TVVS - Test, Verification and Validation of Software

static testing

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Static testing

- Static testing techniques rely on the manual examination (**reviews**) and automated analysis (**static analysis**) of the code or other project documentation.
- Reviews are a way of testing software work products (including code) and can be performed well before dynamic test execution.
- Defects detected during reviews early in the life cycle are often much cheaper to remove than those detected while running tests (e.g., defects found in requirements).

Reviews

- Benefits of reviews include
  - early defect detection and correction
  - development productivity improvements
  - reduced development timescales
  - reduced testing cost and time
  - lifetime cost reductions
  - fewer defects and improved communication
  - reviews can find omissions, for example, in requirements, which are unlikely to be found in dynamic testing
- Compared to dynamic testing, static techniques find causes of failures (defects) rather than the failures themselves.

The values of static analysis

- Early detection of defects prior to test execution.
- Early warning about suspicious aspects of the code or design, by the calculation of metrics, such as a high complexity measure.
- Identification of defects not easily found by dynamic testing.
- Detecting dependencies and inconsistencies in software models, such as links.
- Improved maintainability of code and design.
- Prevention of defects, if lessons are learned in development.

Reviews are cost effective

- 10 times reduction in fault reaching test, testing cost reduced by 50% to 80%
  - [Freedman & Weinberg, Handbook of Walkthroughs, Inspections & Technical Reviews]
- Reduce faults by a factor of 10
  - [Yourdon, Structured Walkthroughs]
- 25% reduction in schedules, remove 80% - 95% of faults each stage, 28 times reduction in maintenace cost, many others
  - [Gilb & Graham, Software Inspection]

Costs of reviews

- Rough guide: 5%-15% of development effort
  - Half a day a week is 10%
- Effort required for reviews
  - Planning (by leader / moderator)
  - Preparation / self study checking
  - Meeting
  - Fixing / editing / follow-up
  - Recording & analysis of statistics / metrics
  - Process improvement (should!)

Reviews

- Typical defects that are easier to find in reviews than in dynamic testing are:
  - deviations from standards,
  - requirement defects,
  - design defects,
  - insufficient maintainability and
  - incorrect interface specifications.

What can be Reviewed/Inspected?

