Editing Code

SWE 795, Fall 2019

Software Engineering Environments

GEORGE MASON UNIVERSITY

Today

- Part 1 (Lecture)(~80 mins)
  - Editing Code
- Break!
- Part 2 (Discussion)(~60 mins)
  - Discussion of readings

What IDE features do you use when editing code?

Demo: JS in WebStorm

Editing Code

- What types of edits do developers make?
- What mistakes occur? How can they be prevented?
- How can developers edit at a level of abstraction beyond lines and characters?

- Techniques we will examine today
  - Structured editors
  - Editable program views
  - Copy & paste reuse
  - Refactoring
  - Systematic edits
  - Exploratory programming

Structured Editors: Motivation

- Syntax can be hard
  - Have to learn the right syntax (challenging for programming or language novices)
  - Getting syntax wrong creates errors

- What if we could have a development environment where it was impossible to have a syntax error

Structured Editors: Idea

- Developers edit code through commands that create program elements
  - e.g., create an if statement through a keyboard shortcut or drag & drop
- Edits are semantic rather than syntactic
  - Individual elements expose specific elements they support
  - Cannot make edits that crosscut element structure

Cornell Program Synthesizer

- Introduced key concepts

IF (*condition*)
THEN *statement*
ELSE *statement*

→

IF (k > 0)
THEN *statement*
ELSE PUT SKIP LIST ('not positive');

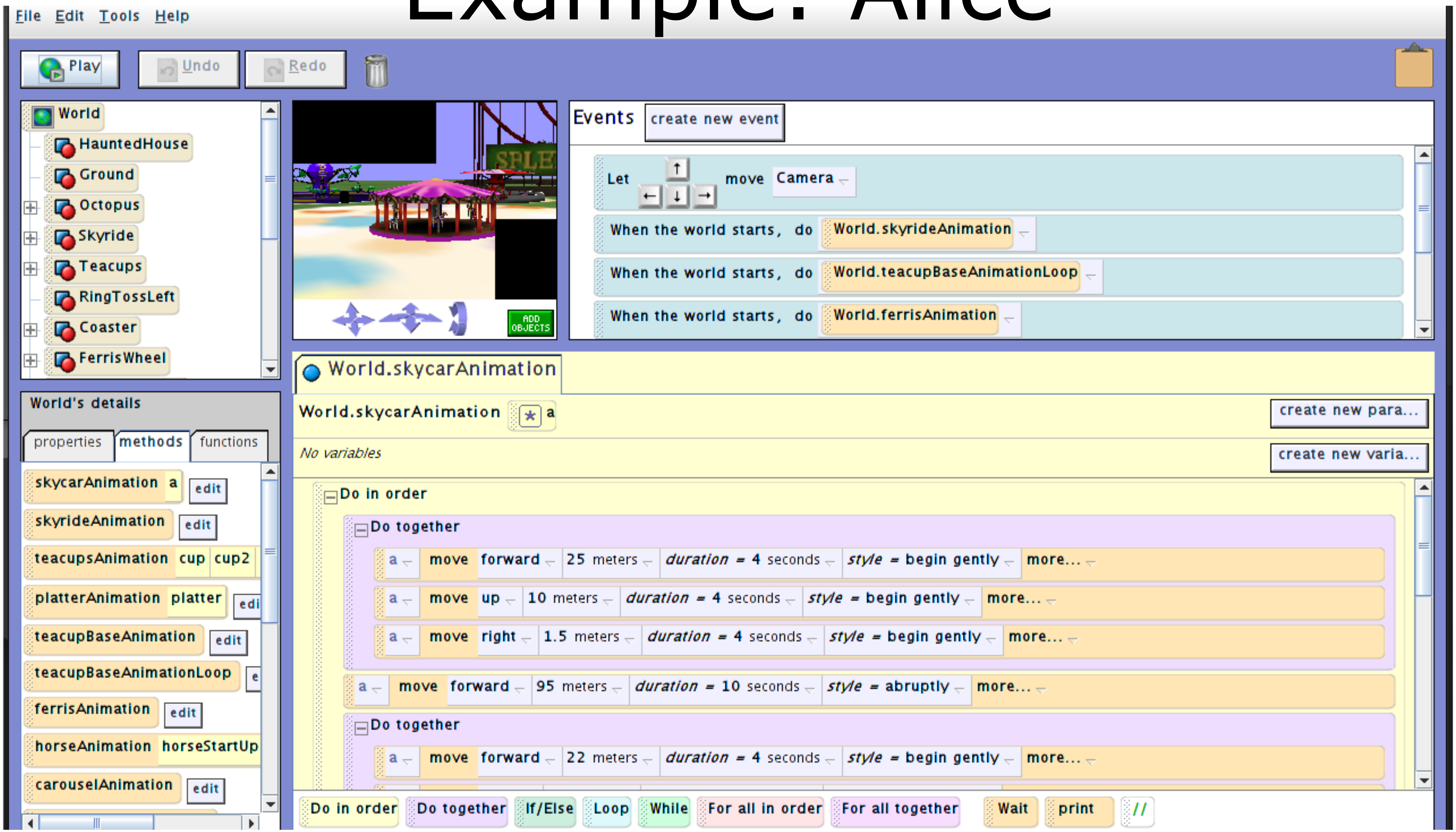
Tim Teitelbaum and Thomas Reps. 1981. The Cornell program synthesizer: a syntax-directed programming environment. Commun. ACM 24, 9 (September 1981), 563-573.

What happened?

