0.1 Polymorphism

Polymorphism is an intriguing notion. Briefly put, polymorphism is the ability of a particular entity (which may be an object, a function, or a variable) to present itself as belonging to multiple types. Object-oriented languages are not unique in their support for polymorphism, but it is safe to say that polymorphism is an important feature of object-oriented languages. As explained in chapter **??**, polymorphism comes in various flavors. With regard to object-oriented languages, we usually mean inheritance or *inclusion* polymorphism. Even within this restricted interpretation, we have to make a distinction between syntactic polymorphism, which requires merely that interfaces conform, and semantic polymorphism, where conformance requirements also include behavioral properties. In this section, we will look at some simple examples in Java that illustrate how we may use the mechanisms of inheritance and (simple) delegation to define objects that have similar functionality but differ in the way that functionality is realized. These examples prepare the way for the more complex idioms and patterns presented later in this chapter. In the rest of this section we will look briefly at the polymorphic constructs offered by C++. We will also study how behavioral conformance can be enforced in C++ by including invariants and assertions. These sections may be skipped on first reading.

0.1.1 Inheritance and delegation in Java

Consider the example below, an *envelope* class that offers a *message* method. In this form it is nothing but a variation on the *hello world* example presented in the appendix.

envelope

0-1

```
public class envelope {

public envelope() { }

public void message() {
        System.out.println("hello ... ");
        }
};
```

Slide 0-1: Hello World

To illustrate the idea underlying idioms and patterns in its most simple form, we will refine the *envelope* class into the collection of classes depicted in slide **??**. We will proceed in three steps: (1) The *envelope* class will be redesigned so that it acts only as an interface to the *letter* implementation class. (2) Then we introduce a *factory* object, that is used to create *envelope* and *letter* instances. (3) Finally, we refine the *letter* class into a *singleton* class, that prevents the creation of multiple *letter* instances.