Understanding and Aiding Code Evolution by Inferring Change Patterns

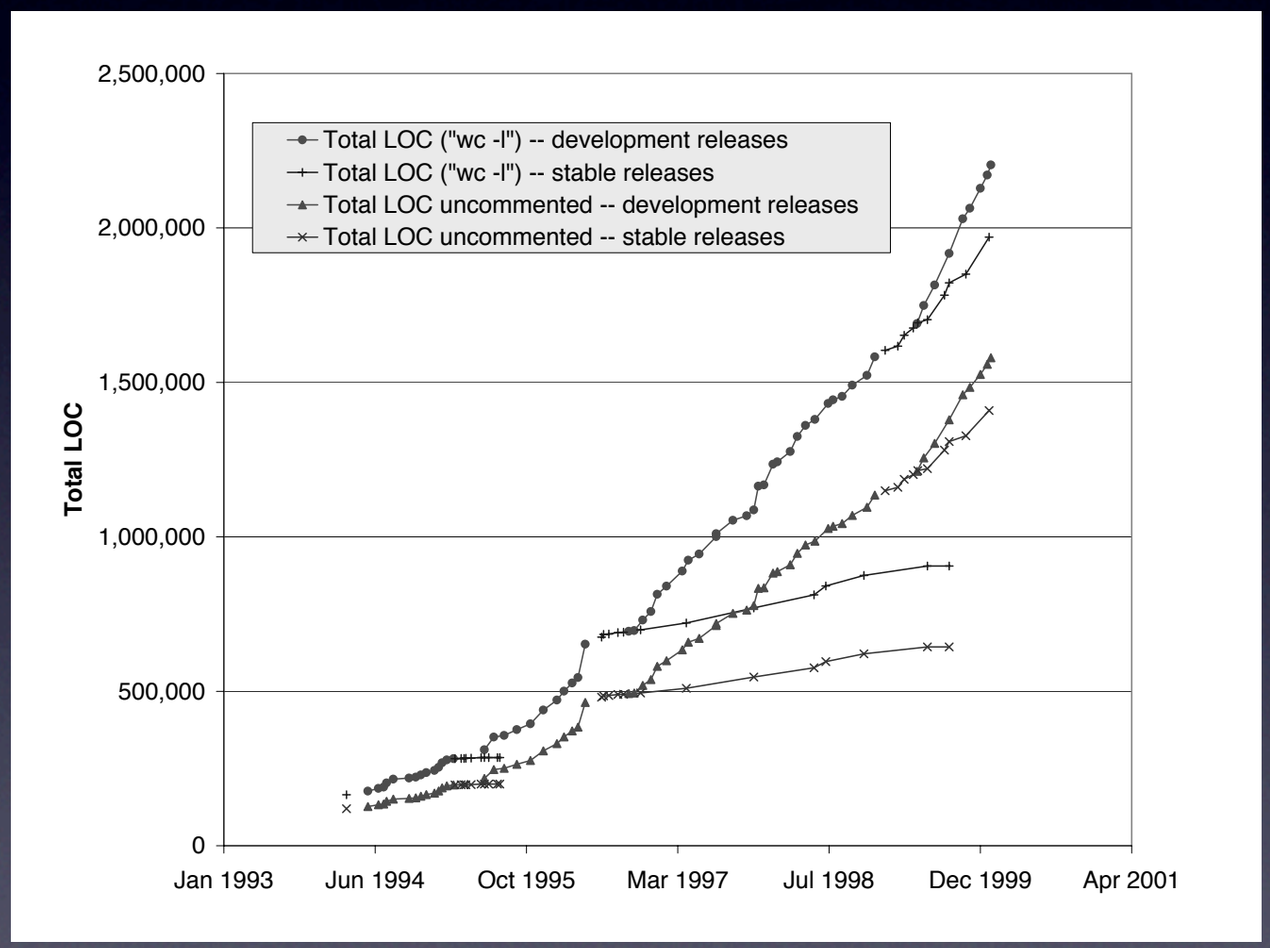
Miryung Kim

Doctoral Symposium
ICSE 2007

1. Title: Understand & Aiding Code Evolution by Inferring Change Patterns. (30 sec)

My name is Miryung Kim & I am a graduate student advised by Dr. David Notkin at the University of Washington. I am going to give a short introduction of what my proposed ph.d thesis will be about.

Classic Studies of Software Evolution



2. Problem: Change pattern is not a first class entity (30 sec)

Though code change is the core of software evolution, most classical studies of code evolution such as Belady & Lehman's study primarily relied on quantitative and statistical analyses of a program over time. We hypothesize that by treating code change patterns as a first class entity, we can better understand code evolution and also aid in programmers changing code.

