply "move method" on "IBAccount" to copy "clearWorkingState", "copyToWorkingState", "commitWorkingState" to "IBLockable>protected-locking"
- ❑ Make "IBLockable::clearWorkingState", … abstract
  - ☞ This is destructive editing and not a refactoring

Are we done?

- ❑ Run the tests …
- ❑ Expand functionality of the IBCustomer

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	+	+	+

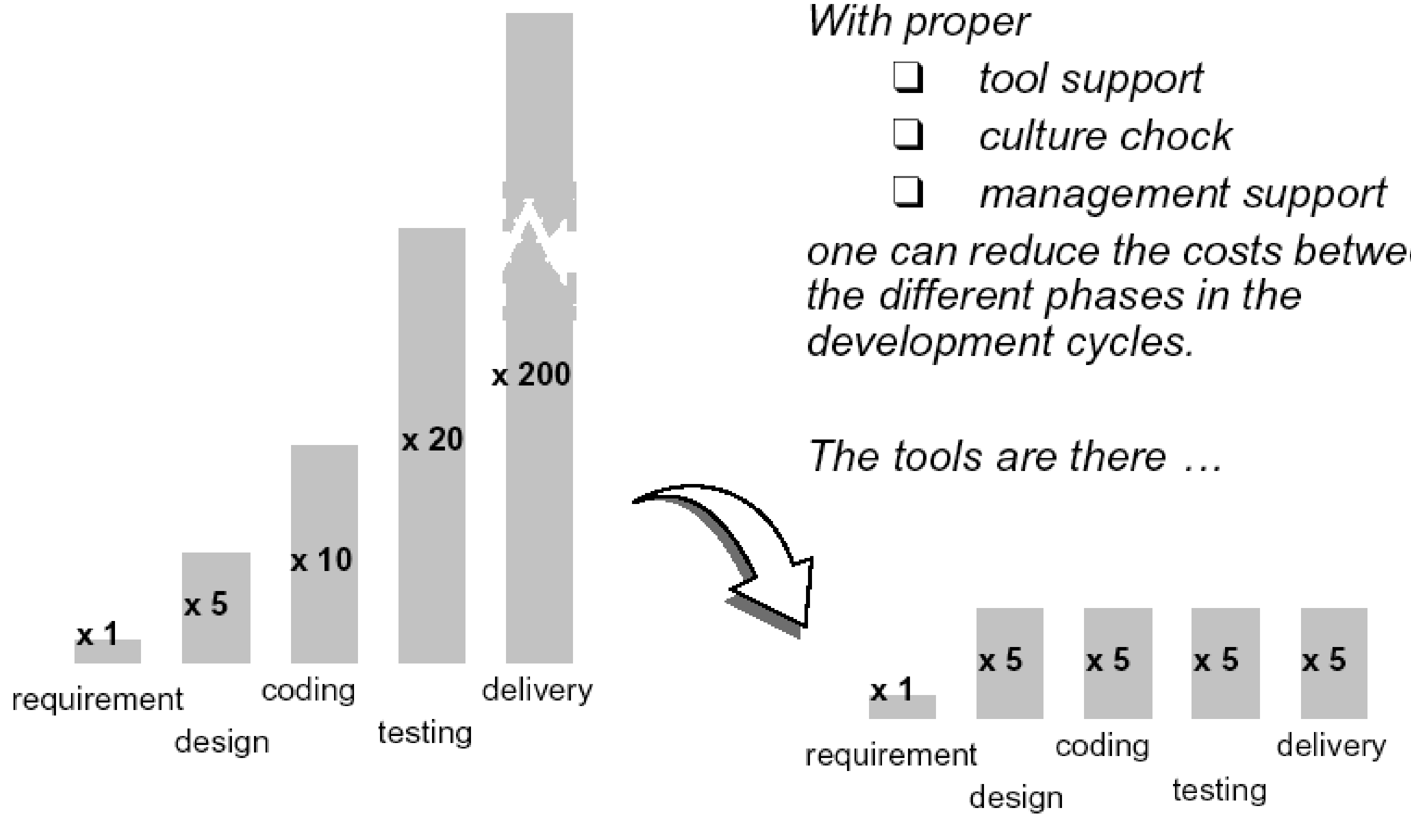
Conclusion (I)

Know when is as important as know-how

- ❑ Refactored designs are more complex
- ❑ Use "code smells" as symptoms
- ❑ Rule of the thumb: State everything "Once and Only Once" (Kent Beck)
  => a thing stated more than once implies refactoring

