17-423/723: Designing Large-scale Software Systems

Design for Testability I & II

Feb 10 & 12, 2025

Logistics

- M1 due today
- M2 released later today; due Feb 28
  - Build and test an initial prototype of the scheduling app
  - Please start early! This will take longer than expected
- Midterm next Monday, Feb 17
  - Covers up to this Wednesday's lecture (testability)
  - Open book, but no electronics (laptop, phone, etc.,) allowed
  - Similar to homework questions and recitation activities

Learning Goals

- Describe the basic elements of testing
- Describe testability and its relationship to testing
- Identify controllability and observability challenges in testing
- Apply test doubles to enable testing with dependencies
- Apply design principles to improve the testability of a system

Testing & Testability

Testing Basics

- **Testing**: Execution of a piece of code on test data in a controlled environment, with an expected output.
- **Test case**: Given a program:
  - A specific set of inputs to that program
  - An expected output
  - An **oracle** that determines whether the actual output of the program matches the expected output.
- **Test suite**: A collection of test cases.
- **Oracle problem**: Figuring out what the expected output of the program should be, for a given input.

Testing Basics

- **Goals of testing**
  - Revealing failures (most common use!)
  - Assessing quality (difficult but still relevant)
  - Identifying the specification for a function/component through the development of oracles
- “Complete testing” is impossible, and testing for bugs is more successful than for correctness.
- BUT testing can be effective at establishing quality attributes when approached in a mindful and disciplined manner

“Testing can show only the presence, not the absence of bugs.”
- Edsger W. Dijkstra

Testability

- The amount of effort required to create and execute automated tests for a system
  - Including: Setting up a test environment, developing the oracle, running a component under a specific input, checking the output
- **Some systems are more testable than others!**
- Testing is nice; testability is better.
- …because testing won't make bad code good, and you can't test well if the code itself is untestable

Example: Online Shopping Site

amazon
Deliver to Ghana
All
Search Amazon
EN
Hello, sign in
Account & Lists
Returns
& Orders
0
Cart

All
Today's Deals
Customer Service
Registry
Gift Cards
Sell
Shop the Gaming Store

Kitchen favorites
under $50

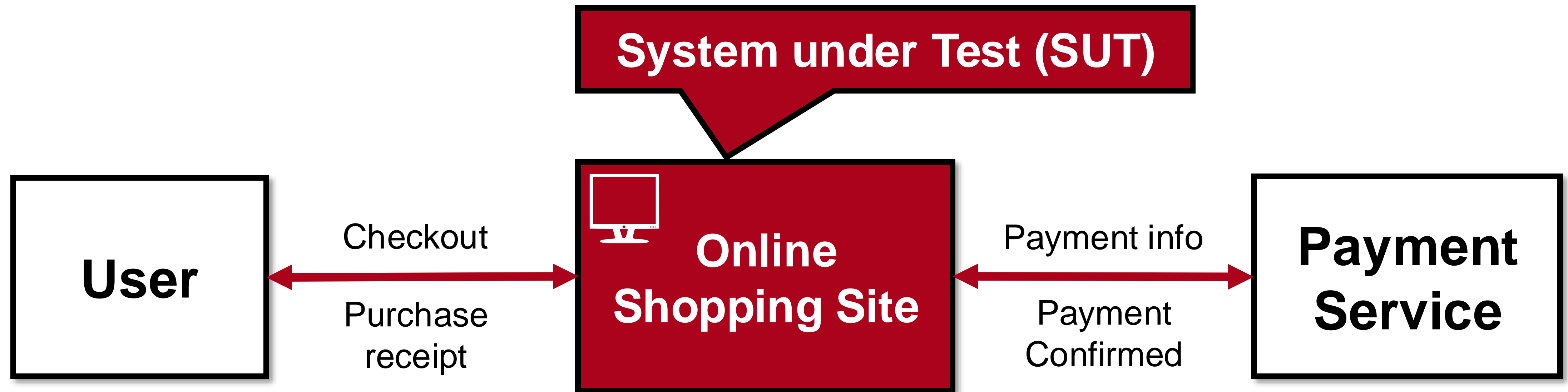
Gaming accessories
Headsets
Keyboards
Computer mice
Chairs

Shop deals in Fashion
Jeans under $50
Tops under $25
Dresses under $30
Shoes under $50

Refresh your space
Dining
Home
Kitchen
Health and Beauty

New home arrivals under $50
Kitchen & dining
Home improvement
Décor
Bedding & bath

Example: Online Shopping Site

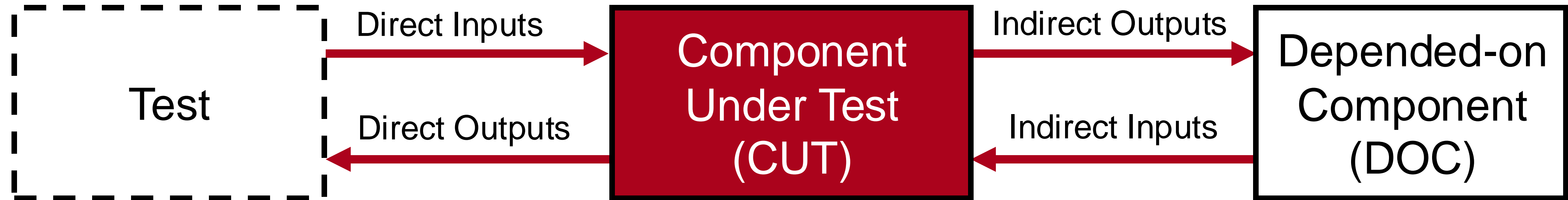


Suppose that we want to test the “checkout” workflow

The user has an option to use an external payment service (e.g., Paypal)

Q. What are some challenges in testing this system?

Dependencies make testing hard

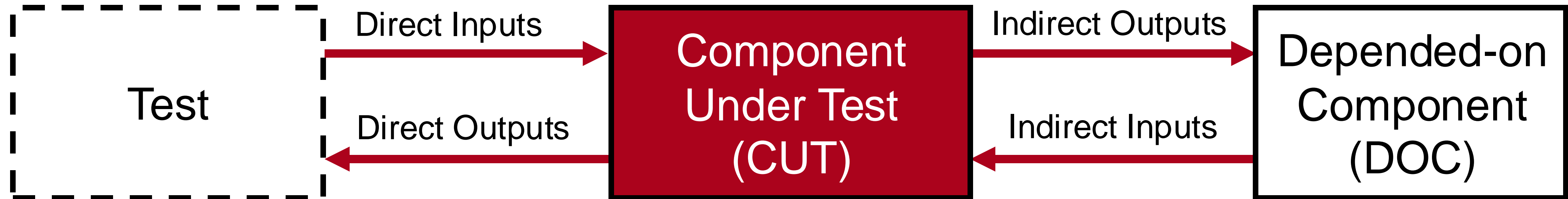


- A program/component to be tested often **depends** on other components (DOC)
- Deploying/executing DOC for testing might be expensive, slow, or infeasible (e.g., external API, file I/O, databases)
- Testing CUT might require executing DOC under specific inputs and/or getting DOC to produce specific outputs

Controllability & Observability

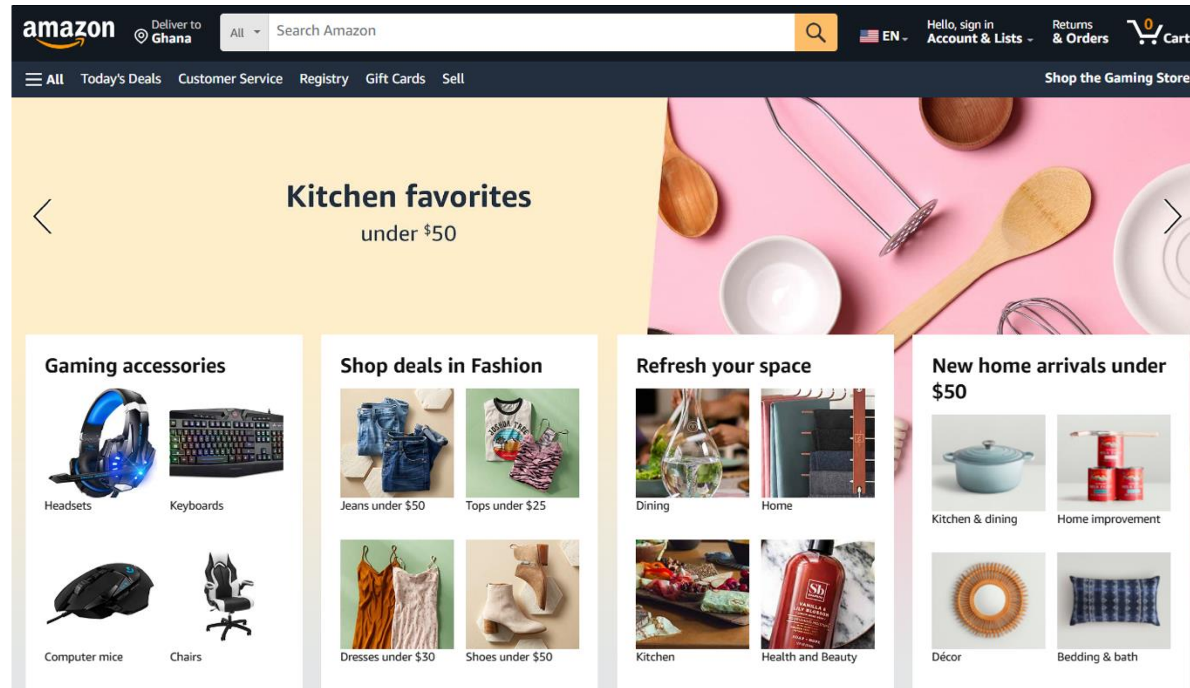
- **Controllability**: How easy is it to bring a program to a particular state and/or inject it with a specific set of inputs?
- **Observability**: How easy is it to observe the behavior of a program, in terms of its outputs, quality attributes, or effects on its state?
- These two factors significantly determine the amount of effort required in creating and running test cases – i.e., testability!