Research Questions

- What is an effective, explicit representation of software change?
- How do we effectively identify common change patterns and from which sources?
- Can we use inferred change patterns to better understand software evolution and ultimately aid programmers in changing software?

3. Research Questions: (1 min)

As a first step to prove this hypothesis, I plan to answer the following research questions. Representation: “What is an effective, explicit representation of software change, including granularity and structure of changes?” Algorithm/ Data Source: “How do we effectively identify common change patterns from which sources?” Evidence / Benefits: “Can we use inferred change patterns to better understand software evolution and ultimately aid programmers in changing software?”

Outline

I. Copy and Paste Study

II. Clone Genealogy Study

III. Automatic Inference of Structural Changes

IV. Logical, Structural Delta

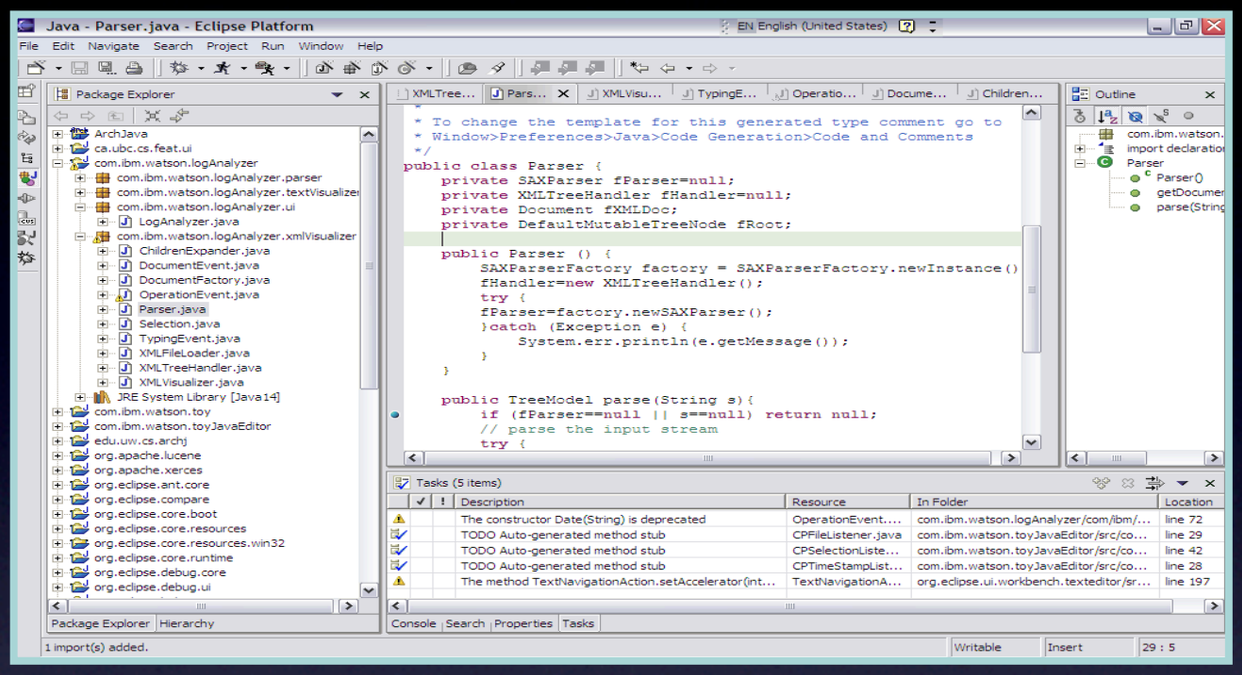
4. Outline (1 min)

In an effort to answer these questions, we have have built several analysis tools that extract or infer code change patterns from different data sources. And we used these tools to study code evolution. The first two studies focus on, in particular, how and why programmers create and maintain duplicated code. The third part of our work involves automatically inferring structural changes --- API changes or refactorings--- to enable multi-version program analyses. Then I am going to describe our on-going work on extracting logical, structural delta to bridge a gap between how programmers view code changes and how popular software engineering tools represent code changes.

