

Effective Automated Testing with Spring

WRITING TESTABLE CODE



Billy Korando

SENIOR SOFTWARE CONSULTANT - KEYHOLE SOFTWARE

@KorandoBilly

Prerequisites Assumptions

Decent understanding of Java

Decent understanding of Spring

Decent understanding of JUnit

Greenfield Project

Familiarity with Maven

Using Spring Tool Suite as IDE

Course Overview

Writing Testable Code

Leveraging Dependency Injection

Writing Tests to Document System Behavior

Mocking out Integration Points with Spring Tools

Writing Integration Tests

Module Agenda

Benefits of writing automated tests

Design principles to make testing easier

Additional design tips

Definitions

Automated Test

Tests that run as a part of your organization's Continuous Integration (CI) process.

Unit tests should run with every build.

Integration tests may run as part of a nightly build.

Dependency

Constructs a class utilizes to perform work.

Examples: Another class, a database, a web service

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

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.

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

Single Responsibility

◀ 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 customer = 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;
}
```

Single Responsibility

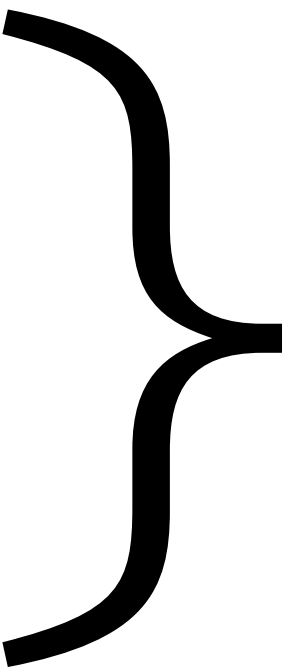
```
public Order createOrder(List<Items> items, String cusId, String ccNum){  
  
    itemService.validateItems(items);  
    Customer customer = customerService.findCustomer(cusId);  
    Payment payment = paymentService.createPayment(ccNum);  
  
    Order order = new Order();  
    order.setItems(items);  
    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.

