Testing Extensible Design Patterns in Object-Oriented Frameworks through Scenario Templates

Wei-Tek Tsai
Yongzhong Tu
Weiguang Shao
Ezra Ebner

Software Engineering Laboratory
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN
email: {tsai, tu, weiguang, ebner}@cs.umn.edu

Abstract

Design patterns have been used in object-oriented frameworks such as the IBM San Francisco framework, Apple's Rhaspody, OpenStep, and WebObjects, and DIWB. However, few guidelines are available on to effectively test the software developed with design patterns. This paper first discusses the issues in testing applications developed with design patterns using an object-oriented framework. Two kinds of design patterns are available, extensible and static patterns, and this paper focuses on testing applications using extensible design patterns. Applications developed using extensible design patterns are difficult to test due to dynamic typing, dynamic binding, extensibility, and communication complexity. This paper then presents a technique, Message Framework Sequence Specifications (MfSS), for generating scenario templates that can be used to generate test cases to test applications developed using extensible design patterns and an object-oriented framework. Various partition test cases and random test cases can be generated from a MfSS. Finally, this paper uses the MfSS technique to test a small bank framework. The test cases generated successfully detected numerous faults that were seeded in the program.

Keywords: Testing, Testing Object-Oriented Design Patterns, Testing Object-Oriented Frameworks, Object-Oriented Frameworks, Extensible Design Patterns, Partition Testing, Random Testing, Scenario Templates.

1. Introduction

Recently, object-oriented design patterns [Gamma 1994, Grand 1998] have been popular. They provide extensibility in object-oriented frameworks such as the IBM San Francisco framework [Bohrer 1997, IBMSF 1997], CORBA [Siegel 1996], Apple's Rhaspody [Feiler 1997], OpenStep [Gervae 1996] and WebObjects [WebObjects 1997], and the DIWB [Ebner 1998]. However, few guidelines are available to effectively test applications developed using design patterns. This paper discusses the issues in testing

design patterns and presents a technique, Message Framework Sequence Specifications (MfSS), for generating scenario templates that can be used to generate various test cases. Both partition test cases and random test cases can be generated using MfSS. MfSS is an extension of Method Sequence Specifications (MtSS) and Message Seuqnence Specifications (MgSS) [Kirani 1994, Pressman 1997] developed earlier to test object-oriented programs. MfSS was designed to test applications developed using object-oriented frameworks.

Although many quality assurance techniques are available, testing is still the primary method used today in the software industry [Beizer 1995, Tsai 1998]. Many testing techniques have been proposed to test object-oriented programs [Kirani 1994, Pressman 1997]. However, testing design patterns, especially testing design patterns used in object-oriented frameworks, has not been addressed in the literature. Frameworks are partially completed applications when they are released [Taligent 1995]. Frameworks use design patterns to allow the application developer to expand the functionality of the framework [Bohrer 1997, IBMSF 1997, Zhu 1998]. Thus, one of the requirements to test applications developed using frameworks is to test those design patterns that are used in the frameworks.

Patterns can be categorized in to two distinct types, static and extensible. Static patterns are design patterns that do not allow easy extension, while extensible patterns are designed to allow the functionality of the application to change, sometimes even after the compilation of the application that uses the pattern. In other words, it is possible to change the system behavior at runtime by adding objects and/or classes at runtime.

The behavior of static patterns does not change after compilation. Thus, it is easier to test static patterns than to test extensible patterns. An example of a static pattern is a Façade (See Figure 1). A Façade provides an interface to a complex set of subsystems. However, the type and number of subsystems is constant and requires a rebuild for modifications.

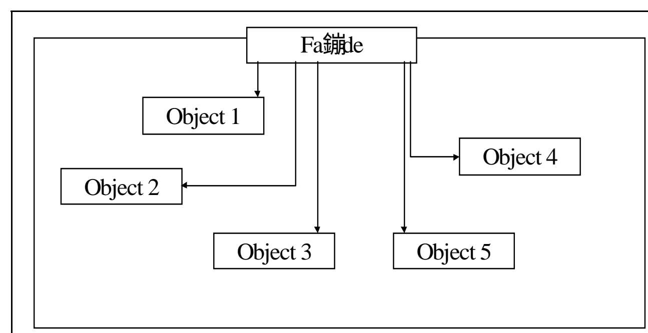

Figure 1: The Façade Design Pattern

Extensible patterns allow the software to expand by adding new classes to the system. The Factory Method and Visitor patterns (see Figure 2) are two such design pattern that can accepts new classes either at compile time or runtime. Testing extensible patterns is difficult because the tester needs to determine the behavior of the pattern in response to *unwritten* code, e.g., for those classes that will be loaded at runtime.

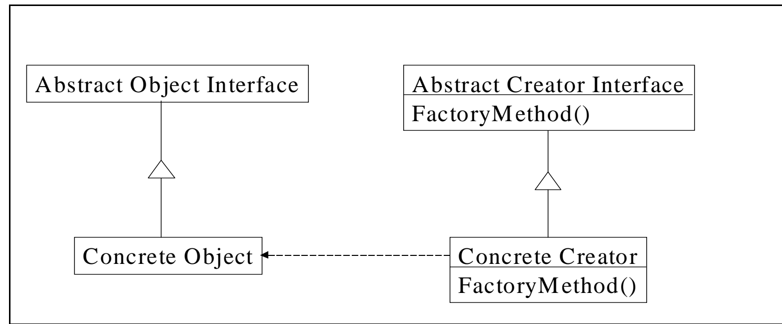

Figure 2: Factory Method Design Pattern

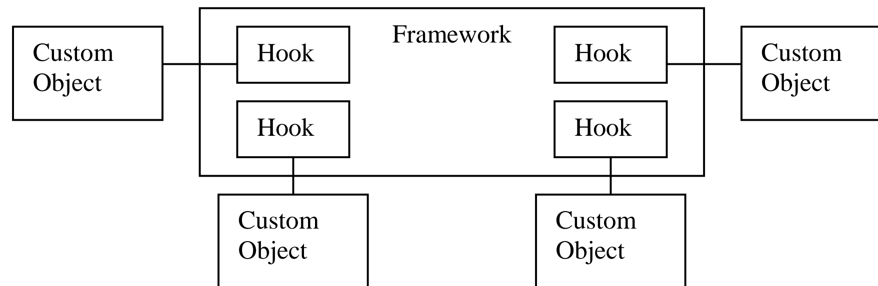

Figure 3: Framework Customization through Hooks

Extensible design patterns are used extensively in object-oriented frameworks such as the IBM San Francisco framework [Bohrer 1997] and WebObjects [WebObjects 1997]. The patterns are used as extension points or hooks where the application developer can use to expand the functionality of the framework (See Figure 3). For example, the Policy pattern is used in the San Francisco framework to allow the developers to provide custom policies for high-level transaction decisions, and the factory pattern is used to provide object creation [Bohrer 1997, IBMSF 1997].

Testing extensible design patterns in the context of an object-oriented framework is difficult because it may be impossible to determine the kind of custom objects that will be used in applications.

1.1. Organization of this Paper

This paper is organized into the following way. Section 2 describes issues in testing extensible design patterns. Section 3 presents the scenario templates to test the extensible design patterns in an object-oriented framework. The key concept is the development of Message Framework Sequence Specifications (MfSS). Section 4 describes an example application that uses design patterns. Section 5 describes various test case generation techniques using scenario templates. Section 6 applies the testing techniques in Section 5 to the example in Section 4, and presents the testing results. Section 7 concludes this paper.

2. Issues in Testing Design Patterns

Extensible design patterns are often implemented in object-oriented languages through dynamic binding and dynamic typing. These mechanisms promote reusability but at the same time introduce indirection as many of decisions are postponed until runtime.