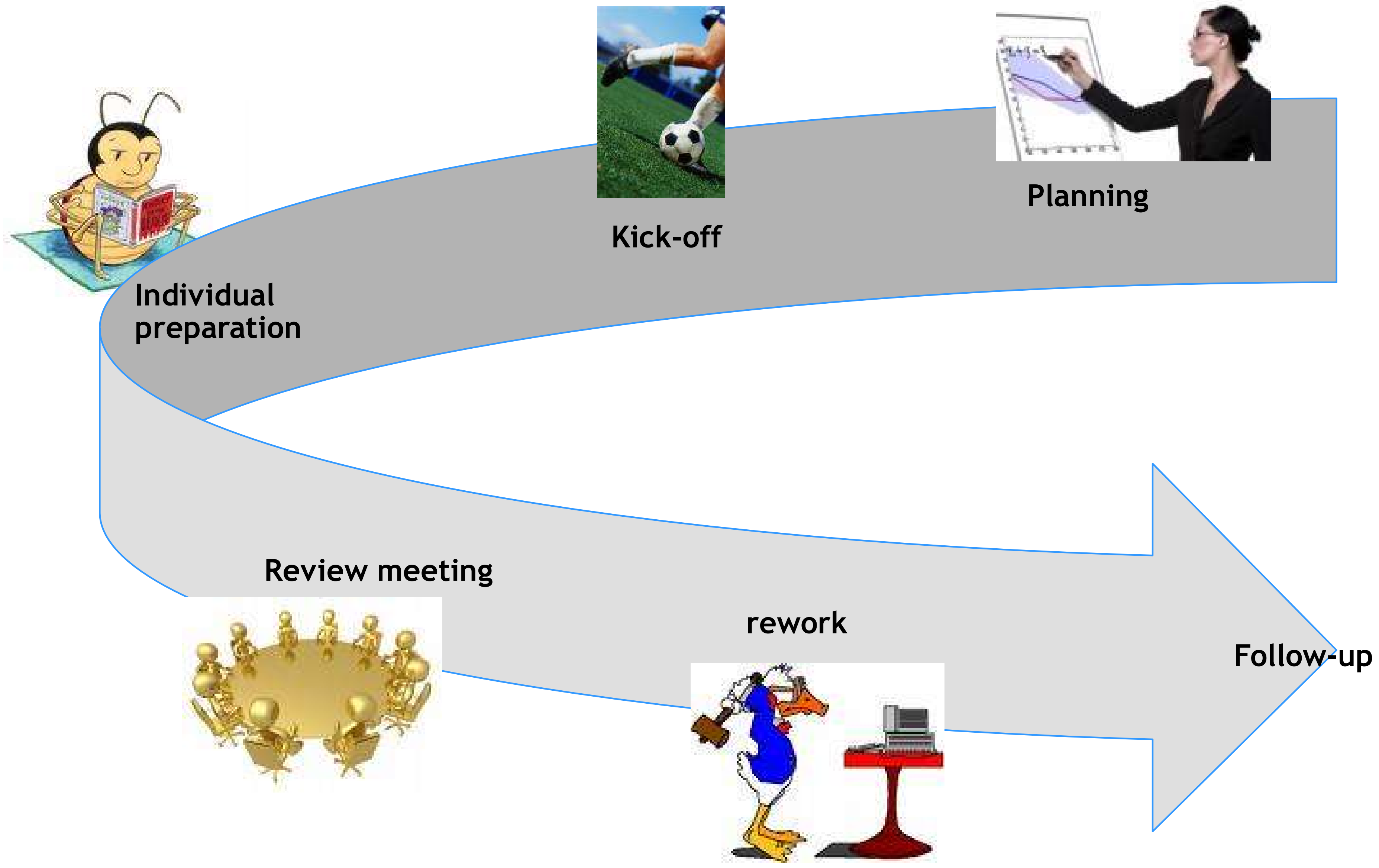
Anything written down can be inspected

- Policy, strategy, business plans, marketing or advertising materials, contracts
- System requirements, feasibility studies, acceptance test plans
- Test plans, test designs, test cases, test results
- System designs, logical & physical
- Software code
- User manuals, procedures, training material

Reviews

- The different types of reviews vary from very informal (e.g., no written instructions for reviewers) to very formal (i.e., well structured and regulated). The formality of a review process is related to factors such as
  - the maturity of the development process,
  - any legal or regulatory requirements,
  - or the need for an audit trail.
- The way a review is carried out depends on the agreed objective of the review (e.g.,
  - find defects,
  - gain understanding,
  - or discussion and decision by consensus).

Phases of a formal review



Phases of a formal review

- **Planning**: selecting the personnel, allocating roles; defining the entry and exit criteria for more formal review types (e.g., inspection); and selecting which parts of documents to look at.
- **Kick-off**: distributing documents; explaining the objectives, process and documents to the participants; and checking entry criteria (for more formal review types).
- **Individual preparation**: work done by each of the participants on their own before the review meeting, noting potential defects, questions and comments.

Phases of a formal review

- **Review meeting**: discussion or logging, with documented results or minutes (for more formal review types). The meeting participants may simply note defects, make recommendations for handling the defects, or make decisions about the defects.
- **Rework**: fixing defects found, typically done by the author.
- **Follow-up**: checking that defects have been addressed, gathering metrics and checking on exit criteria (for more formal review types).

Reviews: Deliverales

- Changes (edits) in review product
- Change requests for source documents (predecessor documents to product being reviewed / Inspected)
- Process improvement suggestions
  - To the review / inspection process
  - To the development process which produced the product just reviewed / inspected
- Metrics

Reviews

Pitfalls (they don’t always work!)

- Lack of training in the technique (especially inspection, the most formal)
- Lack of or quality of documentation – what is being reviewed / inspected
- Lack of management support – “lip service” want them done, but don’t allow time for them to happen in project schedules
- Failure to improve processes (gets disheartening just getting better at finding the same thing over again)

Roles and responsabilities

- **Manager**: decides on the execution of reviews, allocates time in project schedules and determines if the review objectives have been met.
- **Moderator**: the person who leads the review of the document or set of documents, including planning the review, running the meeting, and follow-up after the meeting. If necessary, the moderator may mediate between the various points of view and is often the person upon whom the success of the review rests.
- **Author**: the writer or person with chief responsibility for the document(s) to be reviewed.

