2IMP25 Software Evolution

Architecture Evolution

Alexander Serebrenik

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

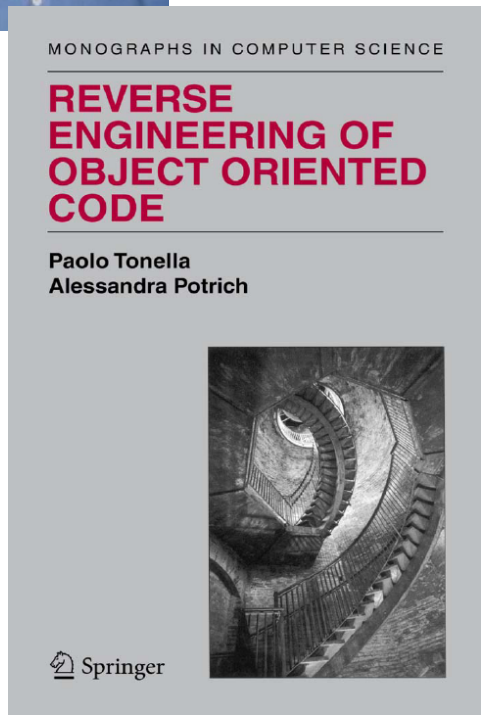
Announcements

- Reminder Assignment 1 **deadline**: Feb 19, 23:59.
  - Do not wait till 23:58…

- Guest lecture by **Prof Vinju** postponed to Feb 24.

Sources

Architecture reconstruction slides Rainer Koschke (in German)
http://www.informatik.uni-bremen.de/st/lehredetails.php?id=3&lehre_id=309



Where are we now?

- Last time: requirements
- This week: architecture

- [IEEE Std. 1471-2000] Software architecture is the fundamental organization of a system embodied in
  - its components,
  - their relationships to each other and to the environment,
  - and the principles guiding its design and evolution.

Components? Rel

- Many views:
  - Kruchten's 4+1
  - Siemens
  - Zachman
  - Perry and Wolf
  - Clements et al
  - …

- M: module: static structure
  - decomposition
  - use
  - generalization
  - layers
- CC: component & connectors: runtime structure
  - pipe and filter
  - shared data
  - publish and subscribe
  - client server
  - peer-to-peer
  - communicating processes
- A: allocation: embedding in organizational development context
  - deployment
  - implementation
  - work assignment

Architecture

- As intended

evolution

- As described
  - Architecture Description Languages
    - Should express different views

evolution

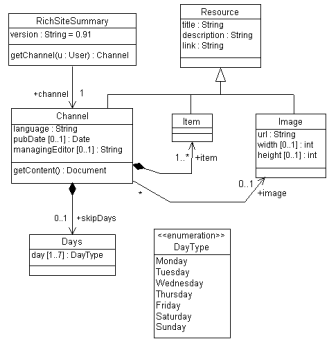
- As implemented
  - Code, deployment
  - From code to architecture: reverse engineering
    - Should extract different views

Reverse Engineering

- **[Chikofsky, Cross II 1990] 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 a higher level of abstraction.**

- **Reverse engineering: different meanings**

```
#include <stdio.h>
#include <string.h>

char stack [3][255];
int stackPosition = 0;

void Push()
{
      strcpy(stack[stackPosition], yytext);
      stackPosition++;
}

char *Pop()
{
      return(stack[--stackPosition]);
}
```



Our Reverse Engineering

- **[Koschke 2008] Architecture reconstruction is the process of analyzing a subject system to reconstruct architectural views.**

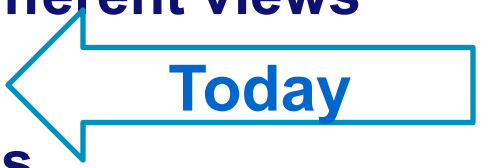
- **Different views require different architecture reconstruction techniques!**
  - **No silver bullets!**

How can we choose the right AR approach? GQ(V)M!

- **Goals**: What problem does the AR try to solve?
  - Ex.: Changes affect too many modules
  - Goal: Assess and improve current modularisation
- **Questions**: What do we need to know to achieve the goal?
  - What are the dependencies between the modules?
  - How can one improve the modularisation?
- **Views**: Which views are needed to answer the questions?
  - Module-Use Views: as-is + improved
- **Metrics**: How can we quantify the answers?
  - Number of dependencies between the modules, …

Architecture Reconstruction ⇒ UML

- **UML**
  - ***De facto* standard architecture description language**
  - **Various diagrams corresponding to different views**
    - **Structure:**
      - **Class diagrams, package diagrams**
    - **Behaviour:**
      - **Sequence diagrams, state machines, activity diagrams**

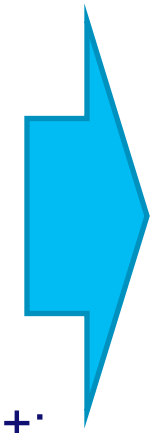


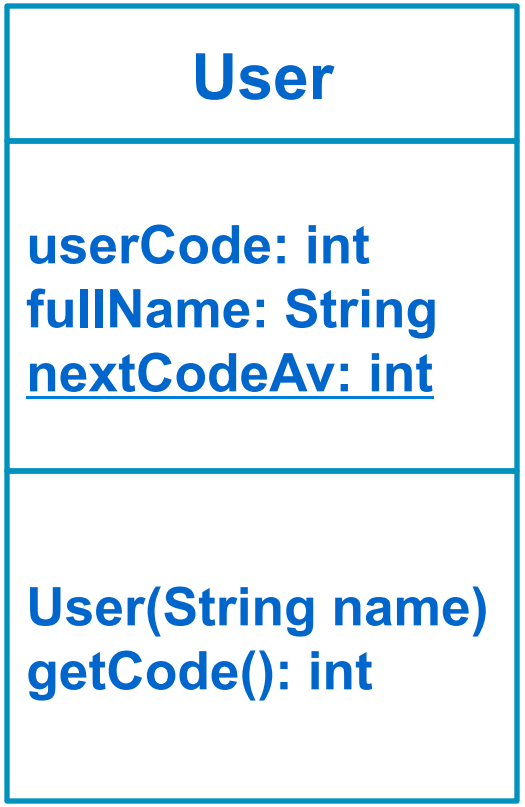
- **AR is implemented in many commercial, open-source and academic tools**
  - **Assignment 2!**
  - **@TUe: CPP2XMI, I2SD**

AR: Class diagrams

- **Basic idea**

```
import java.util.*;

class User {
    int userCode;
    String fullName;
    static int nextCodeAv = 0;
    …
    public User(String name) {
            fullname = Name;
            userCode = User.nextCodeAv++;
    }

    public int getCode() {
            return userCode;
    }
}
```





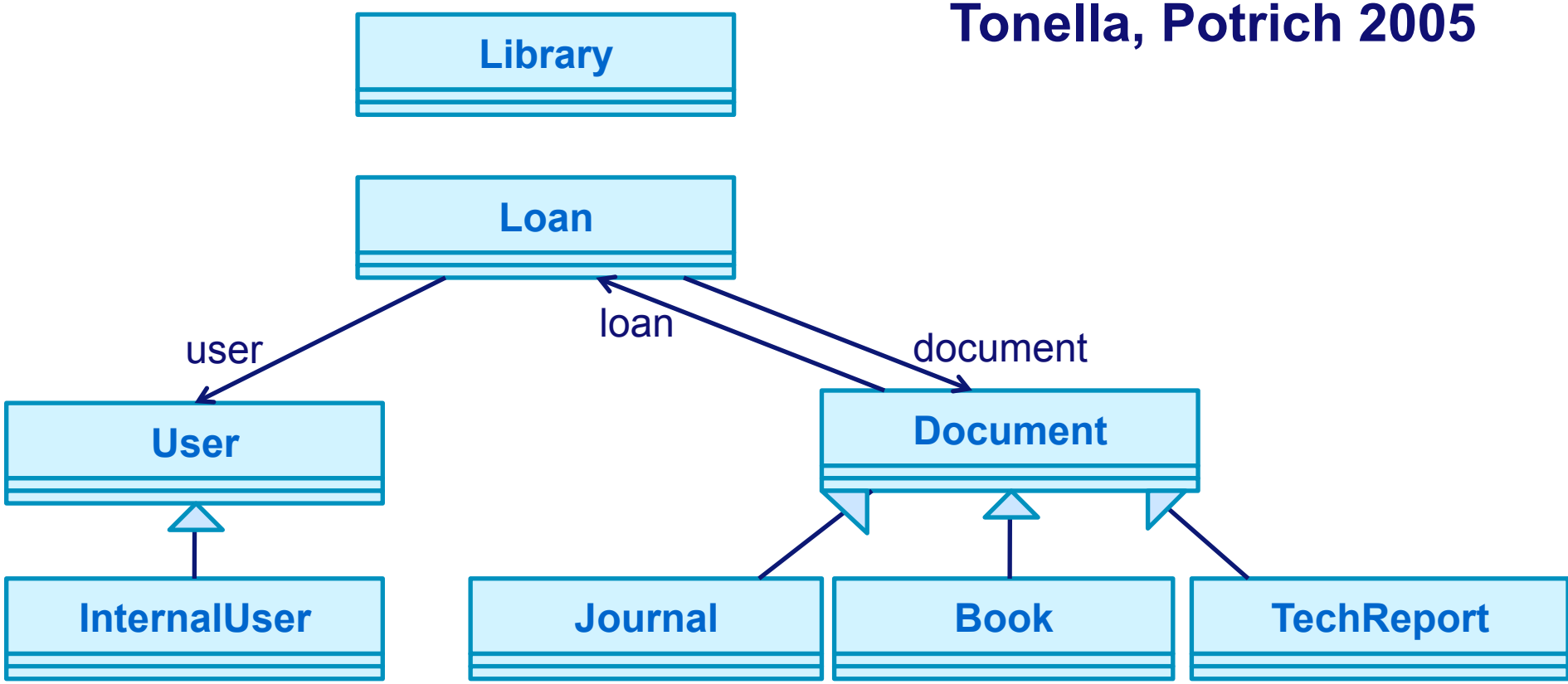
Do not draw the library classes

What about the relationships?

