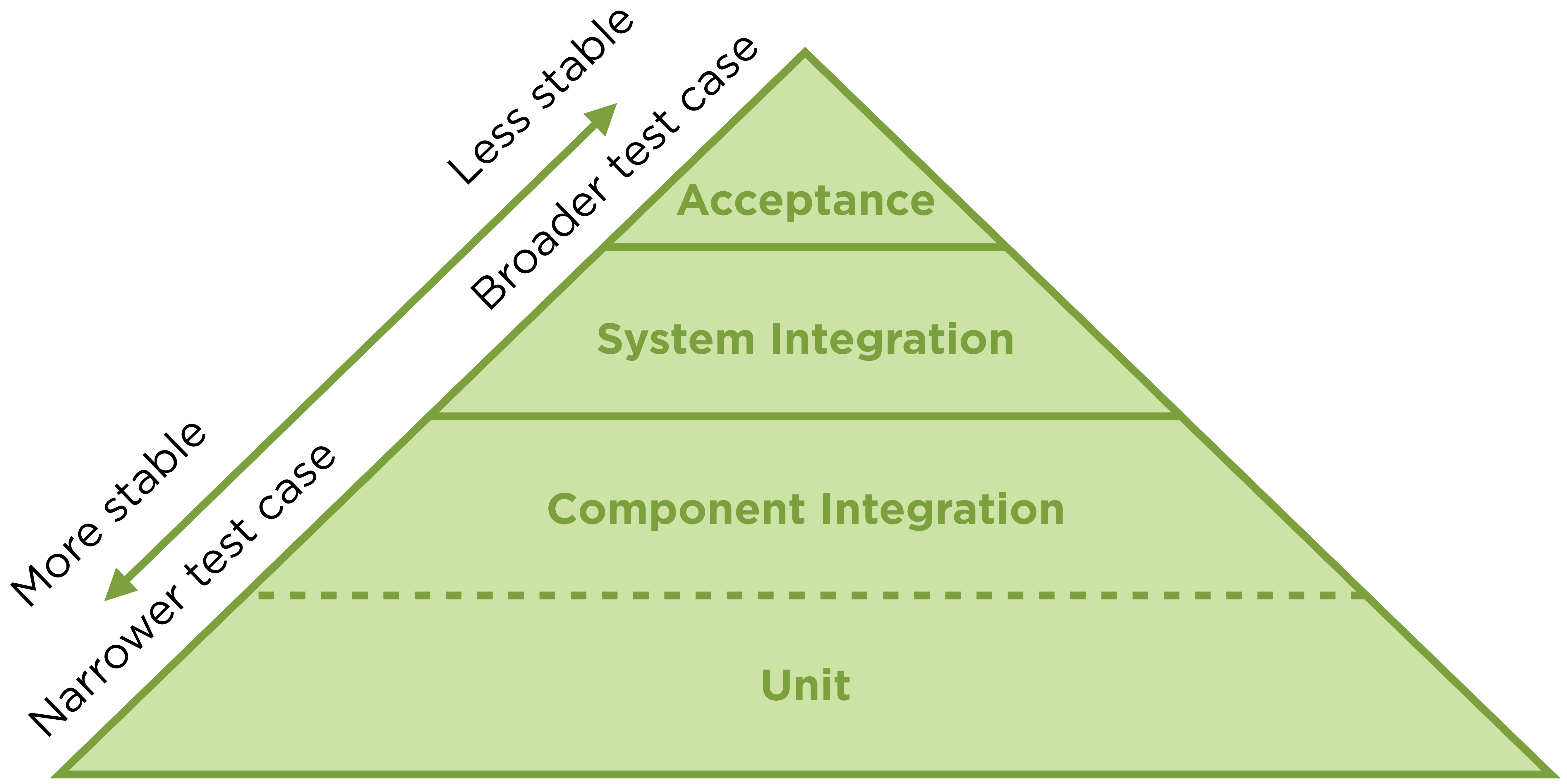
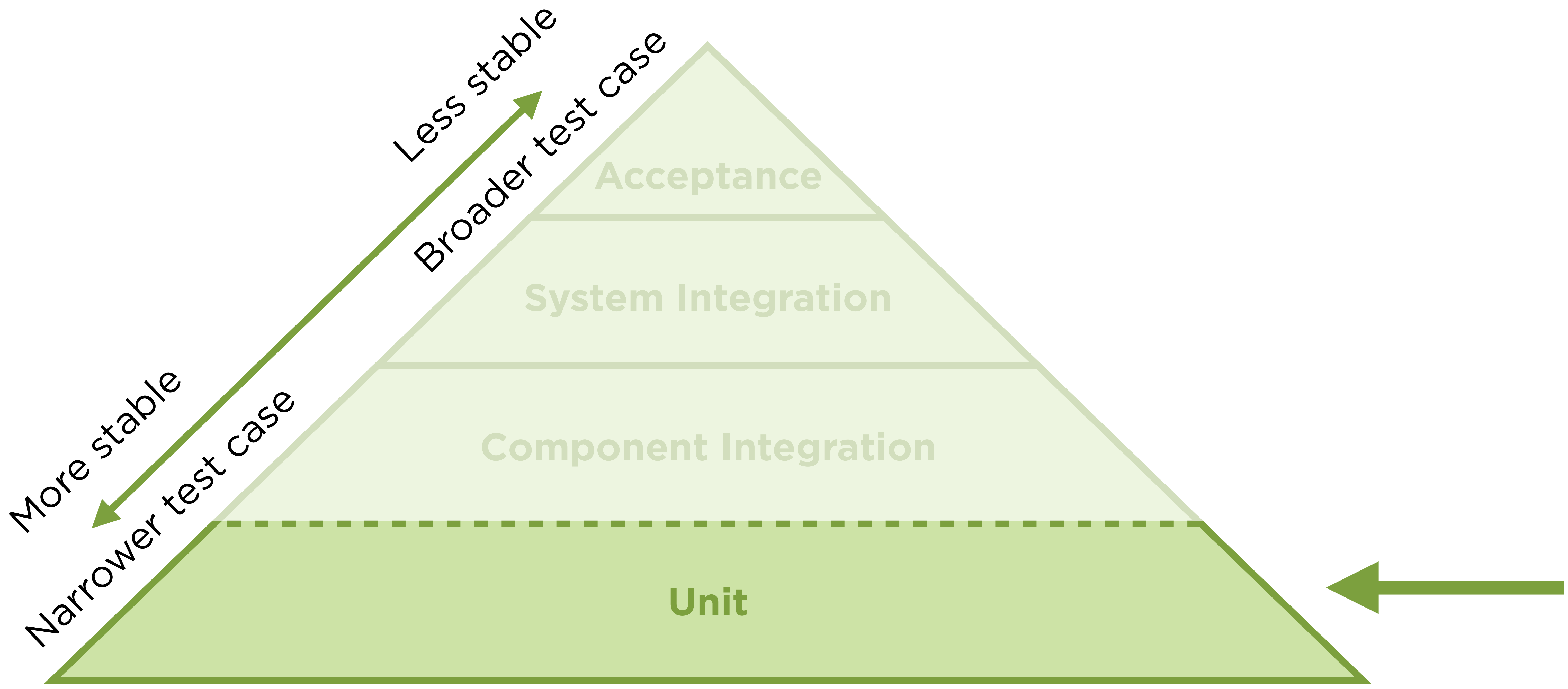
Effective Unit Testing in Spring

