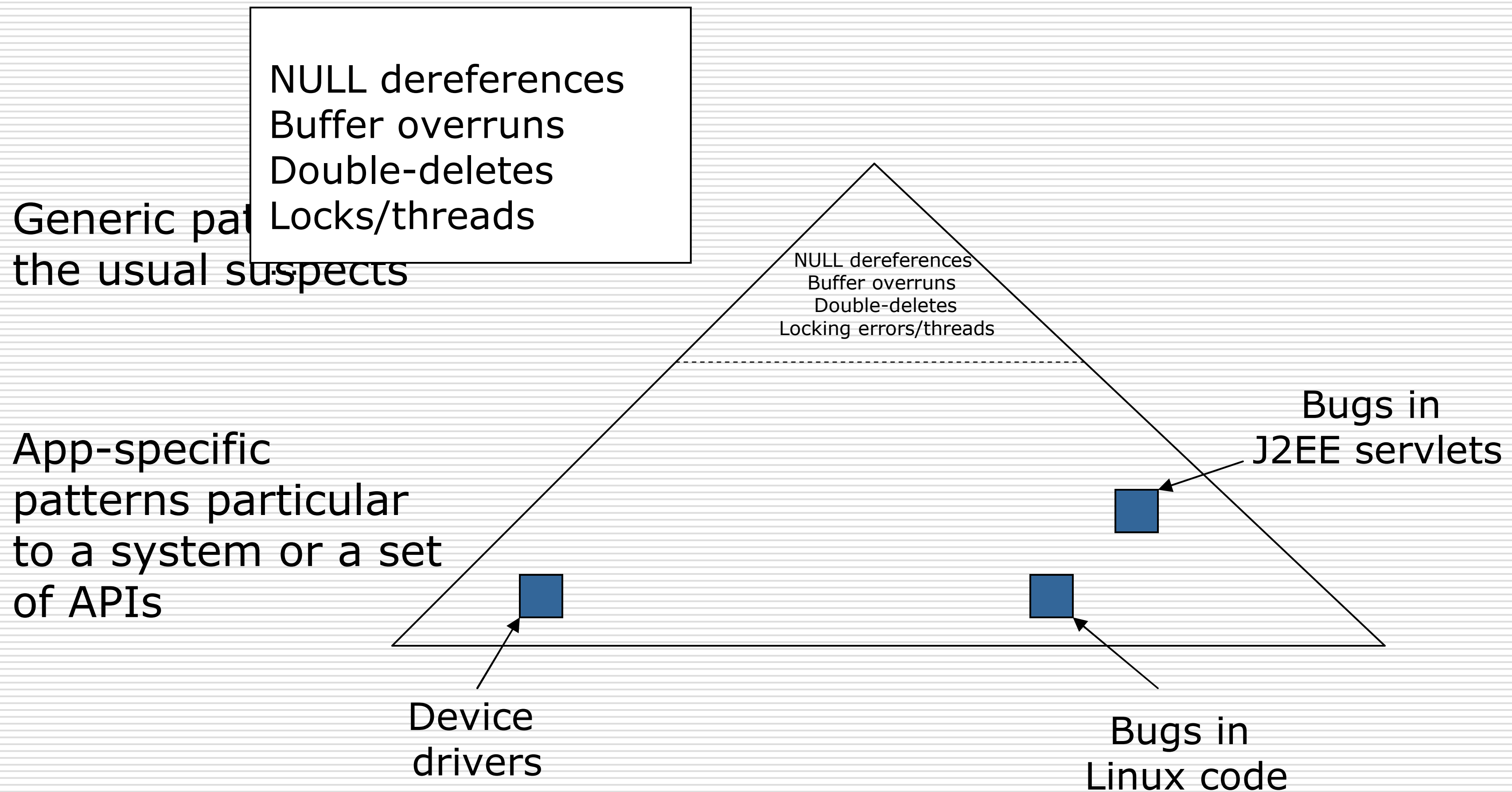
Locating Matching Method Calls by Mining Revision History Data

