Design-for-Testability for Object-Oriented Software[1]

Jeffery E. Payne, Roger T. Alexander, Charles D. Hutchinson

1.0 Introduction

There are many reasons why object-oriented (OO) design and development has become the norm for software creation. Two primary reasons are the positive impacts that abstraction/inheritance and information hiding have on the development process. There is little debate that data abstraction coupled with inheritance provides a powerful software design mechanism. Likewise, hiding data and internal operations behind a public interface encourages development of clean, self-contained software components. Both techniques facilitate software reuse, which many experts feel will result in significant software productivity gains as OO design technology matures.

As much as inheritance and information hiding aid software design, they have disturbingly negative consequences on testing. Studies have shown that information hiding and abstraction can actually decrease the testability of software written in an object-oriented language [1]. Likewise, our experience is that inheritance can significantly complicate the testing of objects. As the effort to test a system represents a significant part of the overall development cost, these studies and experiences raise questions about the true benefit of object-oriented development and the *testability* that OO systems actually have.

1.1 Software Testability

There are many definitions of testability. The most common is the *ease of performing testing* [2]. This definition has its roots in hardware testing and is usually defined in terms of *observability* and *controllability*. Binder defines these two facets of testability succinctly [3]:

> "To test a component, you must be able to control its input (and internal state) and observe its output. If you cannot control the input, you cannot be sure what has caused a given output. If you cannot observe the output of a component under test, you cannot be sure how a given input has been processed."

Based upon these definitions, it is intuitive how controllability and observability impact the ease of testing. Without controllability, seemingly redundant tests will produce different results. Without observability, incorrect results may appear correct as the error is contained in an output that you are unable to see.

Other facets of testability that impact the ease of testing include:

- the amount of difficulty in setting up drivers to execute a component and creating stubs for functionality that does not yet exist,
- the complexity and amount of inherited, parameterized, and polymorphic types in your software, and
- the thoroughness of specification and design information available for test design.

Another definition of testability focuses on the *value of testing*. Voas argues that controllability and observability do not adequately represent all the costs associated with testing, though they are certainly part of the testability equation [4-6]. A key component is the ability of a test to reveal faults. Testing is of less value if a particular testing activity fails to locate existing problems. This value definition of testability attempts to measure the amount of effort necessary to adequately test a system such that all faults are found.

Our research in software testability has yielded evidence that there is a correlation between common OO design techniques (including inheritance and information hiding) and low testability. A study performed by Reliable Software Technologies for the National Institute of Standards and Technology (NIST) discusses

[1] Payne, J. E., R. T. Alexander and C. D. Hutchinson (1997). **Design-for-Testability for Object-Oriented Software**. Object Magazine. **7**(5)**:** 34-43.

these findings [1]. From the ease of testing perspective, abstraction and the subsequent use of inheritance and polymorphism complicate the testing of object hierarchies. From the value of testing perspective, information hiding reduces the ability for faults to propagate to an observable output and hence reduces the likelihood that faults will be revealed during testing.

1.2 Design-for-testability

These concerns regarding the cost-effectiveness of OO testing have spawned research into methods for designing software to be testable. Design-for-testability (DFT) focuses on early life-cycle activities that can increase the testability of systems. The goal of DFT for object-oriented systems is to increase both the ease and value of testing such that the benefits of OO design and development are fully realized.

In traditional hardware systems, design-for-testability is well understood. The specification and creation of built-in tests and test benches are commonplace activities during hardware design. Complex hardware components are designed-for-testability to assure that they operate correctly before mass production. Care is taken to design-for-testability for these systems due to the extremely high costs associated with correcting faults identified after mass production.

Our premise is that as the criticality of OO software continues to increase, similar care is necessary to assure our systems are adequately tested. While software manufacturing is simplistic (copying files is often all that is necessary), the costs associated with correcting faults identified in the field and the loss of market share due to reliability concerns continues to grow. We believe that design-for-testability techniques provide practical mechanisms for assisting the testing process such that the software industry can reap the productivity benefits of object-oriented design and development.

This paper presents a set of practical design-for-testability techniques that can impact both the ease and value of testing. Each of these techniques is based upon the concept of a software contract. We have used these techniques on a wide variety of development projects during the past ten years and have consistently improved the ease and value of our testing.

