Object-Oriented Software Reengineering

Dr. S. Demeyer
Dr. S. Ducasse
Prof. Dr. O. Nierstrasz

Wintersemester 1998/1999

Table of Contents

February 5, 1999

Table of Contents

February 5, 1999

1. Object-Oriented Software Reengineering

Lecturers: Dr. S. Demeyer, Dr. S. Ducasse, Prof. Dr. O. Nierstrasz

WWW: `http://www.iam.unibe.ch/~scg`

Sources

❑   *Software Engineering*, Ian Sommerville, Addison-Wesley, 5th edn., 1995

❑   *Software Reengineering*, Ed. Robert S. Arnold, IEEE Computer Society, 1993

Goals of this course

The "Software Crisis" is an artefact of short-sighted software practices

❑ try to understand factors that lead to software maintenance problems

Legacy systems are "old systems that must still be maintained"

❑ study legacy systems to understand what problems they pose

Reverse Engineering

❑ examine ways to recover design and analysis models from existing systems

Reengineering

❑ explore techniques to transform systems to make them more maintainable

Object-Oriented Reengineering

❑ survey the particular *problems* and *opportunities* of reengineering object oriented legacy systems

Course Overview

1. 30/10 Introduction
2. 06/11 Metrics Introduction
3. 13/11 Metrics in Industry — *Dr. Simon Moser*
4. 20/11 Y2K — *Prof. Dr. Gerhard Knolmayer*
5. 27/11 Design Extraction with UML
6. 04/12 Language Issues and Extraction Tracks
7. 11/12 Detecting Duplicated Code
   18/12 *no course*
   25/12 *winter holiday*
   01/01 *winter holiday*
8. 08/01 Refactoring
9. 15/01 Metrics in Reengineering
10. 22/01 Refactoring Browser
    29/01 *open*
11. 05/02 Code Repositories

What is a Legacy System?

legacy

A sum of money, or a specified article, given to another by will; anything handed down by an ancestor or predecessor.

— *OED*

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❑ outdated development methods used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❑ missing or outdated documentation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]

Corrective maintenance (17%)

fixing reported errors in the software

Adaptive maintenance (18%)

adapting the software to a new environment (e.g., platform or O/S)

Perfective maintenance (65%)

implementing new functional or non-functional requirements

Why is Software Maintenance Expensive?

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

Costs can be high because:

❑ Maintenance staff are often *inexperienced* and *unfamiliar with the application domain*

❑ Programs being maintained may have been developed without modern techniques; they may be *unstructured*, or *optimized for efficiency*, not maintainability

❑ *Changes may introduce new faults*, which trigger further changes

❑ As a system is changed, its *structure tends to degrade*, which makes it harder to change

❑ With time, *documentation may no longer reflect the implementation*

Lehman's Laws

A classic study by Lehman and Belady (1985) 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.

Factors Affecting Maintenance

❑   Module independence

❑   Programming language

❑   Programming style

❑   Program validation and testing

❑   Quality of documentation

❑   Configuration management techniques

❑   Application domain

❑   Staff stability

❑   Age of program

❑   Dependence on external environment

❑   Hardware stability

Maintainability Metrics

Hypothesis: Program maintainability is related to complexity

- ❑ McCabe (1976): measures a program's complexity in terms of the graph of its decision structure

- ❑ Halstead (1977): measures complexity in terms of number of unique operators and operands, and total frequency of operands

- ❑ Kafura and Reddy (1987): used a cocktail of seven different metrics

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."

— Chikofsky and Cross [in Arnold, 1993]

Tools Architectures

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

| Software work product | Parser, Semantic analyzer → Information base → View composer(s) | New view(s) of product |
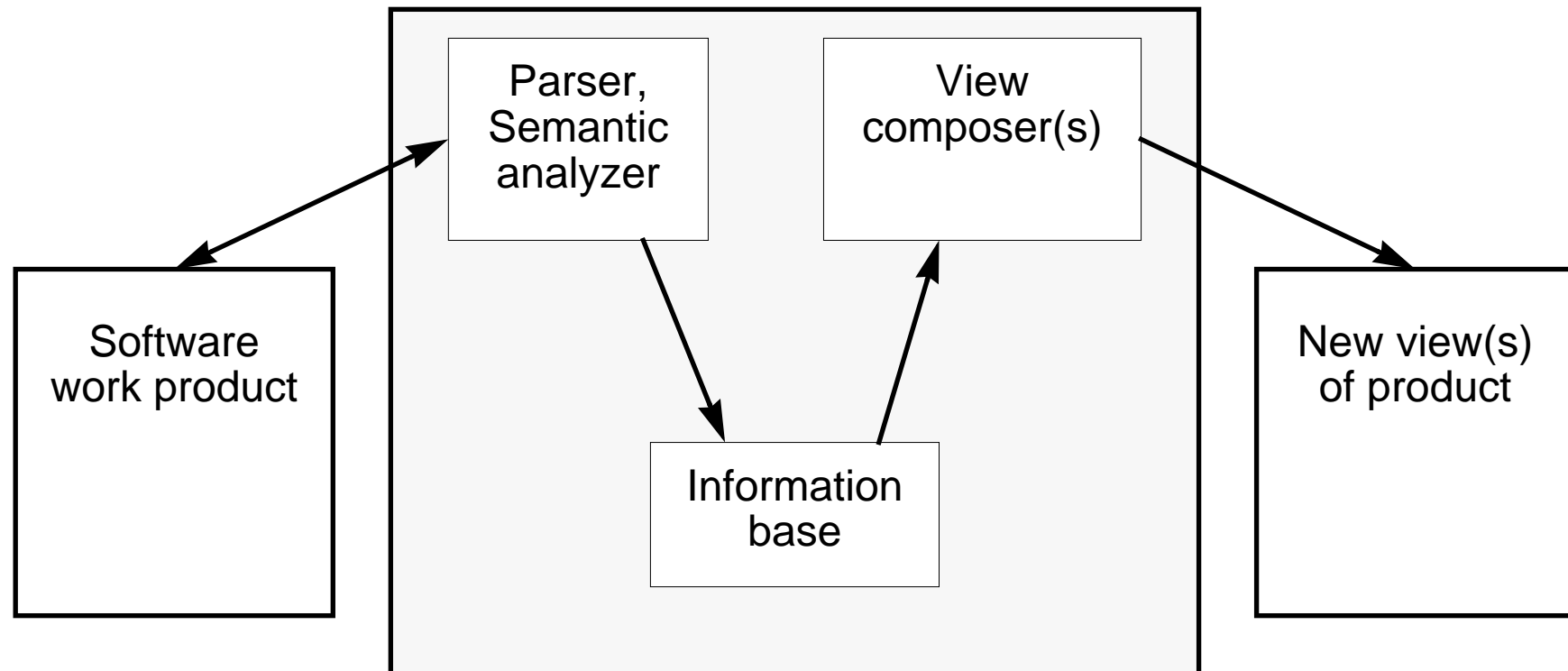
Reverse and Re-engineering
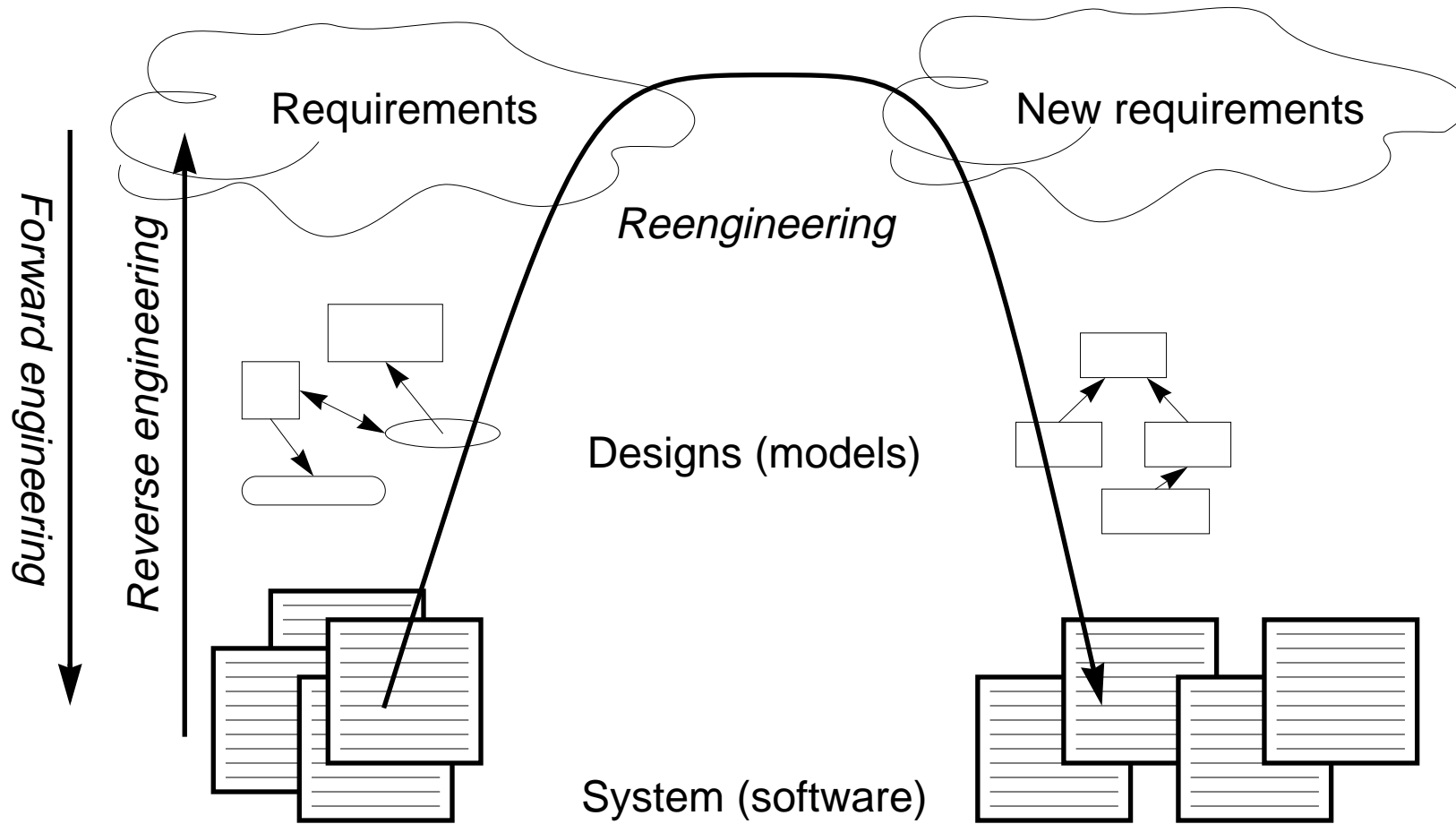
Goals of Reverse Engineering

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

— Chikofsky and Cross [in Arnold, 1993]

Reverse Engineering Techniques

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

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

"<u>Design recovery</u> *recreates design abstractions* from a combination of code, existing documentation (if available), personal experience, and general knowledge about problem and application domains." [Biggerstaff]

- ❑ 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

Summary

❑ We will *always* have legacy systems, because valuable software systems outlive their original requirements

❑ Early adopters of OO methods now find themselves with *OO legacy software*

❑ Reverse engineering techniques help to *recover designs* from legacy software

❑ Reengineering techniques are needed to restructure valuable legacy software so that it can *meet new requirements*, both now, and in the future

2. Metrics

Outline

❑ What are metrics? Why do we need them?

❑ Metrics for cost estimation

❑ Metrics for software quality evaluation

Sources

❑ *Software Metrics: A Rigorous and Practical Approach*, Norman Fenton and Shari Lawrence Pfleeger, 2d edn, PWS Publishing Co., 1997.

❑ *Software Engineering*, Ian Sommerville, Addison-Wesley, 5th edn., 1995

❑ *Tutorial on Software Metrics*, Simon Moser, Brian Henderson-Sellers, C. Mingins, 1997

Why Measure Software?

Estimate cost and effort

❑ measure correlation between specifications and final product

Improve productivity

❑ measure value and cost of software

Improve software quality

❑ measure usability, efficiency, maintainability ...

Improve reliability

❑ measure mean time to failure, etc

Evaluate methods and tools

❑ measure productivity, quality, reliability ...

...

"You cannot control what you cannot measure" — De Marco, 1982

"What is not measurable, make measurable" — Galileo

What is a Metric?

Software metrics

❑ Any type of measurement which relates to a software system, process or related documentation

☞ Lines of code in a program

☞ the Fog index

☞ number of person-days required to develop a component

☞ ...

❑ Allow the software and the software process to be quantified

❑ Measures of the software process or product

❑ Should be captured automatically if possible

GQM

Goal - Question - Metrics approach [Basili et al. 1984]

❑ Define *Goal*

☞ e.g., "How effective is the coding standard XYZ?"

❑ Break down into *Questions*

☞ "Who is using XYZ?"

☞ "What is productivity/quality with/without XYZ?"

❑ Pick suitable *Metrics*

☞ Proportion of developers using XYZ

☞ Their experience with XYZ ...

☞ Resulting code size, complexity, robustness ...

Metrics assumptions

Assumptions

- ❏ A software property can be measured
- ❏ The relationship exists between what we can measure and what we want to know
- ❏ This relationship has been formalized and validated

It may be difficult to relate what can be measured to desirable quality attributes

Measurement analysis

- ❏ Not always obvious what data means. Analysing collected data is very difficult
- ❏ Professional statisticians should be consulted if available
- ❏ Data analysis must take local circumstances into account

Cost estimation objectives

❑ To establish a budget for a software project

❑ To provide a means of controlling project costs

❑ To monitor progress against the budget

☞ comparing planned with estimated costs

❑ To establish a cost database for future estimation

❑ Cost estimation and planning/scheduling are closely related activities

Estimation techniques

❑ Expert judgement
❑ Estimation by analogy
❑ Parkinson's Law
❑ Pricing to win
❑ Top-down estimation
❑ Bottom-up estimation
❑ Algorithmic cost modelling

Algorithmic cost modelling

❑ Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers

❑ The function is derived from a study of historical costing data

❑ Most commonly used product attribute for cost estimation is LOC (code size)

❑ Most models are basically similar but with different attribute values

Measurement-based estimation

A. Measure

Develop a *system model* and measure its size

C. Interpret

Adapt the effort with respect to a specific *development project plan*
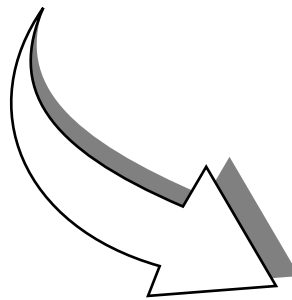
B. Estimate

Determine the effort with respect to an *empirical database* of measurements from *similar projects*
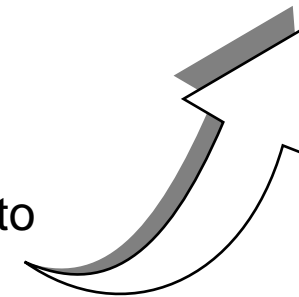
Lines of code

Lines of Code as a measure of system size?

❑ Easy to measure; but not well-defined for modern languages
☞ What's a line of code?
☞ What programs should be counted as part of the system?

❑ Assumes linear relationship between system size and volume of documentation

❑ A poor indicator of productivity
☞ Ignores software reuse, code duplication, benefits of redesign
☞ The lower level the language, the more productive the programmer
☞ The more verbose the programmer, the higher the productivity

Function points

Function Points (Albrecht, 1979)

❑ Based on a combination of program characteristics:
  ☞ external inputs and outputs
  ☞ user interactions
  ☞ external interfaces
  ☞ files used by the system

❑ A weight is associated with each of these

❑ The function point count is computed by multiplying each raw count by the weight and summing all values

❑ Function point count modified by complexity of the project

Good points, bad points

❑ Can be measured already after design

❑ FPs can be used to estimate LOC depending on the average number of LOC per FP for a given language

❑ LOC can vary wildly in relation to FP

❑ FPs are very subjective — depend on the estimator. They cannot be counted automatically

Programmer productivity

A measure of the rate at which individual engineers involved in software development produce software and associated documentation

Productivity metrics

❑ Size related measures based on some output from the software process. This may be lines of delivered source code, object code instructions, etc.

❑ Function-related measures based on an estimate of the functionality of the delivered software. Function-points are the best known of this type of measure

Productivity estimates

❑ Real-time embedded systems, 40-160 LOC/P-month

❑ Systems programs , 150-400 LOC/P-month

❑ Commercial applications, 200-800 LOC/P-month

Quality and productivity

❑ All metrics based on volume/unit time are flawed because they do not take quality into account

❑ Productivity may generally be increased at the cost of quality

❑ It is not clear how productivity/quality metrics are related

The COCOMO model

❑ Developed at TRW, a US defense contractor

❑ Based on a cost database of more than 60 different projects

❑ Exists in three stages

☞ Basic - Gives a 'ball-park' estimate based on product attributes

☞ Intermediate - modifies basic estimate using project and process attributes

☞ Advanced - Estimates project phases and parts separately

Basic COCOMO Formula

❑ Effort = $C \times PM^S \times M$

☞ C is a complexity factor

☞ PM is a product metric (size or functionality)

☞ exponent S is close to 1, but increasing for large projects

☞ M is a multiplier based on process, product and development attributes (~1)

Project classes

❑ *Organic mode* small teams, familiar environment, well-understood applications, no difficult non-functional requirements (EASY)

☞ Effort = $2.4 \, (KDSI)^{1.05} \times M$

❑ *Semi-detached mode* Project team may have experience mixture, system may have more significant non-functional constraints, organization may have less familiarity with application (HARDER)

☞ Effort = $3 \, (KDSI)^{1.12} \times M$

❑ Embedded Hardware/software systems, tight constraints, unusual for team to have deep application experience (HARD)

☞ Effort = $3.6 \, (KDSI)^{1.2} \times M$

COCOMO assumptions

- ❑ Implicit productivity estimate
    - ☞ Organic mode = 16 LOC/day
    - ☞ Embedded mode = 4 LOC/day
- ❑ Time required is a function of total effort NOT team size
- ❑ Not clear how to adapt model to personnel availability

Staffing requirements

- ❑ Staff required can't be computed by dividing the development time by the required schedule
- ❑ The number of people working on a project varies depending on the phase of the project
- ❑ The more people who work on the project, the more total effort is usually required
- ❑ Very rapid build-up of people often correlates with schedule slippage

Product quality metrics

❑ A quality metric should be a predictor of product quality.

❑ Most quality metrics are design quality metrics and are concerned with measuring the coupling or the complexity of a design.

❑ The relationship between these metrics and quality as judged by a human may hold in some cases but it is not clear whether or not it is generally true.

Design maintainability

❑ Cohesion

☞ How closely are the parts of a component related?

❑ Coupling

☞ How independent is a component?

❑ Understandability

☞ How easy is it to understand a component's function?

❑ Adaptability

☞ How easy is to change a component?

Coupling metrics

Associated with Yourdon's 'Structured Design'/ Measures 'fan-in and fan-out' in a structure chart:

❑ High fan-in (number of calling functions) suggests high coupling because of module dependencies.

❑ High fan-out (number of calls) suggests high coupling because of control complexity.

Henry and Kafura's modifications

❑ The approach based on the calls relationship is simplistic because it ignores data dependencies.

❑ Informational fan-in/fan-out takes these into account.

☞ Number of local data flows + number of global data structures updated.

☞ Data-flow count subsumes calls relation. It includes updated procedure parameters and procedures called from within a module.

❑ Complexity = Length * (Fan-in * Fan-out)2

☞ Length is any measure of program size such as LOC.

Validation of quality metrics

❑ Some studies with Unix found that informational fan-in/fan-out allowed compex and potentially fault components to be identified.

❑ Some studies suggest that size and number of branches are as useful in predicting complexity than informational fan-in/fan-out.

❑ Fan-out on its own also seemed to be a better quality predictor.

❑ The whole area is still a research area rather than practically applicable.

Program quality metrics

Design metrics also applicable to programs

- ❑ Other metrics include
    - ☞ Length. The size of the program source code
    - ☞ Cyclomatic complexity. The complexity of program control
    - ☞ Length of identifiers
    - ☞ Depth of conditional nesting
- ❑ Anomalous metric values suggest a component may contain an above average number of defects or may be difficult to understand

Metric utility

- ❑ Length of code is simple but experiments have suggested it is a good predictor of problems
- ❑ Cyclomatic complexity can be misleading
- ❑ Long names should increase program understandability
- ❑ Deeply nested conditionals are hard to understand. May be a contributor to an understandability index

Metrics maturity

❑ Metrics still have a limited value and are not widely collected

❑ Relationships between what we can measure and what we want to know are not well-understood

❑ Lack of commonality across software process between organizations makes universal metrics difficult to develop

Summary

❑ Factors affecting productivity include individual aptitude, domain experience, the development project, the project size, tool support and the working environment

❑ Prepare cost estimates using different techniques. Estimates should be comparable

❑ Algorithmic cost estimation is difficult because of the need to estimate attributes of the finished product

❑ The time required to complete a project is not simply proportional to the number of people working on the project

❑ Metrics gather information about both process and product

❑ Control metrics provide management information about the software project. Predictor metrics allow product attributes to be estimated

❑ Quality metrics should be used to identify potentially problematical components

3. Object-Oriented Metrics in Industry

Invited Lecture by Dr. Simon Moser

See:

`http://www.iam.unibe.ch/~scg/Teaching/OOReen/`

Object-Oriented Software Re-engineering

Universität Bern, IAM, SCG

Object-Oriented Metrics in Industry

Dr. Simon Moser

Release note

This 2 hour lecture emerged from previously developed tutorials held over the years 1994-1998.

It is a development of

Dr. Simon Moser (moser@acm.org).

Outline

Measurement-Based Estimation in Industry

The Effects of Reuse

Industry Benchmarking

Object-Oriented Metrics

Sources

[1] Moser S., Measurement and Estimation of Software and Software Processes, Ph.D. thesis (http://www.iam.unibe.ch/~scg/Archive/PhD/moser-phd.pdf)

[2] Moser S., Nierstrasz O., The Effect of Object-Oriented Frameworks on Developer Productivity, IEEE Software, September 1996, pp. 45-51

[3] IFPUG, International Function Point User Group (http://www.ifpug.org)

[4] Jones C., Programming languages Table (http://www.spr.com/library/0langtbl.htm)

[5] Moser S., Misic V., Class Coupling and Cohesion: A Formal Metamodel Approach, Proc. Of APSEC'97, Hong Kong, Dec. 1997, pp. 31-39

Measurement-Based Estimation in Industry (1/10)

- Example:

  Estimate the development effort for a **„Tiny Order Information System"** (TOIS).

- The steps (cf. foil 28):

  (A) Measure requirements model
  (B) Derive standard development effort
  (C) Adapt with respect to tailored development plan

Measurement-Based Estimation in Industry (2/10)

Customer Information | 5

Order Information | 3

Stock Information | 5

Manage Objects | 4

Do Forecasts | 2

Do Statistics | 2
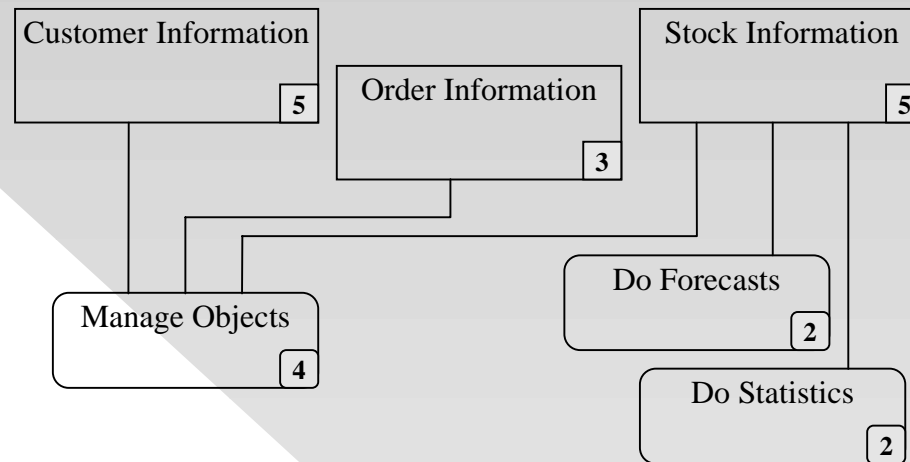
Notation:

= *subject area*   4

= *basic functionality*   4

= *link, i.e. functionality applies for subject area*

Graphical notation of the TOIS preliminary requirements model

Measurement-Based Estimation in Industry (3/10)

```
; PRE description of TOIS, the tiny order information system
Subject Area "Customer Information" 5 .
Subject Area "Order Information" 3 .
Subject Area "Stock Information" 5 .
Functionality "Manage Objects" 4 .
Functionality "Do Statistics" 2 .
Functionality "Do Forecasts" 2 .
Subject Area 'Customer Information' contains 'Manage Objects' .
Subject Area 'Order Information' contains 'Manage Objects' .
Subject Area 'Stock Information' contains 'Manage Objects',
              'Do Statistics', 'Do Forecasts' .
```

Textual notation of the TOIS preliminary requirements model
(language PRE, cf. [1])

Measurement-Based Estimation in Industry (4/10)

```
issue

ma -v -f tois.sdf
```

in a DOS command shell from the **SEBT**-directory. After admittedly a bit lengthy phases of tokenising, scanning and finally counting, the tool will eventually finish its work by printing out a measured number:

```
System Meters = 563
```

Step (A): measurement of the TOIS preliminary requirements model using the utility „ma" resulting in System Meters at PRE level (PRE SM)

Measurement-Based Estimation in Industry (5/10)
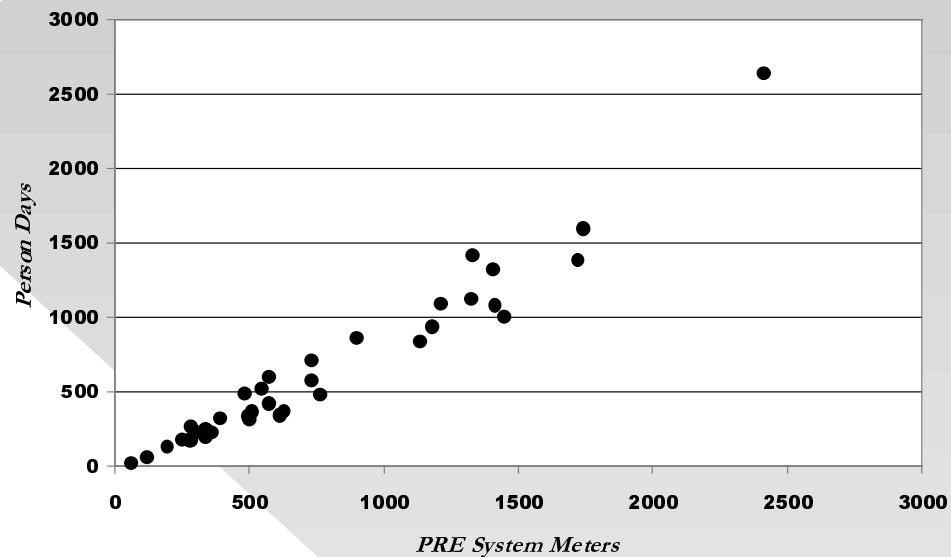
Effort$_{\text{Person Days}}$ = Complexity$_{\text{PRE SM}}$ $\times$ 0.605 + Complexity$_{\text{PRE SM}}^2$ $\times$ 0.0(

```
563 × 0.605 + 563² × 0.0001779 = 340.6 + 56.4 = 397 PD
(95%-error range: +/-33%)
```

Step (B): Entering the empirical database to obtain an estimate for a standard full development process

Measurement-Based Estimation in Industry (6/10)

Level of Elaboration ⟶

Result ↓

		Drafted	Verified	Maintained
1	Preliminary Analysis			
2	Domain Analysis			
3	Specification			
4	Construction (=Design)			
5	Coding and Test			
6	Delivery Preparation			
7	Delivery (Installation, Acceptance Tests and Training)			
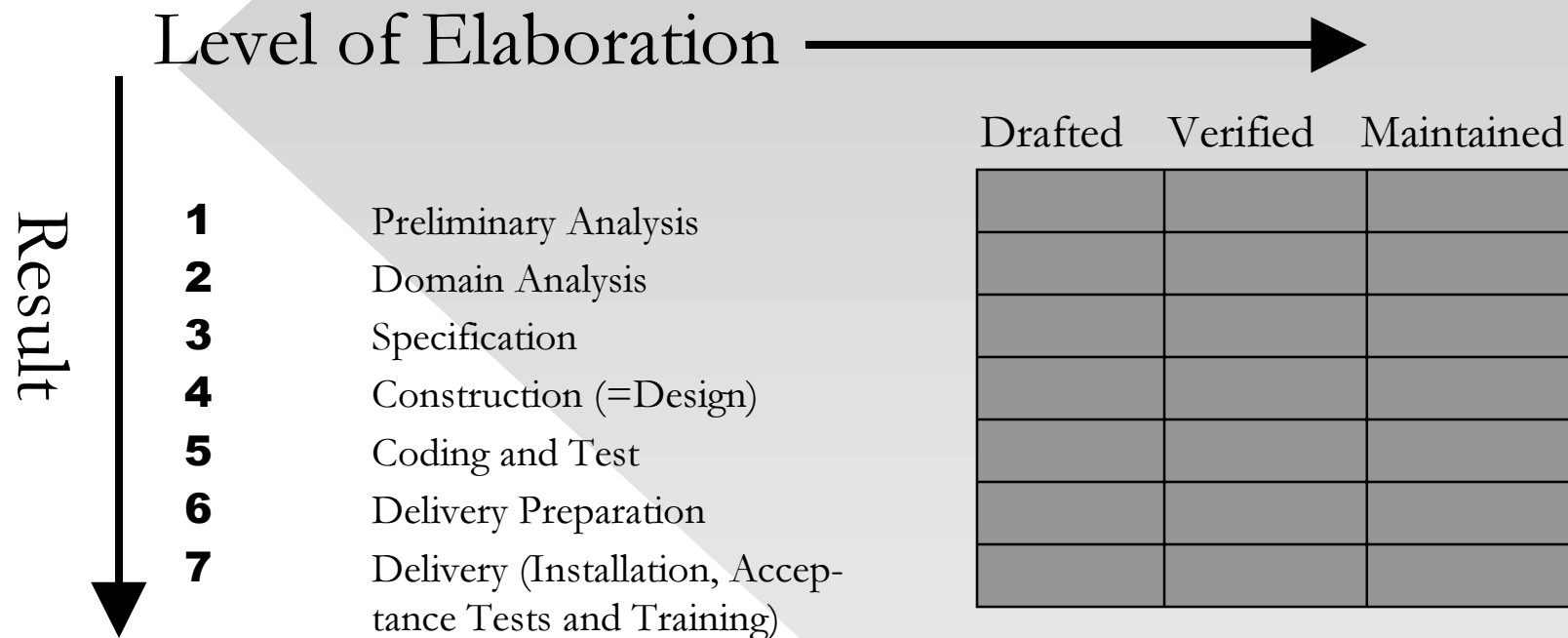
Step (C): Adapting the „normal effort" to the tailored plan:

C.1 Process Completeness Assessment (PCA) ⇨ %C

Measurement-Based Estimation in Industry (7/10)
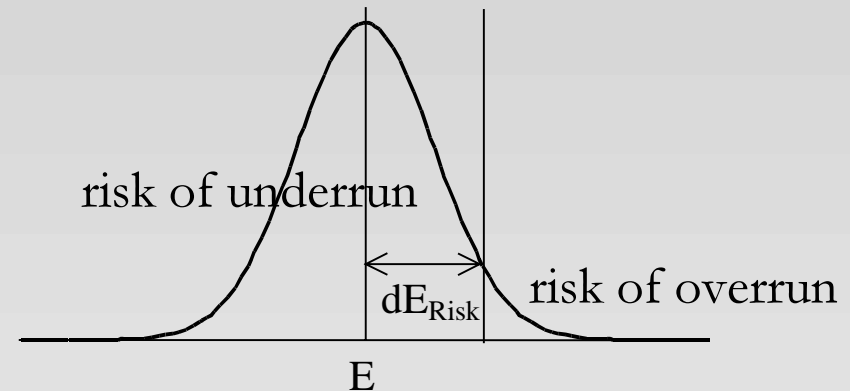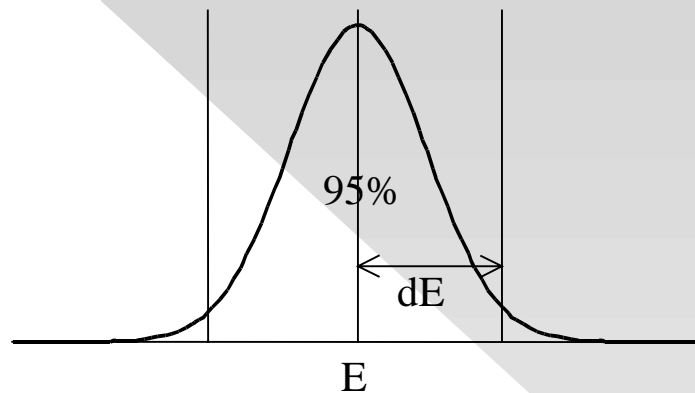
E = estimate, dE = error range of estimate

Step (C): Adapting the „normal effort" to the tailored plan:

C.2 Risk Adaptation $\Rightarrow$ $\%dE_{Risk}$, $E_{C.2} = E \times (1 + \%dE_{Risk})$

Measurement-Based Estimation in Industry (8/10)

Step (C):

C.3: Project Cost Calculation (PCC):

Input: - risk adapted „normal effort"
        - PCA per project phase (optional: per contractor)
        - degree of completeness per subsystem per phase
        - span activities (project mgmt, quality management)
        - tariffs per activity

Output: - efforts per subsystem and phase
        - costs per subsystem and phase

Measurement-Based Estimation in Industry (9/10)

How to use measurement-based estimation in Re-Engineering:

(A) Getting the System Size backwards (Backfire-method):
    1) measure LOC (or KLOC) per language
    2) derive functional size by using empirical DBs per language (cf. [4])

(B) Deriving „normal effort"

(C) Doing PCA for the Re-Engineering process:
    - What do we have in maintained form? Code? Design? Specs? Domain models?
    - What do we want? In what form?
    - What can we obtain in an automated way? (assess as draft)

Measurement-Based Estimation in Industry (10/10)

More topics...

