Abstract: Highlights from Liskov’s book, *Program Development in Java*, which I believe are important to cover while preparing for the software construction Ph.D. qualifying exam at George Mason University.

Contents

1 Procedural Abstraction
2 Exception Handling
3 Data Abstraction
4 Iterators
5 Type Hierarchy
   5.1 Dispatching
   5.2 Inheritance
6 Polymorphic Abstraction
   6.1 Comparable
   6.2 Adder

1 Procedural Abstraction

The behavior of a partial procedure is undefined for some possible inputs. On invalid inputs it could loop forever or, worse, produce erroneous output.

The behavior of a total procedure is defined for all inputs in the domain. Exceptions are a common way of turning partial procedures into total procedures.

Specifications for procedures define three properties:

- **REQUIRES**: States any constraints on use. The absence of a **REQUIRES** clause indicates that the procedure is total. Procedures should verify the constraints when possible, but not all partial procedures can or do validate their inputs.

- **MODIFIES**: Identifies all modified inputs, i.e. side effects. The absence of a **MODIFIES** clause indicates that no inputs are modified.
• **EFFECTS**: Defines the behavior.

Specifications should be written in terms of the abstract state of the object, without reference to its internal representation.

## 2 Exception Handling

**Exceptions** are specified in the `throws` clause of a specification.

If an exception is thrown for a given subset of inputs, then those inputs should *not* be documented as invalid in the `REQUIRES` clause. Throwing an exception is part of the normal behavior of a procedure. Total procedures may throw exceptions.

In Java,

- **Unchecked exceptions** all inherit from `RuntimeException`. Standard unchecked exceptions include `NullPointerException (NPE)`, `IllegalStateException (ISE)`, `IllegalArgumentException (IAE)` and `IndexOutOfBoundsException`.

- **Checked exceptions** are all other subclasses of `Exception`. Java requires checked exceptions to be listed in the `throws` clause, and when a code that might throw a checked exception *must* be handled by a `try-catch` block.

Even though Java doesn’t require it, Liskov’s methodology calls for unchecked exceptions to also be listed in the `throws` clause for documentation purposes.

Liskov recommends using checked exceptions by default, and using unchecked exceptions only if you expect that users will usually write code that ensures the exception will not happen.¹

There are two ways to deal with a thrown exception:

- **Reflection**: The caller also terminate by throwing an exception. It may throw the same exception emitted by the subprocedure (ex. by automatic propagation), or it may catch it and throw a new exception of a different type.

- **Masking**: The caller handles the exception itself and continues with execution.

## 3 Data Abstraction

**Behavioral equivalence**: Two objects should be equal only if it’s not possible to distinguish them via any sequence of calls to the objects’ methods. Since mutating mutable objects makes them distinguishable, mutable objects should be equal only if they are in fact the same object.

The clone method should call `super.clone` to return an object of the appropriate type. Liskov says it should call a constructor – this is flat-out incorrect.

Always implement `toString`! Note that, by the principle of behavioral equivalence, `toString` shouldn’t include private state – just the abstract definition of the object.

¹Yarg! It is widely agreed that checked exceptions are evil and that they break encapsulation. They are tightly coupled to the internal workings of objects, but are an extremely prominent part of the external interface. Change the exceptions emitted by private functions and you break all client code that uses a public method that calls any of those private functions, even if the exceptions themselves are never exercised.
An object’s abstraction function $AF : C \to A$ maps the concrete state of a class of type $C$ to an abstract state of type $A$. Liskov calls for $AF$ to be specified as a comment in the class, but implementing toString is usually sufficient to define the abstraction function.

An object’s representation invariant $J : C \to \mathbb{B}$ is true only if the class is a “legitimate” object, i.e. if its state is consistent. Liskov calls for the invariant to be implemented as the method public boolean repOK() for all objects. It should be checked whenever a constructor or mutator is called. In the former case, if the rep failed, you can throw FailureException.

Note that immutable objects may have a mutable rep. This is the case when all side effects of its methods are benevolent side effects, which modify the rep (concrete state) but not the abstract state.