Tonella, Potrich 2005

Relationships	Code
Association / aggregation 	class A { B b; }
Dependency 	class A { void f(B b) {b.g(); } } class A { void f() {B b; … b.g();} }
Generalization 	class A extends B {…}
Realization 	class A implements B {…}

Library example



- **Looks strange…**
- **No relations between the library, its documents and its users?**

What is going on?

```
class Library {
    Map documents = new HashMap();
    Map users = new HashMap();
    Collection loans = new LinkedList();

    public boolean addUser(User user) {
        if (!users.containsValue(user)) {
            users.put(
                new Integer(
                    user.getCode()),
                user);
            return true;
        }
        return false;
    }
}
```

- **Relation between Library and Map/Collection**
  - **Not drawn: library classes**
- **Containers are weakly typed**
  - **Collect objects of the type that is not yet declared**
- **Relation between Library and User is missed**

What do we need?

- **Flow of information about objects**
  - **creation by allocation statements,**
  - **assignment to variables,**
  - **storage in class fields,**
  - **usage in method invocations**

- **Data-flow analysis**
  - **We need a data structure to propagate the information**
  - **It should represent relations between objects**

$$out(n) = in(n) \cup gen(n) \setminus kill(n)$$

  - **Both backward and forward propagation**

$$in(n) = \bigcup_{p \in succ(n)} out(p) \quad \textbf{of} \quad in(n) = \bigcup_{p \in pred(n)} out(p)$$

Data-flow analysis

Recall:

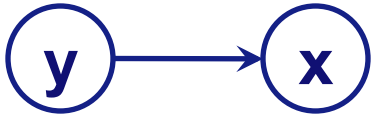
- **We need a data structure to propagate the information**
- **It should represent relations between objects**

Solution: Object Flow Graph

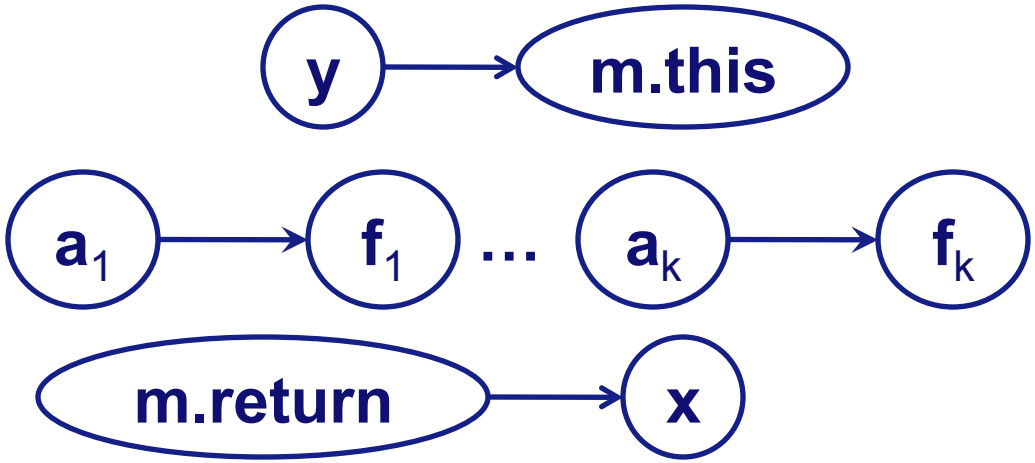
- **Vertices: variables, fields, method parameters (everything that can store data)**
- **Edges: represent data flow (see next slide)**

Basic idea: Object Flow Graph

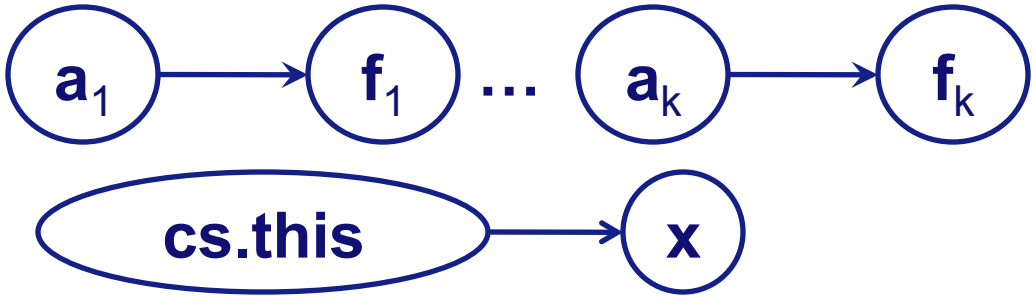
- Vertices: variables, fields, method parameters, method invocations and returns
- x = y



- x = y.m(a_1, …, a_k)
  - Method m(f_1, …, f_k)

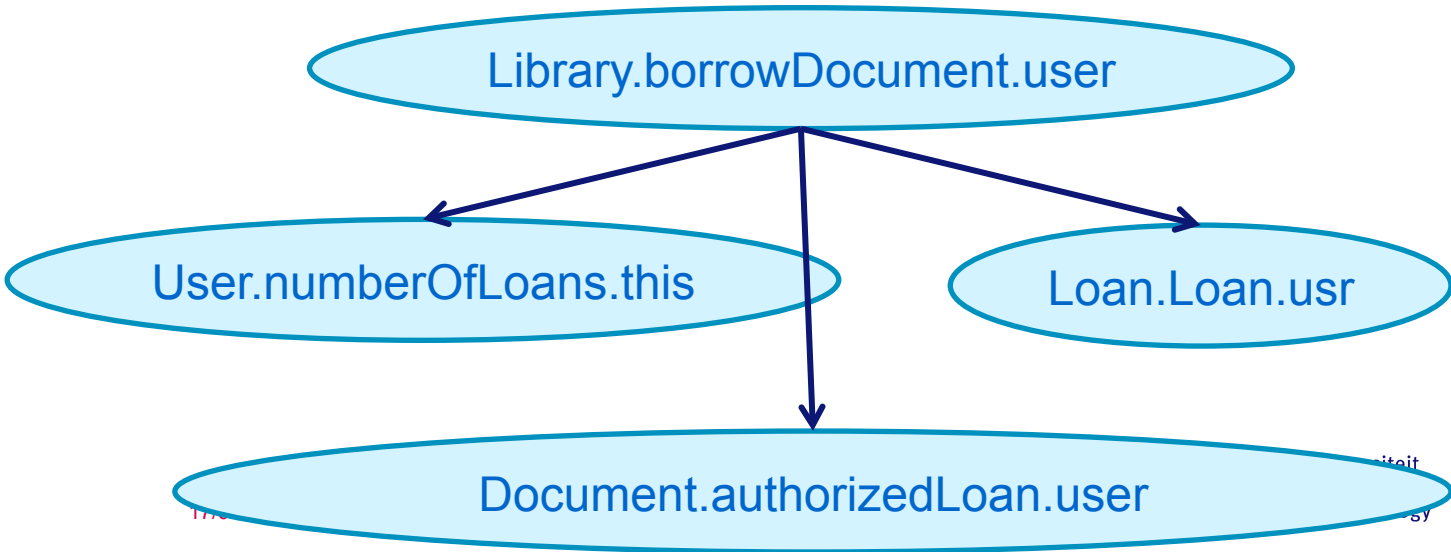


- x = new c(a_1, …, a_k)
  - cs – constructor of c



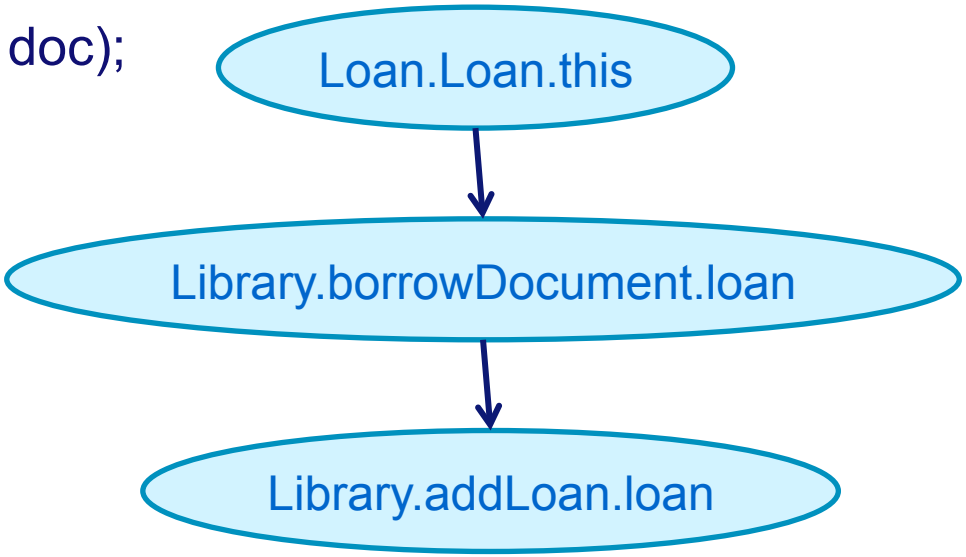
Example: Object Flow Graph

```
public boolean borrowDocument(User user, Document doc) {
        if (user == null || doc == null) return false;

        if (user.numberOfLoans() < MAX_NUMBER_OF_LOANS &&
            doc.isAvailable() && doc.authorizedLoan(user)) {

            Loan loan = new Loan(user, doc);
            addLoan(loan);
            return true;
        }
        return false;
}
```



Example: Object Flow Graph

```
public boolean borrowDocument(User user, Document doc) {
        if (user == null || doc == null) return false;

        if (user.numberOfLoans() < MAX_NUMBER_OF_LOANS &&
            doc.isAvailable() && doc.authorizedLoan(user)) {

            Loan loan = new Loan(user, doc);
            addLoan(loan);
            return true;
        }
        return false;
}
```

