Mining Software Repositories

Mircea Lungu

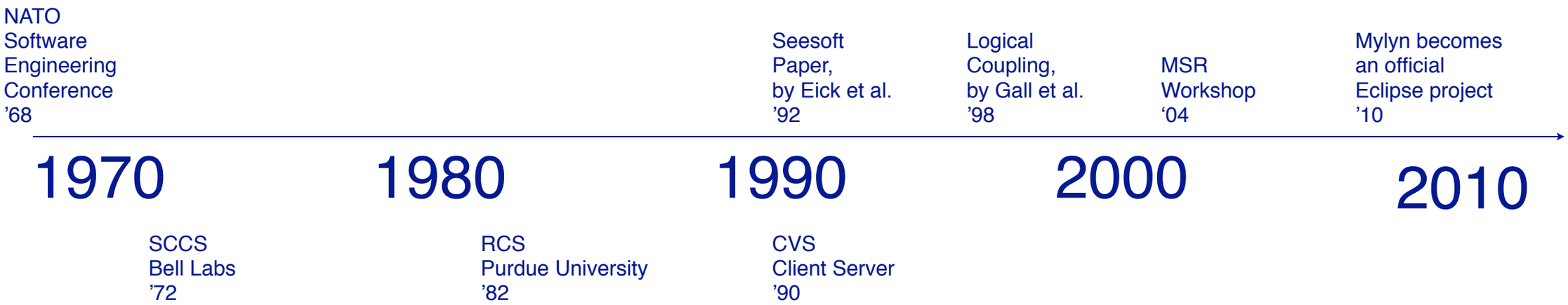
November 23, 2011

b
UNIVERSITÄT
BERN

Roadmap

- **Introduction**
- Mining the history for relationships
  - Logical coupling
- Recovering entity evolution
  - Origin analysis
  - Refactoring detection
- Mining a history for rules
  - Common error patterns
- And more...

A History of Repository-Based Analysis in Software



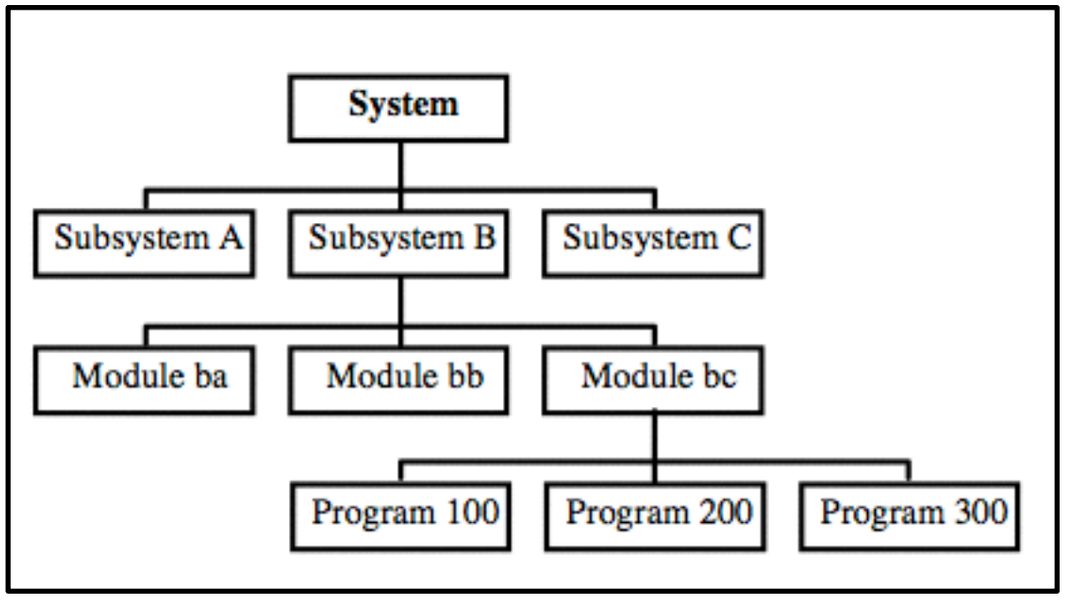
> 20 years of **software engineering** before people start doing research in **analyzing software repositories**

Roadmap

- Introduction
- **Mining the history for relationships**
  - Logical coupling
- Recovering entity evolution
  - Origin analysis
  - Refactoring detection
- Mining a history for rules
  - Common error patterns
- And more...

