Introduction to Software Evolution

Jurgen Vinju
Magiel Bruntink
Vadim Zaytsev

Ashim Shahi
Anastasia Izmaylova
Atze van der Ploeg
Davy Landman
Michael Steindorfer

CWI Centrum Wiskunde & Informatica

Universiteit van Amsterdam

Where are you?

- International conference in Indianapolis U.S.A
  - SPLASH (OOPSLA):Object-oriented programming
  - SLE: Software Language Engineering
  - GPCE: Generative programming
  - Parsing@SLE: interactive workshop
- Papers by our group:
  - "Micro-Machinations: A DSL for Game Economies"
  - "Micropatterns in Grammars"
  - "Safe Specification of Operator Precedence Rules"
  - "The State of the Art in Language Workbenches"

Courses

• Intro	(Jurgen)
• Rascal	(Jurgen)
• Metrics	(Magiel)
• Design Patterns	(Jurgen)
• Visualization	(Jurgen/Paul)
• Mining	(Magiel)
• Refactoring	(Jurgen)
• Technical Debt	(Jurgen)

Global Schedule

Lectures

Mon: 09:00 – 11:00 in SP G2.10, all weeks

Lab (deadlines in the pdf on Blackboard Assignments)

Mon: 11:00 – 17:00 in **G0.23-G0.25** and G0.10-G0.12

Tue: 09:00 – 17:00 in **G0.23-G0.25** and G0.10-G0.12

Prefer G0.23 and let us know if you are in G0.10!

Be there!

Paper sessions (essay deadlines in the same pdf)

Every other week

Mon: SP G2.04, G3.05, **Tue**: G2.04, D1.162

Grades

- Series 0 has no grade, but it trains you for the...
- required Online lab test (Rascal) > 50% correct
- 1/3 paper sessions, required > 5.5
- 1/3 Series 1, required > 5.5
- 1/3 Series 2, required > 5.5
- Overall average required > 5.5

Today!

- 9-11 Introduction to Software Evolution & Rascal
  - Online
  - Next courses will be **in person**
- 11-17:00 Getting started with Series 0
  - this includes lunchtime
  - assistants available!
  - In **G0.23-G0.25** and G0.10-G0.12
- Next week we continue with quick Rascal course

Lab project (Series 1)

- Work in pairs
- Software Assessment
  - **Measuring** source code
  - To find **indications** of good/bad quality
  - **Predicting** hard to maintain, costly, source code
- Software Metrics
  - Mechanics using Rascal
  - Definition and correctness ?!
  - Aggregation ?!
  - Interpretation ?!

Lab project (Series 2)

- Reverse Engineering
  - From source code to design
  - Visualization
- Software Visualization
  - Mechanics using Rascal
  - Software Exploration
  - Software Understanding
  - Link with metrics

Lab project (Advanced Track)

- Starts with Series 2
- On demand, personalized
- Instead of series 2
- Research on Android API evolution
  - Mining infra-structure
  - Analysis
  - Compare with related work
- Very challenging!
- Grading: a successful project gives extra points

Paper sessions

- There is no book with this course
- Instead we read papers about software evolution and discuss them
- You write an outline of a paper: stepping stone towards a great masters thesis!
- Feedback from teachers and lab assistants
- Blackboard -> Assignments

Roadmap

- The Software Volcano
- Introduction to Software Maintenance & Evolution
- Introduction to Software Renovation
- Introduction to Program Analysis and Transformation
- Wrapping up

Software Volcano

Mt. Etna, Sicily, Italy

The Software Volcano: Languages

Distribution of languages in use, worldwide

Language	Used in % of total
COBOL	30
Assembler	10
C	10
C++	10
550 other languages	40

- For mainframe applications 80% is COBOL!
- Figures taken from Capers Jones (Software Productivity Research)

Software Volcano: Volume

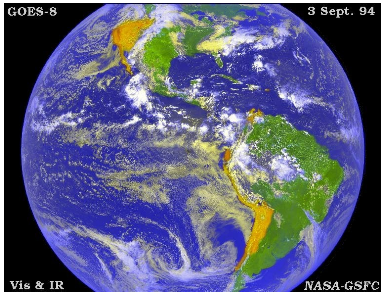
- The total volume of software is estimated at $7 * 10^9$ *function points*
- 1 FP = 128 lines of C or 107 lines of COBOL
- The volume of the volcano is
  - 750 Giga-lines of COBOL code, or
  - 900 Giga-lines of C code

Printed on paper we can wrap planet Earth
9 times!

Software Volcano: Defects

- Observation:
  - on average 5 errors (bugs) per function point
  - includes errors in requirements, design, coding, documentation and bad fixes
- The software volcano, world-wide, contains $5 * 7 * 10^9$ Bugs = 35 Giga Bugs

This means 6 bugs per human being on planet Earth!



Work distribution of programmers

Year	New projects	Enhancements	Repairs	Total
1950	90	3	7	100
1960	8,500	500	1,000	10,000
1970	65,000	15,000	20,000	100,000
1980	1,200,000	600,000	200,000	2,000,000
1990	3,000,000	3,000,000	1,000,000	7,000,000
2000	4,000,000	4,500,000	1,500,000	10,000,000
2010	5,000,000	7,000,000	2,000,000	14,000,000
2020	7,000,000	11,000,000	3,000,000	21,000,000

Now: 60% of the programmers work on enhancement and repair

In 2020: only 30% of all programmers will work on new software

Message

- When an industry approaches 50 years of age it takes more workers to perform maintenance than to build new products (*ex:* automobile industry)
- Maintenance and renovation of existing software become more and more important: avoid that the software volcano explodes
- Making sure existing software keeps its value is an interesting intellectual challenge. Do not dismiss dealing with other people's code.

Roadmap

- The Software Volcano
- Introduction to Software Maintenance & Evolution
- Introduction to Software Renovation
- Introduction to Program Analysis and Transformation
- Course Overview *Software Evolution*

Roadmap

- The Software Volcano
- Introduction to Software Maintenance & Evolution
- Introduction to Software Renovation
- Introduction to Program Analysis and Transformation
- Wrapping up

Introduction to Software Maintenance & Evolution

- What is Software Maintenance?
- Why does software evolve?
- Problems in Software Maintenance
- Solutions

What is Software Maintenance?

- Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment (IEEE 1219, 1993)
- Observe that:
  - maintenance is seen as after-the-fact activity
  - no integration with software development process in this definition

Another Classification

- Software maintenance
  - Changes are made in response to changed requirements
  - The fundamental software structure is stable
- Architectural transformation
  - The architecture of the system is modified
  - Generally from a centralised to a distributed architecture
- Software re-engineering
  - No new functionality is added to the system but it is restructured and reorganised to facilitate future changes

