Testing Java in an Object-Oriented Way

In object-oriented software development, the system is developed as a collaborative collection of objects. Messages are the heart of the communication between these objects. Most Java-based software development projects employ unit testing, which mainly tests the system's behavior. Yet, we rarely test the object-oriented nature of the written program. Object-oriented tests consist of sequences of tests to assess the class hierarchy, inheritance graph, abstractness of classes, encapsulation, and many more object oriented features. This article introduces a list of object-oriented testing methodologies to test your Java programs, implementing a few important design patterns, using the JUnit testing framework.

Characteristics of Objects

An object is defined via its class, which determines every aspect of that object. Objects are individual instances of a class. Software objects that are modeled after real-world objects share two characteristics: they all have state and behavior. Mathematically an object could be represented as:

Object = f(I,S,B)

where I= Unique identity of the object, S=State of the object and B=Behavior of the object

For example, dogs have *state* (name, color, breed, hungry) and *behavior* (barking, fetching, wagging tail, rolling over). A software object maintains its state in one or more variables and use methods to implement its behavior. Each state of an object is initialized by a constructor and executes different behavior through the methods. This is the basic nature across all of the object-oriented programming languages, including Java.

From the above discussion, two different features of object-oriented systems can be identified:

- State-based communication system
- Contract-based communication system

Object-Oriented Software System: Communicating Extended Finite State Machine

In reality, object-oriented software systems, including Java-based systems, can be considered as a *Communicating Extended Finite State Machine* as described below:

CEFSM = f(S,S0,In,Br,Ou,V)

where

S= Finite set of states
S0 = Initial State
In = Finite set of inputs
Br = Business rules that participates in state transition
Ou = Finite set of outputs
V = Variable that represents finite Data

Let's define these concepts briefly:

Finite set of states

Each object-oriented software system steps through several internal states to satisfy expected functional behavior.

Input and Output Data

Each object-oriented software system deals with input and output data during the stepping through of its lifecycle.

Finite data and variables

Each system changes internally through different states. These state data are maintained in a set variable as a finite set of data.

State Transitioning Functions

Internally, every program performs a set of transformations, which includes but is not limited to decision checking, looping, involvement of data-structure, and so on.

Object-Oriented Software System: Contract based communication system

Most of the software system is actually an aggregation and composition of different systems, subsystems, and frameworks, but all of them are considered as a single system under test during the testing phase. This internal communication happens through the messaging between them. To ensure this smooth, error-free, end-to-end communication, it is recommended to maintain/honor a contract. It is highly recommended that object-oriented systems designed in Java or any other language should obey the following five principles to define and maintain the contract:

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP) (PDF)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP) (PDF)
- Interface Segregation Principle (ISP) (PDF)

The Design by Contract (DBC) principle-based software systems ensure that every component of a system respects the specified expectations through contracts. In object-oriented software systems, these contracts are maintained by either interfaces or abstract classes. These contracts ensure the expectations from both sides of the components (namely, the component itself and the client). Object-oriented frameworks are designed to be reused, and therefore enforce the implementation of the abstract classes and maintain the contracts through abstract classes. Object-oriented software frameworks consist of two different contract points, called *frozen spots* and *hot spots*. Frozen spots are the template-based, unalterable design foundation for the overall framework architecture, which define the framework itself (i.e. components of the framework, their intra- and inter-relationship, etc.). On the other hand, hot spots represent the hooks of the framework that actually allow for implementing the framework in a solution/implementation-specific way. Hotspots are basically generic in nature and facilitate being "plugged in" within the application. One good example of a hotspot within the context of the JUnit testing framework would be the TestCase that we must extend in our implementation-specific way.

12 Golden Rules to Maintain a Happy Object-Oriented System

There are 12 golden rules to maintain a happy object-oriented system:

Do	**Don't**
1. Focus on interfaces.	2. Be worried about the implementation.
3. Mention semantic contract using interfaces.	4. Let subtypes break this semantic contract of their parent types.
5. Service decoupling using interfaces.	6. Be coupled with specific concrete implementation.
7. Family extension using interfaces.	8. Break the family relationship using concrete implementation.
9. Establish a family rule by abstract classes.	10. Impose your own rule through concrete classes.

11. Let interfaces answer all "what" about the system.	12. Forget to mention "how" are you answering all "what" from interfaces.