2.0 Software Contracts

A Software Contract is a formal or systematic specification of the behavioral semantics of a class and its associated methods. There are three essential elements:

- an *invariant* expression that defines consistency for the class's state-space,
- a *precondition* for each method that defines the conditions under which the method can be invoked, and
- a *postcondition* for each method that defines precisely what the method does.

These three elements describe both the behaviors of a class (what the implementation must provide) and the requirements for its proper use (what the client must do prior to using the object). This forms the contract. The precondition constrains the client to ensure that the proper conditions exist prior to calling a method; the postcondition constrains the implementer of the method to provide a specific behavior *if* the preconditions are satisfied; the invariant constrains the state of the object within a method. Thus, if the client calls a method *m* and satisfies the precondition, the implementation of *m* must do what the postcondition describes, plus leave the object in a consistent, well-defined state. However, if the client fails to establish the precondition, then the method is not obligated to establish the postcondition. At least two possibilities exist here. The first is the cooperative non-defensive design view advocated by Meyer [7, 8]. In this view, a supplier is not obligated to provide any particular behavior if a method's precondition is not satisfied, i.e. the method's behavior is *undefined.* The second possible view is the defensive view where the supplier of a method provides behavior when a method's precondition is both satisfied and unsatisfied by a client. In the latter case, this behavior includes throwing exceptions, returning values that indicate errors conditions or setting some published global status object.

Considering Binder's definitions of controllability and observability above, preconditions and postconditions are mechanisms to aid in the testability of a class. For a given method, its precondition provides a way to control and observe the input in its implementation. It does this by preventing the execution if the precondition assertion is not satisfied. Later, after the method's implementation, we can observe the method's output in its postcondition. This aids in testability because the method itself can be the only transformation for a given input into a specific output.

The characteristics of Software Contracts described above are part of the *specification* of a class, not its implementation. However, we can make use of the contract elements in a class implementation to increase observability simply by injecting appropriately formed *assertions* at key locations in the source code. An assertion is a boolean expression that expresses some constraint that must be true at the location in the source code where it appears. For example, an assertion on a the *Pop* method for the class *Stack* shown in Listing 1 is that the stack is not empty when the method is invoked. This statement is usually expressed in the syntax of the language being used, such as the *assert* macro in C/C++. In our example we use *Precondition()*, *Postcondition()* and *Invariant()* to distinguish between different aspects of a contract. The implementation of these are part of the particular assertion package that we use at RST. The specific syntax and usage conventions will vary depending on which language (e.g Eiffel , Sather, Clu, etc.) and assertion package you use.

Assertions inherently increase the observability of internal state information. By placing assertions at key locations within a method and monitoring the state-space, the testability of an object is increased as:

- valuable knowledge is gained about the consistency and internal validity of an objects state,
- the incorrect use of methods is identified immediately, and
- implementation faults are identified and pinpointed quicker.

The following sections discuss the use of preconditions, postconditions, invariants, and data assertions to increase the observability and fault revealing ability of your software.

```
template <class T, int kMaxSize>
class Stack
{
public:
        Stack (void) : tos (0) {}

        T Pop (void)
        {
                Invariant ( StackInvariant() );
                Precondition ( ! IsEmpty() ); // stack must not be empty

                T       theResult = itsStackImplementation0tos - 1];

                tos--;

                Postcondition (!IsFull());    // Stack cannot be full
                // Post: Stack is not full and the popped element is
                // no longer an element of the stack.

                Invariant ( StackInvariant() );

                return  theResult;
        }

        void Push (const T& theT)
        {
                Invariant ( StackInvariant() );
                Precondition (! IsFull() );   // stack can't be full

                        itsStackImplementation[tos] = theT;

                tos++;

                PostCondition( itsStackImplementation[tos-1] == theT );
```

```
            // Post: The stack can't be empty, and the element at the top
            // of the stack must be the argument passed to push().

            Invariant ( StackInvariant() );
    }

    bool    IsEmpty() const
    {
            Precondition( true );
            Invariant ( StackInvariant() );
            return tos > 0;
            }

    bool    IsFull() const
    {
            Precondition( true );
            Invariant ( StackInvariant() );
            return tos == kMaxSize;
    }

    private:
    bool    StackIvariant() const
    {
            return tos >= 0 && tos <= kMaxSize;
    }

    int     tos;    // top of stack
    T       itsStackImplementation[kMaxSize];
};
```

