# ***Object-Oriented Software Reengineering***

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

Wintersemester 1999/2000

# Table of Contents


Table of Contents ii
***1. Object-Oriented Software Reengineering 1***
Goals of this course 2
Course Overview 3
Lehman's Laws 4
What is a Legacy System? 5
Software Maintenance 6
Why is Software Maintenance Expensive? 7
Factors Affecting Maintenance 8
What about OO? 9
Definitions 10
Reverse and Reengineering 11
Goals of Reverse Engineering 12
Reverse Engineering Techniques 13
Goals of Reengineering 14
Reengineering Techniques 15
Architectural Problems 16
Refactoring Opportunities 17
Tools Architectures 18
Summary 19
***2. Code Duplication 20***
Overview 21
Code is Copied 22
How Much Code is Duplicated? 23
What Is Considered To Be Copied Code? 24
How Code Gets Copied 25
Why Code Gets Copied 26
What Problems Stem From Copied Code? 27
Code Duplication: Problem Statement 28
Code Duplication Detection 29
General Schema of Detection Process 30
Simple Detection Approach I 31
Simple Detection Approach II 32
Detection Using Parameterized Matching I 33
Detection Using Parameterized Matching II 34
Detection using Abstract Syntax Trees I 35
Detection using Abstract Syntax Trees II 36
Refactoring Duplicated Code I 37
Refactoring Duplicated Code II 38
Visualization of Duplicated Code 39
Visualization of Copied Code Sequences 40
Visualization of Repetitive Structures 41
Visualization of Cloned Classes 42
Visualization of Clone Families 43
Summary 44
References 45
***3. Lab session — Duploc 46***
***4. Design Extraction 47***
Goals 48
Outline 49
Why Design Extraction is needed? 50
UML (Unified Modelling Language) 51
The Little Static UML 52
Road Map 53
Let us practice! 54
A First View 55
Evaluation 56
A Cleaner View 57
Road Map 58
Three Essential Questions 59
Interpreting UML 60
Levels of Interpretations: Perspectives 61
Attributes in Perspectives 62
Operations in Perspectives 63
Associations 64
Associations: Conceptual Perspective 65
Associations: Specification Perspective 66
Arrows: Nagivability 67
Generalization 68
Road Map 69
Need for a Clear Mapping 70
Private you said?! Which one? 71
Class Method Inheritance?! 72
Some Possible Smalltalk Conventions 73
Stereotypes: to Represent Conventions! 74
Another Example: Instance/Class Associations 75
RoadMap 76
Association Extractions (i) 77
Language Impact on Extraction 78
Method Signature for Extracting Relation 79
Convention Based Association Extraction 80
Operation Extraction (i) 81
Operation Extraction (ii) 82
Road map 83
Design Patterns as Documentation Elements 84
Road map 85
Evolution Impact Analysis: Reuse Contract 86
Example 87
Reuse Contracts: General Idea 88
Example 89
Road Map 90
Documenting Dynamic Behaviour 91
Sequence Diagrams 92
Statically Extracting Interactions 93
Dynamically Extracting Interactions 94
Lessons Learnt 95

***5. Software Metrics*** ***96***
Why Measure Software? 97
What is a Metric? 98
GQM 99
Metrics assumptions 100
Cost estimation objectives 101
Estimation techniques 102
Algorithmic cost modelling 103
Measurement-based estimation 104
Lines of code 105
Function points 106
Programmer productivity 107
The COCOMO model 108
Basic COCOMO Formula 109
COCOMO assumptions 110
Product quality metrics 111
Maintainability Metrics 112
Design maintainability 113
Coupling metrics 114
Validation of quality metrics 115
Program quality metrics 116
Metrics maturity 117
Summary 118

***6. Metrics, Visualisations and Interactions for Reverse Engineering*** ***119***
Contents 120
Introduction 121
Metrics 122
Metrics and Measurements 123
Metrics for Reverse Engineering 124
Which Metrics to Collect (Definitions)? 125
Class size 126
Class Complexity 127
Hierarchy Layout 128
Method Size 129
Class Cohesion (i) 130
Class Cohesion (ii) 131
Class Coupling (I) 132
Class Coupling (Ii) 133
Metrics? Stepping Back 134
Visualisation 135
The Motivation: Why are we visualising stuff? 136
Visualisation: Possible Approaches 137
Example: Goose/ Graphlet 138
Example: Mermaid 139
Let's summarise... 140
Our Approach: CodeCrawler 141
The Idea: Visualising Metrics 142
CodeCrawler: Some Examples 143
System Complexity 144
Method Efficiency Correlation 145
Inheritance Classification 146
Service Class Detection 147
CodeCrawler's Logic 148
CodeCrawler: Pro And Contra 149
CodeCrawler: The Case Studies 150
Example: Visualisation of a very large system 151
Example: Flying Saucers 152
Conclusion & Possible Projects 153
Bibliography 154

***7. Lab session — CodeCrawler*** ***155***

***8. Object-Oriented Software Cost Estimation*** ***156***
Topics 157
Measurements & Estimates (1/2) 158
Measurements & Estimates (2/2) 159
A Measurement-Based Estimation Process (1/3) 160
A Measurement-Based Estimation Process (2/3) 161
A Measurement-Based Estimation Process(3/3) 162
Software Process Models (1/2) 163
Software Process Models (2/2) 164
Software Models (Meta Models) (1/4) 165
Software Models (Meta Models) (2/4) 166
Software Models (Meta Models) (3/4) 167
Software Models (Meta Models) (4/4) 168
Software Metrics (1/2) 169
Software Metrics (2/2) 170
Results of a Field Study (1/4) 171
Results of a Field Study (2/4) 172
Results of a Field Study (3/4) 173
Results of a Field Study (4/4) 174
An Example (1/5) 175
An Example (2/5) 176
An Example (3/5) 177
An Example (4/5) 178
An Example (5/5) 179
Future Work 180

***9. Metrics in OO Reengineering*** ***181***
Why Metrics in OO Reengineering? 182
Quantitative Quality Model 183
Process Attributes & External Attributes 184
Internal Product Attributes 185
"Define your own" Quality Model 186
Conclusion: Metrics for Quality Assessment 187
The KISS principle 188
Trend Analysis via Change Metrics 189
Conclusion: Metrics for Trend Analysis 190
Identifying Refactorings via Change Metrics 191
Split into Superclass / Merge with Superclass 192
Example: Inferring the Bridge Protocol 193
Split into Subclass / Merge with Subclass 194
Example: Adding new Functionality 195
Move to Superclass, Subclass or Sibling Class 196
Example: Introducing Layers 197
Split Method / Factor Common Functionality 198
Example: Creation of Template Method 199

Conclusion: Identifying Refactorings 200
Conclusion 201
Questions 202
***10. Tool Integration 203***
Why Integrate Tools? 204
Which Tools to Integrate? 205
Tool Integration Issues 206
Basic Tool Architecture 207
Help Yourself - Parser 208
Help Yourself - File Formats 209
Help Yourself - API 210
Help Yourself - Execution Trace 211
API Example - Java 212
API Example - SNiFF+ 213
API Example - Rational/Rose 214
Exchange Standards 215
Exchange Standards - Reference Format 216
Exchange Standards - Openness 217
Meta Models 218
CDIF sample (propriety syntax) 219
MOF Sample (XML syntax) 220
CORBA Interface for MOF 221
UML shortcomings 222
Conclusion 223
Questions 224
***11. Refactoring 225***
What is Refactoring? 226
Why Refactoring? 227
Iterative Development Life-cycle 228
Example: Rename Class 229
Tool Support for Refactoring 230
Case Study: Internet Banking 231
Prototype Design: Class Diagram 232
Prototype Design: Contracts 233
Prototype Implementation 234
Prototype Consolidation 235
Expansion 236
Expanded Design: Class Diagram 237
Expanded Implementation 239
Consolidation: Problem Detection 240
Consolidation: Refactored Class Diagram 241
Refactoring Sequence (1/5) 242
Refactoring Sequence (2/5) 243
Refactoring Sequence (3/5) 244
Refactoring Sequence (4/5) 245
Refactoring Sequence (5/5) 246
Conclusion (1/2) 247
Conclusion: Culture shock (2/2) 248
Projects and More Information 249
***12. Using Dynamic Information for Reverse Engineering 250***
Outline 251
Why Dynamic Information? 252
Why Dynamic Information (cont'd)? 253
What is Dynamic Information? 254
Static vs. Dynamic Information 255
Problems with using Dynamic Information 256
Roadmap 257
Frequency Spectrum 258
FSA: low vs. high frequencies 259
FSA: related frequencies 260
FSA: specific frequencies 261
Dynamic Differencing 262
Summary of Spectrum Techniques 263
Roadmap 264
Visualization 265
Animated Summaries 266
Animated Summaries Example: Jinsight 267
Information Mural 268
Information Mural Example: ISVis 269
RoadMap 270
A Step Back: Design Recovery 271
Design Recovery through Visualization 272
Selective Instrumentation & Filtering 273
Clustering 274
Recognizing Patterns: example of Jinsight 275
Summary of Visualization for Design Recovery 276
RoadMap 277
Gaudi: overview of Approach 278
Gaudi: Implementation 279
Gaudi: Formulating Derived Relations 283
Gaudi: Using Derived Relations for Querying 284
Gaudi: Simple vs. Composed Views 285
Gaudi: Instance Level View 286
Gaudi Summary 287
Instrumentation 288
References 290

# 1. Object-Oriented Software Reengineering

**Lecturers:** Dr. S. Demeyer, Dr. S. Ducasse, Prof. Dr. O. Nierstrasz
with Michele Lanza, Tamar Richner, Matthias Rieger, Sander Tichelaar

**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. | 29/10 | Introduction |
| 2. | 05/11 | Duplicated code |
| 3. | 12/11 | Lab session — Duploc |
| 4. | 19/11 | UML extraction |
| 5. | 26/11 | Software metrics |
| 6. | 03/12 | Visualizing software metrics |
| 7. | 10/12 | Lab session — Codecrawler |
| 8. | 17/12 | Metrics in industry |
| 9. | 14/01 | Metrics and reengineering |
| 10. | 21/01 | Code repositories |
| 11. | 28/01 | Refactoring |
| 12. | 04/02 | Lab session — Refactoring browser |
| 13. | 04/02 | Exploiting run-time information |

# 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.

# 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

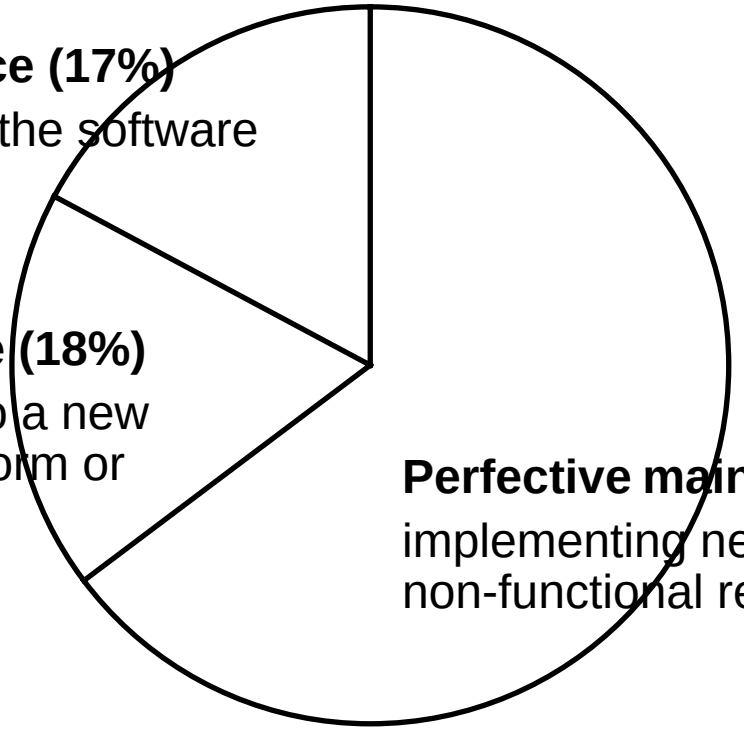
so, further evolution and development may be prohibitively expensive

# 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

# 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

# What about OO?

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

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

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

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

**The claim:**

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

# Definitions

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

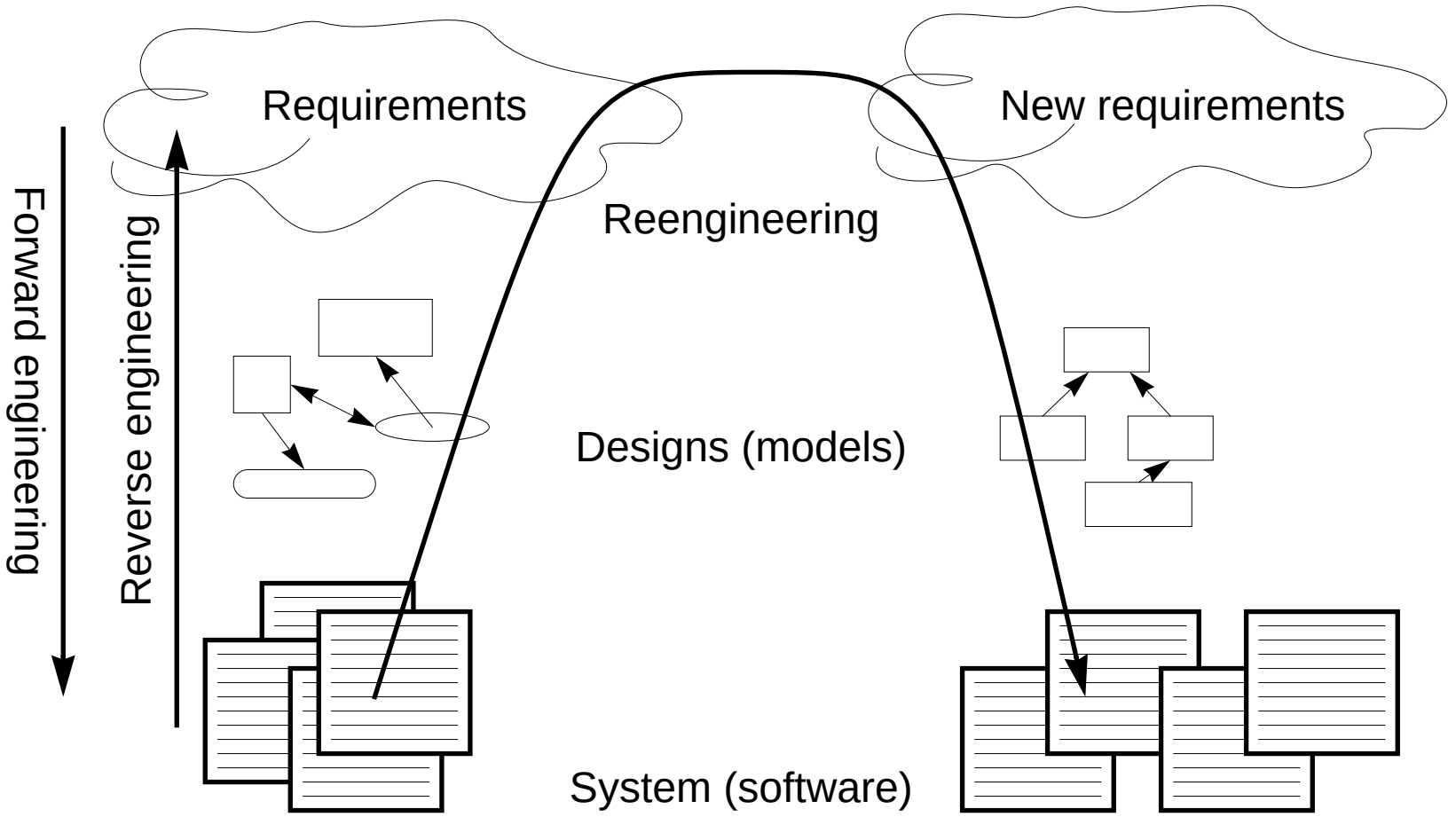
"Reverse Engineering is the process of analyzing a subject system to

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

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

— Chikofsky and Cross [in Arnold, 1993]

# Reverse and Reengineering



# Goals of Reverse Engineering

**Cope with complexity**

- need techniques to understand large, complex systems

**Generate alternative views**

- automatically generate different ways to view systems

**Recover lost information**

- extract what changes have been made and why

**Detect side effects**

- help understand ramifications of changes

**Synthesize higher abstractions**

- identify latent abstractions in software

**Facilitate reuse**

- detect candidate reusable artifacts and components

— Chikofsky and Cross [in Arnold, 1993]

# Reverse Engineering Techniques

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

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

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

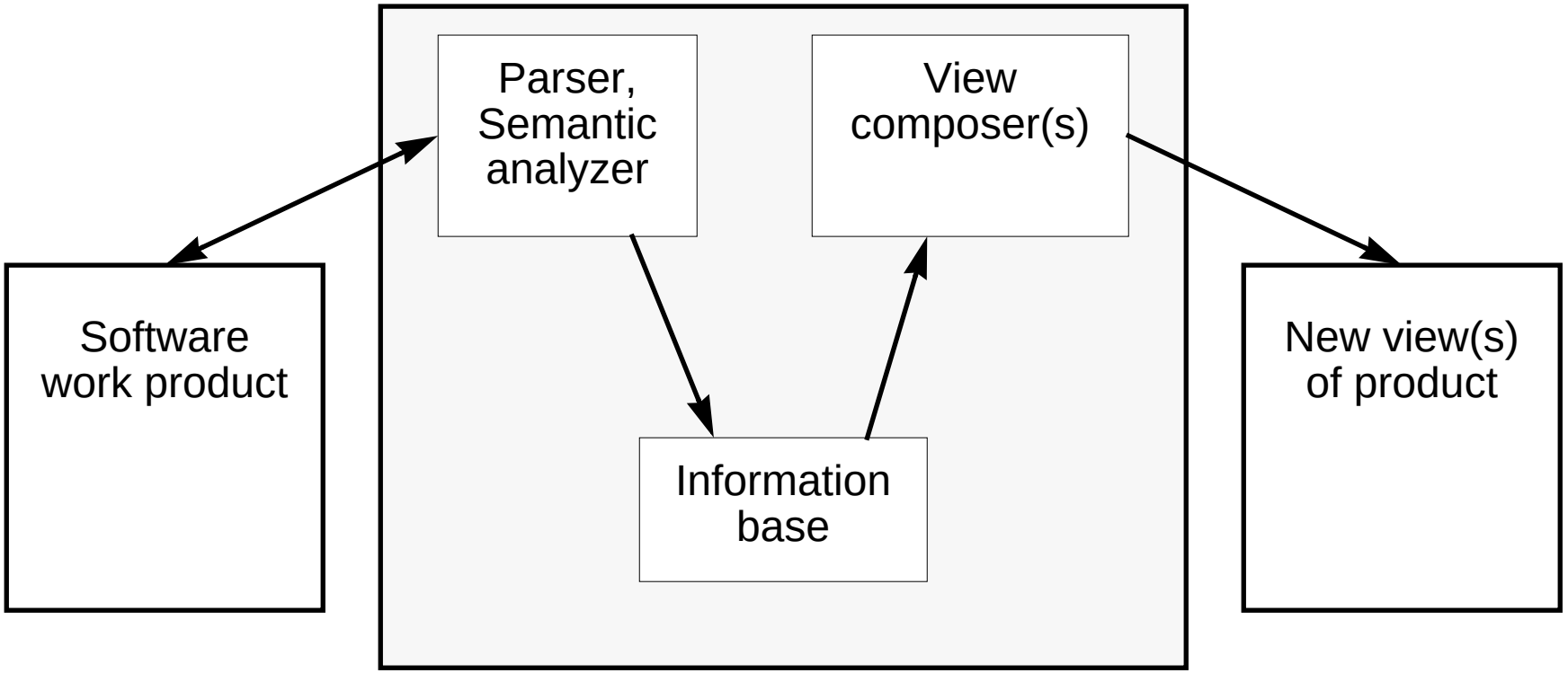
- explicit type-casting, C++ “friends” ...

**Class misuse**

- lack of cohesion — classes as namespaces

# Tools Architectures

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



# 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

# 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*

# Overview

- **What is Code Duplication?**
  - How Much Code Is Copied
  - What Do We Call Copied Code
- **The Life and Times of Copied Code**
  - How Code Gets Copied
  - Why Code Gets Copied
  - What Problems Stem From Copied Code
- **We have to detect Duplicated Code**
  - Simple Detection Approach
  - Detection Using Parameterized Matches
  - Detection Using Abstract Syntax Trees
- **Refactoring Duplicated Code**
- **Visualizing Duplicated Code**

# Code is Copied

- Small Example from the Mozilla Distribution (Milestone 9)

```
[432]  NS_IMETHODIMP
[433]  LocationImpl::GetPathname(nsString
[434]  {
[435]    nsAutoString href;
[436]    nsIURI *url;
[437]    nsresult result = NS_OK;
[438]
[439]    result = GetHref(href);
[440]    if (NS_OK == result) {
[441]  #ifndef NECKO
[442]      result = NS_NewURL(&url, href);
[443]  #else
[444]      result = NS_NewURI(&url, href);
[445]  #endif // NECKO
[446]      if (NS_OK == result) {
[447]  #ifdef NECKO
[448]        char* file;
[449]        result = url->GetPath(&file);
[450]  #else
[451]        const char* file;
[452]        result = url->GetFile(&file);
[453]  #endif
[454]        if (result == NS_OK) {
[455]          aPathname.SetString(file);
[456]  #ifdef NECKO
[457]          nsCRT::free(file);
[458]  #endif
```

```
[467]  NS_IMETHODIMP
[468]  LocationImpl::SetPathname(const nsString
[469]  {
[470]    nsAutoString href;
[471]    nsIURI *url;
[472]    nsresult result = NS_OK;
[473]
[474]    result = GetHref(href);
[475]    if (NS_OK == result) {
[476]  #ifndef NECKO
[477]      result = NS_NewURL(&url, href);
[478]  #else
[479]      result = NS_NewURI(&url, href);
[480]  #endif // NECKO
[481]      if (NS_OK == result) {
[482]        char *buf = aPathname.ToNewCString();
[483]  #ifdef NECKO
[484]        url->SetPath(buf);
[485]  #else
[486]        url->SetFile(buf);
[487]  #endif
[488]        SetURL(url);
[489]        delete[] buf;
[490]        NS_RELEASE(url);
[491]      }
[492]    }
[493]
```

```
[497]  NS_IMETHODIMP
[498]  LocationImpl::GetPort(nsString& aPo
[499]  {
[500]    nsAutoString href;
[501]    nsIURI *url;
[502]    nsresult result = NS_OK;
[503]
[504]    result = GetHref(href);
[505]    if (NS_OK == result) {
[506]  #ifndef NECKO
[507]      result = NS_NewURL(&url, href);
[508]  #else
[509]      result = NS_NewURI(&url, href);
[510]  #endif // NECKO
[511]      if (NS_OK == result) {
[512]        aPort.SetLength(0);
[513]  #ifdef NECKO
[514]        PRInt32 port;
[515]        (void)url->GetPort(&port);
[516]  #else
[517]        PRUint32 port;
[518]        (void)url->GetHostPort(&port);
[519]  #endif
[520]        if (-1 != port) {
[521]          aPort.Append(port, 10);
[522]        }
[523]        NS_RELEASE(url);
```

Extract from `/dom/src/base/nsLocation.cpp`

# How Much Code is Duplicated?

- Usual estimates: 8 to 10% in normal industrial code
- Our Research:

| *Case Study* | *Language* | *LOC* | *Duplication %* |
|---|---|---|---|
| *gcc* | *C* | *460'000* | *8.7% (5.6%)* |
| *Database Server* | *Smalltalk* | *245'000* | *36.4% (23.3%)* |
| *Payroll* | *Cobol* | *40'000* | *59.3% (25.4%)* |
| *Message Board* | *Python* | *6500* | *29.4% (17.4%)* |

# What Is Considered To Be Copied Code?

Duplicated Code = Source code segments that are found in different 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.

---

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

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

could be abstracted to a new function

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

# How Code Gets Copied

Programming Truism: I copy, you copy, we all copy!

**A possible scenario from Software Maintenance:**

14. New functionality similar to the one provided by a sub-component C is needed.
15. C could be extended to assimilate the new functionality, but ...
16. ... this requires lengthy and difficult analysis of C ....
17. ... and significant regression testing to ensure functioning of C in old contexts.

**Add time pressure.**

18. A copy of C is made.
19. The component is tailored to provide the new functionality
20. Code that is not understood and therefore cannot be deleted remains as red herring or even dead code.

# Why Code Gets Copied

Causes apart from **time pressure** that lead to copy&paste programming:

- **Laziness**
  Producing reusable software takes a lot of effort.
- **Efficiency Considerations**
  Procedure Calls can cost too much.
- **Code Ownership**
  I cannot adapt my neighbours code, so I must copy it.
- **Maintaining Versions For Multiple Platforms**
  Separate files instead of a lot of `#ifdef`'s.
- **Programmer Productivity Evaluation Methods**
  It is easy way to boost the number of lines of code produced by copying them.

# What Problems Stem From Copied Code?

**General negative effect:**

- Code bloat

**Negative effects on Software Maintenance:**

- Copied Defects
- Changes take double, triple, quadruple, ... work
- Red herrings and dead code
  - ☞ add to the cognitive load of future maintainers
- Copying as additional source of defects
  - ☞ Errors in the systematic renaming produce unintended aliasing

**Metaphorically speaking:**

Software Aging, “hardening of the arteries”, “Software Entropy” increases

- ☞ even small design changes become very difficult to effect

# Code Duplication: Problem Statement

Frequent consolidation to keep a system flexible and easier to expand:

- Reorganize system components
- Refactor functionality
- Rationalize interfaces

and

- Remove Duplicated Code

**Nontrivial problem:**

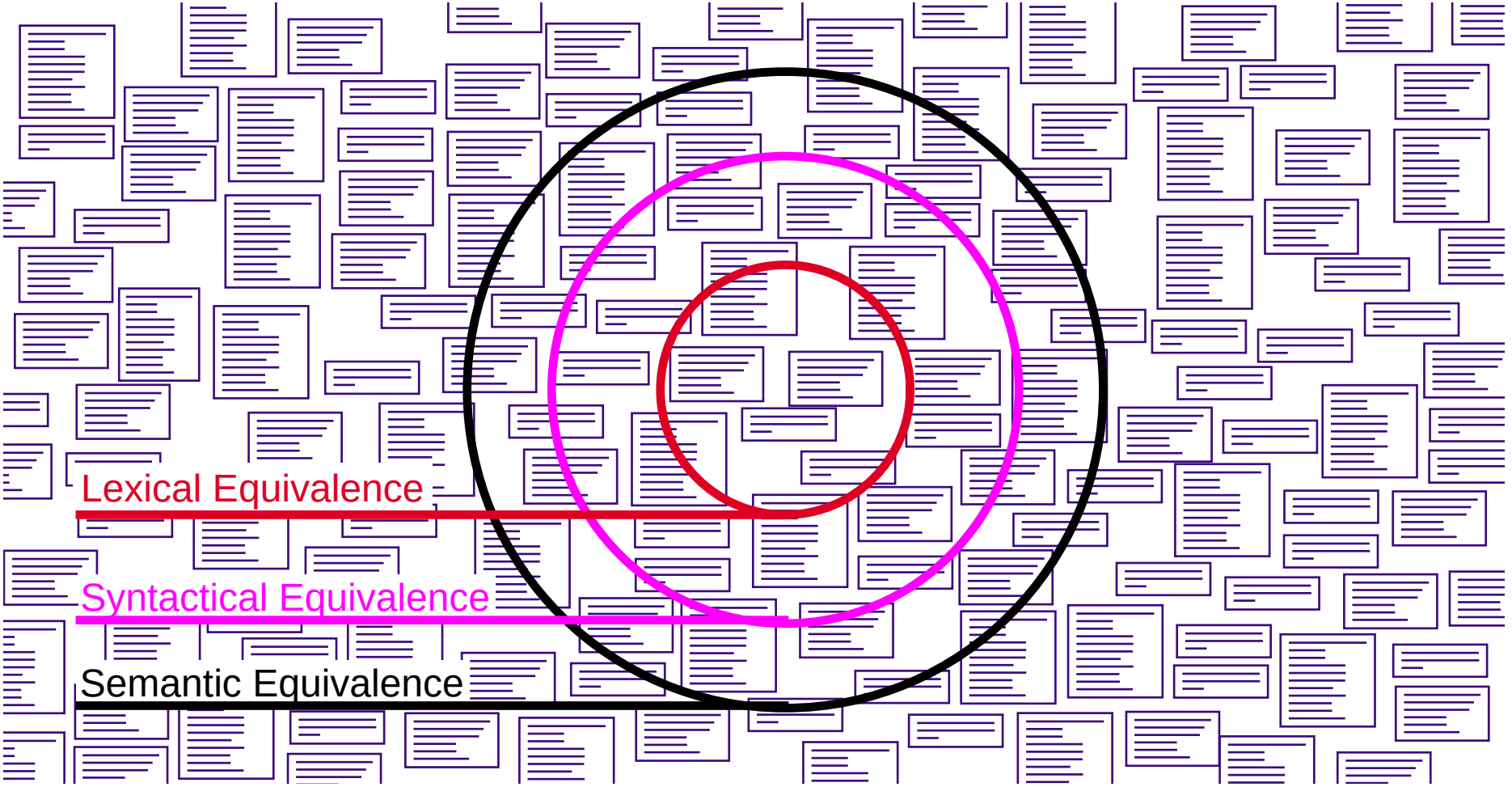
No a priori knowledge about which code has been copied

- ☞ Detect Duplicated Code

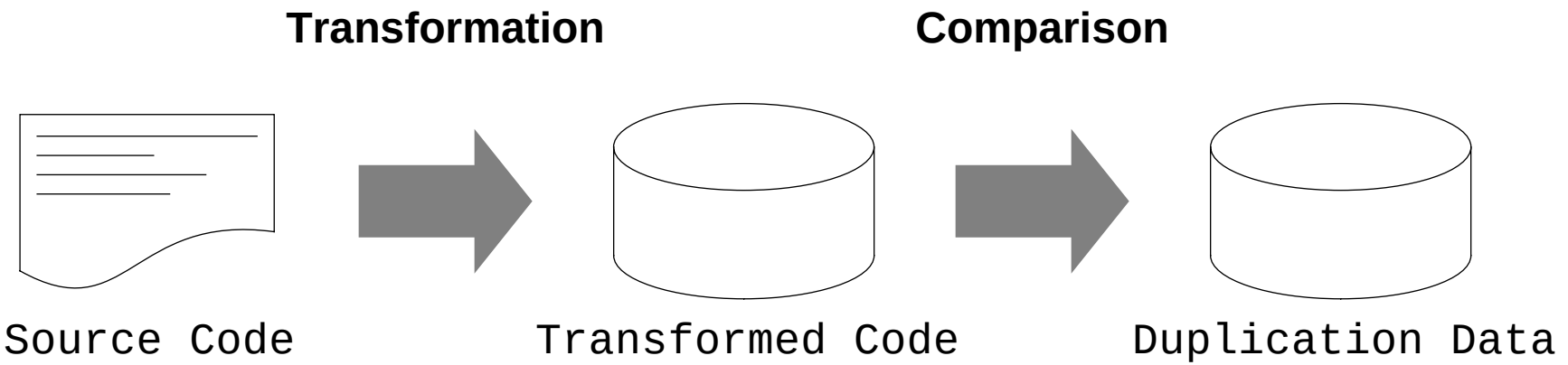
and for that we need tools...

# Code Duplication Detection

How do we find all clone pairs among all possible pairs of segments?



# General Schema of Detection Process




| Author | Level | Transformed Code | Comparison Technique |
|---|---|---|---|
| Johnson, 1994 | Lexical | Substrings | String-Matching |
| Rieger et. al., 1999 | Lexical | Normalized Strings | String-Matching |
| Baker, 1992 | Syntactical | Parameterized Strings | String-Matching |
| Mayrand et. al., 1996 | Syntactical | Metric Tuples | Discrete comparison |
| Kontogiannis, 1996 | Syntactical | Metric Tuples | Euclidean distance |
| Baxter et. al., 1998 | Syntactical | AST | Tree-Matching |

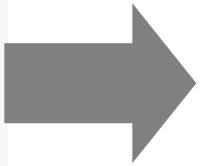
# Simple Detection Approach I

**Assumption:** Code segments are just copied and changed at a few places

**Code Transformation Step**

- remove white space
- remove comments
- remove lines that contain uninteresting code elements (e.g. just '`else`' or '`}`')

```
...
// assign same fastid as container
fastid = NULL;
const char* fidptr = getFastid();
if(fidptr != NULL) {
     int l = strlen(fidptr);
     fastid = new char[l+1];
     char *tmp = (char*) fastid;
     for (int i =0;i<l;i++)
        tmp[i] = fidptr[i];
     tmp[l] = '\0';
}
...
```



```
...
fastid=NULL;
constchar*fidptr=getFastid();
if(fidptr!=NULL){
intl=strlen(fidptr);
fastid=newchar[l+1];
char*tmp=(char*)fastid;
for(inti=0;i<l;i++)
tmp[i]=fidptr[i];
tmp[l]='\0';
...
```

# Simple Detection Approach II

**Code Comparison Step**

- Line based comparison (Assumption: Layout not changed during copying)
- Compare each line with each other line.
  - Reduce search space by hashing:
    1. Preprocessing: Compute the hash value for each line
    2. Actual Comparison: Compare all lines in the same hash bucket
- Collect consecutive matching lines into match sequences

**Evaluation of the Approach**

| | |
|---|---|
| ***Advantages*** | language independent |
| ***Disadvantages*** | misses copies with (small) changes on every line |

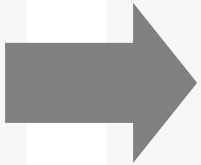
# Detection Using Parameterized Matching I

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

**Code Transformation Step**

- Lexical analysis to generate a token stream
- Replace the identifiers of tokens that represent variables by generic names

```
...
fastid = NULL;
const char* fidptr = getFastid();
if(fidptr != NULL) {
     int l = strlen(fidptr);
     fastid = new char[l+1];
     char *tmp = (char*) fastid;
     for (int i =0;i<l;i++)
        tmp[i] = fidptr[i];
     tmp[l] = '\0';
}
...
```



```
...
P0 = NULL;
const char* P1= getFastid();
if(P1 != NULL) {
     int P3 = strlen(P1);
     P0= new char[P3+1];
     char *P4 = (char*) P0;
     for (int P5=0;P5<P3;P5++)
        P4[P5] = P1[P5];
     P4[P3] = '\0';
}
...
```

- Token stream regarded as one large string

# Detection Using Parameterized Matching II

**Code Comparison Step**

- Find all maximal matching substrings (by using Suffix Tries)

```
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;
```

**Evaluation of the Approach**

| | |
|---|---|
| *Advantages* | finds large range of duplication<br>can generate code that unifies parameterized matches |
| *Disadvantages* | requires lexical analysis, thus language dependent<br>algorithmically complicated |

# Detection using Abstract Syntax Trees I

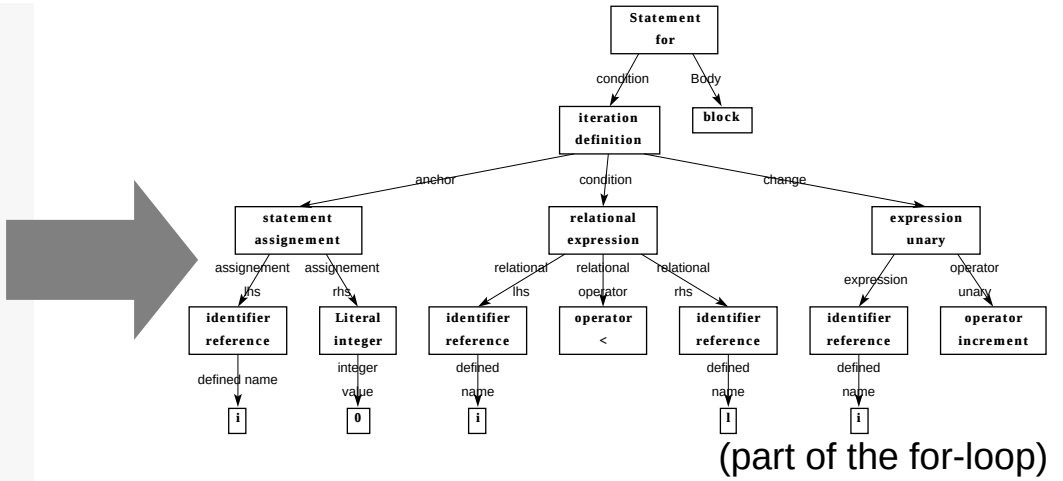
**Idea:** Abstract view on the code is not disturbed by accidental properties like layout, parameter names or even operand order.