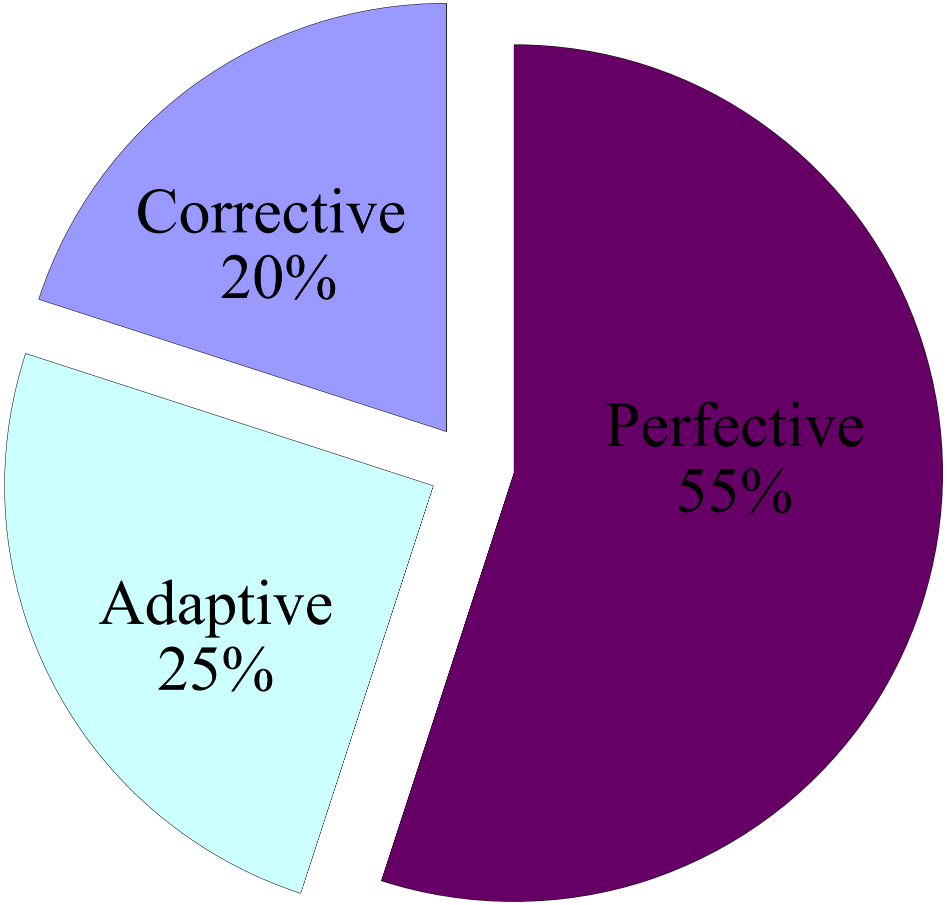
Why systems change

- Correct errors
- Business pull:
  - Business / IT alignment
  - Requirements change (legislation, new insights, efficiency)
  - Re-organization
  - Mergers / take-overs
  - New products, marketing actions
  - Market hypes (CRM, ERP, BPR, STP)
- Technology push:
  - Internet
  - Mobile
  - Updates of operating system, development environment, databases
  - Hardware

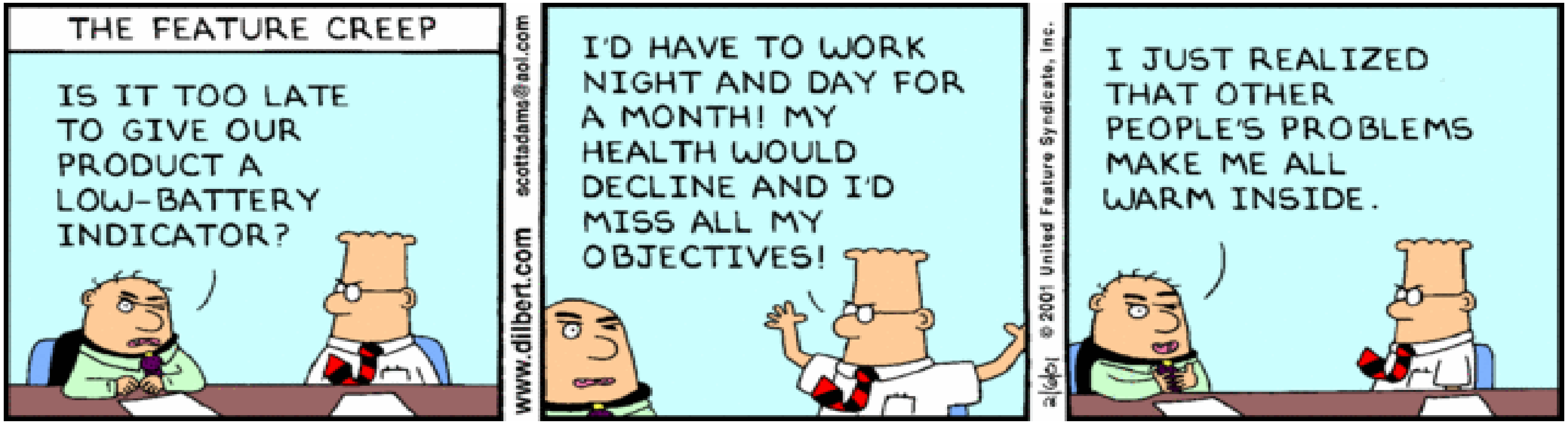
Categories of Maintenance

- Corrective: needed to correct actual errors
- Adaptive: result from changes in the environment
- Perfective: modifications to meet the expanding needs of the user
- Enhancement = Adaptive + Perfective
- Point to ponder: does this classification help?

Cost Distribution per Category



Feature creep


THE FEATURE CREEP
IS IT TOO LATE TO GIVE OUR PRODUCT A LOW-BATTERY INDICATOR?
I'D HAVE TO WORK NIGHT AND DAY FOR A MONTH! MY HEALTH WOULD DECLINE AND I'D MISS ALL MY OBJECTIVES!
I JUST REALIZED THAT OTHER PEOPLE'S PROBLEMS MAKE ME ALL WARM INSIDE.

scottadams@aol.com www.dilbert.com

2/6/01

Costs of Maintenance

- Usually greater than development costs
  - 2 times to 100 times depending on the application
- Affected by both technical and non-technical factors
- Increases as software is maintained
  - Maintenance corrupts the software structure, making further maintenance more difficult
- Ageing software can have high support costs
  - old languages, compilers etc.
- Think of your software as continuously evolving

Cost factors

- Team stability
- Contractual responsibility
- Staff skills
- Program age and structure

Costs and Complexity

- Predictions of maintainability costs may be made by assessing the complexity of system components.
- Most maintenance effort is spent on a relatively small number of system components.
- Complexity depends on
  - Complexity of control structures;
  - Complexity of data structures;
  - Object, method (procedure) and module size
  - Dependencies
  - ***Understandability & Changeability***

Lehman’s Laws for Software Evolution

- Lehman observed that software evolves
- Law of Continuing Change: software needs to change in order to stay useful
- Law of Increasing Complexity: the structure of a program deteriorates as it evolves
  - the structure of a program degrades until it becomes more cost effective to rewrite it

An Example (*Civility*)

- Software for city administration; Old, successful, reliable

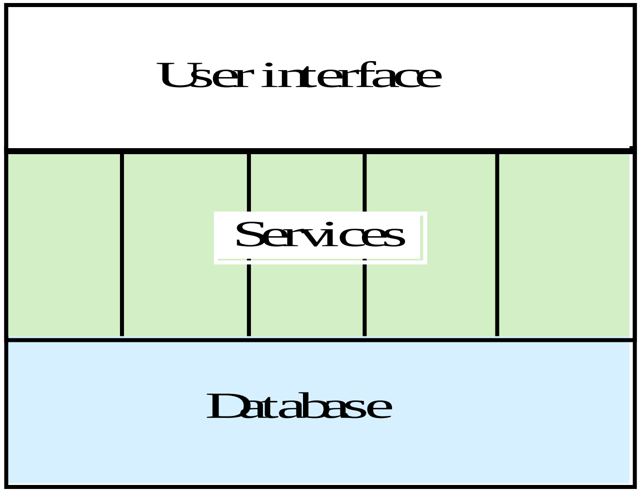
  ➔ large client base
- Complex code (performance, size, many changes)
- No clear separation between Data, Business, Logic and User Interface