Logical Coupling

- Gall et al. ‘98
- Based on an an industrial case study
  - Subsystems
  - Modules
  - Programs
- Two steps
  1. Change Sequence Analysis
  2. Change Report Analysis



How to detect dependencies based on history?

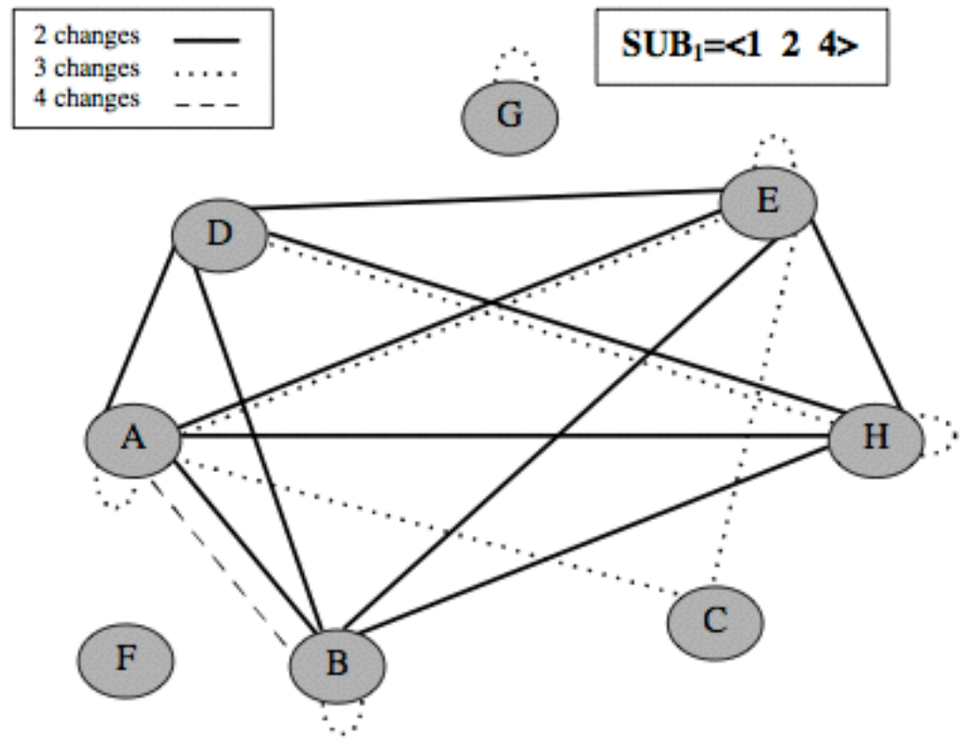
Why history based?

- Structural / Data Flow Analysis
  - Disadvantages:
    - *can not capture all the situations (i.e. writing to a file, reading from a file)*
    - *does not work with documents that are not source code*

Change Sequence Analysis

SUB_2=<1 2 3 4 6 7 9 10 14>											
A.aa.111	**1**	**2**	**3**	**4**	**6**	**7**	**9**	**10**	**14**	17	19
B.ba.222	**1**	**2**	**3**	**4**	**6**	**7**	**9**	**10**	**14**	16	18

