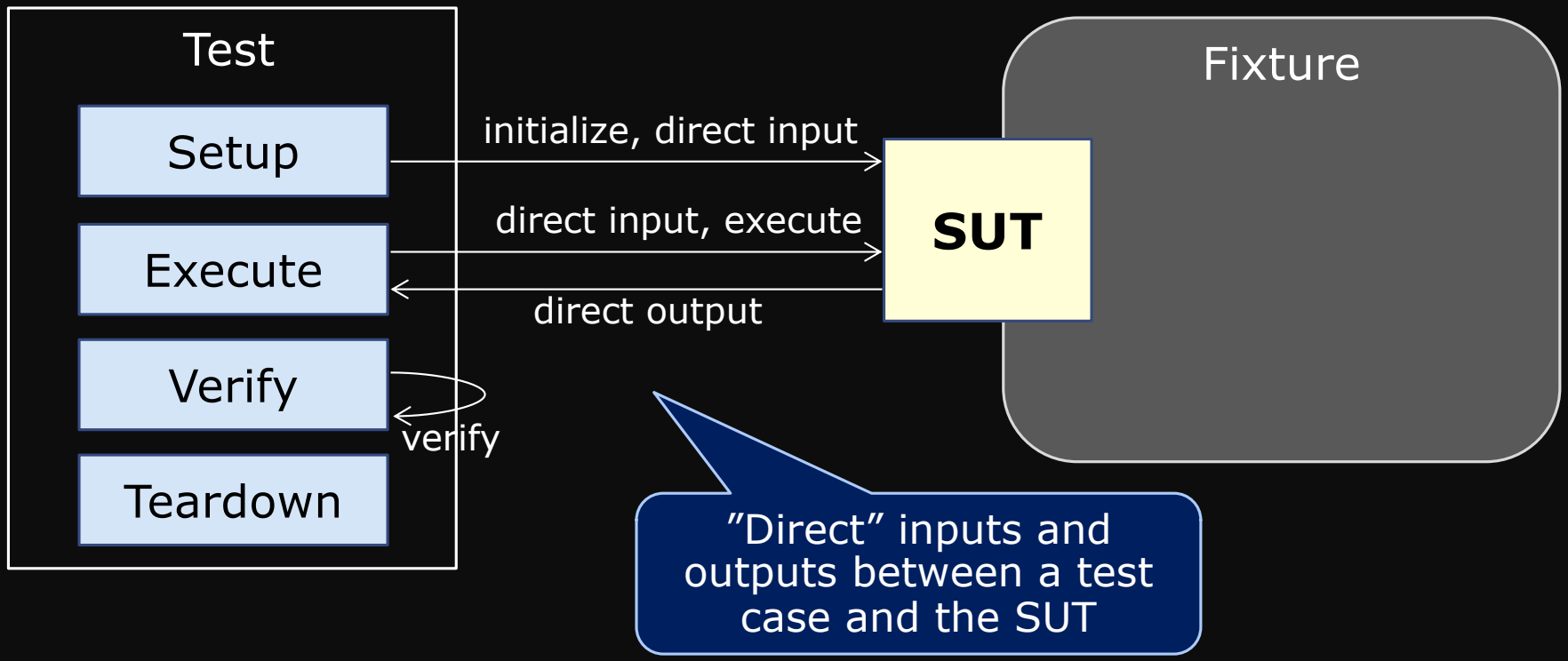
TDD: Test Doubles

CS 3250
Software Testing

[Lasse Koskela, "Test Driven," Chapter 4]
[Tilo Linz, "Testing in Scrum," Chapter 4]
[Frank Appel, "Testing with JUnit," Chapter 3]

