Software Improvement Group

Software Analysis and Testing

Métodos Formais em Engenharia de *Software*

November 2007
Joost Visser

Arent Janszoon Ernststraat 595-H
NL-1082 LD Amsterdam
info@sig.nl
www.sig.nl

Structure of the lectures

Software Improvement Group

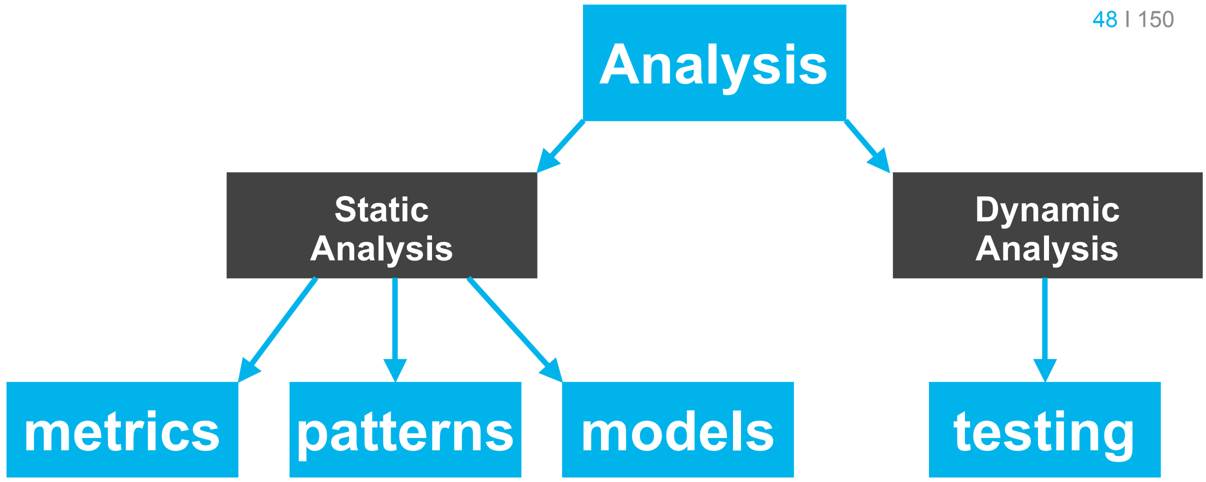
Last week

- Introduction SIG
- General overview of software analysis and testing
- Testing
- Patterns

Today

- Quality & metrics
- Reverse engineering

Structure of the lectures

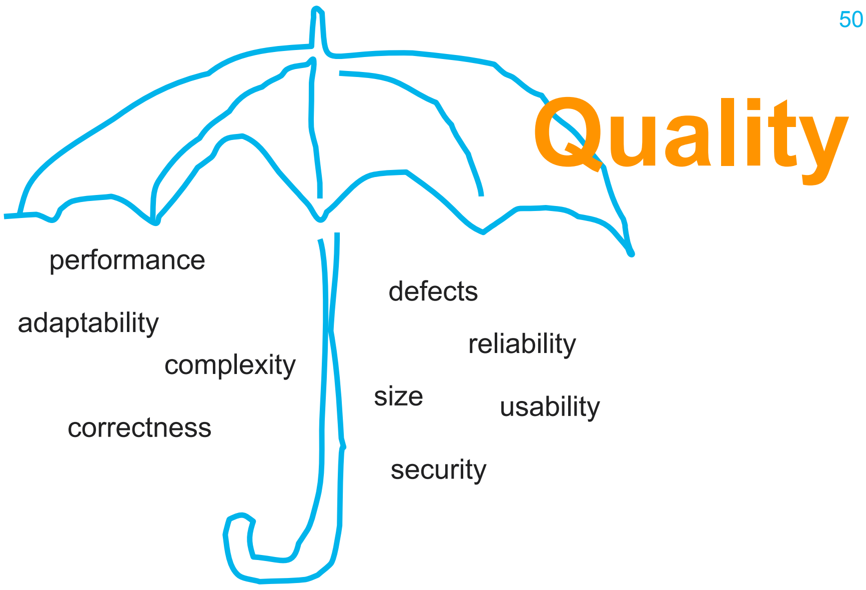


METRICS & QUALITY

Software Analysis and Testing, MFES Universidade do Minho by Joost Visser, Software Improvement Group © 2007.

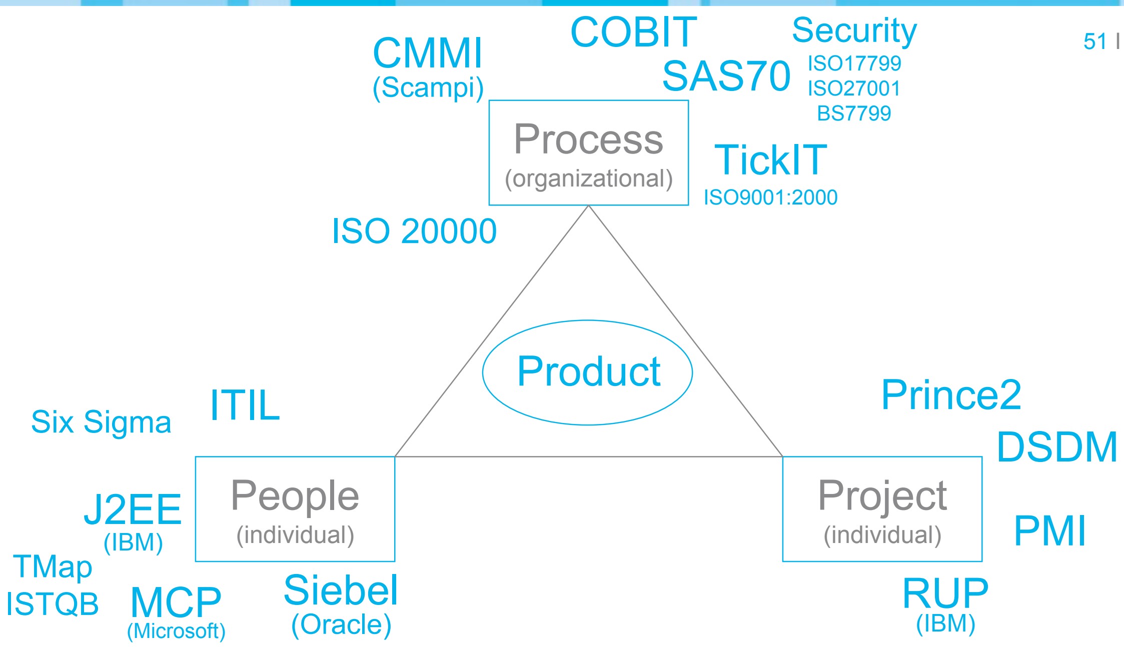
Software analysis
What?

Software Improvement Group



The bermuda triangle of software quality

Software Improvement Group



Software Quality
Process

Software Improvement Group

Capability Maturity Model® Integration (CMMI®)

- "… is a process improvement approach that provides organizations with the essential elements of effective processes.." (SEI)
- CMMI for Development (CMMI-DEV), Version 1.2, August 2006.
- consists of 22 process areas with capability or maturity levels.
- CMMI was created and is maintained by a team consisting of members from industry, government, and the Software Engineering Institute (SEI)
- http://www.sei.cmu.edu/cmmi

The Standard CMMI Appraisal Method for Process Improvement (SCAMPI)

- "… is the official SEI method to provide benchmark-quality ratings relative to CMMI models."

CMMI®

Software Quality
Process

Software Improvement Group

Software Engineering Institute | Carnegie Mellon

http://sas.sei.cmu.edu/pars/

Organization

Organization Name: Accenture
Appraisal Sponsor Name: Jack Ramsay, Marco Spaziani Testa, Maria Angeles Ramirez
Lead Appraiser Name: John Voss
SEI Partner Name: Accenture LLP

Model Scope and Appraisal Ratings

Level 2		Level 3		Level 4		Level 5	
Satisfied	REQM	Satisfied	RD	Out of Scope	OPP	Out of Scope	OID
Satisfied	PP	Satisfied	TS	Out of Scope	QPM	Out of Scope	CAR
Satisfied	PMC	Satisfied	PI				
Not Applicable	SAM	Satisfied	VER				
Satisfied	MA	Satisfied	VAL				
Satisfied	PPQA	Satisfied	OPF				
Satisfied	CM	Satisfied	OPD				
		Satisfied	OT				
		Satisfied	IPM				
		Satisfied	RSKM				
		Satisfied	DAR				

Organizational Unit Maturity Level Rating: 3
Additional Information for Appraisals Resulting in Capability or Maturity Level 4 or 5 Ratings:

Software Quality Process

Levels

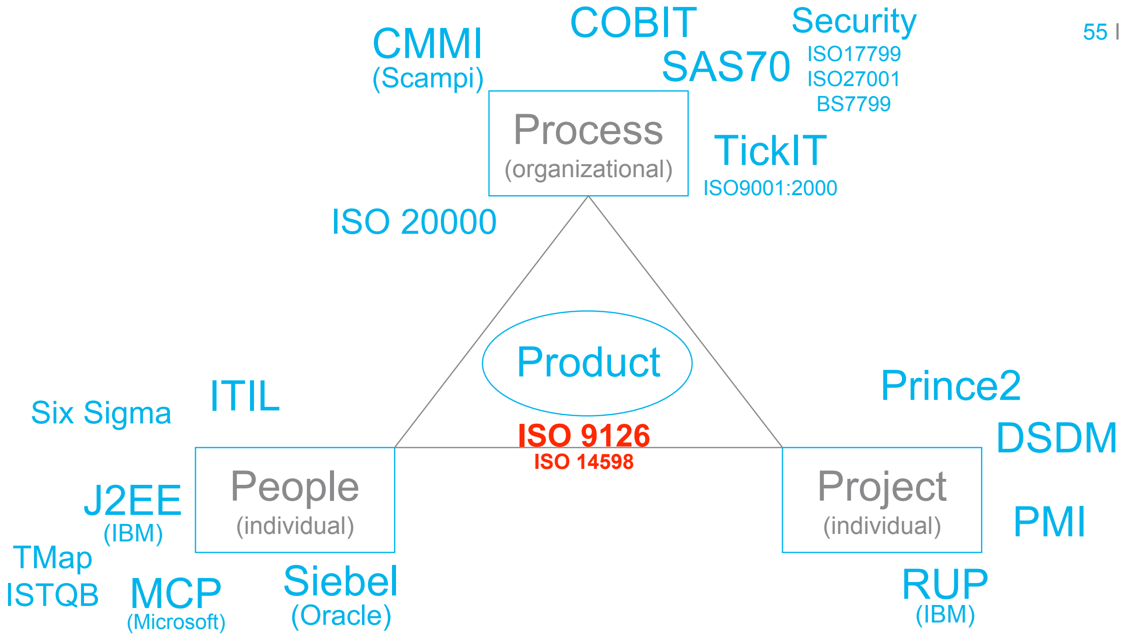
- L1: Initial
- L2: Managed
- L3: Defined
- L4: Quantitatively Managed
- L5: Optimizing

http://www.cmmi.de
(browser)

Process Areas

- Causal Analysis and Resolution
- Configuration Management
- Decision Analysis and Resolution
- Integrated Project Management
- Measurement and Analysis
- Organizational Innovation and Deployment
- Organizational Process Definition
- Organizational Process Focus
- Organizational Process Performance
- Organizational Training
- Product Integration
- Project Monitoring and Control
- CMMI Project Planning
- Process and Product Quality Assurance
- Quantitative Project Management
- Requirements Development
- Requirements Management
- Risk Management
- Supplier Agreement Management
- Technical Solution
- Validation
- Verification

The bermuda triangle of software quality



But …

What is software quality?

What are the technical and functional aspects of quality?

How can technical and functional quality be measured?