2.1. Dynamic Typing

Many object-oriented languages such as objective-C and Java allow dynamic typing where the type of an object is determined at runtime instead of at compile time. For example, an object in Objective-C can be of type "id" and this simply says that the object can be of any type because any other types are subtypes of the type "id". Thus, any message resolution involving an id object must be done at runtime. This type of dynamic typing is used extensively in WebObjects, OpenStep and Rhapsody. Java also provides similar features by having "Object" as the parent type of all other types. Figure 4 illustrates this issue.

Dynamic typing is a powerful mechanism where the same code can be used to handle all kinds of objects, while the actual methods selected for execution will be determined at runtime. In this way, programs can exhibit dynamic behavior by selecting appropriate objects at runtime. The selection can be done by either the application program or the user without modifying the source code. However, it may also make program understanding and testing difficult.

```
//The type of 搊bject_parameter? and 搒creen? are
//unknown inside this method.
//In addition, the type of 搑esult? will be
//determined by the method  performCalculations?

-aMethod:object_parameter display:screen
{
      id result = [object_parameter performCalculations]

      [screen show:result];
}
```

Figure 4: Example WebScript Method with Dynamic Typing

Dynamic typing can be used with extensible design patterns such as Policy, Factory Method, Abstract Method and Visitor. For example, in the Policy (or Strategy) pattern [Gamma 1994], the algorithm selected can be determined at runtime depending on the object that is passed. Thus, extensible design patterns with dynamic typing provide an extremely flexible code that can handle a variety of applications. This is used extensively in WebObjects.

Unfortunately, testing applications using dynamic typing and extensible patterns can be difficult. The tester may not be able to determine the type of an object by looking at the source code alone.

2.2. Dynamic Binding

Object-oriented languages such as Java, C++, Eiffel and Smalltalk often allow dynamic binding where the method to be called is determined at runtime. Dynamic binding poses problems for the tester of the pattern, or an application that uses the patterns. For each test case the tester must determine what method is supposed to be called, since this decision is delayed until runtime, and not necessarily self-evident in the application's requirements or source code (see Figure 5).

```
//In this example from Microsoft祖 MFC, a document may
//have multiple views that a user can select to display
//the document. The actual method called by the
// pView->UpdateWindow()? is determined at runtime.
//In addition, it is impossible to determine the
//actual methods called from this code alone.

void CCustomDocument::OnRepaintAllViews()
{
   POSITION pos = GetFirstViewPosition();
   while (pos != NULL)
   {
      CView* pView = GetNextView(pos);
      pView->UpdateWindow();
   }
}
```

Figure 5: Dynamic Linking in MFC's Document-View Pattern

Extensible design patterns in frameworks contribute additional difficulties because frameworks allow developers to create subclasses of objects given in the framework. These subclasses override existing methods and add new methods to the framework. The framework then uses a callback mechanism to execute the new code.

Recall that many object-oriented languages also allow new classes to be loaded at runtime, and this feature allows the application behavior to change without changing the original source code. This makes testing applications using extensible design patterns even more difficult.

2.3. Extensibility

Extensible design patterns allow application developers to add new classes and methods to frameworks either at compile time or runtime. Considering software testing, one can examine the extensibility from two viewpoints:

1. Framework developers: The framework developers must provide some level of quality assurance for the users of the framework. Therefore, the framework designers must test the extensible design patterns to verify that they do indeed allow the application developer to expand its functionality. Unfortunately, it is not possible to predict what new classes will be added to the framework by the application developers. The customer code supplied could even be malicious (like a virus).

2. Application designers: the application developers must verify that the extension points were properly coded and tested before they can use the extension points to expand the functionality of the framework. Furthermore, the application developer must test the customer code together with the framework code to see if they meet the application requirements. The application developer can not completely trust that the framework will provide all the necessary functions correctly, especially if the framework is developed elsewhere. Unfortunately, if the framework is developed by some one else, the source code may not be available, and in this case, the application developer must treat the framework as a black box.

Therefore, both framework developer and application developer must test the hooks in the framework to verify that they function correctly. However, dynamic typing, dynamic binding and loading of classes at runtime can make testing difficult.

```
//This code allows any DLL that contains a method
//揚olicy *allocatePolicy? to allocate a Policy object
//and link it into the application.  There is no way for
//the developer of this routine to ensure that all of the
//objects linked in will be well behaved, or even that they
//will be Policy objects.
Policy *LoadCustomPolicy( char *filename)
{

   Policy *(*allocatePolicy)(void);
牋燩olicy *policy = 0;

牋牋牋//now load the dll
    HINSTANCE hint = LoadLibrary( filename);
    if( !hint)
    {
        cout << "\nUnable to load library";
        return 0;
    }

    allocatePolicy = (Policy *(* )(void))GetProcAddress(
      hint, "?allocatePolicy@@YAPAVPolicy@@XZ");

        /* If the function address is valid,
           call the function.
         */
    if( allocatePolicy)
        policy = (allocatePolicy)();
    else
        cout << "\nInvalid DLL file.";

    return policy;
}
```

Figure 6: Example Algorithm to Provide Extensibility to a Policy Pattern

Figure 6 shows a routine that allows the Policy pattern to link objects into the application at runtime. The only thing that the routine does is get any procedure named "?allocatePolicy@@YAPAVPolicy@@XZ" from the DLL file. This is the factory method for the Policy object. The routine then executes the code at the location in memory returned by the system. The routine assumes that this code is actually a factory method. Anything returned by the code is treated as a Policy object. There is no guarantee that the code will actually return a Policy object. It could return a virus or other malicious code.

2.4. Communications between framework and its application extension part

Frameworks are different from code libraries because framework objects can call custom objects and vice versa. Both directions of calling need to be considered in testing.

Framework Objects Call Custom Objects

Often, custom objects are developed after the framework has been partially completed. This is true even if the developments of the framework and application are concurrent [Suganuma 1998]. In many cases, the framework developer may not know what kinds of custom objects that will be built in the future. In these cases, it is not possible for framework objects to know exactly which custom objects to call. Framework objects may have to use message and event handling mechanisms such as the Observer pattern or the Strategy pattern [Gamma 1994] to call custom objects at the runtime.

Custom Objects Call Framework Objects

A large framework such as IBM San Francisco framework often chooses to expose certain parts of its software completely with sample code (white box), some other parts as specifications only (gray box), and the rest of parts only as features names (black box). Thus, sometimes a custom object can call a framework object directly (in case of white-box and gray-box exposures), or indirectly via some design patterns (in case of gray-box and black-box exposures).

A framework can be a black box or a white box to the custom objects, depending on the vendor of the framework. So, almost all communication methods are applicable: registration and de-registration (Observer pattern), message return, attempt to change the internal state of the framework (State and Visitor patterns) or direct call.

Frameworks can impose constraints on the way custom objects can interact with framework objects. This is done extensively in the IBM San Francisco framework. Each hook can use a design pattern to help define the interface and behavior between the custom objects and the framework. The framework can define whether the custom object has direct or indirect access to the state of the framework or can define the interface, parameters, and return types for the communication.

Objects communicate through two channels: data structures and messages. Dynamic typing and dynamic binding allow the pattern to abstract the implementation of the

methods used by the custom objects. For example, when using the Factory pattern, the framework defines an interface the custom object must implement. The implementation and data structures used by the factory are hidden from the pattern. If the Visitor pattern is used, the framework defines the interface the Visitor uses to handle the messages, and defines the data structure that the Visitor must use to communicate with the framework. Only the implementation of the methods is hidden from the pattern.

3. Scenario Templates

Multiple custom object types are often associated with one extensible design pattern. For example, in a banking application, withdrawals can be made from a checking account or saving account, and both can be custom objects, but each has its own version of the method *withdraw*. The checking account's *withdraw* may have overdraft protection while the saving account's may not. Fortunately, the overall sequence of code may be similar in spite of the fact that in different custom objects and different custom methods may be used, and in this way the overall sequence is called a *template*. Figure 7 illustrates this.