Listing 1: Class showing use of Invariants, Preconditions, and Postconditions

2.1 Using Preconditions

Any method has a set of preconditions *regardless of whether the designer has explicitly specified them or not*. At the very least, they are present in the assumptions underlying the implementation. If they have not been specified explicitly, then a client of the method is likely to be unaware of the conditions necessary for its correct usage. This becomes apparent when the expected behavior does not occur. Determining the cause is likely to be tricky, particularly if the method's implementation is complicated (assuming that it is available). Injecting an appropriate assertion into the code to check the precondition will ensure that any invocation that fails to satisfy the precondition will be observed in the method's output. Thus, a precondition assertion can be used to increase the observability of a method at the beginning of its invocation just prior to its execution. It also serves as a correctness check on a method's usage by a client.

From a client's perspective, the input domain of a method is explicitly specified by its list of typed parameters. For the *Binomial* function:

double Binomial(double successProb, int nbrTrials, int nbrSuccesses)

the input domain is $double \times int \times int$. This is a *very* large input domain. In reality, the input domain specified by the parameters is often much smaller than stated. For example, in the *Binomial* function, *nbrTrials* must be greater than the *nbrSuccesses* [9], thus reducing the input domain to those combinations of values for which this constraint is satisfied. This fact is not explicit in the function's parameters, but instead is identified in the function's specification. Unfortunately, this is not often done.

In addition to the explicit domain specified by parameters, a method *m* also has an implicit input domain that is defined by the types of the state-variables that *m* is dependent upon in its implementation. This adds an additional dimension to its input space for each dependency. In the *Stack* class mentioned earlier (Listing 1), the method *pop* must make use of the *Stack*'s state-variables that provide the physical representation of the logical stack (*tos* and *itsStackImplementation*). This could be an array or linked-list, or some other data structure. In the example, it is an array of the template parameter *T*. Regardless, *pop* must make use of this representation to provide its behavior. Since the implicit portion of *pop*'s domain is hidden, all of its clients must be unaware of the dependence that *pop* has on the physical representation. Consequently, unless this information is made available somehow, a client will not be able to account for the method's complete input domain. The mechanism for doing this is the precondition that specifies the conditions under which a method (or function) may be called. For *pop*, the precondition is that the stack cannot be empty, as shown on line 7 of Listing 1.

2.2 Using Postconditions

Every method in a class does something. Sometimes this is simply a query that returns information about the state-space (e.g. method *IsEmpty()* in *Stack*). At other times, it is a command that potentially results in a state-change. Regardless of what it does, we want it done correctly, and this must be consistent with the specified semantics. Postcondition assertions can help assure this. Specifically, they can inform us when an implementation is not correct. Like preconditions, postconditions are also checked by injecting an assertion at the appropriate location in the source code. However, postconditions are checked at the end of a method's invocation. Hence, these types of assertions must be placed just prior to locations in the method that return control to the invoking client. This includes all explicit and implicit return statements. Assertions used in this manner perform a check on the implementation of the method to ensure that it has performed its operation correctly. If not, the assertion is violated, and this fact is propagated to the method's output. Thus, the postcondition assertion increases observability and provides a correctness check of a method's implementation.

Referring again to Listing 1, the *pop* method has a postcondition placed at line 10, prior to the *return* statement. Likewise, the postcondition for *push* also appears prior to the return statement at line 20. As this shows, the postcondition is able to check the implementation of the methods in terms of their effect on the state-space of *Stack*. Note, however, that the postcondition assertion on *pop* is not as strong as it should be. In particular, it does not say anything about what must be true of the state-variable *itsStackImplementation*. The postcondition assertion should include a clause that states that the element popped off the stack is no longer a member of the stack. However, in C++, as with most programming languages, there is not a concise way to state this fact. Consequently, in this type of circumstance, the actual postcondition for the method should be included as a comment on the postcondition assertion.

2.3 Using Invariant Assertions

One of the conditions implicit for the successful execution of a method is that the corresponding object's state-space is well defined and consistent. This condition must exist prior to the execution of the method's implementation. If not, the behavior of the method is unpredictable. It is possible for the method's precondition to include constraints that define this consistency requirement. However, every method in the class interface would have to include them explicitly. A better approach is to separate those constraints that must apply at all times (except during state changes) and place them into a special constraint referred to as *a class invariant* [10].

