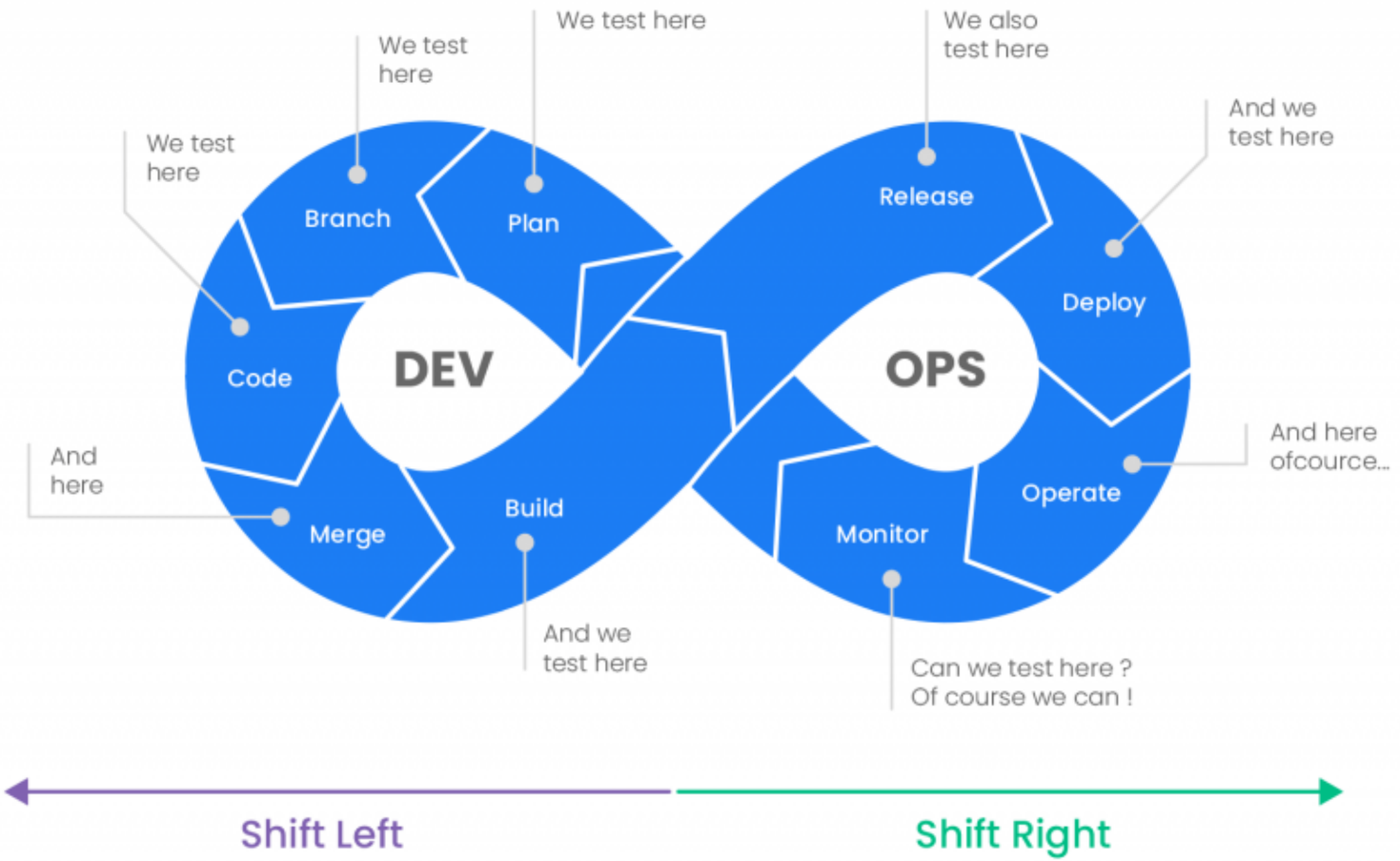
Outline

Planned topics for this lesson:

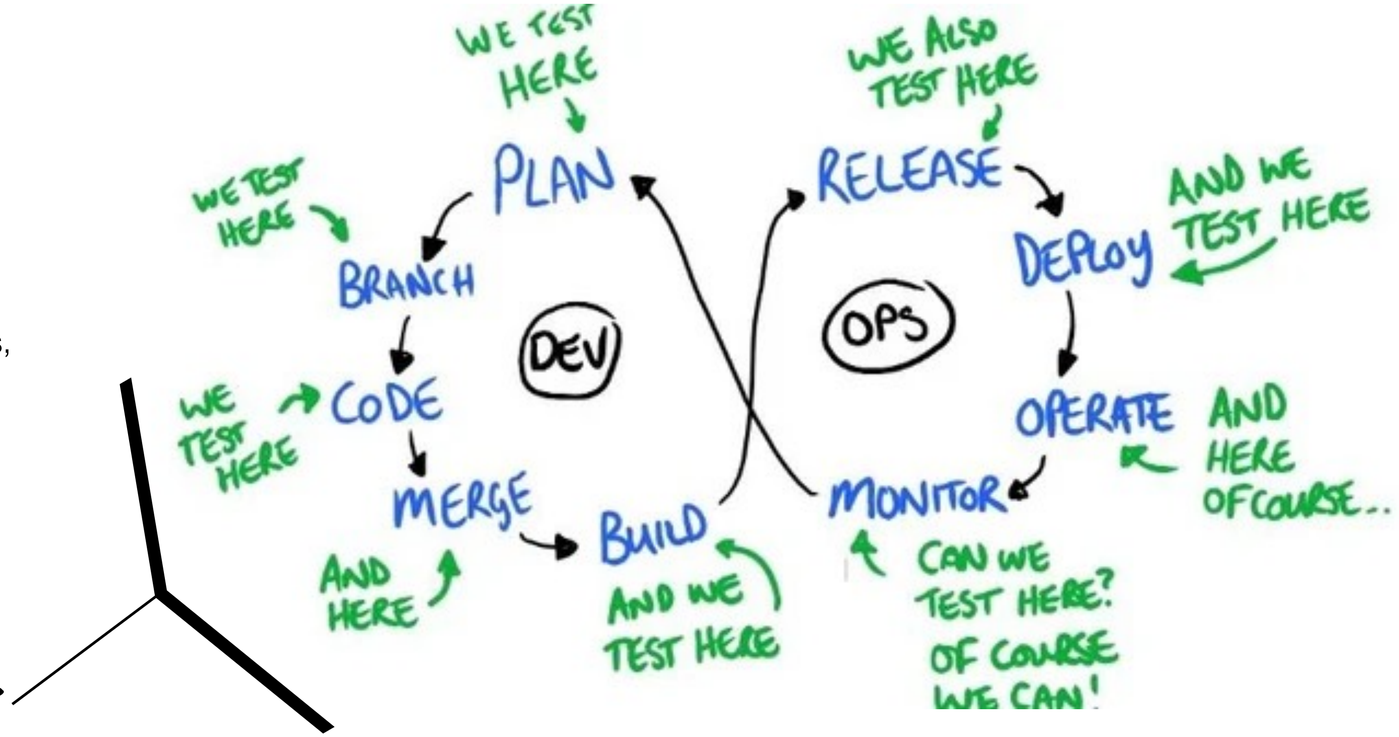
- What is Software Quality Assurance (SQA)
- Software Quality Metrics
  - **PLEASE USE THIS TOOL RESPONSIBLY**
- How to Measure and Improve Code Quality?



Software Quality Assurance

What is it?

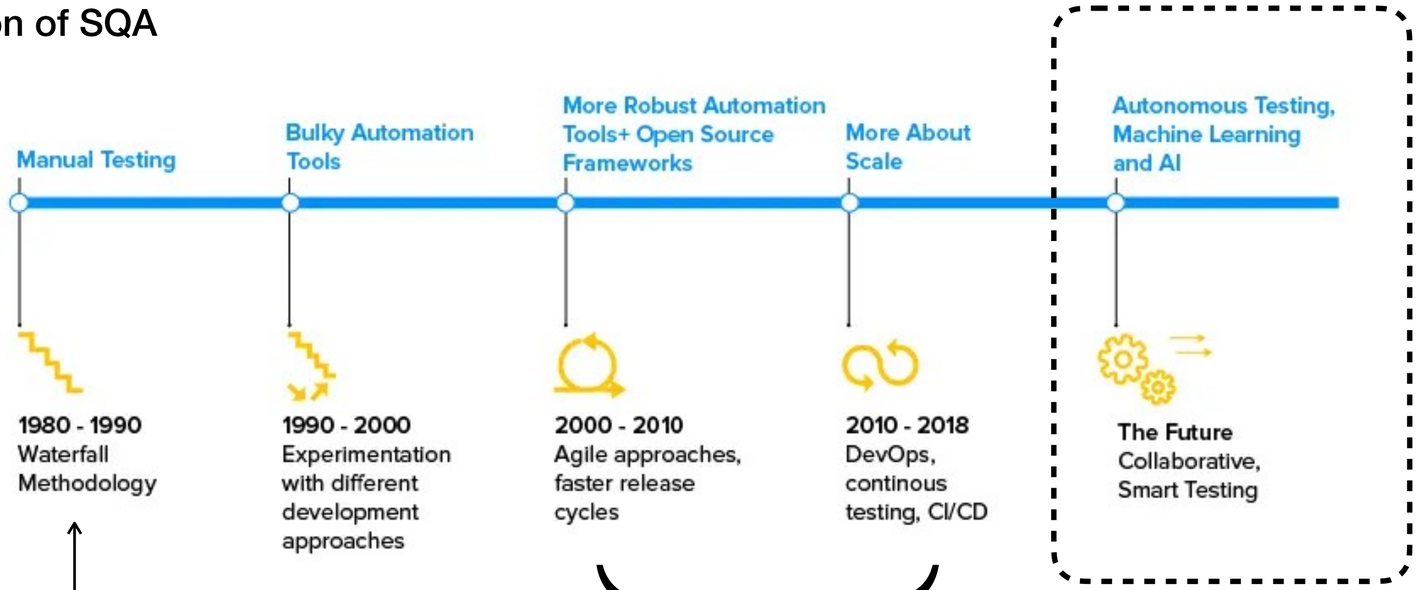
- SQA is a systematic process that ensures software products meet quality standards at each phase of the development lifecycle.
- It's **more than just testing**.
- it involves reviewing the entire development process, including standards, tools, procedures, and methodologies, to ensure the final product meets both functional and non-functional requirements.



SQA IS NOT JUST ABOUT TESTING AT THE END BUT IS EMBEDDED IN EACH PHASE OF THE SDLC.

