Lecture 15

Refactoring Reconstruction

Today's Agenda

- Motivation for Refactoring Reconstruction
- Refactoring Reconstruction
  - UMLDiff: some slides borrowed from Zhenchang Xing (U. Alberta)

Today's Agenda

- Synthesis of Refactoring Reconstruction Techniques
- API Evolution Support
- Bug Cache (MSR Part II)

Motivation for Reconstructing Refactorings from Two Versions

Motivation for Reconstructing Refactorings from Two Versions

Motivation for Reconstructing Refactorings from Two Versions

1. Detecting Possible Sources of Errors
   - Incomplete refactorings can be sources of errors
   - e.g. BarChart.draw() and PieChart.draw() override Chart.draw()
   - e.g. Chart.draw() and PieChart.draw() were renamed to Chart.paint() and PieChart.paint() but not BarChart.draw().

Motivation for Reconstructing Refactorings from Two Versions

2. Capturing Intent of Changes
    - Better empirical studies of code changes
    - Reduce # of conflicts in version merging

Motivation for Reconstructing Refactorings from Two Versions

3. Capturing and Replaying Changes
   - Automated update of client code: e.g. if a parameter was added ton an API, then method invocations in program code using the API is automatically adapted.
4. Longer, continuous evolution history
   - eRose system: when identifying related changes, inferred renamings can be used to combine rules of the previous instance and rules of the new instance

Motivation for Reconstructing Refactorings from Two Versions

5. Relation to Software Metrics
   - Assess what kinds of refactorings increase what kinds of quality metrics

[Source: Identifying Refactorings from Source-Code Changes, Peter Weissgerber and Stephan Diehl ASE 2006]