- Using more detailed models (e.g. business models using UML, cf. [1])

- Deriving ideal team size and elapsed time from the adapted effort (cf. [1])

- Adaptations for very tight schedules (cf. [1])

- Using other functional measures (e.g. Function Points, cf. [3])

The Effects of Reuse (1/4)

- Example:

  For some parts of the TOIS we have reusable parts at hand...

- possible strategies:

  1 Heuristically assess completeness percentages
  (step C.3 PCC)

  2 Rely on special empirical database (step B)

  3 Modify the requirements model (chosen approach...)

The Effects of Reuse (2/4)

```
; PRE description of TOIS, the tiny order information system, with reuse modelling
;ma-entry: category library
Subject Area     "Customer Information"  5 .
;ma-entry: category project
Subject Area     "Order Information"  3 .
Subject Area     "Stock Information"  5 .
;ma-entry: category library
Functionality    "Manage Objects"  4 .
;ma-entry: category project
Functionality    "Do Statistics"  2 .
Functionality    "Do Forecasts"  2 .
;ma-entry: category library
Subject Area     'Customer Information' contains 'Manage Objects'.
;ma-entry: category project
Subject Area     'Order Information' contains 'Manage Objects'.
Subject Area     'Stock Information' contains 'Manage Objects',
                 'Do Statistics', 'Do Forecasts'.
```

Reuse-modified TOIS preliminary requirements model

(language PRE, cf. [1])

The Effects of Reuse (3/4)

```
invoking ma...


System Meters = 415


415 × 0.605 + 415² × 0.0001779 = 251 + 30.6 = 281.6 PD
```

Steps A and B for the reuse-adapted TOIS

The Effects of Reuse (4/4)

Project	Function Point Productivity	Productivity Increase (%)	PRE SM Productivity	Productivity Increase (%)
C1	1.1	N/A	2.47	N/A
C2	1.6	45	2.30	-7
C3	1.6	45	2.12	-14
C4	1.5	36	2.64	7
C5	1.7	55	2.30	-7
C6	1.5	36	2.13	-14
C7	1.6	45	2.42	-2
C8	1.6	45	2.69	9

C1 = Framework project, C2-8 = Reuse projects

Productivity rates of reuse-adapted (PRE SM) and non-adapted (Function Point) model sizes (cf. [2])

Non-adapted sizes show „magical" increase...

Industry Benchmarking (1/1)

- Compare productivity rates (or other target key figures) among companies...

- Analyse company/project characteristics ⇨ possible reasons

- Define corrective actions... and iterate

Object-Oriented Metrics (1/4)

The **generic metamodel** of system descriptions...

contains

isWrittenBy

DESCRIPTION
OBJECT (DO)

name

covers

language DO

library DO

project DO
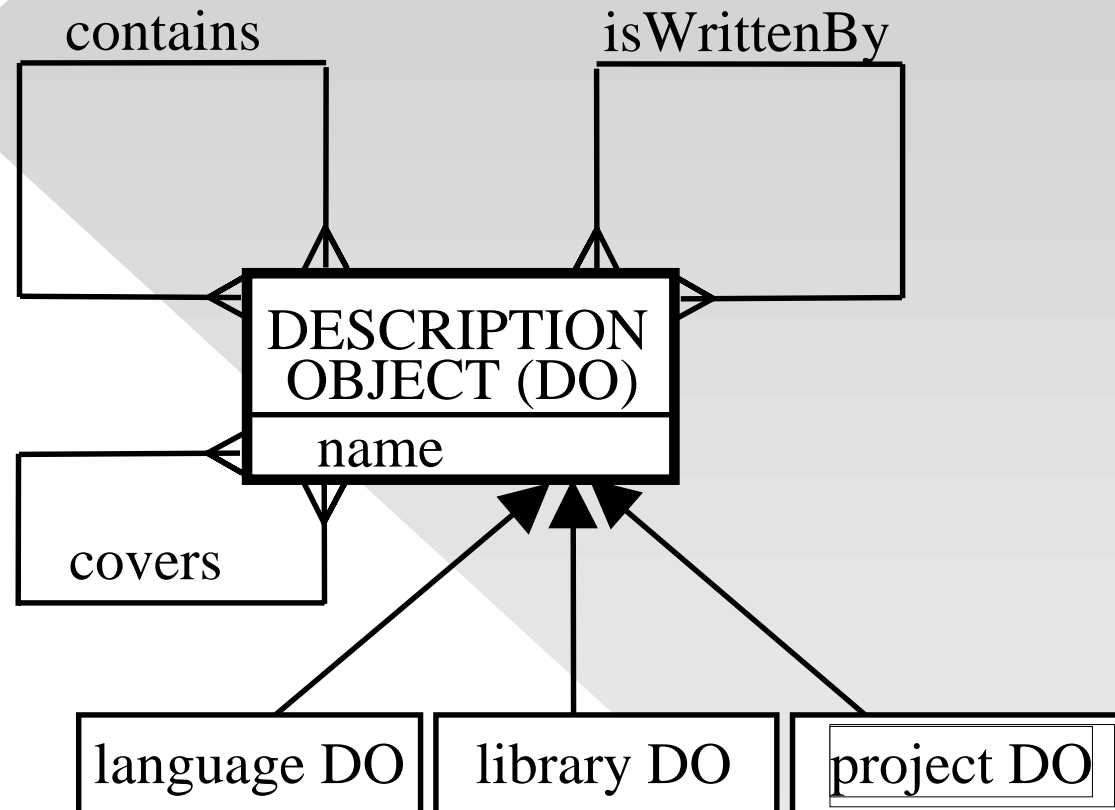
Object-Oriented Metrics (3/4)

The **metamodel hierarchy** for object-oriented code...
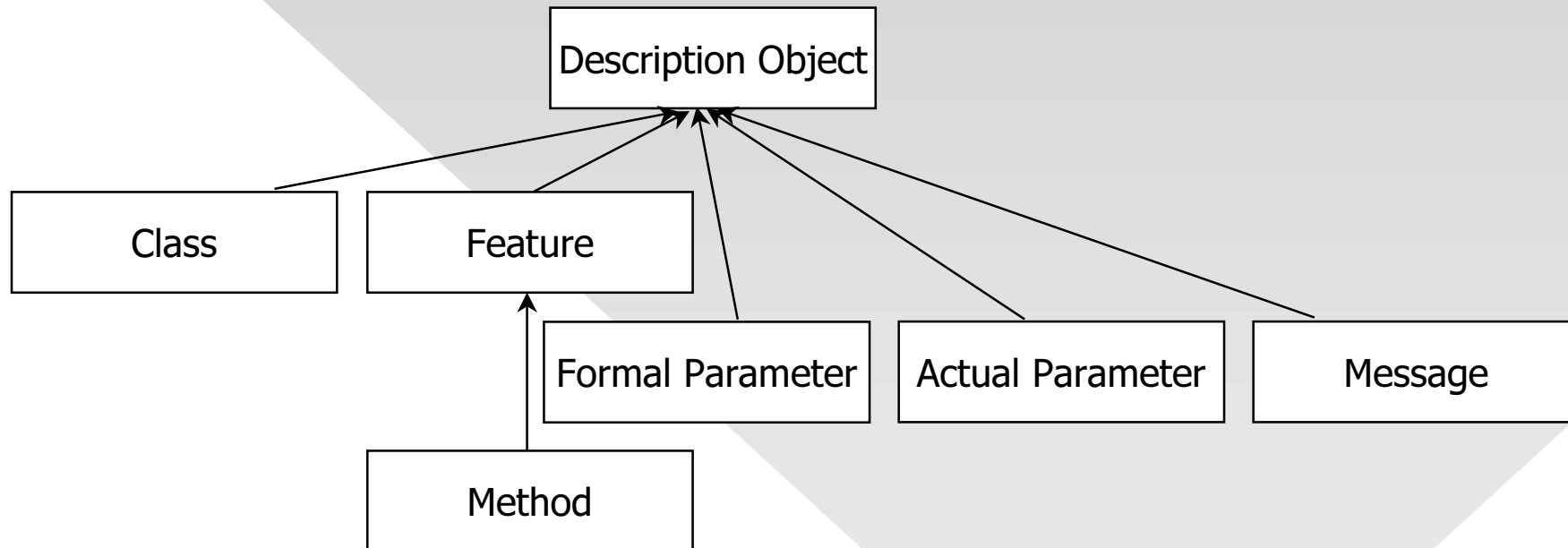
Object-Oriented Metrics (3/4)

Metrics of size, e.g. System Meter

- – Definition (for single description object):
  $$\textbf{Size} \equiv externalSize + internalSize$$
  $$externalSize \equiv numberOfTokens\ (name)$$
  $$internalSize \equiv externalSize\ (isWrittenBy)$$

- – Definition (for sets of description objects):
  $$\textbf{Size} \equiv externalSize\ (library) + Size\ (project)$$

(NB: The above notation is somewhat casual)

e.g. Number of Classes
(easy but correlates with +/-40% to effort only)

Object-Oriented Metrics (4/4)

Quality metrics, e.g. coupling (cf. Mišić, Moser, 1997)

– Definition on generic metamodel:

$$\mathbf{coupling}\,(c1, c2) = 1 - \frac{\displaystyle\sum_{o \in c1.isDefinedWith \cap \left(\{c2\} \cup c2.contains^+\right)} externalSize(o)}{\displaystyle\sum_{p \in c1.isDefinedWith} externalSize(p)}$$

– Informal definition: Coupling is the ratio of dependencies outside some composite description object

Object-Oriented Metrics in Industry - Summary

- **Measurement-based estimation**: transparent, improvable and adaptable (e.g. to re-engineering projects)

- **Reuse modelling**: enables extension of measurement-based techniques to framework and reuse projects

- **Benchmarking**: comparing measured performance, improve towards industry state-of-the-art

- **Object-oriented metrics**: KISS ... and wait for more research results!

4. The Year 2000 Problem

Invited Lecture by Prof. Dr. Gerhard Knolmayer

See:

`http://www.ie.iwi.unibe.ch/zeit/y2k/`

Vergleichskriterien	Titel Zeitsprung 2000 • **Beispiele** • **Checklisten** • **Lösungen** **für Manager und Informatiker**	Das Jahr-2000-Problem, Projektmanagement von Datumsumstellungen in der Software	Das Jahr 2000 in der EDV, Bewältigung des Jahr-2000- Problemes in ihrem Unternehmen	Die Jahr 2000-Lösung So schafft Ihre Software den Sprung ins nächste Jahrtausend; Übersetzung von Raytheon E-Systems: How to 2000
Autoren	Daniel Aebi	Volker Gruhn; Subhash Chavan	Peter Haase	Dean Sims, Jeanne M. Robinson, Christopher Mc Connell, Eileen Silo, Jay Shapiro, Chad Wilbanks
Berufliche Position / Erfahrung der Autoren	Dozent an der ETH Zürich mit Schwerpunkt Wartung von Anwendungssystemen und Berater.	V. Gruhn: Professor an der Universität Dortmund, Lehrstuhl für Software-Technologie, Fachgebiet Spezifika-tionssprachen; S. Chavan: Chavan Software & Services.	Berater (Inhaber der Peter Haase Consulting); Vorsitzender der Aktion2000 (Deutsche Anwendervereinigung zur Jahr-2000-Umstellung in der IT). E-Mail: peter_haase@acm.org	Keine Ausführungen.
Verlag	Carl Hanser Verlag Kolbergerstraße 22, 81679 München http://www.hanser.de E-Mail: info@hanser.de	Addison-Wesley-Longman Wachsbleiche 7-12, 53111 Bonn http://www.addison-wesley.de E-Mail: info@addison-wesley.de	R. Oldenbourg Verlag Rosenheimer Straße 145 81671 München http://www.oldenbourg.de	Int. Thomson Computer Press Königswinterstr. 418, 53227 Bonn http://www.itp.de E-Mail: itp@itp.de Originalausgabe bei IDG Books Worldwide, 1998
Erscheinungsjahr	1998	1997	1998	1998
Anzahl Beiträge	Monographie	Monographie	Monographie	Keine Zuordnung von Autoren zu Kapiteln ersichtlich.
Seitenumfang	225	192	211	634
Abbildungen / Tabellen	38/0	50/6	15/12	237/72
Source Codes / Pseudo Codes	Programme zur Bestimmung von Schaltjahren, Datumstransformationen und Wochentagsberechnungen werden auf rund 5 Seiten dargestellt.	8 Listings auf rund 10 Seiten.	Keine	Keine
Weitergehende Informationsquellen (WWW / Mailing Lists / User Groups)	65/0/1	0/0/0	51/0/2	Keine Angaben im Buch. WWW-Bookmarks für Jahr 2000-Informationen auf der CD-ROM.
Einige typische Überschriften	• Kontakte zu Externen, "Vendor Management" • Eventualplanung • Zwischenbetriebliche Aspekte, EDI • Musterbriefe • Checklisten	• Technische Lösungen zur Herstellung der Jahr-2000-Tauglichkeit • Der Jahr-2000-Projektplan • Versionskontrolle in Jahr-2000-Projekten • Eine Fallstudie	• Kalendergeschichten • Bilanzielle Bewertung • Die vertragliche Haftung der EDV- und Elektronik-Hersteller • Die EDV nach dem Jahr 2000 • Berufsrecht und Berufsethik • Was folgt daraus?	• Deliverables, Aufgaben und Abhängigkeiten • Einflüsse auf das Geschäft, Phasenrisiken, Erfolgsfaktoren • Schnittstellen • Anwendbarkeit von Werkzeugen • Jahr 2000-Projektdatenbank • Jahr 2000-Risikomanagement
Literaturhinweise	48	23	71	Keine

Schlagwortverzeichnis (Anzahl Stichworte)	Keines	87	172	526
Lesbarkeit / Schriftbild	Gut	Gut	Gut	Sehr gut
Art des Einbandes	Hardcover	Hardcover	Hardcover	Hardcover
ISBN-Nr.	3-446-19335-9	3-8273-1251-5	3-486-24737-9	3-8266-0435-1
Preis	DEM 79.-	DEM 69.90	DEM 98.-	DEM 99.-
Problembeschreibung / Problemursachen / Problemfolgen; "Awareness"	Beschreibung und Beurteilung der Problemsituation auf rund 44 Seiten. Knappe Erläuterung der Problemursachen auf 3 Seiten.	Kapitel zur Bedeutung der "Jahrhundertumstellung". Illustration der Auswirkungen mit mehreren Beispielen aus verschiedenen Branchen.	Rund 40 seitiger Überblick über Problemsituation, -ursachen und -folgen. Darstellung möglicher Konsequenzen betroffener Institutionen (z.B. Banken, Krankenhäuser).	Prägnante Problembeschreibung und Ursachenanalyse. Umfangreiche Ausführungen über "awareness". Betonung des Einbezugs aller Managementebenen, Mitarbeiter, Vertrags- und Geschäftspartner.
Projektmanagement	Betonung der Verantwortung der Geschäftsleitung; Hinweise auf externe Ressourcen, finanzielle Mittel, Personal und Projektplan.	Auf rund 9 Seiten werden Überlegungen zum Projektmanagement, Projektkommunikation und -initialisierung sowie zur Auswahl externer Partner angestellt.	Ausführungen über Zeitplan und Einrichtung eines Projektrahmens. Betonung der Bedeutung des Projektkoordinators.	Ausführungen über die Gestaltung von Projektplänen, Projektbudget sowie personelle Ressourcen.
Projektablauf / Vorgehensweise	6 stufiges Phasenmodell für die Problemlösung: Bewußtseinsbildung / Betroffenheitsanalyse / Umsetzung-Korrektur / Testen / Wiederinbetriebnahme/ Nachkontrolle-Abschlußarbeiten.	8 unterschiedliche Projektphasen werden auf rund 20 Seiten dargestellt. Darstellung der Projektphasen in Gantt-Diagrammen für Gesamtprojekt sowie detailliert für die Testphasen.	Mehrfache graphische Darstellung eines Projektrahmens.	Detaillierte Darstellung von 9 Projektphasen. Für jede Projektphase werden Erfolgsfaktoren angegeben und Risiken erörtert.
Kosten / Budgetierung	Angabe von Kostenfaktoren. Verweis auf globale Kostenschätzungen der Gartner Group und von C. Jones mit Hilfe von Function Points. Budgetschätzungen auf Basis LOC.	Verweis auf Schätzungen und Berechnungen der Gartner Group sowie Auflistung der Gesamtkosten verteilt auf Teilaktivitäten. Betont werden die hohen Kosten der Testphase. Beispiel zur Budgetplanung.	Globale Kostenschätzungen ergänzt durch Angaben für einzelne deutsche Unternehmen (Stand Mitte 1997).	Für jede Projektphase werden prozentuale Kosten- und Zeitschätzungen vorgenommen.
Impact Analysis / Repository	Ausführungen zur Impact Analysis auf 10 Seiten. Keine Hinweise auf Repositories.	Auf die Rolle der Impact Analysis wird an mehreren Stellen eingegangen. Anforderungen und Vorgehen der Impact Analysis werden untersucht. Wenige Zeilen zu Repositories.	Methoden der Impact Analysis auf 18 Seiten. Keine Angaben über Repositories.	Auf 60 Seiten werden Aufgaben und Vorgehensweisen erläutert. Ausführungen über die Anwendbarkeit von Tools. In der deutschen Ausgabe wird der 15 seitige Anhang, der mit "Repository" überschrieben war, mit "Projektdatenbank" überschrieben.
Konversionsphase	Darstellung von Lösungsmöglichkeiten auf 10 Seiten.	Vorgehensweisen und Umstellungstechniken werden auf 30 Seiten dargestellt.	Verschiedene Lösungsansätze werden auf 22 Seiten beschrieben.	Generelle Aussagen zur Konversionsphase im Hauptteil des Buches; konkrete Lösungsansätze werden im Anhang B beschrieben.

Testphase	Kurze Behandlung verschiedener Testarten, der Testorganisation und spezieller Probleme beim Testen.	Sehr knappe Ausführungen auf 3 Seiten.	Darstellung von 8 Testvarianten.	Umfangreiche Ausführungen zu Testplanung und Testausführung auf rund 90 Seiten.
Tools / Tool-Anbieter (zum Vergleich: Year 2000 Home Page [deJa98] verweist im August 1998 auf 130 Anbieter und rund 2000 Tools)	Ein Kapitel zum Einsatz von Werkzeugen (10 Seiten). Keine konkreten Tool-Angaben.	Darstellung des Werkzeugeinsatzes in den einzelnen Projektphasen; 14 Werkzeuge verschiedener Anbieter werden (meist in einigen wenigen Zeilen) vorgestellt. Insgesamt 14 Seiten.	Allgemeine Ausführungen zur Auswahl von Anbietern; Verweis auf Marktübersichten. Kurze Ausführungen zur Anwendbarkeit von Tools.	Hinweise über die Beschaffung und Anwendbarkeit von Tools in den jeweiligen Projektphasen. Zusätzliche Beschreibung von Werkzeugkategorien im Anhang.
Service Provider / Beratungsunternehmen	Möglichkeiten und Grenzen von externer Problemlösung werden auf rund 4 Seiten beschrieben.	Überlegungen zur Auswahl externer Partner sowie Darstellung möglicher Organisationsformen auf 5 Seiten.	Rolle externer Berater in Y2K-Projekt; kurze Ausführungen über die Vertrags-situation bei Zusammenarbeit mit externen Beratern.	Kaum angesprochen.
Midrange / PC / LAN; Home-PCs	Rund 15 Seiten zu Problemen bei Arbeitsplatzrechnern.	Fast keine Ausführungen.	Auf einigen Zeilen Ausführungen zu BIOS und RTC.	Eigener Anhang über Y2K-Probleme bei PCs, insbesondere Ausführungen über Microsoft-Produkte.
Embedded Systems	Probleme bei "Embedded Systems" und mögliche Problemlösungen werden auf 5 Seiten angeschnitten.	Keine Ausführungen.	Gestreift bei Darstellung des möglichen "Jahrhundertübergangs" in New York und London (1 Seite).	Knapper Hinweis im Abschnitt über Detailplanung.
Erfahrungsberichte	Keine.	Fallstudie einer US-Versicherungs-gesellschaft auf 26 Seiten.	Allgemeine Ausführungen über Betroffenheit, Situation und Lösungs-möglichkeiten von mehreren Unter-nehmen.	Keine.
Rechtliche Aspekte	Kurze Behandlung auf 3 Seiten ohne Bezugnahme auf konkrete Rechts-ordnung.	Keine Ausführungen.	Beurteilung und Analyse der rechtlichen Situation auf Basis der deutschen Rechtsordnung.	Auf rechtliche und vertragliche Aspekte wird im Anhang auf einer Seite kurz eingegangen.
Hinweise auf ISO 8601	Knapp 2 Seiten	Knapp 3 Seiten	1 Seite	Wenige Zeilen.
Sonstiges	• Hinweis auf KMU • Musterbriefe, Informationsquellen und Checklisten im Anhang • Glossar	• Ausführliche Darstellung strategischer Aspekte des Jahr 2000-Problems • Tabellarische Zusammenstellung von Tool-Anbietern (1 Seite)	• Geschichtliche Informationen über die Entwicklung der Kalender • Informationen zur Bewertung von Software im Jahresabschluß • Einige Bemerkungen zum EURO • Versicherungsfragen • Anwendervereinigungen • Glossar	• Konsequent und sehr detailliert dargestelltes Vorgehensmodell • Ergänzende CD-ROM u.a. mit wenigen WWW-Links und deutsch-sprachigen Präsentationsvorlagen. • Rund 150 Seiten Anhänge
Typisches Zitat	"Es ist kein einfaches Unterfangen, sich ein seriöses Bild der Problemperzeption aus einer globalen Sicht zu machen." (S. 41)	"Es ist auch nicht immer vorteilhaft, mit einem Data Dictionary zu arbeiten, denn dies setzt auch eine saubere Pflege des Dictionary voraus ..." (S. 75) Angabe eines Goethe-Zitats in einem Deutsch geschriebenen Text auf Englisch (S. 81)	"Alle Leser werden gebeten, ihre be-sondere Lage mit ihrem Anwalt für alle im folgenden genannten Problemkreise durchzusprechen." (S. 132)	"Werkzeuge zur Manipulation ausführ-barer Dateien haben nur einen geringen Nutzen, wenn Sie nicht zugleich über den zugehörigen Quellcode des Objekts oder der Datei verfügen, das bzw. die Sie modifizieren wollen." (S. 625)
Gesamtbeurteilung	Gut	Befriedigend	Gut bis befriedigend	Gut

Tabelle 1: Deutschsprachige Bücher zum Jahr 2000-Problem

Titel / Vergleichskriterien	Managing 00: Surviving the Year 2000 Computing Crisis	The Year 2000 Software Problem: Quantifying the Costs and Assessing the Consequences	Year 2000 Software Crisis: Solutions for IBM Legacy Systems	Year 2000 Problem: Strategies and Solutions from the Fortune 100	Solving the Year 2000 Crisis
Autoren	Peter de Jager; Richard Bergeon	Capers Jones	Keith Jones	Leon Kappelman (Hrsg.)	Patrick McDermott
Berufliche Position / Erfahrung der Autoren	Peter de Jager: International Spokesman for the Y2K Problem, Initiator des Information Centers www.year2000.com Richard Bergeon: Vice President der Technology Services von Data Dimensions; Herausgeber des Millennium Journals. E-Mail: pdejager@year2000.com	Gründer und Vorsitzender von Software Productivity Research. Früher u.a. im IBM Labor Santa Teresa und im ITT Programming Technology Center tätig.	Direktor des Worldwide Year 2000 Automation Projects des Institute for Software Quality Automation; Berater.	Professor an der Universität von North Texas; Vorsitzender der Year 2000 Working Group der Society for Information Management. E-Mail: kapp@unt.edu	Inhaber eines Beratungsunternehmens.
Verlag / Herausgebende Institution	Wiley Computer Publishing 605 Third Avenue, New York 10158-0012 http://www.wiley.com/ E-Mail: info@wiley.com	Addison Wesley Longman One Jacob Way Reading, MA 01867-3999 http://www.awl.com/ E-Mail: info@mail.aw.nl	Int. Thomson Computer Press 20 Park Plaza Boston, MA 02116 http://www.itcpmedia.com/ E-Mail: fndit@kiosk.thomson.com	Int. Thomson Computer Press 20 Park Plaza Boston, MA 02116 http://www.itcpmedia.com/ E-Mail: fndit@kiosk.thomson.com	Artech House, Inc. 685 Canton Street Norwood, MA 02062 http://www.artech-house.com/ E-Mail: artech@artech-house.com
Erscheinungsjahr	1997	1998	1997	1997	1998
Anzahl Beiträge	Monographie	Monographie	Monographie	62 Beiträge von 52 Autoren	Monographie
Seitenumfang	226	335	483	447	310
Abbildungen / Tabellen	0 / 6	0 / 64	163 / 20	21 / 74	9 / 52
Source Codes / Pseudo Codes	Keine	Keine	12 Listings auf rund 13 Seiten.	Keine	17 Listings auf rund 15 Seiten.
Weitergehende Informationsquellen (WWW / Mailing Lists / User Groups)	2 / 0 / 0	25 / 0 / 0	27 / 0 / 11	41 / 0 / 45	2 / 0 / 0 .
Einige typische Überschriften	• Houston, We Have a Problem • A Date with Destiny • Bringing Method to the Madness • Taking a Tool Inventory • 2000: A Tool Odyssey • Reading is Fundamental	• Functions Points versus Lines of Codes Metrics for the Year 2000 Problem • Example ROI Calculation • International Year 2000 Repair Effort for 30 Countries • Post-2000 Problem and Recovery Prognosis	• Automated Year 2000 Source Scanning Technology • Automated Year 2000 Project Management Methodology • Automated Year 2000 Software Test Engineering Tools	• Accrued Savings of the Year 2000 Problem • The Role of Accountants and CPAs • Empirical Evidence - Facts on the Fight	• The seven warning signs of Y2K • We do windows • Code archeology • A fool with a tool is still a fool
Literaturhinweise	Keine	95	26	305	39 (am Ende der Kapitel)

Schlagwortverzeichnis (Anzahl Stichworte)	203	379	451	92 (mit zuweilen mehr als 50 Seitenangaben zu einem Schlagwort)	252
Lesbarkeit / Schriftbild	Gut	Gut	Gut	Schlecht	Gut
Art des Einbandes	Paperback	Paperback	Paperback	Paperback	Hardcover
ISBN-Nr.	0-471-17937-X	0-201-30964-5	1-85032-903-6	1-85302-913-3	0-89006-725-2
Preis	USD 15.99	USD 29.95	USD 44.99	USD 44.99	USD 59.-
Problembeschreibung / Problemursachen / Problemfolgen; "Awareness"	Problemerläuterung im Rahmen der Einführung mit Hilfe kleiner Beispiele. Knappe Behandlung der Problemursache. Hinweise auf Bewußtseinsbildung ("wakeup call", "awareness coordinator").	Problembeschreibung und Ausführungen über Ursachen auf 30 Seiten. Angaben zur kostenmäßigen Betroffenheit von Städten (z.B. rund 5 Milliarden DEM für Berlin), Branchen u.a.	Keine Ausführungen über Problemursachen und -folgen. Umfangreiche Informationen über Konsequenzen und Folgekosten, z.T. in Anlehnung an C. Jones.	Kurze Beschreibung des Problems und dessen Ursachen. Darstellung der Konsequenzen und Risiken anhand knapper Beispiele. Empirische Resultate.	Auf 46 Seiten umfangreiche Problembeschreibung. Aufzählung von Symptomen.
Projektmanagement	Hinweise auf Zusammensetzung des Projektteams. Betonung der Bedeutung des Projekt-Managers.	Keine konzentrierten Ausführungen. Einige Hinweise auf die personellen Konsequenzen.	Vorstellung und Verweis auf Projektmanagement-Tools.	Zahlreiche Angaben und Hinweise in Beiträgen verschiedener Autoren.	Ein Kapitel über Strategie- und Projektplanung. Hinweise zur Beschaffung und Nutzung personeller Ressourcen.
Projektablauf / Vorgehensweise	Phasenspezifische Fragen für Zusammenarbeit mit Externen.	Kein konkreter Projektablauf.	Verschiedene Projektabläufe werden graphisch dargestellt.	1 Beitrag mit Projektphasen: Awareness and Acceptance / Inventory and Impact Analysis / Planning and Scheduling / Conversion / Testing / Implementation (S. 88 ff).	6 stufiger Projektablauf: Assessment / Strategy / Repair / Test / Implement / Postimplementation.
Kosten / Budgetierung	Allgemeine Kostenschätzungen anhand von Lines of Code. Bedeutung der verbleibenden Zeit als Kostenfaktor.	Umfangreiche Analysen mit detaillierten Kostenschätzungen auf Basis von Programmiersprachen, Staaten und Branchen. Vergleich zwischen LOC- und Function-Point-Methode als Basis der Kostenschätzungen.	Tabellarische Darstellung der Y2K- Kosten in Anlehnung an C. Jones (1997). Budgetschätzungen anhand LOC. Beschreibung eines Kostenschätzungsprogramms.	Kostenschätzungen (USA/weltweit) nach verschiedenen Ursachen. Hinweise zur Budgetierung. Empirische Resultate.	Kostenvergleich verschiedener Lösungsmethoden.
Impact Analysis / Repository	Fragenkatalog zur Impact Analysis. Ausführungen über Repositories auf 2 Seiten.	Keine Hinweise zum Vorgehen der Impact Analysis. Kostenschätzungen zur Anpassung von Datenbanken, Repositories und Data Warehouses.	Rund 30 Seiten über Betroffenheit verschiedener Branchen, aber keine konkreten Ausführungen über Impact Analysis. Knappe Bemerkungen über Repositories in Verbindung mit Werkzeug-Einsatz.	Umfangreiche Ausführungen verschiedener Autoren zu Vorgehensweisen und Methoden. Keine Ausführungen über Repositories. Empirische Resultate.	Keine Ausführungen.