Software Quality Assurance

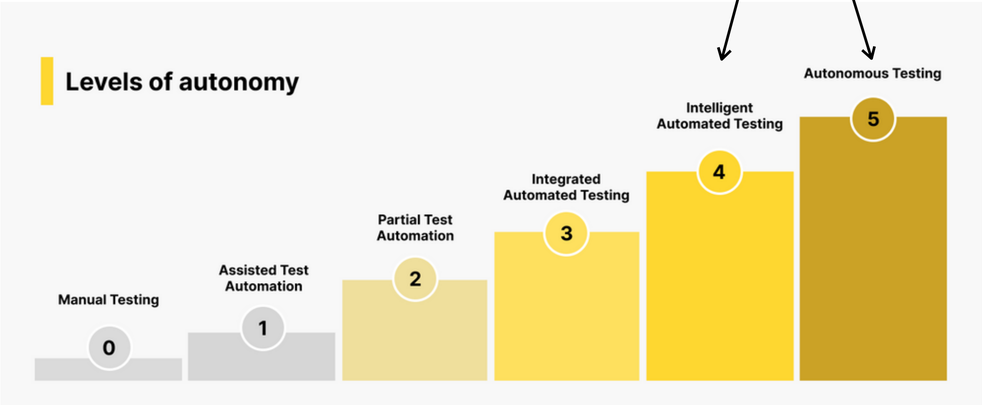
Evolution of SQA



CONDUCTED AFTER THE COMPLETION OF THE DEVELOPMENT PHASE

TEST EVERYWHERE, EVERY TIME — BECOMING A CRITICAL PILLAR IN FAST-PACED DEVELOPMENT CYCLES

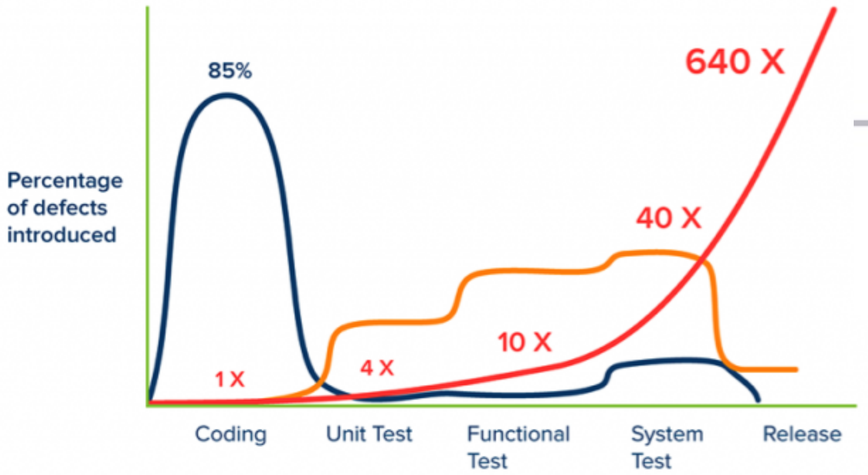
NEXT?