- Structured editors make unstructured edits hard
  - Hard to add / remove lines that crosscut structure
  - Hard to copy and paste in ways that crosscut structure
  - If you already know the syntax, may be slower to select syntax from command or drag and drop than it is to type

- But… if you don't know the syntax at all, can be helpful
  - —> Extensive use of syntax directed editors in programming environments for novice programmers

Example: Alice



http://www.alice.org/3.1/Materials/Videos/01.BriefTour.mp4

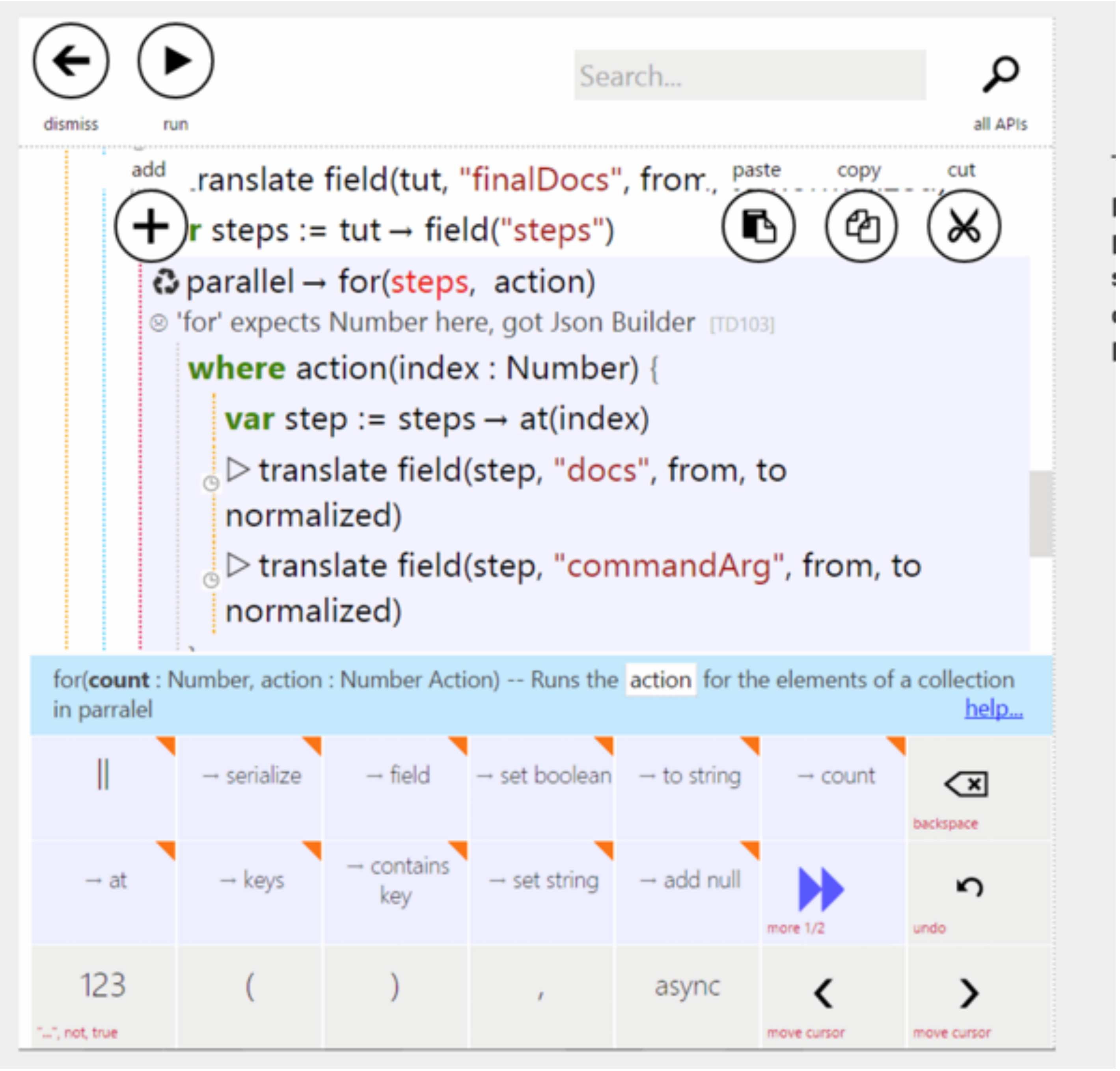
Alice: Lessons Learned from Building a 3D System for Novices. Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, Kevin Christiansen, Rob Deline, Jim Durbin, Rich Gossweiler, Shuichi Kogi, Chris Long, Beth Mallory, Steve Miale, Kristen Monkaitis, James Patten, Jeffrey Pierce, Joe Schochet, David Staak, Brian Stearns, Richard Stoakley, Chris Sturgill, John Viega, Jeff White, George Williams, and Randy Pausch, CHI 2000

Example: Scratch

https://vimeo.com/65583694

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. Commun. ACM 52, 11 (November 2009), 60-67.

Example: TouchDevelop



https://www.youtube.com/watch?v=ve2E90wh-wk

Editable program views

- Expressing code edits through textual changes can be time consuming
  - extra boilerplate, code duplication, etc.

- Key idea: Enable developers to instead interact with abstracted view of code
  - Use edits to abstract view to edit underlying code

Linked Editing

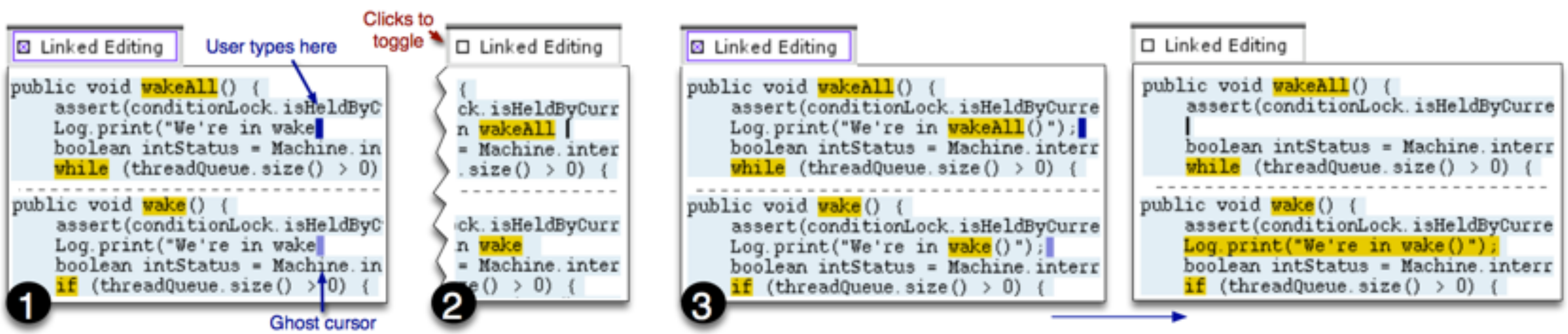


Figure 2. (1) Adding a line to two clones. (2) Modifying one instance. (3) Deleting line in one instance.