Benjamin Livshits
Stanford University
and
Thomas Zimmermann
Saarland University

The Usual Suspects

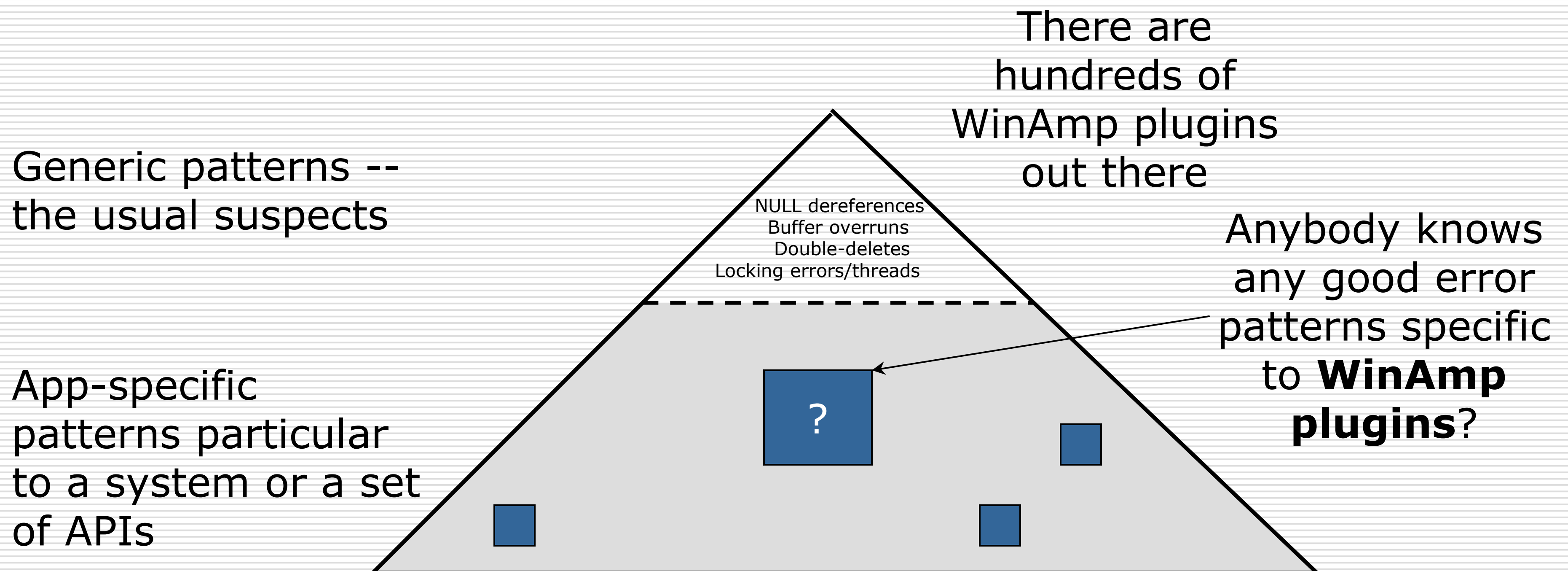
- Much bug-detection research in recent years
- Focus: generic patterns, sometimes language-specific
  - NULL dereferences
  - Security
    - Buffer overruns
    - Format string violations
  - Memory
    - Double-deletes
    - Memory leaks
  - Locking errors/threads
    - Deadlock/races/atomicity
- Let’s look at the space of error patterns in more detail

Classification of Error Patterns



Error Pattern Iceberg

Classification of Error Patterns



- Intuition:
  - **Many** other application-specific patterns exist
  - Much of application-specific stuff remains a gray area so far
- **Goal: Let's figure out what the patterns are**

Focus: Matching Method Pairs

- Start small:
  - Matching method pairs
  - Only two methods
  - A very simple state machine
  - Calls have match perfectly
- Very common, our inspiration is
  - System calls
    - fopen/fclose
    - lock/unlock
    - …
  - GUI operations
    - addNotify/removeNotify
    - addListener/removeListener
    - createWidget/destroyWidget
    - …
- Want to find **more of the same**

