

CS 569

# Selected Topics in Software Engineering: Program Analysis & Evaluation

## **Introduction**

Oregon State University, Winter 2024

# CS 569 Staff

- Instructor:  
Prof. Manish Motwani  
Office: KEC 3043  
Office hours: Tuesdays 2-4 PM  
[motwanim@oregonstate.edu](mailto:motwanim@oregonstate.edu)
- Teaching assistant:  
Suyash Sreekumar  
[sreekums@oregonstate.edu](mailto:sreekums@oregonstate.edu)  
Office hours: Thursdays 2-4 PM



# Today's plan

- What is CS 569 about and why you should take it?
- Foundations
- Introduction to program analysis and testing

# Software Failures Affect Everyone

500 million credentials exposed by data breach at Yahoo



Amongst the most recent [data breaches](#), on September 22, 2016, Yahoo confirmed a data breach that exposed about 500 million credentials that date back to four years. It is considered among the largest credential leaks of 2016. The company believes that this was a state-sponsored breach, where an individual on behalf of a government executed the entire hack. It further urged users to change their passwords and security questions. As a relief for the users, Yahoo stated that sensitive financial data like bank accounts and passwords was not stolen as part of the breach.

99.5% users left in cold after Nest thermostat software update



Software update for the Nest 'smart' thermostat (owned by Google) went wrong and literally left users in the cold. When the software update went wrong, it forced the device's batteries to drain out, leading to a temperature drop. Consequently, the customers could not heat their homes or use any amenities.

Nest claimed that the fault was due to a December 4.0 firmware update, with related issues such as old air filters or incompatible boilers. Later, it released a 4.0.1 software update that solved the issue for 99.5% of affected customers.

3,200 prisoners in WA released in advance due to glitch in behavior monitoring software



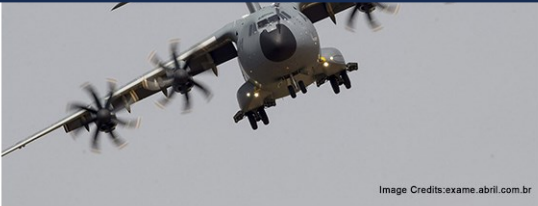
A glitch that occurred in December 2015 led to over 3,200 US prisoners being released before their declared date. The software was designed to monitor prisoners' behavior and was introduced in 2002. The problem occurred for about 13 years; on average, prisoners were released almost 49 days in advance.

600,000 payments failed affecting 6.5 million customers of RBS



About 6 lakh payments failed to get through the accounts of RBS overnight in June 2015, which included wages and benefits. Bank's chief admin officer stated it was a technology fault, and there was no further detail on the real cause. In 2012, about 6.5 million RBS customers had to face an outage caused due to a batch scheduling software glitch, where the bank was fined £56 million.

Software bug caused fatal aircraft crash in Airbus A400M aircraft



In May 2015, Airbus issued an alert for urgently checking its A400M aircraft when a report detected a software bug that had caused a fatal crash earlier in Spain. Before this alert, a test flight in Seville caused the death of four Air Force crew members, and two were left injured.

One-off error caused customers to pay twice at Co-operative Food



In July 2015, Co-operative Food apologized to its customers and promised a refund within 24 hours. The reason was a 'one-off technical glitch' while processing the software that resulted in customers being charged twice.



# Software Failures Affect Everyone

Bug exploited by programmers to enjoy unlimited free rides (OLA)



Ola, India's largest taxi aggregator, faced major security flaws within their system. The software bugs detected helped basic programmers enjoy unlimited free rides – at the expense of Ola and at the expense of users. The issue went public when customers brought up the weaknesses in the system. Ola tried to fix bugs when the complaints soared up and it was alarming for the brand's reputation in the marketplace.

Leeds Pathology IT crash delayed operations for 132 patients



In September 2016, Leeds Teaching Hospitals NHS Trust, one of Europe's largest teaching trusts, witnessed a pathology IT crash that delayed operations for almost 132 patients. Leeds Teaching holds a budget of £1 billion and employs over 16,000 staff. It serves 780,000 people in the city and provides expert care for 5.4 million patients. The outage further affected Bradford Teaching Hospitals NHS Foundation Trust, GP services in Leeds, and a minor number of GP services in Bradford.

Cyber attack on nuclear power plant caused disruption for several years (IAEA)



In October 2016, the head of an international nuclear energy consortium declared that disruption at a nuclear power plant during the last several years was caused due to a 'Cyber Attack'. Yukiya Amano, head of the International Atomic Energy Agency (IAEA) didn't drill the matter much in detail but did alter the potential attacks in the future.

Volkswagen deploys advanced software to cheat emission tests



In September 2015, the US government, in a dramatic move, ordered Volkswagen to recall about 500,000 cars after learning that the company had deployed advanced software to cheat emission tests and allowed its cars to produce 40 times more emissions than the decided limit. The Environment Protection Agency (EPA) accused VW of installing illegal 'defeat device' software that substantially reduces Nitrogen oxide (NOx) emissions only while undergoing emission tests. The company further admitted it and announced a recall as well.

Third-party sellers' products sold for a penny on Amazon



One of the most known glitches in history is the Amazon 1p price glitch, where third-party sellers listed on Amazon saw their products being priced at 1p each. While the products were delivered, numerous small-time retailers had to appeal to the customers to return the items.

Flipkart failed to manage heavy rush on Big Billion Day Sale



In October 2014, Flipkart, an India based e-commerce giant, sent a note to its customers apologizing for the glitches that took place on the Big Billion Day Sale. The site encountered a heavy rush, which it couldn't manage, which resulted in cancellation of orders, delayed delivery, and much more that was beyond them to manage. While the sale helped the e-commerce giant garner a billion hits in a day, it was certainly a PR nightmare for the brand.

# Software Failures Affect Everyone

**606 SOFTWARE FAILURES**  
**1177 NEWS STORIES**  
**314 COMPANIES**



**LOSSES FROM SOFTWARE FAILURES (USD)**

**1,715,430,778,504**

ONETRILLIONSEVENHUNDREDFIFTEENBILLIONFOURHUNDREDTHIRTYMILLIONSEVENHUNDREDSEVENTYEIGHTTHOUSANDFIVEHUNDREDFOUR



*With software growing at ~7% per year, it might be expected that failures would grow at about the same rate, or by about 22.5% through 2020. Extrapolating from the 2017 base, we could say that ~742 failures would likely occur in 2020 in English language media sources, for a total of \$2.08 trillion.*

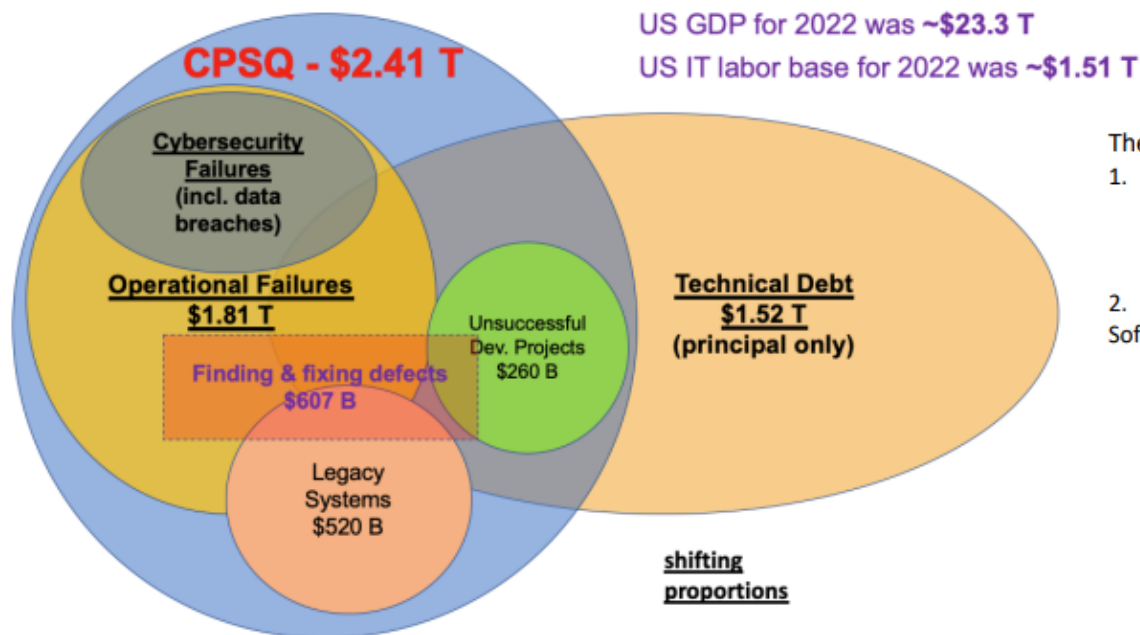
Sources: The Cost of Poor Software Quality in the US: A 2020 Report (<https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf>),

Software Fail Watch: 5th Edition, 2018 (<https://www.tricentis.com/software-fail-watch>)

# Software Failures Affect Everyone

In this 2022 update report we estimate that the cost of poor software quality in the US has grown to at least \$2.41 trillion<sup>1</sup>, but not in similar proportions as seen in 2020. The accumulated software Technical Debt (TD) has grown to ~\$1.52 trillion<sup>1</sup>.