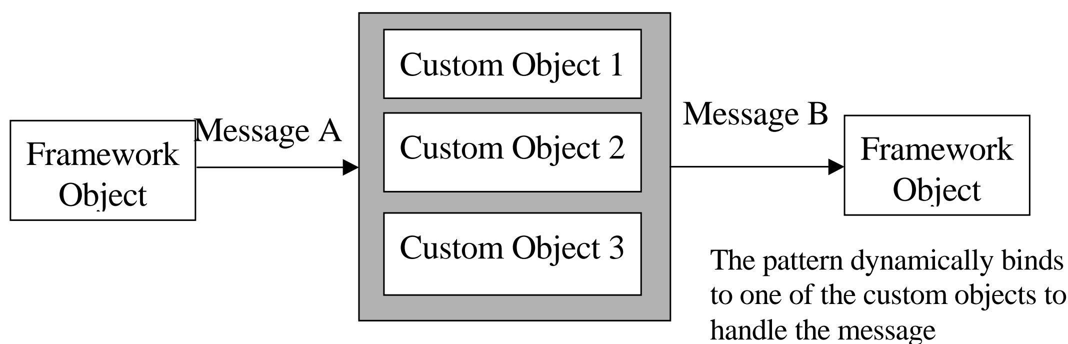

Figure 7: Example message sequence template

This template allows three custom objects are used within the same template. For example, in a banking application all accounts will have the following sequence:

1) Account creation
2) Initial deposit
3) Series of withdrawals and deposits
4) Final withdrawal from the account
5) Close the account.

This sequence is independent of the types of accounts in the banking application, i.e., it is valid for savings, checking, and other kinds of accounts. Therefore, a tester can use this sequence to generate scenarios that are valid for multiple custom objects.

3.1. Message Framework Sequence Specifications (MfSS)

Method Sequence Specification (MtSS), and Message Sequence Specification (MgSS) have been used to specify the message interaction between objects in object-oriented applications [Kirani 1994, Kirani 1994-2, Pressman 1997]. Both MtSS and MgSS were

designed to specify object-oriented applications, and they have been extended to concurrent applications [Wang 1997] and real-time applications [Vishnuvajjala 1996]. This paper extends MtSS and MgSS to MfSS (Message Framework Sequence Specifications), and use it to test applications using extensible design patterns with an object-oriented framework.

The MgSS specifies the message a particular method can send, and MtSS specifies the sequence of messages that can be sent to a class. Figure 8 demonstrates these two concepts. Currenlty, both MtSS and MgSS use the regular expression [Hopcroft 1979]. However, for some complex scenarios, it is not possible to model the sequence constraints using the regular expression. In those cases, it is necessary to use a stronger model such as a Turing machine to express the sequence constraint.

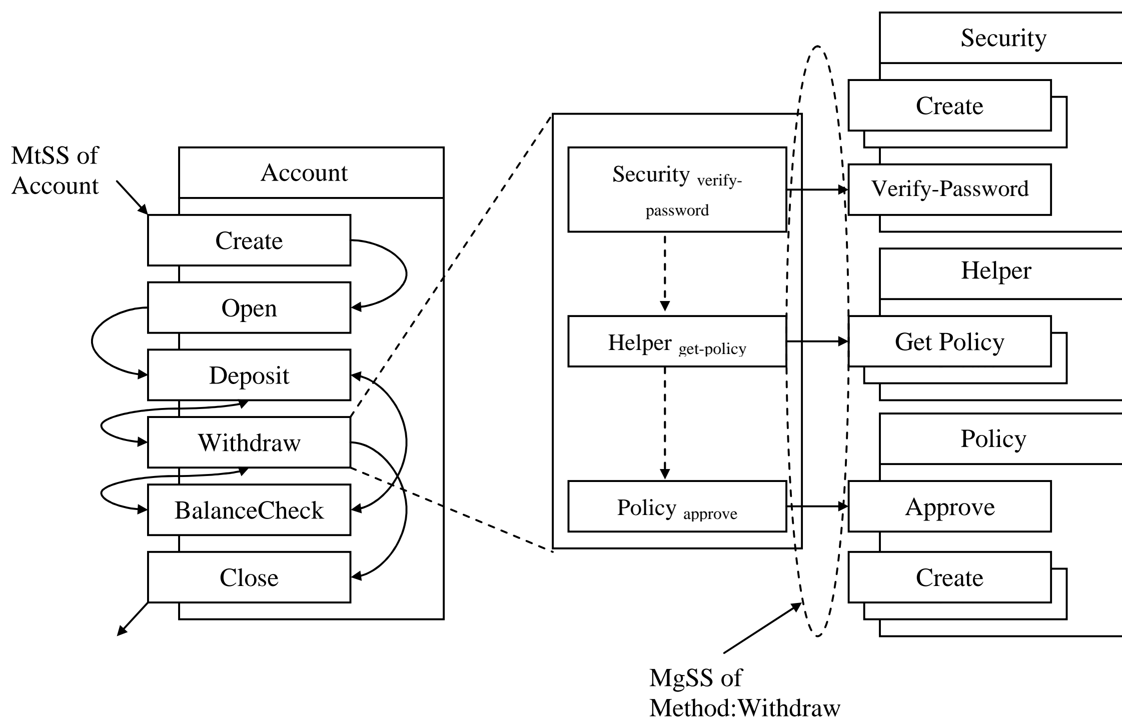

Figure 8. MtSS of Account Class and MgSS of Method *Withdraw*

The MtSS specifies the method invocation constrains for the methods defined in a class. For example, Account class defines *Create*, *Open*, *Deposit*, *Withdraw*, *BalanceCheck*, and *Close* methods. The regular expression of MtSS of an account object is

Create· Open · Deposit · (BalanceCheck | Deposit | Withdraw)* · Withdraw· Close

The MgSS specifies how a particular method call other methods. For example, the *Withdraw* method needs to send three messages to carry out its action. First, it needs to verify the password of the account. It then needs to check whether the money can be withdrawn, but this requires a policy object. Thus, after password verification, it gets a policy object, and call the policy object to check whether money can be withdrawn from the account. Thus, the MgSS of the *Withdraw* method is

$$Security_{verifypassword} \cdot Helper_{get\text{-}policy} \cdot Policy_{approve}$$

MfSS is an extension of MtSS and MgSS, and it specifies the sequence constraints on the interaction between frameworks objects and custom objects. In addition, MfSS also specifies dynamic typing and dynamic binding. Specifically, symbol { } in a MfSS to indicates dynamic typing. Methods in the { } belong to a custom object and the custom objects are loaded at runtime. Thus, the actual method called is determined at runtime. Similarly, symbol [ ] in a MfSS indicates dynamic binding. Methods in the [ ] are determined at runtime. Since the focus is on the interaction between framework objects and custom objects, {} and [] are used to denote custom objects only. Framework objects may also use dynamic typing and dynamic binding within their own computation, but they will not highlighted in MfSS.

MtSS is respect to the methods of a single class, MgSS is respect to a method; MfSS is respect to a set of objects. In addition, MfSS also specifies the dynamic behaviors such as dynamic typing and dynamic binding of the objects in the regular expression. See Table 1.

	Message Source	Message Destination	Constrants
MtSS	Multiple classes	Single class	Sequence that methods of a class are called
MgSS	Single class	Multiple classes	Sequence that a method calls other methods
MfSS	Multiple classes	Multiple classes	Sequence that objects should follow Dynamic binding and dynamic typing involved in the sequence are specified.

Table 1: Differences of MtSS, MgSS and MfSS

MfSS involves with multiple objects, thus a sequence expression must use object name together with its method name

$$ObjectName_{Method} \bullet ObjectName_{Method}$$

In the account example as shown in Figure 8, Account, Security, Helper and Policy objects can be framework objects, but Policy is an extension point where custom objects can be added. Note that a Policy object can be a framework object where a default policy is used or a custom object where a specialized policy is used.