Our Insight

- Our problem:
  - Want to find matching method pairs that are error patterns
- Our technique:
  - Look at revision histories (think CVS)
- Crucial observation:

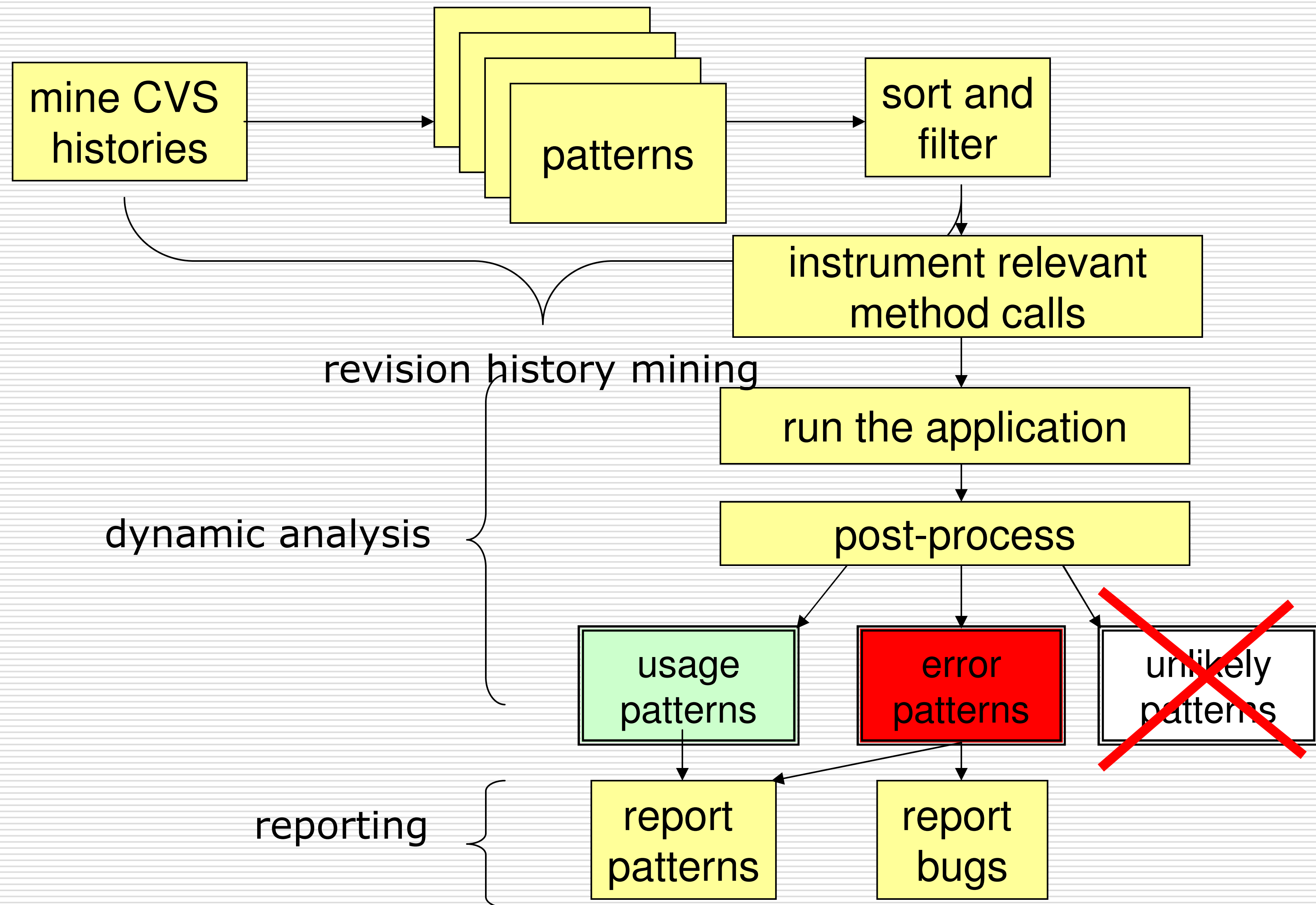
Things that are frequently checked in together often form a pattern