  ➔ High costs for maintenance, hard to change
- Need to change (internet, legislation, process management, CRM)

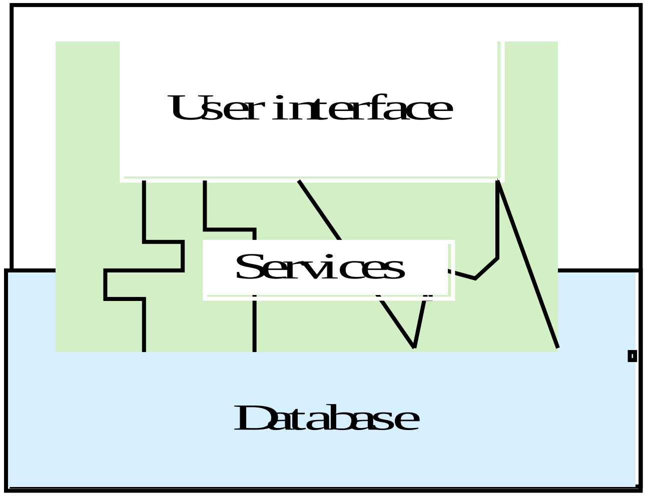
  ➔ Re-engineering and migration

Legacy systems

- Ideally, for distribution, there should be a clear separation between the user interface, the system services and the system data management
- In practice, these are usually intermingled in older legacy systems



Ideal model for distribution



Real legacy systems

Spaghetti code

```
Start:   Get (Time-on, Time-off, Time, Setting, Temp, Switch)
     if Switch = off goto off
     if Switch = on goto on
     goto Cntrld
off:  if Heating-status = on goto Sw-off
     goto loop
on:   if Heating-status = off goto Sw-on
     goto loop
Cntrld:  if Time = Time-on goto on
     if Time = Time-off goto off
     if Time < Time-on goto Start
     if Time > Time-off goto Start
     if Temp > Setting then goto off
     if Temp < Setting then goto on
Sw-off:  Heating-status := off
     goto Switch
Sw-on: Heating-status := on
Switch:  Switch-heating
loop: goto Start
```

Some observations from *Civility*

- Current architecture used to the max
- New requirements require new architecture
- The more stable the functionality, the more the knowledge diminishes
- These systems are *really good*!
- But nobody *knows why* anymore...
- So the *maintenance process* must be very strict:
  - maintenance costs high and flexibility low
- Limited use of tooling

Why Systems Survive

- Organisations have huge investments in their software systems
- Systems are critical business assets
- Organizations depend on the system
- Organizations know how to use their systems
- (Re) building systems is high risk

Business *versus* IT in Software Maintenance

- Low costs
- Opportunistic / flexible
- Quick decision making
- Reliability in short time
- IT should understand business
- Protect initial investment
- Standardization
- Problems with IT systems make companies careful
- ***Quantity***

- Need for adequate resources
- Requires planning / choices
- Hard to predict costs, impact
- Time to deliver quality
- Business should understand IT
- Want something new
- Creativity
- Unpredictability
- Why all these procedures?
- ***Quality***

Major problems in Software Maintenance

- Inadequate testing methods
- Performance measurement difficulties
- Knowledge management / documentation
- Adapting to the rapidly changing business environment
- Large backlog

Major problems in Software Management

- Lack of skilled staff
- Lack of managerial understanding and support
- Lack of maintenance methodology, standards, procedures & tools
- Program code is complex and unstructured
- Integration of overlapping/incompatible systems

Solutions

- Better architecture
- Incremental (agile)
- Batch large-scale source-to-source renovation
- Automated regression testing (why?)
- Knowledge management
- Automated code inspection
- Better organization -> ITIL / CMM

Towards a Software Maintenance Process

- Maintenance should be organized as a structured process (??)
- ISO/IEC 12207: a standard maintenance process
- ITIL: Information Technology Infrastructure Library
- CMMI: Capability Maturity Model
- Gives an impression of the scope and details of the maintenance process
- Read about it, but it is not my cup of tea

ITIL Pointers

- Pink Elephant : www.pinkelephant.com
- ITIL (Libraries) & Service Management directories: www.itil-itsm-world.com/
- British government ITIL: www.ogc.gov.uk/index.asp?id=2261
- techrepublic.com.com/5100-6329-1058517.html - Tech Republic article (subscription required)
- KU's Program & Service Management Office: www.ku.edu/~psmo

Intermezzo; The Metaphor Game

- “Software Maintenance” and “Software Evolution” are metaphors.
- Why these words?
  - Does software wear and tear?
  - Does software procreate and does software select partners?
- What is the intented meaning?

Roadmap

- The Software Volcano
- Introduction to Software Maintenance & Evolution
- Introduction to Software Renovation
- Introduction to Program Analysis and Transformation
- Wrapping up

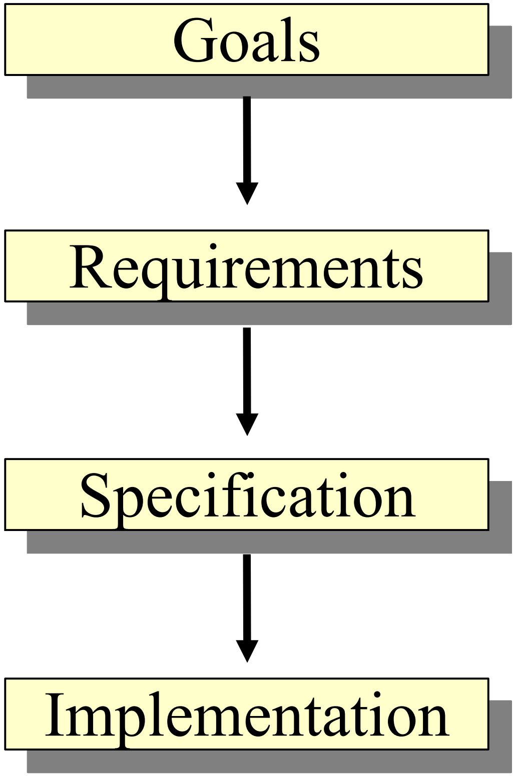
Roadmap

- The Software Volcano
- Introduction to Software Maintenance & Evolution
- Introduction to Software Renovation
- Introduction to Program Analysis and Transformation
- Wrapping up

