

# 1

## Idioms and patterns



Object orientation has brought about a radical shift in our notion of software development. The basic mechanisms of object-oriented programming, *encapsulation* and *inheritance*, have clear advantages when it comes to data-hiding and incremental development.

<b>Idioms and patterns</b>	2
<ul style="list-style-type: none"><li>• polymorphism – inheritance and delegation</li><li>• idioms – realizing concrete types</li><li>• patterns – a catalogue of design patterns</li><li>• events – the reactor pattern</li></ul>	
Additional keywords and phrases: <i>generic types, assertions, canonical classes, event-driven computation</i>	

1-1

Slide 1-1: Idioms and patterns

However, these basic mechanisms alone do not suffice for the realization of more complex systems. In this chapter, we will look at idioms and patterns for object and class composition. Patterns, as originally introduced in [GOF94], characterize a generic solution to a problem or dilemma in design. Idioms may be understood as the implementation techniques underlying the realization of (design) patterns. First we will look at some examples in Java, illustrating the use of inheritance and delegation for the realization of some simple idioms and

patterns. Then, we will briefly deal with polymorphism in C++, including the use of assertions that may be used to enforce contractual obligations. After discussing some of the idioms and patterns that have been employed in the *hush* framework, we will look more closely at the catalogue of design patterns introduced in [GOF94]. Finally, we will study the reactor pattern as introduced in [Schmidt95] and briefly explore event-based software architectures.

## 1.1 Polymorphism

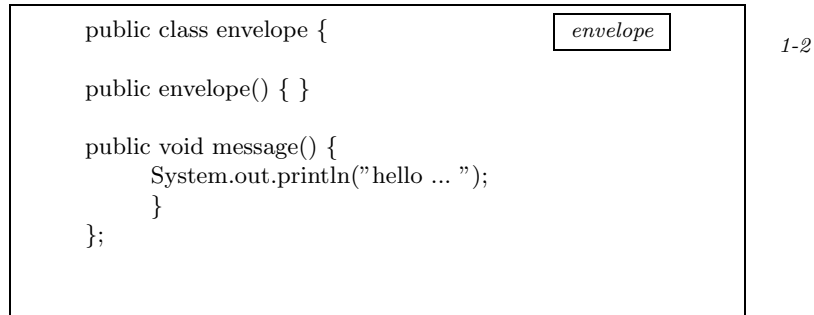
Polymorphism is an intriguing notion. Briefly put, polymorphism is the ability of a particular entity (which may be an object, a function, or a variable) to present itself as belonging to multiple types. Object-oriented languages are not unique in their support for polymorphism, but it is safe to say 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.

### 1.1.1 Inheritance and delegation in Java

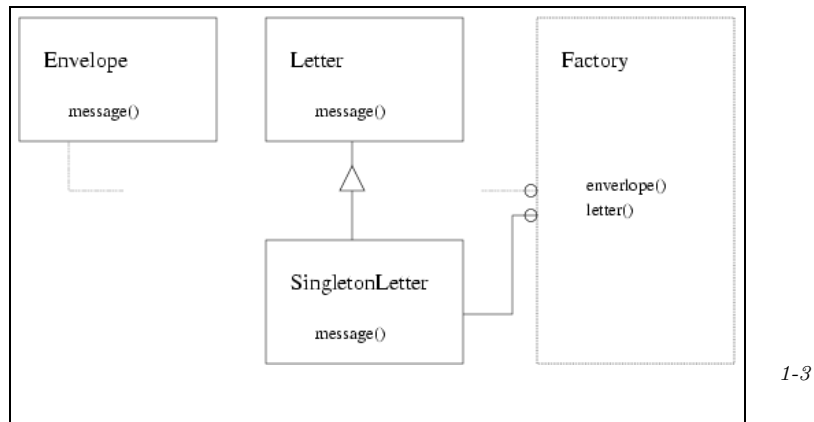
Consider the example below, an *envelope* class that offers a *message* method. In this form it is nothing but a variation on the *hello world* example presented in the appendix.

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 1-3. 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.

**Envelope/Letter** The *Envelope/Letter* idiom was introduced in [Coplien92] as a means to separate interface aspects from implementation aspects. Here the call to *message* is simply forwarded to the *letter* object.



Slide 1-2: Hello World



Slide 1-3: Envelope/Letter Factory

Admittedly, there is no need here to make such a distinction, but the idea speaks for itself. As you will see, this distinction allows us to change the implementation without modifying the *envelope* or interface class.

**Factory** In the next refinement, we introduce a *factory* object, that allows us to create *envelope* and *letter* instances without invoking a constructor.

The *factory* object is used in the *envelope* class to create a *letter*. The advantage here, as will be shown shortly, is that the *envelope* class does not need to have any information about the actual type of the *letter*.

**Singleton letter** Finally, we refine the *letter* class into a *singleton* class. When you inspect the implementation, you will see that only one instance of a *letter* will be created.

Note that the *factory* object must be modified so that the static method *instance* of *singleton* is invoked instead of the original constructor of *letter*.

```
public class envelope {
    letter impl;
    public envelope() {
        impl = new letter();
    }

    public void message() {
        impl.message();
    }
};

public class letter {

    public letter() { }

    public void message() {
        System.out.println("Message in a letter");
    }
};
```

1-4

Slide 1-4: Envelope/Letter

**Discussion** This example, however simple, demonstrates the implementation of some of the idioms and patterns that will be discussed in the rest of this chapter. It shows that the basic mechanisms of inheritance and simple delegation or forwarding are sufficient to implement these idioms and patterns. We have not discussed yet why we need idioms and patterns, but this will hopefully become clear later on.

### 1.1.2 Polymorphism in C++

Polymorphism essentially characterizes the type of a variable, function or object. Polymorphism may be due to overloading, parametrized types or inheritance. Polymorphism due to inheritance is often considered as the greatest contribution of object-oriented languages. This may be true, but the importance of generic (template) types and overloading should not be overlooked.

In slide ?? some examples are given of declarations involving polymorphic types. The function *print* is separately defined for *int* and *float*. Also, a generic *list* class is defined by means of employing *templates*. The list may be used for any kind of objects, for example integers. Finally, a *shape* class is defined from which a *circle* class is derived. An instance of the *circle* may be referred to by using a *shape* pointer, because the type *shape* encompasses *circle* objects.

```
public class factory { factory
    public factory() { }

    letter letter() { return new letter(); }
    envelope envelope() { return new envelope(); }
};

public class envelope { envelope
    letter impl;
    public envelope() {
        factory f = new factory();
        impl = f.letter(); // obtained from factory
    }

    public void message() {
        impl.message();
    }
};
```

1-5

Slide 1-5: Factory

## The Standard Template Library (STL)

The Standard Template Library for C++ provides a generic library of data structures to store, access and manipulate data. It is a generic library based on templates. In fact, it uses templates in such an aggressive way that the C++ standardization committee was forced to reconsider its definition of the generic template facility in C++. See [STL].