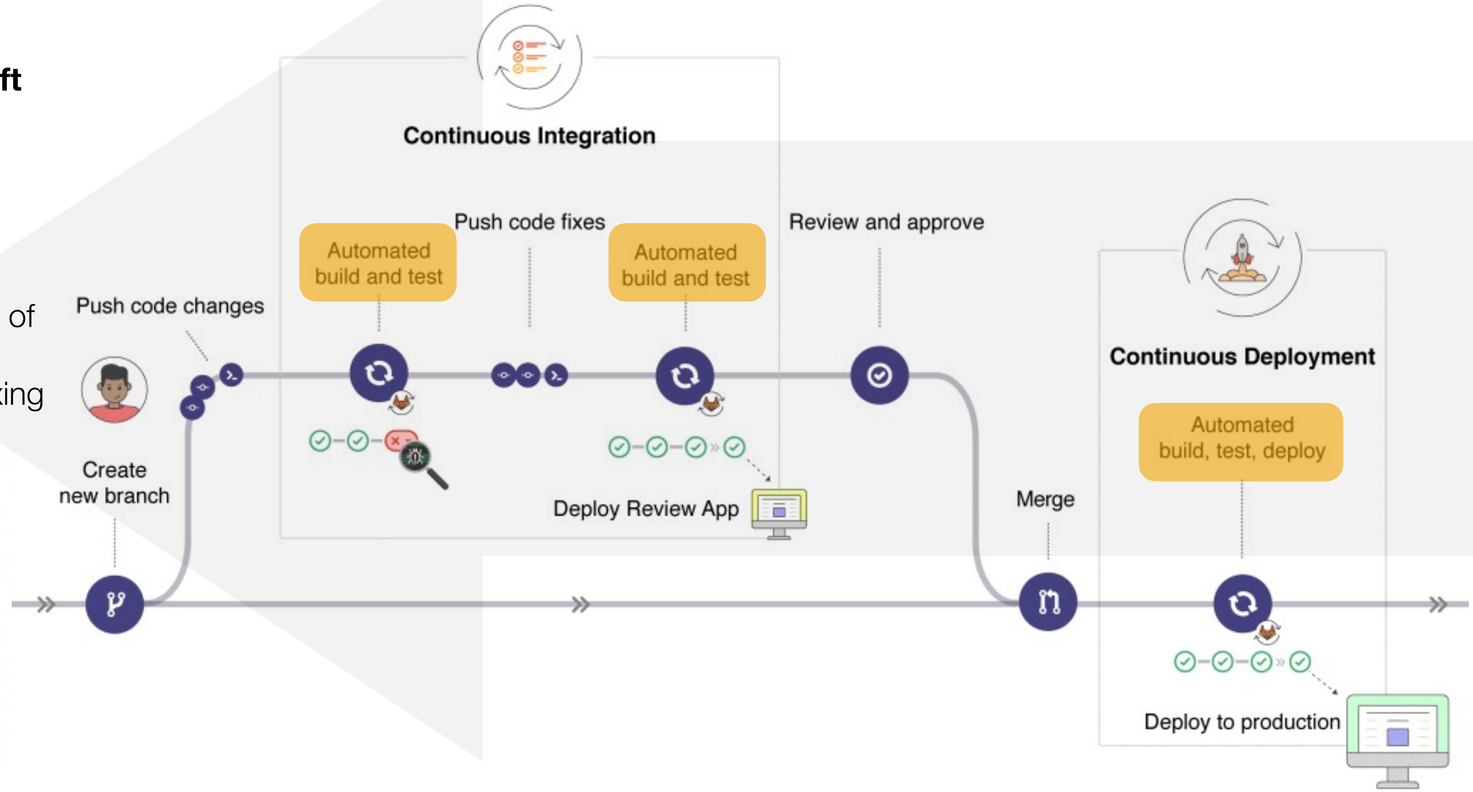
Software Quality Assurance

Shifting Left in SQA

- A major trend in modern software development is to **shift left.**
- This means performing testing and quality assurance activities earlier in the development process.
- By addressing issues during design and coding, instead of waiting for later stages like user acceptance testing or production, teams can reduce the cost and impact of fixing defects



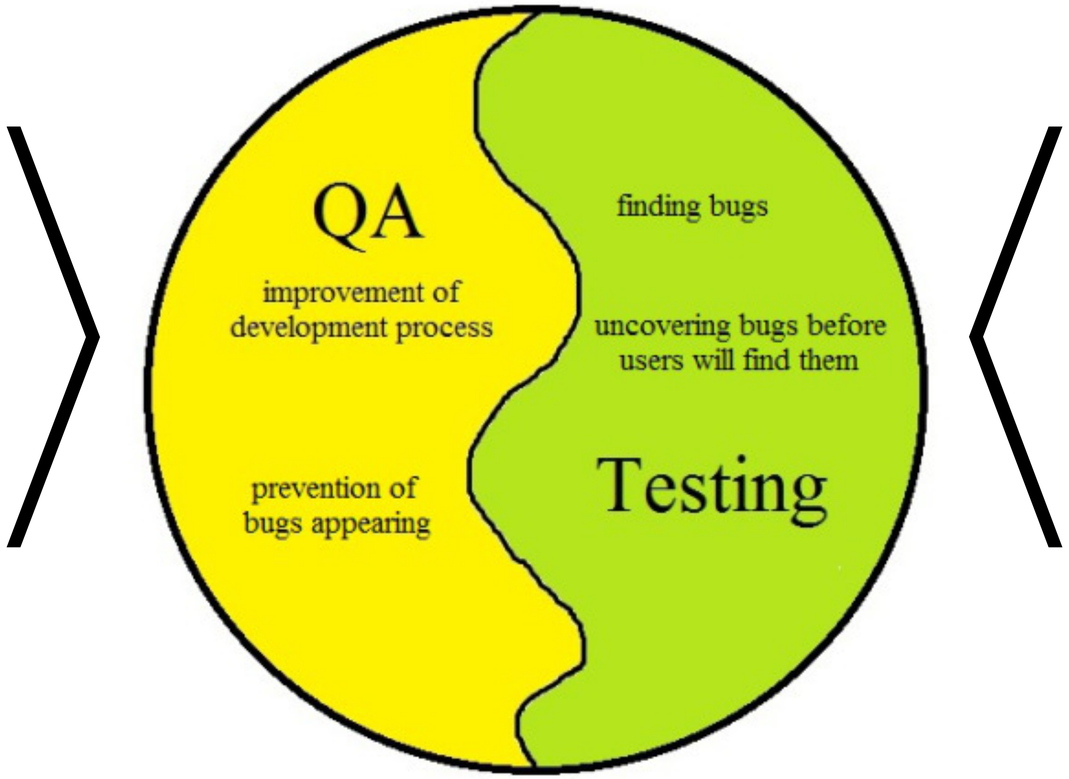
Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*



SQA vs. ST

QA + Testing = good software

- Quality Assurance (QA) is primarily focused on improving and maintaining the integrity of the process that produces the software.
- Rather than looking at the end product, QA evaluates the steps and practices that are followed to create it.
- By refining how the software is built, it reduces the likelihood of defects and enhances the overall quality.