Konversionsphase	Beschreibung von Vorgehensweisen und Lösungen auf 20 Seiten. Fragenkatalog zur Konversionsphase.	Lösungsansätze werden auf ihre kostenverursachende Wirkung untersucht. Hinweise über Vorgehensweisen fehlen.	Umfassendes Kapitel über Umstellungswerkzeuge. Überblick über 7 Tool-Arten und Veranschaulichung je eines Tools. "Conversion tricks" im Anhang auf 9 Seiten.	Zahlreiche unkoordinierte Ausführungen zu Lösungsmethoden. Tabellarischer Vergleich der Methoden.	Detaillierte Beschreibung der Vor- und Nachteile verschiedener Lösungsansätze (über 80 Seiten). Im Anhang kurze Programmlisten zu einzelnen Lösungsmethoden.
Testphase	Betonung der Bedeutung der Testphase. Kurze Beschreibung verschiedener Test-formen.	Ausführungen über Testvorgehen und -formen auf rund 20 Seiten.	Ausführungen zu Vorgehen und Methode beim Testen. Konkrete Beschreibung von 4 Testwerkzeugen. (24 Seiten)	Unkoordinierte Ausführungen über Anforderungen, Vor-gehensweisen und Werkzeuge.	Auf 23 Seiten werden ver-schiedene Testprinzipien und -strategien dargestellt.
Tools / Tool-Anbieter (zum Vergleich: Year 2000 Home Page [deJa98] verweist im August 1998 auf 130 Anbieter und rund 2000 Tools)	Auf 40 Seiten generelle Aus-führungen über verschiedene Typen von Werkzeugen. Keine Beschäftigung mit bestimmten Tools. Nennung von 10 An-bietern	Mit Sternen klassifizierter Return on Investment des Einsatzes von mehr als 125 (sehr unterschiedlichen) "Selected Software Technologies". Keine Ausführungen über Tools.	Vorstellung verschiedener Tools für • Projektplanung • Software Reengineering • Softwareumstellung • Testen • Configuration Management	Ausführliche Darstellung der Einsatzmöglichkeiten von Tools in verschiedenen Projekt-phasen.	Auf 12 Seiten Ausführungen über verschiedene Werkzeugkategorien.
Service Provider / Beratungsunternehmen	Allgemeine und knappe Hin-weise über Service Providers. Liste von in Nordamerika tätigen Beratungsfirmen. Tips zur Gestaltung der Zusammen-arbeit. Phasenspezifische Fra-gen zur Auswahl potentieller Service-Unternehmen.	Wenige Zeilen über die Zusammenarbeit mit Service Providern. 2 Seiten über Outsourcing.	Generelle Aussagen über Zusammenarbeit mit externen Beratern.	Zusammenstellung von Dienstleistern und ihren Angeboten (15 Seiten).	Chancen und Risiken der Zusammenarbeit mit Beratern werden auf 6 Seiten erörtert.
Midrange / PC / LAN; Home-PCs	(Teilweise veraltete) Informa-tionen zur Jahr 2000-Fähigkeit bestimmter PCs und von Micro-soft-Produkten.	1 Seite über PCs.	Knappe Ausführungen über AS/400-Tools.	Je ein Kapitel zu Midrange Systems und PCs.	Ein Kapitel über PCs (6 Seiten). An verschiedenen Stellen Hinweise zu Microsoft-Produkten.
Embedded Systems	Knappe Ausführungen auf rund 2 Seiten.	Knappe Ausführungen auf rund 3 Seiten.	Wenige Zeilen.	Mehrere Kapitel und zahlreiche Hinweise. Liste mit Embedded Systems auf 2 Seiten.	An mehreren Stellen kurze Hinweise auf Embedded Systems.
Erfahrungsberichte	Kurze Erläuterungen über Aktivitäten verschiedener Organisationen.	Fallstudie mit Schwerpunkt Kostenschätzung auf rund 7 Seiten.	Keine.	"War Stories and Tales of Valor" über verschiedene Unternehmen.	Keine.
Rechtliche Aspekte	Forderung nach "Legal Coordinators" im Jahr 2000-Team.	Knappe Hinweise auf Versicherungen, Offenle-gungspflichten, steuerliche Absetzbarkeit.	Darstellung einer Garantie-erklärung für Jahr 2000-fähige Software (4 Seiten). Knappe Hinweise auf die rechtliche Situation.	Kurze Auseinandersetzung mit der Haftung von Softwareher-stellern.	An mehreren Stellen knappe Hinweise auf rechtliche Fragen.
Hinweise auf ISO 8601	Wenige Zeilen.	Vergleich des ISO Standards mit Microsoft- und europä-ischem Datums-Format.	Keine.	Hinweise auf ISO Standard 8601 und andere Standards (10 Seiten).	Wenige Zeilen.

Sonstiges	• Ausführliche Darstellung möglicher Werkzeugunter-stützung.	• Diverse Checklisten (z.B. Anwendungstypen, kritische Software) • Prognosen für die auf 2000 folgenden Jahre • Glossar.	• Checklisten am Ende jedes der 12 Kapitel • Zahlreiche Screenshots und Ausdrucke • Kommentierte Literatur-angaben zu den Grund-lagen von Jahr 2000-Projekten (10 Seiten) • Glossar • CD-ROM u.a. mit Informationen über 200 Service-Anbieter, Vor-gehensmodellen, Pro-grammen etc.	• Mehrere Checklisten • Umfangreiche Resultate empirischer Untersuchungen • Zahlreiche "Short Takes" mit pointierten Zitaten • Glossar • CD-ROM mit Musterbriefen, Programmen, PC Tests etc.	• Ein Kapitel über "Failure points" • Verschiedene kurze Beispielprogramme (in COBOL, C++, Visual Basic) im Anhang.
Typische Zitate	"... the Year 2000 was the best thing that ever happened to the IT industry: It was the catalyst that turned us into profes-sionals." (S. 222)	"Even the proposed four-digit standards for date formats are not fully adequate, since the International Standards Orga-nization (ISO) format and the Microsoft date standard are incompatible, and neither re-cognizes the other's date for-mat as a valid alternative." (S. 21 f)	"Year 2000 is likely to impact more businesses than BIOSes ..." (S. 91)	"An economic analysis of the impact of the decision not to store those two digits of the date indicates that very good business judgement was indeed used!" (S. 54)	"The problem is that there are no standard naming conventions for indicating dates." (S. 107)
Bewertung durch die Leser auf www.amazon.com (Zahl der vergebenen Sterne (max. 5) / Position in Verkaufsrangliste) (alle Angaben as of 1998-09-08)	4.5 / 2358	5 / 6515	5 / 18692	keine Angaben / 26958	4 / 548154
Gesamtbeurteilung	Genügend	Befriedigend bis genügend	Gut	Gut bis befriedigend	Befriedigend

Tabelle 2: Englischsprachige Bücher zum Jahr 2000-Problem

Vergleichende Buchbesprechung

Neue Bücher zum Jahr 2000-Problem

Gerhard Knolmayer, Oliver Klaus

1 Problemstellung

Zahlreiche Prognosen, die Peter de Jager, mittlerweile viel beachteter "International Spokesman on the Year 2000-Problem" Mitte der 90er Jahre abgab, haben sich erfüllt. Eine dieser Vorhersagen war, daß mit der Annäherung zum kritischen Datumssprung die Zahl der diesem Thema gewidmeten Buchveröffentlichungen stark ansteigen würde. Während wir in eine erste vergleichende Besprechung [KnFi97] noch alle bis dahin vorliegenden Buchveröffentlichungen aufnehmen konnten, ist dies rund 1 Jahr später unmöglich geworden: Auf
http://www.year2000.com/y2kbooks.html
waren im September 1998 rund 60 (teilweise noch nicht erschienene)Titel zum Jahr 2000 (Y2K) - Problem angeführt.

Die Auswahl für diese zweite Buchbesprechung wurde unter folgenden Gesichtspunkten getroffen: Zunächst sollte insbesondere die in deutscher Sprache vorliegende Literatur berücksichtigt werden. Bewußt ausgeschlossen haben wir eine detaillierte Besprechung zweier Bücher, die fast ausschließlich rechtliche Aspekte des Jahr 2000-Problems behandeln: Mit [Bart97] liegt eine fundierte, an der Universität Karlsruhe als Dissertation angenommene Veröffentlichung vor, in der insbesondere allgemeine Grundsätze des Schuld- und Handelsrechts auf ihre Konsequenzen im Hinblick auf das Jahr 2000-Problem erörtert werden; dieses Buch wird in dieser Zeitschrift in einer Einzelbesprechung gewürdigt werden. Eine verwandte Veröffentlichung auf Basis kanadischen Rechts mit geringerer wissenschaftlicher Fundierung, aber stärkerer Berücksichtigung der aktuellen Diskussion z.B. um Berichts- und Prüfungspflichten, um die rechtliche Verantwortung der Geschäftsleitungen, um die Auswirkungen auf Firmenzusammenschlüsse und -übernahmen und zu den staatlichen Initiativen liegt mit [Gaht98] vor. Für Nichtjuristen ist [Gaht98] sicherlich besser lesbar als [Bart97], für deutsche Juristen sicherlich [Bart97] hilfreicher.

Eine andere Perspektive als die hier verglichenen Bücher wird auch in [Keye97] entwickelt: Dort werden auf Basis der möglichen ökonomischen Konsequenzen einer ungenügenden Lösung des Jahr 2000-Problems, die den Chefökonomen der Deutschen Bank Securities, Ed Yardeni, bewogen haben, die Wahrscheinlichkeit für das Eintreten einer Jahr 2000-bedingten Rezession innerhalb des letzten Jahres schrittweise von 30 auf 70% anzuheben [Yard98], (stark überholte) Hinweise zur Gestaltung von Aktien-Portefeuilles im Hinblick auf mögliche Auswirkungen des Jahres 2000 gegeben. Im übrigen umfassen die Anhänge, in denen Aussagen von Kongress-Hearings, Pressemitteilungen und Radiointerviews wiedergegeben werden, rund 2/3 dieses Buches. Ebenfalls ausgeschlossen haben wir mit [YoYo98] ein Buch, in dem Ed Yourdon, ein bekannter Vertreter des Software Engineerings, Mitautor ist. Dieses Buch beschäftigt sich überwiegend mit möglichen Auswirkungen des Jahr 2000-Problems auf verschiedene Branchen und ist auf führenden Rängen verschiedener Bestseller-Listen gelandet (vgl.
http://www.yourdon.com/books/fallback/fallbackhome.html). Eine 2. Auflage wird für

Januar 1999 erwartet.

Regionale Aspekte stehen in den Büchern der kanadischen Task Force Year 2000 [TFY298] und in [Good97] für Neuseeland im Vordergrund. Ein zu [TFY298] vergleichbares Buch, das sich vor allem an KMU wendet, fehlt im deutschen Sprachraum. Die in Ordnerform präsentierte Veröffentlichung [Budl97] wurde deshalb nicht in den Vergleich einbezogen, weil in ihr primär COBOL-Listings angeführt werden; diese Veröffentlichung mag für Wartungsprogrammierer hilfreich sein. Schließlich ist die mehr als 900 Seiten umfassende Veröffentlichung [Lefk98] eine schwierig überblickbare, wenngleich in einzelnen Beiträgen durchaus lesenswerte Zusammenstellung verschiedenster Ausführungen in mehr als 100, oft bereits zuvor veröffentlichten Kurzbeiträgen.

Angesichts der hohen Dynamik in diesem Gebiet der Wirtschaftsinformatik ist unstrittig, daß das Internet mit seinen Mailing-Listen [Knol97c] und unzähligen WWW-Präsentationen [Knol97b] wesentlich mehr Detailinformationen bietet als Veröffentlichungen in Buchform. Aus diesem Grund sind die Buchveröffentlichungen besonders für jene Personen wertvoll, die entweder mit der überwiegend in englischer Sprache präsentierten WWW-Information sprachliche Probleme besitzen oder über keinen Zugang zum Internet verfügen. Aus Gründen der Aktualisierbarkeit sind einige Veröffentlichungen, die man sich gut in Buchform vorstellen könnte und auch als Bücher bezeichnet werden, online verfügbar; dazu gehören vor allem die IBM-Präsentation [IBM98] und [Yard98], in der Ed Yardeni vielfältige, nicht-technische Überlegungen zum Jahr 2000-Problem zusammenfaßt.

Einen Überblick zur fast wöchentlich umfangreicher werdenden englsichsprachigen Buchliteratur zum Jahr 2000-Problem gewinnt man durch Zugriff auf folgende WWW-Seiten:

http://www.year2000.com/y2kbooks.html
http://www.year2000.com/y2kbooks-reviews.html
http://www.it2000.com/library/bookshelf.html
http://www.yardeni.com/books/y2k.html
http://www.yourdon.com/books/coolbooks/coolby2000.html
http://www.amazon.com/   mit Suche nach „Year 2000 date conversion"

2   Vorbemerkungen zum Inhalt der Veröffentlichungen

Eigenschaften der unter den oben dargestellten Gesichtspunkten ausgesuchten Veröffentlichungen sind in Tabelle 1 und 2 zusammengestellt. Da eine Strukturierung in zwei Tabellen erforderlich war, haben wir die deutschsprachige und die englischsprachige Literatur in je einer Tabelle verglichen.

Auf die hohe Dynamik der Sichtweisen zum Jahr 2000-Problem wurde bereits hingewiesen. Während zu Beginn der Diskussion um das Jahr 2000-Problem vor allem Informationssysteme im Vordergrund standen [Knol97a], hat sich in den letzten Monaten die Besorgnis um „Embedded Systems" stark erhöht, in denen Chips enthalten sind, die Zeitinformationen speichern und/oder verarbeiten. In diesen Chips sind die Jahresangaben oft zweistellig kodiert. Während man ursprünglich annahm, daß die in ihnen gespeicherte Zeit nur in wenigen Fällen genutzt würde, zeigt eine

Untersuchung des amerikanischen Departments of Defence, daß von rund 200000 untersuchten „Communication and Facility Devices" fast 30 % nicht Jahr 2000-fähig gewesen sind. Derartige Ergebnisse haben zu Befürchtungen geführt, wonach die gesamte technische Infrastruktur unseres Zusammenlebens durch Fehlfunktionen von „Embedded Systems" insbesondere im Bereich der Elektrizität-, Wasser- und Telekommunikationsversorgung, der Medizintechnik sowie der Verkehrssysteme gefährdet sei.

Innerhalb der Beschäftigung mit Informationssystemen treten in der aktuellen Diskussion insbesondere Fragen zur Gestaltung der inner- und überbetrieblichen Tests, mit deren Durchführung rund 50% der Gesamtaufwendungen eines Jahr 2000-Projektes verbunden sein dürften, sowie der Kontingenzplanung in den Vordergrund. Es ist bemerkenswert, daß die meisten der im folgenden besprochenen Bücher die angesprochenen Themen nur sehr beschränkt aufgreifen. Die Lektüre der Bücher kann daher primär als Einstieg und Überblick zur Jahr 2000-Thematik dienen, ersetzt aber nicht einen Erfahrungsaustausch zwischen fachkundigen Kreisen, wie er z.B. in Anwendervereinigungen gepflegt wird [Knol97c].

3   Einzeldarstellung der Buchveröffentlichungen

3.1   Daniel Aebi:
Zeitsprung 2000

Der Autor hat über das Re-Engineering von Informationssystemen dissertiert und verfügt auch über praktische Erfahrungen mit der Lösung des Jahr 2000-Problems. Die Dezimalgliederung kommt mit zwei Stellen aus und ist nicht immer streng strukturiert. Das Buch beginnt mit einem Prolog, in dem Dieter W. am ersten Arbeitstag des Jahres 2000 erwacht und mit einer Reihe von (letztlich undramatischen) Unannehmlichkeiten konfrontiert wird. In der Einführung folgen Bilder von der fehlerhaften Berücksichtigung des Zeitsprungs durch MS-DOS und durch einen Videorecorder sowie mit zweistelligen Jahresangaben versehene Kreditkarten und mit dem Präfix 19 versehene Papierdokumente. Immerhin findet sich aber auch eine Abbildung, die zeigt, wie eine der ISO Norm 8601 entsprechende Zeitangabe unter Windows NT einzustellen ist.

Nach dieser eher populär ausgerichteten Einführung beschäftigt sich der Hauptteil des Buches mit betrieblichen und technischen Aspekten der Problemlösung. Dieser Teil vermittelt dem Leser einen brauchbaren Überblick; insgesamt kann vor allem der in KMUs tätige Praktiker von der Lektüre profitieren.

3.2   Volker Gruhn; Subhash Chavan:
Das Jahr 2000 Problem:
Projektmanagement von Datumsumstellungen in der Software

Die Gliederung dieses Buches beschränkt sich ebenfalls auf zwei Dezimalstellen und ist auch inhaltlich unbefriedigend. So wird kein Vorgehensmodell verwendet, um die bei der Problemlösung anfallenden Aufgaben übersichtlich zu strukturieren. Vor allem die wichtigen Testaufgaben kommen viel zu kurz. Die Fallstudie beschäftigt sich mit einem amerikanischen Versicherungsunternehmen, das Software in Indien umstellen läßt und ist damit auf europäische Gegebenheiten nur bedingt übertragbar.

3.3 Peter Haase:
Das Jahr 2000 in der EDV:
Bewältigung des Jahr-2000-Problems in Ihrem Unternehmen

Das Buch ist insoweit originell gegliedert, als nach einem einführenden Kapitel 6 weitere Kapitel folgen, in denen Gefahren und Chancen der EDV vor, im und nach dem Jahr 2000 dargestellt werden. Wenig Systematik zeigt sich, wenn z.B. ein Abschnitt „Das Jahrhundertereignis in der EDV" in die Unterabschnitte

- Presseschau
- Virenbefall
- Wortwahl
- Globale Kosten
- Was folgt daraus?

unterteilt wird. Etwas penetrant wirken die 21 mit „Was folgt daraus?" überschriebenen Abschnitte.

Das Buch gibt den üblichen Überblick, ohne sich zu sehr mit technischen Details zu beschäftigen. Besonderheiten sind der Exkurs in die Geschichte der Kalendersysteme und die vergleichsweise umfangreichen Ausführungen zu rechtlichen Gesichtspunkten.

3.4 Dean Sims et al.:
Bewältigung des Jahr-2000-Problemes in ihrem Unternehmen:
Die Jahr-2000 Lösung: So schafft Ihre Software den Sprung ins nächste
Jahrtausend (Übersetzung von Raytheon E-Systems: How to 2000)

Das letzte der in deutscher Sprache vorliegenden Bücher ist eine Übersetzung aus dem Amerikanischen und erheblich umfangreicher als die bisher skizzierten Bücher. Wenngleich sich auch hier die Gliederung auf 2 Dezimalstellen beschränkt, ist doch die systematische und straffe Ausrichtung an einem professionell entwickelten, neun Phasen umfassenden Vorgehensmodell eindrucksvoll. Diese Selbstbeschränkung und die Gleichartigkeit des Kapitelaufbaus machen das Buch in der Lektüre trocken, sind aber für eine Projektarbeit sehr hilfreich. Dazu kommen fundiert geschriebene Anhänge. Insgesamt ist das Buch für in Wartungsprojekte involvierte Leser empfehlenswert. An einigen Stellen ist die Übersetzung nicht gut gelungen.

3.5 Peter de Jager; Richard Bergeon:
Managing 00: Surviving the Year 2000 Computing Crisis

Peter de Jager kommt das Verdienst zu, insbesondere seit 1993 intensiv auf die Bedeutung des Jahr 2000-Problems hingewiesen und den öffentlichen Bewußtseinsprozeß erheblich gefördert zu haben. Die Ausführungen in diesem unübersichtlich gegliederten Buch geben einen Überblick über die allgemeine Diskussion zum Jahr 2000-Problem. Details zum Management von Jahr 2000-Projekten oder den einsetzbaren Methoden findet der Leser in diesem Buch nicht.

3.6 Capers Jones:
The Year 2000 Software Problem: Quantifying the Costs and Assessing the Consequences

Capers Jones hat die detailliertesten Berechnungen über mögliche Kosten der Lösung des Jahr 2000-Problems vorgenommen. Auch in diesem Buch sind nur die Kapitelüberschriften numeriert; zwei Kapitel sind gar nicht weiter unterteilt und umfassen nur die Überschrift und Literaturhinweise. Die Sequenz der Kapitel ist nicht immer nachvollziehbar, das Thema der Kostenschätzungen zieht sich durch das ganze Buch. Insgesamt gibt das Buch kaum Hilfen für in Jahr 2000-Projekten tätige Mitarbeiter.

3.7 Keith Jones:
Year 2000 Software Crisis: Solutions for IBM Legacy Systems

Nach 3 einführenden Kapiteln folgen 11 Kapitel, in denen die Einsatzmöglichkeiten und -grenzen von Tools in 8 Projektphasen beschrieben werden. In den Hauptkapiteln wird eine einheitliche Darstellungsweise verwendet. Wenngleich die zur Illustration herangezogenen Werkzeuge eine gewisse Willkürlichkeit aufweisen, wird doch ein detaillierter Eindruck von den Möglichkeiten und Grenzen werkzeuggestützter Vorgehensweisen vermittelt. Insgesamt gibt das Buch eine fachlich fundierte und für Projektmitarbeiter hilfreiche Darstellung.

3.8 Leon Kappelman:
Year 2000 Problem: Strategies and Solutions from the Fortune 100

Der Herausgeber dieses Buches, Leon Kappelman, ist ebenfalls eine „Gallionsfigur" der amerikanischen „Jahr 2000-Szene". Er hat sich insbesondere durch sein frühes Engagement in der Society for Information Management verdient gemacht. Zu den Autoren des Buches zählen auch P. de Jager und C. Jones, deren Bücher in diesen Vergleich ebenfalls einbezogen worden sind.

Wie bei einem Sammelband zu einem derart aktuellen Thema nicht anders zu erwarten, ist das Niveau der einzelnen Beiträge sehr unterschiedlich; es ergeben sich Überschneidungen und Lücken zwischen den teilweise bereits anderswo veröffentlichten Beiträgen, von denen manche in einer sehr populären, zuweilen martialischen Diktion verfaßt sind. Durch das Buch zieht sich das Akronym DRAGON für „Date-Related Abend, Garbage, Or Nothing". Mehrere Beiträge liefern interessante Hintergrundinformationen. Für Projektmitarbeiter ist das Buch nur bedingt hilfreich.

3.9 Patrick McDermott:
Solving the Year 2000 Crisis

Dieses Buch umfaßt 25 Kapitel sowie einen Epilog, was der Übersichtlichkeit nicht gerade förderlich ist: So werden etwa „Fix-it" factories vor der Projektplanung und der „Business perspective" dargestellt, die Buchteile sind teilweise arabisch und zum Teil römisch numeriert. Das Kapitel über Werkzeuge findet man im Teil „The people", das Kapitel „PCs and desktops" im Teil „The business". Gleichwohl sind die

Ausführungen handfester als in der „Unterhaltungsliteratur" zum Jahr 2000. Wegen des unübersichtlichen Aufbaus ist das Buch aber nur bedingt geeignet.

4 Zusammenfassung

Die meisten der zum Jahr 2000-Problem verfaßten Bücher sind nicht für Informatik-Spezialisten geschrieben, sondern wenden sich eher an eine breitere, an technischen Fragen interessierte und von dem Problem potentiell betroffene Öffentlichkeit. Vergleicht man die in [KnFi97] beschriebenen mit den hier besprochenen Veröffentlichungen, so zeigt sich, daß das durchschnittliche Niveau der Darstellungen ziemlich gleich geblieben ist. Die in deutscher Sprache verfügbaren Bücher entsprechen im Niveau etwa jenem der englischen Werke. Generell gewinnt man den Eindruck, die meisten Bücher seien „mit heißer Nadel" und unter starker Einbeziehung des Informationsaustausches am WWW geschrieben worden.

In unserem ersten Vergleich schnitt das Buch von Ulrich und Hayes [UlHa97] am besten ab; dieses Buch würde auch 1998 in der „Spitzengruppe" der Veröffentlichungen liegen. Für die Gestaltung der Projektarbeit scheint uns allerdings das hier besprochene Buch von Sims et al. am besten geeignet zu sein. Deutschsprachige Leser, die einen allgemeinen Überblick über die IV-Aspekte des Problems suchen, können auf die Bücher von Aebi und Haase verwiesen werden.

5 Ausblick

Die hohe Dynamik und der intensive Erfahrungsaustausch, der über das WWW erfolgt, macht die Grenzen traditioneller Formen der Print-Veröffentlichung deutlich. Natürlich kann das WWW die traditionellen Veröffentlichungsformen nicht global ersetzen, aber es wird Segmente substituieren oder ergänzen. Die hohe Aktualität des Jahr 2000-Problems und die überragende Aufmerksamkeit, die ihm im WWW geschenkt wird, machen dieses Thema zu einem derartigen Segment.

Hochschullehrer werden sich in Zukunft vermehrt die Frage stellen müssen, ob nicht auch Dissertationen ein derartiges Segment bilden und ob nicht die Veröffentlichung in Buchform angesichts der komparativen Vorteile des WWW für die Scientific Community und die an wissenschaftlichen Ergebnissen interessierte Praxis einen Anachronismus darstellt.

Literatur

[Bart97]   Bartsch, M., Software und das Jahr 2000, Haftung und Versicherungsschutz für ein technisches Großproblem, Baden-Baden 1998.
[Budl97]   Budlong, M., COBOL, Dates, and the Year 2000, Tujunga: King Computer Service 1997.
[deJa98]   The Year 2000 Information Center / Millennium Bug www.year2000.com
[Gath98]   Gathan, A.M., The Year 2000 Computer Crisis Legal Guide, Ontario 1998.
[Good97]   Good, J., A Survivor's Guide to the Year 2000 Problem, A Non-technical Handbook for All New Zealanders, Christchurch 1997.

[IBM98]   IBM (Ed.), The Year 2000 and 2-Digit Dates: Guide, GC28-1251-08
             [1998-01-23]
             http://ppdbooks.pok.ibm.com:80/cgi-
             bin/bookmgr/bookmgr.cmd/BOOKS/y2kpaper/CCONTENTS

[Keye97]   Keyes, T., The Year 2000 Computer Crisis: An Investor's Survival
             Guide, Brookeville 1997.

[KnFi97]   Knolmayer, G., Fisler, D., Erste Buchveröffentlichungen zur Lösung des
             Jahr 2000-Probelms. In: Wirtschaftsinformatik 39 (1997) 4, S. 418-426.

[Knol97a]   Knolmayer, G., Das Jahr 2000-Problem: Medien-Spektakel oder
             Gefährdung der Funktionsfähigkeit des Wirtschaftssystems?, in:
             Wirtschaftsinformatik 39 (1997) 1, S. 7-18.
             http://www.iid.de/jahr2000/j2p/

[Knol97b]   Knolmayer,G., Das Jahr 2000-Problem im Internet. In:
             Wirtschaftsinformatik 39 (1997) 1, S. 73-76.

[Knol97c]   Knolmayer, G., User Groups zur Lösung des Jahr 2000-Problems. In:
             Informatik/Informatique 4 (1997) 6, S. 11-17.

[Lefk98]   Lefkon, D. (Hrsg.), Year 2000: Best Practices for the Y2K Millennium
             Computing: Panic in The Year Zero, o.O.: AITP-SIG Mainframe 1998.

[TFY298]   Task Force Year 2000 (Ed.), A Call for Action: Report of Task Force
             Year 2000, Ottawa 1998.
             http://strategis.ic.gc.ca/SSG/yk04576e.html

[UlHa97]   Ulrich, W.M., Hayes, J.S., The Year 2000 Software Crisis, Challenge of
             the Century, Upper Saddle River 1997.

[Yard98]   Yardeni, E., YEAR 2000 RECESSION?, Netbook Version 7.2.,
             1998-07-02.
             http://www.yardeni.com/y2kbook.html

[YoYo98]   Yourdon, E., Yourdon, J., Time Bomb 2000 : What the Year 2000
             Computer Crisis Means to You!, Old Tappan 1998.

5. Design Extraction with UML

Lecture by Stéphane Ducasse

Design Extraction - Purpose of this lecture

❏ Design is not code with boxes and arrows

❏ Design extraction is not trivial

❏ Design extraction should scale up

❏ Design extraction can be supported by computers but not fully automatize

❏ Give a critic view on hype: "we read your code and produce design"

❏ Fertilize you with some basic techniques that may help you

❏ Show that UML is not that simple and clear

Outline

- ❑ Why
- ❑ UML Basics (What is design?
- ❑ Basic UML static elements (operations attributes and associations)
- ❑ Displaying code with boxes is not design
- ❑ Languages specific issues
- ❑ Tracks for extraction
- ❑ Case studies

Intro

...

<u>Design In Reengineering LifeCycle</u>

Problem Analysis
(Graphs, Metrics...) $\longrightarrow$ Solution

Views

Design Extraction

Code Model (meta model)

Integration

Code representation
and filtering

Code

Code
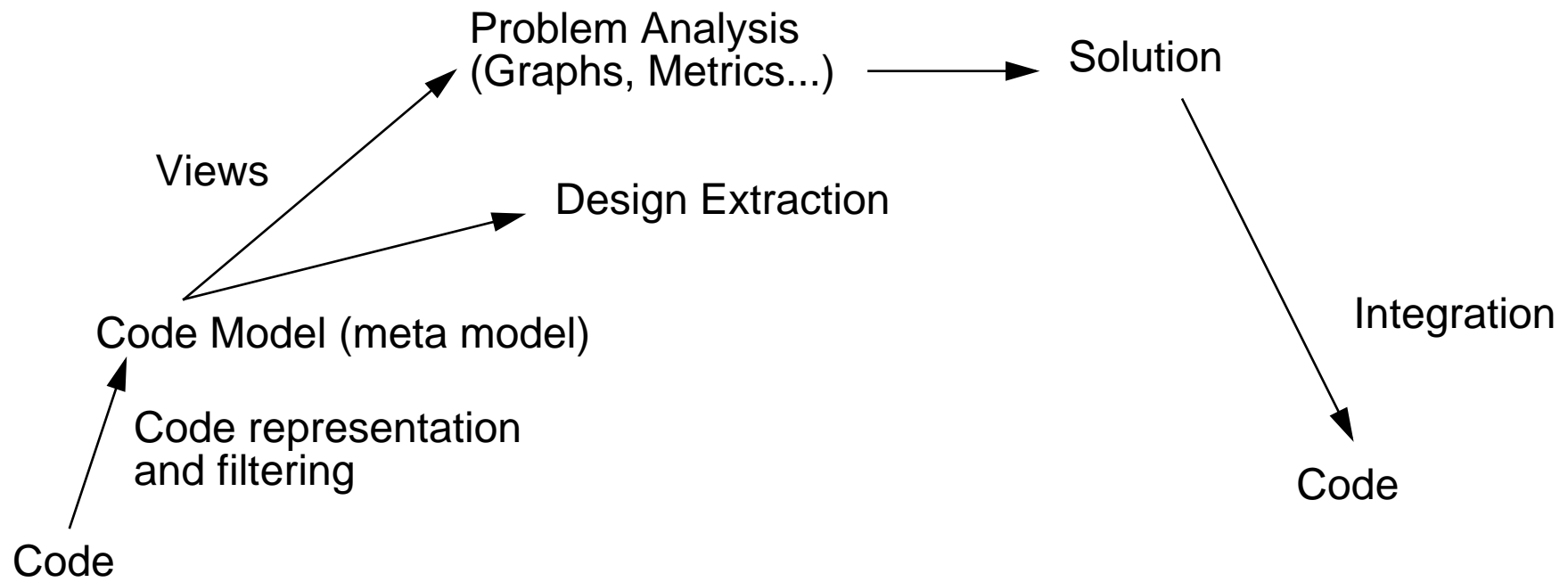
Design In Forward Engineering

Waterfall
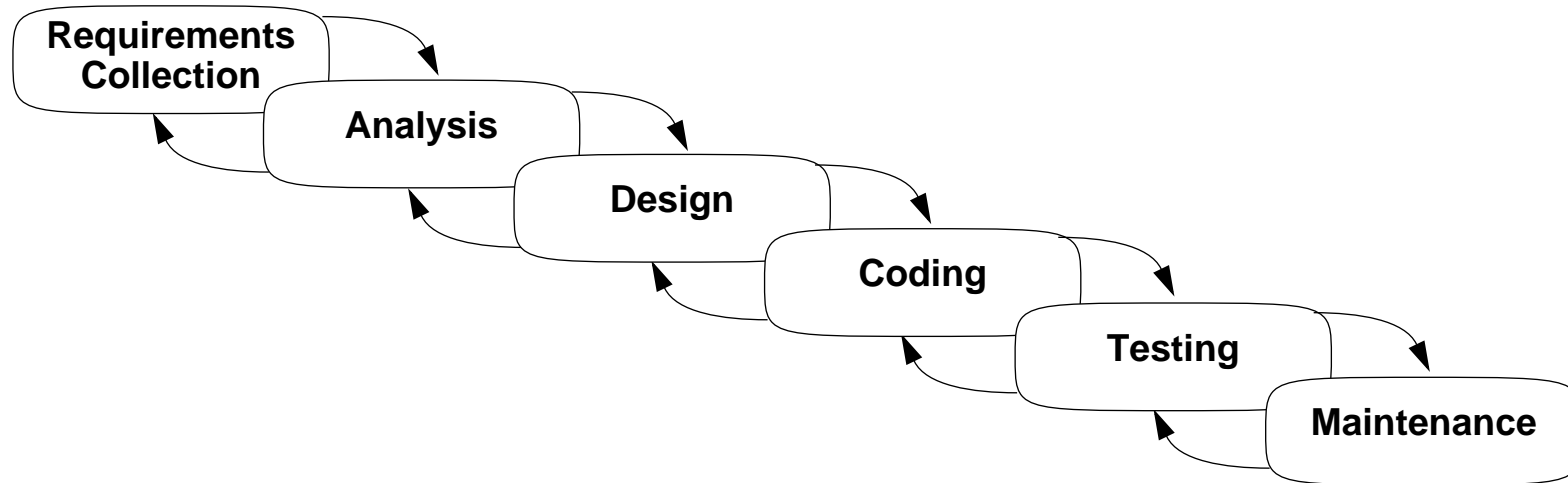
Quote Golberg "..."

Why Design Extraction is needed?

