Using Metaphors from Natural Discussion to Improve the Design of Arcum

Macneil Shonle
The University of Texas at San Antonio
mshonle@cs.utsa.edu

William G. Griswold, and Sorin Lerner
UC San Diego
{wgg,lerner}@cse.ucsd.edu

There are many dimensions to improve usability:

- To minimize steps necessary (ease of *use*)
  - Emacs and vi are very easy to use
- To improve the *ease of learning*
  - Emacs and vi are very hard to learn
- Bottom line: To let users get the most *value* from the tool
  - "You're doing it wrong"

- Our focus: Arcum: A meta-programming tool

2

Arcum is for crosscutting program transformations

- Starts with a query:

```
expr == [System.err]
```

- Then, an alternative:

```
expr ==
  [ErrorLog.getLog()]
```

- Leading to results:

```
System.err
System.err.printf …
sendTo(System.err …
System.err
err
stream = System.err
System.err
System.err
```

- Causing transformation:

```
ErrorLog.getLog()
ErrorLog.getLog().pri
ntf …
sendTo(ErrorLog.getLo
g() …
ErrorLog.getLog()
ErrorLog.getLog()
…
```

3

Arcum uses modular concepts

- Descriptions of idioms implement a common interface:

```
interface ErrorLogAccess() {
  abstract access(Expr expr) {
    check {isA(expr, java.io.PrintStream)}
  }
}
```

```
match access(Expr expr) {
  expr == [System.err]
}
```

```
match access(Expr expr) {
  expr ==
    [ErrorLog.getLog()]
}
```

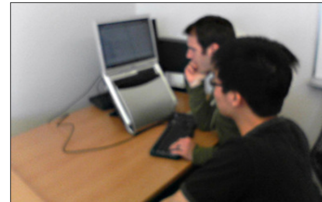
4

Analysis approach for Arcum

- A qualitative, exploratory study:
  - How would users even approach or conceive of a meta system?
- Goal: Guide the design of the language, its examples, and the tutorials.
- Approach: Identify the metaphors that participants used, as a way to know their expectations.
  - Like Grounded Theory, we didn't **start** from a hypothesis
  - Unlike Grounded Theory, we didn't apply coding, etc…

Analysis approach for Arcum

- We conducted a user study of 3 pairs of programmers, for two separate sessions
- Project used: Lobo browser, written in Java



Pairs performed a series of change tasks:

1. Manually externalize a field
2. Repeat changes by using Arcum code provided
3. Modify Arcum code to add special check
4. Port code from using `StringBuffer` to `StringBuilder`
5. Check the usage of Lobo's logging idiom

Participants reasoned about scattering by writing queries

Problems | Console | Error Log | Fragments

Focus on map entry: InternalSt | Refresh | Transform to: ExternalSt | Transform

Concept	Program Fragment	Resource	Path	Line
attrGet	n.next	List.java	/PartTwo/src/edu/ucsd/stu...	38
attrGet	list.next	ListPrinting.java	/PartTwo/src/edu/ucsd/stu...	14
attrGet	list.next	ListPrinting.java	/PartTwo/src/edu/ucsd/stu...	15
attrGet	n.next	ListUtil.java	/PartTwo/src/edu/ucsd/stu...	7
attrSet	3 fragments matched			

A1: "Maybe it's not something we can actually fix with this because it's not our code, it's a bad library dependence."
A2: "Well what we can do is detect where it happens."

A query can create false confidence

- When a query doesn't match the programmer's intentions it can create false confidence about the properties of the program

A2: "Oh, those are, oh we were looking at the wrong thing. Cool. But now we know there are ones we're not getting too, right, because…"
A1: "[..] it can take various sorts of arguments"

- Defensive approaches to checking queries were not observed

Confusing Definition with Reference

- Even the simple `System.err` to `ErrorLog.getLog()` example creates confusion in the best of programmers:
  - "Why not just use Encapsulate Field?"
- Participants were unclear what Arcum's `Type` type meant:
  - When participants searched for `java.lang.StringBuffer` they were surprised to see only one result listed

9

Confusing Definition with Reference

- Participants wanted matches for `Type` to be:
  - the *uses* of the type (e.g., declarations),
  - not the *definition* of the type (compiled JRE binary)
- This meta-level confusion suggests two possibilities to explore:
  - Arcum's type system can be richer, and explicitly make the distinction:
    - `FieldDeclaration`
    - `FieldUse` (a subtype of `Expr`)
  - Arcum's type system can be relaxed, and use-based

10

Use of reference materials

- Participants needed to know what was returned by `Map`'s `put` method and saw the following documentation in Eclipse:

```
V put(K key, V value)
```

A2: "There's some way that it will give you the type. There you go. You do need to mouse over it."
A1: "It's 'value'."
A2: "So it gives you the value. So in this case we can use it. Just like this one. So we can just put this guy."

- But `put` returns the *previous* value in the map.

11

Reasoning over several implementations

- It wasn't surprising that participants used a before-and-after conceptualization for transformation tasks concerning two different implementations
- Error detection declarations were far more difficult to cope with
  - Not only do you need to think of the correct implementation, but also all possible incorrect ones

B1: "It matched but it didn't tell us anything. So we need to do something that detects the error. So we have to capture this in a variable and check that it's of that form. Or something like that. Or maybe not."

12

Keywords should match conceptions

- The keyword "`realize`" had no meaning for the participants, so "`match`" was the keyword we ultimately decided on
- Similarly for "`require`", which became "`check`"
- Considering keywords in the context of metaphors and models that novice programmers have is a serious point of view.

13

Modularity "Worked"?

- On the decomposition of the refactoring process:

B1: "First let's see if we can just find them, and then if we can replace them."

- But, we found programmers struggled with conceiving of multiple implementations simultaneously.

14