Billy Korando
SOFTWARE CONSULTANT - KEYHOLE SOFTWARE
@BillyKorando

What is Automated Testing?



What is Automated Testing?



Benefits of Writing Automated Tests

Verify correctness

Document behavior

Detect regression

Why Don’t We Write Tests?

Time consuming

Difficult to maintain

Lack of value

What Is a Unit Test?

```
public Order createOrder(List<Items> items, String cusId, String ccNum){
   validateItems(items);

   RestTemplate rest = new RestTemplate();
   Customer cust = rest.get(customerUrl + cusId);
   Payment payment = rest.get(paymentUrl + ccNum);

   Order order = new Order();
   order.setItems(items);
   order.setCustomer(customer);
   order.setPayment(payment);

   String sql =
   "INSERT INTO ORDER (ORDER_ID, CUST_ID, PAYMENT_ID) VALUES (?, ?, ?)”;

   jdbcTemplate = new JdbcTemplate(dataSource);
   jdbcTemplate.update(sql, new Object[]  { order.getOrderId(), cust.getId(),
   payment.getId() });

   return order;
}
```

What Is a Unit Test?

```
@Test
public void throwExceptionWhenCustomerNotFound(){

   OrderService service = ...

   try{
      service.createOrder(null, "BAD_CUS_ID", null);
      fail("An Exception should had been thrown");
   } catch(AServiceException e){
   }

}
```

Not a unit test

Reason: not ran in isolation

What Is a Unit Test?

```java
public class Order{
   String orderNumber;

   public void setOrderNumber(String orderNumber){
      this.orderNumber = orderNumber;
   }

   public String getOrderNumber(){
      return orderNumber;
   }
}
```

What Is a Unit Test?

```
@Test
public void testSetOrderNumber()

   Order order = new Order();
   order.setOrderNumber("1234");

   assertEquals("1234", order.getOrderNumber());

}
```

Not a unit test

Reason: not verifying business behavior

What Is a Unit Test?

```java
public Order createOrder(List<Items> items, String cusId, String ccNum){

   itemService.validateItems(items);
   Customer customer = customerService.findCustomer(cusId);
   Payment payment = paymentService.createPayment(ccNum);

   Order order = new Order();
   order.setItems(items);
   order.setCustomer(customer);
   order.setPayment(payment);

   orderDao.insertOrder(order);

   return order;
}
```

What Is a Unit Test?

```
@Test
public void testCreateOrder(){

   OrderService service = new OrderService(itemServiceDummy, customerMock,
   paymentMock, orderDaoMock);

   try{
      service.createOrder(null, “BAD_CUS_ID”, null);
      fail(“An Exception should had been thrown”);
   } catch(ServiceException e){
      assertThat(“Customer Id: BAD_CUS_ID not found”, e.getMessage());
   }

   Order order = service.createOrder(testItemList(), “1234”, “1234”);
   assertNotNull(order);
}
```

Not a unit test

Reason: verifying multiple scenarios

What Is a Unit Test?

```
@Test
public void testCreateOrder(){

   OrderService service = new OrderService(it[illegible]my, customerMock,
   paymentMock, orderDaoMock);

   try{
      service.createOrde[illegible]il);
      fail(“An Exception s[illegible]”);
   } catch(AServiceExceptio[illegible]
      assertThat(“Customer Id:[illegible]not found!”, e.getMessage());
   }
}
```

S.O.L.I.D. Principles

Single Responsibility

Open/Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

S.O.L.I.D. Principles

Cohesion Principles

- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

S.O.L.I.D. Principles

Dependency Abstraction Principles

- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

Single Responsibility

There should only be one reason for a class to change.

Single Responsibility

```
public class MainService {
  createOrder(){...
  }
  findCustomer(){...
  }
  deleteAccount(){...
  }
  updateAccount(){...
  }
  validateOrder(){...
  }
  update(){...
  }
  newCustomer(){...
  }
}
```

- **One service to rule them all**

- The methods have no theme. They cover domains from Order to Customer to Account.