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

Why UML?
- ❑   Standard
- ❑   Communication based on a common language
- ❑   Can support documentationif we are precise about its interpretation
- ❑   (Hype and market)

UML (Unified Modeling Language)

What is the Unified Modeling Language?

- Successor of OOAD&D methods of late 80 & early 90
- Unifies Booch, Rumbaugh (OMT) and Jacobson
- Currently standardized by OMG

- UML = a modeling language and not a methodology (no process)

- UML defines

- a notation (it is the syntax of the modeling language)

Ex:

Customer
name address
creditRating(): String
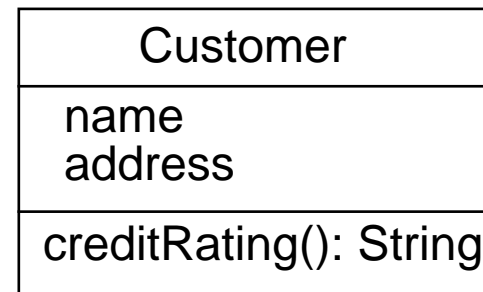
- a meta-model = a model to define the "semantics" of a model
(what is well-formed), defines in itself but weak!

Terminology

UML	Class	Association	Generalization	Aggregation
Booch	Class	Has	Inherits	Containing
Coad	Class & Object	Instance connection	Gen-Spec	Part-Whole
Jacobson	Object	Acquaintance Association	Inherits	Consist of
Odell	Object Type	Relationship	Subtype	Composition
Rumbaugh	Class	Association	Generalization	Aggregation
Shlaer/Mellor	Object	Relationship	Subtype	N/A

Levels of Interpretations: Perspectives

UML purists do not propose different levels of interpretation, they refer to the UML semantics. Perspectives are not part of the formal UML

However levels of interpretation are valuable.

For example:

- ❑ What would be the sense of representing inheritance (subclassing) between two classes using generalization (subtyping)?

  > Dictionary is a subclass of Set in Smalltalk (subclassing)

  > but a Dictionary is not a subtype nor generalization of Set

  > See stereotypes

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 (lot of design patterns are based on interfaces and delegation)

- ❑ Implementation: implementation classes

UML Static Elements

...

Static Class Diagram Elements

A class diagram describes the type of the objects and the various kinds of static relationships that exist among them.

Two main static relationships

- ❑   Association

    ex: a customer rents a video

- ❑   'Is-a'

    ex: a nurse is a kind of person

A class diagram defines:

- attributes
- operations
- constraints that apply to the way objects are connected
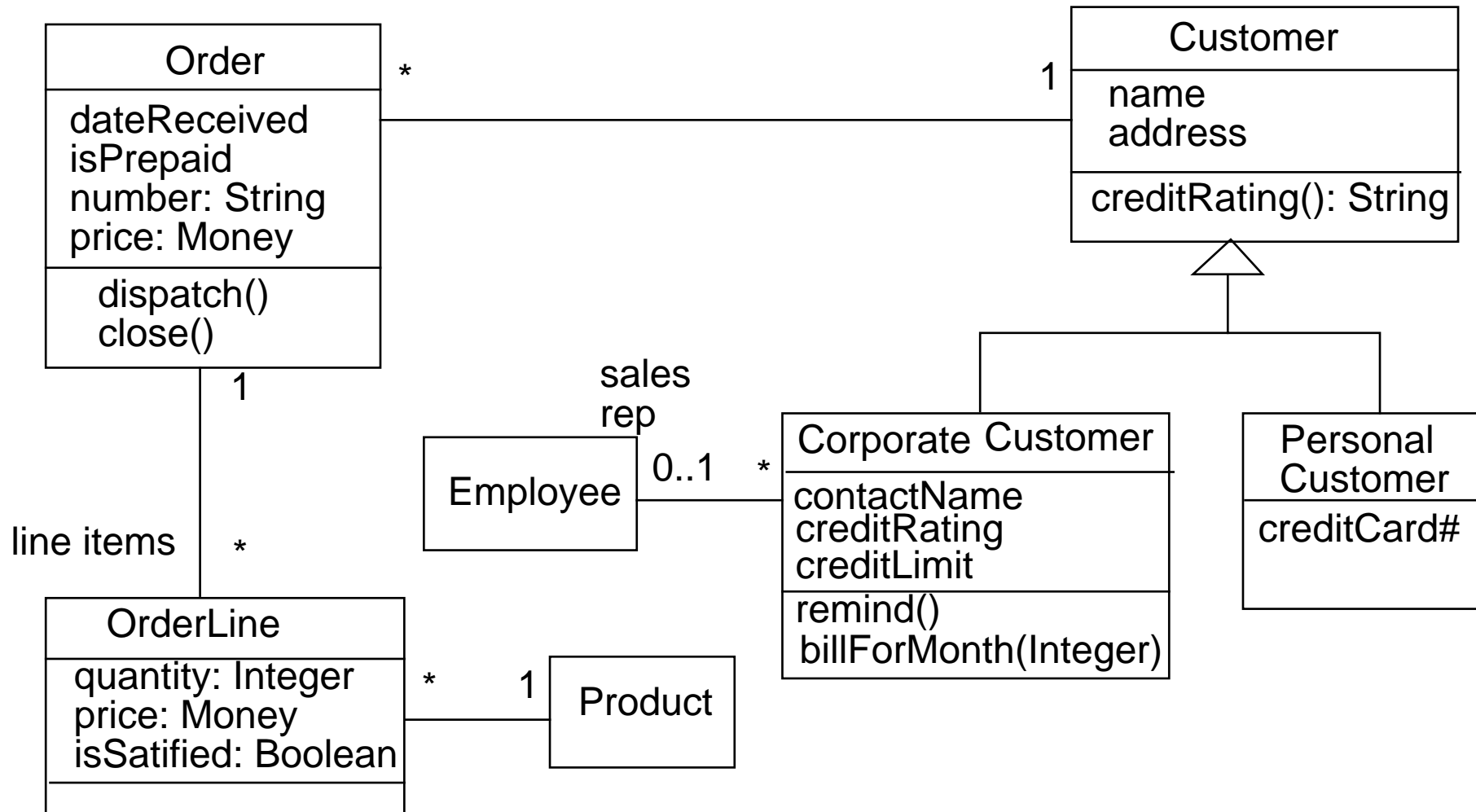
Attributes

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

Can be approximate to methods.
- Conceptual: principal functionality of the object (CRC Cards),
    in design phase, it is often better to use a sentence that describes the functionality
- Specification: public methods on a type
- Implementation: methods

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

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
    - Reuse Contract: To explicit method interdependencies and evaluate impact
    on subclassing and specialising. OCL (Object Constraint Language) Extension:
    You can specify which other methods a method invokes (reuse contracts)
        accept: aPacket {invokes send:}

Associations

- Represents relationships between instances

- Each association has two roles: each role is a direction on the association.
- a role can be explicitly named, labeled 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

Order
dateReceived isPrepaid number: String price: Money
dispatch() close()
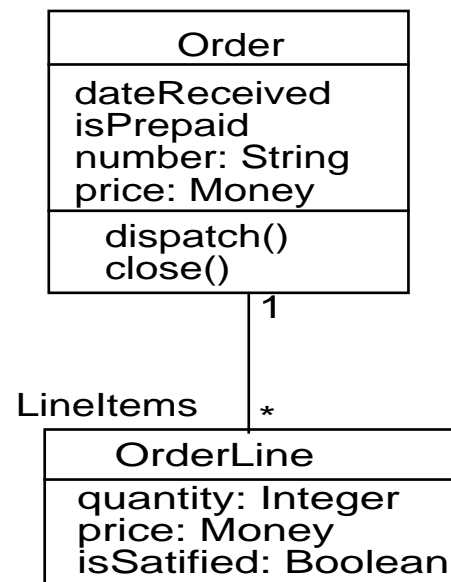
LineItems = role of direction Order to OrderLines
LineItems role = OrderLine role
One Order has several OrderLines

1

LineItems   *

OrderLine
quantity: Integer price: Money isSatified: Boolean

Associations: Conceptual Perspective

Conceptual Perspective: associations represent conceptual relationships between classes

```
┌─────────────────────┐                              ┌─────────────────────────┐
│       Order         │ *                          1 │       Customer          │
├─────────────────────┤                              ├─────────────────────────┤
│ dateReceived        │──────────────────────────────│ name                    │
│ isPrepaid           │                              │ address                 │
│ number: String      │                              ├─────────────────────────┤
│ price: Money        │                              │ creditRating(): String  │
├─────────────────────┤                              └─────────────────────────┘
│ dispatch()          │
│ close()             │ 1
└─────────────────────┘
          │
          │ *
┌─────────────────────┐
│     OrderLine       │
├─────────────────────┤      *      1  ┌─────────────┐
│ quantity: Integer   │───────────────│   Product   │
│ price: Money        │               └─────────────┘
│ isSatified: Boolean │
└─────────────────────┘
```
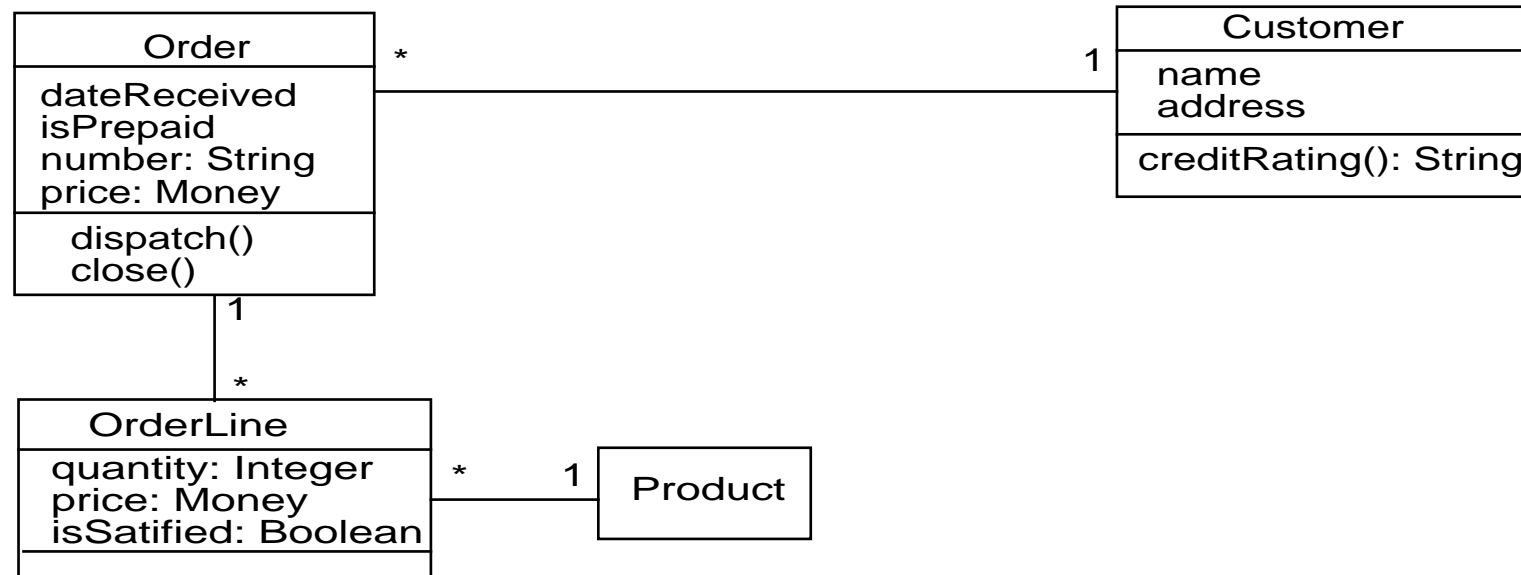
An Order has to come from a single Customer.

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

```
┌─────────────────────┐              ┌─────────────────────┐
│        Order        │ *          1 │      Customer       │
├─────────────────────┤──────────────├─────────────────────┤
│                     │              │                     │
└─────────────────────┘              └─────────────────────┘
```
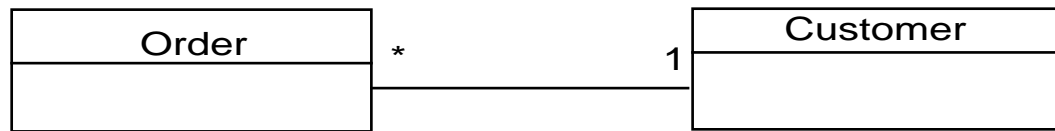
Implications:

- 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

<u>*Arrows: Nagivability*</u>

```
┌──────────────────────┐  *                    1  ┌────────────────────────────┐
│        Order         │─────────────────────────▶│         Customer           │
├──────────────────────┤                          ├────────────────────────────┤
│ dateReceived         │                          │ name                       │
│ isPrepaid            │                          │ address                    │
│ number: String       │                          ├────────────────────────────┤
│ price: Money         │                          │ creditRating(): String     │
├──────────────────────┤                          └────────────────────────────┘
│ dispatch()           │
│ close()              │
└──────────────────────┘ 1
          │
          │ *
┌──────────────────────┐
│      OrderLine       │  *       1  ┌──────────────┐
├──────────────────────┤────────────▶│   Product    │
│ quantity: Integer    │             └──────────────┘
│ price: Money         │
│ isSatified: Boolean  │
└──────────────────────┘
```
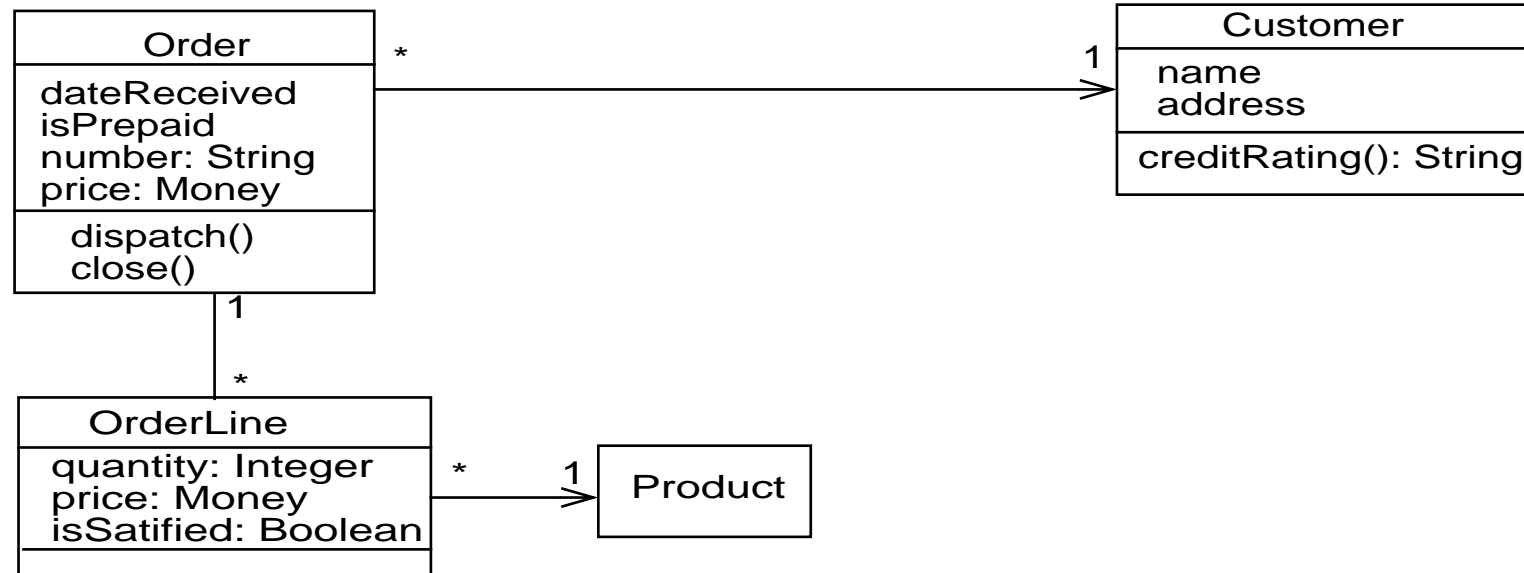
No arrow = navigability in both sides or unknown (so conventions!)

- 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

Attributes vs. Associations

Conceptual: No real difference. Attribute is another convenient notation association may be optional, attribute don't

Specification and Implementation:

      - attributes imply navigability from the type to the attribute only

      - Fowler: "attribute imply that the type contains its own copy of the attribute object. Type used as attribute has value rather than reference semantics." Not true in all the languages!!!

Generalization

UML semantics only supports generalization and not inheritance.

Subtyping is not subclassing!

So we should interpret it.

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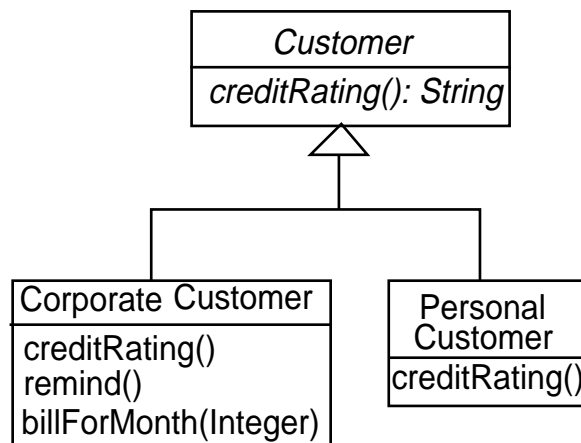
Conceptual: What is true for an instance of a superclass is true for a subclass (associations, attributes, operations).

Corporate Customer is a subtype of Customer

```
Corporate Customer
  creditRating()
  remind()
  billForMonth(Integer)
```

```
  Personal
  Customer
  creditRating()
```

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.

Subclassing is one way to implement generalization.

Abstract classes and Interfaces

Abstract classes and methods are in italic

Abstract classes depending on the language defined only empty methods (Java, C++) or incomplete class behavior (Smalltalk, Eiffel)

- Smalltalk:

```
self subclassResponsibilities
```

- C++:

```
virtual ~Board(void) =0;
```

- Java: abstract

- Eiffel: deferred method

Differences between interfaces and abstract classes:

Interface does not support the definition of templates methods, i.e. do not support the definition of behavior and relationships between methods. Interface are purely methods declaration.

```
┌─────────────────────────┐
│        Customer         │
├─────────────────────────┤
│  creditRating(): String │
└─────────────────────────┘
             △
    ┌────────┴────────┐
┌──────────────────────┐   ┌────────────────┐
│  Corporate Customer  │   │    Personal    │
├──────────────────────┤   │    Customer    │
│ creditRating()       │   ├────────────────┤
│ remind()             │   │ creditRating() │
│ billForMonth(Integer)│   └────────────────┘
└──────────────────────┘
```
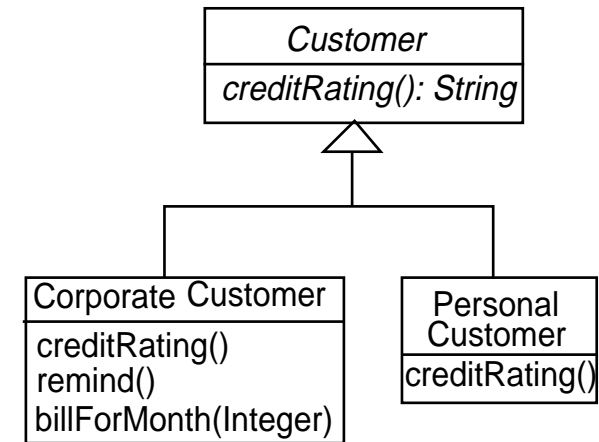
<u>*Experimentations*</u>

...

Extraction without Analysis Works for Toy

Let us do it with a Smalltalk example

```
Node          ('initialize-release' #initialize)
              ('sending/receiving' #accept: #send:)
              ('printing' #printOn:)
              ('private' #name #name: #nextNode #nextNode:)
              ('accessing' #hasNextNode)
Workstation   ('sending/receiving' #accept:)
              ('originating' #originate:)
Printer       ('printing' #print:)
              ('sending/receiving' #accept:)
FileServer    ('sending/receiving' #accept:)
              ('saving' #save:)
Packet        ('printing' #printOn:)
              ('accessing' #isAddressedTo: #isOriginatingFrom:)
              ('initialize-release' #initialize)
              ('private' #addressee #addressee: #contents #contents: #originator #originator:)
```

Extraction without Analysis Works for Toy

A tiny LAN simulator in Smalltalk

Node

name : String
nextNode : Node

send: (p Packet) : Node
accept: (p Packet) : Node
printOn: (s Stream) : Node
name (): String
hasNextNode (): Boolean
initialize (): Node
name (n String) : Node
nextNode(): Node
printOn: (s Stream): Node
nextNode: (n Node) : Node
withName: (s String): Node
withName:nextNode:
  (s String, n Node) : Node

Packet

contents: String
addressee: Symbol
originator: Node

isAddressedTo: (p Packet) : Boolean
isOriginatedFrom: (p Packet) : Boolean
printOn: (s Stream) : Packet
contents () : String
contents: (s String) : Packet
addressee() : Symbol
addressee:(a Symbol) : Packet
originator(): Node
originator(n Node): Node
initialize (p Packet):Packet

send:to: (s String, n Node): Packet

1

1

Workstation

accept: (p Packet) : Node
originate: (p Packet) : Node

Printer

accept: (p Packet) : Printer
print (p Packet) : Printer

FileServer

save: (p Packet) : Node
accept: (p Packet) : Node

Another Tiny 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

Another Tiny Example in C++

BoardGame

+Boardgame (x Integer,y Integer)
+play(x Integer,y Integer)
+valid(x Integer,y Integer):Integer
+gameOver():Integer
+turn(): PlayerId
+winner():PlayerId
+board(): Board

Gomoku

+Gomoku(x Integer, y Integer, p, Integer)
#checkWinner(x Integer, y Integer)
-checkField (x Integer, y Integer, c Integer)

-myPiecesToWin: Integer

Display

Window

Board

BoardWindow

-myEventMask LongInteger
-myDisplay : Display
-myHandle: Window
-myGC: GC
-myPixmapPix Pixmap
-myPixmaPiece: Piece
-myPixmapCount: Integer
-myWidth: Integer
-myHeight: Integer
-myDepth: Integer
-myWhite:LongInteger

+ BoardWindow(String,Integer, Integer,Integer,Integer,String)
+ ~BoardWindow();
+putPiece(x Integer, y Integer, p Piece)
+putText(x Integer, y Integer, t String)
+clear(Integer,Integer,Integer,Integer)
+getEvent(e Event)
+width():Integer
+height():Integer
-addPiece(p Piece)

GC

Piece

Pixmap

Board

+Board (w Integer, h Integer): Board
+Board (b Board)
+~Board()
+=(b Board): Board
+putPiece(x Integer, y Integer, p Piece)
+getPiece(x Integer, y Integer): Piece
+width(): Integer
+height(): Integer

-myWidth: Integer
-myHeight: Integer
- myMap: Piece

Piece

-myType:Integer
-myBitmap:Integer

+Piece():Piece
+Piece(t Integer, sString): Piece
+Piece(o Piece): Piece
+width():Integer
+height():Integer
+=(o Piece):Piece
+type():Integer
+bitmap(): String
== (p Piece):Integer
!=(p Piece):Integer

Player

-myId: Integer
-myBoard: Board
-myWindow: BoardWindow
-myMessage: Message
-myError: Error
-drawPiece(x Integer, y Integer)
-drawText()
-messageString(): String
-errorString(): String

+Player (id Integer, n String, t String, b Board): Player
+ redraw()
+setBoard(nb Board)
+getEvent(e Event)
+message(m Message)
+error(e Error)

Event

Message

Evaluation

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
- ❏

A Cleaner View

BoardGame

+Boardgame (x Integer,y Integer)
+play(x Integer,y Integer)
+valid(x Integer,y Integer):Integer
+gameOver():Integer
+turn(): PlayerId
+winner():PlayerId
+board(): Board

Gomoku

+Gomoku(x Integer, y Integer, p, Integer)

Board

+Board (w Integer, h Integer): Board
+Board (b Board)
+putPiece(x Integer, y Integer, p Piece)
+getPiece(x Integer, y Integer): Piece
+width(): Integer
+height(): Integer

BoardWindow

+ BoardWindow(String,Integer, Integer,Integer,Integer,String)
+putPiece(x Integer, y Integer, p Piece)
+putText(x Integer, y Integer, t String)
+clear(Integer,Integer,Integer,Integer)
+getEvent(e Event)
+width():Integer
+height():Integer

Piece

+Piece():Piece
+Piece(t Integer, sString): Piece
+Piece(o Piece): Piece
+width():Integer
+height():Integer
+type():Integer
+bitmap(): String

Player

+Player (id Integer, n String, t String, b Board): Player
+ redraw()
+setBoard(nb Board)
+getEvent(e Event)
+message(m Message)
+error(e Error)

6. Languages Issues and Extraction Tracks

...

Stereotype: a Way 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

«GUI» BoardWindow
+ BoardWindow(String,Integer, Integer,Integer,Integer,String) +putPiece(x Integer, y Integer, p Piece) +putText(x Integer, y Integer, t String) +clear(Integer,Integer,Integer,Integer) +getEvent(e Event) +width():Integer +height():Integer

Screen
Default
allScreens : Collection open: (s String) : Screen default : () Screen install : () Screen defaultWindowClass() : Class bounds () : Rectangle contentsFromUser () : Rectangle isOpen() : Boolean ....

«metaclass» Screen
Default
allScreens : Collection open: (s String) : Screen default : () Screen install : () Screen defaultWindowClass() : Class

❑ 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)

Languages Specific Issues (i)

- UML has no direct mapping to a specific language. Even if it is close to C++
- We should interpret it and extend it to find the rigth mapping

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»

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

checkWinner

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

width:Integer

Class Method Inheritance

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

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

Board
Board (s String):Board

CustomizedBoard

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
```

Define a stereotype if you need it
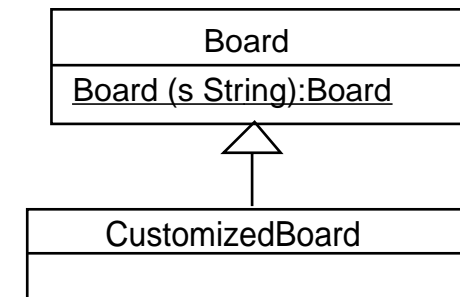
Language Specific Issues (ii)

❑ In Smalltalk return-type is boring because all methods returns self per default. So you may choose to only specify return if it it not the same as the class.

❑ Attributes are all private

❑ All methods are public but 'private categories'

Visibility Semantics Variations

What is the semantics of private, protected and public. For example, 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

Instance/Class Associations

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

UIBuilder (class) is related to the UILookPolicy (class)

UIBuilder

→

UILookPolicy

UIBuilder

«class»
────────→
{class}
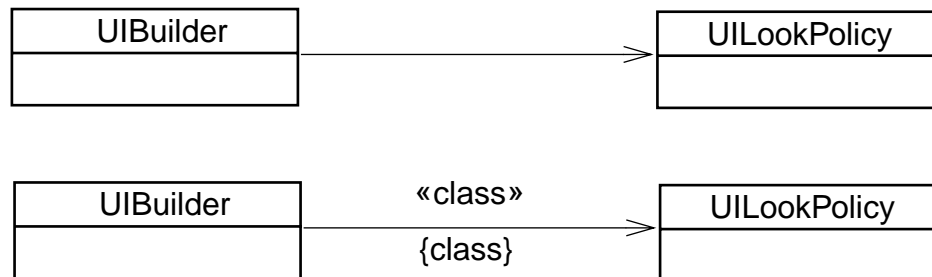
UILookPolicy

But an instance of UIBuilder is also related to an instance of UILookPolicy
Use a stereotype or a constraint

Class Information and Metaclasses

Class information are underlined (attribute and methods)

But class information: methods and attributes are not limited to instance creation methods (constructors static methods).

=> Mixing of levels and confusing!

uniqueInstance (c Class): Scheduler

defaultWindowClass (): Class

returns the class window

Workstation
send: (p Packet): Workstation accept: (p Packet): Workstation <u>withName:(n Name):Workstation</u> <u>withName:nextNode:(n Name, n Node) :Worstation</u>

Screen
<u>Default</u>
<u>allScreens : Collection</u> <u>open: (s String) : Screen</u> <u>default : () Screen</u> <u>install : () Screen</u> <u>defaultWindowClass() : Class</u> bounds () : Rectangle contentsFromUser () : Rectangle isOpen() : Boolean ....

Stereotypes for Class Behavior

Could work for well defined situation

Workstation
uniqueInstance: Workstation
send: (p Packet): Workstation   accept: (p Packet): Workstation   withName:(n Name):Workstation   withName:nextNode:(n Name, n Node) :Worstation   uniqueInstance: Workstation

«Singleton»
Workstation
send: (p Packet): Workstation   accept: (p Packet): Workstation   withName:(n Name):Workstation   withName:nextNode:(n Name, n Node) :Worstation

But still fragile

«Transcient Singleton»
Screen
Default
allScreens : Collection   open: (s String) : Screen   install : () Screen   defaultWindowClass() : Class   bounds () : Rectangle   contentsFromUser () : Rectangle   isOpen() : Boolean

Association Extractions (i)

Goal: Explicit references to domain classes

1: 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: anOrder -> an association

                myDisplay : Display -> not an association

2: Filtering based coding conventions or visibility

In Java, C++

        filter out private attributes

       – *

Association Extractions (ii)

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.

Two classes possess attributes on each other

       => an association with navigability at both side

Code Inferring or Design Extraction

If there are some coding conventions

In Java

```
class Order {

    public Customer customer(); (single value)

    public  Enumerator  orderLines();  (multi-val-
ues)

}
```

In Smalltalk

```
Order>>customer (single value)

Order>>orderLinesCollect: (multi-values)

Order>>orderLinesDo:

Order>>orderLinesSelect:

Order>>orderIsEmpty
```

Order
dateReceived isPrepaid number: String price: Money
dispatch() close()

* 1

Customer
name address
creditRating(): String

1

*

OrderLine
quantity: Integer price: Money isSatified: Boolean
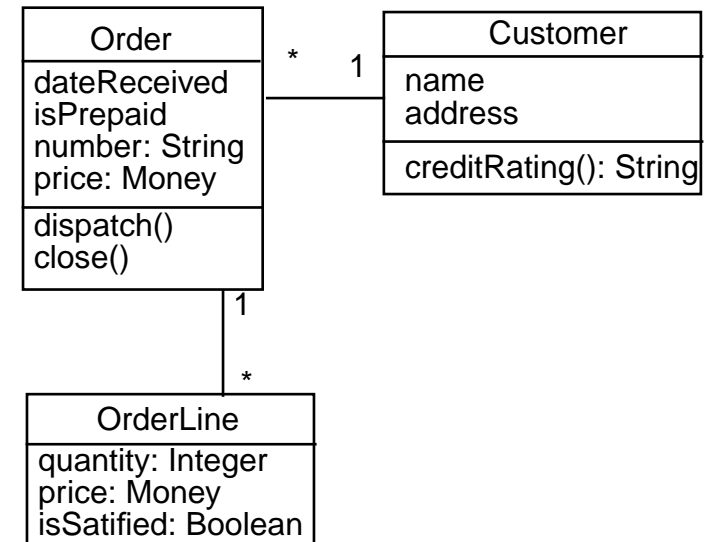
From the specification level note that responsibilities do not imply data structure.
Cannot be able to say if Order contains a pointer to Customer or something else

<u>*Associations: Implementation Perspective*</u>

Associations represent pointers in both directions between related classes

In Java

```
class Order{
    private Customer _customer;
    private Vector _orderLines; }
class Customer {
    private Vector _orders }
```

In Smalltalk

```
Object subclass: #Order
    instanceVariable: 'customer orderLines'
Order>>initialize
    orderLines := OrderedCollection new: 5


Object subclass: #Customer
    instanceVariable: 'orders'
Customer>>initialize
    orders := OrderedCollection new: 5
```

Operation Extraction

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

Operation Extraction (ii)

If there are several methods with more or less the same intent
- select the method with the smallest prefix

      you want to know that the fonctionality exists not all the details
- select the method with the more parameters

      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

Use company conventions to filter
- Access to database
- Calls for the UI
- Naming patterns

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

Case Study

...

UIBuilder of VisualWorks

- ❑ By hand just to prove the point
- ❑ VisualWorks: a middle size frameworks (700 classes)
- ❑ UIBuilder: central to the frameworks, responsible of the building of the interface
- ❑ Quit complex class
- ❑ No type but senders and implementors to identify client

To Help You to Follow

```
UIBuilder

        ('initialize-release' #initialize #newSubBuilder)

        ('accessing' #bindings #bindings: #cacheWhileEditing #cacheWhileEditing: #component #compo-
nent: #componentAt: #componentAt:put: #composite #composite: #converterClass #converterClass: #dec-
orator #decorator: #isEditing #isEditing: #keyboardProcessor #keyboardProcessor: #labels #labels:
#namedComponents #namedComponents: #noTabbing #policy #policy: #source #source: #spec #spec: #visuals
#visuals: #window #window: #windowSpec #windowSpec: #wrapper #wrapper:)