Design Evolution Analysis in support of Evolutionary Software Development

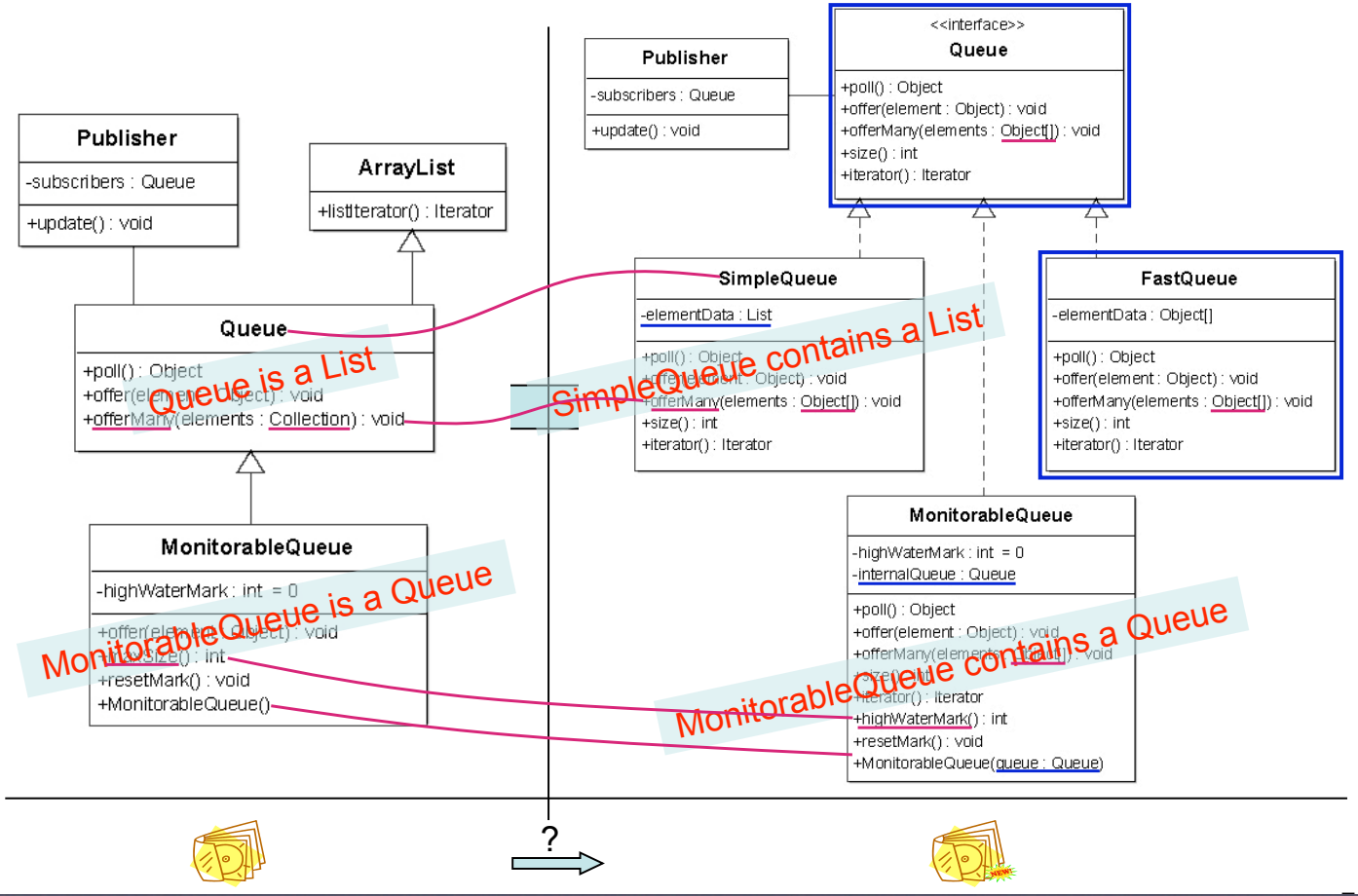
Zhenchang Xing
University of Alberta

Supported by

Why is He Unhappy?

```
public class Course {
    private MonitorableQueue waitingList = new MonitorableQueue();

    private Queue backupTAs = new Queue();

    public void addToWaitingList(Collection waitingStudents) {
        waitingList.offerMany(waitingStudents);
    }

    public void enrolFromWaitingList(int howmany) {
        List list = this.waitingList;
    }

    public void notifyBackupTAs() {
        for(Iterator iterator = backupTAs.listIterator(); iterator.hasNext();) {
        }
    }

    public void reportEnrolmentStatistics() {
        int historyHigh = waitingList.maxSize();
    }
}
```