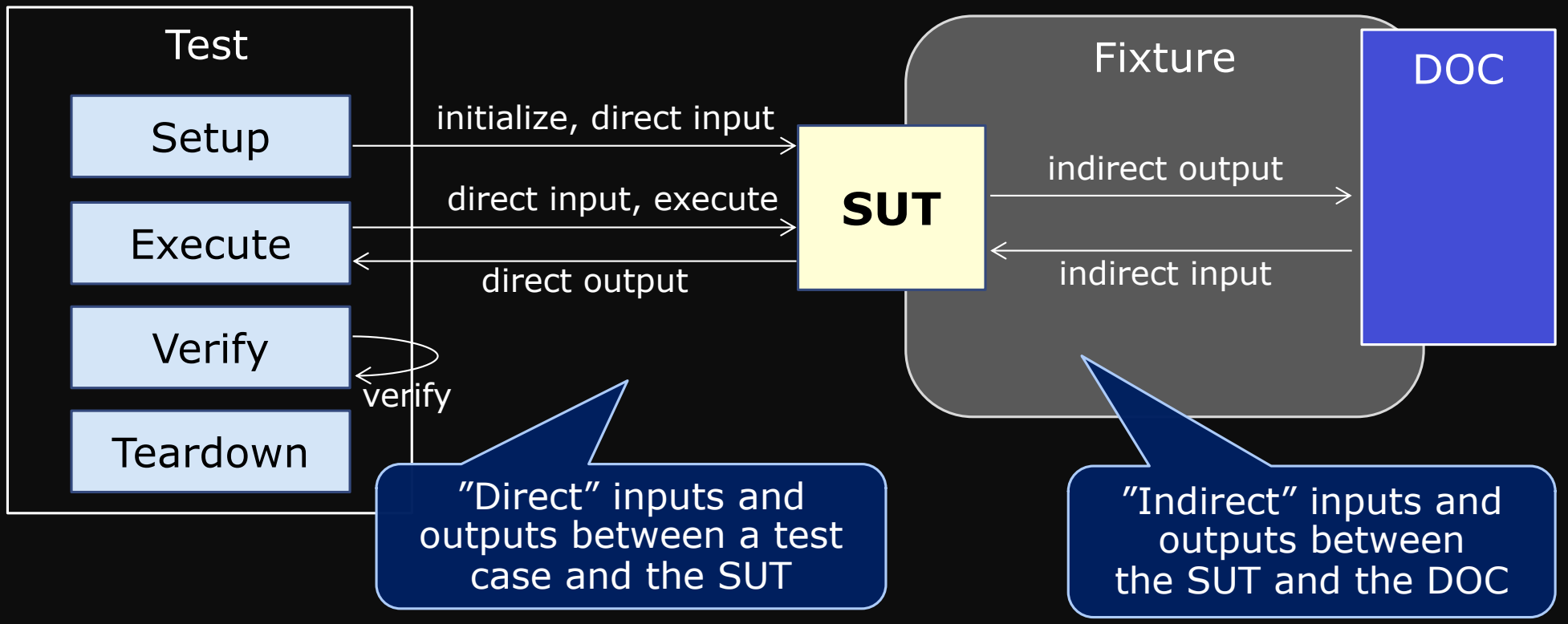
Simple Scenario: Test and SUT



Assert the “direct” output against the expected output

SUT: subject under test (sometimes, system under test, or program under test, PUT)
DOC: Depend-on component

Component Dependencies



Assert the "direct" output (possibly from "indirect" input) against the expected output

SUT: subject under test (sometimes, system under test, or program under test, PUT)
DOC: Depend-on component

Overview of TDD Process

1. From user story to requirements to tests
2. Choosing the first test
3. Breadth-first, depth-first
4. Let's not forget to refactor
5. Adding a bit of error handling
6. Loose ends on the test list
7. Repeat

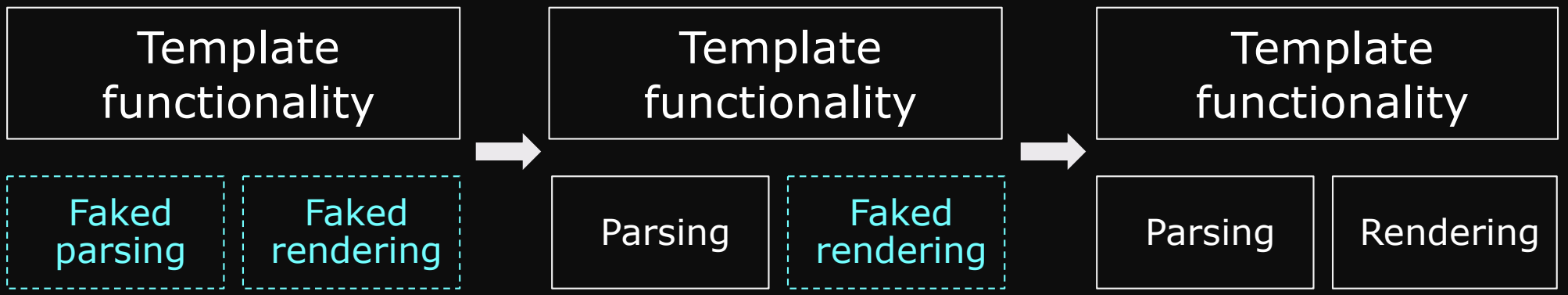
Test first – make it run – make it better

