Faculty of Computer Systems & Software Engineering

Formal methods.

Static Analyses

Vitaliy Mezhuyev

What is Static Analysis

- Static analysis is the process of examining source code prior to compilation (and execution)

- Static analysis is used for:
  - Detecting unsafe programming constructs and coding defects
  - Increase effectiveness of testing
  - Check quality aspects such as maintainability, reliability, understandability and complexity
  - Coding standards compliance issues
  - Checking correspondence to the best programming practices

Automated Static Analysis Tools

Tools for analyzes of programs without executing it

- Not depend on having good test cases or even any test cases
- Generally, doesn't know what your software is *supposed to do*
- Looks for violations in programming (e.g. Null Pointer Exception)
- Good at finding problems on untested paths
- Not a replacement for testing

Automatic static analysis

- 60% of the software faults that were found in released software products could have been detected by means of static analysis
- What is static analysis?
  - Analysis of software artifacts e.g., requirements or code, carried out without execution of these software artifacts
- Objective of static analysis
  - To reveal defects or software parts that are defect-prone
  - To derive metrics in order to measure and prove the quality of the object
- How is static analysis done?
  - Static analysis tools known as static analyzers
- Objects to be analyzed
  - Formal documents that must follow a certain formal structure

FindBugs example

JDK1.6.0-b105
- 379 correctness warnings
- at least 213 of these are serious issues that should be fixed

Google's Java codebase
- over a 6 month period, using various versions of FindBugs
- 1,127 warnings
- 807 filed as bugs
- 518 fixed in code

Can you Find the Bug?

```
if (listeners == null)
   listeners.remove(listener);
```

JDK1.6.0, b105, sun.awt.x11.XMSelection
lines 243-244

What is the Bug here?

```
/** Construct a WebSpider */
public WebSpider() {
WebSpider w = new WebSpider();
}
```

What is the Bug here?

```
public String foundType() {
return this.foundType();
}
```

Why do Bugs Occur?

Nobody is perfect

- Everyone makes syntax errors, but fortunately a compiler catches them
- What about other bugs?

Common types of errors:

- Typos (using wrong operators, forgetting parentheses or brackets, etc.)
- Misunderstood language features, API methods etc.
- Misunderstood OOP principles (e.g. polymorphism)

Compiler as Static Analysis Tool

- Detection of violation of the programming language syntax; reported as an error or warning
- Further information and other checks
  - Generating a cross reference list of the different program elements (e.g. variables, functions)
  - Checking for correct data type usage by data and variables in programming languages with strict typing
  - Detecting undeclared variables
  - Detecting code that is not reachable
  - Detecting overflow or underflow of field boundaries
  - Checking of interface consistency
  - Detecting the use of all labels as jump start or jump target

Stages of Automated Static Analysis

- Syntactic Analysis
- Data Use Analysis
- Control Flow Analysis
- Interface Analysis
- Program Slicing
- Data Flow Analysis
- Path Analysis

Stages of Static Analysis

- Syntactic analysis
  - Missing statements i.e. switch statements

```
switch (expr)
{   case c1: statement1; break; // do these if expr == c1

    case c2: statement2;          // do these if expr == c2

    case c3: statement3;        // we also do c3, because break is missing
}
```

  - Coding standards

Data Use Analysis

- Aim is to identify data flows that do not conform to good programming practices, e.g. variables are not initialized before they are used; inactive code.
- A purely symbolic form of analysis, i.e. no specific data values are considered.
- Based upon a number of relationships between variables and expressions.
- Process involves annotating a program flow graph with each data object definition (D), usage (U) and elimination (E).
- Analysis involves flow graph traversal.

Control Flow Analysis

- This method also involves translating the program into a flow graph.
- Aim is to detect poorly structured code, e.g. multiple exits from a loop, dead code etc.
- By repeated reduction an inaccessible and non-terminated code is identified

Program Slicing

- Program slicing involves focusing on a particular subset of variables within a given program.
- The parts of the program that are relevant to the subset of variables denotes a **program slice**.
- Some applications:
  - Program testing & re-testing: provides focus with respect to test case design and the selection of regression tests.
  - Program comprehension: slicing provides a useful aid to understanding code where no documentation exists.