Four Methods to Be Overridden: Better Way to Create Java Object

Java classes eventually are subclasses of java.lang.Object, and all Java classes inherit methods from Object. Although half of these methods are final and cannot be overridden, a few methods are overridable, and a few often must be overridden:

- clone
- toString
- equals
- hashCode

A Few Other Characteristics of Object

Following are a few unique features of objects in an object-oriented language like Java:

- Polymorphism
- Encapsulation
- Inheritance

A Bird's-Eye View of an Object-Oriented System

Object-oriented systems are therefore built on the solid foundation of classes, components, systems, subsystems, modules, and frameworks. Figure 1 briefly depicts an object-oriented system:

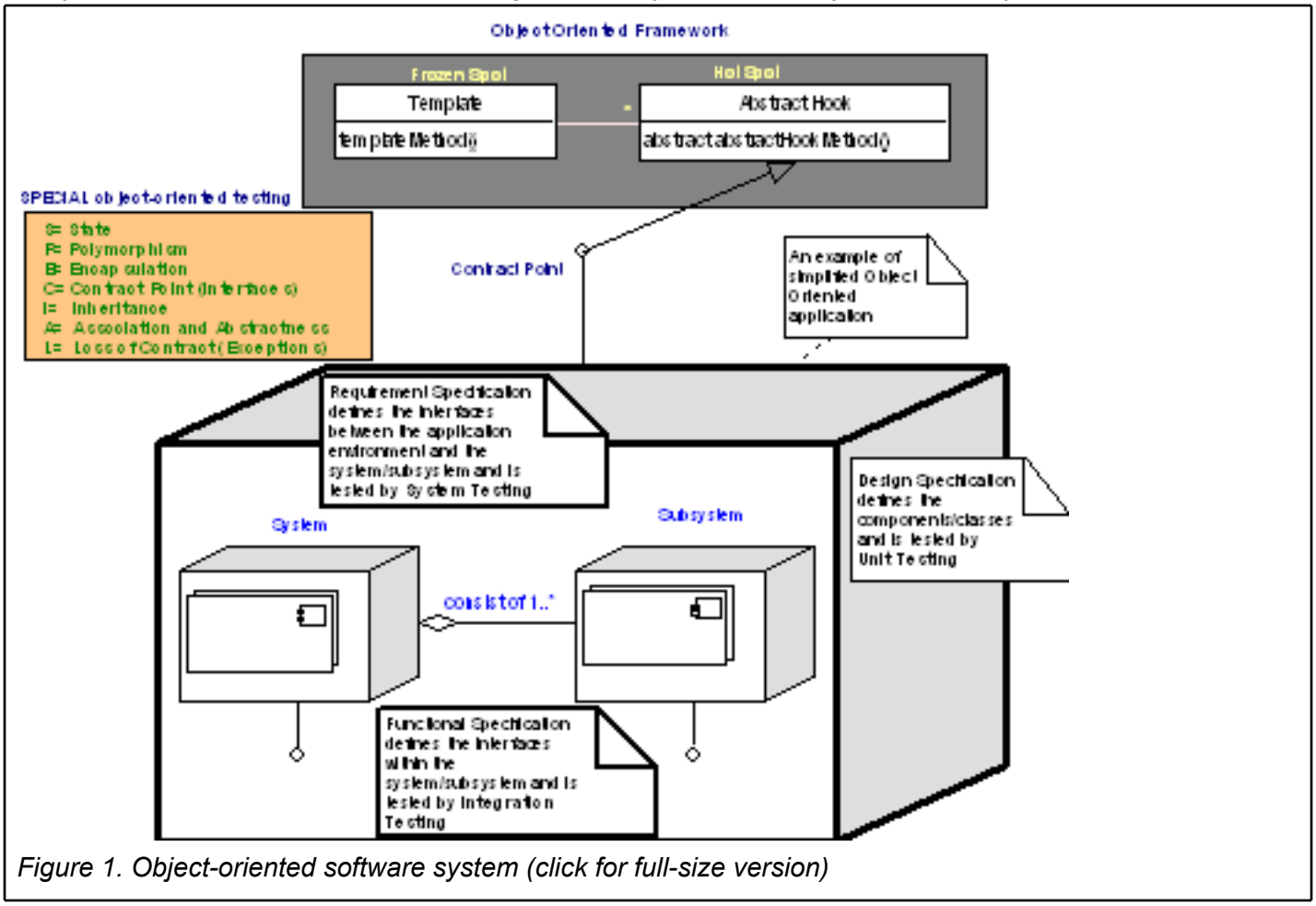

Figure 1. Object-oriented software system (click for full-size version)

Where

S represents the State of the object.
P represents Polymorphism of the object.
E represents Encapsulation of the data.
C represents Contract or Semantic Contract and is implemented through interfaces.
I represents the inheritance hierarchy of the object.
A1 represents the association relationship between different objects.
A2 represents the abstractness of the class. In other words, it checks the abstraction of the class.
L represents the loss of contract during the communication between the objects.