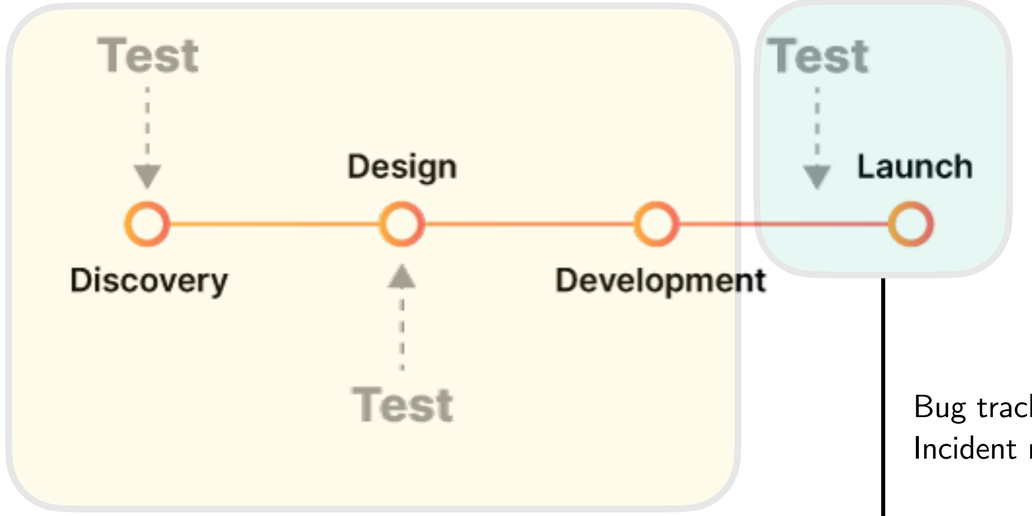
- Software Testing (ST) is concerned with evaluating the final product.
- It identifies defects by validating the functionality, security, and performance of the software.
- Whereas QA seeks to prevent issues, software testing reactively catches defects in the product.
- It focuses on ensuring that the end result meets the original specifications and works as intended.

Proactive vs. Reactive

Approaches in SQA

Proactive strategy

- In proactive software quality assurance, the focus is on preventing defects from being introduced into the system.
- This involves activities such as code reviews, writing tests before development (e.g., TDD), and using tools to analyse the code quality upfront, even before execution.
- The aim is to catch potential issues early, reducing the cost of fixing them later.



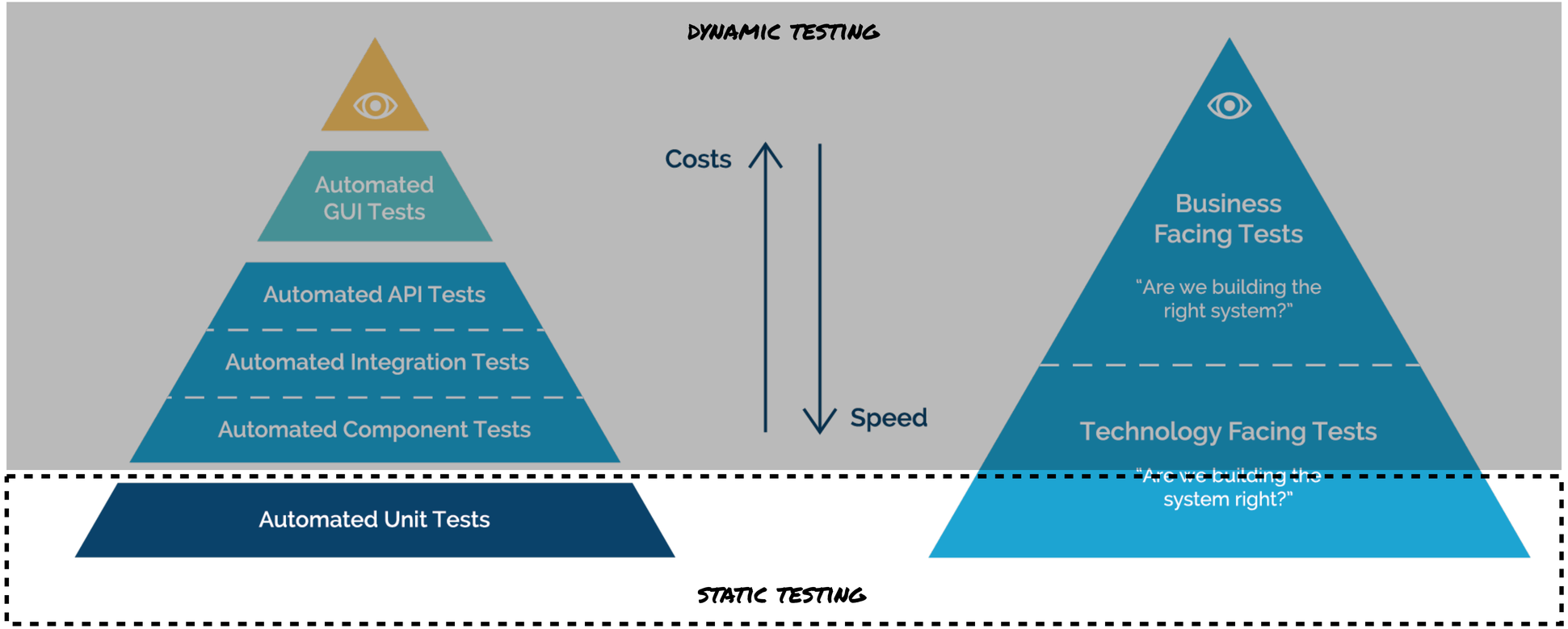
Code reviews,
TDD, etc.

Bug tracking,
Incident reports, etc.

Reactive strategy

- Reactive quality assurance occurs after defects have been introduced, with the focus on identifying and fixing issues after they occur.
- This involves testing the system to uncover defects, tracking bugs, and fixing problems through activities like regression testing and post-deployment incident responses.