3. Breadth-First, Depth-First

- What to do with a "hard" red phase?
  - Issue is "What to fake" vs. "What to build"
- "Faking" is an accepted part of TDD
  - That is, "deferring a design decision"

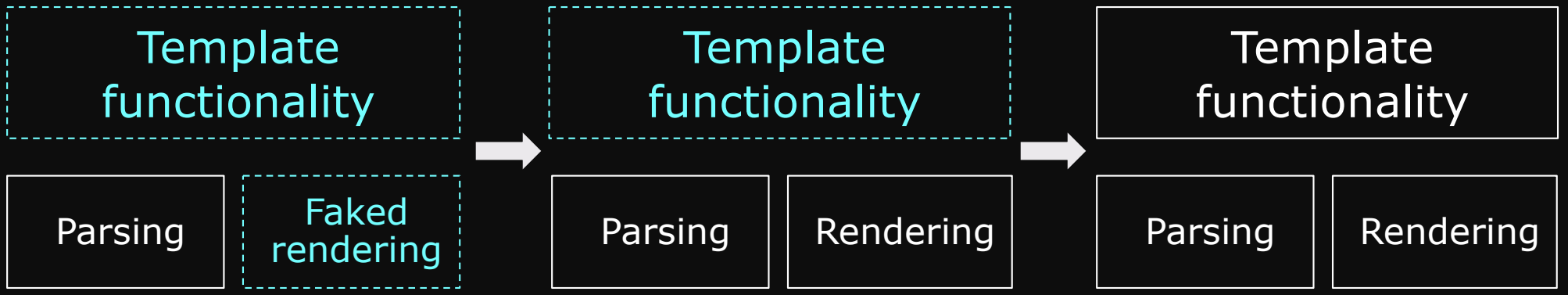
Breadth-First

- Implement the higher-level functionality first by faking the required lower-level functionality



Depth-First

- Implement the lower-level functionality first and only compose the higher-level functionality once all the ingredients are present



Test Double

Movie industry

Stunt double

- Take place of the actor in dangerous scene
- Highly trained
- Meet scene's requirements
- May not be able to act