Every object-oriented systems follow these eight most important characteristics throughout its lifecycle. Therefore they are considered *SPECIAL*.

Object-Oriented Testing

From the above discussion it is clear that object-oriented testing should be performed on the basis of those eight SPECIAL features as defined in Figure 1. In OO testing, we are required to perform testing of interfaces, abstract classes, classes, frameworks, and systems. Figures 2 and 3 depicts the recommended type of testing that needs to be performed for each of these OO constructions apart from a regular behavioral test:

	S	P	E	C	I	A1	A2	L
Interfaces					☑			
Abstract Classes							☑	
System				☑				☑
Class	☑	☑	☑			☑		
Framework				☑			☑	

Figure 2. Object-oriented testing methodologies

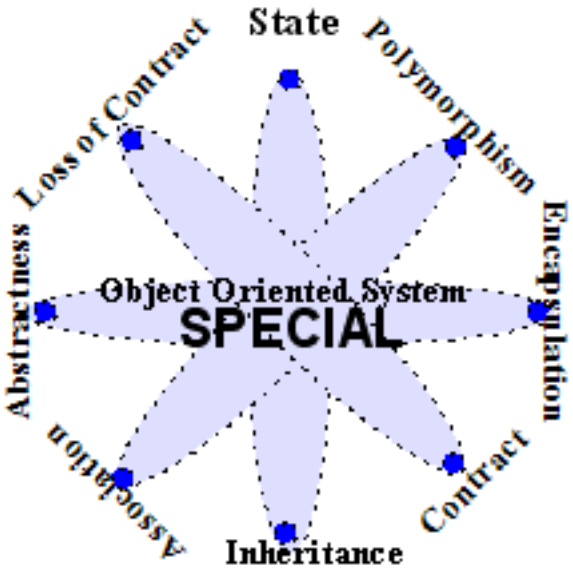

Figure 3. Relationship of OO testing methodologies

SPECIAL is an acronym that you should remember in every piece of object-oriented software to put under test. In the object-oriented sense, the contract needs to be fully testable through one reusable test suite that tests functional compliance (i.e., a module's compliance with some published functional specification, enforced through interfaces or abstract classes) with the interface that could be applied to each of the implementing

classes.
The next few sections will mainly concentrate on pattern-based object-oriented testing strategies and different effective object-oriented testing techniques.

Object-Oriented Testing Strategy: A Strategy-Pattern-Based Approach

Strategy is a bind-once design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. Figure 4 shows the Strategy pattern UML diagram:

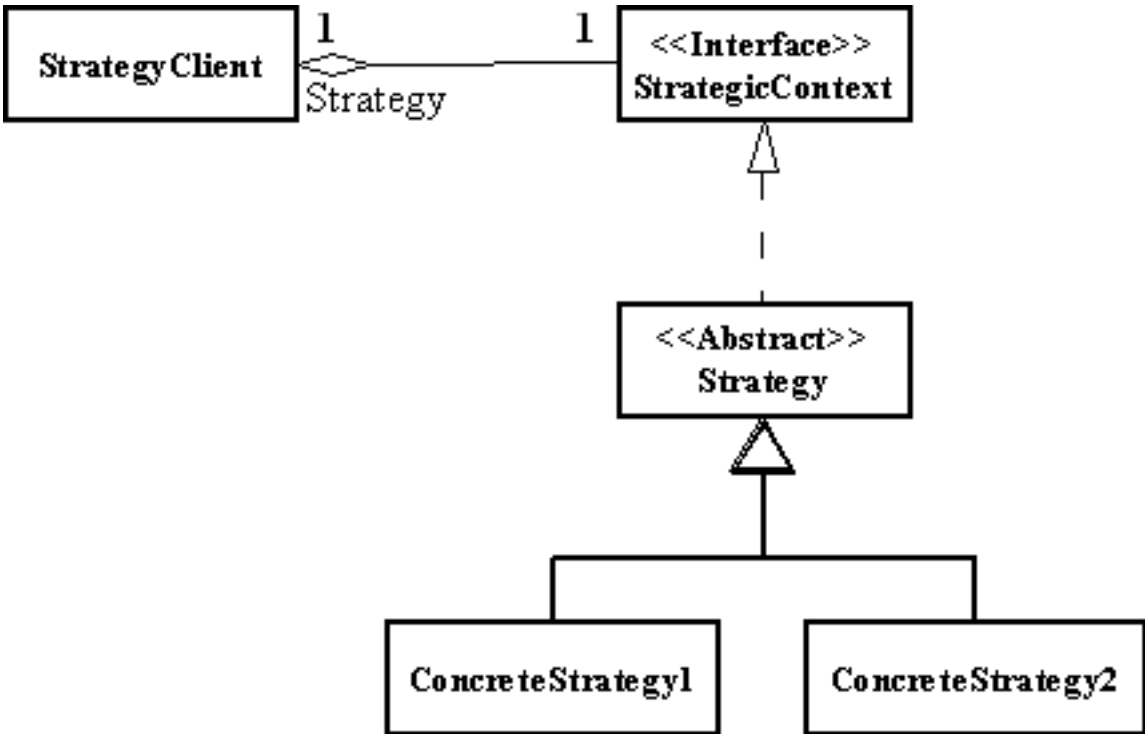

Figure 4. Strategy pattern

Here is a very simple example of an InterestCalculator interface with a few simple methods to calculate the actual interest value. We will construct an abstract strategy for this interface, and then look at the reusability options of the same interface implementations.

```
public interface InterestCalculator
 {
        public void acceptAmount (double dAmt);
        public double getInterestRate (float fInterest);
        public double getInterest();
 }
```

An Abstract Strategy

An abstract test is a test class, declared abstract and based on the Strategy pattern design shown in Figure 4. This abstract test class is the heart of the Strategy design patterns and it has following characteristics:

- Implements a StrategicContext interface.
- Should have a one-to-one relationship between each interface and abstract class.
- Provides a test case that is unintended to be overwritten.
- Provides a factory method to create a testable class instance.

```
public interface IStrategicContext
```

```
 {
        public void testMethod1();
        public void testMethod2();
 }
```

The form of a JUnit abstract test is something like this:

```
import junit.framework.*;
public abstract class AbstractStrategy
extends TestCase implements IStrategicContext
{
 private CandidateInterface candidateInterface;
 public AbstractTestSample(String name)
  {
super(name);
  }

 // Not intended to be overridden by
 // ConcreteStrategy classes
 public final void setUp() throws Exception
 {
        //Create the object to be tested.
        candidateInterface =
        createCandidateInterface();
        //assert that result is non-null
        assertNotNull(candidateInterface);
 }

 // This abstract method is acting as a
 // factory to create the candidate
 // interface
 public abstract CandidateInterface createCandidateInterface()
        throws Exception;

        // testMethod1 is final and
        // can't be overwritten by
        // ConcreteStrategy classes
        public final void testMethod1()
        {
        // Method1 body
        }

        // testMethod2 is final and
        // can't be overwritten by
        // ConcreteStrategy classes
        public final void testMethod2()
        {
        //Method2 body
        }

        //Declare n number of
```

```
        // methods to be tested
        // [...]
}
```

Based on the above recommendations, we can easily create an abstract strategy class to test InterestCalculator.

Concrete Strategy

Now we can think of different implementations of the InterestCalculator interface, like MonthlyInterestCalculator, YearlyInterestCalculator, etc. For each of these implementations, we'd need to construct a separate ConcreteStrategyClass extending AbstractStrategy. Therefore, the AbstractStrategy class is mostly reusable across the same hierarchy testing. However, each ConcreteStrategyClass will be responsible for deriving its own set of strategies required for particular testing and will generally vary widely depending on the requirements.

This abstract-strategy-based test design techniques has several benefits, which are listed below:

- The test itself is object-oriented.
- The test leads to a separation between the interface and its implementation.
- These test strategies could be created by the architects in their architecture proof-of-concept stages and, in later phases, developed and modified by developers. This provides an easy way to achieve a test-first architecture. One noteworthy point here is that test-first *design* is slightly different than test-first *development*. Test-first design is semantic contract testing, while test-first development is the implementation of the contract test.
- Reusable test classes are generated as reusable assets across the interface, which could be reused across the enterprise for similar type of projects. These test classes are good candidates for strategic reuse in the Enterprise Unified Process (EUP) phase .
- Virtually any object-oriented concepts could be tested using these abstract strategies.