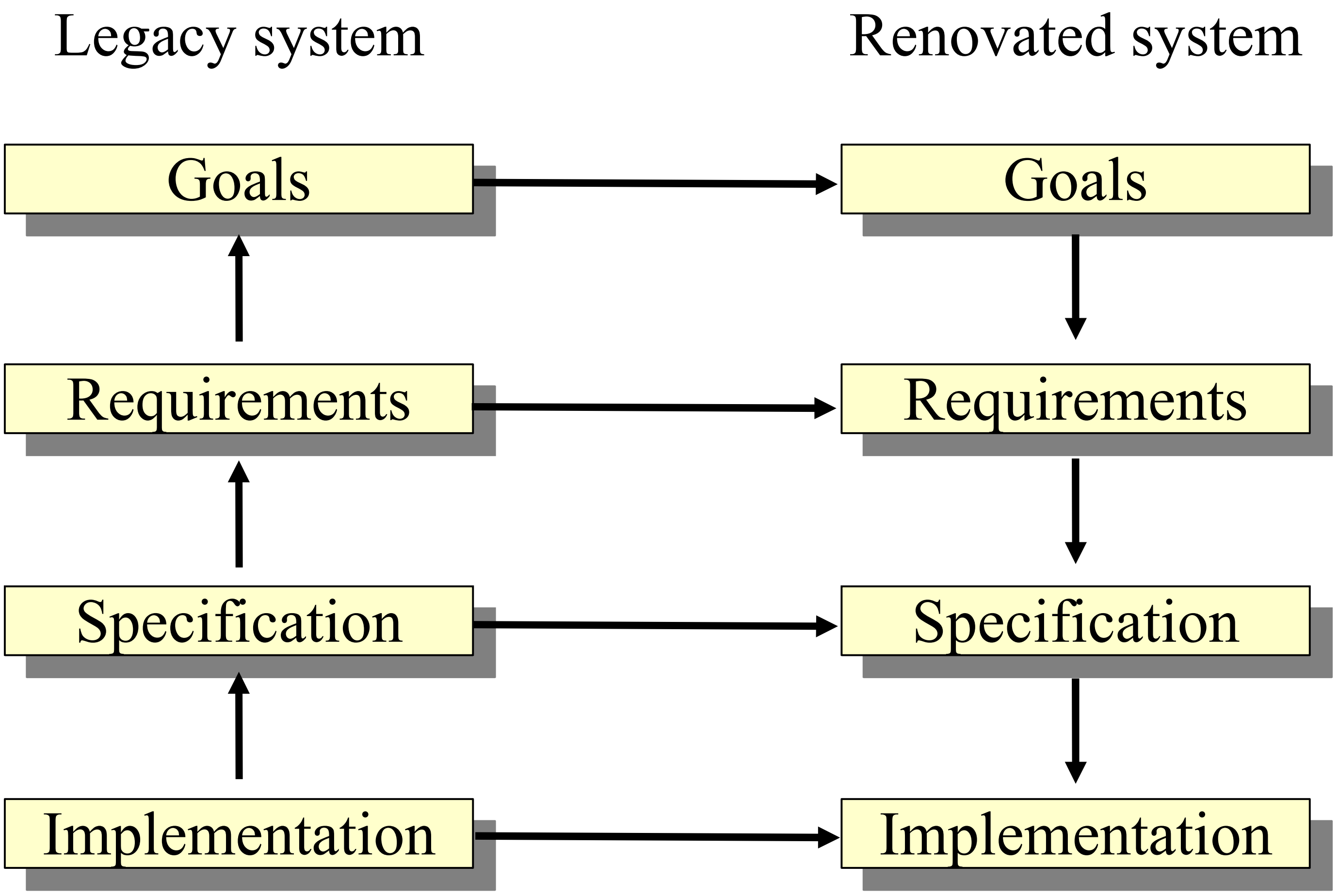
Introduction to Software Renovation

- Legacy system:
  - (information) system that defeats further maintenance, adjustment or renewal due to its size and age
  - requires increasing maintenance costs
- System renovation:
  - understanding and improvement of legacy systems
  - by means of reverse engineering, program understanding, design recovery, transformation, ...

Forward Engineering



Reverse Engineering



Legacy system

Renovated system

Goals

Requirements

Specification

Implementation

Goals

Requirements

Specification

Implementation

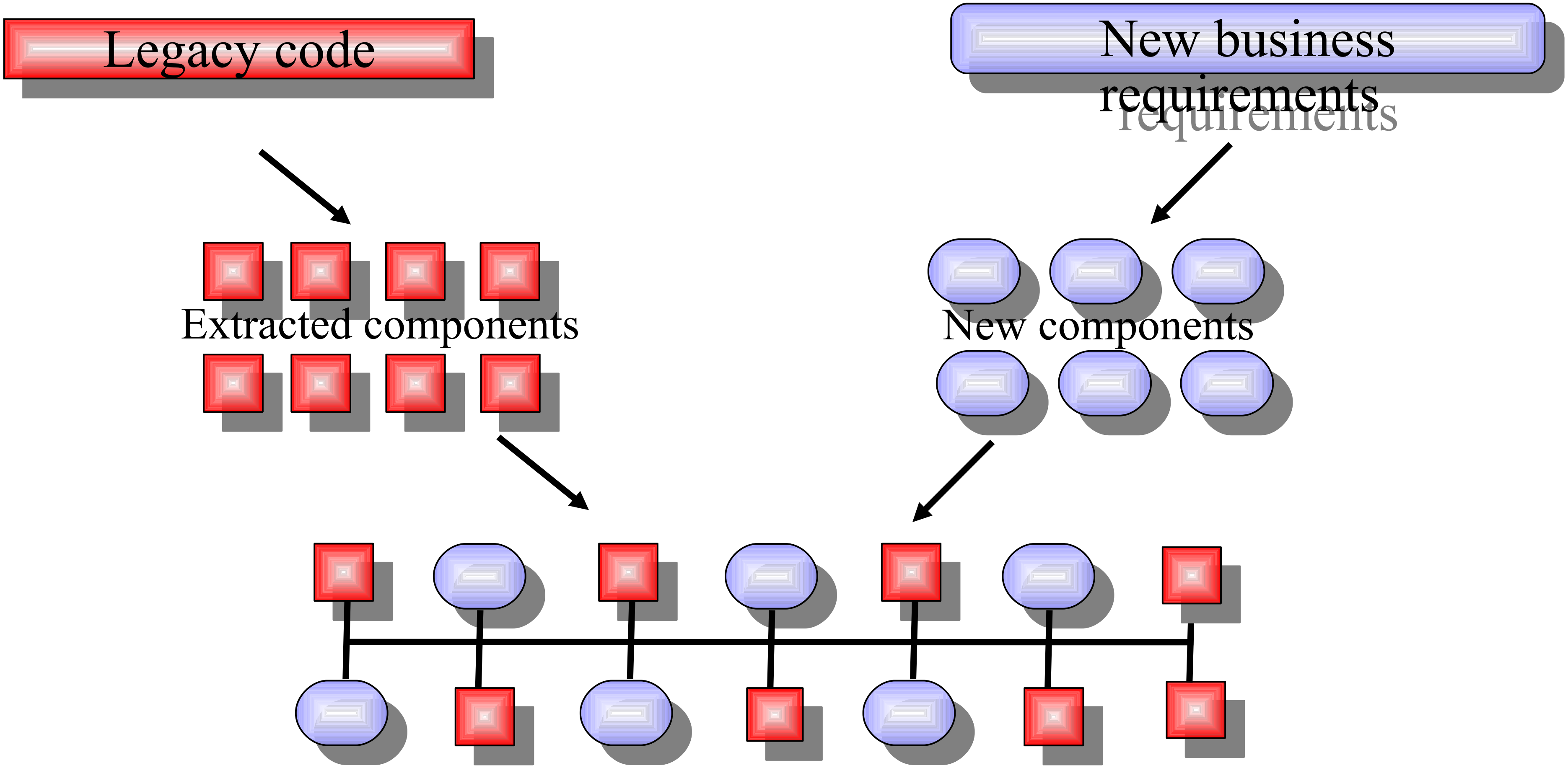
A Typical Legacy System

- Different implementation languages
- Job Control Language scripts serve as glue
- Part of programs/databases are obsolete
- Some source text lost or incomplete; version unknown
- Documentation is incomplete or obsolete

Typical Renovation Questions

- What is the architecture of this system
- Can we improve its structure?
- Can we generate documentation for it?
- Can we migrate it from COBOL 74 to COBOL85?
- Can we connect it to Internet?
- Can we migrate it to a client/server architecture?