The class invariant specifies precisely what consistency means for each object of the class. Further, the invariant must hold prior to a execution of a method in the public interface, and immediately after. To check for this important property, an *invariant assertion* can be injected at the beginning of a method's implementation just prior to the first statement, and before the precondition assertion. The invariant assertion passively checks to make sure that an object's state is consistent prior to execution of a public method. If not, the invariant is violated and this fact is propagated to the method's output.

The other place that an invariant assertion is useful is in checking that the state-space of an object is still consistent after a method has executed. Thus, the implementation of each public method in a class has the responsibility to ensure that the invariant is preserved. State errors resulting from incorrect implementations are often subtle and difficult to find. An Invariant assertion can help by acting as a passive monitor reporting whenever an object is left in an inconsistent state after a method has finished executing.

Like postcondition assertions, invariant assertions are placed immediately prior to those locations that return control to the client, usually just after the postcondition check (shown on lines 11 and 21 for methods *pop* and *push,* respectively). They can also be placed immediately before an exception is thrown in those situations where the invariant of the object has been restored. In the latter case, something has gone wrong, but object will still be left in a consistent state; hence the need for the invariant assertion.

If any aspect of the class invariant is not preserved by a method's implementation, then the assertion is violated and that fact is propagated to the method's output. Thus, an invariant assertion provides an additional correctness check on a method's implementation with respect to the state-space consistency. Combined with appropriately formed postcondition assertions, the two provide increased observability into the general state of an object and the correctness of a particular method's implementation.

2.4 Using Data Assertions

As we have seen, precondition, postcondition, and invariant assertions enhance the testability of object-oriented programs by allowing us to passively observe both the complete input and output domains of a method *m*. However, what they do not check for are intermediate state errors that can occur as the method is executed. It is possible for a fault to exist at some location in *m* that, if executed with a particular input *i*, results in a failure that is not propagated to the output. Statements that are subsequently executed in *m* may mask the failure. In this case, *m* is said to be coincidentally correct with respect to *i*. And there is no way for the client to detect the failure. It is possible that though a failure has occurred, neither the postcondition or invariant assertions can detect it because it has been masked or it is coincidentally correct with respect to the constraints. To make matters worse, the failure may have infected the state-space of the corresponding object, making it likely that future method invocations will also fail. Coincidental correctness impacts the value of testing perspective of testability. If your software successfully masks faults such that failure is infrequenty, it is much more likely that your testing will not identify these faults. Data assertions provide a means ofexamining the internal state of your object to potentially identify corrupt data states before they are masked.

Total observability is what is desired. Unfortunately, this is not always practical, so we must settle for partial observability. As described earlier, what we want is to be guaranteed that if a failure occurs, we find out about it. We want nothing to be hidden in this situation. We can achieve this level of observability by placing *data assertions* at key locations within the body of a method. These assertions express correctness constraints on the partial execution of the method up to a given location with respect to some aspect of the state-space. What we desire is to form a data assertion that is strong enough to detect a failure if the state-space becomes infected at that particular location. Then total observability with respect to the corresponding fault can be achieved. If a failure occurs, we will find out about it because the assertion is violated. Achieving this level of observability is dependent upon a careful analysis of the method's implementation. To assist in this analysis, Voas has developed a technique for identifying locations that are likely to hide failures [4]. This identifies those locations that should be guarded by a data assertion to increase observability.

2.5 Assertions and Inheritance

The discussion above has been limited solely to testability issues related to information hiding and encapsulation. The other dimension that must be considered is inheritance and what it contributes to the problem of testability.

In this article, we consider inheritance solely from the perspective of a sub-typing relationship [11]. If a class *B* is a subtype of a class *A*, then the behavior of a *B* must be consistent with that of *A*. An instance of *B* must be freely substitutable for an instance of *A*. Clients must not be aware of the fact that they are using *B*'s instead of *A*'s. What this means is that any polymorphic method overridden by *B* must have the same semantics as the corresponding method in *A*. If *B* does override the implementation of some method *m* that is defined by *A*, since *B* is a sub-type of *A*, the precondition of *B*'s method m_B, cannot require more than *A*'s method m_A. Similarly, the postcondition of m_B cannot promise less than the postcondition of m_A. Practically speaking, this means that m_B must be able to be called anywhere that m_A can be. Further, m_B must do at least what m_A does. These precondition and postcondition requirements are expressed formally as the assertion constraints $Pre(m_A) \Rightarrow Pre(m_B)$ and $Post(m_B) \Rightarrow Post(m_A)$, respectively.

