CS3213 Project – Week 9

Static Analysis | 16-03-2022

- ❏ Recap: Program Slicing
- ❏ Practical Introduction to Static Analysis
- ❏ Assignment 8 - ”Code + Presentation”

Reducing the program

Recap

- **Program slicing** is the computation of the ***subset of program statements*** (→ the program **slice**).
- The **program slice** includes statements that may affect the values at some point of interest (~ the **slicing criterion**)
- Concept: Select a line to be considered and hide all irrelevant lines.

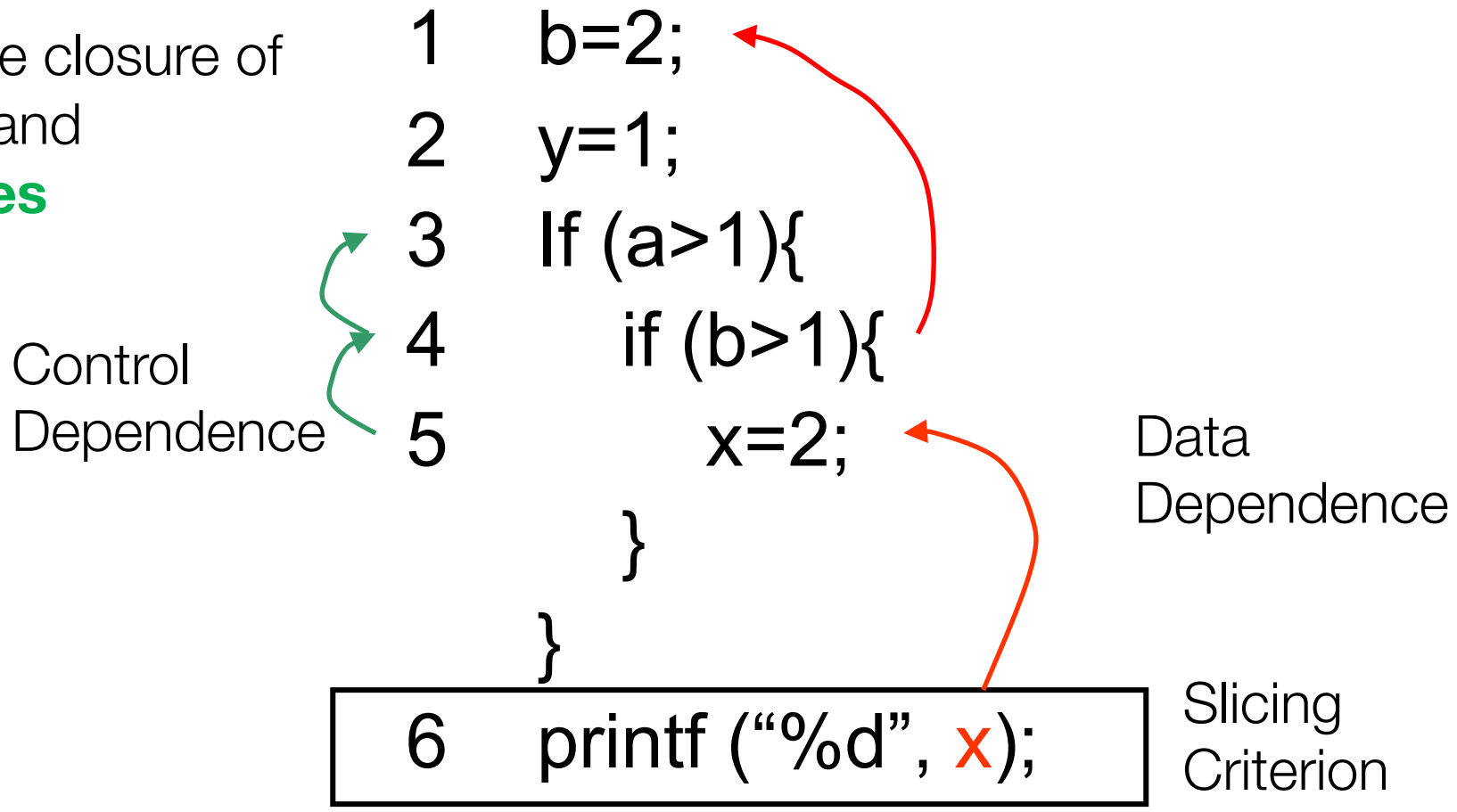
- Dynamic Slicing: slice for a particular program execution
  → dynamic dependencies
- Static Slicing → static dependencies
  - What is **affected** by this slicing criterion? (forward)
  - What is **influenced** the value of this variable? (backward)

Dynamic Slicing

Recap

- Slice backward from the erroneous output of the program
- Dynamic slice includes the closure of
  - **Data dependencies** and
  - **Control dependencies**

For our test case with **a=2**, the value of variable **x** printed in line 6 is **unexpected**.