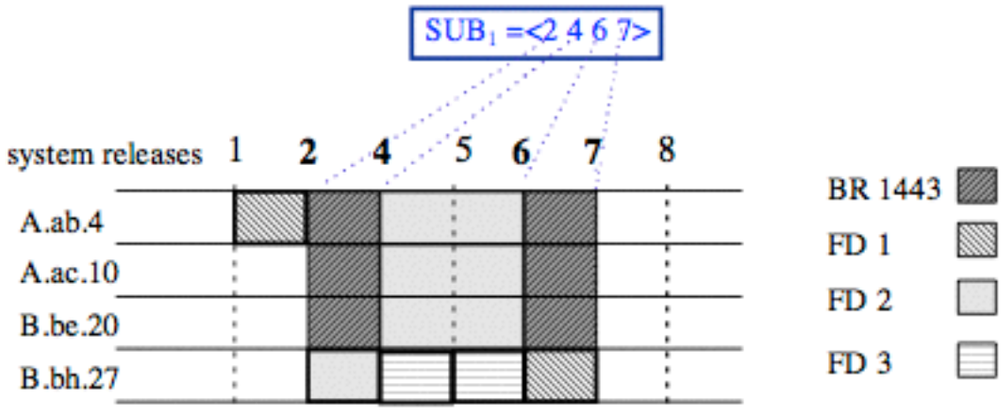
- Detects when two sub-systems change together
- Logical coupling is stronger if the subsequence is larger



Change Report Analysis

- There are two types of changes that are documented
  - Feature additions
  - Bug Requests

- The coupling between subsystems must be verified



Logical Coupling Summary

- Advantages
  - Does not require the code to compile
  - Can work with any types of documents
- Simplification
  - Versioning systems in the real world are a mess (CVS)
    - *does not group*

Alternative read:

Mining Version Histories to Guide Software Changes

Zimmerman et al.

Roadmap

- Introduction
- Mining the history for relationships
  - Logical coupling
- **Recovering entity evolution**
  - Origin analysis
  - Refactoring detection
- Mining a history for rules
  - Common error patterns
- And more...

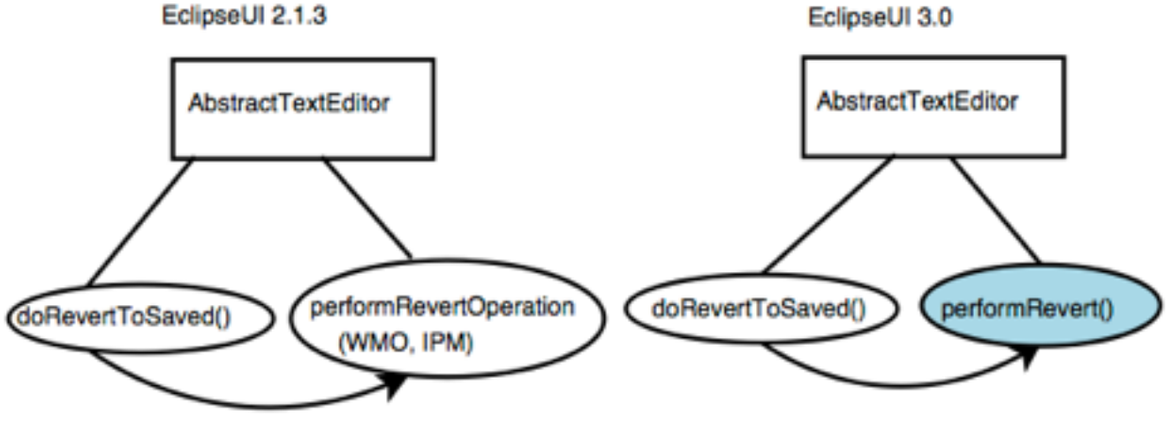
Origin Analysis

- Tu & Godfrey ’02
- Works at the function level
- Combines
  - Bertillonage Analysis
    - *Assumes that Complexity Metrics do not change much*
  - Dependency Analysis
    - *Assumes that relationships do not change much*

How to know if an entity is the same in two versions?