The Standard Template Library (STL) offers *containers*, to hold objects, *algorithms*, that act on containers, and *iterators*, to traverse containers. Algorithms, which are implemented as objects, may use *functions*, which are also defined as objects, overloading the application *operator()* method. In addition, STL offers *adaptors*, to transform objects, and *allocators*, for memory management. STL is supported by C++ compilers that adhere to the C++ standard, including Microsoft Visual C++ and the Cygnus/GNU C++ compilers. A more extensive discussion of STL is beyond the scope of this book, but the reader is advised to consult [STL], which gives an introduction to STL and its history, as well as a thorough course on programming with STL.

### 1.1.3 Assertions in C++

Whatever support a language may offer, reliable software is to a large extent the result of a disciplined approach to programming. The use of assertions has long

<pre> public class singleton extends letter {      static int number = 0;      protected singleton() { }      static letter instance() {         if (number==0) {             theletter = new letter();             number = 1;         }         return theletter;     }      public void message() {         System.out.println("Message in a letter");     }      static letter theletter; }; </pre>	<div style="border: 1px solid black; display: inline-block; padding: 2px 5px;"><i>singleton</i></div>	1-6
---	---	-----

Slide 1-6: Singleton letter

since been recognized as a powerful way in which to check whether the functional behavior of a program corresponds with its intended behavior. In effect, many programming language environments support the use of assertions in some way. For example, both C and C++ define a macro *assert* which checks for the result of a boolean expression and stops the execution if the expression is false. In the example below, assertions are used to check for the satisfying of both the pre- and post-conditions of a function that computes the square root of its argument, employing a method known as Newton iteration.

In the example, the macro *assert* has been renamed *require* and *promise* to indicate whether the assertion serves as, respectively, a pre- or post-condition. As the example above shows, assertions provide a powerful means by which to characterize the behavior of functions, especially in those cases where the algorithmic structure itself does not give a good clue as to what the function is meant to do.

The use of assertions has been promoted in [Meyer88] as a design method for object-oriented programming in Eiffel. The idea is to define the functionality of the various methods by means of pre- and post-conditions stating in a precise manner the requirements that clients of an object must meet and the obligations an object has when executing a method. Together, the collection of methods annotated with pre- and post-conditions may be regarded as a *contract* between

**Overloading**

```
extern void print(int);
extern void print(float);
```

**Generic class – templates**

```
template< class T > class list { ... }
list<int>* alist;
```

**Polymorphism by inheritance**

```
class shape { ... };
class circle : public shape { ... }
shape* s = new circle;
```

print

list< T >

shape

1-7

Slide 1-7: Polymorphic type declarations

**Standard Template Library**

- containers – *to hold objects*
- algorithms – *act on containers*
- iterators – *to traverse containers*
- functions – *as objects*
- adaptors – *to transform objects*
- allocators – *for memory management*

STL

1-8

Slide 1-8: The Standard Template Library

the object and its potential clients. See section ???. Whereas Eiffel directly supports the use of assertions by allowing access to the value of an instance variable before the execution of a method through the keyword *old*, the C++ programmer must rely on explicit programming to be able to compare the state before an operation with the state after the operation.

The annotated *counter* above includes a member function *hold* to store the value of its instance variable. It is used in the *operator++* function to check whether the new value of the counter is indeed the result of incrementing the old value. Assertions may also be used to check whether the object is correctly initialized. The pre-condition stated in the constructor requires that the counter must start with a value not less than zero. In addition, the constructor checks whether the class invariant, stated in the (virtual) member function *invariant*, is

<pre>double sqrt( double arg ) {     require ( arg &gt;= 0 );     double r=arg, x=1, eps=0.0001;     while( fabs(r - x) &gt; eps ) {         r=x; x=r-((r*r-arg)/(2*r));     }     promise ( r - arg * arg &lt;= eps );     return r; }</pre>	sqrt	1-9
---	------	-----

Slide 1-9: Using assertions in C++

satisfied. Similarly, after checking whether the post-condition of the *operator++* function is true, the invariant is checked as well.

When employing inheritance, care must be taken that the invariance requirements of the base class are not violated. The class *bounded*, given above, refines the class *counter* by imposing an additional constraint that the value of the (bounded) counter must not exceed some user-defined maximum. This constraint is checked in the *invariant* function, together with the original *counter::invariant()*, which was declared virtual to allow for overriding by inheritance. In addition, the increment *operator++* function contains an extra pre-condition to check whether the state of the (bounded) counter allows it to perform the operation. From a formal perspective, the use of assertions may be regarded as a way of augmenting the type system supported by object-oriented languages. More importantly, from a software engineering perspective, the use of assertions is a means to enforce contractual obligations.

#### 1.1.4 Canonical class idioms

The multitude of constructs available in C++ to support object-oriented programming may lead the reader to think that object-oriented programming is not at all meant to reduce the complexity of programming but rather to increase it, for the joy of programming so to speak. This impression is partly justified, since the number and complexity of constructs is at first sight indeed slightly bewildering. However, it is necessary to realize that each of the constructs introduced (classes, constructors and destructors, protection mechanisms, type conversion, overloading, virtual functions and dynamic binding) may in some way be essential to support object-oriented programming in a type-safe, and yet convenient, way. Having studied the mechanisms, the next step is to find proper ways, recipes as it were, to use these mechanisms. What we need, in the terminology of [Coplien92], are *idioms*, that is established ways of solving particular problems with the mechanisms we have available. In his excellent book, Coplien discusses a number of advanced C++ idioms for a variety of problem

```

class counter {
public:
  counter(int n = 0) : _n(n) {
    require( n != 0 );
    promise( invariant() ); // check initial state
  }

  virtual void operator++() {
    require( true ); // empty pre-condition
    hold(); // save the previous state
    _n += 1;
    promise( _n == old_n + 1 && invariant() );
  }

  int value() const { return _n; } // no side effects

  virtual bool invariant() { return value() != 0; }

protected:
  int _n;
  int old_n;
  virtual void hold() { old_n = n; }
};

```

*counter*

1-10

Slide 1-10: The counter contract

domains, including signal processing and symbolic computing. In this section, we will look at the concrete class idiom for C++, which states the ingredients that every class must have to behave as if it were a built-in type. Other idioms, in particular an improved version of the *handle/body* or *envelope/letter* idiom that may be used to separate interface from implementation, will be treated in the next section.