For example, by combining the MtSS and MgSS of the account example above, the following MfSS can be generated for an account object.

MtSS of Account:

```
Create • Open • Deposit • (Checkbalance | Deposit | Withdraw)* • Withdraw
• Close
```

MgSS of *Withdraw*:

$$Security_{verifypassword} \bullet Helper_{get\text{-}policy} \bullet Policy_{approve}$$

MgSS of *CheckBalance*:

$$Security_{verify\text{-}password}$$

MfSS of the Account:

```
Account create  • Account open  •  Account deposit  •
(Account BalanceCheck • (Security verify-password )|
Account deposit  |
Account widthdraw •(Security verify-password • {Helper getpolicy }•[Policy approve ]))*
• (Security verify-password • {Helper getpolicy }• [Policy approve ])•
Account close
```

The above MfSS was obtained by inserting the MgSS of *Checkbalance* and *Withdraw* into the MtSS of Account.

As shown in the above example, [$\text{Policy}_{\text{Approve}}$] indicates that dynamic binding for the *approve* method. {$\text{Helper}_{\text{getpolicy}}$} indicates dynamic typing.

In general, an MfSS can be generated by tracing the MtSS of a client object in the framework to the MgSS of custom objects of the framework. Generally, one can start from a client object (using MtSS) in the framework to the hook objects (using MgSS), then from the hook object (using MtSS) to the custom objects of the framework (using MgSS). The sequence is then obtained by tracing the MgSS from one object to another.

3.2. Scenario Templates from the MfSS

Once a MfSS of the application has been determined, then the tester can generate scenario templates that are independent of the different custom object types in the application, by simply selecting one possible sequence of the MfSS. For example, from the MfSS in the previous section, one can generate the following scenario template:

```
Account create  • Account open  •  Account deposit  •
Account BalanceCheck • (Security verify-password ) •
Account widthdraw •(Security verify-password • {Helper getpolicy }•[Policy approve ]) •
Account close
```

To generate an actual scenario, one needs only plug in concrete objects, such as

```
SavingAccount create  • SavingAccount open  •  SavingAccount deposit  •
SavingAccount BalanceCheck • (Security verify-password ) •
SavingAccount widthdraw •(Security verify-password • {Helper getpolicy }•[StandardPolicy
approve ]) •
SavingAccount close
```

The above actual scenario is developed by replacing Account by SavingAccount, and Policy by StandardPolicy. By using different concrete objects, different scenarios can be generated. For example, by replacing the Account by CheckingAccount, and Policy by CustomPolicy, one can obtain the following scenario:

```
CheckingAccount create  • CheckingAccount open  •  CheckingAccount deposit  •
CheckingAccount BalanceCheck • (Security verify-password ) •
CheckingAccount widthdraw •(Security verify-password • {Helper getpolicy }•[CustomPolicy
approve ]) •
```

`CheckingAccount` $_{close}$

Once concrete scenarios are generated, a tester can use them to test various aspects of the software.

3.3. Scenario Template Structure

In general, a scenario template contains three sections: initialization, template path, and conclusion.

Initialization

This part initializes the environment so that it will be ready for actions. In a banking example, accounts must be set up before money can be deposited and withdrawn from them and account setup is the initialization part. The initialization part is often custom object dependent.

In a banking application, the developer may use an abstract base class "Account" to provide an interface to account objects. The developer then adds subclasses of the "Account" class to provide the methods to create a specific account, say Saving Account or Checking Account. Because each kind of account has different parameters, each subclass of account may provide its own methods to provide initialization. For example, for a checking account, it needs a checkPolicy, a checkBook and a userIdentity. So, a possible initialization sequence of the abstract object Account may be:

$Account_{SetCheckPolicy} \bullet Account_{SetCheckBook} \bullet Account_{SetUserIdentity} \bullet Account_{SetBalance}$

For a Saving Account, it has a savingPolicy and a userIdentity. Thus, its initialization sequence may be:

$Account_{SetSavingPolicy} \bullet Account_{SetUserIdentity} \bullet Account_{SetBalance}$

Template Path

The template path specifies the main actions are. This is generally custom object dependent and reused across the different custom object types. In a bank application, the transaction message sequence is sometimes independent of the objects in the transaction. For example, consider the scenario where the customer deposits the check in a new account. It doesn't matter whether the account is a saving or checking account, or whether the check is a personal or cashier's check. This message sequence is: $Account_{create} \bullet Check_{create} \bullet Account_{deposit} \bullet Check_{deposit.}$ The messages required to create the account and the check may depend on the object type, but the deposit sequence is independent of the account or check implementation. Therefore the template path is:

$Account_{deposit} \bullet Check_{deposit}$

Conclusion

The *conclusion* section handles the ending and this section is often custom object dependent. In a banking application, it may be necessary to withdraw all the money in an account before it can be closed. Thus, a reasonable conclusion is:

$(\text{Account}_{\text{withdraw}} \bullet \text{Security}_{\text{verify-password}} \bullet \text{Helper}_{\text{getPolicy}} \bullet \text{Policy}_{\text{approve}} \bullet \text{Check}_{\text{withdraw}} \bullet (\varepsilon \mid \text{Check}_{\text{delete}})) \bullet \text{Account}_{\text{close}}$

3.4 Generating Test Cases

The basic sequence for the generation of the test cases from scenario templates is as follows:

1) Develop the MtSS and MgSS of the framework and its custom objects.
2) Develop the MfSS of the framework by alternately tracing the MtSS and MgSS from object to object.
3) Then for each custom object that requires testing in the applications:
4) Generate the custom object dependant paths and input to provide initialization. For example, in a paint program a circle requires a position and radius for its initialization, while a rectangle requires the coordinates of two opposite corners.
5) Generate the custom object dependent conclusion for the template. For example, in a bank application, a checking account may need to deduct a fee from the account balance for all overdrafts, while a savings account will simply deny the overdraft.
6) Generate each custom object's expected output given the initialization, the template, and the conclusion.

4. Example for Illustration

This section presents a simple banking application to illustrate various testing techniques. The techniques proposed in this paper have been used to test for a variety of applications developed using various frameworks and component models such as CORBA and COM/DCOM. The bank processes checking and saving accounts. It allows a customer to write different types of checks against the balance of a checking account. Before the bank allows withdraws of funds, it must check whether there is sufficient fund in the account as well as the status of the account. A customer cannot withdraw money from a frozen account. The application has been implemented in Java.

4.1. Specifications

The framework implements two types of accounts: checking and saving, and it should allow other types of accounts to be added by application engineers. The bank handles deposits and withdrawals from each account. The bank issues and accepts three kinds of checks:

1) Personal checks, with values from $0.01 to $10,000;

2) Cashier's checks, with values from $0.01 to $1,000,000; and
3) Money orders, with values from $2 to $300.

The framework uses the Factory Method pattern [Gamma 1994] to create checks, and this allows new kinds of checks to be created by custom objects. The framework also uses the Policy pattern [Gamma 1994] to decide whether a check is honored. The value of each check is deducted from the checking account balance, only if the current Policy object approves the check. Currently the framework has two policies:

1) **Standard**. Pay the check as long as the account balance minus the check amount is positive.
2) **Frozen**. The account is frozen. Checks are not honored for the account, regardless of the account balance and the amount of the check.

New policies can be added by application engineers by using the extensible design pattern Policy.

4.2. Framework and Object Models

Based on these specifications, one possible object model developed for the account application is given in figure 8. In this application, classes *Account*, *Check* and *Policy* are implemented in the framework, they are also the extension points of the framework.

4.3. Scenario Templates

Method Sequence Specifications (MtSS)

Based on the specifications, the MtSS for the various objects are as follows:

- Account: different MtSSs are available for this object, here is typical one:

    Create • Open • (BalanceChecking)* • SetupPolicy • Deposit • (SetupPolicy | Deposit | Withdraw | BalanceChecking | WriteCheck)* • Withdraw • (BalanceChecking)* • Close

The method *BalanceChecking* can be invoked anytime after the account is opened and before it is closed. An account should not be closed if money is not withdrawn. After an account is opened, its policy may be changed several times from Standard to Frozen or vice versa.

