L06 Software Testing

Prof. Pramod Bhatotia
Systems Research Group
https://dse.in.tum.de/

Today's learning goals

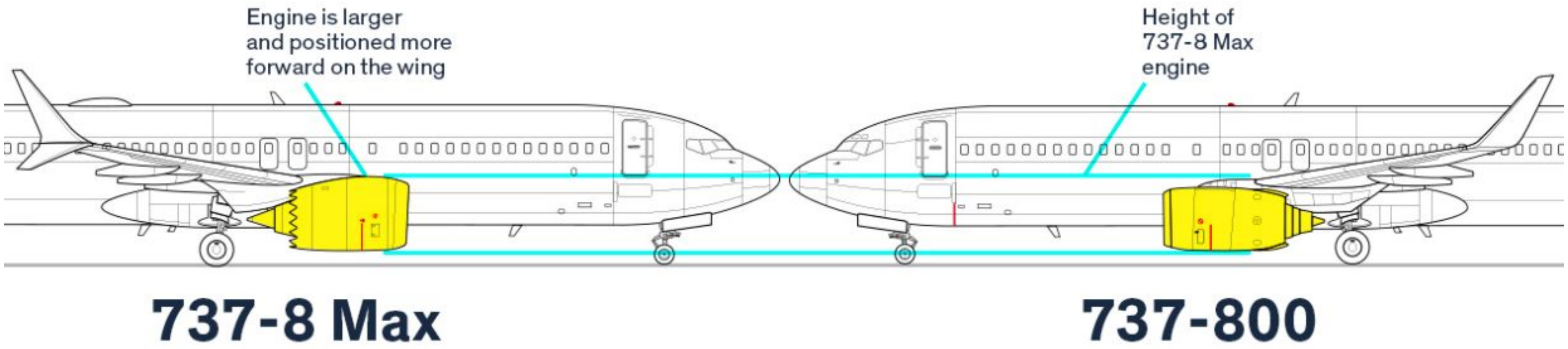
- **Part I:** Testing activities
  - Terminology
  - Unit testing
  - Integration testing
- **Part II:** Automated system testing
  - Fuzzing
  - Symbolic execution
  - Chaos Monkey
- **Part III & IV:** Model-based and object-oriented testing
  - Mock object pattern

Outline

- **Part I: Testing activities**
  - **Terminology**
  - Unit testing
  - Integration testing
- **Part II:** Automated system testing
- **Part III:** Model-based testing
- **Part IV:** Object-oriented testing

Motivation: faults are everywhere

- **Example 1: fatal crashes of Boeing 737 Max**
  - https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looksto-to a-software-developer
- **Result:** nose diving of passenger aircrafts
- **Reason:** faulty reuse of Boeing 737 software in 737 Max



Terminology

- **Failure:** any deviation of the observed behavior from the specified behavior
    - Informally also called **crash**

- **Error:** the system is in a state so that further processing can lead to a failure
    - Also called **erroneous state**

- **Fault:** the mechanical or algorithmic cause of an error
    - Informally also called **bug**

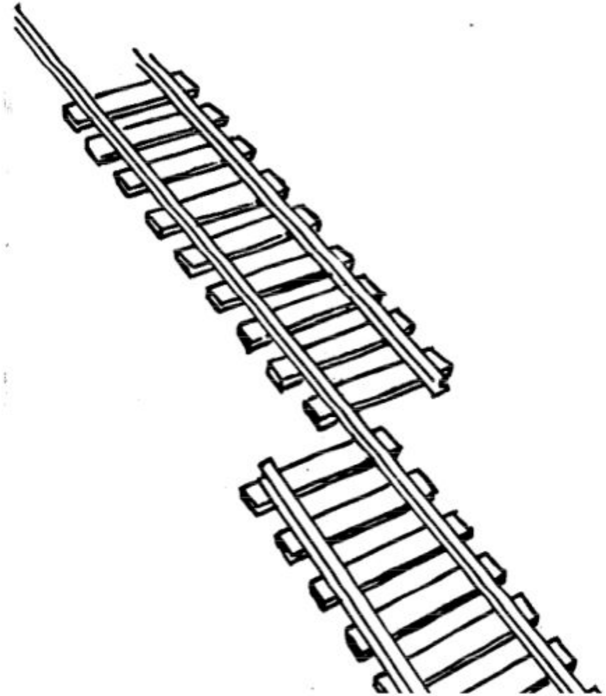
- **Validation:** activity of checking for deviations between the observed behavior of a system and its specified behavior

What is this?

A fault?

An error?

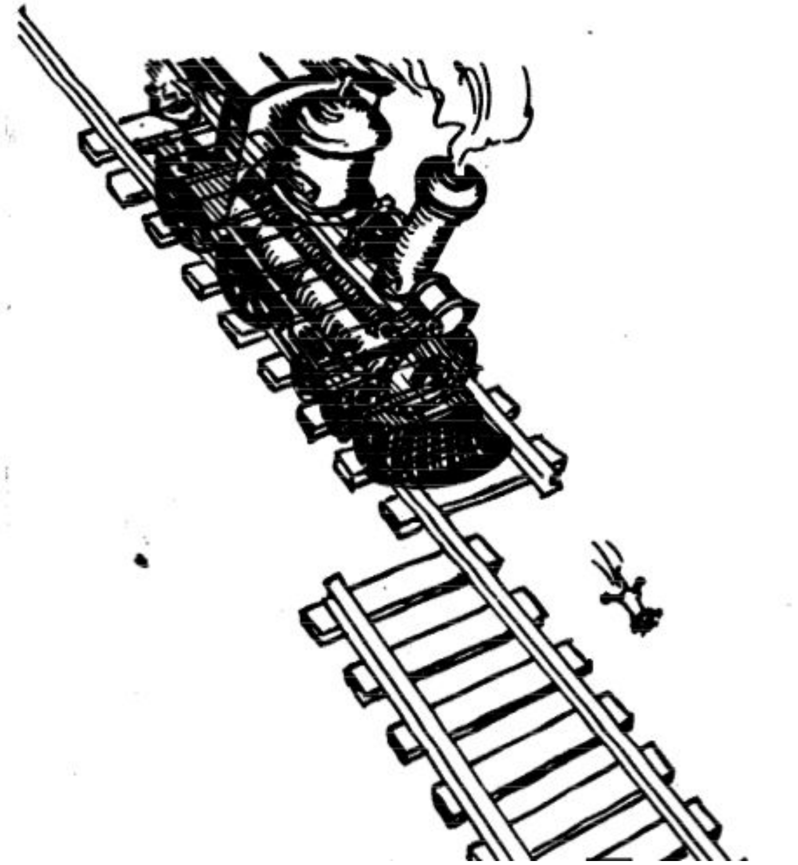
A failure?



- We need to describe the specified behavior first!

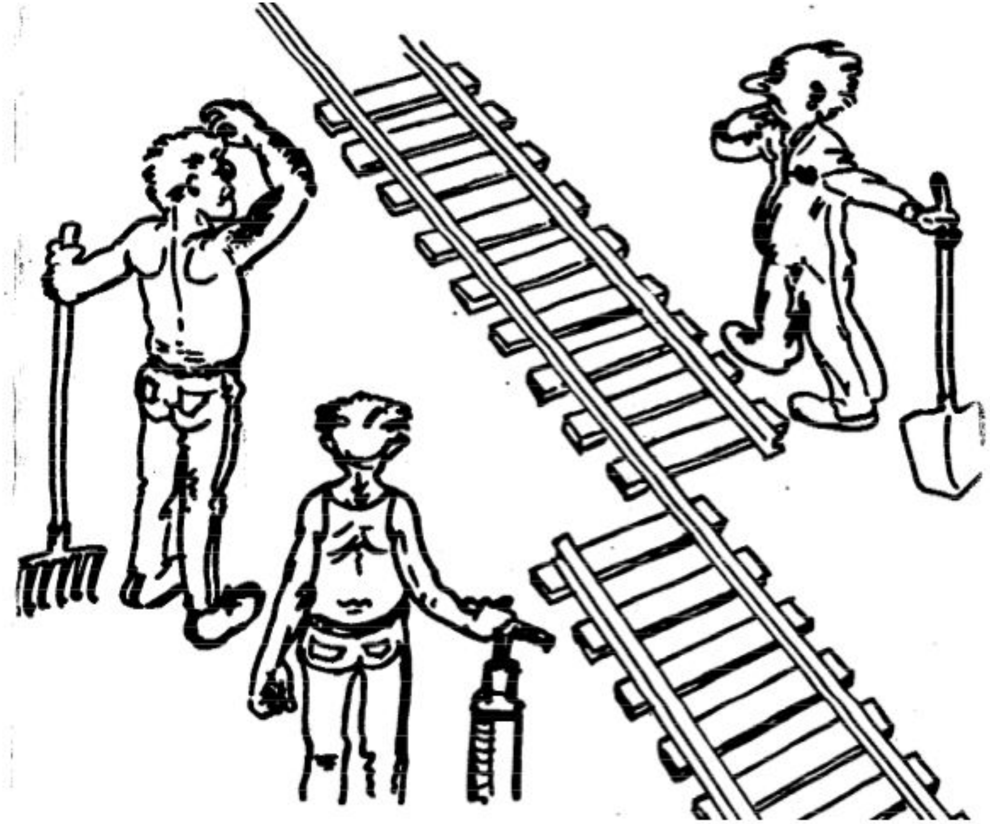
- **Requirements specification**
  - "A track shall support a moving train"

Erroneous state (“Error”)



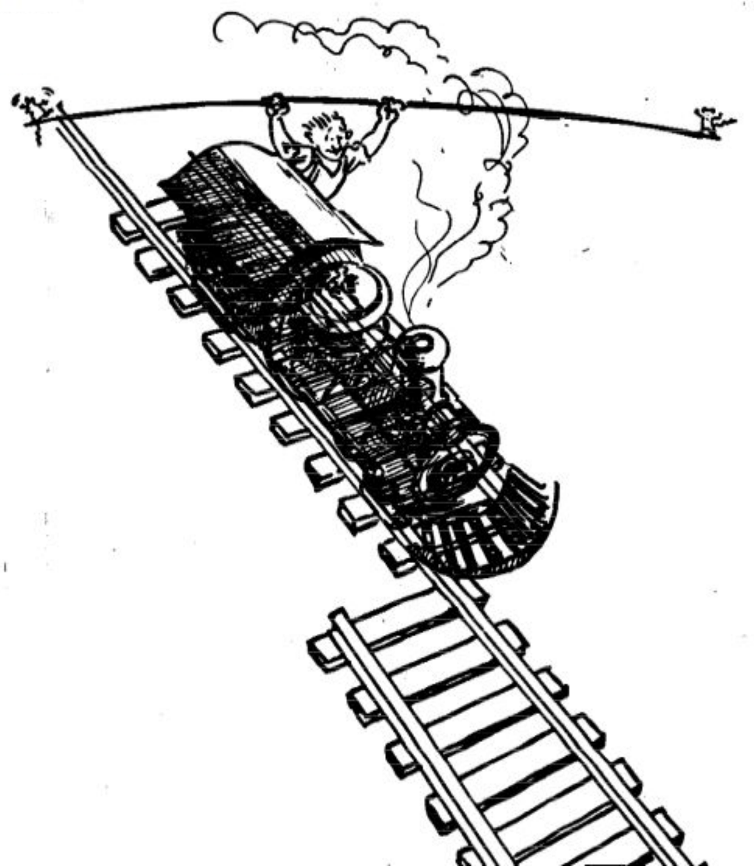
Fault

- **Algorithmic fault**
  - compass shows wrong values
- Or **usage fault**
  - wrong usage of compass
- Or **communication fault**
  - the two teams had problems talking to each other



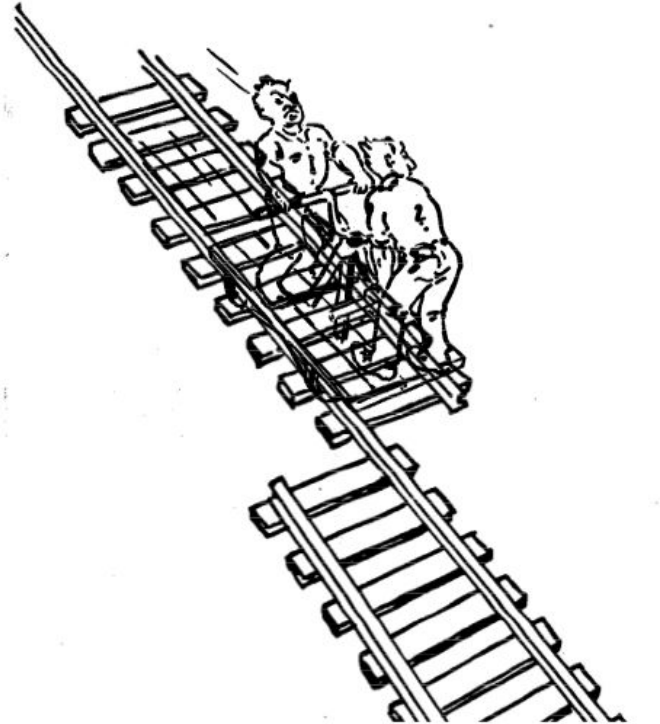
How do we deal with errors, failures and faults?

