Mining Jungloids: Helping to Navigate the API Jungle

David Mandelin, Lin Xu, Rastislav Bodik (UC Berkeley)
Doug Kimelman (IBM)

Presented By: Andrew Tjang

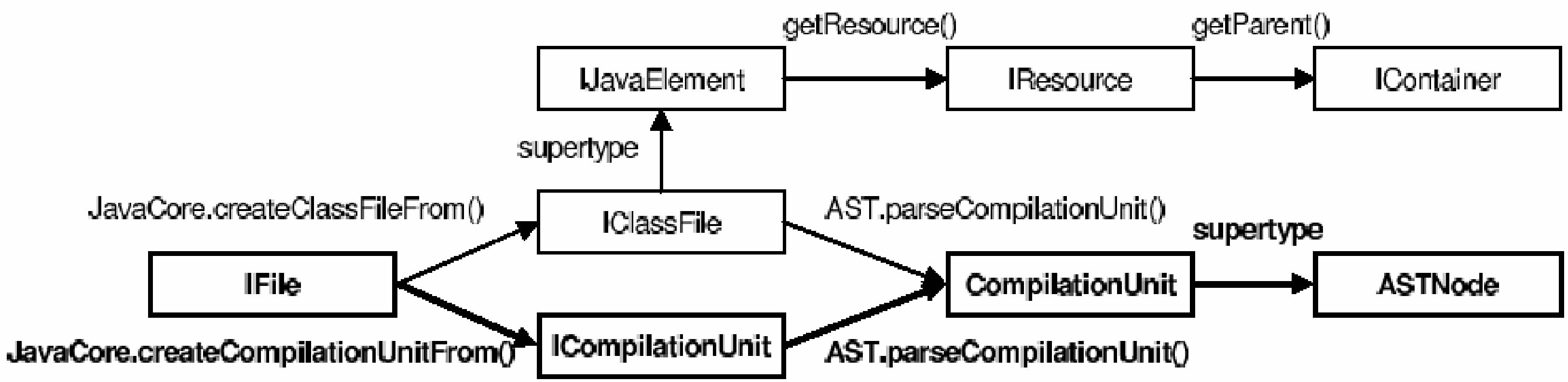
Problem Statement

- APIs allow programmers to reuse code for common s/w tasks
- APIs written with most general purposes in mind
- Reuse makes reuse difficult
- Take J2SE: 21,000 methods in thousands of classes
  - Are javadocs good enough?
- Fine grained method implementations = ease of reuse = hard to use

Introducing Jungloids

- These large APIs sometimes make simple tasks difficult
- Ex:
  - ```
    IFile file = …;
    ICompilationUnit cu =
      JavaCore.createCompilationUnitFrom(file);
    ASTNode ast =
                  AST.parseCompilationUnit(cu, false);
    ```
- Jungloid defined:
  - “a chain of objects and method calls you need to get from something you have to something you need – like a monkey swinging from vine to vine through the jungle”
    ```

Prospector

- Their tool to navigate the jungle
- A search engine to find jungloids
- Given input: source class, target class
- Outputs: a series of jungloids that match constraints

Other applications

- Use Prospector in IDEs to determine correct code path at any given point
  - Determine all classes in scope, and run k queries of type <Ti, Tout>
- K-input jungloids
  - Multiple input classes, with one output class
  - Run prospector successively to find each input

Jungloid Basic Building Blocks

- Method signature
- Field declarations
- Class inheritance declarations
- Form directed graph
  - Nodes: class
  - Edge: method signature

Methods that return Object

- Throws a wrench into producing correct jungloids
- Can downcast to 1 of 50,000 classes at compile time
- Programmers usually look at examples to determine the correct jungloid

Examples

- Fix (somewhat) the Object downcast problem
- Programmers usually use grep to find relevant examples
  - Grep unaware of context and code structure
- Relevant code may span many methods/classes

Combining Signatures and Examples

- Combine best of both worlds
  - Signatures – simple and general
  - Examples – more precise, catch downcasts
- Jungloid graph combines these
  - Each path represents jungloid
  - Examples converted into paths and added
- Use standard graph algorithms to solve queries beginning at Tin and ending at Tout

Elementary Jungloids

- Fields: if class T declares a field U f,
  - J.f : T → U
- Instance Methods: if a class T declares and instance method with no arguments
  - J.m(): T → U
- Static Methods: class C declares static method with one non primitive parameter
  - C.m(J): T → U
- Constructors: Constructor with exactly one non primitive parameter
  - U(J): T → U
- Supertype Conversion: if T is subtype of U
  - J:T → U

Jungloids

- All elementary jungloids are jungloids
- If E1[J]: T→ U is a jungloid, and E2{J): U→V is a elementary jungloid, then E2[E1[J]]:T→V is a jungloid (transitive property)

Example

- Recall:
  - ```
    IFile file = …;
    ICompilationUnit cu =
      JavaCore.createCompilationUnitFrom(file);
    ASTNode ast =
              AST.parseCompilationUnit(cu, false);
    ```
- Composed of 3 elementary Jungloids:
  - Static method jungloid
  - Static method jungloid
  - Supertype jungloid??
    ```