Exercise: Dynamic Slicing

Input: -1

```
1   int x = read(x);
2   if (x < 0) {
3     y = x + 1;
4     z = x + 2;
    } else {
5     if (x == 0) {
6       y = x + 3;
7       z = x + 4;
      } else {
8       y = x + 5;
9       z = x + 6;
      }
    }
10  printf("%d", y);
11  printf("%d", z);
```

Slicing Criterion

Can you specify the resulting dynamic slice?

Exercise: More Dynamic Slicing

Input: 1

```
1   int x = read();
2   int z = 0;
3   int y = 0;
4   int i = 1;
5   while (i <= x) {
6     z = z + y;
7     y = y + 1;
8     i = i + 1;
    }
9   printf("%d", z);
```

Slicing Criterion

Can you specify the resulting dynamic slice?

Journal of Programming Languages **3** (1995) 121–189

A survey of program slicing techniques

FRANK TIP*

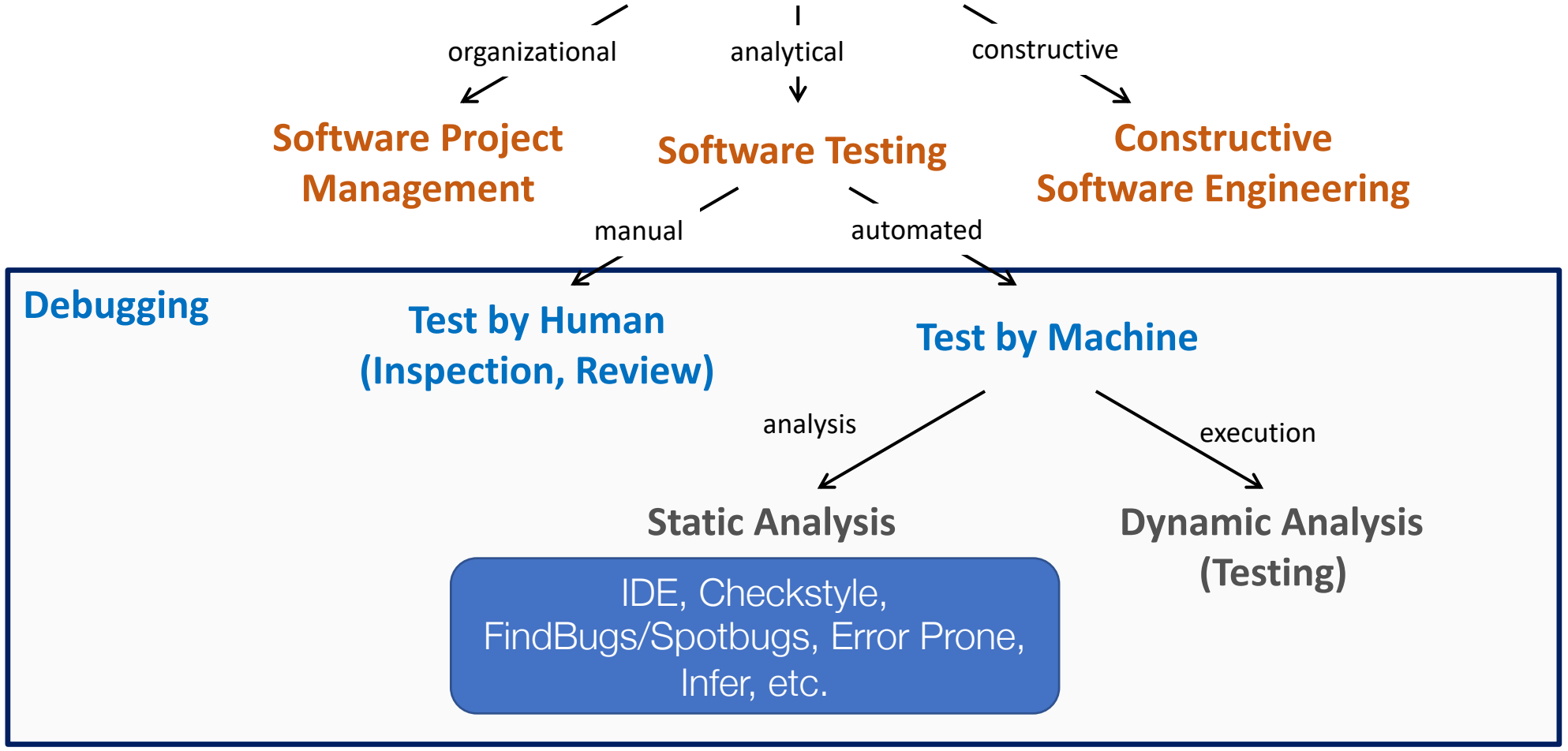
IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, USA