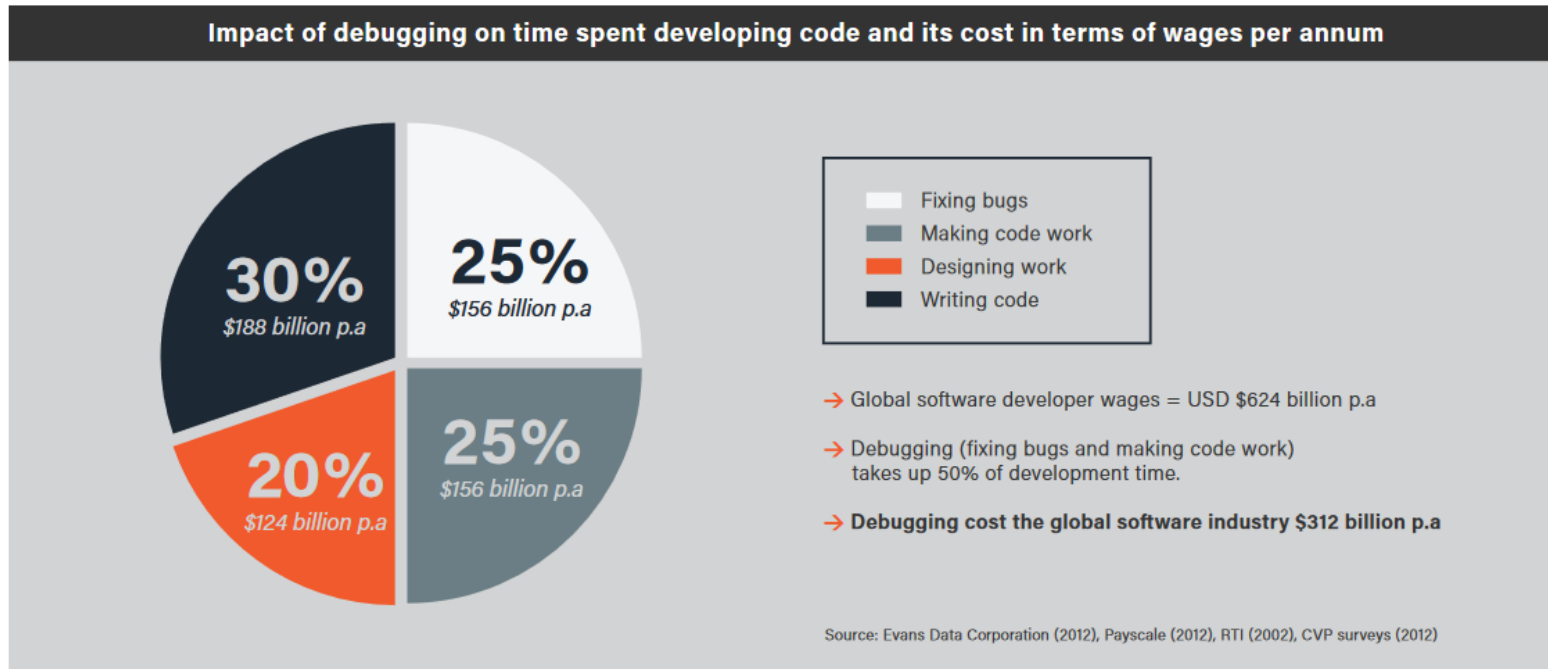
Figure 1-0 CPSQ in US in 2022



The 3 main problem areas that we will focus on this year are:

1. Cybercrime losses due to existing software vulnerabilities jumped way up
  - Losses rose 64% from 2020 to 2021. Those losses have not yet been determined for 2022.
  - Several critical infrastructure attacks cost an unmeasurable amount of pain and suffering over the last 2 years (e.g. Colonial Pipeline)
2. Software supply chain problems with underlying 3rd party components (especially Open Source Software, aka OSS) has risen significantly
  - In 2021, 77% of organizations reported an increase in the use of open source software
  - A medium-sized application (less than 1 million lines of code) carries 200 to 300 third-party components on average.

# Software Debugging Is Expensive



*“Software developers spend 35-50% of their time validating and debugging software. The cost of debugging, testing, and verification is estimated to account for 50-75% of the total budget of software development projects, amounting to more than \$100 billion annually.”*



# Why Software Debugging Is Expensive?

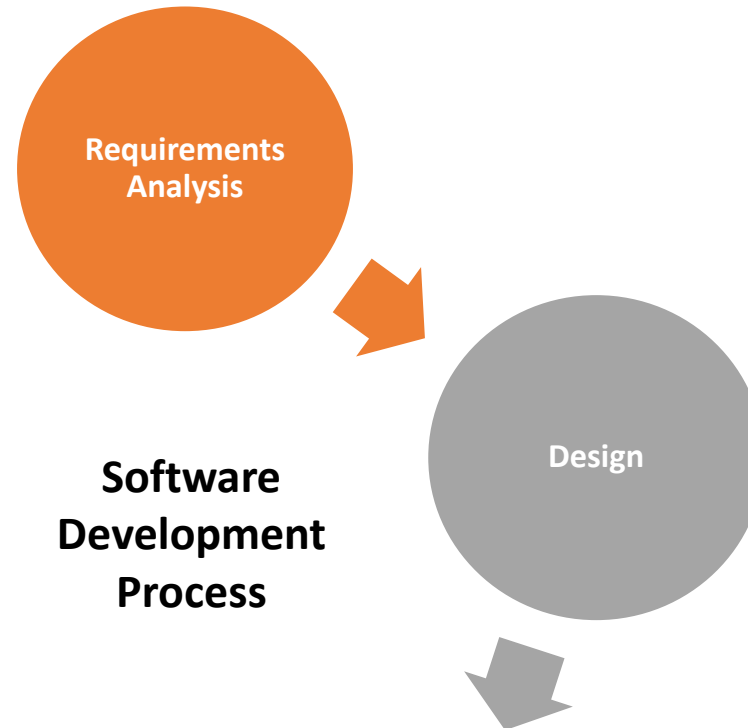
**Software  
Development  
Process**

# Why Software Debugging Is Expensive?

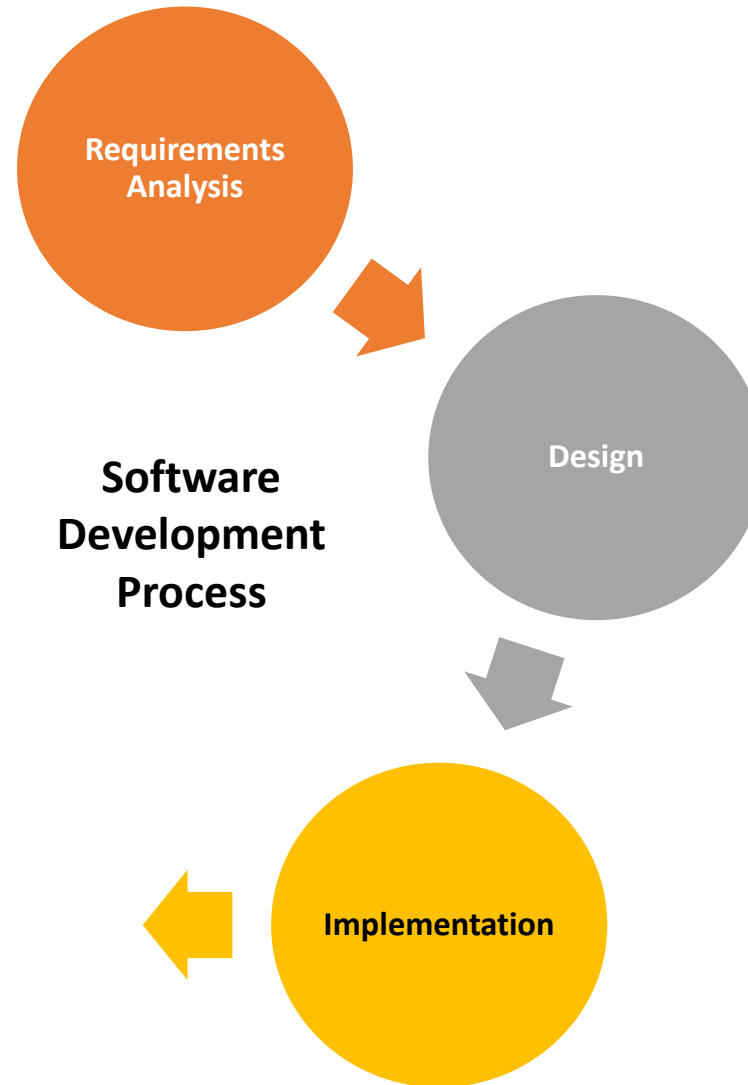


**Software  
Development  
Process**

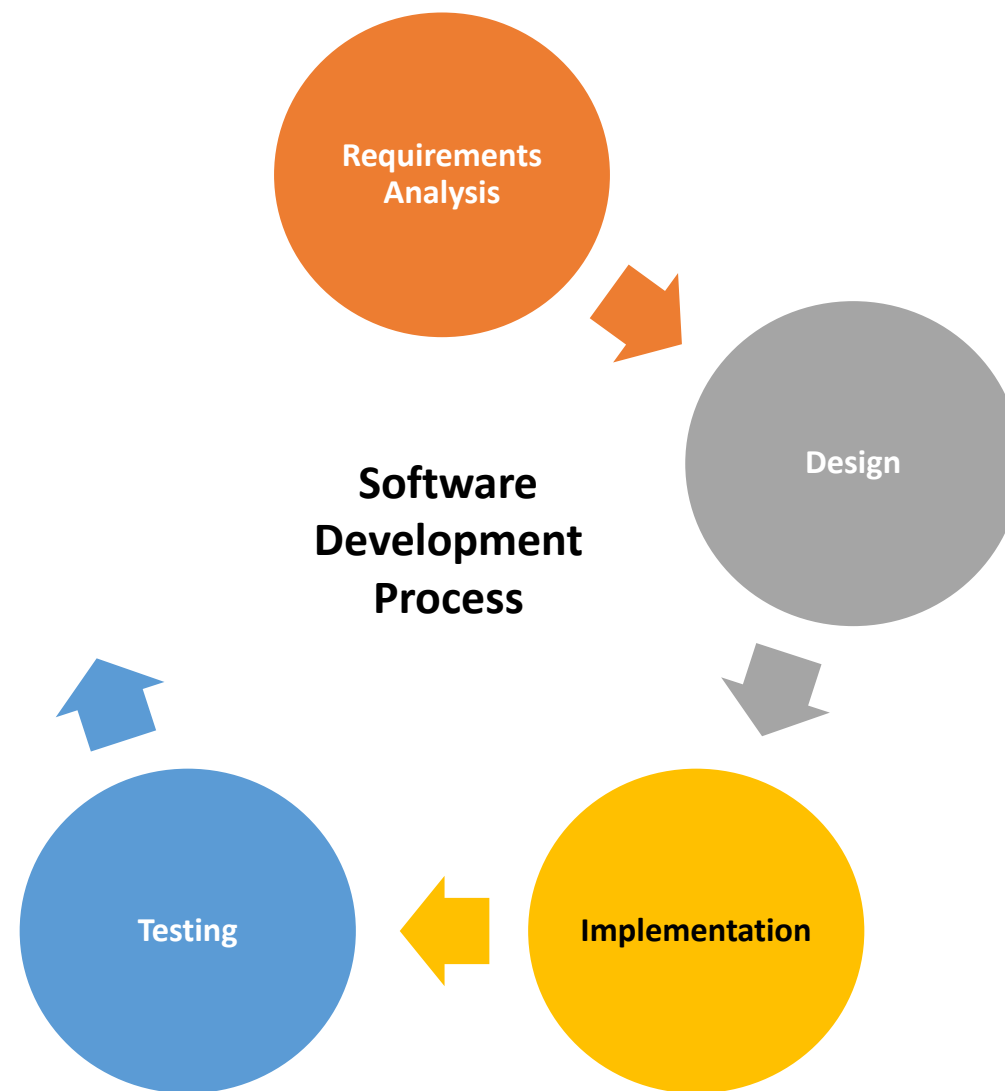
# Why Software Debugging Is Expensive?



