# Lab. 3. Static code analysis

## Work goal

Evaluate selected software quality using static analysis methods and tools.

## Prerequisites:

The student has chosen some software to test. It could be one of previously created by student or any other open source one with source code available. It is recommend choosing software that is implemented in Java programming language, but students are free to choose software implemented in any other language.

## Background

Static code analysis is software code review without executing it in order to find bugs in its implementation. During analysis process team of reviewers of software reads program code and tries to identify the places of code that could contain bugs. If review is performed by person, it’s called code review. That analysis is performed by program – static code analysis. The static analysis is performing according a long checklist that could be automated in an analysis tool. Some of checks are like this in order to identify issues in the code:

1. Threads synchronization (missing locks, releases)
2. Uninitialized variables, null pointers. No checks for null pointers
3. Unreachable code paths.
4. SQL injection
5. Missing constructors, missing destructors
6. Invalid access to class fields.
7. Data overflow.
8. Unhandled exceptions
9. …

The static analysis tools have a long list of checks implanted as rules. The tool parses the software under review source code and checks if piece of code matches all rules. If the rule is not satisfied the tool flags a portion of code for developer to inspect.

Here is an example of Null pointer rule:

1. If we have the following code:

|  |
| --- |
| 1 Person person = map.get("bob");  2 if (person != null) {  3 person.updateAccessTime();  4 }  5 String name = person.getName(); |

1. The static analysis tool would indicate the possible null pointer issue ant the line 5. As getter getName could be executed on the object that, possibly, is null, though the code at line 3 is correct.

The built-in static analysis rules are not always sufficient for software code analysis. Some checks could be not so command and used only in one organization. The developer/tester can also extend static analysis tool by adding their own rules based on specific domain or business requirements.

## Lab work tasks:

1. Code review for the selected on the 1st lab software:
   1. Review several classes,
   2. check for possible issues locations,
   3. Use the code review checklists.
   4. Document possible places in the document (table: source file, class, method, line, unsatisfied checklist rule, comment).
2. Static code analysis for the selected on the 1st lab software:
   1. Select static code analysis tool: possible chooses are JTest, .NET Test, FindBugs, FxCop, etc.
   2. Execute code analysis tool on the selected software.
   3. Design and implement one static code analysis rule.
   4. Execute code analysis tool on the selected software sing newly implemented rule.
   5. Document detected issues in the document (table: source file, class, method, line, unsatisfied checklist rule, comment).

## Lab work defence:

1. Created rule implementation be presented, explained, and committed into version control system at: [http://proin.ktu.lt/svn/]
2. The static analysis and code review report should be presented in the wiki at [http://proin.ktu.lt/wki/]
3. The student should be ready to answer to questions relating to the work.

## The questions for students to look into (defence preparation):

1. What’s the difference between testing ant static analysis?
2. What is false positive?
3. If some static analysis/review rules are not satisfied, does that mean that software is defective?
4. Can the static analysis be used instead of testing?