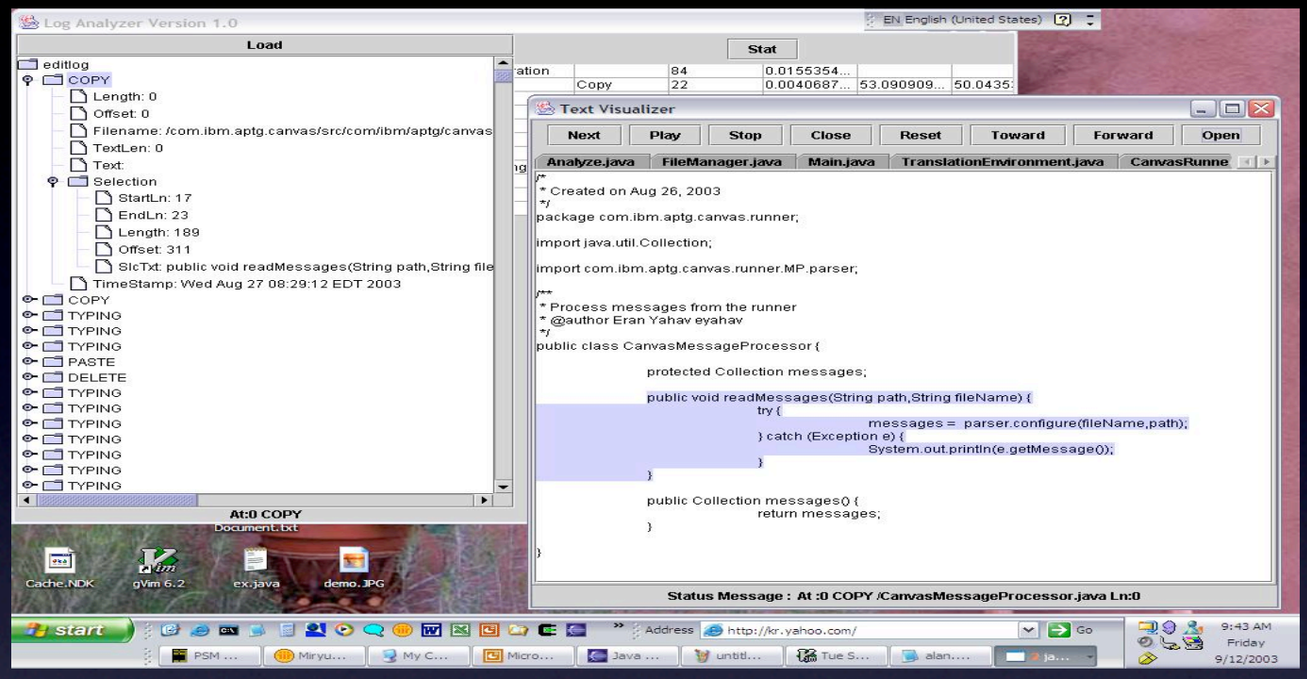
1. A Study of Copy and Paste Programming