Roles and responsabilities

- **Reviewers**: individuals with a specific technical or business background (also called checkers or inspectors) who, after the necessary preparation, identify and describe findings (e.g., defects) in the product under review. Reviewers should be chosen to represent different perspectives and roles in the review process, and should take part in any review meetings.
- **Scribe** (or recorder): documents all the issues, problems and open points that were identified during the meeting.

Types of reviews

- A single document may be the subject of more than one review. If more than one type of review is used, the order may vary. For example, an informal review may be carried out before a technical review, or an inspection may be carried out on a requirements specification before a walkthrough with customers. The main characteristics, options and purposes of common review types are:
    - Informal reviews
    - Walkthrough
    - Technical review
    - Inspection

Informal reviews

Widely viewed as useful and cheap (but no one can prove it!). A helpful first step for chaotic organizations.

- No formal process;
- there may be pair programming or a technical lead reviewing designs and code;
- optionally may be documented;
- may vary in usefulness depending on the reviewer;
- **main purpose**: inexpensive way to get some benefit.

Walkthrough

Author guides the group through a document and his or her thought processes, so all understand the same thing, consensus on change to make

- meeting led by author;
- scenarios, dry runs, peer group;
- open-ended sessions;
- optionally a pre-meeting preparation of reviewers, review report, list of findings and scribe (who is not the author);
- may vary in practice from quite informal to very formal;
- **main purposes**: learning, gaining understanding, defect finding.

Technical review

Includes peer and technical experts, no management participation. Normally documented, fault-finding. Can be rather subjective

- documented, defined defect-detection process that includes peers and technical experts;
- may be performed as a peer review without management participation;
- ideally led by trained moderator (not the author);
- pre-meeting preparation;
- optionally the use of checklists, review report, list of findings and management participation;
- may vary in practice from quite informal to very formal;
- **main purposes:** discuss, make decisions, evaluate alternatives, find defects, solve technical problems and check conformance to specifications and standards.

Inspection

Formal individual and group checking, using sources and standards, according to generic and specific rules and checklists, using entry and exit criteria, leader must be trained & certified, metrics required.

- led by trained moderator (not the author);
- usually peer examination;
- defined roles;
- includes metrics;
- formal process based on rules and checklists with entry and exit criteria;
- pre-meeting preparation;
- inspection report, list of findings;
- formal follow-up process;
- optionally, process improvement and reader;
- **main purpose**: find defects.

Reviews

- Walkthroughs, technical reviews and inspections can be performed within a peer group – colleagues at the same organizational level. This type of review is called a "**peer review**".

Success factors for reviews

- Each review has a clear predefined objective.
- The right people for the review objectives are involved.
- Defects found are welcomed, and expressed objectively.
- People issues and psychological aspects are dealt with (e.g., making it a positive experience for the author).
- Review techniques are applied that are suitable to the type and level of software work products and reviewers.

Success factors for reviews

- Checklists or roles are used if appropriate to increase effectiveness of defect identification.
- Training is given in review techniques, especially the more formal techniques, such as inspection.
- Management supports a good review process (e.g., by incorporating adequate time for review activities in project schedules).
- There is an emphasis on learning and process improvement.

Static analysis by tools

- The objective of static analysis is to find defects in software source code and software models. Static analysis is performed without actually executing the software being examined by the tool; dynamic testing does execute the software code. Static analysis can locate defects that are hard to find in testing. As with reviews, static analysis finds defects rather than failures. Static analysis tools analyze program code (e.g., control flow and data flow), as well as, generated output such as HTML and XML.

Static analysis by tools

Descendent from compiler technology

- A compiler statically analyses code and “knows” a lot about it, e.g., variable usage, finds syntax faults
- Static analysis tools extend this knowledge
- Can find unreachale code, undeclared variables, parameter type mis-matches, uncalled functions & procedures, array bound violations, etc.

Static analysis

Typical defects discovered by static analysis tools include:

- referencing a variable with an undefined value;
- inconsistent interface between modules and components;
- variables that are never used;
- unreachable (dead) code;
- programming standards violations;
- security vulnerabilities;
- syntax violations of code and software models.

Static analysis

- Static analysis tools are typically used
  - **by developers** (checking against predefined rules or programming standards) before and during component and integration testing, and
  - **by designers** during software modeling.
- Static analysis tools may produce a large number of warning messages, which need to be well managed to allow the most effective use of the tool.
- Compilers may offer some support for static analysis, including the calculation of **metrics**.