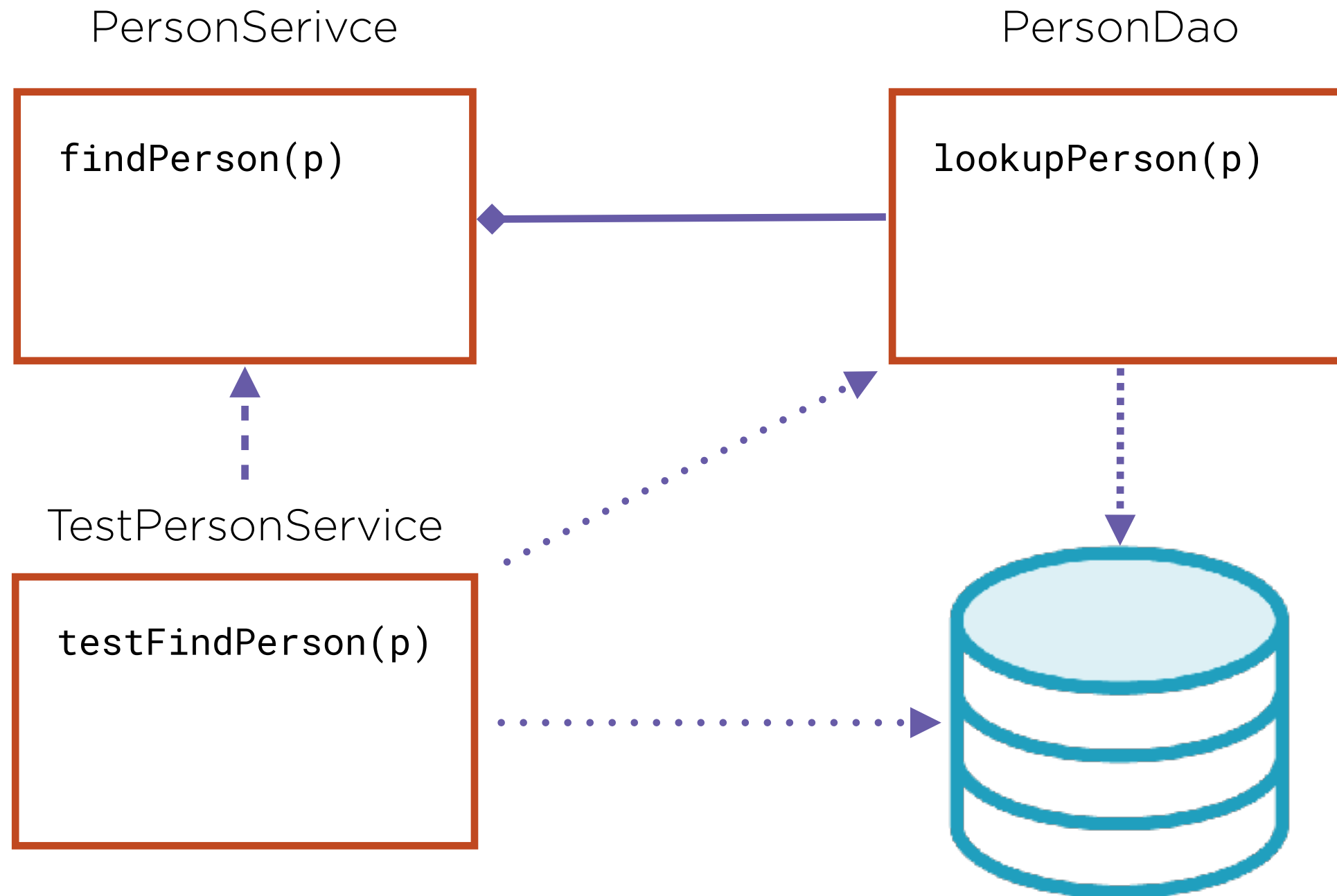
Interface Segregation

- ◀ Fewer methods means easier to mock.

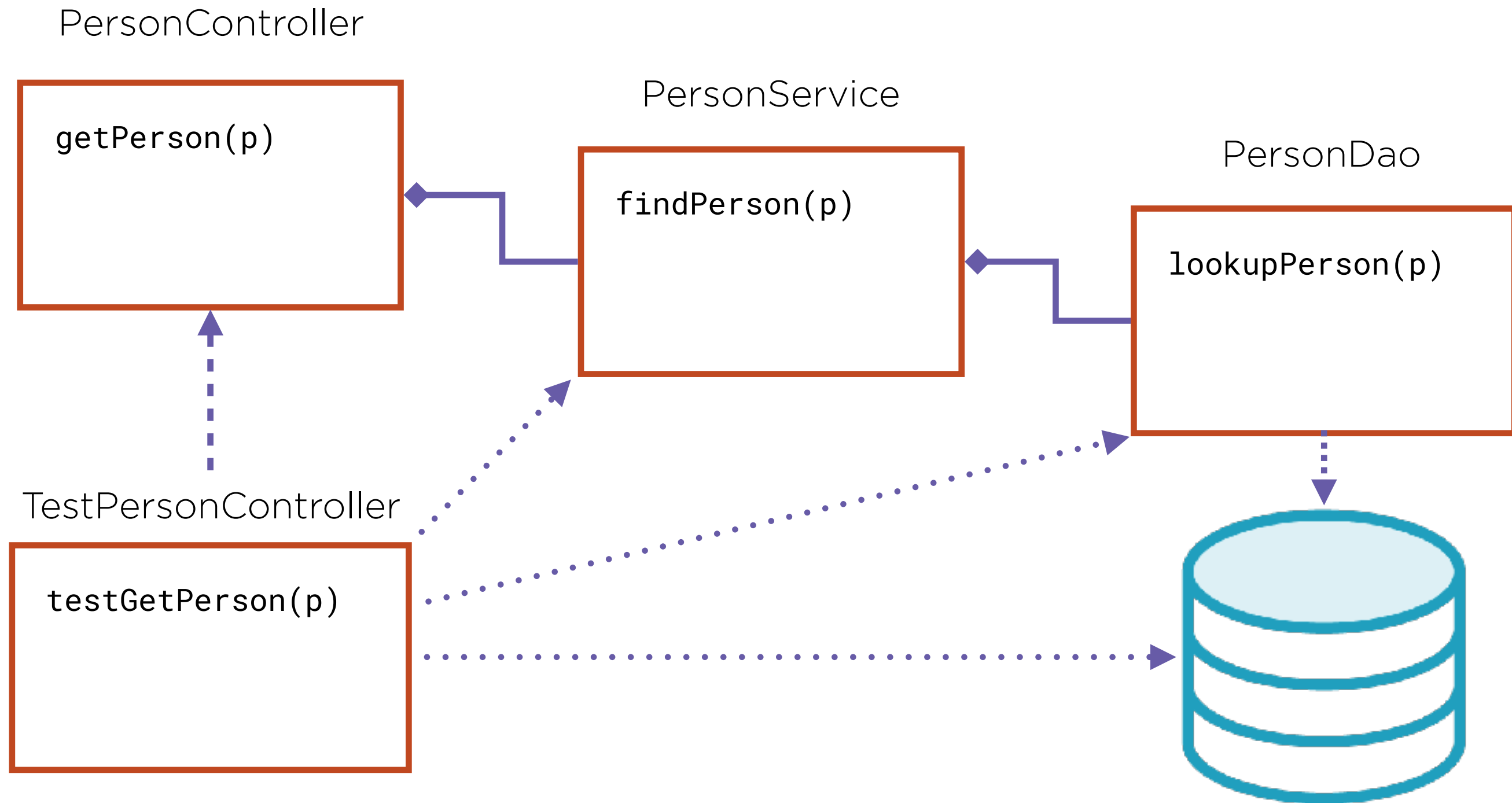
```
public interface OrderDao {  
    insertOrder();  
    lookupOrder();  
    deleteOrder();  
}
```