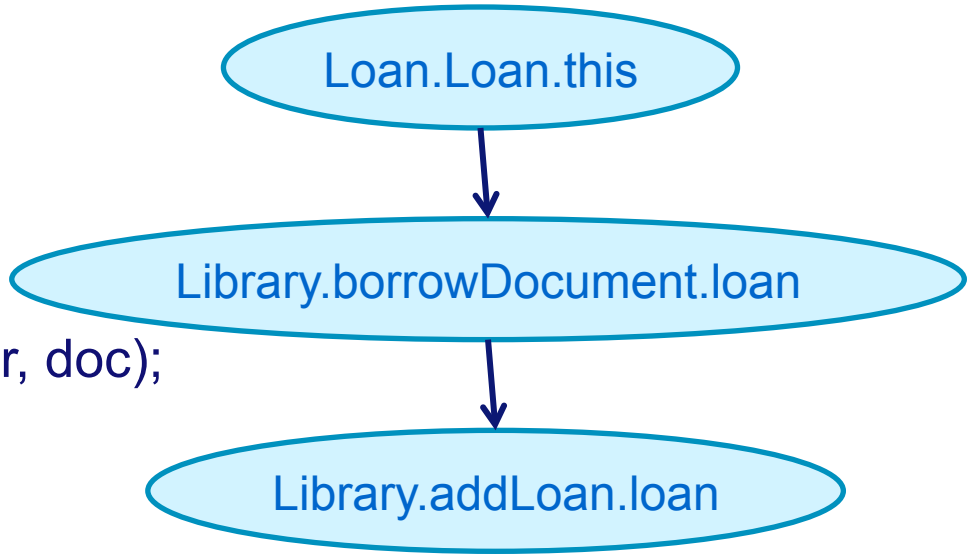


- **Control-flow is ignored**
- **Fields, parameters and methods are encoded with fully qualified names**

Question

- **Does the path in the OFG always correspond to a possible execution?**

```
if (user != null)
    Loan loan = new Loan(user, doc);
if (user == null)
    addLoan(loan);
```



- **This approach is conservative (safe):**
  - **no path in the Object Flow Graph ⇒ no execution can produce the flow**
  - **path in the Object Flow Graph ⇒ ???**

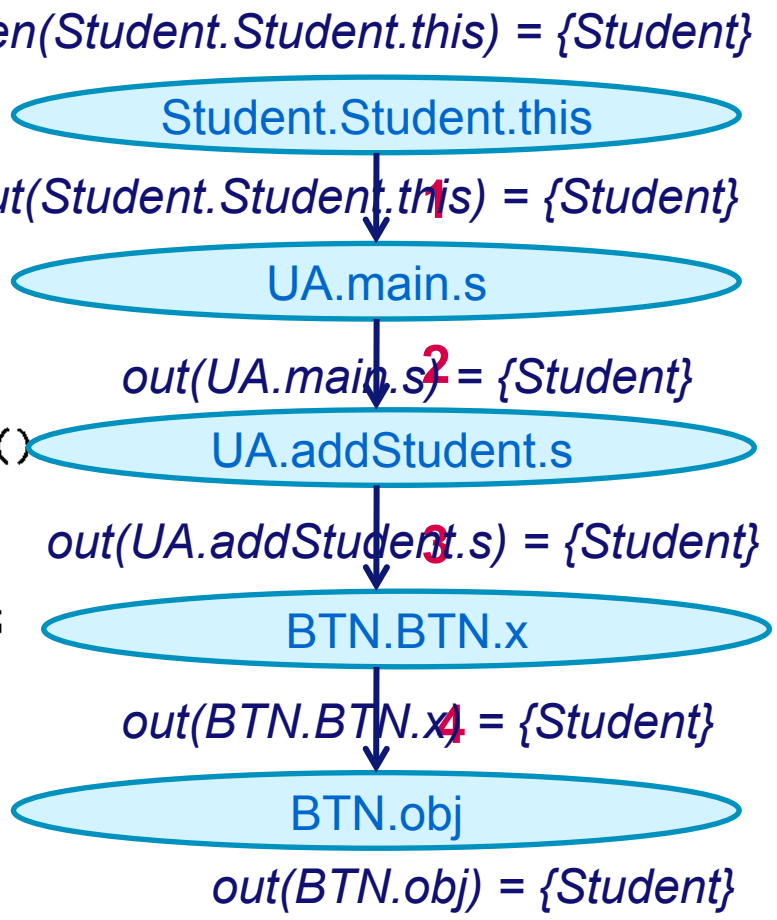
How can we use our graphs?

- **Problem: declared type ≠ actual type**
  - **Subclasses and implemented interfaces**
  - **Inheritance from a library superclass/interface is not visible in a class diagram!**

- **In general: undecidable, we try our best**
  - **For any "x = new c" node record c:** $gen(cs.this) = \{c\}$
  - **Propagate this information through the graph:**
    - **From creation to use (forward)**

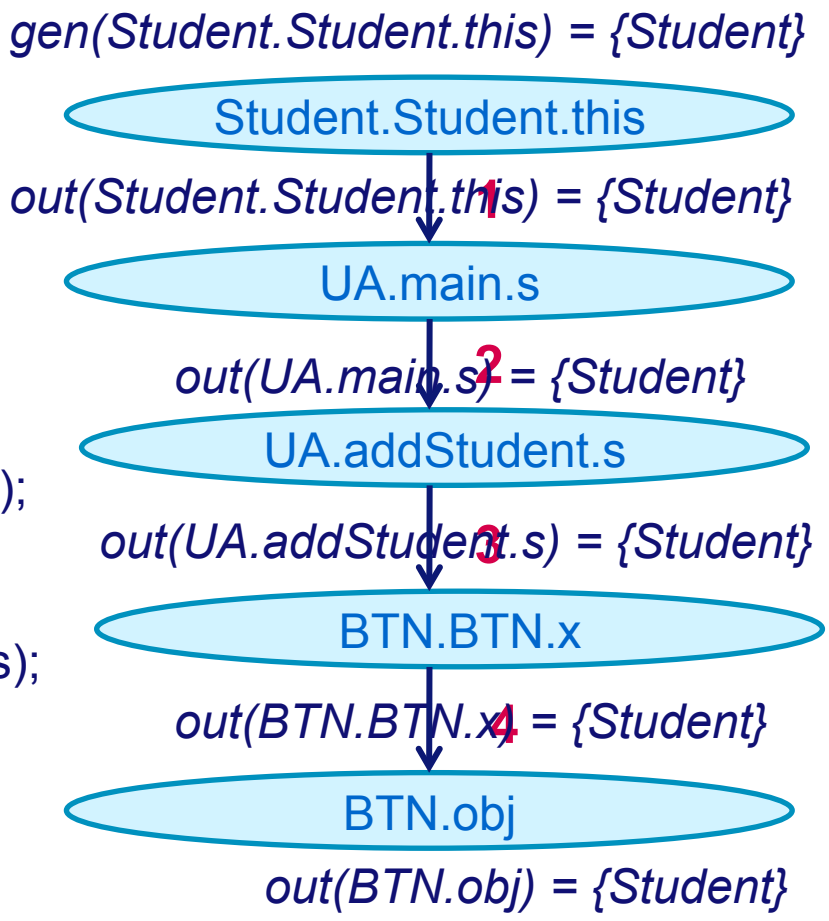
Example

```
class BinaryTreeNode {
  BinaryTreeNode left, right;
  Comparable obj;
  public BinaryTreeNode(Comparable x) {
   4 obj = x;
  }
  ...
}
class UniversityAdmin {
  static BinaryTree students = new BinaryTree();
  ...
  public static addStudent(Student s) {
   3 BinaryTreeNode n = new BinaryTreeNode(s);
     students.insert(n);
  }
  public static void main(String args[]) {
     ...
   1 Student s = new Student("J. Smith");
   2 addStudent(s);
     ...
  }
}
```



Example

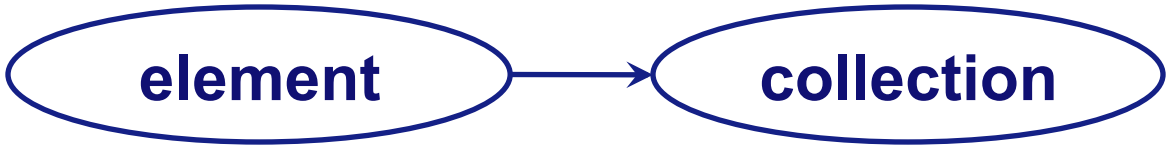
```
class BinaryTreeNode {
      BinaryTreeNode left, right;
      Comparable obj;

      public BinaryTreeNode (Comparable x) {
   4    obj = x;
      }
}
class UniversityAdmin {
      static BinaryTree students = new BinaryTree();

      public static addStudent (Student s) {
   3    BinaryTreeNode n = new BinaryTreeNode(s);
        students.insert(n);
      }

      public static void main(String args[]) {
   1    Student s = new Student("Jane Smith");
   2    addStudent(s);
      }
}
```



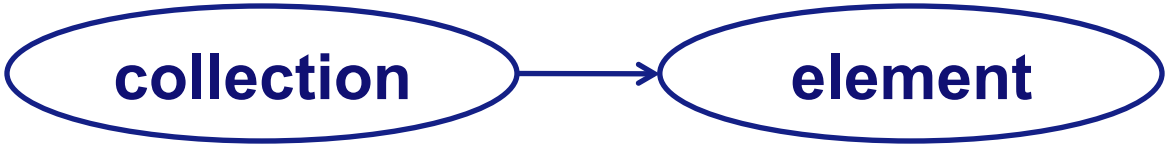
What about containers?

- Containers: Map, List, Collection, Vector…
- Weakly typed: Collect objects of the type that is not yet declared.

- Operations: insert and extract
  - Information flow
    - insert



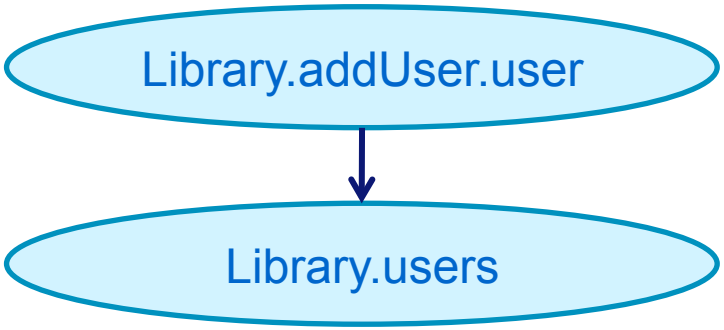
    - extract



- The same approach works!