In addition to the assertion constraints on preconditions and postconditions, the implementation of m_B must assume the same responsibilities that m_A does with respect to the state-space of *A*. m_B must modify at least the same state-variables that m_A does, and in a manner that is consistent with the postcondition of m_A. If m_B does modify additional state-variables in *A*'s state-space, then care must be taken to ensure that no side-effects are introduced that will cause subsequent precondition violations of other methods defined in *A*'s interface. Further, m_B must preserve the invariant of *A*, just as any of *A*'s methods must. If any of these constraints are violated in the implementation of m_B, then the principle of substitution [12] is violated and a client will not be able to substitute an instance of *B* for an instance of *A*.

So, how do polymorphic methods and inheritance affect testability? Observe that all of the issues and techniques discussed in the previous sections above still apply. We want every method, regardless of where it is implemented, to provide as much observability as possible. For inherited non-polymorphic methods, this means that we want the base class to apply assertions as described earlier. This also applies to polymorphic methods that are inherited but not overridden in the derived class. For those inherited polymorphic methods that are overridden, we want to inject assertions to check the preconditions, postconditions, and class invariant, as we described earlier. However, the assertions must conform to the constraints governing precondition and postcondition weakening and strengthening, respectively. We must make sure that the base class's invariant is preserved as well as the derived class's. This is summarized in the rule of consistent observability for sub-typing:

> **Rule of consistent observability (for sub-typing)**: *An overriding method in a derived class must provide at least an equivalent level of observability as the overridden method when an instance of the derived class substituted for an instance of the base class.*

What this really means is that we want the methods in the derived class to provide the same level of observability to a client that the base class has. If we allow the observability to be less for the derived class implementation, then we have effectively reduced the apparent observability of the base class since there are now situations where it will have less observability when an instance of the derived class is substituted. This is due to the lower observability afforded by the implementation of the overridden method. The overriding implementation must at least ensure that the precondition and postcondition are consistent with the overridden method, and that the appropriate assertions are injected into the implementation. Further, the implementation must preserve the base class's invariant, as well that of its own class.

3.0 Use of Software Contracts

The previous sections have described how software contracts and assertions can increase testability by increasing observability. In practice, we have found that the use of assertions is extremely effective at this. They not only increase observability (which aids the ease of testing) and identify internal state errors (which aids the value of testing), but also can be used to properly specify inheritance relationships and control the sequencing of methods and. Both of these properties can ease the testing process. This section extends the use of assertions to handle inheritance relationships and method sequencing.

3.1 Down-calling

Down-call is a technique that can be used to ensure consistency across a class's public interface. Most programming languages consider only the syntactic signature of a method. They do not consider the semantics of the interface, and do not provide a way to consistently evaluate them (i.e. preconditions & postconditions) across all derivations of a class.

Down-calling is a mechanism that can be applied to ensure that the entire interface (both syntactic and semantic) is consistent for both non-polymorphic and polymorphic methods. Of these, the first is trivial. It is not possible to provide a different implementation for a non-polymorphic method in a derived class and expect it to be used when an instance of the derived class is substituted for an instance of the base class. Consequently, derivations of that class will inherit all non-polymorphic methods and their implementations. This includes both precondition and postcondition evaluations made in base class method implementations.

Polymorphic methods are problematic. If the polymorphic method has public visibility, derived classes can provide a different implementation. The problems created include: 1) derived classes may not re-implement the correct precondition and postcondition checks in the method, 2) peer derived classes may implement different precondition and postcondition checks, and 3) the derived class may not implement the precondition and postcondition checks at all. These problems now change the semantics of the interface. The manner that the interface should be used is now based on a particular derivation rather than the semantics of the original abstraction. Fundamentally, it now represents a different abstraction since the class cannot be used in its abstract form consistently across *all* derivations. This violates the sub-typing relationship between base and derived classes. The syntactic signature is of the correct form, but from a behavioral perspective, we have failed to achieve the same semantic form of the abstraction. That is, the derivation is not a sub-type of the base abstraction.