        ('adding' #add: #addCollection: #addCollectionRecyclingSpecs: #addSpec:)

        ('binding' #actionAt: #actionAt:put: #arbitraryComponentAt: #arbitraryComponentAt:put: #as-
pectAt: #aspectAt:put: #clientAt: #clientAt:put: #labelAt: #labelAt:put: #listAt: #menuAt: #menu-
At:put: #metaInfoAt: #metaInfoAt:put: #subCanvasAt:at: #subCanvasAt:at:put: #usingMenuBar: #visu-
alAt: #visualAt:put:)

        ('building composites' #endComposite #endCompositeLayout: #endCompositeLayout:properties:
#newComposite #newComposite: #popComposite)

        ('building windows' #windowLabeled: #windowOn: #windowOn:label:)

        ('scheduling' #doFinalHookup #open #openAroundCursorWithExtent:andType: #openAt: #openDialog
#openDialogWithExtent: #openDialogWithExtent:postOpen: #openIn: #openPopUpIn: #openPopUpIn:type:
#openPopUpIn:type:postOpen: #openWithExtent: #openWithExtent:andType:)

        ('private' #addComponent #applyLayout: #attemptCompositeKey: #cancelAction: #defaultAction:
#defaultProperties#findBindingFor:selector:cache:#fixWindowWith:#raiseErrorMessage:with:#safely-
Perform: #safelyPerform:key: #sendKeyboardTo: #setupKeyboard #setWindow: #stack #stack: #startNew-
Component #wrapWith:)
```

Class Method Extraction

categories: 'class initialization' ' examples' 'private' 'Signal constants'

=> 'policies' 'instance creation'

policies:

defaultPolicyClass defaultPolicyClass: (accessor for a classVariable)

+ only called by the instance level

=> defaultPolicyModel

instance creation: new, newWithPolicy: UILookPolicy

UILookPolicy

UIBuilder
new ():UIBuilder
newWithPolicy: (p UILookPolicy) : UIBuilder
defaultPolicyModel (): Class
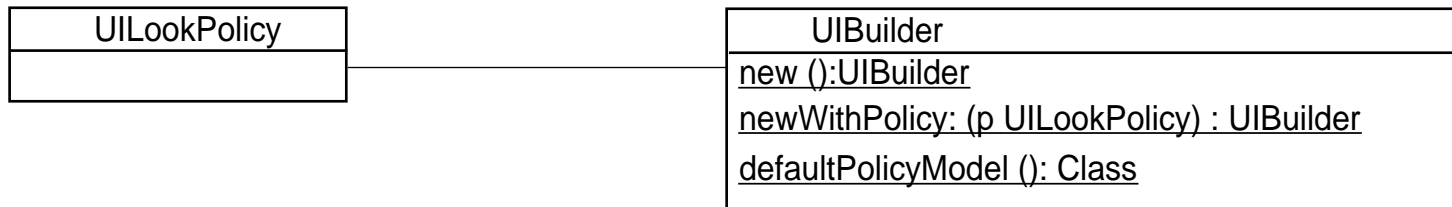
UIBuilder: methods at instance level (i)

categories filtered: 'initialize-release' 'private' 'accessing'

    => 'adding' 'binding' 'building composites' 'building windows' 'scheduling'

'adding' => add: , addSpec:, addCollection:, addCollectionRecyclingSpecs:

    (addSpec: only called by add: and windowOn: Label: of UIBuilder)

    We may apply "select only the simpler prefix here add"

'scheduling' =>   doFinalHookup

                  open (select only the simpler prefix)

'building windows' =>

    windowOn:Label: (select longer parameter)

'building composites' =>

    newComposite:,

    newComposite (called heavily by clients)

    popComposite (only called by UILookPolicy)

    endComposite

'binding' => clientAt:, clientAt:put:,labelAt:,labelAt:put:

    ....

    remove *at: and *at:put:

UIBuilder
add: (u UISPecification) addCollection: (c Collection) addCollectionRecyclingSpecs: (c Collection) open doFinalHookup windowOn:label: (w Window, l String): Window newComposite: newComposite popComposite endComposite

ScheduledWindow
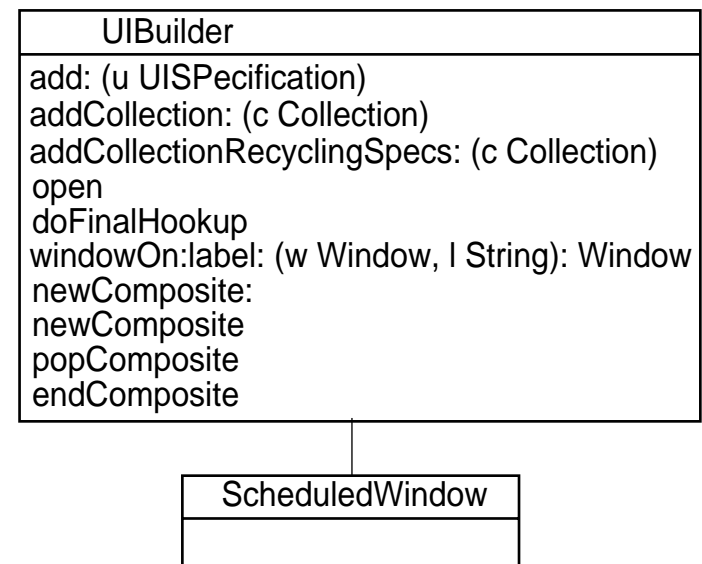
UIBuilder attributes
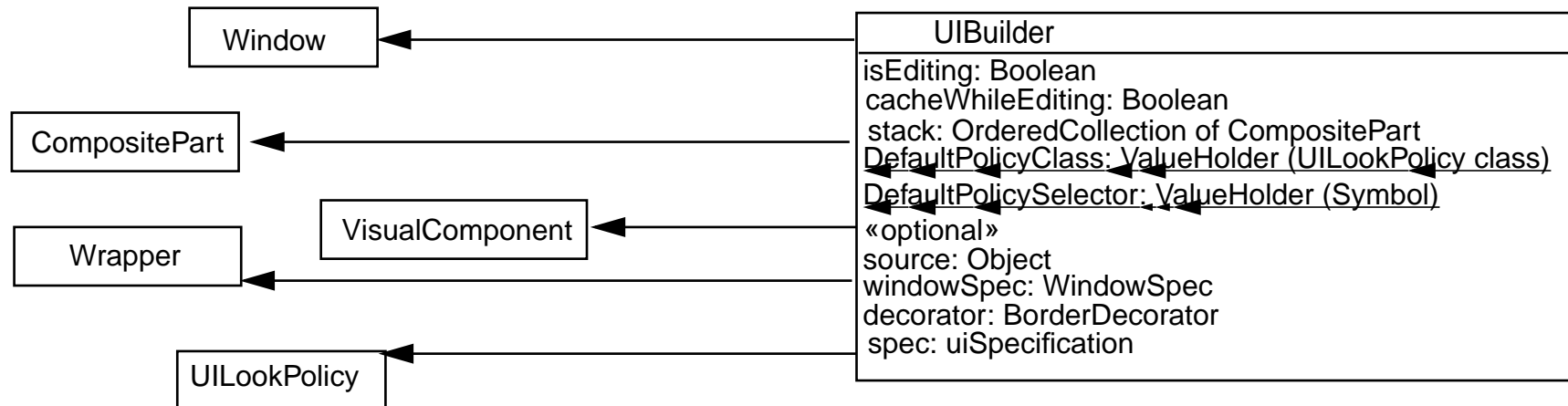
Remove: not domain entities: KeyboardProcessor

As association:

   windowSpec | nil, BorderDecorator | nil -> optional stereotype

   other domain entity: UILookPolicy, UISpecification, BorderDecorator

As attributes: spec are kind of arrays that encode a visual description

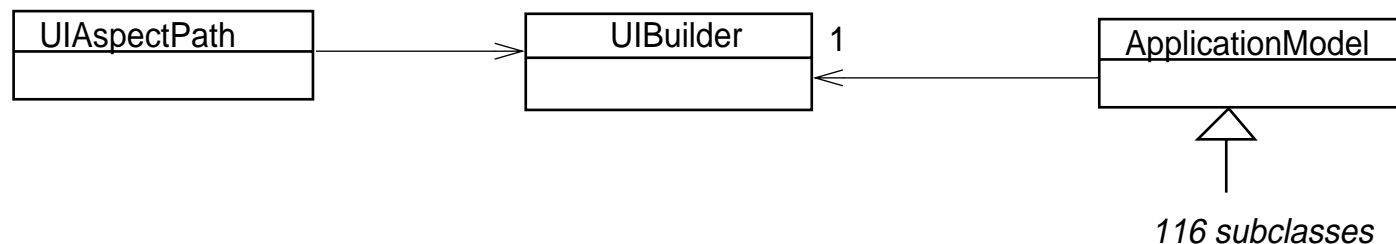Filter out the Dictionary based attributes: Too few information

Window	←	**UIBuilder**

```
+-----------+                              +--------------------------------------------------+
|  Window   |◄─────────────────────────────|                   UIBuilder                      |
+-----------+                              +--------------------------------------------------+
                                           | isEditing: Boolean                               |
                                           | cacheWhileEditing: Boolean                       |
+---------------+                          | stack: OrderedCollection of CompositePart        |
| CompositePart |◄─────────────────────────| DefaultPolicyClass: ValueHolder (UILookPolicy class) |
+---------------+                          | DefaultPolicySelector: ValueHolder (Symbol)      |
                      +-----------------+   | «optional»                                       |
+----------+          | VisualComponent |◄─| source: Object                                   |
| Wrapper  |◄─────────+-----------------+   | windowSpec: WindowSpec                            |
+----------+                              | decorator: BorderDecorator                       |
                                           | spec: uiSpecification                            |
        +--------------+                   |                                                  |
        | UILookPolicy |◄──────────────────|                                                  |
        +--------------+                   +--------------------------------------------------+
```

Classes Pointing to UIBuilder

Smalltalk is dynamically typed, so look for "builder" as attributes and method "builder"

With typed language or a type inference algorithm we would look at attributes with type UIBuilder or methods that return UIBuilder.
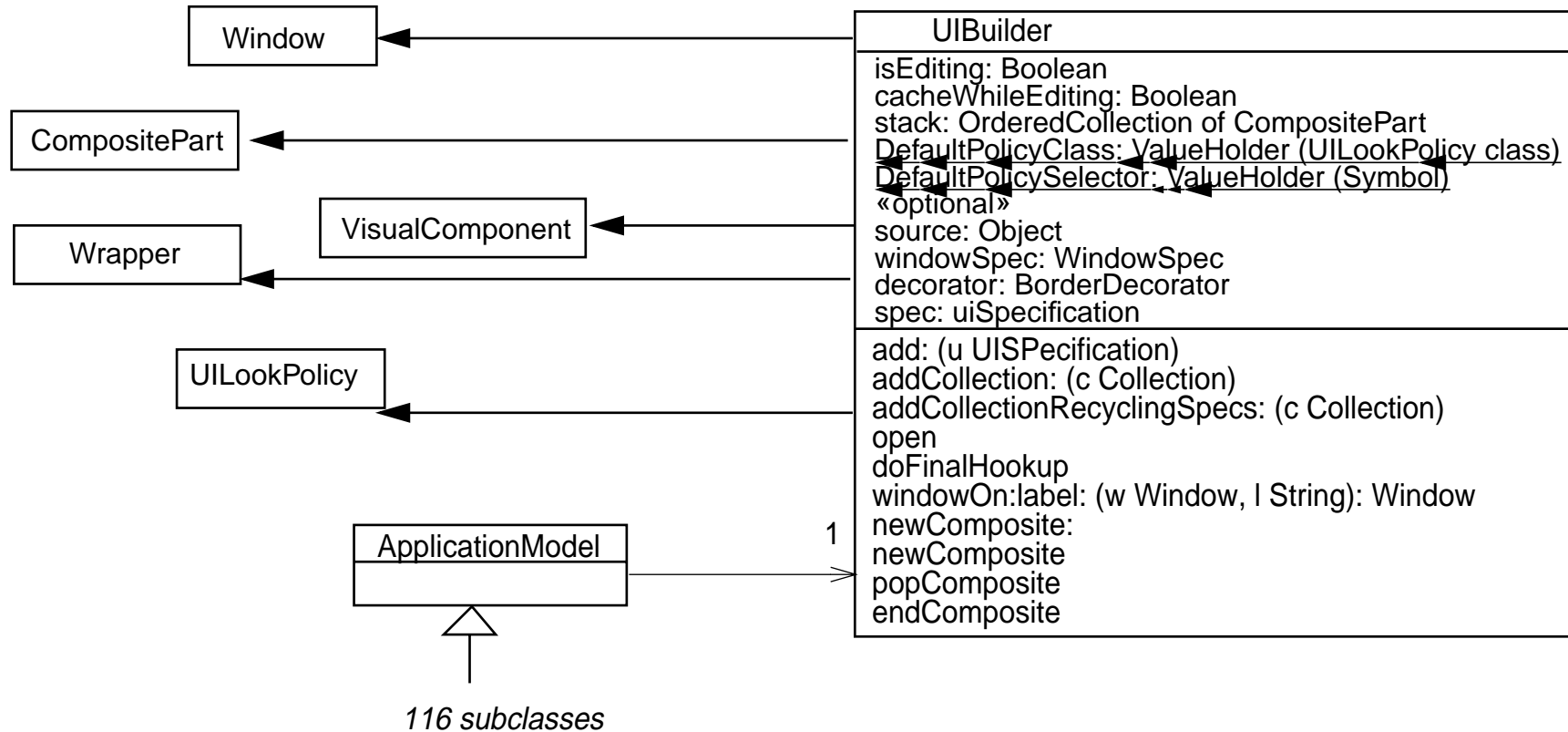
But we restrict the lookup to frameworks itself.

Inheritance impact the result:

-> two classes with builder as attributes (UIAspectPath, ApplicationModel)

-> two classes with accessors (ApplicationModel, UIPainterController but via a model)

```
┌─────────────────┐        ┌─────────────────┐        ┌─────────────────┐
│  UIAspectPath   │        │    UIBuilder    │  1     │ ApplicationModel │
├─────────────────┤───────>├─────────────────┤ <───── ├─────────────────┤
└─────────────────┘        └─────────────────┘        └─────────────────┘
                                                              △
                                                              │
                                                       116 subclasses
```

The Overall Picture

| Window |

| CompositePart |

| VisualComponent |

| Wrapper |

| UILookPolicy |

UIBuilder
isEditing: Boolean cacheWhileEditing: Boolean stack: OrderedCollection of CompositePart DefaultPolicyClass: ValueHolder (UILookPolicy class) DefaultPolicySelector: ValueHolder (Symbol) «optional» source: Object windowSpec: WindowSpec decorator: BorderDecorator spec: uiSpecification
add: (u UISPecification) addCollection: (c Collection) addCollectionRecyclingSpecs: (c Collection) open doFinalHookup windowOn:label: (w Window, l String): Window newComposite: newComposite popComposite endComposite

| ApplicationModel |

1

116 subclasses

=> Better knowledge of the domain class

A tool should help an expert of the domain, it will not deduce everything automatically

Qualified Associations

Associative arrays, maps, dictionaries

Order
Product

Catalogue
Part Number

```
0..1          lineItem
```

OrderLine

```
0..1
```

Part

In the context of an Order you need a product to identify anOrderLine.

Conceptual: you cannot have two OrderLines within an Order for the same product

Specification:

```
class Order {
    public OrderLine lineItem (Product aProduct);
    public void addLineItem(number Amount, Product forProduct);
```

=> access to a given LineItem requires aProduct

Implementation:

```
class Order {
    private Dictionary _lineItems; (key is a product)
```

Aggregation

Aggregation is part-of relationship

Ex: A car has an engine and wheels as its parts

Simple at first glance but complex in fact !!

Difficult because there are a lot of different notions and definitions. Even Gurus disagree!

Aggregation is just a special kind of association.

```
┌───────────┐                    ┌──────────────┐
│Company    │                    │ Department   │
│           │◇──────────────────│              │
└───────────┘                    └──────────────┘
      1                               *
```

The distinction is purely conceptual.

The open diamond distinguishes the whole from the part.

Does not change the navigation or the lifetimes of the whole/part

Aggregation and Composition

UML proposes a limited view of aggregation where lifetime and sharable issues are linked

In UML composition implies that

- (non-sharable) parts belong to only one whole

- (constrained lifetime) parts live and die with the whole

```
                          3..*  ┌─────────┐
                        ┌──────→│ Point   │
          ┌─────────┐◆──┘       │         │
          │ Polygon │           └─────────┘
          │         │◇──┐
          └─────────┘   │       ┌─────────┐
                        └──────→│ Style   │
                                │         │
                                └─────────┘
```
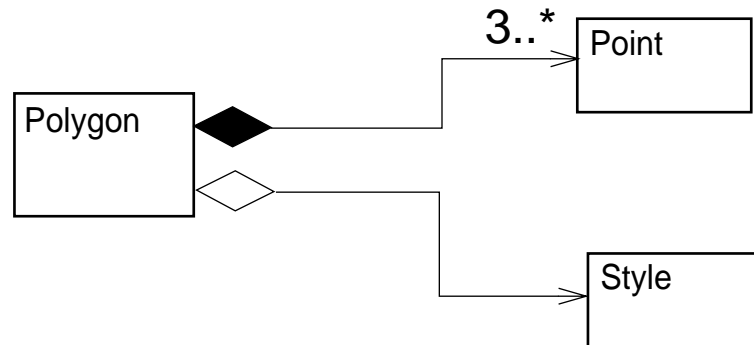
UML does not offer predefined way to express:

- different liteime sharing dependency

- reference, embedded value

USe stereotype

Aggregations or/and Multiplicity Extraction

Iterators or collections =>

      - aggregation

      - associations 1..* between class and the class of the collection elements.

Private Inheritance semantics = aggregation [Riel96]

About Aggregations: an Analysis View

From "A Taxonomy..." article (Linguistic, logic and cognitive psychology) different part-of relationship are identified:

Component integral, material-object, portion-object, place-area, collection-members, container-content, member-partnership

Three criteria:

- Configuration: whether or not parts have a functional and structural relationship either to one another or to the whole they constitute

- Homeomerous: whether or not parts are the same kind of thing as the whole

- Invariance: whether or not parts can be separated from the whole

	Configurational	*Homeomeric*	*Invariant*
Component integral	*yes*	*no*	*no*
Material object	*yes*	*no*	*yes*
Portion	*yes*	*yes*	*no*
Place-Area	*yes*	*yes*	*yes*
Member	*no*	*no*	*no*
Partnership	*no*	*no*	*yes*

<u>*Component-Integral Object Aggregation*</u>

Configurational: yes Homeomeric: no Invariant: no

"Defines a configuration of parts within a whole"

- Also called assembly-parts

- Component parts are objects in their own right even when they are not part of the whole

- Parts can be removed

- The configuration of the parts should be particular (structural or functional)

- The integral object exibits a patterned structure organization

- But a part cannot be removed without affecting the concept of the whole

Examples:

       Scenes are part of films

       Wheels are part of a car, a keyboard is part of a computer

       Bristles are part of toothbrush

       Nuclear physics is part of physics

       Projective geometry is part of mathematics

When a component ceases to support the overall pattern of an object a different kind of association result (part-of -> piece-of)

Material-Object Aggregation

Configurational: yes Homeomeric: no Invariant: yes

"Defines an invariant configuration of parts within a whole"

- Define what objects are made of
- Parts cannot be removed
- Parts lose their identity when they are used to make a whole

With material obejct relationships, the relationship between the parts is no longer known once they become part of the whole.

Example:

A cappucino is partly milk

A car is made of iron

Portion-Object Aggregation

Configurational: yes Homeomeric: yes Invariant: no

"Defines a homoemeric configuration of parts within a whole"

- The parts are the "same" kind as the whole
- The parts can be removed
- Certain properties of the whole can apply to the parts

Examples:

- A slice of bread is a portion of a loaf of bread
- A meter is part of a kilometer
- An hour is part of a day

Place-Area Aggregation

Configurational: yes Homeomeric: yes Invariant: yes

"Defines a homeomeric and invariant configuration of parts within a whole"

- Non sharable configuration of the parts within a whole.
- Parts cannot be removed

Example:

        San-Franscico is part of california

        A peak is part of a mountain

Odell said that this is UML composition but it is not!!

Member-Bunch Aggregation

Configurational: no Homeomeric: no Invariant: no

"Defines a collection of parts as a whole"

- Also called Collection-Member

- Parts can be removed

- No organization pattern but may have an implicit ordering

- Based on spatial, temporal or social connectivity

- Different from classification

Examples:

      A tree is part of a forest

      An employee is part of an union

      A day is part of a montly planner

Member Parternship Aggregation

Configurational: no Homeomeric: no Invariant: yes

"Defines an invariant collection of parts as a whole"

- Member cannot be removed without destroying the partnership

Examples:
- Laurel is part of Laurel&Hardy

Non-Aggregation Forms

- Spatial inclusion: relation between a container , an area or a temporal duration and that which is contained

- Classification inclusion: Christine is a car

- Attribution: properties of an object (height, color)

- Attachement:
  - Toes are attached to feet (part-of)
  - Earings are attached to Ear but are not part-of Ear

in UML & OML

May consider to group:

        - configurational aggregations as aggregation

        - non-configurational aggregations as membership

In UML no configurability issues

        - sharable

        - lifetime

<u>*Documenting for Evolution: Reuse Contract*</u>

How to identify the impact of changes?

change
propagation

domain

application

Example

Set
add(Element)
addAll(Set)

Set (Optimized)
add(Element)
addAll(Set)

CountingSet
count
add(Element)
addAll(Set)

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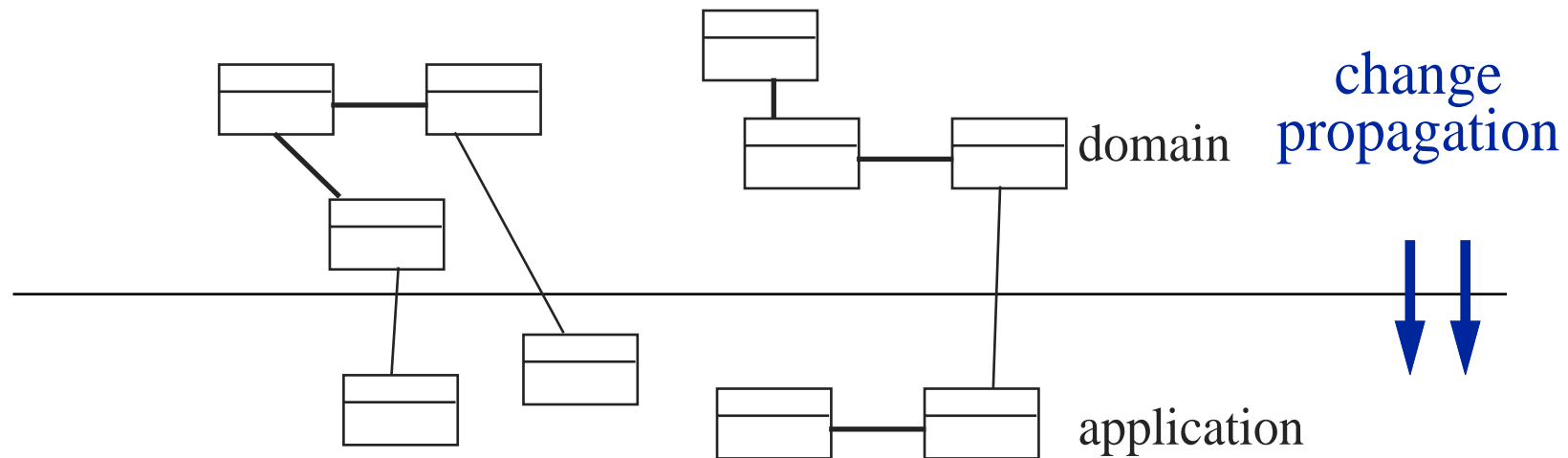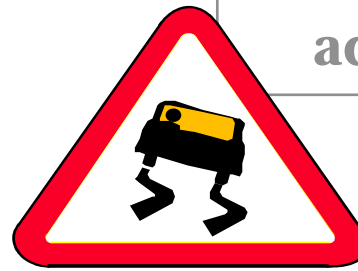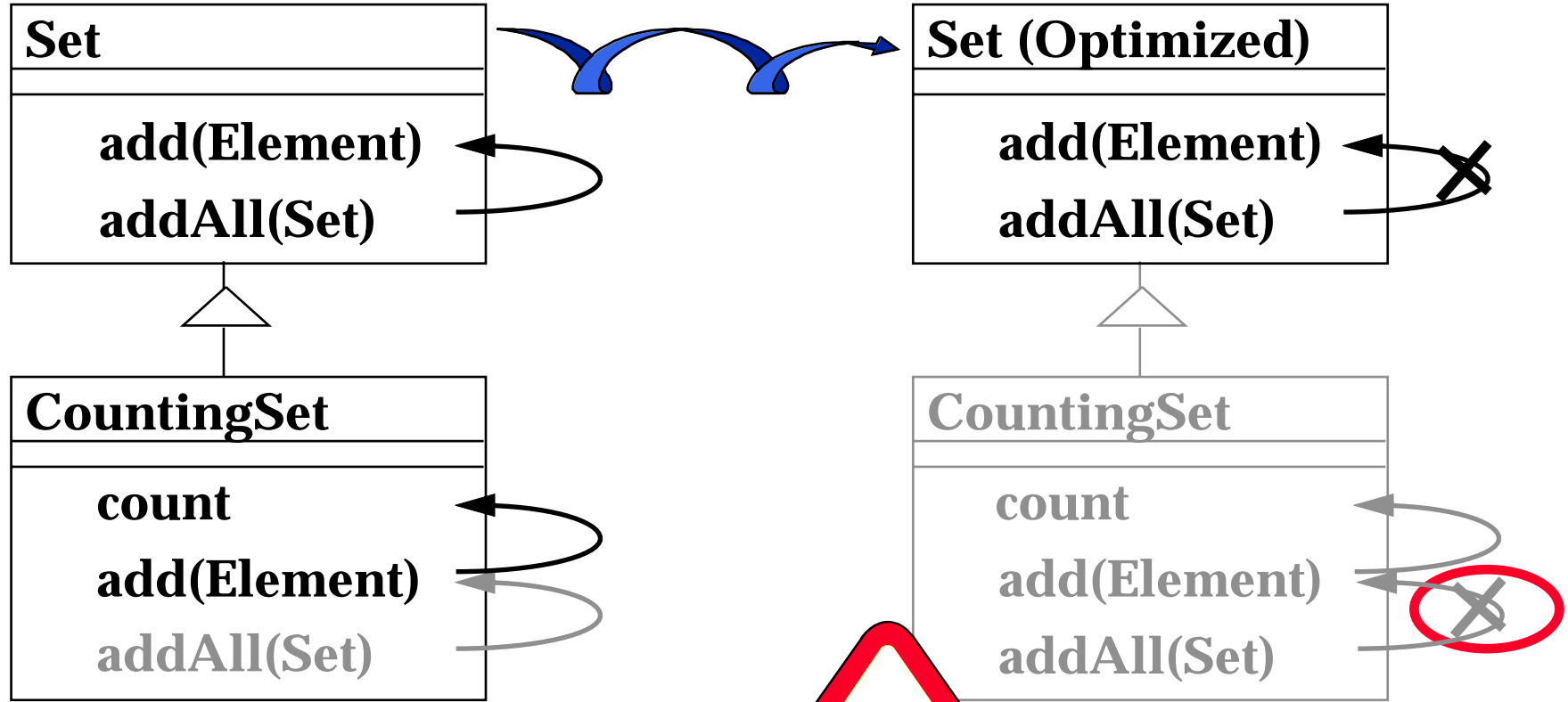
**not all elements
are counted**

Reuse Contracts: General Idea

domain

application

provider
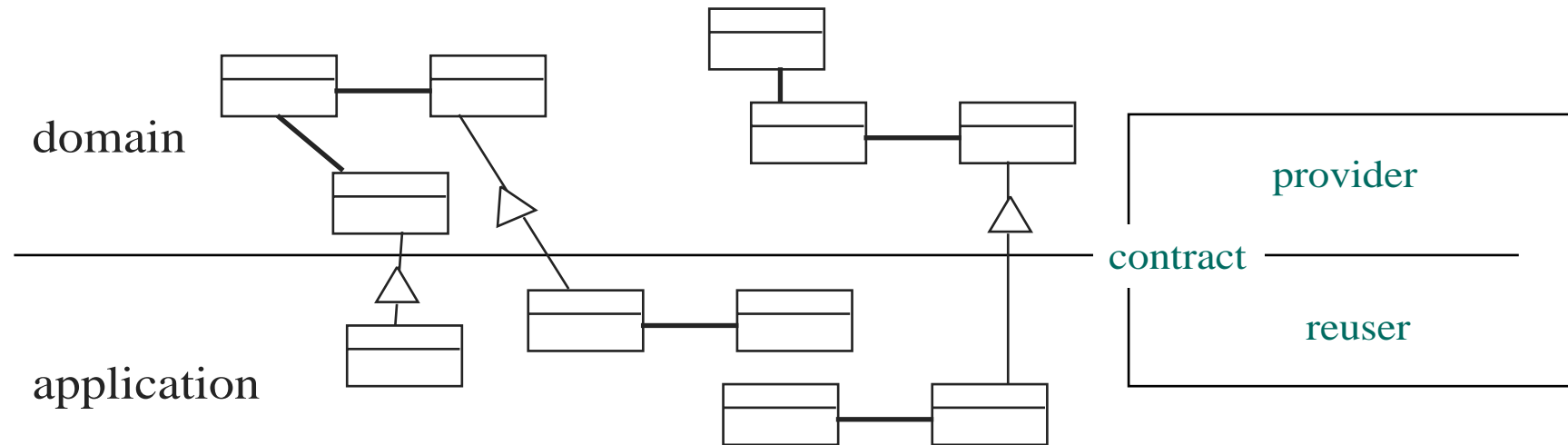
contract

reuser

Propose a methodology to:

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

Example

Coarsening

✓ addAll [-add]

Set
add   addAll **[add]**

Set
add   addAll **[add]**

Refinement

✓ add [+count]

CountingSet
count   add **[count]**

CountingSet
count   add **[count]**

--- effort estimate
   # addAll needs to be
   overridden too
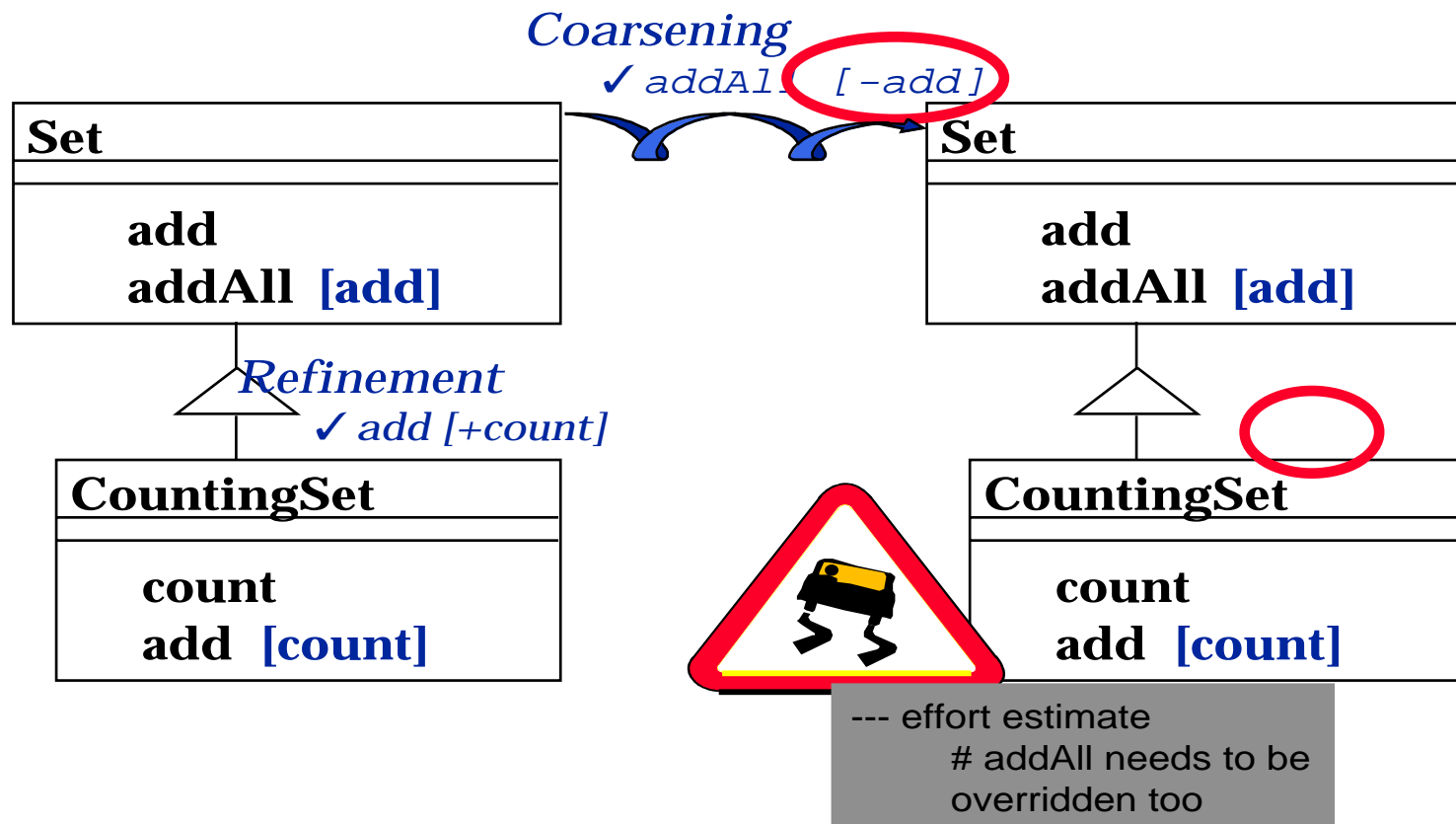
You can specify which other methods a method invokes (reuse contracts)

In class Set

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

Dynamic Behavior

You want to document a dynamic aspect of a system that static documentation is not able to show.

- Interaction Diagrams = Sequence Diagram or Collaboration Diagram

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!

Caller	Phone Line	Callee
caller lifts receiver →		
← dial tone begins		
dial (1) →		
← dial tone ends		
dial (2) →		
dial (2) →		
← ringing tone	phone rings →	
	← answer phone	
← tone stops	← ringing stops	
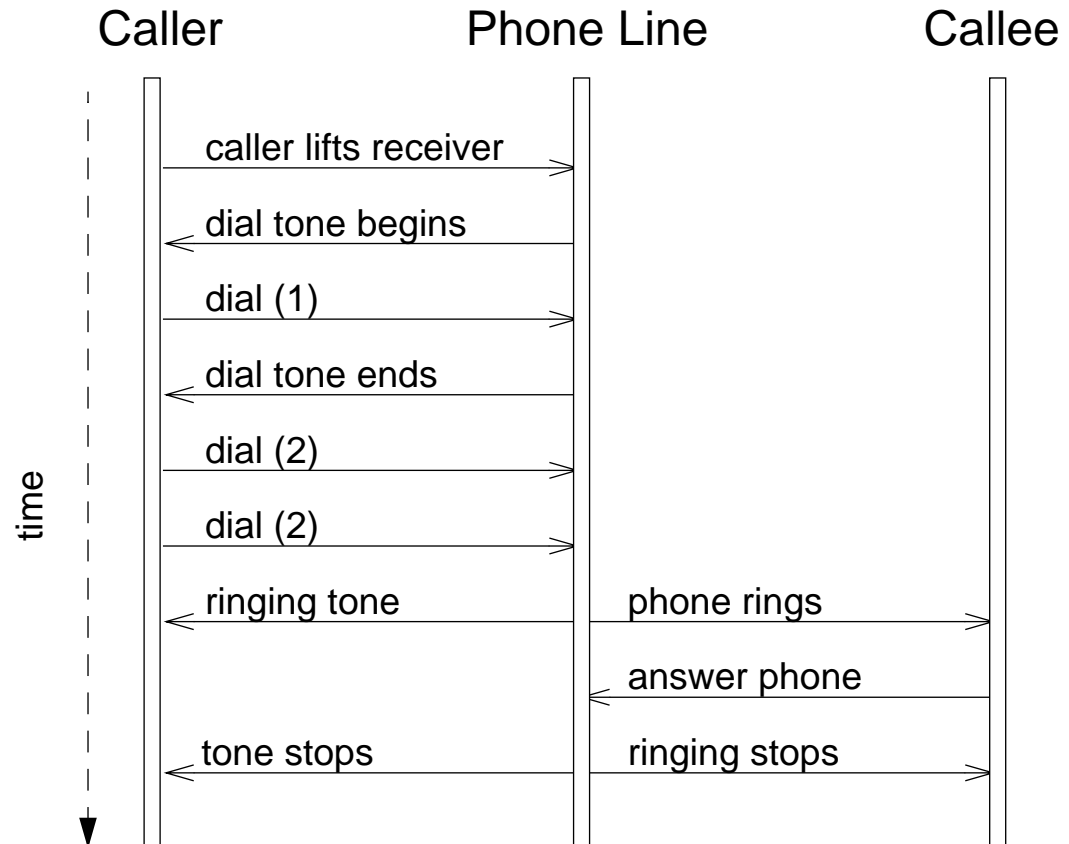
time

Collaboration Diagrams

Collaboration diagrams depict scenarios as flows of messages between objects:
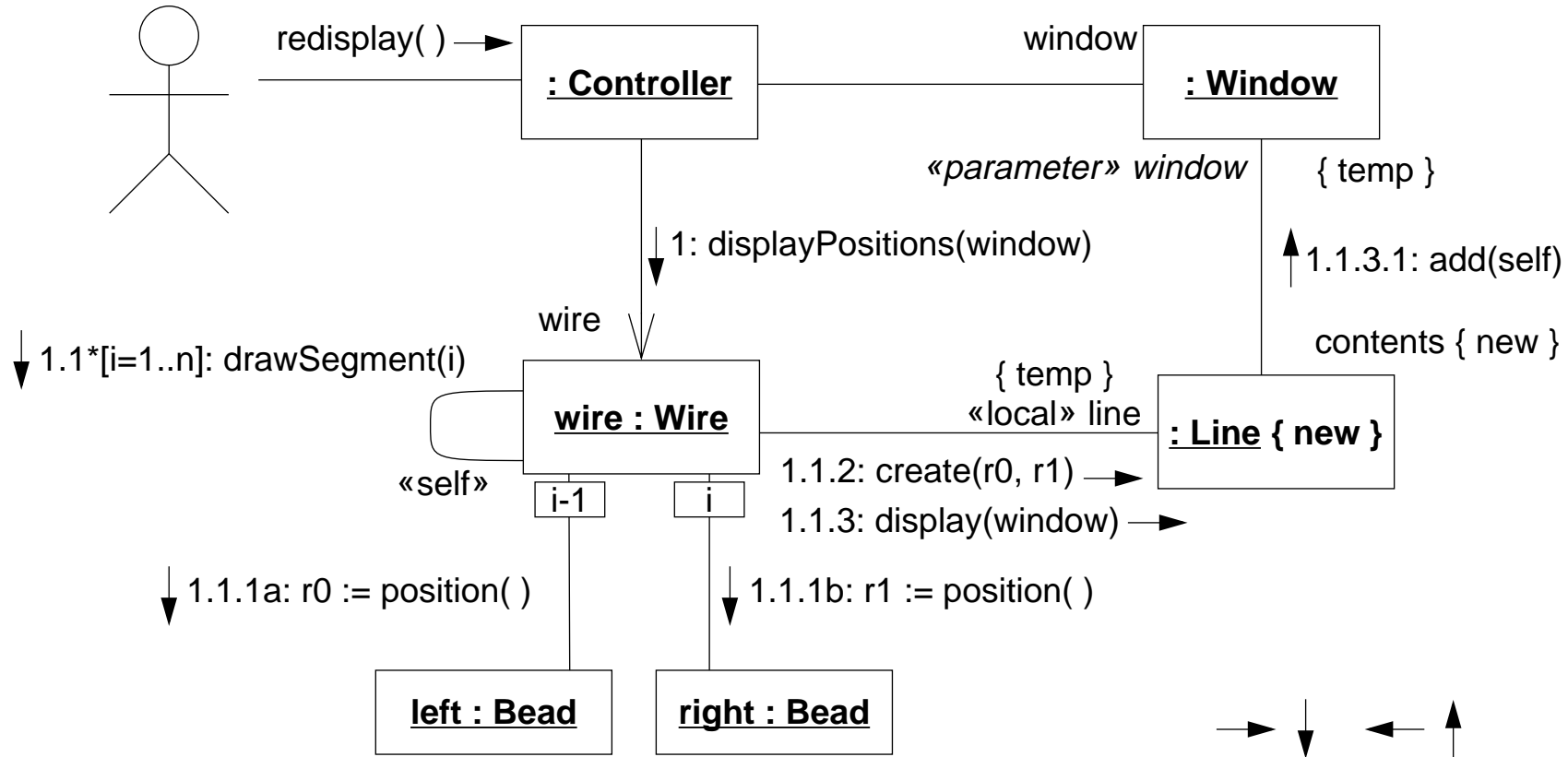
Implications Under Considerations

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

UML as Support for Design Extraction

❑ Influenced by C++ private protected public
❑ Generalization and not inheritance
❑ Composition aggregation limited (see OML references)
❑ Do not support well reflexive models
❑ Fuzzy: Packages
❑ But UML is extensible, define your own stereotype
❑ You should be clear about:

home conventions like navigability

level of interpretations

<u>References</u>

- A Foundation for Composition, Joop, 7(6), Oct 94
- A More Complete Model of Relationships and Their Implementation Mapping, Joop, vol. 10, n 6, octobre 1997
- Clarifying Specialized Forms of Association in UML & OML, Henderson-Sellers and Joop,vol. 11, n 2, may 1998
- Distilled UML, Martin Fowler, 1997, Addison Wesley, ISBN: 0-201-32563-2
- A Taxonomy of Part-Whole Relations, Winston, Chaffin and Herrmann, Cognitive Science 11, 1987, p417-444
- Six Different Kinds of Aggregation, Odell, Joop, january 1994
- 37 Things that Don't Work in Object-Oriented Modelling with UML, A. Simmons, I. Graham, 1998
- The Unified Modeling Language User Guide, G. Booch, J. Rumbaugh, I. Jacobson, Addison-Wesley,1998

Other References

- UML and C++: a Practical Guide to Object-Oriented Development, Lee and Tepfemhart, Prentice-Hall, 1997, ISBN: 0-13-619719-1
- Designing Object Systems: Object-Oriented Modeling with Syntropy, S. Cook and J. Daniels, Prentice Hall, 1997
- Object-Oriented Design Heuristics, A. Riel, Addison-Wesley,1996
- Basic Relationship Patterns, J. Noble, Plopd4 submission 1998

7. Code Duplication

Code Duplication

a.k.a. Software Cloning, Copy&Paste Programming, Code Scavenging

Matthias Rieger

FAMOOS Project, Software Composition Group

University of Berne

rieger@iam.unibe.ch

What is Code Duplication?

Duplicated Code = Source code segments that are found indifferent places of a system.
- ❑ in different files
- ❑ in the same file but in different functions
- ❑ in the same function

The segments must contain some logic or structure that can be abstracted, i.e.