Software product quality standards

ISO/IEC 9126
Software engineering -- Product quality

1. Quality model
2. External metrics
3. Internal metrics
4. Quality in use metrics

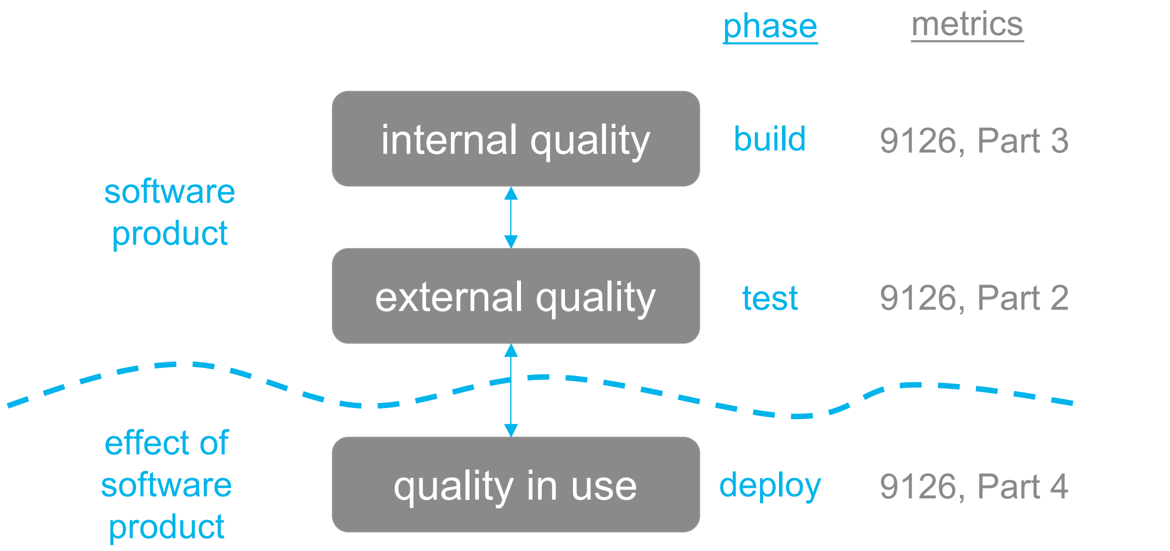
ISO International Organization for Standardization

ISO/IEC 14598
Information technology -- Software product evaluation

1. General overview
2. Planning and management
3. Process for developers
4. Process for acquirers
5. Process for evaluators
6. Documentation of evaluation modules

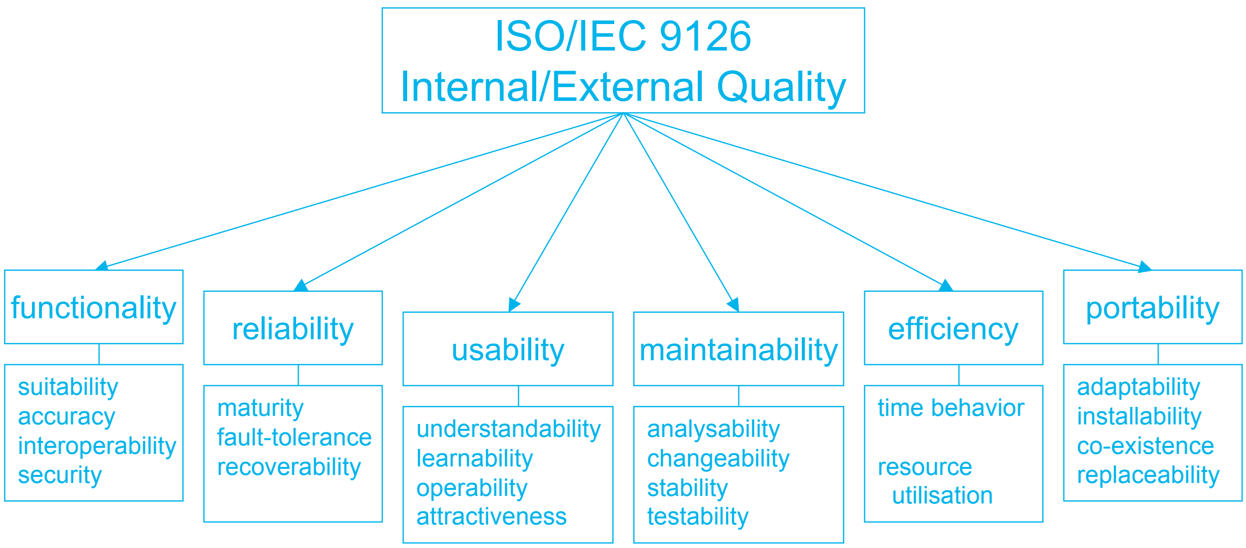
ISO/IEC 9126, Part 1
Quality perspectives

Software Improvement Group



ISO/IEC 9126, Part 1
Product quality model: internal and external

Software Improvement Group

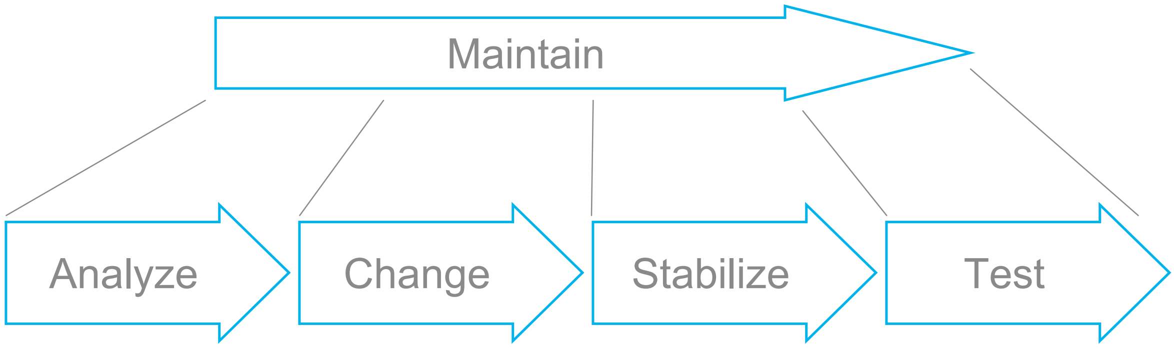


ISO 9126, Part 1
Maintainability (= evolvability)

Software Improvement Group

Maintainability =

- *Analyzability*: easy to understand where and how to modify?
- *Changeability*: easy to perform modification?
- *Stability*: easy to keep coherent when modifying?
- *Testability*: easy to test after modification?

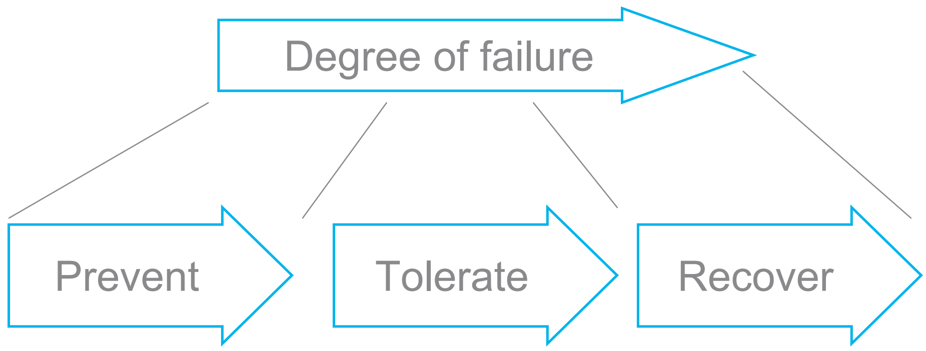


ISO 9126, Part 1
Reliability

Software Improvement Group

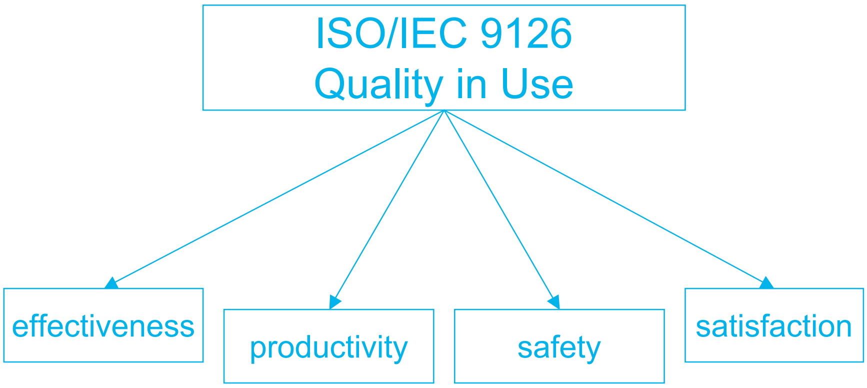
Reliability =

- *Maturity*: how much has been done to prevent failures?
- *Fault tolerance*: when failure occurs, is it fatal?
- *Recoverability*: when fatal failure occurs, how much effort to restart?



ISO/IEC 9126, Part 1
Product quality model: quality-in-use

Software Improvement Group



ISO 9126
Part 2,3: metrics

Software Improvement Group

External metrics, e.g.:

- Changeability: “change implementation elapse time”,
  time between diagnosis and correction
- Testability: “re-test efficiency”, time between correction and conclusion of test

Internal metrics, e.g.:

- Analysability: “activity recording”,
  ratio between actual and required number of logged data items
- Changeability: “change impact”,
  number of modifications and problems introduced by them

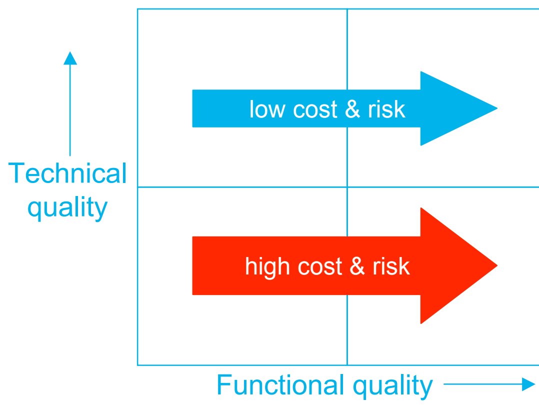
Critique

- Not pure *product* measures, rather *product in its environment*
- Measure *after* the fact
- No clear distinction between functional and technical quality

The issue

- Companies innovate and change
- Software systems need to adapt in the same pace as the business changes
- Software systems that do not adapt lose their value
- The technical quality of software systems is a key element

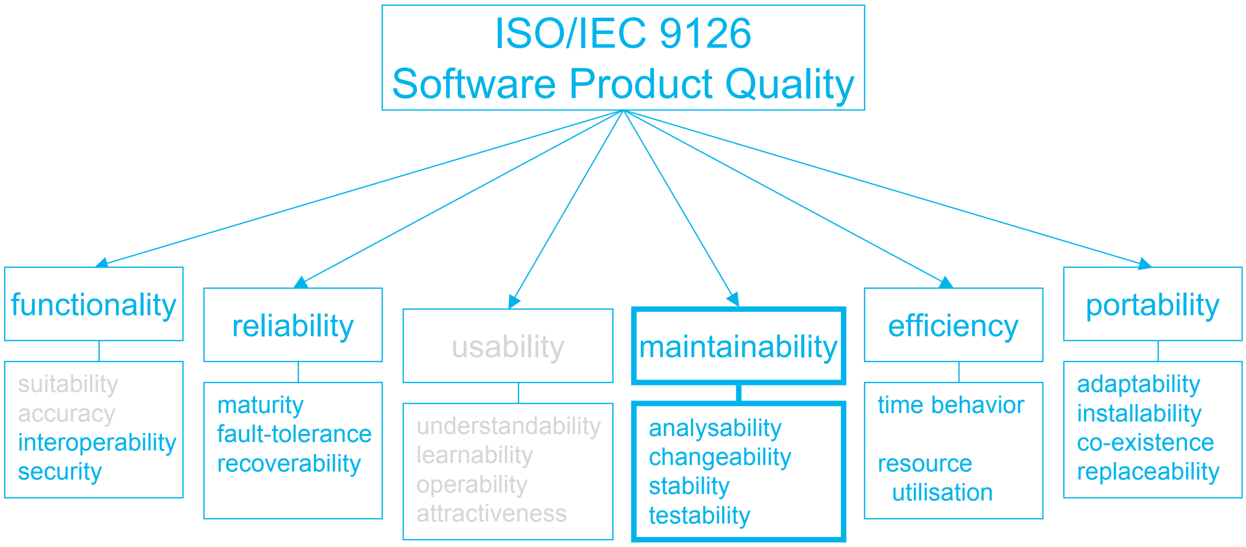
Functional *vs* technical quality



Software with high technical quality can evolve with low cost and risk to keep meeting functional and non-functional requirements.

Software Analysis and Testing, MFES Universidade do Minho by Joost Visser, Software Improvement Group © 2007.

ISO/IEC 9126, Part 1
Product quality model: technical quality



So …

What is software quality?

What are the functional and technical aspects of quality?

How can technical quality be measured?

A Challenge

Software Improvement Group

Use source code metrics to measure technical quality?

Plenty of metrics defined in literature

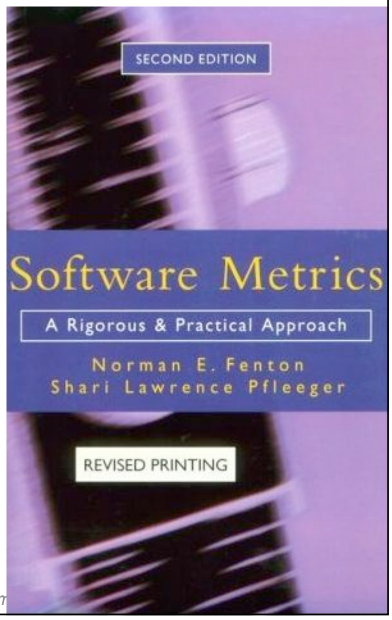
- LOC, cyclomatic complexity, fan in/out, coupling, cohesion, …
- Halstead, Chidamber-Kemener, Shepperd, …

Plenty of tools available

- Variations on Lint, PMD, FindBugs, …
- Coverity, FxCop, Fortify, QA-C, Understand, …
- Integrated into IDEs

But:

- Do they measure technical quality of a system?