Static analysis tools

- Who and when used static analysis tools?
  - Developers
  - Before and during component or integration testing
    - To check if guidelines or programming conventions are adhered to
  - During integration testing : analyze adherence to interface guidelines
- What are produced by static analysis tools?
  - List of warnings and comments
    - **Syntax violation**
    - **Deviation from conventions and standards**
    - **Control flow anomalies**
    - **Data flow anomalies**
    - **Metrics**

Program Slicing Example

Program

```
read(X);
read(Y);
Q := 0;
R := X;
while R >= Y do
        begin
        R := R - Y;
        Q := Q + 1
end;
print(Q);
print(R);
```

A Program Slice

```
read(X);
read(Y);
R := X;
while R >= Y do
        begin
        R := R - Y;
end;
print(R);
```

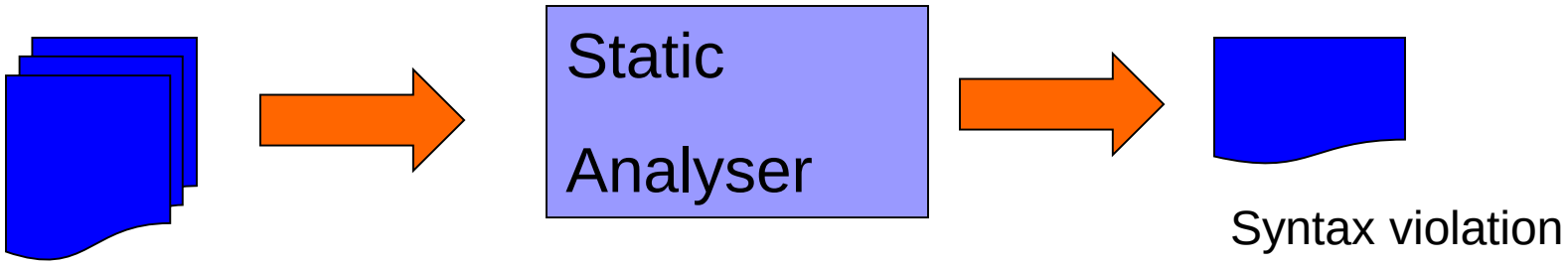
Program Slice for the variable ‘R’

Types of program Slicing

- Backward:
  - For a given statement S, a backward slice through a program contains all statements that effect whether control reaches S and also all statements that effect the value of variables that occur in S.
- Forward:
  - For a given statement S, a forward slice through a program contains all statements that are affected by S.
- Static:
  - A static program slice is calculated symbolically, i.e. takes no account of concrete data values.
- Dynamic:
  - A dynamic program slice is calculated based upon particular data values.

Note that forward and backward slices can be calculated either statically or dynamically.

Automatic static analyzers: General form



Statically analyze program code using compilers, data flow analysis, and control flow analysis

Frama-C Example

```
/*@ ensures \result >= x && \result >= y;
    ensures \result == x || \result == y;
*/
int max (int x, int y)
{ return (x > y) ? x : y; }
```

SPARK based example

- Exploits software annotations, i.e. meta-data that asserts properties that should hold at particular points during program execution.

```
procedure Exchange(X, Y:in out Float)
--# derives X from Y &
--# Y from X;
is
T:Float;
begin
        T:=X; X:=Y; Y:=T;
end Exchange;
```

- Note: derives defines a dependency relation between variables that is checked against the code automatically by the Spark Examiner static analyzer.

Static Analysis

- If a static analysis is performed before a review, a number of defects can be found and the number of the aspects to be checked in the review clearly decreases
  - Thus much less effort in a testing
- Not all defects can be found using static testing
  - Some defects become apparent only when the program is executed (runtime)
    - Example: division by zero valued variable

Data Flow Analysis

- What is it?
  - A form of static analysis based on the definition and usage of variables
- How it is performed?
  - Analysis of data use
    - **The usage of data on paths through the program code is checked**
- Use to detect data flow anomalies
  - Unintended or unexpected sequence of operations on a variable
- What is an anomaly?
  - An inconsistency that can lead to failure, but does not necessarily so
  - May be flagged as a risk