SPECIAL Object-Oriented Testing Techniques

In this section we will discuss a few important object-oriented feature testing techniques in Java.

State Testing

In Java, the State design pattern is the best way to perform state-based testing arrangements. The diagram in Figure 5 represents the State design pattern, or the design strategy of a state machine:

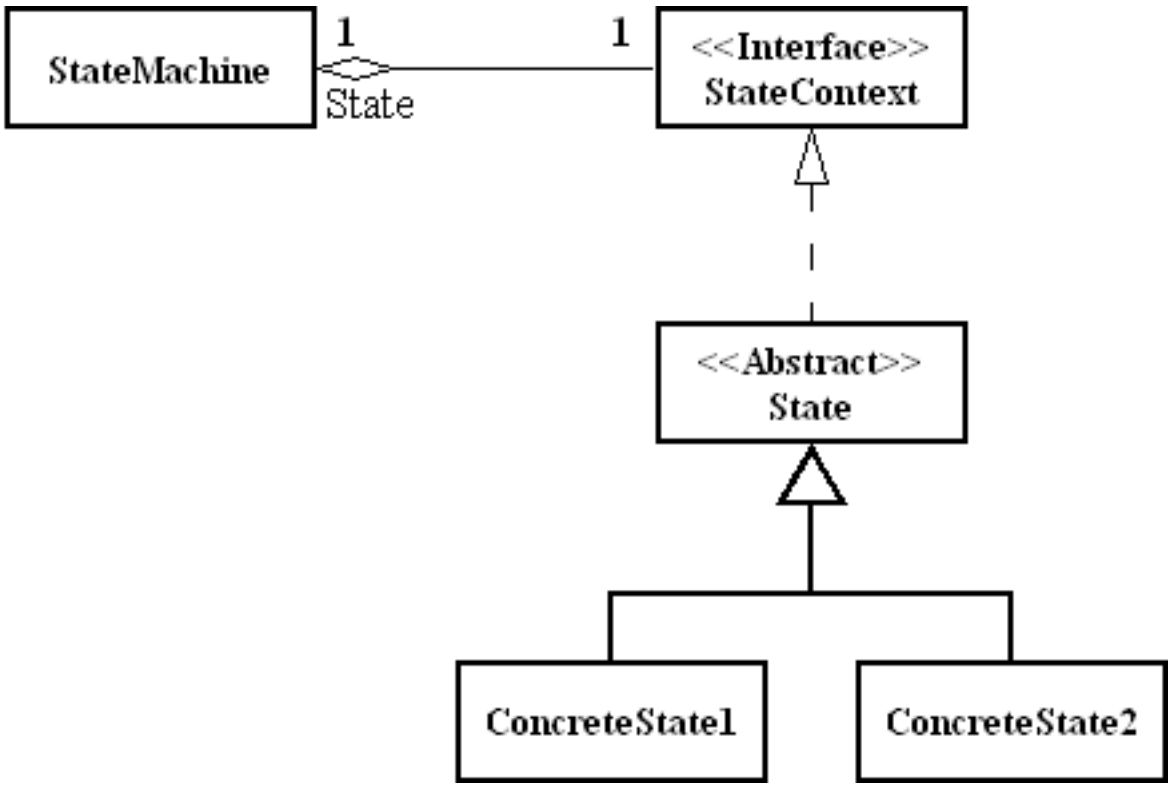

Figure 5. State design pattern

State needs to be abstracted around a state class. Each implementation is built as a separate ConcreteState class. The Strategy pattern described in the previous section is the best way to test Java-based object state. The Java API provides a class called java.util.Observable, which can be extended by any Java class to provide state-change notification.

Polymorphism Testing

Polymorphism is the property of a Java class to be in a different forms. Polymorphism may be single or multiple, and may be of a parametric type or ad-hoc (operator-overloading type). Somewhat differently, Luca Cardelli and Peter Wegner categorize polymorphism under two major categories: *ad hoc* and *universal* (see "On Understanding Types, Data Abstraction, and Polymorphism" [PDF]). These two categories may be further classified as:

- Ad hoc:
- Coercion
- Overloading
- Universal
- Parametric
- Inclusion

Among all these four different types of polymorphism, parametric is the best way to achieve polymorphism. Inclusion is also good. Java compilers extensively use the coercion and overloading types of polymorphism internally (e.g., implicit type conversion, operator overloading, etc.). However, developers mostly use the parametric and inclusion types of polymorphism to generate different subtypes and use them in a type-safe way.