Synergy between Renovated and New Components



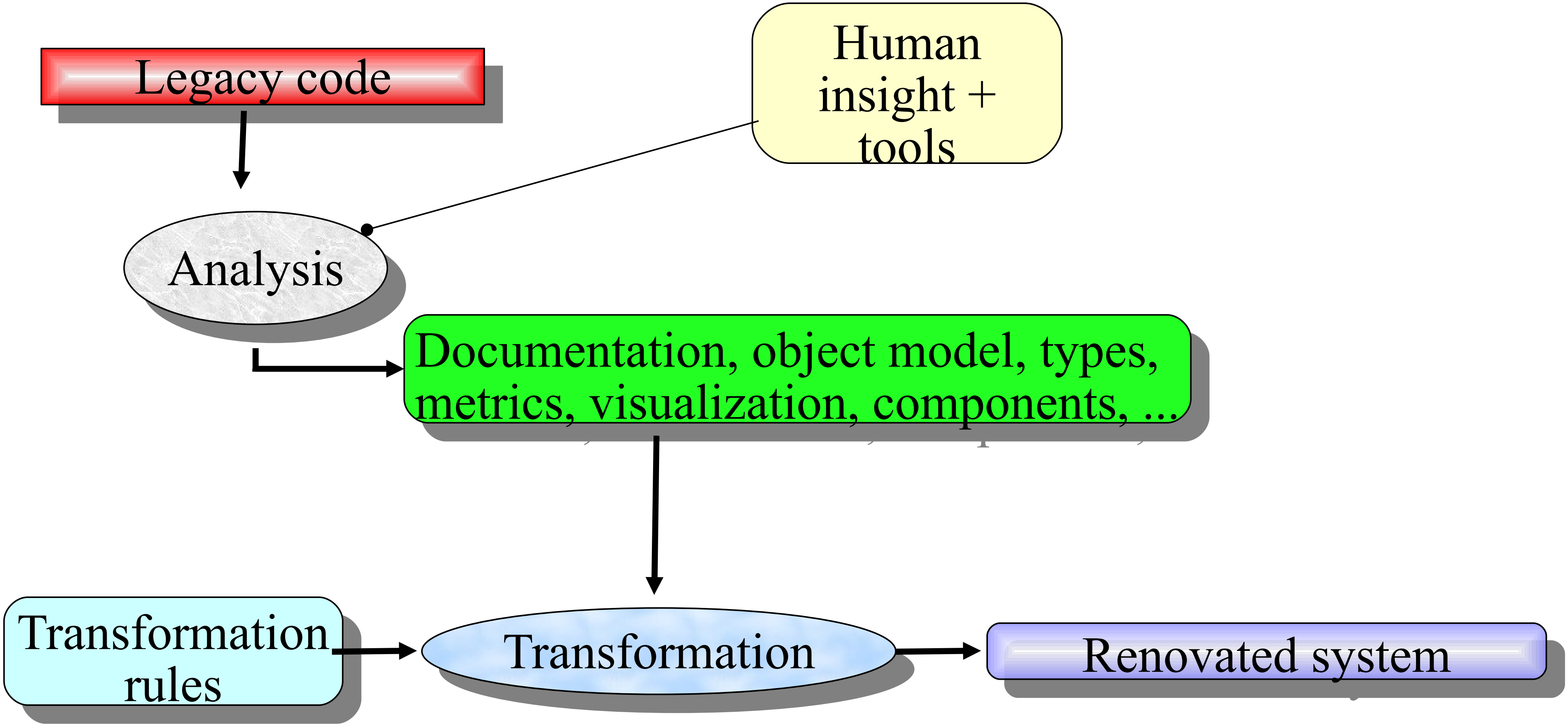
Legacy code

New business requirements

Extracted components

New components

Renovation = Analysis + Transformation



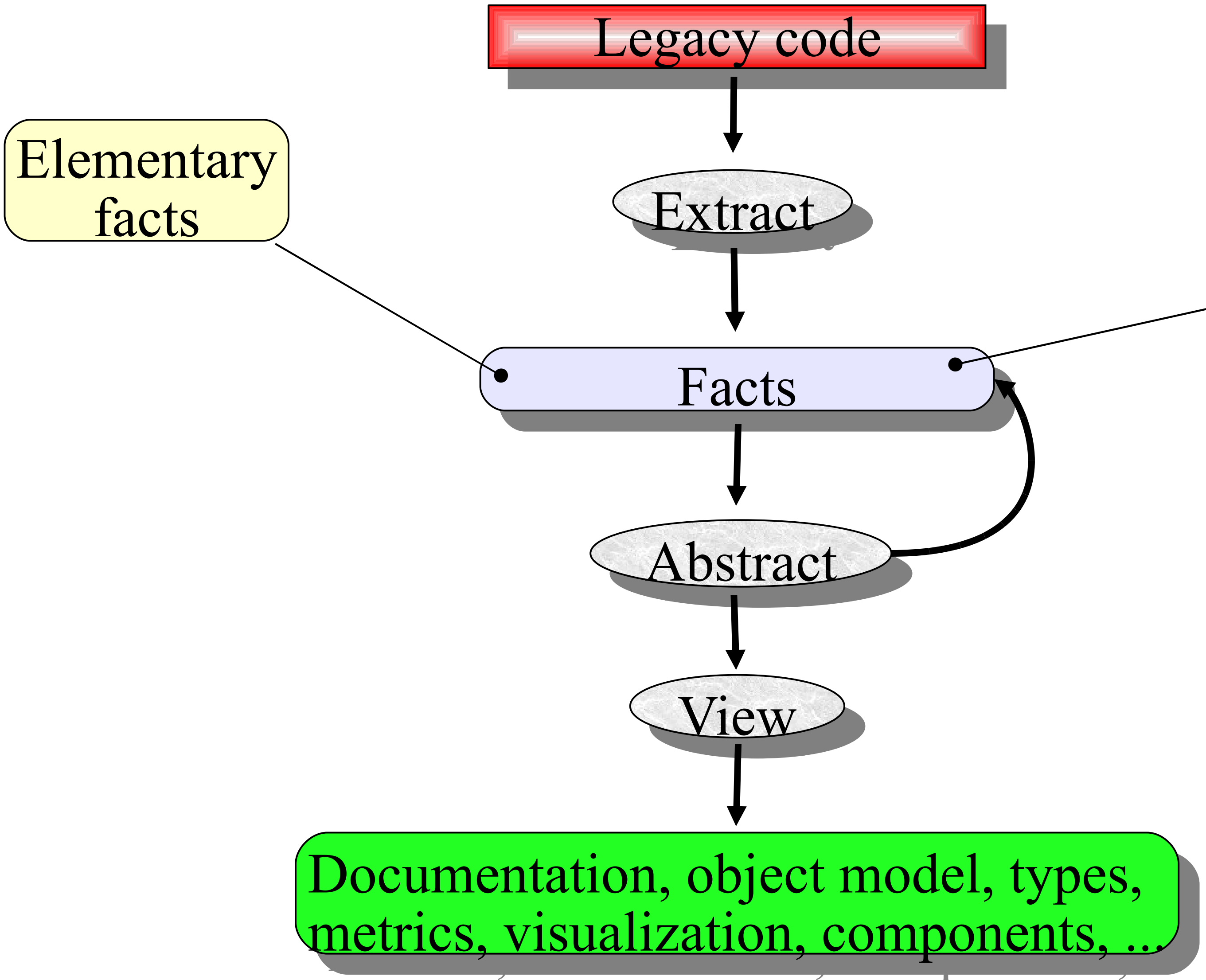
Software Renovation

- Analysis (partly supported by tools):
  - architecture recovery
  - system understanding
- Transformation (mostly supported by tools):
  - systematic repairs
  - code improvement/dialect conversion/translation
  - architecture improvement/change