Dependencies make testing hard



- Observing (1) indirect input & output interactions between CUT and DOC and (2) internal state of CUT is an **observability** challenge.
- Getting CUT and DOC to behave in a particular way (e.g., generate a particular output) is a **controllability** challenge.

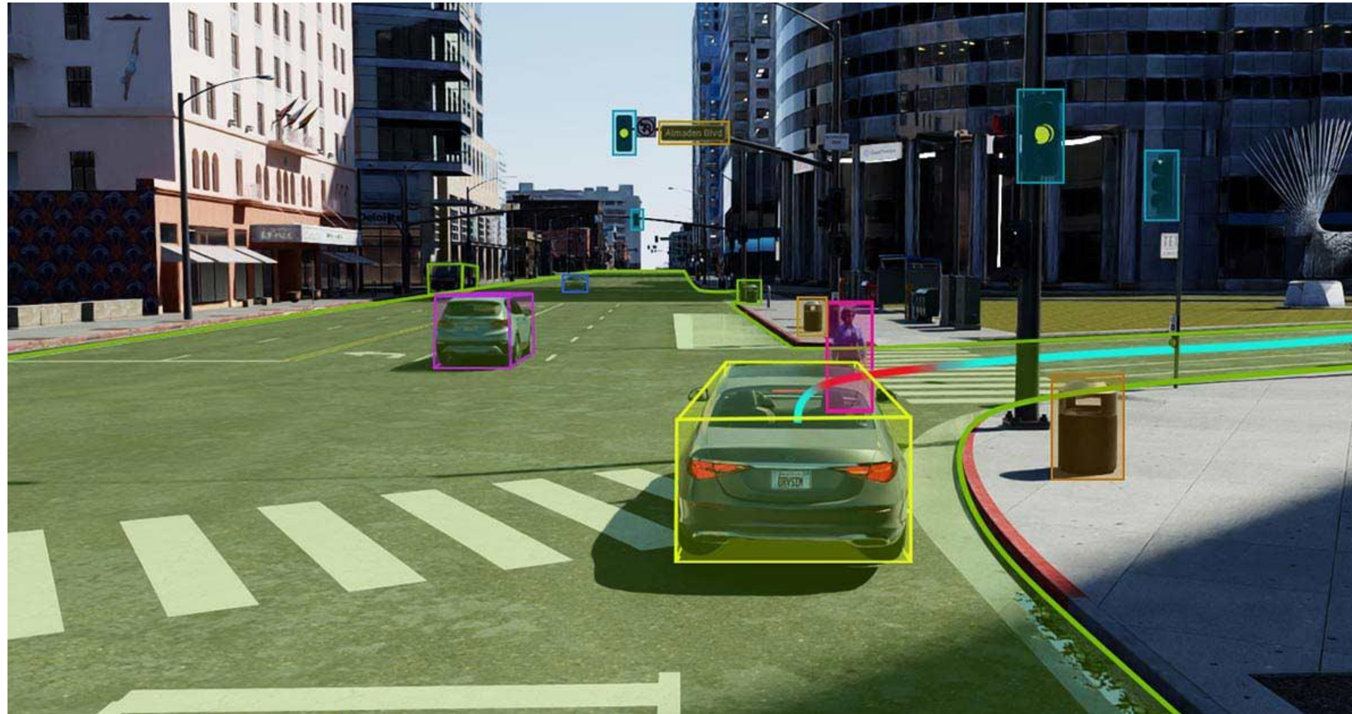
Controllability & Observability: Examples



Online shopping site

- **Controllability**: How to get the payment service to respond with a particular output (e.g., deny payment for an invalid credit card number)?
- **Observability**: How to observe the status of the checkout process when the payment is denied?

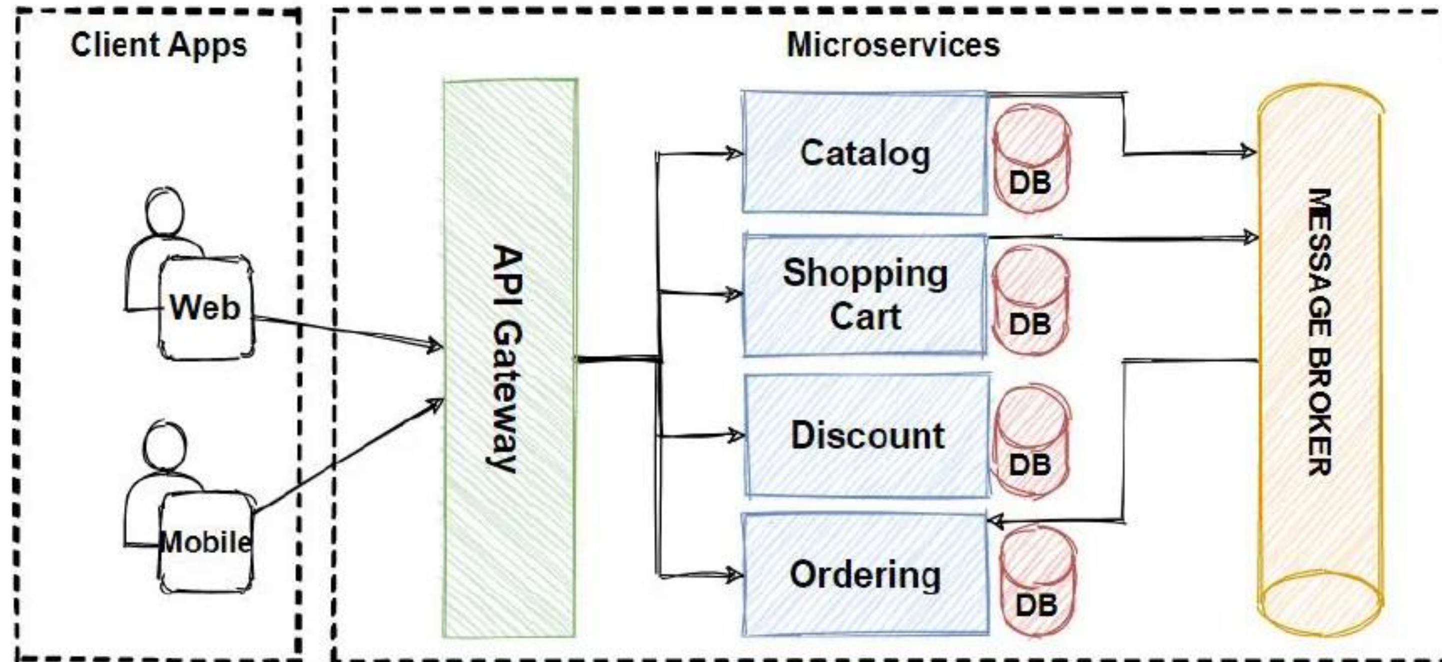
Controllability & Observability: Examples



Self-driving car simulator

- **Controllability**: How to set up the simulation environment to test the vehicle software under a particular road setting?
- **Observability**: How to track the locations of the car and obstacles to detect when a collision is possible?