- ❑ Wiki-web
  `http://c2.com/cgi/wiki`

Conclusion (II)



With proper

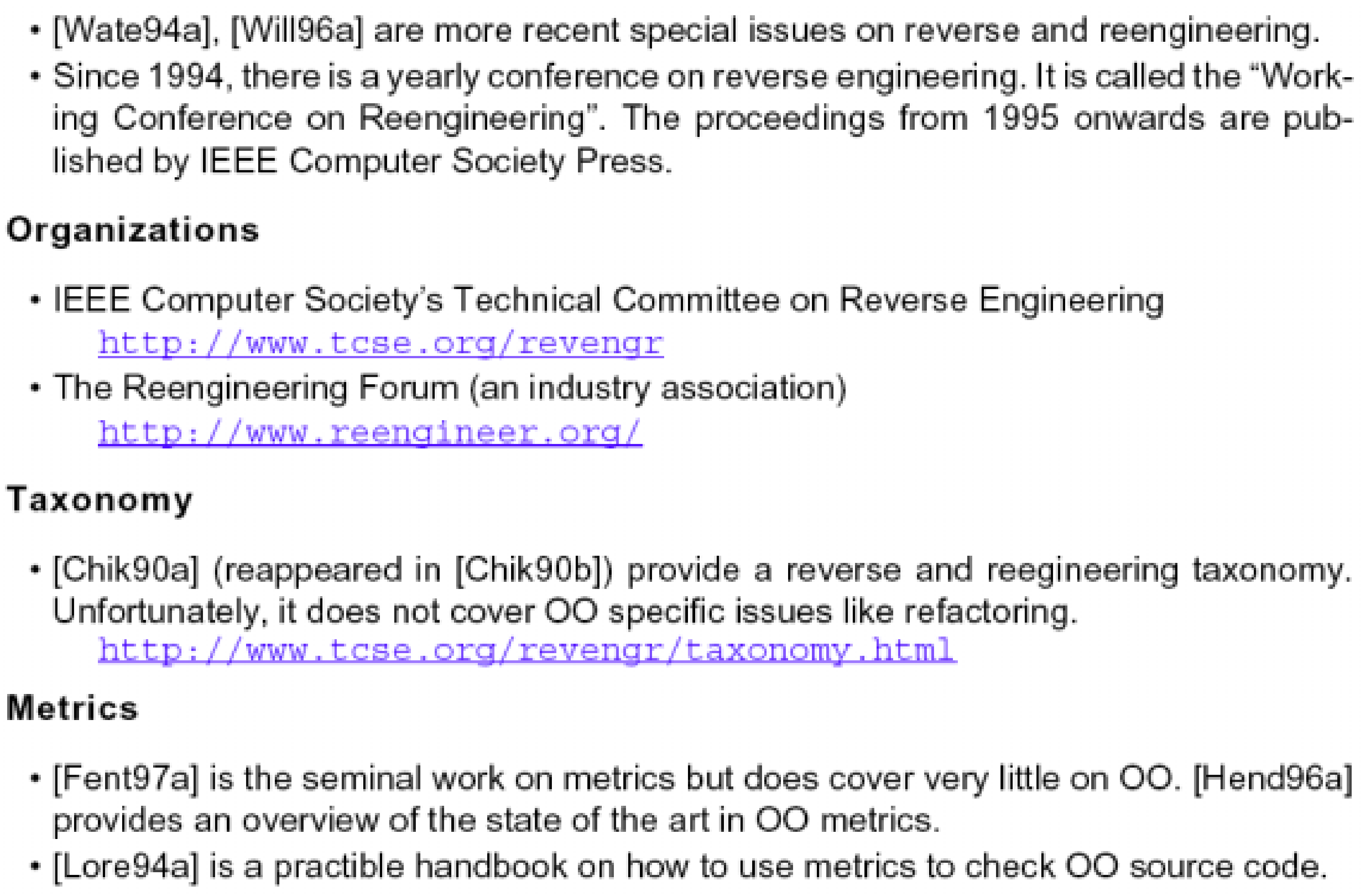
- *tool support*
- *culture chock*
- *management support*

one can reduce the costs between the different phases in the development cycles.

The tools are there …

Bibliography

Annotated biliography (I)



- [Wate94a], [Will96a] are more recent special issues on reverse and reengineering.
- Since 1994, there is a yearly conference on reverse engineering. It is called the "Working Conference on Reengineering". The proceedings from 1995 onwards are published by IEEE Computer Society Press.