Software Renovation: Analysis

- Extraction of procedure calls and call graph
- Database usage between programs
- Dataflow analysis (at program and system level)
- Type analysis
- Cluster and concept analysis
- Metrics
- Visualization

Software Renovation: analysis



Legacy code

Extract

Facts

Abstract

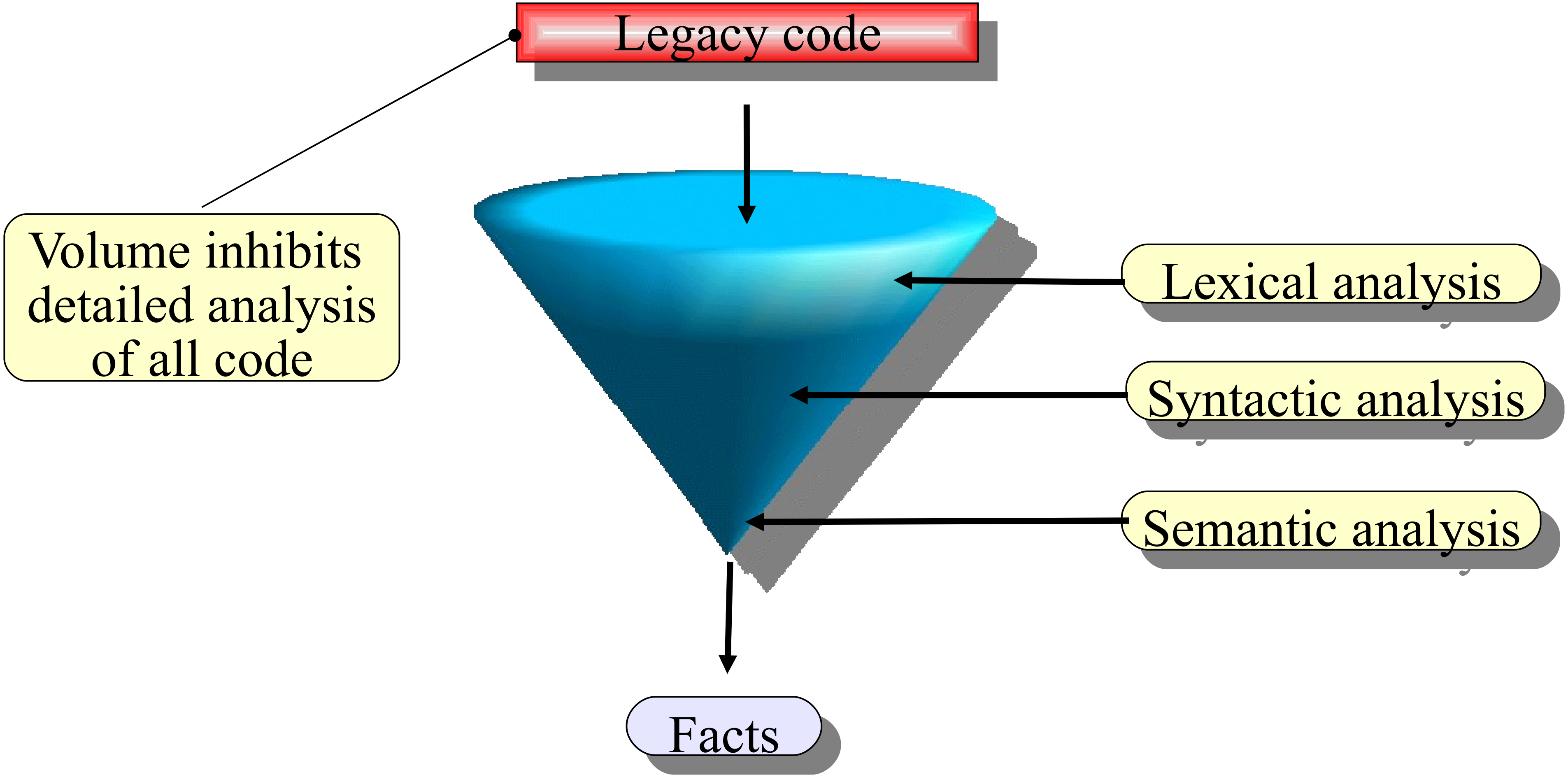
View

Documentation, object model, types, metrics, visualization, components, ...

Elementary facts

Enriched by semantic queries

The Analysis Funnel

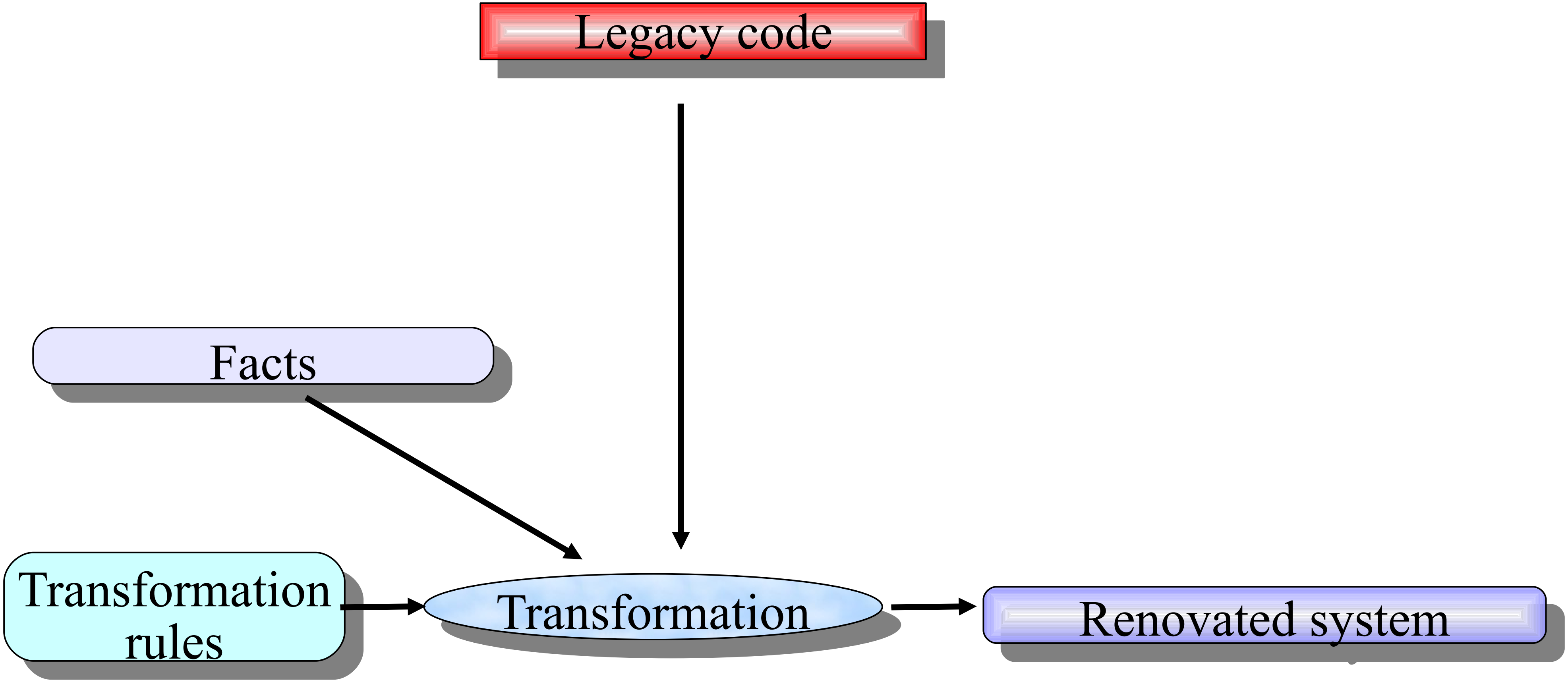


Volume inhibits detailed analysis of all code

Example: DocGen

- Given the sources of a legacy system, web-based documentation is generated containing
  - overall architecture
  - module dependencies & internal structure modules
  - database usage
  - simple metrics
- Fact: code reading finds two times more defects than testing