Data Flow Analysis

- Examples of data flow anomalies
  - Reading variables without previous initialization
  - Not using the values of a variable at all
- The usage of every variable is inspected
- Three types of usage or states of variables
  - Defined (d) : the variable is assigned a value
  - Reference (r) : the value of the variable is read and/or used
  - Undefined (u) : the variable has no defined value

Data Flow Analysis

- Three types of data flow anomalies
  - ur-anomaly : an undefined value (u) of a variable is read on a program path (r)
  - du-anomaly : the variable is assigned a value (d) that becomes invalid/undefined (u) without having been used in the meantime
  - dd-anomaly : the variable receives a value for the second time (d) and the first value had not been used (d)

Data Flow Analysis: Example

The following function is supposed to exchange the integer Value of the parameters Max and Min with the help of the variable Help, if the value of the variable Min is greater than the value of the variable Max

```
void exchange (int& Min, int& Max)
{
        int Help;
        if (Min > Max)
          {
            Max = Help;
            Max = Min;
            Help = Min;
           }
}
```

Data Flow Analysis: Example

```
int Help;
        if (Min > Max)
          {
             Max = Help;
             Max = Min;
             Help = Min;
           }
```

- The following anomalies detected:
  - ur-anomaly of the variable Help
    - The domain of the variable is limited to the function **exchange**
    - The first usage of the variable is on the right side of an assignment
    - At this time, the variable still has an undefined value, which is referenced there
    - There was no initialization of the variable when it was declared

Data Flow Analysis: Example

```
int Help;
        if (Min > Max)
          {
             Max = Help;
             Max = Min;
             Help = Min;
           }
```

- The following anomalies detected:
  - dd-anomaly of variable Max
    - The variable is used twice consecutively on the left side of an assignment and therefore is assigned a value twice
    - Either the first assignment can be omitted or the use of the first value has been forgotten

Data Flow Analysis: Example

```
int Help;
        if (Min > Max)
          {
             Max = Help;
             Max = Min;
             Help = Min;
           }
```

- The following anomalies detected:
  - du-anomaly of the variable Help
    - In the last assignment of the function the variable Help is assigned another value that cannot be used anywhere
    - This is because the variable is only valid inside the function

Correct code

```
void exchange (int& Min, int& Max)
    {
        int Help;
        if (Min > Max)
          {
            Max = Help;
            Max = Min;
            Help = Min;
          }
    }
```

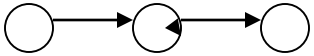
```
Help = Max;
Max = Min;
Min = Help;
```

Control Flow Analysis

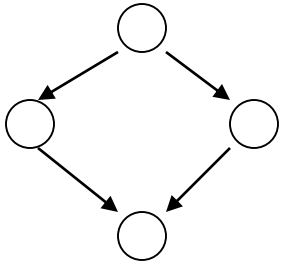
- What is control flow?
  - An abstract representation of all possible sequences of events (paths) in the execution of a component or system
- A program structure is represented (modeled) by a Control Flow Graph (CFG)
- CFG is a directed graph that shows a sequence of events (paths) in the execution through a component or system
- CFG consists of nodes ○ and edges ⟶
  - Node represents a statement or a sequence of statements
  - Edge represents control flows from one statement to another