If a method exposes the underlying representation, allowing the user to modify it, we can no longer use local reasoning to guarantee that the class is correct. For instance, a getter might return a reference to a mutable object without making a defensive copy.

**Definition 1.** An object’s methods can be classified as follows:
- **Creators** create objects without taking any objects of their types as inputs. Most constructors are creators. Copy constructors are not.
- **Producers** take objects of their type as inputs and create objects of their type.
- **Mutators** modify objects of their type.
- **Observers** take objects of their type and return other types.

4 Iterators

An iterator is a procedure that returns a generator. Generators (i.e. Java Iterator objects) have methods to get the next element and to determine whether there are any more elements.

To implement an iterator, we need to write the iterator procedure and create a nested class that implements the Iterator interface to serve as the generator.

The abstract state of all generators is simply the sequence of items that remain to be generated. Liskov recommend writing a rep invariant for generators, but not implementing a method to check it.

5 Type Hierarchy

5.1 Dispatching

The compiler knows the apparent type of an object, but in general there is no way for it to know the actual type. For instance, consider the following:

```java
String t = "ab";
Object o = t + "c";
String r = "abc";
boolean b = o.equals(r);
```

Dynamic dispatch is the method by which Java determines that it must call String.equals in the fourth line of the code above rather than Object.equals. One way of doing this is to maintain a dispatch vector with a pointer to the appropriate method. For all objects, the compiler automatically generates code such that, when a method is invoked on an object, the dispatch vector is consulted to find the location of the appropriate method.
5.2 Inheritance

The rep invariant of a subtype should check that the rep invariant of the supertype holds, but not check it directly. Maintaining the supertype rep is the responsibility of the supertype, and the subtype shouldn’t have access to private state of the supertype.

The Liskov substitution principle states that the anything that is true about the behavior of a supertype must be true about the behavior of a subtype. Specifically, these three properties must be supported by the objects specifications:

- **Signature rule**: The subtype objects must have all the methods of the super types and their signatures must be compatible (that is, identical, except that a subtype method can have fewer exceptions than the corresponding supertype method).

- **Methods rule**: Calls of these subtype methods must “behave like” calls to the corresponding supertype methods. In particular, the subtype is only permitted to
  - *Weaken the precondition*: \( \text{pre}_{\text{super}} \Rightarrow \text{pre}_{\text{sub}}, \) and to
  - *Strengthen the postcondition*: \( (\text{pre}_{\text{super}} \land \text{post}_{\sub}) \Rightarrow \text{post}_{\text{super}}. \)

- **Properties rule**: The subtype must preserve all properties that can be proved about supertype objects.

The signature rule is checked by the compiler, but the other two rules cannot be checked at compile-time.

The equals method poses a problem to inheritance. Liskov points out that a subclass Point3 subclass of Point2 object can achieve meaningful behavior by providing separate overrides equals(Object p), equals(Point2 p), equals(Point3 p) – but she fails to observe that this breaks symmetry and transitivity, not to mention the Liskov substitution principle. In general, inheritance makes it impossible to satisfy the Java equals contract, which requires equals to be an equivalence relation (reflexive, symmetric and transitive).

6 Polymorphic Abstraction

6.1 Comparable

The Comparable interface provides a compareTo(Object x) method. If the comparison doesn’t make sense (such as a string comparing itself to an int), it throws a ClassCastException. Comparable allows us to create polymorphic types like an OrderedList that work with objects of many different types.

For already-existing types that do not inherit from Comparable, Java provides the Comparator interface for defining custom comparison methods for use with algorithms.

6.2 Adder

Similar to Compartor, Liskov’s Adder example is an interface that allows us to add two objects, subtract, or return an object to represent zero. Custom Adder implementations can be written for polynomials, etc. This allows her to build a polymorphic class SumSet that maintains a mutable set of objects and also keeps track of their sum as an Object computed by an Adder.

Clearly polymorphic abstraction in Java has been vastly extended by the introduction of generics, but Liskov’s book was written before generics were introduced.