Single Responsibility

```
public Order createOrder(List<Items> items, String cusId, String ccNum){
   validateItems(items);

   RestTemplate rest = new RestTemplate();
   Customer cust = rest.get(customerUrl + cusId);
   Payment payment = rest.get(customerUrl + ccNum);

   Order order = new Order();
   order.setItems(items);
   order.setCustomer(cust);
   order.setPayment(payment);

   String sql =
   "INSERT INTO ORDER (ORDER_ID, CUST_ID, PAYMENT_ID) VALUES (?, ?, ?)”;

   jdbcTemplate = new JdbcTemplate(dataSource);
   jdbcTemplate.update(sql, new Object[]  { order.getOrderId(), cust.getId(),
   payment.getId() });

   return order;
}
```

Single Responsibility

```java
public Order createOrder(List<Items> items, String cusId, String ccNum){

   itemService.validateItems(items);
   Customer customer = customerService.findCustomer(cusId);
   Payment payment = paymentService.createPayment(ccNum);

   Order order = new Order();
   order.setItems(items);
   order.setCustomer(customer);
   order.setPayment(payment);

   orderDao.insertOrder(order);

   return order;
}
```

Interface Segregation

Better to have many client specific interfaces than a single general purpose interface.

Interface Segregation

```
public interface MainDao {
  insertOrder();
  lookupOrder();
  deleteOrder();
  lookupCustomer();
  insertCustomer();
  deleteCustomer();
  insertPayment();
}
```

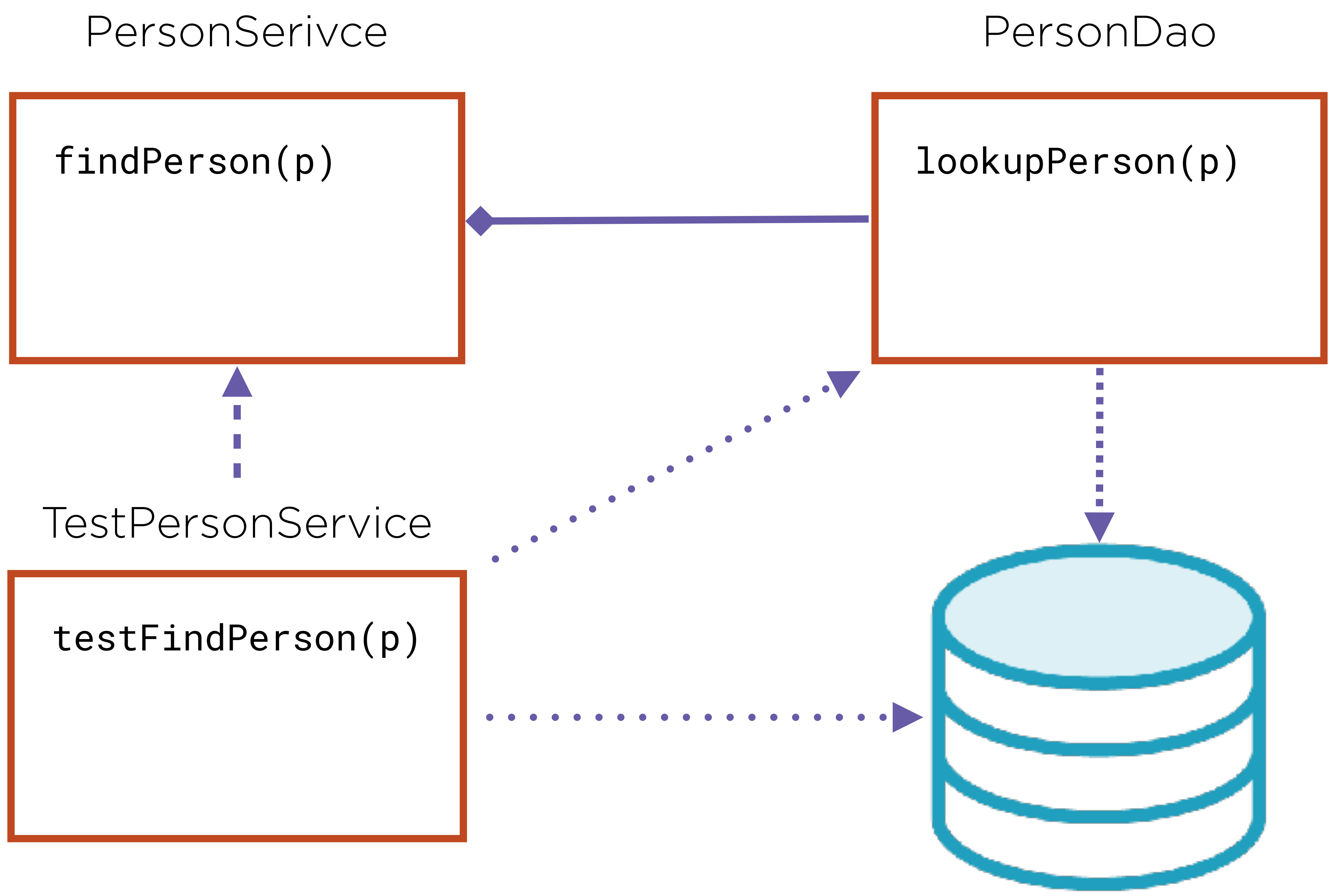
All of these methods will need to be implemented in a mock implementation.