**Concrete data types in C++** A concrete data type is the realization of an abstract data type. When a concrete data type is correctly implemented it must satisfy the requirements imposed by the definition of the abstract data type it realizes. These requirements specify what operations are defined for that type, and also their effects. In principle, these requirements may be formally specified, but in practice just an informal description is usually given. Apart from the demands imposed by a more abstract view of the functionality of the type, a programmer usually also wishes to meet other requirements, such as speed, efficiency in terms

```

class bounded : public counter {
public:
    bounded(int b = MAXINT) : counter(0), max(b) {}
    void operator++() {
        require( value() < max ); // to prevent overflow
        counter::operator++();
    }

    bool invariant() {
        return value() <= max && counter::invariant();
    }

private:
    int max;
};

```

bounded

1-11

Slide 1-11: Refining the counter contract

of storage and error conditions, to prevent the removal of an item from an empty stack, for example. The latter requirements may be characterized as requirements imposed by implementation concerns, whereas the former generally result from design considerations.

**Canonical class in C++**

- default constructor
- copy constructor
- destructor
- assignment
- operators

Abstract data types must be indistinguishable from built-in types

1-12

Slide 1-12: Canonical class

To verify whether a concrete data type meets the requirements imposed by the specification of the abstract data type is quite straightforward, although not always easy. However, the task of verifying whether a concrete data type is optimally implemented is rather less well defined. To arrive at an optimal implementation may involve a lot of skill and ingenuity, and in general it is hard to decide whether the right choices have been made. Establishing trade-offs and making choices, for better or worse, is a matter of experience, and crucially depends upon the skill in handling the tools and mechanisms available. When

defining concrete data types, the list of requirements defining the *canonical class* idiom given in slide ?? may be used as a check list to determine whether all the necessary features of a class have been defined. Ultimately, the programmer should strive to realize abstract data types in such a way that their behavior is in some sense indistinguishable from the behavior of the built-in data types. Since this may involve a lot of work, this need not be a primary aim in the first stages of a software development project. But for class libraries to work properly, it is simply essential.

## 1.2 Idioms in *hush*

The *hush* framework, developed by the author and his colleagues, aims at providing an easy-to-use and flexible, multi-paradigm environment for developing distributed hypermedia and web-based applications. Actually *hush*, which stands for *hyper utility shell*, is a part of the DejaVU framework which has been developed at the Free University in Amsterdam over the last five years. The DejaVU framework is meant as an umbrella for our research in object-oriented applications and architectures. Many of the examples in this book are in some way derived from *hush* or applications developed within the DejaVU project. The *hush* library was originally developed in C++, but parts of it have been ported to Java using the Java native runtime interface. You will see examples of *hush* in chapters ??, ??, ??, ?? and ??. In this section a brief overview will be given of the basic concepts underlying *hush*. Then we will discuss the idioms used in realizing *hush* and its extensions, in particular an adapted version of the *handle/body* idiom originally introduced in [Coplien92], the *virtual self-reference* idioms and the *dynamic role switching* idiom. At the end of this section we will discuss the implications these idioms have for developing *hush* applications. Readers not interested in *hush* may safely skip the introduction that follows and the discussion at the end of this section. The *hush* framework is object-oriented in that it allows for a component-wise approach to developing applications. Yet, in addition to object class interfaces, it offers the opportunity to employ a script language, such as Tcl and Prolog, to develop applications and prototypes. The *hush* framework is a multi-paradigm framework, not only by supporting a multi-lingual approach, but also by providing support for distributed client/server solutions in a transparent (read CORBA) manner. In this section we will look at the idioms employed for the realization of the framework. In developing *hush* we observed that there is a tension between defining a clean object model and providing the flexibility needed to support a multiparadigm approach. We resolved this tension by choosing to differentiate between the object model (that is class interfaces) offered to the average user of the framework and the object model offered to advanced users and system-level developers. In this approach, idioms play a central role. We achieved the desired flexibility by systematically employing a limited number of basic idioms. We succeeded in hiding these idioms from the average user of the framework. However, the simplicity of our original object model is only apparent. Advanced or system-level developers who intend to define extensions to

the framework must be well aware of the patterns underlying the basic concepts, that is the functionality requirements of the classes involved, and the idioms employed in realizing these requirements.

### The *hush* framework – basic concepts

Application development generally encompasses a variety of programming tasks, including system-level software development (for example for networking or multimedia functionality), programming the user interface (including the definition of screen layout and the responsivity of the interface widgets to user actions), and the definition of (high-level) application-specific functionality. Each of these kinds of tasks may require a different approach and possibly a different application programming language. For example, the development of the user interface is often more conveniently done using a scripting language, to avoid the waiting times involved in compiling and linking. Similarly, defining knowledge-level application-specific functionality may benefit from the use of a declarative or logic programming language. In developing *hush*, we decided from the start to support a multiparadigm approach to software development and consequently we had to define the mutual interaction between the various language paradigms, as for example the interaction between C++ and a scripting language, such as Tcl. Current scripting languages, including Python and Tcl, provide facilities for being embedded in C and C++, but extending these languages with functionality defined in C or C++ and employing the language from within C/C++ is rather cumbersome. The *hush* library offers a uniform interface to a number of script languages and, in addition, it offers a variety of widgets and multimedia extensions, which are accessible through any of the script interpreters as well as the C++ interface.

These concepts are embodied in (pseudo) abstract classes that are realized by employing idioms extending the *handle/body* idiom, as explained later on.

#### Basic *hush* classes

- *session* – to manage (parts of) the application
- *kit* – to provide access to the underlying system and interpreter
- *handler* – to bind C++ functionality to events
- *event* – stores information concerning user actions or system events
- *widget* – to display information on a screen
- *item* – represents an element of a widget

1-13

Slide 1-13: Basic *hush* classes

Programming a *hush* application requires the definition of an application class derived from *session* to initialize the application and start the (window environment) main loop. In addition, one may bind Java or C++ *handler* objects to

script commands by invoking the *kit::bind* function. Handler objects are to be considered an object realization of callback functions, with the advantage that client data may be accessed in a type-secure way (that is either by resources stored when creating the handler object or by information that is passed via events). When invoked, a handler object receives a pointer to an *event* (that is, either an X event or an event related to the evaluation of a script command). Both the *widget* and (graphical) *item* class are derived from *handler* to allow for declaring widgets and items to be their own handler.

**Embedding script interpreters** The *hush* framework offers a generic *kit* that may be used as the interface to any embedded interpreter. The public interface of the *kit* class looks as follows:

```
interface kit { kit

    void eval(string cmd);
    string result();

    void bind(string name, handler h);
};
```

The function *eval* is used for evaluating (script) commands, and *result* may be used to communicate data back. The limitation of this approach, obviously, is that it is purely string based. In practice, however, this proves to be flexible and sufficiently powerful. The *bind* function may be used to define new commands and associate it with functionality defined in *handler* objects, which are introduced below.