- **Declaring the bug as a feature**



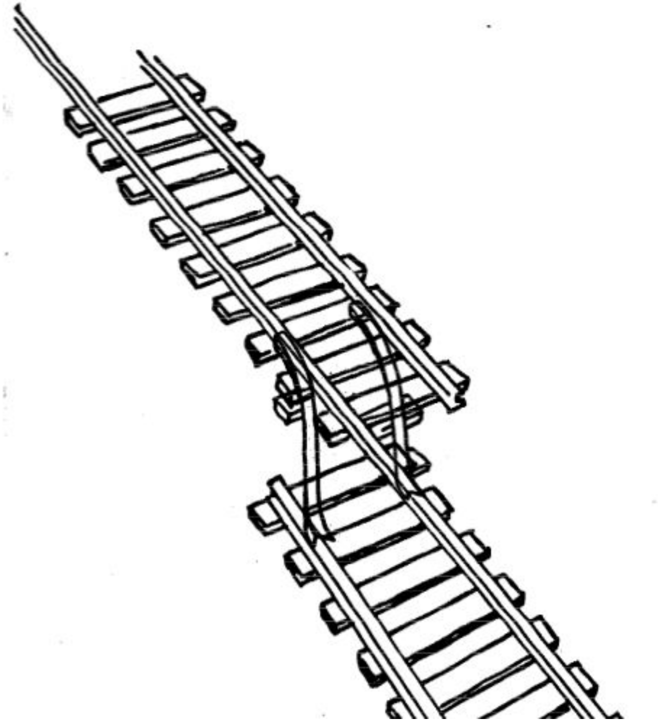
How do we deal with errors, failures and faults?

- **Testing**



How do we deal with errors, failures and faults?

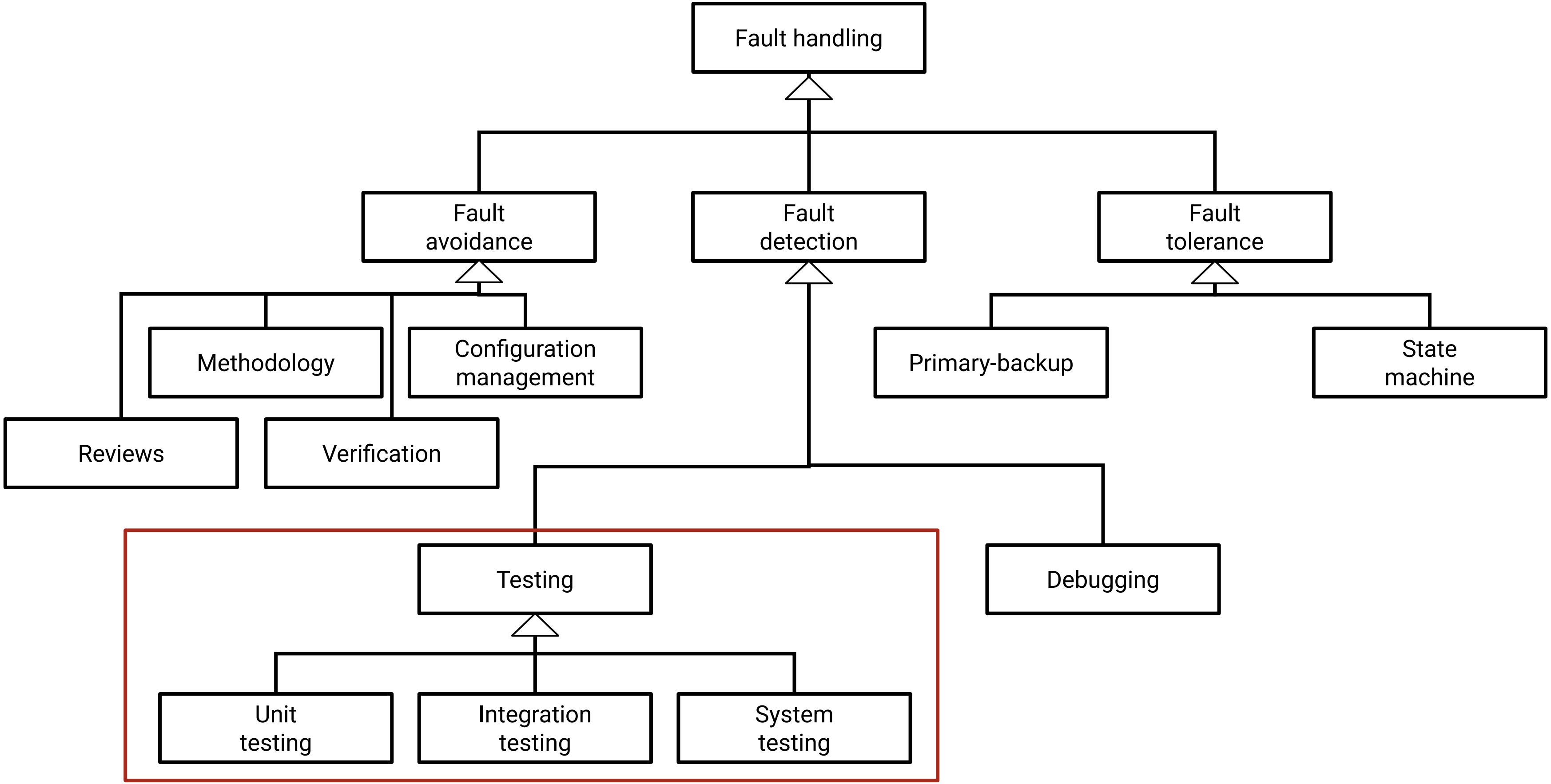
- **Bug fixing**



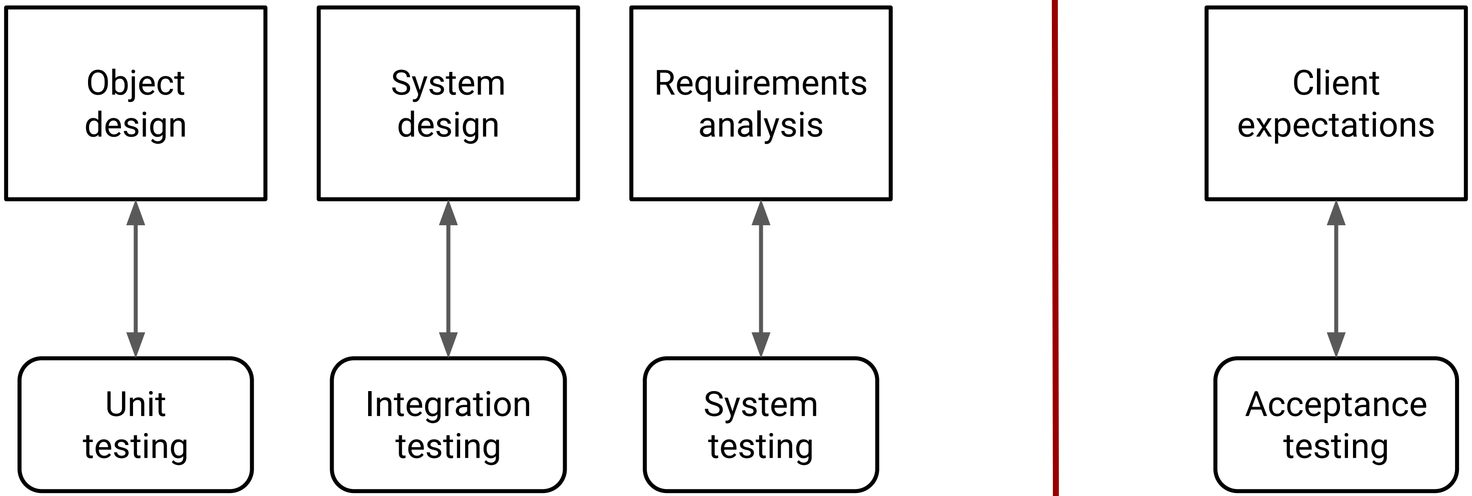
Another view on how to deal with faults

- **Fault avoidance**
  - Use a methodology to reduce complexity
  - Use configuration management to prevent inconsistencies
  - Apply verification to prevent algorithmic faults
  - Use reviews to identify faults already visible in the design
- **Fault detection**
  - **Testing:** activity to provoke failures in a planned way
  - **Debugging:** activity to find and remove the cause (fault) of an observed failure
  - **Monitoring:** activity to deliver information about state and behavior
- **Fault tolerance**
  - Primary-backup replication
  - State machine replication

Taxonomy for fault handling techniques



Taxonomy for fault handling techniques



Internal (e.g. testers in the development team)

External (e.g. client)

Types of testing (overview)

Unit testing
- The development team tests individual components (method, class, subsystem)
- **Goal:** confirm the component is correct and carries out the intended functionality

System testing
- The development team tests the entire system
  - **Functional** testing validates functional requirements
  - **Structure** testing validates the subsystem decomposition
  - **Performance** testing validates non-functional requirements
- **Goal:** determine if the system meets the requirements (functional and non-functional)

Integration testing
- The development team tests groups of subsystems (eventually the entire system)
- **Goal:** test the interfaces of the subsystems

Acceptance testing
- The client evaluates the system delivered by developers (in the target environment)
- **Goal:** demonstrate that the system meets the requirements and is ready to use

Outline

- **Part I: Testing activities**
  - ~~Terminology~~
  - **Unit testing**
  - Integration testing
- **Part II:** Automated system testing
- **Part III:** Model-based testing
- **Part IV:** Object-oriented testing

Unit testing

- A testing method where individual **units** in a program are tested
  - Procedural programming: function or procedure
  - Object-oriented programming: **class**
    - A unit can also be an **attribute**, an individual **method** or the **interface** of the class
- Unit tests are short code fragments created by developers or testers during the development process
- Unit tests form the basis of integration testing

Guidance for unit test case selection

- **Black box testing:** use **analysis knowledge** about functional requirements
  - Scenarios and use cases
  - Expected input data
  - Invalid input data
- **White box testing:** use **design and implementation knowledge** about system structure, algorithms, data
  - Control structures
  - Test branches, loops, ...
  - Classes and data structures
  - Test methods, attributes, records, fields, arrays, ..

Black box testing

- **Focus:** input / output behavior
  - If we can predict the output for any given input, the unit passes the test
    - Almost always impossible to generate all possible inputs ("test cases")
- **Goal:** reduce the number of test cases by **equivalence partitioning**
  - Divide inputs into equivalence classes
  - Choose test cases for each equivalence class
    - **Example:** if an object is supposed to accept a negative number, testing one negative number is enough
    - **Example:** for an enumerated type or array: below the range, in the range, above the range

White box testing

- **Focus:** code coverage
- Use the design & implementation knowledge to ensure
  - **Statement testing:** tests each statement
  - **Loop testing**
    - Loop to be executed exactly once / more than once
    - Cause the execution of the loop to be skipped completely
  - **Path testing:** makes sure all paths in the program are executed
  - **Branch testing** (conditional testing)
    - Ensure that each outcome in a condition is tested at least once

Comparison of white and black box testing

White box testing

- Potentially infinite number of paths
- Often tests what is done, instead of what should be done
- Cannot detect missing use cases

Black box testing

- Potential combinatorial explosion of test cases (valid & invalid data)
- Does not discover extraneous use cases ("features")

→ Both types of testing are needed

- Any test is in between white and black box testing and depends on the following
    - Number of possible logical paths
    - Nature of input data
    - Amount of computation
    - Complexity of algorithms and data structures

Execution of test cases

- There are two ways to execute test cases
  - **Manually:** testers set up the test data, run the tests and examine the results; success and/or failure of tests is determined through observation by the tester
  - **Automatically:** testers write source code to execute tests (including setup of test data) with a test system and compare the results against the oracle
- **Advantages of automated testing**

  \+ Less boring for the tester

  \+ Better test thoroughness

  \+ Reduces the cost of test execution

  \+ Indispensable for regression testing

What constitutes successful testing?