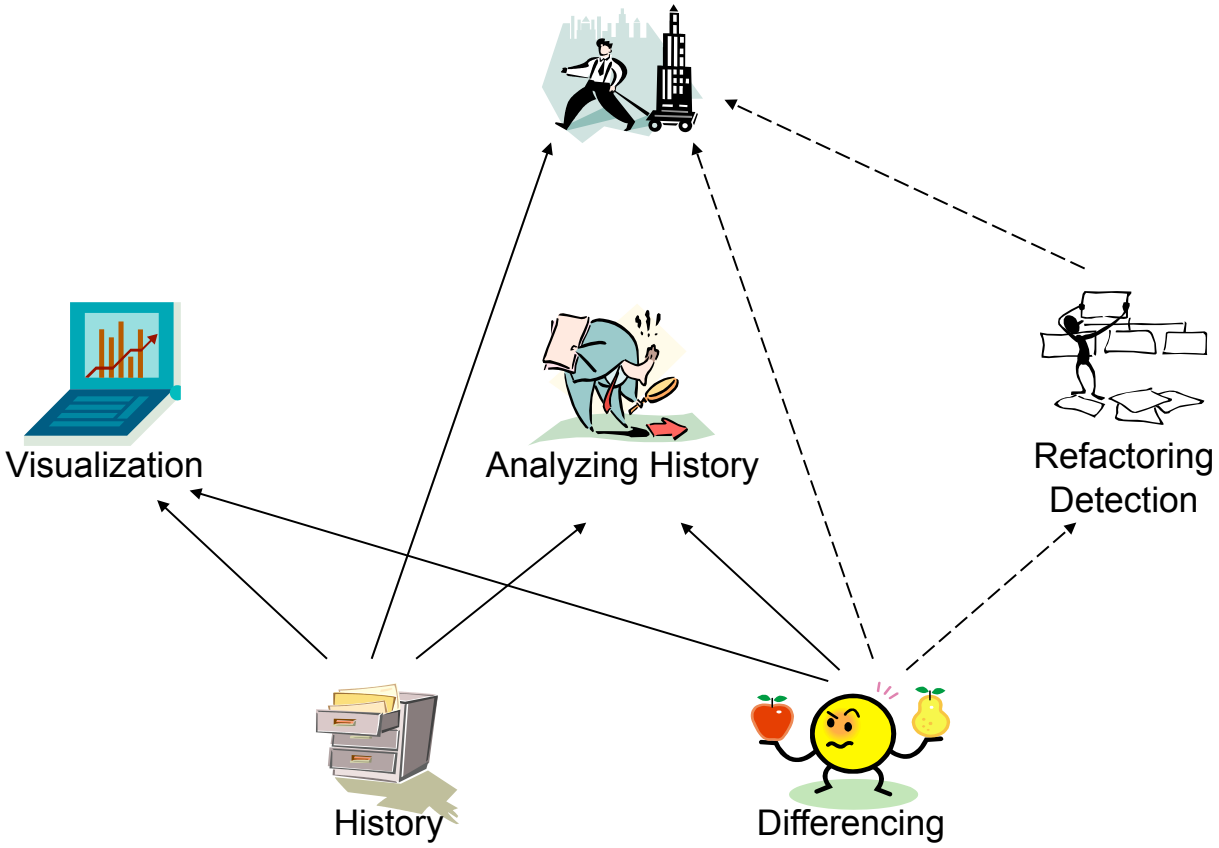
What I Will Tell Him



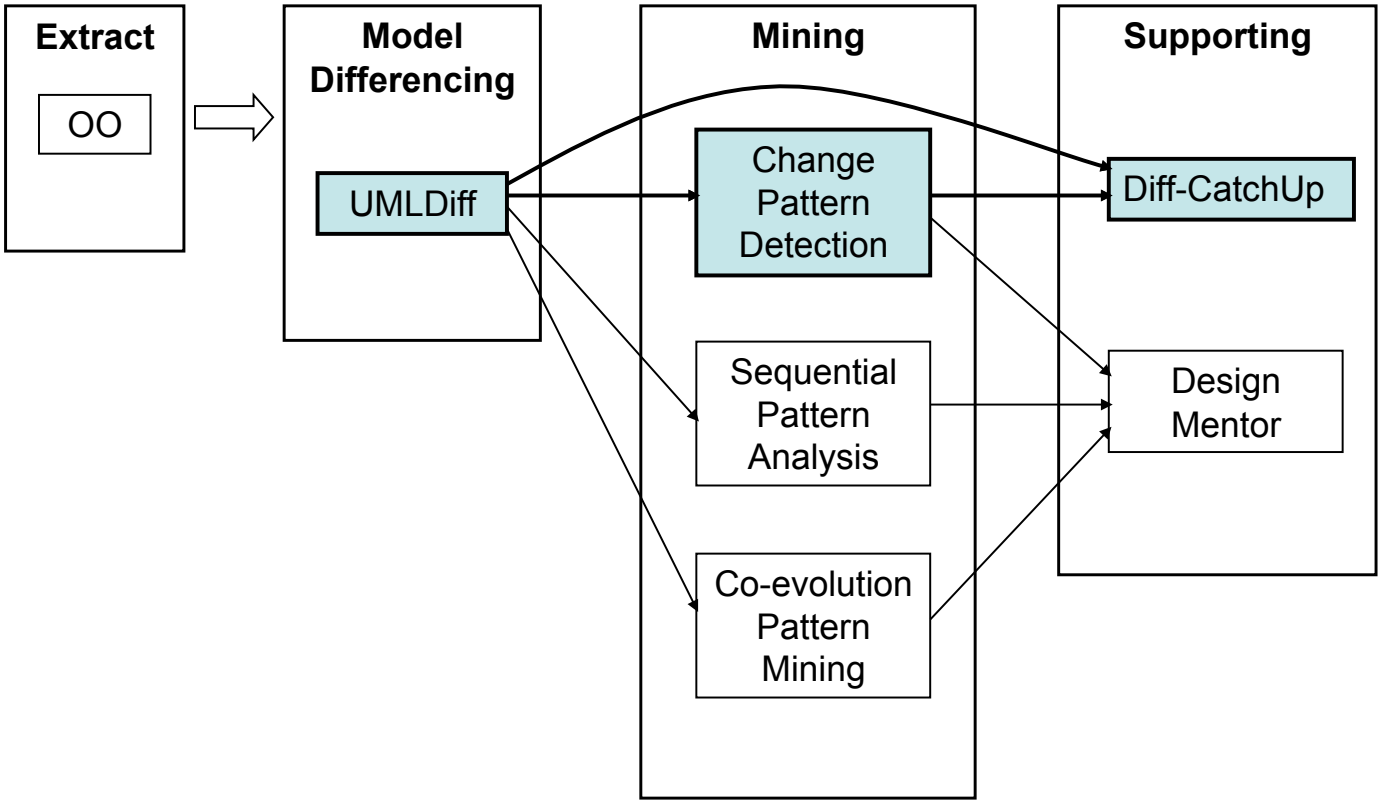
The Research Questions

- What exactly has been changed in the design context and how?
- Why has it been changed in the way it has?
- How can this information be used to support developers and in what tasks?

The World


Visualization
Analyzing History
Refactoring Detection
History
Differencing

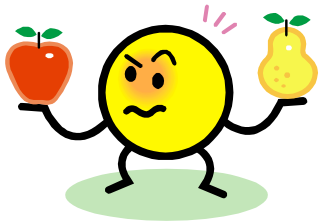
The Methodology



Model differencing with UMLDiff

Journal of Automated Software Engineering, 2007
The 20th ACM/IEEE International Conference on Automated Software Engineering, 2005

What exactly has been changed and How?