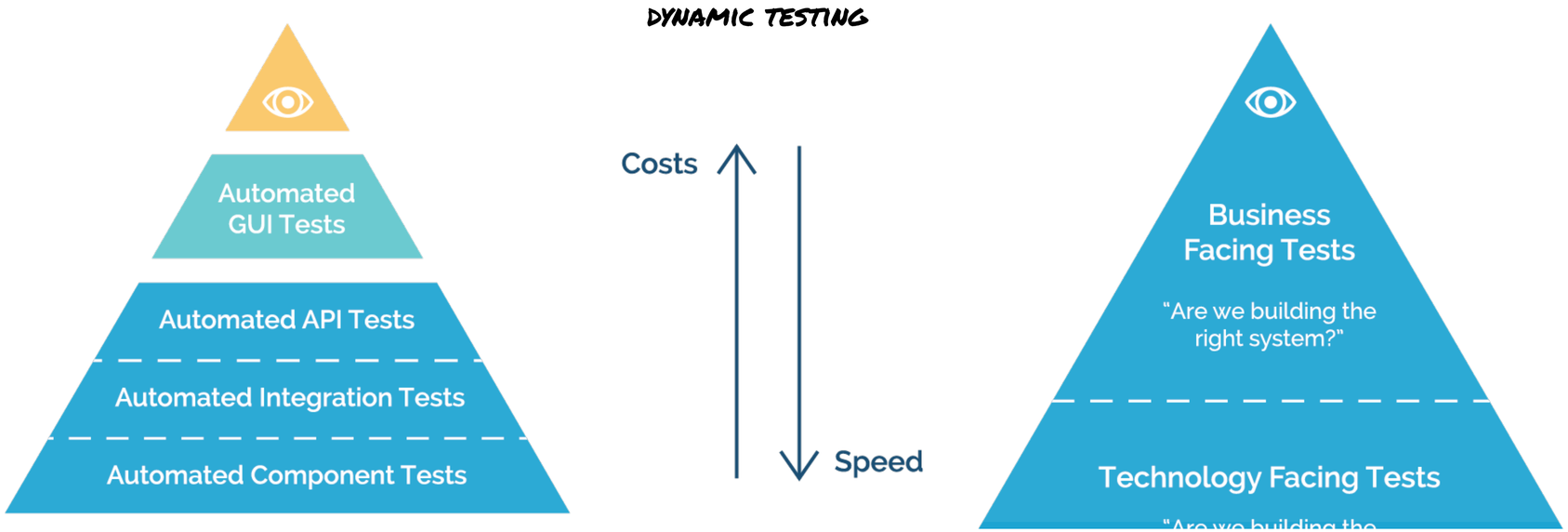
Static vs. Dynamic



Static vs. Dynamic

- Static testing involves examining the code or software artefacts (like design documents) without executing the code.
- Key Techniques:
  - Manual inspection of code for potential defects.
  - Static analysis tools (e.g., SonarQube) analyse the source code for syntax errors, vulnerabilities, and adherence to coding standards.
- Finds issues early in the development cycle, like coding errors or security vulnerabilities, before running the application.

Static vs. Dynamic



- Dynamic testing involves executing the code to validate its behavior against expected outcomes.
- Key Techniques:
  - Unit Testing, Integration Testing, System Testing.
  - Functional & Non-Functional Testing
- Identifies real-time issues in a running application, such as memory leaks or functionality bugs.

Software Quality Metrics

Code Coverage

- Code coverage measures how much of your source code is tested by automated tests.
- It identifies untested code paths and functions.
- Key Types of Coverage:
  - ✓ **Line Coverage** — Measures the percentage of executed lines of code during a test suite.
  - ✓ **Branch Coverage** — Evaluates whether both true and false branches of every control structure (like if-statements) are tested.
  - ✓ **Function Coverage** — Verifies that all methods/functions are executed at least once during testing.
- High code coverage can signal robust testing but does not guarantee code quality.
- It ensures that critical code paths are not left untested, which could lead to defects in production.

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	221	84%	2970/3513	81%	859/1060	1.727
junit.extensions	6	82%	52/63	87%	7/8	1.25
junit.framework	17	76%	399/525	90%	139/154	1.605
junit.runner	3	49%	77/155	41%	23/56	2.225
junit.textui	2	76%	99/130	76%	23/30	1.686
org.junit	14	85%	196/230	75%	68/90	1.655
org.junit.experimental	2	91%	21/23	83%	5/6	1.5
org.junit.experimental.categories	5	100%	67/67	100%	44/44	3.357
org.junit.experimental.max	8	85%	92/108	86%	26/30	1.969
org.junit.experimental.results	6	92%	37/40	87%	7/8	1.222
org.junit.experimental.runners	1	100%	2/2	N/A	N/A	1
org.junit.experimental.theories	14	96%	119/123	88%	37/42	1.674
org.junit.experimental.theories.internal	5	88%	98/111	92%	39/42	2.29
org.junit.experimental.theories.suppliers	2	100%	7/7	100%	2/2	2
org.junit.internal	11	94%	149/157	94%	53/56	1.947
org.junit.internal.builders	8	98%	57/58	92%	13/14	2
org.junit.internal.matchers	4	75%	40/53	0%	0/18	1.391
org.junit.internal.requests	3	96%	27/28	100%	2/2	1.429
org.junit.internal.runners	18	73%	306/415	63%	82/130	2.155
org.junit.internal.runners.model	3	100%	26/26	100%	4/4	1.5
org.junit.internal.runners.rules	1	100%	35/35	100%	20/20	2.111
org.junit.internal.runners.statements	7	97%	92/94	100%	14/14	2
org.junit.matchers	1	9%	1/11	N/A	N/A	1
org.junit.rules	20	89%	203/226	96%	31/32	1.444
org.junit.runner	12	93%	150/161	88%	30/34	1.378
org.junit.runner.manipulation	9	85%	36/42	77%	14/18	1.632
org.junit.runner.notification	12	100%	98/98	100%	8/8	1.162
org.junit.runners	16	98%	321/327	96%	95/98	1.737
org.junit.runners.model	11	82%	163/198	73%	73/100	1.918

Report generated by Cobertura 1.9.4.1 on 12/22/12 2:25 PM.