Signature Graph



Non-useful Jungloids

- Fails in user context: returns null or throws exception
- Fails for all program inputs the user plans to use
- Returns normally, but doesn't satisfy user's intent

Ranking Jungloids

- Put short jungloids at top of result list
  - Programmer not likely to write a jungloid w/ 300 method calls
  - Shorter jungloids likely to return normally
- Shortest arbitrarily chosen jungloid in result set satisfied programmers intent in 9/10 times
- Presented the top k matches

Limitations

- input types as Object
- String as intermediate type
- May produce unwanted jungloids
- Downcasts

How to handle downcasts

- Create new downcast elementary jungloid
- Can't add all downcast edges based on signatures
  - makes for many unwanted jungloids
  - makes for short jungloids (ranking problems)
- Ideally: Include downcasts that do not fail runtime type check (ClassCastException)
  - can be approximated by adding based on examples
  - obtain corpus of code
  - extract casts (mining)
  - make extracted info more general

Casting

```
protected IJavaObject getObjectContext() {
  IDebugView view = theDebugView;
  ISelection s = view.getViewer().getSelection();
  IStructuredSelection sel =
    (IStructuredSelection) s;
  Object selection = sel.getFirstElement();
    IJavaVariable var = (IJavaVariable) selection;
  ...
}
```

Figure 4: An API usage example containing a cast found in a corpus of sample client code.

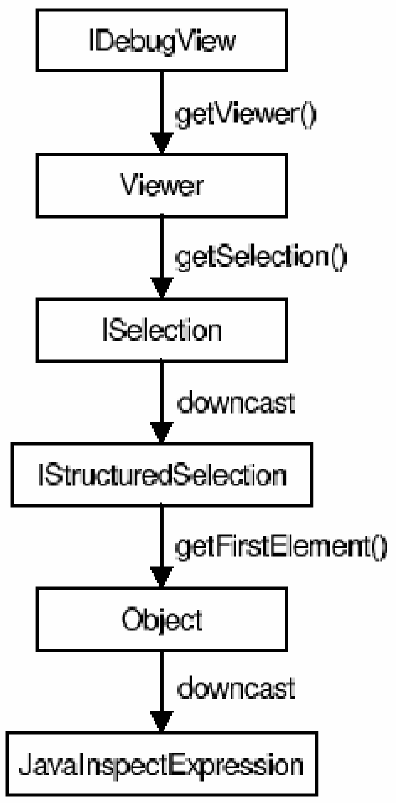


Figure 5: Example jungloid mined from code in Figure 4.

More casting

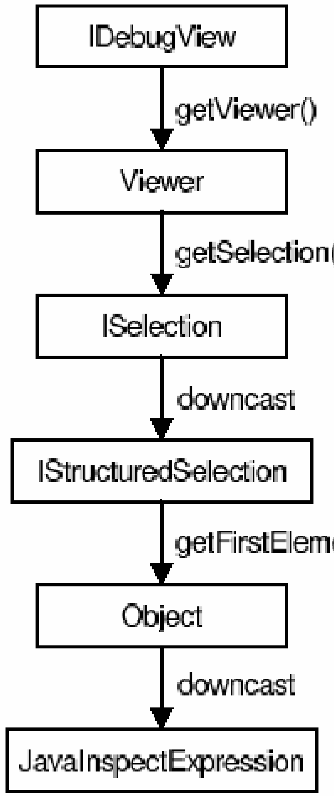


Figure 5: Example jungloid mined from code in Figure 4.

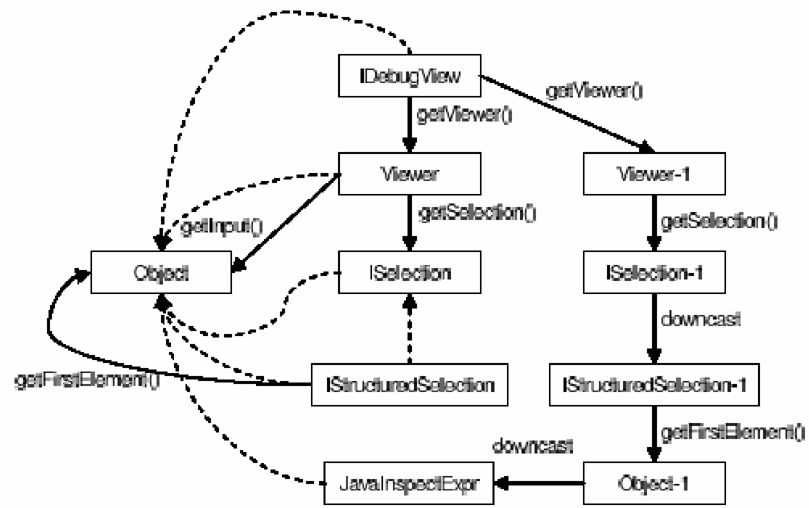


Figure 6: Part of the jungloid graph, formed from signatures and an example. All nodes have supertype conversion edges leading to `Object`, but some have been omitted for legibility.