Source code metrics
Lines of code (LOC)

Software Improvement Group

- Easy! Or …
- SLOC = Source Lines of Code
  - Physical (≈ newlines)
  - Logical (≈ statements)
- Blank lines, comment lines, lines with only “}”
- Generated *versus* manually written
- Measure effort / productivity: specific to programming language

Source code metrics
Function Point Analysis (FPA)

- A.J. Albrecht - IBM - 1979
- Objective measure of functional size

- Counted manually
  - IFPUG, Nesma, Cocomo
  - Large error margins

- Backfiring
  - Per language correlated with LOC
  - SPR, QSM

- Problematic, but popular for estimation

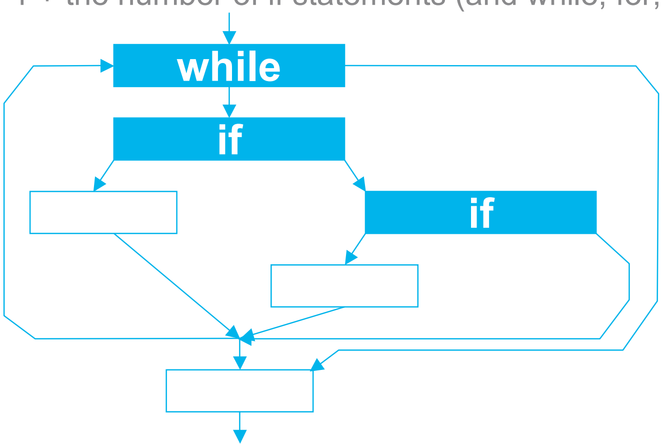
Table 2. Sample Function Point Calculations

Raw Data	Weights		Function Points
1 Input	X 4	=	4
1 Output	X 5	=	5
1 Inquiry	X 4	=	4
1 Data File	X 10	=	10
1 Interface	X 7	=	7

Unadjusted Total			30
Compexity Adjustment			None
Adjusted Function Points			30

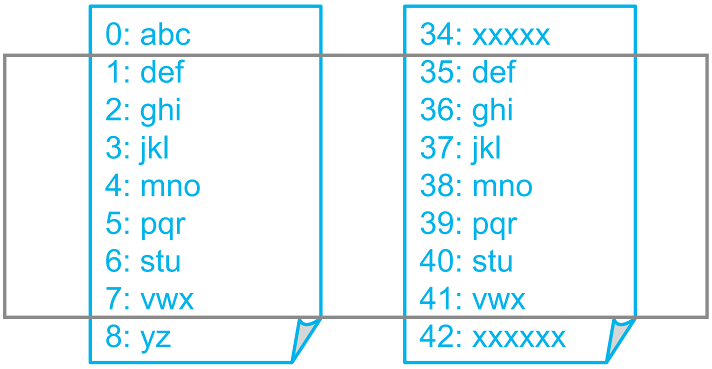
Source code metrics
Cyclomatic complexity

- T. McCabe, *IEEE Trans. on Sw Engineering*, 1976
- Accepted in the software community
- Number of independent, non-circular paths per method
- Intuitive: number of decisions made in a method
- 1 + the number of if statements (and while, for, ...)



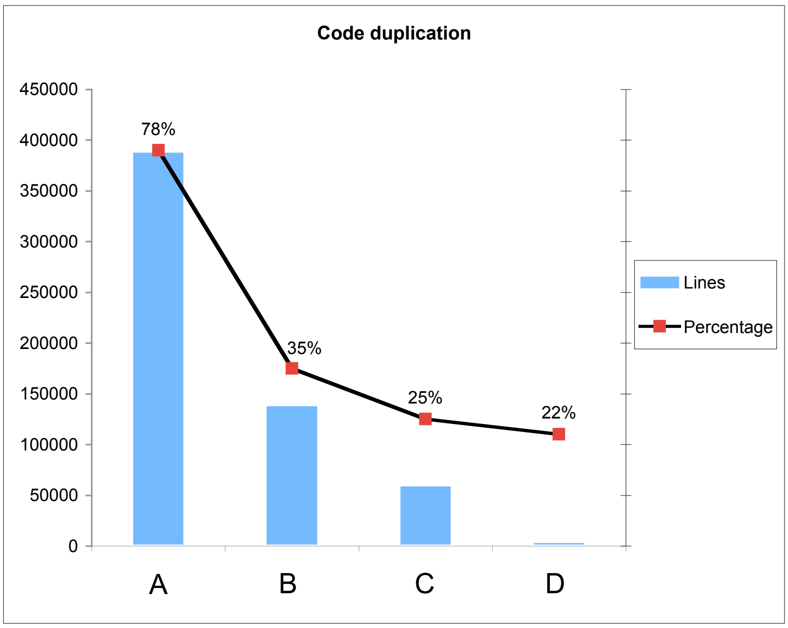
Code duplication
Definition

Code duplication measurement



Number of duplicated lines: 14

Code duplication



Source code metrics
Coupling

- Efferent Coupling (Ce)
  - How many classes do I depend on?
- Afferent Coupling (Ca)
  - How many classes depend on me?
- Instability = Ce/(Ca+Ce) ∈ [0,1]
  - Ratio of efferent *versus* total coupling
  - 0 = very stable = hard to change
  - 1 = very instable = easy to change

Figure 1. Coupling graph

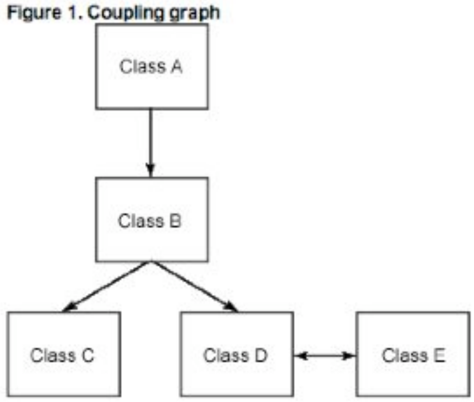
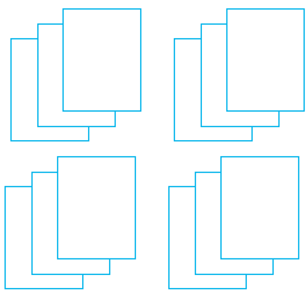


Table 1. Results of compiling a single class

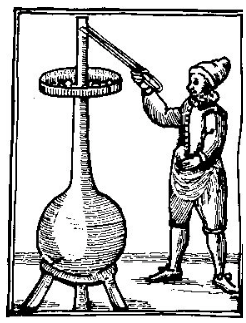
Class to Compile	Other Classes Compiled	Afferent Couplings	Efferent Couplings	Instability Factor
A	B,C,D,E	0	4	1
B	C,D,E	1	3	0.75
C	-	2	0	0
D	E	3	1	0.25
E	D	3	1	0.25

A Challenge
Do metrics measure technical quality?



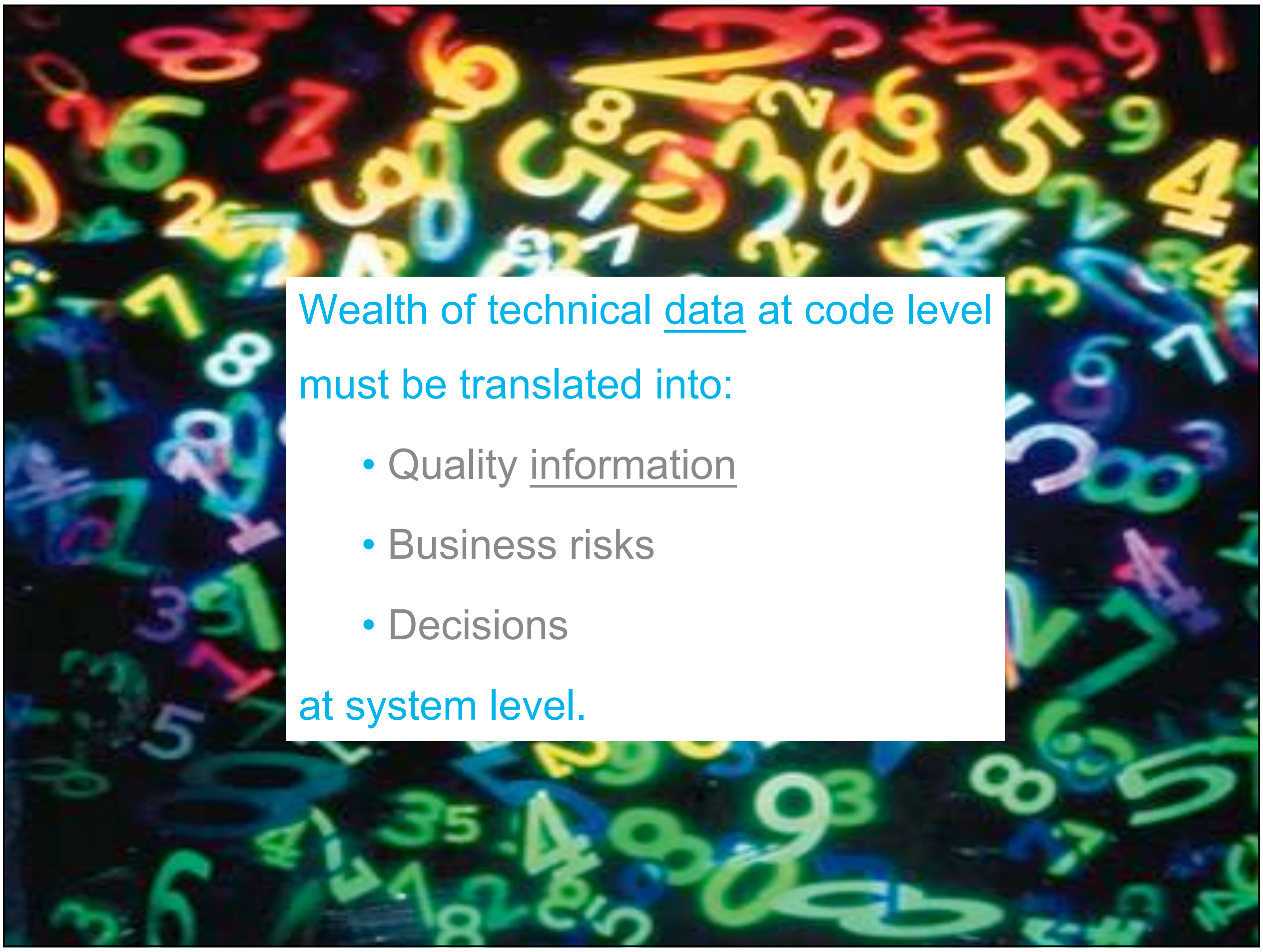
500.000 LOC Java code

+



source code analyzer

= ?


Wealth of technical data at code level
must be translated into:
• Quality information
• Business risks
• Decisions
at system level.

Source code metrics
Cyclomatic complexity

Software Improvement Group

- T. McCabe, *IEEE Trans. on Sw Engineering*, 1976
- Accepted in the software community
- Academic: number of independent paths per method
- Intuitive: number of decisions made in a method
- Really, the number of if statements (and while, for, ...)
- Software Engineering Institute:

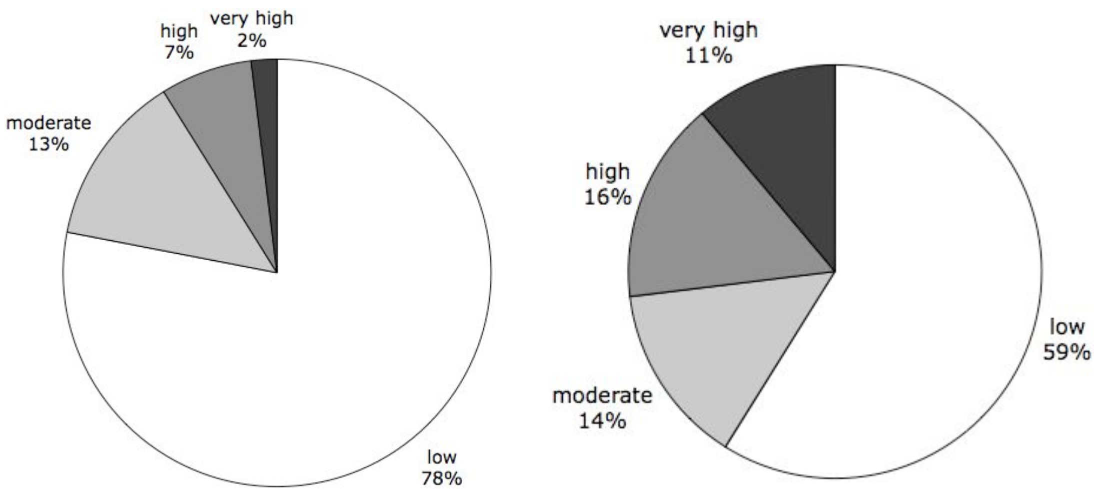
Table 4: Cyclomatic Complexity

Cyclomatic Complexity	Risk Evaluation
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk program
greater than 50	untestable program (very high risk)

Complexity per unit
Quality profiles

Software Improvement Group

Aggregation by averaging is fundamentally flawed



Software Analysis and Testing, MFES Universidade do Minho by Joost Visser, Software Improvement Group © 2007.

Quality profiles, in general

Input

- type Input metric = Map x (metric,LOC)

Risk groups

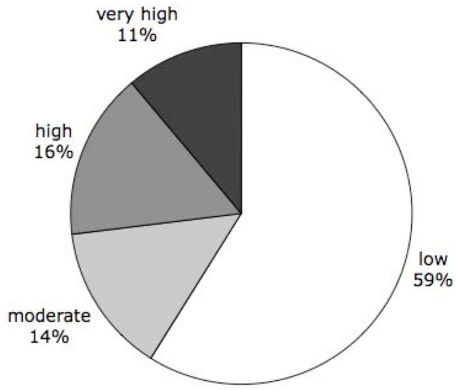
- type Risk = Low | Moderate | High | Very High
- risk :: metric → Risk

Output

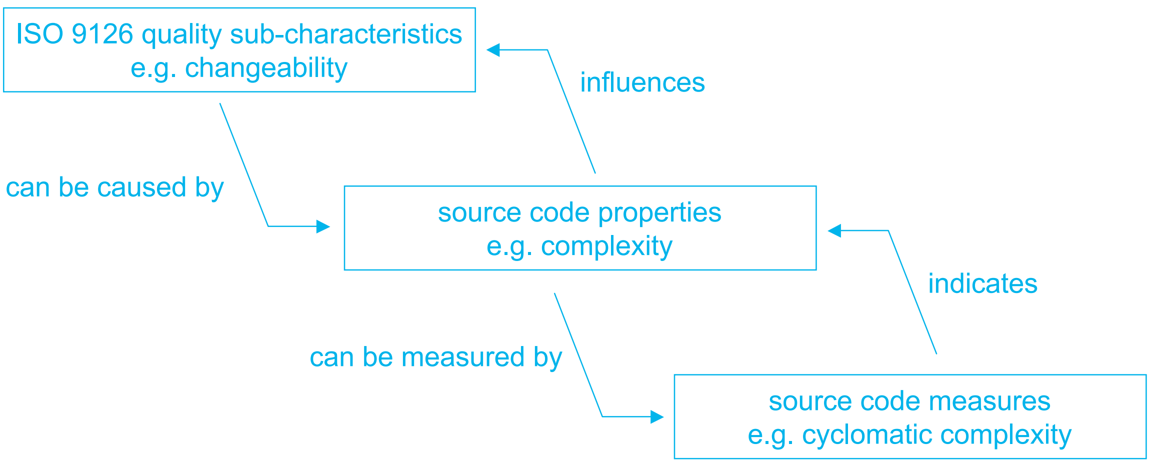
- type ProfileAbs = Map Risk LOC
- type Profile = Map Risk Percentage

Aggregation

- profile :: Input metric → Profile



Combining metrics
The SIG approach



Mapping source code properties onto quality sub-characteristics

	Volume	Complexity	Unit size	Duplication	Unit testing	
Analysability	X		X	X		
Changeability		X		X		
Stability					X	
Testability		X	X		X	

Source code properties and metrics

Volume

- LOC, within the context of a single language
- Man years via backfiring function points

Complexity per unit

- McCabe's cyclomatic complexity, SEI risk categories, %LOC for each category

Duplication

- Duplicated blocks, threshold 6 lines, %LOC

Unit size

- LOC, risk categories, %LOC for each category

Unit testing

- Unit test coverage
- Number of assert statements (as validation)

Software Analysis and Testing, MFES Universidade do Minho by Joost Visser, Software Improvement Group © 2007.