Software Renovation: Transformation


Legacy code

Facts

Transformation rules

Transformation

Renovated system

Typical Transformations

- Year 2000
- Euro
- Extending bank account numbers to 10 digits
- Goto elimination
- OO restructuring
- Dialect translation (Cobol 74 -> Cobol 85)
- Language conversion (Cobol -> Java)

Observations

- Most legacy systems are multi-lingual
- A generic approach is needed to describe all forms of analysis and transformations for all required languages
- Languages like COBOL and PL/I are big:
  - getting the right grammar is difficult
  - many parsing techniques break down

Needed Technologies

- Lexical scanning & Parsing
- Fact repository & queries
- Search
- Replacement

Take home messages

- Software evolves in order to stay useful
- Maintenance (= 80% enhancement) enables this evolution
- Maintenance should be based on a well-defined process
- Software renovation is needed to extend the life cycle of a system
- Software renovation can be supported by tools

The role of Rascal

- Rascal is designed for ***all*** this work
  - Parsing and lexical analysis
  - Relation modeling (facts!)
  - Source code locations (links!)
  - Patterns (search)
  - Visits (replacement)
- Libraries
  - Visualization
  - SVN, Git, SSH access
  - Etc. etc.

Roadmap

- The Software Volcano
- Introduction to Software Maintenance & Evolution
- Introduction to Software Renovation
- Introduction to Program Analysis and Transformation
- Wrapping up

Roadmap

- The Software Volcano
- Introduction to Software Maintenance & Evolution
- Introduction to Software Renovation
- Introduction to Program Analysis and Transformation
- Wrapping up

Introduction to Program Analysis and Transformation

- Lexical syntax
- Context-free syntax
- Static semantics
- Dynamic semantics
- Static versus dynamic analysis
- Control flow graph
- Data flow graph
- Call graph
- Examples of transformations

Lexical Syntax

- What are the keywords (if, return, while)
- What are identifiers (rather_long_identifier)
- What are the constants (123, “a string”, false)
- What are the layout symbols (space, tab, newline)
- What are the comments (// ..., /* ... */)
- Related notions:
  - lexical grammar (describes lexical syntax)
  - lexical scanner (recognizes lexical syntax)

Context-free Syntax

- What is the structure of declarations/statements (if <expr> then <stat> else <stat> end)
- Related notions:
  - grammar (describes the context-free syntax)
  - syntax analyser, parser (recognizes context-free syntax and builds a parse tree)
  - parse tree, syntax tree (tree that describes structure of a text, including all layout, keywords, etc.)
  - abstract syntax tree (parse tree with textual elements like layout, keywords, etc. removed)

Static Semantics

- Pre-execution meaning of language elements:
  - are all variables declared?
  - are all expressions type correct?
  - are all procedure/methods called with correct parameters?
- Static semantics is conservative: run-time values are unknown and all possibilities should be considered
- Related notions:
  - type checking, compile-time analysis, model checking, abstract interpretation

Dynamic Semantics

- Execution-time meaning of language elements:
  - what is the effect of an assignment?
  - what is the value of an expression?
  - which method should be called?
  - what is the result of executing a procedure call?
- Execution behaviour depends on specific input values
- Related notions:
  - run-time semantics, interpreters, compilers

Static versus Dynamic Analysis, 1

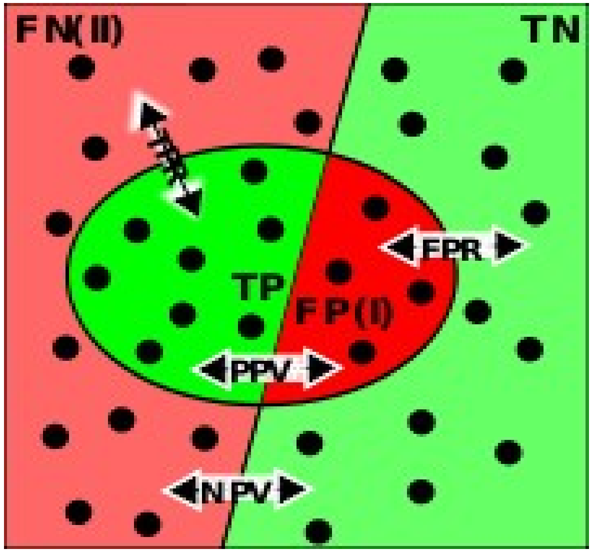
- Many analysis problems can be solved with only static analysis:
  - count number of class declarations
  - count number of goto statements
  - determine the methods with more than 25 lines of code
  - determine the methods with McCabe complexity larger than 3

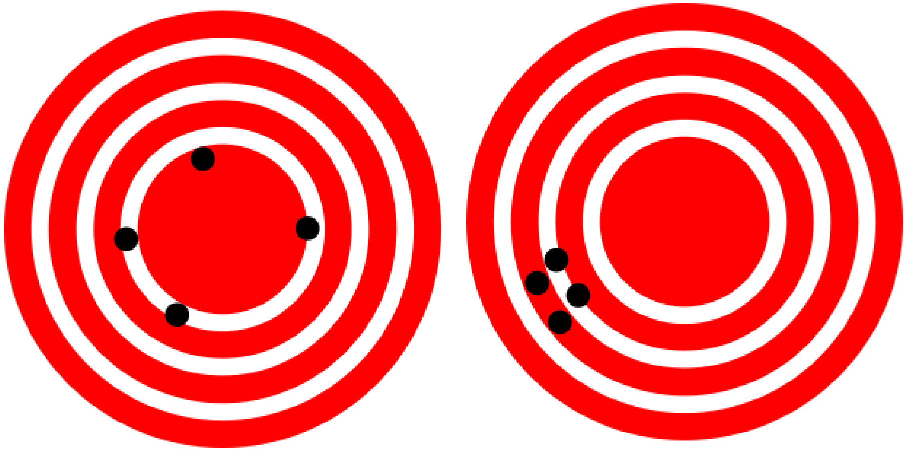
Static versus Dynamic Analysis, 2

- For other analysis problems, static analysis can only provide a conservative approximation:
  - call graph construction
  - dead code determination
- Some language constructs hinder static analysis:
  - run-time method selection in Java
  - reflection in Java
  - pointer indirection in C
  - run-time execution of strings as code

Intermezzo: Quality of analysis

- Binary classification
  - False/true positives/negatives
  - PPV: precision
  - TPR: recall
- Measurement
  - Precision vs Accuracy
  - Significant digits!
  - Units of measure!
- Trade-off efficiency