Heuristics in UMLDiff

- Additions and removals are easy
- Renamings are difficult
  - Lexical similarity of names and comments:
    - LCS, Adjacent pair
  - Structural similarity of relations
- Moves are even harder
  - The context from and to which elements are moved
    - Relationships: inheritance, containment, usage
    - Lexical and structural similarity of source and target contexts
  - The number of potential moves
- What if a set of elements are all renamed and/or moved?
  - Multiple rounds of renaming/move recognition

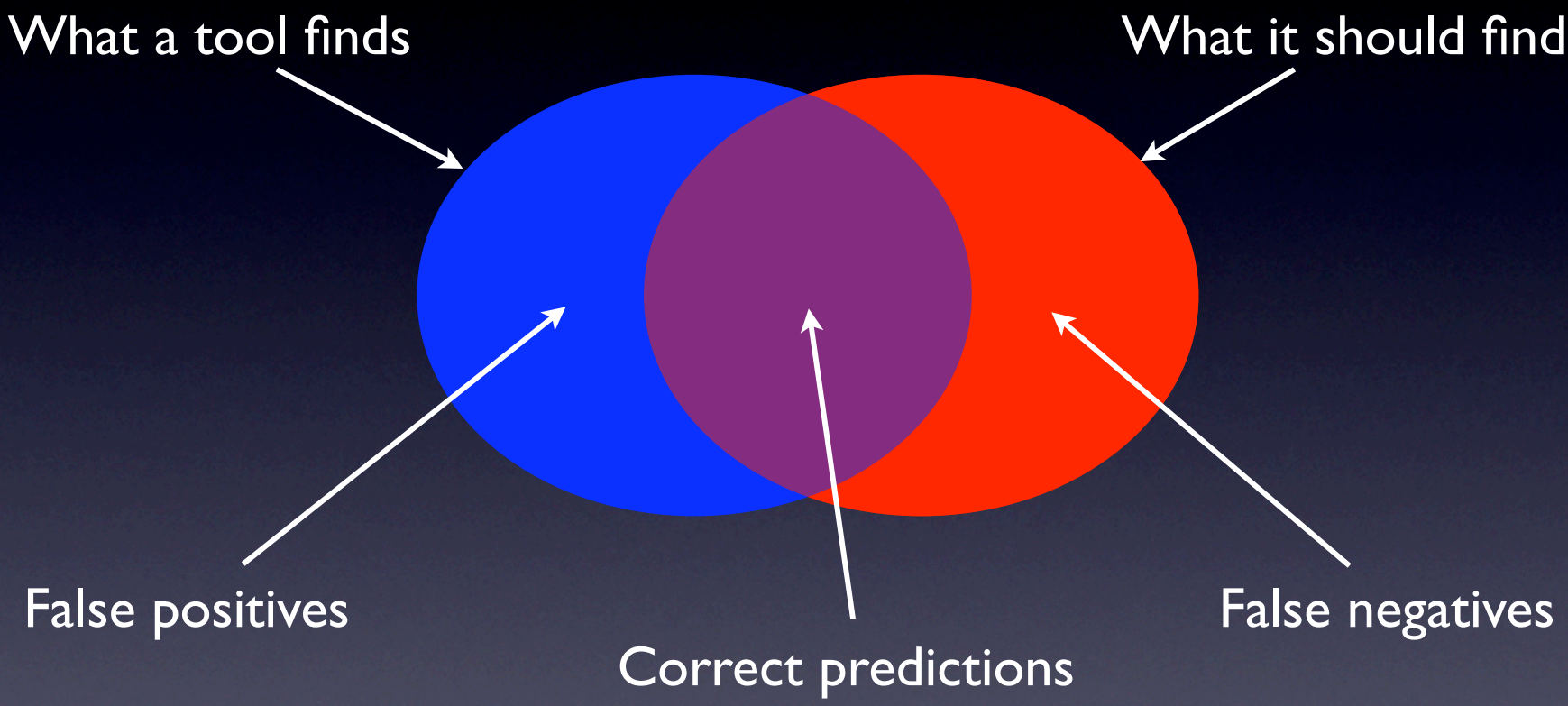
UMLDiff Process

- **Input**: $Model_{before}$ and $Model_{after}$
- **UMLDiff** is a heuristic differencing algorithm
  1. Mapping model elements
     - Lexical and structural similarity
  2. Mapping relationships
     - The same relation type and the model elements they relate are mapped
  3. Recognizing extract/inline operations (not limited to class internals)
     - Usage dependency changes
  4. Compare attributes of mapped model elements
- **Output**: A set of elementary design change facts
  - Additions, removals, matches, renamings, moves of model elements
  - Extract and inline operations
  - Changes to relationships (inheritance, association, usage)
  - Changes to attributes (visibility, deprecation-status, …)