Back to the original problem: Missing link

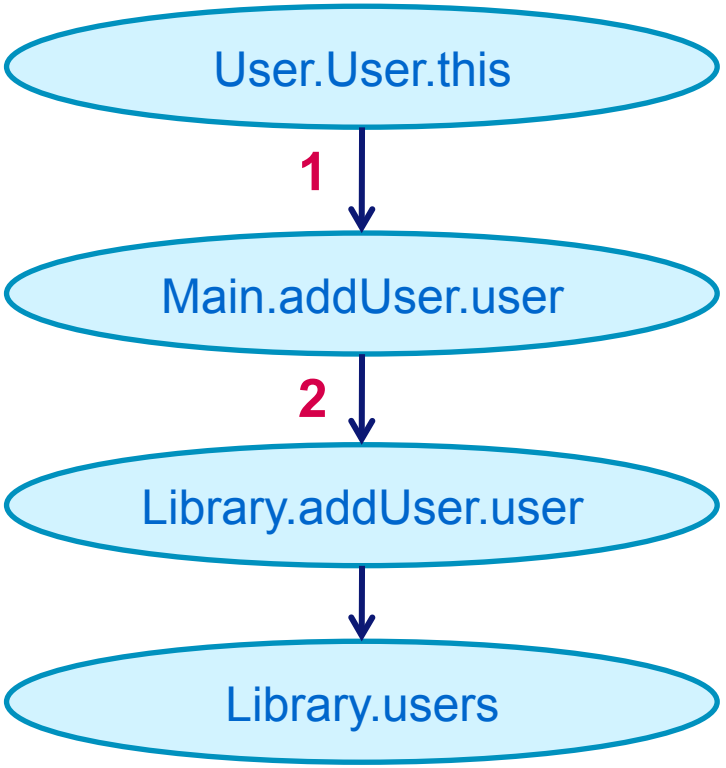
```
class Library {
    Map documents = new HashMap();
    Map users = new HashMap();
    Collection loans = new LinkedList();

    public boolean addUser(User user) {
        if (!users.containsValue(user)) {
            users.put(
                new Integer(
                    user.getCode()),
                user);
            return true;
        }
        return false;
    }
}
```



Back to the original problem: Missing link

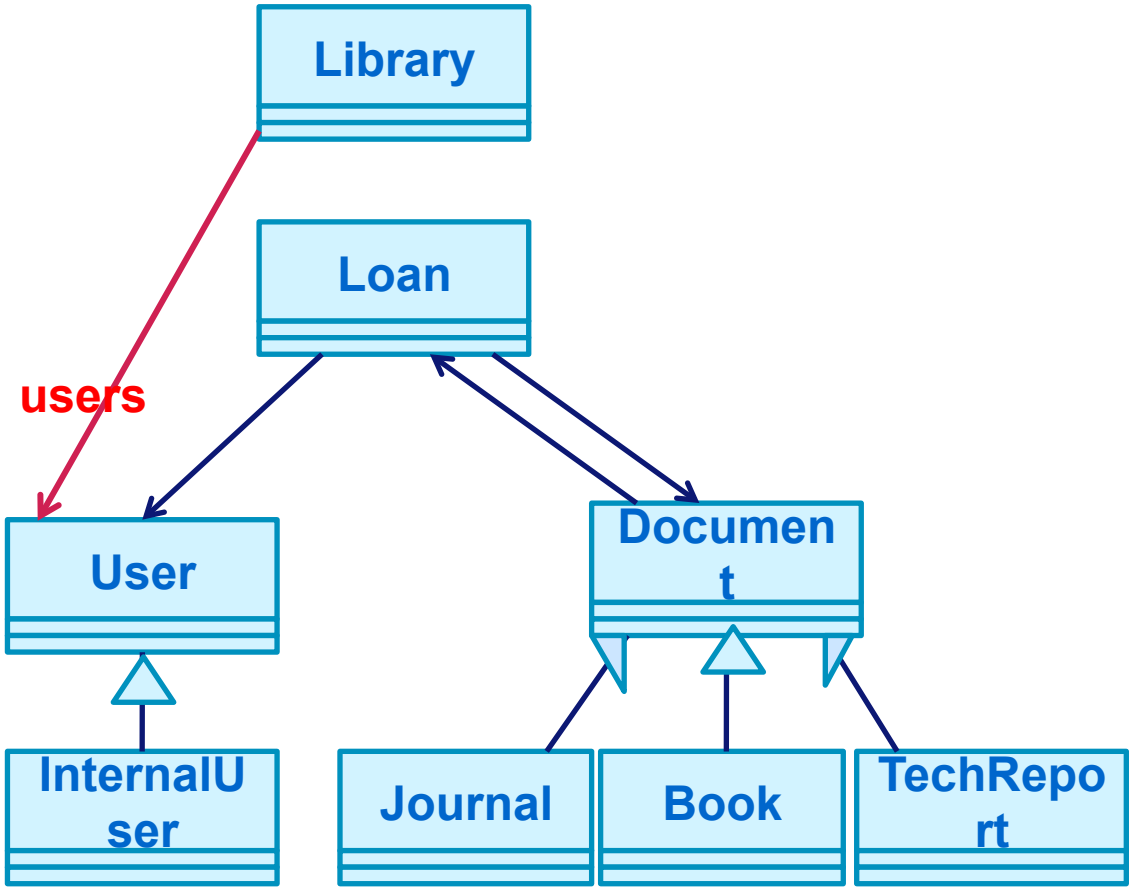
```
class Main {
    static Library lib = new Library();
    …
     public static void addUser(String cmd) {
         …
    1    User user = new User(args[0], args[1])
    2    lib.addUser(user);
         …
     }
}
```

gen(User.User.this) = {User}

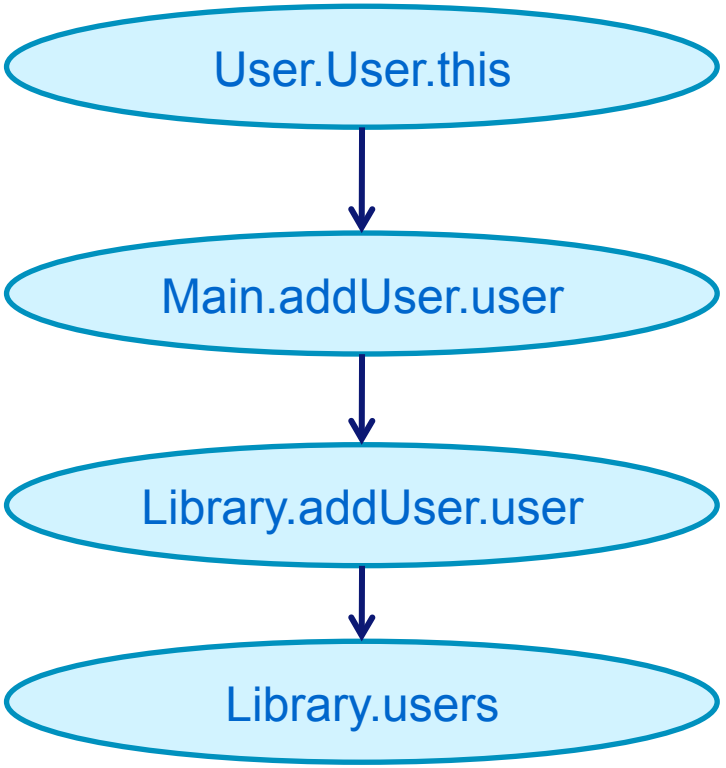


out(Library.users) = {User}

Back to the original problem: Missing link



gen(User.User.this) = {User}



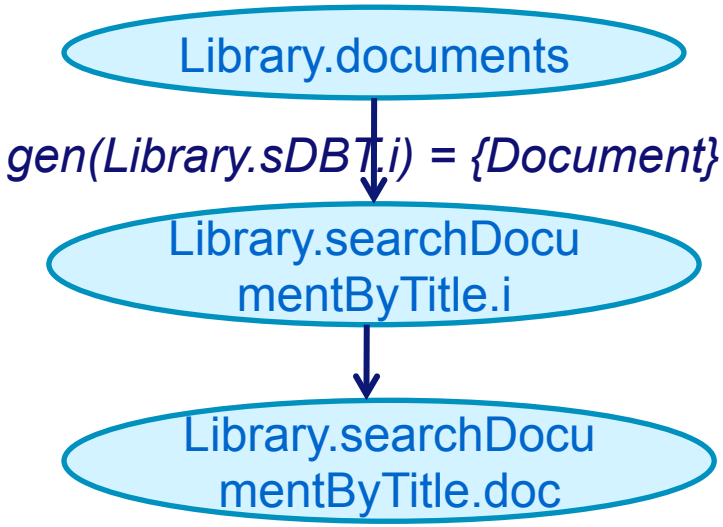
out(Library.users) = {User}

Is this all?