```
...
computeIt(a,b,c,d);
...
```

```
...
computeIt(w,x,y,z);
...
```

is not considered duplicated code.

```
...
getResult(hash(tail(a)));
...
```

```
...
getResult(hash(tail(z)));
...
```

could be abstracted to a new function

Copied artefacts range from expressions, to functions, data structures to entire subsystems.

Copying: *I do it, you do it, everybody does it!*

Reasons for Copy & Paste Programming

❑ **Time Pressure**

There is no time to design and implement and test a component. And the problem has been solved before.

❑ **Lazyness**

Making something reusable always takes some extra effort over making the code work for one particular case only.

Rethinking an implementation if you suddenly see similarities between different parts of the implementation, requires energy.

❑ **Efficiency Considerations**

In time critical applications, a procedure call may seem to cost too much.

❑ **Maintaining Multiple Versions**

Keeping the code for multiple versions in separate files may be preferable to working with a lot of `#ifdef`'s.

❑ **Programmer Productivity**

If productivity is evaluated by looking at the number of lines of code written, there is incentive to produce new code rather than working on old code.

How Code Duplication happens

A possible scenario.

Context: Software Maintenance, enhancements of a legacy system.

12. A sub-component similar to an existing sub-component is defined.
13. New functionality requires substantial change on the exisiting component. Current uses of the existing component will be affected.
14. The analysis of the exisiting component will be lengthy and difficult. Significant regression testing will be required to ensure functioning in the old context.

Now add a little time pressure. This is when it happens!

15. A copy of the exisiting component is made. Systematic renaming is done to avoid naming conflicts. The component is tailored to its new use.
16. Code that is not understood and therefore cannot be deleted remains as *red herring* or even *dead code*.

Problems entailed by Code Duplication

General negative effect:

- ❏ Code bloat

Negative effects on Software Maintenance:

- ❏ Defects are found in a segment that has possibly been copied
  - ☞ find the clones of the code segment
  - ☞ assess the impact of the correction in each new context
- ❏ Errors in the systematic renaming
  - ☞ unintended aliasing
  - ☞ latent bugs that show up much later
- ❏ Red herrings and dead code
  - ☞ add to the cognitive load of future maintainers

Metaphorically speaking:

software aging, "hardening of the arteries", "software entropy" increases
  - ☞ small design changes become very difficult to effect

Justified Duplication

Advantages of code cloning:

❑ helps to meet short term goals

☞ the code is already designed, implemented *and* debugged!

❑ necessary if programming language lacks an abstraction mechanism (e.g. no mechanism for generic programming in C).

Duplication that is less problematic:

❑ C++ templates ("glorified macros")

❑ Inlining

☞ duplication is hidden from the programmer and handled by the compiler

Code Duplication: Problem Statement

Code Duplication: a fact of life as a programmer/maintainer

Deal with it!

Locate the 'enemy'

☞Code Duplication Detection

Other Reasons to Detect Duplicated Code

Program understanding: looking at a system through its structure of repetitive repetitive source code patterns, or programming idioms

Extraction of a library from a system: library functionality should be removed completely from the application

Understanding change: comparing different versions of the same program unveils the hot spots of change

Repairing Design flaws: In OOP, duplicated code may indicate design problems like missing use of inheritance

System partitioning: code cloning as a clustering criterion

Code Duplication Detection

What can be called a clone of a code segment?

Syntactic Equivalence

with variability

Semantic Equivalence

string matching algorithms
statistical methods

language
semantics
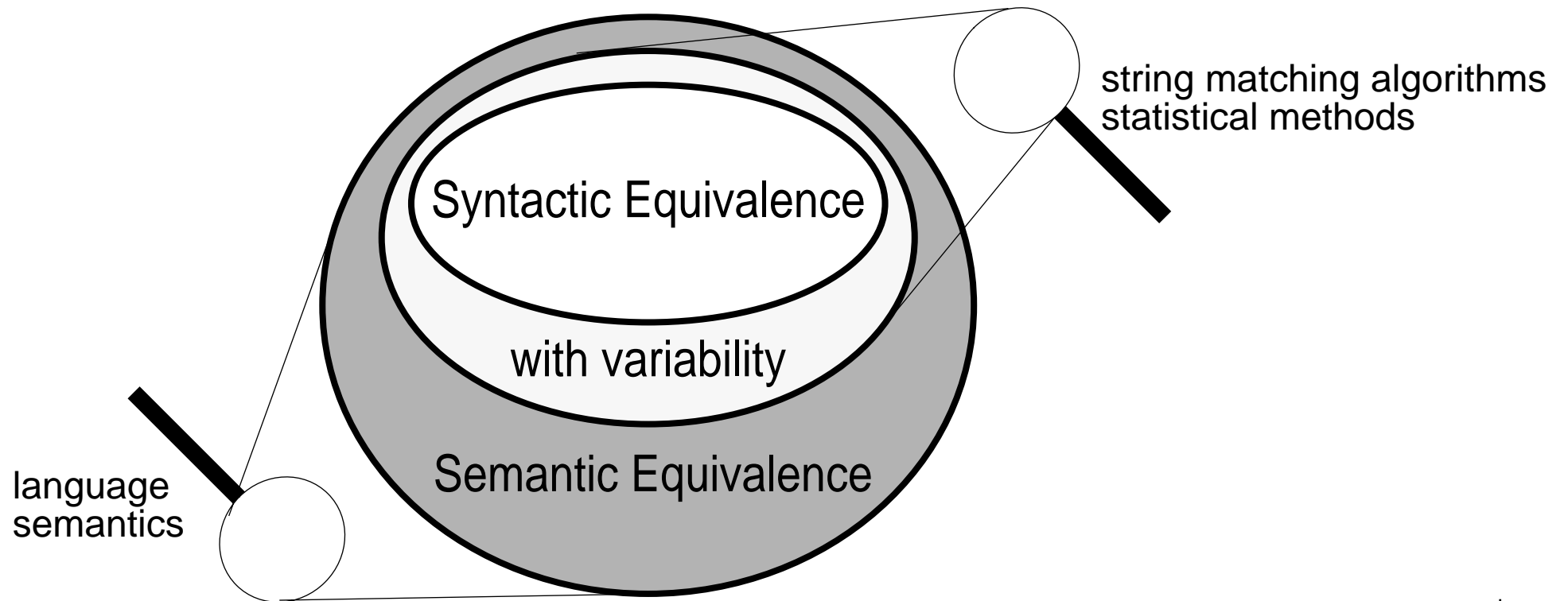
Definition of a Clone

Refining the notion of what we are looking for.

For two code fragments C_1, and C_2, C_2 is a clone of C_1 if

1. C_1 and C_2 are syntactically identical (e.g. are found identical using the Unix utility `diff`).

2. C_1 and C_2 have the same structure but modified variable names or data types.

3. C_1 and C_2 have the same structure but modified statements or expressions.

4. one differs from the other on inserted, deleted or substituted statements and expressions (the bigger the set of insertions and deletions is, the less similar the two fragments are to each other).

The selection of the category of clones determines the algorithmic effort.

Exact Matches

Only simple algorithms needed to detect exact matches

❑ Multiple possibilitites for preprocessing:

– remove white space, except for line separators (layout preserving)

– remove white space including line separators

– remove comments

– replace identifiers with an identifier marker (requires lexical analysis)

❑ Generate the substrings that will be tested against each other

– number and size of substrings determines size of possible matches and size of search space

❑ Match each substring against each other

☞ hashing used to reduce the search space quickly in a first pass

☞ sorting reduces $O(n^2)$ to $O(n*\log n + n)$

☞ recursing over suffix trees reduces to $O(n+m)$

Parameterized Matches I

Assumption: Programmers copy code segments and systematically replace variable names to fit in the new context.

- ❑ Lexical analysis of the source Code.
  - ☞ token names that are a candidate for a parameter are changed into a `P. x = 3*y; => P = P * P; `.
- ❑ Search exact matches among the transformed lines
  - ☞ Substring size = 1 line (preservation of layout)

```
...
P = P - P;
if (P>P) P = 1;
P = f(P);
P = P;
...
```

```
...
P = P - P;
if (P>P) P = 1;
P = f(P);
P = P;
...
```

Parameterized Matches II

❑ Exact matches are examined further to find parameterized matches.
☞ one-to-one correspondence between the parameters of the two code segments

```
x = y - z;

if (y>z) m = 1;

h = f(x);

y = x;
```
}}
y=b
z=c
m=n

y=c
{{
```
x = b - c;

if (b>c) n = 1;

h = f(x);

c = x;
```
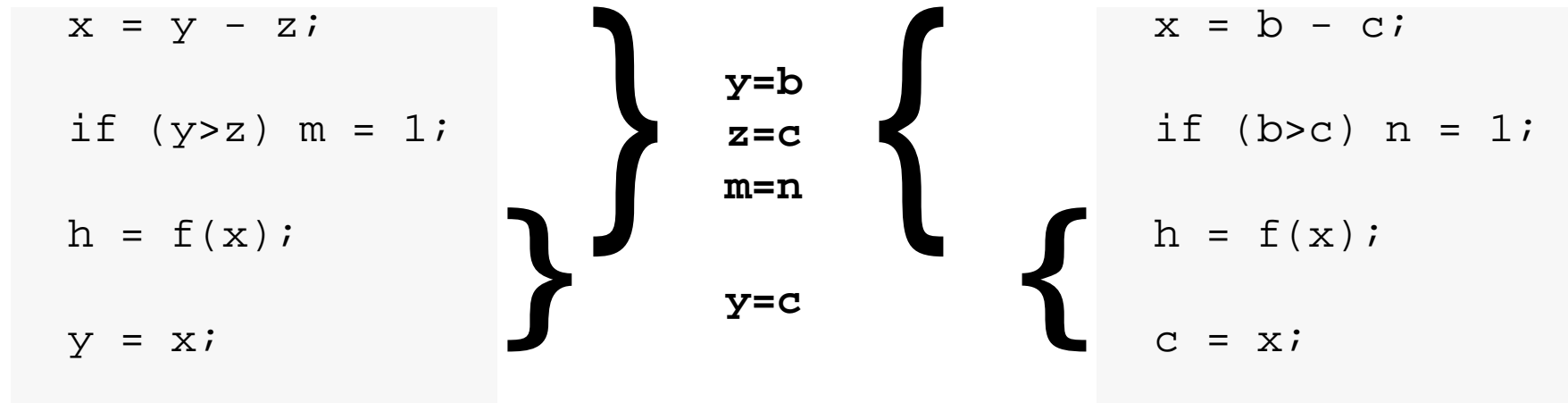
Syntax Based Detection: Metrics

Creation of an Abstract Syntax Tree (AST)

Important Nodes of the AST (allows for matching at different levels of granularity):

- epressions
- statements
- blocks
- functions

Calculation of a number of data and control flow related metrics -> metrics signature for each of these nodes

Compute the Euclidean distance between the signatures.

Benefits: –fast

- allows for matching in the presence of arbitrary local changes in the code

Drawback: low precision

Visualization of Duplicated Code I

Technique from DNA Analysis: Scatterplots

Certain dot configurations can be interpreted right away:
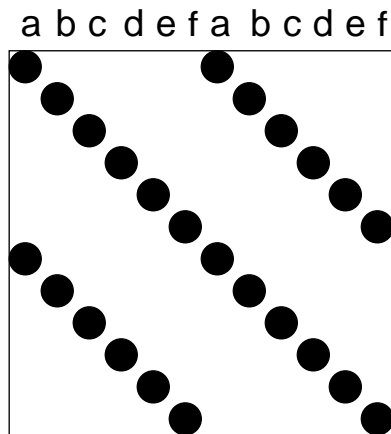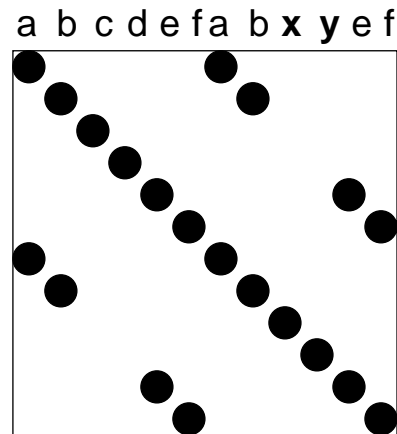
| Diagonals | D. with Holes | Broken Diagonals | Rectangles |
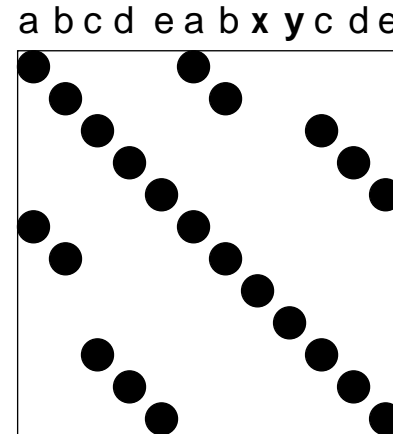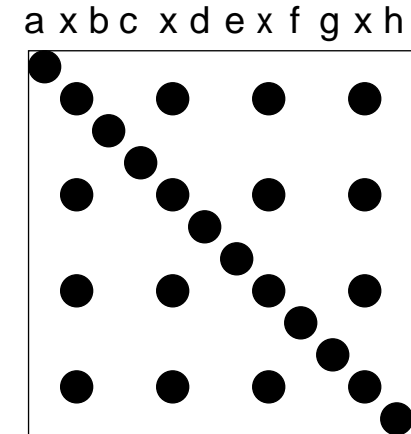
☞ intuitive insights into the duplication situation

☞ anomalies may be detected (be open for surprises)

☞ quick source code access

Visualization of Duplicated Code II

Example: Software Evolution, immediately graspable through the eye
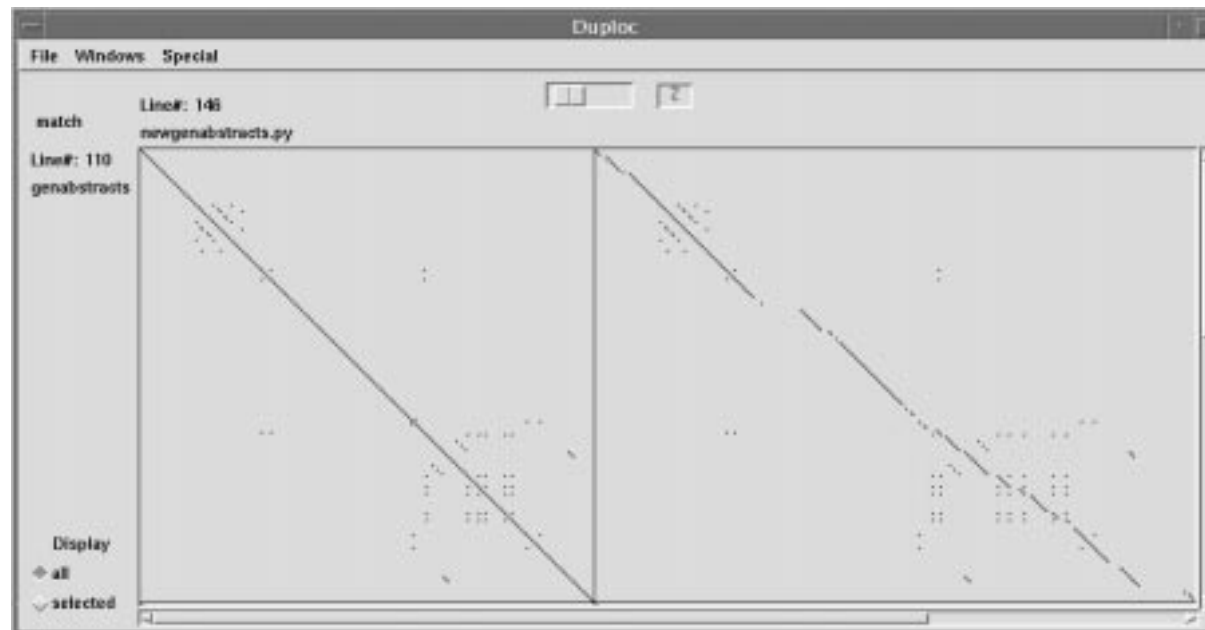
On the left, the file compared with itself. On the right, the file compared to a newer version of itself.

Tradeoffs

Language Dependency:

- – String Matching

- – Lexical Analysis

- – Parsing

Pattern space:

❏ we know what we are looking for

☞ pattern matching approaches (grep) allow fine-tuning

❏ we do not know what weare looking for

☞ the patterns to look for are derived from the search space

Refactoring Duplicated Code I

What can we do with found instances of duplication?

Redesigning is too much effort (and introduces too much new bugs).

Refactoring is Redesigning in the small.

When you refactor, you do not change the functionality of your program; rather you change its internal structure in order to make it easier to understand and work with.

Refactoring steps are usually very lightweight:
- ❏ renaming a method
- ❏ moving a field from on e class to another
- ❏ consolidating two similar methods into a superclass

> *Write every piece of logic once and only once (Kent Beck)*

Refactoring Duplicated Code II

❑ If you have a piece of duplication in methods of the same class
☞ refactor code into a function or method

❑ If you have pieces of duplication in two sibling subclasses.
☞ refactor code into a method and put it into the common superclass

❑ If the code in the subclasses is not the same but just similar, (e.g. the same algorithmic structure with some differences in the details)
☞ consider applying the *Template Method Pattern* (Gamma et. al., 1995)

❑ If you have found duplicated code in unrelated classes
☞ consider making the code into a component (function object) or think again, in which class the method really belongs.

❑ If you have found repetitive code
☞ try to rewrite it using arrays and loops

<u>*References*</u>

Brenda S. Baker. On Finding Duplication and Near-Duplication in Large-Software-Systems. In *Proceedings Second Working Conference on Reverse Engineering*, pages 86-95, IEEE Computer Society, 1995.

Jonathan Helfman. Dotplot Patterns: A Literal Look at Pattern Languages. TAPOS, 2(1):31-41,1995.

J Howard Johnson. Substring Matching For Clone Detection and Change Tracking. In Proceedings of the International Conference on Software Maintenance (ICSM), pages 120-126, 1994.

Kostas Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics, In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 44-54, IEEE Computer Society, 1997.

8. Refactorings

"Interface design and functional factoring constitute the

key intellectual content of software and are far more difficult to create and re-create than code"

> Peter Deutch

Beware of 'pet alligator' phenomenon !!!

> "It's cute while it's small, but it keeps growing"

> Anonymous

Outline

❏   Definitions

❏   Why?

❏   Low-Level Refactorings

❏   High-Level Refactorings

❏   Strategies for Refactoring

❏   Obstacles

❏   Bibliography

Some Definitions

- ❑ The process of redesigning the abstractions in a program
- ❑ A behavior-preserving source-to-source program transformation [Roberts&Brant98]
- ❑ The process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [Fowler99]
- ❑ A change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable state of that software component [Fowler99]
- ❑ Refactoring = reorganization plan for a system to improve reuse

Why? The Lie and the Reality

"Build one to throw away" Fred Brooks

The lie:

- ❑ *First,* you gather requirements
- ❑ *Then*, you analyse them
- ❑ *Then*, you design the system
- ❑ *Then*, you code it
- ❑ *Then*, you test it
- ❑ *Then*, you are done

The reality:

- ❑ You will never get it right the first time
  - Can't fully understand the problem domain
  - Can't understand user requirements (when the user itself really knows)
  - Can't understand how the system will evolve
- ❑ Result
  - Original design is inadequate
  - System becomes convoluted and brittle
  - Changes become more and more difficult

Why? The Obvious

Programs

❑   that are hard to read are hard to understand and to modify

❑   that have duplicated logic are hard to modify

❑   where additional behavior requires you to change running code are hard to modify

❑   with complex conditional logic are hard to modify

Good programmers refactor because:

Even if a good design comes first and the coding comes second.

☞   Over the time the code will get modified and the structure of the system according to the design gradually fades

Fit in Evolutionary Software Development

" Grow, don't build software" Fred Brooks

❏ Prototype

   - solidifies user requirements

   - sketch of system design

❏ Expansion

   - add functionality

   - determine hot-spots

❏ Consolidation

   - correct design defects

   - introduce new abstractions

Cultural Issues

❑ Clean *first*

❑ *Then* add new functionality

❑ 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

☞ Refactoring help to concentrate the optimizations

Obstacles to Refactorings

❑ Complexity

- Understanding the design is hard

- Changing the design of an existing system is hard

- Errors can be introduced

❑ Schedules and Culture

- Every software product is under time pressure

    (refactoring pahse should occur after delivery)

- Get paid to add new features

- If it doesn't break, doesn't fix it

- Refactoring can take time

- Producing negative lines of code is out of our culture

❑ Databases Mapping

❑ Changing Interfaces

What Refactorings Are Good For?

❑ Help you to develop software more quickly
❑ Improve the structure of existing code
❑ Enable sharing of logic
❑ Isolate change
❑ Reduce the code size
❑ Eliminate duplicated code
❑ Make your code more readable (Reveal intention)
❑ Help you to understand unfamiliar code
❑ Enhance code to support new requirementsRefactorings

Refactorings are

❑ Behavior-preserving (observable behavior does not changed)
❑ Some are semantics-preserving i.e. we can garantee that there is no change
   Example: renaming an instance variable
❑ Have precondition to ensure their semantics preserving behavior

Renaming Methods

Renaming Methods Help You to Understand Code

Example

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

Give to the reader a good idea of the functionality and not about the implementation

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

The code that uses the method will gain in readibility!!

Examples of Refactoring Scripts

❑ Creating a new empty class

pred: a class with the same name does already exist

refact: create a class

❑ Renaming all methods with a given name:

pred: no method locally exists with the same name

5. Check if a method does not exist in the class and superclass/subclasses with the same "name"
6. Browse all the implementers
7. Browse all the senders
8. Edit and rename all implementers
9. Edit and rename all senders
10. Remove all implementers
11. Test

Some Refactorings

Create Empty Class

Add Instance Variable

Add Method

Delete Class

Delete Instance Variable

Delete Methods

Rename Class

Rename Instance Variable

Rename Method

Convert Variable Reference to Message

Extract Code as Method

Inline Method

Change Superclass

Move Instance Variable into Component

Move Method into Component

Add Method Argument

Delete Method Argument

Reorder Method Argument

Pull Up Instance Variable

Pull Up Method

Push Down Instance Variable

Push Down Method

Low-Level (i): Adding/Deleting

❑ Creating a program entity

- create empty class: define a new class with no instance variables, nor methods

- create instance variable: add an unreferenced instance variable to a class

- create method: add a method to a class that either is unreferenced or is identical to an already inherited method

❑ Deleting a program entity

- delete unreferenced class

- delete unreferenced instance variable

- delete methods: delete a set of methods; either each method is redundant, or the only references to it are by other methods that are also being deleted

Low-Level (ii): Changing a Program Entity

❑ change class name (the precondition ensures distinct class names)

❑ change variable name

> variable name should not already exist in superclasses/subclasses

❑ change method name: change the name of a method and any corresponding methods defined in subclasses, and any corresponding message sends

❑ change type (C++): change the type of a set of (pointer) variables and functions (i.e. change the types of the variables and return types of the functions)

❑ change access control mode: (e.g. 'private', 'protected', 'public')

Low-Level (iii): Renaming

❑ add method argument: to the definition of a method in its class (and subclasses that override it). In each method invocation, add an argument which is a dynamically created instance

❑ delete method argument: from the definition of a method in its class (and subclasses that override it). In each method invocation, the argument is removed

❑ reorder method arguments: in a method definition in its class (and subclasses that override it) and in all invocations of that method

❑ add method body: an existing method (signature)

❑ delete method body: from an existing method (making it a signature only)

Low-Level (iv): Moving

Moving an instance variable

- ❏ pull up instance variable to superclass:

  from all subclasses where it is defined

- ❏ push down instance variable to subclasses:

  from its current containing class to each of its immediate subclasses

Composite refactorings

- ❏ abstract access to instance variable:

  by defining accessor and mutator methods, and replacing all variable references with invocations of these methods

- ❏ Extract method

  define a new method, with a unique name, whose body is equivalent to a segment of an existing method. Replace that segment in the method with an invocation of the new method

- ❏ move class

  change the superclass of a class

Strategies for Refactoring

Top Ten of Code Bad Smells

"Let the code tell you where to refactor" and "If it stinks, changes it" [Kent Beck]
http://c2.com/cgi/wiki?CodeSmells

- ❑ Duplicated Code
- ❑ Long Method
- ❑ Large Class
- ❑ Long Parameter List
- ❑ Case Statement
- ❑ Divergent Change
- ❑ Shotgun Surgery
- ❑ Feature Envy
- ❑ Co-occuring Parameters
- ❑ Parallel Inheritance Hierarchies
- ❑ Lazy Class
- ❑ Similar Subclasses
- ❑ Temporary Field

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)

Long methods

- ❑ A method is the smallest unit of overriding
    - ☞ Extract pieces as smaller method
    - ☞ Comments are good delimiters
- ❑ Method body should be at the same level of abstraction: Template Method
- ❑ Except for crazy code (e.g. 600 lines to 5000 lines of code) metrics will not really help!
- ❑ Lot of temporary variables
    - ☞ Replace Method With Method Object (A method is really complex and can be an object)
- ❑ Long list of parameters
    - ☞ Introduce Parameter Object
    - ☞ Replace parameter with Method (send a message to an object you know instead of using a parameter)

Large 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
❏ Except for crazy code, metrics will not really help!
❏ For crazy method with lot of temporary
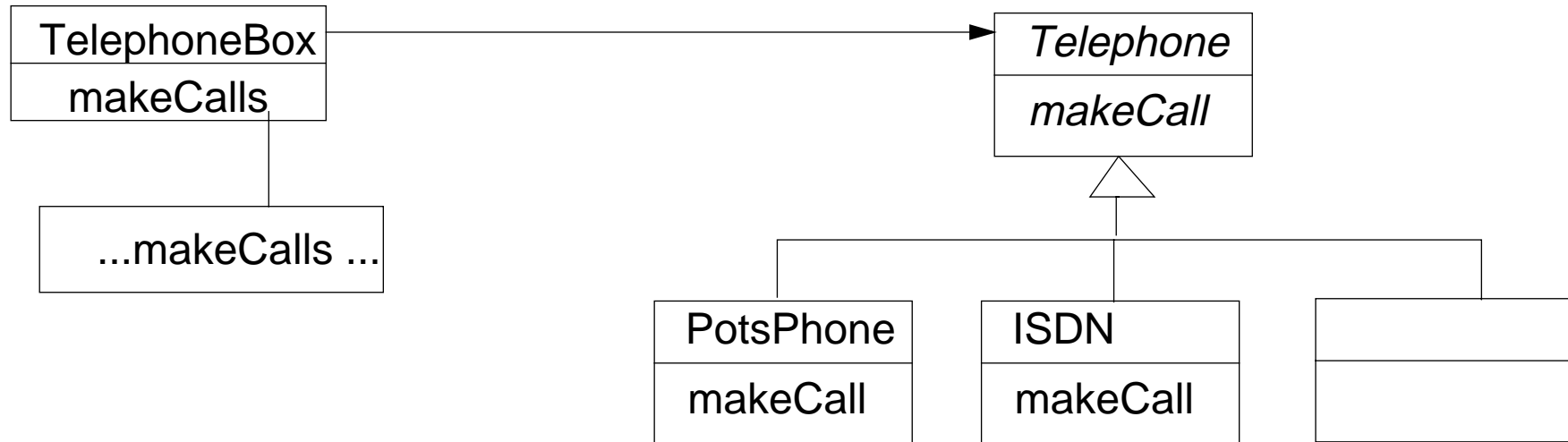   ☞ Replace Method with Object Method

Nested Conditionals

New cases should ideally not require changing existing code

☞   Use dynamic dispatch

☞   May apply the State / Strategy pattern