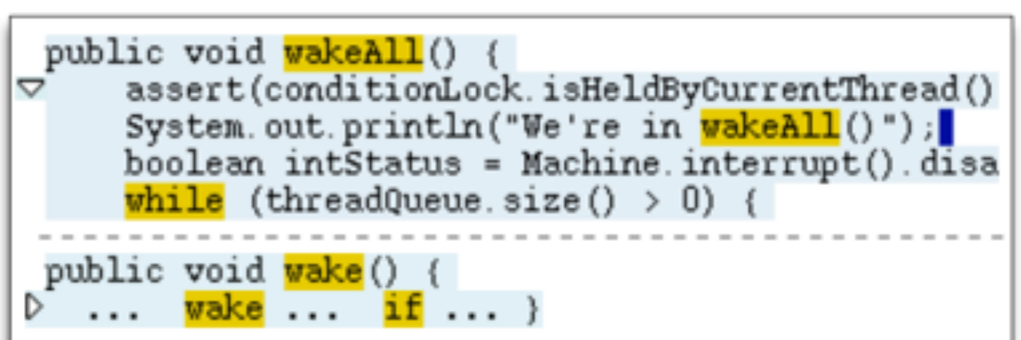


Figure 3. An elided clone looks similar to a function definition and use

Michael Toomim, Andrew Begel, and Susan L. Graham. 2004. Managing Duplicated Code with Linked Editing. In Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC '04). IEEE Computer Society, Washington, DC, USA, 173-180.

Registration-based language abstractions

```
public class WorkbenchHistoryPageSite implements IHistoryPageSite {

    GenericHistoryView part;  |Getter: public IWorkbenchPart get

    IPageSite site; |Getter: public getWorkbenchPageSite
                    |Delegates Implementation of IHistoryPageSite (3 of 8 methods):
                    |  public setSelectionProvider
                    |  public getSelectionProvider
                    |  public getShell
```

```
public static BundleDesc[] getDependentBundles(BundleDesc root) {
    BundleDesc[] imported = getImportedBundles(root);
    BundleDesc[] required = getRequiredBundles(root);
    BundleDesc[] dependents = imported + required;
    return dependents;
}
```

(a) An array-concatenation registration. The presentation uses an overloaded "+" to indicate the concatenation of two arrays through calls to `System.arraycopy`.

```
public static BundleDesc[] getDependentBundles(BundleDesc root) {
    BundleDesc[] imported = getImportedBundles(root);
    BundleDesc[] required = getRequiredBundles(root);
    BundleDesc[] dependents = new BundleDesc[imported.length + required.length];
    dependents[0 : *] = imported[0, imported.length];
    dependents[imported.length : *] = required[0, required.length];
    return dependents;
}
```

(b) Two arraycopy registrations. The notation "0 : *" indicates that the elements are copied into the indices starting at 0. An icon is used to disambiguate the syntax, by making it clear that the dependents array is not truncated to the length of the copied elements.

Samuel Davis and Gregor Kiczales. 2010. Registration-based language abstractions. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10). ACM, New York, NY, USA, 754-773.

Copy & paste code reuse

- A very common way to edit code is by copying existing code. —> copy & paste reuse
- Creates code duplication
  - But… ok if this code duplication does not represent new abstraction

- Studies have attempted to understand when code duplication introduced by copy & paste is bad

- Many tools to detect code clones introduced by copy & paste

Slides for this section adapted from 05-899D Human Aspects of Software Development Spring 2011, “Software Evolution” by YoungSeok Yoon

Why do developers copy & paste code?

- structural template (the most common intention)
  - relocate, regroup, reorganize, restructure, refactor
- semantic template
  - design pattern
  - usage of a module (following a certain protocol)
  - reuse a definition of particular behavior
  - reuse control structure (nested if~else or loops)

M. Kim, L. Bergman, T. Lau, and D. Notkin (2004), “An ethnographic study of copy and paste programming practices in OOPL,” in *Proceedings of International Symposium on Empirical Software Engineering (ISESE’04)*, pp. 83-92.

Why do developers copy & paste?

- Forking
    - Hardware variations
    - Platform variation
    - Experimental variation
- Templating
    - Boiler-plating due to language in-expressiveness
    - API/Library protocols
    - General language or algorithmic idioms
- Customization
    - Bug workarounds
    - Replicate and specialize

C. Kapser and M. W. Godfrey (2006), "'Cloning Considered Harmful' Considered Harmful," in *13th Working Conference on Reverse Engineering (WCRE '06)*, 2006, pp. 19-28.

Properties of copy & paste reuse

- Unavoidable duplicates (e.g., lack of multiple inheritance)

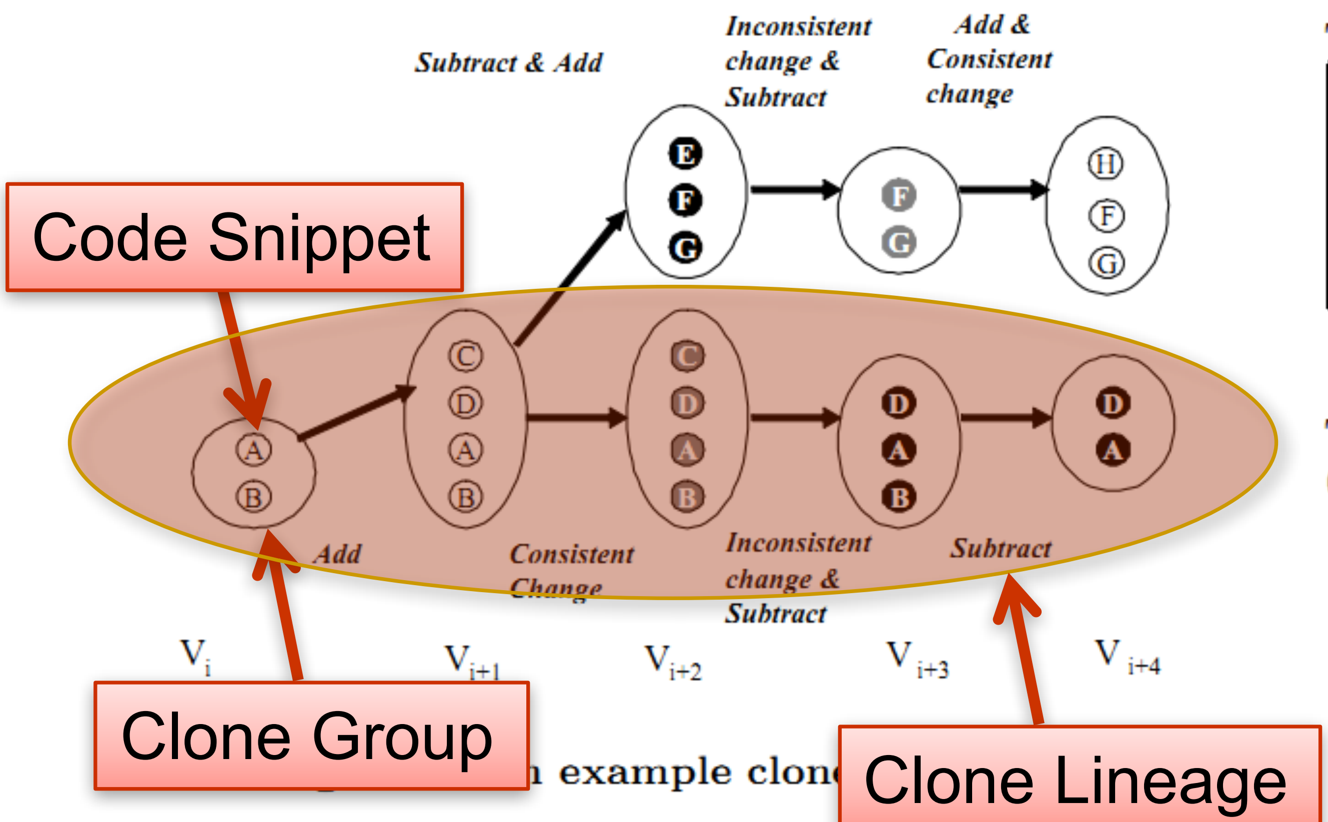
- Programmers use their memory of C&P history to determine when to restructure code
  - delaying restructuring helps them discover the right level of abstraction