Duplication

Software Improvement Group

	Volume	Complexity	Unit size	Duplication	Unit testing	
				★★★☆☆		
Analysability	X		X	X		
Changeability		X		X		
Stability					X	
Testability		X	X		X	

Duplicate blocks

- Over 6 lines
- String comparison
- Remove leading spaces

Rank	duplication
★★★★★	0-3%
★★★★☆	3-5%
★★★☆☆	5-10%
★★☆☆☆	10-20%
★☆☆☆☆	20-100%

Complexity

Software Improvement Group

	Volume	Complexity	Unit size	Duplication	Unit testing	
		★★★☆☆				
Analysability	X		X	X		
Changeability		X		X		
Stability					X	
Testability		X	X		X	

Software Engineering Institute

complexity	risk
1-10	low
11-20	medium
21-50	high
>50	very high

	Maximum relative LOC		
Rank	moderate	high	very high
★★★★★	25%	0%	0%
★★★★☆	30%	5%	0%
★★★☆☆	40%	10%	0%
★★☆☆☆	50%	15%	5%
★☆☆☆☆	-	-	-

Rating example

	Volume	Complexity	Unit size	Duplication	Unit testing	
	★★★★★	★☆☆☆☆	★★☆☆☆	★★☆☆☆	★★★☆☆	
Analysability	X		X	X		★★★☆☆
Changeability		X		X		★★☆☆☆
Stability					X	★★★☆☆
Testability		X	X		X	★★☆☆☆

That's all?

Practical

- Fast, repeatable, technology independent
- Sufficiently accurate for our purposes
- Explainable

Beyond core model ...

- Only one instrument in Software Risk Assessments and Software Monitor
- Weighting schemes
- Dynamic analysis
- Quality of process, people, project

Further reading

See

I. Heitlager, T. Kuipers, J. Visser.
A pragmatic model for measuring maintainability.
QUATIC 2007.

What is software quality? ✓

What are the technical aspects of quality? ✓

How can technical quality be measured? ✓

Projects

"interpretation of source code clusters"

"evaluation of QuickCheck solutions for Java"

Interpretation of Source Code Clusters

Clustering
- Data mining technique
- Input: N measurement values for each item
- Groups together "similar" items based on measurement values

Problem
- Apply to source code metrics of large software system
- **BUT**: clusters have no "meaning"

Solution
- ISO/IEC 9126: standard for software product quality
- Give clusters meaning by scoring against ISO 9126 using SIG method

Fun
- Apply your tool to REAL BIG systems from our clients!

Further reading

See
Yiannis Kanellopoulos, I. Heitlager, J. Visser.
Interpretation of source code clusters in terms of ISO 9126 quality characteristics.
Draft.

Case 1
Curbing Erosion

System

- About 15 years old
- Automates primary business process
- Maintenance has passed through various organizations
- New feature requests at regular intervals

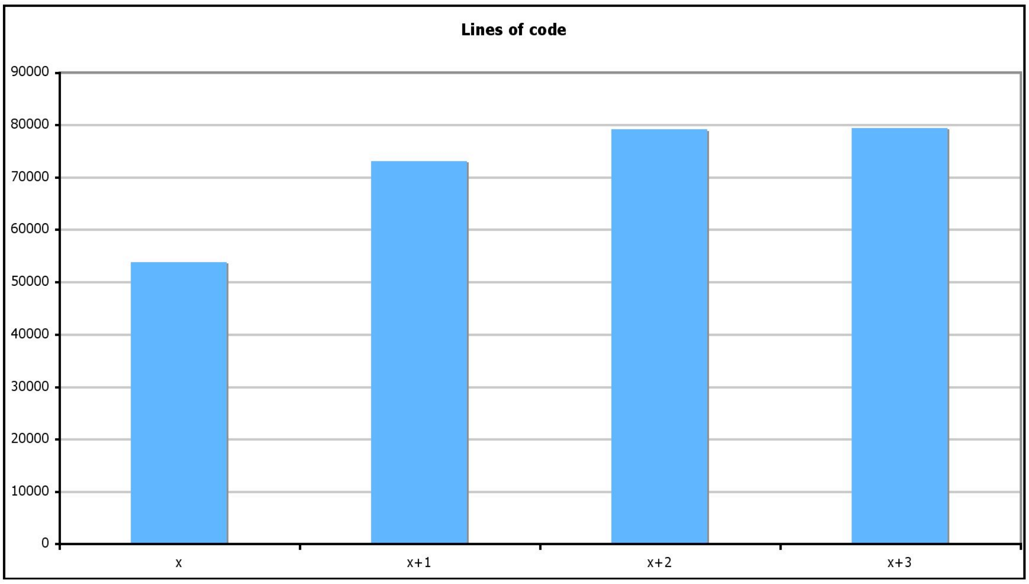
Questions

- Improve management's control over quality and associated costs

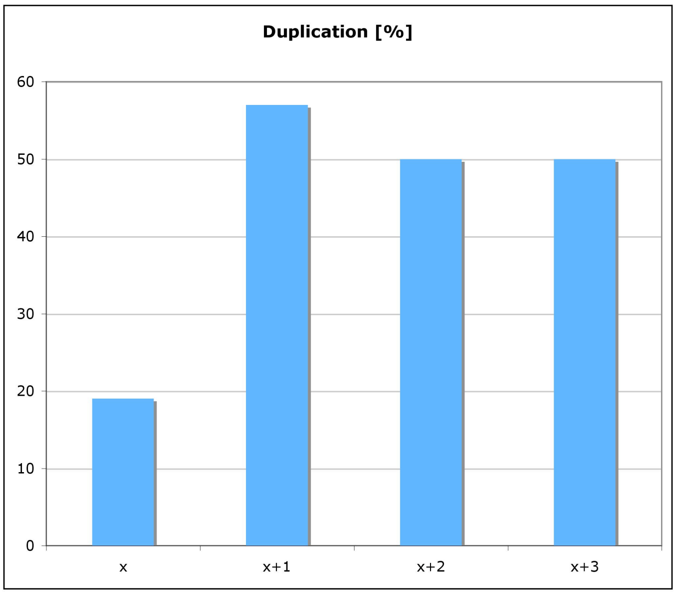
Metrics in this example

- Volume
- Duplication

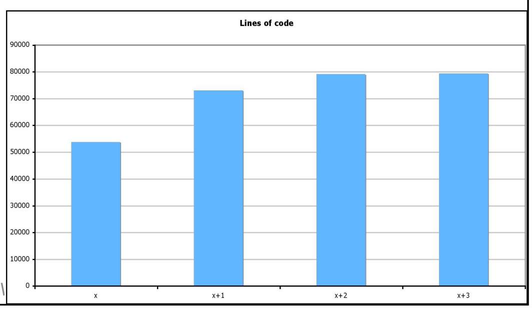
Case 1
Curbing Erosion



Case 1
Curbing erosion



- All growth is caused by duplication
- There is no “real” productivity



Case 2
Systems accounting from code churn

System

- 1.5 MLOC divided over 7000 files
- Estimated 240 people divided over 25 subcontractors

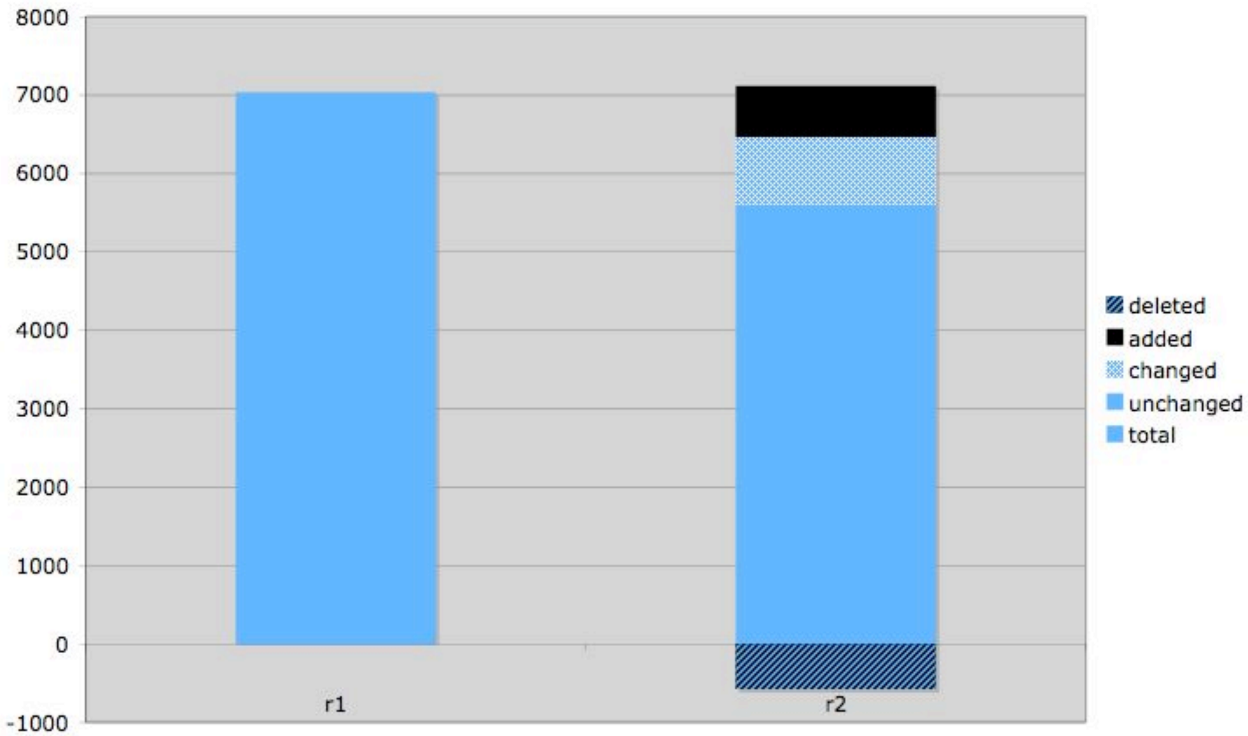
Questions

- Is staffing justified?