In Java, polymorphism is manifested in three ways: method overloading, method overriding through inheritance, and method overriding through the Java interface. Polymorphism testing is required to ensure type safety.

In fact, Java polymorphism could me understood as a particular state of an object at a particular moment. Therefore, to test polymorphism of any Java class, we need to create a one-to-one concrete strategy to test the class for each concrete state.

Encapsulation Testing

Encapsulation refers to information hiding achieved via access modifiers in Java. It is sometimes necessary to test a private variable in Java, especially when it is related to state testing. However, in JUnit, there is no direct support of performing private method testing. But using JUnit add-ons such as junitx.util.PrivateAccessor, we can access private methods and test the values with regular JUnit tests.

Contract or Interface Testing

Interfaces and abstract classes are the only way to specify the contract between different Java objects. Interfaces are the heart of real object-oriented designs and therefore must be tested properly using the Abstract Strategy pattern discussed earlier.

Inheritance Testing

Inheritance can be used to implement specialization relationships. There are different types of inheritances possible, which is beyond the scope of this article. However, it is always recommended that you use the *specialization* and *extension* types of inheritance, as the resultant classes better support the aforementioned five object-oriented principles.

On the other hand, *specification*, *construction*, and *limitation* type inheritance are not recommended, as they restrict class reusability. Inheritance testing is really important for large organizations where objects are built in geographically distributed enterprise environments and may lose traceability. Figure 6 shows one of the recommended ways to derive inheritance hierarchy.