- Use data mining techniques to find method that are often added at the same time

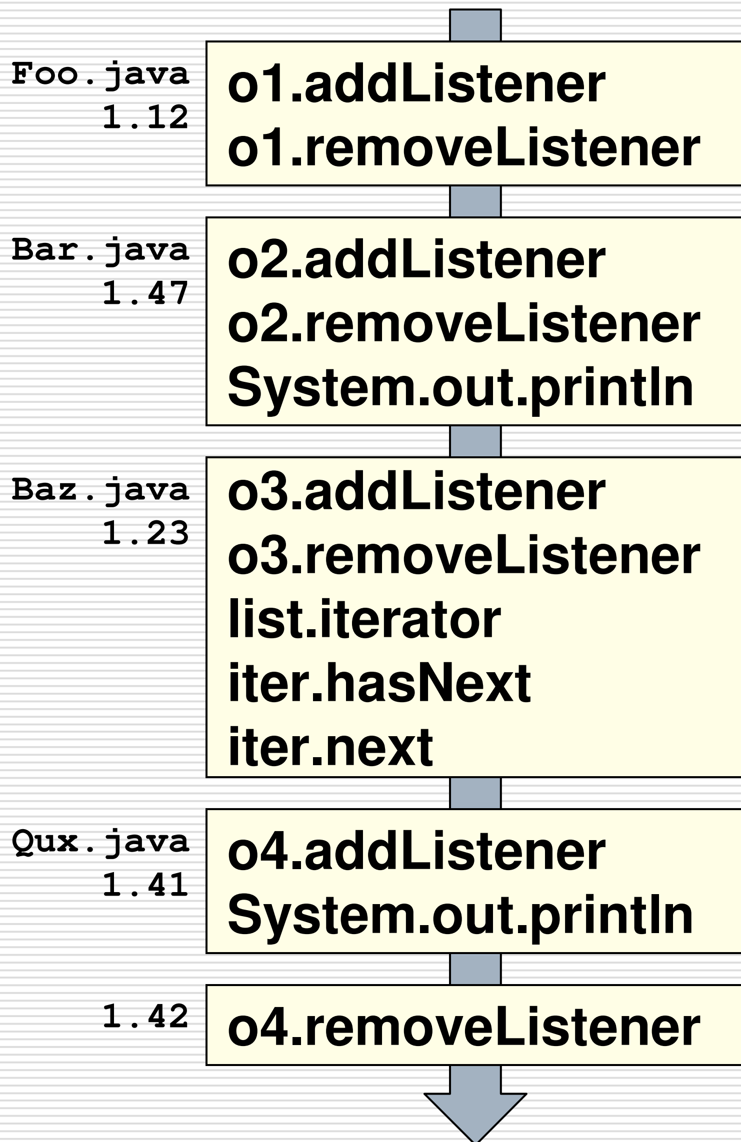
Our Insight (continued)

- Now we know the potential patterns
- "Profile" the patterns
  - Run the application
  - See how many times each pattern
    - hits – number of times a pattern is followed
    - misses – number of times a pattern is violated
- Based on this statistics, classify the patterns
  - **Usage** patterns – almost always hold
  - **Error** patterns – violated a large number of the times, but still hold most of the time
  - **Unlikely** patterns – not validated enough times