- Check: a check can be used to withdraw money at first and then used to deposit or vice versa, but cannot be deposited or withdrawn more than once.
    create • (ε | withdraw • deposit | deposit • withdraw) • delete
- Policy: create • (approve)* • delete
- Security: create • (verify-password)* • delete
- Helper: create • (getPolicy)* • delete

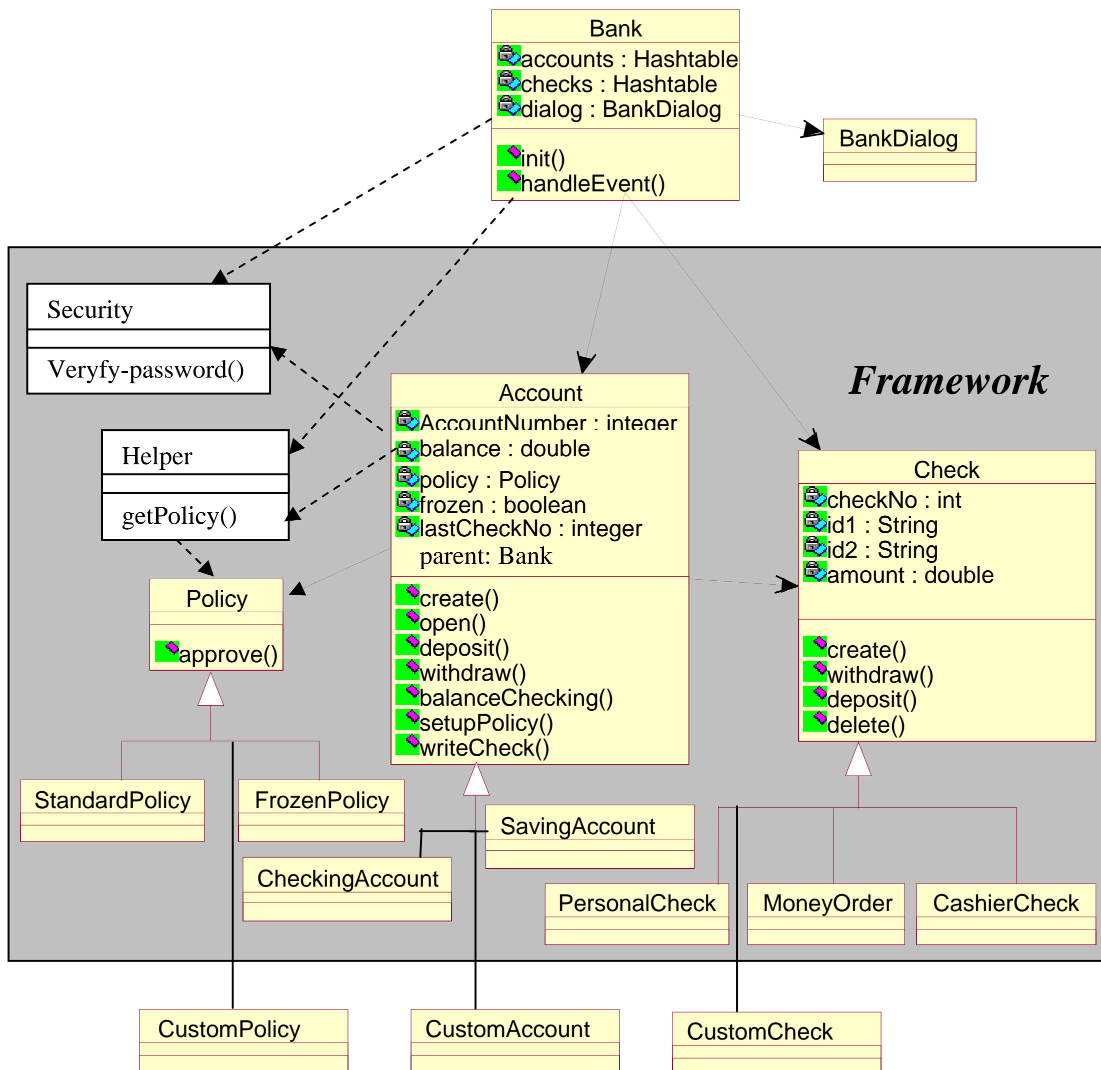

Figure 8: Object model for the account application(With GUI)

Message Sequence Specifications (MgSS)

The MgSS for the Account object's methods are listed in the Table 2.

Method	Object called	Method of the object called
Withdraw	Security Helper Policy, Check	$\text{Security}_{\text{verify-password}}$ • $\text{Helper}_{\text{getPolicy}}$ • $\text{Policy}_{\text{Approve}}$ • $\text{Check}_{\text{Withdraw}}$
Deposit	Check	$\text{Check}_{\text{Deposit}}$
WriteCheck	Check	$\text{Check}_{\text{Create}}$
Create, Open, SetupPolicy BalanceChecking and close	None	None

Table 2: MgSS for Account

There are no messages sent by any of the Policy object's methods, thus no MgSS. Check object's methods have no messages sent out either.

Message Framework Sequence Specifications (MfSS)

Start with the MtSS of account objects, one can specify one possible MfSS for the application by tracing MgSS of the methods in the MtSS:

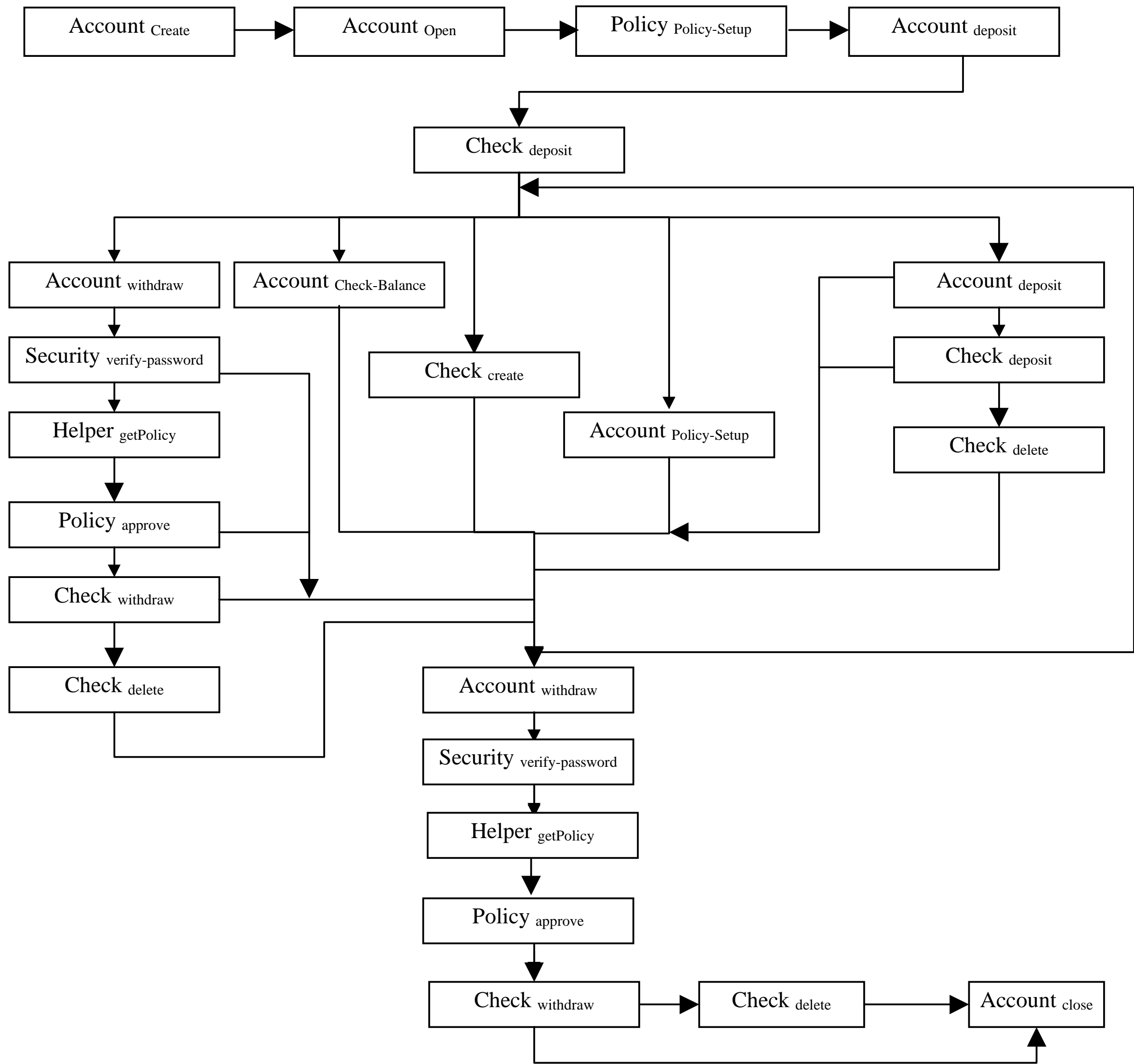

Figure 9: Message framework sequence specification for an account

Account $_{create}$ • Account $_{open}$ • Account$_{setupPolicy}$ • (Account $_{deposit}$ • Check $_{deposit}$ • (ε | Check $_{delete}$)) •

((Account $_{deposit}$ • Check $_{deposit}$ • (ε | Check $_{delete}$)) | (Account $_{withdraw}$ • Security $_{verify\text{-}password}$ • (ε | Helper $_{getPolicy}$ • Policy $_{approve}$ • (ε | Check $_{withdraw}$ • (ε | Check $_{delete}$)))) | (Account $_{writeCheck}$ • Check $_{create}$) | Account $_{checkBalance}$| Account$_{setupPolicy}$)* •
(Account $_{withdraw}$ • Security $_{verify\text{-}password}$ • Helper $_{getPolicy}$ • Policy $_{approve}$ • Check $_{withdraw}$ • (ε | Check $_{delete}$)) • Account $_{close}$

In the MfSS, Account $_{create}$ • Account $_{open}$ • Account$_{setupPolicy}$ • (Account $_{deposit}$ • Check $_{deposit}$ • (ε | Check $_{delete}$)) is the initialization part, (Account $_{withdraw}$ • Security $_{verify\text{-}password}$ • Helper $_{getPolicy}$ • Policy $_{approve}$ • Check $_{withdraw}$ • (ε | Check $_{delete}$)) • Account $_{close}$ is the conclusion part, the rest is the template path. This MfSS can be illustrated by Figure 9.

5. Generating Test Cases

Given the scenario templates, testers can focus their testing on different aspects of the application by using common testing techniques such as slicing, partition testing, stress testing, boundary testing, random testing, positive testing and negative testing. Positive testing involved in feeding the system with valid inputs, while negative testing with invalid inputs. One can obtain various types of partition test cases once the scenario templates are available, and some can be involved in boundary testing and stress testing. Random sequences can also be generated giving the scenario templates.

5.1. MfSS Slicing

Program slicing has been a popular technique in software maintenance [Huang 1996]. One can apply the slicing techniques to sequence specifications [Wang 1996] to obtain a subsequence related to certain aspects. This allows testers to focus on a certain aspect of the system. For example, given a MfSS sequence of:

Object1 $_{Message1}$ • Object2 $_{Message2}$ • Object1 $_{Message3}$

The sequence can be easily sliced with respect to Object1 and obtain the following subsequence:

Object1 $_{Message1}$ • Object1 $_{Message3}$.

It is possible to slice a sequence with respect a certain object type, a group of object types, or a collection of methods. By using various criteria, a tester is able to focus on testing certain aspects of the system.

5.2. Partition Testing

Giving a scenario template, various partition testing can be used to generate test cases. This allows for systematic testing of the pattern and the custom objects.

Partitioning According to Object Types

One possible partition of the scenarios templates allow is partitioning based on object types. This allows the developers to test the ability of the pattern to handle different objects, and it allows the developers to test the behavior of individual custom objects. For example, the bank example allows for the following partitions based on object types:

- One partition for each acceptable object type
  - CheckingAccount
  - PersonalCheck
  - CashiersCheck
  - MoneyOrder
  - StandardPolicy
  - FrozenPolicy
- Partition for invalid objects
  - IRASavingsAccount
  - IRACheckingAccount

Partition According to the Number of Objects

The number of objects in the pattern also partitions the input space. One can partition according to zero objects, one or more objects, and a large number of objects (for example, let the number of objects approaches to the available space of the machine).

Partitioning According to Groups of Objects

The scenario temples also allow the developer to see the groups of objects that will work together in the application. This allows the tester to partition the input space not only by objects, but by groups of objects; thus testing the interaction between sets of objects. For example, in the banking example one can have nine partitions based on valid combinations of objects:

- No objects
- Checking Account + no checks + Standard Policy
- Checking Account + Personal Checks + Standard Policy
- Checking Account + Cashiers Checks + Standard Policy
- Checking Account + Money Orders + Standard Policy
- Checking Account + Personal Checks + Cashiers Checks + Standard Policy
- Checking Account + Personal Checks + Money Orders + Standard Policy
- Checking Account + Cashiers Checks + Money Orders + Standard Policy
- Checking Account + Personal Checks + Cashiers Checks + Money Orders + Standard Policy

In addition, the developer can create partitions for invalid combinations of objects, thus testing the ability of the framework to handle erroneous user input and failures in the custom objects. For the banking example one such invalid combination is:

- No Checking Account + Cashiers Checks

Partitioning According to the Object Input Space

Once the patterns input space has been partitioned according to object or groups of objects, partitioning according to the input space of the individual custom objects can refine the partitions. In the banking example, the checks have different acceptable values that they can be written for, so this can be used to create the following partitions for each check type:

- Personal Checks
  - Amount < $0.01
  - $0.01 <= Amount <= $10,000
  - $10,000 < Amount
- Cashier's Checks
  - Amount < $0.01
  - $0.01 <= Amount <= $1,000,000
  - $1,000,000 < Amount
- Money Orders
  - Amount < $2
  - $2 <= Amount <= $300
  - $300 < Amount

Once the input space of the pattern is partitioned, the developer has a set of partitions that focus on various aspects of the development. This allows the systematic development of testes by generating the input and expected output for each partition of interest.

Stress Testing

Stress testing runs applications with large number of data. It can be considered as one kind of partition testing where input data are selected from a domain of large values. One can generate the following sample stress test cases for the banking example:

- Case 1: Open 100M checking accounts.
- Case 2: Create 100M personal checks.
- Case 3: Write 100M personal checks with the standard policy.

Boundary Testing

Once an input space has been partitioned, one can generate test cases near the boundaries between domains. For the banking example, the following sample boundary test cases can be generated:

- Write personal checks for $0.00, $0.01, $9999.99, $10,000.00, and $10,000.01

5.3. Negative Testing

Negative testing gives the application invalid input to measure the robustness of the application. Several negative test cases have been discussed in the above subsection. In general, one can generate a variety of invalid objects such as those that are not executable, e.g., wrong binary format and corrupted file.

For the banking application, use the following scenario template

$$\text{Account}_{\text{withdrawal}} \bullet \text{Policy}_{\text{approve}} \bullet \text{Check}_{\text{withdraw}}$$

with these invalid objects

- Account: Checking account that raises an invalid account exception.
- Checks: Personal Check that raises an invalid check exception.
- Policy: Policy that raises an invalid policy exception.