Controllability & Observability: Examples



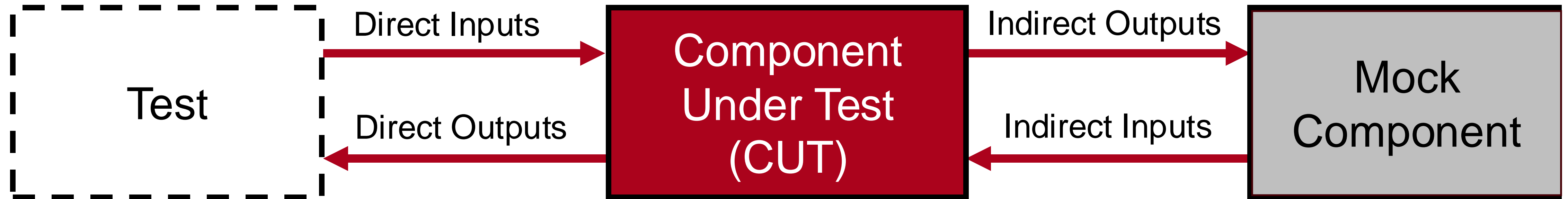
Distributed system

- **Controllability**: How to set up the distributed network to test the system under a failure scenario (e.g., certain servers being down)?
- **Observability**: How to measure the availability of a service during the failure scenario?

Test Doubles

- Components that act as a replacement for a dependency (DOC)
  - Enables a component (CUT) to be tested in isolation without the presence of DOC
- **Test stub**: Provides predefined responses to a function
- **Mock component:** Simulates the behavior of a component in limited ways
- **Test spy**: Track method calls and arguments, to be used for later verification (e.g., check whether a method was called)

Mock Component



- Simulates the behavior of a component in limited ways
- Useful for testing when the actual component:
  - Has states that are difficult to create or reproduce
  - Returns non-deterministic outputs
  - Is slow to run (e.g., database query)
  - Does not exist yet

```python
class PaymentProcessor:
    def charge(self, amount):
        """Calls an external API to process the payment"""
        raise NotImplementedError(
                    "Real payment processing is not implemented!")

class ShoppingCart:
    def __init__(self, payment_processor):
        self.items = []
        self.payment_processor = payment_processor

    def add_item(self, name, price):
        self.items.append({"name": name, "price": price})

    def get_total(self):
        return sum(item["price"] for item in self.items)

    def checkout(self):
        total = self.get_total()
        # External dependency!
        return self.payment_processor.charge(total)
```

External dependency

Component under test

```python
import unittest
from unittest.mock import Mock

class TestShoppingCart(unittest.TestCase):
    def test_checkout_calls_payment_processor(self):
        mock_payment_processor = Mock()
        mock_payment_processor.charge.return_value =
            "Payment Successful"

        cart = ShoppingCart(mock_payment_processor)
        cart.add_item("Laptop", 1000)
        cart.add_item("Mouse", 50)

        result = cart.checkout()

        # Verify return value
        self.assertEqual(result, "Payment Successful")
        # Verify that the mock method was called
        mock_payment_processor.charge.
            assert_called_once_with(1050)

if __name__ == "__main__":
    unittest.main()
```

Mock object library in Python

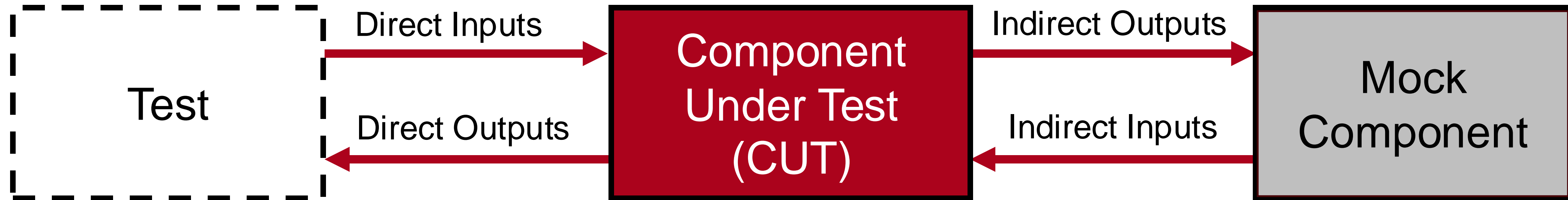
Set up the mock object with pre-determined value

Create a shopping cart with the mock payment processor

Call the function to test

Check whether the test passed

Mock Component



- Simulates the behavior of a component in limited ways
- Useful for testing when the actual component:
  - Has states that are difficult to create or reproduce
  - Returns non-deterministic outputs
  - Is slow to run (e.g., database query)
  - Does not exist yet
- **Improving testability makes it easier to create/use test doubles!**

Design Principles for Testability

Changeability & Testability

- Changeability is strongly related to testability!
- **Recall**: Principles for changeability
  - Information hiding
  - Single-responsibility
  - Interface segregation
  - Dependency inversion
- They have a common goal: Reduce dependencies between components to make them easier to change independently
- **Discussion**: How do these principles help improve (or reduce) testability?

Changeability & Testability

- Information hiding principle
  - Makes a component easier to test in isolation, by reducing dependencies
  - Hides details that may be needed for testing (reduce observability)
- Single-responsibility principle
  - Helps make test suite for a component more focused and simpler
- Interface segregation principle
  - Makes it easier to create stubs/mocks for dependencies, by reducing the size of the interface to be implemented
- Dependency inversion principle
  - Makes it easier to create stubs/mocks, by abstracting away details irrelevant to the high-level business logic

Design Principles for Testability

1. Separate business logic from infrastructure code
2. Improve controllability through dependency injection
3. Improve observability through accessor methods
4. Reduce test complexity through separation of concerns

Design Principles for Testability

1. **Separate business logic from infrastructure code**
2. Improve controllability through dependency injection
3. Improve observability through accessor methods
4. Reduce test complexity through separation of concerns

Separate Business Logic from Infrastructure Code

- **Infrastructure**: Parts of the system that handles an external dependency
  - Database queries, calls to web services, file read/writes, etc.,
- Business/application logic often depends on the infrastructure
- To test business logic, also need to observe & control the interactions with the infrastructure
  - Ideally, business logic should be tested in isolation without dealing with details about the infrastructure
  - Much easier to do if there is a clear separation between the two!

```java
1 public class InvoiceFilter {
2   private List<Invoice> all () {
3     try {
4       Connection connection = DriverManager.getConnection("db", "root", "");
5       PreparedStatement ps =
6               connection.prepareStatement("select * from invoice"));
7       Results rs = ps.executeQuery();
8       List<Invoice> allInvoices = new ArrayList<>();
9       while (rs.next()) {
10         allInvoices.add(new Invoice(
11           rs.getString("name"), rs.getInt("value")));
12       }
13       ps.close ();
14       connection.close();
15       return allInvoices;
16      } catch (Exception e) {
17      // ..handles ….
18    }
19    public List<Invoice> lowValueInvoices () {
20     List <Invoice> issuedInvoices = all();
21     return issuedInvoices.all().stream().
22         filter(invoice -> invoice.value < 100). Collect(toList());
23    }
24 }
```

Gets all invoices from a database.

Code execute “select” query (details unimportant)

Database APIs often throw exceptions.

Returns all low value Invoices, relying on the private **all**() method.

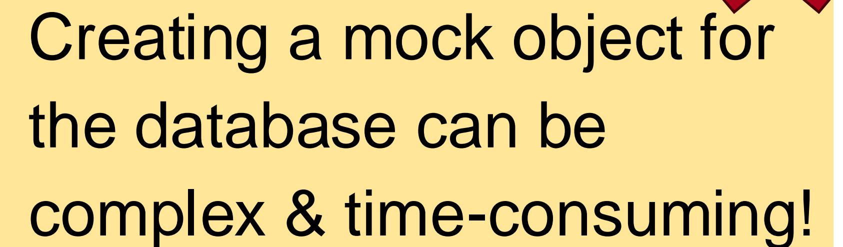
```
1 public class InvoiceFilter {
2   private List<Invoice> all () {
3     try {
4       Connection connection = DriverManager.getConnection("db", "root", "");
5       PreparedStatement ps =
6               connection.prepareStatement("select * from invoice"));
7       Results rs = ps.executeQuery();
8       List<Invoice> allInvoices = new ArrayList<>();
9       while (rs.next()) {
10         allInvoices.add(new Invoice(
11           rs.getString("name"), rs.getInt("value")));
12       }
13       ps.close ();
14       connection.close();
15       return allInvoices;
16      } catch (Exception e) {
17      // ..handles ….
18    }
19    public List<Invoice> lowValueInvoices () {
20     List <Invoice> issuedInvoices = all();
21     return issuedInvoices.all().stream().
22         filter(invoice -> invoice.value < 100). Collect(toList());
23    }
24 }
```