System Architecture

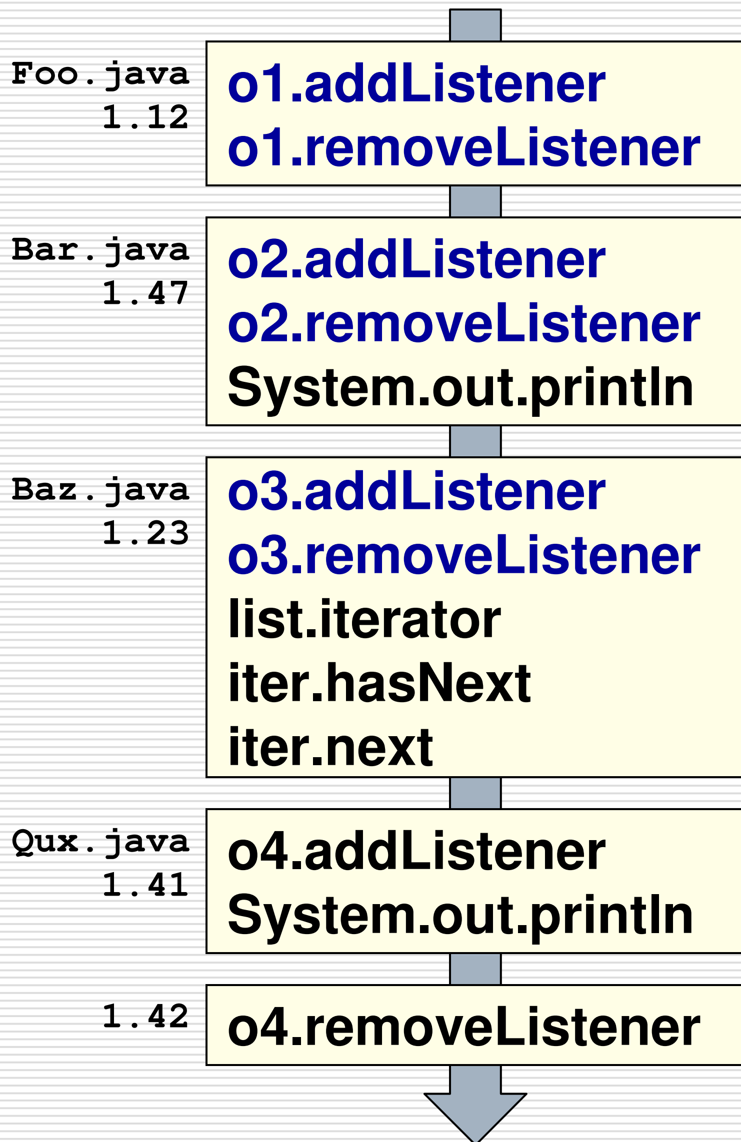


Mining Basics



- Sequence of revisions
- Files Foo.java, Bar.java, Baz.java, Qux.java
- Simplification: look at method calls only
- Look for **interesting patterns** in the way methods are called

Mining Matching Method Calls



- Use our observation:
  - Methods that are frequently added simultaneously often represent a use pattern
- For instance:

```
…
addListener(…);
…
removeListener(…);
…
```