ASTs are handy format for a large number of computations

## Code Transformation Step

- Parse source code into an abstract syntax tree

```
...
fastid = NULL;
const char* fidptr = getFastid();
if(fidptr != NULL) {
     int l = strlen(fidptr);
     fastid = new char[l+1];
     char *tmp = (char*) fastid;
     for (int i =0;i<l;i++)
        tmp[i] = fidptr[i];
     tmp[l] = '\0';
}
...
```




(part of the for-loop)

- Calculate tuples of metric values for specific subtrees (e.g. functions)

# Detection using Abstract Syntax Trees II

**Code Comparison Step**

- Tree matching

or

- Comparison of metrics tuples (Euclidean distance)

**Evaluation of the Approach**

| | |
|---|---|
| *Advantages* | - fine grained similarity analysis is possible<br>☞ approach finds largest range of duplication<br>- code generation can be done directly from the AST |
| *Disadvantages* | - very language dependent<br>- scalability is a problem since ASTs require la lot of memory |

# Refactoring Duplicated Code I

**The Mantra of the Ideal Programmer**

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

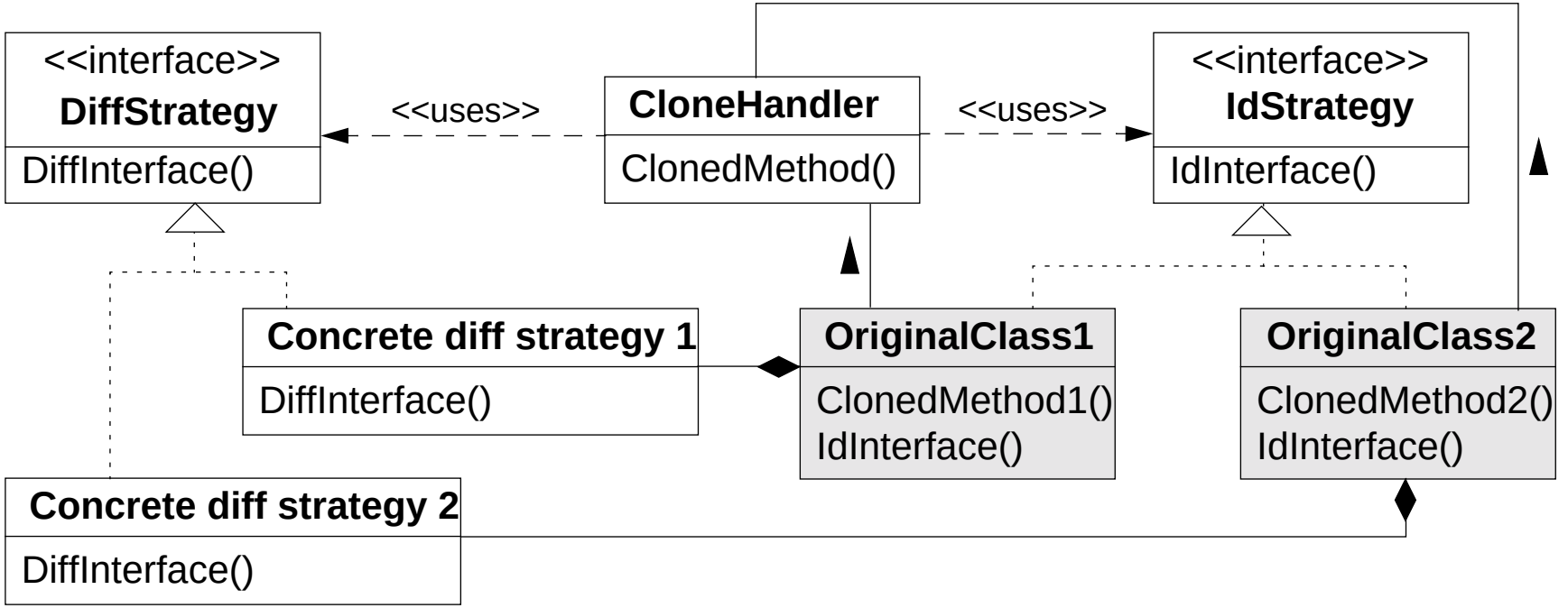
**Refactoring in a Class Hierarchy**

- 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)

# Refactoring Duplicated Code II

**Automatic refactoring of Java clones using the strategy design pattern**

- `CloneHandler` class captures the duplicated functionality
- `IdStrategy` interface connects the CloneHandler to the context from which it is called
- `DiffStrategy` interface holds the variant functionality that occurs in the different clones

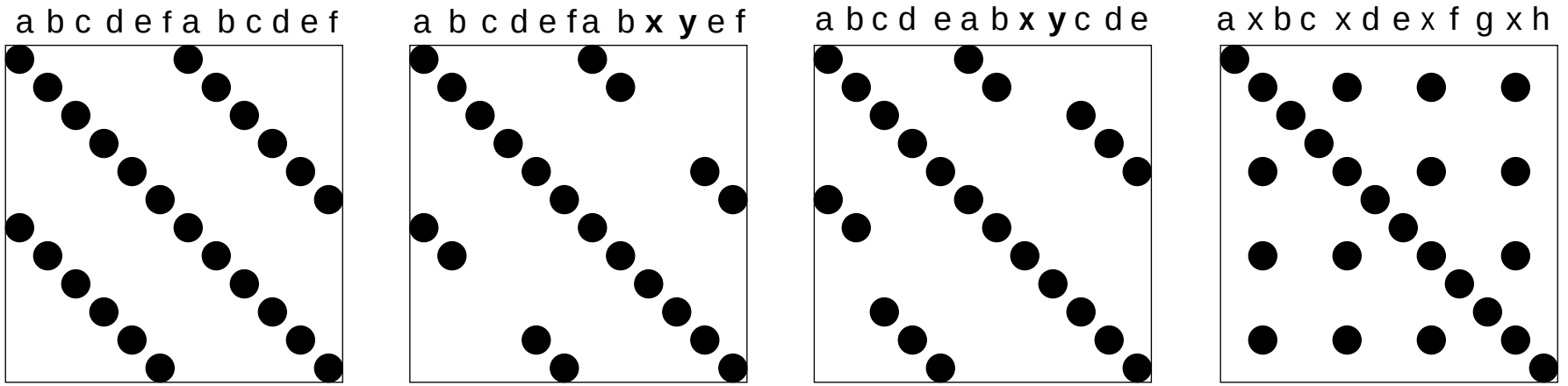


# Visualization of Duplicated Code

**Scatterplots-Technique from DNA Analysis**

- Code is put on vertical as well as horizontal axis
- A match between two elements is represented as a dot in the matrix

**Interpretation of Dot Configurations**




Exact Copies

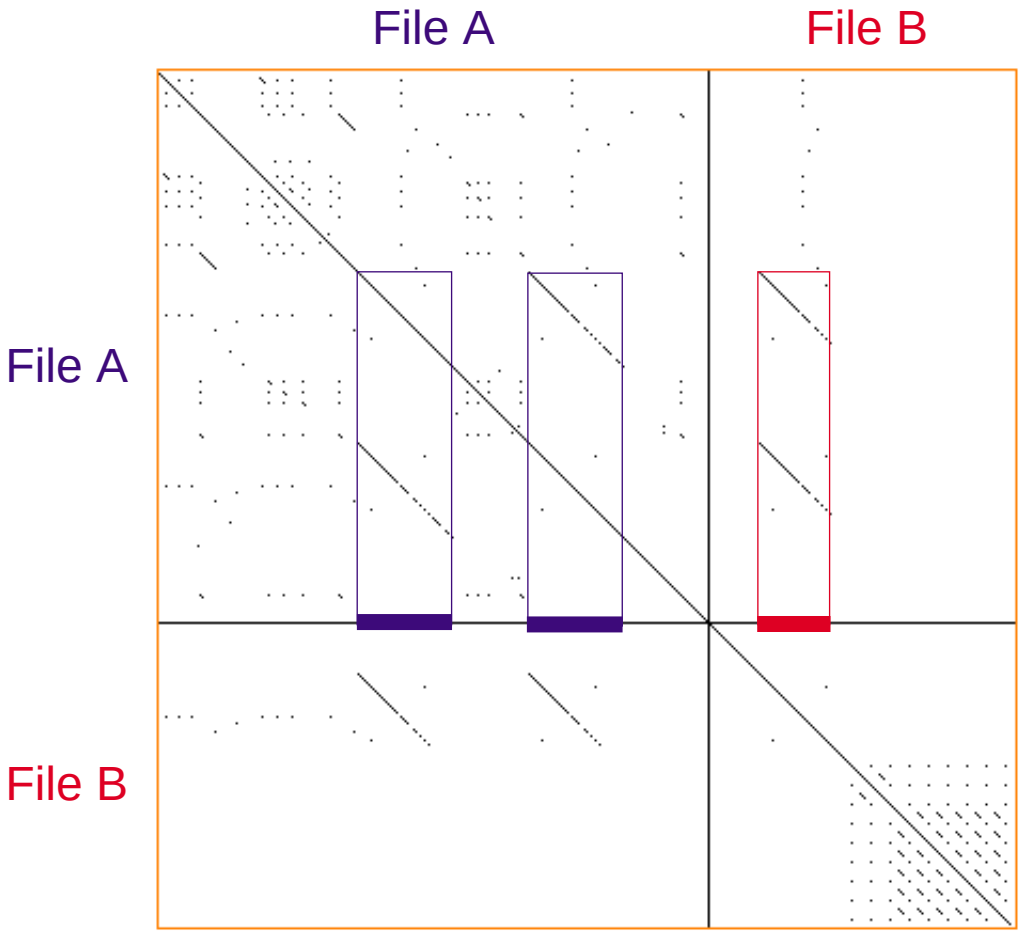
Copies with Variations

Inserts/Deletes

Repetitive Code Elements

- Visualization allows intuitive insights into the duplication situation
- Easy source code access is important

# Visualization of Copied Code Sequences



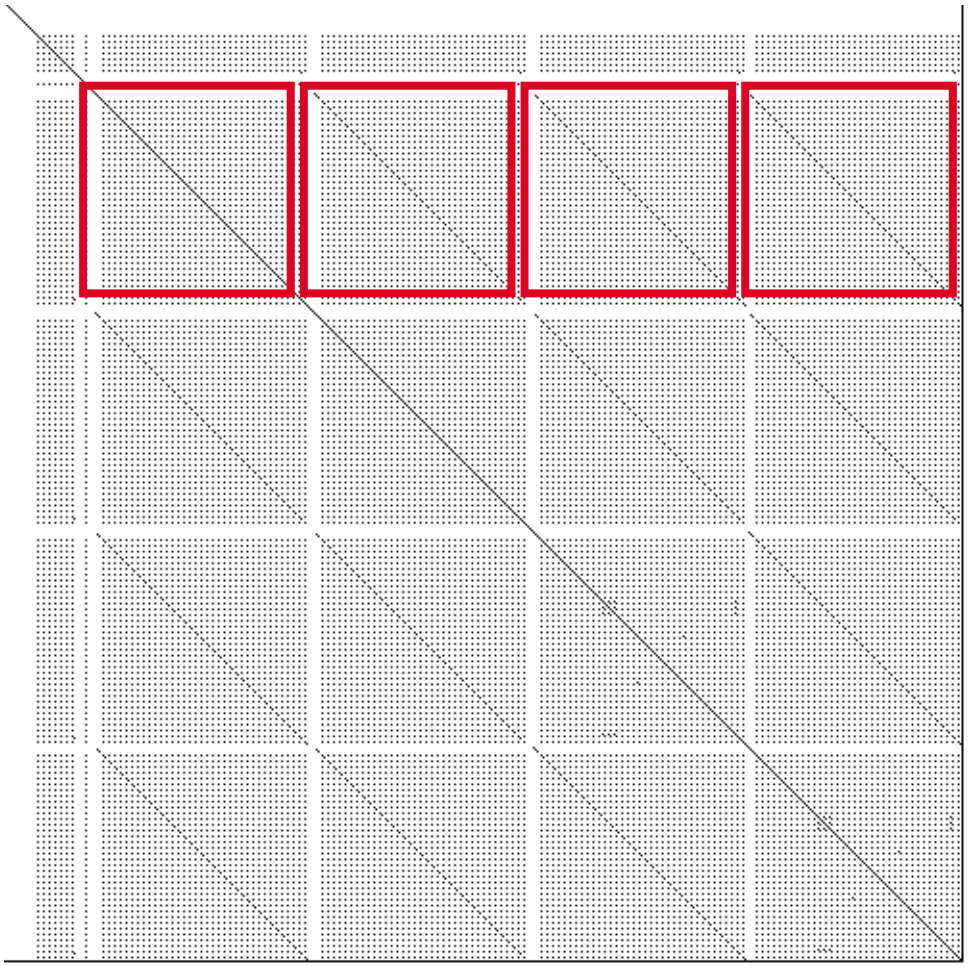

**Detected Problem:**

File A contains two copies of a piece of code.

File B contains another copy of this code.

All Examples on this an the following slides are from an industrial case study (1 Mio LOC C++ System)

# Visualization of Repetitive Structures



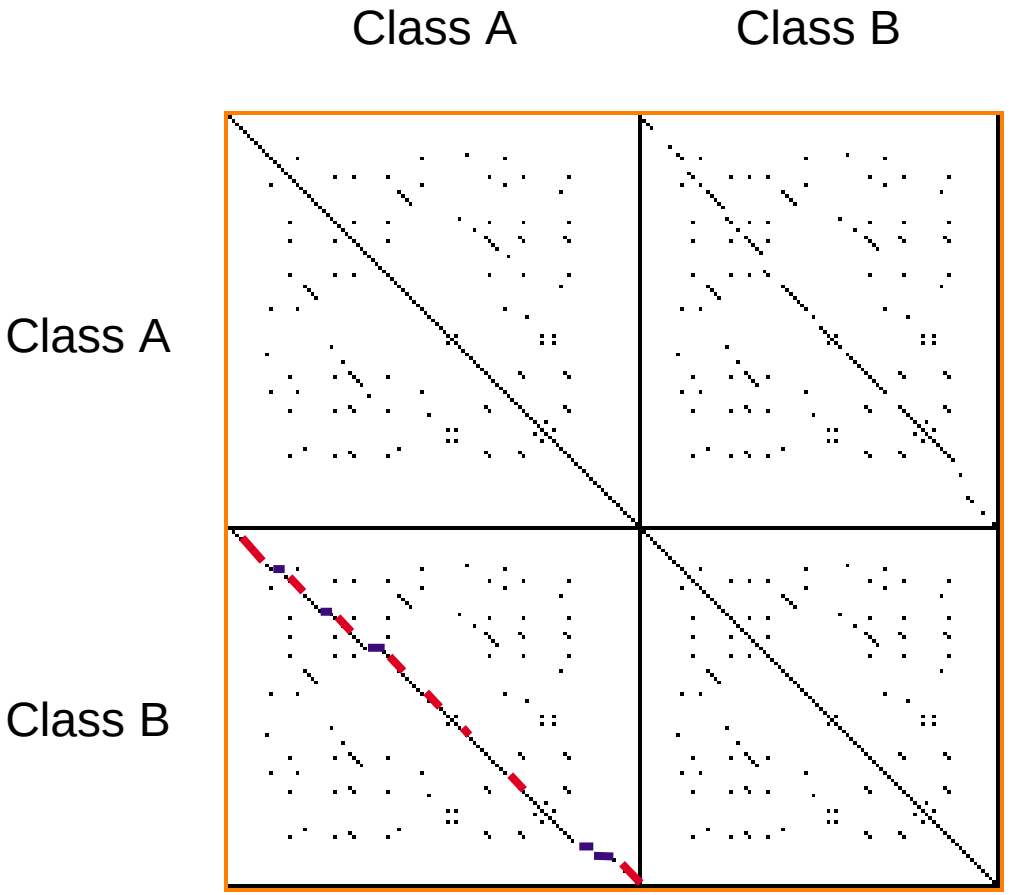
**Detected Problem:**

4 Object factory clones: a switch statement over a type variable is used to call individual construction code.

**Possible Solution:**

Strategy Method

# Visualization of Cloned Classes

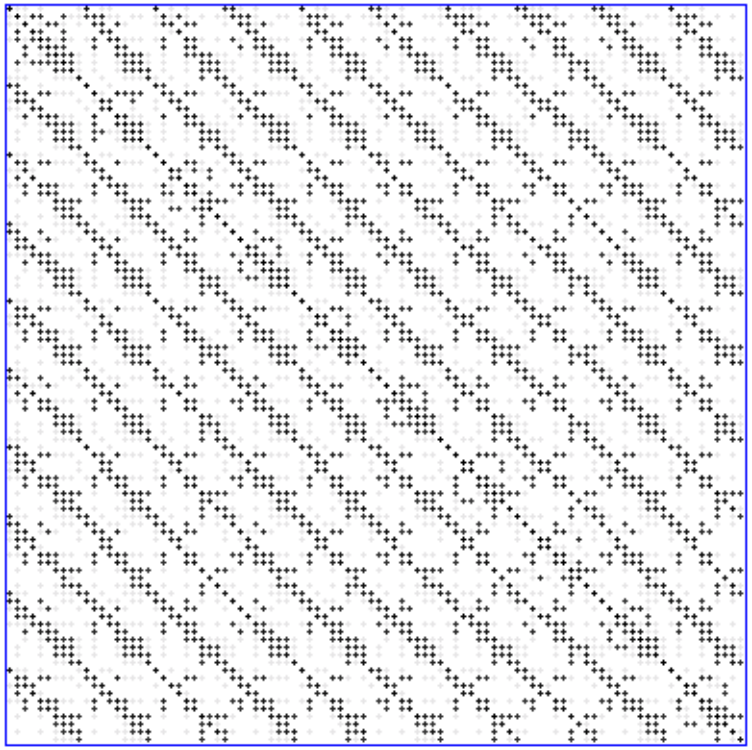



**Detected Problem:**
Class A is an edited copy of class B:
Editing & Insertion

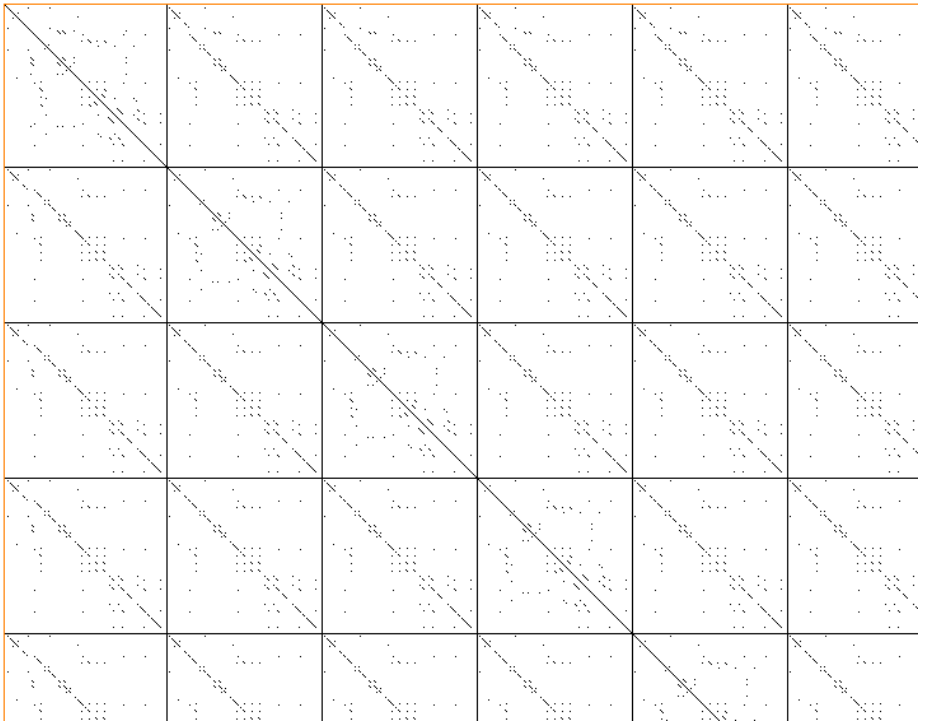
Typical case of code scavenging.

# Visualization of Clone Families

## Overview



## Detail



20 Classes implementing lists for different data types

# Summary

- Duplicated code is a real problem
- Duplicated Code makes a system progressively harder to change
- To Detect Duplicated Code is a hard problem
  - ☞ tool support is needed
- Refactoring duplicated code can be done automatically
- Code Duplication Detection and Removal is an active research area

# References

M. Balazinska et. al. Partial Redesign of Java software Systems Based on Clone Analysis, Proceedings Sixth Working Conference on Reverse Engineering, pages 326-336, IEEE Computer Society, 1999.

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 WCRE, pages 44-54, IEEE Computer Society, 1997.

# 3. Lab session — Duploc

# 4. Design Extraction

War story:

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

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

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

# Goals

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

# Outline

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
- Interpreting Uml
- Language Specific Issues
- Tracks For Extraction
- Extracting Intention: Design Pattern
- Extraction For The Reuser
- Extraction of Interaction
- Conclusion

# Why Design Extraction is needed?

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

Why UML?

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

# UML (Unified Modelling Language)

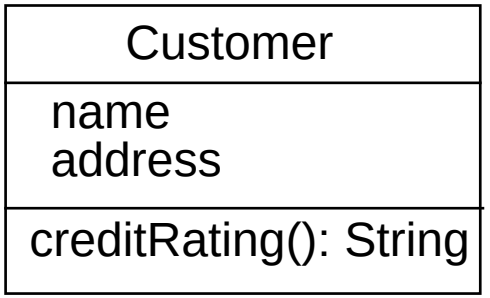
What is the Unified Modelling Language?

- Successor of OOAD&D methods of late 80 & early 90
- Unifies Booch, Rumbaugh (OMT) and Jacobson [Booc98a] [Rumb99a]
- Currently standardized by OMG

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

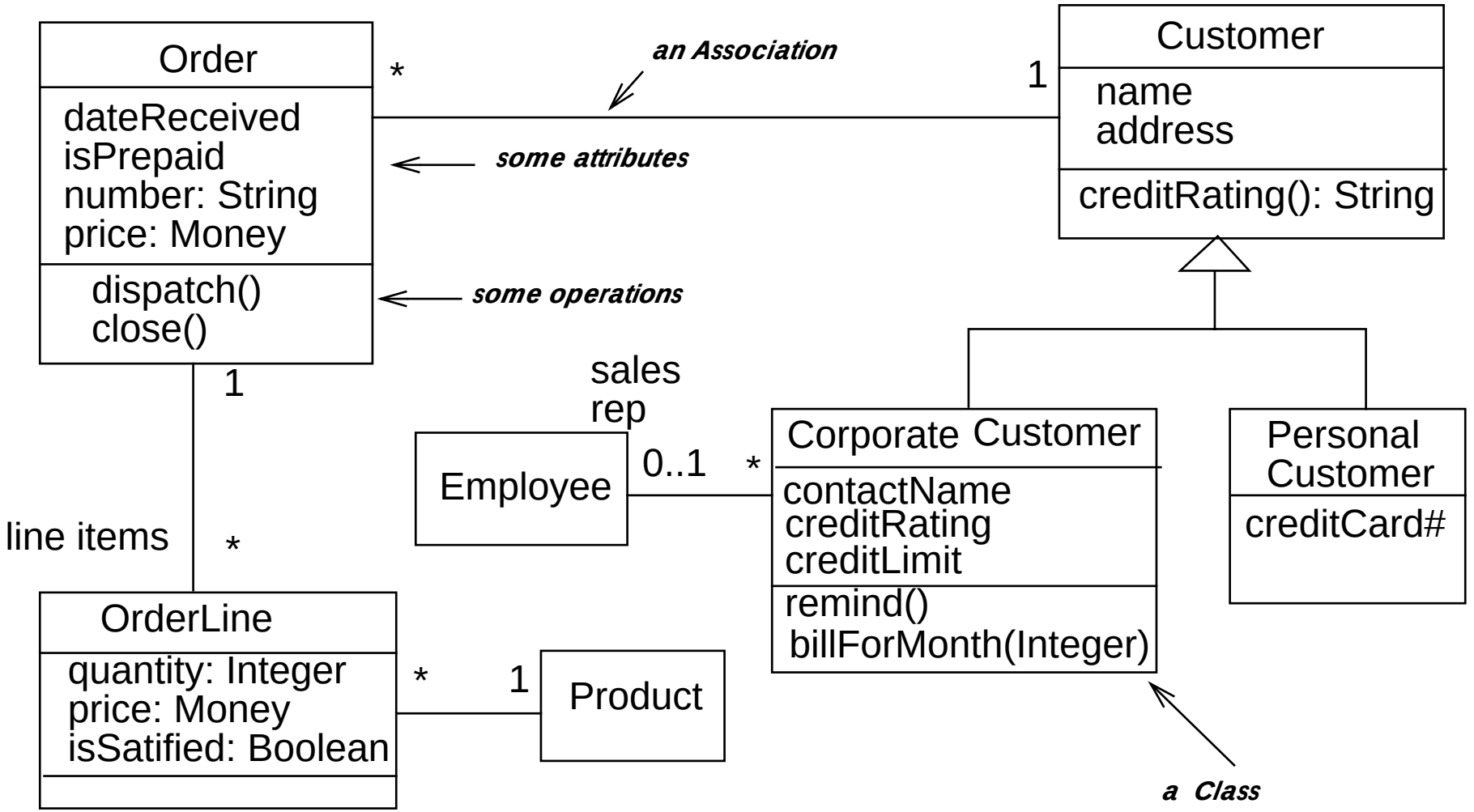
- UML defines
  - a notation (the syntax of the modelling language)

    Ex:




  - a meta-model = a model to define the “semantics” of a model (what is well-formed), defines in itself but weak!

# The Little Static UML



# Road Map

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
  - Experimenting With Extraction
- Interpreting Uml
- Language Specific Issues
- Tracks For Extraction
- Extracting Intention: Design Pattern
- Extraction For The Reuser
- Extraction of Interaction
- Conclusion

# Let us practice!

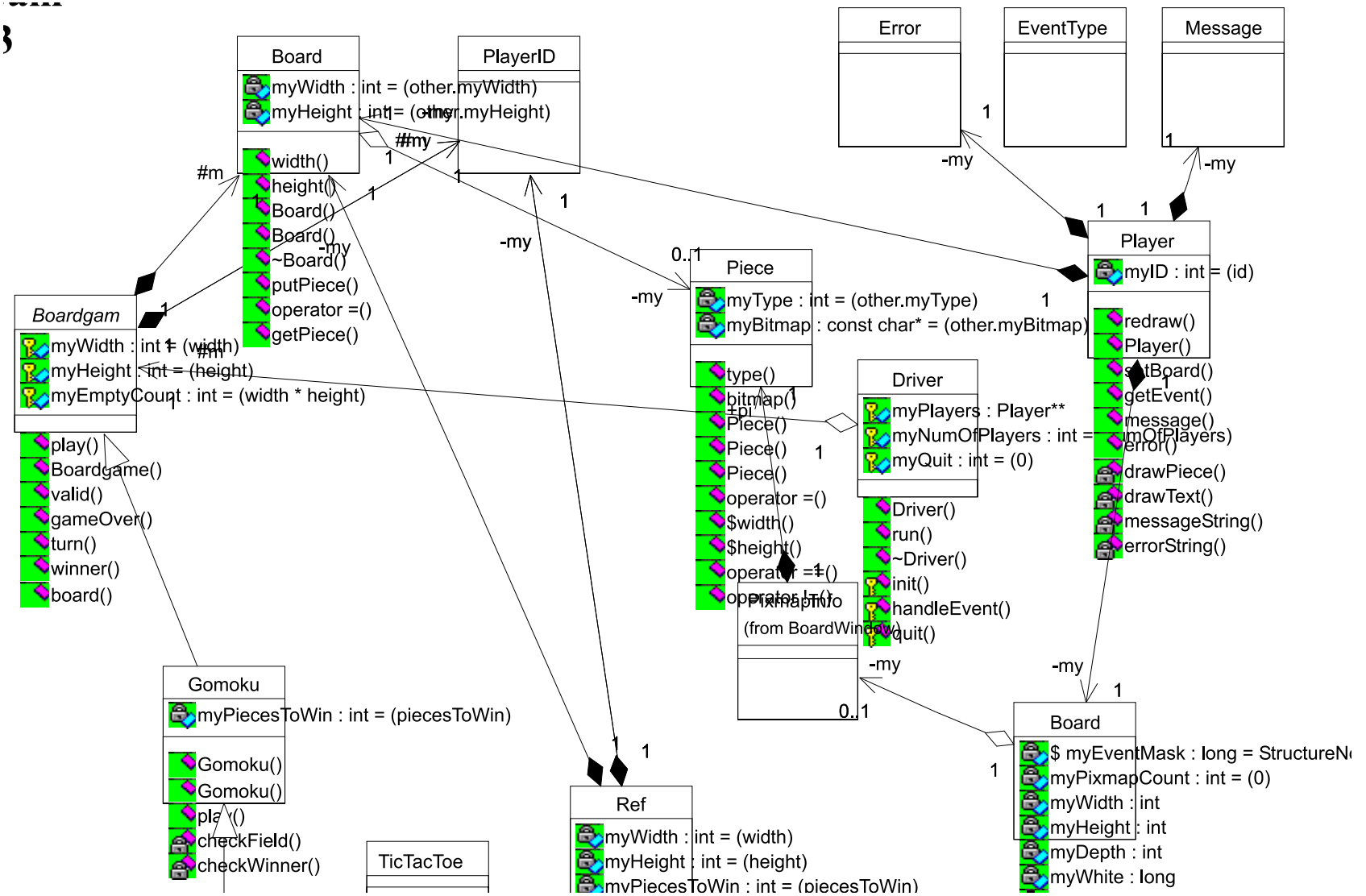
A small example in C++: A Tic-Tac-Toe Game!

You will do it now.....

But:

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

# A First View


Board
myWidth : int = (other.myWidth)
myHeight : int = (other.myHeight)
width()
height()
Board()
Board()
~Board()
putPiece()
operator =()
getPiece()
PlayerID
Error
EventType
Message
Boardgam
myWidth : int = (width)
myHeight : int = (height)
myEmptyCount : int = (width * height)
play()
Boardgame()
valid()
gameOver()
turn()
winner()
board()
Piece
myType : int = (other.myType)
myBitmap : const char* = (other.myBitmap)
type()
bitmap()
Piece()
Piece()
Piece()
operator =()
$width()
$height()
operator ==()
operator !=()
Player
myID : int = (id)
redraw()
Player()
getEvent()
message()
error()
drawPiece()
drawText()
messageString()
errorString()
Driver
myPlayers : Player**
myNumOfPlayers : int =
myQuit : int = (0)
Driver()
run()
~Driver()
init()
handleEvent()
quit()
PixmapInfo
(from BoardWindow)
Gomoku
myPiecesToWin : int = (piecesToWin)
Gomoku()
Gomoku()
play()
checkField()
checkWinner()
TicTacToe
Ref
myWidth : int = (width)
myHeight : int = (height)
myPiecesToWin : int = (piecesToWin)
Board
$ myEventMask : long = StructureN
myPixmapCount : int = (0)
myWidth : int
myHeight : int
myDepth : int
myWhite : long
#m
#my
-my
1
0..1

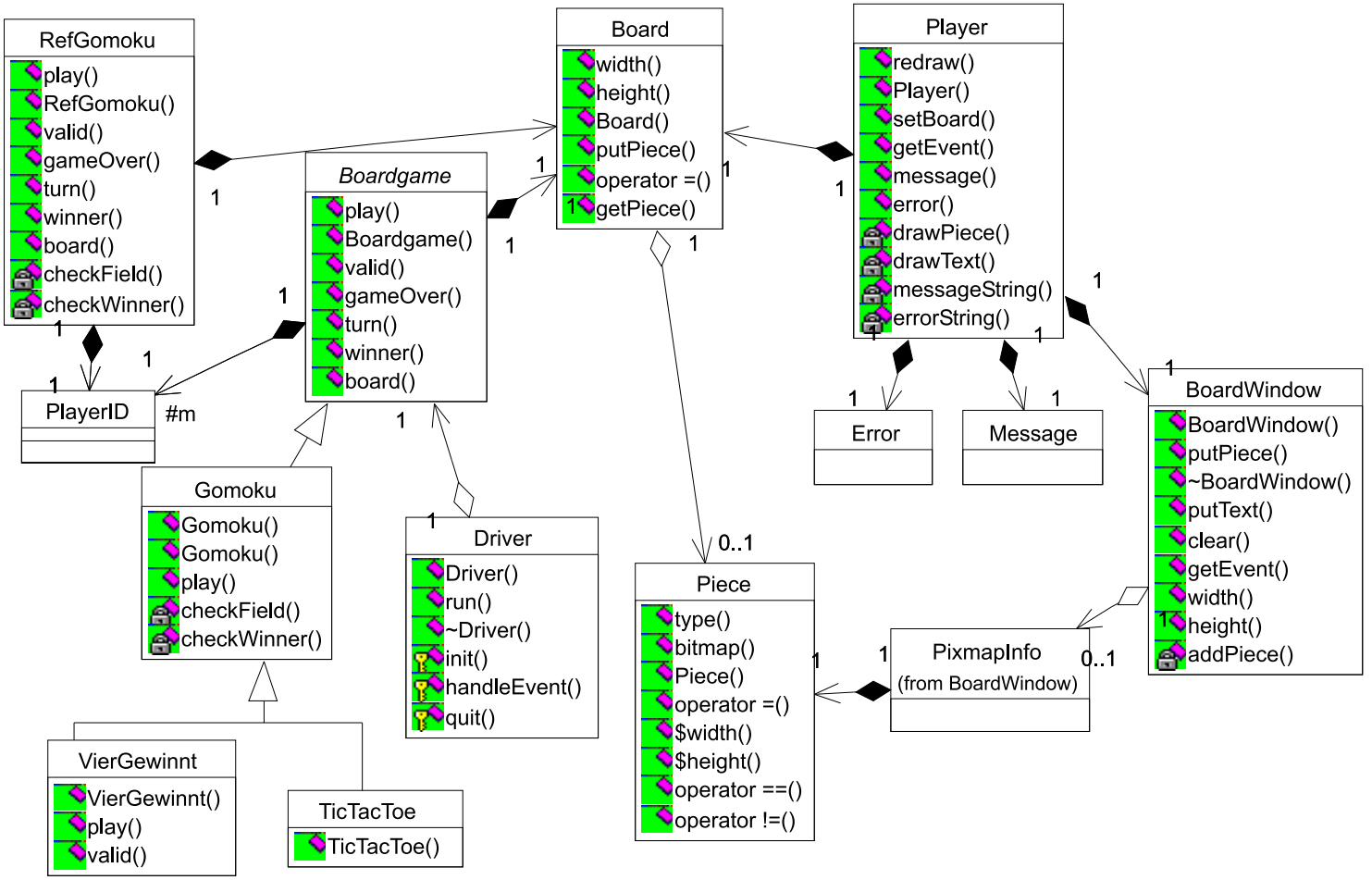
# Evaluation

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



# Road Map

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
  - Interpreting Uml
- Language Specific Issues
- Tracks For Extraction
- Extracting Intention: Design Pattern
- Extraction For The Reuser
- Extraction of Interaction
- Conclusion

# Three Essential Questions

When we extract design we should be precise about:

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

# Interpreting UML

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

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

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

  Dictionary is a subclass of Set in Smalltalk (subclassing)

  but a Dictionary is not a subtype nor generalization of Set

So at the minimum we should have:

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

# Levels of Interpretations: Perspectives

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

- conceptual
- specification
- implementation

Three Perspectives:

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

# Attributes in Perspectives

Syntax:

visibility attributeName: attributeType = defaultValue

+ name: String

Conceptual:

Customer name = Customer has a name

Specification:

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

Implementation:

Customer has an attribute that represents its name

Possible Refinements

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

# Operations in Perspectives

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

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

Operations can be approximate to methods but they are more like abstract methods

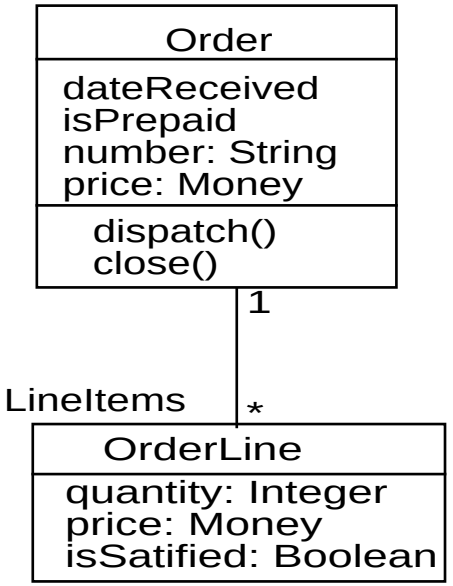
Possible Refinements:

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

# Associations

- Represent relationships between instances

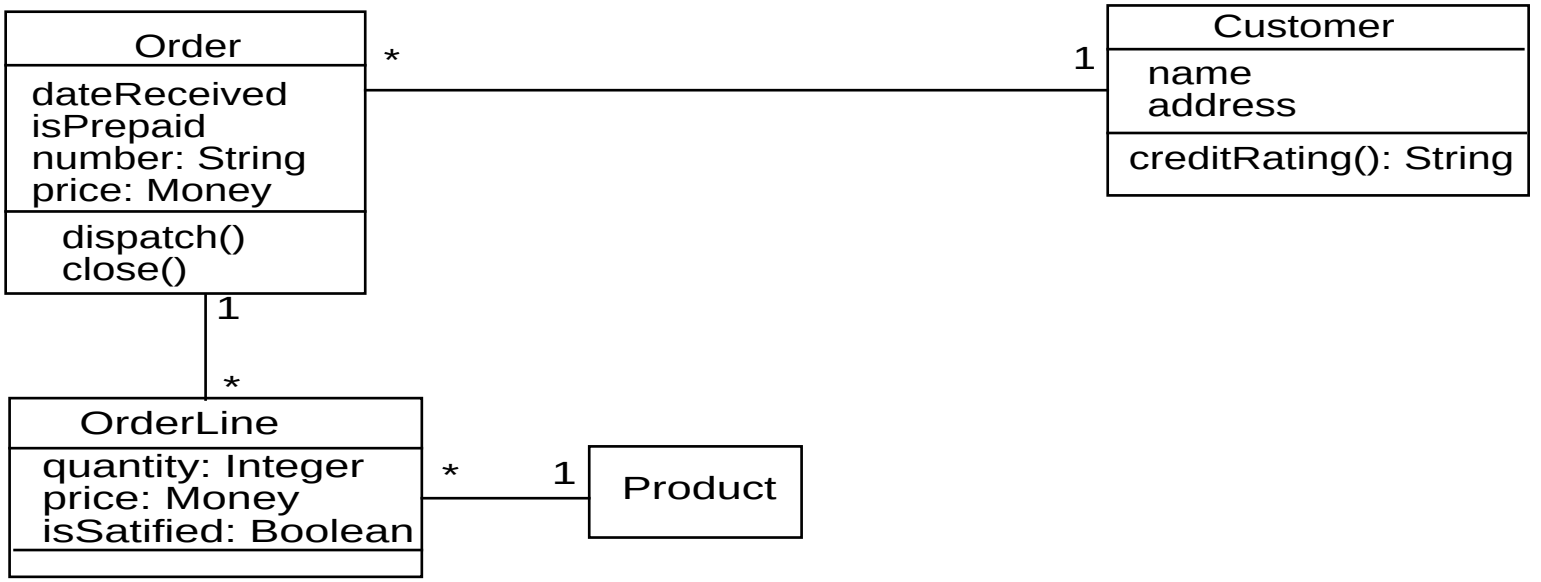
- Each association has two roles: each role is a direction on the association.
  - a role can be explicitly named, labelled near the target class
    if not named from the target class and goes from a source class to a target class
  - a role has a multiplicity: 1, 0, 1..*, 4




LineItems = role of direction Order to OrderLines
LineItems role = OrderLine role
One Order has several OrderLines

# Associations: Conceptual Perspective

Conceptual Perspective: associations represent conceptual relationships between classes




An Order has to come from a single Customer.

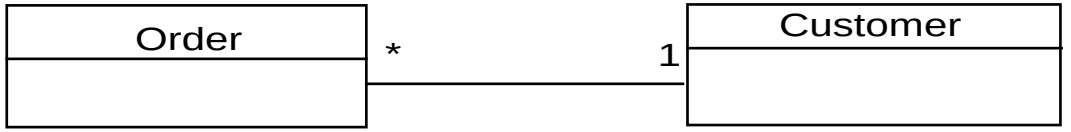
A Customer may make several Orders.

Each Order has several OrderLines that refers to a single Product.

A single Product may be referred to by several OrderLines.

# Associations: Specification Perspective

Specification Perspective: Associations represent responsibilities



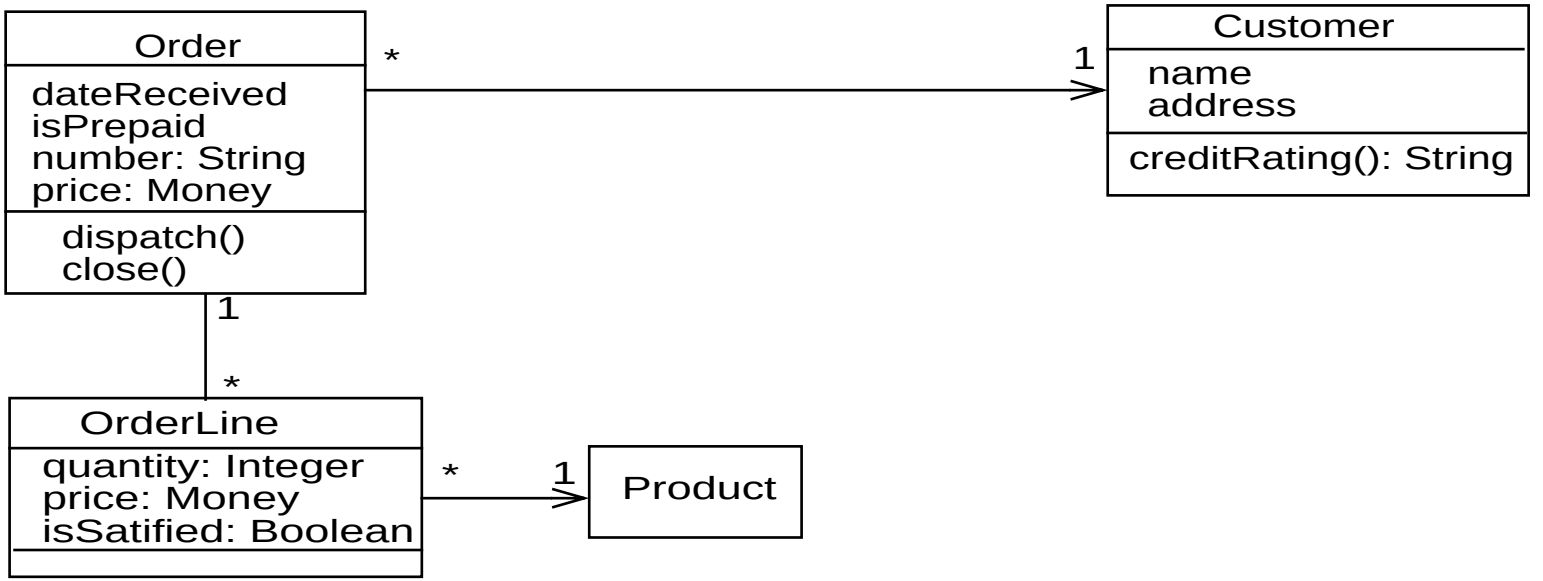

Implications:

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

Associations also implies responsibilities for updating the relationship, like:

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

# Arrows: Nagivability




No arrow = navigability in both sides or unknown

☞ conventions needed!!

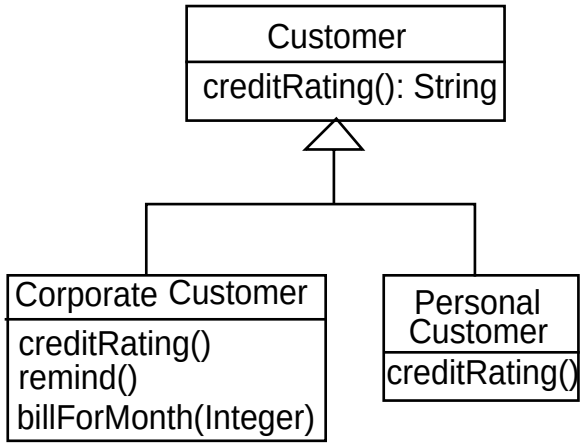
- Conceptual perspective: no real sense
- Specification perspective: responsibility

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

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

# Generalization

UML semantics only supports generalization and not inheritance.




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

Corporate Customer is a Customer

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

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

# Road Map

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
- Interpreting Uml
  - Language Specific Issues
- Tracks For Extraction
- Extracting Intention: Design Pattern
- Extraction For The Reuser
- Extraction of Interaction
- Conclusion

# Need for a Clear Mapping

UML

- language independent even if influenced by C++
- fuzzy (navigability, package...)
  - We should define how we interpret it
  - Define some conventions

In C++, examples show that:

```
Board& board()
```

board(): Board

```
Piece* myMap;
```

myMap: Piece

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

«public inherits»

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

checkWinner

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

width:Integer

# Private you said?! Which one?

What is the semantics of private, protected and public.

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

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

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

class based private

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

- instance based private
- methods are public

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

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

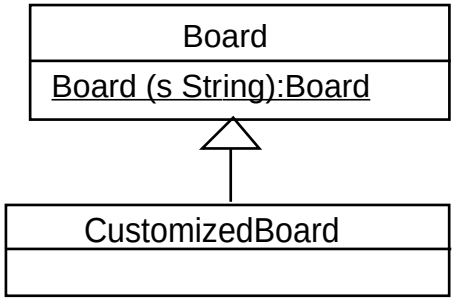
=> protected is more public than package

- classes can be marked as public or package

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

# Class Method Inheritance?!

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




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

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

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

☞ Conventions needed

# Some Possible Smalltalk Conventions

- In Smalltalk all methods returns self per default, so you may choose to only specify return type if it is not the same as the class.
- Attributes are all private
- All methods are public but there are ‘private categories’
- How do I distinguish between class instance variables and instance variables of the class?
- UML can be confusing when classes are objects too

  - uniqueInstance (c Class): Scheduler

    returns an instance of Scheduler class

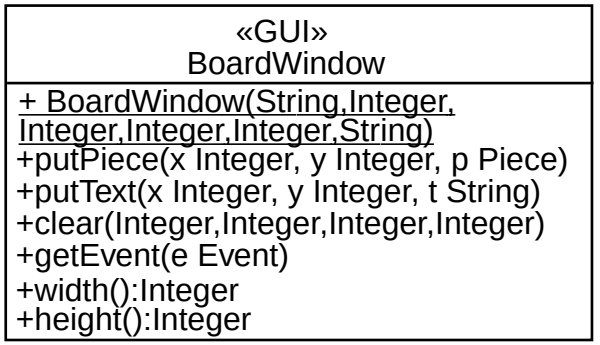
  - defaultWindowClass (): Class

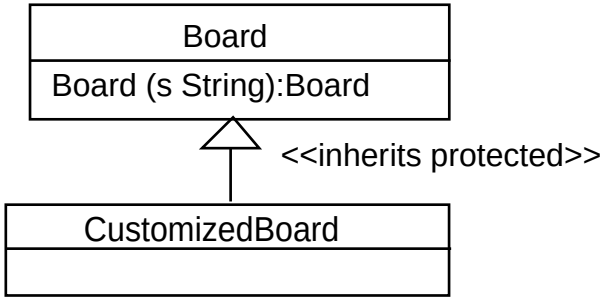
    returns the class window instance of class Class

# Stereotypes: to Represent Conventions!

Mechanism to specialize the semantics of the UML elements

- New properties are added to an element
- When a concept is missing or does not fit your needs select a close element and extend it.