Data flow analysis

This is study of program variables

- Variable defined* when a value is stored into it
- Variable used when the stored value is accessed
- Variable is undefined before it is defined or when it goes out of scope

```
x = y + z
if a>b then read(s)
```

x is defined, y and z are used

a and b are used, s is defined

*defined should not be confused with declared

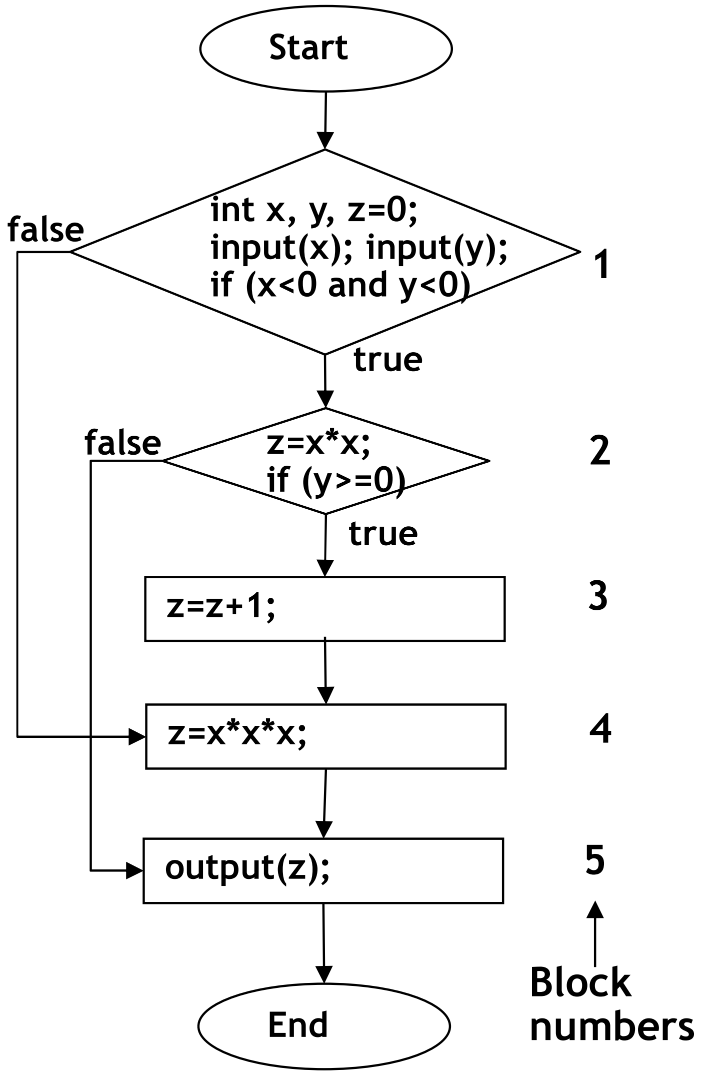
Data flow analysis

```
n:=0
read(x)
n:=1
while x>y do
    begin
        read(y)
        write(n*y)
        x:=x-n
end
```

Data flow anomaly: n is re-defined without being used

Data flow fault: y is used before it has been defined (first time around the loop)

Control flow analysis



- Nodes not accessible from start node
- Infinite loops
- Multiple entry to loops
- Whether code is well structured, i.e., reducible
- Whether code conforms to a flowchart grammar
- Any jumps to undefined labels
- Any labels not jumped to
- Cyclomatic complexity and other metrics