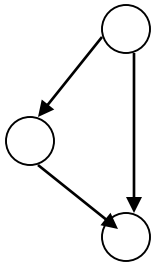
Control Flow Analysis

- Basic constructs of CFG
  - Sequence of assignment statements



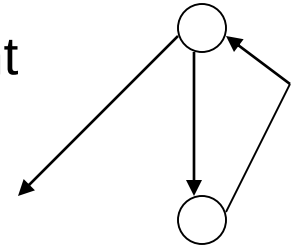
  - IF … THEN … ELSE statement

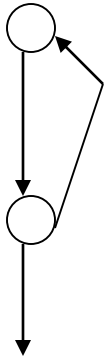




IF … THEN

  - WHILE DO statement



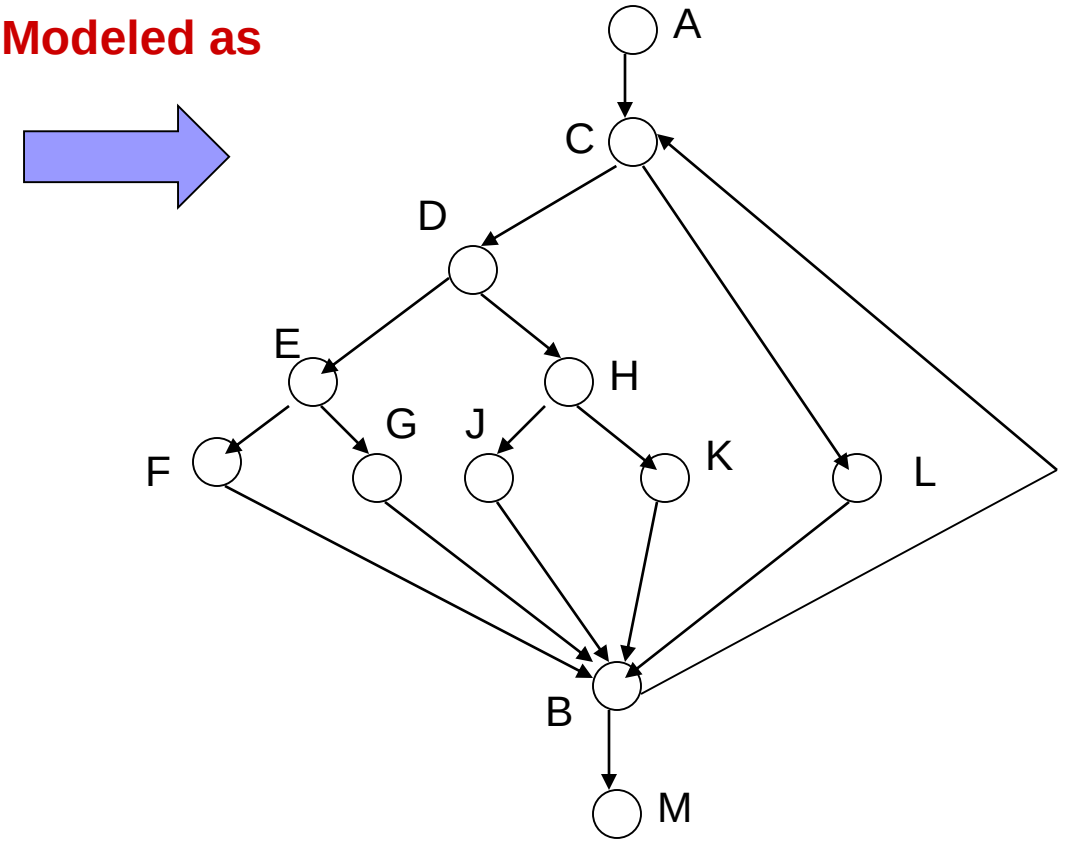


DO WHILE

Control Flow : example

```
A
DO
        IF C THEN
          IF D THEN
            IF E THEN F
            ELSE G
          ELSE IF H THEN J
                  ELSE K
        ELSE L
WHILE B
M
```

Modeled as



Control Flow Anomalies

- Statically detected anomaly in the control flow of a test object
- Example
  - Jumps out of a loop body
  - Program structure has many exits

Determining Metrics

- Quality characteristics can be measured with metrics
- The intention is to gain a quantitative measure of software whose nature is abstract
- Example:
  - McCabe's metric or cyclomatic complexity, V
  - Measures the structural complexity of program code
  - Based on CFG
  - V(G) = e - n + 2

    where V(G) is cyclomatic number of graph G

    e = number of edges in G

    n = number of nodes in G

Determining Metrics

Example: for CFG in previous slide

V(G) = e - n + 2 = 16 – 12 + 2 = 6

V(G) higher than 10 can not be tolerated and rework of the source code has to take place

- V(G) specifies the number of linearly independent paths in the program
- V(G) can be used to estimate the testability and maintainability

Use of Static Analysis for Secure Coding

- Safety Vulnerabilities:
  - Buffer Overflows
  - Memory Leaks
  - Integer Overflows
- Security Vulnerabilities:
  - Cross-site Scripting (XSS)
  - SQL Injection
  - Command Injection