Down-calling solves this consistent semantic interface problem associated with polymorphic methods. The polymorphic call is structured so that a method's preconditions and postconditions are always evaluated consistently across all derivations of a class. The technique uses two types of methods to achieve this consistency: interface methods, and implementation methods. Interface methods are public and visible to the outside word, and can be invoked directly. However, by design, they are not polymorphic since derived classes are not permitted to provide a different implementation. Instead, the derived class will inherit the base class's implementation of the interface method. This represents a separation of concerns since the interface method manages the preconditions and postconditions for the abstraction (i.e. the semantics), and the implementation method provides a suitable conforming implementation.

Unlike interface methods, implementation methods are not public to the outside world. Their visibility is restricted to the inheritance hierarchy, and they are polymorphic. These methods are used to provide an implementation for a particular interface method. Since they are polymorphic, they allow derived classes to provide specific implementations suitable for their particular behaviors. Many languages, such as C++ and Java, provide mechanisms to support this type of restricted visibility for polymorphic methods.

Interface methods are responsible for ensuring that the logical semantics of a method are enforced. For a particular method, a check is done to ensure that the client invoking the method has established its precondition. If not, a precondition violation occurs and this fact is propagated to the method's output. Otherwise, the interface method invokes the implementation method, forwarding any parameters passed by the client. After the implementation method has executed, the interface method checks the postcondition to ensure that the implementation executed correctly. If so, any return value is passed back to the client. Otherwise, a postcondition violation occurs and this fact is also propagated to the method's output. Note that interface method also has the responsibility for checking the class invariant both before and after the implementation executes. Similarly, if an invariant violation occurs, then that information is propagated to the method's output. In this manner, the interface provides the necessary constraints to ensure that all implementations of a method across all derivations of an abstraction are correct.

To understand the use of interface and implementation methods, consider the C++ example in Listing 2, which shows a simple class *Telephone*. This example provides both the specification and implementation for the class. In the specification for class *Telephone*, the *Dial()* method is an interface method, and method *DialImplementation()* is its corresponding implementation method. In *Dial()*'s implementation, a check of the precondition is done. It then invokes the implementation method *DialImplementation().* Finally, it evaluates the postcondition.

The telephone example also illustrates that the postcondition of a method must also consider the set of exceptions that can be thrown. The exceptions thrown by derived classes must be consistent from the view of a client using the public interface. Only those exceptions that are part of the semantic interface should be thrown.

```
class Telephone
{
public:
        // Exception class thrown when unable to dial phone.
        class       Busy
        {
        };

        void Dial (void);
                // Interface Method
                // The semantics this interface provides is to dial a phone.
                // If unable to dial, the callee should catch the Busy exception.
protected:
        virtual void DialImplementation (void) = 0;
                // Implementation method that derived classes must provide to dial
                // a phone.  If unable to dial the phone derived classes must throw
                // Telephone::Busy exception

private:
        bool        offline;
};

// ----------------------------------------------------------------------------------------------------------------------------------------

void Telephone::Dial ()
{
Precondition (offline == true);

try
{
        // Call down the hierarchy with the polymorphic method.
        DialImplementation ();
}
catch (const Busy& theBusySignal)
{
        // this is expected and is rethrown to client
        // it is considered part of the interface.
        throw;
}
catch (...)
{
        // check to make sure that exceptions were not thrown that are
        // not considered part of the interface.  this is part of the postcondition
        // that needs to be evaluated
        throw PostconditionFailure ();
}

offline = false;        // set state variable to record state of object

Postcondition (offline == false);
}

// ----------------------------------------------------------------------------------------------------------------------------------------
```

Listing 2: Telephone abstraction

Now consider the C++ example in Listing 3 that extends class *Telephone*. The two derivations for class *Telephone* are *TouchToneTelephone*, and *InternetTelephone.*

```
class TouchToneTelephone : public Telephone
{
public:
protected:
        void DialImplementation (void);
                // Implementation method that dials a touch tone  telephone.
                // If unable to dial the phone, the Telephone::Busy exception
                // is thrown.
private:
};

// ----------------------------------------------------------------------------------------------------------------------------------------

void TouchToneTelephone:: DialImplementation()
```

```
{
        // Do whatever is necessary to dial a touch tone telephone.
}

// ---------------------------------------------------------------------------------------------------------------------------------------

class InternetTelephone : public Telephone
{
public:
protected:
        void DialImplementation (void);
                // Implementation method that dials an internet telephone.
                // If unable to dial the phone, the Telephone::Busy exception
                // is thrown.
private:
};

// ---------------------------------------------------------------------------------------------------------------------------------------

void InternetTelephone:: DialImplementation()
{
        // Do whatever is necessary to dial an internet telephone.
}

// ---------------------------------------------------------------------------------------------------------------------------------------
```