Note: Infrastructure code is intermixed with business logic! Can't avoid database access when testing "lowValueInvoices"

More complex code, more bugs possible! (e.g., bugs related to SQL and business logic)

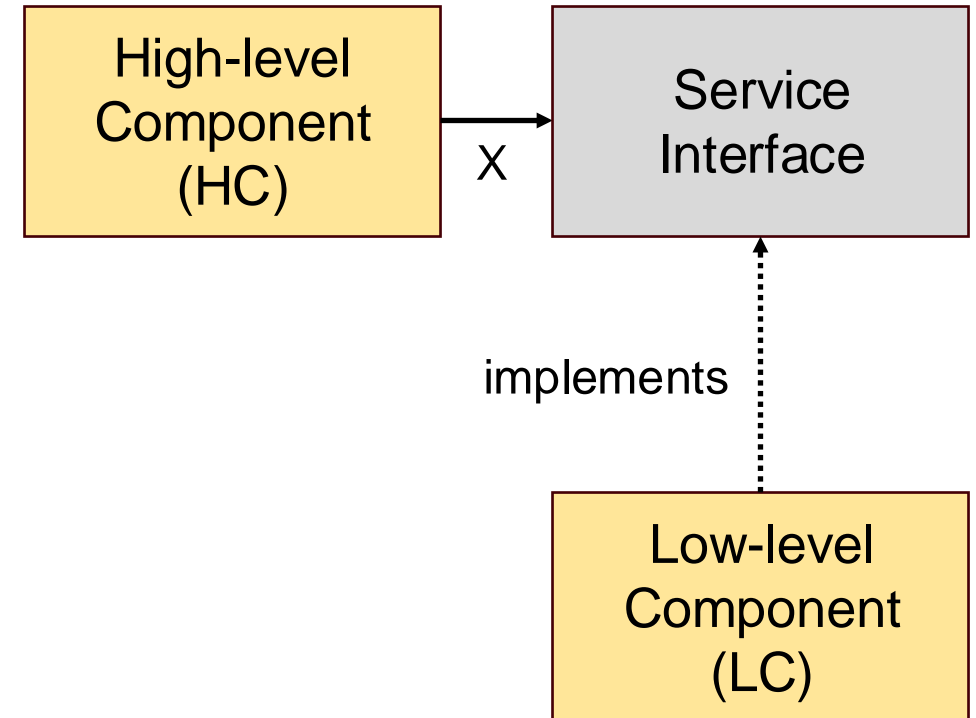
(UI code is another example of code that's often mixed into business logic)

```java
private MockedConstruction<DatabaseConnection> databaseConstruction;
private MockedConstruction<IssuedInvoices> issuedConstruction;

private void setUpConstruction(MockInitializer<DatabaseConnection> databaseInitialize,
MockInitializer<IssuedInvoices> issuedInitializer){
   databaseConstruction = mockConstruction(DatabaseConnection.class, databaseInitialize);
   issuedConstruction = mockConstruction(IssuedInvoices.class, issuedInitializer);
}

@Test
public void filterInvoices(){
   // ...

   setUpConstruction((mock, context) -> { // database initializer
       // Probably some other internal databaseConnection stubs
   }, (mock, context) -> { // issued invoice initializer
       when(mock.all()).thenReturn(listOfInvoices);
   });

   InvoiceFilter filter = new InvoiceFilter();

   assertThat(filter.lowValueInvoices()).containsExactlyInAnyOrder(john, steve);

   // ...
}
```

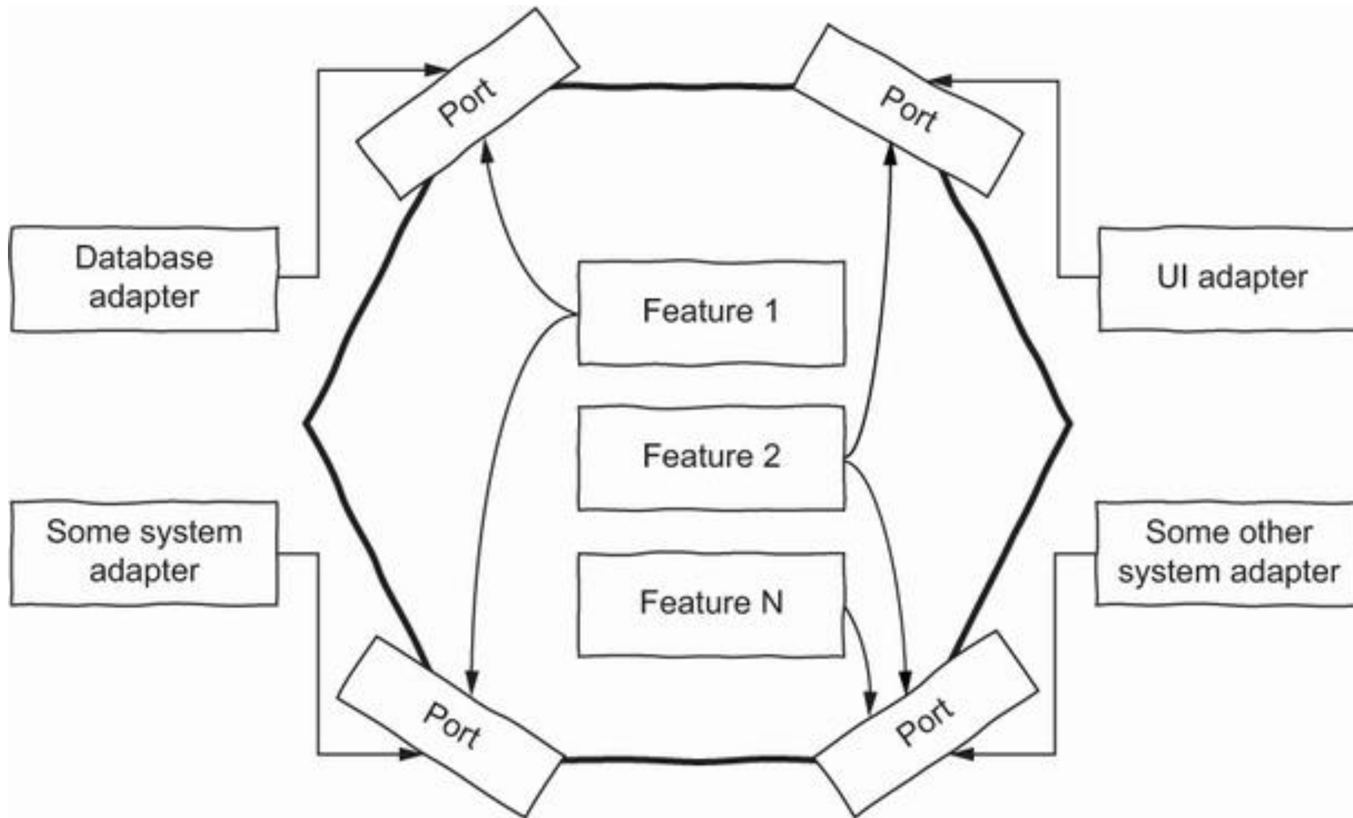


Recall: Dependency Inversion Principle (DIP)

- “High-level” components (i.e., business logic) should not directly depend “low-level” components (i.e., infrastructure)
- Invert the dependency from HC to LC by introducing an intermediate abstraction (i.e., interface)
- HC depends on the interface; details about LC are hidden
- This makes it easier to test HC by inserting a **mock for the interface (not LC)**!