**Handler objects** The problem of extending the script language with functionality defined by the application, is (as already indicated above) addressed by defining a generic *handler* object class. Handler objects may be regarded as a generalization of callback functions, in the sense that they are activated whenever the corresponding script command is evaluated. The advantage of using objects for callbacks instead of functions, obviously, is that we no longer need type-insecure casts, or static or global variables to pass information around. The public interface of the *handler* class looks as follows:

```
interface handler { handler
    int dispatch( event e ); // to dispatch events
    int operator();
};
```

The *dispatch* function is called by the underlying system. The *dispatch* function receives a pointer to an event which encodes the information relevant for that particular callback. In its turn *dispatch* calls the *operator()* function. Classes derived from *handler* need only redefine the *operator()* function. Information

needed when activating a handler object must be provided when creating the object, or obtained from the event for which the handler is activated. The use of handler objects is closely connected to the paradigm of event-driven computation. An *event*, conceptually speaking, is an entity that is characterized by two significant moments, the moment of its *creation* and the moment of its *activation*, its occurrence. Naturally, an event may be activated multiple times and even record a history of its activation, but the basic principle underlying the use of events is that all the information that is needed is stored at creation time and, subsequently, activation may proceed blindly. See section ??.

**User actions** Another use of handler objects (in *hush*) is for defining what must be done in response to user events, resulting from actions such as moving the mouse, or pressing a button, or selecting an entry from a menu. This is illustrated by the public interface of the generic *widget* class:

```
interface widget : handler {
    ...
    void bind( handler h );
    void bind( string action, handler h );
    ...
};
```

*widget*

The first member function *bind* may be used for installing a handler for the default bindings of the widget, whereas the second *bind* function is to be used for overriding any specific bindings. (Recall that the class *widget* is derived from *handler* class to allow the widget to be its own handler. In this way inheritance or the delegation to a separate handler object may be used to define the functionality of a widget.) In addition to the *widget* class, the *hush* library also provides the class *item*, representing graphical items. Graphical items, however, are to be placed within a canvas widget, and may be tagged to allow for the groupwise manipulation of a collection of items, as for example moving them in response to dragging the mouse pointer.

**Programmer-defined events** User interface events occur in response to actions by the user. They are scheduled by the underlying window system, which invokes the handler whenever it is convenient or necessary. When getting used to event-driven computation, system designers and programmers may feel the need to have events at their disposal that may be scheduled at will, under the programmer's control. It will come as no surprise that another use of handler objects is to allow for programmer-defined events. The public interface of the class *event* looks as follows:

```
interface event : handler {
    operator();
};
```

*event*

Actual event classes are derived from the generic class *event*, and a scheduler is provided to activate events at the appropriate time. (In effect, we provide a fully functional discrete event simulation library, including facilities for generating random distributions and analysing the outcome of experiments. Business process simulations done with this library are discussed in Chapter 11.) Note that there is an important difference between programmer-defined events and system-defined events. System-defined events are delivered to the user by activating a handler callback. In contrast, programmer-defined events are (directly) activated by a scheduler. They contain, so to speak, their own handler.

**Discussion** What benefits do we derive from employing handler objects and their derivatives? One advantage is that we have a uniform way to define the functionality of script commands, callbacks to user actions and programmer-controlled events. Another, less apparent advantage, is that it allows us to incorporate a variety of functionality (including sound synthesis facilities, digital video and active documents) in a relatively straightforward fashion.

### 1.2.1 The handle/body idiom

The handle/body class idiom, originally introduced in [Coplien92], separates the class defining a component's abstract interface (the handle class) from its hidden implementation (the body class). All intelligence is located in the body, and (most) requests to the handle object are delegated to its implementation. In order to illustrate the idiom, we use the following class as a running example:

<pre>class A {      public A() { }      public void f1() { System.out.println("A.f1"); f2(); }     public void f2() { System.out.println("A.f2"); } };</pre>	<div><i>A – naïve</i></div>	1-14
--	-----------------------------	------

Slide 1-14: Running example

The implementation of *A* is straightforward and does not make use of the handle/body idiom. A call to the *f1()* member function of *A* will print a message and make a subsequent call to *f2()*.

Without any modification in the behavior of *A*'s instances, it is possible to re-implement *A* using the handle/body idiom. The member functions of class *A* are implemented by its body, and *A* is reduced to a simple interface class:

Note that the implementation of *A*'s body can be completely hidden from the application programmer. In fact, by declaring *A* to be the superclass of its body class, even the existence of a body class can be hidden. If *A* is a class provided

```

class A {

    public A() { body = new BodyOfA(this); }
    protected A(int x) { }

    public void f1() { body.f1(); }
    public void f2() { body.f2(); }
    public void f3() { System.out.println("A.f3"); }

    private A body;
};

```

A

  
1-15

Slide 1-15: Interface: A

by a shared library, new implementations of its body class can be plugged in, without the need to recompile dependent applications:

```

class BodyOfA extends A {

    public BodyOfA() { super(911); }

    public void f1() { System.out.println("A.f1"); f2(); }
    public void f2() { System.out.println("A.f2"); }

};

```

*BodyOfA – naive*

  
1-16

Slide 1-16: Naive: BodyOfA

In this example, the application of the idiom has only two minor drawbacks. First, in the implementation below, the main constructor of A makes an explicit call to the constructor of its body class. As a result, A's constructor needs to be changed whenever an alternative implementation of the body is required. The *Abstract Factory* pattern described in [GOF94] may be used to solve this problem in a generic and elegant way. Another (aesthetic) problem is the need for the dummy constructor to prevent a recursive chain of constructor calls. But the major drawback of the handle/body idiom occurs when deriving a subclass of A which partially redefines A's virtual member functions. Consider this definition of a derived class C:

Try to predict the output of a code fragment like:

The behavior of instances of C does indeed depend on whether the hidden implementation of its base class A applies the handle/body idiom or not! If it does, the output will be `A.f1()` `A.f2()`. because the indirect call to `f2()` in `f1()`

```
class C extends A {
  public void f2() { System.out.println("C.f2"); }
};
```

C

1-17

Slide 1-17: Usage: C

```
C c = new C; c.fl();
```

1-18

Slide 1-18: Example: calling C

will (unexpectedly) not call the redefined version of `f2()`. The original definition of `A` would of course yield `A.f1()` `C.f2()`. but this can only be obtained by deriving `C` directly from the (hidden) body class.

Note that this is an illustration of one of the main drawbacks of the OOP paradigm: the inability to change base classes at the top of a hierarchy without introducing errors in derived classes.

**Explicit invocation context** In both implementations of `A`, the call to `f2()` in `f1()` is an abbreviation of `this.f2()`. However, in the first, naive implementation of `A`, the implicit `this` reference refers to the handle object (which can be an instance of a derived class). In contrast, `this` in the `BodyOfA` will refer to the body object. As a consequence, the body object is unable to make calls to functions redefined by classes derived from the base class `A`.