# Why Software Debugging Is Expensive?

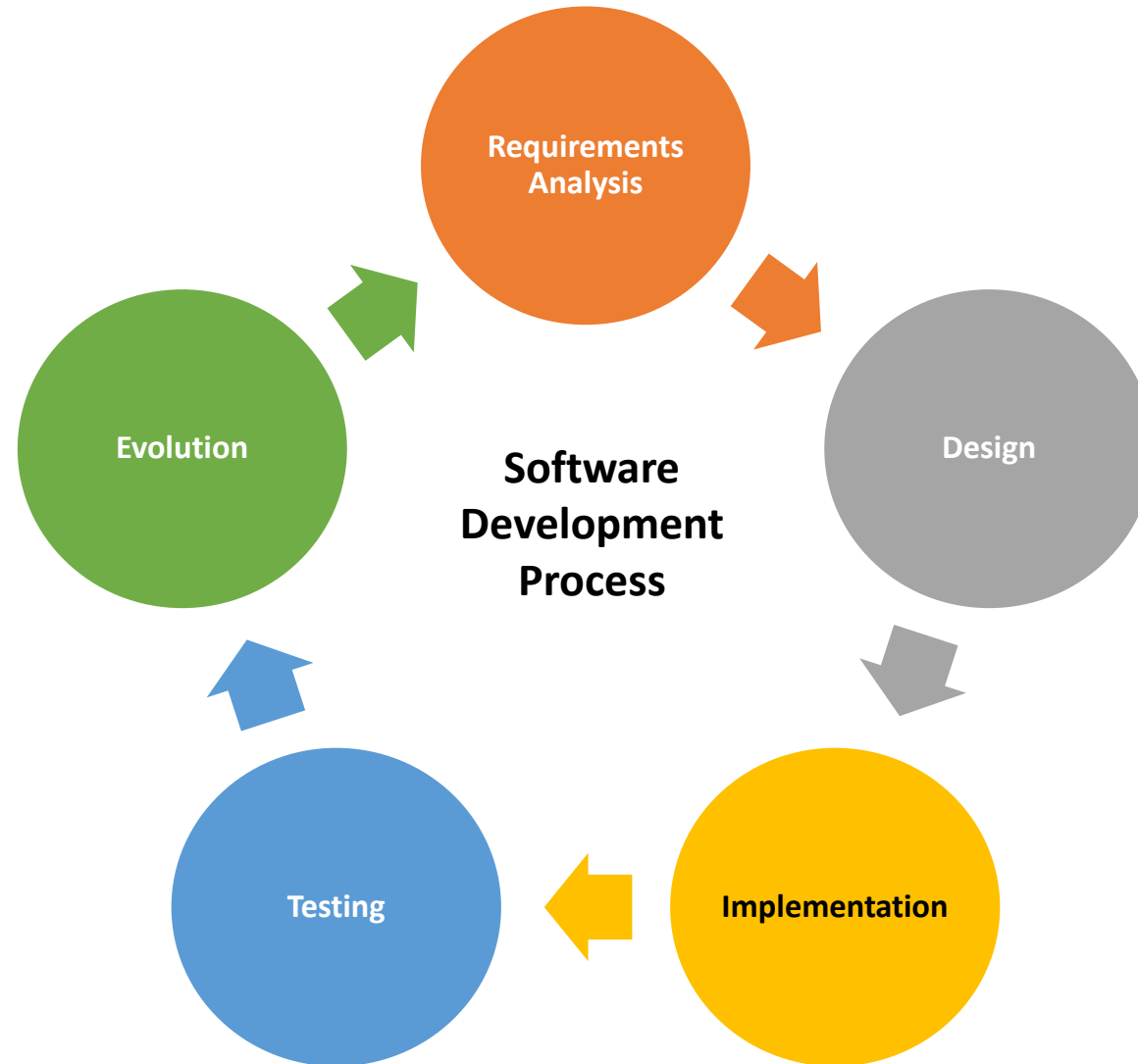


# Why Software Debugging Is Expensive?

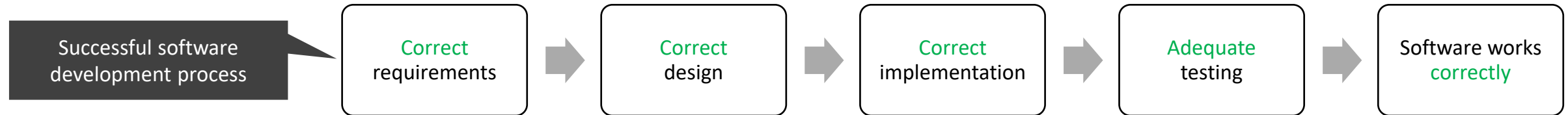




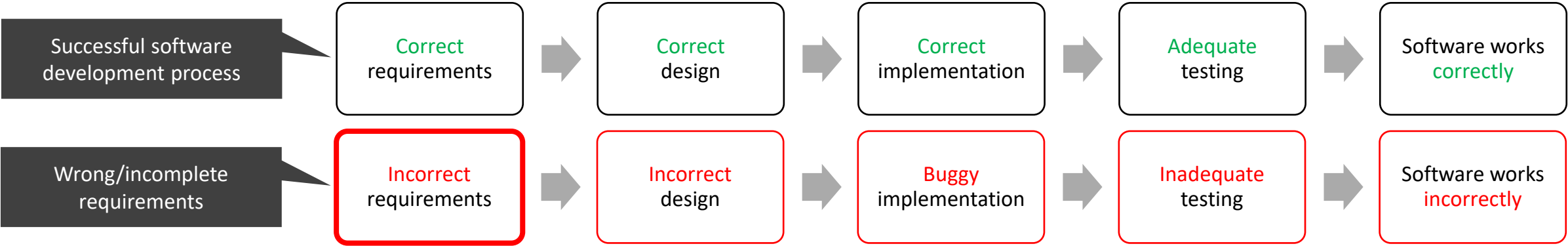
# Why Software Debugging Is Expensive?



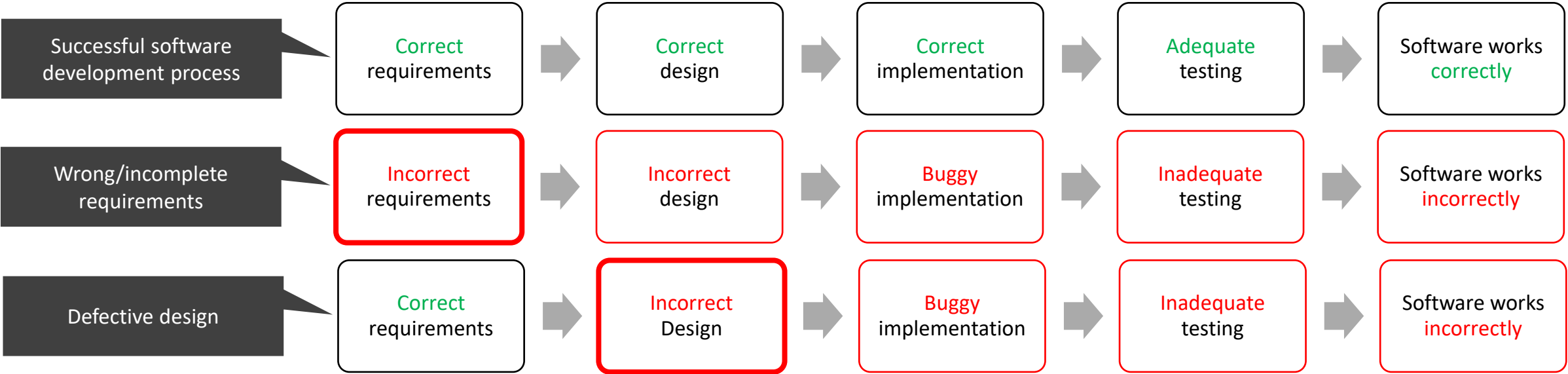
# Why Software Fails?



