JavaOne™

JavaOne™

Custom Static Code Analysis

Jan Lahoda
Software Developer, NetBeans

Motivation

- Big projects tend to have project-specific antipatterns
- In NetBeans, rather then (for `Project p;`): `p.getLookup().lookup(ProjectInformation.class)`
- One should do: `ProjectUtils.getProjectInfo(p)`
- Is there a way to guard against the anti-patterns?

Custom Static Code Analysis

- There is: use NetBeans' language for declarative refactorings?
  - Can use just the "find" part to implement code analysis
  - Rule:
```
$prj.getLookup().lookup(ProjectInformation.class)
 :: $prj instanceof Project
=> ProjectUtils.getProjectInfo($prj);;
```
- How to check violations?
  - Netbeans shows them in the editor&Source/Inspect…
  - Why not check for them also during continuous build?

Standalone runner

- "Standalone" runner for the custom rules: https://bitbucket.org/jlahoda/jackpot30/wiki/StandaloneJackpot
- Can also run the standard NetBeans warnings (hints)
- Can even perform the changes
- Contains bindings to ant and maven
- Still work in progress

Custom Declarative Refactorings Language

- Allows to define refactorings (almost) declaratively
- Custom Java-like DSL
- Makes API/structure/relationship changes much easier
- Can check for antipatterns
- Can also be used to ask questions like
  How many clients are calling this method (in a specified context)?

Custom Declarative Refactorings Language

History

- Project Jackpot: founded 2000 to improve IDEs and the way devs develop (by Tom Ball, Michael Van De Vanter, James Gosling)
- Incl. code transformation engine
  - Structure/AST based
  - Working prototype existed
  - Transformations: in Java or a custom declarative language
- Its write model adopted by NetBeans in 2006 (NB 6.0)
- Declarative language for transformations revived for NetBeans IDE 7.1 project “Jackpot 3.0”

The Language

File Format

- Basic rule format:

```
   <source-pattern> :: <conditions>
=> <target-pattern> :: <conditions>
=> <target-pattern> :: <conditions>
;;
```

- Any number of such rules in a file
- Place the rule in a `.hint` file into `META-INF/upgrade`

Patterns

Basics

- Java expression, statement(s), class, variable, method
- Identifiers starting with `$` represent variables: a tree node will be bound to them: `$1`, `$lock`, etc.
- Identifiers starting and ending with $ consume any number of tree nodes:
  - `java.util.Arrays.asList($param)`
  - `java.util.Arrays.asList($params$)`

Patterns

Repeated Variables

- A variable used multiple times: all must be the same
- `$var = $var` ("assignment to itself")

```
private int a, b;
a = a;
this.a = a;
a = b;
```

Patterns

Special Forms

- Statement: `$statement;` (more: `$statements$;`)
- 0 or 1: `if ($cond) $then; else $else$;`
- Modifiers: `$mods$ $type$ $variableName;`
- Multiple catches: `try {} $catches$ finally {}`
- More special forms for specific uses

Conditions

- Three types:
  - Language – `instanceof`, `otherwise`
  - Standard (predefined) – method invocations
  - Custom
- Condition result can be negated (`!`)
- `&&` works on condition results

Conditions

Language Conditions

- `$variable instanceof <type>`
  - True if expression bound to `$variable` of type `<type>`
- `otherwise`
  - Valid only on fixes
  - True when no other fix available for that rule

Conditions

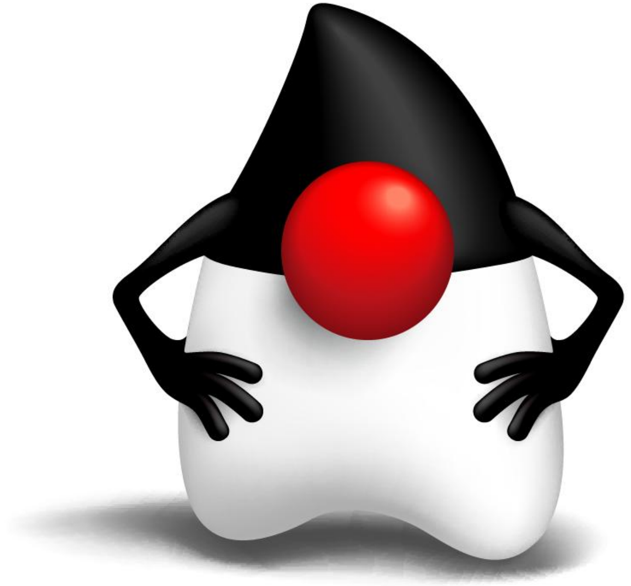
Standard Conditions

- Like method invocations:

```
hasModifier($variable, PRIVATE)
```

- Conditions for:
  - Pattern matching (parentMatches, matchesAny, ...)
  - Referred element inspection (hasModifier, elementKindMatches)
  - Occurrence location (inClass, inPackage)
  - Assorted checks (isNullLiteral, sourceVersionGE, referencedIn)

Demo – Jackpot 3.0 in Hudson



See Also

- Much more details on the language in:
  Custom Declarative Refactoring (TUT3702)
  tomorrow, 3:00 PM, Hilton - Continental Ballroom 1/2/3
- Static Analysis with Javac Tutorial (TUT4285)
  tomorrow, 12:30 PM, Hilton - Continental Ballroom 1/2/3
- Language description:
  http://wiki.netbeans.org/JavaDeclarativeHintsFormat
- Examples:
  https://bitbucket.org/jlahoda/jackpot30-demo-examples/

Conclusion

- Use the NetBeans’ custom refactorings to check for antipatterns
- Can check for violations during continuous build
- Can also produce standard NetBeans warnings

Q&A

JavaOne™