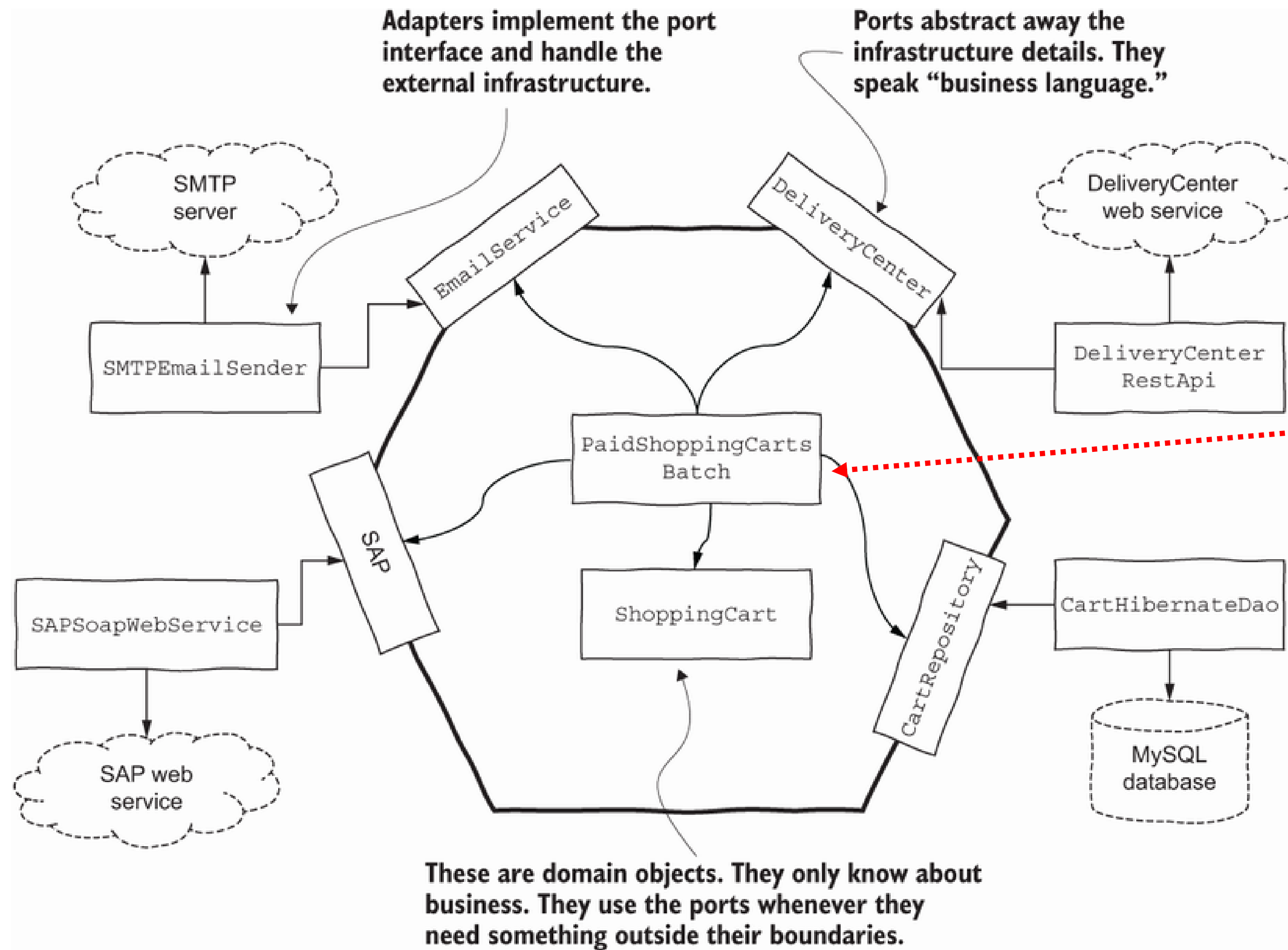


Recall: Hexagonal Architecture



- **Inward** dependency only: All external components depend on core business logic (**dependency inversion**!)
- **Port:** An interface between the core logic and an external component
- **Adapter**: Implements a port interface; links the interface to a concrete implementation

Example: Shopping Cart Checkout



The shopping cart logic does not need to know how the external services are implemented!

```java
public class PaidShoppingCartsBatch {
  // Ports (i.e., interfaces to external services)
  private ShoppingCartRepository db;
  private DeliveryCenter deliveryCenter;
  private CustomerNotifier notifier;

  public PaidShoppingCartsBatch(ShoppingCartRepository db,
          DeliveryCenter deliveryCenter, CustomerNotifier notifier) {
    // initialize object
  }

  public void processAll() {
    // Get all carts paid today
    List<ShoppingCart> paidShoppingCarts = db.cartsPaidToday();
    for (ShoppingCart cart : paidShoppingCarts) {
      // Create delivery order for the items in the cart
      LocalDate estimatedDayOfDelivery = deliveryCenter.deliver(cart);
      cart.markAsReadyForDelivery(estimatedDayOfDelivery);
      // Update the information about the cart
      db.persist(cart);
      // Notify the user of the estimated delivery date
      notifier.sendEstimatedDeliveryNotification(cart);
    }
  }
}
```

- **Feature to test**: Batch process the set of shopping carts for the day. For each shopping cart:
  - Get estimated delivery date from the delivery center
  - Mark the cart as being delivered & update in DB
  - Notify the user of the delivery date
- Each of these three tasks involves an external dependency
- **Goal**: Test that the processAll() calls these services correctly

Port for Shopping Cart DB

```
public interface ShoppingCartRepository {
  List<ShoppingCart> cartsPaidToday();
  void persist(ShoppingCart cart);
}
```

Provides an abstraction over all database related operations

Adapter for ShoppingCartRepository Port

```
public class ShoppingCartHibernateDao
  implements ShoppingCartRepository {
  @Override
  public List<ShoppingCart> cartsPaidToday() {
    // A query to get the list of all
    // invoices that were paid today
  }

  @Override
  public void persist(ShoppingCart cart) {
    // A query to persist the cart
    // in the database
  }
}
```

Connects the port to a specific external database service (Hibernate + MySQL DB); can be substituted with adapters for alternative DB engines

Mocks created with the Mockito framework

```java
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
public class PaidShoppingCartsBatchTest {
  @Mock ShoppingCartRepository db;
  @Mock private DeliveryCenter deliveryCenter;
  @Mock private CustomerNotifier notifier;
  @Test
  void theWholeProcessHappens() {
    PaidShoppingCartsBatch batch = new PaidShoppingCartsBatch(db,
                                         deliveryCenter, notifier);
    ShoppingCart someCart = new ShoppingCart();
    LocalDate someDate = LocalDate.now();

    when(db.cartsPaidToday()).thenReturn(Arrays.asList(someCart));
    when(deliveryCenter.deliver(someCart)).thenReturn(someDate);

    batch.processAll();
    // Verify the test outcome by checking the states of components
    verify(deliveryCenter).deliver(someCart);
    verify(notifier).sendEstimatedDeliveryNotification(someCart);
    verify(db).persist(someCart);
  }
}
```

Create mock objects

Specify how the mocks should behave

Verify that the methods were called with the specific arguments

Separate Business Logic from Infrastructure Code

- Infrastructure: Parts of the system that handles an external dependency
  - Database queries, calls to web services, file read/writes, etc.,
- Business/application logic often depends on the infrastructure
- To test the business logic, also need to observe & control the interactions with the infrastructure
- **Apply an interface abstraction** to separate the two as much as possible!
  - This makes it easier to create stubs/mocks to test the business logic in isolation; these can be created without knowing the details of the infrastructure code

Design Principles for Testability

1. Separate business logic from infrastructure code
2. **Improve controllability through dependency injection**
3. Improve observability through accessor methods
4. Reduce test complexity through separation of concerns

Improve Controllability through Dependency Injection

- Dependency injection
  - A component **receives** one or more components that it depends on
  - Dependencies are created and “injected” into the component by an **external** entity (i.e., client), instead of being created internally

```python
class Processor:
    def process(self):
        fetcher = DataFetcher()
        data = fetcher.fetch_data()
        return f“Processed: {data}”

class DataFetcher:
    def fetch_data(self):
        fetched = fetch_external_data()
        return fetched

# Unit test
import unittest
class TestProcessor(unittest.TestCase):
    def test_process(self):
        processor = Processor()
        result = processor.process()
        expected = … # expected data for this test
        # Cannot easily control the return value of fetch_data()
        self.assertEqual(result, "Processed: ” + expected)

if __name__ == "__main__":
    unittest.main()
```

fetcher is a dependency of Processor

Processor directly instantiates DataFetcher
Difficult to control how this is done outside of Processor!