Interface Segregation

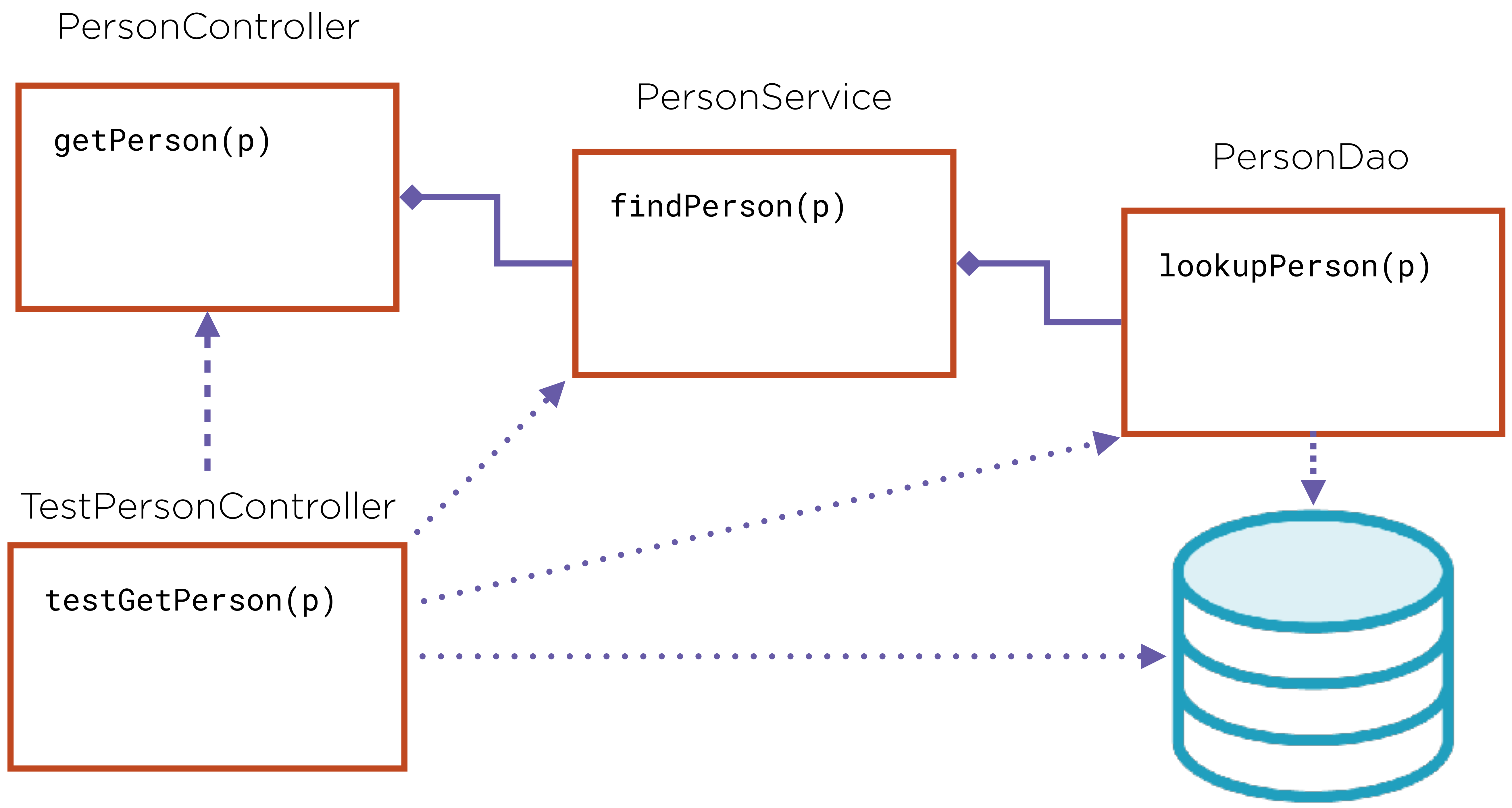
```
public interface OrderDao {
  insertOrder();
  lookupOrder();
  deleteOrder();
}



public interface
CustomerDao {
  lookupCustomer();
  insertCustomer();
  deleteCustomer();
}
```

- **Fewer methods means easier to mock.**

Dependency Abstraction Principles



Dependency Abstraction Principles



Dependency Abstraction Principles

- Open for Extension/Closed For Modification

  The behavior of a class can be extended. The extended behavior should not modify the code of the class.