- The purpose of testing is to generate failures
- Two ways to express the success of testing a component
  - 1. The test was **successful** because **it did not generate a failure**
    - Commonly used by many programmers
    - The goal is to show the absence of failures
  - 2. The test was **successful** because **it generated a failure**
    - Karl Popper: the goal is the falsification of a model
    - "A theory in the empirical sciences can never be proven, but it can be falsified, meaning that it should be scrutinized by decisive experiments"

JUnit overview

- A Java framework (test system) for writing and executing unit tests
  - JUnit is open source: www.junit.org and https://github.com/junit-team/junit5/
- Designed initially by Kent Beck and Erich Gamma with “test first” in mind
  - Tests are written before implementing the system → Test-driven development (TDD)
  - Observe those test cases that create failures
  - Write new code or fix existing code to make the tests pass
- Uses Java **annotations** and Java **assertions**

JUnit example: testing Money class

- **Problem statement:** storing, adding, and subtracting real money in a computer is currently not possible. We can do these operations only on integers. As a result, we might accidentally add 5 Euros (€) and 7 US Dollars ($), which is, of course invalid
- **Solution:** create a **Money** class that provides the currency abstraction (encapsulating **amount** and type of **currency**)
- Functional requirements
  - Store an amount and currency
  - Add and subtract money with the same currency
  - Adding and subtracting money with different currencies is invalid (e.g. €6+ $5 is invalid)

Money class

```java
public class Money {

      private final int amount;
      private final Currency currency;

      public Money(int amount, Currency currency) {
            this.amount = amount;
            this.currency = currency;
      }

      public int amount() {
            return amount;
      }

      public Currency currency() {
            return currency;
      }

      //...
}
```

Money class

```java
public class Money {

	//...
	public Money add(Money money) {
		return new Money(amount() + money.amount(), getCurrency());
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj) {
			return true;
		}

		if (obj == null) {
			return false;
		}
		if (getClass() != obj.getClass()) {
			return false;
		}
		Money other = (Money) obj;
		return amount == other.amount && currency == other.currency;
	}

}
```

A unit test for the add() method

- The unit test for the class **Money** should test the **add()** method
- Below is an **example** test for the addition of money

```
import org.junit.jupiter.api.Test;

class MoneyTest {

    @Test
    void testSimpleAdd() {
        Money m12CHF = new Money(12, Currency.CHF);
        Money m14CHF = new Money(14, Currency.CHF);
        Money expected = new Money(26,
        Currency.CHF);
        Money observed = m12CHF.add(m14CHF);

    }
}
```

@Test annotation:
testSimpleAdd()
is the name of the test

This is still an incomplete unit test as we do not yet compare expected and observed state

The test calls the method add() from the system model

Annotations in JUnit 5

- **@Test public void exampleTest()**
  - Annotation @Test identifies that exampleTest() is a test method
- **@Timeout(1)**
  - The test fails if it takes longer than 1 second
- **@BeforeEach public void setUpTest()**
  - Perform setUpTest() before executing any test method
- **@AfterEach public void tearDownTest()**
  - Perform tearDownTest() after executing any test method
- **@BeforeAll public void beforeClassSetUp()**
  - Perform beforeClassSetUp() before the start of all tests: perform time intensive activities, e.g. to connect to a database
- **@AfterAll public void afterClassTearDown()**
  - Perform afterClassTearDown() after all tests have finished: perform clean-up activities, e.g. to disconnect from a database
- **@Disable**
  - Disable or ignore the test method: useful if the code has been changed but the test has not yet been adapted

Ensuring pre- and postconditions for **single** tests

- Any method can be annotated with **@BeforeEach** and **@AfterEach**

```
class CalculatorTest {

      @Test
      void addTest() { }

      @Test
      void subTest() { }

      @BeforeEach
      //executed before every test (i.e. 2x in this example)
      void setupTestData() { }

      @AfterEach
      //executed after every test (i.e. 2x in this example)
      void teardownTestData() { }
}
```

Ensuring pre- and postconditions for **multiple** tests

- A test class can have static methods annotated with **@BeforeAll** and with **@AfterAll**
- Useful for expensive setups that do not need to run for every test, e.g. setting up a database connection

```
@Test
void addTest() { }
@Test

void subTest() { }
@BeforeAll // executed once at instantiation of class (before all tests)
static void setupDatabaseConnection() { }

@AfterAll // executed once after removing instance of class (after all tests)
static void teardownDatabaseConnection() { }
```

Omitting tests

- There are situations where certain tests should not be executed
- Any test method can be omitted using **@Disable**
- **Example**
  - The current release of a third-party library used in the system has a bug in a routine
  - We cannot test it until the bug is fixed

```
class CalculatorTest {
      @Disable
      @Test
      public void test() {}
}
```

Make sure tests are short

- Unit tests should be short
- But some tests take their time, particularly if network connectivity is involved
- In these cases, it is recommended to set an upper bound for the test using a timeout

```
class CalculatorTest {

    @Timeout(value = 500, unit = TimeUnit.MILLISECONDS)
    void testNetworkOperation() {
    //...
    }
}
```

Assertions in JUnit 5

assertTrue(condition, [message]);
- Checks if condition evaluates to true; otherwise throws an Exception

assertFalse(condition, [message]);
- Checks if condition evaluates to false; otherwise throws an Exception

assertEquals(expected, actual, [message]);
- Checks if the values/objects expected and actual are equal; otherwise throws an Exception

assertEquals(expected, actual, delta, [message]);
- Used for float and double; delta specifies the number of decimals which must be the same

fail(message);
- Let the method fail, useful to check that a certain part of the code is not reached

Note: [message] is an optional parameter

assertNull(object, [message]);
- Checks if the object is null; otherwise throws an Exception

assertNotNull(object, [message]);
- Check if the object is not null; otherwise throws an Exception

assertSame(expected, actual, [message]);
- Check if both variables expected and actual refer to the same object; otherwise throws an Exception

assertNotSame(expected, actual, [message]);
- Check that both variables expected and actual do not refer to the same object; otherwise throws an Exception

assertThrows(expectedType, executable, [message]);
- Check that execution of the supplied executable (e.g. lambda expression) throws an exception of the expectedType and returns the exception

assertEquals

- The unit test for the class **Money** should test all public and protected methods in the class, except getters and setters
- **Example** test for the **add()** method of Money

```
class MoneyTest {

    @Test
    void testSimpleAdd() {
        Money m12CHF = new Money(12, Currency.CHF);
        Money m14CHF = new Money(14, Currency.CHF);
        Money expected = new Money(26, Currency.CHF);
        Money observed = m12CHF.add(m14CHF);
        assertEquals(expected, observed);
    }
}
```

The test passes, if both parameters are equal, otherwise the test throws an exception of type **AssertionError**

assertThrows

- **Example**

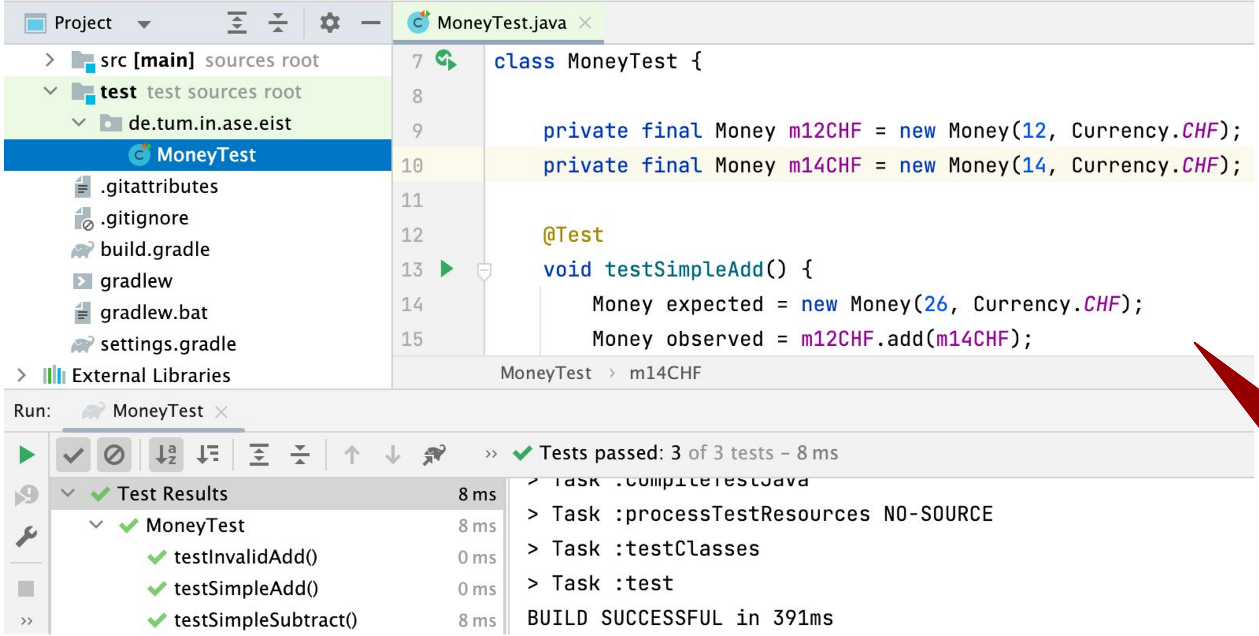
```java
class ExceptionTest {

    @Test
    void testExpectedException() {
        // 1st argument specifies the expected exception
        // It expects that code block will throw NumberFormatException
        // 2nd argument passes an executable code block or lambda
        expression
        assertThrows(NumberFormatException.class, () -> {
            Integer.parseInt("One");
        });
    }
}
```

The test passes, if a NumberFormatException is thrown within the lambda expression

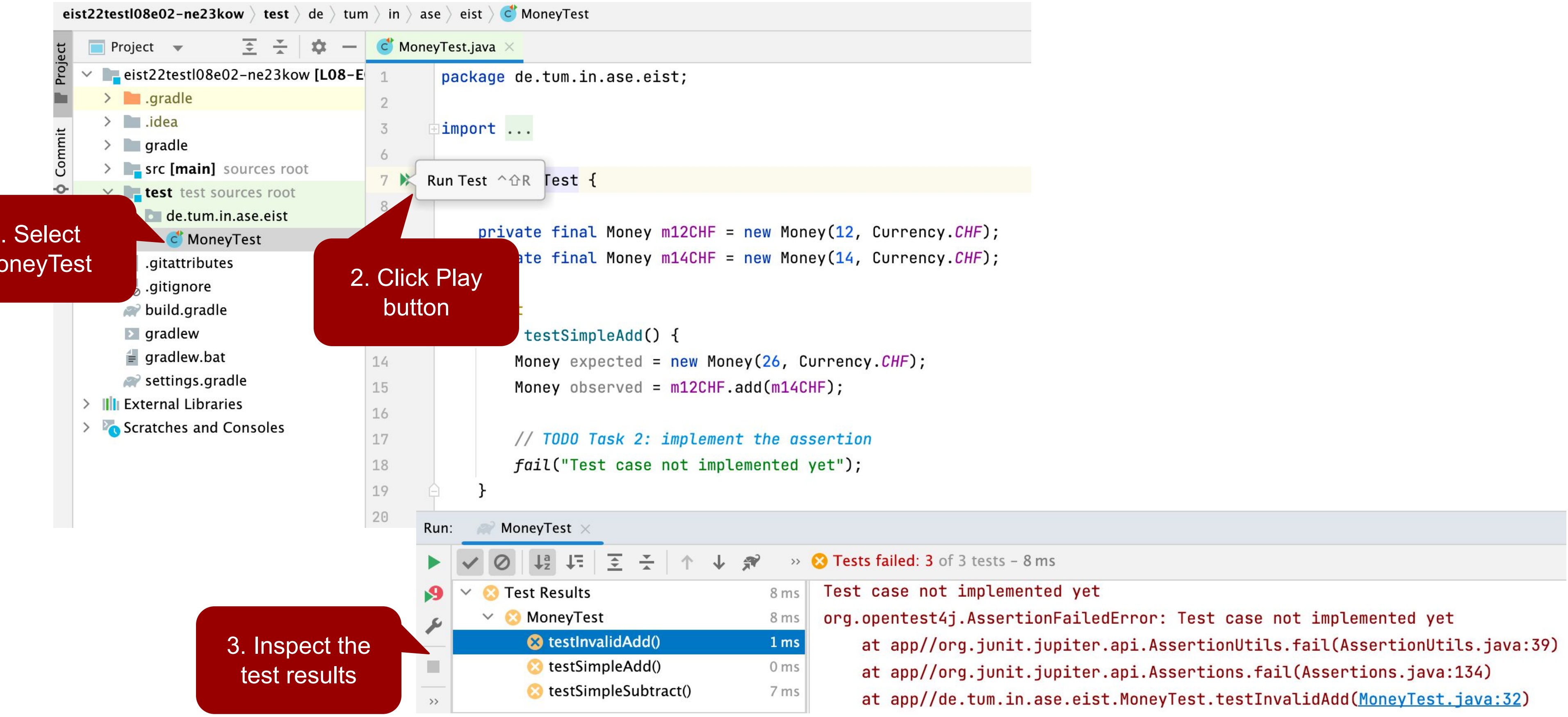
L07P01: Unit testing

- **Problem statement**
  - Execute the test case **MoneyTest**
  - Complete the implementation of **Money** and **MoneyTest**