Summary

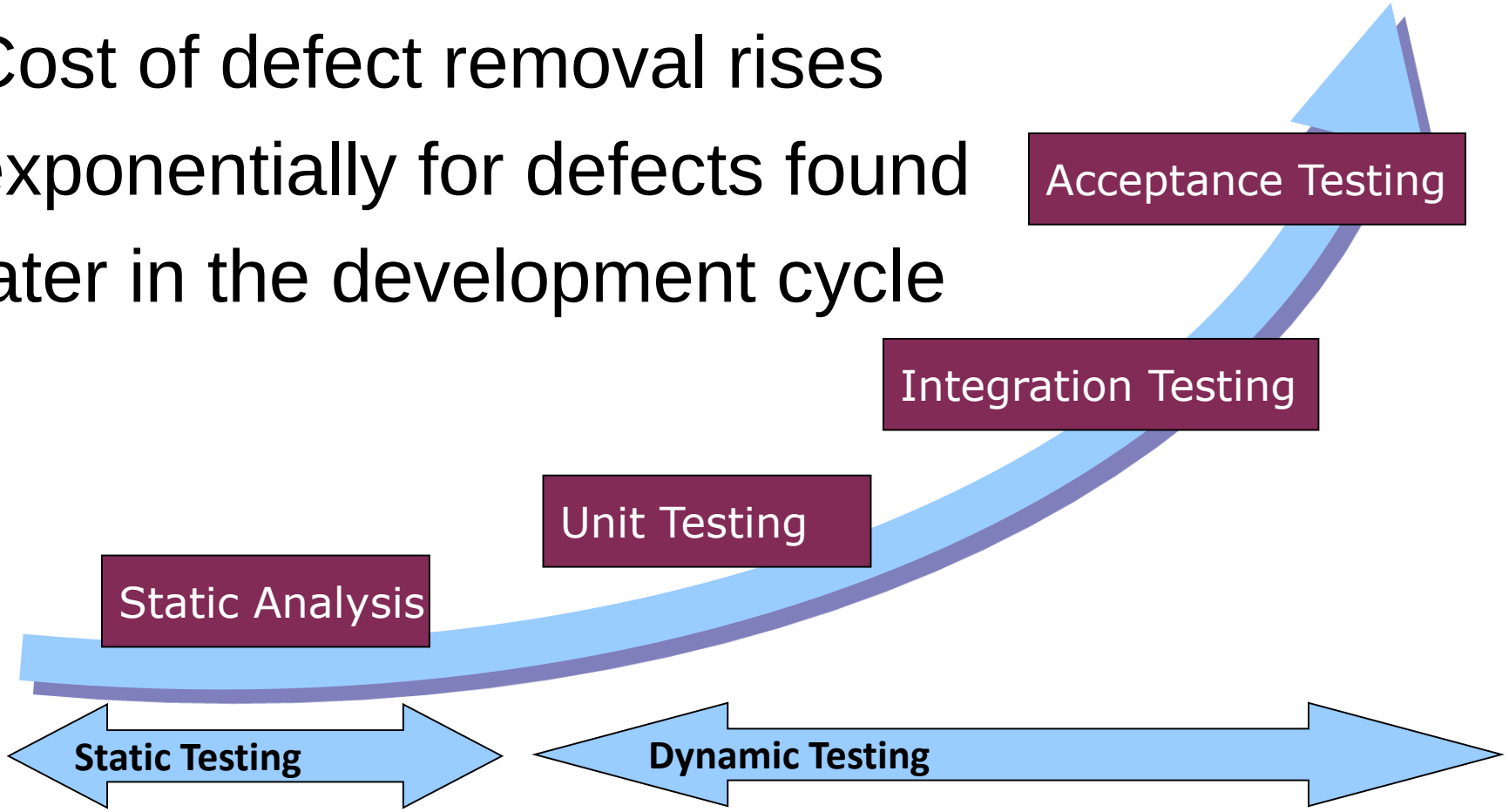
- Static analysis can be done to find defect and deviation using:
  - Compiler
  - Data flow analysis
  - Control flow analysis