Software industry

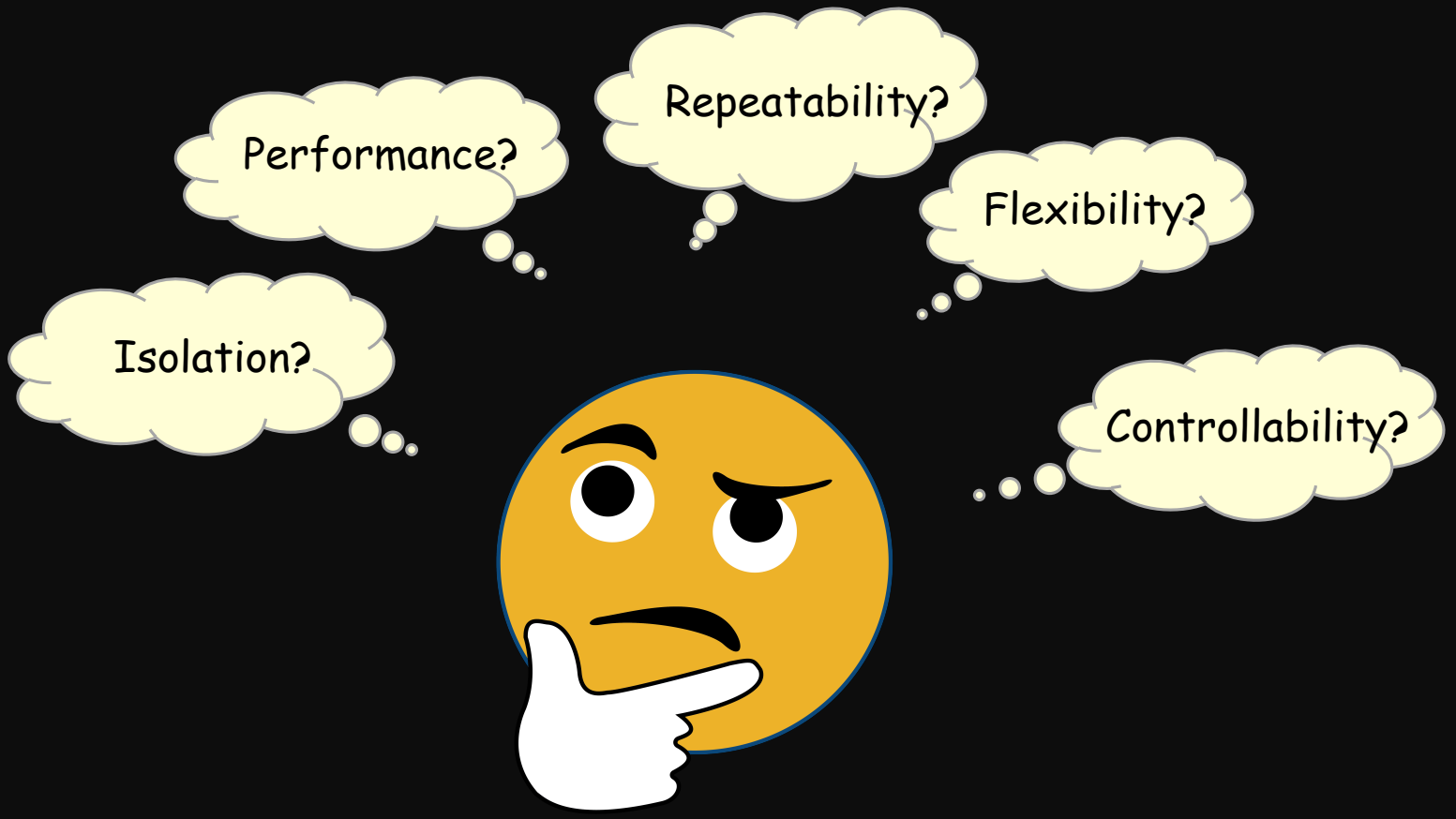
Test double

- Replace the real depend-on component (that may be unavailable, unusable, expensive, dangerous, complicated, or take too long to run, …)
- Look or behave like the depend-on component
- Simpler than the depend-on component

Stand-in for something that would be real in the program

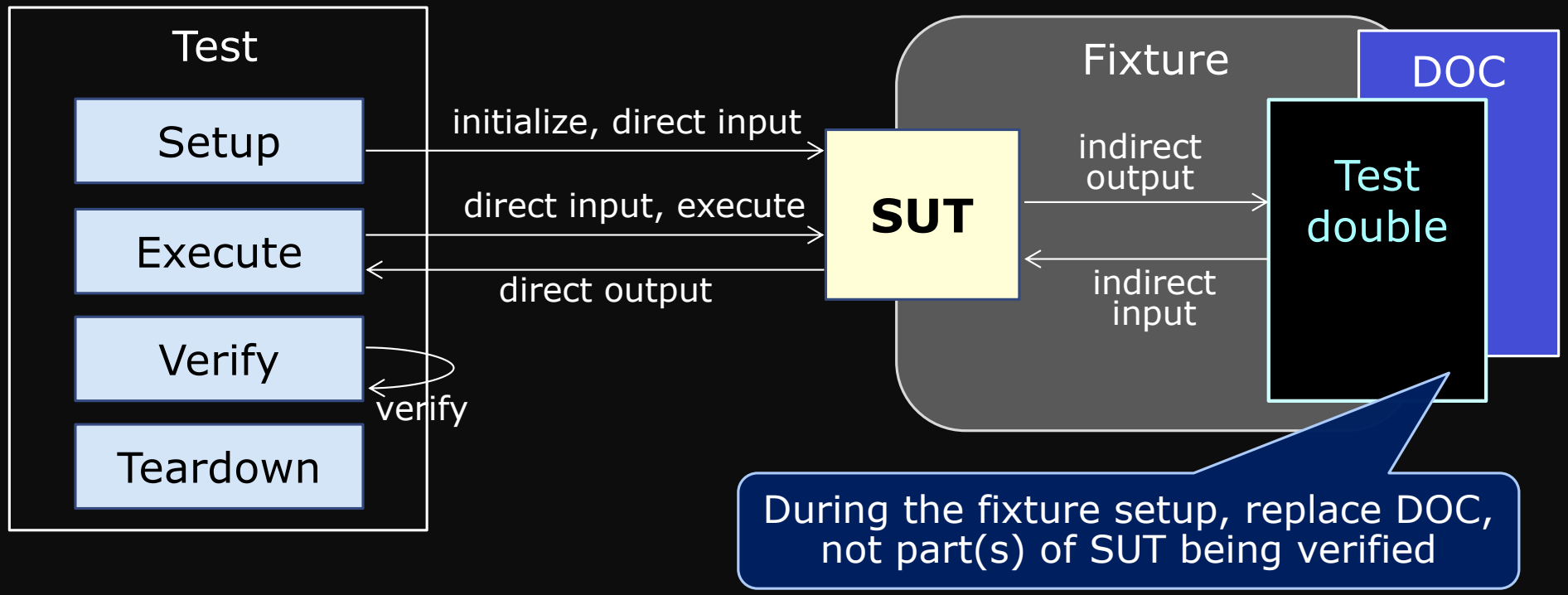
Replace a component the SUT depends on with a "test-specific equivalent"