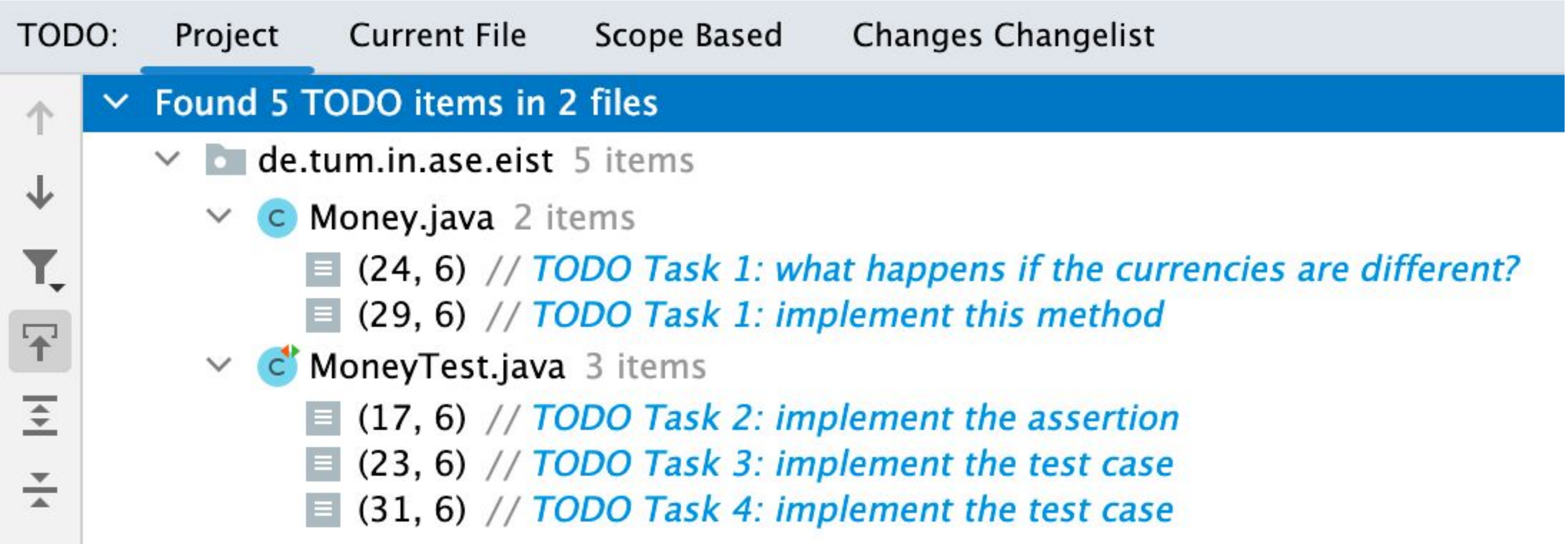


Edsger Dijkstra: Testing shows the presence, not the absence of faults ("bugs")

L07P01: Unit tests - Run the test



L07P01: Unit tests - Follow the TODOs



Example solution: Money

```
public class Money {

    //...
    public Money add(Money money) {
        if(currency != money.currency()) {
        throw new IllegalArgumentException("Different currencies not
        supported!");
        }
        return new Money(amount() + money.amount(), currency());
    }

    public Money subtract(Money money) {
        if(currency != money.currency()) {
        throw new IllegalArgumentException("Different currencies not
        supported!");
        }
        return new Money(amount() - money.amount(), currency());
    }

}
```

If the currencies are not the same, throw an IllegalArgumentException

If the currencies are not the same, throw an IllegalArgumentException

Example solution: MoneyTest

```
class MoneyTest {
      private Money m12CHF = new Money(12, Currency.CHF);
      private Money m14CHF = new Money(14, Currency.CHF);

      @Test
      void testSimpleAdd() {
            Money expected = new Money(26, Currency.CHF);
            Money observed = m14CHF.add(m12CHF);
            assertEquals(expected, observed);
      }

      @Test
      void testSimpleSubtract() {
            Money expected = new Money(2, Currency.CHF);
            Money observed = m14CHF.subtract(m12CHF);
            assertEquals(expected, observed);
      }

      @Test
      void testInvalidAdd() {
            Money m14USD = new Money(14, Currency.USD);
            assertThrows(IllegalArgumentException.class, () -> {
                  m12CHF.add(m14USD);
            });
      }
}
```

Check if the expected amount is the same as the observed amount

Check if the expected amount is the same as the observed amount

If the currencies are not the same, expect an IllegalArgumentException

Outline

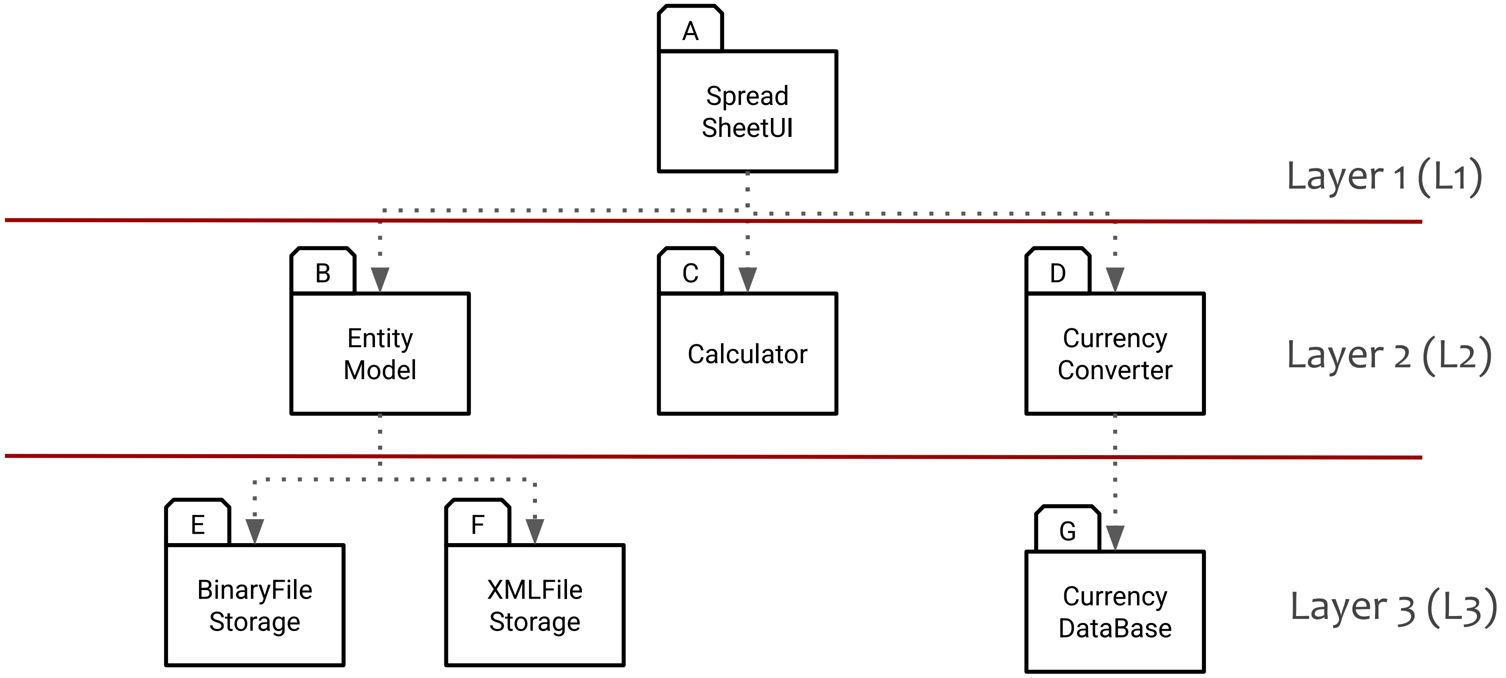
- **Part I: Testing activities**
  - ~~Terminology~~
  - ~~Unit testing~~
  - **Integration testing**
- **Part II:** Automated system testing
- **Part III:** Model-based testing
- **Part IV:** Object-oriented testing

Integration testing

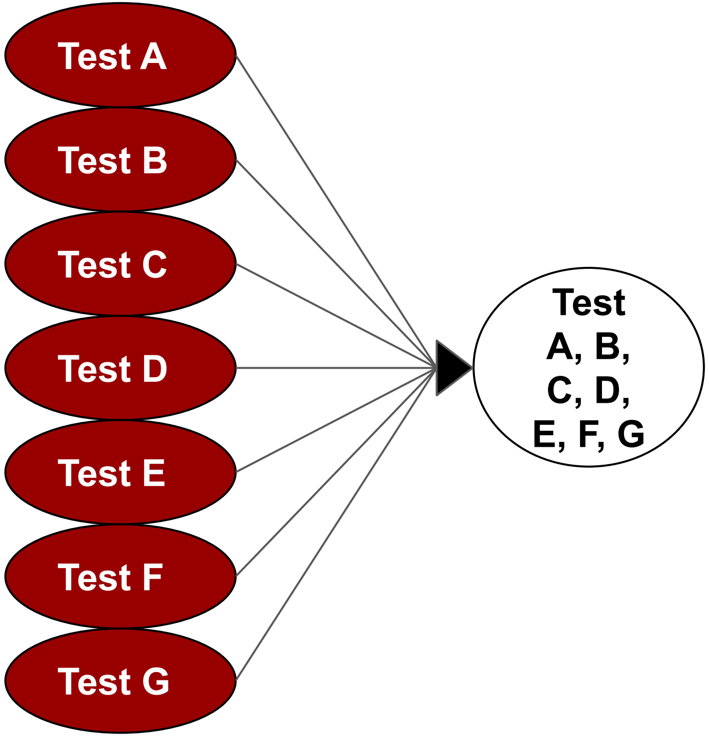
- The entire system is viewed as a collection of subsystems (set of classes) determined during the system and object design
- **Goal:** test all interfaces between subsystems and the interaction of subsystems
- The **integration testing strategy** determines the order in which the subsystems are selected for testing and integration
  - Big bang integration
  - Bottom up testing
  - Top down testing
  - Vertical integration

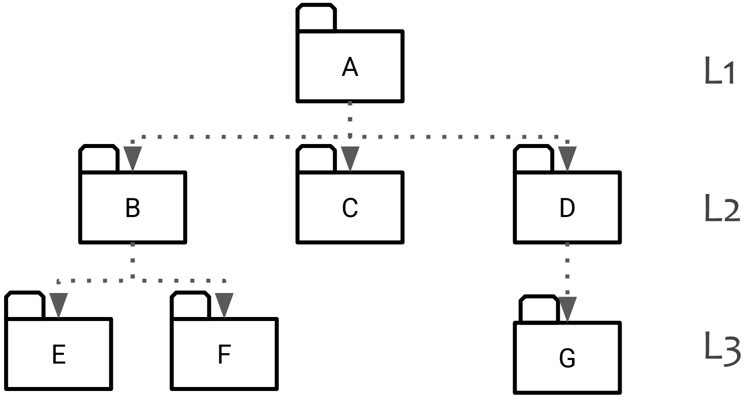
Horizontal integration

Example: integration testing for a 3 layer design



Big bang approach





Pitfall of the “big bang” approach

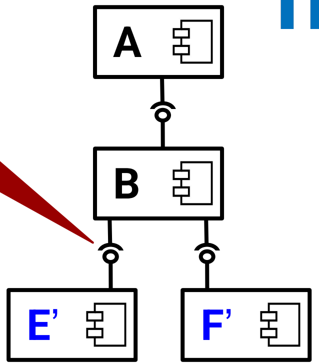
- The system integration will be **late** in the software development cycle
- **Difficult to identify problems** instead of “piece-wise” identification of problems → overwhelming for the developers
- It is **not possible** to test the system "early on” since it requires the implementation of all components

- **Alternative strategies**
  - Bottom-up integration
  - Top-down integration

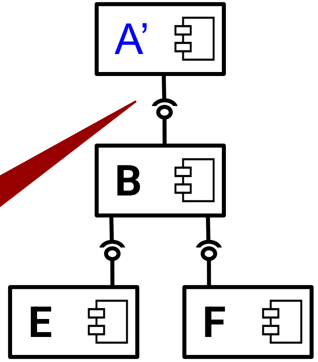
Drivers and stubs

- Both are doubles that replace the actual component (subsystem or class) during testing
- **Stub**
  - **Provides** the same interface as the actual (replaced) component
  - Each operation is implemented very simply (e.g. always returns the same value)
  - Allows to test other components (which **require** the interface and **invoke** it)
  - Used in top-down integration
  - **Example: E’ and F’**
- **Driver**
  - **Invokes** and **requires** the same interface as the actual (replaced) component
  - Each operation of the interface is invoked for testing purposes
  - Allows to test other components (which **provide** the interface)
  - Used in bottom-up integration
  - **Example: A’**