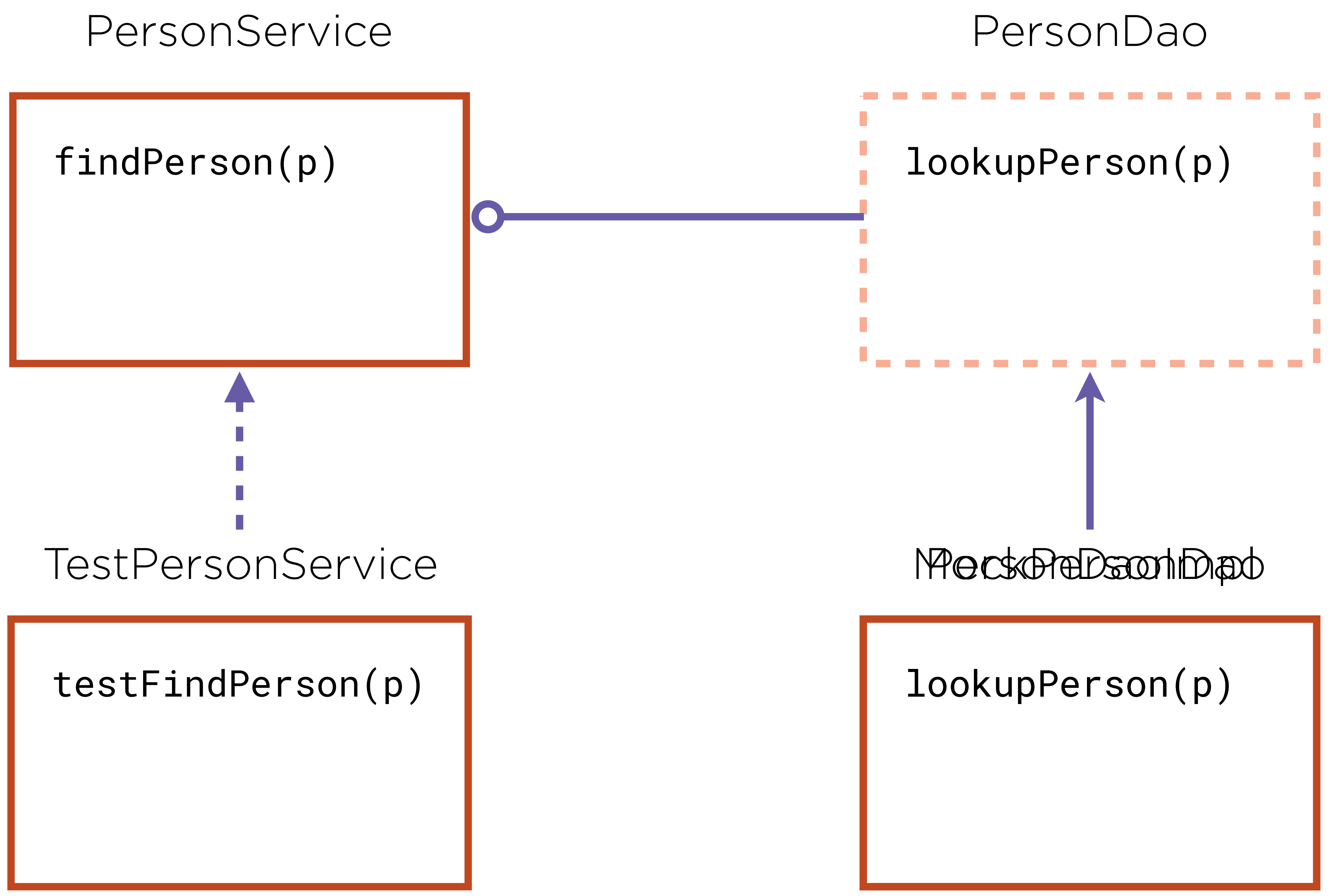
- Liskov Substitution

  The behavior of code should not change if a different subtype is used.