A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a *slicing criterion*, and is typically specified by a location in the program in combination with a subset of the program's variables. The task of computing program slices is called *program slicing*. The original definition of a program slice was presented by Weiser in 1979. Since then, various slightly different notions of program slices have been proposed, as well as a number of methods to compute them. An important distinction is that between a *static* and a *dynamic* slice. Static slices are computed without making assumptions regarding a program's input, whereas the computation of dynamic slices relies on a specific test case. This survey presents an overview of program slicing, including the various general approaches used to compute slices, as well as the specific techniques used to address a variety of language features such as procedures, unstructured control flow, composite data types and pointers, and concurrency. Static and dynamic slicing methods for each of these features are compared and classified in terms of their accuracy and efficiency. Moreover, the possibilities for combining solutions for different features are investi-

Frank Tip, "A survey of program slicing techniques", Journal of Programming Languages, vol. 3, pages 121-189, 1995.
https://www.franktip.org/pubs/jpl1995.pdf

Any more questions for program slicing?

Software Quality Assurance



Software Quality Assurance

Software quality assurance includes organizational, constructive, and analytical measures to provide confidence that the software meets the required quality.

- **Organizational Measures**: Introduction of programming guidelines.
- **Constructive Measures**: Tools for implementing the guidelines, e.g., a program code formatter.
- **Analytical measures**: Audits/Reviews with the guidelines to detect violations.

Testing as Quality Assurance

Software testing is a measure for software quality assurance.

- **Organizational Measures**: Specifications for the test.
- **Constructive Measures**: Avoiding errors by using appropriate languages and techniques.
- **Analytical measures**: Detect errors.

Tool Support for Software Quality Assurance

checkstyle

https://checkstyle.sourceforge.io
https://plugins.jetbrains.com/plugin/1065-checkstyle-idea

https://spotbugs.github.io
https://plugins.jetbrains.com/plugin/14014-spotbugs
https://plugins.jetbrains.com/plugin/3847-findbugs-idea

https://pmd.github.io
https://plugins.jetbrains.com/plugin/1137-pmdplugin

- Static code analysis to check **programming guidelines**
- own configuration may be necessary
- integration into build process or directly into IDE
- https://checkstyle.sourceforge.io
  https://plugins.jetbrains.com/plugin/1065-checkstyle-idea
- **Checks**
  - https://checkstyle.sourceforge.io/checks.html
  - E.g., AvoidStarImport, EmptyCatchBlock, EqualsHashCode, FallThrough, ReturnCount
  - You can add more checks:
    https://checkstyle.sourceforge.io/writingchecks.html

checkstyle

JTodoFrame.java

```
167     */
168    private JButton createLoadButton() {
169        JButton loadButton = new JButton(bundle.getString("JTodoFrame.open")
170                + " ...", openIcon);
171        loadButton.addActionListener(new ActionListener() {
172            @Override
173            public void actionPerformed(ActionEvent e) {
174                /*
175                 * Show the file chooser and if a file is selected load it using
176                 * the application facade.
177                 */
178                if (fileChooser.showOpenDialog(JTodoFrame.this) == JFileChooser.APPROVE_OPTION) {
179                    try {
180                        File file = fileChooser.getSelectedFile();
181                        Application.instance.load(file.getAbsolutePath());
182                        SwingUtilities.invokeLater(new Runnable() {
183                            @Override
184                            public void run() {
185                                todoList.updateUI();
186                            }
187                        });
188                    } catch (ApplicationException exc) {
189                        JOptionPane.showMessageDialog(
190                                JTodoFrame.this,
191                                "<html><b>"
192                                        + bundle.getString("JTodoFrame.error")
193                                        + ":</b><p>" + exc.getMessage()
194                                        + "</p></html>",
195                                bundle.getString("JTodoFrame.error"),
196                                JOptionPane.ERROR_MESSAGE);
197                    }
198                }
199            }
```

bad naming

line too long

More examples in the lab.

SpotBugs

- SpotBugs is a fork of FindBugs (which is now an abandoned project)
- Static analysis tool for finding errors in Java programs based on **bug patterns**.
- ***"A bug pattern is a code idiom that is often an error."***
  (~Anti-Pattern, Code Smell)
- based on **Java bytecode**
- https://spotbugs.github.io
  https://plugins.jetbrains.com/plugin/14014-spotbugs
  https://plugins.jetbrains.com/plugin/3847-findbugs-idea
- But Patterns / Descriptions
  https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html

SpotBugs

Violations of recommended and essential coding practice.