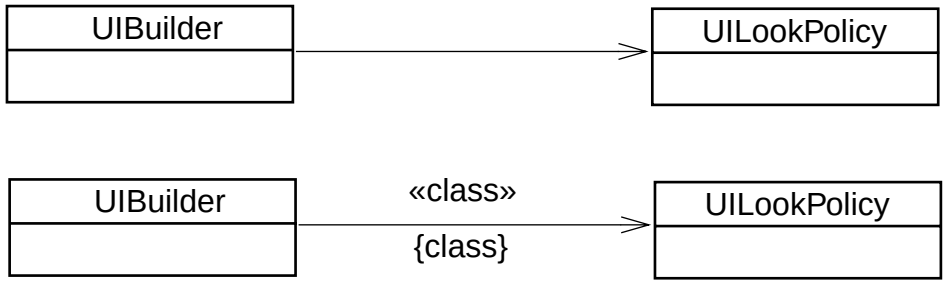




- 40 predefined stereotypes (c = class, r = relation, o = operation, a = attribute, d = dependency, g = generalization): metaclass (c), instance (r), implementation class (c) constructor (o), destructor(o), friend (d), inherits (g), interface (c), private (g), query (o), subclass (g), subtype (g), utility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

# Another Example: Instance/Class Associations

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

In VisualWorks, UIBuilder class is related to UILookPolicy class




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

- ☞ Use a stereotype or a constraint

# RoadMap

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
- Interpreting Uml
- Language Specific Issues
  - Tracks For Extraction
- Extracting Intention: Design Pattern
- Extraction For The Reuser
- Extraction of Interaction
- Conclusion

# Association Extractions (i)

Goal: Explicit references to domain classes

- Domain Objects
  Qualify as attributes only implementation attributes that are not related to domain objects.

  Value objects -> attributes and not associations,
  Object by references -> associations
  Ex: String name -> an attribute
  Order order -> an association
  Piece myPiece (in C++) -> composition

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

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

# Language Impact on Extraction

Attributes interpretation (like in the pictures)

- In C++ =>
  - Piece* myPiece ---> aggregation or assocation
  - Piece& my Piece ---> aggregation or association
  - Piece myPiece (copied so not shared) ----> composition

- In Smalltalk and Java
  - Aggregation and composition is not easy to extract
  - Piece myPiece ----> attribute or association or aggregation

# Method Signature for Extracting Relation

- Having attributes is not always necessary to interact with an object,
  => temporary references exist: temporary variable, method parameter, returned value

  - An instance can be dynamically created
  - An instance can pass itself as a parameter

Some relevant idioms: Self Delegation, Dispatched Interpretation [Beck97], Double Dispatch,...

=> Do not limit yourself to attributes, methods also contain implicit relationships

void putPiece (int x, int y, Piece piece)

=> relation between a Board and a Piece

When should we extract an aggregation and not a relation is not clear!

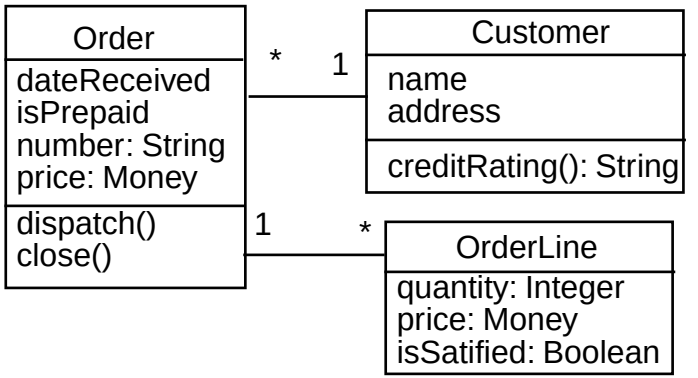
=> Analyse the language semantics (by copy, by reference)

=> Consider the various semantics of composition [Winston87]

# Convention Based Association Extraction

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

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



# Operation Extraction (i)

You may not extract

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

Example in Smalltalk, do not show

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

Use company conventions to filter

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

# Operation Extraction (ii)

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

- if you want to know that the functionality exists not all the details
  => select the method with the smallest prefix

- if you want to know all the possibilities but not all the ways you can invoke them
  => select the method with the more parameters

- if you want to focu on important methods
  => categorize methods according to the number of time they are referenced by clients
  => but a hook method is not often called but still important

**What is important to show: the Creation Interface.**

- Smalltalk class methods in ‘instance creation’ category,
- Non default constructors in Java or C++

# Road map

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
- Interpreting Uml
- Language Specific Issues
- Tracks For Extraction
  - Extracting Intention: Design Pattern
- Extraction For The Reuser
- Extraction of Interaction
- Conclusion

# Design Patterns as Documentation Elements

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

**But.**

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

  What is the difference between a State and a Strategy from the code point of view?

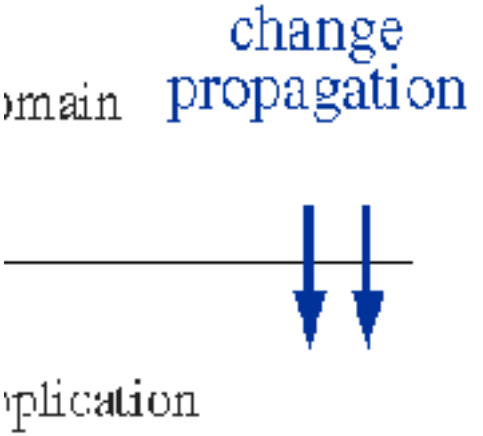
- Need somebody that knows
- Lack of support for code annotation so difficult to keep the use of patterns and the code evolution [Flor97a]

# Road map

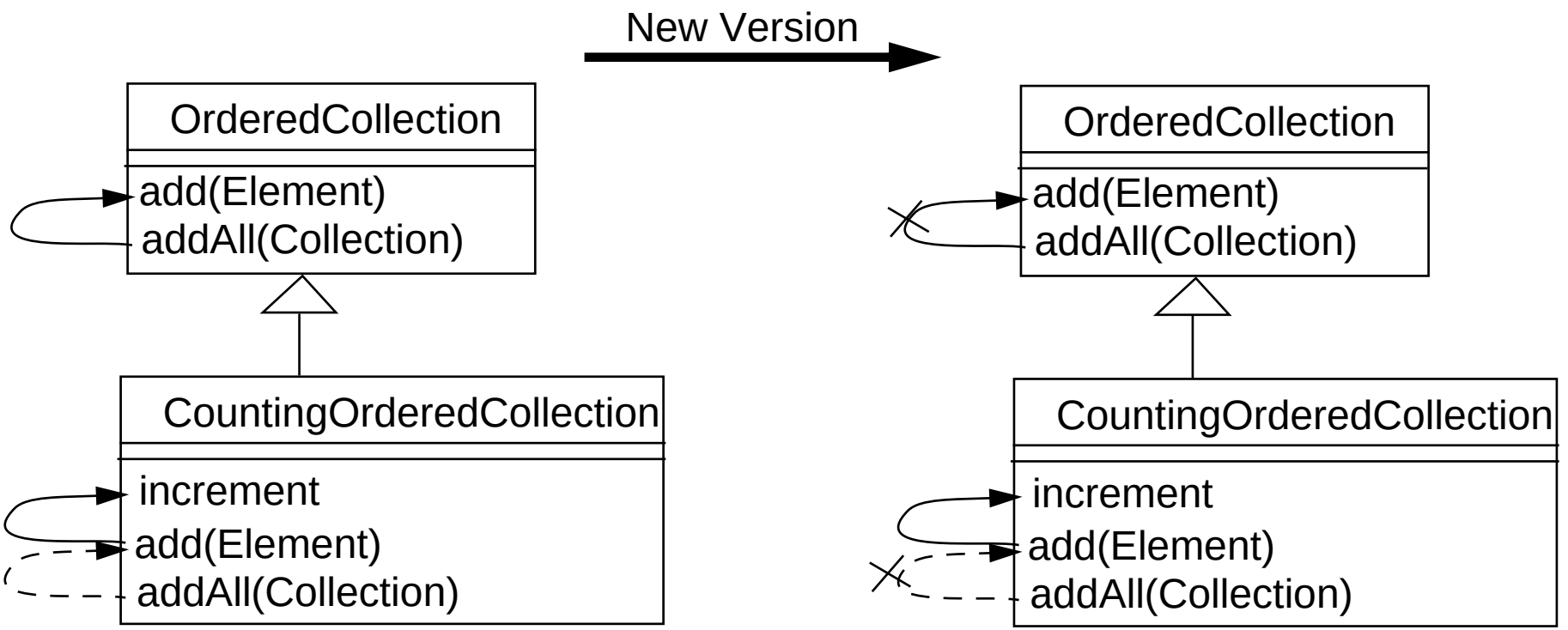
- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
- Interpreting Uml
- Language Specific Issues
- Tracks For Extraction
- Extracting Intention: Design Pattern
  - Extraction For The Reuser
- Extraction of Interaction
- Conclusion

# Evolution Impact Analysis: Reuse Contract

How to identify the impact of changes?

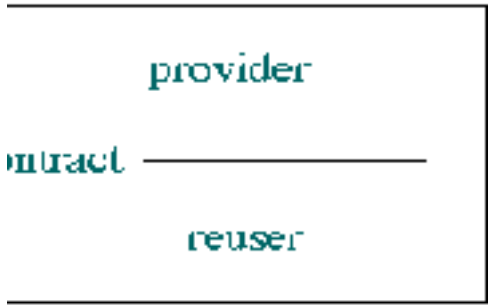


# Example




Not all the elements are counted

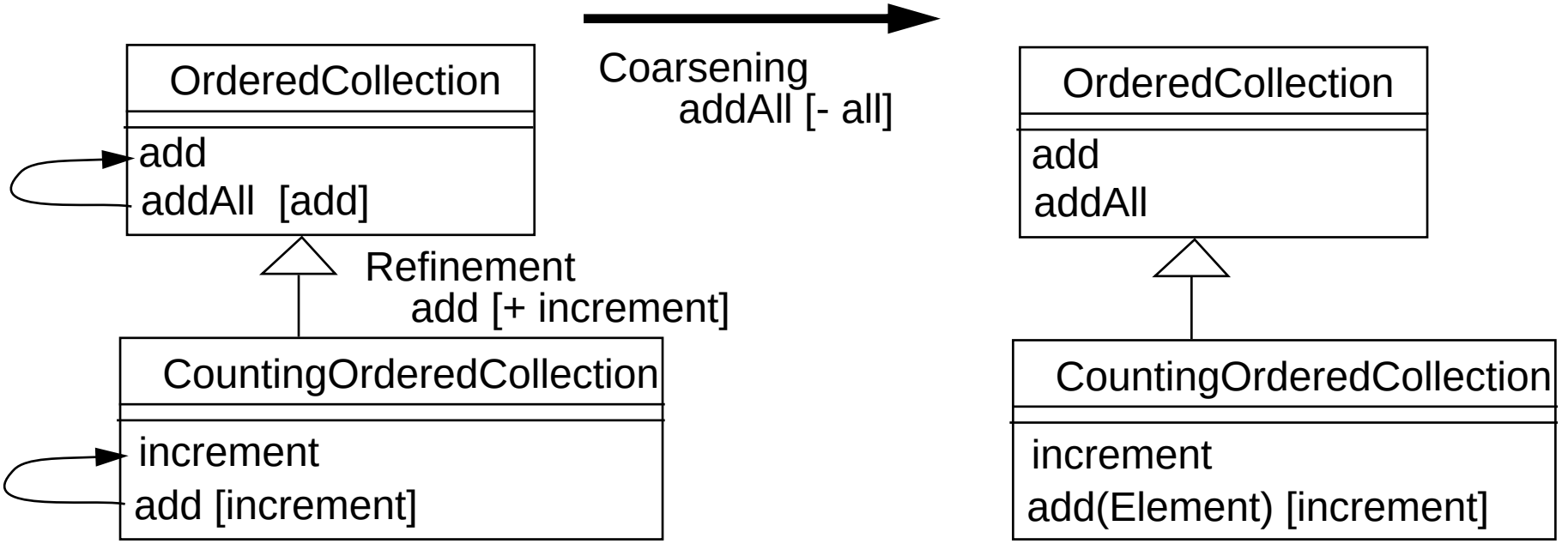
# Reuse Contracts: General Idea




Reuse Contracts [Stey96a] propose a methodology to:

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

# Example




effort estimate
addAll needs to be overrident too

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

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

# Road Map

- Why Extracting Design? Why Uml?
- Basic Uml Static Elements
- Experimenting With Extraction
- Interpreting Uml
- Language Specific Issues
- Tracks For Extraction
- Extracting Intention: Design Pattern
- Extraction For The Reuser
  - Extraction of Interactions
- Conclusion

# Documenting Dynamic Behaviour

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

- Calling relationships is well suited for
  - method interrelationships
  - class interrelationships

UML proposes Interaction Diagrams = Sequence Diagram or Collaboration Diagram

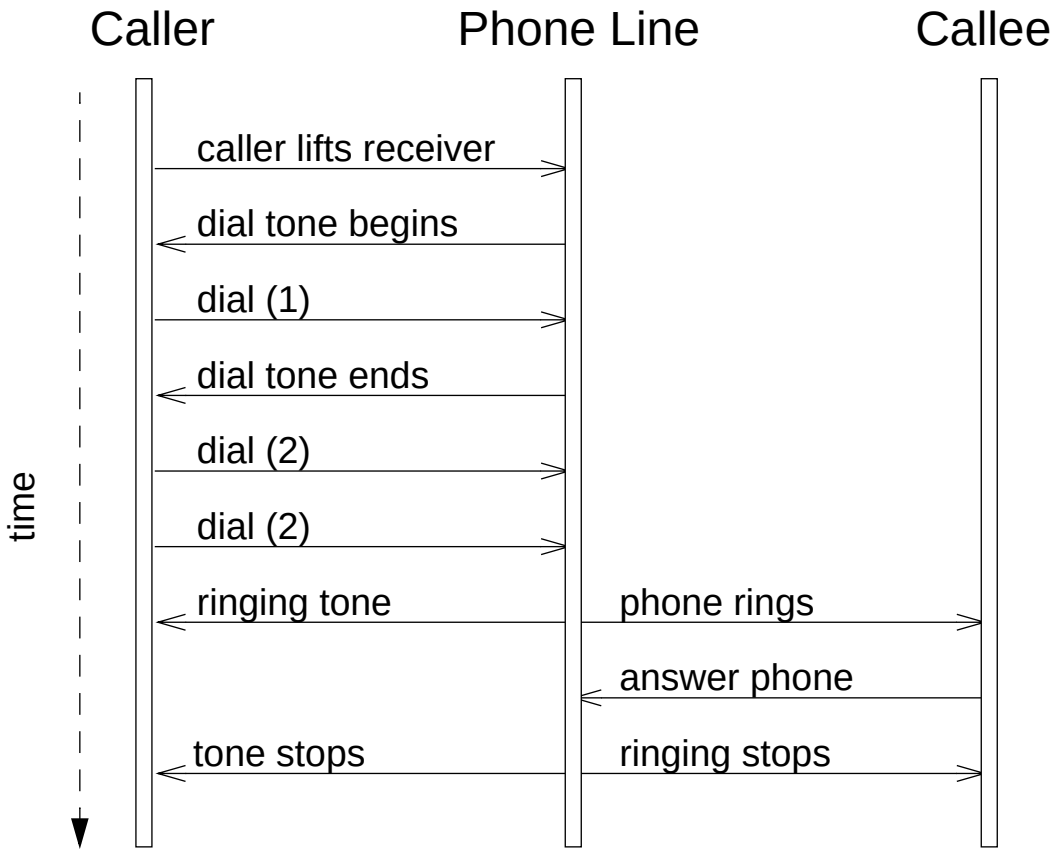
# Sequence Diagrams

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!



# Statically Extracting Interactions

**Pros:**

- Limited resources needed
- Do not require code instrumentation

**Cons:**

- Need a good understanding of the system
  - state of the objects for conditional
  - compilation state #ifdef...
  - dynamic creation of objects
- Potential behavior not the real behaviour
- Blur important scenario

# Dynamically Extracting Interactions

**Pros:**

- Help to focus on a specific scenario
- Can be applied without deep understanding of the system

**Cons:**

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

For dealing with the huge amount of information

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

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

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

# Lessons Learnt

**You should be clear about:**

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

**For Future Development**

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

**UML as Support for Design Extraction**

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

# 5. Software 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

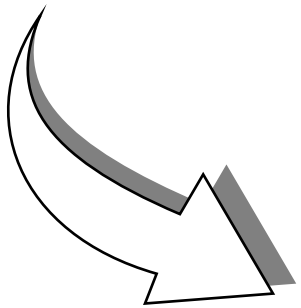
# Algorithmic cost modelling

- 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

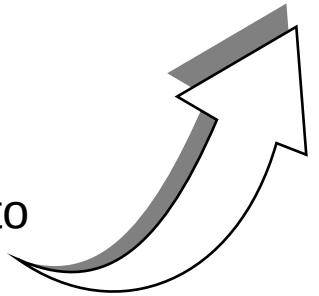
**A. Measure**

Develop a system model and measure its size



**B. Estimate**

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



**C. Interpret**

Adapt the effort with respect to a specific development project plan

# 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) $^{1.2} \times M$

NB: KDSI = Kilo Delivered Source Instructions

# 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.

# 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

# 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 complex and potentially faulty 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

# 6. Metrics, Visualisations and Interactions for Reverse Engineering

Michele Lanza
lanza@iam.unibe.ch
031 631 3547

Stéphane Ducasse
ducasse@iam.unibe.ch
031 631 4903

# Contents


- Introduction
- Metrics and Measurements
- Visualisation
  - Possible Approaches
  - Examples
- Our Approach: CodeCrawler
  - Examples
- Online Demo
- Conclusion

# Introduction

- Goals of this Lecture:
  - Metrics. Why? Which ones?
  - Visualisation. Why? How?
  - CodeCrawler: An example of a Reverse Engineering platform.
  - Industrial Experiences.
  - Online Demo: Preparation for the Lab Experience next week.

# Metrics

- Metrics and Measurement
- Metrics for reverse engineering
- Selection of OO metrics
- Step back and look

# Metrics and Measurements

[Wey88] defined nine properties that a software metric should hold [Fenton for critics]. For OO only 6 properties are really interesting [Chid 94, Fenton]

**Noncoarseness:** Given a class P and a metric m, another class Q can always be found such that m (P) != m(Q)
-> not every class has the same value for a metric

**Nonuniqueness.** There can exist distinct classes P and Q such that m(P) = m(Q)
-> two classes can have the same metric

**Design Details are Important.** The specifics of a class must influence the metric value. Even if a cclass performs the same actions details should have an impact on the metric value.

**Monotonicity.** m(P) <= m (P+Q) and m(Q) <= m (P+Q), P+Q is the combination of the classes P and Q.

**Nonequivalence of Interaction.** m(P) = m(Q) ! -> m(P+R) = m(Q+R) where R is an interaction with the class.

**Interaction Increases Complexity.** m(P) + (Q) < m (P+Q).
-> when two classes are combined, the interaction between the too can increase the metric value

Conclusion: Not every measurement is a metric.
But take care because this is fuzzy and academic

# Metrics for Reverse Engineering

Pragmatic Criteria to evaluate OO metrics

- Easy to Compute (E)
- Based on Code
- Simple stable definition (S)

Size of the system, system entities

- Class size, method size, inheritance

  The intuition: a system should not contain too much big entities

  Pro really big entities may be problematic

  Cons can be really difficult and complex to understand

Cohesion of the entities

- Class internals,

  The intuition: a good system is composed by cohesive entities

Coupling between entities

- Within inheritance: coupling between class-subclass
- Outside of inheritance

  The intuition: the coupling between entities should be limited

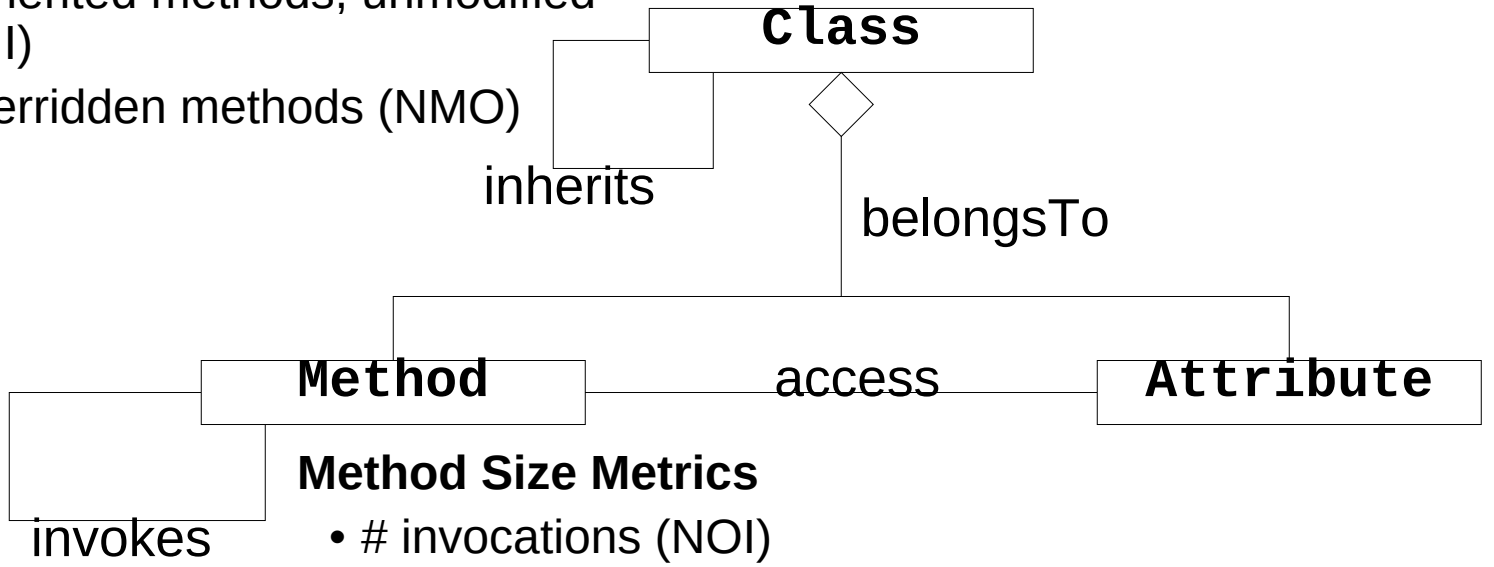
# Which Metrics to Collect (Definitions)?

**Inheritance Metrics**

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

**Class Size Metrics**

- # methods (NOM)
- # instance attributes (NIA, NCA)
- # Σ of method size (WMC)
- Cohesion (LCOM), CBO




**Method Size Metrics**

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

# Class size

- (NIV) [Lore94] Number of Instance Variables (E++, S++)
- (NCV) [Lore94] Number of Class Variables (static) (E++, S++)
- (NOM) [Lore94] Number of Methods (public, private, protected)(E++, S++)
- (LOC) Lines of Code (E+, S++)
- (NSC) Number of semicolons [Li93]-> number of Statements (E, S+)
- (WMC) [Chid94] Weighted Method Count (E--, S++)

  WMC = SUM $c_i$

  where c is the complexity of a method (number of exit or McCabe Cyclomatic Complexity Metric)

# Class Complexity

- (RFC) Response For a Class [Chid94]

Response Set for a Class (RS) is the set of methods that can be executed in response to a message.

$$RS = \{M\} \text{ Union}_i \{R_i\}, \quad RFC = | RS |$$

where $\{R_i\}$ is the set of methods called by method i and {M} the set of all the methods in the class.

# Hierarchy Layout

- (HNL) [Chid94] Hierarchy Nesting Level , (DIT) [Li93] Deep of Inheritance Tree (E++, S++)
  HNL, DIT = max hierarchy level
- (NOC) [Chid94] Number of Children (E++, S++)
- (WNOC) Total number of Children (E++, S++)
- (NMO, NMA, NMI, NME) [Lore94] (E+, S++)
  Number of Method Overriden, Added, Inherited, Extended (super call)
- (SIX) [Lore94] (E+,S+, Sceptic interpretation)
  SIX (C) = NMO * HNL / NOM
  Weighted percentage of Overriden Methods

# Method Size

- (MSG) Number of Message Sends
- (LOC)
- (MCX) Method complexity (E-, S+)
  Total Number of Complexity / Total number of methods
  API calls= 5, Assignment = 0.5, arithmetics op = 2, messages with para = 3....
- (NP) Number of Parameters

# Class Cohesion (i)

- (LCOM) [Chid94] Lack of Cohesion in Methods (E,S--, not reliable) [Hitz95a]

  $I_i$ = set of instance variables used by method $M_i$

  let $P = \{ (I_i, I_j) \mid$ Intersection $(I_i, I_j)$ is Empty,

  $Q = \{ (I_i, I_j) \mid$ Intersection $(I_i, I_j)$ is not Empty

  if all the sets are empty, P is empty

  $LCOM = |P| - |Q|$ if $|P| > |Q|$

  $= 0$ otherwise

# Class Cohesion (ii)

- (TCC) [Biem95] Tight Class Cohesion (E,S, not really used, complex)
  TCC is the relative number of directly connected methods
  TCC = NDC / NP
  NDC = Number of Direct Connection
  NP = n * (n -1) /2 = Maximum possible direct and indirect connected methods
  A class is represented by a collection of Abstract Method (AM)
  AM (M) = set of directly and indirectly accessed instance variables by M
  Abstracted Class: AC = [AM (M) | M belongs to V(C)]
  V(C) = Visible method of C and C's ancestors.
  NP (C) = total number of abstracted method pairs in AC(C)

- (LCC) [Biem95] Loose Class Cohesion (E,S not really used, complex)
  TCC is the relative number of directly or indirectly connected methods
  LCC = (NDC + NIC) / PC
  NIC = Number of Indirect Connections

# Class Coupling (I)

- (CBO) [Chid94] Coupling Between Objects
  CBO = number of other class to which it is coupled (E, S+, fuzzy definition)
  See [Hitz94] for a discussion
- (DAC) [Li93] Data Abstraction Coupling (E-, S+)
  DAC = number of ADT's defined in a class
- (CDBC) [Hitz96] Change Dependency Between Classes (E-, S+, not simple, not used, not commented in the literature)
  Impact of changes from a server class (SC) to a client (CC).
  CDBC(CC,SC)= min (n, A)
  n = number of methods of CC
  A = SUM (m1, ai)+ (1-k) SUM (m2, ai)
  1-k = degree of stability of SC
  a = number of methods of CC potentially affected by a change
  m1 accesses of CC to the implemention of SC
  m2 accesses of CC to the interface of SC

# Class Coupling (Ii)

- (LD) [Hitz96] Locality of Data (E+,S+, not used, not commented)
  LD = SUM $|L_i|$ / SUM $|T_i|$
  Mi = methods without accessors
  Li = non public instance variables, inherited protected of superclass, static variables of the class
  Ti = all variables used in Mi, except non-static local variables (??)

# Metrics? Stepping Back

About the impact of the computation

Example:

- number of attributes

  should we count private attributes in NIV?

  Why not?
- number of methods (private, protected, public, static, instance, operator, constructeurs, friends)

What to do?

- Try first simple metrics, with simple extraction
- Take care about absolute threshold

  Metrics are good as a differential

  Metrics should be etalonned
- Do not numerically combine them: what is the multiplication of oranges and apples: Jam!

# Visualisation

- The Motivation: why are we doing it?
- Possible Approaches
  - Examples
- Our Approach: CodeCrawler
  - The Idea
  - Examples
  - The Interaction

# The Motivation: Why are we visualising stuff?

"Software is intangible, having no physical shape or size. Software visualisation tools use graphical techniques to make software visible by displaying programs, program artifacts and program behaviour."

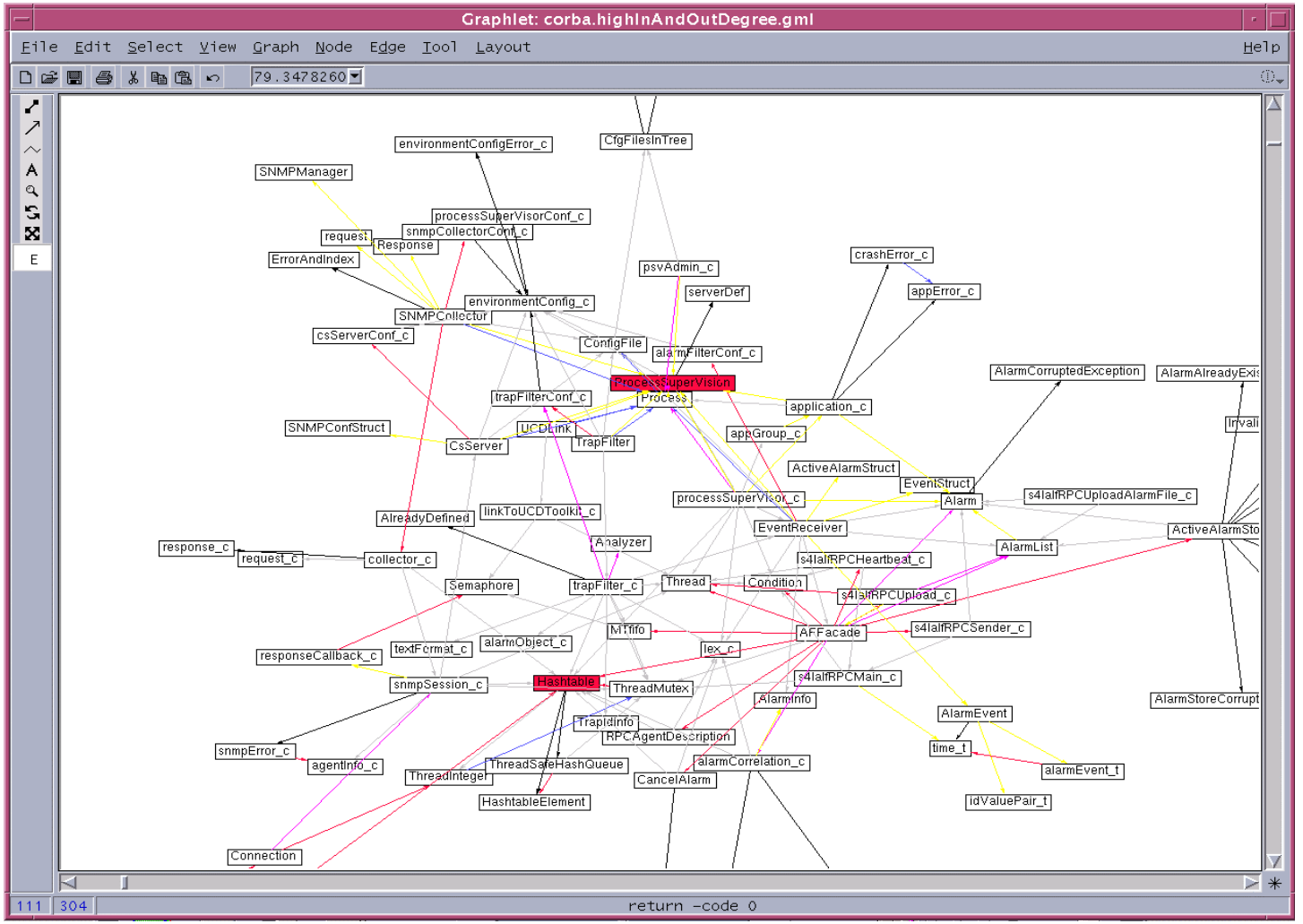
T.S. Ball & S.E.Eick

- Reduction of Complexity:
  - Transformation from purely text-based form to a higher abstract representation
- Generate different views on software system.
- Let the system tell you what it's all about
- Documentation of the system

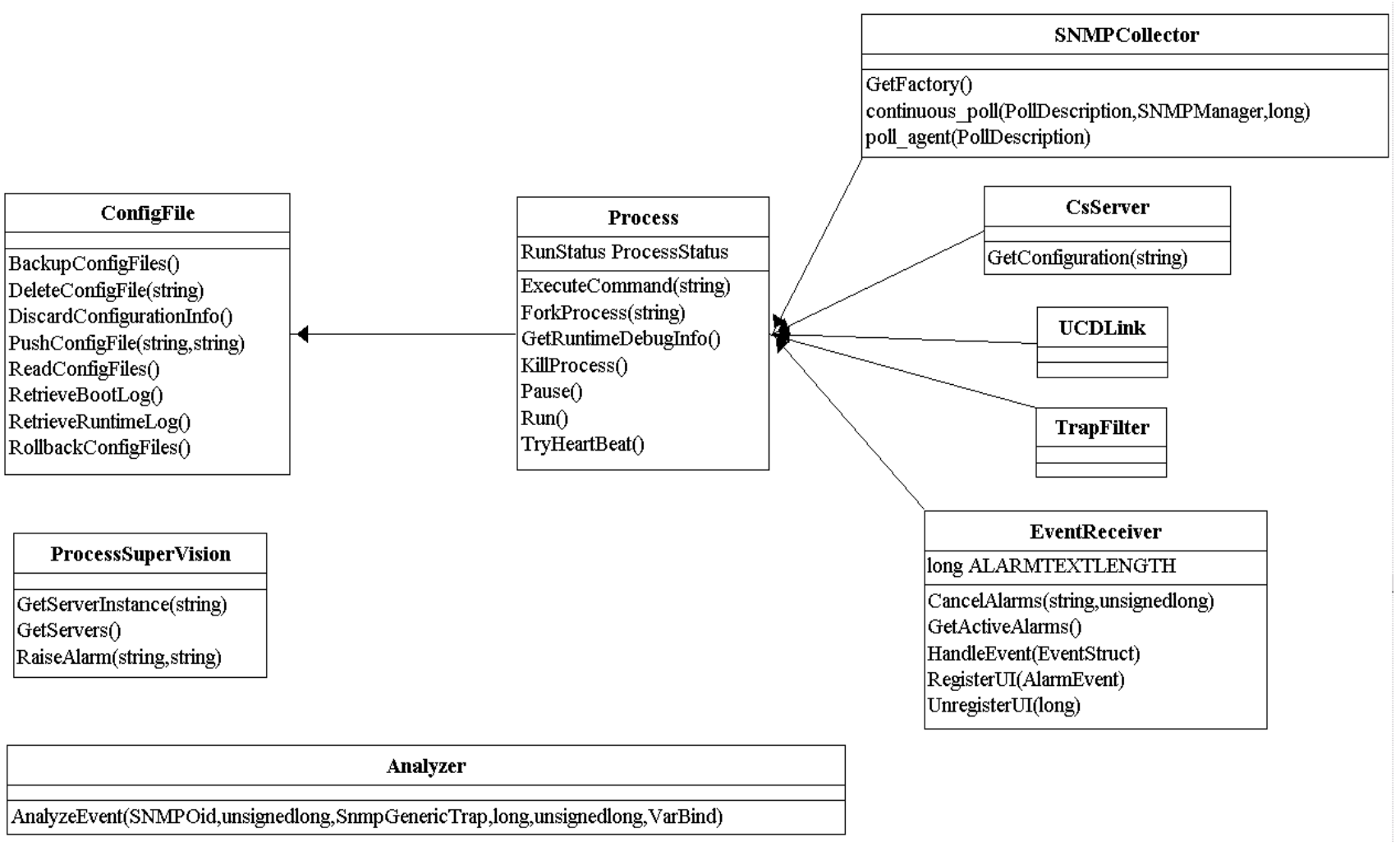
# Visualisation: Possible Approaches

- A decent graph layout can be a hard task...
  - Efficient space use (physical limits of a screen)
  - Edge crossing problem
  - UML
  - Colors are nice, but... there are no conventions!
- Tradeoff between usefulness and complexity
- Keeping a focus is hard:
  - Where should we look?
  - What should we look for?
- Examples from real-world visualisation systems

# Example: Goose/ Graphlet



# Example: Mermaid



# Let's summarise...

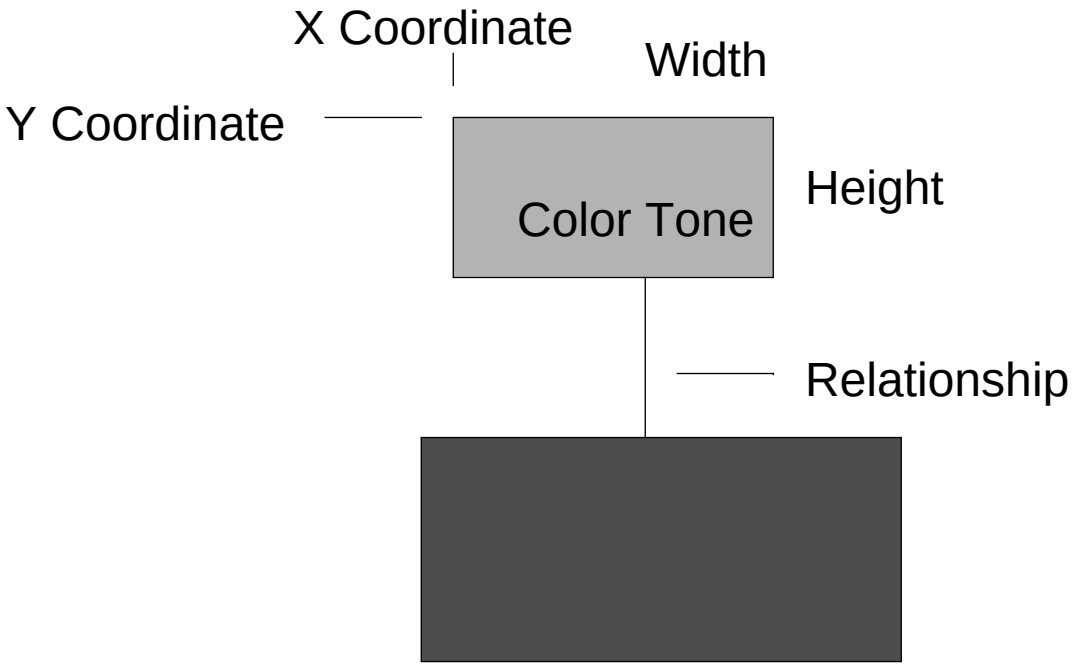
- What kind of information do we want to convey about an entity?
  - Name
  - Structure
  - Size
  - Role
  - etc.
- How do they communicate and how do we want to see that?
  - Colored Edges
  - Weighted Edges
  - Edges?
  - etc.
- At what granularity level can we apply a certain display?
  - Full system
  - Single class or small subsystem

# Our Approach: CodeCrawler

- A lightweight combination of:
  - Visualisation
  - OO Metrics
  - Interaction
- The main constraint is:
  - Simplicity
- OO Entities are rendered as colored rectangles:
  - Classes, Methods, Attributes, etc.
- OO Relationships are rendered as edges:
  - Inheritance, Invocation, Access, etc.