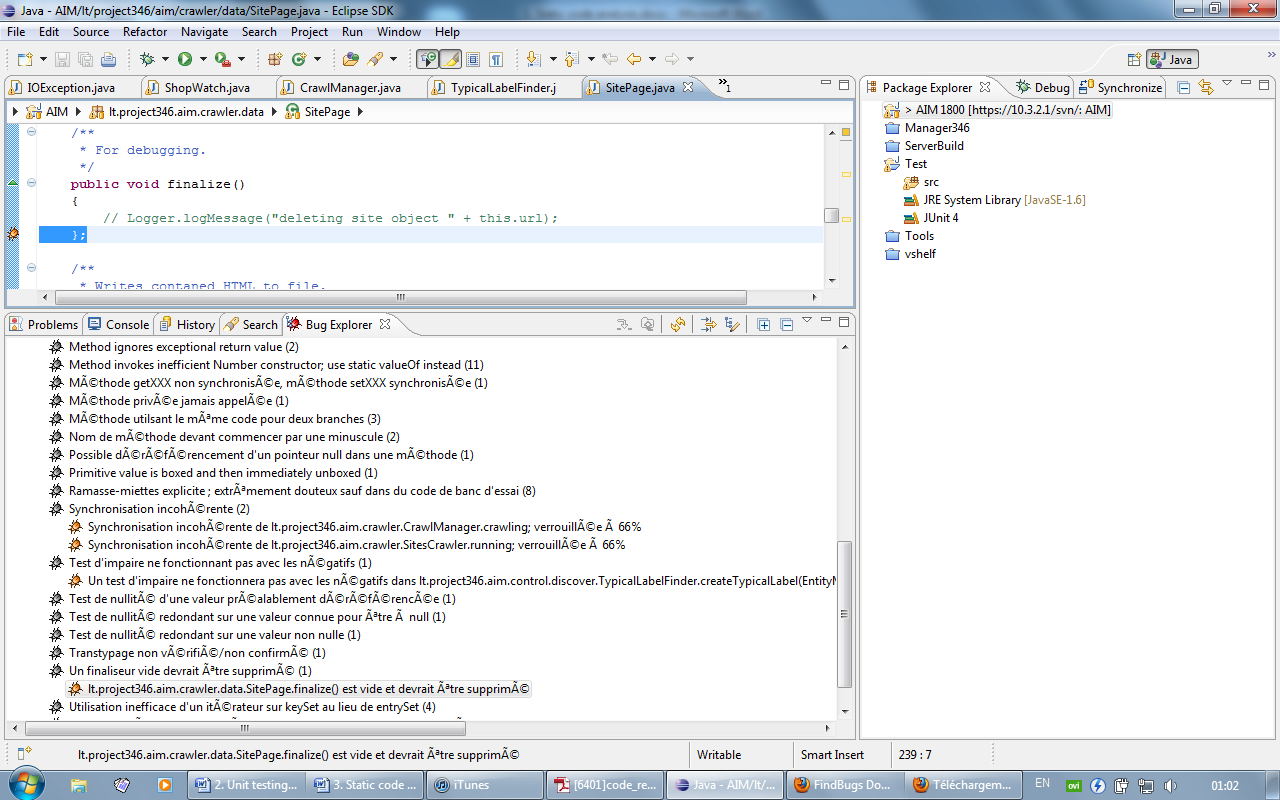
## References:

1. Brian Chess, Jacob West. Secure Programming with Static Analysis, Addison-Wesley Professional, 624p.

## Sample code review checklist:

1. Structure
   1. Does the code conform to coding standards?
2. Documentation
   1. Are all comments consistent with the code?
3. Variables
   1. Are all variables properly defined with meaningful, consistent, and clear names?
   2. Do all assigned variables have proper type consistency or casting?
4. Arithmetic Operations
   1. Does the code avoid comparing floating-point numbers for equality?
   2. Does the code systematically prevent rounding errors?
   3. Are divisors tested for zero or noise?
5. Loops and Branches
   1. Does every case statement have a default?
   2. Does the code in the loop avoid manipulating the index variable or using it upon exit from the loop?
6. Defensive Programming
   1. Are indexes, pointers, and subscripts tested against array, record, or file bounds?
   2. Are all output variables assigned?
   3. Is every memory allocation de-allocated?
   4. Are files checked for existence before attempting to access them?
   5. Are all files and devices are left in the correct state upon program termination?