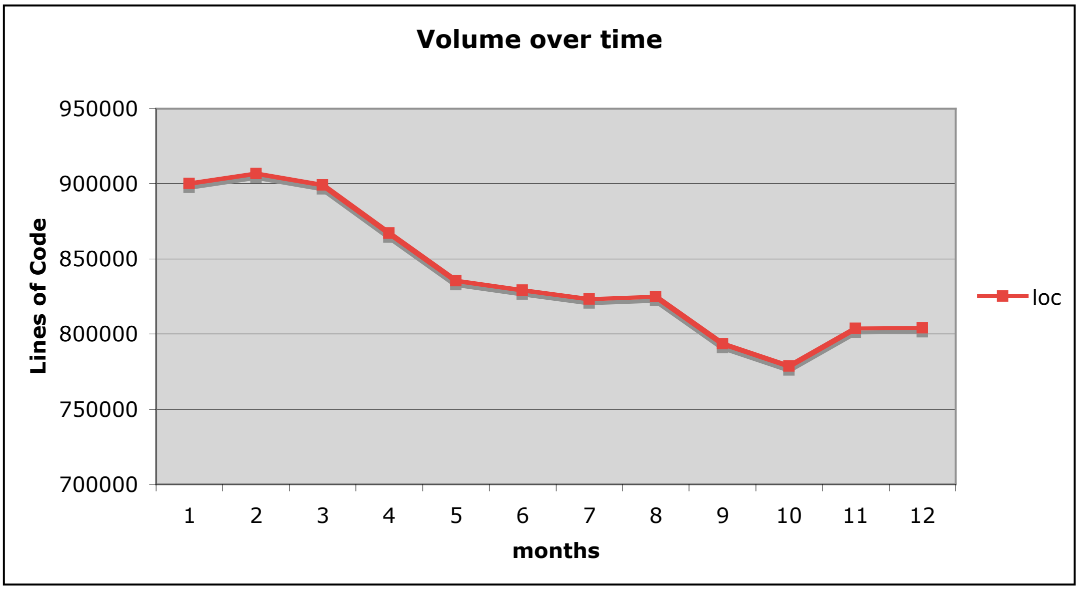
Metrics in this example

- Code churn = number of added, changed, deleted LOC

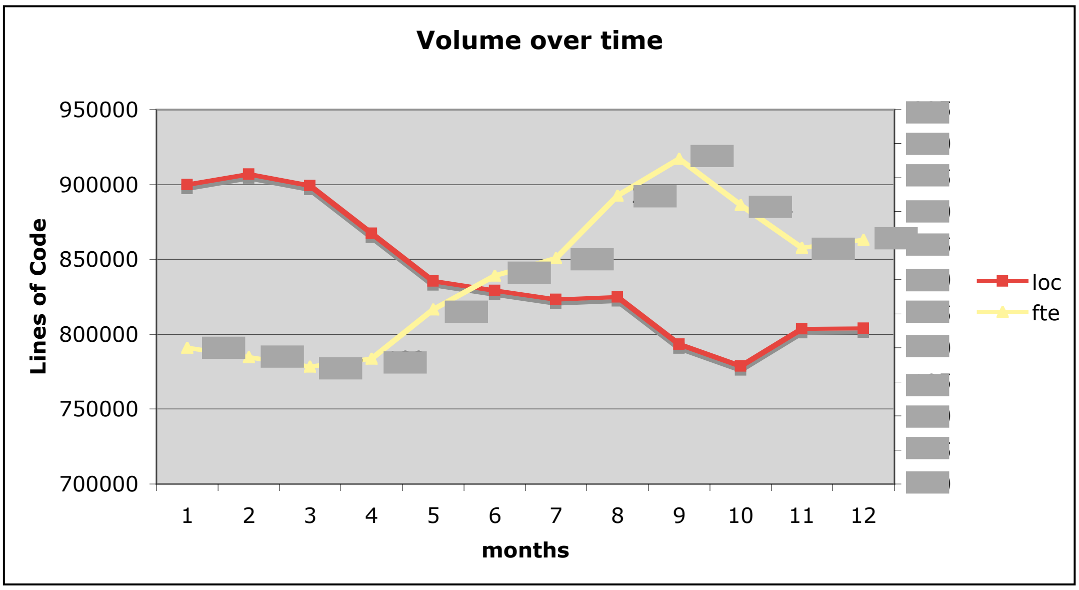
Case 2
Systems accounting from code churn



Case 2
Systems accounting from code churn



Case 2
System accounting from code churn



Case 3
Learn from failure

Software Improvement Group

System

- Electronic commerce
- Replacement for functionally identical system which failed in rollout
- Outsourced development

Questions

- Monitor productivity and quality delivered by the developer

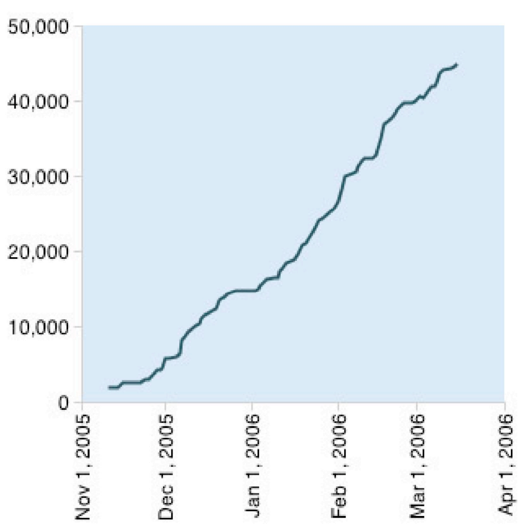
Metrics in this example

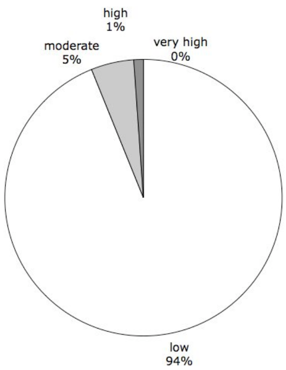
- Volume
- Complexity

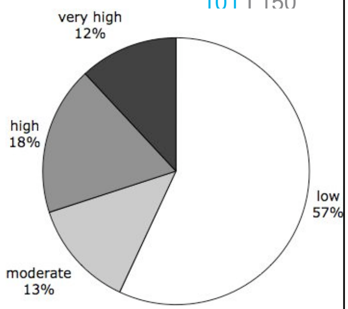
Case 3
Learn from failure

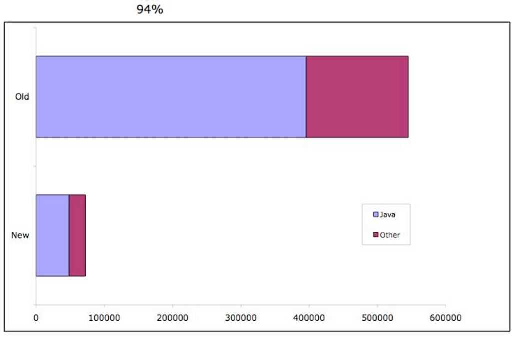
Software Improvement Group

Total lines sources









What should you remember (so far) from this lecture?

Software Improvement Group

Testing

- Automated unit testing!

Patterns

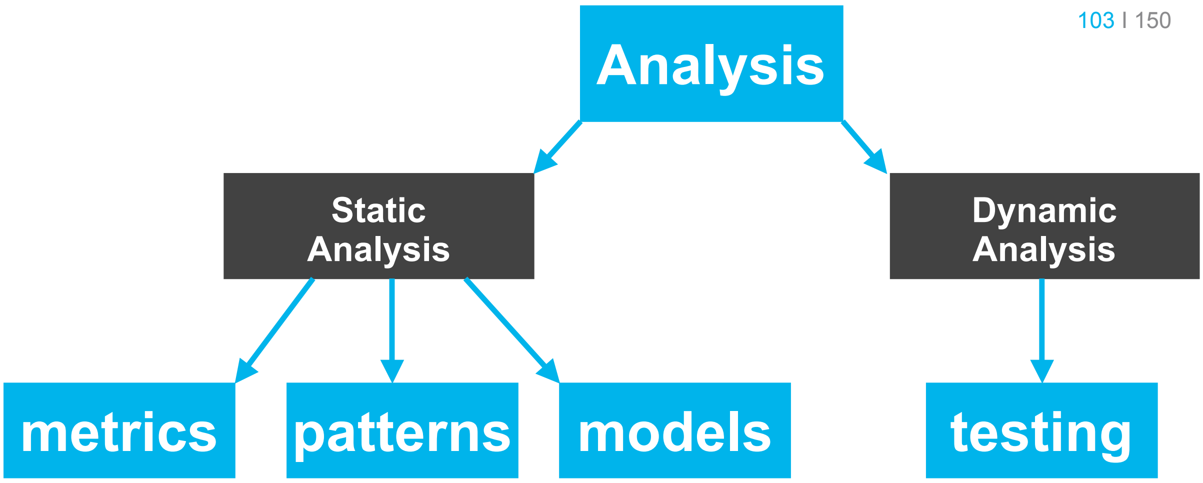
- Run tools!

Quality and metrics

- Technical quality matters in the long run
- A few simple metrics are sufficient
- If aggregated in well-chosen, meaningful ways
- The simultaneous use of distinct metrics allows zooming in on root causes

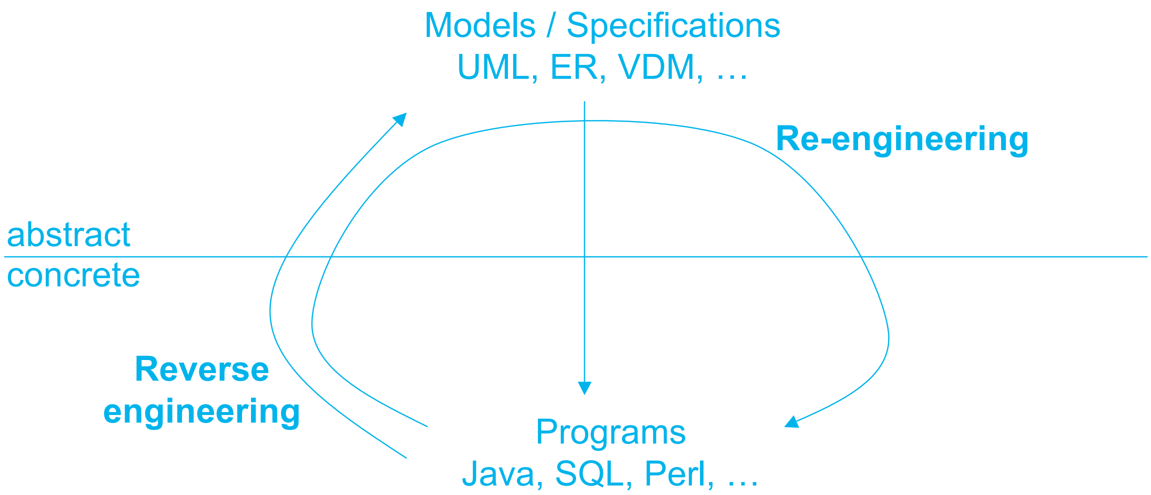
Structure of the lectures

Software Improvement Group



REVERSE ENGINEERING

Terminology



Reverse engineering

Dependencies and graphs

- Extraction, manipulation, presentation
- Graph metrics
- Slicing

Advanced

- Type reconstruction
- Concept analysis
- Programmatic join extraction

Reverse engineering trinity

Extraction

From program sources, extract basic information into an initial source model.

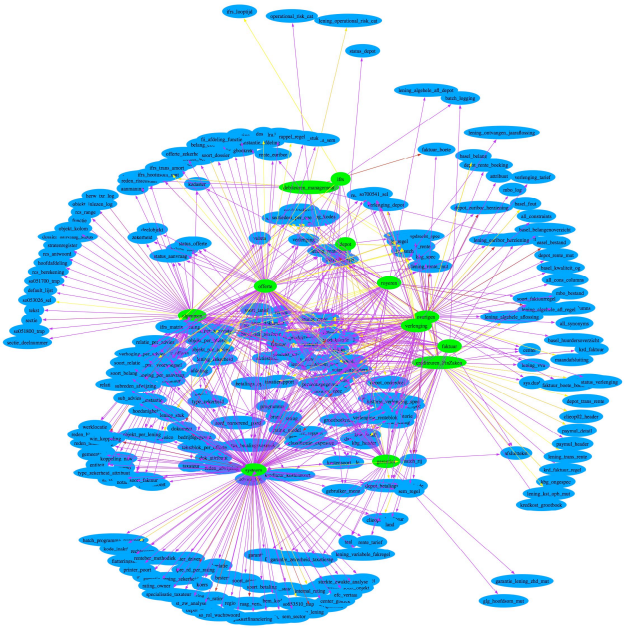
Manipulation

Combine, condense, aggregate, or otherwise process the basic information to obtain a derived source model.

Presentation

Visualize or otherwise present source models to a user.

Example



Green oval = module

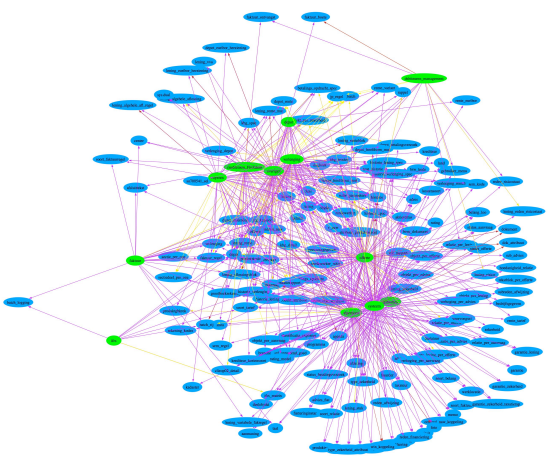
Blue oval = table

Purple arrow = select operation

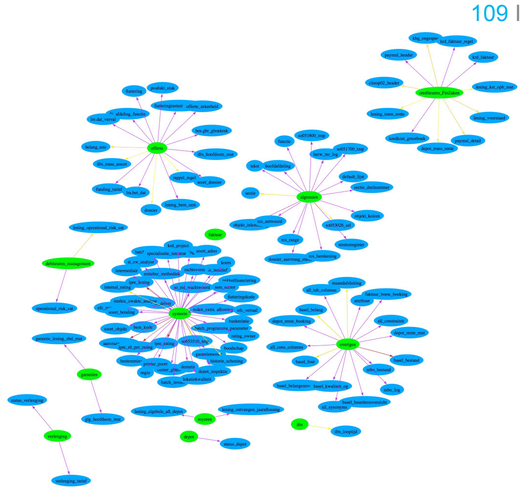
Yellow arrow = insert/update operation

Brown arrow = delete operation

Example



Tables used by multiple modules.



Tables used by a single module.