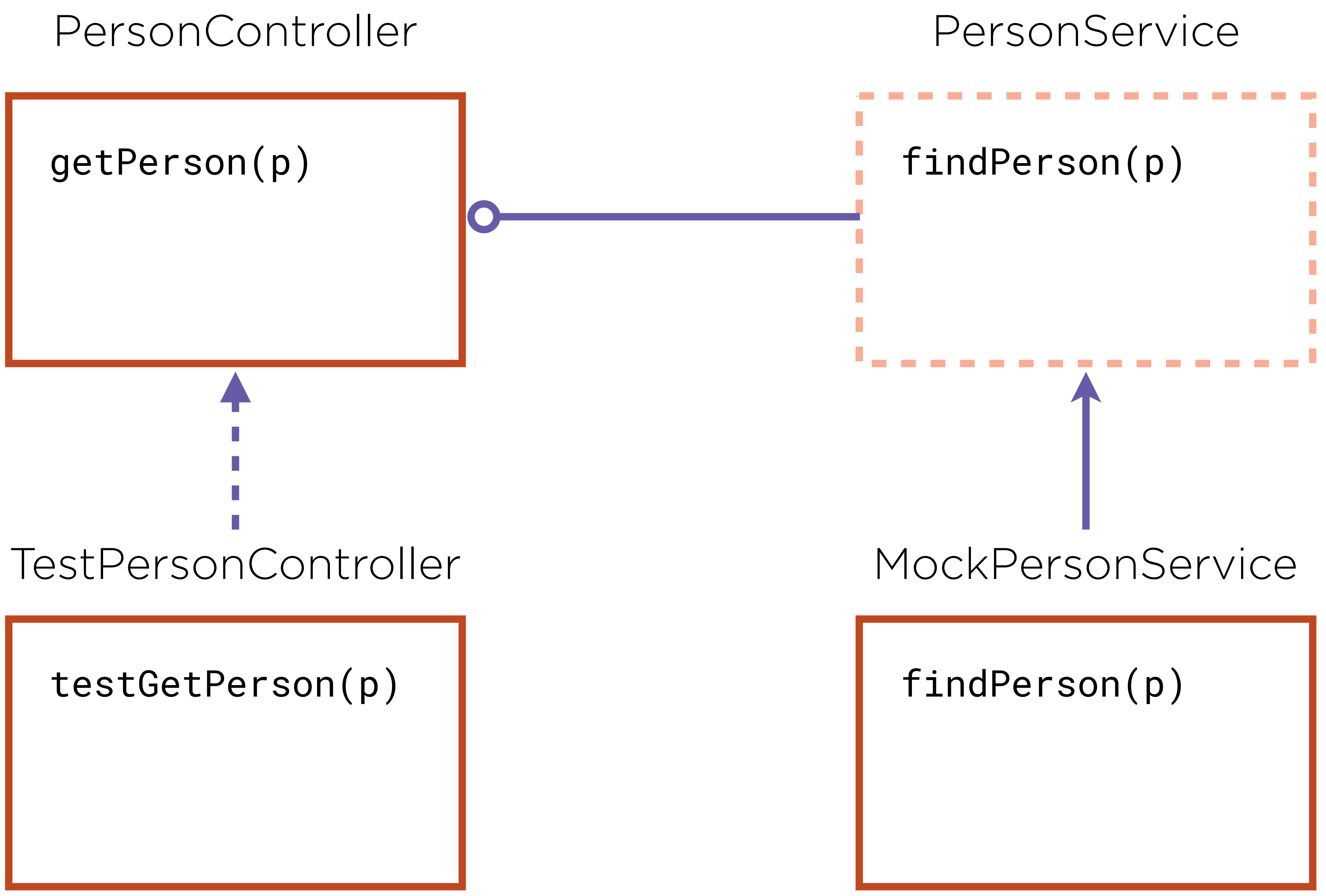
- Dependency Inversion

  High level classes should not depend on low level classes. Both should depend upon an abstraction.

Dependency Abstraction Principles



Dependency Abstraction Principles



Additional Design Considerations

```
public class PersonService{

~~@Autowired~~
private PersonDao dao;

   …
}
```

Do Not Use Field Injection

Using field injection means all tests depend on the Spring container.

“Field injection causes a unit test to break every time.”

Pivotal Team

```java
@Component
public class PersonService{

private PersonDao dao;

  public PersonService(PersonDao dao){
    this.dao = dao;
  }
}
```

Do Use Constructor Injection

By passing in our dependencies through a constructor our tests no longer require the Spring to work!

Note: As of Spring 4.3, if you only have a single constructor in a class Spring will auto-detect it for autowiring. Hint! Hint!

```
public class PersonService{

private PersonDao dao;

  @Autowired(required=false)
  public void setPersonDao(PersonDao dao){
    this.dao = dao;
  }
}
```

Do Use Setter Injection

Use when a dependency is optional

```
public class Name {
  private String firstName;
  private String lastName;
  private String middleName;
  …
  public Name(){…
  public Name(String firstName, String lastName, String
  middleName,…){…
}
```

Provide an Default Constructor

Helpful for when a test doesn’t care about the contents of an object.

```java
public class NameBuilder {
  private String firstName;
  private String lastName;
  …
  public NameBuilder firstName(String firstName){…
  public NameBuilder lastName(String lastName){…
  public Name build(){…
}
```

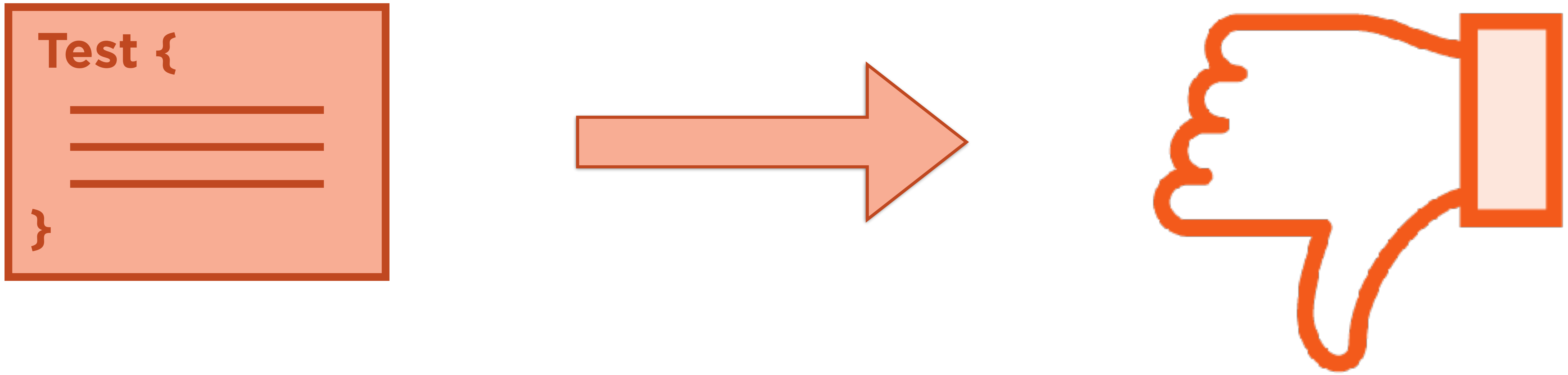
Use Builder Pattern

If some fields have constraints, like not being null, but other fields do not.

Note: Particularly helpful if a class has a lot of fields with the same type.