Static and Dynamic Testing

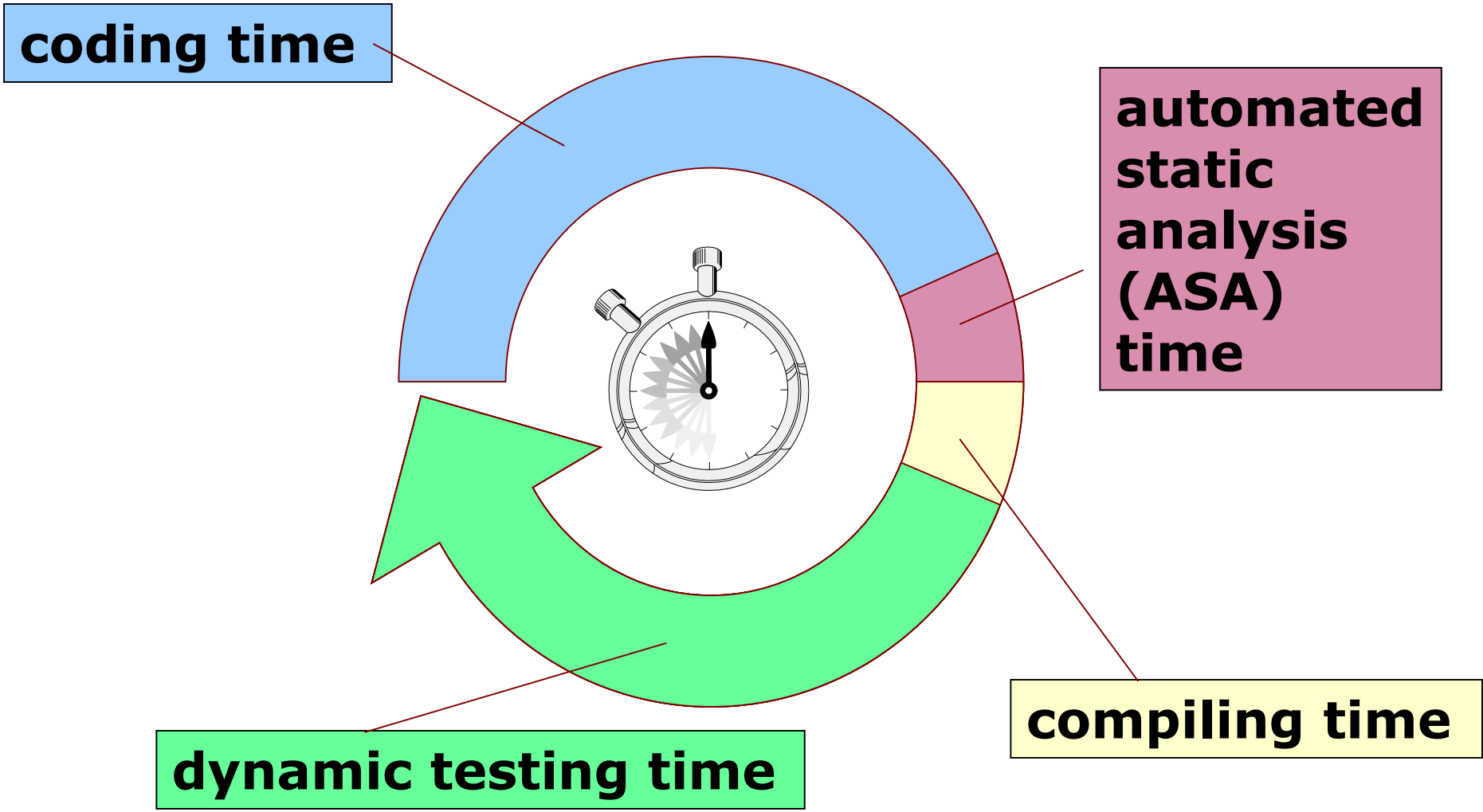
- Static and Dynamic testing are supplementary – static analysis does not replace dynamic testing but can significantly reduce dynamic testing effort
- Static testing achieves 100% statement coverage
- Including explicit static analysis in test coverage:
  - Improves overall test quality and test planning
  - Results in shorter dynamic testing time
  - Allows stronger focus testing on complex and crucial modules

Defect Removal Cost

Cost of defect removal rises exponentially for defects found later in the development cycle