Relations and graphs

Relation

```
type Rel a b = Set (a,b)
```
set of pairs

Graph

```
type Gph a = Rel a a
```
endo-relation

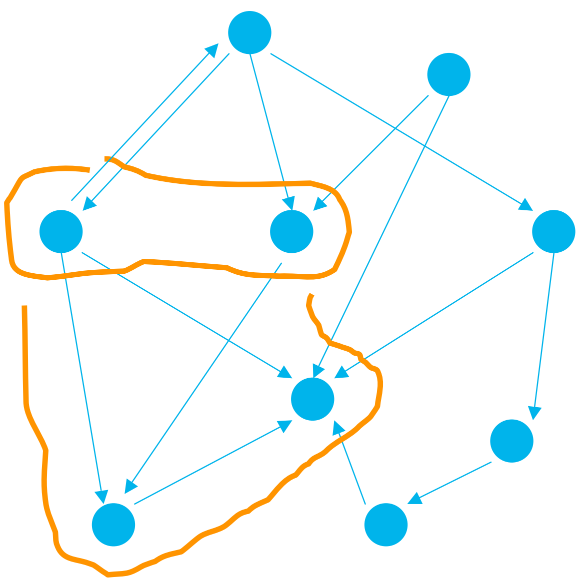
Labeled relation

```
type LRel a b l = Map (a,b) l
```
map from pairs

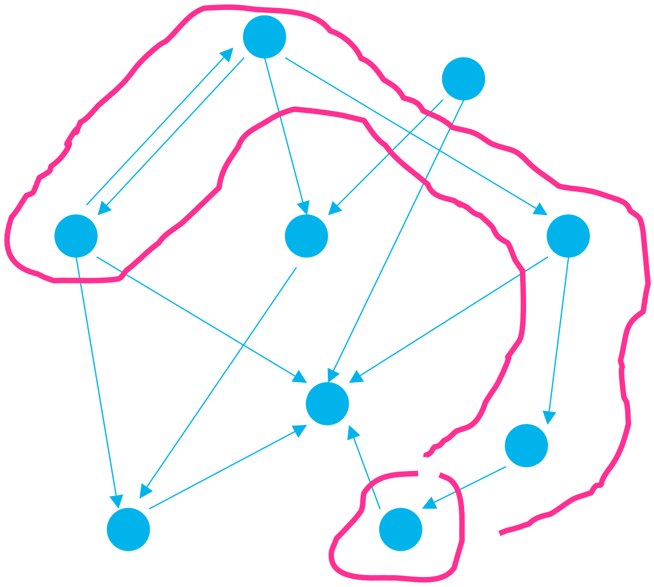
Note

```
Rel a b = Set(a,b)= Map(a,b)()= LRel a b ()
```

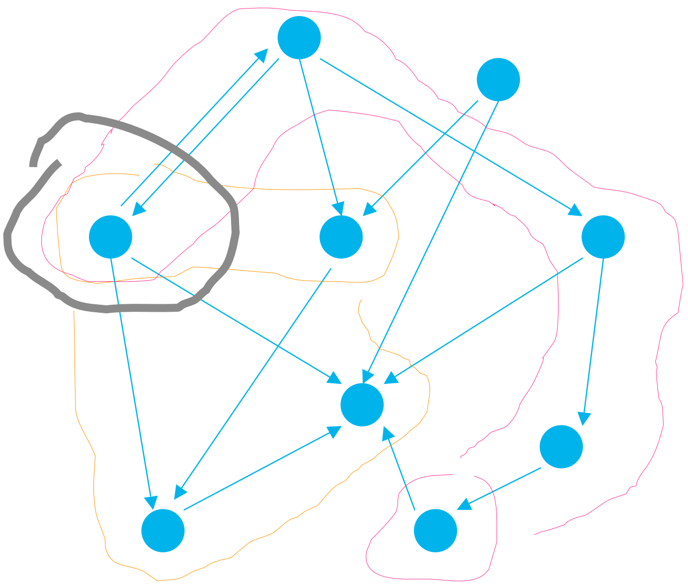

Slicing (forward)



Slicing (backward)

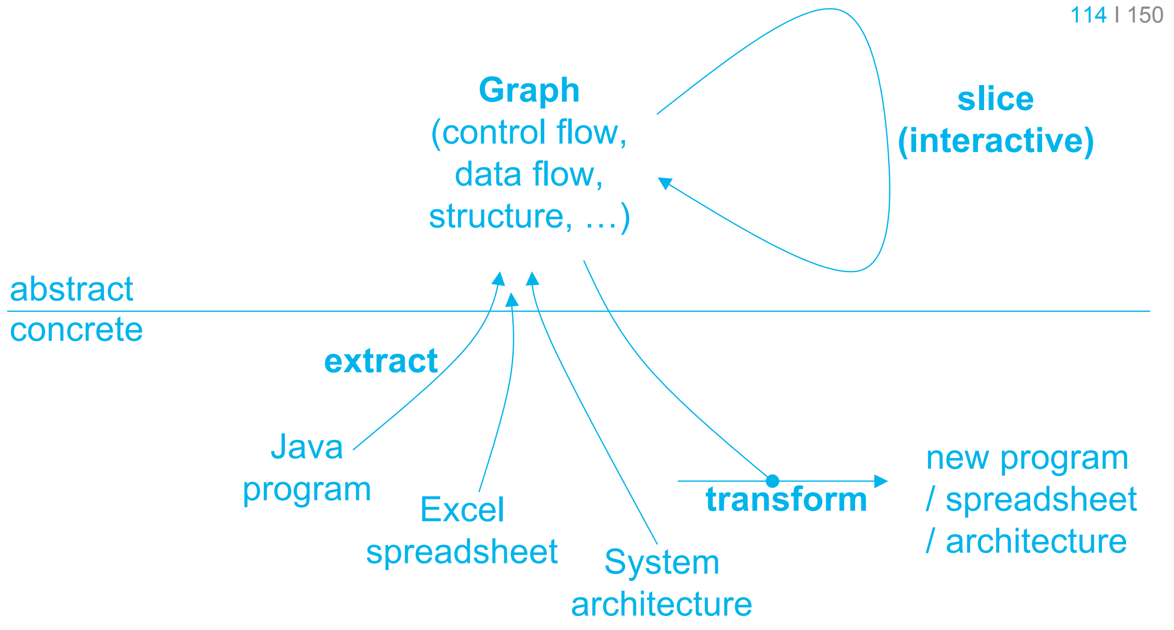


Chop = Forward ∩ Backward



Generic slicing

Software Improvement Group



Further reading

Software Improvement Group

See

Arun Lakhotia.
Graph theoretic foundations of program slicing and integration.
The Center for Advanced Computer Studies, University of Southwestern Louisiana.
Technical Report CACS TR-91-5-5, 1991.

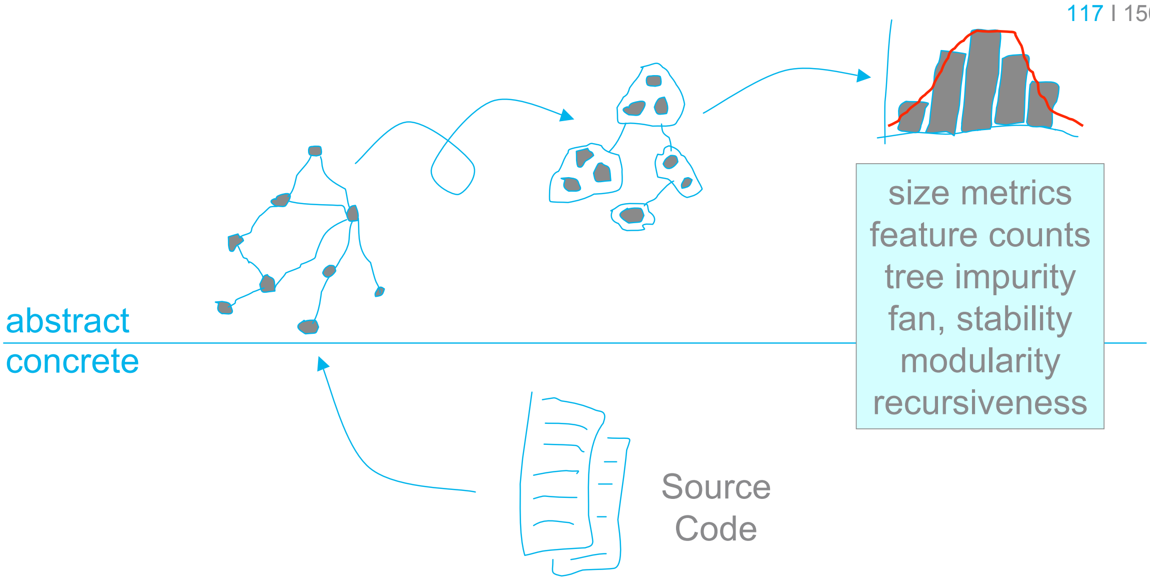
Questions about graphs

Software Improvement Group

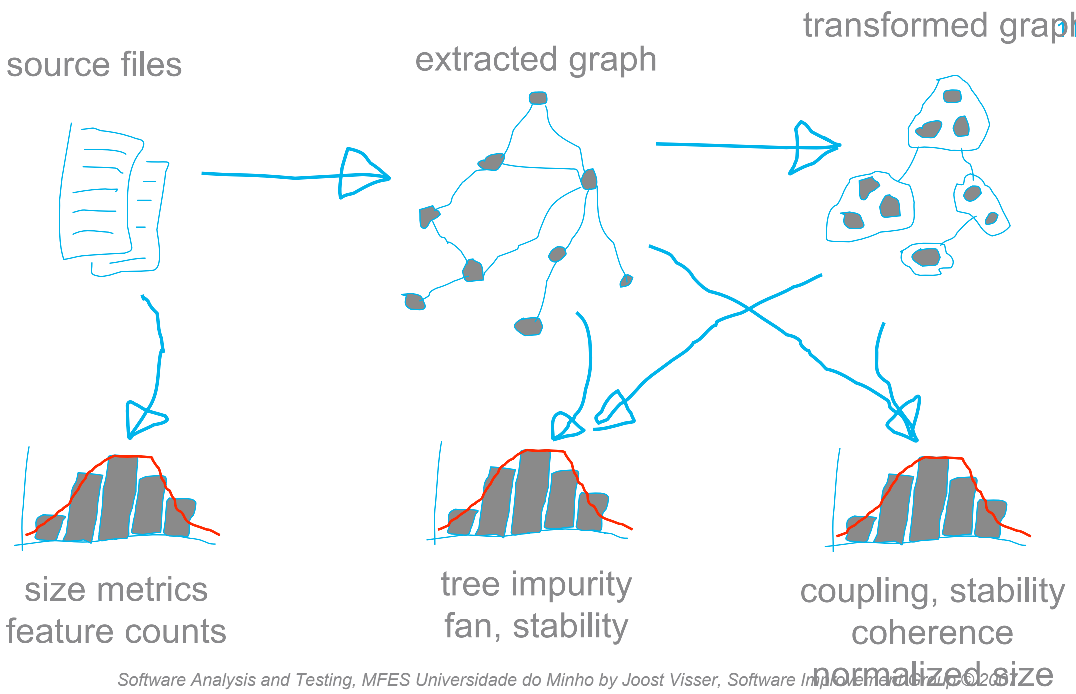
How big?
How complex?
Recursive? To what degree?
Modular? To what degree?
Degree of internal reuse?

Graph metrics

Software Improvement Group



Software Analysis and Testing, MFES Universidade do Minho by Joost Visser, Software Improvement Group © 2007.