E’ provides the interface —> B can be tested which invokes the interface



Use of stubs (E’ and F’) when top-down testing B



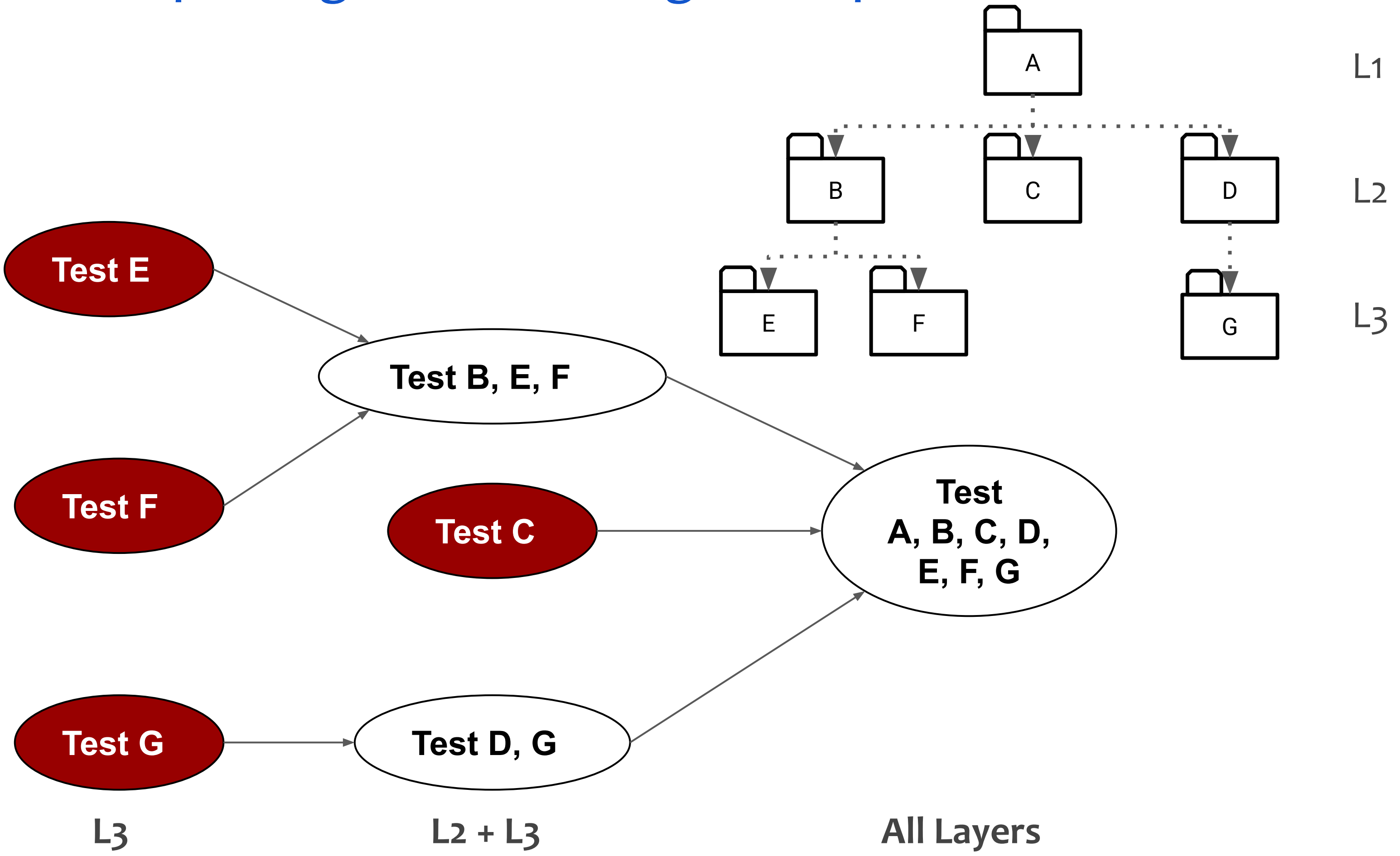
A’ invokes the interface —> B can be tested which provides the interface

Use of a driver A’ when bottom-up testing B

Bottom up testing strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then, the subsystems above this layer are tested which call the previously tested subsystems
- This is repeated until all subsystems are included

Bottom up integration testing example



Pros and cons: bottom up integration testing

- **Pros**

  + No stubs needed

  + Useful for integration testing of

    - Object-oriented systems
    - Systems with strict performance requirements, e.g. real-time systems
- **Cons**

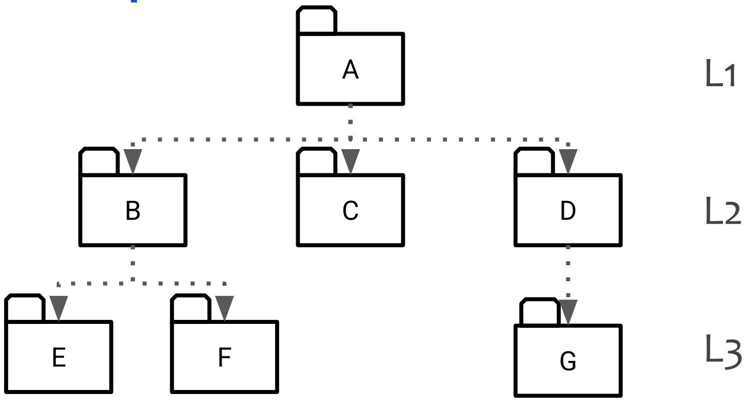
  – Tests an important subsystem (the user interface) last

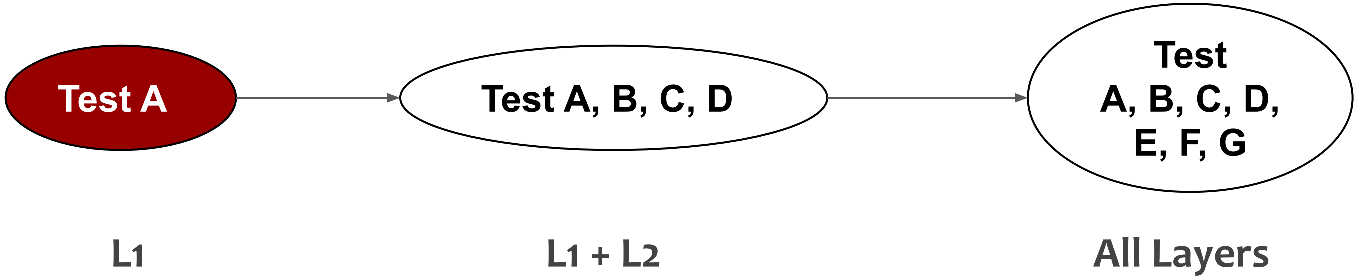
  – Drivers are needed

Top down testing strategy

- Test the subsystems in the top layer first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the tests

Top down integration testing example





Pros and cons: top down integration testing

- **Pros**

  \+ Test cases can be defined in terms of the functional requirements of the system

  \+ No drivers needed

- **Cons**

  – Stubs are needed

  – Writing stubs is difficult: they must allow all possible conditions to be tested

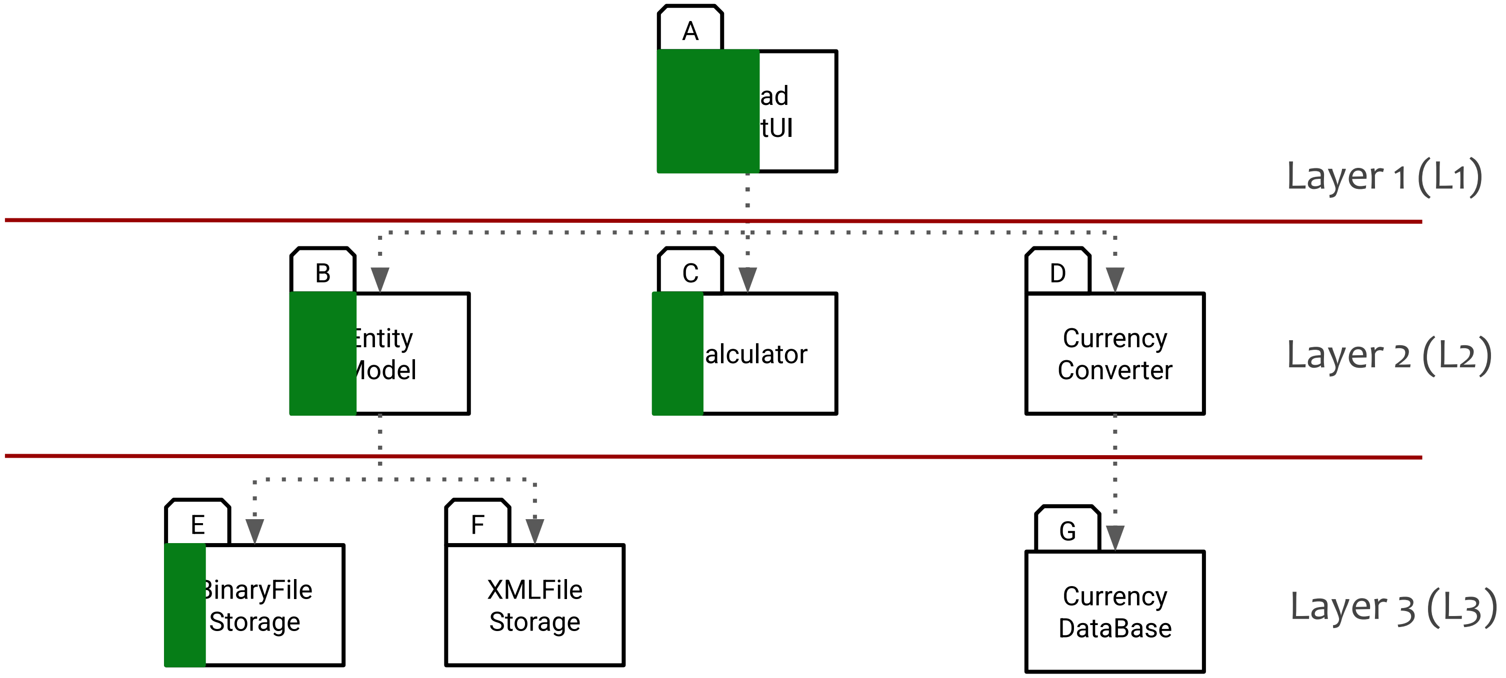
  – Large number of stubs may be required, especially if the lowest level of the system contains many methods

- Solution to avoid too many stubs: **sandwich testing strategy**
  - Test each layer of the system decomposition individually before merging the layers
  - **Disadvantage**: both stubs and drivers are needed

Horizontal integration testing risks

- **Risk #1:** The higher the complexity of the software system, the more difficult is the integration of its components
- **Risk #2:** The later integration occurs in a project, the bigger the risk that unexpected failures occur
- Horizontal integration strategies (bottom up, top down) don't do well with **risk #2**
- **Vertical integration** addresses these risks by building as early and frequently as possible
  - Used in scenario-driven design: scenarios are used to drive the integration
  - Used in Scrum: user stories are used to drive the integration → **potentially shippable product increment**
- **Advantages of vertical integration**
  - There is **always** an executable version of the system (→ **continuous integration**)
  - All the team members have a **good overview** of the project status

Vertical integration testing



Outline

- ~~Part I: Testing activities~~
  - ~~Terminology~~
  - ~~Unit testing~~
  - ~~Integration testing~~
- **Part II: Automated system testing**
  - Fuzzing
  - Symbolic execution
  - Crashing: Chaos Monkey
- **Part III:** Model-based testing
- **Part IV:** Object-oriented testing

I: Automated test case generation w/ Fuzzing

```
   Write manual tests

myAPI(test-case-1);
myAPI(test-case-2);
myAPI(test-case-3);
...
```

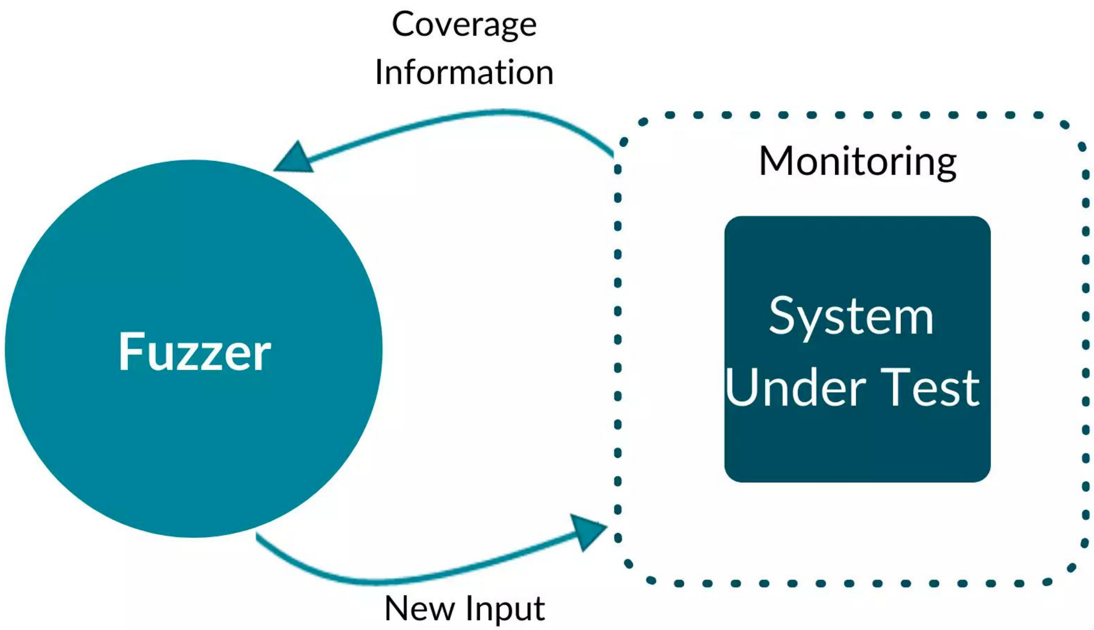
```
   Generate automated tests

while(terminating_condition) {
     myAPI(generate_input());
}
```

Fuzz testing

- **(or simply) fuzzing**
  - Run program on many **random, abnormal** inputs and look for **bad behavior** in the responses
  - Bad behaviors such as crashes or hangs
  - Extensively used to find reliability and security issues
- What are the benefits compared to manual testing?
  - (Semi-)automated way of generating a large set of test cases
  - Program is tested using (ab-)normal inputs