For testing, want to control the return value of fetch_data

```python
class Processor:
    def __init__(self, fetcher: DataFetcher):
        self.fetcher = fetcher  # Dependency injection
    def process(self):
        data = self.fetcher.fetch_data()
        return f"Processed: {data}"

class DataFetcher:
    def fetch_data(self):
        fetched = fetch_external_data()
        return fetched

# Unit test
import unittest
from unittest.mock import Mock
class TestProcessor(unittest.TestCase):
    def test_process(self):
        expected = … # expected data for this test
        mock_fetcher = Mock()
        mock_fetcher.fetch_data.return_value = expected
        processor = Processor(mock_fetcher)
        result = processor.process()

        self.assertEqual(result, "Processed: " + expected)
```

Processor expects "fetcher" through its constructor instead of instantiating it

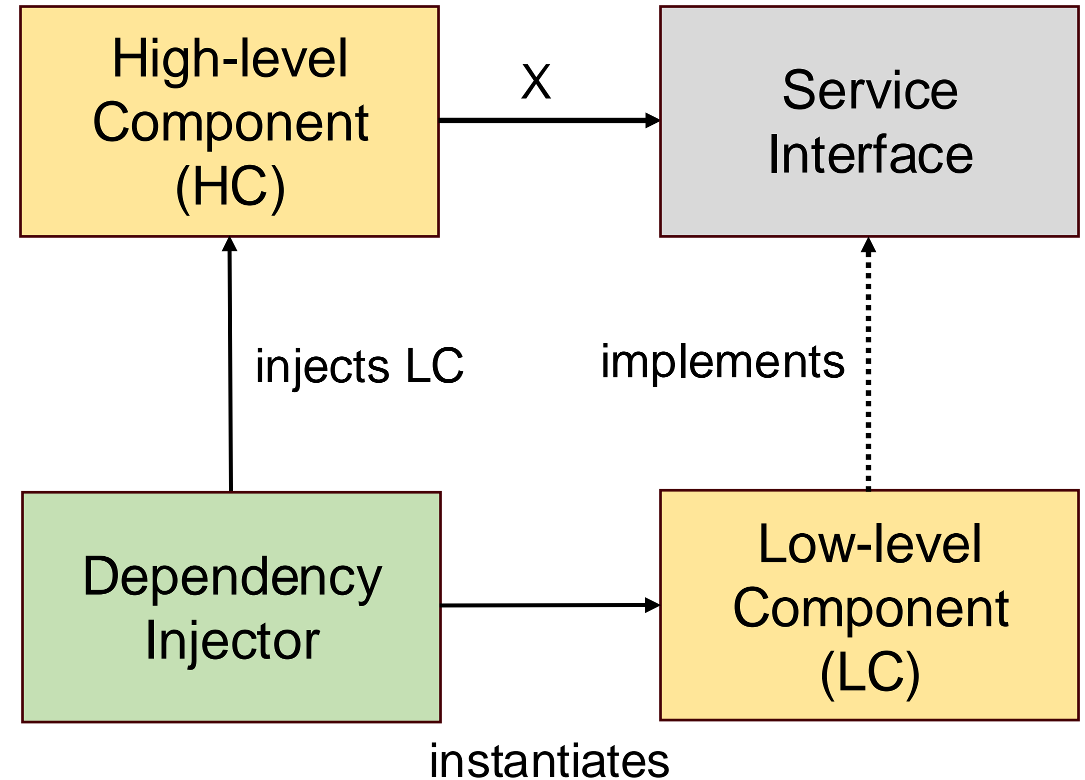
DataFetcher doesn't change

fetcher can be mocked with a specific return value for fetch_data

The mock fetcher is "injected" into Processor

Dependency Injection & Dependency Inversion Principle (DIP)

- DIP is often achieved through dependency injection
  - Select an LC that implements Service Interface
  - Inject that LC into HC
  - HC interacts with LC through the interface; does not need to directly reference or interact with LC

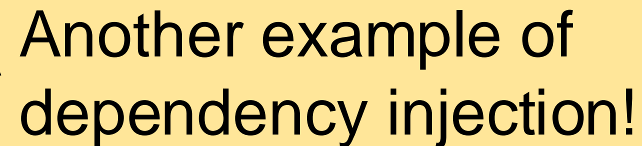


Port for Shopping Cart DB (i.e., service interface)

```
 public interface ShoppingCartRepository {
  List<ShoppingCart> cartsPaidToday();
  void persist(ShoppingCart cart);
}
```

Logic for batch processing carts (i.e., high-level component)

```
public class PaidShoppingCartsBatch {
  // Ports (i.e., interfaces to external services)
  private ShoppingCartRepository db;
  private DeliveryCenter deliveryCenter;
  private CustomerNotifier notifier;

  public PaidShoppingCartsBatch(ShoppingCartRepository db,
          DeliveryCenter deliveryCenter, CustomerNotifier notifier) {
    // initialize object
  }
  ...
}
```



Improve Controllability through Dependency Injection

- Dependency injection
  - A component **receives** one or more components that it depends on
  - Dependencies are created and "injected" into the component by an **external** entity (i.e., client), instead of being created internally
- **Benefit**: Separates the logic of creating dependencies from the receiving component
  - Improves **controllability** for testing, since dependencies can be more easily mocked with specific behaviors and injected into CUT
- **Q. Any potential downsides to dependency injection?**

```java
 1 public class ChristmasDiscount {
 2     private final Clock clock;
 3     public ChristmasDiscount(Clock clock) { … }
 4 
 5     public double applyDiscount(double rawAmount) {
 6       LocalDate today = clock.now();
 7       double discount = 0;
 8       boolean isChristmas = today.getMonth() == Month.DECEMBER
 9         && today.getDayOfMonth() == 25;
10       if(isChristmas);
11         discount = 0.15;
12       return rawAmount - (rawAmount * discount);
13     }
14 }
15 
16 public class ChristmasDiscount {
17     public double applyDiscount(double rawAmount, LocalDate today) {
18       double discount = 0;
19       boolean isChristmas = today.getMonth() == Month.DECEMBER
20         && today.getDayOfMonth() == 25;
21       if(isChristmas);
22         discount = 0.15;
23       return rawAmount - (rawAmount * discount);
24     }
25 }
```

- Two versions of code that applies a holiday discount
- Bottom uses a dependency injection for the date
- **Q. Which is better?**
- Bottom is simpler but forces a dependency onto clients; all callers of applyDiscount must pass a LocalDate!
- i.e., dependency injection introduces **trade-offs** between controllability vs. complexity of client code

Design Principles for Testability

1. Separate business logic from infrastructure code
2. Improve controllability through dependency injection
3. **Improve observability through accessor methods**
4. Reduce test complexity through separation of concerns

Improve Observability through Accessor Methods

- **Oracle**: For each test case, determine whether the test passed successfully or not
- Sometimes, this requires observing the internal state/behavior of a component(s) that is, by default, not publicly accessible
- One way to improve observability is to augment the component with **accessor** methods

```python
class Order:
    def __init__(self, total_amount):
        self._total_amount = total_amount  # Private field
        self._paid = False
        self._shipped = False
    def pay(self):
        """Marks the order as paid."""
        self._paid = True
    def ship(self):
        """Ships the order, but only if it's paid."""
        if self._paid:
            self._shipped = True

import unittest

class TestOrder(unittest.TestCase):
    def test_order_shipment_fails_if_not_paid(self):
        order = Order(100)
        order.ship()  # Should not ship since it's not paid

        # No way to directly check if order was shipped!
        ... # ??

if __name__ == "__main__":
    unittest.main()
```

The state “paid” and “shipped” are internal to Order

Test whether shipping before payment fails (as it should)

Q. How do we check whether the test passed?

```python
class Order:
    ... # same as on the previous slide
    def get_status(self):  # Getter that provides meaningful state
        """Returns the status based on payment and shipping state."""
        if self._shipped:
            return "Shipped"
        elif self._paid:
            return "Paid"
        else:
            return "Pending Payment”

class TestOrder(unittest.TestCase):
    def test_order_shipment_fails_if_not_paid(self):
        order = Order(100)
        order.ship()  # Should not ship since it's not paid

        # Check that the order is not shipped before payment
        self.assertEqual(order.get_status(), ”Pending Payment")

if __name__ == "__main__":
    unittest.main()
```

Provides visibility into the status of the order

Use the accessor method as part of test oracle

Q. Why not create getters for “shipped” and “paid”?

Improve Observability through Accessor Methods

- **Oracle**: For each test case, determine whether the test passed successfully or not
- Sometimes, this requires observing the internal state/behavior of a component(s) that is, by default, not publicly accessible
- One way to improve observability is to augment the component with **accessor** methods
- Accessors expose details about a component to external entities
  - This conflicts with the information hiding principle!
  - **Trade-offs** between observability vs. changeability!
- Consider: Remove/hide testing accessors from the production system if possible

Design Principles for Testability

1. Separate business logic from infrastructure code
2. Improve controllability through dependency injection
3. Improve observability through accessor methods
4. **Reduce test complexity through separation of concerns**

Reduce Test Complexity through Separation of Concerns

- Components with multiple responsibilities are harder to test, require larger test suites, has more complex dependencies
- **Recall**: Single-responsibility principle (SRP)
  - Break large components into smaller ones, each with a **single, cohesive purpose**
- If there are multiple external dependencies (e.g., API, I/O, DB) within a single component, consider separating them into different components
- If a private function within a component is complex enough to deserve its own tests, consider extracting it into its own component

Testing offers clues about the quality of your design

All tests basically do the following three things. If any of them is difficult, can we re-designed to make it easier?