In the Factory Method, a method allocates a concrete implementation of an abstract class. To perform negative testing, the Factory Method can return objects with wrong binary format, or not executable code at all, or objects that are not the implementation of the abstract base class.

5.4. Testing Dynamic Typing

Dynamic typing can be tested by running different objects in test cases. For example, the subsequence $\text{Account}_{\text{deposit}} \bullet \text{Check}_{\text{deposit}}$ can be run using *CheckingAccount* objects, *SavingingAccount* objects, or *customAccount* (such as an *IRASavingAccount*) objects.

This template verifies that the different accounts checks for deposit. It demonstrates that the casts used by the framework and custom objects do not cause failures. When used with the group partitions, it provides an exhaustive test suite for the typing the deposit sequence.

5.5. Testing Dynamic Binding

Dynamic typing implies dynamic binding, but not vice versa. Thus, other than dynamic typing, dynamic binding involved in selecting a method at runtime. In the banking example, *approve()* is a common method for the *StandardPolicy* and *FrozenPolicy*, however, both classes have their individual implementation of the *approve()*.

- Trying to withdraw money from a frozen account:
  $\text{FrozenPolicy}_{\text{approve}} \bullet \text{PersonalCheck}_{\text{withdraw}}$
- Trying to withdraw money from a standard account:
  $\text{StandardPolicy}_{\text{approve}} \bullet \text{MoneyOrder}_{\text{withdraw}}$

5.6. Testing Extensibility

One way to illustrate the extensibility is to use those scenario templates that contain custom objects that can be replaced or determined at runtime. The following scenario template illustrates this:

$$\text{Account}_{\text{setpolicy}} \bullet \text{Account}_{\text{withdraw}} \bullet \{\text{Helper}_{\text{getpolicy}}\} \bullet [\text{Policy}_{\text{approve}}]$$

As shown in this MfSS scenario template, dynamic typing and dynamic binding are used. Specific test cases can be generated for Helper and Policy class. The test cases developed for dynamic typing and dynamic binding can be used to test the actual custom object such as a concrete policy and the functionality of the framework, such as the Helper.

5.7. Testing Communications

The scenarios allow for partitioning of the test cases to generate templates that will systematically test communication between framework objects and custom objects. The scenario templates provide suites of test cases that could test the interaction of all of the different account, policy, and check types. The following sequence tests the interaction of the three classes of objects in the withdrawal process.

$$\text{Account}_{\text{withdraw}} \bullet \text{Policy}_{\text{approve}} \bullet \text{Check}_{\text{withdraw}}$$

Then one can run test cases based on the scenario on each possible combination of types.

- Checking Account + Personal Checks + Standard Policy
- Checking Account + Cashiers Checks + Standard Policy
- Checking Account + Money Orders + Standard Policy
- Checking Account + Personal Checks + Frozen Policy
- Checking Account + Cashiers Checks + Frozen Policy
- Checking Account + Money Orders + Frozen Policy

5.8. Random Testing

One can also generate random test cases from the scenario templates no matter they are negative or positive.

Positive Random Testing

Giving a scenario template, one can generate random test cases according to the following criteria:

- Random sequences

One may randomly select a random message sequence from the MfSS. For example,

- $\text{CheckingAccount}_{\text{Deposit}} \bullet \text{CheckingAccount}_{\text{checkBalance}}$
- $\text{CheckingAccount}_{\text{Withdraw}} \bullet \text{CheckingAccount}_{\text{setupPolicy}}$

- Random objects

Once a specific sequence is chosen, one can choose some random objects to run the sequence. For example:

  - $\text{tomCheckingAccount}_{\text{create}}$ • $\text{jerrySavingAccount}_{\text{deposit}}$
  - $\text{jerrySavingAccount}_{\text{withdraw}}$ • $\text{jerryCheckingAccount}_{\text{writeCheck}}$ • $\text{MoneyOrder}_{\text{create}}$

- Random values

Once the sequence and objects are chosen, one can select random values for these objects. For example,

Case 1: Deposit $23.34 to a checking account
Case 2: Write a $99.89 personal check to Tom

Negative Random Testing

Random testing can also be negative testing. Random invalid sequences can be generated by:

- Random sequences

Randomly select a message sequence, for example, incorrect order of sequences,

$\text{CheckingAccount}_{\text{create}}$ • $\text{CheckingAccount}_{\text{Withdraw}}$
$\text{SavingAccount}_{\text{deposit}}$ • $\text{SavingAccount}_{\text{delete}}$

- Random objects

Objects are randomly selected, for example:

$\text{Check}_{\text{withdraw}}$ • $\text{Check}_{\text{withdraw}}$
$\text{Check}_{\text{delete}}$ • $\text{Check}_{\text{deposit}}$

- Random values

Select random value for objects

  - Case 1: Deposit negative dollars to a checking account
  - Case 2: Write a 0.001 dollars personal check to a custom

5.9. Combination of Partition Testing and Random Testing

It is possible to use a combination of partition testing and random testing. For example, once a random sequence is chosen, one may choose a specific set of objects with specific set of values. By using specific combination of random testing and partition testing, a tester is able to test a specific aspect of system. For example, one can generate numerous valid sequences, but with checking accounts cash deposit only, and with each transaction value more than $10,000.

6. Test Results

This section applied the test cases developed earlier to the banking example. To illustrate this, we first seeded forty-nine bugs into the code to see if the test cases developed are capable of detecting these bugs. Most of these bugs affect the functionality of the example, and forty of them are seeded in the framework, while the rest in the custom code.

6.1. Test cases

The test cases used in testing the framework are listed in Appendix 4. They follow the testing strategies described earlier.

Test techniques	Category	Test cases
Partition testing	Partition according to object types	6, 7, 8, 9, 10, 11
	Partition according to the number of objects	4, 5, 6, 12
	Partition according to groups of objects	6, 7, 8, 9, 10, 11
	Partition according to object input space	1, 2, 3
Stress testing		4, 5
Boundary testing		1, 2, 3, 24
Random testing	Random sequences	15, 26
	Random objects	6, 7, 8, 9, 10, 11
	Random values	15, 16, 19, 20, 21, 24

Table 3: Positive Testing

Techniques used	Category	Test cases
Partition testing	invalid objects	39, 40
Random testing	Random invalid sequences	17, 18, 20, 22, 23, 25, 27
	Random invalid objects	17, 18
	Random invalid values	13, 14

Table 4: Negative Testing

Purpose	Test cases
Testing dynamic typing	6, 7, 8, 9, 10, 11, 39, 40
Testing dynamic binding	6, 7, 8, 9, 10, 11, 39, 40
Testing extensibility	6, 7, 8, 9, 10, 11, 39, 40
Testing communications	6, 7, 8, 9, 10, 11

Table 5: Special Testing

Other random testing includes test cases 6, 7, 8, 9, 10, 11, 12.

6.2.Test results

All of forty-nine bugs seeded have been detected using the test cases. Table 3 shows the results. F# denotes the fault number seeded in the framework, C# stands the fault number

seeded in the custom code. From the table, only Test Case 26 did not detect any fault, and in most cases, a single test case detects multiple faults.

Test cases used	Seeded fault detected	Test cases used	Seeded fault detected
1	F35, F36, F37	14	F10
2	F32, F33, F34	15	F3, F5, F8, F22, F23, F24, C1, C2, C3
3	F29, F30, F31	16	F1, F23, F24, C4
4	F26, F28	17	F25, F9
5	F26, F28	18	F13, F27, C8
6	F14, F16	19	F2, F17, F20, F21
7	F14, F16	20	F12, F17, F19
8	F14, F16	21	F18, C6, C9, F39
9	F14, F16	22	F12
10	F14, F16	23	F27
11	F14, F16	24	F20
12	F6, C5	25	F38
13	F7,	26	
14	F10	27	F12

Table 6: Detected faults by the test cases

7. Conclusions

This paper addresses testing extensible design patterns in an object-oriented framework. Testing these patterns are difficult due to dynamic typing, dynamic binding, extensibility, communication between framework objects and custom objects. This paper proposes using MfSS to specify the interaction between framework objects and custom objects, and use the MfSS to generate scenario templates that can be used to generate actual test cases. This paper then illustrate several testing techniques such as positive testing, negative testing, test slicing, partition testing, boundary testing, random testing and stress testing based on scenario templates. Finally, this paper uses the proposed techniques to test a small banking framework with several extensible design patterns. Various partition testing criteria were used including partition based on object types, the number of objects, groups of objects and object input space. The test cases developed successfully detected all the bugs (49 of them) that were initially seeded in the code, either in the framework or in the custom code.