Refactoring Detection

- Dig et al. ’06 detect refactorings of Packages, Classes, Methods
- Combination of **syntactic** and **semantic** analysis
- Shingles Algorithm



How to detect refactorings in object-oriented systems?

The Approach of Dig et al.: The Shingles Algorithm

- Input
  - sequence of tokens representing method body without signature
- Output
  - Multi-sets of integers
  - Similar inputs generate similar outputs

- Algorithm
  - W: window size
  - S: maximum set size
  - Compute hashes while sliding the window
  - Sort shingles and keep the first S

The Approach of Dig et al.: Shingles Algorithm (Example with W=2 and S=10)

```
void doRevertToSaved() {
IDocumentProvider p= getDocumentProvider();
if (p == null)
   return;
performRevertOperation(createRevertOperation(),
                getProgressMonitor());
}
```

Shingles: { -1942396283, -1672190785,
-12148775115, -56733233372, 208215292,
1307570125, 1431157461,
190471951, 969607679 }

```
void doRevertToSaved() {
IDocumentProvider p= getDocumentProvider();
if (p == null)
   return;
performRevert();
}
```

Shingles: {-1942396283, 1672190785,
-1214877515, -5673233372, 208215292,
1307570125, 1431157461, 577482186 }

The Approach of Dig et al.: Semantic Analysis

- Seven Detection Strategies
  - applied in order
  - based on dependencies between artifacts
    - *method calls*
    - *subclassing*
    - *fields*
    - *arguments*
    - *parameters*

1. RenamePackage (RP)
2. RenameClass (RC)
3. RenameMethod (RM)
4. PullUpMethod (PUM)
5. PushDownMethod (PDM)
6. MoveMethod (MM)
7. ChangeMethodSignature (CMS)

The Approach of Dig et al.: Results

- More than 85% Precision and Recall on
  - Eclipse
  - Struts
  - HotDraw
- What's next? CatchUp! (Automatically refactoring clients)

Automated Detection of Refactorings in Evolving Components
Dig et al., ECOOP 2006

Roadmap

- Introduction
- Mining the history for relationships
  - Logical coupling
- Recovering entity evolution
  - Origin analysis
  - Refactoring detection
- **Mining a history for rules**
  - Common error patterns
- And more...

Common Error Patterns

- Livshits & Zimmermann ’05
- Data mining reveals frequent patterns
  - Matching Method Pairs
  - State Machines

How to detect bugs in apps that use APIs about which you do not have knowledge?

Principles

1. API specific errors
2. Co-addition is a pattern
3. Small commits are fixes

File	Revision	Added method calls
`Foo.java`	1.12	`o1.addListener` `o1.removeListener`

`Bar.java`	1.47	`o2.addListener`
		`o2.removeListener`
		`System.out.println`

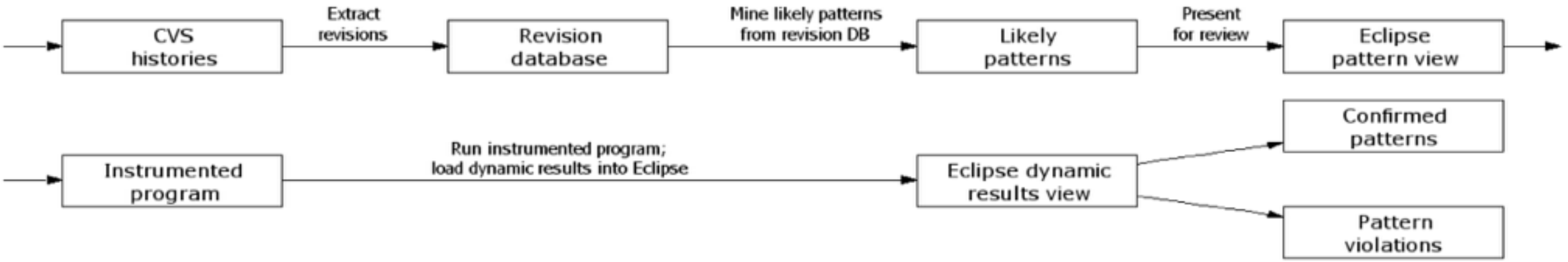
`Baz.java`	1.23	`o3.addListener`
		`o3.removeListener`
		`list.iterator`
		`iter.hasNext`
		`iter.next`