Example
source = XML Schema


```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Emp" type="EmployeeType"/>
  <xs:complexType name="EmployeeType">
    <xs:sequence>
      <xs:element name="Emp" type="EmployeeType"/>
    </xs:sequence>
    <xs:attribute name="EmployeeID" type="xs:ID"/>
    <xs:attribute name="FirstName" type="xs:string"/>
    <xs:attribute name="LastName" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

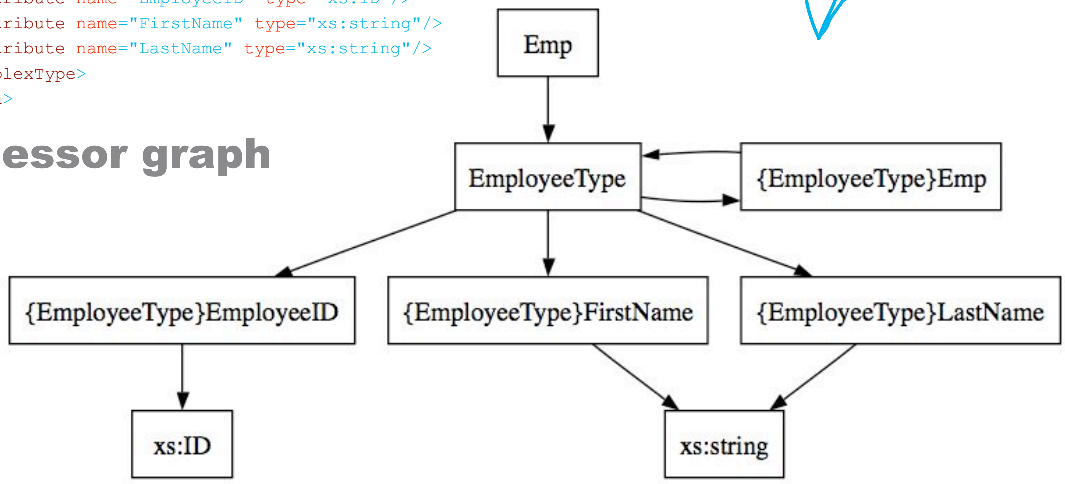
Adapted from the online .NET Framework Developer's Guide at http://msdn.microsoft.com/library/.

Example

Software Improvement Group

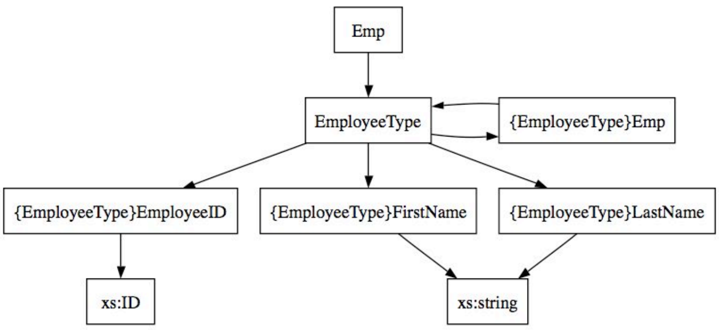

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Emp" type="EmployeeType"/>
  <xs:complexType name="EmployeeType">
    <xs:sequence>
      <xs:element name="Emp" type="EmployeeType"/>
    </xs:sequence>
    <xs:attribute name="EmployeeID" type="xs:ID"/>
    <xs:attribute name="FirstName" type="xs:string"/>
    <xs:attribute name="LastName" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Successor graph



Example

Software Improvement Group



Successor graph

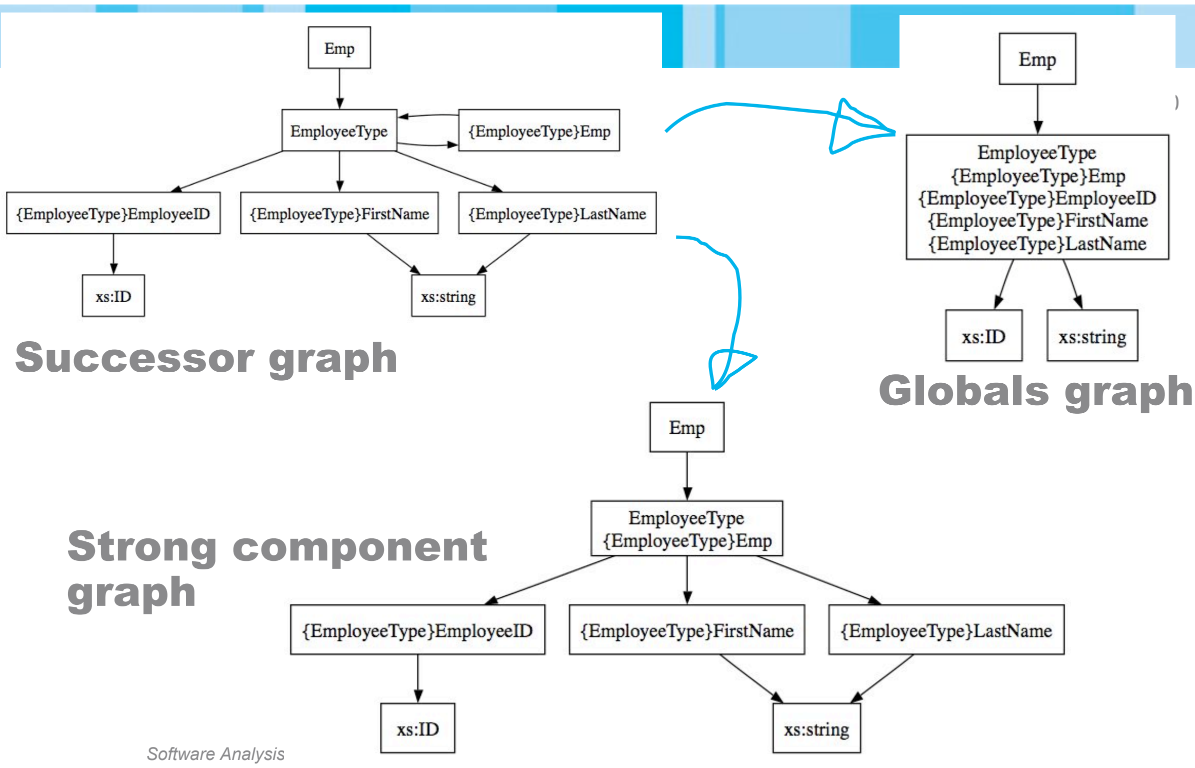
Internal reuse — Tree impurity = $\frac{2(e-n+1)}{(n-1)(n-2)}$ = 4.76%

Spot extremes { Fan-in$_{max}$ = 2; Fan-out$_{max}$ = 4

Ease of change — Instability$_{avg}$ = $\frac{fanout}{fanout + fanin}$ = 45.8%

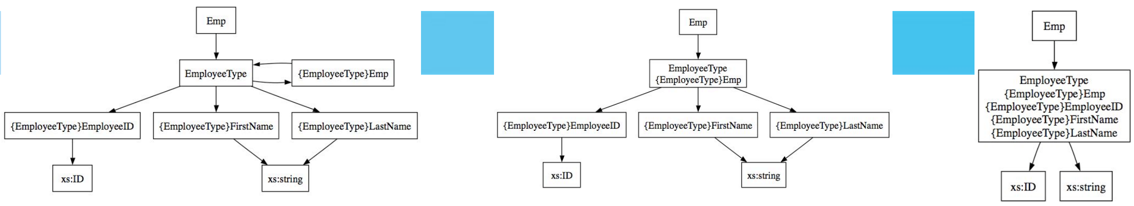
Example

Software Improvement Group



Example: transformed graphs

Software Improvement Group

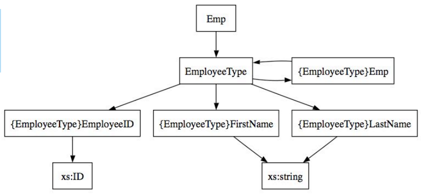


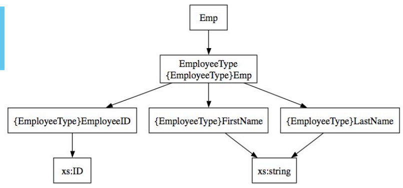
150

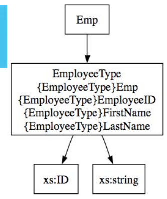
	Successors	**Strong components**	**Globals**
Tree impurity	= 4.76%	= 6.67%	= 0%
Fan-in$_{max}$	= 2	= 2	= 1
Fan-out$_{max}$	= 4	= 3	= 2
Fan instability$_{avg}$	= 45.8%	= 46.4%	= 41.7%
Afferent coupling$_{max}$		= 2	= 2
Efferent coupling$_{max}$		= 3	= 3
Coupling instability$_{avg}$		= 46.43%	= 43.75%
Coherence$_{avg}$		= 90.5%	= 88.9%
Tree impurity$_{avg}$		= 85.7%	= 75.0%

Example: transformed graphs

Software Improvement Group







	Successors	Strong components	Globals
Tree impurity	= 4.76%	= 6.67%	= 0%
Fan instability$_{avg}$	= 45.8%	= 46.4%	= 41.7%
Coupling instability$_{avg}$		= 46.43%	= 43.75%
Coherence$_{avg}$		= 90.5%	= 88.9%
Tree impurity$_{avg}$		= 85.7%	= 75.0%
Node count$_{max}$		= 2	= 5
Normalized group count		= 85.7%	= 50.0%
		Recursion	Encapsulation

Schemas sampled

Software Improvement Group

Google

AMBER ALERT

DIVA

CODEWARE

amazon.com

ebay

IBM

OASIS

DSML
ebXML
SAML
SPML
UBL
UDDI v2, v3
XACML v1,v3

GPX

MARC 21

W3C

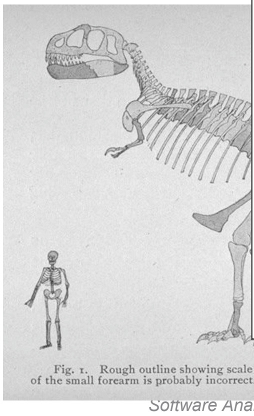
MathML
XML Schema

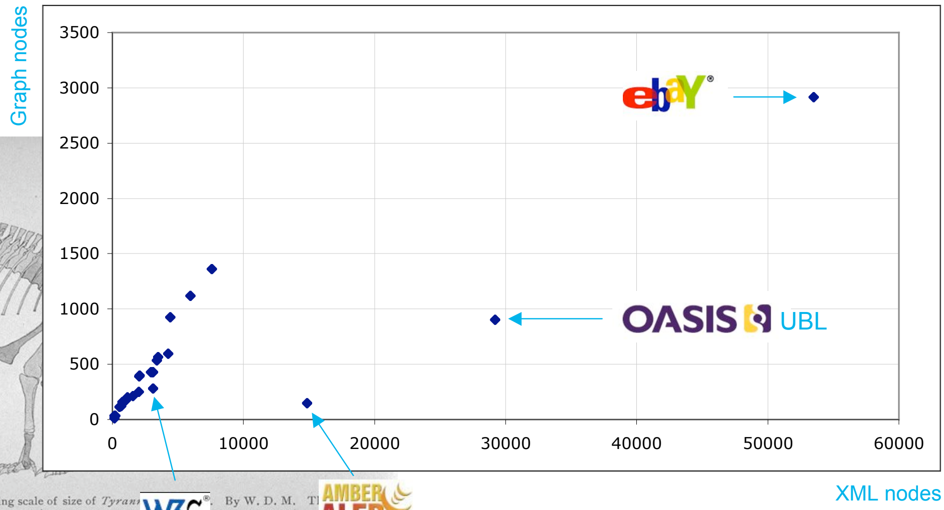
Streamlined Sales Tax Governing Board, Inc.

How big?

Software Improvement Group

	avg	max
Bytes	**199.749**	**2.170.773**
XML nodes	**5.857**	**53.493**
Graph nodes	**497**	**2.916**

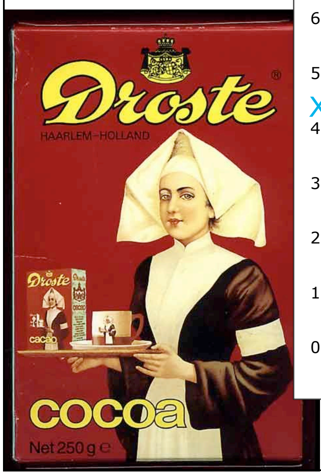


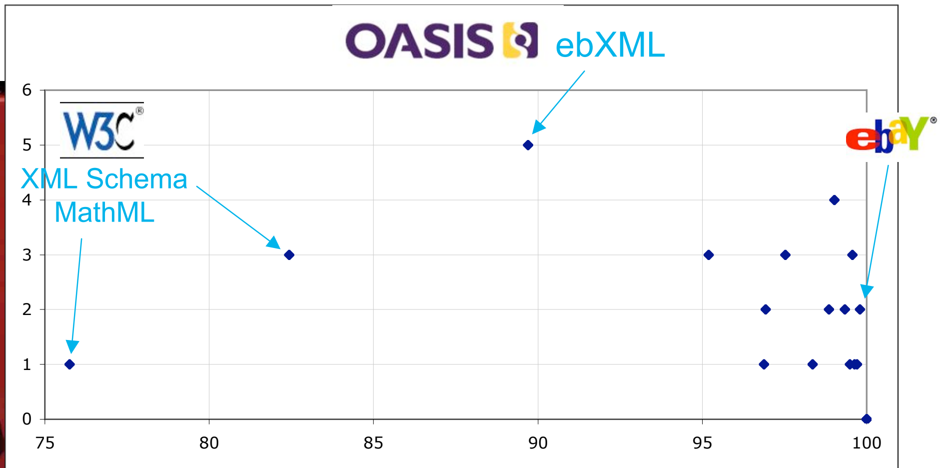


How much recursion?

Software Improvement Group

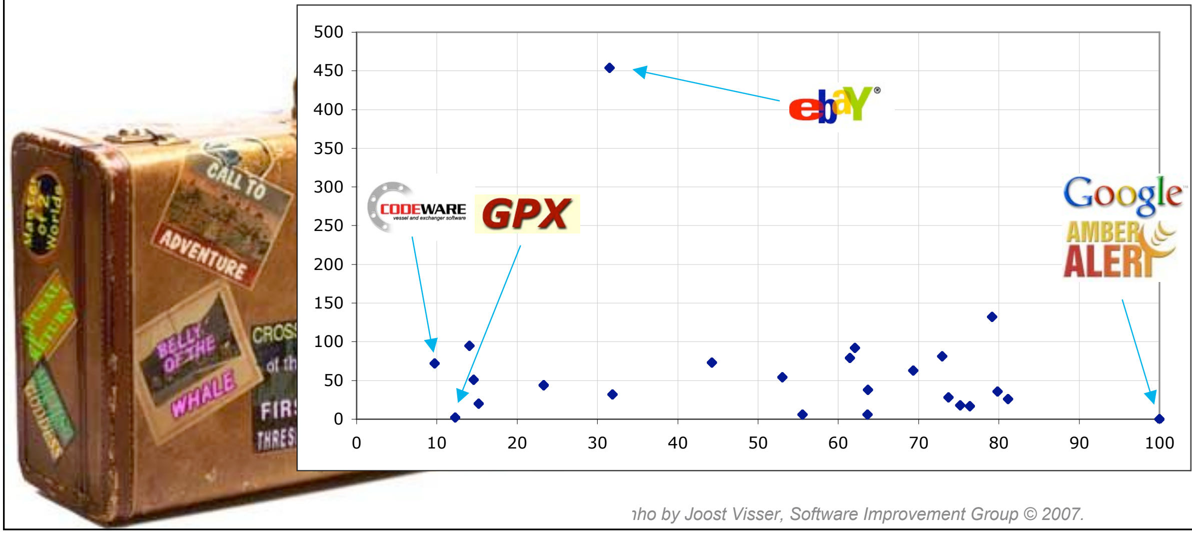
61.5% contain recursion	avg	min	max
Normalized group count	**97.2%**	**75.8%**	**100%**
Non-singleton groups	**1.35**	**0**	**5**





How much encapsulation

	min	avg	max
Normalized group count	**9.71%**	**53.4%**	**100%**
Non-singleton groups	**0**	**60.1**	**454**



Further reading

See

Joost Visser.
Structure Metrics for XML Schema.
XATA 2006.

See

- Arie van Deursen and Leon Moonen. *An empirical Study Into Cobol Type Inferencing*. Science of Computer Programming **40**(2-3):189-211, July 2001

Basic idea

1. Extract basic relations (entities are variables)
   - assign: ex. `a := b`
   - expression: ex. `a <= b`
   - arrayIndex: ex. `A[i ]`
2. Compute derived relations
   - typeEquiv: variables belong to the same type
   - subtypeOf: variables belong to super/subtype
   - extensional notion of type: set of variables

Type reconstruction
(from type-less legacy code)

Pseudo code from paper

arrayIndexEquiv := $\text{arrayIndex}^{-1} \circ$ arrayIndex

typeEquiv := arrayIndexEquiv $\cup$ expression

subtypeOf := assign

repeat

  subtypeEquiv := equiv(subtypeOf+ $\cap$ $(\text{subtypeOf+})^{-1}$)

  typeEquiv := equiv(typeEquiv $\cup$ subtypeEquiv)

  subtypeOf := subtypeOf \ typeEquiv

  subtypeOf := subtypeOf $\cup$ subtypeOf $\circ$ typeEquiv $\cup$ typeEquiv $\circ$ subtypeOf

until fixpoint of (typeEquiv, subtypeOf)

Software Analysis and Testing, MFES Universidade do Minho by Joost Visser, Software Improvement Group © 2007.

Type reconstruction (from type-less legacy code)

Data