Evaluation

- How did they create the ground truth?
  - Use a very low threshold 1% and manually inspect all of them
  - Changes identified by UMLDiff and the ones UMLDiff missed, which were manually added through their manual inspection using JDEvAn tool
- Precision
- Recall

Precision vs. Recall



precision = % of returned entities are relevant

recall = what % of relevant entities are returned

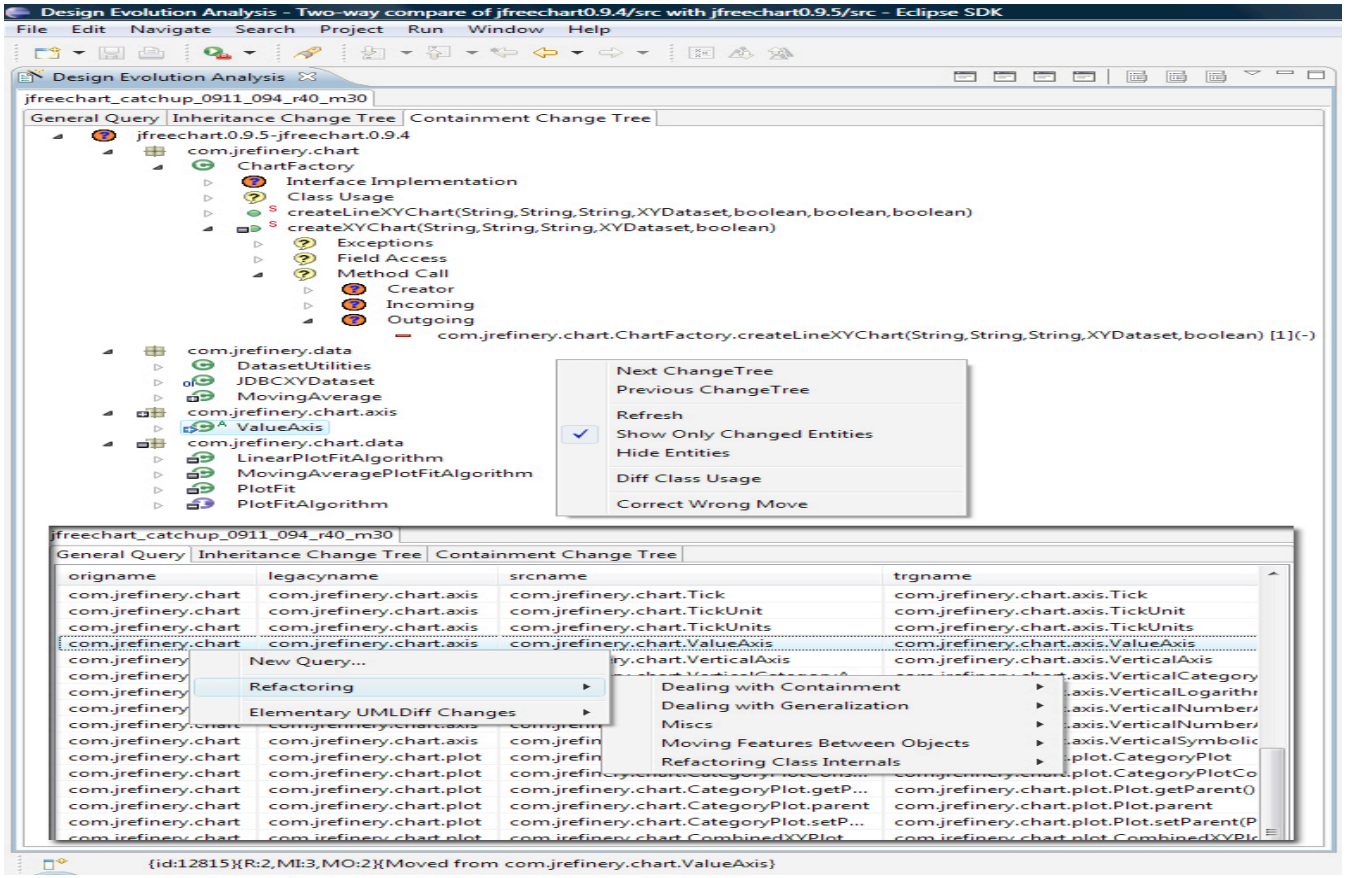
How good is UMLDiff?