Code Coverage

Line vs. Branch

- Line coverage measures how many statements you took
  - A statement is usually a line of code, not including comments, conditionals (if-then-else), and method headers)
- Branch coverage checks if you took the true and false branch for each conditional (if, for, while).
  - You'll have twice as many branches as conditionals

```java
public void setBalance(int balance)
{
    if(balance>100000)
    {
        System.out.println("You are very rich");
        this.balance = balance;
    }
    else
    {
        System.out.println("You are well off");
        this.balance = balance;
    }
}
```

METHOD FOR TESTING

```java
public class BalanceTest {

    @Test
    public void evaluatesSetBalance() {
    Account acc = new Account("Enda", 0);
    acc.setBalance(100001);
    assertEquals(acc.getBalance(), 100001);
  }

}
```

UNIT TEST

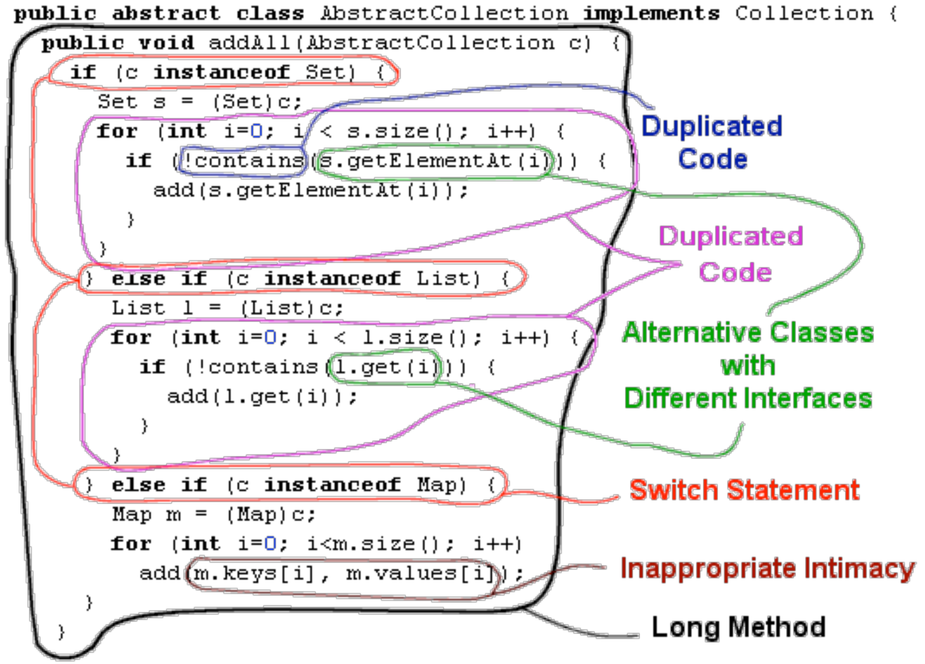
CALCULATE THE LINE COVERAGE AND BRANCH COVERAGE – WHAT DO YOU THINK?

50/50 , 50/25, , 62/50 , 100/50

Software Quality Metrics

Code Smells

- Code smells refer to any symptom in the source code that indicates deeper problems or technical debt.
- Examples of Common Code Smells:
  - Long Methods — Functions that are excessively long, making them difficult to understand and maintain.
  - Duplicated Code — Code that is repeated across the codebase, increasing maintenance effort and risk of errors.
  - Large Classes — Classes that have too many responsibilities, violating the Single Responsibility Principle (SRP).
  - Excessive Comments — Too many comments may indicate code that is hard to understand or not self-explanatory.
- Code smells increase the complexity of the codebase, making it harder to maintain and test.
- Addressing them early improves readability and maintainability.



Software Quality Metrics

Cyclomatic Complexity

- Cyclomatic complexity is a software metric used to indicate the complexity of a program by measuring the number of independent paths through the source code.
- It counts the number of decision points (e.g., if, for, while statements) in a function, method, or code block. The higher the number, the more complex the code.

$$CC = E - N + 2P$$

- E, Number of edges in the control flow graph.
- N, Number of nodes in the control flow graph.
- P, Number of connected components or exit points in the code

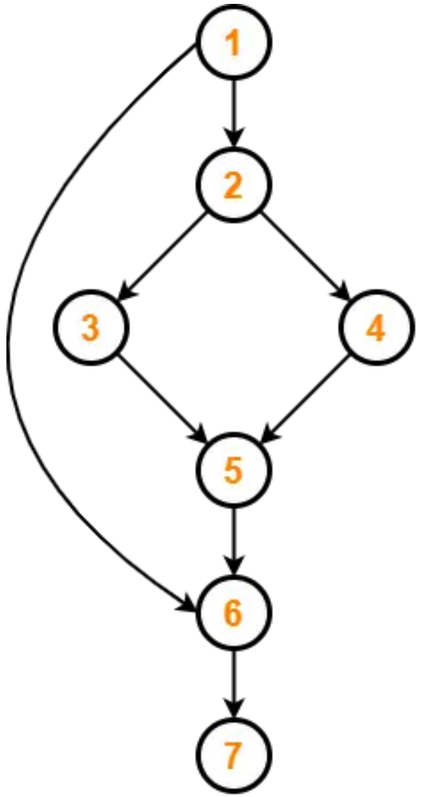
- What Does It Mean?