Listing 3: Telephone abstraction subtypes

A client that uses *Telephone*'s interface is assured that the preconditions & postconditions on its public interface are assessed consistently across all derivations, even when different kinds of *Telephone* abstractions are used.

The previous two examples show how the consistency of a class's interface can be ensured, including both the syntactic and semantic portions. This consistency reduces programming complexity, and eliminates the possibility of errors that are often made by ensuring that preconditions and postconditions are always evaluated consistently. This aids observability by satisfying the rule of consistent observability presented earlier.

3.2 Sequencing Constraints

A sequencing constraint is a variation of a precondition that requires a class's interface to be used in a particular order. There is a specific sequence in which the methods must be invoked. If the methods are invoked in an incorrect order, then the interface is being used improperly by some client. This ordering can be modeled with state transition diagrams where the event that triggers the transition are the method invocations. For example, consider the C++ example shown in Listing 4 for class *File*. In this example, there are four methods: *Open(), Close(), Read(),* and *Write().* A typical use of this class would be to open a file, read or write some data from it, and finally close it. An important problem is: how can we ensure that all clients who use class *File* will use it correctly ? Will all clients first invoke *Open()* and then some number of *Read()* and *Write()* and finally *Close()* ? Or will the client invoke first *Open()* and then *Read(),* and finally *Write()* –forgetting to close the file.

```
class   File
{
public:
        void Open (const string& theFileName);
        void Close (void);
        void Read (void* theBuffer, unsigned long theSize);
        void Write (const void* theBuffer, unsigned long theSize);

protected:
private:
};
```

Listing 4: Abstraction without sequencing constraints

Sequencing preconditions provide a mechanism to manage the states that are acceptable for a given object based on the context of how it was designed. An internal state variable can be used to keep track of the current state of the object (i.e. its history of method invocations). The implementation of each method's precondition inspects this state variable determining whether the method is being used in the proper sequence. An example of this is shown in Listing 5. This is the same example shown in Listing 4, but with sequencing preconditions added. In the figure, the state variable *itsFileState* holds the current state of the file object. This variable is based on the *FileState* enumeration type that specifies the different states of an instance of *File*. As illustrated by the state-transition diagram shown in Figure 1, there are two states that the file can occupy: *Ready For Opening*, and *Ready For I/O*. Analyzing the preconditions for the public methods, we see that *Read(), Write(), and Close()* can be called when the file is in the *Ready for I/O* state. The method *Close()* provides the transition from the *Ready For I/O* state to the *Ready For Opening* state. This is the only state that it is valid to invoke this method.

```
class       File
{
public:
            File (void);
            ~File (void);

            void Open (const string& theFileName);
            void Close (void);
            void Read (void* theBuffer, unsigned long theSize);
            void Write (const void* theBuffer, unsigned long theSize);

protected:
private:
            enum        FileState
            {
                        eReadyForOpening = 0,       // allowed to open a file & initial state
                        eIOAllowed                  // allowed to read and write to a file
            };

            FileState   itsFileState;
};

// ------------------------------------------------------------------------------------------------------------------------------------------

File::File (void)
:           itsFileState  (eReadyForOpening)
{
}

// ------------------------------------------------------------------------------------------------------------------------------------------

File::~File (void)
{
Precondition  (itsFileState == eReadyForOpening);
}

// ------------------------------------------------------------------------------------------------------------------------------------------

void File::Open (const string& theFileName)
{
Precondition  (itsFileState == eReadyForOpening);

...

itsFileState = eIOAllowed;

Postcondition  (itsFileState == eIOAllowed);
}
```

```
// ---------------------------------------------------------------------------------------------------------------------------------------------

void File::Close (void)
{
Precondition  (itsFileState == eIOAllowed);

...

itsFileState = eReadyForOpening;

Postcondition  (itsFileState == eReadyForOpening);
}

// ---------------------------------------------------------------------------------------------------------------------------------------------

void File::Read (void* theBuffer, unsigned long theSize)
{
Precondition (itsFileState == eIOAllowed);
...
Postcondition (true);
}

// ---------------------------------------------------------------------------------------------------------------------------------------------

void File::Write (const void* theBuffer, unsigned long theSize)
{
Precondition (itsFileState == eIOAllowed);
...
Postcondition (true);
}

// ---------------------------------------------------------------------------------------------------------------------------------------------
```