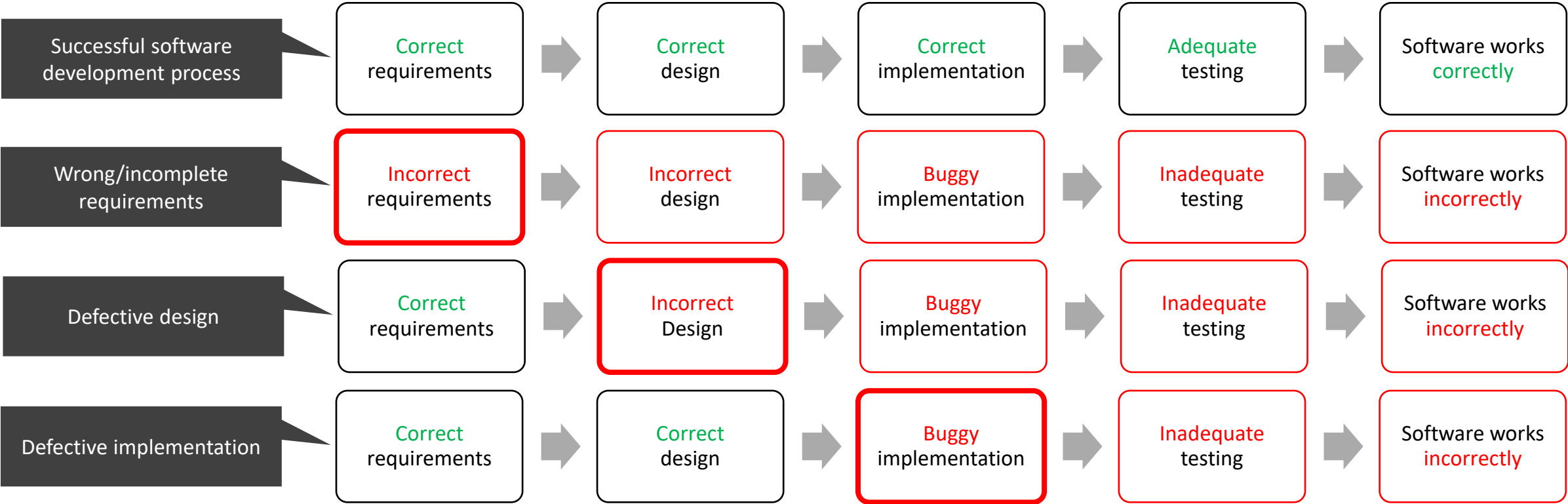
# Why Software Fails?



# Why Software Fails?

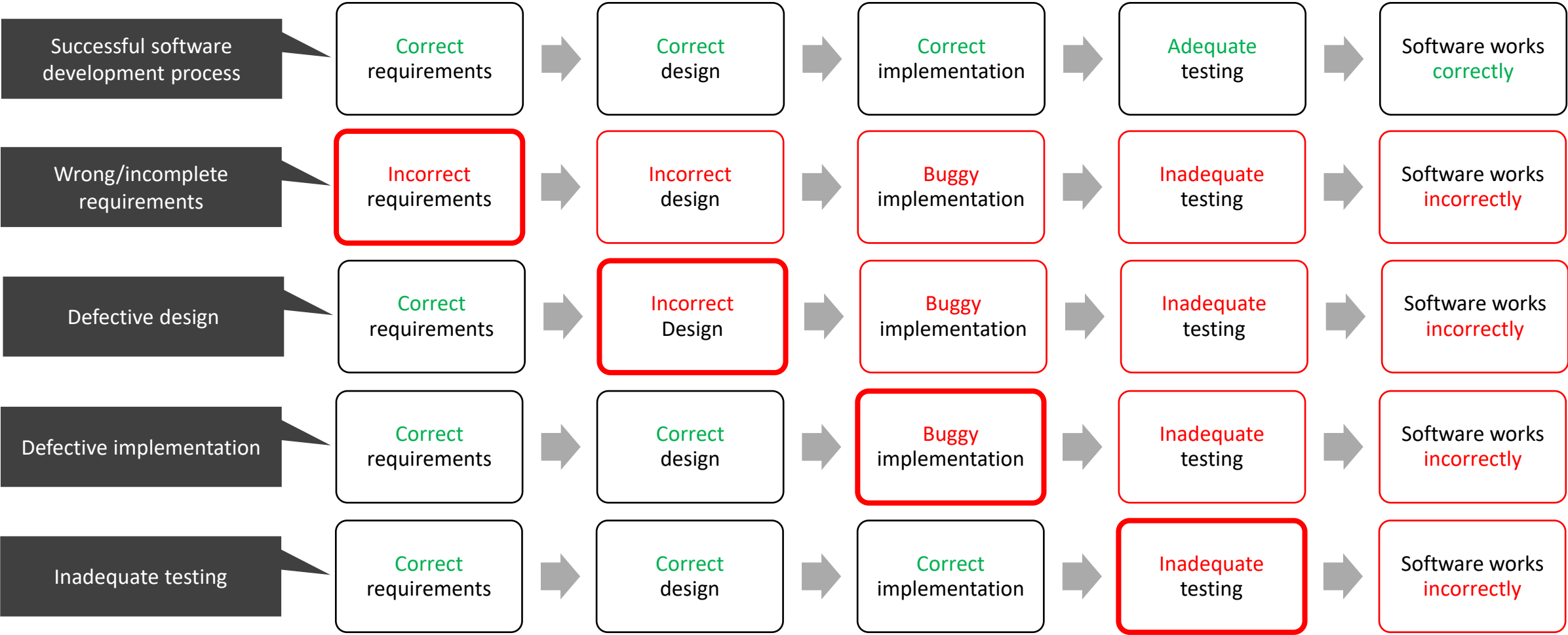


# Why Software Fails?





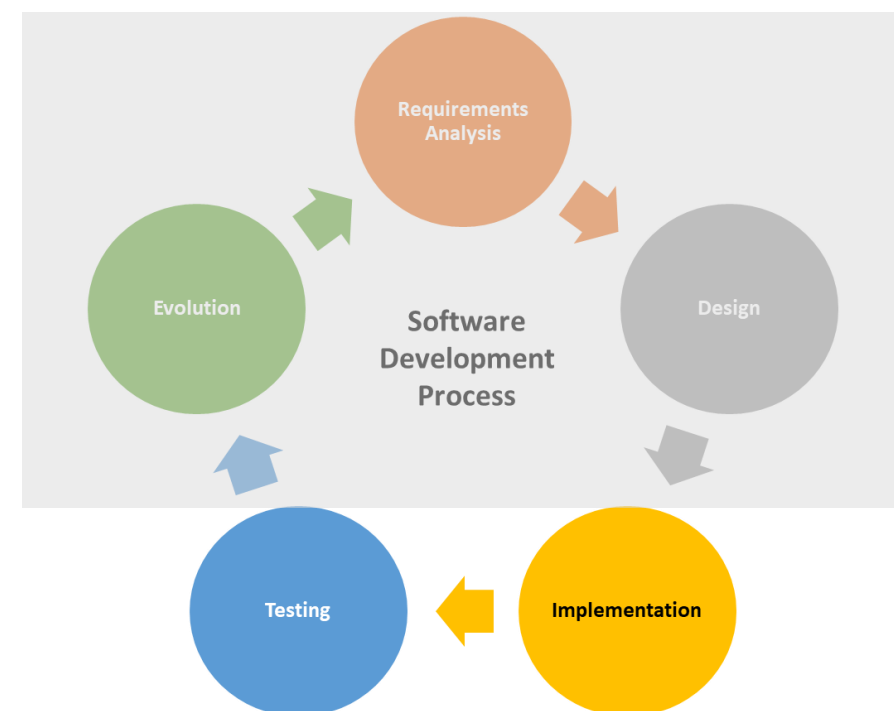
# Why Software Fails?



# What is CS 569 about?



Analyzing and evaluating software programs to verify and validate their “correctness”



# What is CS 569 about? The bigger picture

- **CS 361, SOFTWARE ENGINEERING I:** front-end software development (requirements analysis and specification; design techniques; project management)
- **CS 362, SOFTWARE ENGINEERING II:** back-end software development (verification and validation; debugging; maintenance)
- **CS 561, SOFTWARE ENGINEERING METHODS:** methods and supporting tools in the context of different SDLC methods (e.g., agile, waterfall)
- **CS 562, SOFTWARE PROJECT MANAGEMENT:** managing software projects to deliver high-quality systems in timely and cost-effective manner
- **CS 563, SOFTWARE MAINTENANCE AND EVOLUTION:** maintaining and evolving mature and deployed software systems with constantly changing technology and system requirements
- **CS 569, SELECTED TOPICS IN SOFTWARE ENGINEERING: PROGRAM ANALYSIS AND EVALUATION:** methods and techniques to verify and validate software systems

# What is CS 569 about?

- **CS 569, SELECTED TOPICS IN SOFTWARE ENGINEERING: PROGRAM ANALYSIS AND EVALUATION:** methods and techniques to verify and validate software systems
- This is not a programming course!
- Please visit <https://mmotwani.com/courses/SkillAssessment.html> to brush-up the recommended skillset required to do homework and project.
- The course focuses on state-of-the-art techniques for software analysis and testing
- State-of-the-art means exploring or cutting-edge research
- Students will learn the latest techniques in improving software quality
- Students will advance the state-of-the-art by developing their own techniques!

# What is CS 569 about?

- **CS 569, SELECTED TOPICS IN SOFTWARE ENGINEERING: PROGRAM ANALYSIS AND EVALUATION:** methods and techniques to verify and validate software systems
- **TOPICS COVERED:**
  - Introduction and basics
  - Dynamic analysis (Delta debugging, Mutation testing)
  - Static analysis (Data flow analysis, Call graphs)
  - Automated testing (Symbolic execution, Random test generation/Fuzzing)
  - Software quality assurance



# CS 569 policies

- **Grading policy:**
  - Class participation: 10%
  - Paper presentation: 20% (5% paper selection, 7% report, 8% oral)
  - Homework: 30% (10% each homework)
  - Project: 40% (15% project idea presentation and report, 10% project plan presentation and report, 15% final presentation and report)
- **Late policy:** All submission deadlines are sharp, and no extensions will be granted after the deadline. Early requests for extensions will be considered only in extenuating circumstances. No more than one extension will be granted per student/group.
- **Attendance policy:** All lectures will be in-person and not recorded. All students are expected to attend the in-person lectures to do well in assignments. However, no penalty will be imposed for missing any lectures.