We use the term *invocation context* to denote a reference to the context in which the original request for a specific service is made, and represent this by a pointer to the handle object. In other words, the handle object needs a pointer to its body to be able to delegate its functionality, and, symmetrically, the body needs a pointer to the handle in order to be able to use any redefined virtual functions.

The body can be redefined as:

The new body class is aware of the fact that it is implementing services which are accessed via the handle object. Consequently, it can use this information and is able to make calls to functions which might be redefined by descendants of `A`. Note that this solution does require some programming discipline: all (implicit) references to the body object should be changed into a reference to the invocation context. Fortunately, this discipline is only required in the body classes of the implementation hierarchy.

Descendants of the handle classes in the public interface hierarchy can share and redefine code implemented by the hidden body classes in a completely transparent way, because all code sharing takes place indirectly, via the interface

```

class BodyOfA extends A {
    public BodyOfA(A h) { super(911); handle = h; }

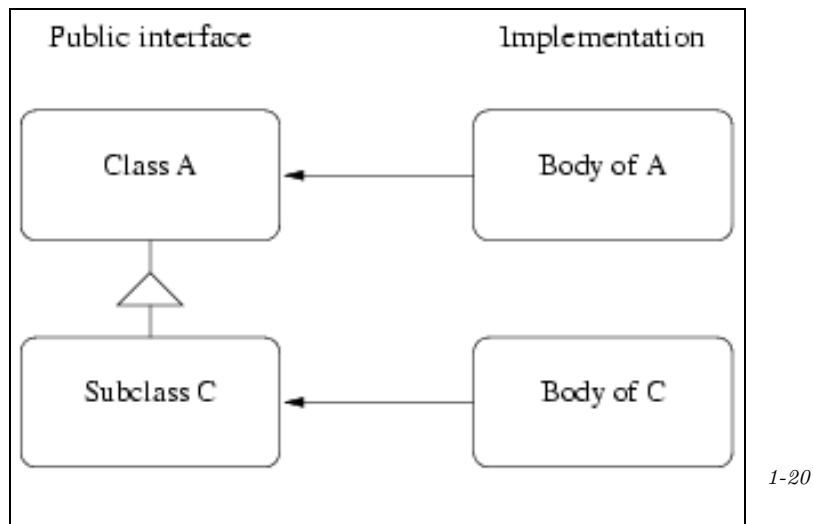
    public void f1() { System.out.println("A.f1"); handle.f2(); }
    public void f2() { System.out.println("A.f2"); }

    A handle; // reference to invocation context
};

```

1-19

Slide 1-19: Handle/Body: BodyOfA



Slide 1-20: Separating interface hierarchy and implementation

provided by the handle classes. However, even other body classes will typically share code via the handle classes. Also derived classes can use the handle/body idiom, as depicted in slide ??.

### 1.2.2 Virtual self-reference

A special feature of the *hush* widget class library is its support for the definition of new composite widgets, which provide to the application programmer the interface of a built-in (or possibly other composite) widget. To realize this flexibility, we introduced a *self()* function that adds another level of indirection to self-reference. For example, look at the *item* class below:

```
class item {  
    public item(String x) { _name = x; _self = null; }  
  
    String name() { return exists()?self().name():_name; }  
  
    public void redirect(item x) { _self = x; }  
  
    boolean exists() { return _self != null; }  
    public item self() { return exists()?_self.self():this; }  
  
    item _self;  
    String _name;  
};
```

item

1-21

Slide 1-21: Item with self()

The *item* class has an instance variable `_self`, that can be set to an arbitrary instance of *item* by invoking *redirect*. Now, when we ask for the *name* of the *item*, it is checked whether a redirection *exists*. If so, the call is redirected to the instance referenced by `self()`, otherwise the name of the *item* itself is returned.

```
public class go {  
  
    public static void main(String[] args) {  
        item a = new item("a");  
        item b = new item("b");  
        a.redirect(b);  
        System.out.println(a.name());  
    }  
  
};
```

1-22

*indeed, b*

Slide 1-22: item: go

In combination with the *handle/body* idiom, we can create composites offering the interface of *item*, providing access to one or more (inner) items. This will be further illustrated in chapter 4. Those well-versed in design patterns will recognize the *Decorator* patterns (as applied in the Interviews *MonoGlyph* class, [Interviews]).

### 1.2.3 Dynamic role-switching

For many applications, static type hierarchies do not provide the flexibility needed to model dynamically changing roles. For example we may wish to consider a person as an actor capable of various roles during his lifetime, some of which may even coexist concurrently. The characteristic feature of the *dynamic role switching* idiom underlying the *actor* pattern is that it allows us to regard a particular entity from multiple perspectives and to see that the behavior of that entity changes accordingly. We will look at a possible realization of the idiom below. Taking our view of a person as an actor as a starting point, we need first to establish the repertoire of possible behavior.

*actor*

```

class actor {

    public static final int Person = 0;
    public static final int Student = 1;
    public static final int Employer = 2;
    public static final int Final = 3;

    public void walk() { if (exists()) self().walk(); }
    public void talk() { if (exists()) self().talk(); }
    public void think() { if (exists()) self().think(); }
    public void act() { if (exists()) self().act(); }

    public boolean exists() { return false; }
    public actor self() { return this; }

    public void become(actor A) { }
    public void become(int R) { }
};

```

1-23

Slide 1-23: actor.java

Apart from the repertoire of possible behavior, which consists of the ability to *walk*, *talk*, *think* and *act*, an actor has the ability to establish its own identity (*self*) and to check whether it exists as an actor, which is true only if it has become another self. However, an actor is not able to assume a different role or to become another self. We need a *person* for that! Next, we may wish to refine the behavior of an actor for certain roles, such as for example the *student* and *employer* roles, which are among the many roles a person can play.

Only a *person* has the ability to assume a different role or to assume a different identity. Apart from becoming a *Student* or *Employer*, a person may for example become an *adult\_person* and in that capacity again assume a variety of roles.

A person may check whether he exists as a *Person*, that is whether the *Person* role differs from the person's own identity. A person's *self* may be characterized

```
class student extends actor { student
public void talk() { System.out.println("OOP"); }
public void think() { System.out.println("Z"); }
};

class employer extends actor { employer
public void talk() { System.out.println("money"); }
public void act() { System.out.println("business"); }
};
```

1-24

Slide 1-24: Students and Employers

as the actor belonging to the role the person is playing, taking a possible change of identity into account.

When a person is created, his repertoire is still empty. Only when a person changes identity by becoming a different actor (or person) or by assuming one of his (fixed) roles, is he capable of displaying actual behavior. Assuming or ‘becoming’ a role results in creating a role instance if none exists and setting the `_role` instance variable to that particular role. When a person’s identity has been changed, assuming a role affects the actor that replaced the person’s original identity. (However, only a person can change roles!) The ability to become an actor allows us to model the various phases of a person’s lifetime by different classes, as illustrated by the *adult* class.