Why Test Double



[Ref: emoji by Ekarin Apirakthanakorn]

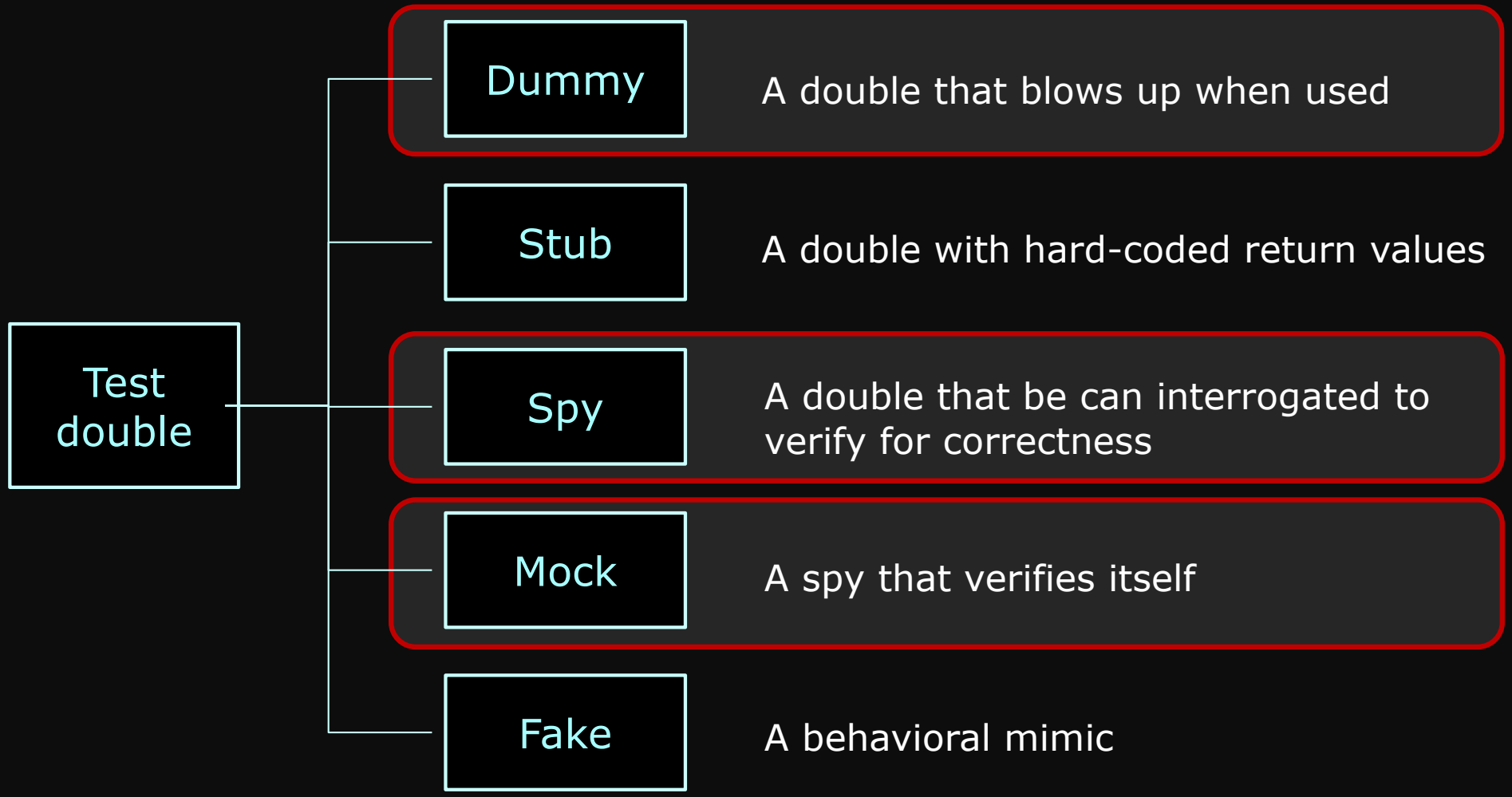
How Test Double Works



- We want to verify code independently from the rest of the system, but the code it depends on is unavailable or unusable
- Need an object that looks (or behaves) like the real component, but is simplified – provides the same APIs so that SUT thinks it is the real one