Fuzzing


Coverage
Information
Monitoring
Fuzzer
System
Under Test
New Input

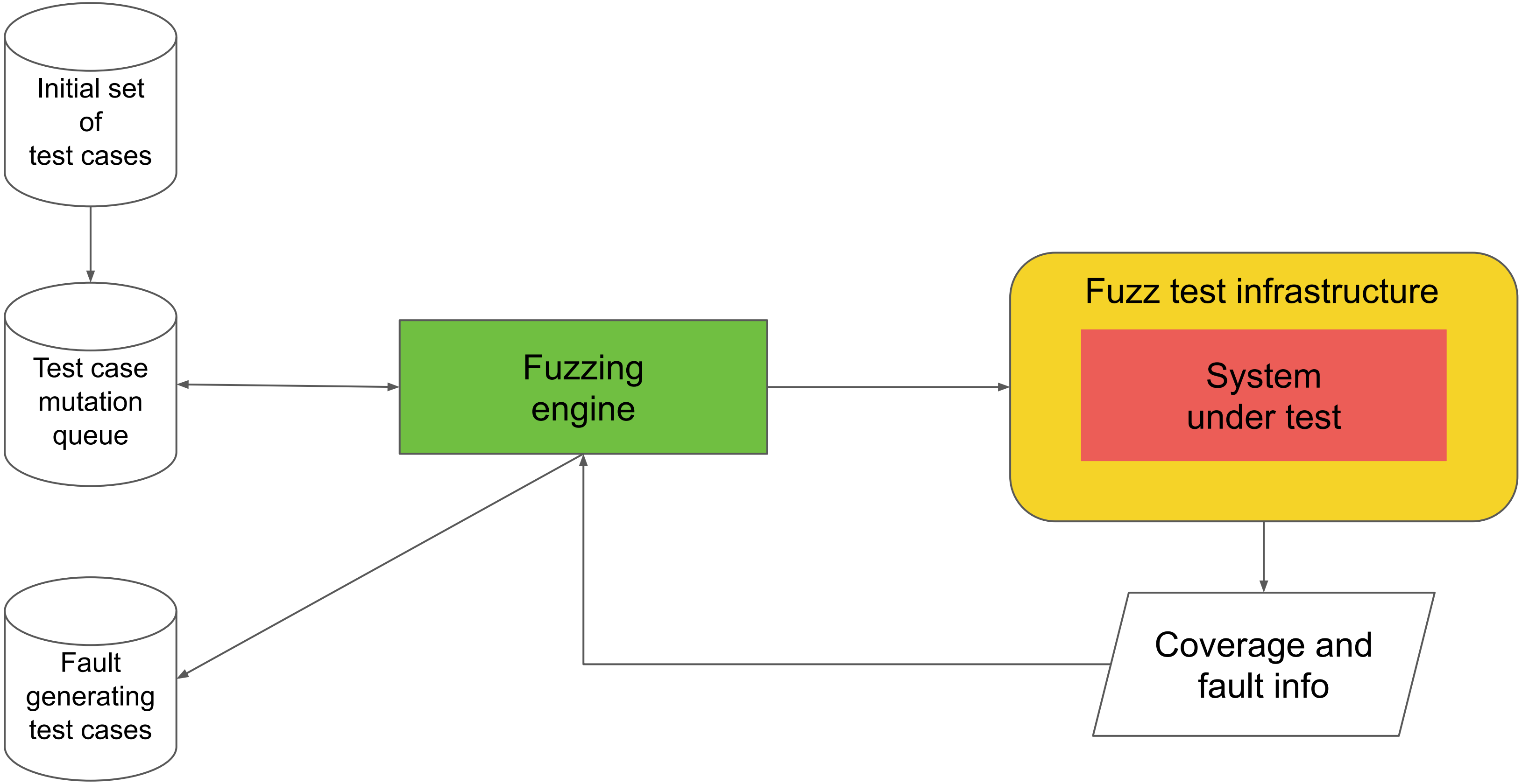
Types of fuzzing

- **Black-box fuzzing:** generates input without any knowledge of the program
  - **Mutation-based:** starts with one or more seed inputs → these seeds are modified to generate new inputs: random mutations are applied to the input
  - **Generation-based:** inputs are generated from scratch → structural specification of the input is provided, new inputs are generated to meet the grammar
  - **Example:** Peach
- **Grey-box fuzzing:** involves program instrumentation to get feedback and steer the fuzzer
  - Program is instrumented at the compile time and an initial input seed corpus is provided
  - Seed input is mutated to generate new inputs
  - Generated inputs that cover new control locations (increasing coverage) are added to the seed input
  - **Examples:** AFL, Libfuzz
- **White-box fuzzing:** based on "symbolic execution" that involves program analysis to systematically exercise all paths in the program
  - **Examples:** KLEE, SAGE, Angr, S2E

Trade-offs

	Black-box	Grey-box	White-box
Pros	+ Easy to setup and automate + Little or no knowledge of the system	+ Feedback-driven fuzzing with coverage	+ Systematic exploration of (all/interesting) paths
Cons	- Limited by initial corpus - Fails to capture the complexities of modern systems	- Still relies on random mutation of the input	- Difficult to scale (path explosion problem)

Grey-box fuzzing architecture



Google AFL: American Fuzzy Lop

- A state-of-the-art grey-box fuzzer extensively used in production
    - Load user-supplied input test cases into the queue
    - Takes next input test case from the queue
    - Repeatedly mutates the test cases based on fuzzing strategies
    - If generated mutation exercises a new path, add it to the queue for further exploration
    - Repeat!

More info: https://github.com/google/AFL

II: Symbolic execution

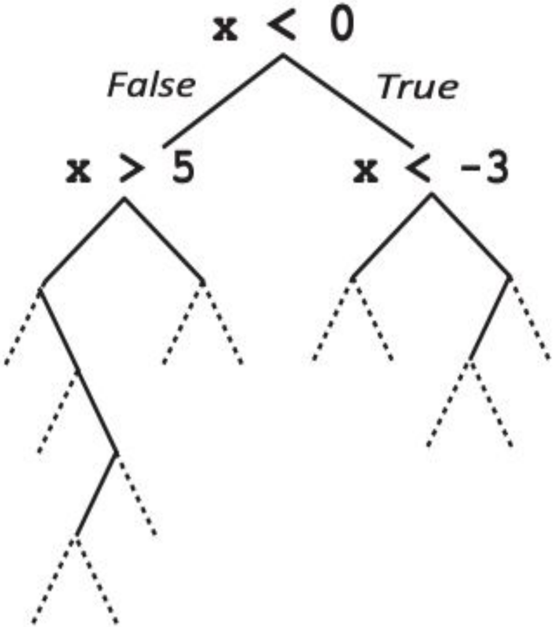
- Testing works*
  - But, each test only explores one possible execution
    - assert(f(3) == 5)
  - We hope test cases generalize, but no guarantees
- Symbolic execution generalizes testing
  - Allows unknown symbolic variables in evaluation

Symbolic execution

- A **symbolic execution engine** executes a program with “symbolic” inputs instead of running the program with regular inputs
  - For e.g., an integer input x is given as value a symbol α that can take on any integer value
- When the program encounters a branch that depends on x, the program state is forked to produce two parallel executions (if and else path)
  - For e.g., make the branch condition evaluate to true (e.g., $\alpha<0$), respectively false (e.g., $\alpha\geq 0$)

Symbolic execution example

```
void read( int x ){
  if (x < 0) {
    if (x < -3)
      foo(x);
    else {
      ...
    }
  } else {
    if (x > 5)
      bar(x);
    else {
      ...
  ...
```



How symbolic execution finds a bug?

- When an execution encounters a testing goal (e.g., a bug), the constraints collected from the root to the goal leaf can be solved to produce concrete program inputs that exercise the path to the bug
  - Path condition is a logical formula, i.e., a set of constraints collected for an execution
  - SMT solvers solve these path conditions to produce a test case that led to the failure

Symbolic execution engines

- **KLEE Symbolic execution engine**
  - http://klee.github.io/
  - KLEE is a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure
  - A source-level symbolic execution engine
- **S2E: A Platform for In-Vivo Analysis of Software Systems**
  - https://s2e.systems/
  - S^2E runs unmodified x86, x86-64, or ARM software stacks, including programs, libraries, the kernel, and drivers
- **angr: an open-source binary analysis platform for Python**
  - https://angr.io/
  - A binary level symbolic execution engine, constraint-solving, and instrumentation

III: Choas Monkey

- Chaos Monkey randomly terminates instances in production to ensure that engineers implement their services to be resilient to instance failures
    - A widely used tool, developed at Netflix, to test the resilience of microservices
- How does it work?
    - Set up a cron job that calls Chaos Monkey periodically to create a schedule of terminations
- Tool:
    - https://github.com/Netflix/chaosmonkey

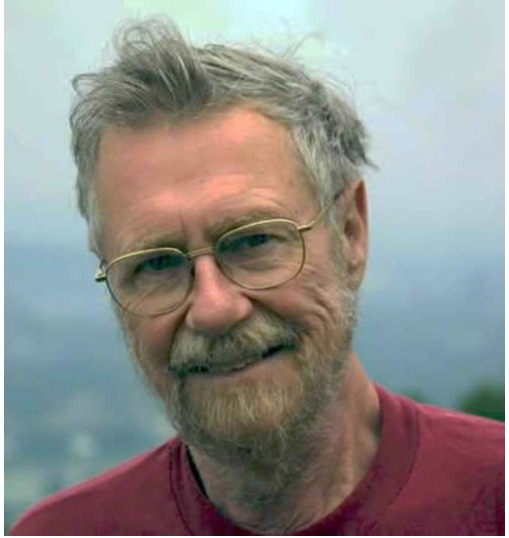
Outline

- ~~Part I: Testing activities~~
  - ~~Terminology~~
  - ~~Unit testing~~
  - ~~Integration testing~~
- ~~Part II: Automated system testing~~
- **Part III: Model-based testing**
- **Part IV:** Object-oriented testing

Observations

- It is **impossible** to completely test any nontrivial module or system
  - Practical limitations: complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g. halting problem
- "Testing can only show the presence of bugs, not their absence" (Dijkstra)
- **Testing is not free**

➡ Define your goals and priorities



Testing takes creativity

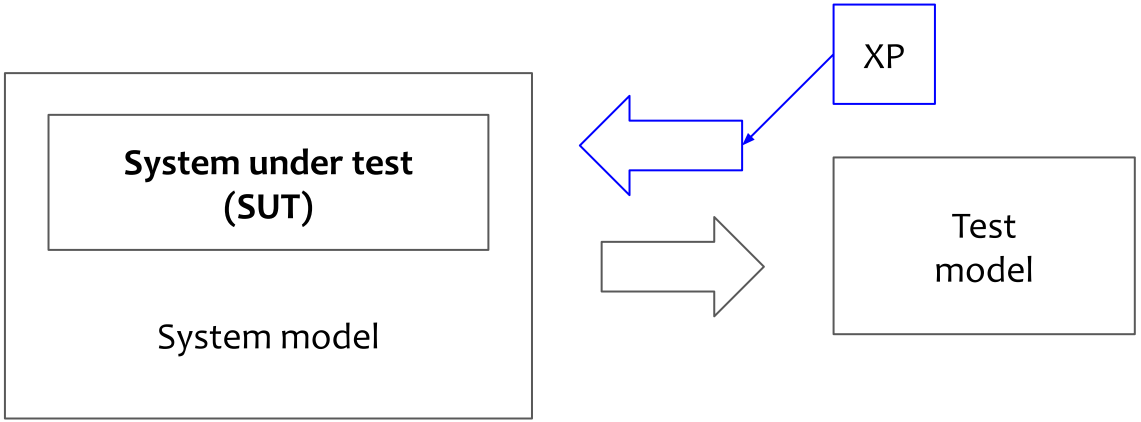
- To write an effective test case, the following is necessary
  - Detailed understanding of the system
  - Application and solution domain knowledge
  - Knowledge of the testing techniques
  - Skill to apply these techniques
- Testing is done best by independent people
  - Developers often have a certain mental attitude that the program should behave in a certain way when in fact it does not
  - Developers often stick to the data set that makes the program work
  - A program often does not work when tried by somebody else

Test model

- Consolidates all test related decisions and components into one package (sometimes also test package or test requirements)
- Contains
  - **Test cases:** functions usually derived from use cases (specification of behavior realizing one or more test objectives)
  - **Input data:** data needed for the execution of the test cases
  - **Oracle:** predicts the expected output data
  - **Test system:** a framework or software component (e.g. JUnit) that executes the tests under varying conditions and monitors the behavior and output

Model based testing

- The **system model** is used for the generation of the **test model**
- Extreme programming **(XP)** variant
  - The **test model** is used for the generation of the **system model**
  - Test-driven development: test → code → refactor
- System under test **(SUT):** part of the system model which is being tested

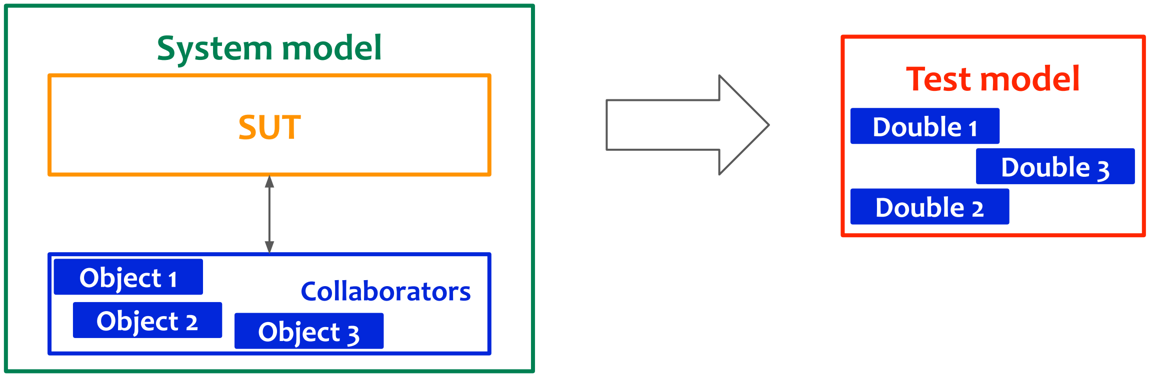


Outline

- ~~Part I: Testing activities~~
  - ~~Terminology~~
  - ~~Unit testing~~
  - ~~Integration testing~~
- ~~Part II: Automated system testing~~
- ~~Part III: Model-based testing~~
- **Part IV: Object-oriented testing**