In the example code below we have a person talking while assuming different roles. Note that the person’s identity may be restored by letting the person become its original self.

The *dynamic role switching* idiom can be used in any situation where we wish to change the functionality of an object dynamically. It may for example be used to incorporate a variety of tools in a drawing editor, as illustrated in chapter 4.

### 1.2.4 The art of *hush* programming

For the average user, programming in *hush* amounts (in general) to instantiating widgets and appropriate handler classes, or derived widget classes that define their own *handler*. However, advanced users and system-level programmers developing extensions are required to comply with the constraints resulting from the patterns underlying the design of *hush* and the application of their associated idioms in the realization of the library. The design of *hush* and its extensions can be understood by a consideration of two basic patterns and their associated idioms, that is the *nested-component* pattern (which allows for nesting components that have a similar interface) and the *actor* pattern (which allows for attributing different modes or roles to objects). The realizations of these patterns are based on idioms that extend an improved version of the familiar *handle/body* idiom. Our

improvement concerns the introduction of an *explicit invocation context* which is needed to repair the disruption of the virtual function call mechanism caused by the delegation to ‘body implementation’ objects. In this section, we will first discuss the *handle/body* idiom and its improvement. Then we will discuss the two basic patterns underlying the design of *hush* and we will briefly sketch their realization by extensions of the (improved) *handle/body* idiom.

**Invocation context** The *handle/body* idiom is one of the most popular idioms. It underlies several other idioms and patterns (e.g. the envelope/letter idiom, [Coplien92]; the Bridge and Proxy patterns, [GOF94]).

However, despite the fact that it is well documented there seems to be a major flaw in its realization. Its deficiency lies in the fact that the dynamic binding mechanism is disrupted by introducing an additional level of indirection (by delegating to the ‘body’ object), since it is not possible to make calls to member functions which are refined by subclasses of the (visible) handle class in the implementation of the (hidden) body class. We restored the working of the normal virtual function mechanism by introducing the notion of *explicit invocation context*. In this way, the *handle/body* idiom can be applied completely transparently, even for programmers of subclasses of the handle. The (improved version of) the idiom is frequently used in the *hush* class library. The widget library is build of a stable interface hierarchy, offering several common GUI widgets classes like buttons, menus and scrollbars. The widget (handle) classes are implemented by a separate, hidden implementation hierarchy, which allows for changing the implementation of the widget library, without the need to recompile dependent applications. Additionally, the idiom helps us to ensure that the various widget implementations are used in a consistent manner.

**The nested component pattern** The *nested component* pattern has been introduced to support the development of compound widgets. It allows for (re)using the script and C++ interface of possibly compound widgets, by employing explicit redirection to an inner or primary component.

Inheritance is not always a suitable technique for code sharing and object composition. A familiar example is the combination of a Text object and two scrollbars into a ScrollableText object. In that case, most of the functionality of ScrollableText will be equal to that of the Text object. This problem may be dealt with by employing multiple inheritance. Using single inheritance, it may be hard to inherit this functionality directly and add extra functionality by attaching the scrollbars, especially when interface inheritance and implementation inheritance coincide. The *nested component* pattern is closely related to the Decorator pattern treated in [GOF94] and InterViews’ notion of MonoGlyph, [Interviews]. Additionally, by using explicit delegation it provides an alternative form of code sharing to inheritance, as can be found in languages supporting *prototypes* or *exemplars*, see section ???. The *nested component* pattern is realized by applying the *virtual self-reference* idiom. Key to the implementation of that idiom is the virtual *self()* member of a component. The *self()* member returns a reference to the object itself (e.g. *this* in C++) by default, but returns the inner component

if the outer object explicitly delegated its functionality by using the *redirect()* method. Note that chasing for *self()* is recursive, that is (widget) components can be nested to arbitrary depth. The *self()* member must be used to access the functionality that may be realized by the inner component. The *nested component* pattern is employed in designing the *hush* widget hierarchy. Every (compound) widget can delegate part of its functionality to an inner component. It is common practice to derive a compound widget from another widget by using interface inheritance only, and to delegate functionality to an inner component by explicit redirection.

**The actor pattern** The *actor* pattern provides a means to offer a multitude of functional modes simultaneously. For example, a single *kit* object gives access to multiple (embedded) script interpreters, as well as (possibly) a remote kit.

The characteristic feature of the actor pattern is that it allows us to regard a particular entity as being attributed various roles or modes and that the behavior of that entity changes accordingly. Changing roles or modes can be regarded as some kind of state transition, and indeed the actor pattern (and its associated *dynamic role-switching* idiom) is closely related to the State pattern treated in [GOF94]. In both cases, a single object is used to access the current role (or state) of a set of several role (or state) classes. In combination with the *virtual self-reference* idiom, our realization of the *actor* pattern allows for changing the role by installing a new actor. The realization of the actor pattern employs the *dynamic role-switching* idiom, which is implemented by extending the handle class with a set of several bodies instead of only one. To enable role-switching, some kind of indexing is needed. Usually, a dictionary or a simple array of roles will be sufficient. In the *hush* library the actor pattern is used to give access to multiple interpreters via the same interface class (i.e. the *kit* class). The pattern is essential in supporting the multi-paradigm nature of the DeJaVU framework. In our description of the design of the Web components in section ??, we will show how *dynamic role-switching* is employed for using various network protocols via the same (*net*)*client* class. The actor pattern is also used to define a (single) viewer class that is capable of displaying documents of various MIME-types (including SGML, HTML, VRML).

### 1.3 A catalogue of design patterns

Why patterns, you may wonder. Why patterns and why not a method of object-oriented design and an introduction in one or more object-oriented languages? The answer is simple. Patterns bookmark effective design. They fill the gap between the almost infinite possibilities of object-oriented programming languages and tools and the rigor of methodical design. As Brian Foote expressed it in [POPL3], patterns are the footprints of design, paving the way for future designs. They provide a common design vocabulary and are also helpful in documenting a framework. And, as we will see later, patterns may also act as a target for

redesign, that is when the current design no longer offers the desired functionality and flexibility.

The Gang of Four book, *Design Patterns* by [GOF94], was immediately recognized as an important contribution to object-oriented software development. Not only because of the actual patterns presented, but also because of the style in which they were presented, crisp problem-oriented descriptions of actual solutions to real design problems, written with scientific rigor and accuracy. As Brian Foote remarked, actual design became a legitimate subject of computer science research. The *pattern schema*, or rather a simplified version thereof, is depicted in slide ???. Each pattern must have a *name*, which acts as a handle in discussions about the design. Being able to speak about specific pattern solutions, such as a *factory*, greatly facilitates discussions about design.

Other important entries in the pattern schema are, the *problem* indicating what the patterns is all about, the *solution* describing the general arrangement of the classes and objects involved, and the *consequences* or tradeoffs that a particular solution entails.