Change Patterns from Recorded Editing Operations

1. Capture Edit Operations from IDE

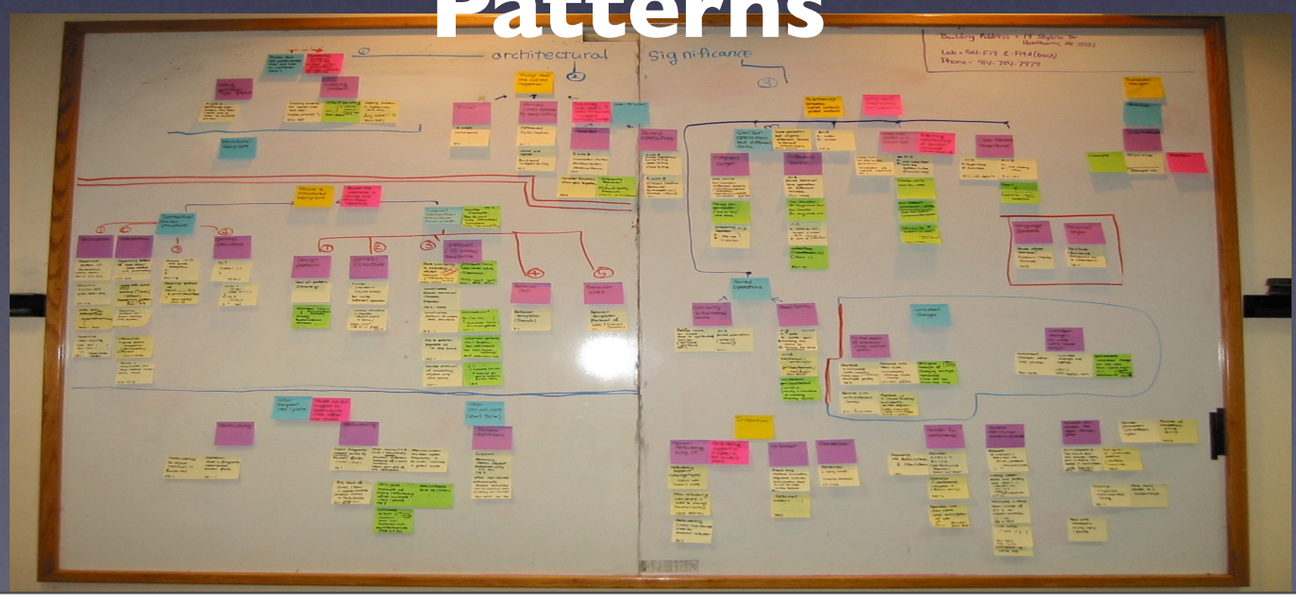


2. Replay and Reconstruct Editing Context



3. Semi-Structured Interviews

4. Create a Taxonomy using Copy&Paste Patterns



5. Copy & Paste Study (1min)

Though copying and pasting code is very common, implication of copy and paste usage patterns have not been studied previously. To understand common C&P patterns, we used an edit capture and replay approach. We developed an Eclipse IDE plugin that logs text edit operations as well as keystrokes and a replayer that play back the edit logs. The change granularity of edit logs is too low level to find any meaningful change patterns. So we resorted to some manual analysis and semi-structured interviews in addition to replaying the edit logs. Our analysis method & results are detailed in our ISESE paper published in 2004.

II. Clone Genealogy Study

Change Patterns from a Set of Program Versions

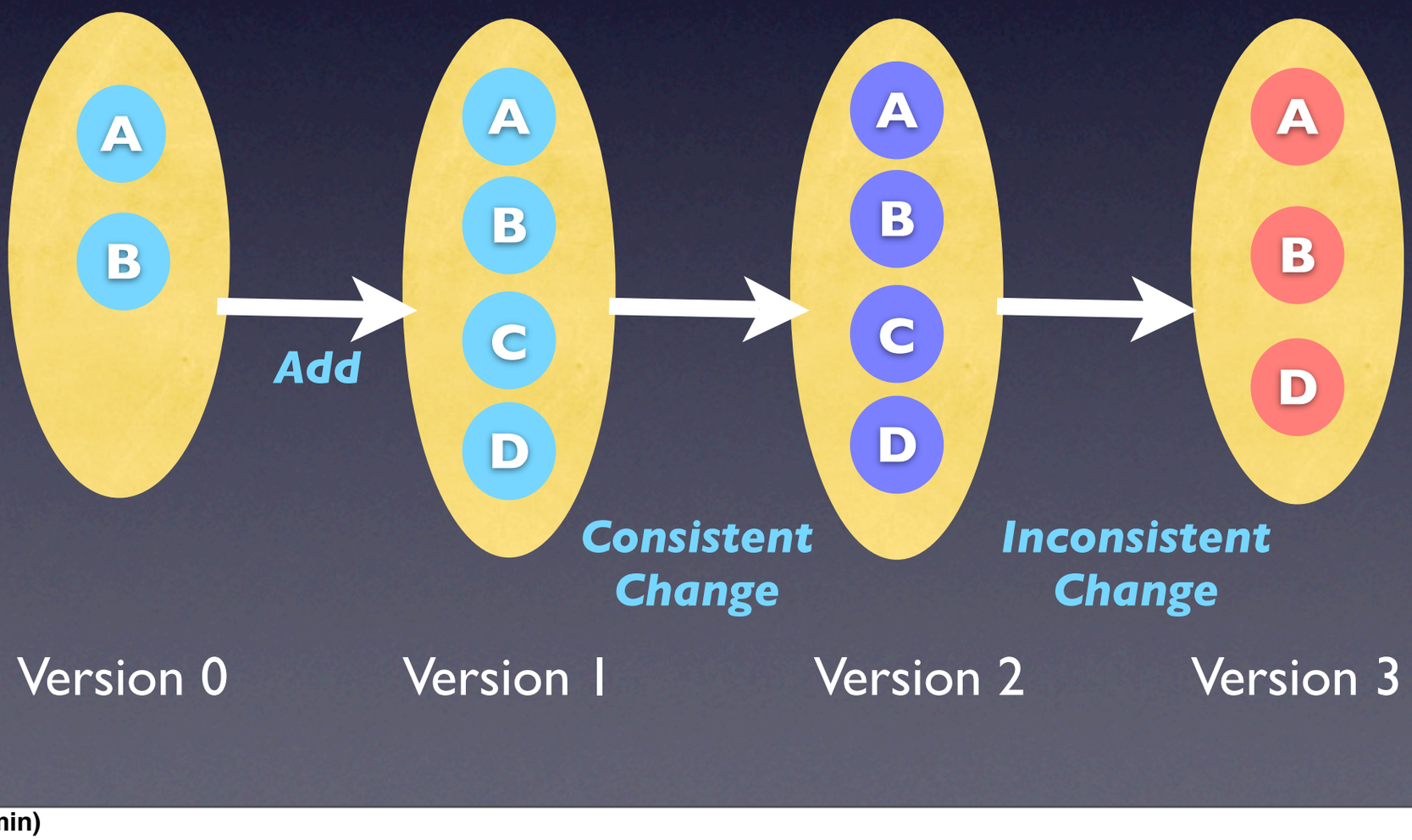
- Most projects do not retain archives of editing logs but rather have CVS.
- Capturing edits in an IDE is limited to a single programmer workspace.
- A longitudinal analysis is not feasible due to high cost of analyzing edits.