Low Complexity	CC = 1 to 10	Easier to understand and maintain
Moderate Complexity	CC = 11 to 20	Increased testing effort, some challenges in maintainability
High Complexity	CC = 21 to 50	Difficult to maintain and test; higher risk for bugs
Very High Complexity	CC > 50	Unmanageable, likely requiring refactoring

Software Quality Metrics

Cyclomatic Complexity

```
IF A = 354
    THEN IF B > C
            THEN A = B
            ELSE A = C
    END IF
END IF
PRINT A
```



Control Flow Graph

$$= E - N + 2P$$
$$= 8 - 7 + 2(1)$$
$$= 3$$

The Tools

Code Coverage

- **JACOCO** Java Code Coverage — A widely used tool for measuring test coverage in Java projects. It integrates seamlessly with popular build tools like Maven and Gradle.
-  — A popular tool for JavaScript applications. It provides coverage reports for Mocha, Jest, and other testing frameworks.

Cyclomatic Complexity

- **sonarqube** — A static analysis tool that integrates with multiple languages and IDEs to calculate cyclomatic complexity, alongside other code quality metrics.
- 

A tool that supports complexity analysis and gives feedback on issues like high cyclomatic complexity in code

Code Smell

- **sonarlint** — A plugin that integrates with IDEs to analyse code on the fly for code smells (e.g., dead code, long methods, poor naming conventions)

Apple's SSL Bug — 2014

Case Study

- A bug in the Apple's SSL implementation that affected iOS 6, iOS 7, and OS X 10.9
- It was undiscovered for about TWO (2) years
- The bug compromised SSL-based HTTPS connections that use specifically **Diffie-Hellman (DH) key exchange** during the initial handshake phase
  - It allowed an attacker to launch a Man-in-the-Middle attack and compromise the secure communication between client and server

→

- Both client and server exchange public kets, which are used by each side to calculate a common session key
- To deter man-in-the-middle attack, the server digitally signs its public key before sending it to the client
- The client in turn validates the signature before accepting the public key

Apple's SSL Bug — 2014

Case Study

- A bug in the Apple's SSL implementation that affected iOS 6, iOS 7, and OS X 10.9
- It was undiscovered for about TWO (2) years
- The bug compromised SSL-based HTTPS connections that use specifically **Diffie-Hellman (DH) key exchange** during the initial handshake phase
  - It allowed an attacker to launch a Man-in-the-Middle attack and compromise the secure communication between client and server

→

- Both client and server exchange public kets, which are used by each side to calculate a common session key
- To deter man-in-the-middle attack, the server digitally signs its public key before sending it to the client
- ~~The client in turn va ates the signature before accepting the publ~~
- client simply accepted public key regardless
- man-in-the-middle attack happens !

Apple's SSL Bug — 2014

Case Study

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

These checks got skipped

Best Practices

Maintaining Code Quality

Regular Code Reviews:

- Code reviews ensure that different developers review each other's work, catching bugs or bad practices early.
- It fosters team collaboration, helps in knowledge sharing, and enforces consistency in coding standards.
- Encourage pair programming or peer reviews as part of the development culture.

Automated Testing in CI/CD:

- Automated tests in CI/CD pipelines ensure that new code doesn't break existing functionality.
- Write unit tests, integration tests, and use code coverage metrics.
- Automating tests saves time, reduces manual testing effort, and quickly identifies bugs after every code change.

Monitoring & Continuous Improvement:

- Implement dashboards with tools like **SonarQube**, **Jenkins**, or **CircleCI** to monitor ongoing code quality metrics.
- Set thresholds for when alerts should be triggered if quality metrics degrade over time.

Best Practices

Maintaining Code Quality

```java
public class OrderService {
    public void processOrder(Order order) {
        if (order != null && order.getStatus() == Status.NEW) {
            System.out.println("Processing order: " + order.getId());
            if (order.getItems().size() > 0) {
                for (Item item : order.getItems()) {
                    if (item.isInStock()) {
                        System.out.println("Shipping item: " + item.getName());
                    } else {
                        System.out.println("Item out of stock: " + item.getName());
                    }
                }
            }
        } else {
            System.out.println("Order invalid or already processed.");
        }
    }
}
```

```java
public class OrderService {

    public void processOrder(Order order) {
        if (isOrderValid(order)) {
            processItems(order);
        } else {
            logInvalidOrder(order);
        }
    }

    private boolean isOrderValid(Order order) {
        return order != null && order.getStatus() == Status.NEW;
    }

    private void processItems(Order order) {
        order.getItems().forEach(this::processItem);
    }

    private void processItem(Item item) {
        if (item.isInStock()) {
            System.out.println("Shipping item: " + item.getName());
        } else {
            System.out.println("Item out of stock: " + item.getName());
        }
    }

    private void logInvalidOrder(Order order) {
        System.out.println("Order invalid or already processed.");
    }
}
```