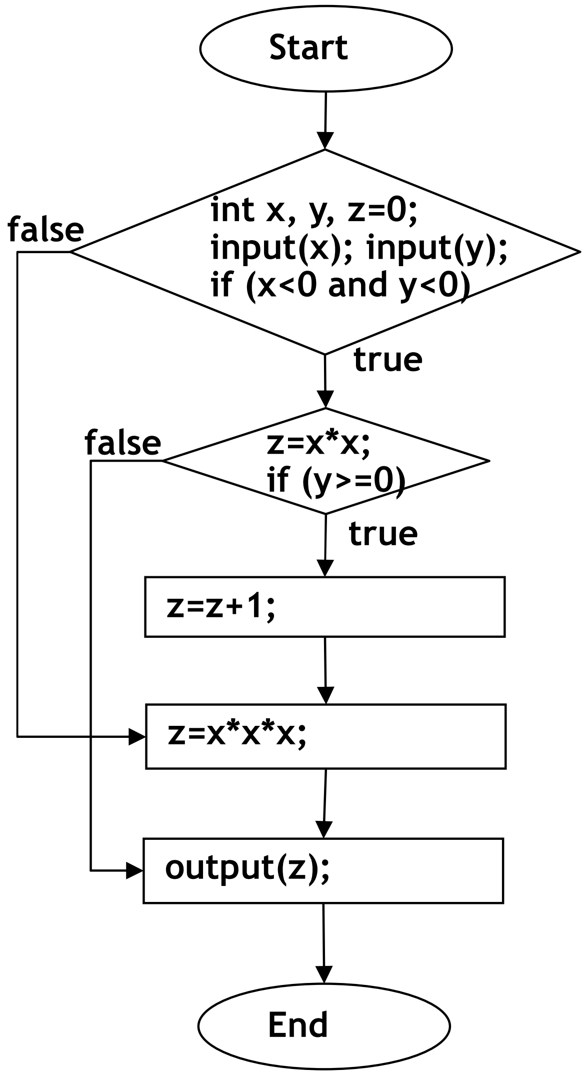
Unreacheable code example

- Macro definitions (different for different platforms the code runs on)

```
Buffsize: 1000
Mailboxmax: 1000
    if Buffsize < Mailboxmax then
        Error-Exit
    endif
```

- Static analysis finds the then clause unreachable, so will flag a fault

Cyclomatic complexity



- Cyclomatic complexity is a measure of the complexity of a flow graph
  - (and therefore the code that the flow graph represents)
- The more complex the flow graph, the greater the measure
- It can most easily be calculated as
  - Complexity = number of dicisions + 1 or
  - Complexity = Number of edges – number of node + 2

Other static metrics

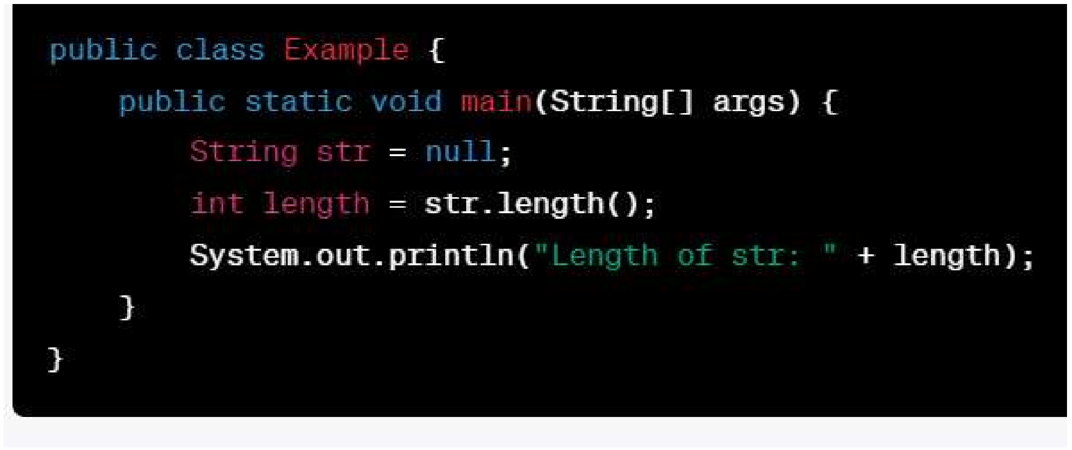
- Lines of code (LOC)
- Operands & operators (Halstead’s metrics): count operators such as “+”, “-”, “/”, and “x”, but this measure is not much used now.
- Structural fan-in & fan-out: modules with high fan-in are found at the bottom of hierarchies or in libraries where they are frequently called. Modules with high fan-out are typically at the top of hierarchies, because they call to many modules (e.g., the main menu)
- Nesting levels: relate to how deeply nested statements are within other IF statements.
- Function calls
- OO metrics: inheritence tree depth, number of methods, coupling & cohesion

Example 1

Static Analysis Report

Project: MyJavaApp
Analysis Date: 2023-09-24
Tool: StaticAnalyzerY

Summary:
- Total Files Analyzed: 1
- Total Issues Found: 1
- Issues by Severity:
- High: 1



```
public class Example {
    public static void main(String[] args) {
        String str = null;
        int length = str.length();
        System.out.println("Length of str: " + length);
    }
}
```