- **Associations can be introduced by extraction:**

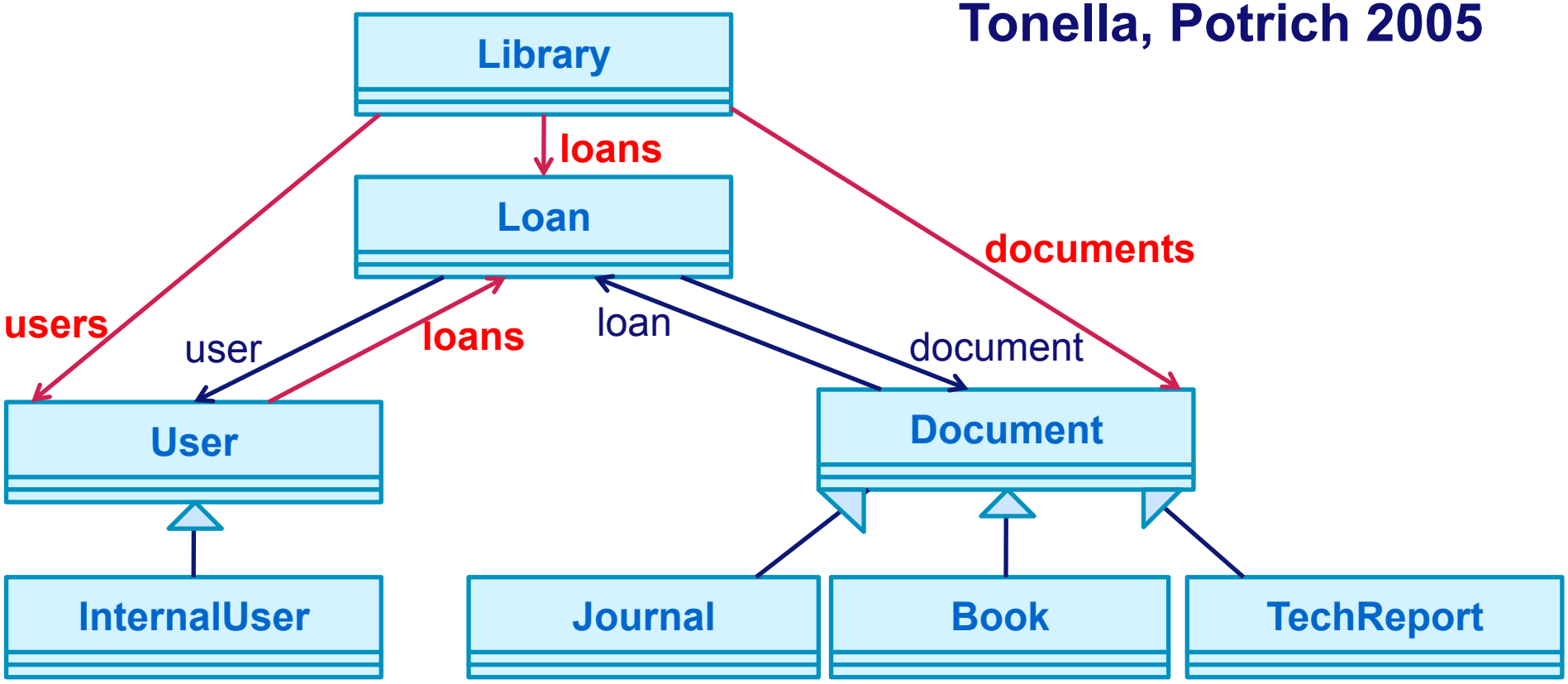
```
public List searchDocumentByTitle(String title) {
    List docsFound = newLinkedList();
    Iterator i = documents.values().iterator();

    while (i.hasNext()) {
        Document doc = (Document)i.next();
        if (doc.getTitle().indexOf(title) != -1)
            docsFound.add(doc);
    }
    return docsFound;
}
```

out(Library.documents) = {Document}



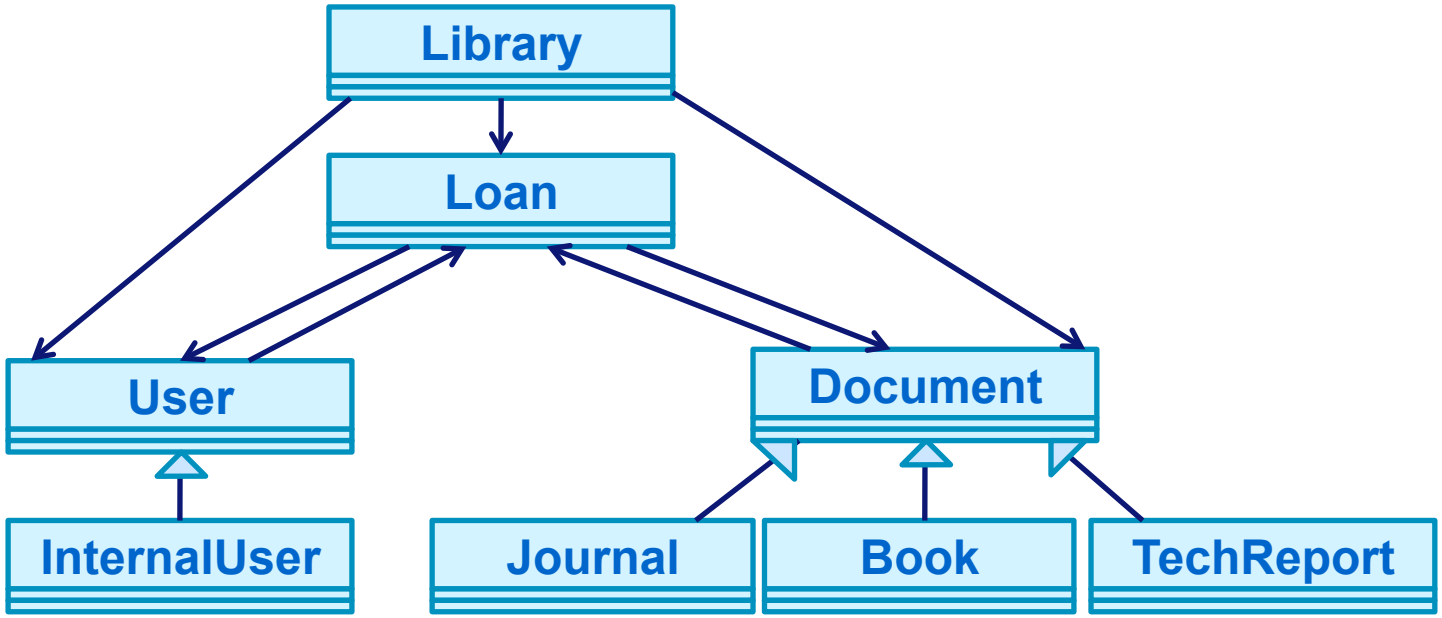
- **Data-flow analysis progresses backwards!**

Concluding the example…



- **Red associations were omitted by the “basic” AR technique.**

Concluding the example…



- **Red associations were omitted by the “basic” AR technique.**

Summary so far: AR – Class diagrams

- Implemented in many tools
  - How precise? Depends on the tool…
- Basic technique is very simple but imprecise
- Improving precision necessitates data-flow analysis

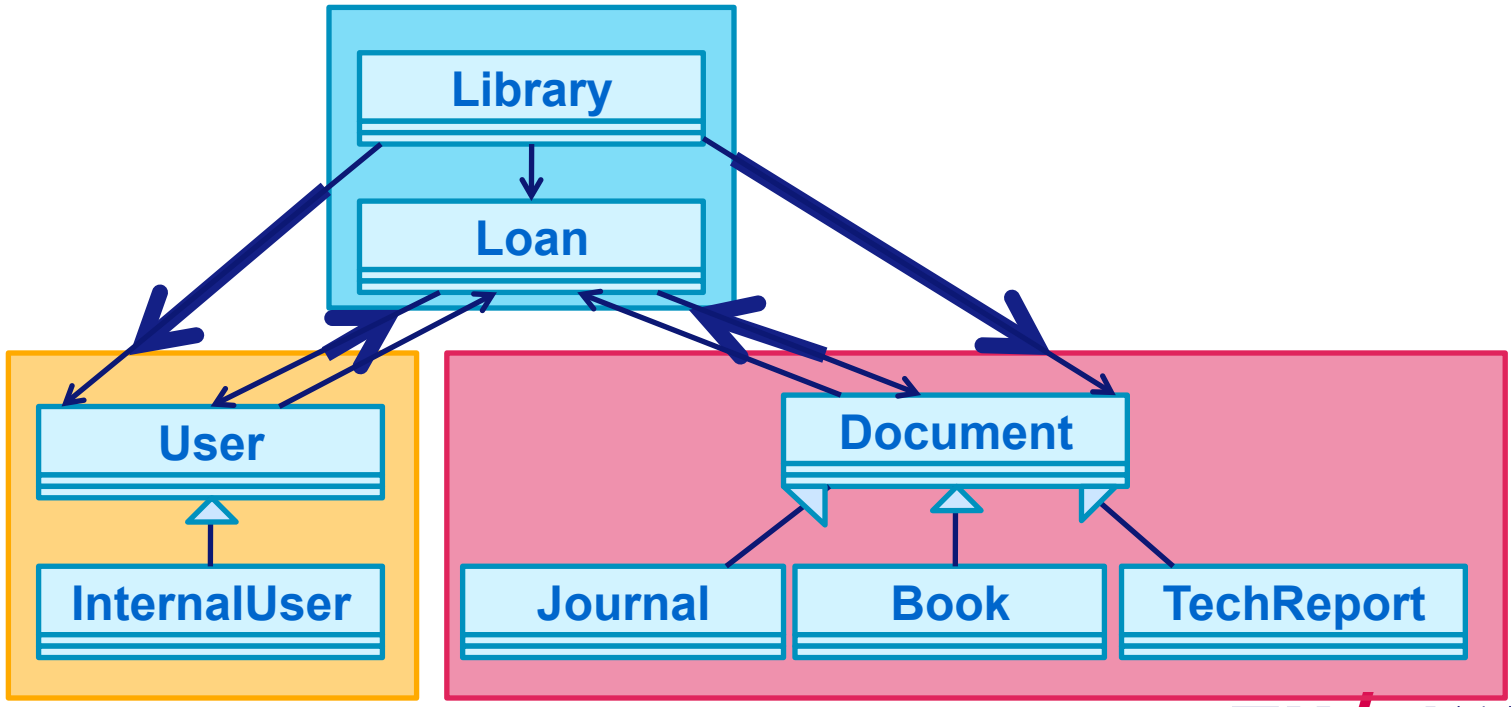
Is this the Holy Grail?..