## Static analysis using Find Bugs tool:

1. Preparation
   1. Install Eclipse development tool
   2. Import software under test project into eclipse (same as in lab 2)
   3. Install find bugs plug-in into eclipse (help->install new software, enter <http://findbugs.cs.umd.edu/eclipse> address and install all components)
2. Launching static code analysis on software project.
   1. Select on navigator window software under test project
   2. Select find bugs->find bugs from context menu.
   3. Observe static analysis results in Bugs explorer window
   4. 
   5. In this example the tree shows static analysis rules, and after expanding the tree node the code locations with possible issues are displayed.
   6. In this example the static analysis tool found unnecessary finalizer declared.
3. Creating a custom static analysis rule:
   1. Define rule: if there is a setter method in a class, there should be getter method as well.
   2. Each static testing framework has specific rules defining interfaces. In this example the FindBugs tool is used.
   3. In order to create static rule create a new Java project in the eclipse: File->new Project->java project.
   4. Name the new project “GettersDetector” – the static analysis rules in find bugs are called detectors.
   5. Link find bugs library files: select libraries tab in project->properties->java build path from context menu.
   6. Add required jar files: findbugs.plugin.jar, usually located at “eclipse installation” / /plugins/edu.umd.cs.findbugs.plugin.eclipse\_1.3.9.20090821/lib directory.
   7. Crate the detector class :

|  |
| --- |
| package my.detector;  import org.apache.bcel.classfile.JavaClass;  import org.apache.bcel.classfile.Method;  import edu.umd.cs.findbugs.BugInstance;  import edu.umd.cs.findbugs.BugReporter;  import edu.umd.cs.findbugs.bcel.OpcodeStackDetector;  public class GettersDetector extends OpcodeStackDetector {  private BugReporter bugReporter;  public GettersDetector(BugReporter bugReporter) {  this.bugReporter = bugReporter;  };  public void visit(JavaClass someObj) {  // iterate through all class methods  for (Method method : someObj.getMethods()) {  // do we have a getter?  if (method.getName().startsWith("get")) {  String setterName = "set" + method.getName().substring(3);  boolean found = false;  // iterate through all class methods while searching for the  // setter  for (Method setterMethod : someObj.getMethods()) {  // do we have a getter?  if (setterMethod.getName().equals(setterName)) {  found = true;  break;  }  }  // so no setter was found  if (false == found) {  // report the issue  bugReporter.reportBug(new BugInstance(this, "GETSET\_BUG",  *NORMAL\_PRIORITY*).addClassAndMethod(this)  .addString("").addSourceLine(this));  }  }  }  super.visit(someObj);  }  public void sawOpcode(int arg0) {  }  } |

* 1. Create detector descriptor named findbugs.xml and place it into project root directory:

|  |
| --- |
| <FindbugsPlugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:noNamespaceSchemaLocation="findbugsplugin.xsd"  pluginid="lt.ktu.soften.findbugs.plugingetset"  provider="Getter Setter rule"  website="http://soften.ktu.lt">  <Detector class="my.detector.GettersDetector" reports="GETSET\_BUG" />  <BugPattern type="GETSET\_BUG" abbrev="GS" category="CORRECTNESS"/>  </FindbugsPlugin> |

* 1. This xml file instructs find bug tool how to find detector class.
  2. Then create messages.xml (displaying discovered issues descriptions) at the project root directory.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <MessageCollection xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:noNamespaceSchemaLocation="messagecollection.xsd">  <Plugin>  <ShortDescription>FindBugs get/set rule Plugin</ShortDescription>  <Details>Provides detectors as part of the FindBugs detector</Details>  </Plugin>  <Detector class="my.detector.GettersDetector">  <Details>Checks if there exists matching get/set methods pair</Details>  </Detector>  <BugPattern type="GETSET\_BUG">  <ShortDescription>Getter should contain matching setter</ShortDescription>  </BugPattern>  <BugCode abbrev="GS">gettet/setter</BugCode>  </MessageCollection> |

* 1. Export project as jar file, File->export->Java->JAR file. Make sure that xml files are included.
  2. Copy exporter jar file into find bugs directory (like at : “eclipse installation” / /plugins/edu.umd.cs.findbugs.plugin.eclipse\_1.3.9.20090821/plugins )
  3. Re-launch eclipse.
  4. Repeat actions from the 3rd step.