	HtmlUnit	JFreeChart	Eclipse JDT
Type	Unit testing framework for web apps	Java library for drawing charts	IDE and Plugin-based framework
Major releases	11 (~4 years)	31 (~5 years)	6 (~3 years)
Average #Class	~200	~450	~4000
Renamings* [Threshold 0.3] (Precision) (Recall)	97.2% 98.5%	95.2% 96.4%	93.8% 96.6%
Moves* [Threshold 0.4] (Precision) (Recall)	99.5% 99.9%	91.1% 97.1%	84.8% 90.3%

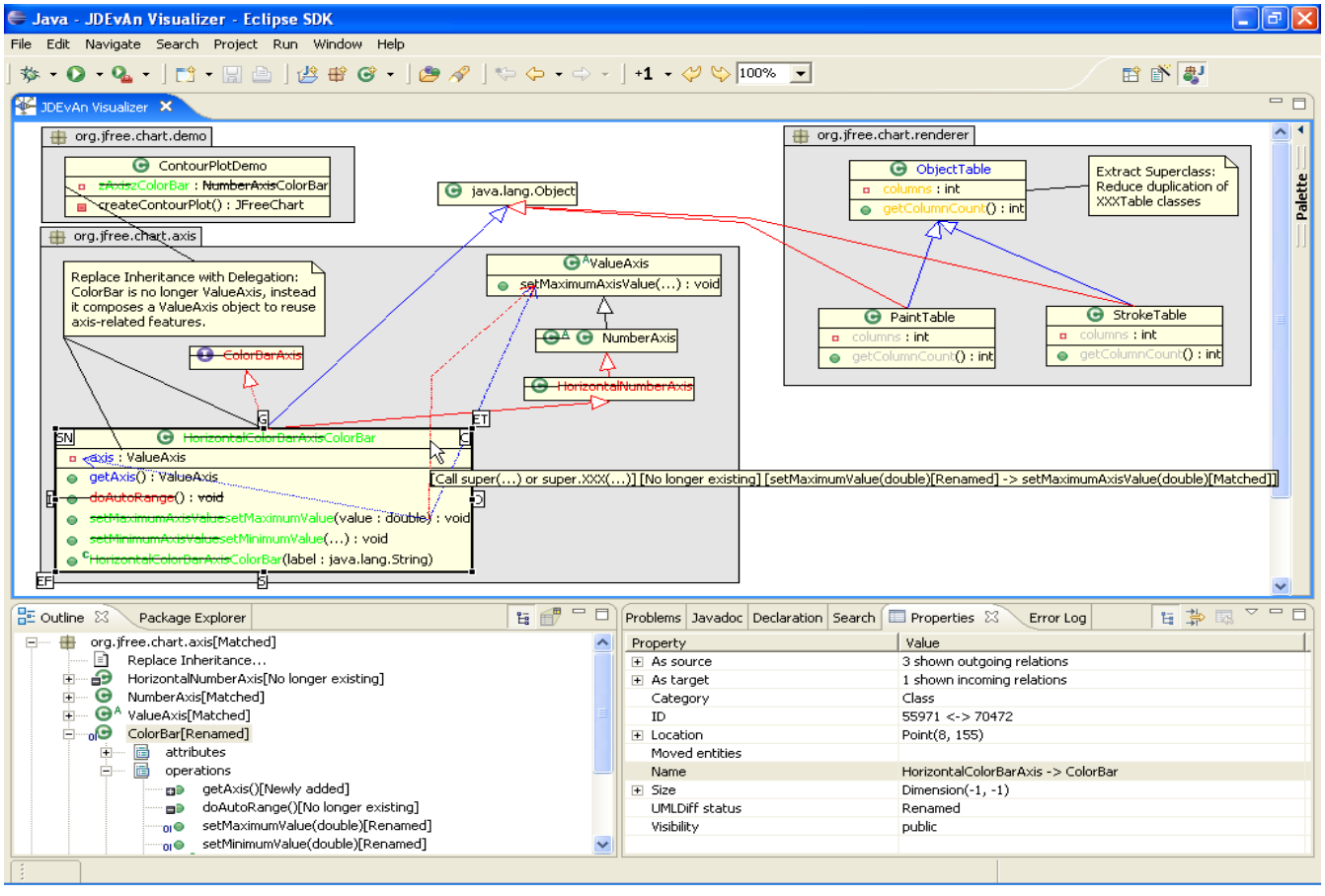
*Results with heuristics: Name, Comment, Structure, Src/ TrgContext, #PotentialMoves, TransitiveUsage, Round=3