SUT: subject under test (sometimes, system under test, or program under test, PUT)
DOC: Depend-on component

Types of Test Doubles



Example

- Imagine you are writing and testing a program that controls a rocket launching

```
static void launchRocket(Rocket rocket, LaunchCode code) {
    // ...
}
```

- Can't interact with a real, live rocket
- Need a stand-in for that rocket
- Rely on the idea of a rocket, and allow the runtime to provide a rocket to work with

Rocket interface

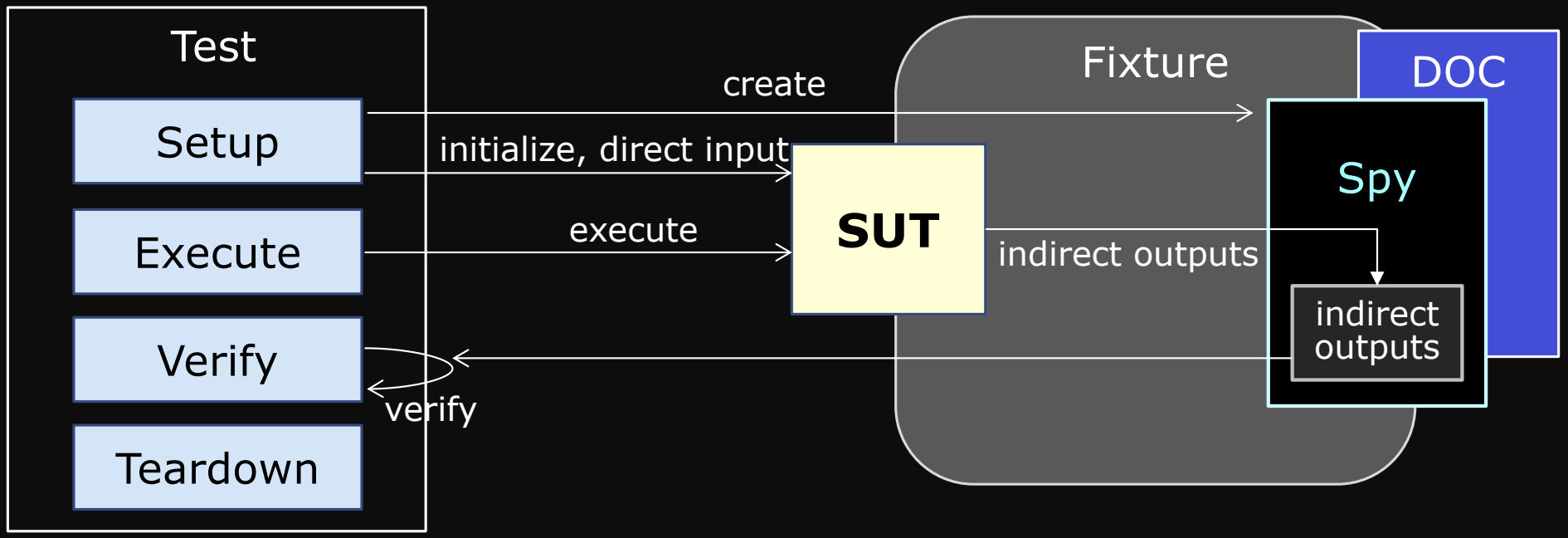
Example: Dummy

```
class DummyRocket implements Rocket {
    @Override
    void launch() {
        throw new RuntimeException();
    }
}
...
static void launchRocket(Rocket rocket, LaunchCode code) {
    try {
        rocket.launch()
    } catch (Exception e) { }
}
```

```
@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
    launchRocket(new DummyRocket(), expiredCode);
}
```

- For the situation when an expired or invalid code is given, use a dummy to ensure the rocket is not fired
- (+) Simple
- (-) May not be intuitive; no traditional setup-act-assert test structure

How Spy Works



- Captures indirect outputs of the SUT and saves them for later used in assertion – act as an "observation point"