Organizations

- IEEE Computer Society's Technical Committee on Reverse Engineering
  http://www.tcse.org/revengr
- The Reengineering Forum (an industry association)
  http://www.reengineer.org/

Taxonomy

- [Chik90a] (reappeared in [Chik90b]) provide a reverse and reegineering taxonomy. Unfortunately, it does not cover OO specific issues like refactoring.
  http://www.tcse.org/revengr/taxonomy.html

Metrics

- [Fent97a] is the seminal work on metrics but does cover very little on OO. [Hend96a] provides an overview of the state of the art in OO metrics.
- [Lore94a] is a practible handbook on how to use metrics to check OO source code.

Annotated biliography (II)

- [Fowl97a] provides a fast introduction to UML including the notion of “perspectives” which is quite interesting from a reverse engineering point of view because it is a way to specify how a certain UML diagram should be interpreted (i.e., on a Conceptual, Specification or Implementation level).
- [Booc98a], [Rumb99a] provide a good user reference and language reference for UML.
- [John92a] [Oden97a] present how patterns can support the documentation of a frameworks.
- [Brow96c], [Wuyt98a], [Prec98a] present some possible approaches to support design patterns extraction.
- [Flor97a] shows how design patterns can be supported at the development environemt level.
- [Stey96a] presents Reuse Contracts a way to document frameworks for evolution.
- [Wins87a] presents some discussion about variety of composition relationships.

Annotated biliography (III)

Refactoring and Code Smells

- The Ph.D. work of Opdyke [Opdy92b] resulted in a number of papers describing incremental redesign performed by humans supported by refactoring tools [Opdy93a], [John93b].
- [Fowl99a] summarises practical experience with refactorings and code smells.
- The Refactoring Browser —a Smalltalk tool that represents the state of the art in the field— is described in [Robe97a] and can be obtained from
  http://st-www.cs.uiuc.edu/
- Both Casais [Casa91b], [Casa92a], [Casa94a], [Casa95a] and Moore ([Moor96a]) report on tools that optimise class hierarchies without human intervention. Schulz et al. illustrate the feasability of refactorings on a subset of C++ [Schu98a].
- There exists a web-page discussing "code smells", i.e. suspicious symptoms in source code that might provide targets for refactoring
  http://c2.com/cgi/wiki?CodeSmells

Meta-Meta Models

- CDIF (CASE data interchange format)
  http://www.eigroup.org/
- MOF (Meta-Object Facility)
  http://www.omg.org/

References (I)

[Arno92a] Robert S. Arnold, Software Reengineering, IEEE Computer Society Press, Los Alamitos, CA, 1992.

[Bigg89c] T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse," IEEE Computer, IEEE Computer Society Press, October 1997, pp. 36-49.

[Bigg89d] T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse," Software Reengineering, Robert S. Arnold (Ed.), IEEE Computer Society Press, 1992, pp. 520-533.

[Booc98a] Grady Booch , James Rumbaugh and Ivar Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1998, ISBN: 0-210-57168-4.

[Brow96c] Kyle Brown, "Design Reverse-engineering and Automated Design Pattern Detection in Smalltalk," Technical Report, no. TR-96-07, North Carolina State University, 1996.

[Brow96c] Kyle Brown, Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk, TR-96-07, North Carolina State University, 1996, Ph.D. Thesis available from http://www.ksccary.com/kbrown.htm.

[Casa91b] Eduardo Casais, "Managing Evolution in Object Oriented Environments: An Algorithmic Approach," Ph.D. thesis (no. 369), Centre Universitaire d'Informatique, University of Geneva, May 1991.

[Casa92a] Eduardo Casais, "An Incremental Class Reorganization Approach," Proceedings ECOOP'92, O. Lehrmann Madsen (Ed.), LNCS 615, Springer-Verlag, Utrecht, The Netherlands, June/July 1992, pp. 114-132.

References (II)

[Casa94a] Eduardo Casais, "Automatic Reorganization of Object-Oriented Hierarchies: A Case Study," Object-Oriented Systems, vol. 1, no. 2, Chapman & Hall, December 1994, pp. 95-115.

[Casa95a] Eduardo Casais, "Managing Class Evolution in Object-Oriented Systems," Object-Oriented Software Composition, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 201-244.

[Casa97a] Eduoardo Casais and Antero Taivalsaari, "Object-Oriented Software Evolution and Re-engineering (Special Issue)," Journal of Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, 1997, pp. 233-301.

[Chik90a] E.J. Chikofsky and J.H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software Engineering, 1990 January, pp. 13-17.

[Davi95a] Alan M.Davis, 201 Principles of Software Development, McGraw-Hill, 1995.