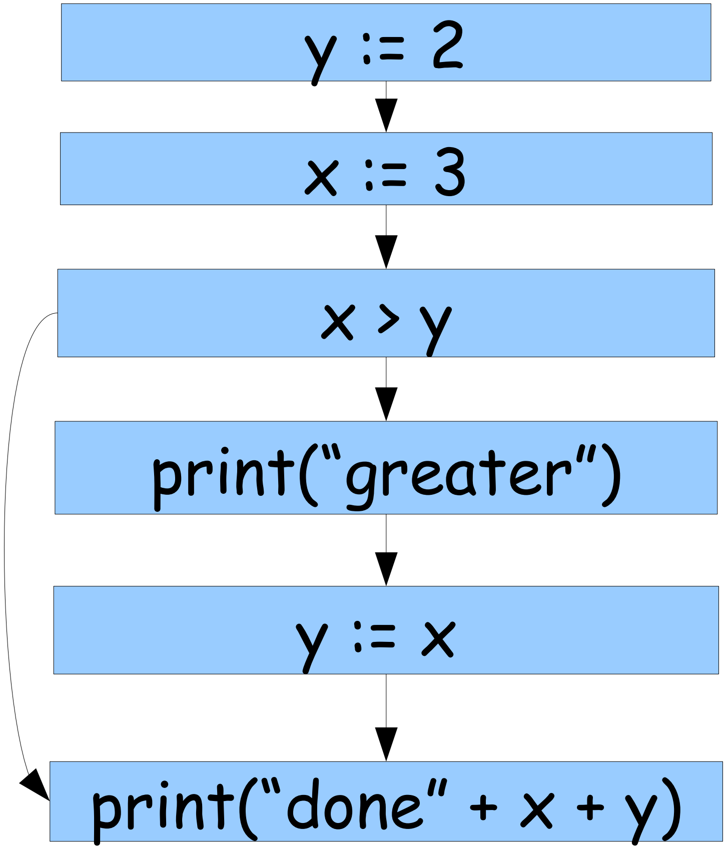




Control Flow graph

- Connects statements in the order in which they may be executed

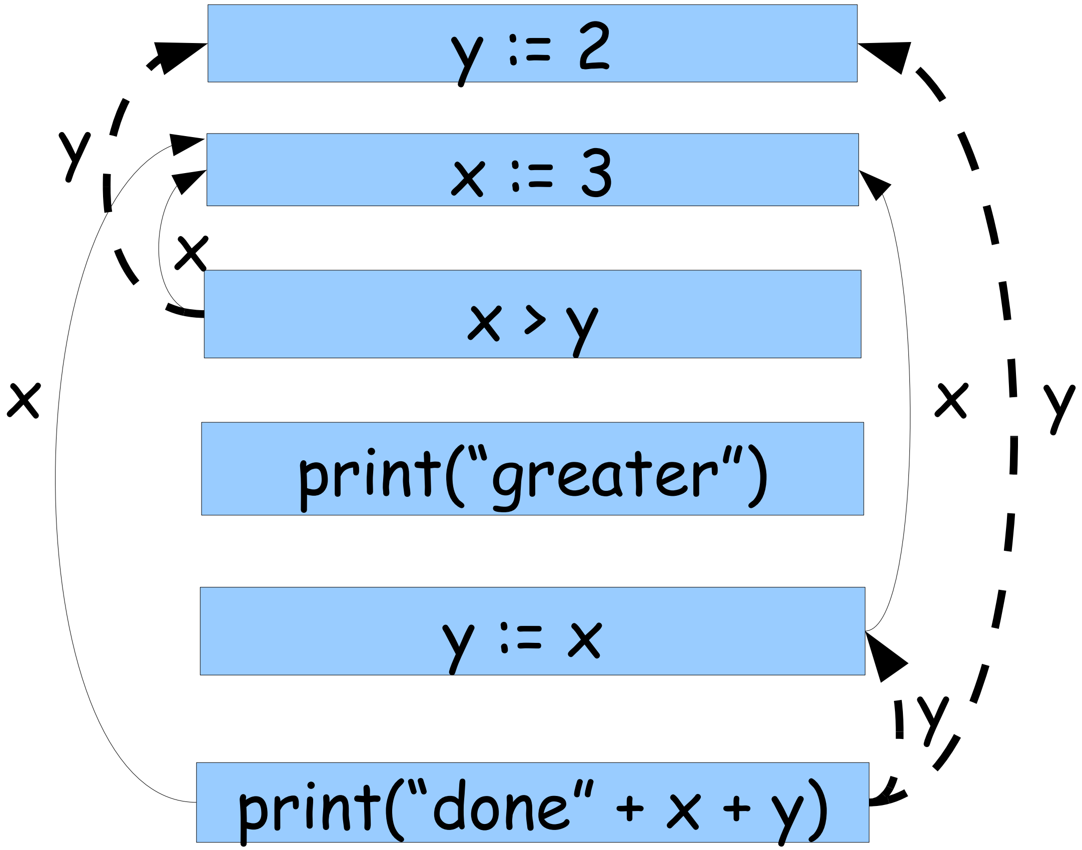
```
y := 2;
x := 3;
if x > y then
   print("greater");
   y := x
endif
print("done" + x + y)
```



Data Flow graph

- Connects variable uses with their definitions

```
y := 2;
x := 3;
if x > y then
   print("greater");
   y := x
endif
print("done" + x + y)
```



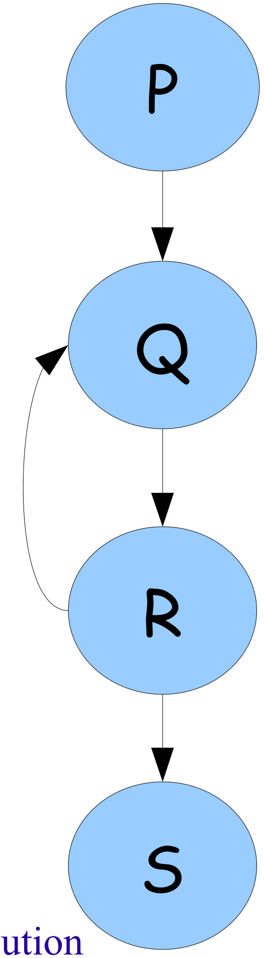
Call Graph

- Connects procedure calls with their definitions

```
proc P { … call Q …}

proc Q { … call R…}

proc R { … call Q …
           … call S … }
proc S { … }
```



Examples of Program Transformations

- Change the layout of the code according to standard rules
- Change method names
- Remove goto's
- Remove dead code
- Transform C to Java (very hard!)
- Migrate from some other (incompatible) library
- Migrate to another database system

Meta programming

- Type-checkers
- Refactoring
- Source-to-source
- Reverse engineering
- Reengineering
- Documentation generation
- Mining version repositories...
- All in the Rascal domain

Roadmap

- The Software Volcano
- Introduction to Software Maintenance & Evolution
- Introduction to Software Renovation
- Introduction to Program Analysis and Transformation
- Wrapping up

Roadmap

- The Software Volcano
- Introduction to Software Maintenance & Evolution
- Introduction to Software Renovation
- Introduction to Program Analysis and Transformation
- Wrapping up

Resources

- Blackboard: blackboard.ic.uva.nl
- Course: 2011.Software Evolution
- http://www.rascal-mpl.org
- www.acm.org/dl (ACM Digital Library)
- www.computer.org/portal/site/csdl (IEEE digital Library)
- Access to DLs is restricted (only via UvA).

Now

- Coffee
- At 11:00 in **G0.23-G0.25** and G0.10-G0.12
  - Installing and starting Eclipse + Rascal
  - Rascal tutor exploring
- Next Monday
  - Intro evolution and Rascal (here)
  - Rascal interactive course (**G0.23-G0.25**)

Take Home Messages

- Software Evolution is a real problem
- Software Maintenance is hard but interesting
- We designed Rascal for meta programming
- The lab is difficult but teaches you a lot
  - Metrics
  - Visualization
- The essay is important
  - Think of your thesis!