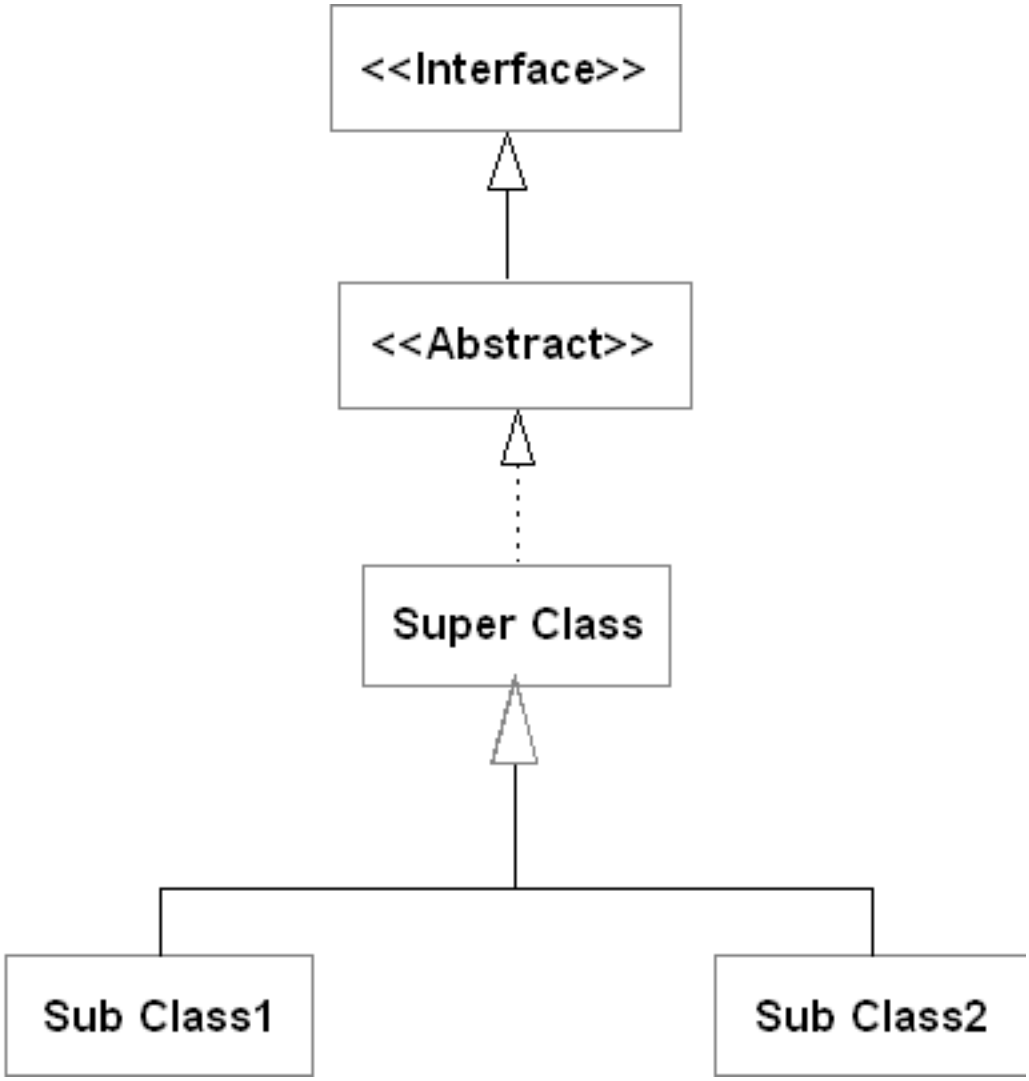

Figure 6. Class hierarchy and family extension

A factory method that actually helps the polymorphism is the one of the best ways to use the inheritance.

```
Public abstract class CreditCard
{
public abstract void createCard();
  //...
}
Public class MasterCard extends CreditCard
{
        public void createCard()
        {
        // do implementation
        }
        //...
}
Public class VisaCard extends CreditCard
{
        public void createCard()
        {
        // do implementation
        }
```

```
        //...
}
Public class CreateCardPrintDept
{
        public void getCard(CreditCard creditCard )
        throw Exception
        {
        createCard(new VisaCard());
        // Check null objects etc.
        //...
        createCard(new MasterCard());
        // Check null objects etc.
        //..
        }
        public static void printCard
        (CreditCard creditCard)
        {
                creditCard.createCard();
        }
}
```

This example also points out that using factory type object-creation techniques results in type-safe object creation. In other words, these subtypes satisfy the "principle of substitutability," allowing objects to be used interchangeably, which is an important objective of using inheritance. Inheritance is easily testable through the previously discussed AbstractStrategy classes and using instanceof methods like:

```
if (obj instanceof class1)
  ...
else if (obj instanceof class2)
        ...
else if (obj instanceof class3)
```

Association Testing

Association could be of three different types:

- Association (e.g., createAccount (SavingsAccount))
- Aggregation (e.g., private Account account)
- Composition (e.g., private Account savings; Account = new Account())

For association testing, follow these tips:

- Create the object associations.
- Simulate a particular state on that dependent object.
- Test that state using strategy.

You may use the JUnit assert*XXX* method to compare these values as regular JUnit tests.

Abstract Class Testing

This is the same as interface testing.

Loss-of-Contract Testing

Many large-scale object-oriented applications are developed and maintained by many people working across locations. Most of these object-oriented systems, APIs, frameworks, etc. are intended to be reusable. However, erroneous implementations of these methods can cause a semantic breaking of contracts and make it impossible for others to subtype them.

Therefore, another very important type of testing is to ensure that broken contracts are well-tested. This type of testing ensures that in case of a broken contract, system behaviors are correct and graceful decision logic is maintained. During loss-of-contract scenarios, the system should raise appropriate exceptions. State machines are very useful to test these exceptional situations; mock objects are another candidate to test exceptions.

Enterprise Development Lifecycle: Where Does OO Testing Fit?

So far we have discussed different kinds of object-oriented testing techniques, which are very important from any object-oriented system's point of view. However, another important part of these testing strategies is to understand and identify an appropriate lifecycle phase for each of these types of testing. The primary difference between object-oriented testing and normal procedural testing is that object-oriented testing is better performed by state and strategy patterns, which need to be very carefully designed by the architects. Otherwise, it will be trapped under the common symptoms of testing objects in a procedural way. There are different processes and methodologies available on the market. However, EUP is the only process that extends enterprise and RUP is the development oriented process. The following table provides the phases where OO testing might add significant value:

Phase	**Process or methodology**	**What do we test?**
Enterprise architecture	EUP	Framework
Analysis and design	RUP	Interfaces, abstract classes and system
Implementation	RUP	Concrete implementation
Test	RUP	Actual testing

Summary

The following table briefly summarizes OO testing concepts:

What (feature of the OO concept)	**Why (do need to test it)**	**How (do you achieve that)**	**Which (stage of the process)**	**Who (typically responsible for)**
State	Determine the correct state	State pattern	Development	Architect, developer
Polymorphism	To confirm the type safety	State pattern	Architecture design, development	Architect, developer
Encapsulation	Correct data	JUnit Add-on	Development	Developer
Contract/Interfaces	Correct contract/ interface specification	Strategy pattern	Architecture design, development	Architect, developer
Inheritance	Proper hierarchy/ subtype	Strategy pattern	Architecture design, development	Architect, developer

Association	Correct relationship	State and Strategy patterns	Development	Developer
Abstraction	Correct behavior	Strategy	Architecture design, development	Architect, developer
Loss of contract	Correct behavior even in the case of exception or lost-a-contract scenario	State, mock object	Development	Architect, developer

Note: All of these state- and strategy-pattern-based testing frameworks should be designed by the architects, with the help of the developers. However, in the actual development stage, developers will be responsible to ensure correct usage of them.

Conclusion

There are different schools of thought in the market. Moreover, in this consulting world, people argue against the usage of OO testing and brand it as a useless waste of time. In reality, it's really very important to perform object-oriented testing, if your software is written with the intention of reusability from either system or framework points of view. It is very important to test open source and third-party software and, OO testing is the only way to test semantic contracts. In any organization, creating an object-oriented testing framework can reduce the cost of testing an application by a significant order of magnitude, because it lets you reuse both design and code.

References

Standards

- RUP
- EUP

Bibliographies

- *Testing Object-Oriented Systems: Models, Patterns, and Tools*, by Robert V. Binder (The Addison-Wesley Object Technology Series)
- *A Practical Guide to Testing Object-Oriented Software*, by John D. McGregor, David Sykes (Addison-Wesley Professional)
- *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (GoF) (Addison-Wesley Professional Computing Series)

Research Papers and Publications

- "Patterns in Testing," by Soumen Chatterjee, International Testing Conference organized by QAI
- *The Anthology of the Finite State Machine Design Patterns*, by Paul Adamczyk, University of Illinois at Urbana-Champaign
- "Fragment Class Analysis for Testing of Polymorphism in Java Software," by Atanas Rountev, The Ohio State University; Ana Milanova, Rutgers University; and Barbara G. Ryder, Rutgers University
- "Object Test Coverage Using Finite State Machines," by Oscar Bosman and Heinz W. Schmidt, The Australian National University, 1995

- "Interface-Driven, Model-Based Generation of Java Test Drivers," by Mark Blackburn, Robert Busser, Aaron Nauman, T-VEC Technologies/SPC, Ramaswamy Chandramouli, National Institute of Standards and Technology

Internet Materials

- Object concepts
- Mock objects (PDF)
- Abstract test
- "Object and Classes"
- "Principles of Object-Oriented Design"
- A fantastic article on an abstract testing implementation example
- JUnit add-ons
- Grobo utils
- "Testing Interface Compliance with Abstract Test"
- "Constructing Testing Hierarchies"
- MockCreator
- JMock
- EasyMock
- jsystem
- "JUnit and Its Extensions Make Unit Testing Your Java Code Much Easier," by Bryan Dollery
- "Testing Private Methods with JUnit and SuiteRunner"
- Evolving framework
- "The Essence of OOP Using Java, Polymorphism Based on Overloaded Methods"
- "On Understanding Types, Data Abstraction, and Polymorphism" (PDF)
- "Reveal the Magic Behind Subtype Polymorphism"

Resources

- "Design Guidelines for 'Tailorable' Frameworks," by S. Demeyer, T. D. Meijler, O. Nierstrasz, and P. Steyaert, *Communications of the ACM*
- "Systematic Framework Design by Generalization," by Schmid, *Communications of the ACM*
- "Frameworks = (Components + Patterns)," by R. E. Johnson, *Communications of the ACM*
- "Object-Oriented Testing, Part 1"
- "Advanced Coverage Metrics for Object-Oriented Software" (PDF)
- "Object-Oriented Testing, Part 1"
- "Object-Oriented Testing: Myth and Reality"
- "Object-Oriented Testing" (PowerPoint file)
- "Fragment Class Analysis for Testing of Polymorphism in Java Software" (PDF)
- "The Full Lifecycle Object-Oriented Testing (FLOOT) Method"
- For a better understanding of inheritance in Java: *Understanding Object-Oriented Programming with Java*, by Timothy Budd (Addison Wesley)