- C&P dependencies are worth observing and maintaining

M. Kim, L. Bergman, T. Lau, and D. Notkin (2004), "An ethnographic study of copy and paste programming practices in OOPL," in *Proceedings of International Symposium on Empirical Software Engineering (ISESE'04)*, pp. 83-92.

Code clone genealogies

- Investigates the validity of the assumption that code clones are bad
- Defines clone evolution model



- Built an automatic tool to extract the history of code clones from a software repository

Table 1: Description of Two Java Subject Programs

Program	*carol*	*dnsjava*
URL	carol.objectweb.org	www.dnsjava.org
LOC	7878 ~ 23731	5756 ~ 21188
duration	26 months	68 months
# of check-ins	164	905

Table 2: Clone Genealogies in *carol* and *dnsjava* (min_{token} **=30,** $sim_{th} = 0.3$)

# of genealogies	carol	dnsjava
total	122	140
false positive	13	15
true positive	109	125
locally unfactorable	70 (64%)	61 (49%)
consistently changed	41 (38%)	45 (36%)

11

M. Kim, V. Sazawal, D. Notkin, and G. Murphy (2005), "An empirical study of code clone genealogies," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13).*

Refactoring: Motivation

"Refactoring is the process of changing a software system in such a way that it ***does not alter the external behavior*** of the code yet ***improves its internal structure***." [Fowler 1999]

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts (1999), "*Refactoring: Improving the Design of Existing Code*", 1st ed. Addison-Wesley Professional.

Slides for this section adapted from 05-899D Human Aspects of Software Development Spring 2011, "Software Evolution" by YoungSeok Yoon

First tool: A Refactoring Tool for Smalltalk

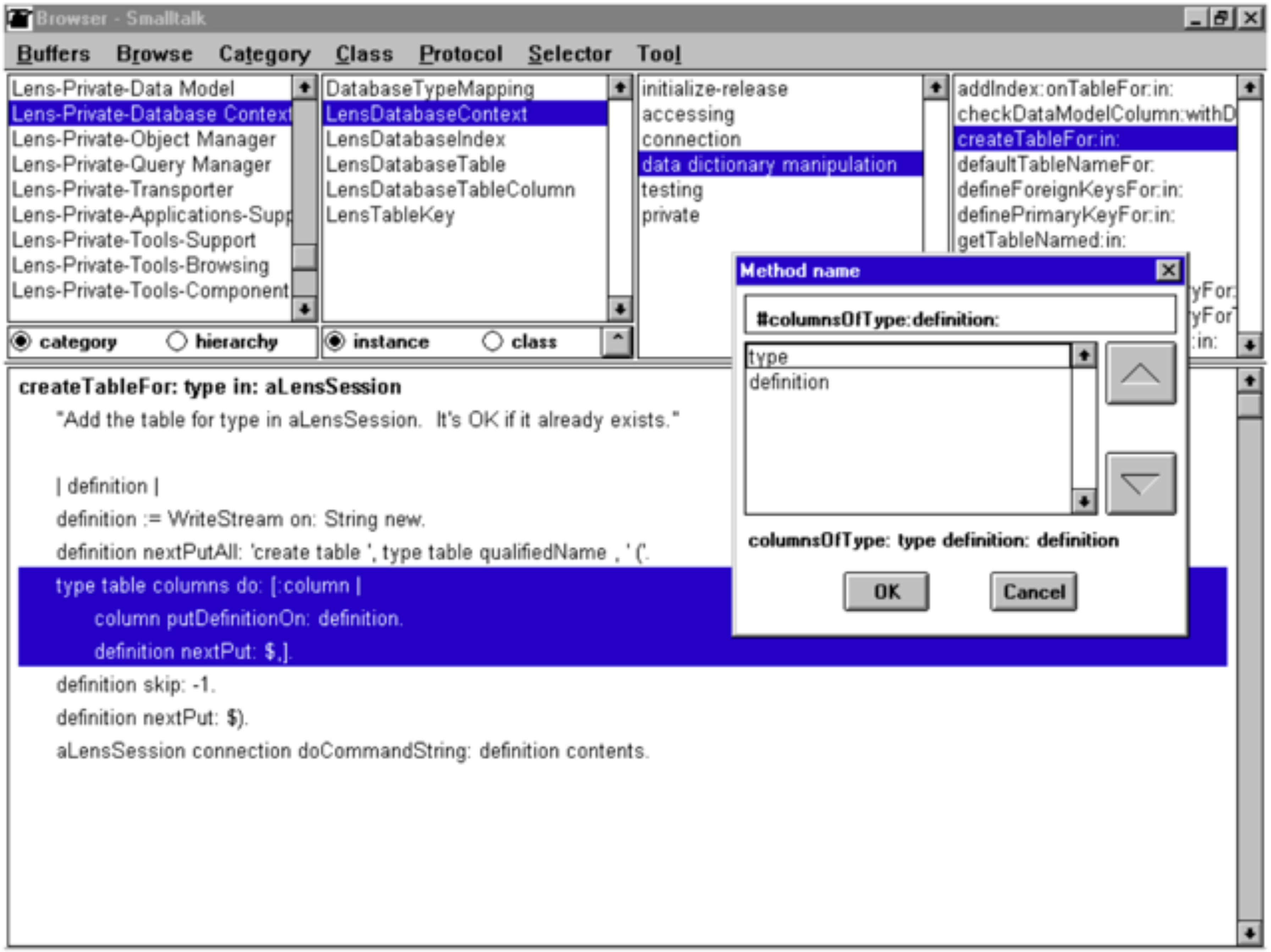


Figure 2 - Screenshot of Refactoring Browser during extract code as method refactoring