Listing 5: Abstraction with sequencing constraints

At construction time *itsFileState* is initialized to the *Ready For Opening* state. Also, the destructor precondition inspects *itsFileState* to determine whether the file has been closed. This helps to determine if somewhere in the program, the implementation has failed to invoke the Close () method.

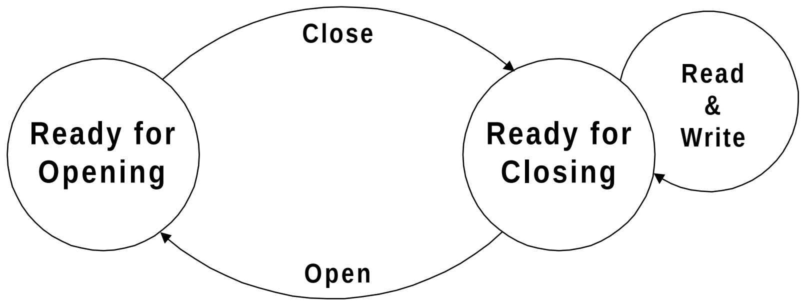

Figure 1: State-Transition Diagram modeling sequencing constraints

Sequencing preconditions aid observability by ensuring that clients using the interface do so in a manner that is semantically correct. This ordering is considered part of the contract that clients must adhere to when invoking the methods. It is dynamic in nature in the sense that it can only be inspected at execution time.

4.0 Conclusions

The benefits of object-oriented development are threatened by the testing burden that inheritance and information hiding place upon objects. In terms of both the ease of testing and the value of testing, object-oriented software has been demonstrated to have lower testability than procedural implementations. To address these concerns, software design-for-testability is becoming important. There are numerous practical approaches to increasing the testability of systems during design. We advocate the use of software contracts to increase the observability of objects and also to identify corrected internal states within a program. Our experience suggests that the use of assertions can have a significant impact on the overall testability of object-oriented software.

References

1. Corporation, R.S.T., *Testability of Object-Oriented Systems*, . 1995, National Institute of Standards and Technology: Gaithersburg, MD. Report Number: NIST GCR 95-675

2. *IEEE Standard Glossary of Software Engineering Terminology*, . 1990, IEEE Computer Sociey.

3. Binder, R.V., *Design for Testability with Object-Oriented Systems.* Communications of the ACM, 1994. 37(9): p. 87-101.

4. Voas, J.M., *PIE: a dynamic failure-based technique*. IEEE Transactions on Software Engineering, 1992. 18(8): p. 717-27.

5. Voas, J.M. and K.W. Miller, *Software Testability: The New Verification*. IEEE Software, 1995. 12(3): p. 17-28.

6. Friedman, M.A. and J.M. Voas, *Software Assessment*. 1995, New York: John Wiley & Sons.

7. Meyer, B., *Design By Contract*, in *Advances in Object-Oriented Software Engineering*, D. Mandrioli and B. Meyer, Editors. 1991, Prentice Hall: Englewood Cliffs, N.J. p. 1-50.

8. Meyer, B., *Applying 'design by contract'*. Computer, 1992. 25(10): p. 40-51.

9. Snedecor, G.W. and W.G. Cochran, *Statistical Methods*. 8 ed. 1989: IOWA State University Press.

10. Meyer, B., *Object-Oriented Software Construction*. 2 ed. 1997, Englewood Cliffs, New Jersey: Prentice-Hall.

11. Liskov, B. and J.M. Wing. *Specifications and their use in defining subtypes*. in *8th Annual ACM Conference on Object-Oriented Programming Systems,*

Languages, and Applications, OOPSLA 1993. 1993. Washington, DC, USA.

12. Liskov, B. *Data abstraction and hierarchy*. in *OOPSLA '87: Conference on Object Orientated Programming, Systems, Languages and Applications.* 1987. Orlando, FL, USA.