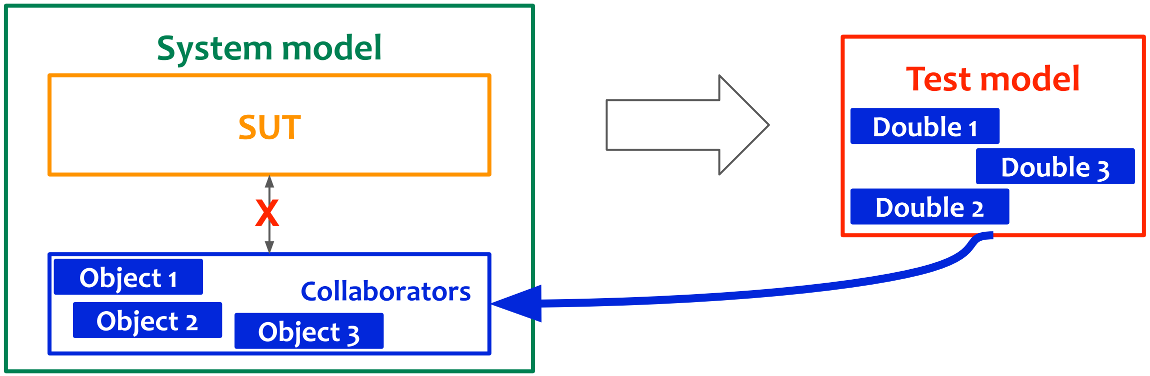
Object oriented test modeling

- Start with the **system model**
- The system contains the **SUT** (system under test)
- The **SUT** does not exist in isolation, it interacts with other participating objects in the system model that are not yet implemented: **collaborators**
- The **test model** is derived from the **SUT**
- To be able to interact with **collaborators**, we add objects to the **test model**
- These are called **test doubles**



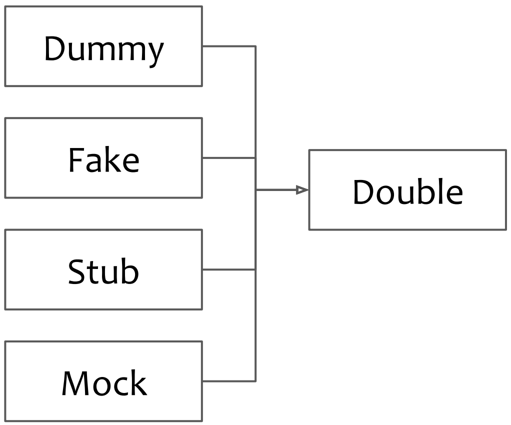
Object oriented test modeling

- Start with the **system model**
- The system contains the **SUT** (system under test)
- The **SUT** does not exist in isolation, it interacts with other participating objects in the system model that are not yet implemented: **collaborators**
- The **test model** is derived from the **SUT**
- To be able to interact with **collaborators**, we add objects to the **test model**
- These are called **test doubles** (substitutes for the **collaborators** during testing)



Taxonomy of test doubles

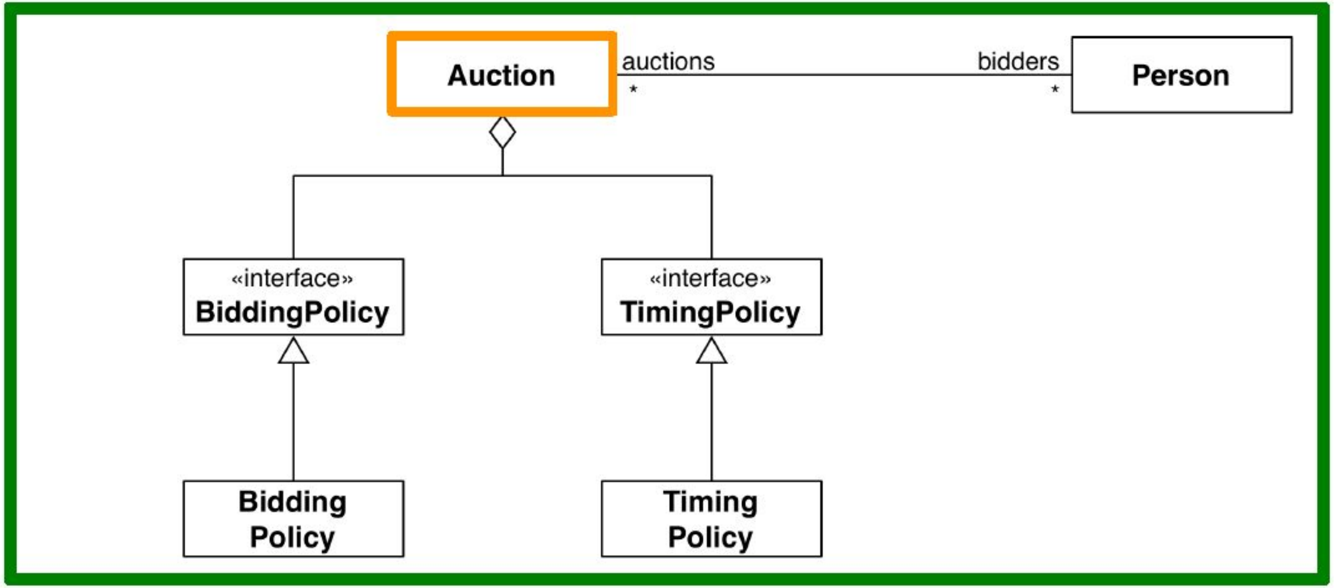
- **Dummy:** often used to fill parameter lists, passed around but never actually used
- **Fake:** a working implementation that contains a “shortcut” which makes it not suitable for production code
  - **Example:** a database stored in memory instead of on a disk
- **Stub:** provides canned answers (e.g. always the same) to calls made during the test
  - **Example:** random number generator that always return 3.14
- **Mock:** mimic the behavior of the real object and know how to deal with a specific sequence of calls they are expected to receive



Good design is crucial when using mock objects: the real object (subsystem) must be specified with an interface (façade) and a class for the implementation

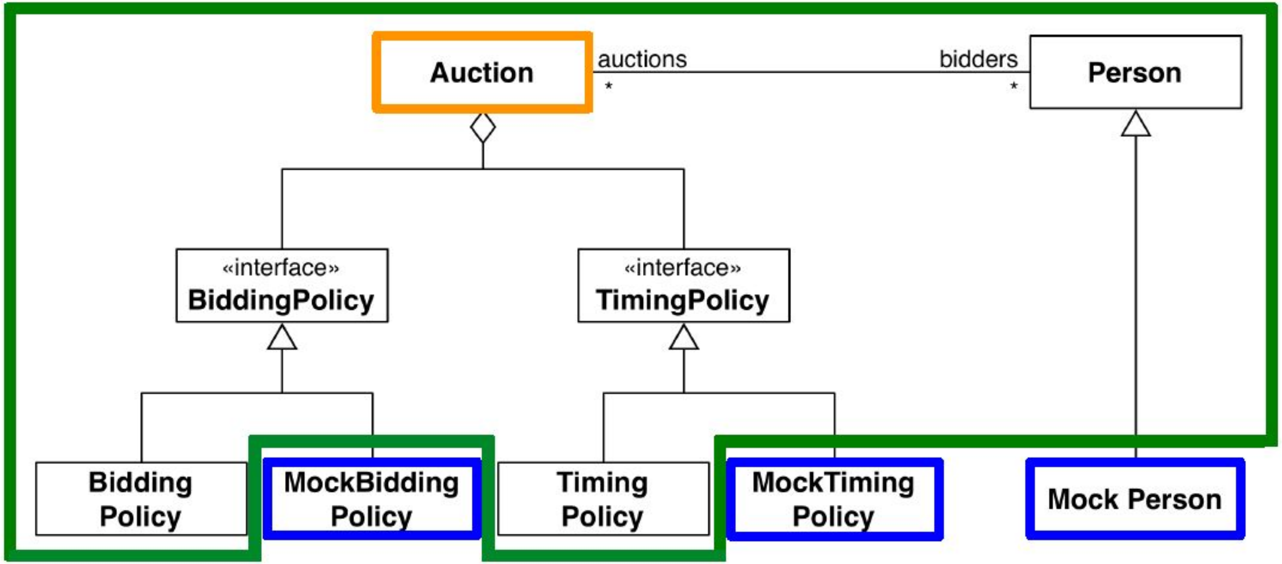
Motivation for mock objects

- There is a **system model** for an auction system with 2 types of policies
- We want to unit test Auction, which is the **SUT**

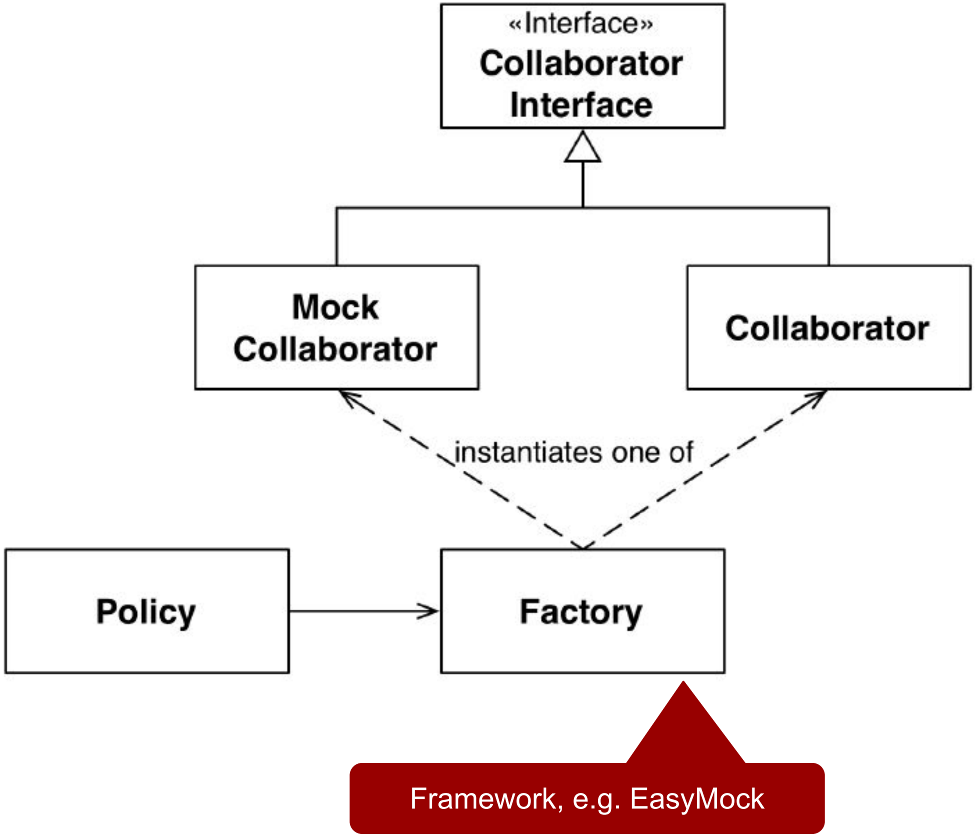


Motivation for mock objects

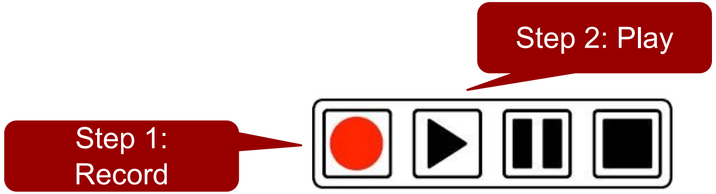
- There is a **system model** for an auction system with 2 types of policies
- We want to unit test **Auction**, which is the **SUT**
- The mock object test pattern is based on the idea to replace the interaction with the collaborators in the system model, that is **Person, BiddingPolicy and TimingPolicy**, by **mock objects**
- These mock objects are created at startup time