D. Roberts, J. Brant, and R. Johnson (1997), "A refactoring tool for smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, pp. 253-263.

(Very) brief story of refactoring

- Started with academic work defining idea of refactoring
  - William F. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois, 1992.
- Academic work for tools quickly followed (e.g., [Brant TPOS97])
  - Built in real IDE for Smalltalk from beginning
- Disseminated by agile thought leaders like Martin Fowler
- Adopted into mainstream IDEs like Eclipse, Visual Studio
- Became standard accepted feature of IDES
- Research continued
  - Do developers use refactoring tools?
  - Could they use them more?
  - How could refactoring tools better support developers?

Developers manually perform refactorings not yet supported by tools

- About 70% of structural changes may be due to refactorings
- About 60% of these changes, the references to the affected entities in a component-based application can be automatically updated
- State-of-the-art IDEs only support a subset of common low-level refactorings, and lack support for more complex ones

Type of refactoring	# detected	Eclipse support
Convert anonymous class to nested*[2]	12	√
Convert nested type to top-level	19	√
Convert top-level type to nested	20	×
Move member class to another class	29	√
Extract package	16	×
Inline package	3	×

Type of refactoring	# detected	Eclipse support
Pull up field/method	279	√
Push down field/method	53	√
Extract interface	28	√
Extract superclass	15	×
Extract subclass	4	×
Inline superclass	4	×
Inline subclass	7	×

Type of refactoring	# detected	Eclipse support
Extract constant interface	5	√
Inline constant interface	2	×
Extract class	95	×
Inline class	31	×

Type of refactoring	# detected	Eclipse support
Information hiding	751	×
Generalize type	107	√
Downcast type	85	×
Introduce factory	19	√
Change method signature	4497	√
Introduce parameter object*	4	×
Extract method*	45	√
Inline Method*	31	√

Z. Xing and E. Stroulia (2006), “Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study,” in *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM ‘06)*, 2006, pp. 458-468.

Supporting systematic edits

- Developers sometimes make edits to multiple files that are very similar

- Tool idea: find commonality in edits between 2 or more examples, generalize to others

Example

A_{old} to A_{new}

```
  public void textChanged (TEvent event) {
    Iterator e=fActions.values().iterator();
- print(event.getReplacedText());
- print(event.getText());
    while(e.hasNext()){
-    MVAction action = (MVAction)e.next();
-    if(action.isContentDependent())
-      action.update();
+    Object next = e.next();
+    if (next instanceof MVAction){
+      MVAction action =(MVAction)next;
+      if(action.isContentDependent())
+        action.update();
+    }
    }
    System.out.println(event + " is processed");
}
```

B_{old} to B_{new}

```
  public void updateActions () {
    Iterator iter = getActions().values().iterator();
    while(iter.hasNext()){
-     print(this.getReplacedText());
-     MVAction action=(MVAction)iter.next();
-     if(action.isDependent())
-       action.update();
+     Object next = iter.next();
+     if (next instanceof MVAction){
+       MVAction action =(MVAction)next;
+       if(action.isDependent())
+         action.update();
+     }
+     if (next instanceof FRAction){
+       FRAction action = (FRAction)next;
+       if(action.isDependent())
+         action.update();
+     }
    }
    print(this.toString());
}
```

C_{old} to C_{new}

```
  public void selectionChanged (SEvent event) {
    Iterator e = fActions.values().iterator();
    while(e.hasNext()){
-     MVAction action=(MVAction)e.next();
-     if(action.isSelectionDependent())
-       action.update();
+     Object next = e.next();
+     if (next instanceof MVAction){
+       MVAction action =(MVAction)next;
+       if(action.isSelectionDependent())
+         action.update();
+     }
    }
}
```

Fig. 1. A systematic edit to three methods based on revisions from 2007-04-16 and 2007-04-30 to org.eclipse.compare

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 502-511.

Locating and applying systematic edits

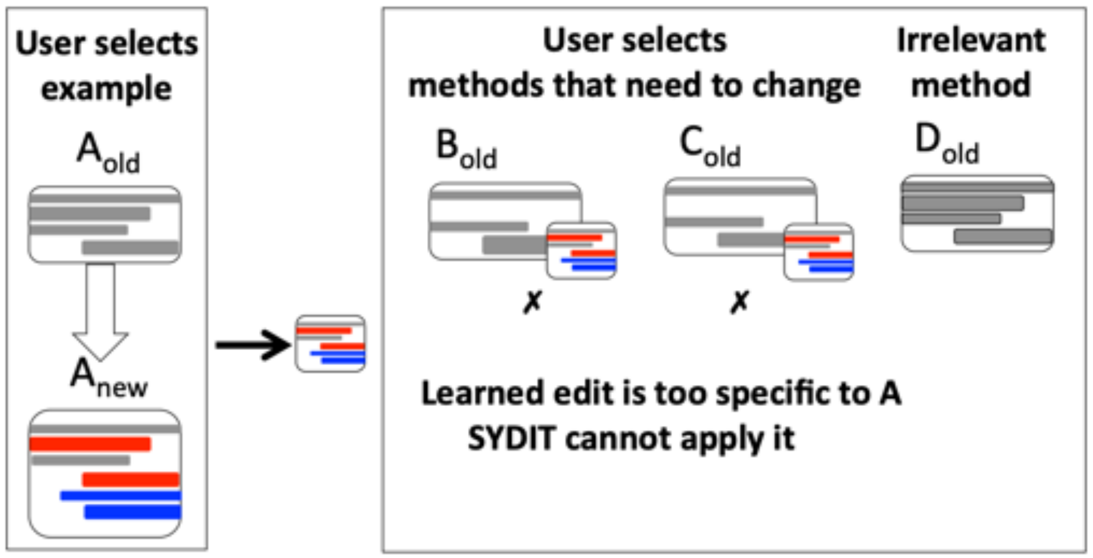


Fig. 2. SYDIT learns an edit from one example. A developer must locate and specify the other methods to change.