The actual patterns presented in [GOF94] are the result of the authors' involvement in developing various GUI toolkits, in particular Interviews, [Interviews], and ET++, [ET], and applications such as, for example, interactive text and image editors. In the course of developing a toolkit or application there are many occasions for redesign. Reasons why you may need to redesign are listed in slide ??, along with an appropriate selection of patterns from [GOF94].

Following [GOF94], we may distinguish between *creational* patterns that govern the construction and management of objects, *structural* patterns that define the static relationships between objects, and *behavioral* patterns that characterize the dynamic aspects of the interaction between objects. In this section we will look at a brief overview of the classification as originally presented in [GOF94]. The patterns themselves will be treated only briefly. The reader is invited to consult the original source and the many publications that followed: [POPL1], [POPL2], [POPL3].

### 1.3.1 Creational patterns

Design for change means to defer commitment to particular object implementations as long as possible. Due to inheritance, or rather subtyping, the client, calling a particular method, can choose the most abstract class, highest in the hierarchy. However, when it comes to creating objects, there seems to be no other choice than naming the implementation class explicitly. Wrong. Creational patterns are meant to take care of that, that is to hide the actual class used as far away as possible.

Creational patterns come in various flavors. In section 1.1.1 some example realizations were presented. The *factory* class, for example, is a rather static way of hiding the implementation classes. As an alternative, you may use a *factory method*, similar to the *instance* method of the *singleton* class. If you prefer a more dynamic approach, the *prototype* pattern might be better. A *prototype* is an object that may be used to create copies or clones, in a similar way as instances

are created from a class. However, cloning is much more dynamic, the more so if delegation is used instead of inheritance to share resources with some ancestor class. See section ???. The advantage of using a *factory*, or any of the other creational patterns, is that exchanging product families becomes very easy. Just look for example at the Java Swing library. Swing is supported under Unix, Windows and MacOS. The key to multiple platform support is here, indeed, the use of factories to create widgets. Factories are also essential when using CORBA, simply because calling a constructor is of no use for creating objects on a remote site.

### 1.3.2 Structural patterns

Objects rarely live in isolation. In slide ?? a selection of the structural patterns treated in [GOF94] is collected. Structural patterns indicate how classes and objects may be composed to form larger structures.

#### Structural patterns

- object and class composition

Pattern	Alias	Remarks
Composite	part/whole	collections of components
Flyweight	handle/body	extrinsic state, many objects
Adaptor	wrapper	resolve inconsistency between interfaces
Bridge	handle/body	relate abstraction to implementation
Decorator	handle/body	to introduce additional functionality
Facade	handle/body	provides unified interface
Proxy	handle/body	to defer ... remote, virtual, protection

Imagine, for example, an application for interactive text processing. Now, the *Composite* pattern may be used to combine text, images and also compound components, that may itself consist of other components. Closely related to the *Composite* pattern is the *Flyweight* pattern, which is needed when the number of components grows very large. In that case, the components themselves must for obvious reasons carry as little information as possible. Context or state information must then be passed as a parameter. To give some more examples, suppose there exists a nice library for formatting text and images, but unfortunately with only a procedural interface. Then the *Adaptor* pattern may be used to provide a interface that suits you, by wrapping the original library. The *Bridge* pattern is in some sense related to the *Factory*. In order to work with a platform-independent widget library, you need, as has been explained, a *factory* to hide the creation of widgets, but you also need to bridge a hierarchy of platform-dependent implementation classes to the more abstract platform-independent widget set. When creating widgets to display text or images it may be very inconvenient to create a separate class, for example when adding scrolling functionality. The *Decorator* pattern allows you to insert additional functionality without subclassing. Now think of a networked application, for example to be

able to incorporate components that are delivered by a server. The library may provide a number of networking classes that deal with all possible communication protocols. To simplify access to these classes a *Facade* may be built, hiding the complexity of the original class interfaces. Alternatively, remote components may be available through a *proxy*. The *Proxy* pattern describes how access may be regulated by an object that acts as a surrogate for the real object. Like *composites* and *decorators*, *proxies* may be used for recursive composition. However, *proxies* primarily regulate access, whereas *decorators* add responsibilities, and *composites* represent structure.

### 1.3.3 Behavioral patterns

Our final category of patterns, *behavioral patterns*, concern the construction of algorithms and the assignment of responsibilities to the objects that cooperate in achieving some goal.

A first distinction can be made between patterns that involve the composition of classes (using inheritance) and patterns that rely on object composition. As an example of the *Template Method* pattern, think of a compiler class that offers methods for scanning and the creation of a parse tree. Each of these methods may be refined without affecting the structure of the compilation itself. An *interpreter* allows for evaluating expressions, for example mathematical formula. Expressions may be organised in a hierarchy. new kinds of expressions can be inserted simply by filling in details of syntax and (semantic) evaluation. Object composition, which employs the *handle/body* idiom and delegation, is employed in the *Mediator* pattern, the *Chain of Responsibility* pattern and the *Observer* pattern. The actual task, such as for example updating the display of information when the actual information has changed, is delegated to a more specialized object, to achieve a loose coupling between components. The difference between a *mediator* and *chain of responsibility* is primarily the complexity of co-ordinating the tasks. For example, changing the format of a single image component from one image type to another image type may be done simply by using an image converter (*mediator*), whereas exporting the complete document to a particular format such as HTML may involve delegating control to a specialized converter that itself needs access to the original components (*chain of responsibility*). We will discuss the *Observer* pattern in more detail later.

A common characteristic of the patterns listed in slide ?? is that functional behavior is realized as an object. Semantically, objects are more powerful than functions, since objects can carry a state. Hence, the imperative *objectify* pays off when we need functions that must know their invocation history. As an example of the *Command* pattern, think of how you would realize insertion and deletion commands in an interactive editor, with undo! Turning these commands into an object in which the information necessary for undoing the command can be stored, for example having a snapshot of the state stored in a *Memento*, it suffices to stack the actual command objects. To undo a command, pop the stack and invoke the undo method. The *Strategy* pattern may be used to hide the details of the various layout algorithms that are available. For example, you may

use a straightforward algorithm that formats the text line by line, or you may use the much more advanced formatting algorithm of  $\text{\TeX}$ , which involves the minimalization of penalties. These alternatives can be collected in a formatting *strategy* hierarchy, that hides the implementation details from the client by a common interface. When doing the formatting, you may wish to separate the traversal of the component tree structure from the actual formatting operations. This may be accomplished by employing the *Visitor* pattern. In general it is recommended to abstract from the data structure and use a more abstract way, such as an *Iterator* or *Visitor* to access and traverse it. The *State* pattern is similar to the *dynamic role switching* idiom that has been discussed in section ???. As an example, think of providing viewers for alternative document formats, such as VRML or PDF, in your application. Using the *State* pattern, it suffices to have a single viewer that changes itself according to the format of the document viewed.