# CS 569 material

- All the lectures and assignments related to homework, paper presentation, and project will be available on Canvas (<https://canvas.oregonstate.edu/>)
- Please check announcements on Canvas to stay updated.
- All submissions and grading will be done via Canvas.
- Use Canvas Discussions (preferred) or Discord (<https://discord.com/invite/xG6tZajCXt>) for asking questions and sharing ideas.

# CS 569 Content Organization

Lectures

Homework

Project

Paper presentation

week	day & date	topic	reading	homework	project	paper presentation
week 1	Mon, 01/08/24	Course Introduction				
	Wed, 01/10/24	Invariant Detection	<u>Dynamically discovering likely program invariants to support program evolution</u> ➡			
week 2	Mon, 01/15/24	No class (MLK day)				
	Wed, 01/17/24	Software Specifications		<u>Homework 1</u> (due 01/29/24, 11:59 PM)	<u>Project Idea Report</u> (due 01/29/24, 11:59 AM)	<u>Paper Selection</u> (due 01/22/24, 11:59 PM)

<https://canvas.oregonstate.edu/courses/1964338/assignments/syllabus>

# CS 569 Nondiscrimination policy

Software engineering is at its nature a collaborative activity and it benefits greatly from diversity. This course includes and welcomes all students regardless of age, background, citizenship, disability, sex, education, ethnicity, family status, gender, gender identity, geographical origin, language, military experience, political views, race, religion, sexual orientation, socioeconomic status, and work experience. Our discussions and learning will benefit from these and other diverse points of view.

**Any kind of language or action displaying bias against or discriminating against members of any group or making members of any group uncomfortable are against the mission of this course and will not be tolerated.** The instructor welcomes discussion of this policy and encourages anyone experiencing concerns to speak with him.

# CS 569 Academic integrity

- **Students are allowed to work together on all aspects of this class. However, for the homework assignments, each student must submit his or her own write up, clearly stating the collaborators. Your submission must be your own. When in doubt, contact the instructors about whether a potential action would be considered plagiarism.** If you discuss material with anyone besides the class staff, acknowledge your collaborators in your write-up. If you obtain a key insight with help (e.g., through library work or a friend), acknowledge your source and write up the summary on your own. It is the student's responsibility to remove any possibility of someone else's work from being misconstrued as the student's. Never misrepresent someone else's work as your own. It must be absolutely clear what material is your original work. Plagiarism and other anti-intellectual behavior will be dealt with severely. **Note that facilitation of plagiarism (giving your work to someone else) is also considered to be plagiarism and will carry the same repercussions.**
- Students are encouraged to use the Internet, literature, and other publicly-available resources, except the homework solutions and test (including quizzes, midterms, finals, and other exams) solutions, from past terms' versions of this course and other academic courses, whether at OSU and at other institutions. **To reiterate, the students are not allowed to view and use past homework and test solutions, unless explicitly distributed by the CS 569 staff as study material. Whenever students use Internet (including ChatGPT), literature, and other publicly-available resources, they must clearly reference the materials in their write ups, attributing proper credit. This cannot be emphasized enough: attribute proper credit to your sources.** Failure to do so will result in a zero grade for the assignment and possibly a failing grade for the class, at the instructor's discretion. **Copying directly from resources is not permitted, unless the copying is clearly identified as a quote from a source. Most use of references should be written in the words of the student, placing the related work in proper context and describing the relevant comparison.**



# 10 Min Break (Q/A)

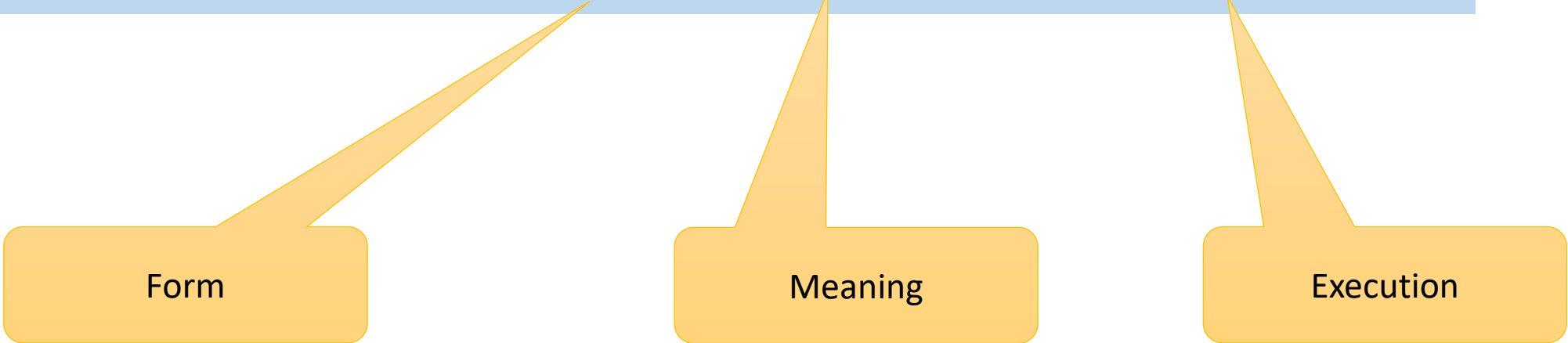
09:44



Source: <https://xkcd.com/1838/>

# Foundations

**Programming Language = Syntax + Semantics + Implementation**



Form

Meaning

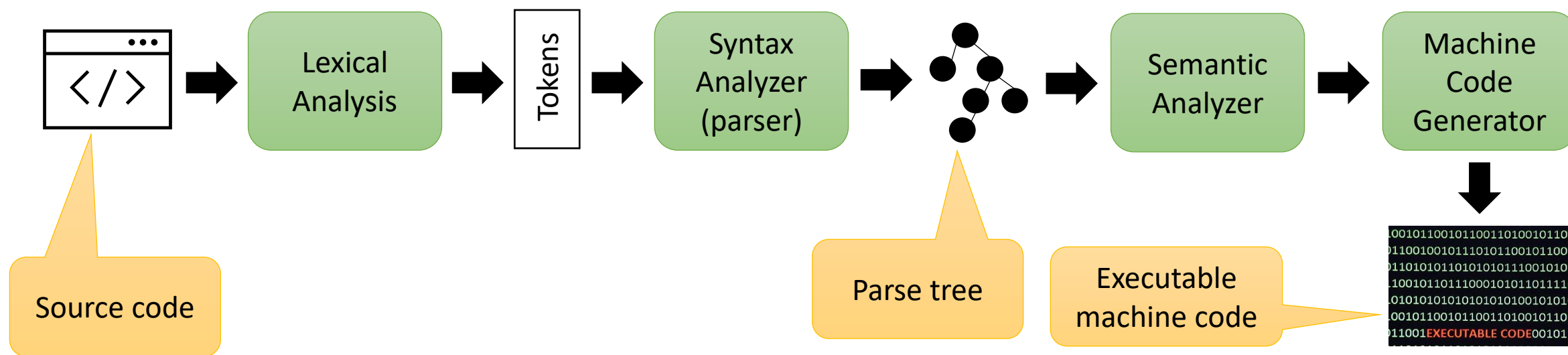
Execution

# Foundations

**Programming Language = Syntax + Semantics + Implementation**

Programming Language Implementation:

## 1. Compiler (C, C++)

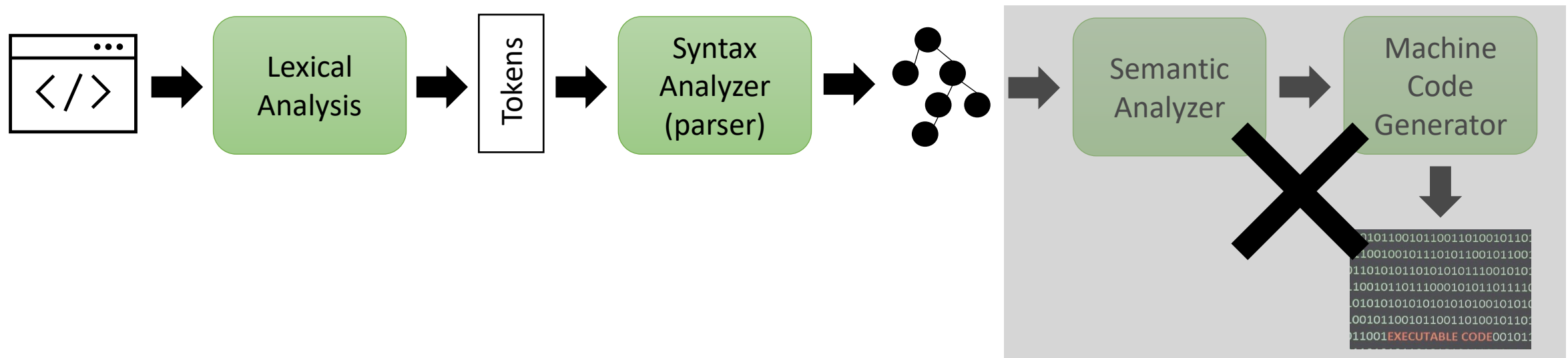


# Foundations

**Programming Language = Syntax + Semantics + **Implementation****

Programming Language Implementation:

1. Compiler (C, C++)
- 2. Interpreter (Python, Ruby)**

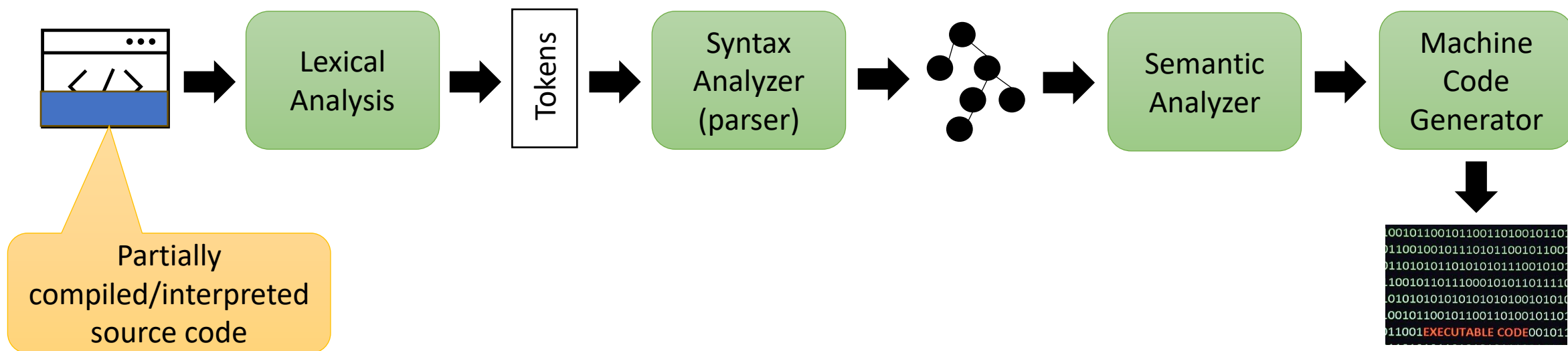


# Foundations

**Programming Language = Syntax + Semantics + Implementation**

Programming Language Implementation:

1. Compiler (C, C++)
2. Interpreter (Python, Ruby)
- 3. Hybrid (Java, C#)**



# Foundations

**Programming Language = Syntax + Semantics + Implementation**

## Programming Language Syntax

### 1. **Grammar** (which program is syntactically correct?)

- a) Terminals **T**
- b) Non-terminals **N**
- c) Production rules **P**
- d) Start symbol **S**  $\in$  **N**

### **Example:** Arithmetic Expression

**T:** { 0, 1, 2, ..., 9, +, - }

**N:** { Exp, Num, Op, Digit }

**P:**

- $\text{Exp} \rightarrow \text{Num} \mid \text{Exp Op Exp}$
- $\text{Op} \rightarrow + \mid -$
- $\text{Num} \rightarrow \text{Digit} \mid \text{DigitNum}$
- $\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

**S:** Exp

# Foundations

**Programming Language = Syntax + Semantics + Implementation**

## Programming Language Syntax

**1. Grammar** (which program is syntactically correct?)

- a) Terminals **T**
- b) Non-terminals **N**
- c) Production rules **P**
- d) Start symbol **S**  $\in$  **N**

**Example:** Arithmetic Expression

**T:** { 0, 1, 2, ..., 9, +, - }

**N:** { Exp, Num, Op, Digit }

**P:**

- $\text{Exp} \rightarrow \text{Num} \mid \text{Exp Op Exp}$
- $\text{Op} \rightarrow + \mid -$
- $\text{Num} \rightarrow \text{Digit} \mid \text{DigitNum}$
- $\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

**S:** Exp

**Quiz:**

Which of the following is part of the Arithmetic Expression language?

- 1.  $12 - 2$
- 2.  $2 + ( 5 - 34 )$
- 3.  $11 / 4$
- 4. 123400987
- 5.  $12 * 5 + 2$

# Foundations

**Programming Language = Syntax + Semantics + Implementation**

## Programming Language Syntax

### 1. Grammar (which program is syntactically correct?)

- a) Terminals **T**
- b) Non-terminals **N**
- c) Production rules **P**
- d) Start symbol **S**  $\in$  **N**

### Example: Arithmetic Expression

**T:** { 0, 1, 2, ..., 9, +, - }

**N:** { Exp, Num, Op, Digit }

**P:**

- $\text{Exp} \rightarrow \text{Num} \mid \text{Exp Op Exp}$
- $\text{Op} \rightarrow + \mid -$
- $\text{Num} \rightarrow \text{Digit} \mid \text{DigitNum}$
- $\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

**S:** Exp

### Quiz:

Which of the following is part of the Arithmetic Expression language?

- 1.  $12 - 2$  ✓
- 2.  $2 + ( 5 - 34 )$  ✗
- 3.  $11 / 4$  ✗
- 4.  $123400987$  ✓
- 5.  $12 * 5 + 2$  ✗



# Foundations

**Programming Language = Syntax + Semantics + Implementation**

## Programming Language Syntax

1. Grammar (which program is syntactically correct?)
  - a) Terminals **T**
  - b) Non-terminals **N**
  - c) Production rules **P**
  - d) Start symbol **S**  $\in$  **N**

## 2. Abstract Syntax Tree

Abstract grammar

Example,

**P:**

- $E \rightarrow N \mid \text{Op } (E, E)$
- $\text{Op} \rightarrow + \mid -$

**T:** program tokens

# Foundations

**Programming Language = Syntax + Semantics + Implementation**

## Programming Language Syntax

1. Grammar (which program is syntactically correct?)
  - a) Terminals **T**
  - b) Non-terminals **N**
  - c) Production rules **P**
  - d) Start symbol **S**  $\in$  **N**

## 2. Abstract Syntax Tree

Abstract grammar

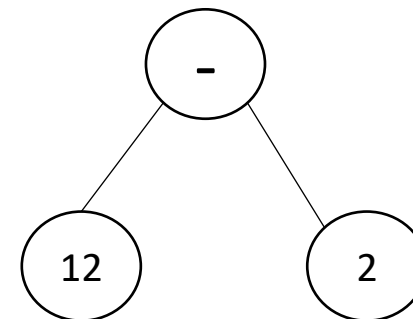
Example,

**P:**

- $E \rightarrow N \mid \text{Op}(E, E)$
- $\text{Op} \rightarrow + \mid -$

**T:** program tokens

12 - 2



# Foundations

## Control Flow Graphs (CFGs)

- Models flow of control through program
- Directed Graph  $G = (N, E)$ , where
  - **N** : basic blocks (sequence of operations executed together)
  - **E** : possible transfer of control during execution of program

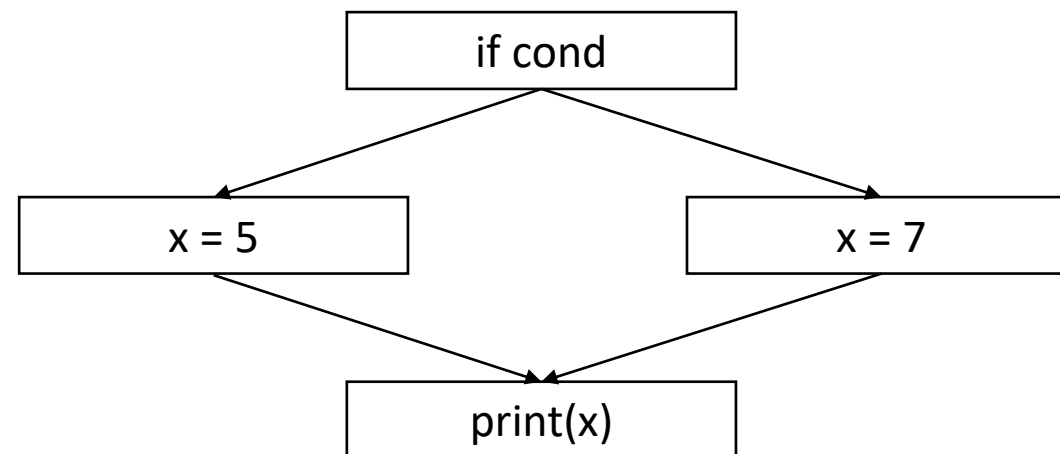
# Foundations

## Control Flow Graphs (CFGs)

- Models flow of control through program
- Directed Graph  $G = (N, E)$ , where
  - **N** : basic blocks (sequence of operations executed together)
  - **E** : possible transfer of control during execution of program

Example:

```
if cond:  
    x = 5  
else:  
    x = 7  
print(x)
```



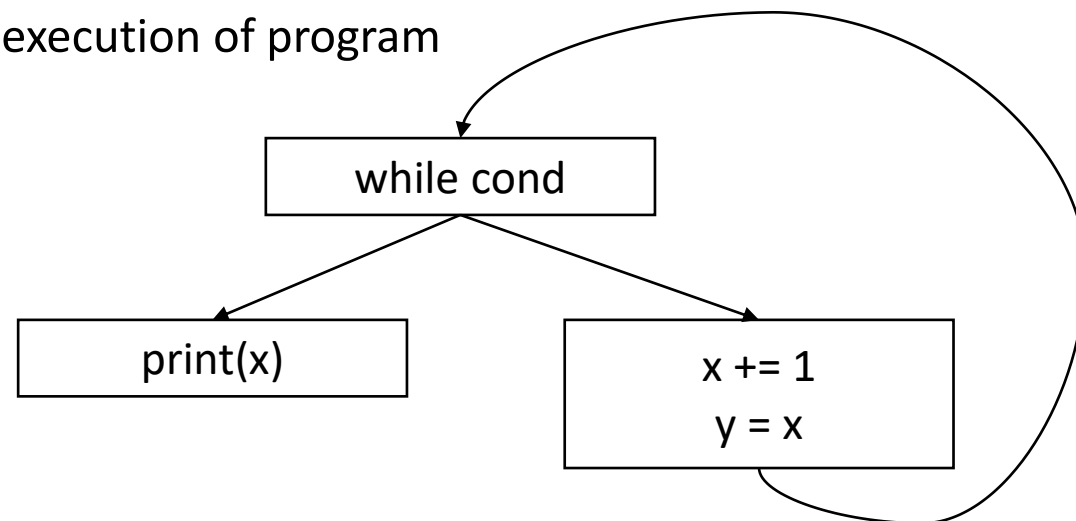
# Foundations

## Control Flow Graphs (CFGs)

- Models flow of control through program
- Directed Graph  $G = (N, E)$ , where
  - **N** : basic blocks (sequence of operations executed together)
  - **E** : possible transfer of control during execution of program