Category:	Bad Practice
Examples:	❑ ***Dm: Method invokes System.exit(…)*** Invoking System.exit shuts down the entire Java virtual machine. This should only been done when it is appropriate. Such calls make it hard or impossible for your code to be invoked by other code. Consider throwing a RuntimeException instead. ❑ ***HE: Class defines equals() and uses Object.hashCode()*** This class overrides equals(Object), but does not override hashCode(), and inherits the implementation of hashCode() from java.lang.Object (which returns the identity hash code, an arbitrary value assigned to the object by the VM). Therefore, the class is very likely to violate the invariant that equal objects must have equal hashcodes.

https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#bad-practice-bad-practice

SpotBugs

Probable bug - an apparent coding mistake resulting in code that was probably not what the developer intended.

Category:	Correctness
Examples:	❑ ***NP: Method with Optional return type returns explicit null*** The usage of Optional return type (java.util.Optional or com.google.common.base.Optional) always means that explicit null returns were not desired by design. Returning a null value in such case is a contract violation and will most likely break client code. ❑ ***Eq: equals method always returns false*** This class defines an equals method that always returns false. This means that an object is not equal to itself, and it is impossible to create useful Maps or Sets of this class. More fundamentally, it means that equals is not reflexive, one of the requirements of the equals method.

https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#correctness-correctness

SpotBugs

Code that is not necessarily incorrect but may be inefficient.

Category:	Performance
Examples:	❑ ***IIO: Inefficient use of String.indexOf(String)*** This code passes a constant string of length 1 to String.indexOf(). It is more efficient to use the integer implementations of String.indexOf(). f. e. call myString.indexOf('.') instead of myString.indexOf(".") ❑ ***WMI: Inefficient use of keySet iterator instead of entrySet iterator*** This method accesses the value of a Map entry, using a key that was retrieved from a keySet iterator. It is more efficient to use an iterator on the entrySet of the map, to avoid the Map.get(key) lookup.

https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#performance-performance

SpotBugs

A use of untrusted input in a way that could create a remotely exploitable security vulnerability.

Category:	Security
Examples:	❑ ***Dm: Hardcoded constant database password*** This code creates a database connect using a hardcoded, constant password. Anyone with access to either the source code or the compiled code can easily learn the password. ❑ ***HRS: HTTP cookie formed from untrusted input*** This code constructs an HTTP Cookie using an untrusted HTTP parameter. If this cookie is added to an HTTP response, it will allow a HTTP response splitting vulnerability. See http://en.wikipedia.org/wiki/HTTP_response_splitting for more information.

https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#security-security

SpotBugs

Category:	Dodgy Style
Examples:	❑ ***UwF: Field not initialized in constructor but dereferenced without null check*** This field is never initialized within any constructor, and is therefore could be null after the object is constructed. Elsewhere, it is loaded and dereferenced without a null check. This could be either an error or a questionable design, since it means a null pointer exception will be generated if that field is dereferenced before being initialized. ❑ ***PZLA: Consider returning a zero length array rather than null*** It is often a better design to return a length zero array rather than a null reference to indicate that there are no results (i.e., an empty list of results). This way, no explicit check for null is needed by clients of the method.

https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#dodgy-code-style

PMD

- PMD is a **source code** analyzer.
- It finds **common programming flaws** like unused variables, empty catch blocks, unnecessary object creation, etc.
- It supports Java, JavaScript, Salesforce.com Apex and Visualforce, PLSQL, Apache Velocity, XML, XSL.
- Additionally, it includes Code Clone Detection with CPD, the copy-paste-detector.
- CPD finds duplicated code in Java, C, C++, C#, Groovy, PHP, Ruby, Fortran, JavaScript, PLSQL, Apache Velocity, Scala, Objective C, Matlab, Python, Go, Swift and Salesforce.com Apex and Visualforce.
- https://pmd.github.io
  https://plugins.jetbrains.com/plugin/1137-pmdplugin

PMD – Java Rules

- **Best Practices**: Rules which enforce generally accepted best practices, e.g.,

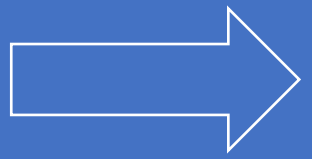
  `JUnitTestsShouldIncludeAssert`
  JUnit tests should include at least one assertion. This makes the tests more robust, and using assert with messages provide the developer a clearer idea of what the test does.

- **Code Style**: Rules which enforce a specific coding style.
- **Design**: Rules that help you discover design issues.
- **Documentation**: Rules that are related to code documentation.
- **Error Prone**: Rules to detect constructs that are either broken, extremely confusing or prone to runtime errors.
- More: **Multithreading**, **Performance**, **Security**

---

https://pmd.github.io/pmd-6.43.0/pmd_rules_java.html

What is the difference?



Will be discussed in the lab tomorrow.

Conclusion

- Static Analysis based techniques can be easily integrated into the software development workflow!
- Note: quality **cannot** be introduced by testing…
  → analytical vs constructive methods!

Next Week (Project-Part) – Week 10: **Implementation**

- Implementation (Clean Code)
- Documentation & Reusability
- Assignment 9: Final Report