# The Idea: Visualising Metrics

- Directly render up to five metrics on node node:
  - Size (2)
  - Color (1)
  - Position (2)

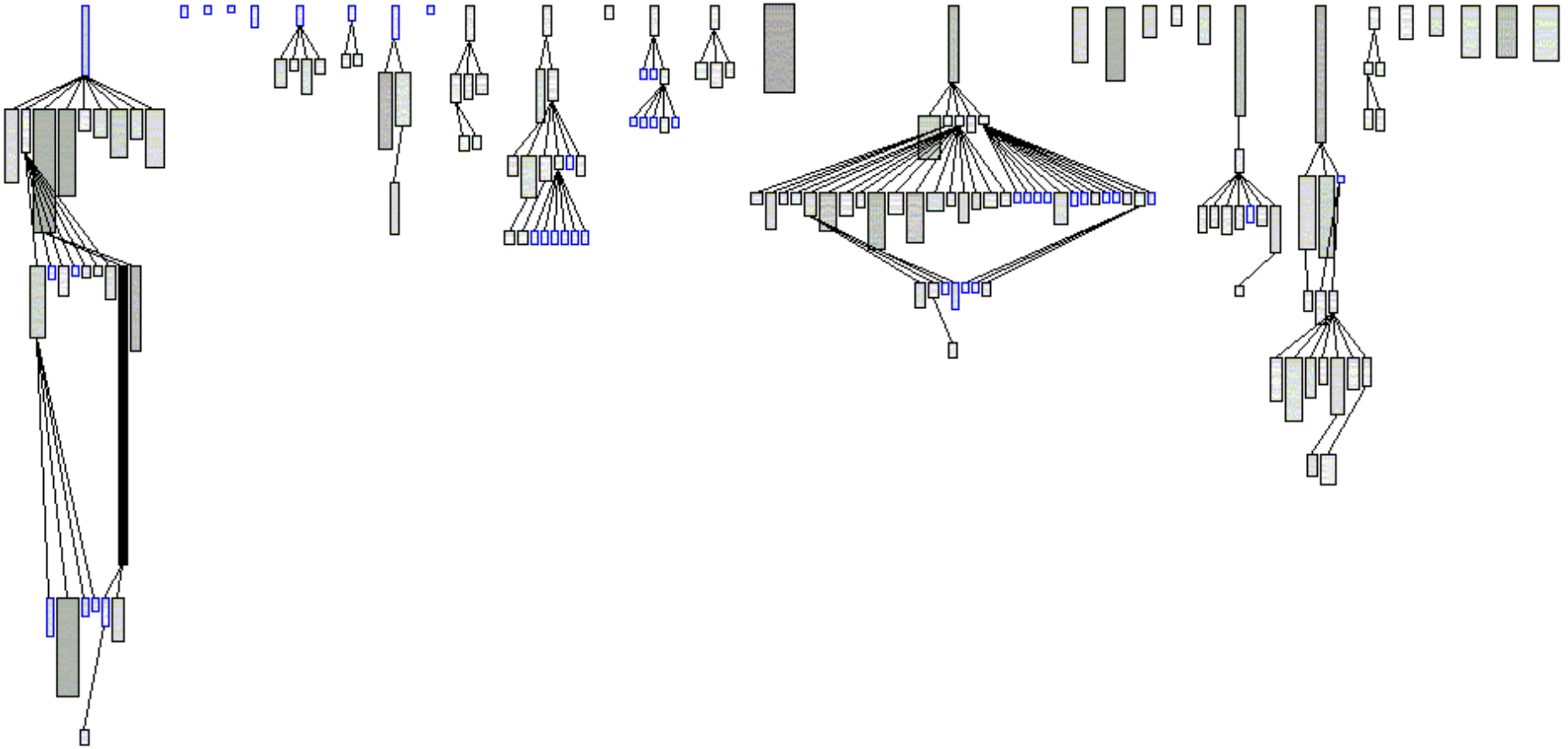


# CodeCrawler: Some Examples

- Taken from the Refactoring Browser
- Try to understand and interpret the following graphs...
  - System Complexity
  - Method Efficiency Correlation
  - Inheritance Classification
  - Service Class Detection

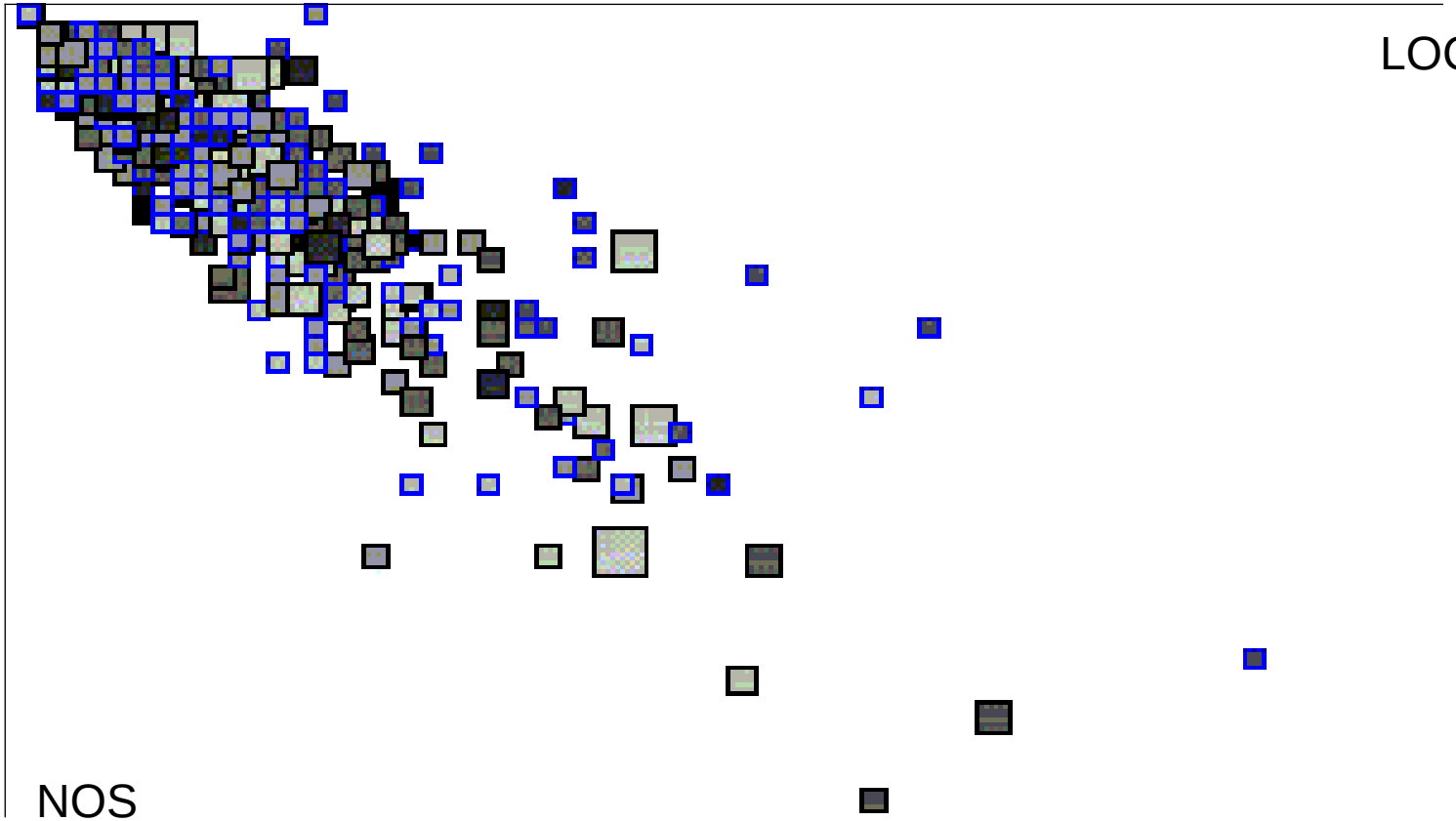
# System Complexity

Metrics: NIV, NOM, LOC



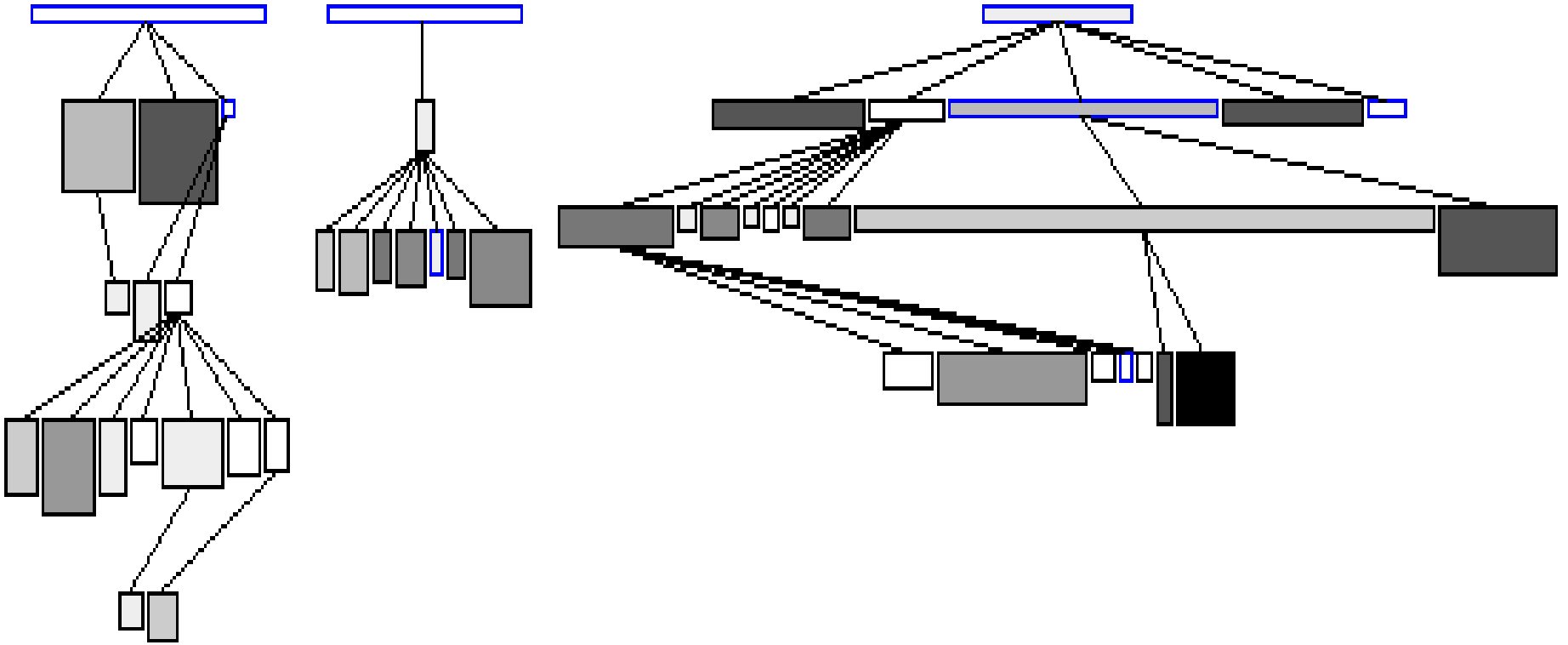
# Method Efficiency Correlation

Metrics: NOP, NOP, HNL, LOC, NOS



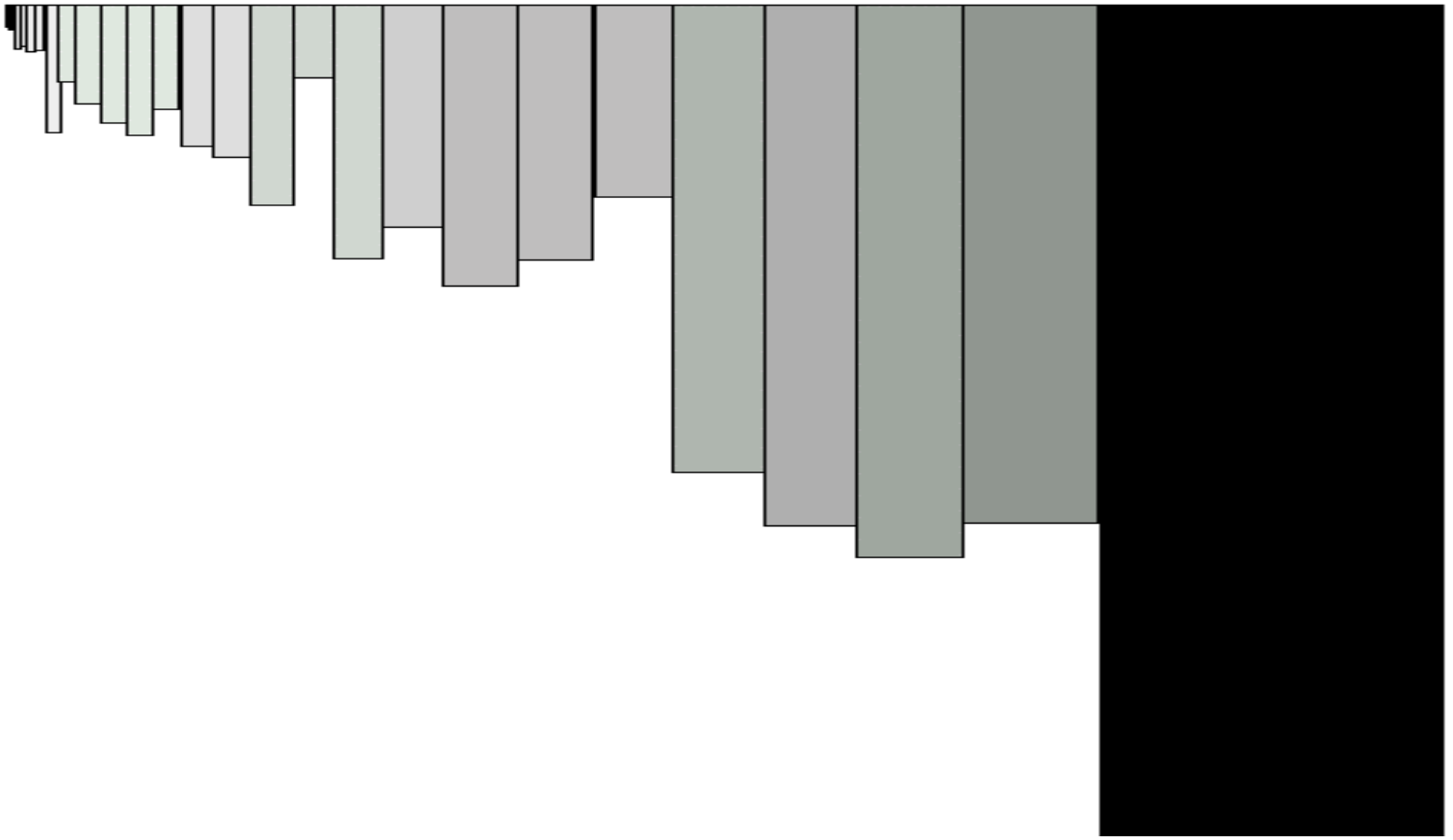
# Inheritance Classification

Metrics: NMA, NMO, NME



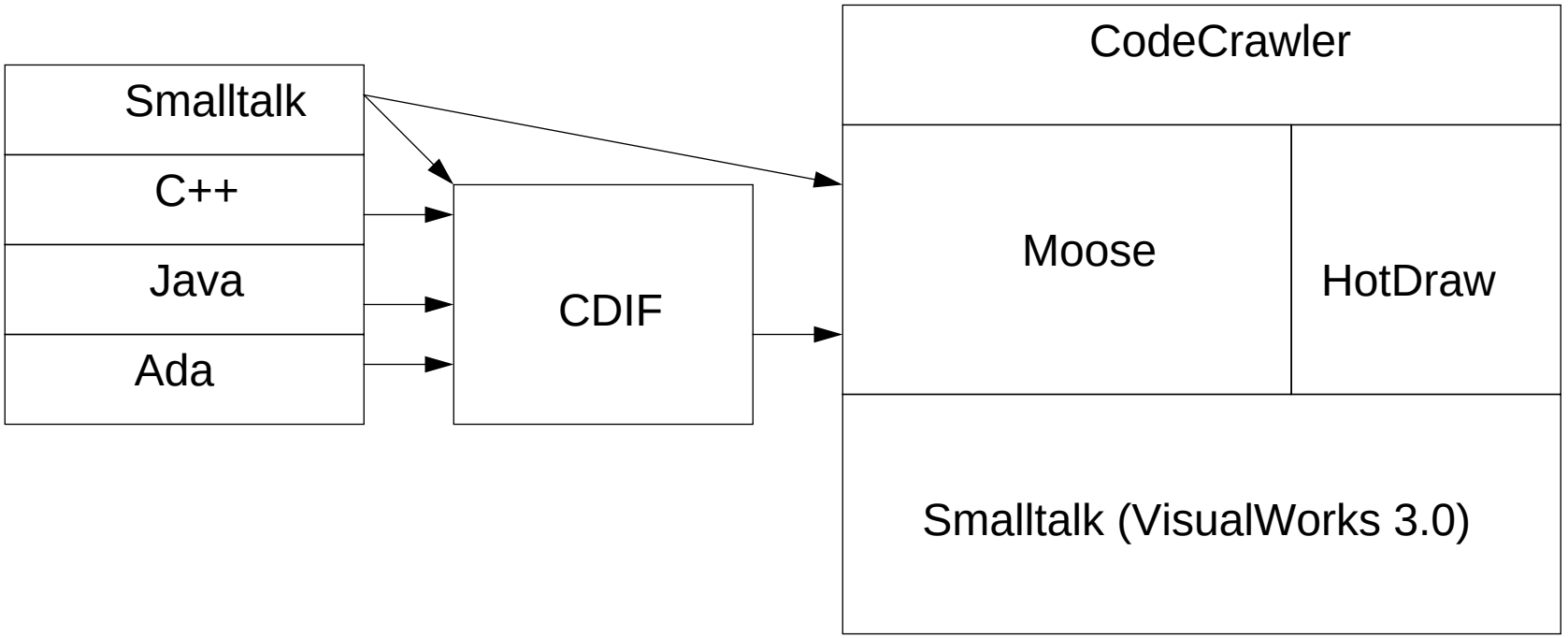
# Service Class Detection

Metrics: NOM, LOC, LOC



# CodeCrawler's Logic

- Language Independent (CDIF Interface)
- Platform Independent (Smalltalk)



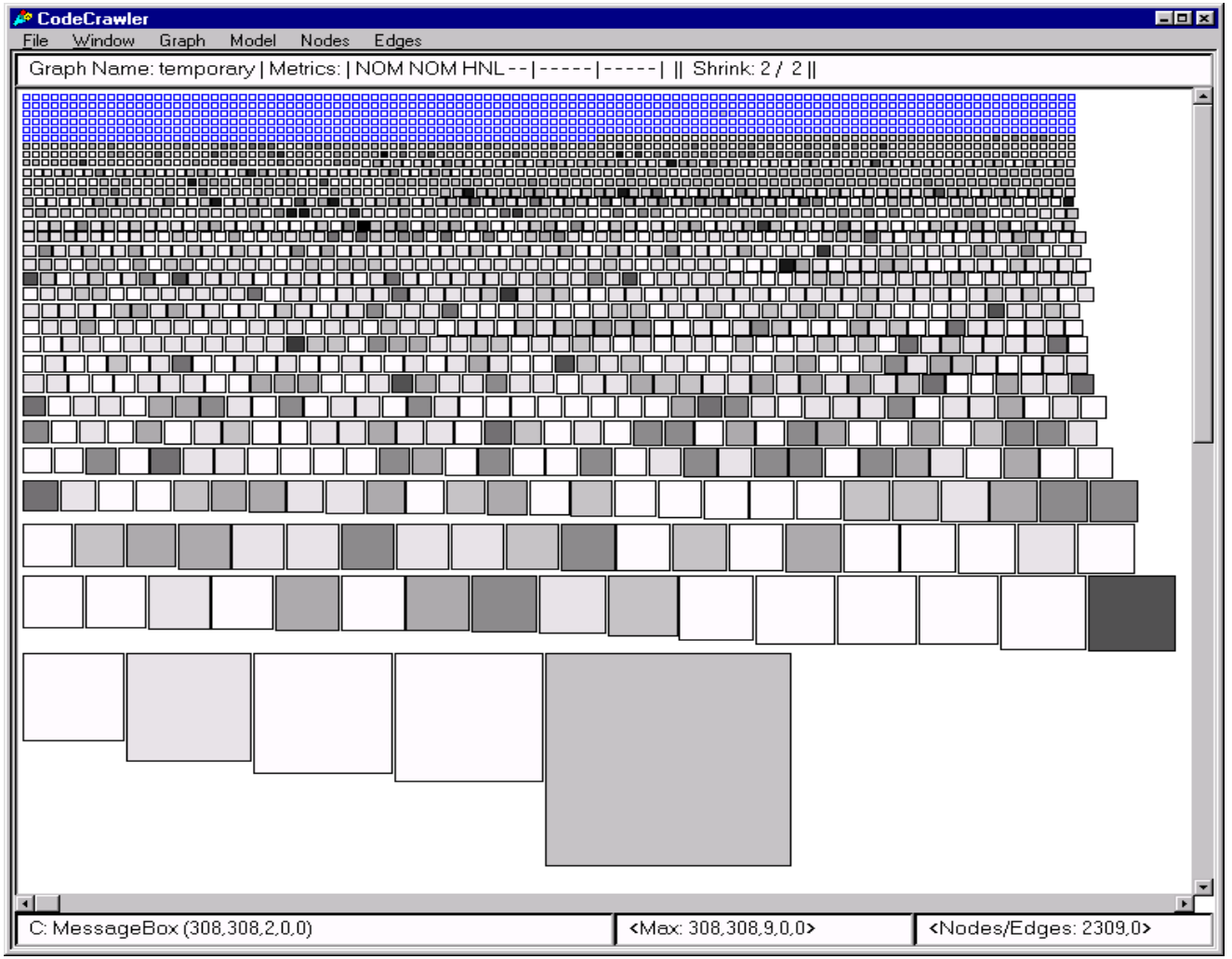
# CodeCrawler: Pro And Contra

- Pro:
  - Intuitive Approach: simple is beautiful
  - Quick Insights
  - Language Independence
  - Platform Independence
- Contra:
  - Simplicity
  - Its reliability depends on several external factors, i.e. parsing. The language idependence does come at a certain cost...

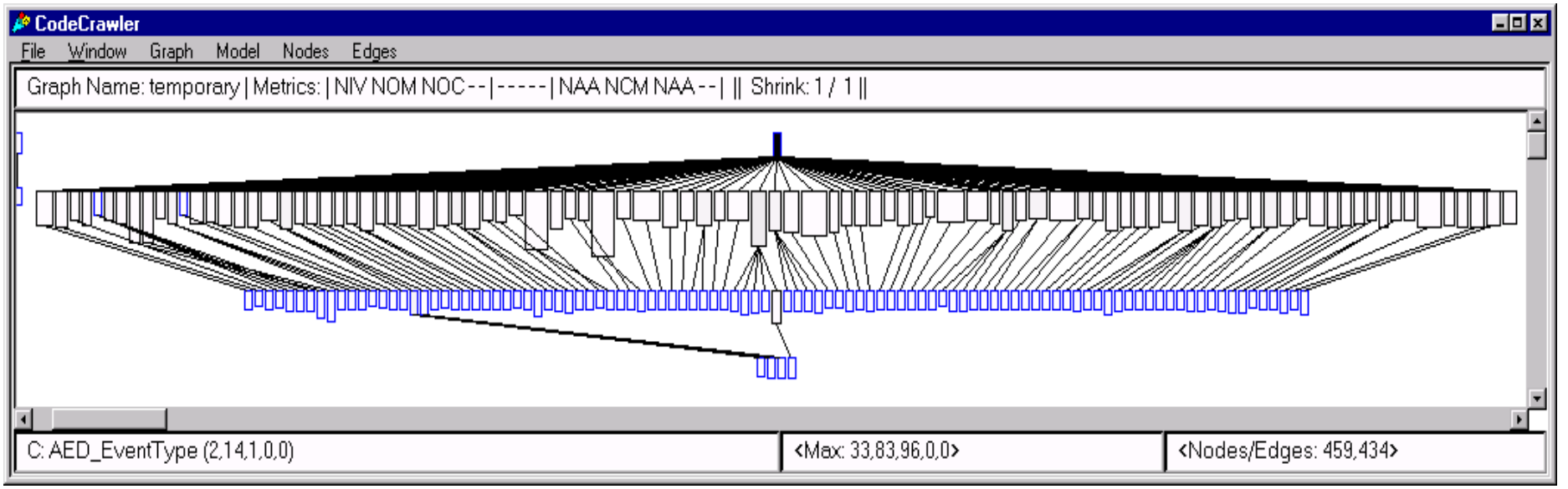
# CodeCrawler: The Case Studies

- Academic:
  - VisualWorks 3.0 ( > 500 classes)
  - Refactoring Browser ( > 150 classes)
  - Duploc (> 100 classes)
- Industrial:
  - XXX (C++, 1.2 MLOC, > 2300 classes)
  - XXY (C++/Java, 120 kLOC, > 400 classes)
- The Approach Works!
  - Let's have a look at some examples...

# Example: Visualisation of a very large system



# Example: Flying Saucers



# Conclusion & Possible Projects

- Visualisation is necessary, because...
  - Systems have become too complex to cope with in their pure textual form

- Possible Projects
  - Add Grouping Techniques, i.e. collapsing of nodes
  - Generate Graph Views based on OO Heuristics
  - Add (animated?) Spring Layouting Algorithms: the system will find its own layout.
  - Closer views on a class: how can a class be displayed to make it tell you what kind it is...

There's a lot to be done...come around and ask!

# Bibliography

[Weyu88] E. Weyuker, "Evaluating Software Complexity Measures", IEEEE Transactions on Software Engineering, vol 14, n 9. 1988.

[Chid94] S. Chidamber, C Kemerer "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, vol 20, n 6, June 1994

[Li93] W. Li, S. Henri, "Maintenance Metrics for the Object Oriented Paradigm", IEEE Proc. First International Software Metrics, 1993

[Hitz95a] M. Hitz, B. Montazeri, "Measuring Coupling and Cohesion in Object-Oriented Systems", Proceedings International Symposium on Applied Corporate Computing, 1995.

[Hitz96] M. Hitz, B. Montazeri, "Measuring Coupling in Object-Oriented Systems", Object Currents, Vol 1, N 4, 1996

[Lanz99a] M. Lanza, "Combining Metrics and Graphs for Object Oriented Reverse Engineering",University of Bern,1999

[Deme99] S. Demeyer, S. Ducasse and M. Lanza, "A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization", WCRE'99 Proceedings (6th Working Conference on Reverse Engineering), 1999

# 7. Lab session — CodeCrawler

BEDAG INFORMATIK

# OBJECT-ORIENTED SOFTWARE COST ESTIMATION

December 1999

**Dr. Simon Moser**

moser@acm.org

# Topics:

The Importance of Measurements & Estimates

A Measurement-Based Estimation Process

Software Models (Meta-Models)

Software Metrics

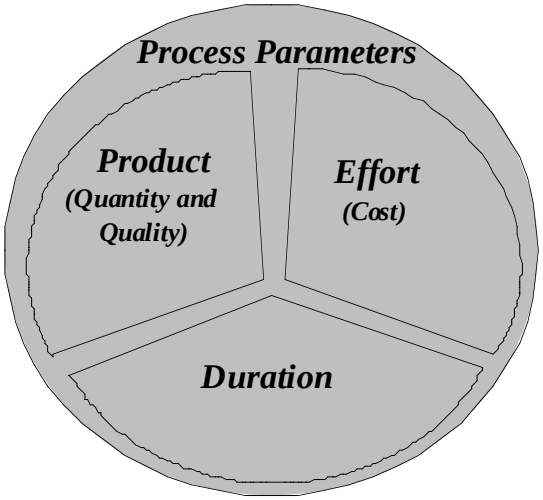
Results of a Field Study

An Example

Future Work

# The Importance of Measurements & Estimates (1/2)

## 3 Process Parameters to Control:




**Measurement = Knowing where you stand**

*Prerequisite for:*

- Generic problem solving
- Process improvement / Quality management

# The Importance of Measurements & Estimates (2/2)

**Estimate = The expectations of a project**

*Under-estimates:*
time pressure ⇨ stress ⇨ frustration ⇨ people turnover
too tight budget ⇨ save on functionality and quality ⇨ "maintenance dilemma"
no more money ⇨ late project cancelation

*Over-estimates:*
time for fancy stuff ⇨ the over-estimate will turn into an under-estimate

*Estimation evaluation criteria:*
(1) Accuracy
(2) Cost and speed of the overall estimation process

# A Measurement-Based Estimation Process (1/3)

[People throwing darts to a calender, the date hit will be the estimated deadline...]

*A non-measurement-based estimation process!*

# A Measurement-Based Estimation Process (2/3)

A non-software example: estimating the duration of a bush-walk

(**A**) Measure the walk distance on a map
(**B**) Derive a first duration according to some rule-of-thumb
(**C**) Interpret this estimate to specifics (restaurants on the way, ...)

= measurement-based estimation

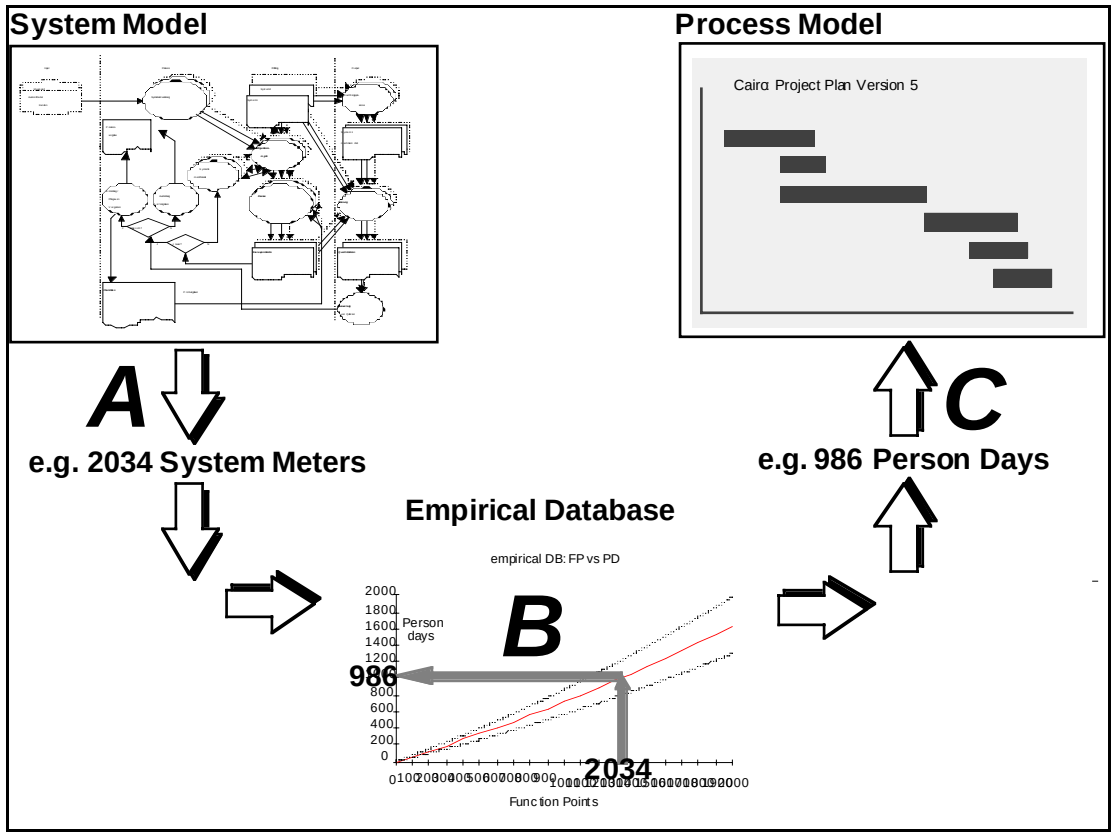
*Improvements with respect to accuracy:*

- more detailed map or model
- better metric (e.g. taking height differences into account)
- specific empirical database instead of general rule-of-thumb

*Improvements with respect to cost of estimation:*

- lower-resolution map

# A Measurement-Based Estimation Process(3/3)




(adapted from T. DeMarco, 1982 [1])

# Software Process Models (1/2)

[The "standard" software process is like biking on gravel roads in the montains, encountering lots of detours and ... wasting lots of money (it should **not** be like this!)]

What is the standard software process?

# Software Process Models (2/2)

Process Standardisation through Artefact Standards and Process Completeness Percentages ([2])
(per Artefact release state: draft% - validated% - final&maintained%)

| | ***Artefact*** | ***Process Completeness Percentage*** |
|---|---|---|
| 1 | Requirements (=Analysis) | 10% - 20% - 35% |
| 2 | Design | 4% - 9% - 11% |
| 3 | Test-suites | 3% - 6% - 8% |
| 4 | Code | 10% - 25% - 35% |
| 5 | Documentation | 1% - 3% - 5% |
| 6 | Installation/Acceptance | 2% - 4% - 6% |
| ... | [optional/repeated artefacts] | [additional %] |

+ supporting artefacts:

a) Project Management results (plans, reports, ...) - 4% - 6% - 10%

b) Quality Management results (risk analysis, quality plan, ...) - 3% - 5% - 8%

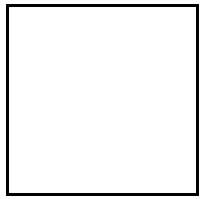
# Software Models (Meta Models) (1/4)

[Nobody agrees on what systems or system models are...]

When we want to measure a system (model), the first question is:
**What is a system (model)?**

# Software Models (Meta Models) (2/4)

*Layers of Software Product and Process (Software-Life-Cycle):*



*Most relevant for estimation:*
Analysis models = Requirements models

# Software Models (Meta Models) (3/4)

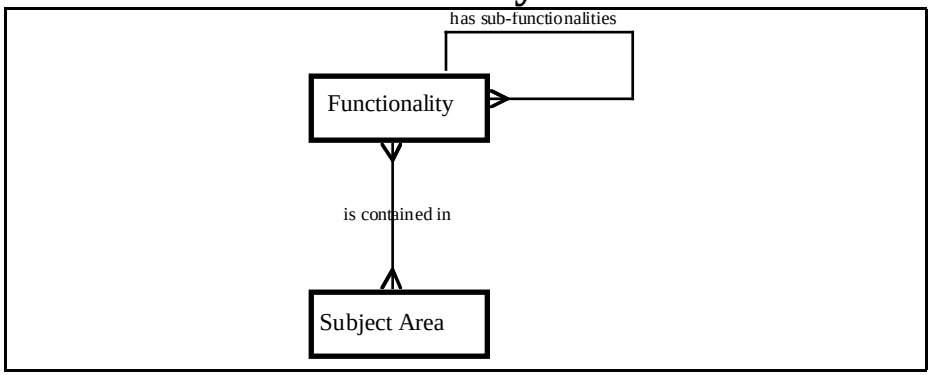
*Further sub-layering of the analysis layer*

(1) Preliminary Analysis = coarse real-world system modelling

(2) Domain/Business Analysis = detailed real-world system modelling

(3) Application Analysis = user view of the computer system (=user manual)

## Preliminary Model:




**Functionality** "name" [ complexity number ] **.**
**Functionality** "name" = { 'sub-functionality', } **.**
**Subject Area** "name" number of classes **.**
**Subject Area** 'name' **contains** 'functionality' **, ... .**

# Software Models (Meta Models) (4/4)

## A Domain Model - Metamodel (compliant with most standards [4], [5]):

### *(1) Domain Analysis Class Model*

**Domain Class** <ddd> { **isSubTypeOf** <base-class> } { **contains** <n> **attr** < class model object>} **.**
**Domain Association** <assoc-name> **one**|**many** <class1> **to one**|**many** <class2> **.**
**Function Type** <ddd> [ **ofKind** <parameter-name> **,** ... ] **.**
**Consistency Rule** <rrr> **=** < class model object > **,** ... **.**

### *(2) Use Case Model*

**Use Case** <rrr> **isTriggeredBy** <event/time indication> [ **=** <class model object > **,** ... ] **.**
**Signal** [ <sss> ] **of** <use case / function type> [ from|to <actor> ] = <class model object > **,** ... **.**
**Domain Subsystem** <dss> = <use case> **,** ... **.**

### *(3) State Transition Model*

**State** <st> **isSubStateOf** <state/class> [= < class model object > **,** ... ] **.**
**Transition** <tr> **startsAt** <state1> **endsAt** <state2> [= < class model object > **,** ... ] {**triggers**|**isTriggeredBy** <signal>}**.**

# Software Metrics (1/2)

## Function Point (Allan J. Albrecht, IFPUG [3]):

(1) Classifying the domain classes into easy-medium-complex and giving „points“
(2) Analogue procedure for rating the persistency-accesses to classes in the use cases
(3) Sum of all points = "unadjusted" Function Points
(4) Rating of 10 influence factors with percent point
(5) Adjustment (70%-130%) of the "unadjusted" Function Points = "adjusted" Function Points

*Advantages:*

- understandable / „intuitive“
- useful for database applications

*Disadvantages:*

- restricted to database applications
- requires the business model (modelling effort high)
- does not take reuse into account
- needs expert assessment (no fully automatable measurement)
- formally unsound

# Software Metrics (2/2)

## New approach: System Meter (Moser, 1995 cited in [6])

(1) External complexity of a single model entity:
= #new tokens in the name (+ 1, if old tokens are contained in the name)
= 1, if object is anonymous
[z.B. "theCurrentWindow" = 3; "theNewWindow" = 2; "theNewCurrentWindow" = 1]

(2) Internal complexity of a single model entity
= Sum of the external complexity of those other model entities that define the one in focus
[z.B. a class is defined through its super-classes and „members“,
a method through its parameters and implementing „messages“]

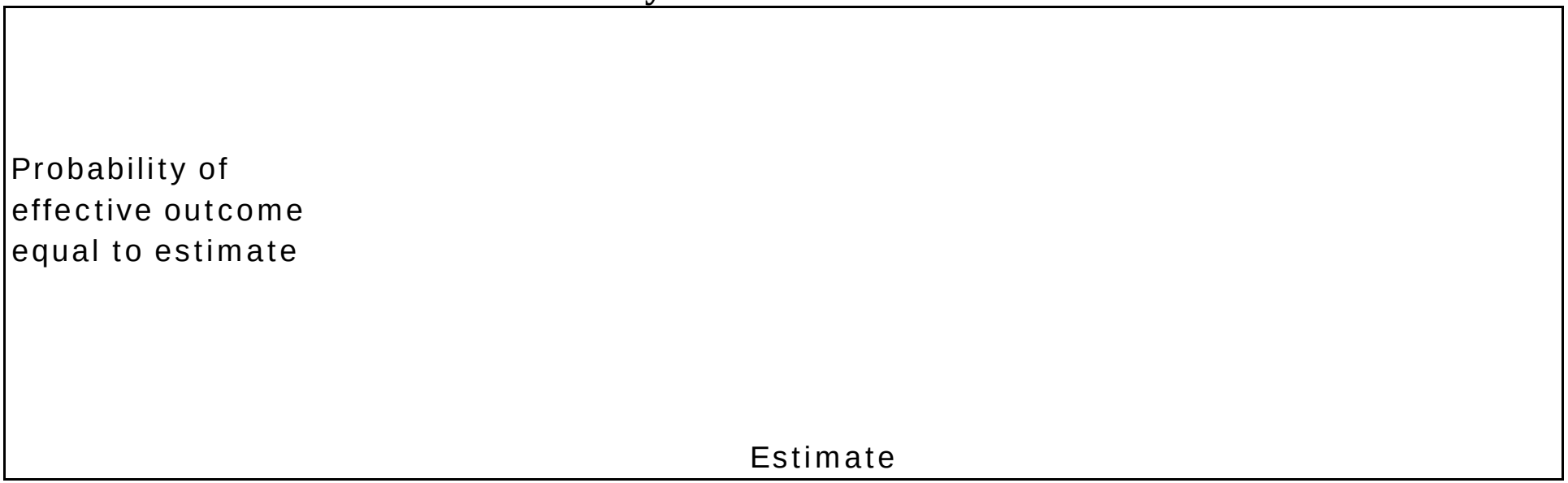
(3) Sum up all complexities (just the external complexity for reusable objects)

*Advantages:*

- generic (also non-persistency features are counted)
- takes reuse into account
- can be applied on preliminary models, business models as well as code
- measurement is fully automated and objective

# Results of a Field Study (1/4)

## Main analysis: Effort estimation bias

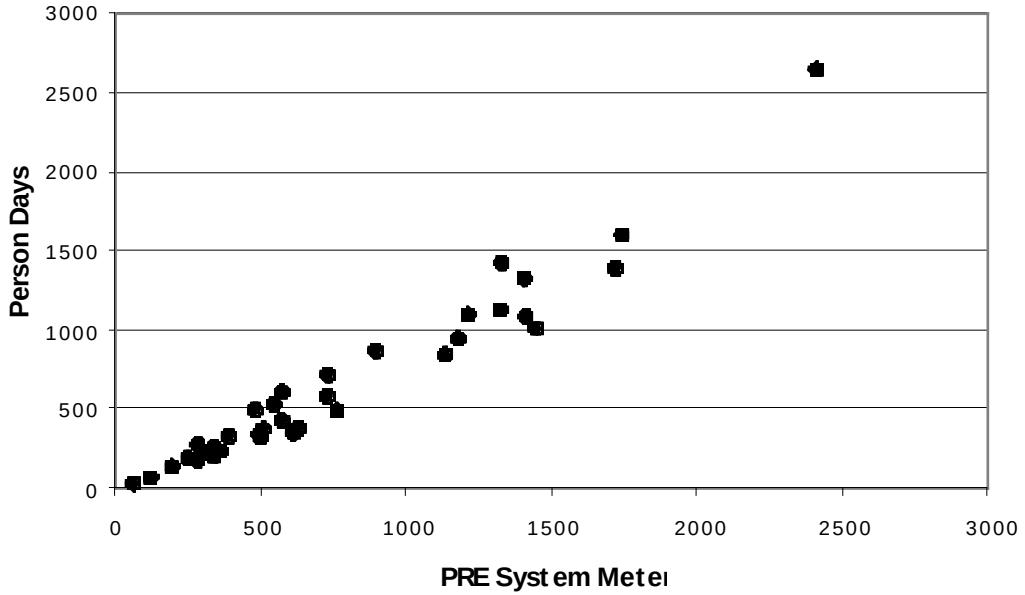



*36 Projects:*

- 33 industry projects (6 companies); 3 universitary projects
- time span: completion date mainly in 1994/95)
- C++: 4  4GL: 6  Smalltalk: 26
- client/server database-applications: 29