Example:

```
while cond:
    x += 1
    y = x
print(x)
```



# Foundations

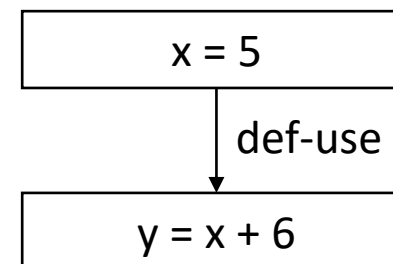
## Data Dependence Graphs (DDGs)

- Models flow of data from “definitions” to “uses”
- Directed graph  $G = (N, E)$ , where
  - **N**: operations
  - **E**: possible def-use relations
  - Edge  $e = (n_1, n_2)$  means node  $n_2$  **may** use data defined at node  $n_1$

Example:

$x = 5$

$y = x + 6$



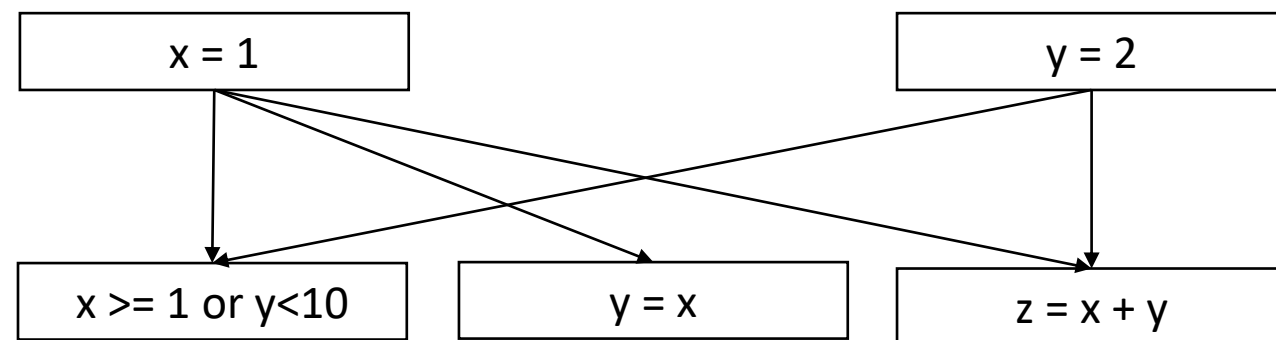
# Foundations

## Data Dependence Graphs (DDGs)

- Models flow of data from “definitions” to “uses”
- Directed graph  $G = (N, E)$ , where
  - N**: operations
  - E**: possible def-use relations
  - Edge  $e = (n_1, n_2)$  means node  $n_2$  **may** use data defined at node  $n_1$

Example:

```
x = 1
y = 2
if x >= 1 or y < 10:
    y = x
z = x + y
```



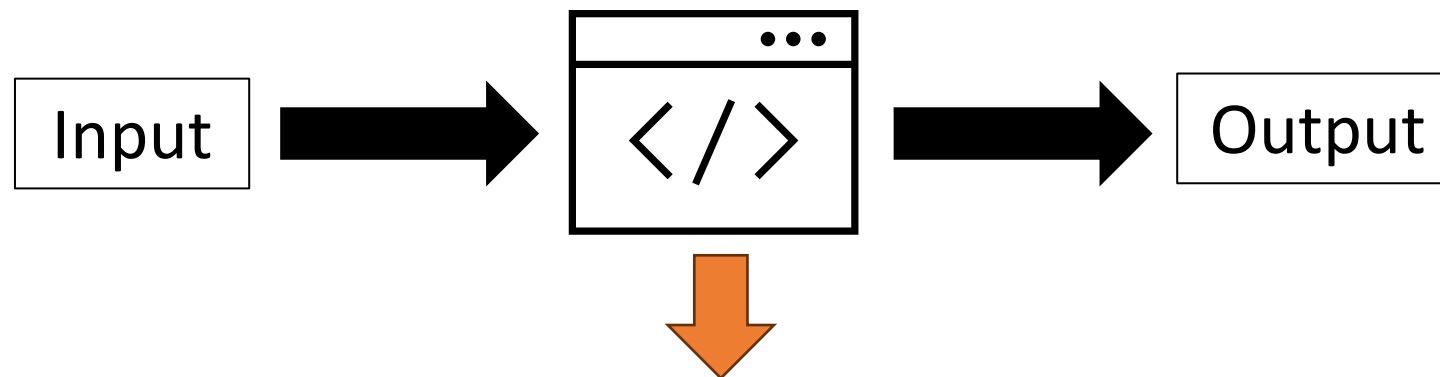
# What is Program Analysis?

Automated analysis of program to find “facts” about the program, e.g.,:

- Programming errors
- Performance errors
- Security vulnerabilities

Different kinds of software bugs

Program



Program “facts” based on reasoning about inputs and observed outputs



# Static vs. Dynamic Analysis

Static	Dynamic
Analyze source code, byte code, or binary	Analyze program execution traces
Typically: <ul style="list-style-type: none"> <li>• Consider all inputs</li> <li>• Overapproximate possible behavior</li> </ul>	Typically: <ul style="list-style-type: none"> <li>• Consider given inputs</li> <li>• Underapproximate possible behavior</li> </ul>
E.g., Compilers, Linters	E.g., invariant detection tools, automated testing

# Example-1

```
// JavaScript
var r = Math.random(); // value in [0, 1)
var out = "yes";
if (r < 0.5)
    out = "no";
if (r == 1)
    out = "maybe"; // infeasible branch
console.log(out)
```

## What are the possible outputs?

- **Static Analysis**
  - Considering all the paths that are feasible based on limited knowledge, possible outputs are: "yes", "no", "maybe"
  - **Overapproximation (Unsound but Complete)**

# Example-1

```
// JavaScript
var r = Math.random(); // value in [0, 1)
var out = "yes";
if (r < 0.5)
    out = "no";
if (r == 1)
    out = "maybe"; // infeasible branch
console.log(out)
```

## What are the possible outputs?

- **Static Analysis**
  - Considering all the paths that are feasible based on limited knowledge, possible outputs are: "yes", "no", "maybe"
  - **Overapproximation (Unsound but Complete)**
- **Dynamic Analysis**
  - Considering some number of program executions, possible outputs are: "yes" (assuming  $r \geq 0.5$  in those executions)
  - **Underapproximation (Sound but Incomplete)**

# Example-1

```
// JavaScript
var r = Math.random(); // value in [0, 1)
var out = "yes";
if (r < 0.5)
    out = "no";
if (r == 1)
    out = "maybe"; // infeasible branch
console.log(out)
```

## What are the possible outputs?

- **Static Analysis**
  - Considering all the paths that are feasible based on limited knowledge, possible outputs are: "yes", "no", "maybe"
  - **Overapproximation (Unsound but Complete)**
- **Dynamic Analysis**
  - Considering some number of program executions, possible outputs are: "yes" (assuming  $r \geq 0.5$  in those executions)
  - **Underapproximation (Sound but Incomplete)**
- **Desired Output**
  - Considering *actually* feasible paths, possible outputs are: "yes", "no"
  - **Sound and Complete**

# Example-2

```
// JavaScript  
var r = Math.random(); // value in [0, 1)  
var out = r * 2;  
console.log(out)
```

**What are the possible outputs?**

# Example-2

```
// JavaScript  
var r = Math.random(); // value in [0, 1)  
var out = r * 2;  
console.log(out)
```

## What are the possible outputs?

- Static Analysis
  - Considering all the paths that are feasible based on limited knowledge, possible outputs are: *All numbers*
  - **Overapproximation (Unsound but Complete)**

# Example-2

```
// JavaScript  
var r = Math.random(); // value in [0, 1)  
var out = r * 2;  
console.log(out)
```

## What are the possible outputs?

- **Static Analysis**
  - Considering all the paths that are feasible based on limited knowledge, possible outputs are: *All numbers*
  - **Overapproximation (Unsound but Complete)**
- **Dynamic Analysis**
  - Considering some number of program executions, possible outputs are: 1.2, 0.45, 1.1 (values in range [0,2))
  - **Underapproximation (Sound but Incomplete)**

# Example-2

```
// JavaScript  
var r = Math.random(); // value in [0, 1)  
var out = r * 2;  
console.log(out)
```

Exploring all possible outputs  
is practically infeasible

This is true for most real-  
world programs

## What are the possible outputs?

- **Static Analysis**
  - Considering all the paths that are feasible based on limited knowledge, possible outputs are: *All numbers*
  - **Overapproximation (Unsound but Complete)**
- **Dynamic Analysis**
  - Considering some number of program executions, possible outputs are: 1.2, 0.45, 1.1 (values in range [0,2))
  - **Underapproximation (Sound but Incomplete)**
- **Desired Output**
  - **Sound and Complete**