File: Example.java

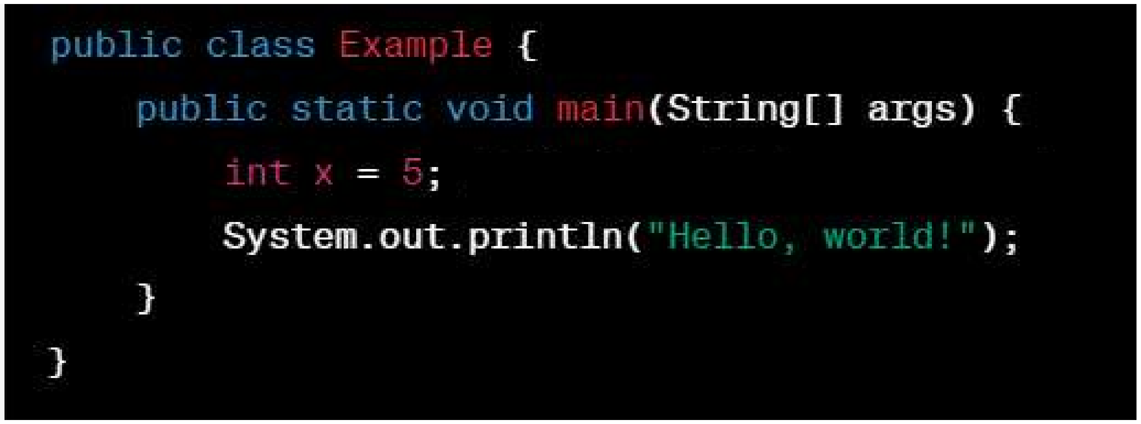
Issue 1 (High):
- Description: Null Pointer Dereference
- Location: Line 4, Column 21
- Recommendation: Check if the reference 'str' is null before invoking 'length()'.

Example 2

Static Analysis Report

Project: MyJavaApp
Analysis Date: 2023-09-24
Tool: StaticAnalyzerX

Summary:
- Total Files Analyzed: 1
- Total Issues Found: 1
- Issues by Severity:
  - Low: 1



```
public class Example {
    public static void main(String[] args) {
        int x = 5;
        System.out.println("Hello, world!");
    }
}
```

File: Example.java

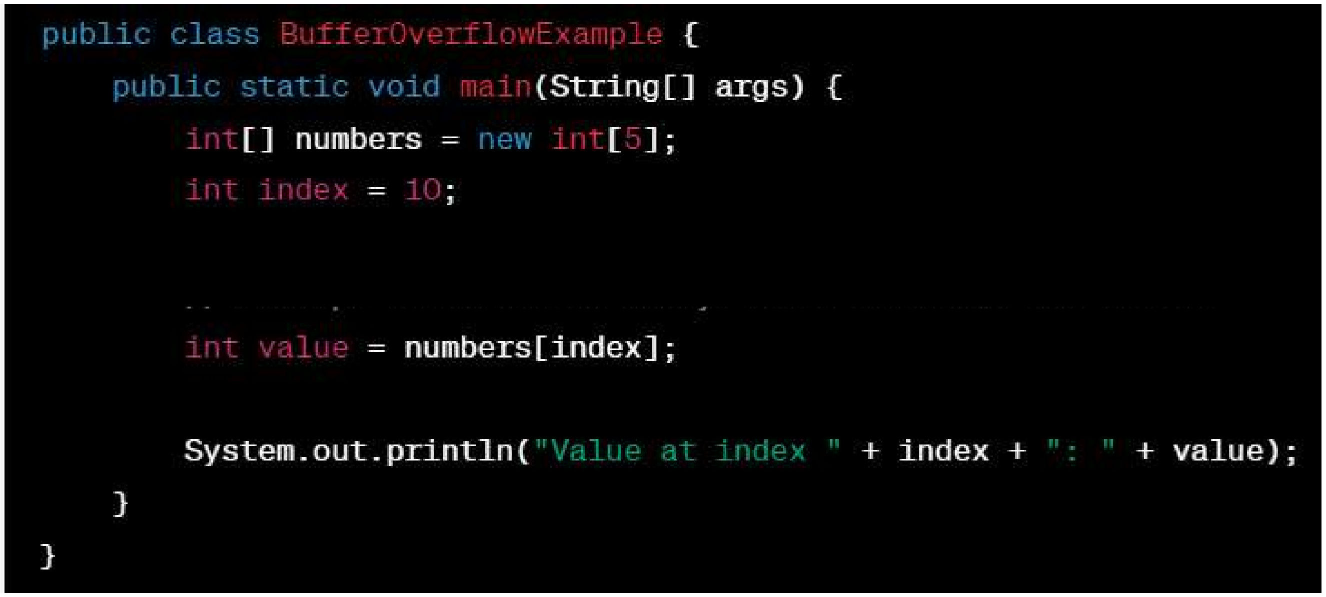
Issue 1 (Low):
  - Description: Unused Variable
  - Location: Line 3, Column 9
  - Recommendation: Remove or refactor the unused variable 'x'.