6. Transition to CGE (1min)

Through our copy and paste study, we have learned that, recording and replaying edits is very good at forming insights about change patterns, it is limited in a number of ways. First, most projects do not retain archives of editing logs but rather they have version control systems or software release archives. Second, while most projects are developed by more than one developer in a collaborative work environment, capturing edits in an IDE is often limited to a single programer workspace. Third a longitudinal analysis is not feasible due to high cost of analyzing edits.

Clone Genealogy

Clone genealogy is a representation that captures clone change patterns over multiple versions of a program.



7. Clone Genealogy Study (1min)

These limitations motivated our second study and analysis technique that infer clone evolution patterns from a set of program version stored in a source code repository. We developed a representation that captures clone change patterns, which we call as a clone genealogy. And we built a tool that automatically extracts clone genealogies.

An Empirical Study of Code Clone Genealogies

- We studied how often and in which way clones change in open source projects.
- Our study results show that the problem of code clones is not so black and white.
- There are several types of clones that refactoring may not be the best solution.

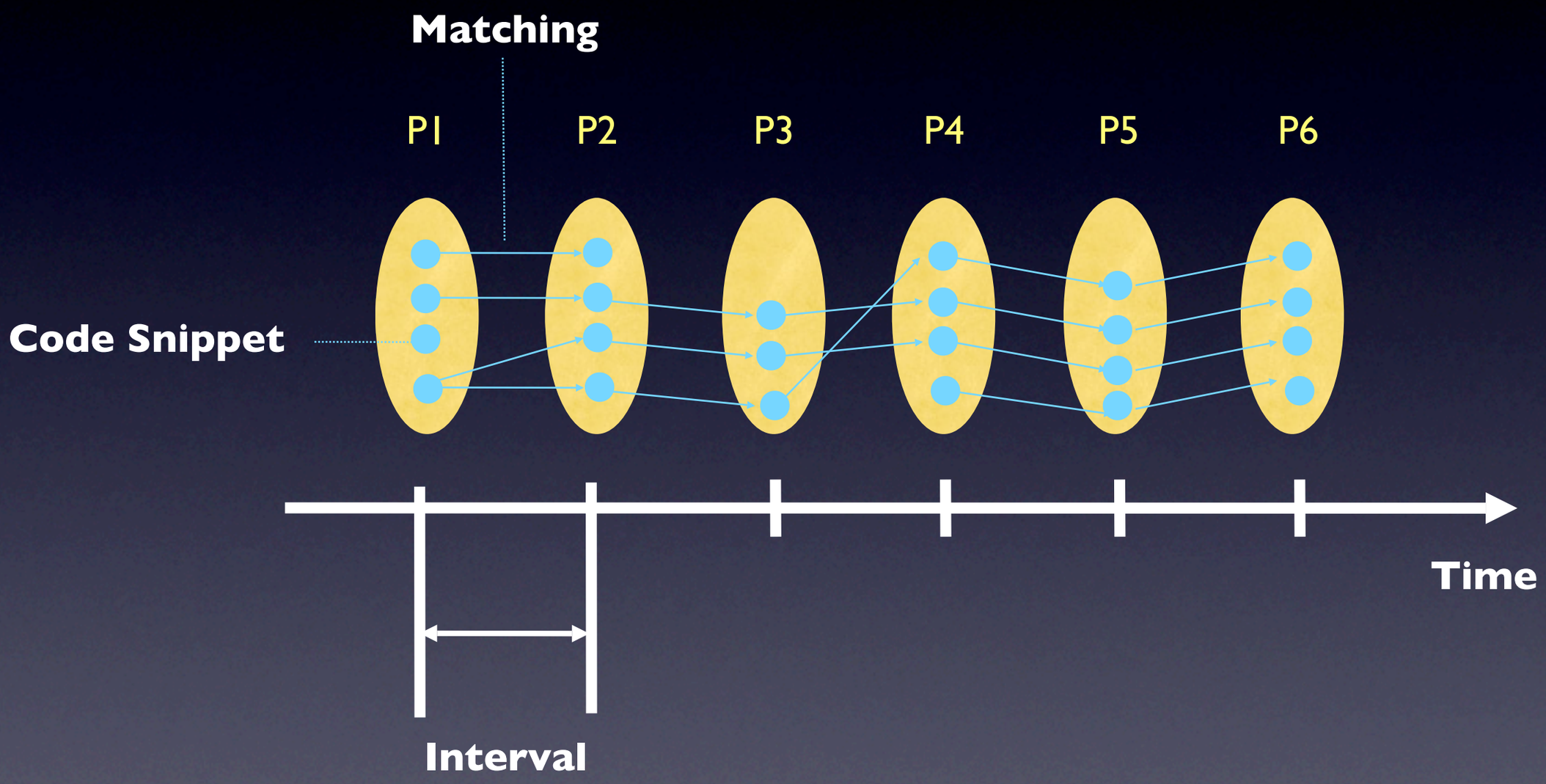
[ESEC/FSE '05, Kim et al.]