`Qux.java`	1.41	`o4.addListener`
	1.42	`o4.removeListener`

When to look for pattern violations?

- Compile Time
  - — complicated (inter procedural analysis)
- Runtime
  - \+ Scalability
  - \+ Simplicity (no interprocedural analysis)
  - \+ Counting occurrences
  - \+ Zero False Positives
  - — Coverage

Dynamine: The Approach



- Human Input is required
- Mines from the history
- Validates at runtime

Mining for Likely Patterns: The Apriori Algorithm

- Concepts
  - Usage Pattern
  - Transaction
  - Support Count
- Input
  - Minimum Support
- Output
  - Frequent Patterns
- Implementation
  - Iterative
  - Exponential

Pattern Filtering

- Consider a subset of the methods
  - ignore initial revisions
  - ignore common calls
- Consider small patterns only
  - group calls by access path

Pattern Ranking & Classification

- Lexicographically on support count
- Corrective ranking
  - assumes on one-line changes are bug-fixes
  - used as first lexicographic category improves bug finding
- Classification
  - Usage
  - Error
  - Unlikely

Results

	Method pair $\langle a, b \rangle$		Confidence			Support	Dynamic		Static		Type
	Method a	**Method** b	$conf$	$conf_{ab}$	$conf_{ba}$	$count$	v	e	v	e	
Corrective Ranking											
Eclipse (16 pairs)	`NewRgn`	`DisposeRgn`	0.76	0.92	0.82	49					
	`kEventControlActivate`	`kEventControlDeactivate`	0.69	0.83	0.83	5					
	`addDebugEventListener`	`removeDebugEventListener`	0.61	0.85	0.72	23	4	1	4	1	Unlikely
	`beginTask`	`done`	0.60	0.74	0.81	493	332	759	41	28	Unlikely
	`beginRule`	`endRule`	0.60	0.80	0.74	32	7	0	4	0	Usage
	`suspend`	`resume`	0.60	0.83	0.71	5					
	`NewPtr`	`DisposePtr`	0.57	0.82	0.70	23					
	`addListener`	`removeListener`	0.57	0.68	0.83	90	143	140	35	29	Error
	`register`	`deregister`	0.54	0.69	0.78	40	2,854	461	17	90	Error
	`malloc`	`free`	0.47	0.68	0.68	28					
	`addElementChangedListener`	`removeElementChangedListener`	0.42	0.73	0.57	8	6	1	1	1	Error
	`addResourceChangeListener`	`removeResourceChangeListener`	0.41	0.90	0.46	26	27	1	21	1	Usage
	`addPropertyChangeListener`	`removePropertyChangeListener`	0.40	0.54	0.73	140	1,864	309	54	31	Error
	`start`	`stop`	0.39	0.59	0.65	32	69	18	20	9	Error
	`addDocumentListener`	`removeDocumentListener`	0.36	0.64	0.56	29	38	2	14	2	Usage
	`addSyncSetChangedListener`	`removeSyncSetChangedListener`	0.34	0.62	0.56	24					
jEdit (8 pairs)	`addNotify`	`removeNotify`	0.60	0.77	0.77	17	3	0	3	0	Unlikely
	`setBackground`	`setForeground`	0.57	0.67	0.86	12	75	175	5	5	Unlikely
	`contentRemoved`	`contentInserted`	0.51	0.71	0.71	5	17	11	7	5	Error
	`setInitialDelay`	`start`	0.40	0.80	0.50	4	0	32	0	2	Unlikely
	`registerErrorSource`	`unregisterErrorSource`	0.28	0.45	0.62	5					
	`start`	`stop`	0.20	0.39	0.52	33	83	98	10	13	Error
	`addToolBar`	`removeToolBar`	0.18	0.60	0.30	6	24	43	5	5	Error
	`init`	`save`	0.09	0.40	0.24	31					
(24 pairs)	**Subtotals for the corrective ranking scheme:**						**5,546**	**2,051**	**241**	**222**	3 U, 8 E