Example 3

Static Analysis Report

Project: MyJavaApp
Analysis Date: 2023-09-24
Tool: StaticAnalyzerZ

Summary:
- Total Files Analyzed: 1
- Total Issues Found: 1
- Issues by Severity:
  - High: 1



```java
public class BufferOverflowExample {
    public static void main(String[] args) {
        int[] numbers = new int[5];
        int index = 10;


        int value = numbers[index];

        System.out.println("Value at index " + index + ": " + value);
    }
}
```

File: BufferOverflowExample.java

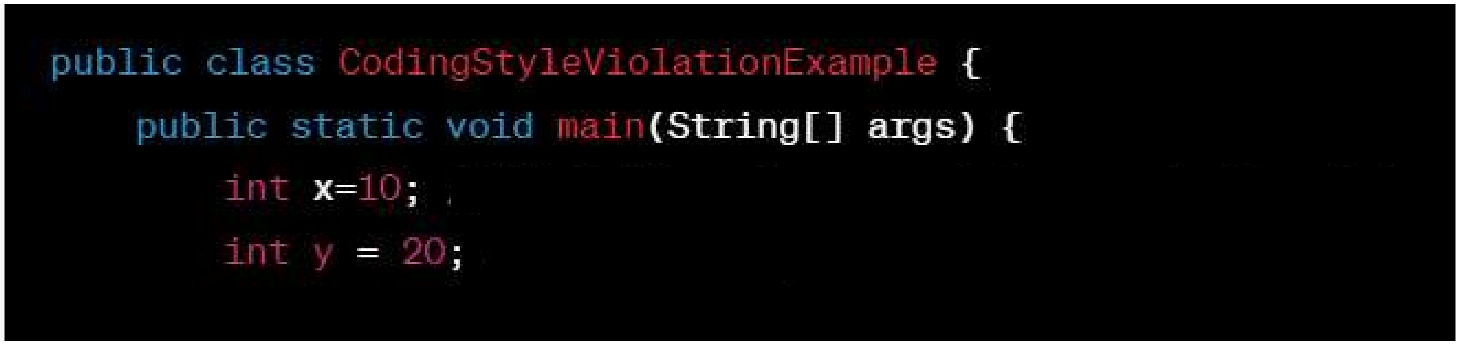
Issue 1 (High):
   - Description: Potential Buffer Overflow
   - Location: Line 7
   - Recommendation: Ensure that the array index 'index' is within the valid range before accessing the array element.

Example 4

Static Analysis Report

Project: MyJavaApp
Analysis Date: 2023-09-24
Tool: StaticStyleChecker

Summary:
- Total Files Analyzed: 1
- Total Issues Found: 2
- Issues by Severity:
  - Low: 2



```
public class CodingStyleViolationExample {
    public static void main(String[] args) {
        int x=10;
        int y = 20;
```

---

File: CodingStyleViolationExample.java

---

Issue 1 (Low):
   - Description: Coding Style Violation (Variable Declaration)
   - Location: Line 3
   - Recommendation: Use spaces before and after '=' in variable declarations for better readability.