Based on extracted clone genealogies, we investigated how long clones stay in a system and how often and in which way clones change in open source projects. It has been broadly assumed that clones are inherently bad and eliminating clones by refactoring would solve the problem. Our study results indicated that the problem of code clone is not so black and white and there are several types of clones that refactoring may not be the best solution. Our study method and results are detailed in our ESEC/FSE published in 2005.

III. Automatic Inference of Structural Change

for Matching Across Program Versions

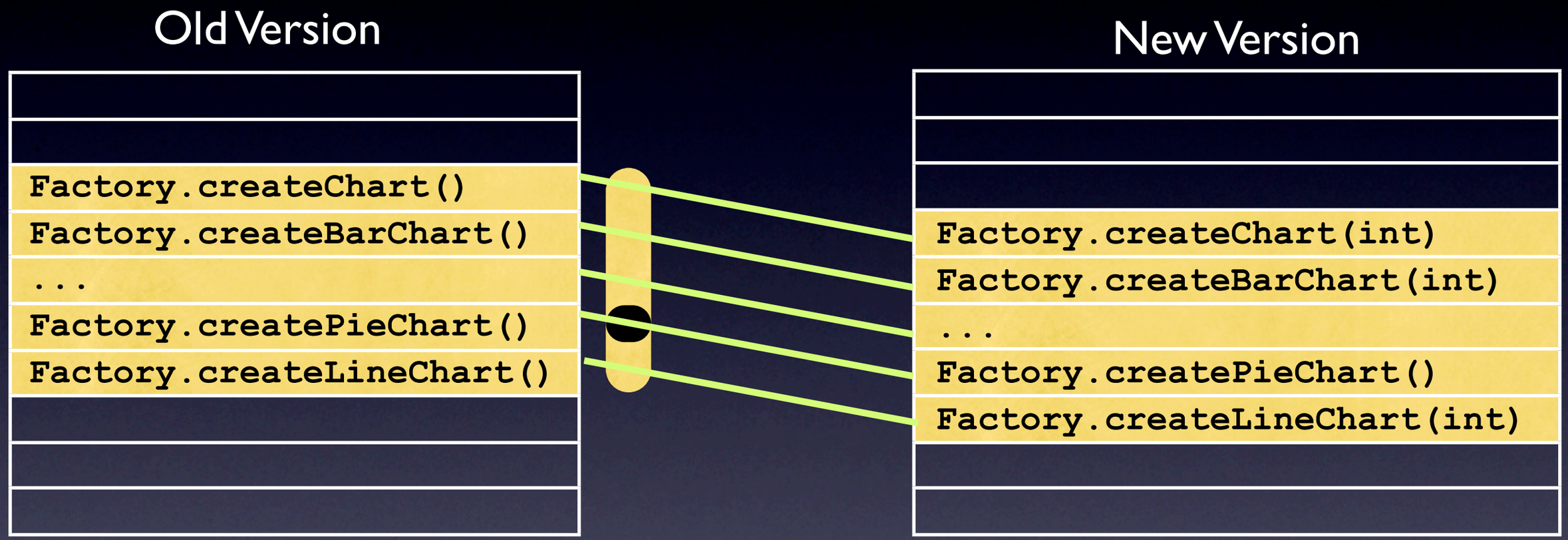
Towards Multi Version Program Analyses



8. Transition to Code Element Matching: Motivating the Matching Problem (30 sec)

Based on our clone genealogy study, we became more interested in analyzing a set of program versions stored in a chronological order to infer fine-grained code level change patterns. We soon realized that analyzing code over time require matching code elements such as files and functions in one version of a program to corresponding entities in another version of a program. This code element matching problem is the dual problem of identifying changes from one version to another.