Evaluation

JDEvAn in Eclipse



JDEvAn Viewer in Eclipse



Synthesis of Refactoring Reconstruction Techniques

Method	Program Element Characteristics	Versions
Origin Analysis 2005	name similarity, code metrics, calls	two complete versions selected manually
UMLDiff 2005	name similarity, code relationships	two complete versions selected manually
M. Kim et al. 2007	name similarity	two complete versions selected manually
S. Kim et al. 2005	name similarity, code metrics, calls, textual similarity	two complete versions selected manually
Dig et al. 2006	syntactical similarity, code relationships	two complete versions selected manually
Weissgerber et al. 2006	structural and code clone differences	all change sets between two versions
SemDiff 2008	structural and outgoing call differences	all change sets between any versions

[Source: Recommending Adaptive Changes from Framework Evolution, Barthelemy Dagenais and Martin Robillard, ICSE 2008]

API-Evolution Support with Diff-CatchUp

IEEE Transactions on Software Engineering, 2007

How can this information be used to support developers and in what tasks?

Diff-CatchUp Approach

- Automatically recover the evolution of framework APIs
  - UMLDiff and change-pattern queries
- Suggest ways to migrate client applications
  - Refactored API
    - Present the refactorings that the API is involved in and its renaming/move counterparts in new version if any
  - Removed (deprecated, visibility-restricted, no-longer-inherited, and class-made-abstract) API
    - Locate "voluntary" migration examples
    - Recommend replacing APIs

Migrate to Refactored API

- RenameMethod(maxSize(), highWaterMark())

  Prob #1: The method maxSize() is undefined for the type MonitorableQueue
  Reason: The method name changed
  Solution: Update the method call with new name

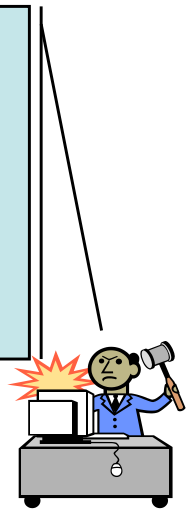
- ChangeParamType(offerMany(…), Collection, Object[])

  Prob #2: The method offerMany(Object[]) in the type MonitorableQueue is not applicable for the argument (Collection)
  Reason: Parameter type changed
  Solution: Obtain Object[] from Collection (e.g. Collection.toArray())

Migrate to Refactored API

- **RenameClass**(Queue, SimpleQueue)
- **ExtractInterface**(SimpleQueue, Queue)
- **AddAbstraction**(FastQueue, Queue)
- **AddAbstraction**(MonitorableQueue, Queue)

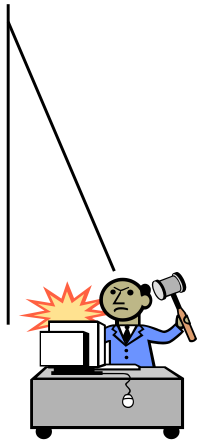
Prob #3: Cannot instantiate the type Queue
Reason: The Queue represents a newly introduced interface in the new version.
The original class Queue is renamed as SimpleQueue.
Solution: Create SimpleQueue object, or
See if the interface Queue's other implementation classes can be used as well.



Migrate to Refactored API

- **ReplaceInheritanceWithDelegation**(MonitorableQueue, SimpleQueue, internalQueue, Queue)
- **ReplaceInheritanceWithDelegetion**(SimpleQueue, ArrayList, elementData, List)
- **ExtractInterface**(SimpleQueue, Queue)
- **AddAbstraction**(MonitorableQueue, Queue)

Prob #4: Type mismatch: cannot convert MonitorableQueue to List
Reason: MonitorableQueue is no longer SimpleQueue, which is no longer List
Solution: Stop using MonitorableQueue as List object
May use it as a Queue object