Data Mining Summary

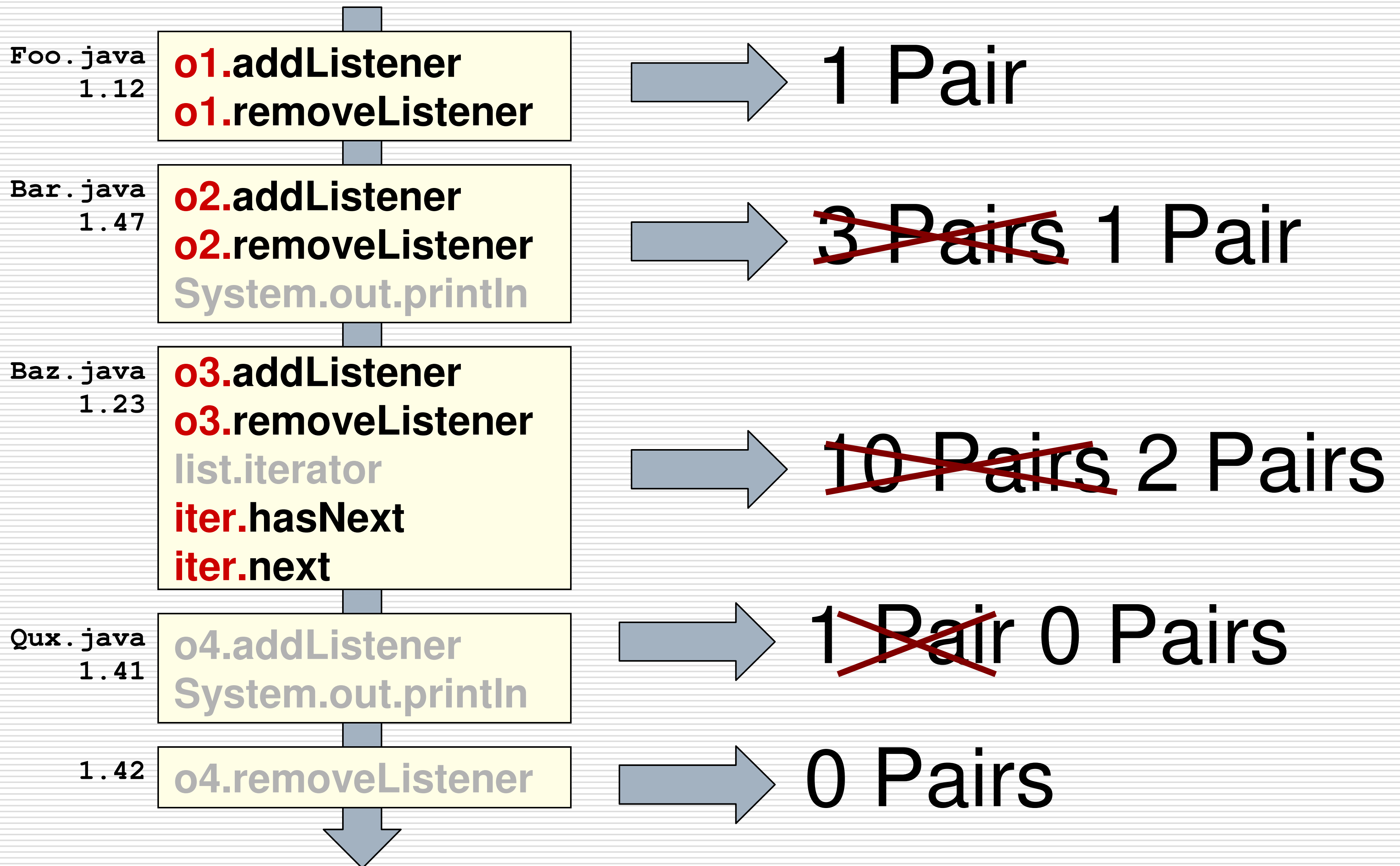
- We consider method calls added in each check-in
  - We want to find patterns of method calls
- Too many potential pairs to consider
  - Want to **filter** and **rank** them
  - Use **support** and **confidence** for that
- Support and confidence of each pattern
  - Standard metrics used in data mining
  - **Support** reflects how many times each pair appears
  - **Confidence** reflects how strongly a particular pair is correlated
- Refer to the paper for details