Code Element Matching



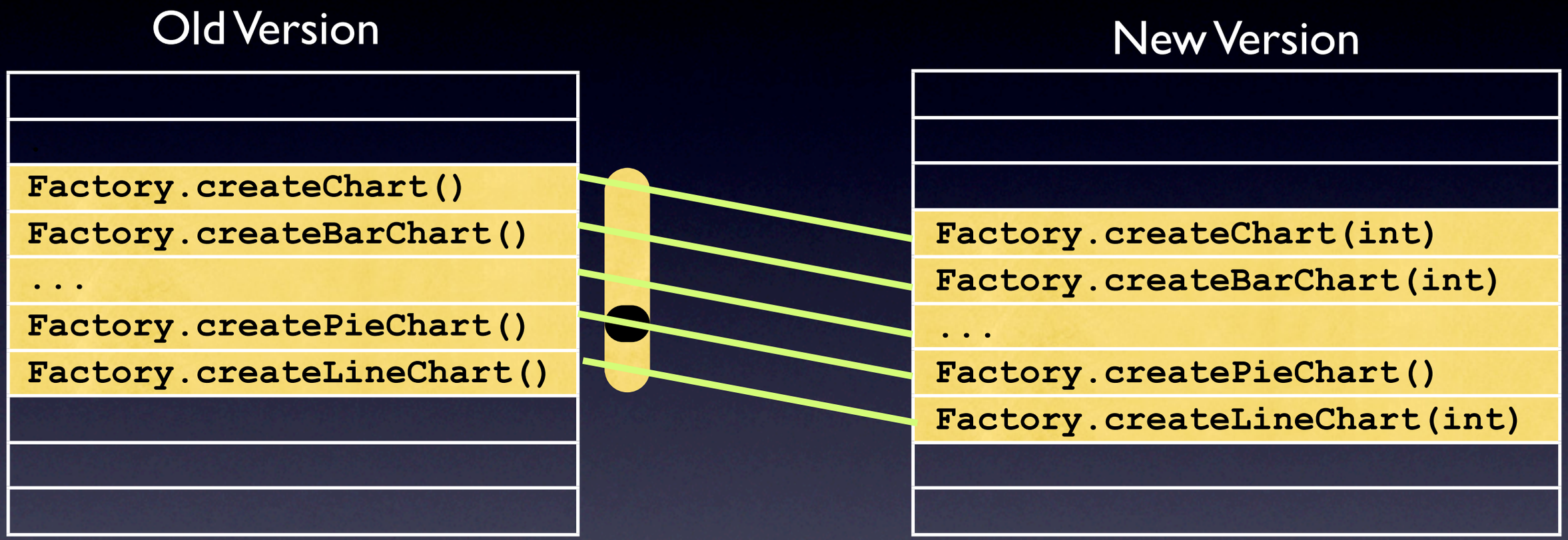
9. Change Rule based on First Order Logic (1 min)

As a first step, we solve the matching problem at or above the level of method headers by inferring structural changes---API changes or refactorings. To represent structural changes we developed our change vocabulary, change rule that is based on first order logic rule. This is based on our observation that often a conceptually simple change involves applying the same atomic transformation on multiple places in the code base.

For example, suppose that a programmer adds an additional boolean input argument to chart creation APIs. Though the goal of this change is simple, it requires modifying all chart creation APIs in a chart factory class. In our representation, this type of change is concisely represented as a single rule.

We developed an inference algorithm that automatically finds a set of likely change rules by comparing two version of a program. Our ICSE paper that will be presented this friday details the description of change rules, evaluation of our tool on multiple open source projects, and potential applications that leverage our inferred change rules in addition to code element matching.

Change Rule



For all x in Factory.create*Chart(*)
except {Factory.createPieChart()}
argAppend(x, [int])
14 matches and 1 exception

9. Change Rule based on First Order Logic (1 min)

As a first step, we solve the matching problem at or above the level of method headers by inferring structural changes---API changes or refactorings. To represent structural changes we developed our change vocabulary, change rule that is based on first order logic rule. This is based on our observation that often a conceptually simple change involves applying the same atomic transformation on multiple places in the code base.