SUT: subject under test (sometimes, system under test, or program under test, PUT)
DOC: Depend-on component

Example: Spy

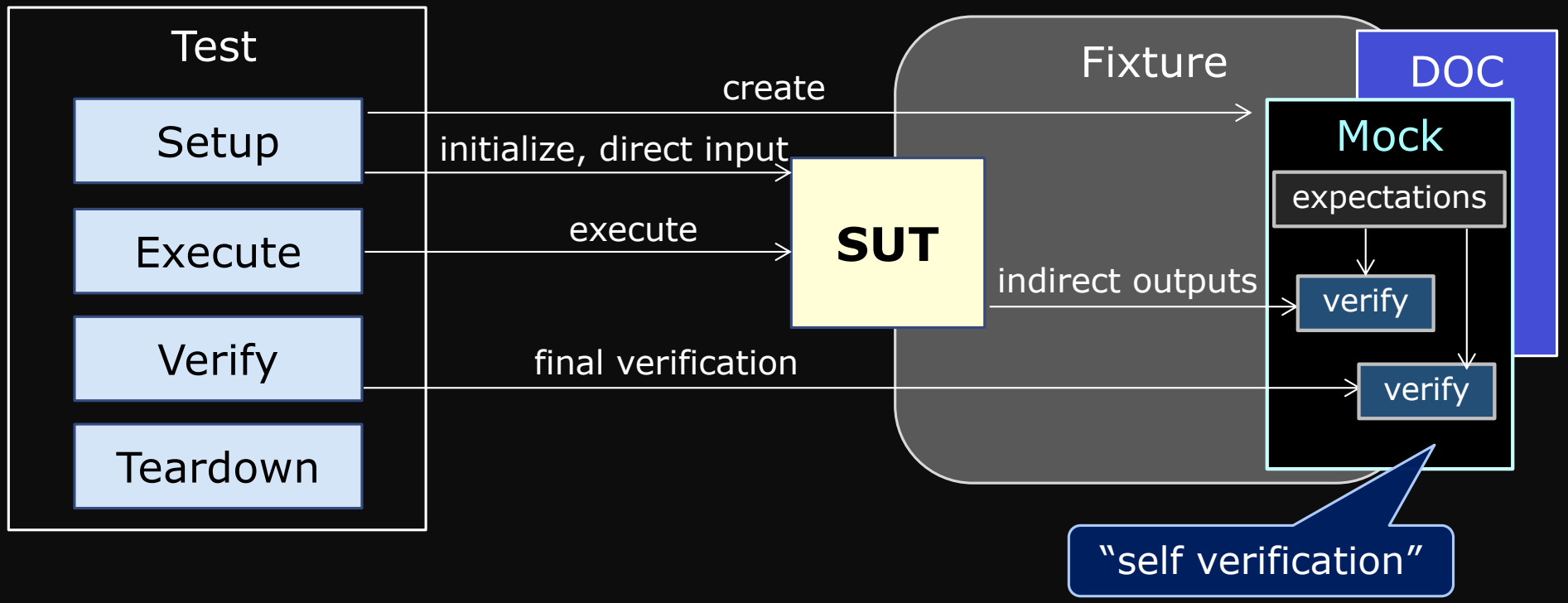
```
class SpyRocket implements Rocket {
   private boolean launchWasCalled = false;
   @Override
   void launch() {
      launchWasCalled = true;
   }
   boolean launchWasCalled() {
      return launchWasCalled;
   }

   ...
   static void launchRocket(Rocket rocket, LaunchCode code) {
      try {
         // if code is invalid or expired, do not lunch
         // else
         rocket.launch()
      } catch (Exception e) { }
   }
}
```

Example: Spy (2)

```
@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
    SpyRocket spy = new SpyRocket();
    launchRocket(spy, expiredCode);
    assertEquals(false, spy.launchWasCalled();
}
```

- Use a spy so that a test can interrogate
- (+) More readable; traditional setup-act-assert test structure
- (-) Tests are coupled to the implementation (must know about the implementation, instead of just focusing on behavioral outputs)

How Mock Works



- Verify that it is being used correctly by the SUT
- Uses as an observation point to verify behavior while avoiding test code duplication

SUT: subject under test (sometimes, system under test, or program under test, PUT)
DOC: Depend-on component