Improvements Over the Traditional Approach

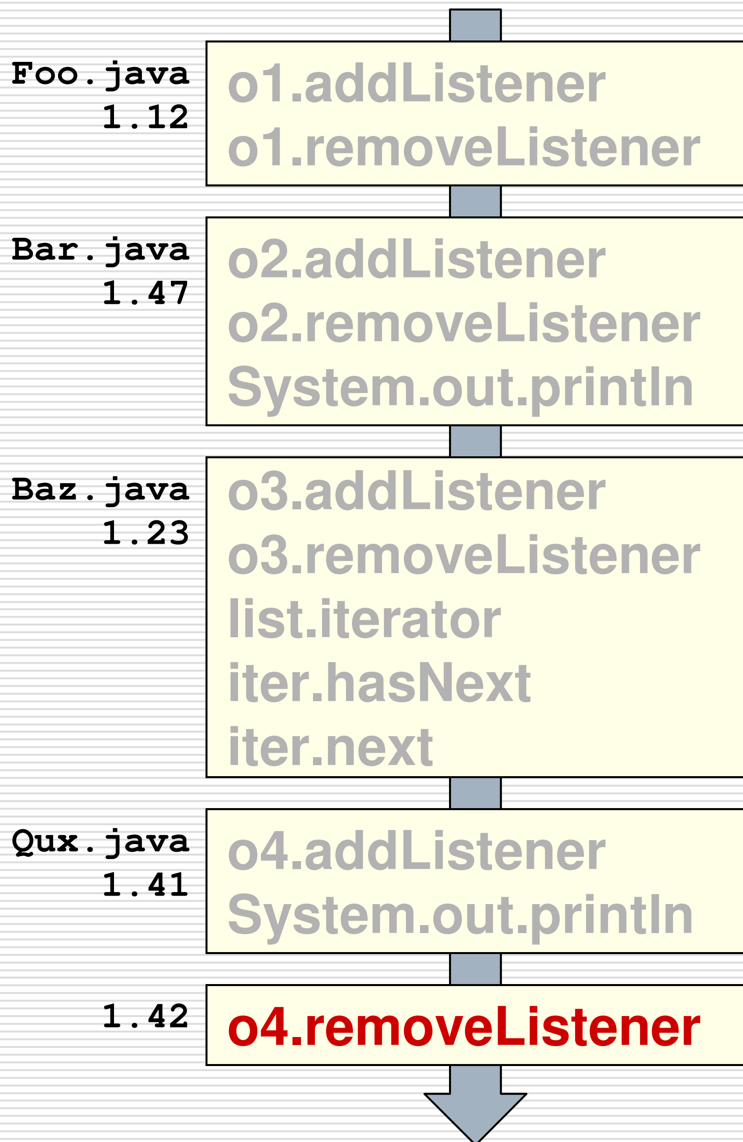
- The default data mining approach doesn't really work
  - Filters based on confidence and support
  - Still too many potential patterns!

1. Filtering:
   - Consider only pairs with the **same initial subsequence** as potential patterns
2. Ranking:
   - Use one-line "fixes" to find likely error patterns

Matching Initial Call Sequences



Using Fixes to Rank Patterns



- Look for one-call additions which likely indicate fixes.
- Rank pairs for such methods higher.

This is a fix!
Move pairs containing removeListener up

Mining Setup

- Apply these ideas to the revision history of Eclipse
  - Very large open-source project
  - Many people working on it
- Perform filtering based on
  - Pattern support
  - Pattern strength
- Get **32** strongly correlated method pairs in Eclipse

Some Interesting Method Pairs (1)

kEventControlActivate	kEventControlDeactivate
addDebugEventListener	removeDebugEventListener
beginRule	endRule
suspend	resume
NewPtr	DisposePtr
addListener	removeListener
register	deregister
addElementChangedListener	removeElementChangedListener
addResourceChangeListener	removeResourceChangeListener
addPropertyChangeListener	removePropertyChangeListener
createPropertyList	reapPropertyList
preReplaceChild	postReplaceChild
addWidget	removeWidget
stopMeasuring	commitMeasurements
blockSignal	unblockSignal
HLock	HUnlock
OpenEvent	fireOpen
...	

Some Interesting Method Pairs (2)

kEventControlActivate	kEventControlDeactivate
addDebugEventListener	removeDebugEventListener
beginRule	**endRule**
suspend	resume
NewPtr	
addListener	
register	
addElementChangedListener	
addResourceChangeListener	removeResourceChangeListener
addPropertyChangeListener	removePropertyChangeListener
createPropertyList	reapPropertyList
preReplaceChild	postReplaceChild
addWidget	removeWidget
stopMeasuring	commitMeasurements
blockSignal	unblockSignal
HLock	HUnlock
OpenEvent	fireOpen
...	

Begin applying a thread scheduling rule to a Java thread

Some Interesting Method Pairs (3)

kEventControlActivate	kEventControlDeactivate
addDebugEventListener	removeDebugEventListener
beginRule	endRule
suspend	resume
NewPtr	DisposePtr
addListener	removeListener
register	deregister
addElementChangedListener	removeElementChangedListener
addResourceChangeListener	removeResourceChangeListener
addPropertyChangeListener	removePropertyChangeListener
createPropertyList	reapPropertyList
preReplaceChild	postReplaceChild
addWidget	**removeWidget**
stopMeasuring	commitMeasurements
blockSignal	
HLock	
OpenEvent	
...	