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

Dependency Abstraction Principles



Dependency Abstraction Principles

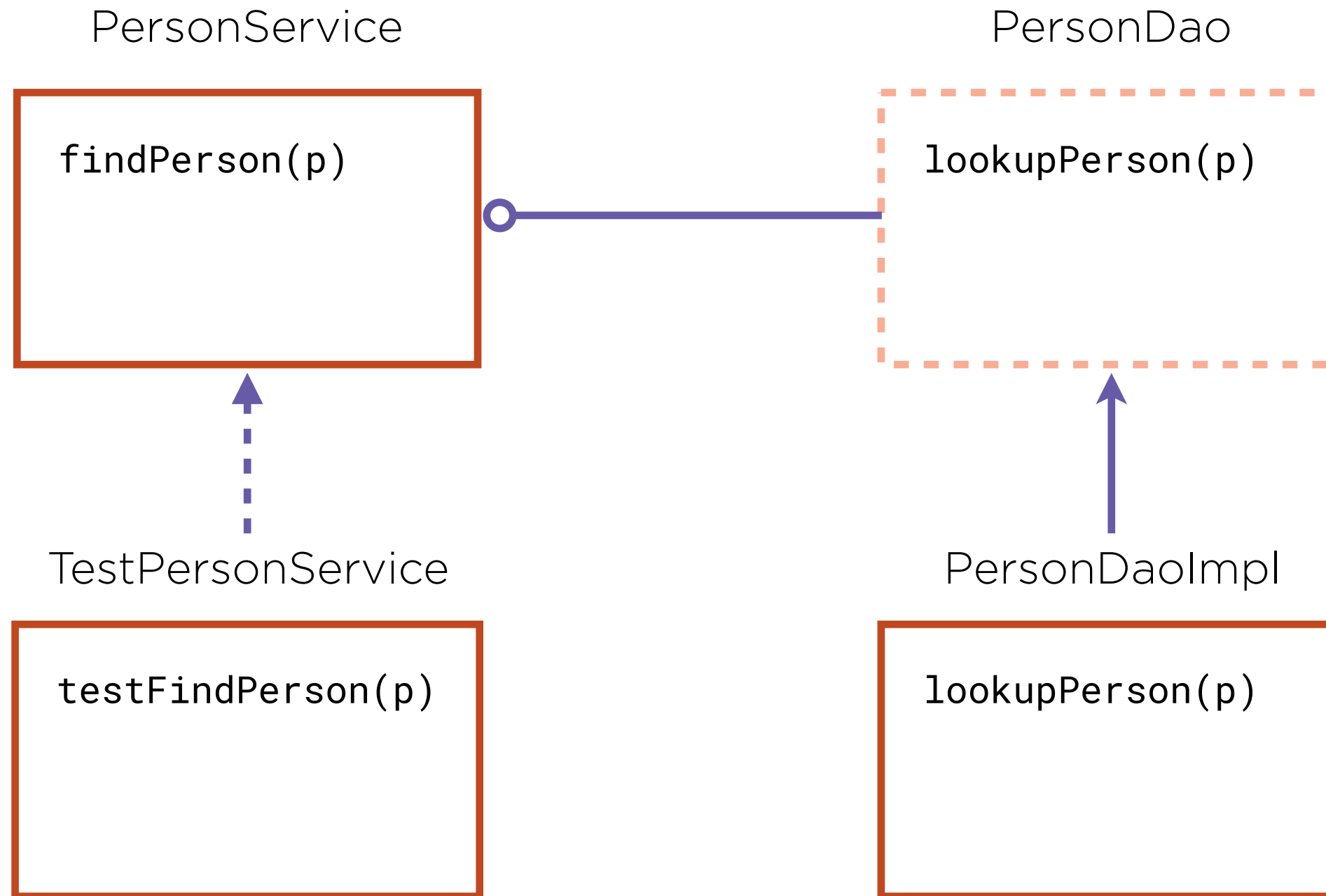


Dependency Abstraction Principles

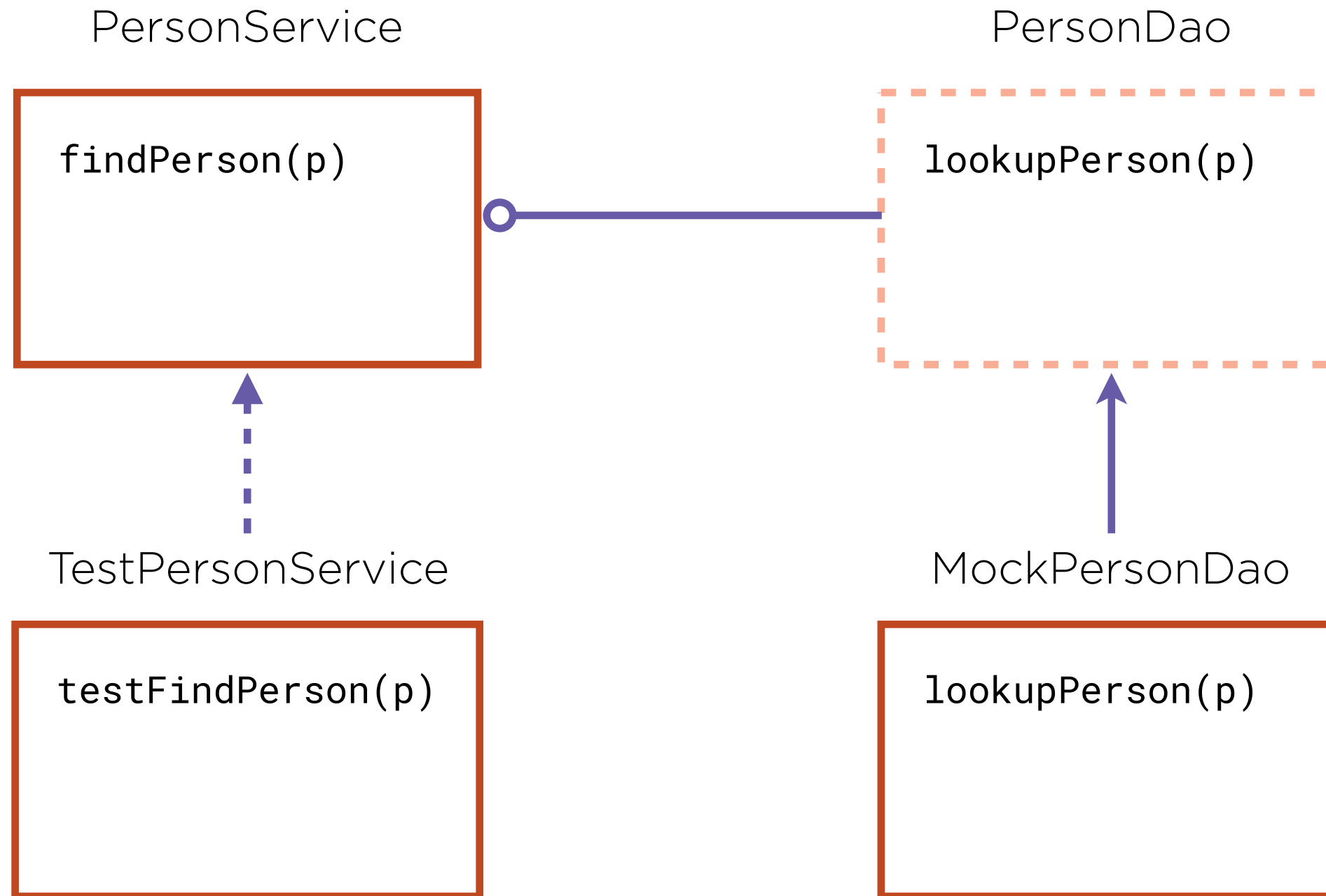


- Open for Extension/Closed For Modification
The behavior of a class can be extended. The extended behavior should not modify the code of the class.
- Liskov Substitution
The behavior of code should not change if a different subtype is used.
- Dependency Inversion
High level classes should not depend on low level classes. Both should depend upon an abstraction.