1. Set up a component to be tested (and its dependencies)
   - ❑ Can it be designed with fewer dependencies?
2. Invoke a method, after satisfying certain conditions
   - ❑ Are the conditions hard to satisfy? Can we improve controllability?
3. Assert that the method behaved as expected
   - ❑ What additional information do we need for the assertion? Can observability be improved?

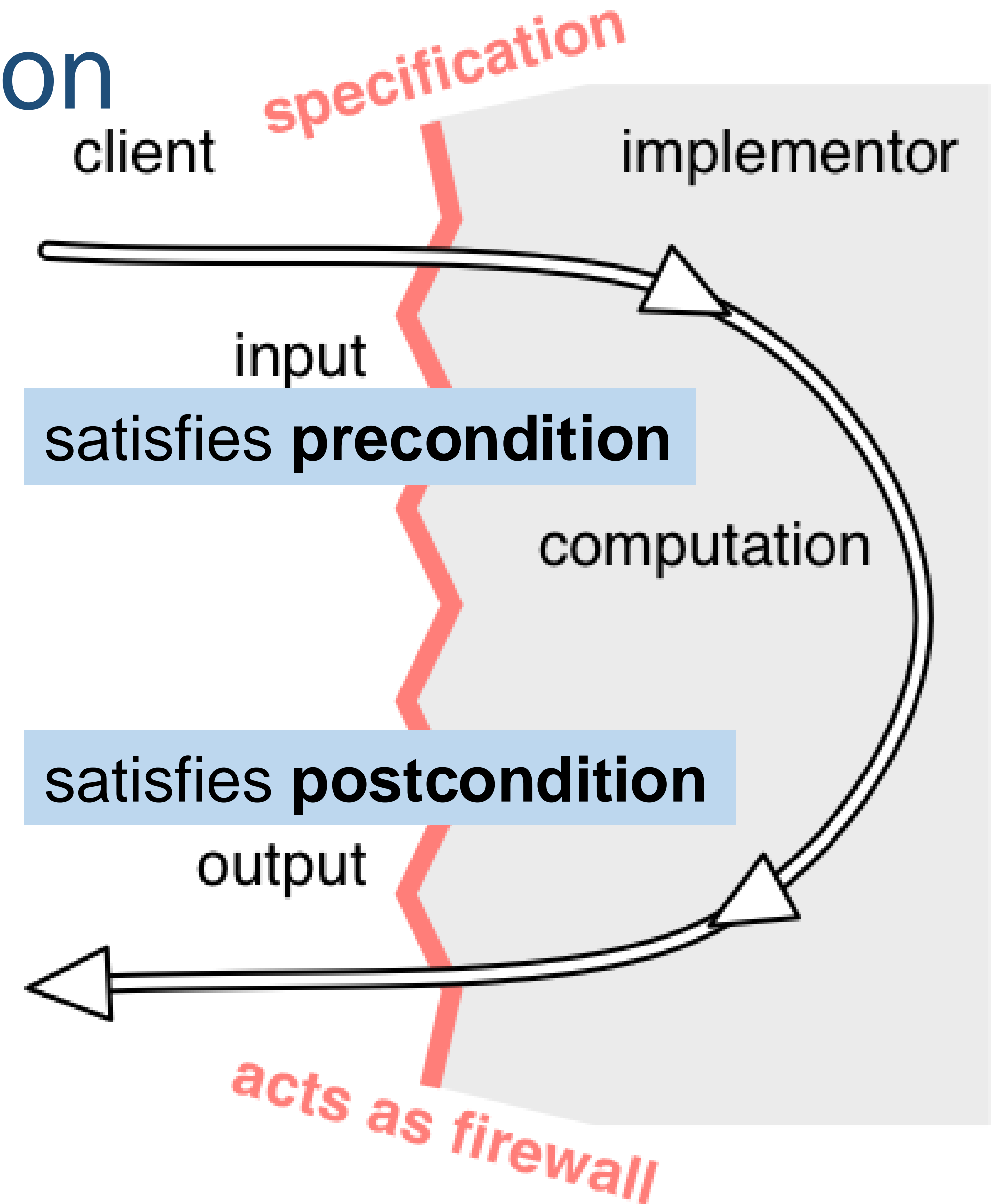
Design by Contract (DbC)

Design by Contract (DbC)

- Also called contract-based programming
- Each component is associated with a **contract** that describes its expected behavior given some assumption about its input
  - An interface specification (from last Wed's class) is one type of contract

Recall: Interface Specification

- **Pre-condition**
  - What the component **expects from the client**, expressed as a condition over the function input and/or component state
- **Post-condition**
  - What the component **promises to deliver**, as a condition over the function output and/or component state
- **Meaning:** Pre-condition holds ⇒ post-condition holds



Design by Contract (DbC)

- Also called contract-based programming
- Each component is associated with a **contract** that describes its expected behavior (i.e., **post-condition**) given some assumption (**pre-condition**) about its input
  - An interface specification (from last Wed's class) is one type of contract
- In addition to using contracts as a documentation, DbC also involves **checking** that a component and its client(s) fulfill their contract **during execution**
- Complementary to testing: Detect bugs in scenarios that are not covered by test cases

Checking Contracts using Assertions

- Assertion: A statement asserting that a condition must hold at a particular point in the execution
  - If the assertion fails, throw an error or an exception
  - Built-in support in many languages; can also simulate using conditional statements and exceptions (i.e., “if (…)”)

```
class BankAccount {
    private double balance;

    public void deposit(double amount) {
        assert amount > 0 : “Deposit amount must be positive”;
        balance += amount;
    }
}
```

Checking Contracts using Assertions

- Assertion: A statement asserting that a condition must hold at a particular point in the execution
- Assume a component method with a pre-condition and a post-condition
- Add an assertion at the *beginning* of the method to check that the *pre-condition* holds
  - If it fails, it indicates an invalid input from the client
- Add an assertion at the *end* of the method to check that the *post-condition* holds
  - If it fails, it indicates a bug in the method

```java
public class Basket {
  private double totalValue = 0;
  private Map<Product, Integer> basket = new HashMap<>();

  // requires: product is not null; quantity is greater than 0
  // effects: product is added to the basket
  public void add(Product product, int qtyToAdd) {
    // add the product
    // update the total value
    ...
  }

  // requires: product exists in the basket
  // effects: product is removed from the basket
  public void remove(Product product) {
    // remove the product from the basket
    // update the total value
    ...
  }
}
```

Pre-condition

Post-condition

```java
public class Basket {
  private double totalValue = 0;
  private Map<Product, Integer> basket = new HashMap<>();

  // requires: product is not null; quantity is greater than 0
  // effects: product is added to the basket
  public void add(Product product, int qtyToAdd) {
    // check the post-condition holds on the exit
    assert product != null : "Product cannot be null";
    assert qtyToAdd > 0 : "Cannot add 0 quantity";

    // add the product
    // update the total value
    ...

    // check the post-condition holds on the exit
    assert basket.containsKey(product) :
           "Failed to add the product to the basket ;
  }
}
```

Assert that the pre-condition holds

Assert that the post-condition holds

```java
public class Basket {
  private double totalValue = 0;
  private Map<Product, Integer> basket = new HashMap<>();

  // requires: product is not null; quantity is greater than 0
  // effects: product is added to the basket
  //          total value is greater than prev. total value
  public void add(Product product, int qtyToAdd) {
    // assert(total value is greater than prev. total value)
    assert ??
  }
}
```

How do I write this assertion?

Checking Post-conditions

- Post-conditions are sometimes expressed over the state of the component before and after a method
  - These are also called **pre-state** and **post-state**, respectively
- To assert such a post-condition, we must be able to refer to the pre-state at the end of the method
- **Solution**: Store the pre-state in an additional local variable
- These variables are also called **specification variables**
  - They do not alter the behavior of the method and adds no new information, but exist for the purpose of specification only

```java
public class Basket {
  private double totalValue = 0;
  private Map<Product, Integer> basket = new HashMap<>();

  // requires: product is not null; quantity is greater than 0
  // effects: product is added to the basket
  //          total value is greater than prev. total value
  public void add(Product product, int qtyToAdd) {
    // Specification variable: Prev. total value
    double oldTotalValue = totalValue;

    // assert(total value is greater than prev. total value)
    assert (totalValue >= oldTotalValue);
  }
}
```

Add a specification variable to store the pre-state

Check the post-condition over the pre- & post-states

Invariant

- A condition over the state of a component that must always hold throughout execution
- An important part of a contract, in addition to pre- & post-conditions
  - Describes what it means for the component to be in a valid state
  - Clients rely on the invariant being true
  - It is the responsibility of the component to ensure that it is never broken (otherwise, it may break the client's code!)
- Invariants are associated with a component, not with a particular method
  - An invariant holds in the pre- and post-state of **every** method

Examples: Invariant or not?