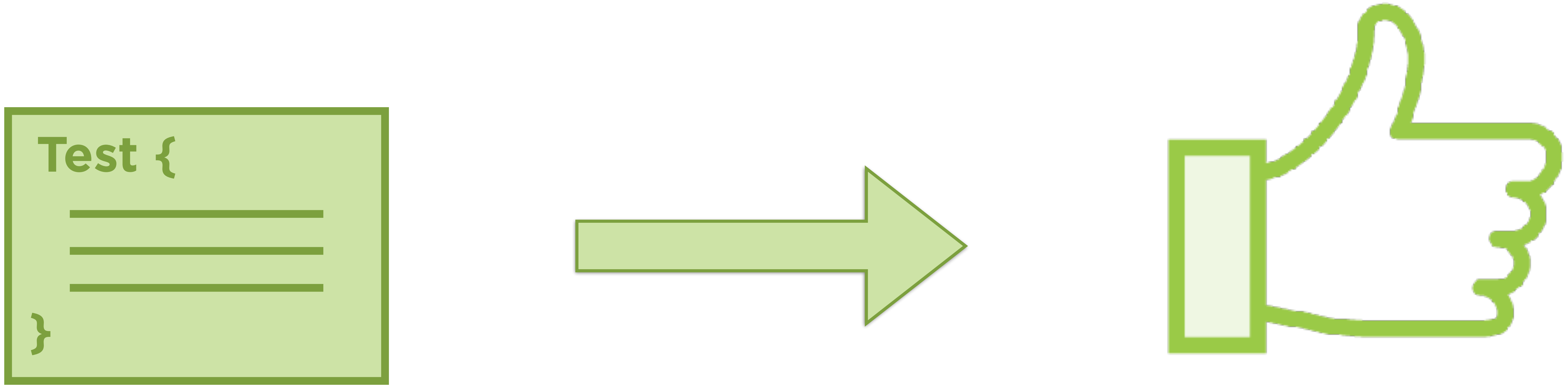
Test Driven Development

Write a Failing “Red” Test

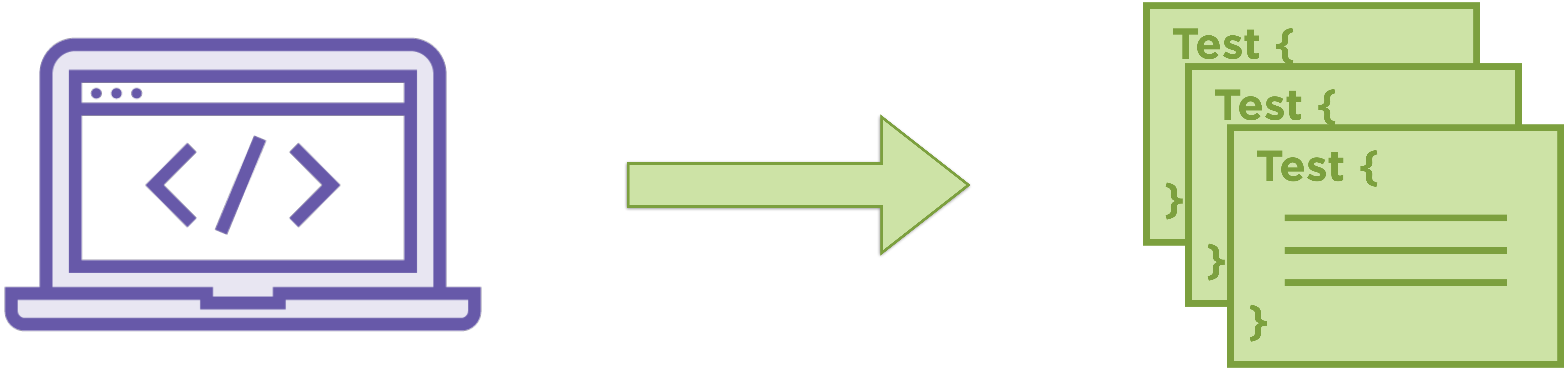


Implement the Feature

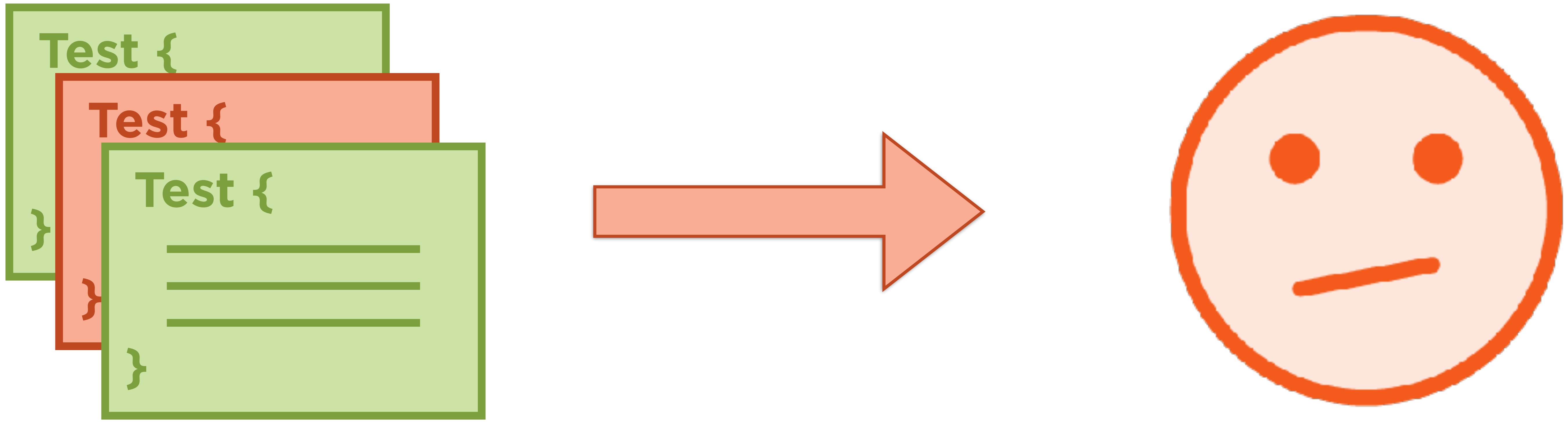
Run the Test Until It Passes (“Green”)