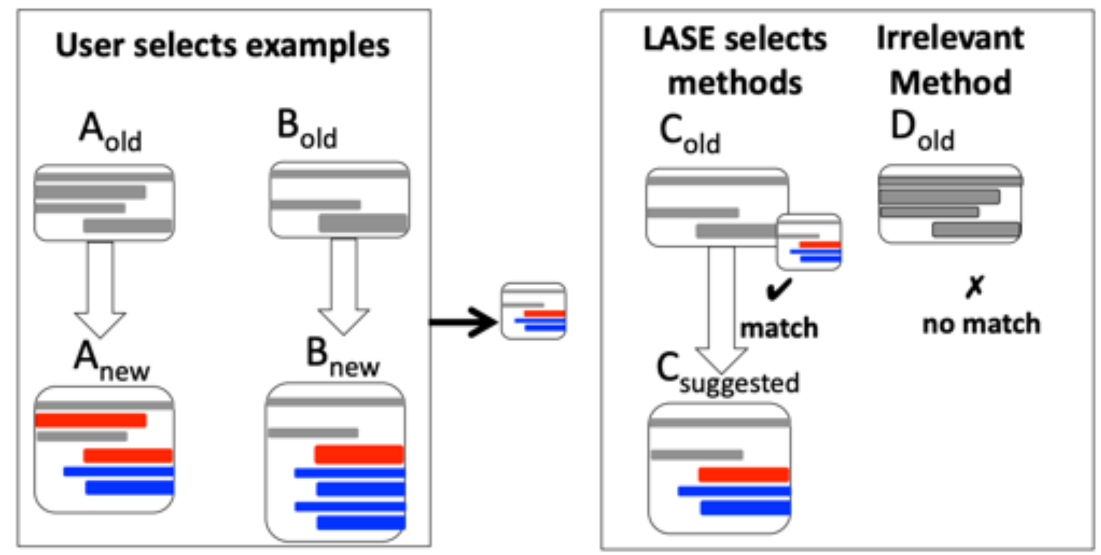


Fig. 3. LASE learns an edit from two or more examples. LASE locates other methods to change.

```
1. … … method_declaration(… …){
2.    T$0 v$0 = v$1.m$0().m$1();
3.    DELETE: m$2(v$2.m$3());
4.    DELETE: m$2(v$2.m$4());

3.    while(v$0.m$5()){

4.       UPDATE: T$1 v$3 = (T$1)v$0.m$6();
5.           TO: T$2 v$4 = v$0.m$6();
6.       if(v$3.m$7()){
7.        … …
8.       }
MOVE 9.  INSERT: if(v$4 instanceof T$1){
10.          INSERT: T$1 v$3 = (T$1)v$4;
11.          … …
12.             }
```

Fig. 4. Edit script from SYDIT abstracts all concrete names. Gray marks edit context, red marks deletions, and blue marks additions.

```
1. … … method_declaration(… …){
2.    Iterator v$0 = u$0:FieldAccessOrMethodInvocation
                          .values().iterator();
3.    while(v$0.hasNext()){
4.       UPDATE: MVAction action = (MVAction)v$0.next();
5.           TO: Object next = v$0.next();
6.       if(action.m$0()){
7.        … …
8.       }
MOVE 9.  INSERT: if(next instanceof MVAction){
10.          INSERT: MVAction action = (MVAction)next;
11.          … …
12.             }
```

Fig. 5. Edit script from LASE abstracts code names that differ in the examples and uses concrete names for common ones. Gray marks edit context, red marks deletions, and blue marks additions.

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 502-511.

Specifying program transformations

Before:

```
if s != nil {
 for _, x := range s {
    ...
 }
}
```

After:

```
for _, x := range s {
   ...
}
```

Match template:

```
if :[var] != nil {
 for :[_] := range :[var] {
    :[body]
  }
}
```

Rewrite template:

```
for :[_] := range :[var] {
   :[body]
}
```

Figure 1. *Top:* A textual description for simplifying a `nil` check Go code, taken from the Go `staticcheck` tool. *Bottom:* Our match template and rewrite templates for the `nil-check` pattern above.

```
func (c *SymbolCollector) addContainer(...) {
  if fields.List != nil {
    for _, field := range fields.List {
      if field.Names != nil {
        for _, fieldName := range field.Names {
          c.addSymbol(field, fieldName.Name)
        }
      }
    }
  }
  ...
}
```

(a) Highlighted lines 2 and 4 contain redundant `nil` checks in Go code: iterating over a container in a for loop implies it is non-nil.

```
func (c *SymbolCollector) addContainer(...) {
    for _, field := range fields.List {
        for _, fieldName := range field.Names {
          c.addSymbol(field, fieldName.Name)
        }
    }
    ...
}
```

(b) Rewrite output simplifying the Go code above.

Figure 2. Redundant code pattern and simplification.

Exploratory Programming

- Developers sometimes explore programs without knowing a priori what behavior they want to create or the best way to implement it

- Goal: enable developers to explore *variations* in programs

Domains for exploratory programming

- Learning programming through play
- Digital art and music: generative music, live coding, performance
- Data science: tasks analyzing data, building a machine learning model
- Software engineering: backtracking, commenting out or undoing different ideas; figuring out how an API should be used

M. Beth Kery and B. A. Myers, "Exploring exploratory programming," *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Raleigh, NC, 2017, pp. 25-29. doi: 10.1109/VLHCC.2017.8103446

Code quality tradeoffs

- Often associated with code being hard to read
  - If rapidly changing it, no sense in spending time making it clear and easy to read
  - “I know how to write code. And I know that I could write functions to reuse functions and I could try to modularize things better, and sometimes I just don’t care because why am I going to put effort in that if I’m not going to use it again?”
  - In TDD methodology, make it work (functional), make it right (easy to read), make it fast (performant) are 3 separate stages and should not progress till finished previous

M. Beth Kery and B. A. Myers, "Exploring exploratory programming," *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Raleigh, NC, 2017, pp. 25-29. doi: 10.1109/VLHCC.2017.8103446