unused classes

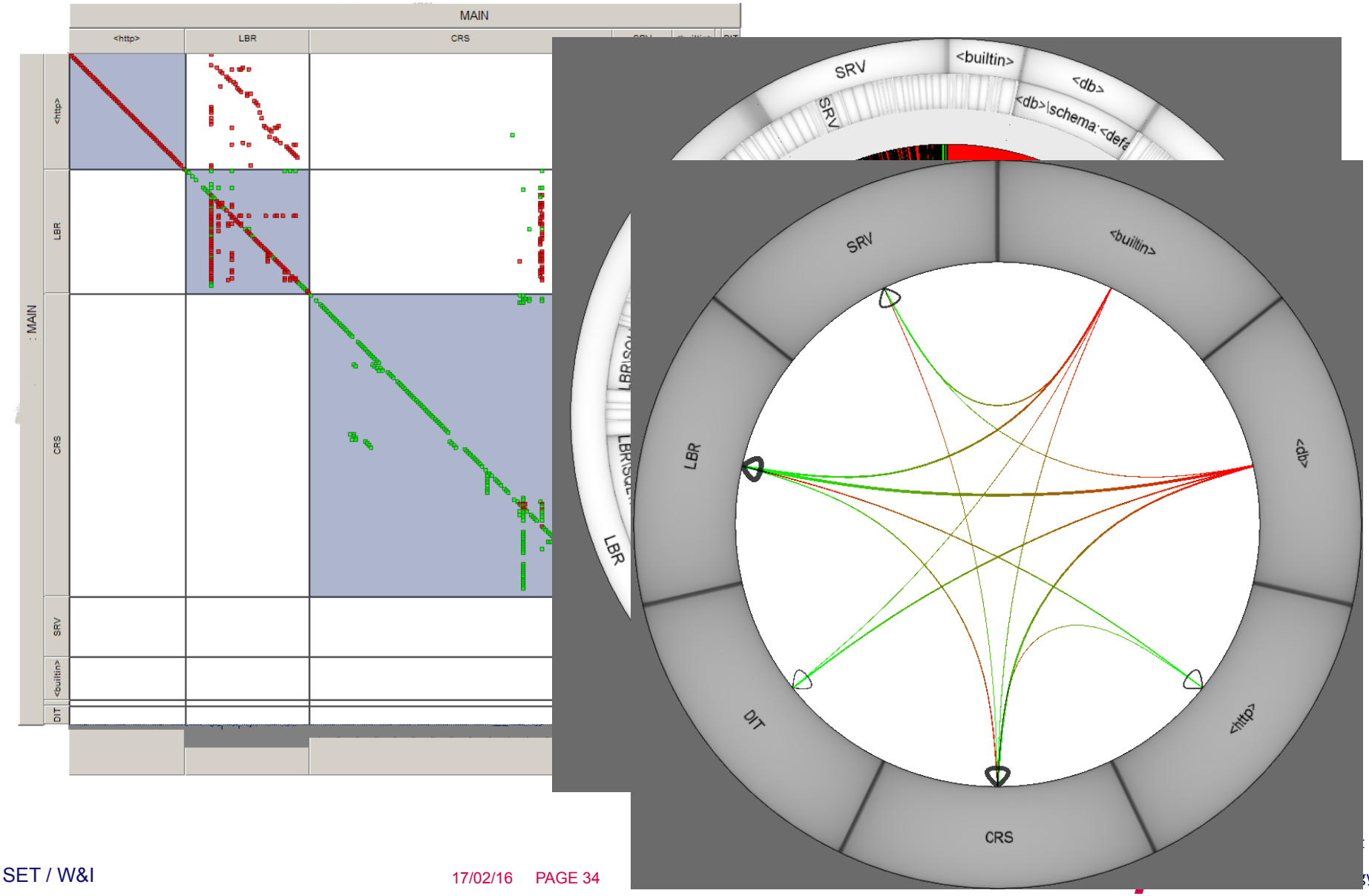
We need something better...

Packages

- Higher level of abstraction
- OO-languages usually contain explicit mechanism:
  - packages (Java), namespaces (C++), modules (Modula)
- Easily derived from the class diagram



Better visualizations

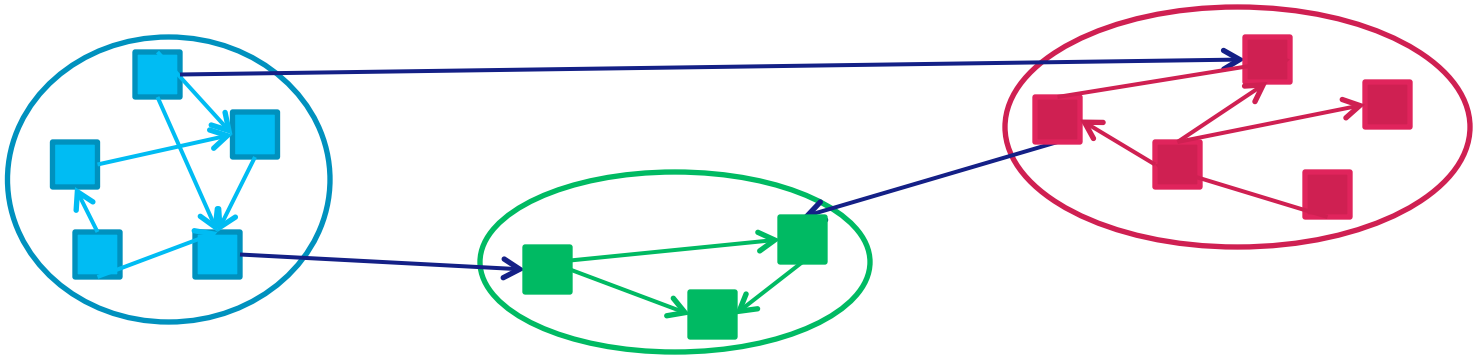


What if the packages are unknown?

- When can it happen?
  - Suitability for evolution
    - Packaging is being assessed for appropriateness
  - Implementation of evolution
    - Legacy language with no “packaging”
    - Inappropriate packaging

- What can we do?
  - Let the user do it her/himself!
  - Do it automatically:
    - Join “similar” classes together: clustering
    - Packages can be composed into larger packages:
      - Hierarchical clustering

What is a good modularization?



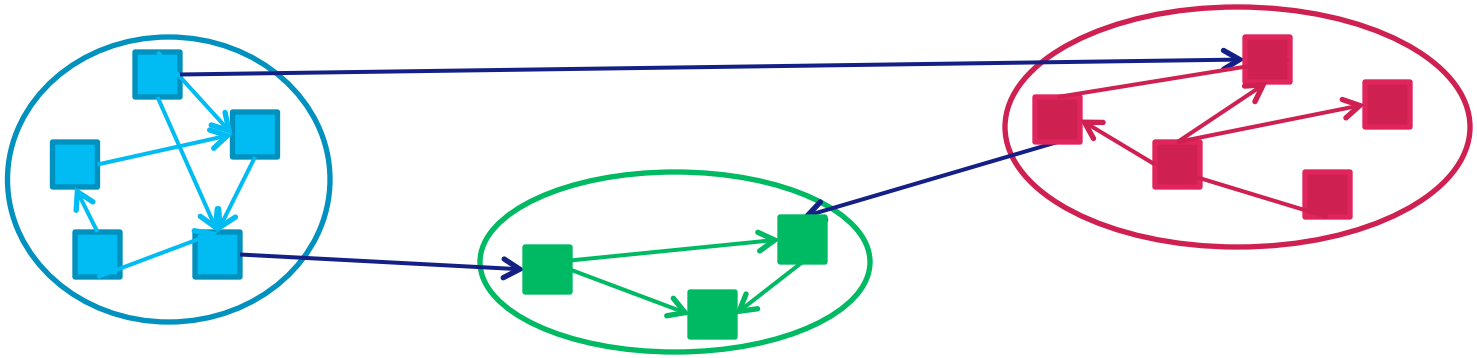
- **Many intra-package dependencies: high cohesion**

$$A_i = \frac{\mu_i}{N_i^2} \quad \text{or} \quad A_i = \frac{\mu_i}{N_i(N_i - 1)}$$

μ_i - **Number of dependencies in package i**

N_i - **Number of classes in package i**

What is a good modularization?



- **Many intra-package dependencies: high cohesion**

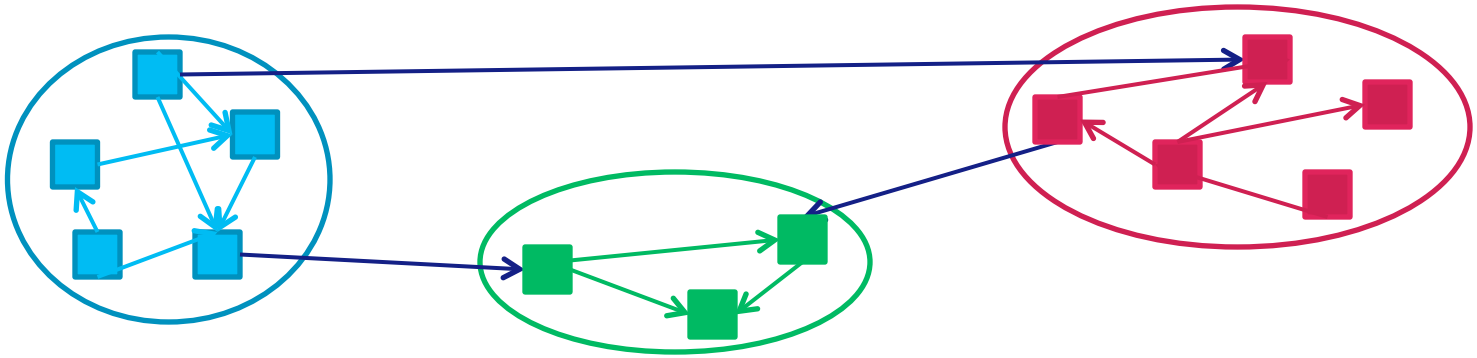
$$A_i = \frac{\mu_i}{N_i^2} \quad \text{or} \quad A_i = \frac{\mu_i}{N_i(N_i - 1)}$$

- **Few inter-package dependencies: low coupling**

$$E_{i,j} = \frac{\varepsilon_{i,j}}{2N_iN_j}$$

$\varepsilon_{i,j}$ **- Number of dependencies between packages i and j**

What is a good modularization?