# Results of a Field Study (2/4)

*PRE System Meter*



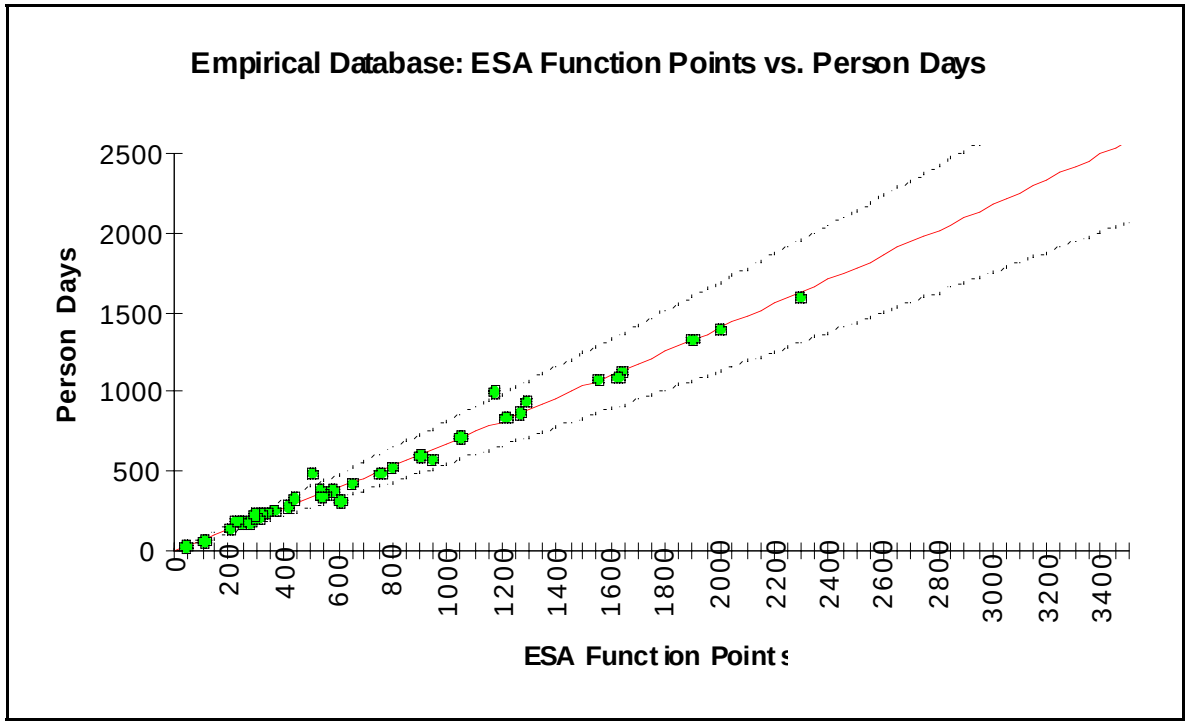

*PRE-SM Survey Results: $A = 0.605 \cdot s + 0.0001779 \cdot s^2$ , $dA = \pm 33\%$*

*Additional analysis (compared to Function Points):*

- Better adjustment for reuse
- Better correlation in the 7 non-IS projects

# Results of a Field Study (3/4)

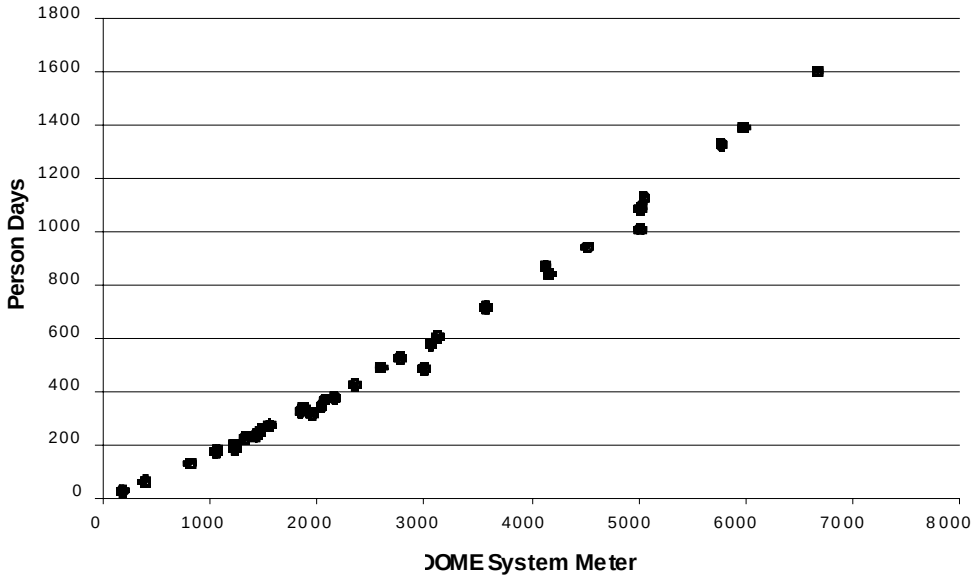
*Function Points:*




*FPM survey results: $A = 0.656 \cdot s + 0.000235 \cdot s^2$ , $dA = \pm 20\%$*

# Results of a Field Study (4/4)

*DOME System Meter:*




*DOME-SM Survey Results: $A = 0.151 \cdot s + 0.0000126 \cdot s^2$ , $dA = \pm 9\%$*

*Additional analysis (Wilcoxon-signed-ranksum-test):*

- The correlation improvement over FP is significant

# An Example (1/5)

Prerequisite for Step A: a System Model ...

```
; 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’ .
```

... eventually refined with information about reuse:

```
...
;ma-entry: category library
Functionality “Manage Objects“ 4 .
;ma-entry: category project
...
```

# An Example (2/5)

Step A: measuring the System Model

Measurement should be algorithmic ⇒ use an automated tool (download, e.g., `SEBT`, the software estimation basic toolkit from ftp://ftp.csse.swin.edu.au/outgoing/simonm/sebt.zip, use the unzipper ftp://ftp.csse.swin.edu.au/outgoing/simonm/pkunzip.exe)

Measurement is then as simple as typing some (DOS) command ...

```
ma -v -f tois.sdf
```

... and watch the result to plop out:

```
System Meters = 563
```

**November 1999: New tool with GUI:** http://www.softengprod.com

# An Example (3/5)

Prerequisite for Step B: setting-up or obtaining an Empirical Database
= measuring predictor and result values of completed projects

Use the databases (edb_pre.xls, edb_dome.xls) contained in SEBT (37 projects)

Step B: using the Empirical Database

*PRE-SM Survey Results:* $A = s \cdot 0.605 + s^2 \cdot 0.0001779$ , $dA = \pm 33\%$

$$563 \times 0.605 + 563\_ \times 0.0001779 = 340.6 + 56.4 = 397 \text{ PD}$$

# An Example (4/5)

## Prerequisite for Step C: a Software Process Model (**XX%** = standard, * = repeatable)

| BIO Layer | Result | Exp. % | Evol. % | Full % | BIO Layer | Result | Exp. % | Evol. % | Full % |
|---|---|---|---|---|---|---|---|---|---|
| **Preliminary Analysis 5%** | Subject Areas | 1/2 % | 1 % | **2 %** | | *[replication cont.]* | | | |
| | Goals | 1/2 % | 1 % | **3 %** | | User Manual / Online Help * | 1 % | 3 % | 4 % |
| **Domain Analysis 14%** | Use Case Model | 1 % | 3 % | **5 %** | | Forms (for manual processes) | 1/2 % | 1/2 % | 1 % |
| | Domain Class Model | 1 % | 3 % | **5 %** | **Delivery 6%** | Acceptance | 1/2 % | 2 % | **3 %** |
| | State-Transition Models | 1 % | 2 % | 4 % | | Installations * | 1/4 % | 1 % | **1 %** |
| | Non-essential Requirements | 1 % | 2 % | **4 %** | | User Instruction * | 1% | 1 1/2% | **2 %** |
| **Application Analysis 18%** | Specification Types * | 1 % | 3 % | **5 %** | | Organisational Changes | 1/2 % | 1 1/2 % | 2 % |
| | Models | 2 % | 3 % | **5 %** | | Data Migration | 2 % | 3 % | 4 % |
| | System States | 2 % | 3 % | 4 % | **Project Management 10%** | Plans | 1 % | 1 1/2 % | **2 %** |
| | Application Class Model | 2 % | 4 % | **6 %** | | Estimates | 1/2 % | 1 % | **1 %** |
| | Non-functional Requirements | 1/2 % | 1 % | **2 %** | | Configuration Mgmt | 2 % | 2 % | **3 %** |
| **Construc-tion 19%** | Implementation Patterns * | 2 % | 4 % | **5 %** | | Problem and Change Mgmt | 1 % | 2 % | **3 %** |
| | Relational Model | 1 % | 3 % | **4 %** | | Controlling and Reporting | 1 % | 1 % | **1 %** |
| | Technical Class Model | 1 % | 2 % | **2 %** | | Evaluations * | 1/2 % | 2 % | 3 % |
| | Test Data | 1 % | 3 % | **4 %** | | Prep. Organisational Changes | 1/2 % | 2 % | 3 % |
| | Test Cases | 2 % | 3 % | **4 %** | | Prep. User Instructions | 1 % | 2 % | 3 % |
| **Replica-tion 38%** | Tuned Items * | 2 % | 4 % | 5 % | | Prep. Data Migration | 2 % | 2 % | 3 % |
| | Code | 10 % | 25 % | **30 %** | **Quality Management 8%** | Risk Analysis / Quality Plans | 1 % | 1 1/2 % | **2 %** |
| | Admin. & Installation Code | 1 % | 4 % | **5 %** | | Measurements | 1 % | 2 % | **3 %** |
| | Platform Port * | 2 % | 8 % | 10% | | Defining Standards | 1/2 % | 1 1/2 % | **3 %** |
| | Layout (GUI) Translation * | 4 % | 5 % | 5 % | | Developer Instruction | 1/4 % | 1/2 % | 1/2 % |
| | System Admin. Manual * | 1 % | 2 % | **3 %** | | Project Reviews | 1/4 % | 1/2 % | 1/2 % |

# An Example (5/5)

Step C: adapting the original estimate to the tailored process model

Example: we conduct a full application analysis and a prototypical construction focussing on 3 implementation patterns without formal test preparation:
= 18% + 3x2% + 1% + 1% = 26%

The resulting effort estimate therefore is:

$$397PD \times 26\% = 103PD$$

Additional adaptations:

- Optimum team sizes, maximising speed of development
- Reducing budget overrun risks by adding "buffer effort"

# Future Work

- The System Meter is used in industry

- Due to its formal properties the System Meter may be used in derived measures

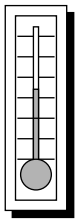
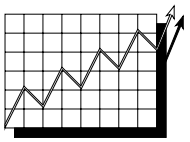
## *References:*

1. DeMarco T, Controlling SW Projects, Prentice-Hall, Englewood Cliffs, N.J., 1982
2. Moser S, Cherix R, Flueckiger J, HERMES/Bedag Informatik Vorgehensmodell, Bedag Informatik, Berne, Switzerland, 1993-1999
3. IFPUG, Counting Practices Manual V4.0, Westerville, Ohio, USA, 1996
4. Rumbaugh J, Jacobson I, Booch G, The Unified Modeling Language (UML) Ref. Manual, Addison-Wesley, Reading MA, 1999
5. Firesmith D, Henderson-Sellers B, Graham I, OPEN Modeling Language (OML) Ref. Manual, SIGS Books, NY, 1997
6. Moser S, Measurement and Estimation of Software and Software Processes, Ph.D. thesis, University of Berne, Berne, Switzerland, 1996
7. Henderson-Sellers B, Graham IM, Younessi H, The OPEN Process Specification, Addison-Wesley, UK, 1998

# 9. Metrics in OO Reengineering

**Outline**

- Why Metrics in OO Reengineering?
- Applicability for...
  - Quality Assessment
  - Process Control
  - 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?

**Estimating Cost**

- Is it worthwhile to reengineer, or is it better to start from scratch?
  => See previous lectures

**Assessing Software Quality**

- Which components have poor quality? (Hence should be reengineered)
- Which components have good quality? (Hence should be reverse engineered)
  => Metrics as a reengineering tool!

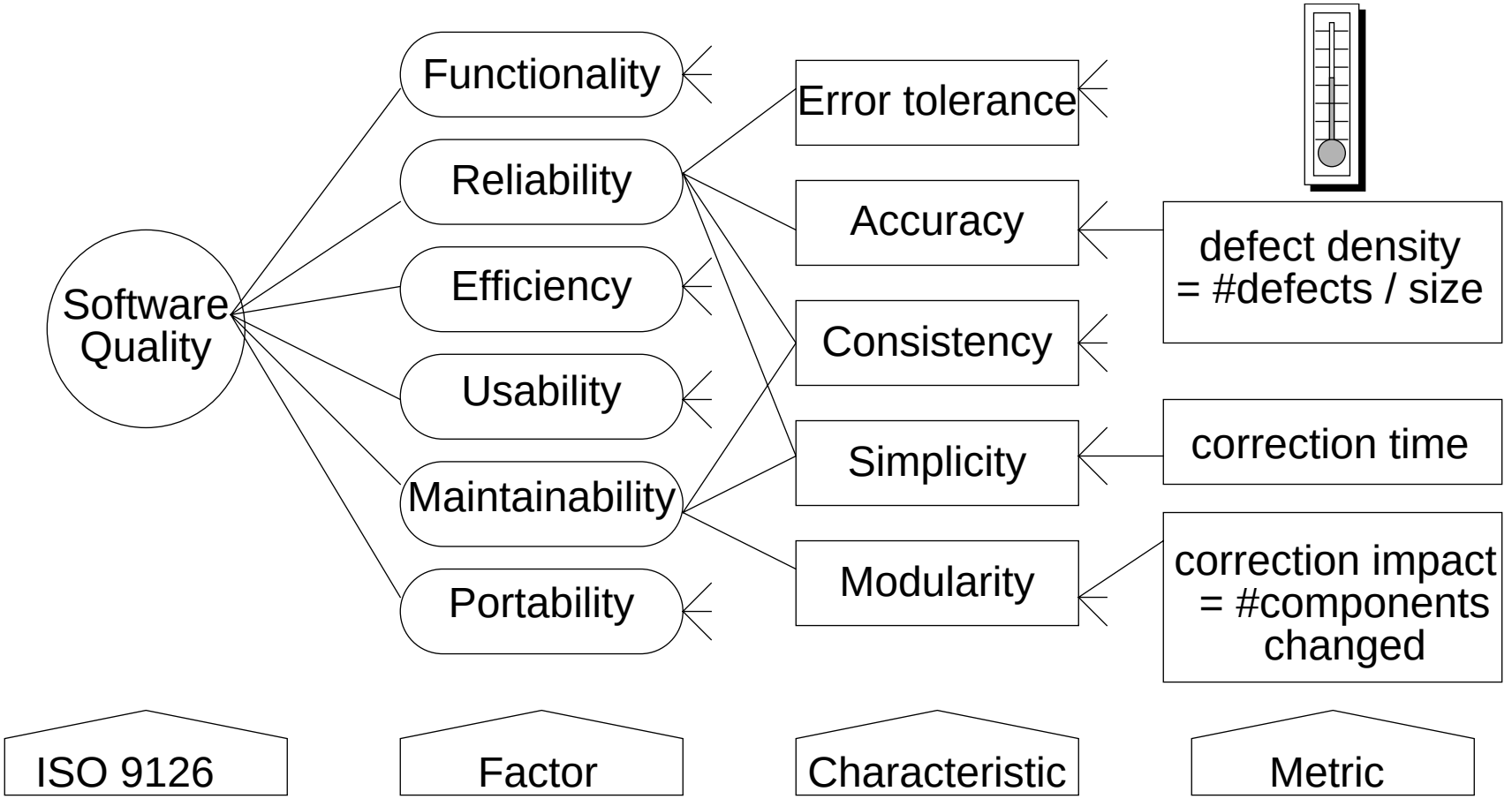
**Controlling the Reengineering Process**

- Trend analysis: which components did change?
- Which refactorings have been applied?
  => Metrics as a reverse engineering tool!

# Quantitative Quality Model

**Quality according to ISO 9126 standard**

- Divide-and conquer approach via “hierarchical quality model”
- Leaves are simple metrics, measuring basic attributes



# Process Attributes & External Attributes

**Process Attribute**

- **Definition**: measure aspects of the process which produces a product
- example: time to correct defect, number of components changed per correction

**Product Attribute**

- **Definition**: measure aspects of artifacts delivered to the customer

**External Product Attribute**

- **Definition**: measures how the product behaves in its environment
- example: number of system defects perceived, time to learn the system

**Pros and Cons**

- advantages:
  - close relationship with quality factors
- disadvantages:
  - measure only after the product is used or process took place
  - data collection is difficult often involves human intervention/interpretation
  - relating external effect to internal cause is difficult

# Internal Product Attributes

**Internal Product Attribute**

- **Definition**: is measured purely in term of the product, separate from behaviour
- example: method size, class coupling and cohesion

**Quality Assumption**

☞ Internal product attributes directly affect quality

**Pros and Cons**

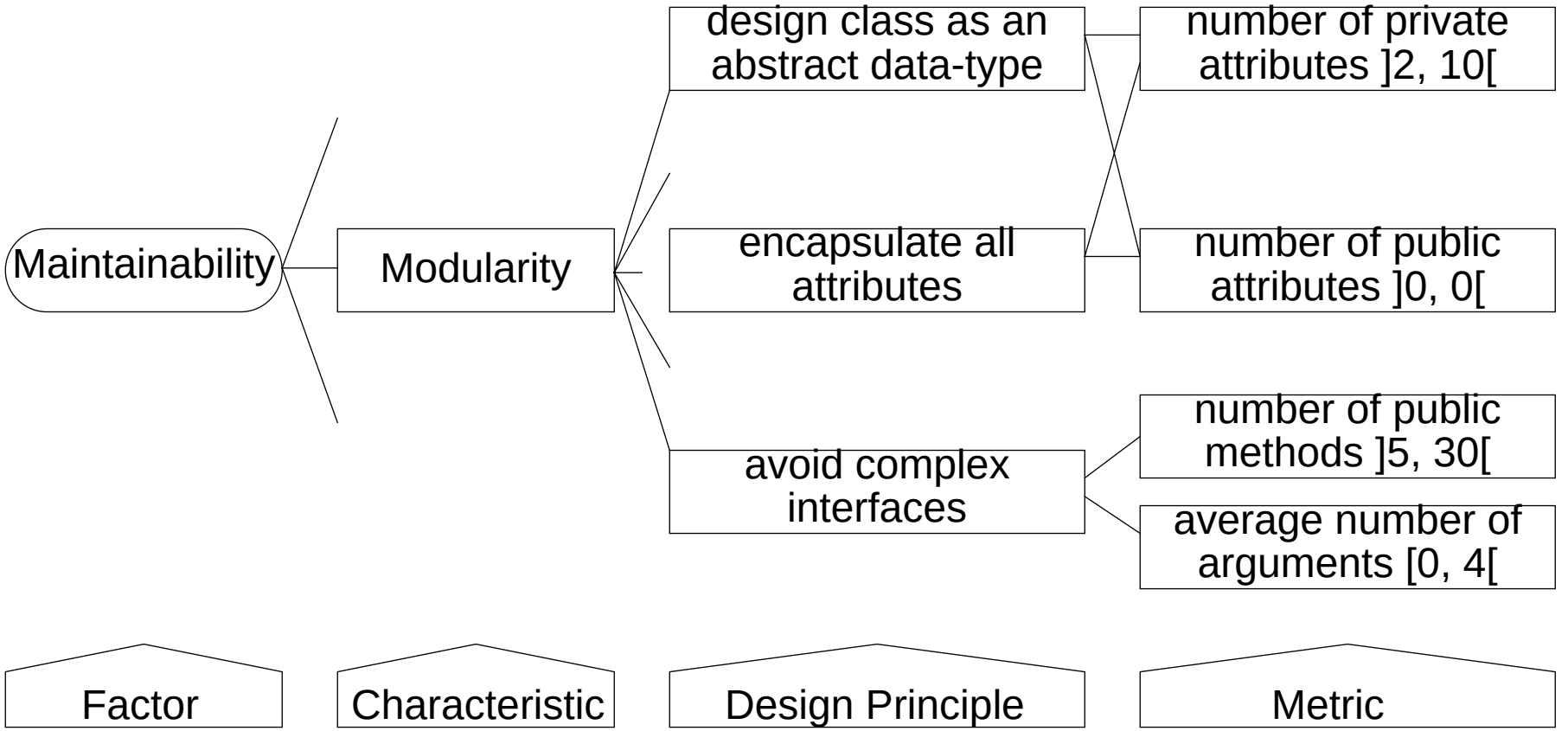
- advantages:
  - can be measured at any time
  - data collection is quite easy and can be automated
  - direct relationship between measured attribute and cause
- disadvantage:
  - relationship with quality factors is not empirically validated

☞ measurements may only be used as indicators, i.e. a heuristic

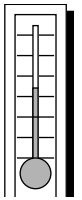
# *"Define your own" Quality Model*

**Define the quality model with the development team**

- Team chooses the characteristics, design principles, metrics...
- ... and the thresholds



# Conclusion: Metrics for Quality Assessment



**Question:**

- Can internal product metrics reveal which components have good/ poor quality?

**Yes, but...**

- Not reliable
  - false positives: “bad” measurements, yet good quality
  - false negatives: “good” measurements, yet poor quality
- Heavy Weighth Approach
  - Requires team to develop (customize?) a quantitative quality model
  - Requires definition of thresholds (trial and error)
- Difficult to interpret
  - Requires complex combinations of simple metrics

**However...**

- Cheap once you have the quality model and the thresholds
- Good focus (± 20% of components are selected for further inspection)
  Note: focus on the most complex components first!

# *The KISS principle*

**K**eep
**I**t
**S**tupidly
**S**imple

**Question**

- Wouldn't there lightweight approaches to exploit metrics during reengineering?

# Trend Analysis via Change Metrics

**Change Metric**

- **Definition**: difference between two metric values for the same metric and the same component in two subsequent releases of the software system
- Examples:
  - difference between number of methods for class “Event” in release 1.0 and 1.1
  - difference between lines of code for method “Event::process()” in release 1.0 and 1.1

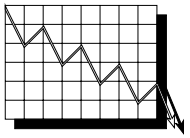
**Change Assumption**

- ☞ Changes in metric values indicate changes in the system

# Conclusion: Metrics for Trend Analysis

**Question:**

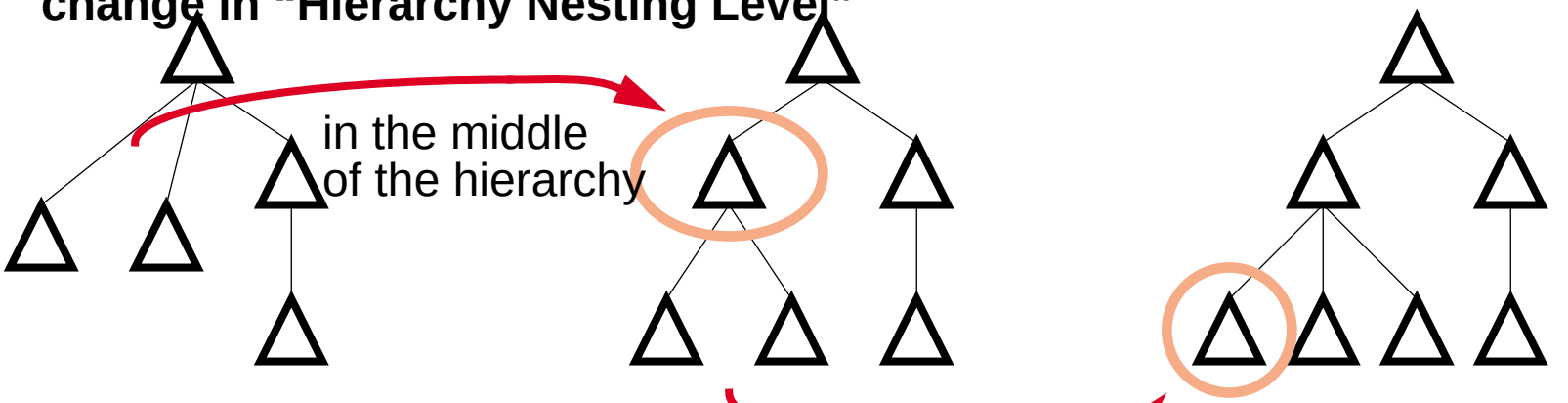
- Can internal product metrics reveal which components have been changed?


- changes may go unnoticed
  => false negatives are possible
- all detected changes are real
  => no false positives (but lot of noise)

Sometimes the kind of changes are revealing!

**change in "Hierarchy Nesting Level"**




**change in "Number of Children"** at the leaf of the hierarchy

# Identifying Refactorings via Change Metrics

**Refactorings Assumption**

- ☞ Decreases (or Increases) in metric values indicate movement of functionality

**Basic Principle of "Identify Refactorings" Heuristics**

- ❑ Use one change metric as an indicator (1)
- ❑ Complement with other metrics to make the analysis more precise
- ❑ Include other metrics for quicker assessment of the situation before and after

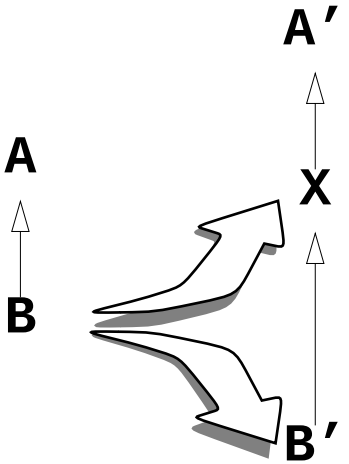
(1) Most often we look for decreases in size, as most refactorings redistribute functionality by splitting components.

# Split into Superclass / Merge with Superclass

**Recipe**

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

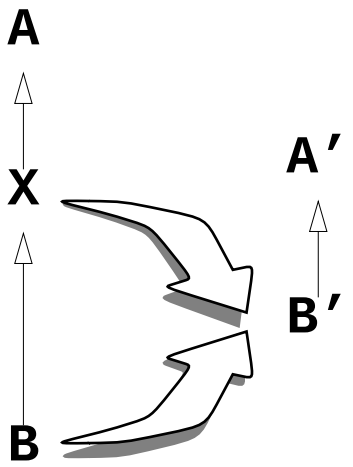
**SPLIT**




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))

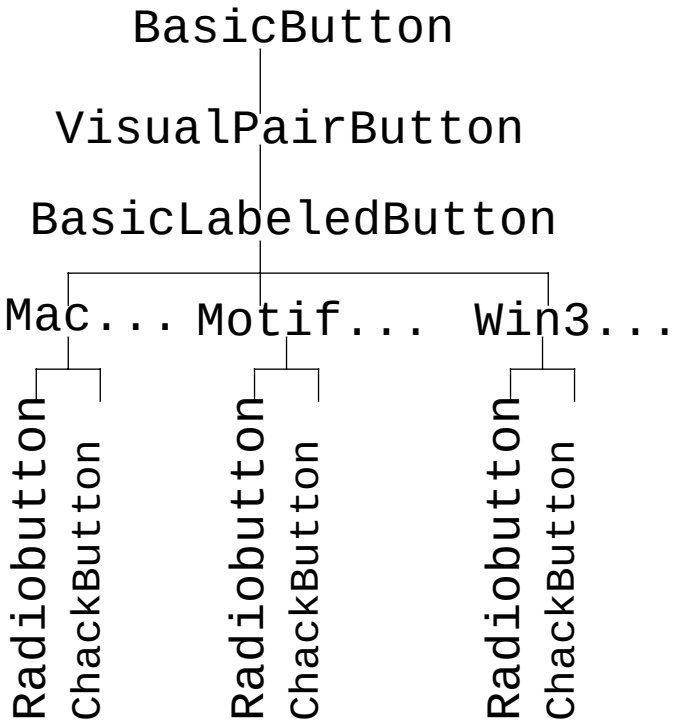
Merge X and B into B’
(delta_HNL(B’) < 0) and
( (delta_NOM(B’) >0)
or (delta_NIA(B’) > 0)
or (delta_NCA(B’) > 0))

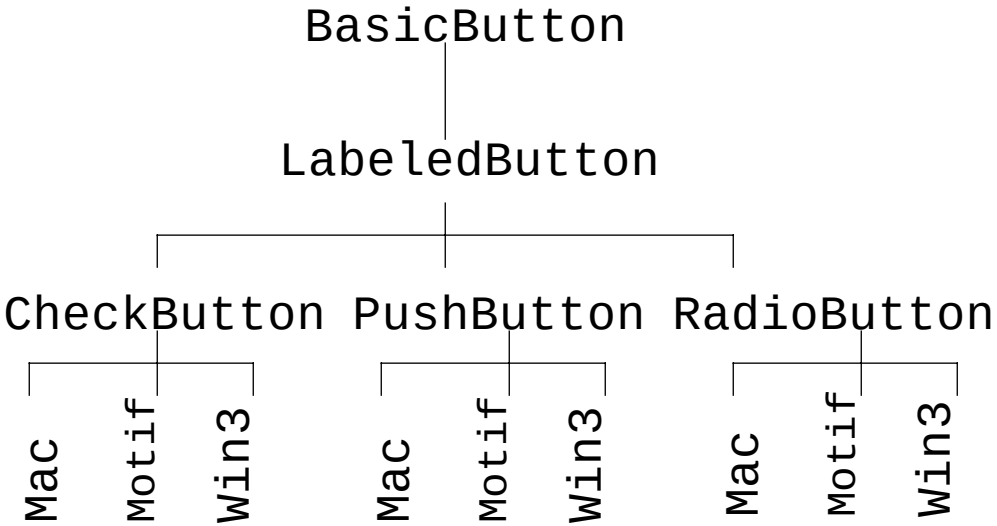
**MERGE**



# Example: Inferring the Bridge Protocol

In VisualWorks we detected a “Merge with Superclass” which revealed parts of the interaction protocol of the Bridge Pattern



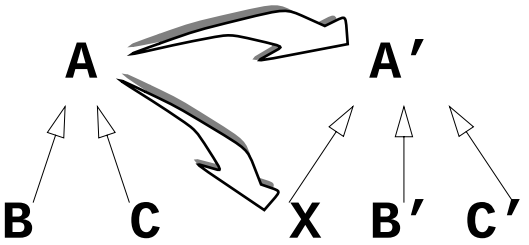



# Split into Subclass / Merge with Subclass

**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’

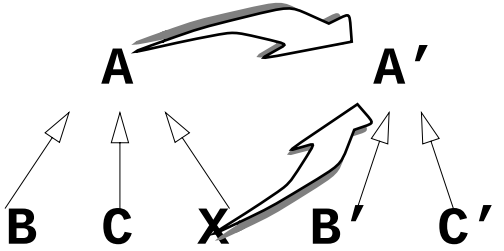
(delta_NOC(A’) <> 0) and

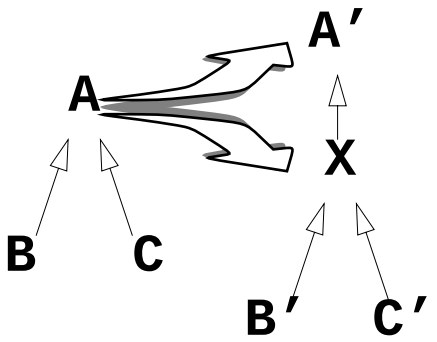
( (delta_NOM(A’) < 0)

or (delta_NIA(A’) < 0)

or (delta_NCA(A’) < 0))

**MERGE**



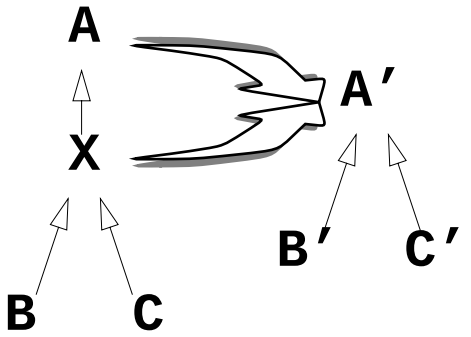

Merge X and A into A’

(delta_NOC(A’) <> 0) and

( (delta_NOM(A’) >0)

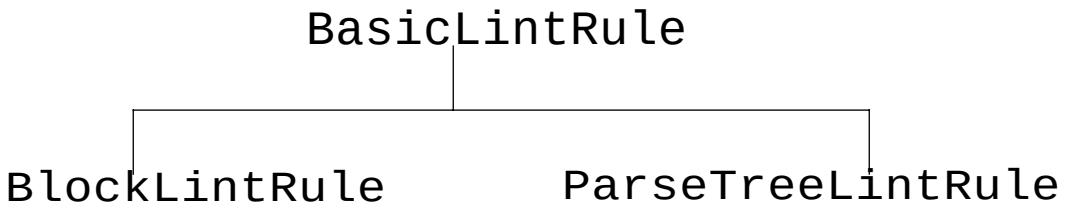
or (delta_NIA(A’) > 0)

or (delta_NCA(A’) > 0))



# Example: Adding new Functionality

In the Refactoring Browser we detected a “Split into Subclass” which enabled adding new functionality.



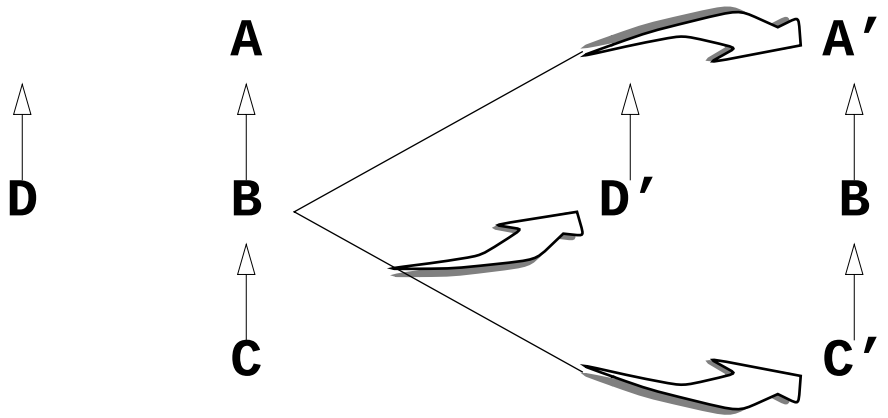

2 subclasses of BasicLintRule have been added.
2 attributes have been pushed down into BlockLintRule
70 methods have been redistributed across the two subclasses

# 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**

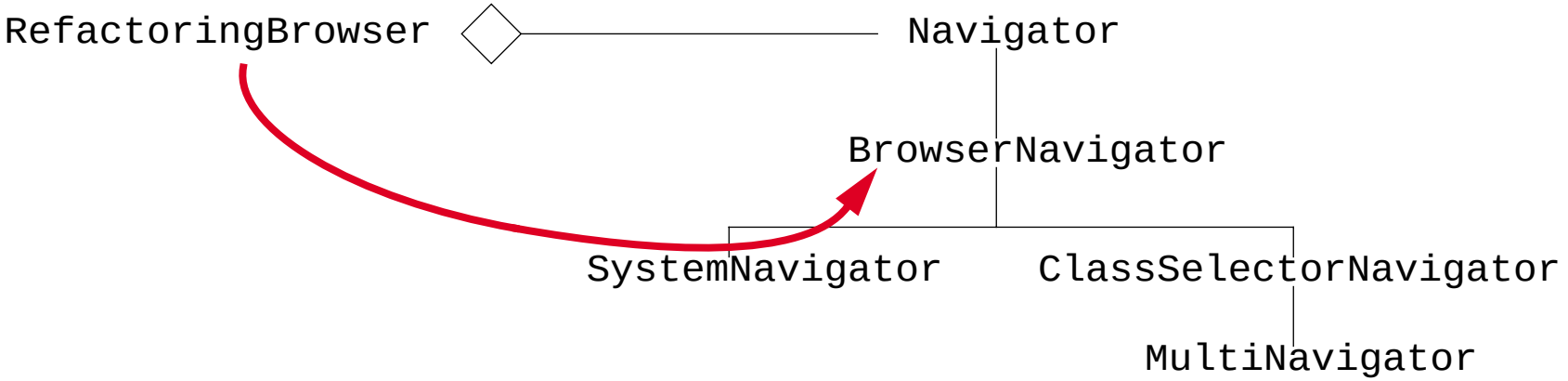



Move from B to A’, C’ or D’

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

# Example: Introducing Layers

In the Refactoring Browser, we detected a “Move to Sibling” introducing layers.



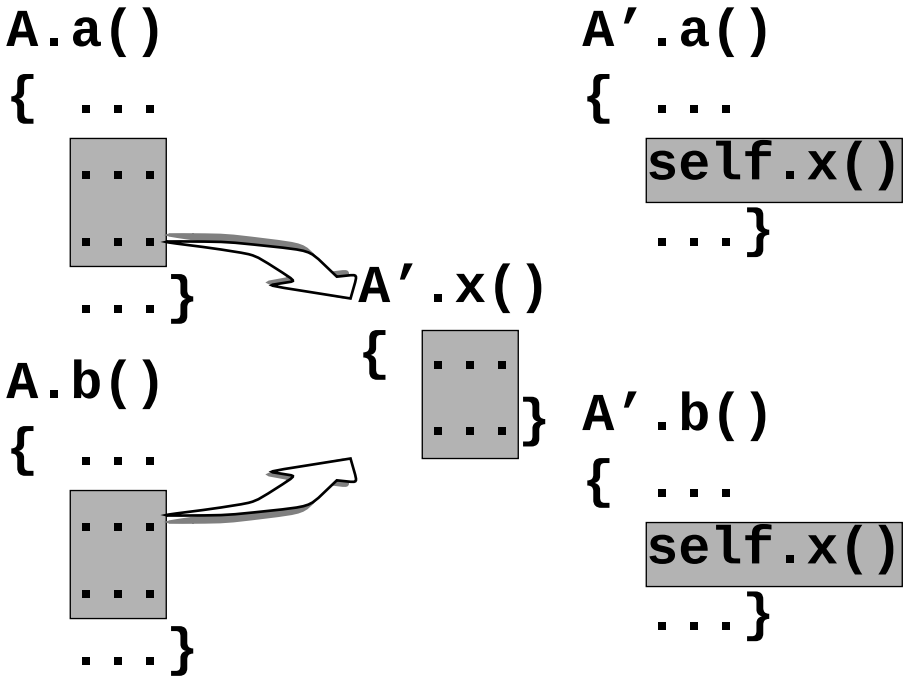

+- 50 methods have been moved.

Result: Methods in navigator do not call any more on their aggregate.

# Split Method / Factor Common Functionality

**Recipe**

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




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

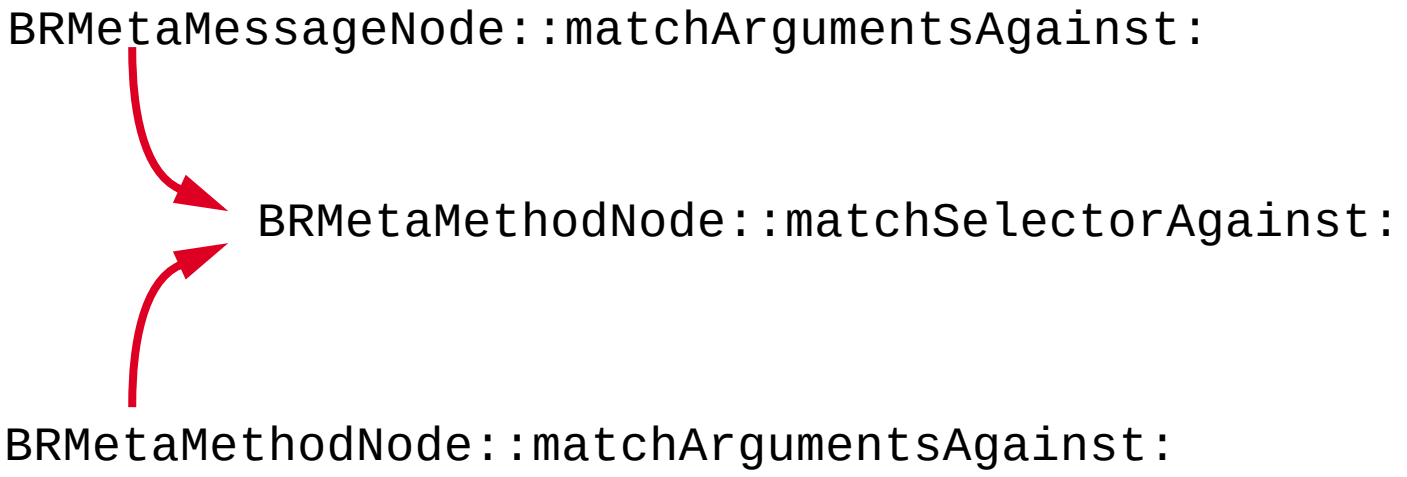
(delta_NOI(A'.a()) < 0)

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

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

# Example: Creation of Template Method

In the Refactoring Browser we detected a “Split Method” which corresponded with the introduction of a template method.



# Conclusion: Identifying Refactorings

**Question:**

Can internal product metrics reveal which refactorings have been applied?

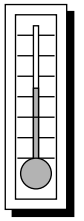
- vulnerable to renaming
- imprecise for many changes
- requires experience
- considerable resources

=> inherent to reverse engineering based on source code

- good focus (scaleability)
- reliable
- reveals class interaction
- unbiased

=> good in the early stages

# Conclusion



**Question**

Can metrics [1] help to answer the following questions?

1. Which components have good/poor quality? *Not reliably*
2. Which components did change? *Yes*
3. Which refactorings have been applied? *Yes*

[1] Metrics = Measure internal product attributes (i.e., size, inheritance, coupling, cohesion,...)

# Questions

**You should know the answers to these questions.**

- What's the difference between an internal and an external product attribute? What's the difference between a product and a process attribute?
- Why is it preferable to use internal product attributes instead of process attributes or external product attributes?
- Why is it possible to have false negatives for change metrics?
- Why do we state for "Move to Superclass, Subclass or Sibling Class on page 196" that you should select only those cases where "# immediate children" (NOC) and "Hierarchy Nesting Level" (HNL) remain equal?

**Can you answer the following questions?**

- Is the quality assumption (i.e., Internal product attributes directly affect quality) reasonable? Find both arguments for and against.
- When would you apply a quantitative quality model in a reengineering project?
- If you are looking for refactorings, why is it better to look for decreases in size?
- Why do you think that change metrics are vulnerable to renaming?

# 10. Tool Integration

**Outline**

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

**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 Integrate Tools?

**Tool Adage**

Tools are necessary to improve productivity.