- A Set data structure does not contain duplicate elements
- The bank account balance is always positive
- The remove() method removes the largest element in the input list
- The list of items in a cart is sorted by the order in which they were added
- The account balance after deposit is greater than the previous balance
- The scheduling database does not contain multiple entries with an overlapping appointment time

Checking Invariants

- Each invariant should be checked in (1) the initial state of the component and (2) the post-state of each method
- **Initial state**
  - Ensure that the component state is properly set up during the initialization
- **Post-state of every method**
  - Ensure that the method preserves the invariant after potentially modifying the component state
- **Q. Do we also not need to check the invariant at the beginning of each method (i.e., the pre-state)?**

```java
public class Basket {
  private double totalValue;
  private Map<Product, Integer> basket = new HashMap<>();
  // invariant: totalValue is never negative

  // constructor
  public Basket() {
    // initialize the component state
    totalValue = 0;
    basket = new HashMap<>();
    // check that the component has been properly constructed
    // i.e., it satisfies the invariant
    totalValue >= 0;
  }
  // requires: product is not null; quantity is greater than 0
  // effects: product is added to the basket
  public void add(Product product, int qtyToAdd) {
    // add the product
    // update the total value
    ...
    // check that the method preserves the invariant
     totalValue >= 0;
  }
}
```

Invariant documentation

Check that invariant holds in initial state

Check that invariant holds in the post-state

```java
public class Basket {
  private double totalValue;
  private Map<Product, Integer> basket = new HashMap<>();

  // invariant: totalValue is never negative
  private boolean invariant() {
    return totalValue >= 0;
  }

  // constructor
  public Basket() {
    ...
    checkInvariant();
  }
  public void add(Product product, int qtyToAdd) {
    ...
    checkInvariant();
  }
  public void remove(Product product) {
    ...
    checkInvariant();
  }
}
```

Factor out the invariant

Reuse the invariant across every method

Design by Contract: Benefits

- **Improves reliability**: Ensures functions only run with valid inputs and produce expected results.
- **Easier debugging**: Errors are detected immediately when a contract is violated, rather than producing unexpected behavior later.
- **Defensive programming**: Prevents invalid states from propagating through the system.
- **Clear documentation**: The contract defines explicit rules for component inputs and outputs.

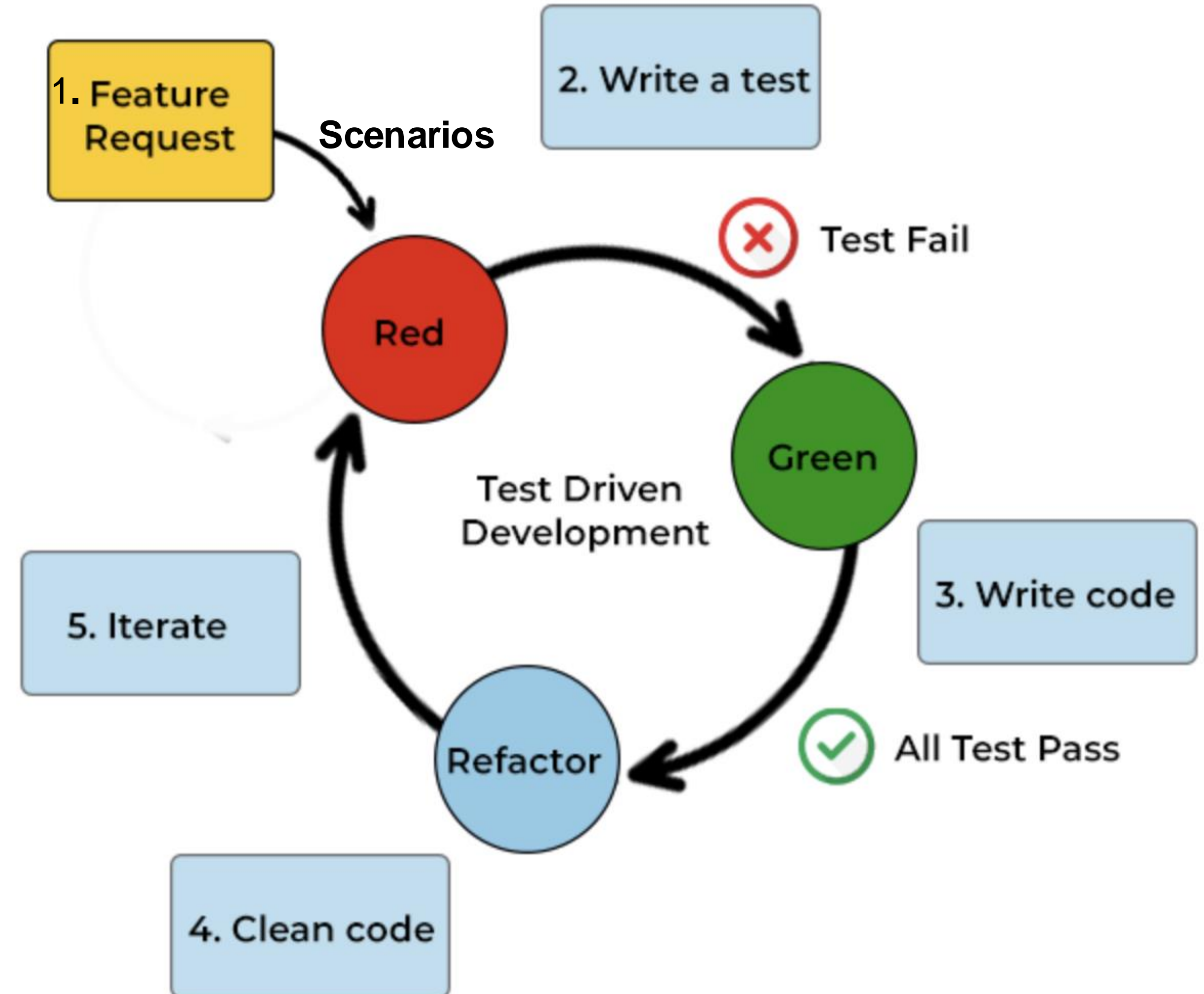
Design by Contract: Pitfalls

- Some contract conditions can be difficult to specify as an assertion
  - e.g., “The total value is equal to the sum of the prices of all items”
- Assertions can introduce performance overhead
  - Can disable them in the production-ready version
- Throwing an error effectively terminates the program execution; this is not always desirable behavior
  - An exception may be thrown instead, to be caught and handled by an external component
  - **Q. But exceptions also have downsides! What are some of these?**
- Contracts may be too restrictive or weak (recall the lecture on interface specification!)

Test-Driven Development (TDD)

Test-Driven Development

- An approach to developing software
- **Idea**: Write tests first *before* writing any code
  - Tests will fail; write minimal code to pass the tests
- **Claim**: Result in more tests written, less buggy code, and reduced debugging time



TDD Example: Factorial

- **Goal**: Write a program that computes a factorial
- **Q. What are possible inputs that we need to test for?**
  - Factorial of 0
  - Factorial of a positive number
  - Factorial of a negative number
  - Factorial of a non-integer

TDD from Requirements

- Creation of tests are driven by a feature request or a system requirement (e.g., achieve a quality attribute)
- Given a requirement, think of different use case scenarios ("variants")
- Each scenario variant becomes a test case to pass
- **Example**: Scheduling application
  - **Requirement**: Allow the user to search & book an appointment
  - **Scenarios**:
    - Log in, view available slots, select a time slot, enter patient info, confirm appointment
    - Log in, view available slots, no slots available
    - Log in, view available slots, select a time slot, enter patient info, cancel appointment
    - others...

TDD: Benefits & Pitfalls

- **Discussion: What are some benefits of TDD? Potential downsides?**

TDD: Benefits & Pitfalls

- By writing tests first, you are forced to think about requirements and different scenarios – this is always beneficial!
  - And you are more likely to end up with tests than if you did not apply TDD
- TDD is *code-centric*: The goal is to write code that passes all tests
  - Risks of neglecting high-level design considerations (e.g., changeability)
  - TDD does not exclude good design, but the focus on *passing tests* may put design as a secondary concern
- Following TDD does not automatically lead to high-quality design!
  - Still need to think carefully about component responsibilities, and assumptions about the client (i.e., contracts!), corner cases, and designing to be ready for changes

Summary

- Exit ticket!