```
void makeCalls (Telephone * phineArray[])

    {

    for (Telephone *p = phoneArray;p;p++{

        switch (p->phoneType()){

        case TELEPHONE::POTS{

        POTSPhone * potsp = (POTSPhone *) p;

        potsp->tourneManivelle();

        potsp->call(); break}

        case TELEPHONE::ISDN

        ISDNPhone * isdnp = (ISDN*) p;

        isdnp->initializeLIne ().....

        ...

        }

    }
```

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

Nested Conditionals (ii)

```
┌─────────────────┐                                          ┌─────────────────┐
│ TelephoneBox    │ ───────────────────────────────────────▶│ Telephone       │
├─────────────────┤                                          ├─────────────────┤
│ makeCalls       │                                          │ makeCall        │
└─────────────────┘                                          └─────────────────┘
         │                                                            △
┌─────────────────┐                                    ┌──────────────┼──────────────┐
│ ...makeCalls ...│                          ┌──────────────┐  ┌──────────────┐  ┌──────────────┐
└─────────────────┘                          │ PotsPhone    │  │ ISDN         │  │              │
                                             ├──────────────┤  ├──────────────┤  ├──────────────┤
                                             │ makeCall     │  │ makeCall     │  │              │
                                             └──────────────┘  └──────────────┘  └──────────────┘
```
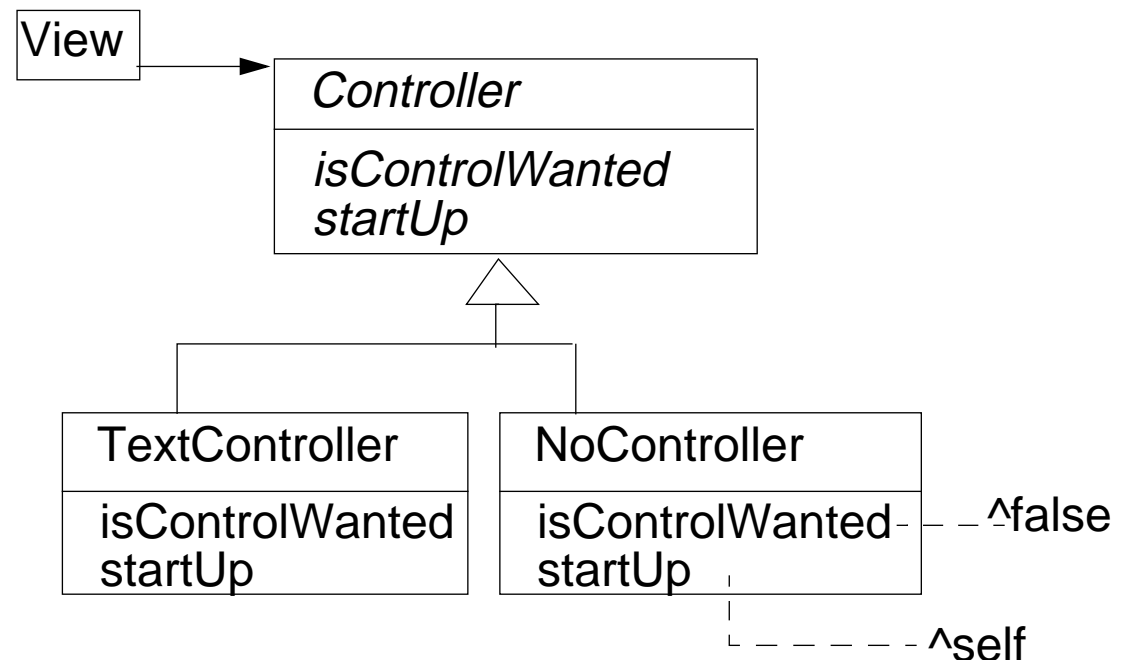
Nested Conditionals

❑ For default value testing

☞ consider the Null Object Pattern

```
VisualPart>>objectWaintingControl

    ...

    ^ ctrl isNil ifFalse:

            [ctrl isControlWanted

                ifTrue: [self]

                ifFalse: [nil]]


VisualPart>>objectWantingControl

    ...

    ^ctrl isControlWanted

                ifTrue: [self]

                ifFalse: [nil]]
```

```
+-------+          +------------------------+
| View  |--------->| Controller             |
+-------+          +------------------------+
                   | isControlWanted        |
                   | startUp                |
                   +------------------------+
```

```
+-------------------+      +-------------------+
| TextController    |      | NoController      |
+-------------------+      +-------------------+
| isControlWanted   |      | isControlWanted --|-- ^false
| startUp           |      | startUp           |
+-------------------+      +-------------------+
                                         |
                                         +------- ^self
```

Others

Co-occuring parameters (also called Data Clumps)

- ❑ Create missing abstraction (x y -> point)
- ❑ Pass it around
- ❑ Find methods that should be on the new object

Divergent Change

- ❑ If you have to change always the same methods when introducing new objects (database, financial instrument)
  - ☞ Extract Component

Shotgun Surgery

- ❑ Lot of little changes to a lot of different classes
  - ☞ Move Method, Move Instance Variable -> into a single class
  - ☞ Create a new class if needed

Feature Envy

- ❑ Functionality not define in the right class (e.g. a method calling a lot of getter from another object)
  - ☞ Extract Method, Move Method to the class that has most of the data

Summary

❑ Refactorings fit in real development process

- Clean the code then add new functionality

- Write negative lines of code if needed!

❑ Refactorings help you

- to write code that lives

- to build reliable software

- to write code that communicates their intent

❑ If you develop real software in Smalltalk, try the refactoring browser to make your own opinion

Tool Support and Bibliography

❑   In Smalltalk use the Refactoring Browser

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

Bibliography

❑   W. Opdyke, "Refactorings Object-Oriented Frameworks", PhD Thesis, University of Illinois, 1992

❑   R. Johnson and W. Opdyke, ``Refactoring and Aggregation", Object Technologies for Advanced Software, LNCS, vol. 742, Springer-Verlag, 1993,

❑   W. Opdyke and R. Johnson, ``Creating Abstract Superclasses by Refactoring," Proceedings of the 1993, ACM Conference on Computer Science, ACM Press, 1993, pp. 66-73.

❑   Fowler'99, ??, Addison-Wesley,1999

❑   D.Roberts, J. Brant and R. Johnson: "Using the Refactoring Browser Will it increase your efficiency?", The Smalltalk Report, Vol 6, Sep, 1997

❑   D. Roberts, J. Brant and R. Johnson, "A Refactoring Tool for Smalltalk", Theory and Practice of Object Systems special Issue on Software Reengineering, 1998

9. Refactoring Tools

Outline

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

Why Refactoring Tools?

Relative Effort of Maintenance
[Software Engineering
(Sommerville)]
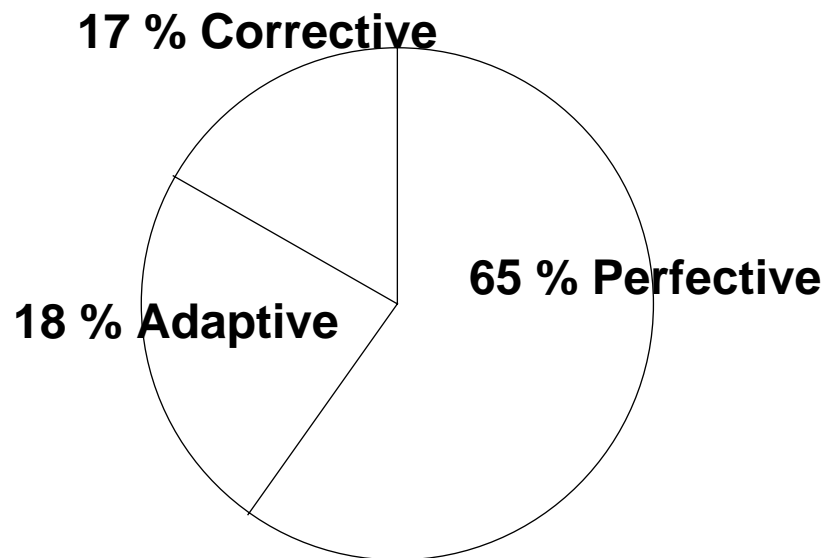
Relative cost of fixing mistakes
[201 Principles of Software
Development (Davis)]

17 % Corrective

18 % Adaptive

65 % Perfective

x 200

x 20

x 10

x 5

x 1

requirement   coding   delivery

design   testing

Iterative Development Life-cycle

**Initial
Requirements**

PROTOTYPING

**New / Changing
Requirements**

EXPANSION

CONSOLIDATION

**More
Reuse**
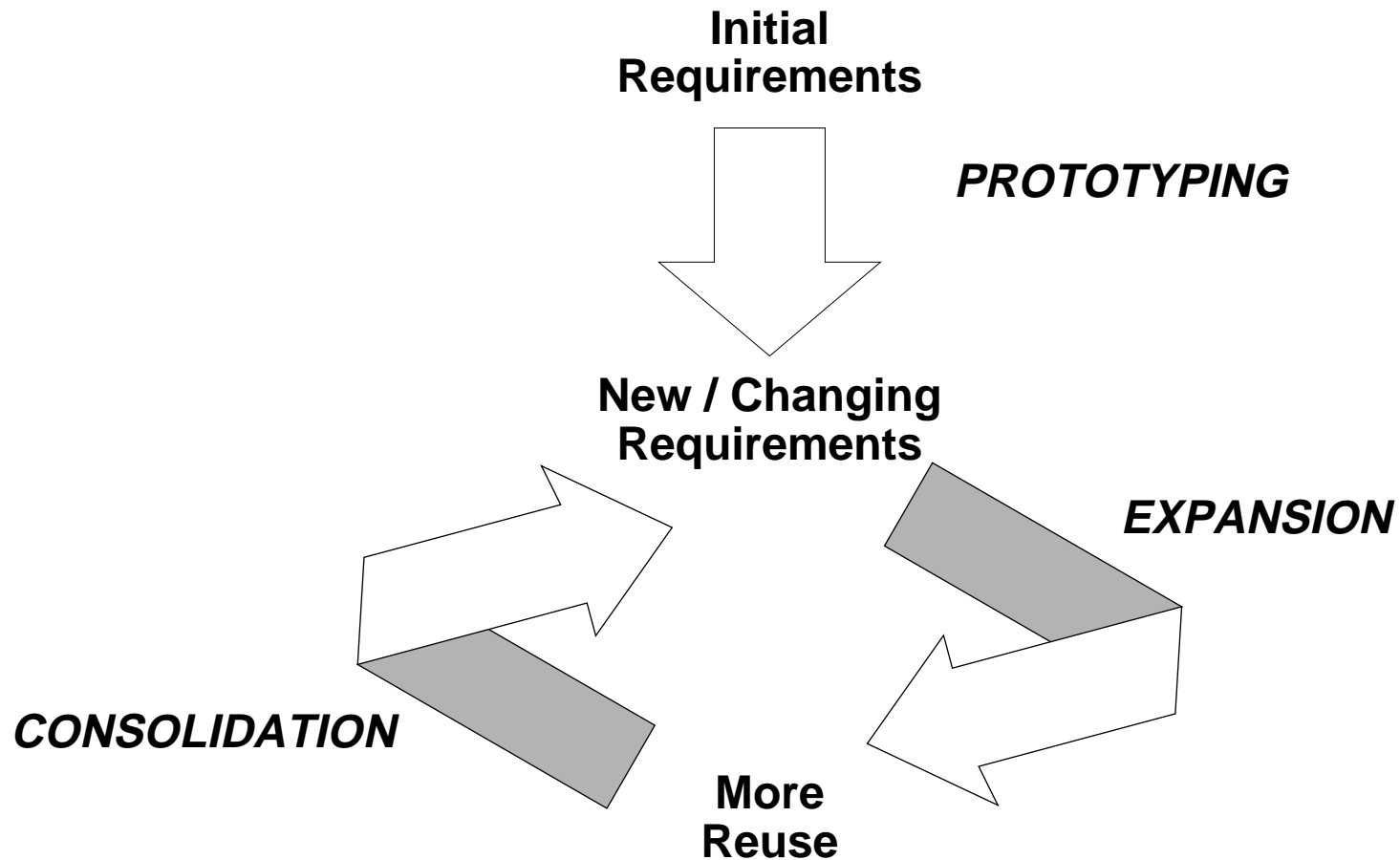
Which Refactoring Tools?

Change Efficient

Refactoring

- ❑ Source-to-source program transformation
- ❑ Behaviour preserving

=> *ameliorate the program structure*

Programming Environment

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

=> *convenient for "local" amiliorations*

Failure Proof

Regression Testing

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

=> *verify if ameliorated structure does not screw up previous work*

Configuration & Version Management

- ❑ keep track of versions that represent project milestones

=> *possibility to go back to previous version*

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: Class Diagram

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: Contracts

Ensure the "*secure*" and "*reliable*" requirements.

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


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


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

Prototype Implementation

=> see demo "IBanking1"

Include test cases for

❑ IBCustomer
  - customerNr()
❑ IBAccount
  - getBalance()
  - setBalance() [#Demo of rapid edit/compile/run#]
❑ IBBank
  - createAccountForCustomer() (initBank)
  - 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 reverse the above
    + 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

❑ *concurrency*: concurrent access of customers

Add test case for

❑ IBBank

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

Expanded Design: Class Diagram

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
...
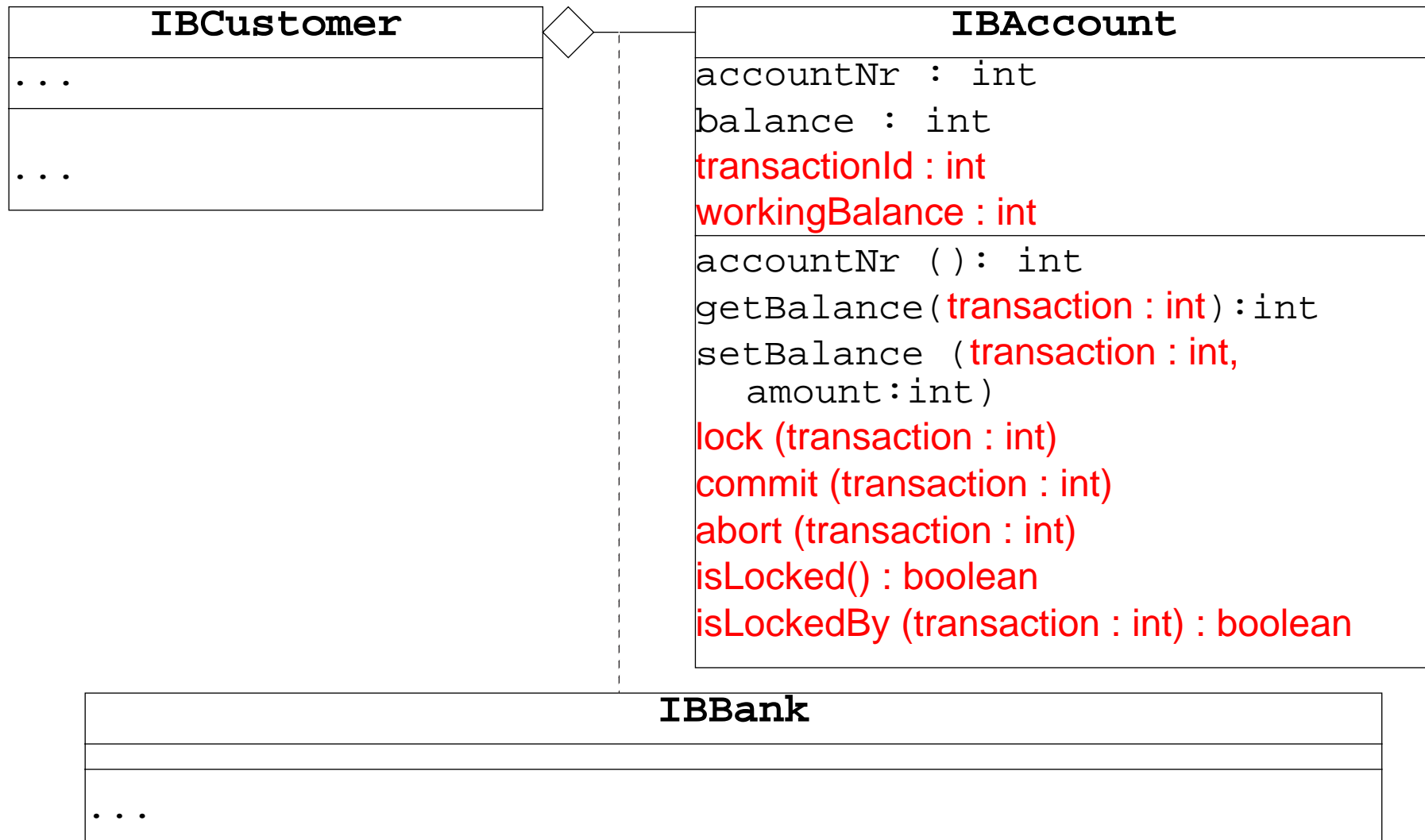
Expanded Design: Contracts

```
getBalance(transaction:int): int
        require: isLockedBy(transaction)
        ensure:
setBalance(transaction:int, amount: int)
        require: isLockedBy(transaction)
        ensure: getBalance(transaction) = amount
lock(transaction:int)
        require:
        ensure: isLockedBy(transaction)
commit(transaction:int)
        require: isLockedBy(transaction)
        ensure: NOT isLocked()
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"
- ❏ apply "add class" to create "IBTransactionManager"
- ❏ use normal editing to expand functionality of "seeBalance()" and "transfer()"
  => load "IBanking2"

Expand Tests
- ❏ previous tests for "getBalance()" and "setBalance()" should now fail
- ❏ new tests for the transaction contracts, incl. commit and abort
- ❏ testConcurrent works!
  => we can confidently ship a new release

Consolidation: More Reuse (Problem Detection)

More Reuse

Assume the following scenario

- ❏ Other people applied a similar transaction interface to customer (change of name, address, password, ...)
- ❏ A design review reveals that this "transaction" stuff is a good idea and should become a separate component

Code Smells

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

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: More Reuse *(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

Consolidation: More Reuse *(Refactoring Sequence)*

❑ apply "create subclass" to create "IBLockable" with "IBAccount" as subclass

❑ apply "abstract attribute" to "transactionId" [#check source code#]

❑ apply "pull up attribute" to move "transactionId" up

❑ apply "push up" to move accessor methods on transactionId up

❑ apply "push up" to "isLocked", "isLockedBy", "notLocked"

❑ apply "push up" to "abort:", "commit:", "lock:"

- fails; accesses local attributes

- apply "extract method" on groups of accesses to local attributes
  (Do you want to extract assignment ? -> Yes)

- apply "push up" to "abort:", "commit:", "lock:"

- define the extracted methods as new abstract methods in the hierarchy
  (this is done via normal editing; not semantic preserving !)

❑ apply "inline all self sends" on accessor methods on transactionId up

❑ check references to "transactionId" to check if all of them are local

Conclusion (1/2)

Do not apply refactoring tools in isolation

Combine with ...

	Smalltalk	*C++*	*Java*
❑ refactoring tools	+	- (?)	...
❑ rapid edit-compile-run cycles	+	-	+-
❑ reverse engineering facilitie	+-	+-	+-
❑ regression testing !!	+	+	+
❑ version & configuration management	+	+	+

Know when is as important as know-how

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

Conclusion (2/2)
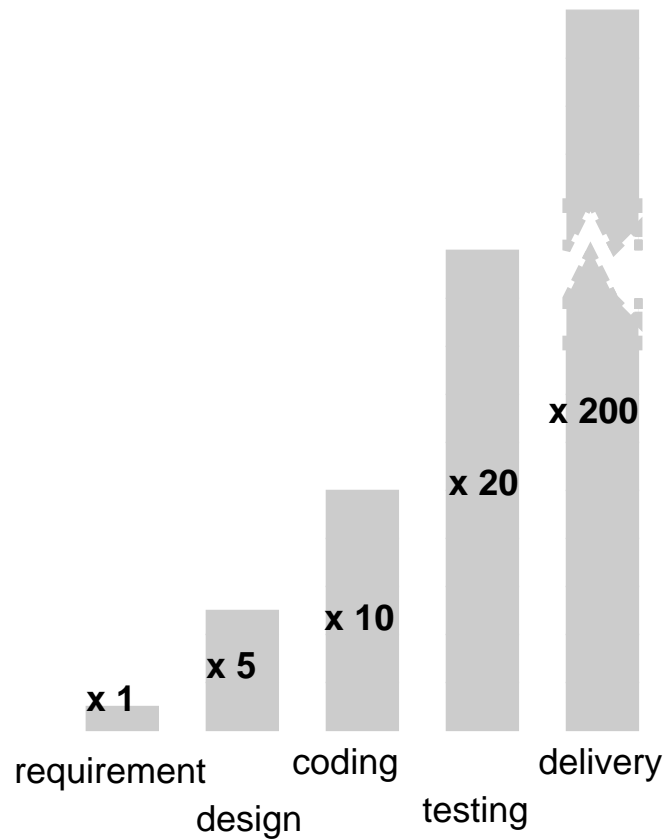
With proper

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

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

The tools are there ...

x 200

x 20

x 10

x 5

x 1

requirement

design

coding

testing

delivery

x 5 **x 5** **x 5** **x 5**

x 1

requirement

design

coding

testing

delivery
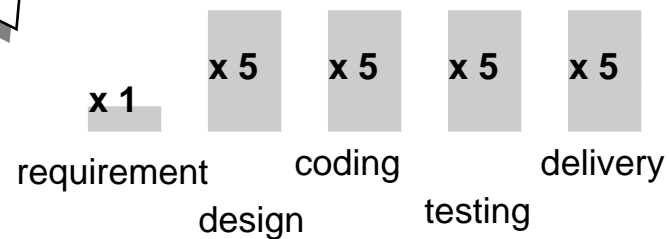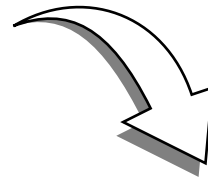
10. Metrics in OO Reengineering

Outline

❏ Why Metrics in OO Reengineering?

❏ Which Metrics to Collect?

- Goal-Question-Metric paradigm

- Metric Definitions

❏ Case studies

❏ Applicability for...

- Problem Detection

- Stability Assessment

- Reverse Engineering

❏ Conclusion
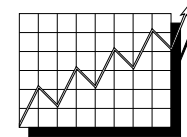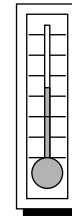
Literature

❏ Norman E. Fenton, Shari l. Pfleeger, "Software Metrics: A rigorous & Practical Approach", Thompson Computer Press, 1996.

❏ Mark Lorenz, Jeff Kidd, "Object-Oriented Software Metrics", Prentice Hall, 1994.

❏ Brian Henderson-Sellers, "Object-Oriented Metrics: Measures of Complexity", Prentice Hall, 1996.

Why Metrics in OO Reengineering?

OO Reengineering = Iterative Development

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

   => See lecture 3, "Metrics in Industry"

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 reengineering tool!

Which Metrics to Collect (GQM)?

Goal

❑ Support reverse and reengineering of object-oriented programs in an iterative development context.

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?
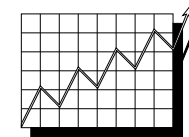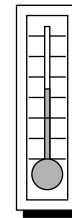
Metric

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

=> Collect from source code

Which Metrics to Collect (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)

inherits

Class

belongsTo

Method   access   **Attribute**

invokes
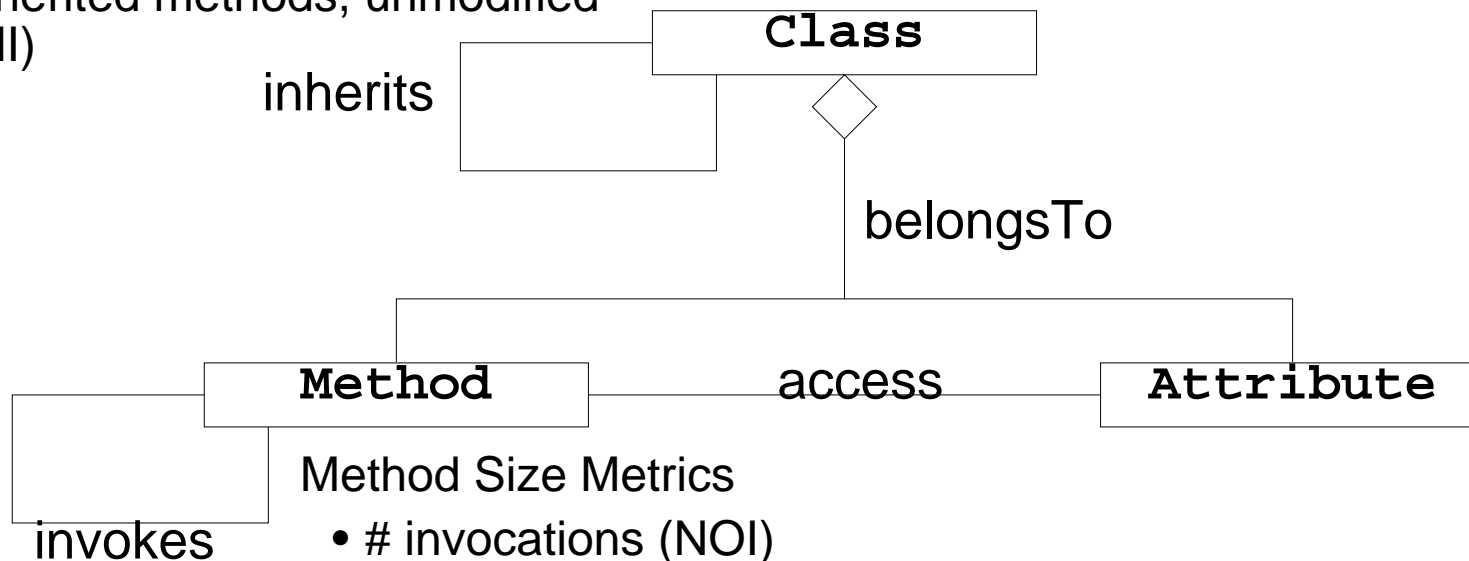
Method Size Metrics
- # invocations (NOI)
- # statements (NOS)
- # Lines of Code (LOC)

Case Studies

Requirements

- ❑ *Representative* for iterative OO development.
  => acceptance of results
- ❑ Development occurred *independent*.
  => experiments did not influence results
- ❑ Source code is publicly *accessible*.
  => reproducing results
- ❑ Functionality changes are *documented*.
  => validating experimental findings

Case Studies

- ❑ VisualWorks (4 releases)
- ❑ HotDraw (3 releases)
- ❑ Refactoring Browser (2 releases)

Problem Detection: Experiment

Question:

❑ Can metrics predict which parts of the design will cause trouble in future extensions?

Release n   *Release n+1*

{$\forall$ Metric m, Component c_n |
    $m(c_n)$ > threshold}
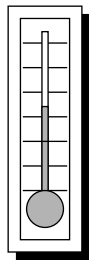
1. c_{n+1} deleted
2. $m(c_{n+1})$ <= threshold
3. $m(c_{n+1})$ $m(c_n)$
4. $m(c_{n+1})$ > $m(c_n)$

Cases 1 & 2 confirm the prediction capabilities of metric m
Cases 3 & 4 disprove the prediction capabilities of metric m

Problem Detection: Results

- 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

- some metrics perform better than others (WMC-NOI, WMC-NOS)?