[Fent97a] Norman Fenton and Shari Lawrence Pfleeger, Software Metrics: A Rigorous and Practical Approach, Second edition, International Thomson Computer Press, London, UK, 1997.

[Flor97a] Gert Florijn, Marco Meijers and Pieter van Winsen, "Tool Support for Object-Oriented Patterns," Proceedings ECOOP'97, Mehmet Aksit and Satoshi Matsuoka (Ed.), LNCS 1241, Springer-Verlag, Jyvaskyla, Finland, June 1997, pp. 472-495.

[Fowl97a] Martin Fowler, UML Distilled, Addison-Wesley, 1997.

[Fowl99a] Martin Fowler, Refactorings (Working Title), Addison-Wesley, 1999.

[Gold95a] Adele Goldberg and Kenneth S. Rubin, Succeeding With Objects: Decision Frameworks for Project Management, Addison-Wesley, Reading, Mass., 1995.

References (III)

[Hend96a] Brian Henderson-Sellers, Object-Oriented Metrics: Mesures of Complexity, Prentice-Hall, 1996.

[John92a] Ralph E. Johnson, "Documenting Frameworks using Patterns," Proceedings OOPSLA '92, ACM SIGPLAN Notices, vol. 27, no. 10, Oct. 1992, pp. 63-76.

[John93b] Ralph E. Johnson and William F. Opdyke, "Refactoring and Aggregation," Object Technologies for Advanced Software, First JSSST International Symposium, Lecture Notes in Computer Science, vol. 742, Springer-Verlag, Nov. 1993, pp. 264-278.

[Lehm85a] Lehman M. M. and Belady L., Program Evolution - Processes of Software Change, London Academic Press, 1985.

[Lore94a] Mark Lorenz and Jeff Kidd, Object-Oriented Software Metrics: A Practical Approach, Prentice-Hall, 1994.

[Moor96a] Ivan Moore, "Automatic Inheritance Hierarchy Restructuring and Method Refactoring," Proceedings of OOPSLA '96 Conference, ACM Press, 1996, pp. 235-250.

[Oden97a] Georg Odenthal and Klaus Quibeldey-Cirkel, "Using Patterns for Design and Documentation," Proceedings of ECOOP'97, LNCS 1241, 1997, pp. 511-529.

[Opdy92b] William F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. thesis, University of Illinois, 1992.

[Opdy93a] William F. Opdyke and Ralph E. Johnson, "Creating Abstract Superclasses by Refactoring," Proceedings of the 1993 ACM Conference on Computer Science, ACM Press, 1993, pp. 66-73.

References (IV)

[Prec98a] Lutz Prechelt and Christian Kramer, "Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns",Journal of Universal Computer Science, 1998, 4, 12, 866-882,december

[Press94a] Roger S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, 1994.

[Robe97a] Don Roberts, John Brant and Ralph E. Johnson, "A Refactoring Tool for Smalltalk," Journal of Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, 1997, pp. 253-263.

[Rumb99a] James Rumbaugh , Ivar Jacobson and Grady Booch, The Unified Modeling Language Reference, Addison-Wesley, 1999, 0-210-30998-X.

[Schu98a] Benedikt Schulz, Thomas Genssler, Berthold Mohr and Walter Zimmer, "On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems.," Proceedings of the TOOLS 27 Conference (Asia '98), IEEE Computer Society Press, 1998.

[Somm96a] Ian Sommerville, Software Engineering Fifth Edition, Addison-Wesley, 1996.

[Stey96a] Patrick Steyaert, Carine Lucas, Kim Mens and Theo D'Hondt, "Reuse Contracts: Managing the Evolution of Reusable Assets," Proceedings of OOPSLA '96 Conference, ACM Press, 1996, pp. 268-285.

[Wate94a] Richard C. Waters and Elliot Chikofsky, "Reverse Engineering: Progress Along Many Dimensions (Special Issue)," Communications of the ACM, vol. 37, no. 5, May 1994, pp. 22-93.

[Wins87a]MortonE.Winston,RogerChaffinandDouglasHerrmann,"ATaxonomyofPart-WholeRelations", Cognitive Science, 1987, 11, 417-444

[Will96a] Linda Wills and Philip Newcomb, "Reverse Engineering (Special Issue)," Automated Software Engineering, vol. 3, no. 1-2, June 1996, pp. 5-172.

References (V)

[Wuyt98a] Roel Wuyts, "Class-management using Logical Queries, Application of a Reflective User Interface Builder," Proceedings of the TOOLS 26 Conference (USA '98), IEEE Computer Society Press, 1998.