Example 5

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class SQLInjectionExample {
    public static void main(String[] args) {
        String userInput = “Ana' OR '1'='1";
        String sqlQuery = "SELECT * FROM users WHERE username = '" + userInput + "'";

        try {
            // Establish a database connection
            Connection connection = DriverManager.getConnection(
                                    "jdbc:mysql://localhost:3306/mydb",
                                    “username", "password");

            // Create a SQL statement
            Statement statement = connection.createStatement();

            // Execute the SQL query
            ResultSet resultSet = statement.executeQuery(sqlQuery);

            // Process the query results...
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

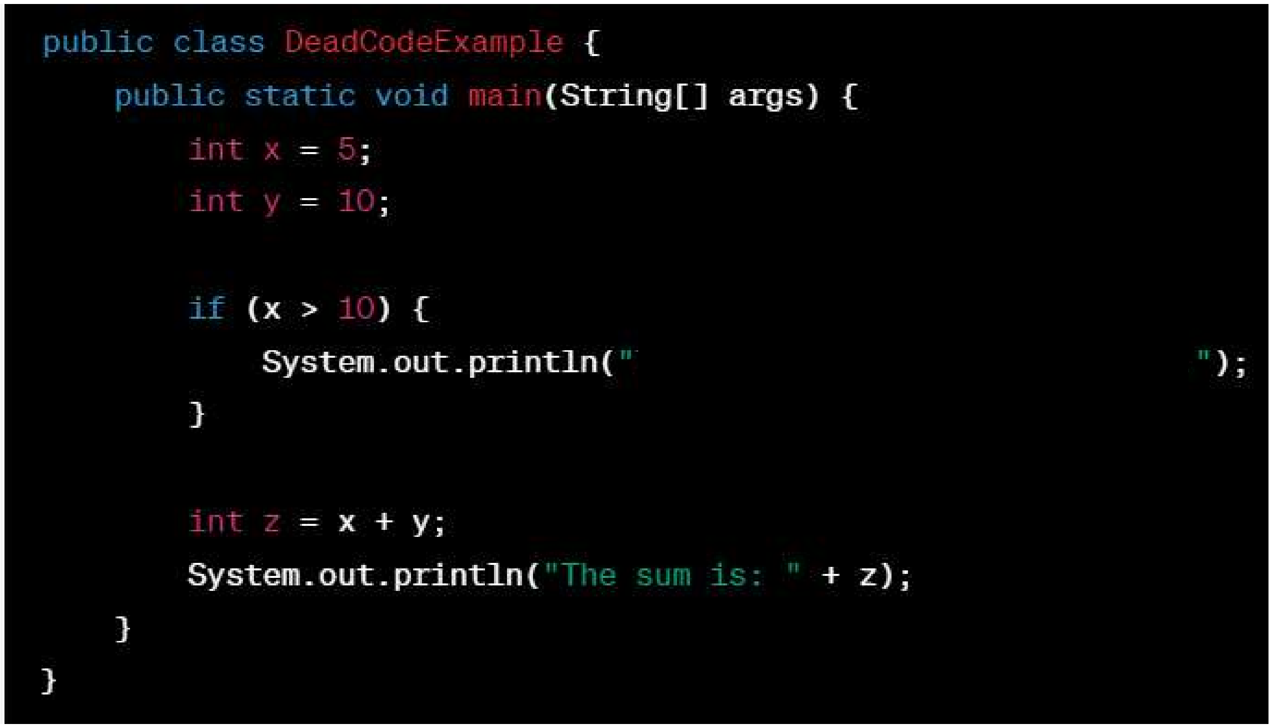
```
Static Analysis Report

Project: MyJavaApp
Analysis Date: 2023-09-24
Tool: StaticAnalyzerX

Summary:
- Total Files Analyzed: 1
- Total Issues Found: 1
- Issues by Severity:
  - High: 1

----------------------------------------------
File: SQLInjectionExample.java
----------------------------------------------
Issue 1 (High):
   - Description: Security Vulnerability (SQL Injection)
   - Location: Line 8
   - Recommendation: Use parameterized queries or prepared statements to construct SQL queries with
user input to prevent SQL injection attacks.
```

Example 6

Static Analysis Report

Project: MyJavaApp
Analysis Date: 2023-09-24
Tool: StaticAnalyzerX

Summary:
- Total Files Analyzed: 1
- Total Issues Found: 1
- Issues by Severity:
  - Low: 1



```
public class DeadCodeExample {
    public static void main(String[] args) {
        int x = 5;
        int y = 10;

        if (x > 10) {
            System.out.println("                              ");
        }

        int z = x + y;
        System.out.println("The sum is: " + z);
    }
}
```

File: DeadCodeExample.java

Issue 1 (Low):
  - Description: Dead Code
  - Location: Line 7
  - Recommendation: Remove the dead code block to improve code readability and maintainability.