- **Many intra-package dependencies: high cohesion**

$$A_i = \frac{\mu_i}{N_i^2} \quad \text{or} \quad A_i = \frac{\mu_i}{N_i(N_i - 1)}$$

- **Few inter-package dependencies: low coupling**

$$E_{i,j} = \frac{\varepsilon_{i,j}}{2N_iN_j}$$

- **Joint measure**

$$MQ = \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{2}{k(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k} E_{i,j}$$

k **- Number of packages**

Modularity Quality

Cohesion

Coupling

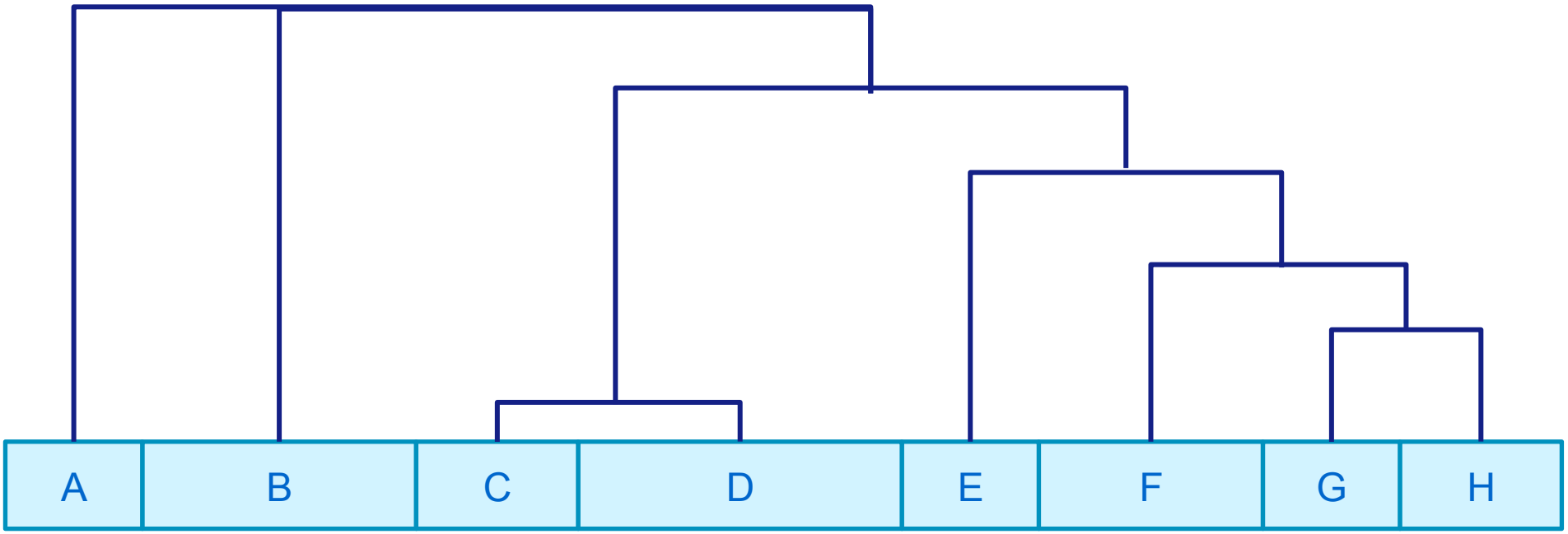
$$MQ = \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{2}{k(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k} E_{i,j}$$

- What does MQ = -1 mean?
  - No cohesion, maximal coupling
- MQ = 1?
  - No coupling, maximal cohesion

Hierarchical Clustering

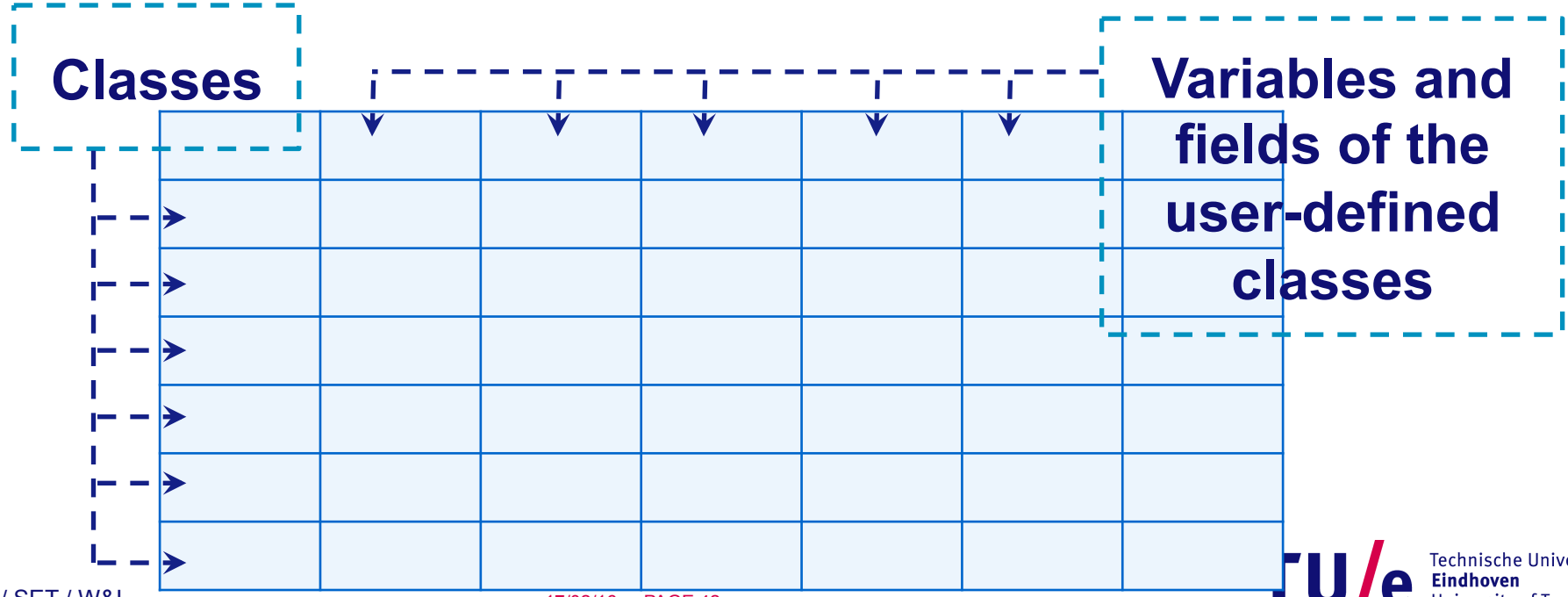
- Init: every element is a cluster on its own
- While (#clusters > 1) do
  - Calculate similarity between all pairs of clusters
  - Select two most similar clusters
    - Single linkage: max pairwise similarity
      - Reduces coupling
    - Complete linkage: min pairwise similarity
      - Increases cohesion
      - *Usually* more important
  - Merge these clusters

Hierarchical clustering example



Hierarchical clustering: Classes

- Class description: feature vectors
  - Dimension: feature
    - Methods called, types used, words in the description
  - Coordinate: number of references to this feature
  - NB: Should be “discriminating enough”

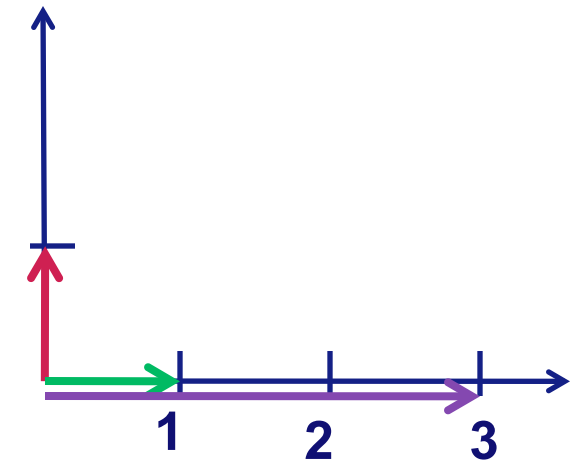


When are two vectors similar?

- **Distance between $\langle x_1,\ldots,x_n \rangle$ and $\langle y_1,\ldots,y_n \rangle$:**

$$\sum_{i=1}^{n} |x_i - y_i| \quad \text{or} \quad \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

- **What happens if the vectors are almost empty?**
  - **Distance is small**
  - **Clustering is imprecise**
    - **d(red,green) = d(green,purple)**
    - **green and purple are more similar**

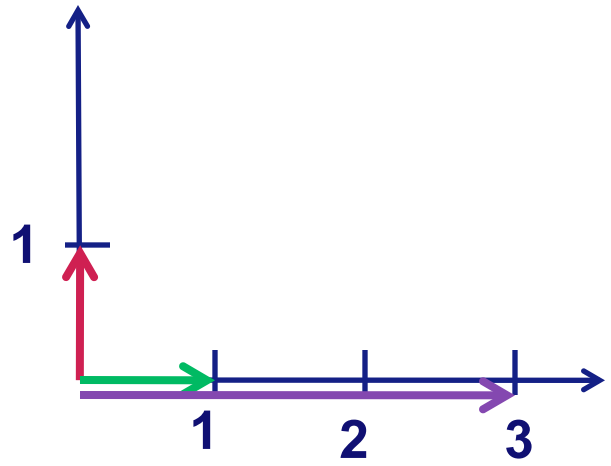


Better alternative: Similarity measures

- **Cosine measure:**

$$\frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

  - sim(red,green) = 0
  - sim(green,purple) = 1



- **Many more can be found in the literature**

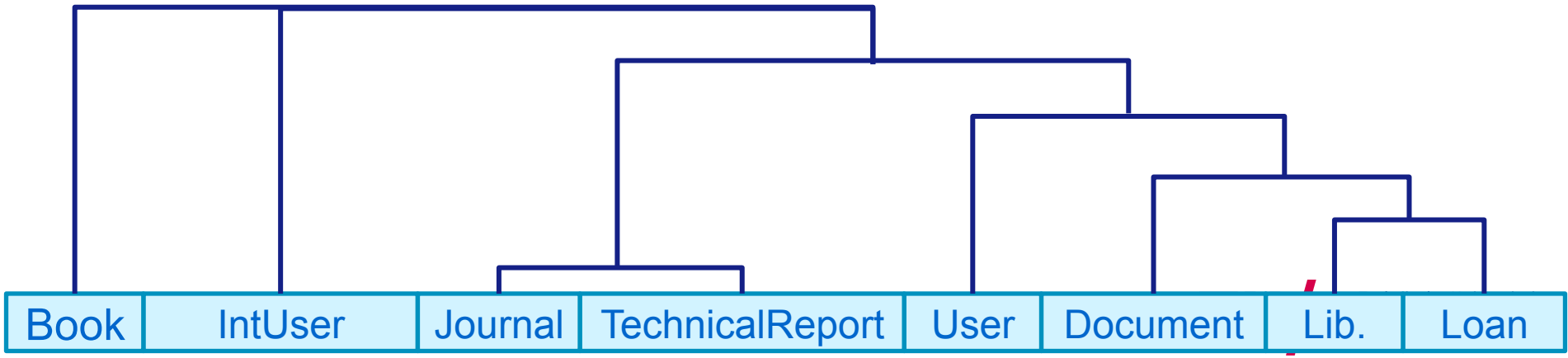
Back to the library example

- **Class description: feature vectors**
  - **Dimension: feature**
    - **Methods called, types used, words in the description**
  - **Coordinate: number of references to this feature**