Exploration process

- Backtracking: 2 or more edit run cycles that are close in time and affect the same code
- Exploration scale:
  - tuning a single variable or parameter to observe effect
  - iterating variations of a function
  - trying out different larger snippets of code
- Exploration duration: transient to long term
- Using exploratory history
  - Often use code history to understand a change or bug

M. Beth Kery and B. A. Myers, "Exploring exploratory programming," *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Raleigh, NC, 2017, pp. 25-29.
doi: 10.1109/VLHCC.2017.8103446

Backtracking in programming

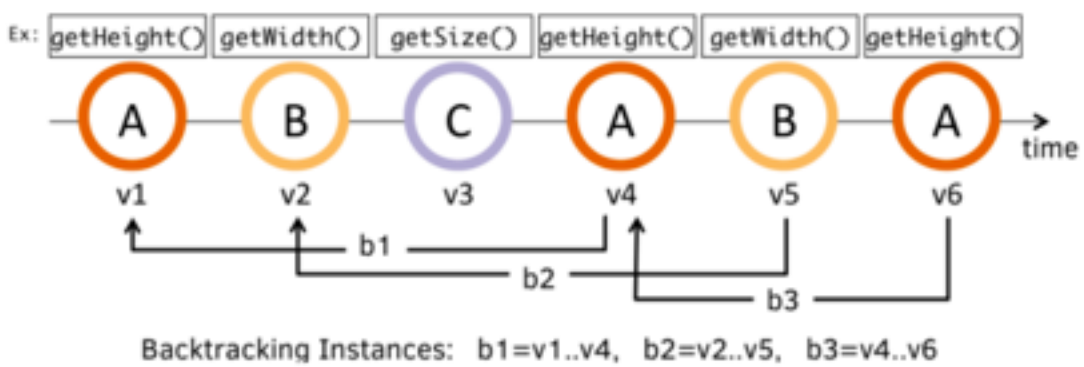


Fig. 1. An example of a node evolution history, which contains three backtracking instances. The node first appeared in the code as "getHeight();" (v1), changed a few times (v2 through v5), and finally ended up back at the original code (v6). The different contents are symbolized as capital letters A, B, and C. There are three backtracking instances in this node history indicated as black backward arrows.

```
Backtracking instance: [4263, 4629]
v1 [4263]return new Point(getWidth(),getHeight());
v2 [4555]return new Point(getWidth() - MARKER_SIZE,getHeight());
v3 [4567]return new Point(getWidth() - MARKER_SIZE,getHeight() - MARKER_SIZE);
v4 [4623]return new Point(getWidth() - MARKER_SIZE,getHeight() - MARKER_SIZE);
v5 [4629]return new Point(getWidth(),getHeight() - MARKER_SIZE);
```

Fig. 2. An example output of our analyzer, showing the history of a statement node. Each row maps to each version (v1,v2, ..., v5). This node contains a single backtracking instance, which is v1...v5. The edit operation IDs were originally 6-digits long (e.g., 184263), but were shortened for brevity.