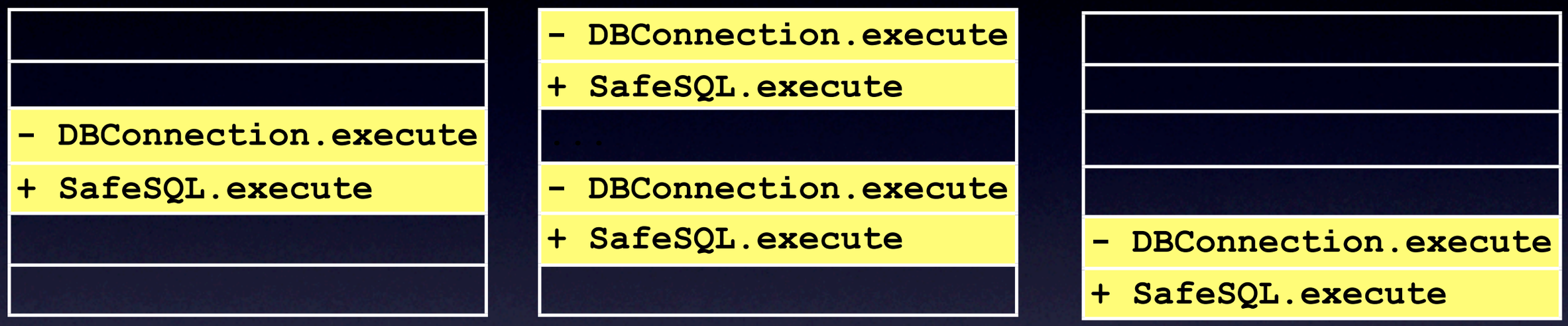
For example, suppose that a programmer adds an additional boolean input argument to chart creation APIs. Though the goal of this change is simple, it requires modifying all chart creation APIs in a chart factory class. In our representation, this type of change is concisely represented as a single rule.

We developed an inference algorithm that automatically finds a set of likely change rules by comparing two version of a program. Our ICSE paper that will be presented this friday details the description of change rules, evaluation of our tool on multiple open source projects, and potential applications that leverage our inferred change rules in addition to code element matching.

IV. Logical Structural Delta

to Enhance Program Understanding Tasks

Logical Structural Delta



changedClass(Foo), changedClass(Bar), changedClass(Baz)
inherits(Foo, Baz), inherits(Bar, Baz)

for all x changedMethod(x), before_calls(x, DBConnection.execute)
=> after_calls(x, SQLSafeUtil.execute)

10. Logical, Structural Delta (1min)

Now I am going to share a little bit about our on-going work. We believe that there is a significant gap between how programmers think about a program change and how such change is represented in most software engineering tools we use daily: diff, CVS, unix patch tool, etc. Programmers often think of changes in terms of structural, semantic information. However, commonly used diff tool represents such changes in terms of textual delta. To bridge this gap, our goal is to augment diff outputs by extracting program structure information. We plan to represent such structural information as logical facts and rules by borrowing notation and concept from logic programming. For example, suppose that a programmer added two classes Foo and Bar that both inherit the base class “Base”. Logical structural delta for this change will include the following ground predicates to show how added code elements are related to one another.

addedClass(Foo), addedClass(Bar), after_inherits(Foo, Base), after_inherits(Bar, Base)

As another example, suppose that a programmer changed all invocation to DBConnection.execute with SQLSafeUtil.execute. Diff output may involve textual modifications in multiple files. In this case, the inferred logical structural delta concisely represents such changes in first order logic rule.

for all x changedMethod(x), before_calls(x, DBConnection.execute) => after_calls (x, SQLSafeUtil.execute)"

We believe this meta level information will improve the effectiveness of many program understanding tasks: understanding other developers’ change, invoking queries of program structure on recent changes, finding relevant code examples, etc. We plan to conduct case studies to show how the extracted LSD can augment current program understanding tasks.

Expected Contributions

- A demonstration of edit capture & replay approach
- A *clone genealogy* representation and an empirical study
- A change rule representation and an inference algorithm
- Initial evidence that shows the benefits of *logical, structural delta*

11. Summary (1min)

In summary, through my doctoral dissertation work, I am planning to make the following contributions.

1. We demonstrated an edit capture & replay approach toward gathering insights about software change patterns.

2. We developed a clone genealogy representation and analyzed code evolution not only quantitatively but qualitatively based on genealogies extracted from a set of program versions.

3. We developed a change vocabulary that concisely describes a set of related structural changes and a tool that automatically infer change rules.

4. We plan to gather initial evidence that augmenting logical, structural delta on regular diff output can enhance program understanding tasks and enable software engineering tools that utilize explicit, high level change patterns.

9:30 mins