Register/unregister the current widget with the parent **display** object for subsequent event forwarding

Some Interesting Method Pairs (4)

kEventControlActivate	kEventControlDeactivate
addDebugEventListener	removeDebugEventListener
beginRule	endRule
suspend	resume
NewPtr	DisposePtr
addListener	removeListener
register	deregister
addElementChangedListener	removeElementChangedListener
addResourceChangeListener	removeResourceChangeListener
addPropertyChangeListener	**removePropertyChangeListener**
createPropertyList	reapPropertyList
preReplaceChild	
addWidget	
stopMeasuring	
blockSignal	
HLock	HUnlock
OpenEvent	fireOpen
...	

Add/remove listener for a particular kind of GUI events

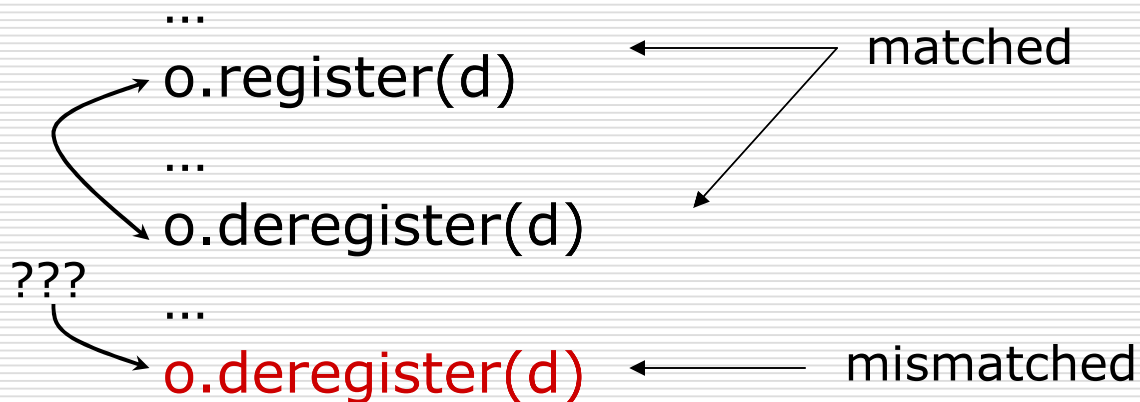
Some Interesting Method Pairs

kEventControlActivate	kEventControlDeactivate
addDebugEventListener	removeDebugEventListener
beginRule	endRule
suspend	resume
NewPtr	DisposePtr
addListener	removeListener
register	deregister
addElementChangedListener	removeElementChangedListener
addResourceChangeListener	removeResourceChangeListener
addPropertyChangeLis	
createPropertyList	
preReplaceChild	
addWidget	
stopMeasuring	commitMeasurements
blockSignal	unblockSignal
HLock	**HUnlock**
OpenEvent	fireOpen
...	

Use OS native locking mechanism for resources such as icons, etc.

Dynamically Check the Patterns

- Home-grown bytecode instrumenter
  - Get a list of matching method pairs
  - Instrument calls to any of the methods to dump parameters
- Post-processing of the output
  - Process a stream of events
  - Find and count matches and mismatches



Experimental Setup

- Applied our approach to Eclipse
  - One of the biggest Java applications
  - 2,900,000 lines of Java code
  - Included many Eclipse plugins consisting of lower quality code than the core
  - Chose 32 matching method pairs
- Times:
  - 5 days to fetch and process CVS histories
  - 30 minutes to compute the patterns
  - An hour to instrument Eclipse
  - And we are done!

Experimental Summary

- Pattern classification:
  - 5 are usage patterns
  - 5 are error patterns
  - 5 are unlikely patterns
  - *17 were not hit at runtime*
- Error patterns
  - Resulted in a total of **107** dynamically confirmed bugs
  - Results for a single run of Eclipse

A Preview of Coming Attractions…

- We have a paper in FSE 2005
- Describes a tool called DynaMine
- Provides various extensions:
  - More complex patterns:
    - State machines
    - Grammars
  - More applications analyzed
  - More bugs found