References

[Beizer 1995] B. Beizer, *Black-Box Testing, Techniques for Functional Testing of Software and Systems*, Wiley & Sons, 1995.

[Bohrer 1997] K. Bohrer, V. Johnson, A. Nilsson, B. Rudin, “The San Francisco Project, An Object-Oriented Framework Approach to Building Business Applications”, Proc. of IEEE COMPSAC 1997, pp. 416-424.

[Ebner 1999] E. Ebner, W. G. Shao and W. T. Tsai, “The Five-Module Framework for Internet Application Development”, ACM Computing Survey, 1999.

[Feiler 1997] J. Feiler, *Rhapsody: Developer’s Guide*, AP Professional, Boston, MA, 1997

[Gamma 1994] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[Gervae 1996] N. Gervae and P. Clark, *Developing Business Applications with OpenStep*, Springer Verlag, 1996

[Grand 1998] M. Grand, *Patterns in Java*, John Wiley & Sons, 1998.

[Hopcroft 1979] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[Huang 1996] H. Huang and W. T. Tsai, “Generalized Program Slicing”, Proc. of SEKE, 1996.

[IBMSF 1997] IBM, “IBM San Francisco Framework Programmer's Guide”, http://www.ibm.com/java/sanfrancisco/, 1997.

[Kirani 1994-1] S. H. Kirani, and W. T. Tsai, “Method Sequence Specification and Verification of Classes", Journal of Object-Oriented Programming, 1994.

[Kirani 1994-2] S. H. Kirani and W.T. Tsai, “Specification and Verification of Object-Oriented Programs”, Technical Report, Department of Computer Science, University of Minnesota, December, 1994.

[Pressman 1997] R. S. Pressman, *Software Engineering - A Practitioner's Approach*, 4th edition. McGraw-Hill, 1997.

[Siegel 1996] J. Siegel, et al., *CORBA Fundamentals and Programming*, Wiley & Sons, 1996

[Suganuma 1998] H. Suganuma, et al., “Concurrent Development on a Framework and its Application”, Proc. of IEEE COMPSAC, 1998.

[Taligent 1995] *The Power of Frameworks*, Taligent, 1995.

[Tsai 1998] W. T. Tsai, R. Mojdehbakhsh and F. Zhu, “Ensuring System and Software Reliability in Safety-Critical Systems”, Proc. of IEEE ASSET, 1998, pp.48-53.

[Vishnuvajjala 1996] R. Vishnuvajjala, W. T. Tsai, R. Mojdehbakhsh and L. Elliott, “Specifying Timing Constraints in Real-Time Object-Oriented Systems”, Proc. of IEEE High-Assurance Systems Engineering (HASE) Workshop, Oct. 1996, pp. 32-39.

[Wang 1996] Y. Wang, W. T. Tsai, X. P. Chen and S. Rayadurgam, “The Role of Program Slicing in Ripple Effect Analysis”, in Proc. of Software Engineering and Knowledge Engineering, 1996, pp. 369-376.

[Wang 1997] Y. Wang, R. Vishnuvajalla and W. T. Tsai, “Sequence Specification for Concurrent Object-Oriented Applications”, Proc. of IEEE WORDS, 1997.

[WebObjects 1997] "WebObjects Documents" at www.apple.com.

[Zhu 1998] F. Zhu and W. T. Tsai, “Framework-Oriented Analysis”, Proc. of IEEE COMPSAC, 1998, pp. 324-329.

Appendix 1: Seeded faults in the framework

Class	Method/Attributes	Fault
CheckingAccount	Account	1. The identity not initialized 2. Policy not initialized 3. Do not initialize balance 4. Do not initialize check 5. Initialize balance to 1000 6. Create CheckingAccount from the abstract class "Account"
	Deposit	7. Not check the sign of the depositing amount 8. Write code: balance = amount 9. Don't call the method deposit of the check object
	withdraw	10. Not check the sign of the withdrawing amount 11. Write code balance = amount 12. Don't call the policy to approve the withdrawal 13. Don't call the withdraw method of check
	checkCreate	14. Not create a check 15. Check number is not created 16. Check type is not checked
	policySetup	17. Policy type is wrongly set
	BalanceChecking	18. Return check number
	delete	38. MfSSConstraints not maintained
Account	withdraw	39. Delete this method
Policy	Approve	40. Delete this method
FrozenPolicy	Approve	19. Always return true
StandardPolicy	Approve	20. Delete "=" from "if (balance-amount>=0) …" 21. Always return false
Check	Create	22. Exchange name1 with name2 23. Amount not saved 24. MfSSConstraints not maintained
	Deposit	25. MfSSConstraints not maintained 26. Don't delete the check after the check is depositted and withdrawed
	Withdraw	27. MfSSConstraints not maintained 28. Don't delete the check after the check is depositted and withdrawed
	Delete	
MoneyOrder	Create	29. Boundary values are modified 30. Logic operator ">" is modified to "<" 31. Create MoneyOrder from the abstract class "Check"
CashierCheck	Create	32. Boundary values are modified 33. Logic operator "<" is modified to ">" 34. Create CashierCheck from the abstract class "Check"
PersonalCheck	Create	35. Boundary values are modified 36. Logic operator "=" is deleted 37. Create PersonalCheck from the abstract class "Check"

Appendix 2: **Faults seeded in the custom objects**

Code	Faults seeded
Create new check	1. Exchange the two names 2. Check object is not saved 3. Create result is not checked
Create account	4. The account object is not saved 5. Create checking account from abstract class *Account*
Withdraw	6. Check to withdraw is not located
Deposit	7. Check to deposit is not located
SetupPolicy	8. Mix account object "account1" with "account2"
BalanceChecking	9. Account object is not checked to be null

Appendix 3: **Test cases used**

Test Type	**Test cases**
Boundary testing	1. Create personal checks with the amount $0.001, $0.01, $10,000, and $10,001 2. Create cashier checks with the amount $0.001, $0.01, $1,000,000 and $1,000,001 3. Create money orders with the amount $0.1, $2, $300, $301
Stress testing	4. Create 1,000,000 accounts through the custom objects 5. Create 1,000,000 checks through the custom objects
Testing communication	6. Create a checking account using personal checks with the standard policy 7. Create a checking account using personal checks with the frozen policy 8. Create a checking account using cashier checks using the standard policy 9. Create a checking account using cashier checks with the frozen policy 10. Create a checking account using money orders with the standard policy 11. Create a checking account using money orders with the frozen policy

Test message sequences	12. Create a checking account 13. Deposit a negative amount of money 14. Withdraw a negative amount of money 15. Deposit a positive amount of money, and then check the balance 16. Deposit a positive amount of money with a cashier check 17. Deposit with the same check twice 18. Withdraw with the same check twice 19. Withdraw a positive amount of money with a standard account 20. Withdraw a positive amount of money from a frozen account 21. Withdraw a positive amount of money from a standard account, and then check balance 22. Setup a frozen account and then withdraw money from it 23. Setup a standard account and then withdraw money from it 24. Withdraw $200 from an standard account which exactly has $200 25. Delete an account after deposit 26. Check the balance after withdraw, and then delete the account 27. Withdraw money just after an account is created