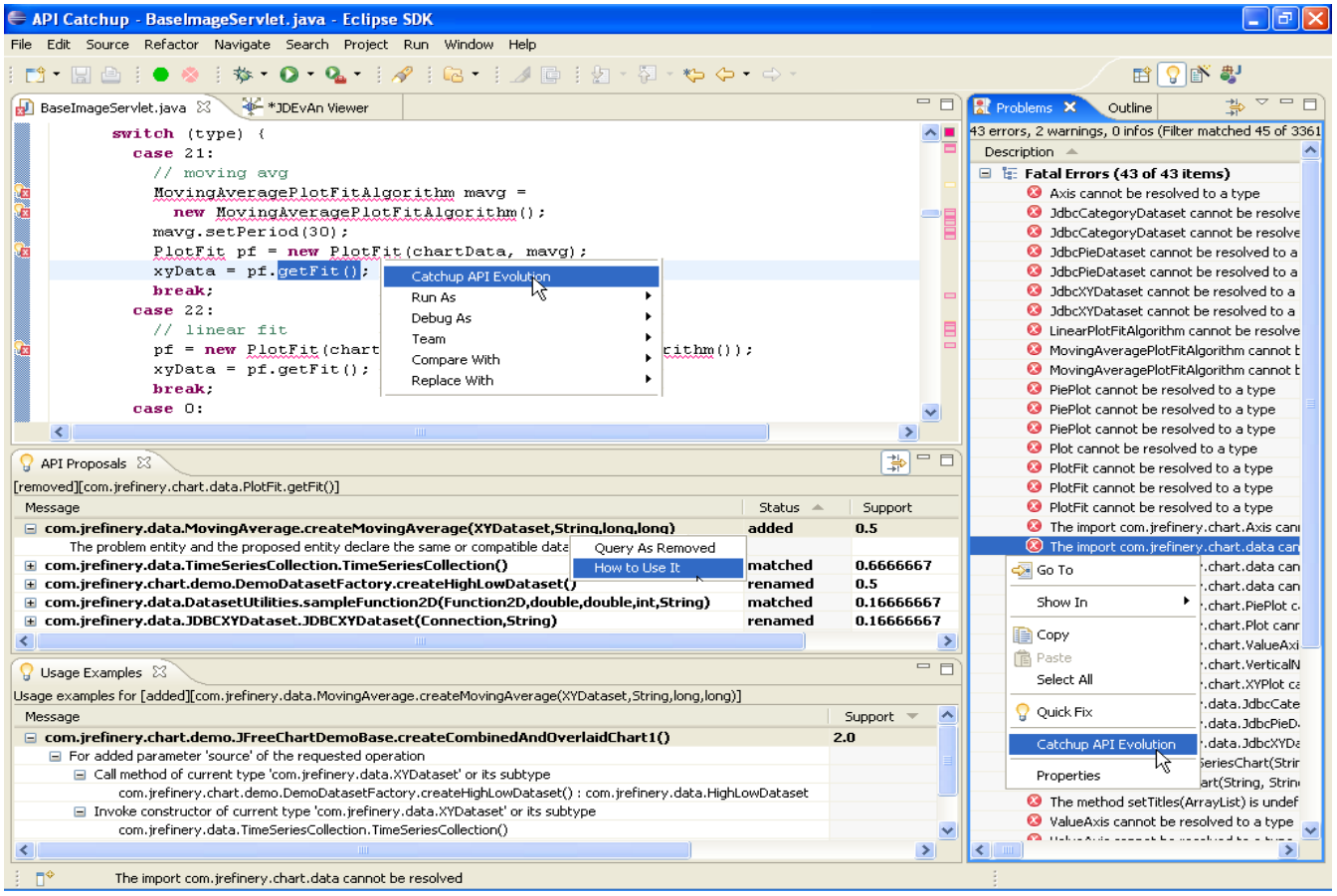
Migrate to "Removed" API

- **ReplaceInheritanceWithDelegation**(SimpleQueue, ArrayList, elementData, List)

Prob #6: The method listIterator() is undefined for the type Queue
Reason: The original Queue class used to be a List; it inherits listIterator() from its superclass ArrayList, but no longer doing so.

This is essentially a "**removed**" API.
How am I going to **replace** it?

Diff-CatchUp in Eclipse



How Good is Diff-CatchUp?

Type of problem	#broken API	#success proposal	%
	JFreechart		
ImportNotFound	17	17	100
UndefinedType+ImportNotFound+UndefinedName	254	247	97.2
InvalidClassInstantiation	1	1	100
UndefinedMethod/Constructor	180	151	83.9
ParameterMismatch	54	54	98.1
UndefinedField+UndefinedName	33	29	87.9
UsingDeprecatedType	3	3	100
UsingDeprecatedMethod/Constructor	35	34	97.1
Total	**577**	**535**	**92.7**
	HTMLUnit		
UndefinedType	1	1	100
UndefinedMethod/Constructor	11	9	81.8
ParameterMismatch	3	3	100
UsingDeprecatedType	1	0	0
Evaluation ecatedMethod/Constructor	10	7	70
Total	**26**	**20**	**76.9**

My thought on Refactoring Reconstruction Research

- Promising ways to allow programmers to understand code changes at a high level
- Still long ways to go to automatically reconstruct design intent from source code
- It can be applied to mining software repository research.
- This is a challenging problem:
  - heuristics-based, often requiring many similarity thresholds
  - hard to evaluate this type of work in general.

Preview for Monday after Spring Break

- First of all--- have a fun & productive spring break!
- Crosscutting Concerns
  - Why some code changes are crosscutting?
  - Read Visitor Pattern from Design Patterns book--- We may have a quiz on crosscutting concerns (using the visitor pattern code example) on Monday.