```
type VariableGraph v array
  = (Rel v v, Rel v array, Rel v v)

type TypeGraph x
  = (Rel x x, Rel x x)   -- subtypes and type equiv
```

Operation

```
typeInference
  :: (Ord v, Ord array) =>
     VariableGraph v array -> TypeGraph v
```


Formal concept analysis

See

- Christian Lindig. *Fast Concept Analysis*. In Gerhard Stumme, editors, *Working with Conceptual Structures - Contributions to ICCS 2000*, Shaker Verlag, Aachen, Germany, 2000.

Basic idea

1. Given formal context
   - matrix of objects vs. properties
2. Compute concept lattice
   - a concept = (extent,intent)
   - ordering is by sub/super set relation on intent/extent

Used in many fields, including program understanding.

Formal concept analysis pseudo-code (1/2)

$\textsc{Neighbors}\ ((G, M), (\mathcal{G}, \mathcal{M}, \mathcal{I}))$

$\quad \text{min} \leftarrow \mathcal{G} \setminus G$

$\quad \text{neighbors} \leftarrow \emptyset$

$\quad \textbf{foreach}\ g \in \mathcal{G} \setminus G\ \textbf{do}$

$\quad\quad M_1 \leftarrow (G \cup \{g\})'$

$\quad\quad G_1 \leftarrow M_1'$

$\quad\quad \textbf{if}\ ((\text{min} \cap (G_1 \setminus G \setminus \{g\})) = \emptyset)\ \textbf{then}$

$\quad\quad\quad \text{neighbors} \leftarrow \text{neigbors} \cup \{(G_1, M_1)\}$

$\quad\quad \textbf{else}$

$\quad\quad\quad \text{min} \leftarrow \text{min} \setminus \{g\}$

$\quad \textbf{return}\ \text{neighbors}$

Note that _′ operation denotes computation of intent from extent, or vice versa, implicitly given a context.

Formal concept analysis pseudo-code (2/2)

$\textsc{Lattice}\ (\mathcal{G}, \mathcal{M}, \mathcal{I})$

$\quad c \leftarrow (\emptyset'', \emptyset')$

$\quad \text{insert}\ (c, L)$

$\quad \textbf{loop}$

$\quad\quad \textbf{foreach}\ x\ \textbf{in}\ \textsc{Neighbors}\ (c, (\mathcal{G}, \mathcal{M}, \mathcal{I}))$

$\quad\quad\quad \textbf{try}\ x \leftarrow \text{lookup}\ (x, L)$

$\quad\quad\quad \textbf{with}\ \text{NotFound} \rightarrow \text{insert}\ (x, L)$

$\quad\quad\quad x_* \leftarrow x_* \cup \{c\}$

$\quad\quad\quad c^* \leftarrow c^* \cup \{x\}$

$\quad\quad \textbf{try}\ c \leftarrow \text{next}\ (c, L)$

$\quad\quad \textbf{with}\ \text{NotFound} \rightarrow \textbf{exit}$

$\quad \textbf{return}\ L$

Transposition to Haskell?

Formal concept analysis

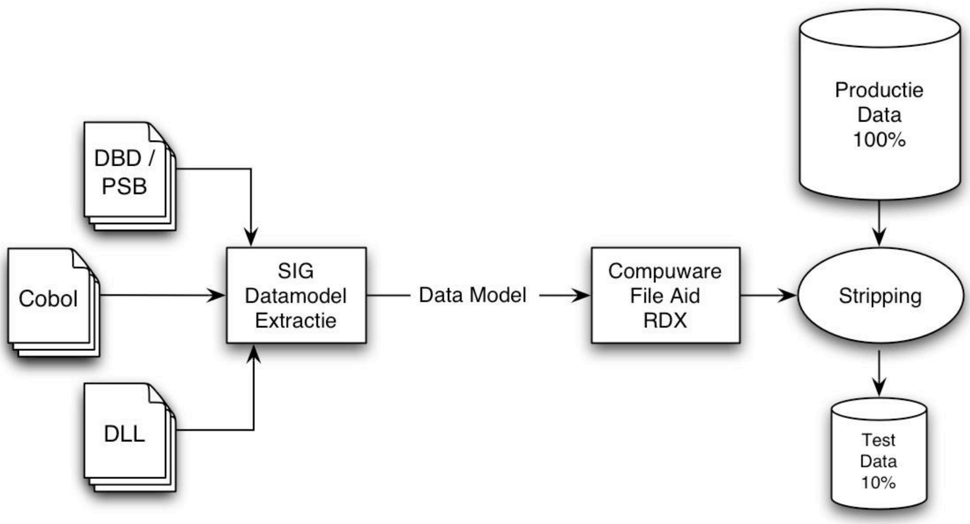
Representation

```
type Context g m = Rel g m
type Concept g m = (Set g, Set m)
type ConceptLattice g m
   = Rel (Concept g m) (Concept g m)
```

Algorithm

```
neighbors :: (Ord g, Ord m)
          => Set g              -- extent of concept
          -> Context g m        -- formal context
          -> [Concept g m]      -- list of neighbors

lattice :: (Ord g, Ord m)
        => Context g m          -- formal context
        -> ConceptLattice g m   -- concept lattice
```


Case: Test data stripping



Goal: extract DB2 en IMS relations through program source analysis.

Especially programmatic relations:

- not defined explicitly in DB definitions,
- rather: encoded in application programs,
- can occur across modules, programs, systems.

Kinds of DB relationships

Explicitly modeled relationship

- Referential Integrity relationships (foreign keys in SQL)

Relationships in queries

- Implicit join in SQL, Explicit JOIN in SQL
- Joins between sub-queries, Joins in views

Programmatic relationships

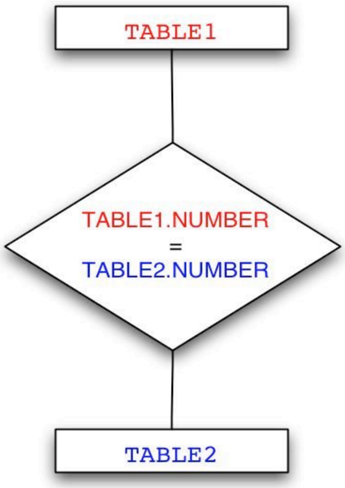
- Programmatic join within 1 program via dataflow en compares
  (SQL: where clause; IMS: segment search argument)
- Programmatic join across programs via calls
- Programmatic join across systems

Cod. standard: between systems no RI relationships can be defined.

Note: Relationships between IMS en DB2 occur as well.

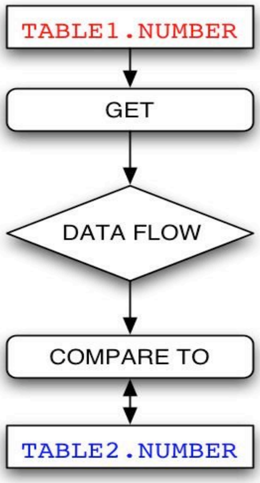
A simple programmatic join


```
EXEC SQL
  SELECT NUMBER INTO FIELD1
  FROM TABLE1.
END-EXEC.

-    -

MOVE FIELD1 TO FIELD2.

[...]

EXEC SQL
  SELECT *
  FROM TABLE2
   WHERE
 NUMBER = FIELD2.
END-EXEC.
```



A simple programmatic join with dataflow

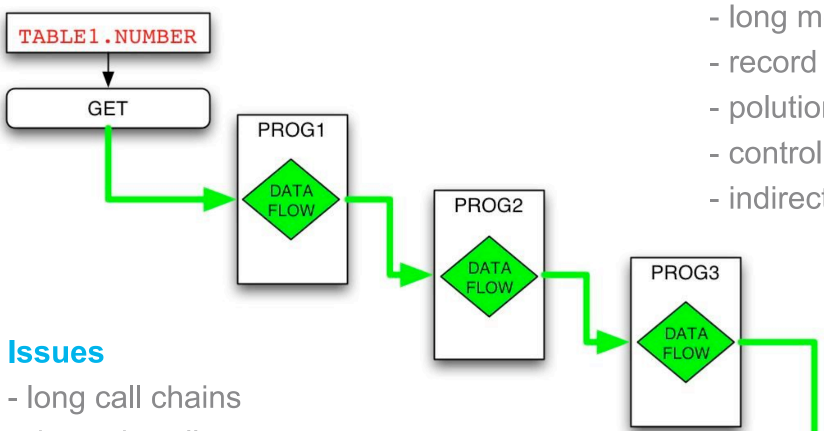

```
EXEC SQL
   SELECT NUMBER INTO FIELD1
   FROM TABLE1.
END-EXEC.

[...]

MOVE FIELD1 TO FIELD2.

[...]

EXEC SQL
   SELECT *
   FROM TABLE2
    WHERE
 NUMBER = FIELD2.
END-EXEC.
```



A simple programmatic join across programs

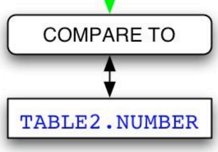
Issues

- long move chains
- record moves
- polution due to helper fields
- control flow
- indirection via views



Issues

- long call chains
- dynamic calls
- utility calls
- calls across systems
- asynchronous calls (IMS triggers, MQ)



IMS
Hierarchical database system

Software Improvement Group

General

- IMS = Information Management System.
- Developed by IBM in the late 60s.

Databases

- Data is organized in tree structures.
- Nodes of trees are *segments*, which are sequences of *fields*.
- *Logical* databases define a selective view on a physical database.
- All of this defined in DBD = Data Base Definition

Access

- PSB = Program Specification Block.
- Define which segments are accessible (sensitive) to which programs.
- Database operations are performed via utility calls with appropriate args.
- SSA = Sensitive Search Argument can be passed as argument.

IMS versus SQL
A rough correspondance

Software Improvement Group

IMS	SQL
segment	**table**
field	**column**
SSA	***where* clause**
logical database	**view**
utility call	**query**
DBD	**DDL**
PSB	-

Software Analysis and Testing, MFES Universidade do Minho by Joost Visser, Software Improvement Group © 2007.

Approach
Sketch of solution

Dataflow analysis

- Find data-flow paths between column occurrences.
- Multilingual: Cobol, IMS, DB2.
- Across modules, programs, systems.
- Scalable to complete portfolio.

Path selection

- Select paths that indicate data model relationships.

Validation

- On the basis of types, naming, indexing, path length, etc.

Approach
Some basic required ingredients

Parsing

- (embedded) DB2 SQL parser
- Parsers for IMS definitions
- Parser for Cobol

Name resolution

- SQL column names: find corresponding tables, possibly via aliases.
- Cobol field names: find corr. field declaration, possibly via redefines.
- IMS segment and field names: reconstruct correspondence of Cobol field names to IMS fields via PCBs, logical databases, memory layout.
- Link SQL host names to Cobol field names.

Type matching

- SQL column types ~ Cobol record member types ~ IMS field types.

Approach
Dataflow

Challenges

- Record moves.
- Dynamic calls, call handlers, asynchronous calls (IMS triggers).
- Compound keys (follow parallel dataflow).
- Cursors.
- Nested queries, complex queries, joins in views.
- Indirection via views.
- Scalability.
- Pollution, due to auxiliary variables, utility call parameters, …

Requirements

- Modular (for scalability).
- Generic (for manageability).
- Customizable (to reduce pollution and silences) …

Approach
Risks and their mitigation

Silences (false negatives, incorrectly not found)

- *Source:* missing dataflow links, e.g. due to non-resolvable dynamic calls.
- *However:* relationship between 2 columns/fields only absent if ALL dataflow paths are missing a link.

Noise (false positives, incorrectly found)

- *Source:* tangling dataflow links, e.g. due to auxiliary variables or utility parameters.
- *Counter measure:* fine-tuning of heuristics to suppress noise-generating links.

Scale

- Dataflow analysis for an entire portfolio is a resource intensive computation.
- *Counter measure:* modularization of the analysis algorithm, persistency for (partial) data flow graphs, hardware.

Study characteristics

General

- 7 systems (out of about 250)
- DB2: 992 tables, 882 views, 106 foreign keys
- IMS: 2110 databases, 4778 segments, 8163 fields, 8143 sensitive segments, 7716 sensitive fields
- Cobol (9 MLoc in 2786 listings), 2203 selects and fetches, 272 inserts and updates, 5993 IMS operations using 1446 sensitive search arguments.

Findings

Observations

- Small number of foreign key definitions given the number of tables.
- Only a handful of non-programmatic joins.
- Only a handful of views with a complex query.
- Programmatic joins often via cursor, often via call.
- Dataflow paths length for programmatic joins within a single program usually between 3 en 8.
- Programmatic joins between IMS en DB2 occur.

More info? Feel free to contact…

Dr. ir. Joost Visser

E: j.visser@sig.nl
W: www.sig.nl
T: +31 20 3140950