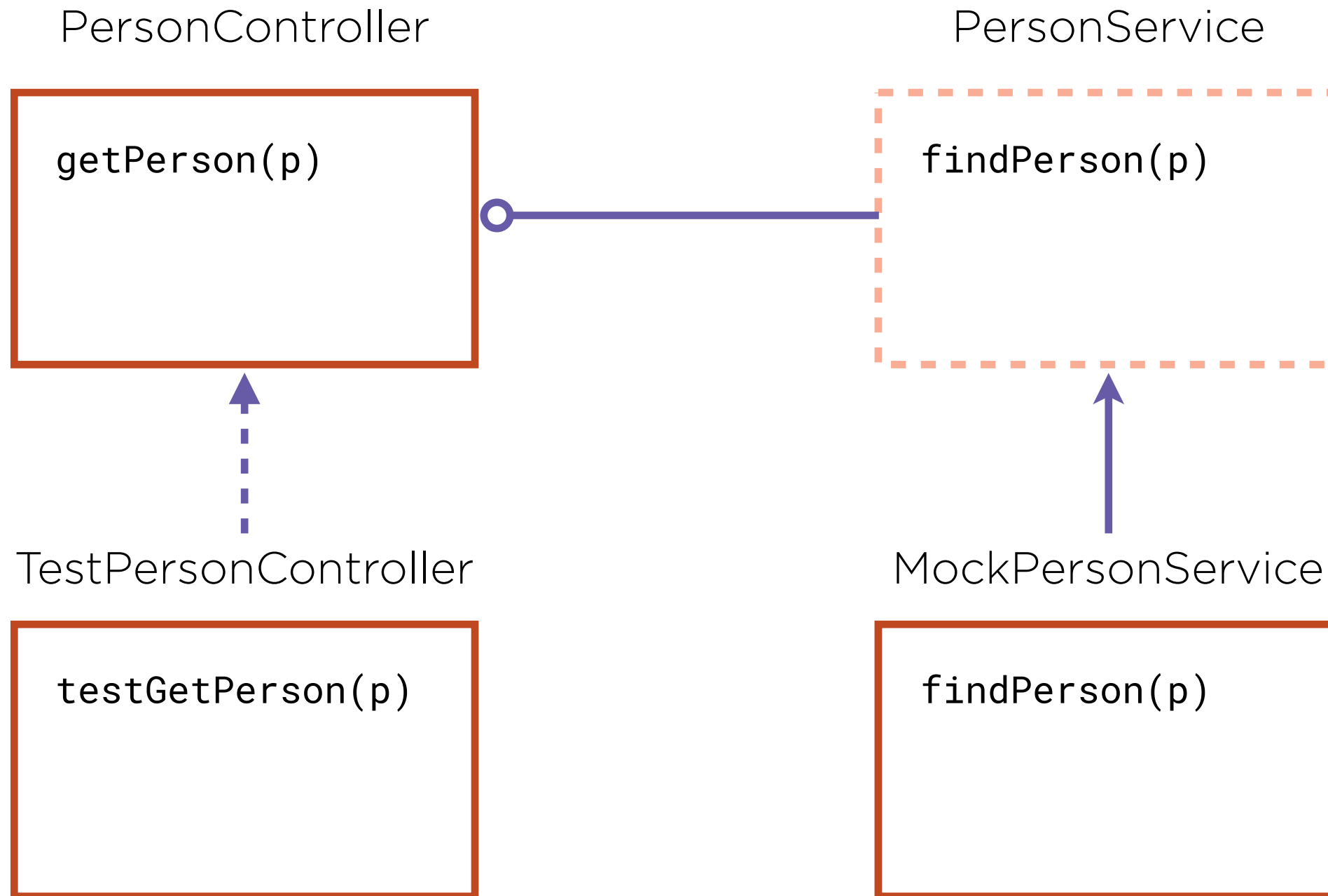
Dependency Abstraction Principles



Dependency Abstraction Principles



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

```
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  
    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.

```
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.


```
public class PersonService {  
    public void save(Person p){  
        if(PersonValidationUtils.validate(p)){  
            dao.save(p);  
        }  
    }  
}
```

Be Wary of the Use of Static

Mocking static methods will increase the complexity of tests, so avoid the usage of static where possible.

Summary

Benefits of automated testing

Impediments to automated testing

Learned how to write code to be testable