=> expensive parsing

- 80%-20% distribution as a litmus test
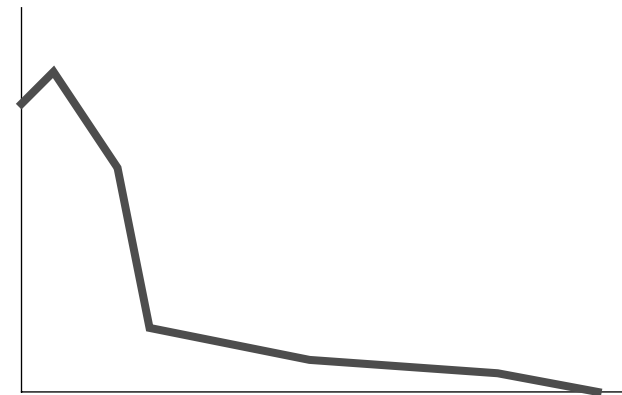
Stability Assessment: Experiment

Question:

❑   Can metrics assess which parts of the design are unstable?

Release n-1   *Release n*

$\{\forall$ Metric m, Component c_{n-1}, Component c_n |
$m(c_{n-1})$ $m(c_n)\}$

Qualitative Analysis
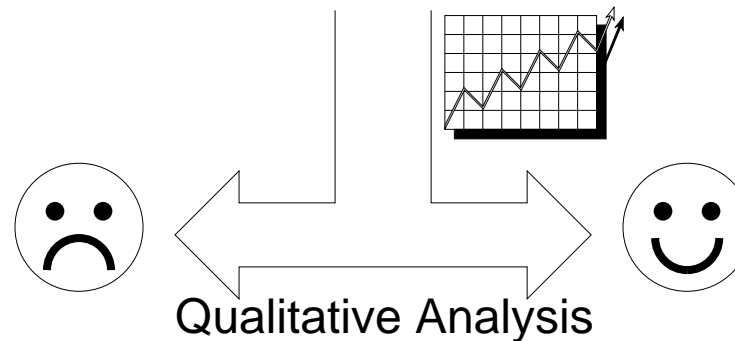
Stability Assessment: Results

- all documented changes are detected
- some undocumented changes are detected
- all changes are relevant

=> no false positives

- changes may go unnoticed

=> false negatives are possible

Reverse Engineering

Question:

❑   Can metrics reveal which parts of the design have been refactored?

Four heuristics for reverse engineering

❑   Split into Superclass / Merge with Superclass

❑   Split into Subclass / Merge with Subclass

❑   Move to Other Class (Superclass, Subclass, Sibling class)

❑   Split Method / Factor Out Common Functionality

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
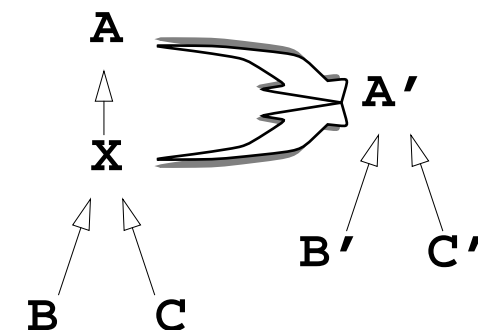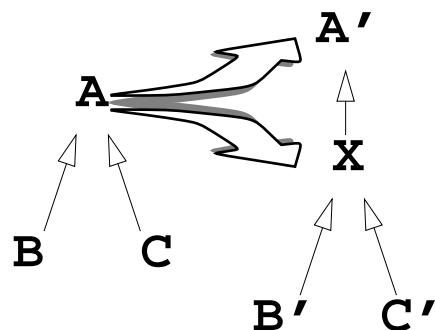
❑ Include changes in "# inherited methods" (NMI) to assess overall protocol

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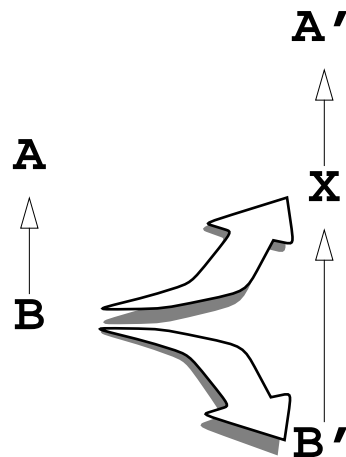
Split B into X and B'
(delta_HNL(B') > 0) and
    ((delta_NOM(B') < 0)
    or   (delta_NIA(B') < 0)
    or   (delta_NCA(B') < 0))

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))
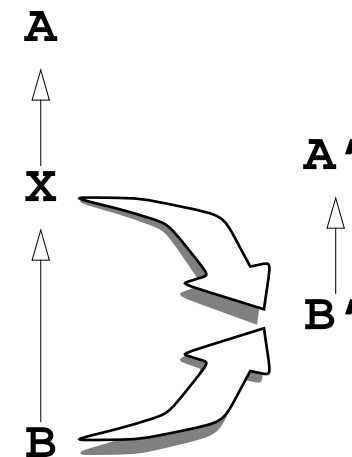
MERGE

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 $(delta\_NIA(A') < 0)$
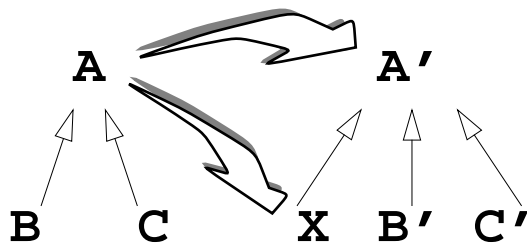
or $(delta\_NCA(A') < 0))$

MERGE

Merge X and A into A'

$(delta\_NOC(A') <> 0)$ and

($(delta\_NOM(A') > 0)$

or $(delta\_NIA(A') > 0)$
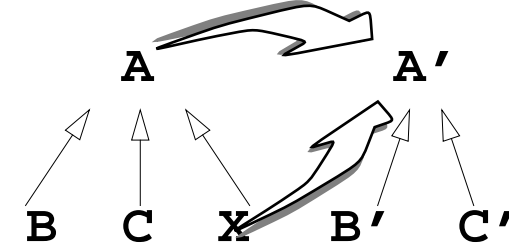
or $(delta\_NCA(A') > 0))$

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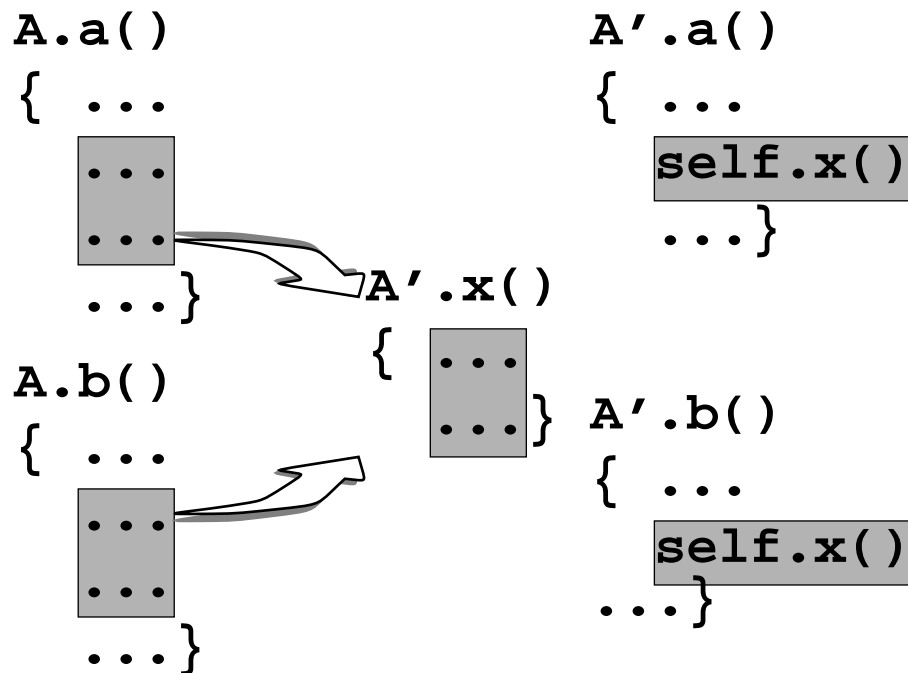
MOVE
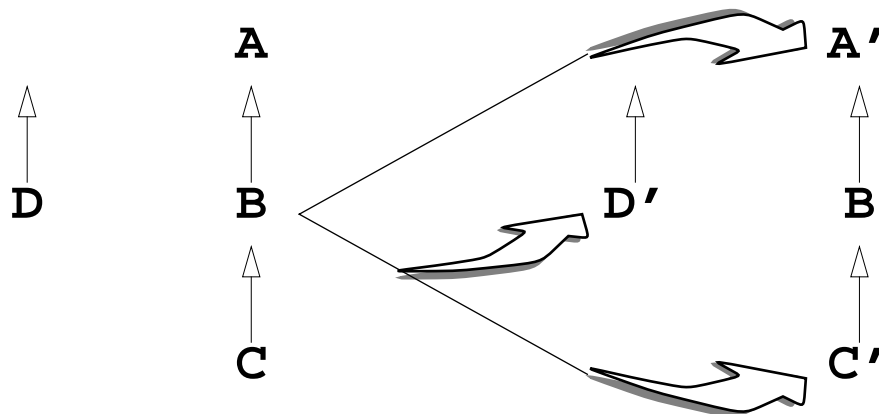
Move from B to A', C' or D'

$($ (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

```
A.a()
{ ...
  ...
  ...
  ...}

A.b()
{ ...
  ...
  ...
  ...}
```

```
A'.a()
{ ...
  self.x()
  ...}

A'.x()
{ ...
  ...}

A'.b()
{ ...
  self.x()
  ...}
```

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()))

Heuristics Performance

	Accuracy	*False Negatives*	*False Positives*
Split into Superclass / Merge into Superclass	*0%-3% (0-20)*	*Yes, but...*	*Yes, but...*
Split into Subclass / Merge into Subclass	*1% - 6% (11-43)*	*Yes, but...*	*Yes, but...*
Move to Superclass, Subclass, Sibling Class	*1%-10% (12-79)*	*Yes, but...*	*Yes, but...*
Split Method / Factor Common Functionality	*1%-3% (42-211)*	*Yes, but...*	*Yes, but...*

It works, but...

- ❏   vulnerable to renaming
- ❏   imprecise for large restructuring sequences
- ❏   by experts only
- ❏   heavyweight technique

Main advantage: focuses on interactions between classes and methods!

Conclusion (1/2)
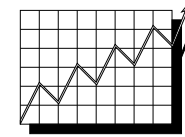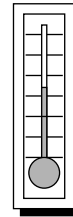
Question

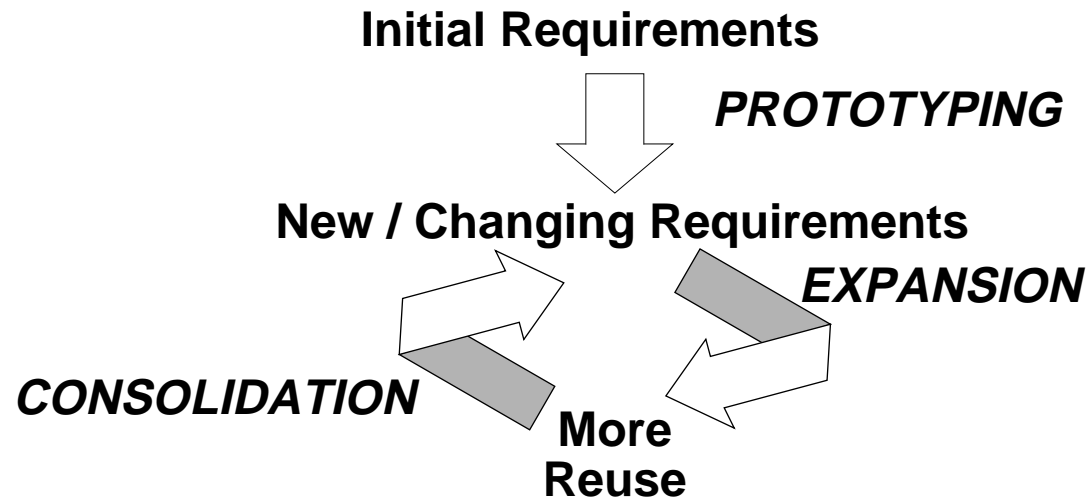*Can metrics help to answer the following questions?*

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

Conclusion (2/2)

Initial Requirements

PROTOTYPING

New / Changing Requirements

EXPANSION

CONSOLIDATION

**More
Reuse**

❑ Iterative development is both a blessing and a curse
=> exploit the presence of versions
=> metrics are but one of the ways

11. Reengineering Repositories

Outline

- ❏ Why Reengineering Repositories?
- ❏ What is a Reengineering Repository?
- ❏ Taxonomy (Functionality + Integration Options)
- ❏ What to Store?
- ❏ How to Obtain Data?
- ❏ Exchange Standards

Literature

- ❏ Ian Sommerville, Software Engineering Fifth Edition, Addison-Wesley, 1996.
- ❏ Roger S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill, 1994.
- ❏ Alan M.Davis, 201 Principles of Software Development, McGraw-Hill, 1995.

Why Repositories - CARE

Tools

Tools are necessary to improve productivity.

However: Give Software Tools to Good Engineers [Davis'95]
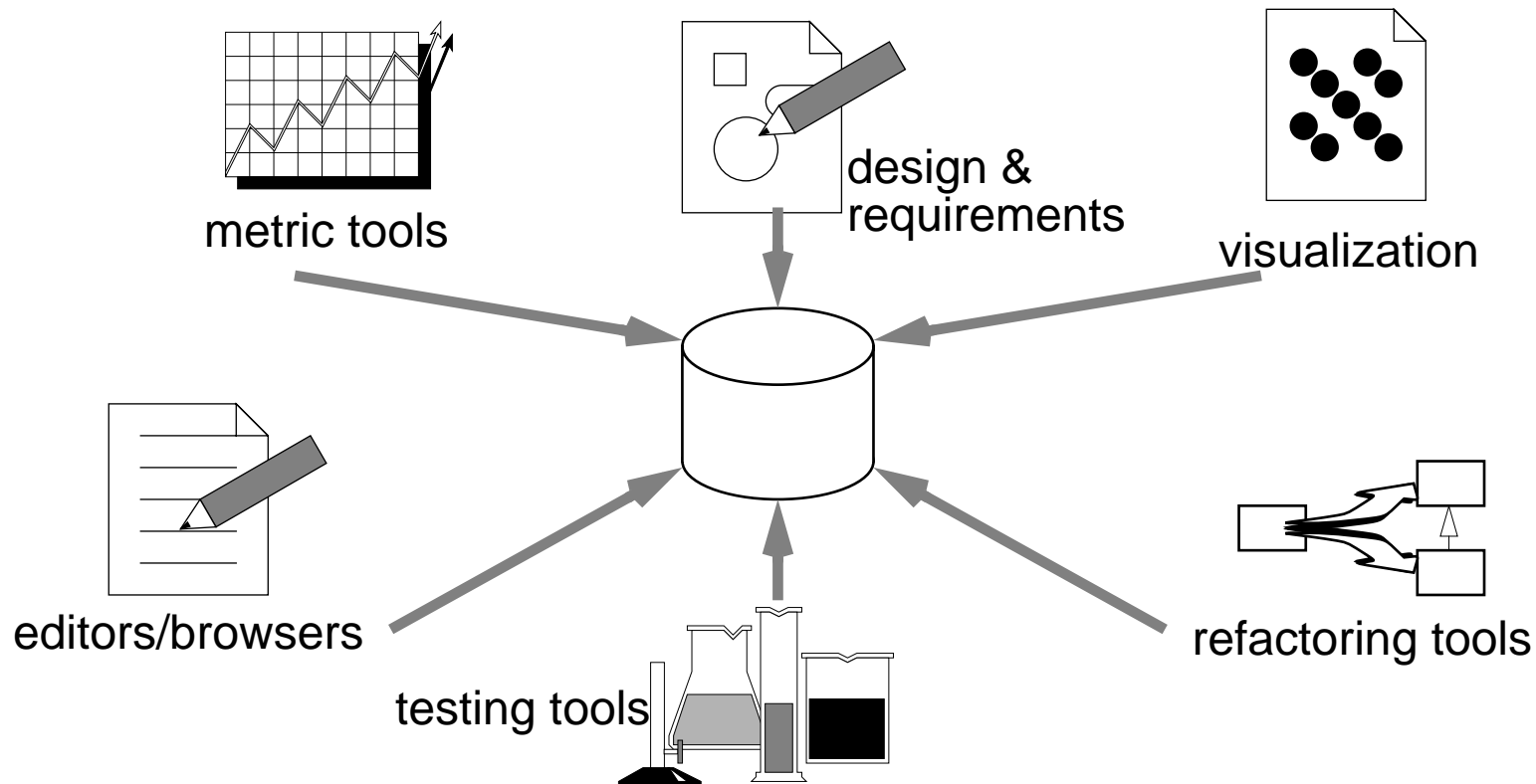*(You want bad engineers to produce less, not more, poor-quality software)*

Towards CARE

❑ Late 70's: CAD/CAM = Computer Aided Design / Manufacturing

= Drawing tools to create and validate design diagrams and steer manufacturing processes

❑ Late 80's: CASE = Computer Aided Software Engineering

= Tool support for (parts of) the Software Engineering Process

❑ CARE = Computer Aided Reengineering

= Automated support for Software Reengineering

- Y2K tools

- Iterative Development (i.e., *round-trip engineering*)

Why Repositories - Tool Integration

Reengineering tools must work together

❑   They are supplied by different vendors

❑   They must share data

   => need a common "database"

metric tools

design & requirements

visualization

editors/browsers

testing tools

refactoring tools

What is a Repository - Definition?

Definitions for Repository

- ❑ A repository is a shared database of information about engineered artifacts. (Philip Bernstein, "An overview of Repository Technology", Proceedings of the 20th VLDB Conference - 1994)

- ❑ A repository is a database that acts as the centre for both accumulation and storage of software engineering information. [Pressman'94]

- ❑ The tools are integrated around an object management system which includes a public, shared data model describing the data entities and relationships which can be manipulated by the tools. This model is accessible to all tools but not an inherent part of them. [Sommerville'96]

=> Reengineering Repository

- ❑ A reengineering repository is a common database used for integrating reengineering tools and providing all relevant information about the software system to be reengineered.

What is a Repository - Issues?

A reengineering repository is a common <u>database</u> used for <u>integrating</u> reengineering tools and <u>providing</u> all <u>relevant</u> information about the software system to be reengineered.

Issues

❑ Wide range of supporting "database" technology

Flat file <-> Relational Database <-> Object Database

❑ PROVIDING all information?

Often a reference to the actual place where the information is stored

=> Repository maintains meta-data, i.e. data about the data

❑ RELEVANT information?

Who decides what is relevant?

=> open and extensible database schema (i.e., repository model)

❑ Tool integration?

Not all tools work together smoothly

- different information needs

- not designed to work together

=> redundant information + necessary patching

Taxonomy - Functionality

What kind of functionality does one expect from a CARE-tool?

Adapted from [Sommerville'96]

Classification	Examples of tool functionality	CARE-specific
• Tracking	Requirement Tracking, Bug Tracking	x
• Configuration Management	Version Mngmnt., Configuration Mngmnt	
• Source Code Manipulation	Code Editors, Browsers	
• Restructuring	Pretty-printing, Refactoring	x
• Data Collection	Parsers, Run-time Data Collectors	x
• Data Analysis	Metrics, Visualisation, Clustering	
• Testing	Regression Tests, Coverage Analysis	

Buying all these tools from a single vendor is not a realistic scenario
=> need for integration facilities

Taxonomy - Integration

What kind of integration facilities does one expect in a CARE-tool?

Adapted from [Pressman'94]

Classification	Examples of integration facilities
• Completely Closed	--
• Data Exchange	Import/Export facilities available in most tools
• Common Tool Access	Unix shell, emacs
• Common Data Management	Link libraries, Java byte-code
• Data Sharing	java.lang.reflect
• Interoperability	Most "open environments" from a single vendor

Tool integration is even more important in reengineering than in forward engineering
- ❑ forward engineering tools are chosen deliberately
- ❑ reengineering tools must integrate with what's already in place

Integration Example (1/3)

Data Sharing

Here's a piece of Java-code that uses the reflection facilities to inspect a class

```java
import java.lamg.reflect.*;

public class ClassInspector
{
    ... /* definition of auxiliary methods Print... */

    public static void Inspect (Class c) {
       System.out.println("Contents of class " + c.getName());
       PrintFields (c.getFields());
       PrintConstructors(c.getConstructors());
       PrintMethods(c.getMethods());
    }
}
```

Integration Example (2/3)

Interoperability

Here's a piece of C-code which accesses the SNiFF+ API

```
int main ( int argc, char *argv[] )
{ SNiFFACCESS slot;
  .... /*other declarations */

  ParseArgs( argc, argv, &host, &proj, &session );
  __si__module__init( );
  slot = si_open( session, host );
  if( slot && si_open_project( slot, proj ) )
    {full = si_Query(eQImplFiles,eSGlobal,0);
    .... /* enumerate pointer structure in 'full' */
    si_close_project( slot, proj );
    }
  si_exit(slot);
return 0;
}
```

Integration Example (3/3)

Interoperability

Here's a piece of VisualBasic-code which accesses the Rational/Rose API

```
Sub GenerateClassIn (theClassName As String,
      theCategory As Category)
   Dim theClass As Class
   Set theClass = theCategory.AddClass(theClassName)
End Sub
Sub GenerateInheritanceIn (theSubclassName As String,
      theSuperclassName As String, theCategory As Category)
   Dim theSub As Class
   Dim theInherit As InheritRelation
   Set theSub = theCategory.GetAllClasses().GetFirst(theSubclassName)
   Set theInherit = theSubclass.AddInheritRel("", theSuperclassName)
End Sub
Sub Main
   Dim theCategory As Category
   Set theCategory = RoseApp.CurrentModel.RootCategory
   GenerateClassIn "JPeg1", theCategory
   GenerateClassIn "JPEGImageDescription", theCategory
   GenerateInheritanceIn "JPEGImageDescription", "JPeg1", theCategory
End Sub
```

What to Store?

- Core OO constructs
  - ❏ Classes
  - ❏ Inheritance
  - ❏ Methods
  - ❏ Attributes
  - ❏ Types
- Procedural constructs
  - ❏ Packages
  - ❏ Functions & Procedures
  - ❏ Global Variables
- Details
  - ❏ Formal Parameters
  - ❏ Other Variables (local,...)
  - ❏ Arguments

- Relations
  - ❏ {Methods, Functions, Procedures} *access* {Attributes, Variables}
  - ❏ {Methods, Functions, Procedures} *invoke* {Methods, Functions, Procedure}
  - ❏ {Classes} *own* {Methods, Attributes,...}
  - ❏ {Packages} *own* {Classes, ,...}
  - ❏ {Methods, Functions, Procedures} *accept* {Formal Parameters}
  - ❏ {Invocations} *bind* {Formal Parameters} *to* {...}
  - ❏ {Types} *correspond to* {Classes}
  - ❏ {Attributes, Formal Parameters, Variables,...} *have* {Types}
  - ❏ {Types} *correspond to* {Classes}

How to Obtain Data (1/4)?

Build your own parser

- Technique
  - ❑ Use parser generator to build a parser for the language
- Advantage
  - ❑ Full control {dialects, pre-compilers}
- Disadvantage
  - ❑ Experts only (formal syntax grammars)
  - ❑ Costly
  - ❑ Uncertain about reliability and scalability
  - ❑ Build your own = Maintain your own
  - ❑ Tools to integrate with require source code or API
- Remarks
  - ❑ C++ requires full control (lot's of dialects + pre-compiling tricks)
  - ❑ ... but 100% reliability is very difficult for parser generators

How to Obtain Data (2/4)?

Translate between file-formats

- Technique
  - ❑ Build gateways between existing tools by translating import/export file formats
- Advantage
  - ❑ Relatively cheap (assuming formats are documented)
  - ❑ Offers reasonable integration
  - ❑ Reasonable scalability (limited by file system)
- Disadvantage
  - ❑ Faith in external tools
  - ❑ Maintenance is difficult (future releases easily change file-formats)
  - ❑ Effort to be duplicated for every tool
- Remarks
  - ❑ Works only when few gateways must be build
  - ❑ Standardization efforts are under way (CDIF, MOF)
    => tackles "maintenance" and "duplication of efforts" problems
    => improves scalability and allows multiple tools

How to Obtain Data (3/4)?

**Communicate via API's (application programmer's interface)**

- Technique
    - ❑ Build gateways between existing tools using wrappers that extract info via API's
- Advantage
    - ❑ Cheap
    - ❑ Good integration
    - ❑ Good scale-up (limited by wrapping tool)
    - ❑ Maintenance effort is reasonable (API's don't change that frequently)
- Disadvantage
    - ❑ Faith in external tools
    - ❑ Effort to be duplicated for every tool
    - ❑ Robustness
- Remarks
    - ❑ Works only when few gateways must be build
    - ❑ May be combined with "Translate between file-formats"

How to Obtain Data (4/4)

Collect Execution Traces

- Technique
  - ❑ Acquire traces of sequences of method invocations

    (code instrumentation, method wrapping, debuggers, virtual machines)
- Advantage
  - ❑ Good insight in the 'real' execution trace
- Disadvantage
  - ❑ Expensive with current state of the art
  - ❑ Relies on reliable usage scenarios
  - ❑ Explosive data-growth
- Remarks
 - ❑ Currently not often used, but gives spectacular results

Exchange Standards

EXCHANGE VIA FLAT ASCII STREAMS
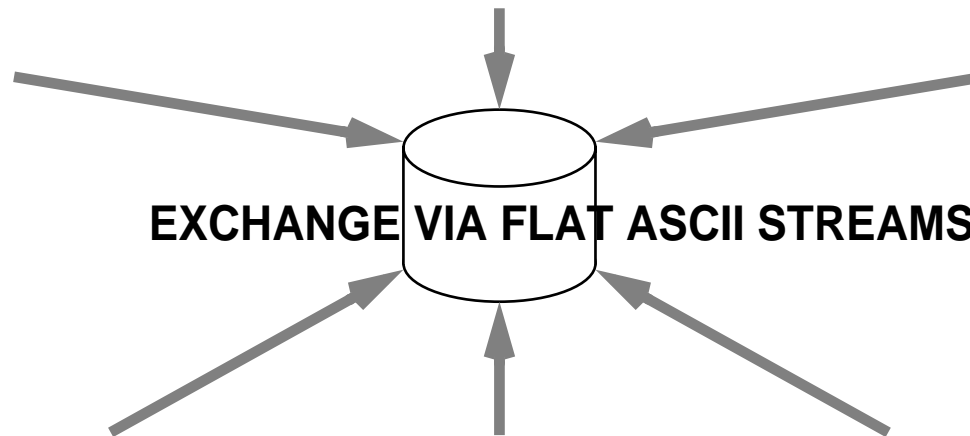
Standardization Efforts

❑ CDIF (CASE data interchange format) - see http://www.eigroup.org/
Mature standard (approved by ISO).
Little commitment from tool vendors

❑ MOF (Meta-Object Facility) from OMG - see http://www.omg.org/
Currently immature.
Major commitment from tool vendors to be expected
Builds on UML and XML.

Exchange Standards
- Meta Models

Issue: Who decides what is relevant information? Tools have different information needs

> => open and extensible repository model

> => need for a meta-model
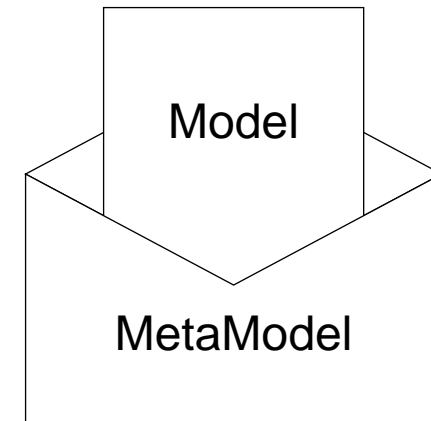
Model

MetaModel

Layer	Description	Example
User Objects	Describes a specific situation in an information domain.	Student#3, Course#5, Student#3.enrolled_in.Course#5
Model	Defines a language to describe an information domain.	Student, Course, enrolled_in
Meta Model	Defines a language for specifying Models	Class, Attribute, Operation
Meta Meta Model	Defines the core ingredients sufficient for defining languages for specifying meta-models	MetaEntity, MetaRelationship

<u>*Exchange Standards - CDIF example*</u>

```
CDIF, SYNTAX "SYNTAX.1" "02.00.00", ENCODING "ENCODING.1" "02.00.00"

(:HEADER ...
   (:META-MODEL
      (:SUBJECTAREAREFERENCE Foundation
         (:VERSIONNUMBER "01.00"))
      ...)
```

► Obligatory Introduction Stuff

```
   (MetaEntity Class
      (Name *Class*))
   (MetaAttribute nameClass
      (Name *name* )
      (DataType <StringValue>)
      (isOptional -FALSE-))
   (MetaAttribute.IsLocalMetaAttributeOf.AttributableMetaObject
      nameClass Class)
   ...
)
```

► Definition of a meta-model concept "Class" as having one attribute "name"

```
(:MODEL ...
   (Class 001
      (Name "Student"))
   (Class 002
      (Name "Course"))
   ...)
```
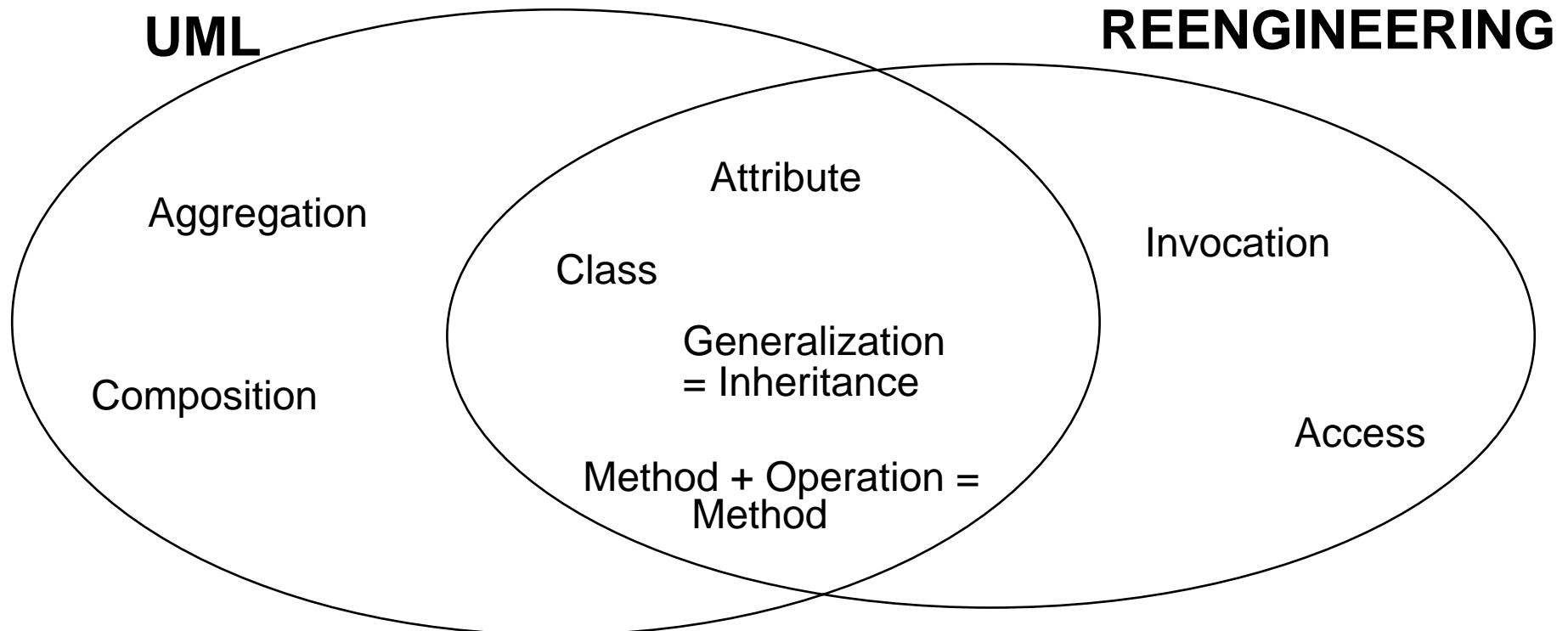
► Definition of 2 classes "Student" & "Course"

Exchange Standards - UML ?

Current standardization efforts are geared towards UML. This may not be enough for reengineering purposes.

UML   **REENGINEERING**

Aggregation

Attribute

Class

Invocation

Generalization
= Inheritance

Composition

Access

Method + Operation =
Method

- ❑   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
- ❑ Tool integration via data exchange and/or data sharing
  - Repositories as the central "database"
  - A repository is not so much a passive data store ...
  - ... but more an object which knows where the data is
- ❑ "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 emphasis may cause problems