DynaMine: Finding Common Error Patterns by Mining Software Revision Histories

Livshits and Zimmerman, FSE 2005

Roadmap

- Introduction
- Mining the history for relationships
  - Logical coupling
- Recovering entity evolution
  - Origin analysis
  - Refactoring detection
- Mining a history for rules
  - Common error patterns
- **And more**...

Associating Artifacts with Tasks

- Kersten & Murphy ‘05
- Mylar/Mylin
- Task-Focused Interface
- Degree of Interest ranking

How to filter the large amount of information available in the IDE?

Further Directions (Kagdi et al. ’07)

- Change-Based repositories
  - Replay (Hattori et. al ’11)
- Bug prediction
  - Extensive comparison of approaches (D’Ambros et al. ’10)
- Risk Prediction
  - The Code Orb, (Lopez ’11)

Enablers of Historical Analysis

- Versioning systems
- Availability of different types of data
  - developer interaction
  - bug/issue tracking
- Modern IDE's
  - plugin philosophy
    - *collecting data*
    - *playground for features*

Benefits of Historical Analysis

- Increase the amount of available information
- Allows temporal mining
- Predict various aspects of the system based on the past
- Allows empirical validation of hypotheses based on mining many systems

Roadmap

- Introduction
- Mining the history for relationships
  - Logical coupling
- Recovering entity evolution
  - Origin analysis
  - Refactoring detection
- Mining a history for rules
  - Common error patterns
- And more...

What you should know!

- What are some of the problems that can be solved by mining software repositories
- What is the problem of origin analysis and how to solve it
- How can logical coupling be detected, and why it matters
- What kind of error patterns can be detected by mining software repositories
- How to architect a system that can recommend entities that are likely to need to be changed

Can you answer these questions?

- How would you solve the problem of origin analysis?
- What are some of the challenges when in comes to automatically detecting refactorings?
- What are the advantages of taking into account dynamic analysis when mining API protocols?
- What heuristics would you use to predict classes that change together and why?
- Can you discuss some of the advantages and some of the disadvantages of the shingles technique?

References

- **Mandatory Reading (minimum one paper)**
  - **Detection of Logical Coupling Based on Product Release History,** Gall et al., ’98
  - **Automated Detection of Refactorings in Evolving Components,** Dig et al., ’06
  - **Mining Version Histories to Guide Software Changes,** Zimmermann et al. ‘04
  - **DynaMine: finding common error patterns by mining software revision histories,** Livshits & Zimmerman, ’05

- **Further Reading**
  - An Integrated Approach for Studying Architectural Evolution, Tu & Godfrey, ’02
  - Mylar, a Degree of Interest model for IDE’s, Kersten & Murphy ’05
  - The Role of Refactorings in API Evolution, Dig & Johnson, ’05
  - The code orb: supporting contextualized coding via at-a-glance views, Lopez ’11
  - Modeling History to Understand Software Evolution, Girba, ’05
  - An extensive comparison of bug prediction approaches, D’Ambros et al., ’10
  - Software Evolution Comprehension: Replay to Rescue, Hattori et al., ’11
  - A survey and taxonomy of approaches for mining software repositories in the context of software evolution, Kagdi et al. ’07

creative commons

COMMONS DEED

Attribution-ShareAlike 2.5

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/2.5/