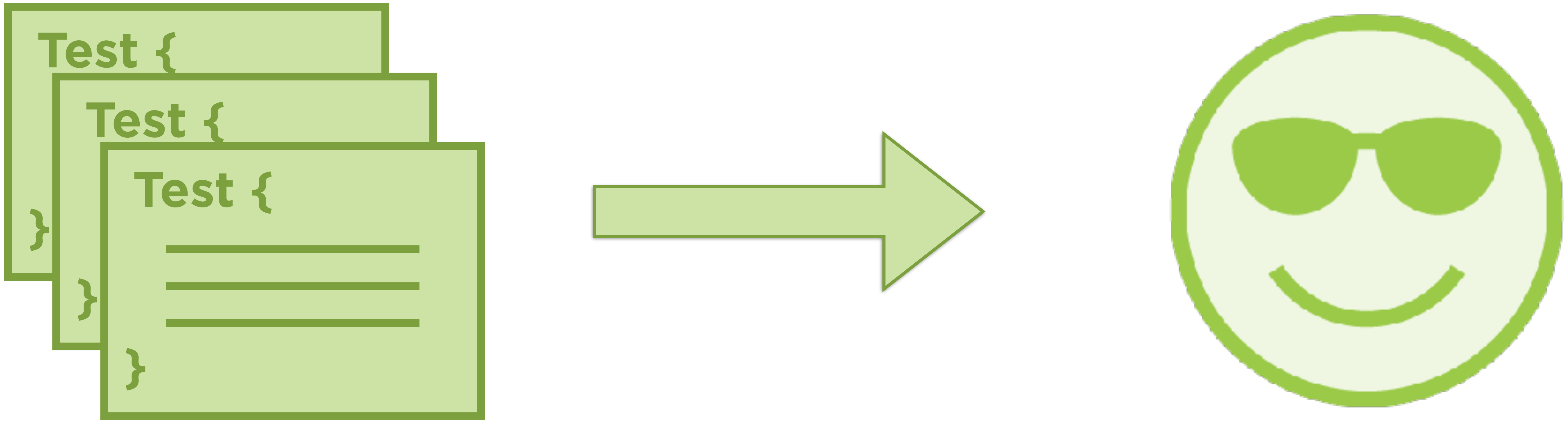
Add New Features and Test Cases

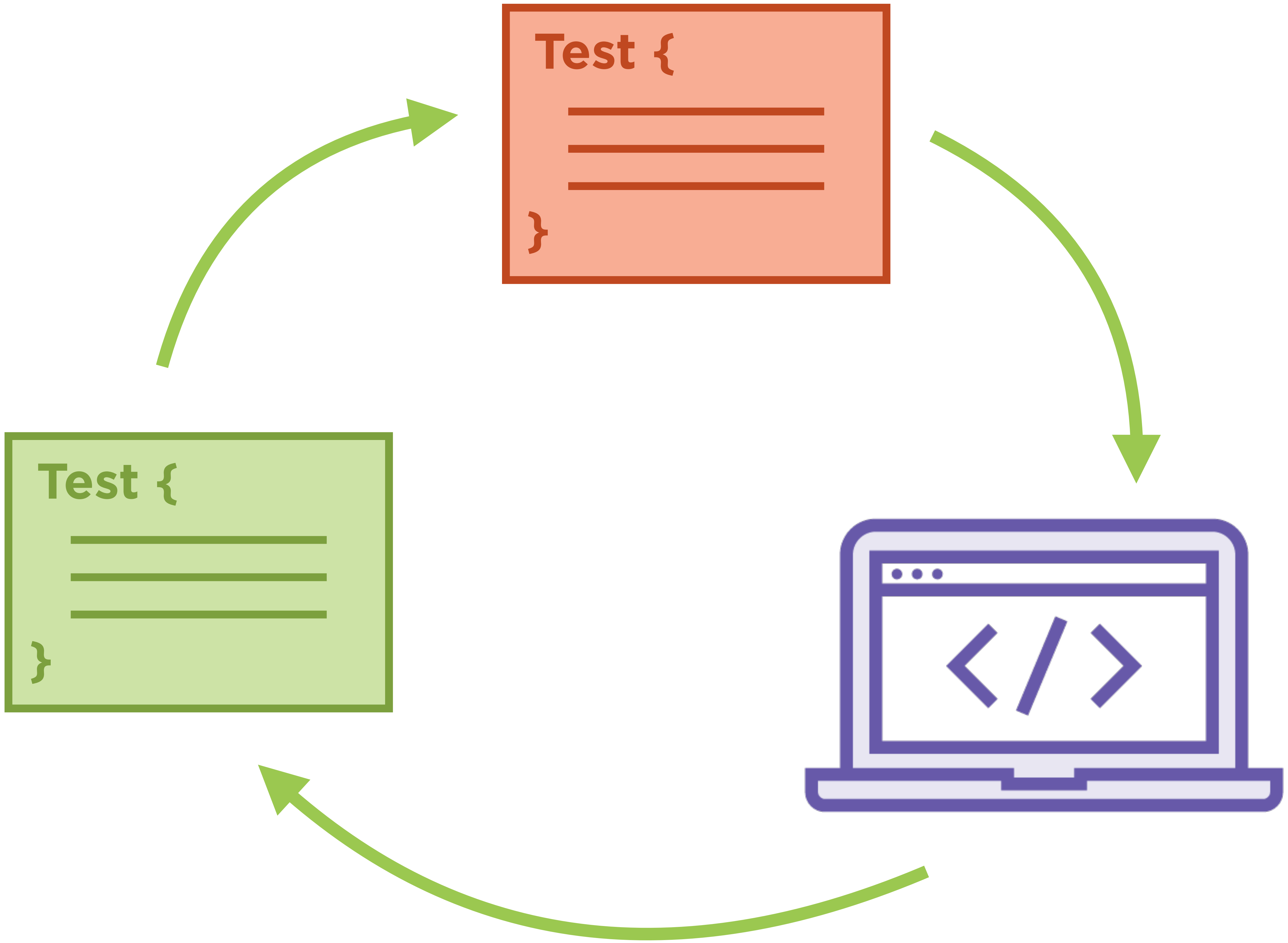


Detect Regressions



Refactor with Confidence




Test {
}
Test {
}