Mock object pattern

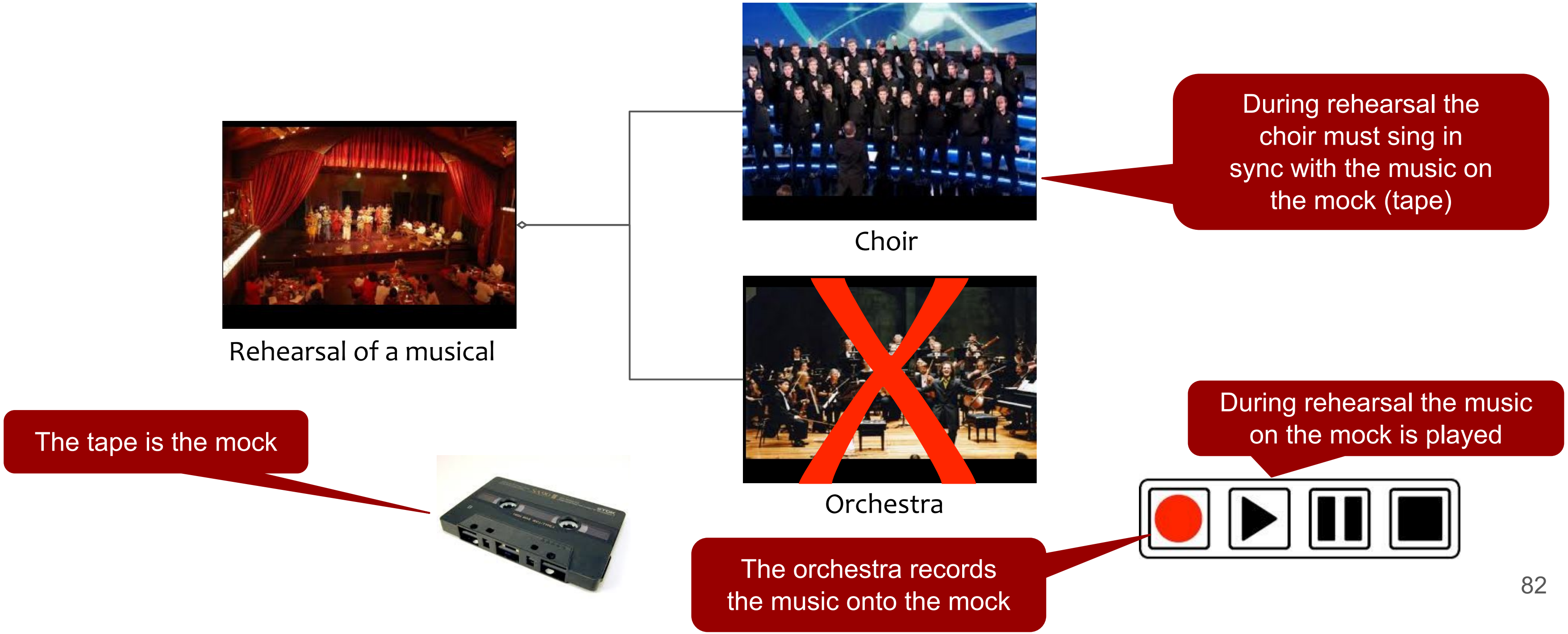


- A **mock object** replaces the behavior of a real object called the collaborator and returns hard-coded values
- A mock object can be created at startup time with the **factory pattern**
- Mock objects can be used for testing the state of individual objects and the interaction between objects
- The use of mock objects is based on the **record play metaphor**



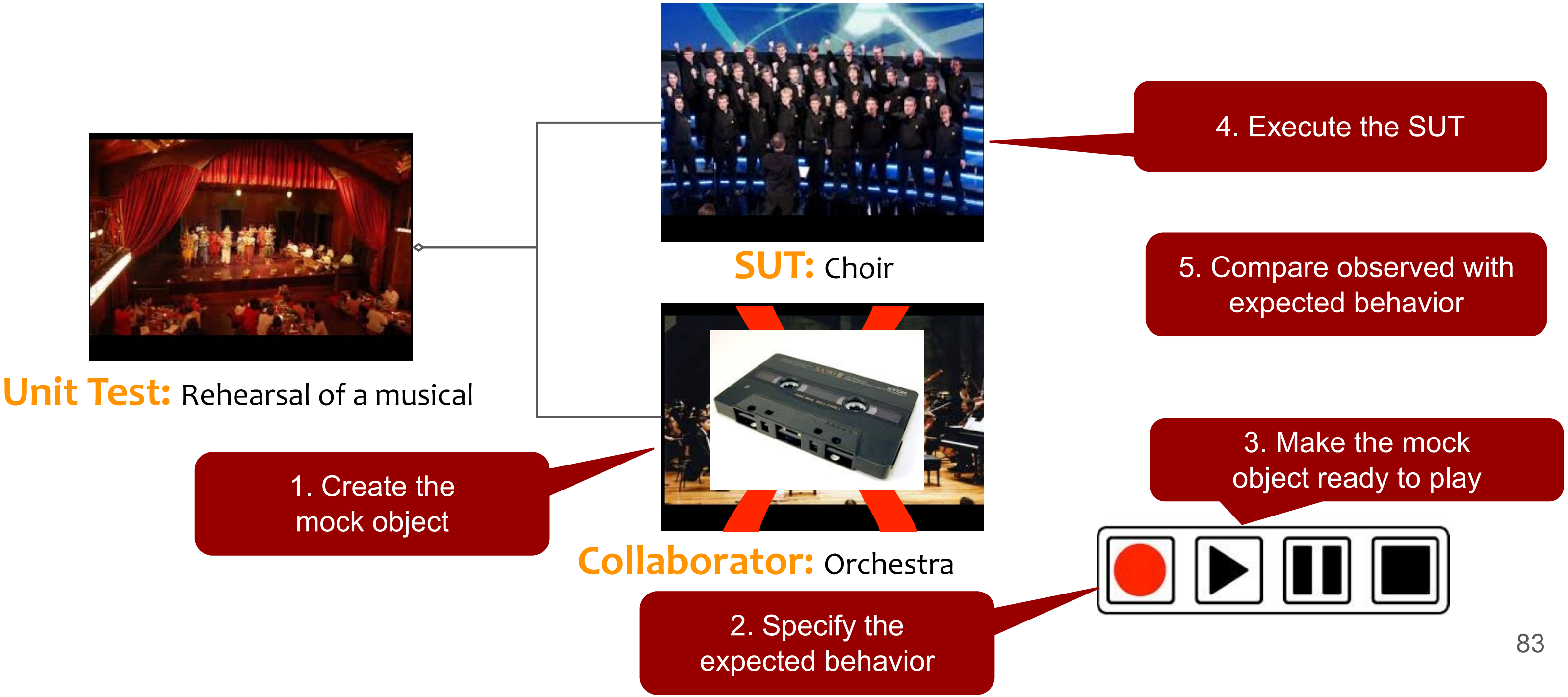
Record play metaphor

Assume you want to perform a musical, which requires an orchestra and a choir. Most of the time the orchestra will not be available (too expensive), when the choir practices. But the choir needs to be accompanied by the music played by the orchestra when rehearsing the musical:



Record play metaphor for mock objects

Mock objects are proxy collaborators in tests where the real collaborators are not available



EasyMock

- Open source testing framework for Java
- Uses annotations for test subjects (=SUT) and mocks

```
@TestSubject
private ClassUnderTest classUnderTest = new ClassUnderTest();
@Mock
private Collaborator mock;
```

- Specification of the behavior

```
expect(mock.invoke(parameter)).andReturn(42);
```

- Make the mock ready to play

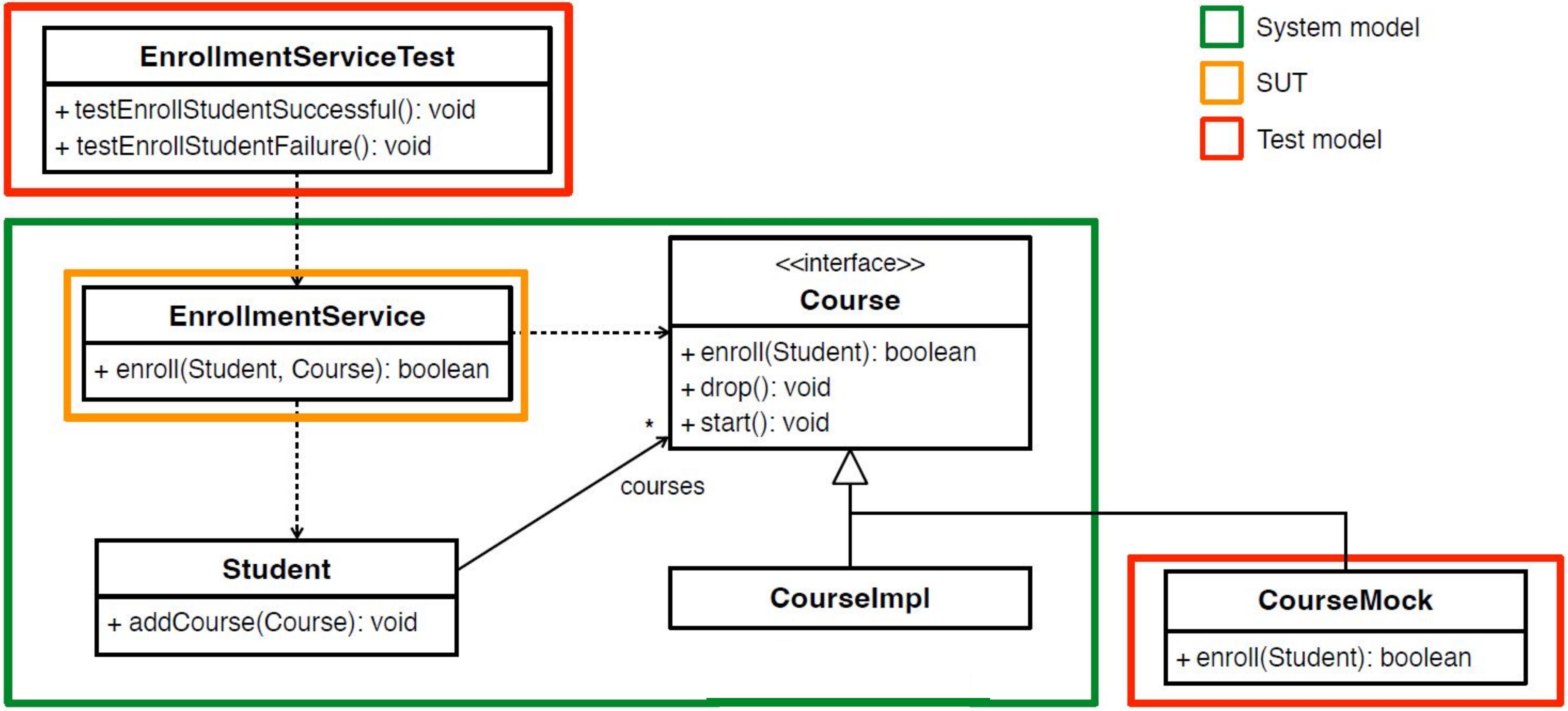
```
replay(mock);
```

- Make sure the mock has actually been called in the test (additional assertion)

```
verify(mock);
```

- Documentation: http://easymock.org/user-guide.html

Example: university app with a mock object



L07P02: Mock Object Pattern

- **Problem statement**
  - Apply the mock object pattern using EasyMock
  - Implement **testEnrollStudentSuccessful()**
  - Optional challenge: implement **testEnrollStudentFailure()**

Example solution: unit test for enrolling students with EasyMock

```
@ExtendWith(EasyMockExtension. class)
class EnrollmentServiceTest {
    @TestSubject
    private EnrollmentService enrollmentService = new
    EnrollmentService();

    @Mock
    private Course courseMock;

    @Test
    void testEnrollStudentSuccessful () {
        Student student = new Student();
        int expectedSize = student.getCourses().size() + 1;
        expect(courseMock.enroll(student)).andReturn(true);
        replay(courseMock);
        enrollmentService.enroll(student, courseMock);
        assertEquals(expectedSize, student.getCourses().size());
        verify(courseMock);
    }
}
```

1. Instantiate the SUT

2. Create the mock object

3. Specify the expected behavior

4. Make the mock object ready to play

5. Execute the SUT

6. Validate observed against expected behavior

7. Verify that enroll() was invoked on courseMock once

References

- Kent Beck, Erich Gamma, Junit Cookbook
  http://junit.sourceforge.net/doc/cookbook/cookbook.htm
- JUnit 5: https://junit.org/junit5/
- Martin Fowler, Mocks are not Stubs: http://martinfowler.com/articles/mocksArentStubs.html
- Brown & Tapolcsanyi: Mock Object Patterns. In Proceedings of the 10th Conference on Pattern Languages of Programs, 2003. http://hillside.net/plop/plop2003/papers.html
- Herman Bruyninckx, Embedded Control Systems Design, WikiBook, Learning from Failure:
  http://en.wikibooks.org/wiki/Embedded_Control_Systems_Design/Learning_from_failure
- Joanne Lim, An Engineering Disaster: Therac-25
- http://www.bowdoin.edu/~allen/courses/cs260/readings/therac.pdf
- Peter G. Neumann, Computer-Related Risks, Addison-Wesley, ACM Press, 384 pages, 1995
- Philipp Hauer: Modern Best Practices for Testing in Java,
  https://phauer.com/2019/modern-bestpractices-testing-java
- EasyMock: http://easymock.org/user-guide.html

Summary

- Testing is difficult, but many rules and heuristics are available
- Unit testing with JUnit
  - Assertions
  - Annotations
- Integration testing
  - Horizontal vs. vertical testing
- System testing
  - Fuzzing, symbolic execution, and crashing
- Object-oriented testing
  - Mock object pattern
  - EasyMock