Example: Mock

```
@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
   MockRocket mockRocket = new MockRocket();          // setup
   launchRocket(mockRocket, expiredCode);             // execute
   assertEquals(false, mockRocket.launchWasCalled();  // verify
   assertEquals(true, mockRocket.disableWasCalled();
}
```

```
@Test
void givenUnencryptedLaunchCode_RocketNotLaunched() {
   MockRocket mockRocket = new MockRocket();          // setup
   launchRocket(mockRocket, unencryptedCode);         // execute
   assertEquals(false, mockRocket.launchWasCalled();  // verify
   assertEquals(true, mockRocket.disableWasCalled();
}
```

- For multiple tests with duplicate assertions, the duplicated assertions may be moved to a helper method in a mock class – "avoid test code duplication"

```
void verifyCodeRedAbort() {
   assertEquals(false, mockRocket.launchWasCalled();
   assertEquals(true, mockRocket.disableWasCalled();
}
```

Example: Mock (2)

```
class MockRocket implements Rocket {
   private boolean launchWasCalled = false;
   private boolean disabledWasCalled = false;

   @Override
   void launch() {
      launchWasCalled = true;
   }

   @Override
   void disable() {
      disableWasCalled = true;
   }

   boolean launchWasCalled() {
      return launchWasCalled;
   }

   boolean disabledWasCalled() {
      return disabledWasCalled;
   }

   void verifyCodeRedAbort() {
      assertEquals(false, launchWasCalled();
      assertEquals(true, disableWasCalled();
   }
   ...
   static void launchRocket(Rocket rocket, LaunchCode code) {
      try {
         rocket.launch()
      } catch (Exception e) { }
   }
}
```

Example: Mock (3)

```
@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, expiredCode);
    assertEquals(false, mockRocket.launchWasCalled();
    assertEquals(true, mockRocket.disableWasCalled();
}
```

```
@Test
void givenUnencryptedLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, unencryptedCode);
    assertEquals(false, mockRocket.launchWasCalled();
    assertEquals(true, mockRocket.disableWasCalled();
}
```

- The tests can no longer interrogate the mock through its public interface. They can only verify that a code red abort happened.

Example: Mock (4)

```
@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, expiredCode);
    mockRocket.verifyCodeRedAbort();
}
```

```
@Test
void givenUnencryptedLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, unencryptedCode);
    mockRocket.verifyCodeRedAbort();
}
```

Example: Mock (5)

```
@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, expiredCode);
    mockRocket.verifyCodeRedAbort();
}
```

```
@Test
void givenUnencryptedLaunchCode_RocketNotLaunched() {
    MockRocket mockRocket = new MockRocket();
    launchRocket(mockRocket, unencryptedCode);
    mockRocket.verifyCodeRedAbort();
}
```

- Notice the duplicate set up, refactor the tests

Example: Mock (6)

```
...
MockRocket mockRocket;
...
@BeforeEach
void setup () {
   mockRocket = new MockRocket();
}
```

```
@Test
void givenExpiredLaunchCode_RocketNotLaunched() {
   launchRocket(mockRocket, expiredCode);
   mockRocket.verifyCodeRedAbort();
}
```

```
@Test
void givenUnencryptedLaunchCode_RocketNotLaunched() {
   launchRocket(mockRocket, unencryptedCode);
   mockRocket.verifyCodeRedAbort();
}
```

- Refactor a spy to a mock
- Refactor the code to clean up and remove code smells (extract and move)
- (+) Decrease duplication, centralize the assertions, increase maintainability
- (-) To understand the tests, must inspect the mock

Be Careful When Using Replacements

- We are testing the SUT and test double in a different configuration from that which will be used in production
- If a stand-in component does not mimic the DOC behavior correctly, it can falsify the test results.
- We must emphasize on meeting the common specification.

We don't want to build perfect cars for crash-test dummies, but fail on real humans

Just enough to pass a test

- The replacements often simulate only partial behavior.
- There may be many ways to implement a certain functionality.
- Using third-party libraries or framework interfaces may introduce different behavior and increase complexity in implementation.
- The different implementation may make it difficult to refactor the code without breaking our tests.

Summary

- Test doubles serve various purposes including:
  - Indirect input provisioning
  - Recording of indirect output
  - Immediate verification of interactions
- Fake it till you make it
  - Fast or independent from environmental influences
- Verify behavior with mocks
- Increase efficiency with mock frameworks
  - EasyMock (http://easymock.org/) – used in Koskela book
  - Jmock (http://www.jmock.org/)
  - Mockito (http://site.mockito.org/) – popular