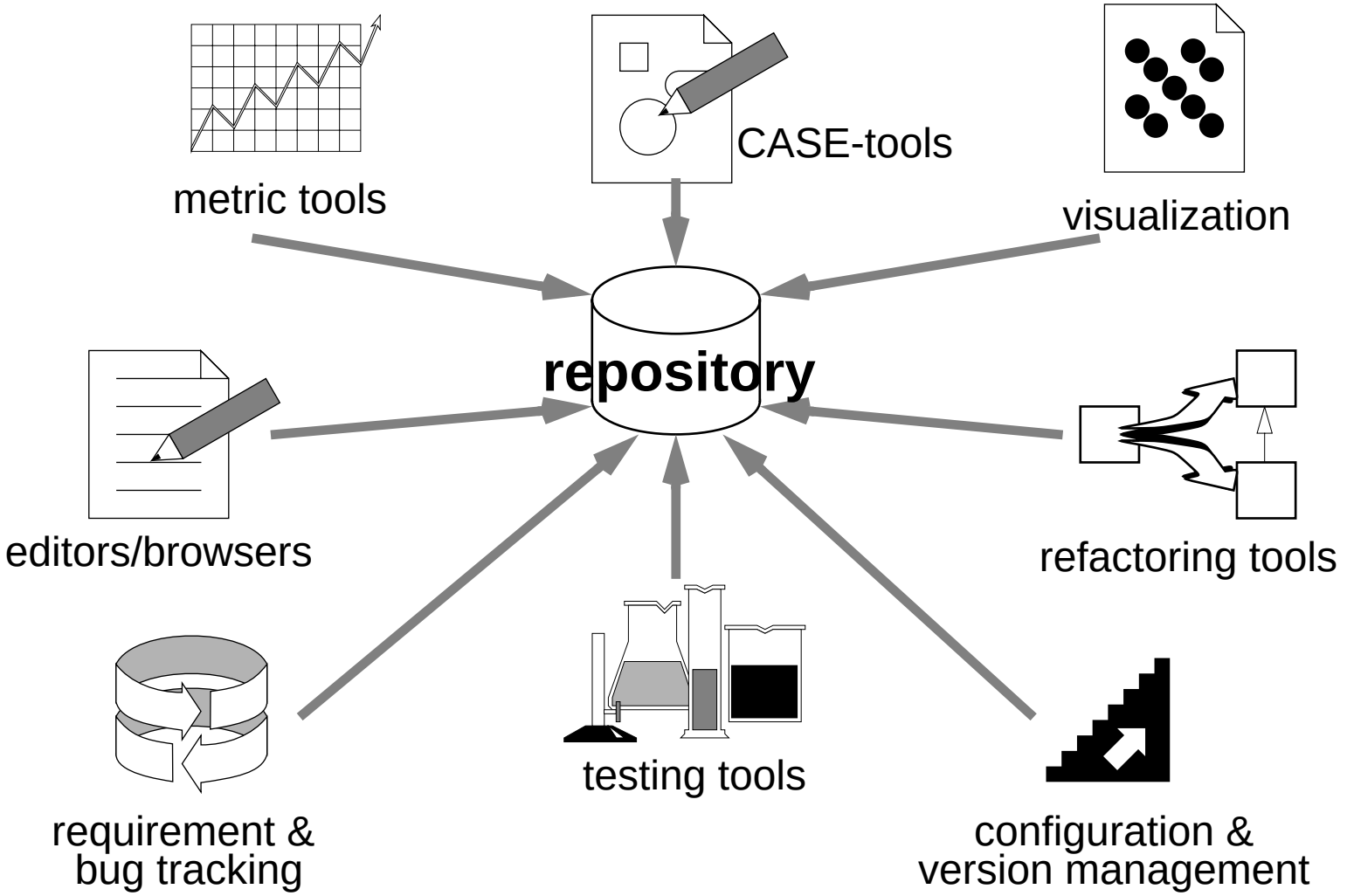
**Tool Principle**

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

**Towards CARE**

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

# Which Tools to Integrate?



# Tool Integration Issues

**Reengineering vs. forward engineering**

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

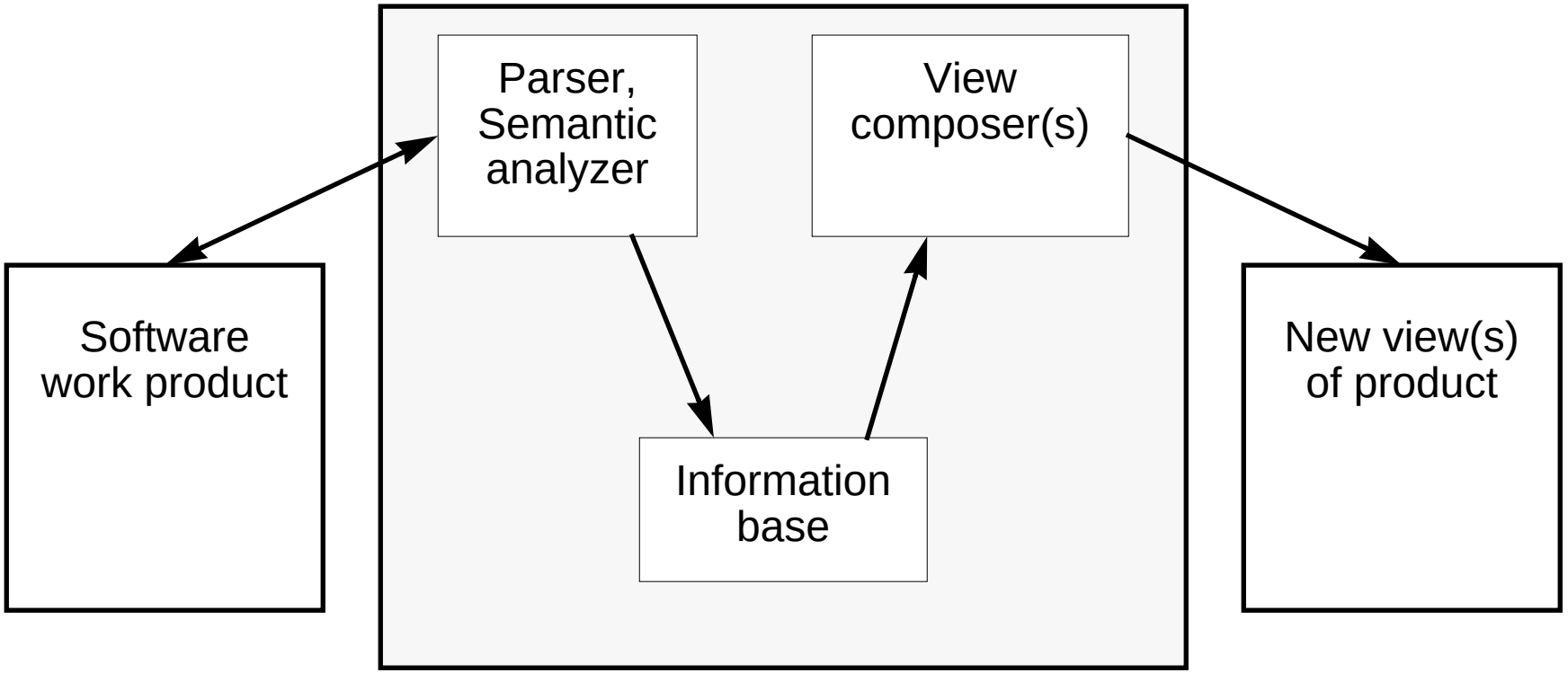
  - ☞ Tool integration in reengineering is harder
    ... but we can rely on forward engineering experience
  - ☞ "Help yourself" approach

**Tools must work together**

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

# Basic Tool Architecture

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



# Help Yourself - Parser

**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

# Help Yourself - File Formats

**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

# Help Yourself - API

**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"

# Help Yourself - Execution Trace

**Collect Execution Traces**

- Technique
  - Acquire traces of sequences of method invocations
    (code instrumentation, method wrapping, debugger, 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

# API Example - Java

A piece of Java-code using the reflection facilities to inspect class elements

```
import java.lang.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());
   }
}
```

# API Example - SNiFF+

A piece of C-code which accesses the SNiFF+ API to query the symbol table

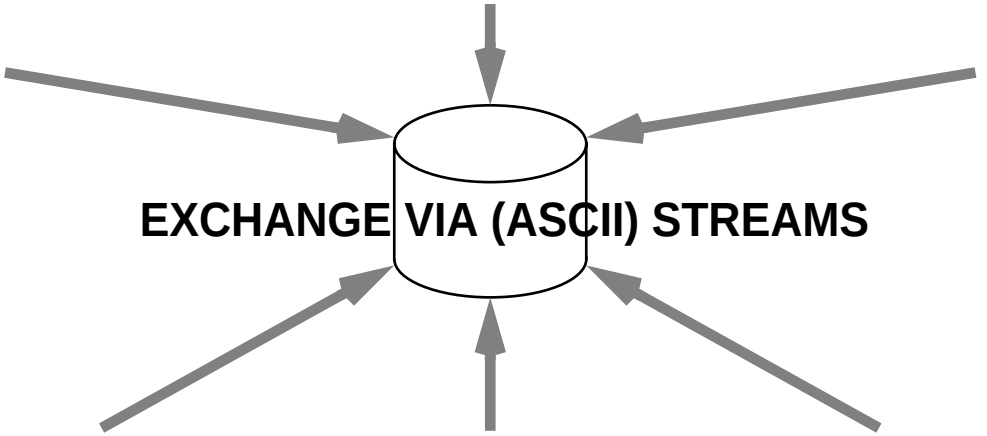
```
int main ( int argc, char *argv[] )
{ SNiFFACCESS slot;
  .... /*other declarations */

  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;
}
```

# API Example - Rational/Rose

Pieces of VisualBasic-code to **generate** elements into the Rational/Rose repository

```
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
```

# Exchange Standards



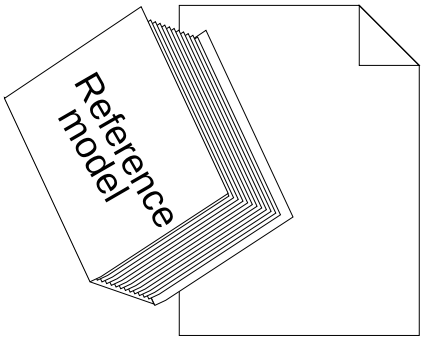

Standardization Efforts

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

# Exchange Standards - Reference Format

- **Issue**

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




- **Answer**

  Tools agree on a single reference model

  **reference model = meta model**

- **Analogy**

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

- **Advantage**

  Only need for one translation dictionary

- **Disadvantage**

  Centralised reference models do not work in practice

  - Need for specialised constructs (i.e. jargon)
  - Cannot predict future specializations

# Exchange Standards - Openness

## Specialised Constructs

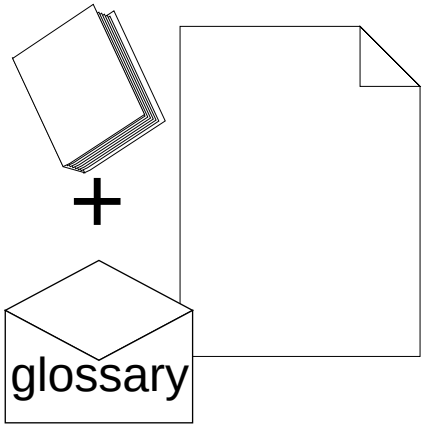
- **Issue**

  How can tools extend the meta model with specialised constructs?

- **Answer**

  Each tool includes an extra glossary, explaining the specialised constructs in terms of a core reference model.

  **core reference model = meta meta model**



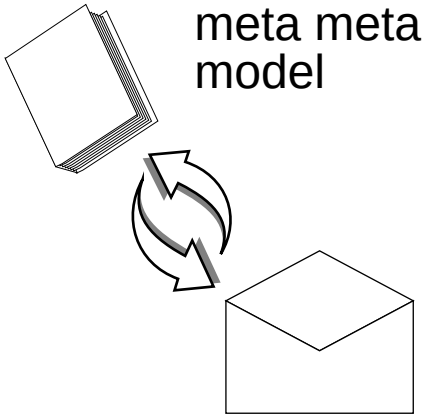

## Multiple Standards

- **Issue**

  How can tools deal with future extensions?

- **Answer**

  All glossaries (=meta model extensions) define mapping with the core reference model (= meta meta model)



# Meta Models

Exchange standards community cultivated specialised terminology

☞ the Four Layer Metamodeling Architecture

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

# CDIF sample (propriety syntax)

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

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

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

Obligatory Introduction Stuff

Definition of a meta-model concept “Class” as having one attribute “name”

Definition of 2 classes “Student” & “Course”

# MOF Sample (XML syntax)

```
<?xml version="1.0" encoding=”ISO-8859-1”?>
<!DOCTYPE XMI SYSTEM "mof.dtd">
<XMI xmi.version="1.0">
  <XMI.header>
    <XMI.metamodel xmi.name="uml" xmi.version="1.1" />
  </XMI.header>
  <XMI.content>
    <MoF.Model.Package xmi.id="i00000001">
      <Mof.Model.ModelElement.name>package1</Mof.Model....>
      <Mof.Model.ModelElement.annotation/>
      <Mof.Model.GeneralizableElement.isRoot
          XMI.value="yes"/>
      ...
      <Mof.Model.Namespace.contents>
        <Mof.Model.Class xmi.id="i00000002">
          <Mof.Model.ModelElement.name>class1</Mof.....>
```

Obligatory Introduction Stuff

Load predefined UML meta model

Definition of a package with name “package1” and some attributes

This package contains class named “class1”

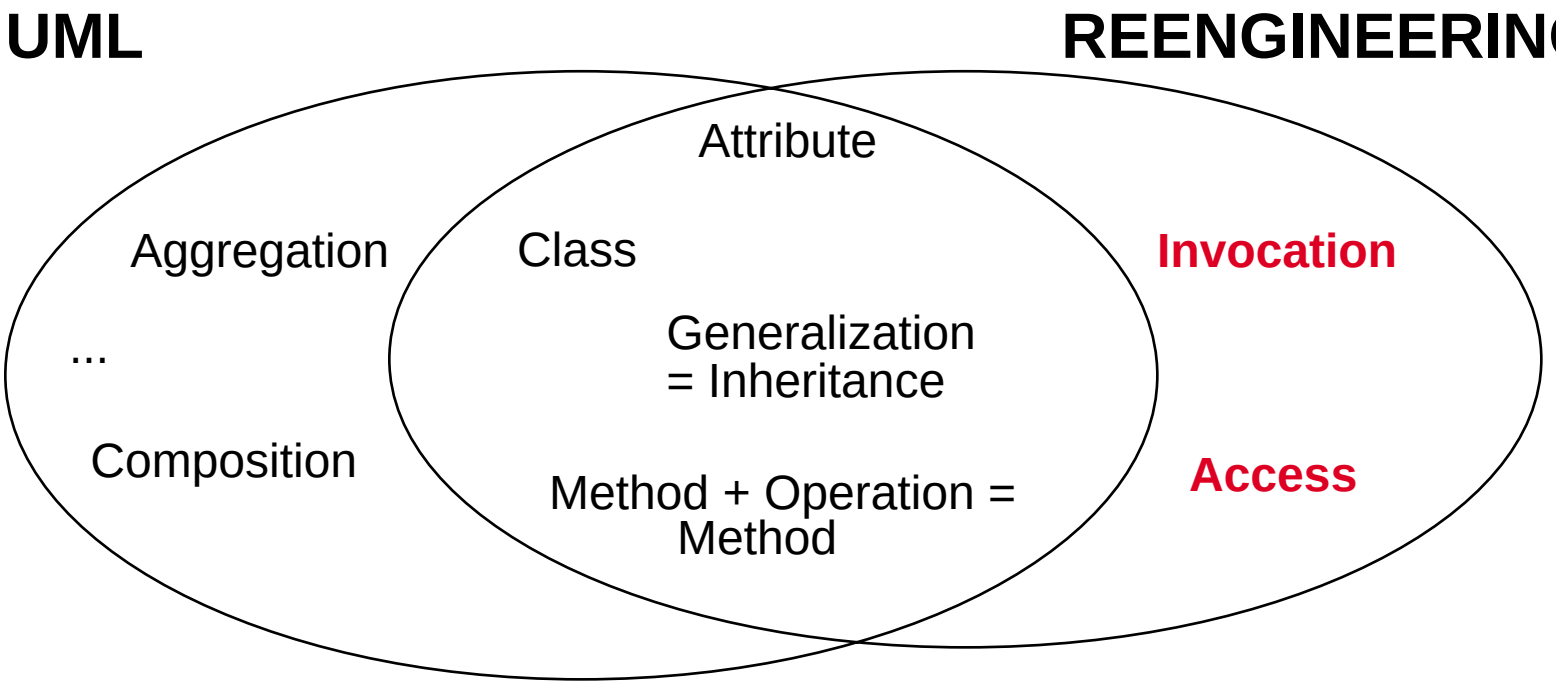
# CORBA Interface for MOF

```
interface MofAttributeClass : StructuralFeatureClass {
  readonly attribute
    MofAttributeUList all_of_kind_mof_attribute;
  readonly attribute
    MofAttributeUList all_of_type_mof_attribute;

MofAttribute create_mof_attribute (
  /* from ModelElement */ in ::Model::NameType name,
  ...
}; // end of interface MofAttributeClass

interface MofAttribute : MofAttributeClass, StructuralFeature
{
  boolean is_derived ()
    raises (Reflective::StructuralError,
      Reflective::SemanticError);
  void set_is_derived (in boolean new_value)
    raises (Reflective::SemanticError);
```

# UML shortcomings

Current standardization efforts are geared towards UML.

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




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

# Conclusion

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

# Questions

**You should know the answers to these questions.**

- What's the difference between tool integration in forward angineering and reengineering?
- If you need to build a tool that generates UML from Java source code, how would you conceive it ? Why ?
- Why do we need a meta meta model when exchanging information between tools?

**Can you answer the following questions?**

- How would you explain the "Four Layer Metamodeling Architecture"

# 11. Refactoring

**Outline**

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

# What is Refactoring?

**Some definitions**

- The process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure [Fowl99a]
- A behaviour-preserving source-to-source program transformation [Robe98a]
- A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality - simplicity, flexibility, understandability, performance [Beck99a]

**Typical Refactorings**

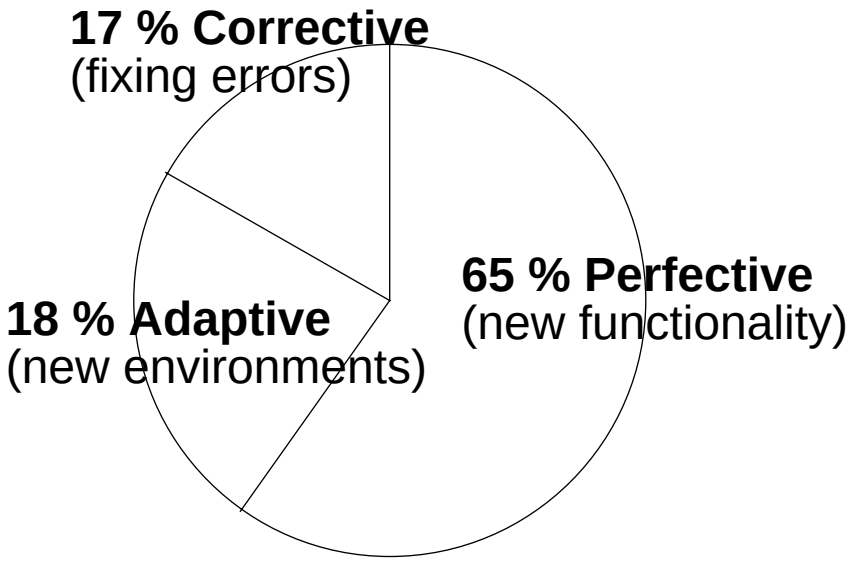
| Class Refactorings | Method Refactorings | Attribute Refactorings |
|---|---|---|
| add (sub)class to hierarchy | add method to class | add variable to class |
| rename class | rename method | rename variable |
| remove class | remove method | remove variable |
| | push method down | push variable down |
| | push method up | pull variable up |
| | move method to component | create accessors |

# Why Refactoring?

**Relative Effort of Maintenance**

[Somm96a]

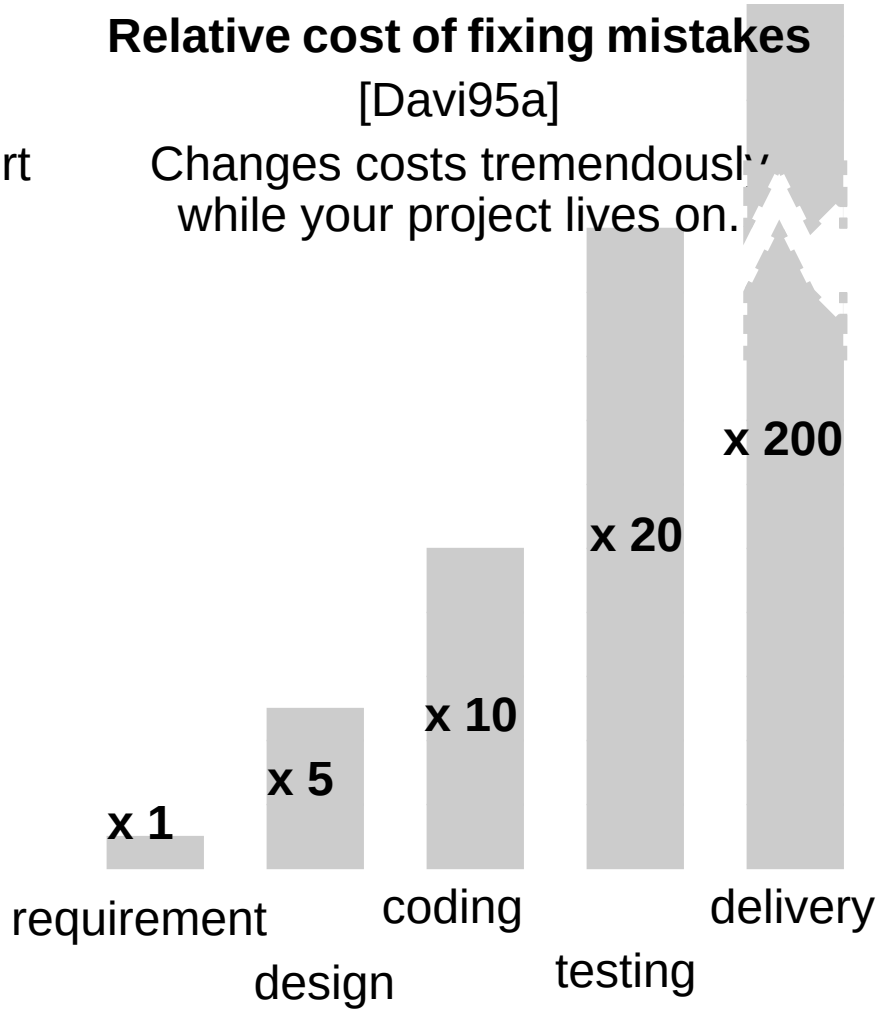
Between 50% and 75% of available effort is spent on maintenance. 65% of that concerns new functionality, which you could not foresee when you started.




**Relative cost of fixing mistakes**

[Davi95a]

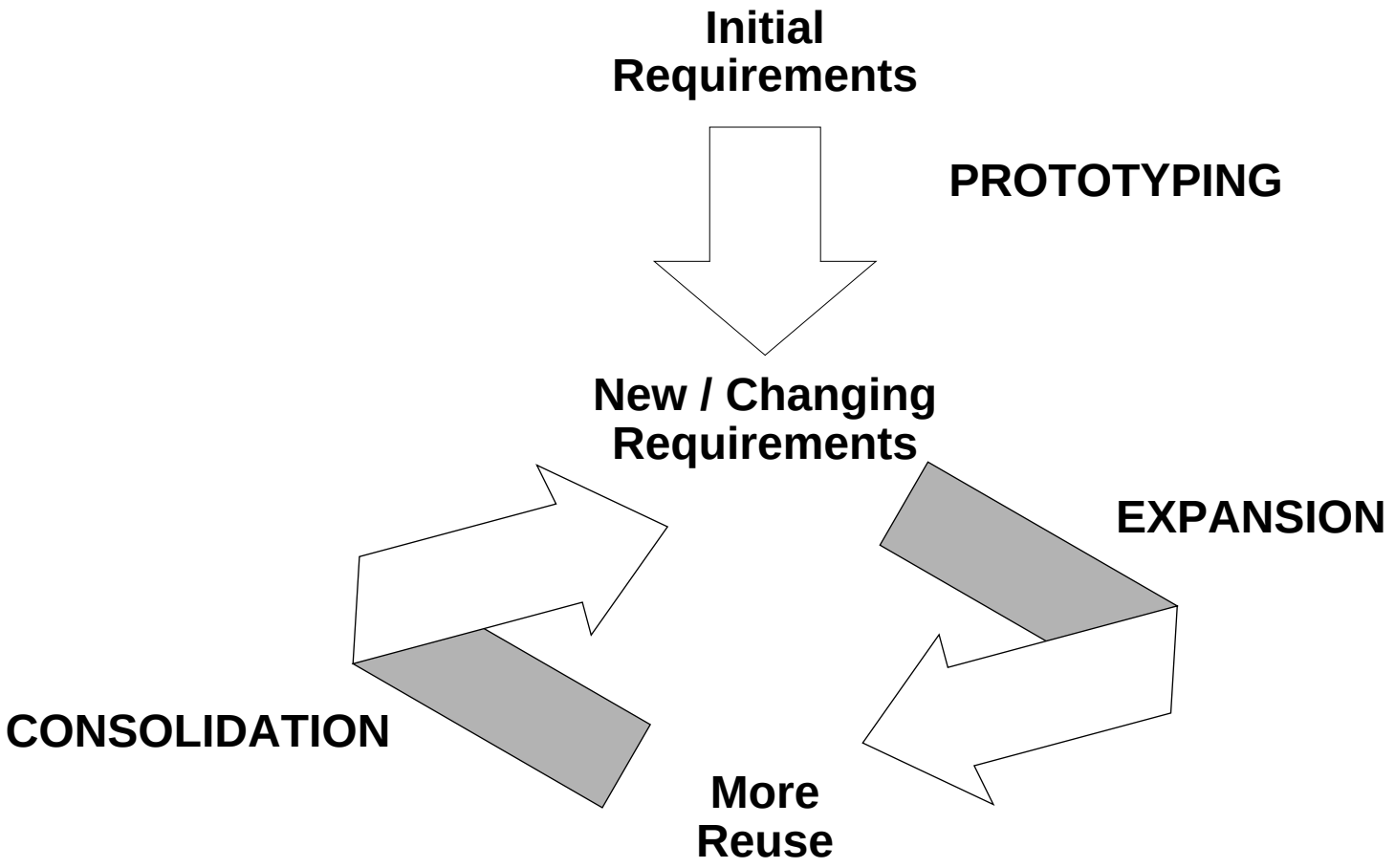
Changes costs tremendously while your project lives on.



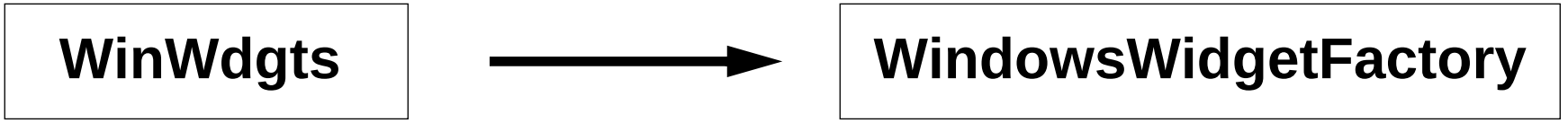

✔ make change less costly in later stages!

# Iterative Development Life-cycle

Change is the norm, not the exception !



# Example: Rename Class




| | |
|---|---|
| subclasses: | `MyWidgets extends` **`WinWdgts`** |
| contructors: | **`WinWdgts`**`()` |
| | and their calls: `widgets = new` **`WinWdgts`**`()` |
| types: | **`WinWdgts`** `currentWidgets;` |
| | `public` **`WinWdgts`** `getWidgets() {...}` |
| | `public void setWidgets(`**`WinWdgts`** |
| `widgets){...}` | |
| class method calls: | **`WinWdgts`**`.instance();` |
| class attribute accesses: | **`WinWdgts`**`.properties;` |
| casts: | `(`**`WinWdgts`**`) Object` |
| imports: | `import gui.widgets.`**`WinWdgts`**`;` |
| filename: | **`WinWdgts`**`.java` |

+ precondition checking

# Tool Support for Refactoring

*Change Efficient* *Failure Proof*

**Refactoring**

- Source-to-source program transformation
- Behaviour preserving

=> improve the program structure

**Regression Testing**

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

=> verify if improved structure does not damage previous work

**Programming Environment**

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

=> convenient for “local” ameliorations

**Configuration & Version Management**

- keep track of versions that represent project milestones

=> possibility to go back to previous version

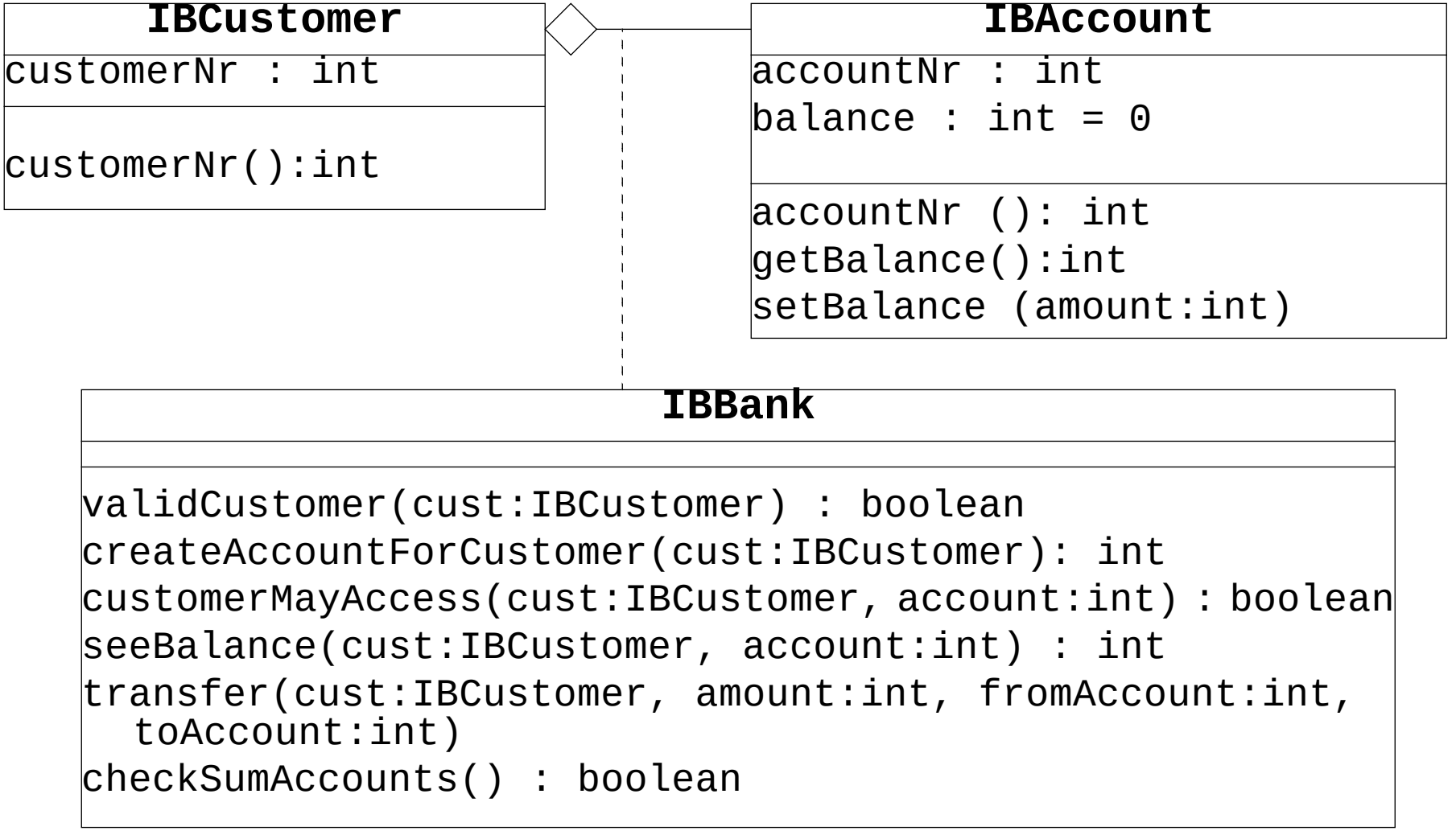
# Case Study: Internet Banking

**Initial Requirements**

- a bank has customers
- customers own account(s) within a bank
- with the accounts they own, customers may
  - deposit / withdraw money
  - transfer money
  - see the balance

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

# Prototype Design: Class Diagram



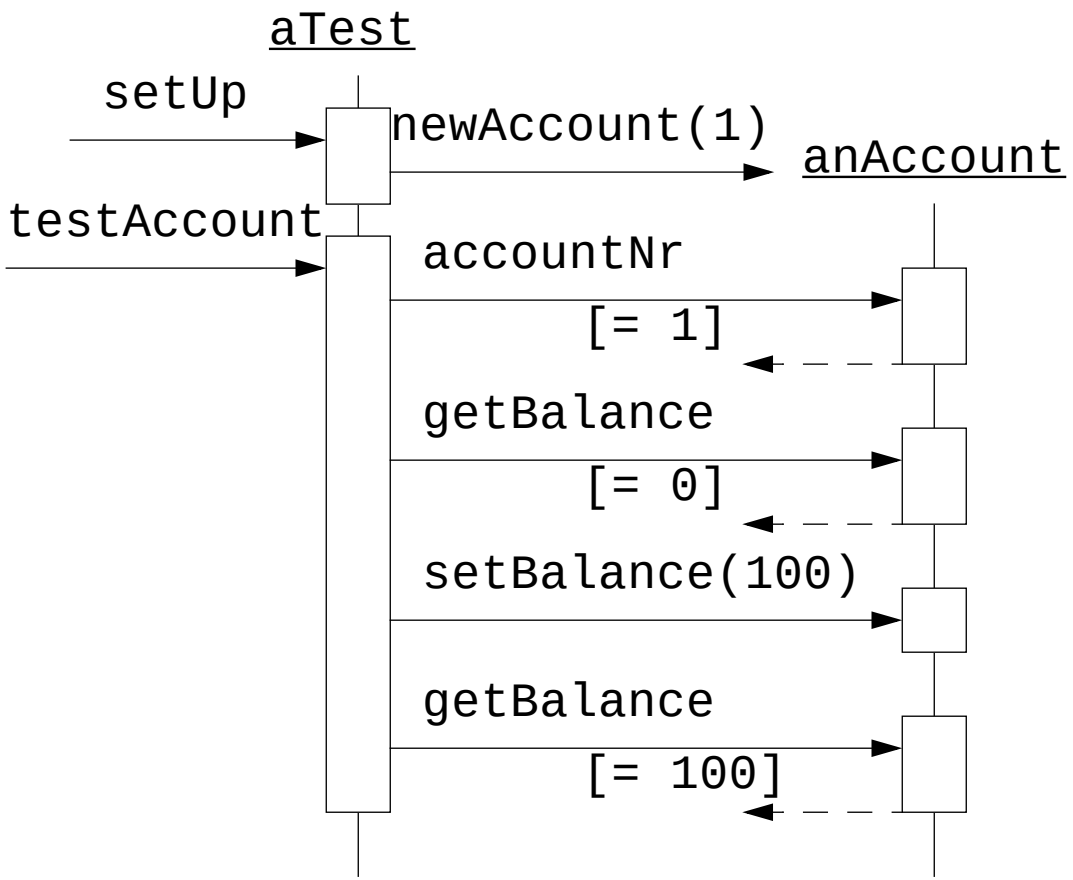
# Prototype Design: Contracts

Ensure the “secure” and “reliable” requirements.

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

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

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

# Prototype Implementation

=> see demo “IBanking1”




**Include test cases for**

- ❑IBCustomer
  - -customerNr()
- ❑IBAccount
  - -getBalance()
  - -setBalance()
- ❑IBBank
  - -createAccountFor Customer()
  - -transfer() / seeBalance() (single transfer)
  - -transfer() / seeBalance() (multiple transfers)

# Prototype Consolidation

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

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

# Expansion

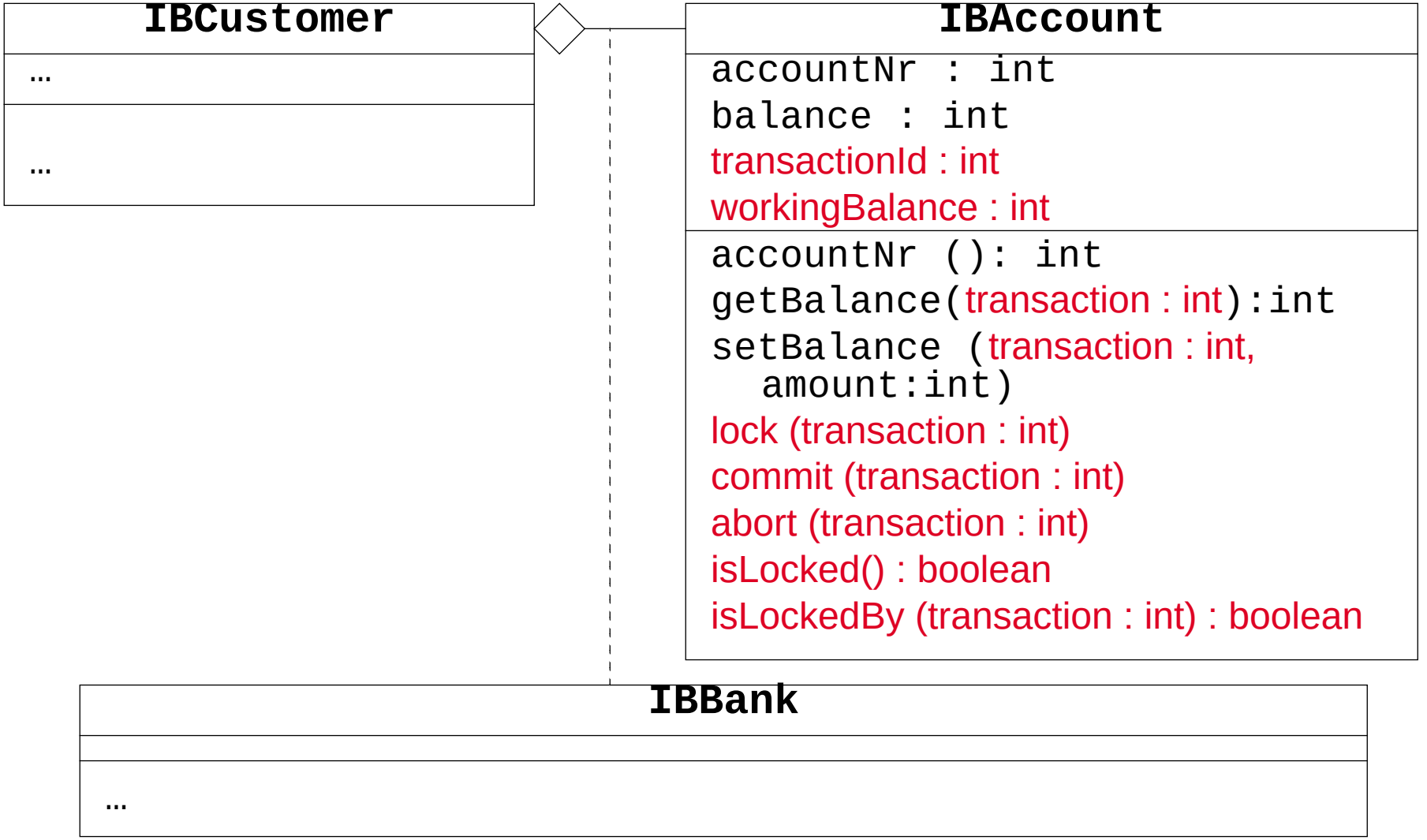
**Additional Requirement**

- concurrent access of accounts

**Add test case for**

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

# Expanded Design: Class Diagram



# Expanded Design: Contracts

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

# Expanded Implementation

**Adapt implementation**

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

**Expand Tests**

- previous tests for “getBalance()” and “setBalance()” should now fail
  => adapt tests
- new contracts, incl. commit and abort
  => new tests
- testConcurrent works!
  => we can confidently ship a new release

# Consolidation: Problem Detection

**More Reuse**

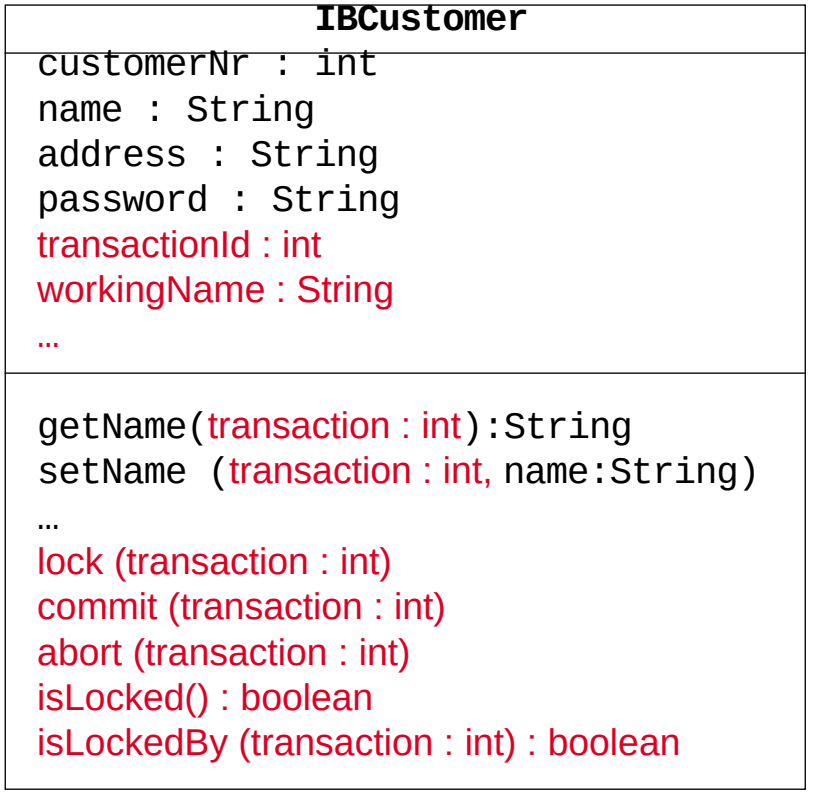
- A design review reveals that this “transaction” stuff is a good idea and should be applied to IBCustomer as well.