How to Mine

- Create signature graph
- Prefix truncate to generalize
- Merge with signature graph

Extraction Algorithm

- Construct data dependence graph of corpus
- Methods treated as expressions
  - (can be entered, but not done)
- Find all cast expressions, and extract backward acyclic paths
- Convert to example jungloid

Prefix Truncation

- Casts with unnecessary prefixes should be truncated
  - may be too specific and prevent mining
- Views jungloids as a set of stings
- Remove layers not needed to distinguish between two different down casts

Truncating

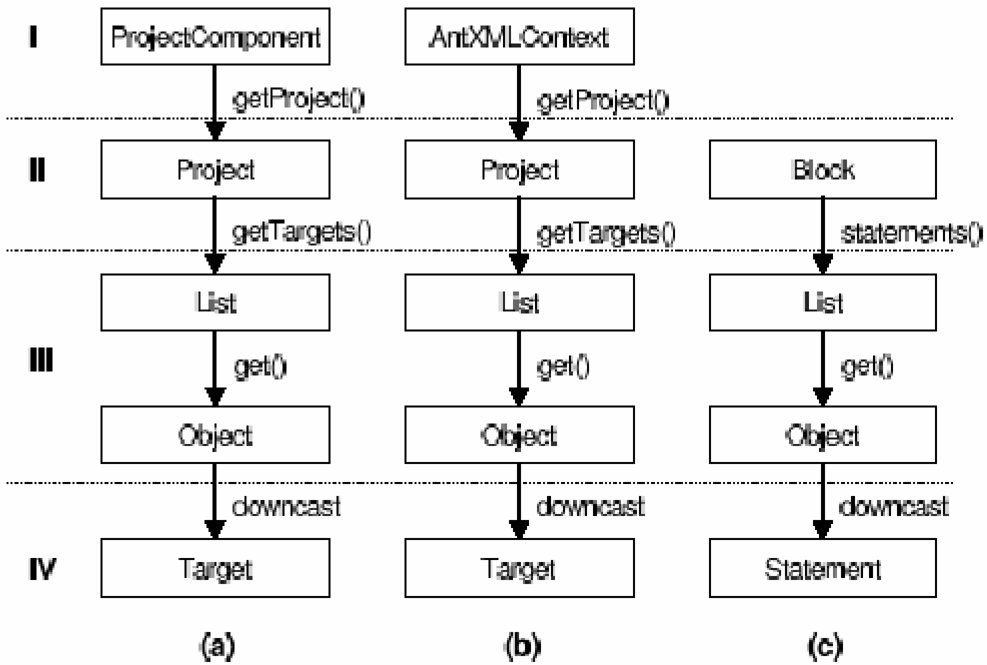


Figure 9: Example jungloids with unneeded prefixes, shown with dashed lines. The list returned by `Project.getTargets()` contains `Target` objects, regardless of the methods called to obtain the `Project`.

Mining Accuracy

- Completeness and Soundness
- Completeness - Any valid jungloid can be found
- Soundness - the search only returns valid jungloids
- Valid jungloid - jungloids that return normally for at least one context and program input

How to achieve mining accuracy?

- Corpus must approach certain ideal properties:
- Corpus Coverage Property
  - Corpus contains all API usage scenarios containing casts that return normally (At least once)
  - the larger the better
- Corpus Cast Property
  - The corpus never throws ClassCastExpression
  - Contains no dead-code jungloids with casts

Experiments

- Performed two experiments:
  - Test Prospector's query processing accuracy (finds the right jungloids for solving problems)
  - Study performance on developers.(do developers solve problems

Accuracy Results

Table 1: Query processing accuracy test results, showing the rank of the desired solution jungloid in PROSPECTOR results for 20 real world queries.

Jungloid rank	Number of queries	Fraction of queries	Cumulative fraction
1	10	.50	.50
2	3	.15	.65
3	3	.15	.80
4	1	.05	.85
Not found	3	.15	1.00

User Results

Table 2: User study results for six programming problems. The *Prospector* and *Baseline* columns show the number of users that successfully completed the problem out of the number that attempted the problem.

Problem	Success		Average Time (min)	
	Prospector	Baseline	Prospector	Baseline
1	2/2	5/5	4.5	8.4
2	0/2	6/6	-	11.16
3	5/5	1/2	13.4	20
4	1/2	0/2	25	-
5	1/3	0/2	20	-
6	1/2	4/4	9	12

Conclusions

- Automatic analysis can yield jungloids
- Jungloids can ease the burden of figuring out APIs
- In practice, it seemed to be a useful tool for developers.