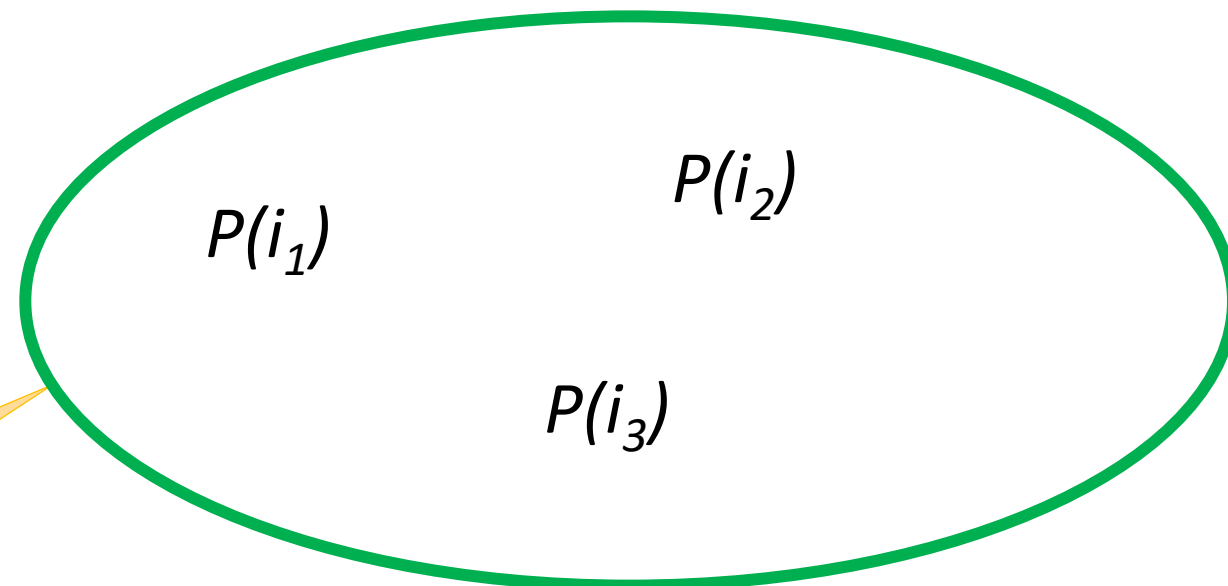
# Soundness vs. Completeness

**Program  $P$**

**Input  $i_k$**

**Program behavior  $P(i_k)$**

All possible behaviors

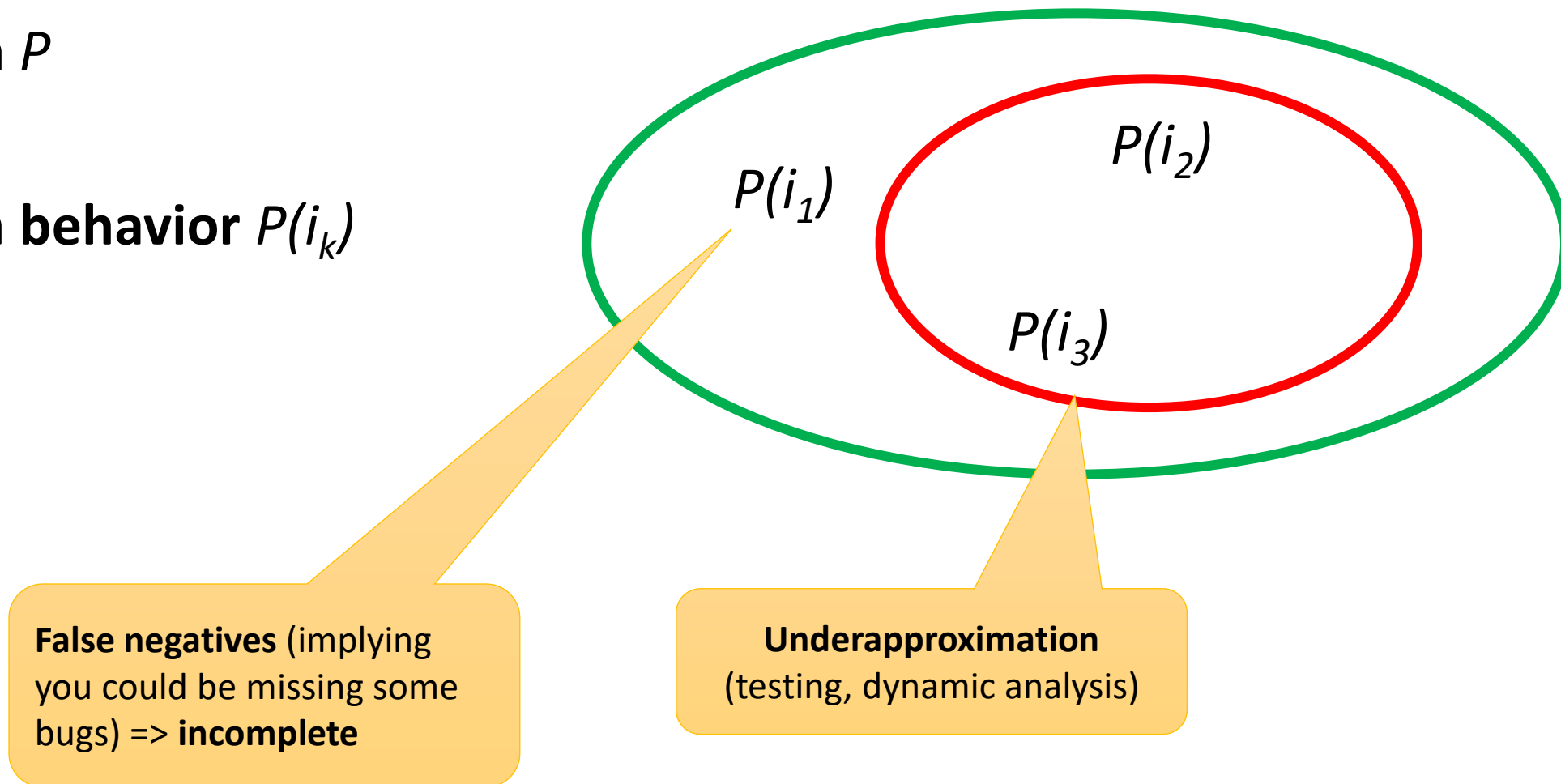


# Soundness vs. Completeness

**Program  $P$**

**Input  $i_k$**

**Program behavior  $P(i_k)$**

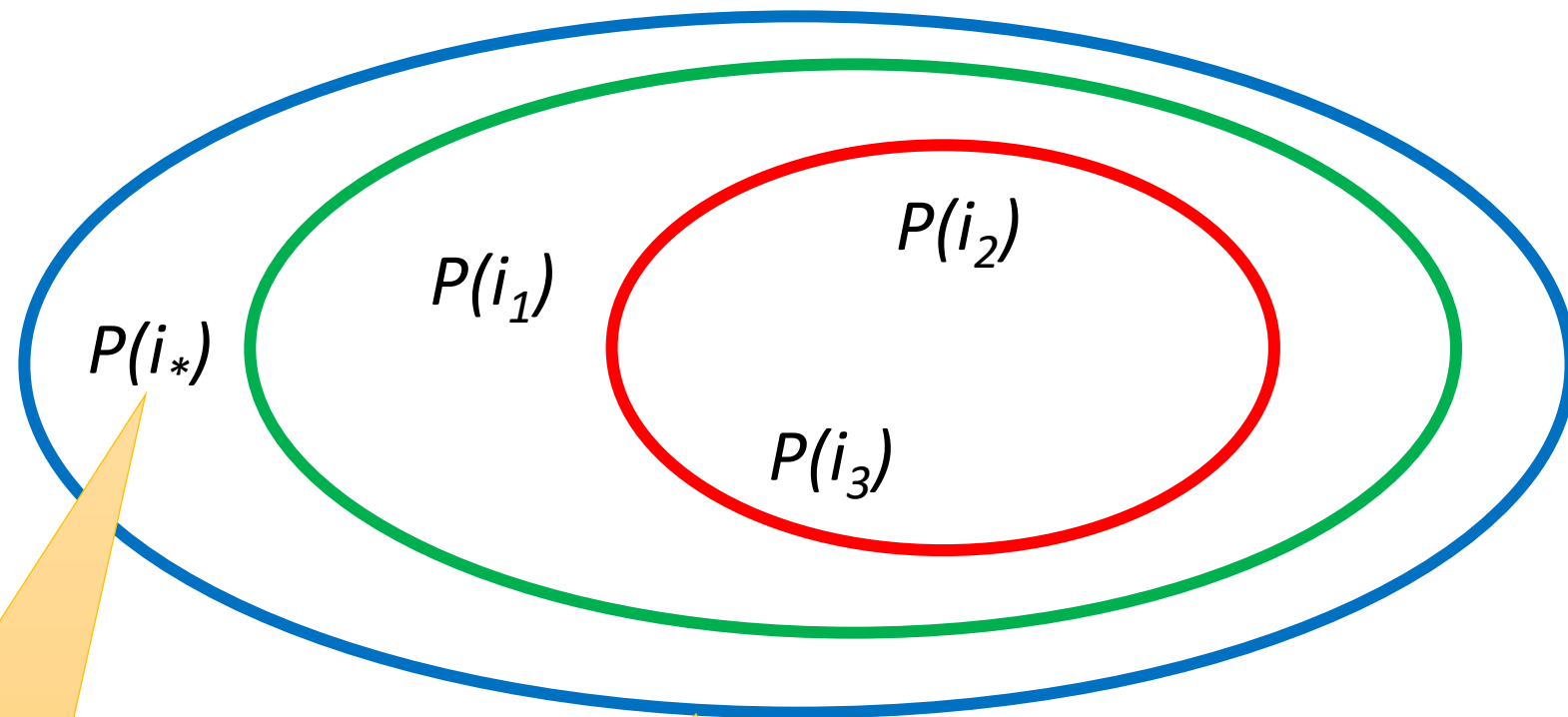


# Soundness vs. Completeness

**Program  $P$**

**Input  $i_k$**

**Program behavior  $P(i_k)$**



**False positives** (warning about bugs that can never occur) => **unsound**

**Overapproximation**  
(static analysis)

# Test Generation

- Creates inputs automatically
- Enables Dynamic Analysis (requires input to run the program)
- Example techniques:
  - Generate unit tests
    - Input = sequence of method calls
  - UI-level tests
    - Input = sequence of UI events
  - Fuzz-test a compiler
    - Input = program

# How Does All This Help Me?

- Learn to use program analysis and testing tools
  - Improve the quality of your code
- Understand program analysis
  - Helps to better understand and comprehend program behavior
- Create your own techniques to improve compilers, software quality tools, IDEs

# Ariane 5 Disaster



# Next Class

## Detecting Program Invariants