**=> Code Smells**

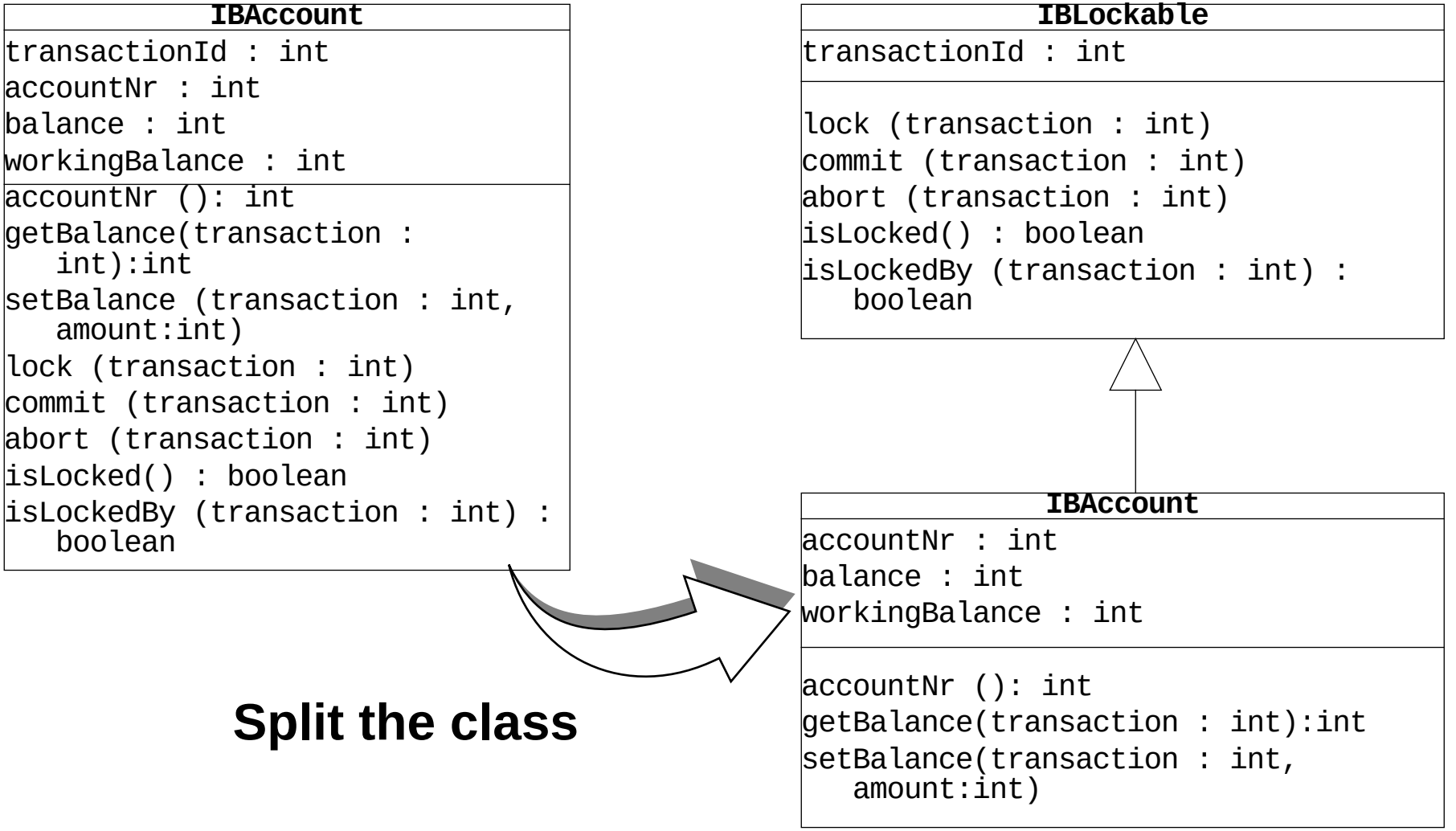
- duplicated code (lock, commit, abort + transactionId)
- large classes (extra methods, extra attributes)

**=> Refactor**

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



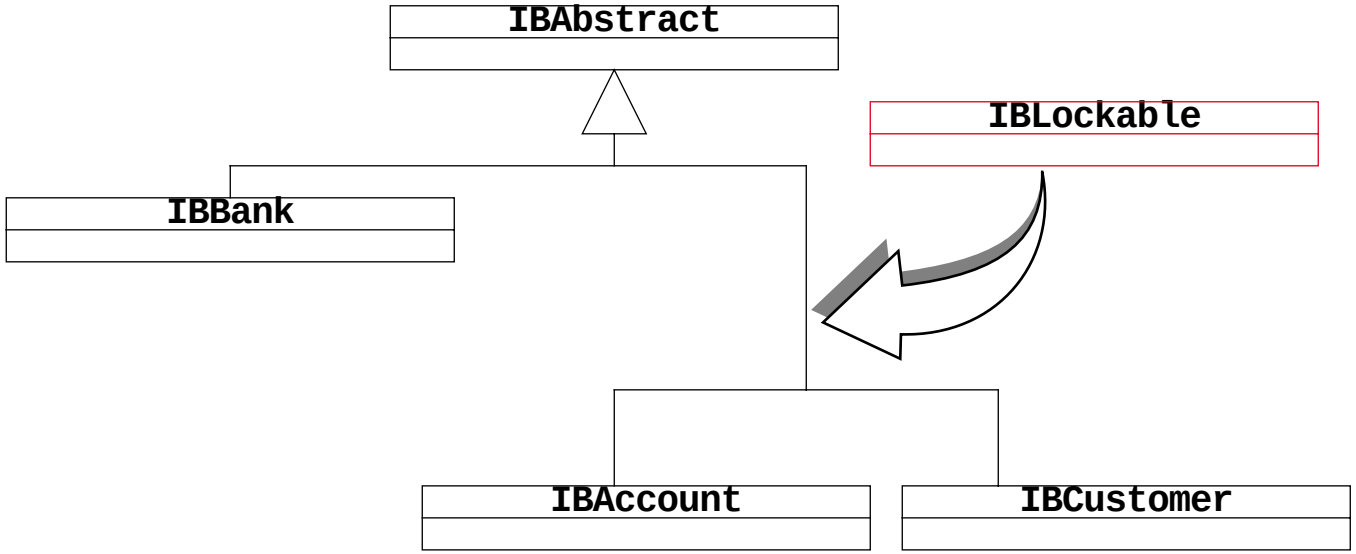
# Consolidation: Refactored Class Diagram



# Refactoring Sequence (1/5)

**Refactoring: Create Subclass**

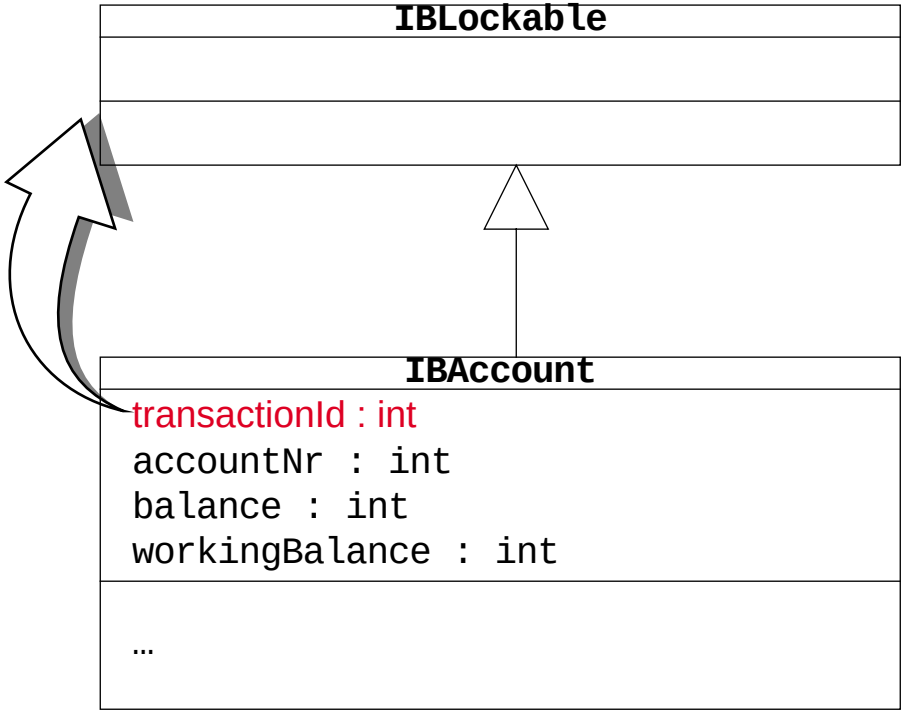
- apply “Create Subclass” on “IBAbstract” to create an empty “IBLockable” with subclass(es) “IBAccount” & “IBCustomer”



# Refactoring Sequence (2/5)

**Refactoring: Pull Up Attribute**

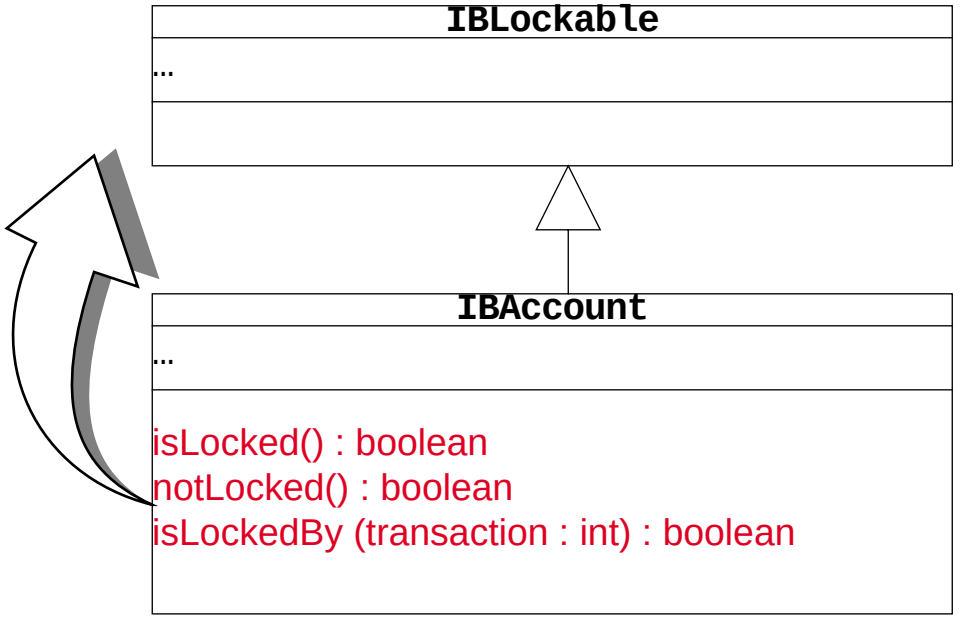
- apply “pull up attribute” on “IBLockable” to move “transactionId” up



# Refactoring Sequence (3/5)

**Refactoring: Pull Up Method**

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



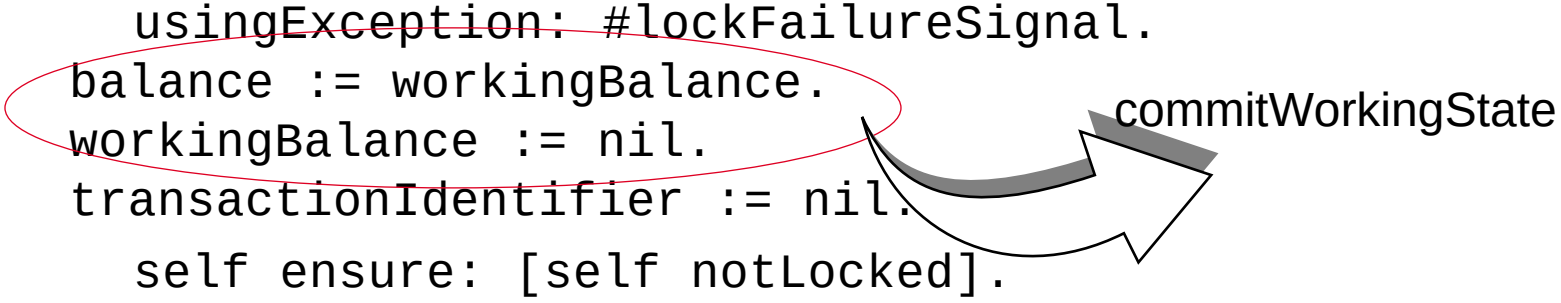

- apply “pull up” to “abort:”, “commit:”, “lock:”
  => failure: accesses to “balance” and “workingBalance” attributes

# Refactoring Sequence (4/5)

**Refactoring: Extract Method + Pull Up Method**

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

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

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




- similar for “abort:” (-> clearWorkingState) and “lock:” (-> copyToWorkingState)
- apply “pull up method” on “IBAccount” to move “abort:”, “commit:”, “lock:” up

# Refactoring Sequence (5/5)

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

- Apply “rename protocol” on “IBAccount” to rename “public-locking” into “protected-locking”

**Refactoring: Copy Method**

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

**Are we done?**

- Run the tests …
- Expand functionality of the IBCustomer

# Conclusion (1/2)

**Refactoring Philosophy**

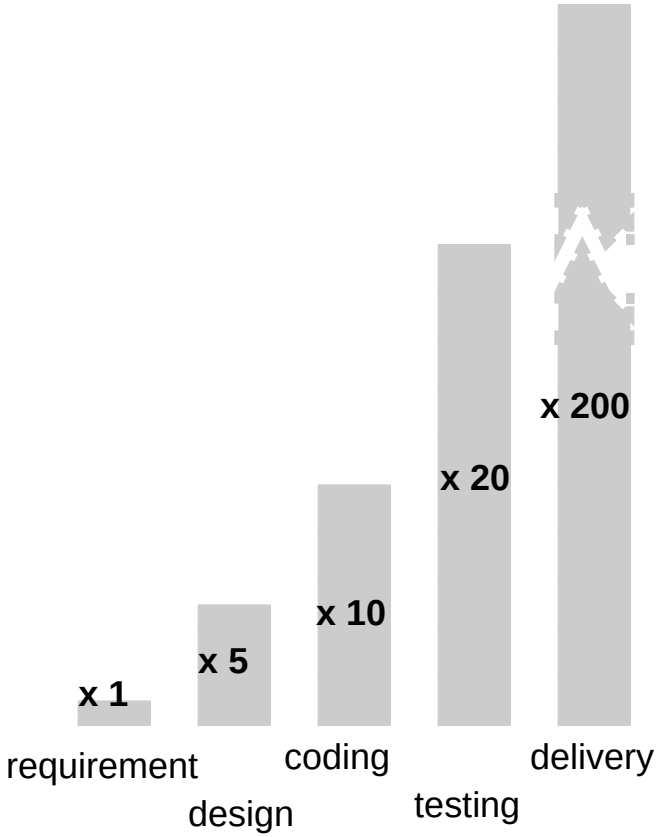
- Combine simple refactorings into larger restructuring
  => improved design
  => ready to add new functionality
- Do not apply refactoring tools in isolation

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

**Know when is as important as know-how**

- Refactored designs are more complex
- Use “code smells” as symptoms
- Rule of the thumb: State everything “Once and Only Once” (Kent Beck)

# Conclusion: Culture shock (2/2)

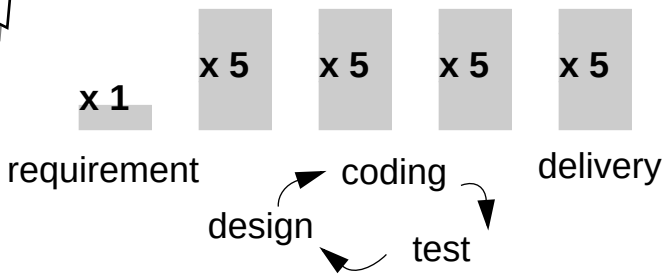



With proper

- tool support
- culture shock
- management support

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

The tools are there …



# ***Projects and More Information***

**Possible projects:**

- Analysis when to apply refactorings
  - resolve duplicated code
  - resolve design problems (such as big classes)
  - resolve unwanted dependencies
  - enforce architectures
- Refactorings in Java (some open source efforts on their way already)

**More about code smells and refactoring**

- Book on refactoring [Fowl99a].
  `http://cseng.aw.com/bookdetail.qry?ISBN=0-201-48567-2`
- Wiki-web with discussion on code smells
  `http://c2.com/cgi/wiki?CodeSmells`

# 12. Using Dynamic Information for Reverse Engineering

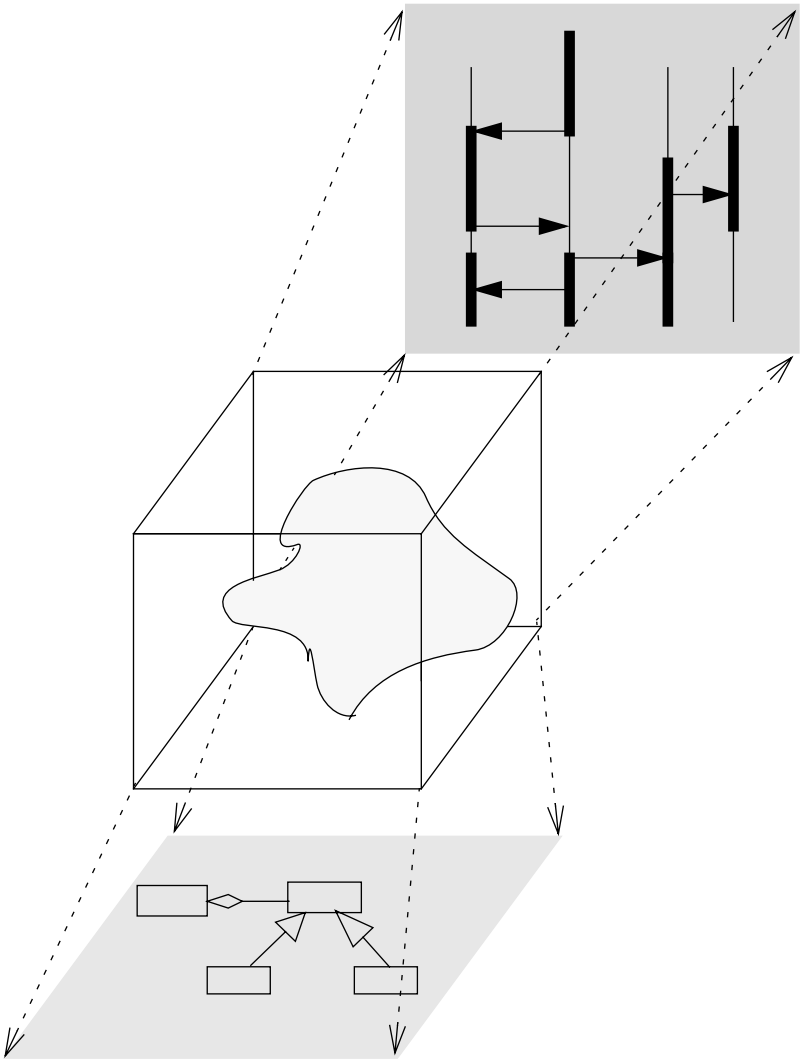
Tamar Richner
Software Composition Group

- dynamic information is important for program understanding
- how dynamic information can be used in reverse engineering
- problems in analyzing and interpreting dynamic information

# Outline

- Why dynamic information?
- What is dynamic information?
  - dynamic vs. static information
  - problems with using dynamic information
- Frequency spectrum analysis
- Visualization
- Design Recovery
- Queries and Views: Gaudi
- Instrumentation
- Conclusions

# Why Dynamic Information?



# Why Dynamic Information (cont'd)?

we are already familiar with its use for:

- debugging: examine program state
- analysing memory use
- profiling: measure time spent executing

For reverse engineering:

**functionality in OO programs comes from collaborations of objects**

but,

- control flow is hard to derive statically
- polymorphism makes it hard to figure out which method is actually executing

# What is Dynamic Information?

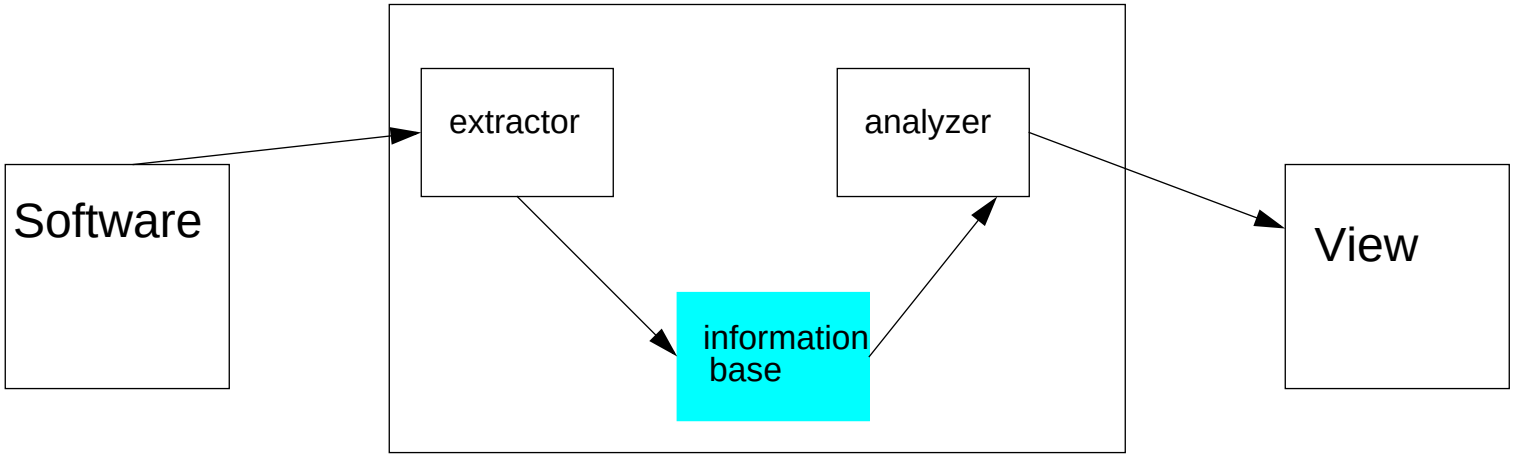
any information that we can collect from executing a program.

for example:

- value of a variable at time t
- number of milliseconds spent executing method m
- instance x of class X created 25 instances of class Y from t1 to t2
- methods on the call stack at time t
- X.x invokes method m on Y.y at time t

and so on.....

**Which kind of information is useful for reverse engineering?**



# Static vs. Dynamic Information

| | Static | Dynamic |
|---|---|---|
| **Precision** | | ✔ |
| **Completeness** | ✔ | |

| | False Positives | No False Positives |
|---|---|---|
| **False Negatives** | | **dynamic** |
| **No False Negatives** | **static** | |

- dynamic information relates a scenario to behavior
- static and dynamic information complement each other

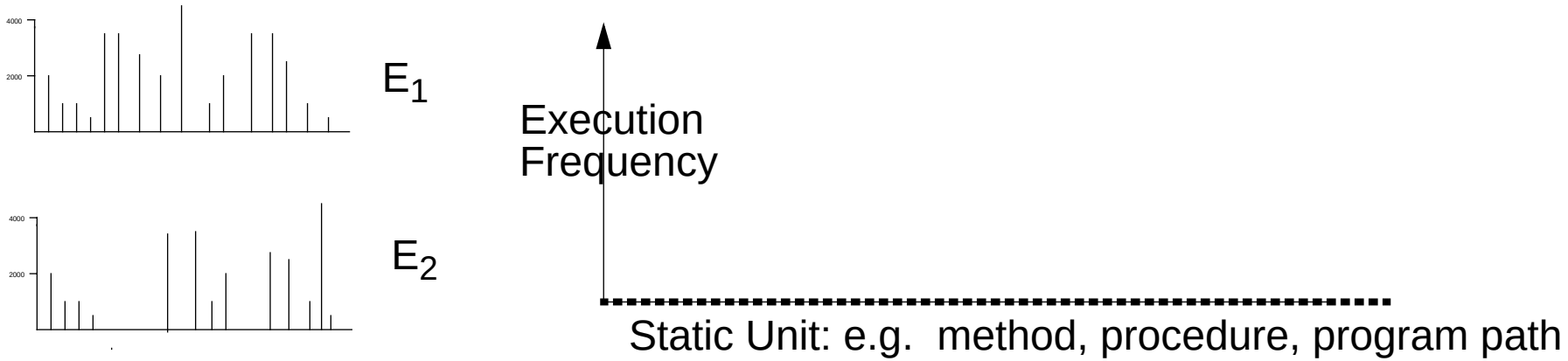
# Problems with using Dynamic Information

- huge amount of information generated by tracing
- from low-level information to high-level model (as for static information)
- problem of coverage (as for testing)
- instrumentation (not always easy)
- how do we express behavioral models of OO software?

# Roadmap

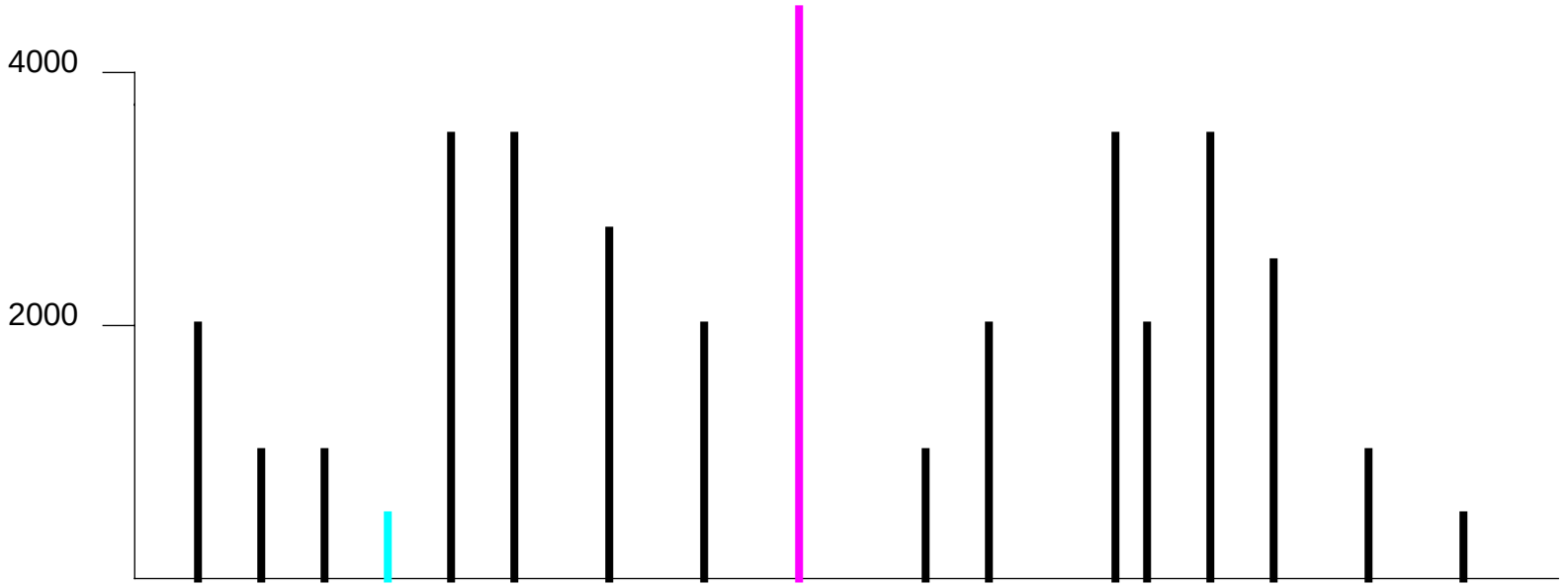
- Why dynamic information
- What is dynamic information
  - dynamic vs. static information
  - problems with using dynamic information
- Frequency spectrum analysis
  - ☞ looking at execution frequencies
  - some heuristics
- Visualization
- Design Recovery
- Queries and Views: Gaudi
- Instrumentation
- Conclusions

# Frequency Spectrum



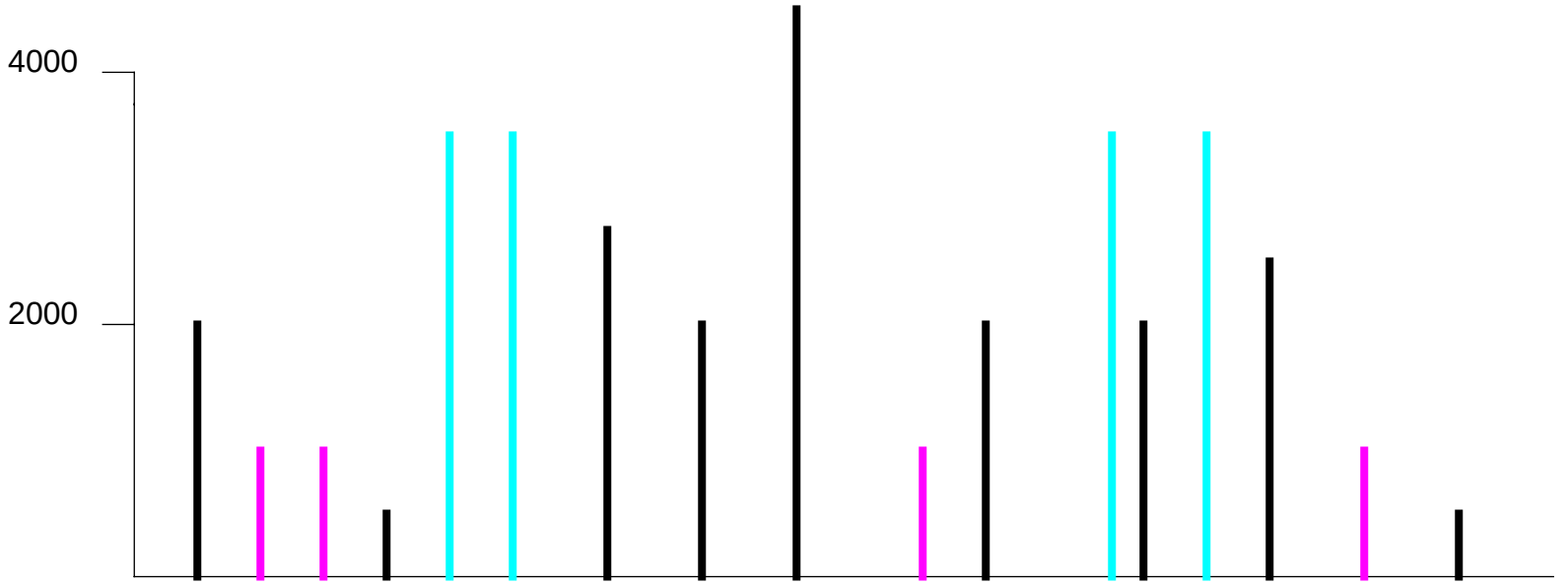

- For a single execution:
  frequency spectrum analysis (FSA) [Ball99]
  - low vs. high frequencies
  - related frequencies
  - specific frequencies
- Comparing executions:
  - Dynamic Differencing [Reps97][Agra98]
  - Concept Coverage Analysis [Ball 99]

# FSA: low vs. high frequencies



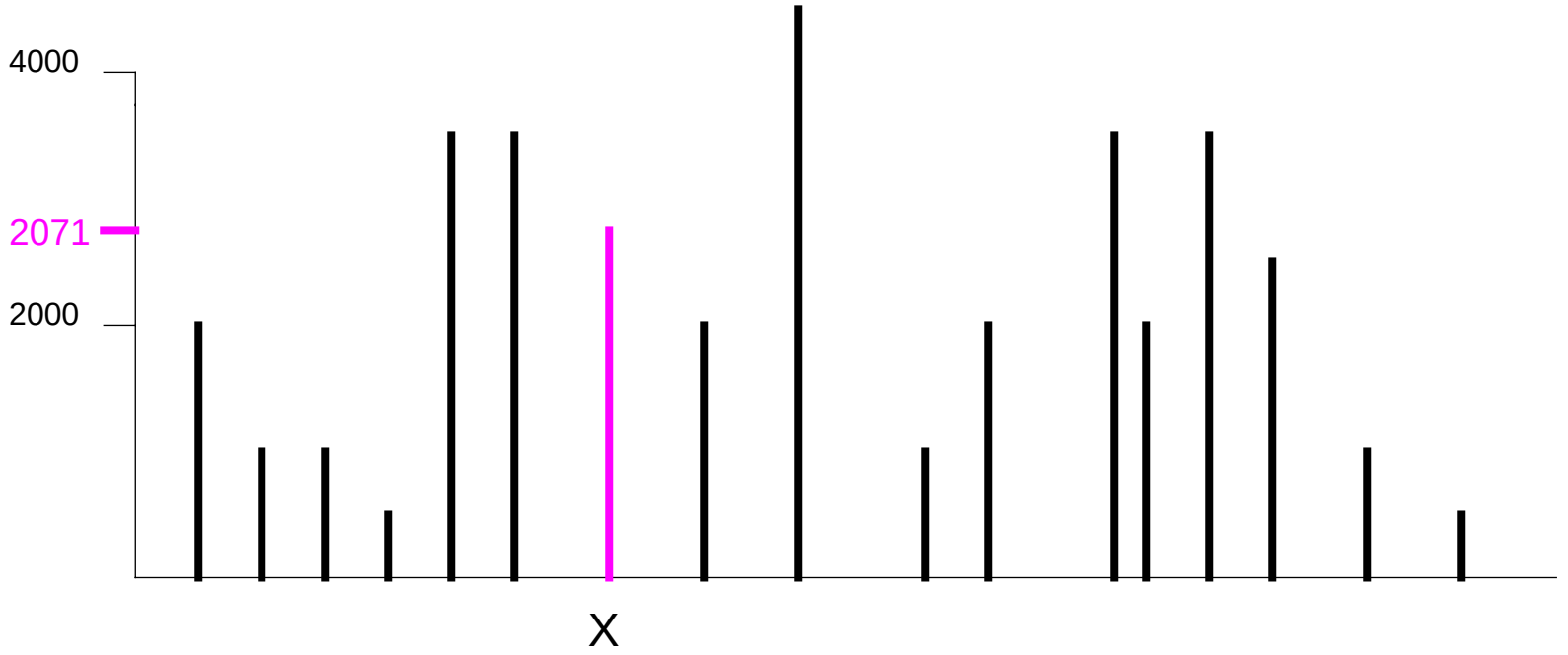

- high frequencies -> lower level abstractions
- low frequencies -> higher level abstractions

# FSA: related frequencies



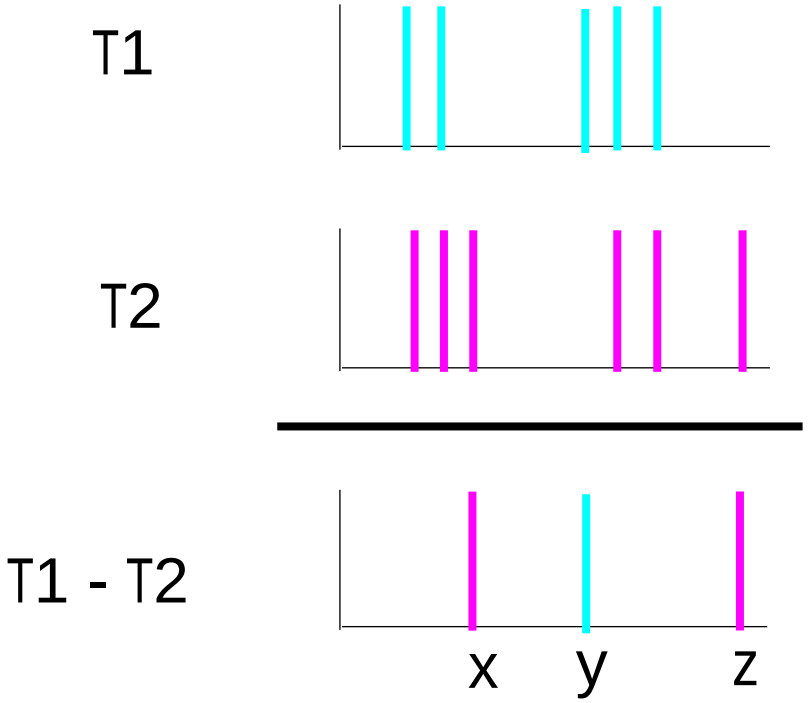

- same frequency -> frequency clusters

# FSA: specific frequencies




- associate frequency to input, e.g. input file with 2071 records -> method X handles this input

# Dynamic Differencing




- locate where a feature is implemented

  e.g.

  telephony features like call setup, call waiting

  data-sensitive code (e.g. year 2000 problem)

  - ➪ test case selection is important to get good results

# Summary of Spectrum Techniques

are these really used in practice?

- on a small scale - yes.
- on a large scale - probably not:
  - test case preparation is critical and number of test cases necessary to get meaningful information is large. Work reported is in research labs.

# Roadmap

- Why dynamic information
- What is dynamic information
- Frequency spectrum analysis
- Visualization
  - ☞ visualization techniques
  - – some examples
- Design Recovery
- Queries and Views: Gaudi
- Instrumentation
- Conclusions

# Visualization

of what?

- summary information about the execution
- sequence diagrams: showing message sends between objects

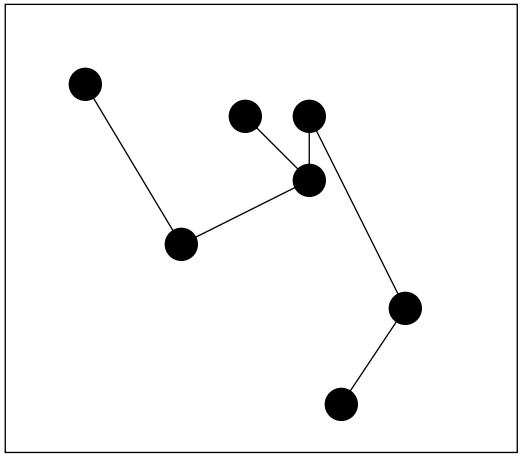
how?
techniques for **displaying** lots of information:

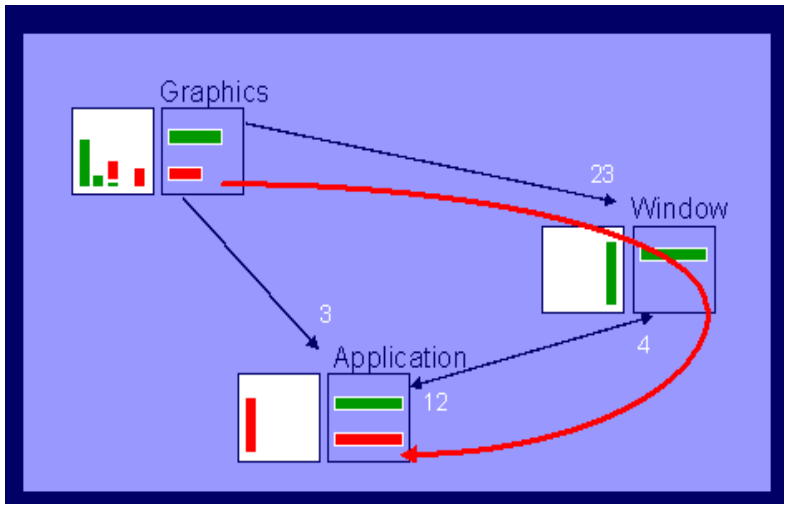
- remove time element through animation [DePa94][Sefi97][Walk98]
- navigation through hyperlinks [Kosk96]
- compress information into visual pattern: information mural [Jerd98]

# Animated Summaries

- affinity diagram
  - inter-class call graph
- histogram
  - number of instances

these can be animated real-time or offline - they can also summarize data without animation



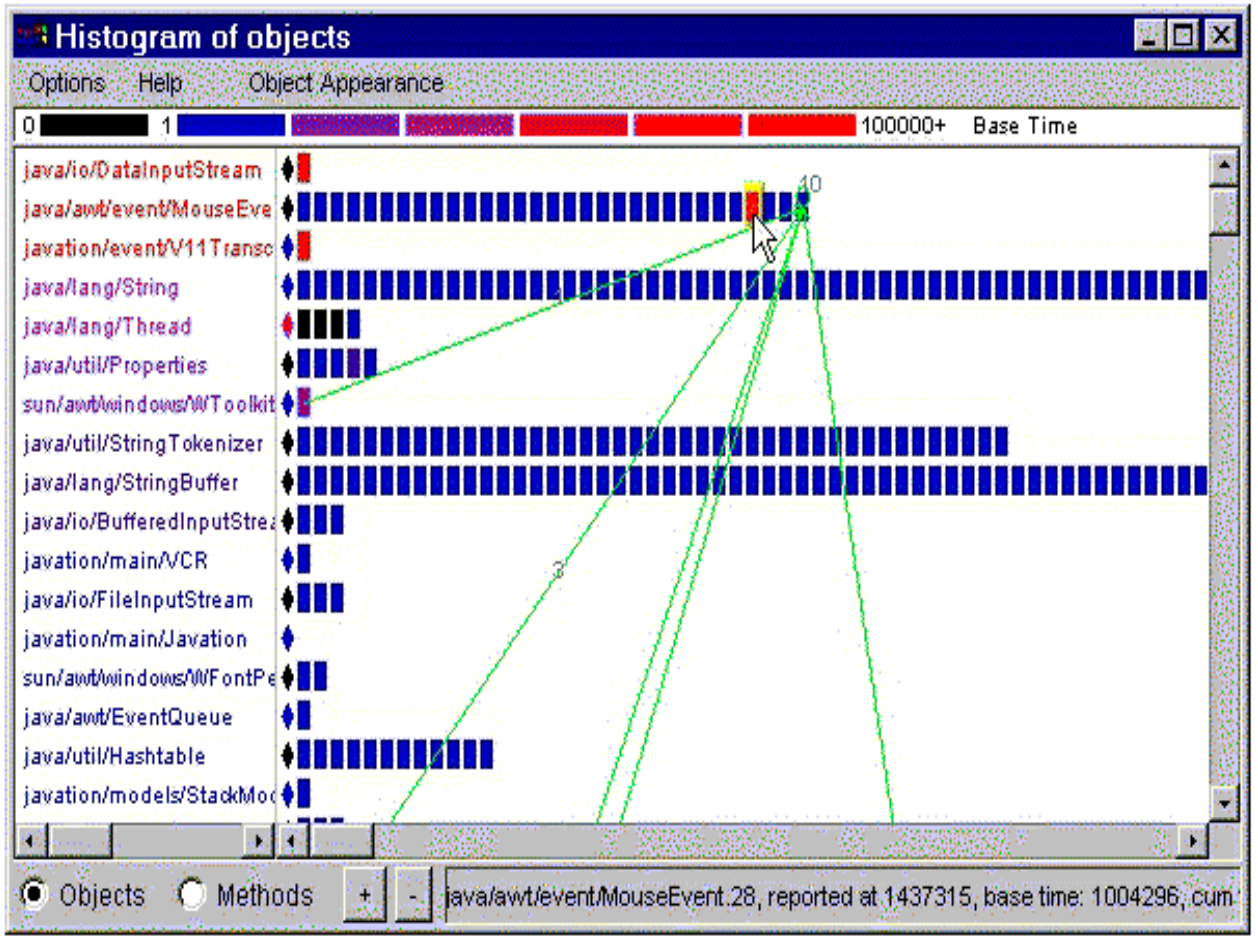



**animates the class affinities (frequency of calls)[Sefi97]**

**shows total # of objects allocated (green) and deallocated (red), animated through a high-level model [Walk98]**

# Animated Summaries Example: Jinsight

[DePa94]



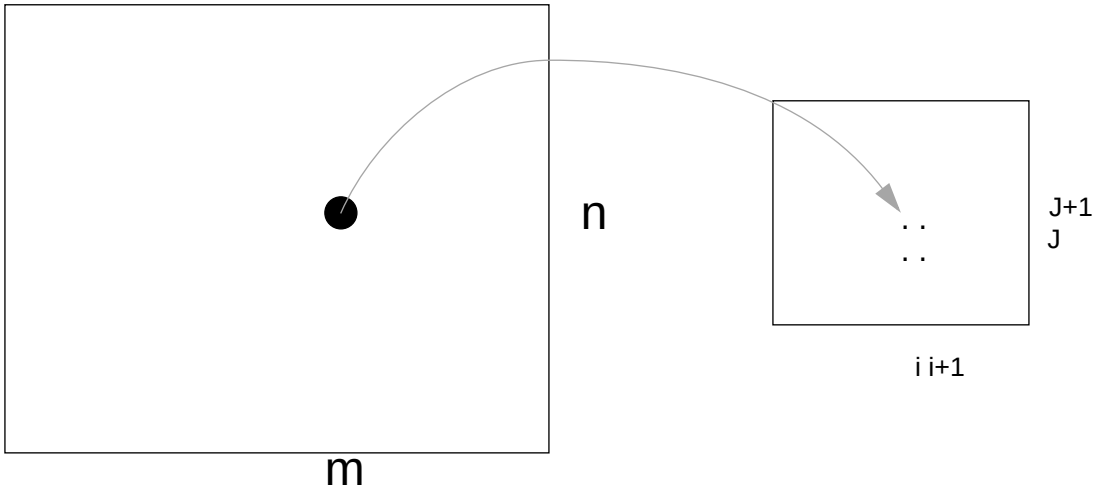

Show instances or methods grouped by class, and indicates their level of activity.