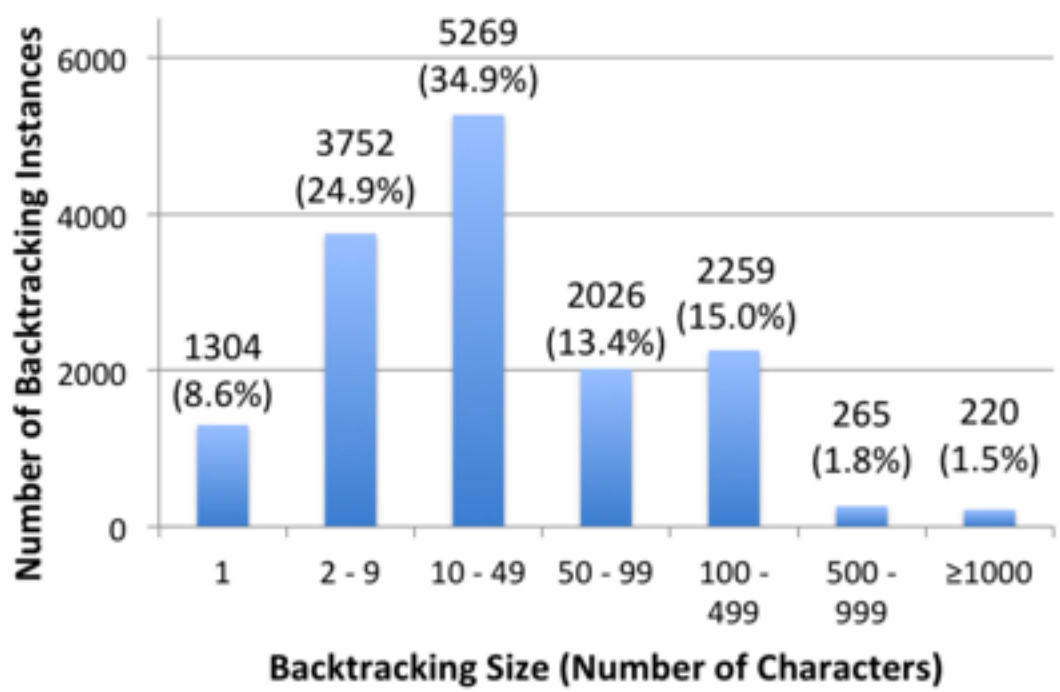


Fig. 3. Distribution of all the detected backtracking sizes

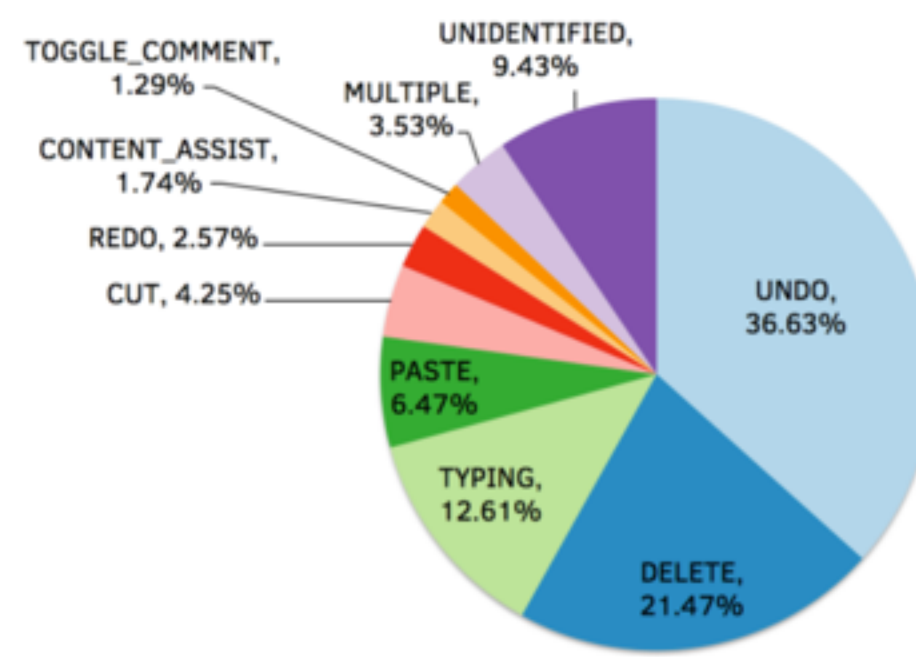


Fig. 5. The identified backtracking tactics

Y. S. Yoon and B. A. Myers, "A longitudinal study of programmers' backtracking," 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Melbourne, VIC, 2014, pp. 101-108.

Exploration by data scientists

- Notebooks used in 3 ways: (1) preliminary scratch pad work, (2) production work, (3) shared work
- Scratch pad use: preliminary and short-lived, answers a specific question: how to debug a piece of code, test out example from internet, test if idea worth pursuing
    - ”I was just testing to make sure I had the syntax right on these tuples.” - IP13
    - “OK so can we do k-means on this dataset and like does it make sense” - IP11
- Sometimes occurs with individual cells, sometimes with whole notebooks

Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18). ACM, New York, NY, USA, Paper 174, 11 pages. DOI: https://doi.org/10.1145/3173574.3173748

Iteration behavior

- Organizing the notebook
  - Examples:
    - Most recent code at the bottom
    - Debugging at the bottom
    - Function refs at top
  - Add cels where the original data analysis took place
- Expand then reduce
  - “So at the beginning it's usually a lot of little code cells that are one at a time... just making things work... I end up with this huge mess where there are several threads in sort of the same series. So I usually go back and start deleting things or combining cells” - IP17
  - Cells enable viewing intermediate results
- Narrative structure: some used note book chronologically following steps in analysis; others were non-chronological, following important decisions

Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18). ACM, New York, NY, USA, Paper 174, 11 pages. DOI: https://doi.org/10.1145/3173574.3173748

Supporting data scientists

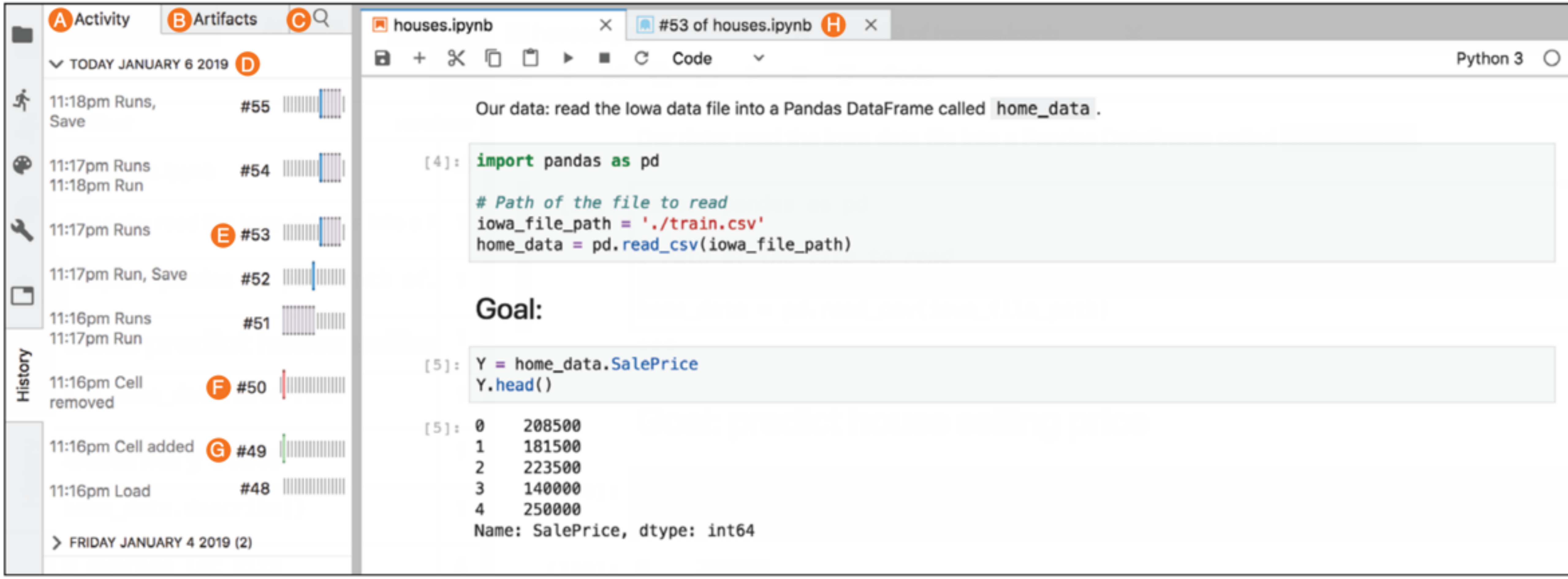


Figure 3: The history tab opens the sidebar for Verdant containing three tabs: Activity (A), Artifacts (B & Fig. 5), and Search (C & Fig. 7). The Activity tab, shown open here, displays a list of events. A date (D) can be opened or collapsed to see what happened that day. Each row shows a version of the notebook (e.g. version #53) with a text description and visual minimap. The minimap shows cells added in green (see G) and deleted in red (F). In (E), a cell was edited and run (in blue), and the following cells were run but remained the same (in grey). The user can open any version (e.g., #53, H & Fig. 8) in a ghost notebook tab for quick reference.

Video: https://dl.acm.org/citation.cfm?doid=3290605.3300322

Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. 2019. Towards Effective Foraging by Data Scientists to Find Past Analysis Choices. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19). ACM, New York, NY, USA, Paper 92, 13 pages. DOI: https://doi.org/10.1145/3290605.3300322