	Library	Loan	Document	Book	Journal	TR	User	Internal User
Library	<0,	5,	12,	0,	0,	0,	10,	0>
Loan	<0,	1,	3,	0,	0,	0,	3,	0>
Document	<0,	2,	1,	0,	0,	0,	3,	0>
Book	<0,	0,	0,	0,	0,	0,	0,	0>
Journal	<0,	0,	0,	0,	0,	0,	1,	0>
TR	<0,	0,	0,	0,	0,	0,	1,	0>
User	<0,	3,	1,	0,	0,	0,	1,	0>
InternalUser	<0,	0,	0,	0,	0,	0,	0,	0>

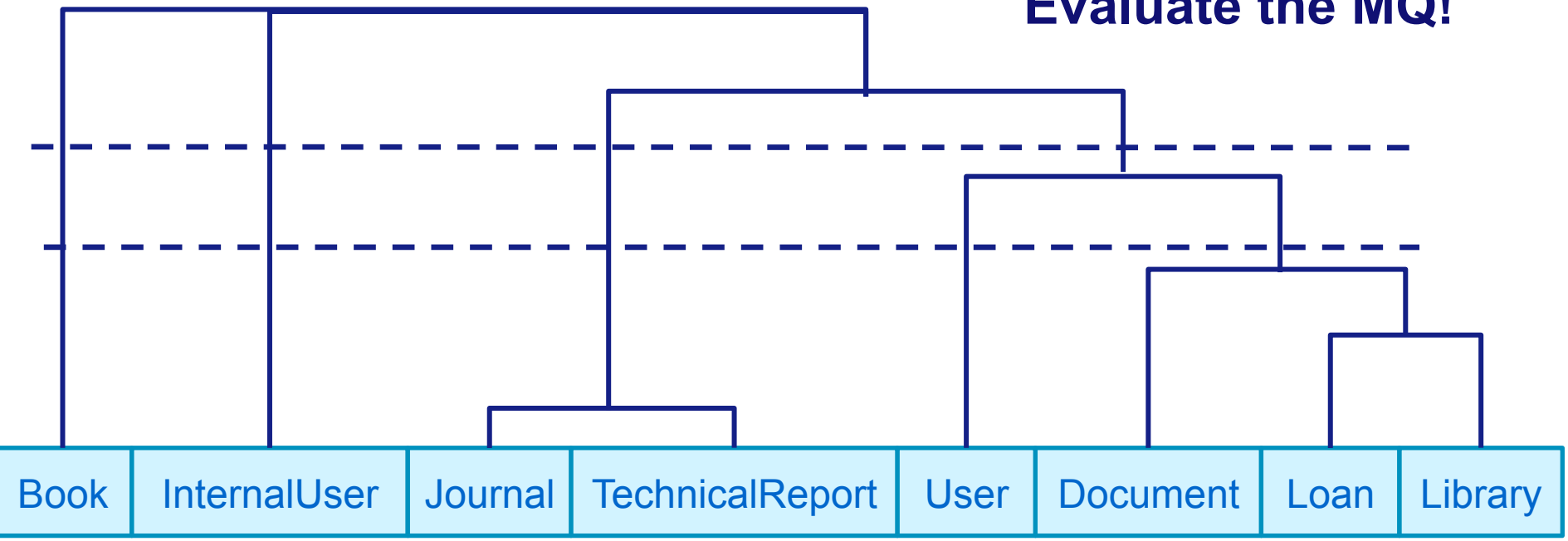
Hierarchical clustering: Library

	Library	Loan	Document	Book	Journal	TR	User	Internal User
Library	<0,	5,	12,	0,	0,	0,	10,	0>
Loan	<0,	1,	3,	0,	0,	0,	3,	0>
Document	<0,	2,	1,	0,	0,	0,	3,	0>
Book	<0,	0,	0,	0,	0,	0,	0,	0>
Journal	<0,	0,	0,	0,	0,	0,	1,	0>
TR	<0,	0,	0,	0,	0,	0,	1,	0>
User	<0,	3,	1,	0,	0,	0,	1,	0>
InternalUser	<0,	0,	0,	0,	0,	0,	0,	0>



From clusters to packages

Where can we put the cutline(s)?
Evaluate the MQ!



- **Packages are created by cut points**
- **Subpackages are created by more cut points**

Algorithm: what have we done?

We can stop earlier, when the MQ no longer improves

Some cluster partition

- Init: every element is a cluster on its own
- While ~~(#clusters > 1)~~ do
  - Calculate similarity between all pairs of clusters
  - Select two most similar clusters
    - Single linkage: max pairwise similarity
      - Reduces coupling
    - Complete linkage: min pairwise similarity
      - Increases cohesion
      - *Usually* more important
  - Merge these clusters

Minor change in cluster partition

This is a combinatorial optimization problem!

- **Hill Climbing**:
  - Start with a random cluster partitioning
  - As long as the optimization function increases
    - Select a best “neighbour” partition
      - Neighbour: small change
      - Best: with the highest opt. func. value
      - A: there may be more than one such partition

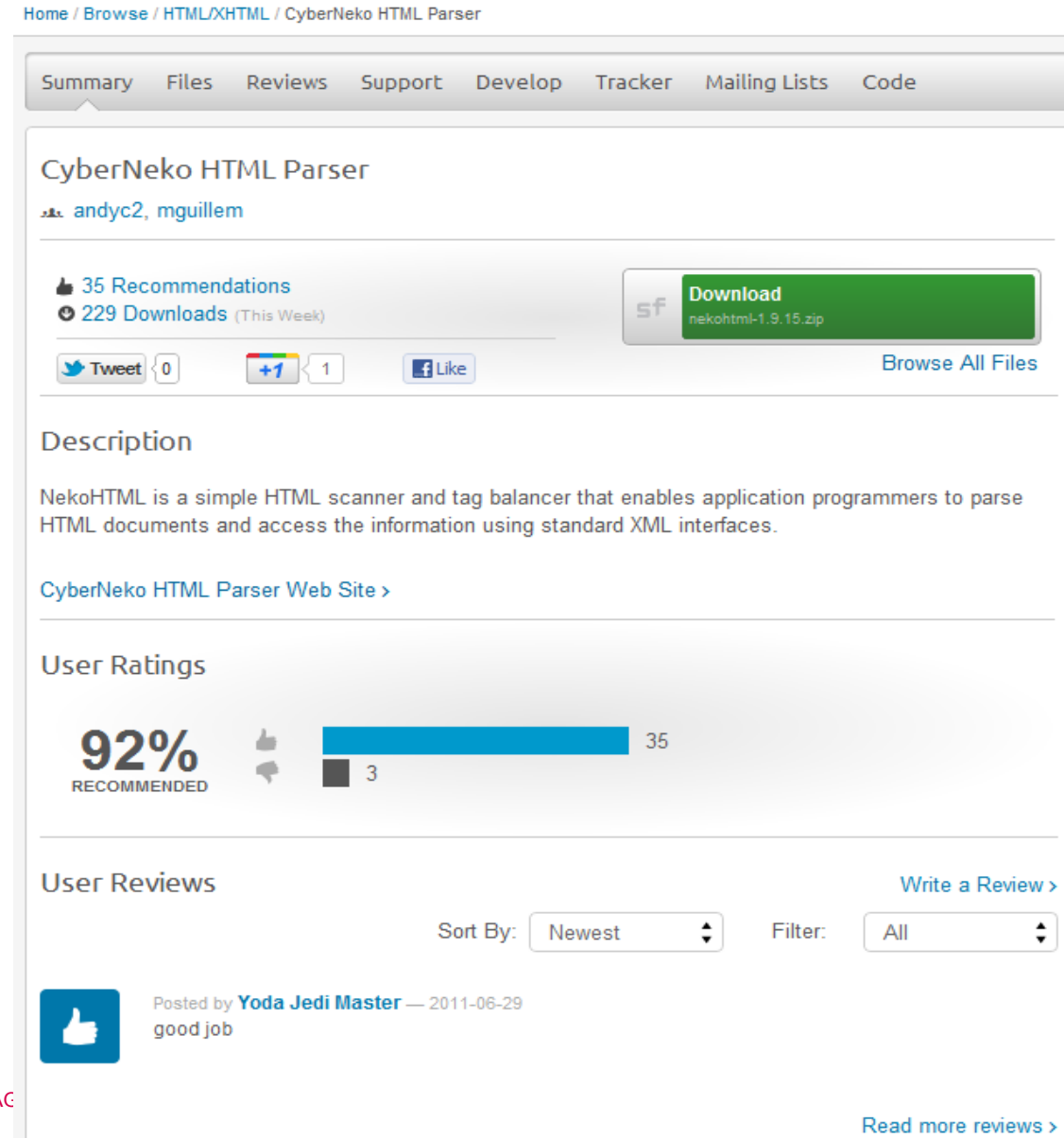
- **Problems:** local maxima, plateau, ridges
- There are better search techniques
  - Search-Based Software Engineering

What have we seen today?

- Reverse engineering/architecture reconstruction
  - Different views – different approaches
  - Goal – Question – Views – Metrics
- Class diagrams
  - Basic approach: simple, imprecise
  - Better precision requires data-flow analysis
- Package diagrams
  - “Fixed” or “user-defined” packages
  - Automated techniques:
    - Clustering
    - Search-based, e.g. Hill climbing

Assignment 2: Till March 9!

- Pairs
- Architecture reconstruction:
  - Using Rascal
  - build a small architecture reconstruction tool for Java, and
  - apply it to study evolution of CyberNeko HTML Parser
    - Dec 2007 vs June 2014



Rascal???

- Meta-programming language
- Developed at CWI
  - Jurgen Vinju and his team
  - http://www.rascal-mpl.org/

- Next Wednesday:

  Guest lecture of Jurgen Vinju about Rascal