# Information Mural

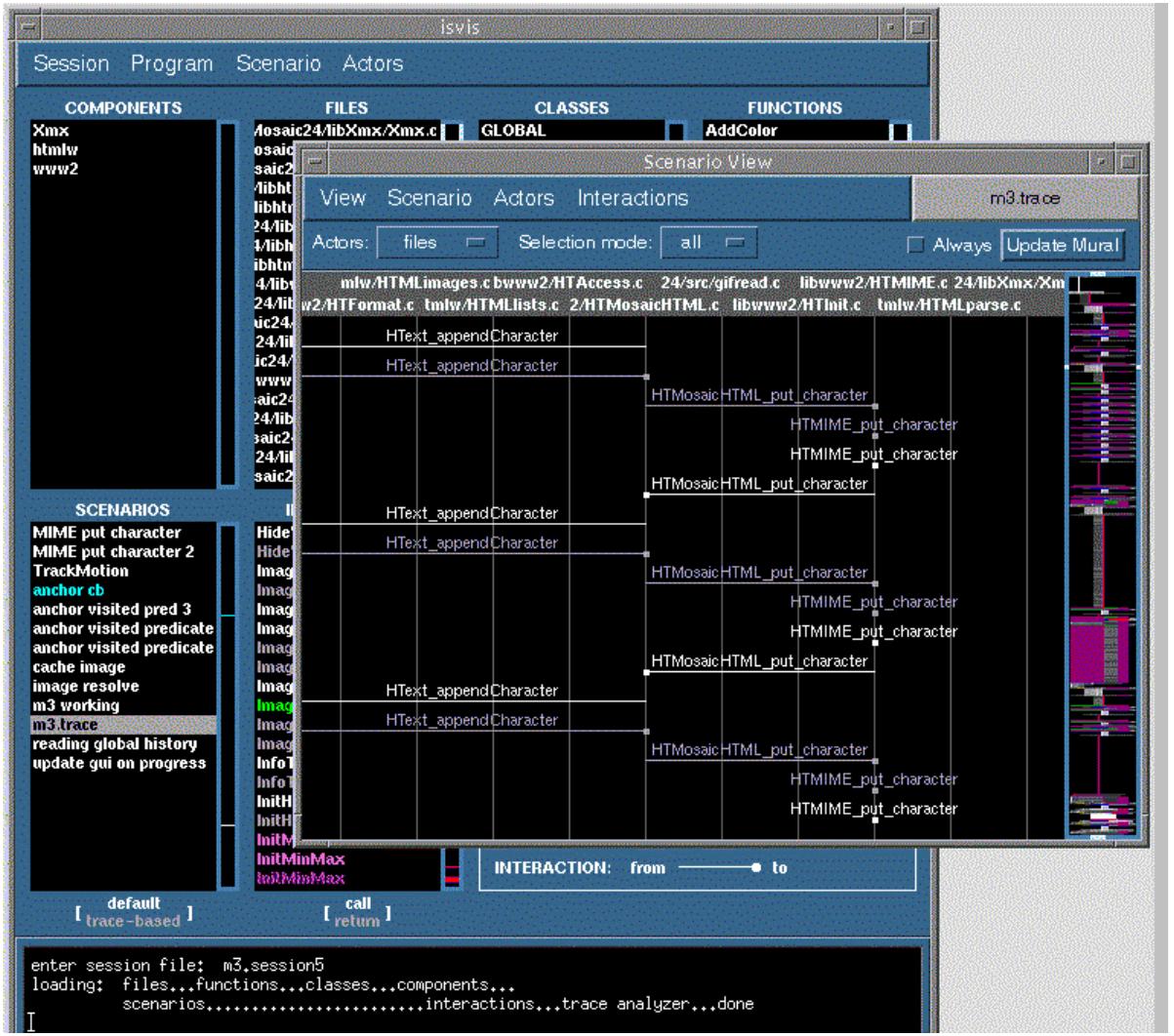
display all the information on one window or screen - mimicing what information would look like in its entirety [Jerd98] :

- represent the relative information **density** at each pixel instead of presence of absence of information.
- density is visuallized as grey scale value, or with colours

.



# Information Mural Example: ISVis




mural is a navigational guide through the long sequence diagram

visual pattern recognition

[Jerd98]

# RoadMap

- Why dynamic information
- What is dynamic information
- Frequency spectrum analysis
- Visualization
- Design Recovery
  - ☞ requirements: focus and granularity
  - – using clustering and filtering in visualization
- Queries and Views: Gaudi
- Instrumentation
- Conclusions

# A Step Back: Design Recovery

issues in dealing with information extracted:

- **Granularity**:
  build high-level model of the software
- **Focus**:
  need a model which describes the aspect of the software of interest for the task

# Design Recovery through Visualization

- techniques for **displaying** lots of information

But, more important:

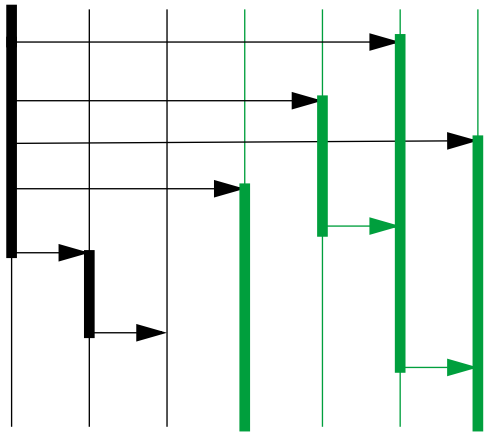
- focusing on relevant information:
  - **instrumenting** selectively
  - **filtering** out uninteresting information
- higher granularity
  - **clustering** elements to create higher-level abstractions

# Selective Instrumentation & Filtering

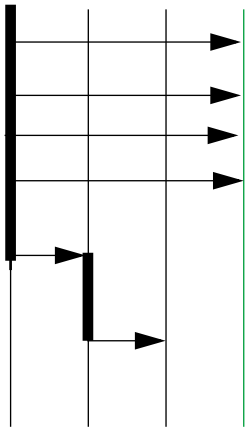
- look at dynamic information only for certain classes, methods.
- eliminate message sends based on certain criteria, e.g. self-sends, sends to metaclass, constructors, etc.)

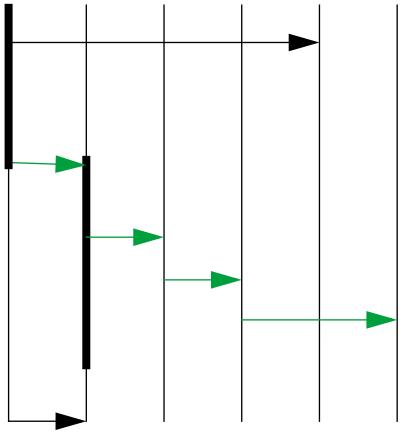
Example: ISVis [Jerd97]: can edit the sequence diagram to remove actors (vertical line) or interactions (several horizontal lines).

# Clustering

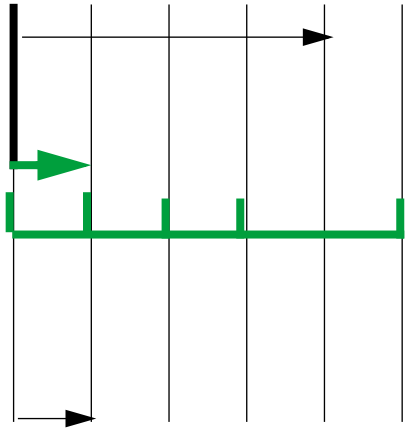


clustering objects



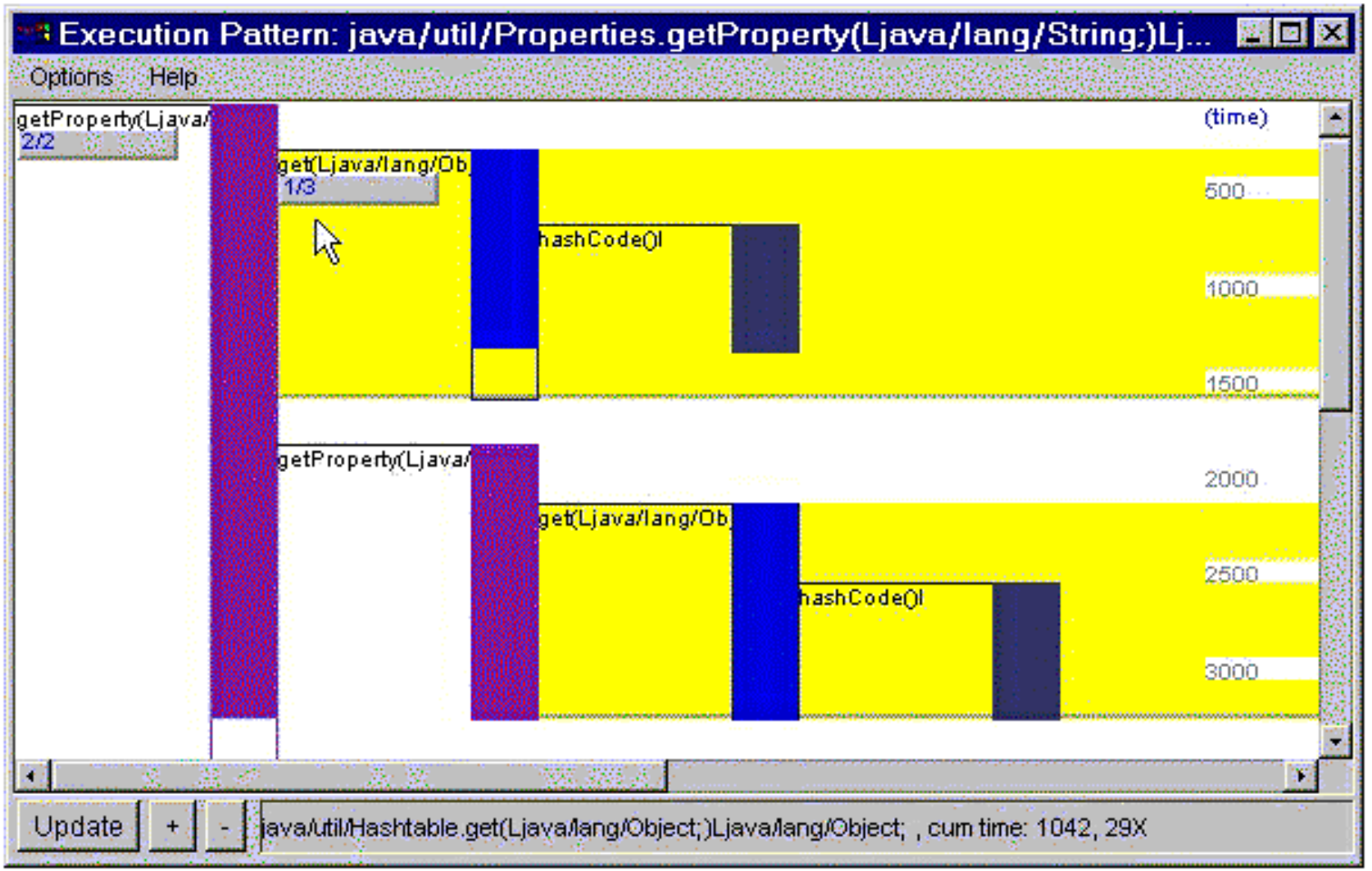


clustering events



clustering events: can use pattern matching to find recurring interactions

# Recognizing Patterns: example of Jinsight




Browse recurring patterns of communication arising from a selected method, each displayed as a function of time.

# Summary of Visualization for Design Recovery

good for:

- localizing interesting or strange interactions
- debugging, performance, space analysis

disadvantages:

- still too low-level: hard to navigate, too close to debugging
- models are mental models: they can not be manipulated

Despite its immediate appeal, it is not clear what the real benefits are of visualization vs. textual feedback about the system.

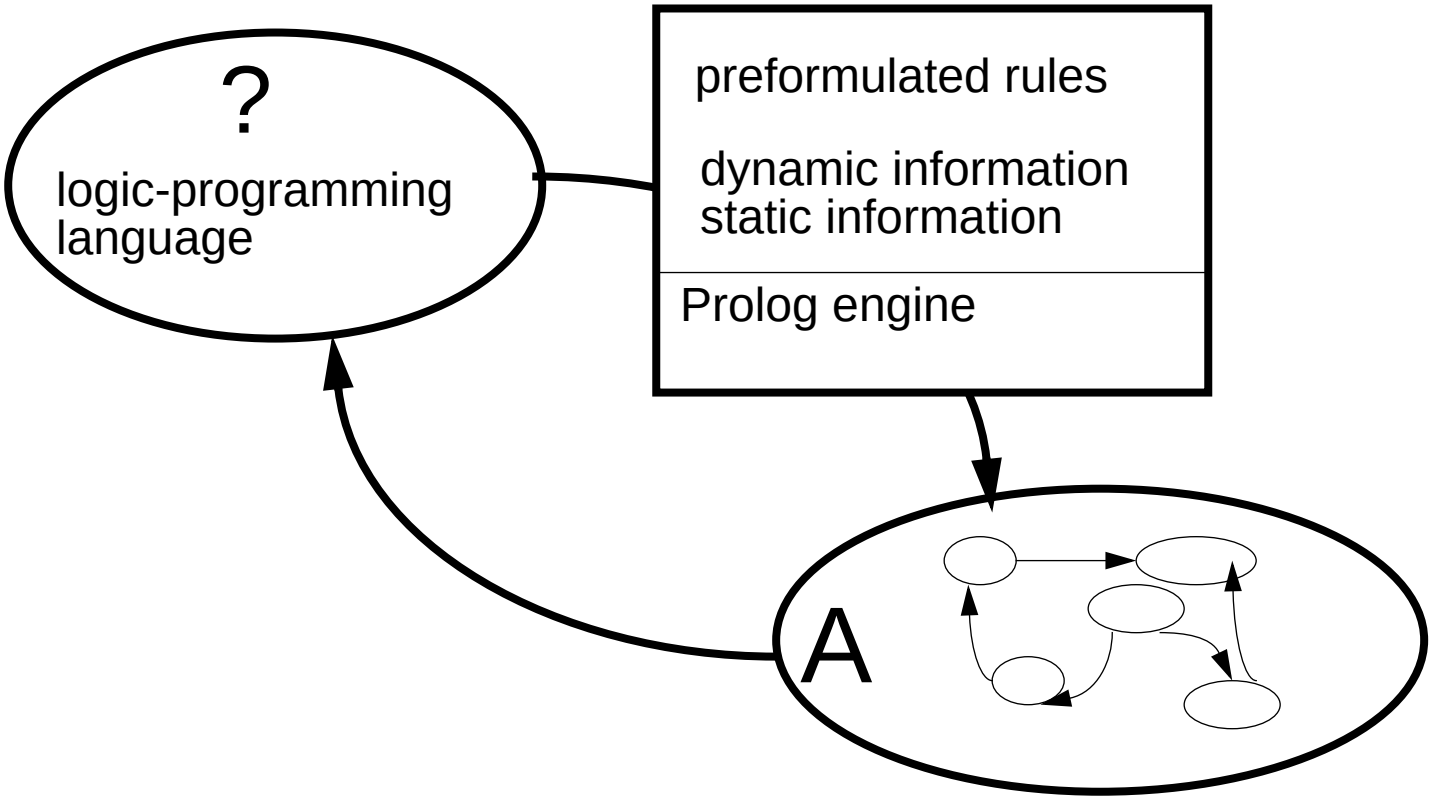
What are good OO models for expressing behavior?

- UML interaction diagrams for expressing relevant, critical scenarios: how do we find these in the trace?
- role models: look at the roles that classes play in interactions

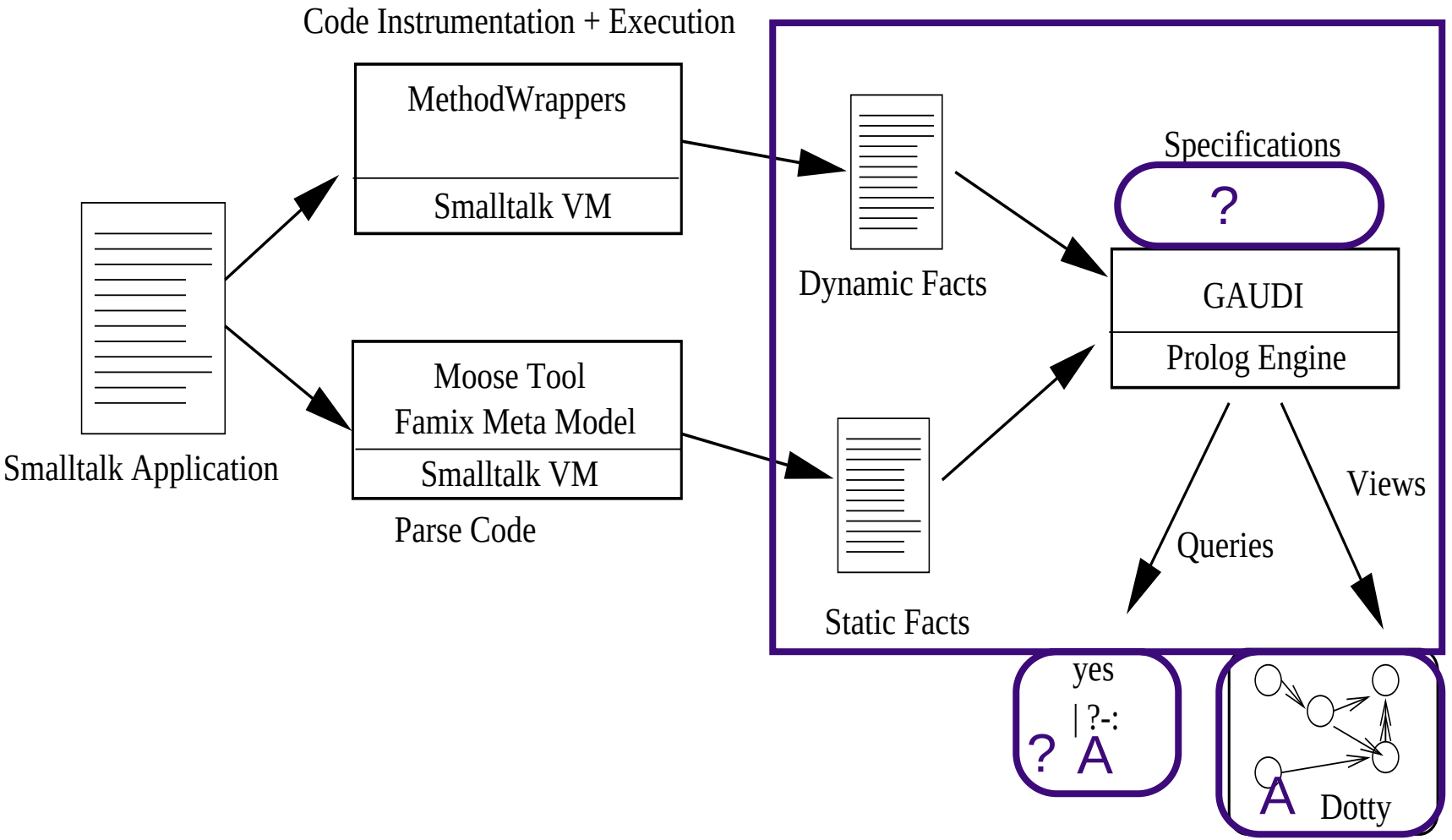
# RoadMap

- Why dynamic information
- What is dynamic information
- Frequency spectrum analysis
- Visualization
- Design Recovery
- Queries and Views: Gaudi
  - ☞ overview of approach
  - – modelling static and dynamic information
  - – queries and views
- Instrumentation
- Conclusions

# Gaudi: overview of Approach



# Gaudi: Implementation



# Modelling OO Programs and their Execution: the Basic Relations

Static Information:

```
class(ClassName,SourceAnchor).
superclass(SuperClass,SubClass).
method(Class,MethodName,IsClassMethod,Category).
```

Dynamic information:

```
send(SN,SL,Class1,Instance1,Class2,Instance2,Method).
```

```
send(1,1,'WidgetDragDropCallbacks',650,SystemNavigator',10956,'classWantToDrag:').
send(2,2,'SystemNavigator',10956,'SystemNavigator',10956,'className').
send(3,3,'SystemNavigator',10956,'SystemNavigator',10956,'classNames').
send(4,4,'SystemNavigator',10956,'SystemNavigator',10956,'viewCategory').
send(5,4,'SystemNavigator',10956,'SystemNavigator',10956,'classNames').
send(6,5,'SystemNavigator',10956,''SystemNavigator',10956,'classList').
send(7,5,'SystemNavigator',10956,''BRMultiSelectionInList',5250,'selections').
send(8,1,'MessageChannel',10775,'SystemNavigator',10956,'changeRequest').
send(9,2,'SystemNavigator',10956,'changeRequest','CodeModelLockPolicy_class',13067,'flushCache').
send(10,2,'SystemNavigator',10956,'changeRequest','CodeModel',12429,'updateRequest').
send(11,3,'CodeModel',12429,'StateLockPolicy',6170,'isLocked').
send(12,3,'CodeModel',12429,'CodeModel',12429,'updateRequest').
send(13,4,'CodeModel',12429,'CodeModel',12429,'subcanvases').
send(14,5,'CodeModel',12429,'CodeModel',12429,'subcanvases').
send(15,5,'CodeModel',12429,'CodeModel',12429,'tool').
send(16,5,'CodeModel',12429,'CodeModel',12429,'tool').
send(17,4,'CodeModel',12429,'ClassNavigatorTool',11142,'updateRequest').
send(18,5,'ClassNavigatorTool',11142,'ClassNavigatorTool',11142,'subcanvases').
send(19,6,'ClassNavigatorTool',11142,'ClassNavigatorTool',11142,'subcanvases').
send(20,6,'ClassNavigatorTool',11142,'ClassNavigatorTool',11142,'subcanvas').
send(21,5,'ClassNavigatorTool',11142,'BrowserClassTool',3963,'updateRequest').
send(22,6,'BrowserClassTool',3963,'BrowserClassTool',3963,'updateRequest').
send(23,7,'BrowserClassTool',3963,'BrowserClassTool',3963,'subcanvases').
send(24,6,'BrowserClassTool',3963,'BrowserClassTool',3963,'isEditing').
send(25,7,'BrowserClassTool',3963,'BrowserClassTool',3963,'isEditing').
send(26,8,'BrowserClassTool',3963,'BrowserClassTool',3963,'subcanvases').
send(27,7,'BrowserClassTool',3963,'BrowserClassTool',3963,'textController').
send(28,8,'BrowserClassTool',3963,'BrowserClassTool',3963,'controllerFor:').
send(29,1,'DependentsCollection',8382,'BRMultiSelectionInList',5250,'update:with:from:').
send(30,1,'BRMultiSelectionView',2857,'BRMultiSelectionView',2857,'updateSelectionChannel').
send(31,1,'MessageChannel',2123,''SystemNavigator',10956,'changedClass').
send(32,2,'SystemNavigator',10956,'SystemNavigator',10956,'updateProtocolList').
```

send(33,3,’SystemNavigator’,10956,’SystemNavigator’,10956,’protocols’).
send(34,4,’SystemNavigator’,10956,’SystemNavigator’,10956,’protocolList’).
send(35,4,’SystemNavigator’,10956,’BRMultiSelectionInList’,12982,’selections’).
send(36,3,’SystemNavigator’,10956,’’SystemNavigator’,10956,’newProtocolList:’).
send(37,4,’SystemNavigator’,10956,’SystemNavigator’,10956,’newProtocolListNoUpdate:’).
send(38,5,’SystemNavigator’,10956,’SystemNavigator’,10956,’selectedClass’).
send(39,6,’SystemNavigator’,10956,’SystemNavigator’,10956,’nonMetaClass’).
send(40,7,’SystemNavigator’,10956,’SystemNavigator’,10956,’className’).
send(41,8,’SystemNavigator’,10956,’SystemNavigator’,10956,’classNames’).
send(42,9,’SystemNavigator’,10956,’SystemNavigator’,10956,’viewCategory’).
send(43,9,’SystemNavigator’,10956,’SystemNavigator’,10956,’classNames’).
send(44,10,’SystemNavigator’,10956,’SystemNavigator’,10956,’classList’).
send(45,10,’SystemNavigator’,10956,’BRMultiSelectionInList’,5250,’selections’).
send(46,7,’SystemNavigator’,10956,’SystemNavigator’,10956,’classForName:’).
send(47,6,’SystemNavigator’,10956,’SystemNavigator’,10956,’isMeta’).
send(48,7,’SystemNavigator’,10956,’SystemNavigator’,10956,’meta’).
send(49,5,’SystemNavigator’,10956,’SystemNavigator’,10956,’category’).
send(50,6,’SystemNavigator’,10956,’SystemNavigator’,10956,’categories’).
send(51,7,’SystemNavigator’,10956,’SystemNavigator’,10956,’categoryList’).
send(52,7,’SystemNavigator’,10956,’BRMultiSelectionInList’,4697,’selections’).

▪

▪

▪

send(1187,13,’BrowserClassTool’,3963,’BrowserClassTool’,3963,’textHolder’)..

# Gaudi: Formulating Derived Relations

sendsCreate(C1,C2).

```
sendsCreate(C1,C2):-
invokesMethodClass(C1,C2,M),
metaclassOf(MC,C2),
methodCategory(MC,M,'instance creation').
```

invokesMethodClass(C1,C2,M).

```
invokesMethodClass :-send(_,_,C1,_,C2,_,M).
```

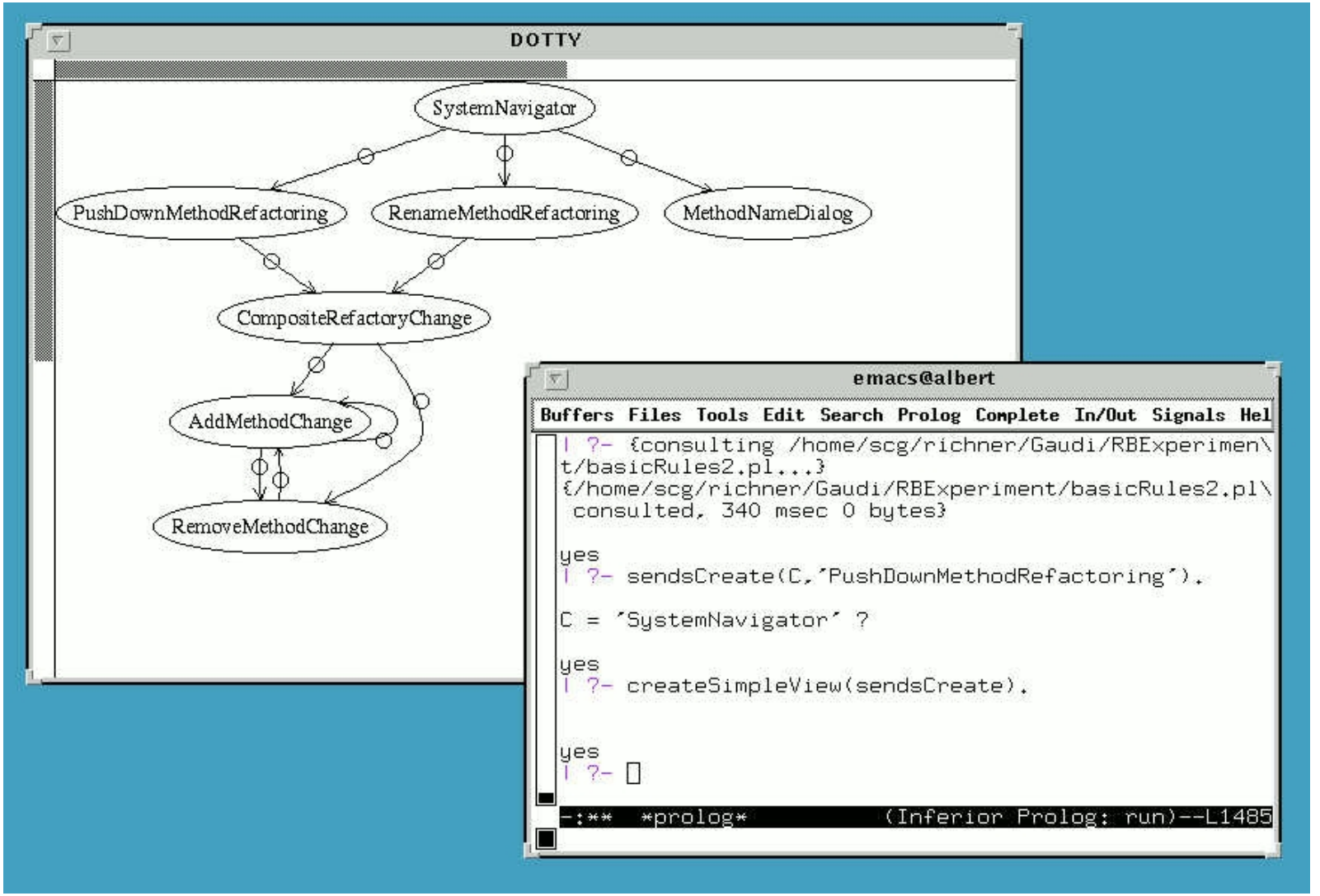
methodCategory(C,M,Cat).

```
methodCategory(C,M,Cat):-method(C,M,_,Cat).

methodCategory(C,M,Cat):-
inHierarchy(Superclass,Class),
method(Superclass,Method,_,Category).
```

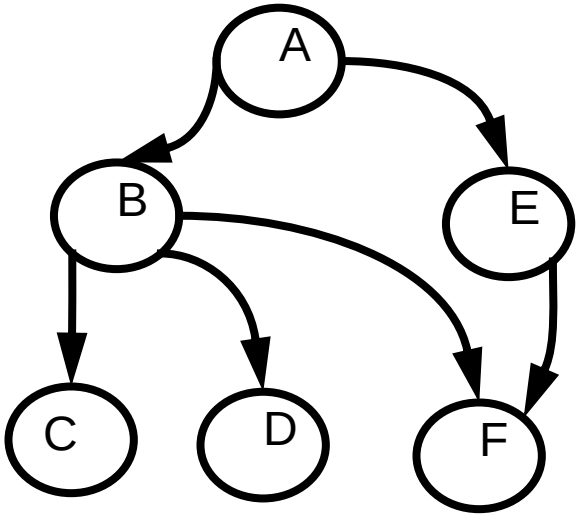
inHierarchy(Class,Subclass).

```
inHierarchy(Class,Superclass).
inHierarchy(Class,Superclass):-
superclass(Class,Subclass).

inHierarchy(Class,Superclass)
superclass(Superclass,Subclass),
inHierarchy(Class,Superclass).
```

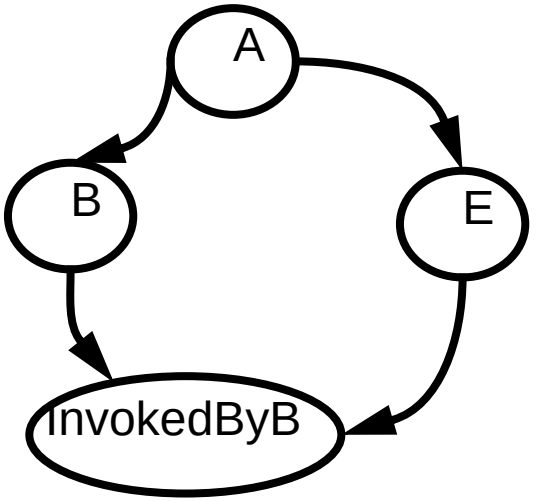
# Gaudi: Using Derived Relations for Querying



# Gaudi: Simple vs. Composed Views



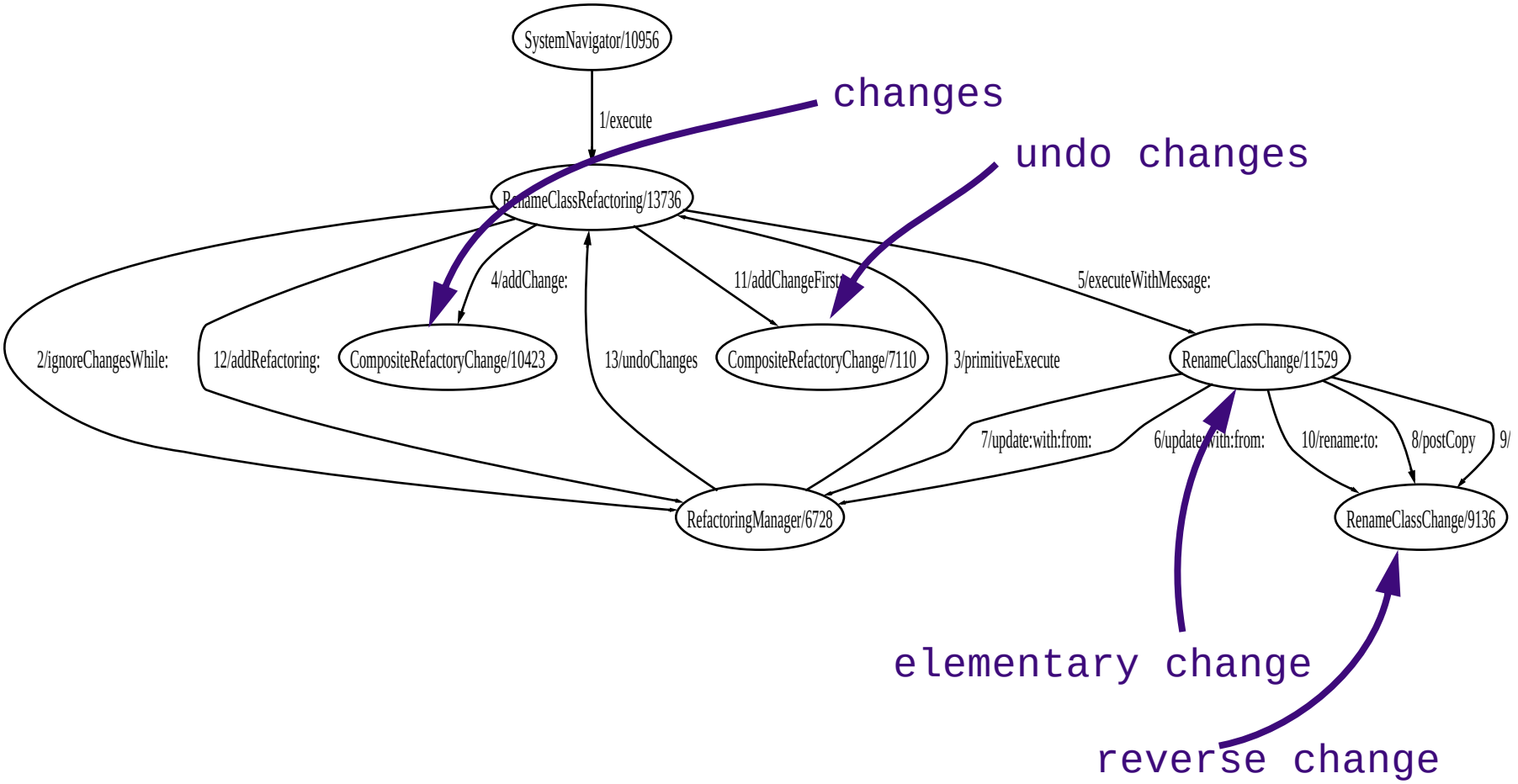

createSimpleView(invokesClass).




createView(invokedClass,component).

```
component('invokedByB',L) :-
    setof(ClassinvokesClass('B',Class),L).
```

# Gaudi: Instance Level View



# Gaudi Summary

[Rich99]

uses Prolog rules to:

- query the database of static and dynamic information
- create views of the information

views can be:

- high-level: e.g. send and create relationships between classes, clusters of classes
- low-level: e.g. show sequence of message sends between instances

methodology:

- start with a question to be answered.
- create a high-level view in order to locate what we are interested in.
- focus the search by creating more fine-grained views and iterate.

# Instrumentation

how do we collect dynamic information?

- with reflective language support: e.g. Smalltalk
- without (C++, Java) : insert instrumentation code and recompile, or modify the VM.

problems:

- a flexible facility for instrumenting selectively
- a non-intrusive instrumentation

# Conclusions

we did not talk about using dynamic information for:

- typing and refactoring
- reverse engineering structural relationships (aggregation, composition - mutable, variable)
- generation of state diagrams for objects
- understanding concurrent programs

Summary:

- dynamic information is important for program understanding
- how dynamic information can be used in reverse engineering
  - most of work for OO is related to visualization
- problems in analyzing and interpreting dynamic information
  - handling the large amount of information
  - creating meaningful abstractions
  - expressing behavior concisely

# References

Frequency Analysis:

[Ball99] T. Ball, The Concept of Dynamic Analysis, Proceedings ESEC ’99. pp. 218-234.

[Agra98] H. Agrawal et al., Mining System Tests to Aid Software Maintenance, IEEE Computer, July 1998, pp. 64-73.

[Rep97] T. Reps et al., The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem, Proceedings ESEC ’97, pp. 432-449.

Visualization of Object-oriented Applications:

[DePa94] W. De Pauw, D. Kimelman and J. Vlissides, Modeling object-oriented program execution, Proceedings ECOOP ’94, LNCS 821, pp. 163-182.

[Lang95] D.B. Lange and Y. Nakamura, Interactive visualization of design patterns can help in framework understanding, Proceedings OOPSLA ’95, pp. 342-357.

[Kosk96] K. Koskimies and H. Moessenboek, Scene: using scenario diagrams and active test for illustrating OO programs, Proceedings ICSE 96, pp.366-374.

[Jerd97] D. Jerding and S. Rugaber, Using visualization for architectural localization and extraction, Proceedings Working Conference on Reverse Engineering (WCRE ’97), pp. 56-65.

[Walk98] R. Walker et al., Visualizing dynamic software system information through high-level models, Proceedings OOPSLA ’98, pp. 271-283.

Gaudi:

[Rich99] T. Richner and S. Ducasse, Recovering high-level views of object-oriented applications from static and dynamic information, Proceedings International Conference on Software Maintenance (ICSM ’99), pp. 13-22.

Visualization tools:

interaction diagram (for Smalltalk) :

http://st-www.cs.uiuc.edu/users/brant/Applications/WrapperApplications.html

Jinsight (for Java) :

http://www.research.ibm.com/jinsight/

ISVis (for C++) :

http://www.cc.gatech.edu/morale/tools/isvis/isvis.html