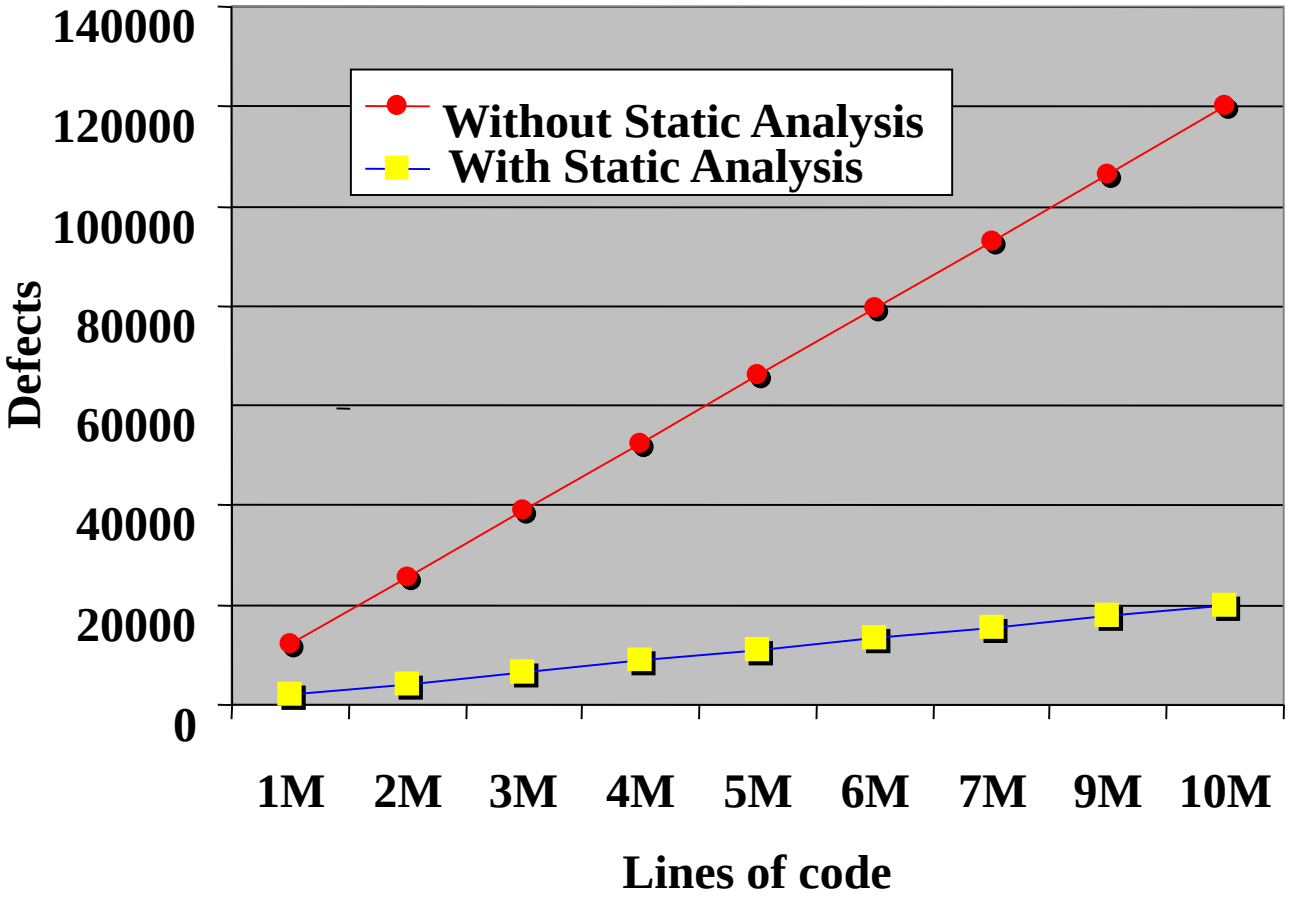


Time (cost) required for ASA is low



Impact (benefit) of ASA is high

Static Analysis may reduce defects by a factor of 6!



Source: Capers Jones, Software Productivity Group, Inc.

Thank you for your attention!
Please ask questions