### The Observer pattern

The *Observer* pattern is a variant of the famous *Model-View-Control* (MVC) pattern, that governed the creation of the graphical user interface of the Smalltalk environment and many Smalltalk applications.

The basic idea is simple, to decouple information management and the display of information. In other words, a distinction is made between the *model* or *subject*, that carries the information, and the *views* or *observers*, that present that information in some format. As a consequence, when a change occurs, the *viewers* or *observers* have only to be notified to update their presentation. In effect, MVC or the *Observer* pattern can be regarded as a simple method for constraint propagation. An advantage is that unexpected updates can be easily dealt with.

The objects involved in realizing the *Observer* pattern are depicted in slide ???. The *subject* object must allow for *observers* to be attached and detached. Note that *observers* must also have a reference to the *subject*. In particular, concrete observers must know how to obtain information about the state of the *subject*, to be able to update their view. What the abstract *subject* and *observer* classes supply are the facilities for attachment and mechanisms for notification and updates. In the implementation of the *Observer* pattern there are a number of problems and tradeoffs that must be considered. For example, do we allow one *observer* to be attached to more than one *subject*? Do we allow for alternative update semantics, for example observer-pull instead of subject-push? Do we provide facilities for specifying aspects of interest, so that updates only need to concern those aspects? And finally, how do we guarantee mutual consistency between subjects and observers when we do allow for alternative update semantics?

## 1.4 Event-driven computation

Event-driven computation underlies many applications, ranging from graphical user interfaces to systems for discrete event simulation and business process

modeling. An important characteristic of event-driven computation is that control is relinquished to an environment that waits for events to occur. Handler function or handler objects are then invoked for an appropriate response. In this section we will look at the *Reactor* pattern that explains the interaction between objects and the environment. We will also look at an event system, in which the event types are defined by the application programmer. In this application, events are used to maintain global consistency, similar to the *Observer* pattern.

#### 1.4.1 The Reactor pattern

The *Reactor* pattern has been introduced in [Schmidt95] as a general architecture for event-driven systems. It explains how to register handlers for particular event types, and how to activate handlers when events occur, even when events come from multiple sources, in a single-threaded environment. In other words, the *reactor* allows for the combination of multiple event-loops, without introducing additional threads.

The abstract layout of the software architecture needed to realize the pattern is depicted in slide ???. The *reactor* environment must allow for binding *handlers* to particular types of *events*. In addition, it must be able to receive events, and select a handler to which the event can be dispatched.

Events may be organized in a hierarchy. There are two possible choices here. Either the topmost event class has a fat interface, containing all the methods that an event may ever need to support, or the topmost event class can be lean, so that additional methods need to be added by the subclasses of event. The first solution is chosen for *hush*, because in C++ it is not possible to load new classes dynamically. The latter solution is the way Java does it. In Java new event types can be added at the *reactor* level without recompiling the system. In the Java AWT and Swing libraries, handlers are called *Listeners*. Concrete handlers, derived from an abstract handler, must provide a method, such as `operate(Event)` that can be called by the *reactor* when the handler is selected after receiving an event.

The interaction between the application, its handlers, the *reactor* and the environment from which the events originate is depicted in slide ???. First, the *reactor* must be initialized, then one or more handlers can be registered, providing a binding for particular types of events. The *reactor* must then start to execute its eventloop. When it receives an event from the environment, it selects a handler and dispatches the event to that handler, by calling `operate(Event)`.

**Consequences** Modularity is one of the advantages of an event-driven software architecture. Handlers can be composed easily, since their invocation is controlled by the *reactor*. Another advantage is the decoupling of application-independent mechanisms from application-specific policies. In other words, handler objects need not be aware of how events are dispatched. This is the responsibility of the system or framework. The fact that control is handed over to the environment has, however, also some disadvantages. First of all, as experience with student assignments shows, it is difficult to learn in the beginning. But even when

mastered, applications may be hard to debug, since it is not always clear why a particular handler was invoked, and because it may be difficult to repeat the computation preceding the fault.

**Applicability** Some variant of the *reactor* pattern is used in Unix (X) Windows, (MS) Windows, and also GUI libraries such as Interviews, ET++ and *hush*. Another example is the Orbacus object request broker, that supports a *reactor* mode for server objects, which allows for receiving messages from multiple sources in a single thread. The Orbacus broker, however, also allows for multi-threaded servers.

### 1.4.2 Abstract event systems

To conclude this chapter about idioms and patterns, we will look at a somewhat more detailed example employing (user-defined) events to characterize and control the interaction between the objects. The example is taken from [Henderson93]. The *abstract system*, or repertoire of statements indicating the functionality of our application is depicted in slide ??.

First, we will define the functional behavior of the system (in this case a collection of thermometers that record and display temperature values, as characterized above). Then we will introduce the user interface classes, respectively to update the temperature value of a thermometer and to display its value. After that we define a concrete event class (derived from an abstract event class) for each of the possible kinds of interactions that may occur. Then, after installing the actual objects comprising the system, we will define the dependencies between (actual) events, so that we can guarantee that interactions with the user will not result in an inconsistent state.

**Functional behavior** A thermometer must provide the means to store a temperature value and allow for the changing and retrieving of this value. The temperature values are assumed to be stored in degrees Kelvin.

```
class thermometer {
    protected thermometer( float v ) { temp = v; }

    public void set(float v) { temp = v; }
    public float get() { return temp; }

    protected float temp;
};
```

*thermometer*

Since only derived classes can use the protected constructor, no direct instances of *thermometer* exist, so the class is abstract.

We will distinguish between two kinds of thermometers, measuring temperatures respectively in centigrade and fahrenheit.

```
class centigrade extends thermometer {

    public centigrade() { super(0); }
    public void set(float v) { temp = v + 273; }
    public float get() { return temp - 273; }
};
```

*centigrade*

The class *centigrade* redefines the methods *get* and *set* according to the measurement in centigrade, and in a similar way we may define the class *fahrenheit*.

```
class fahrenheit extends thermometer {

    public fahrenheit() { super(0); }
    public void set(float v) { temp = (v - 32) * 5/9 + 273; }
    public float get() { return temp * 9/5 + 32 - 273; }
};
```

*fahrenheit*

Both the thermometer realization classes take care of performing the conversions necessary to store and retrieve the absolute temperature value.

**User interface** We will define two simple interface classes, of which we omit the implementation details. First, we define the interface of the *displayer* class, needed to *put* values to the screen.

```
class displayer extends window {

    public displayer() { ... }
    public void put(String s) { ... }
    public void put(float f) { ... }
};
```

*displayer*

And secondly, we define a *prompter* class, which defines (in an abstract way) how we may get a value from the user (or some other component of the system).

```
class prompter extends window {

    public prompter(String text) { ... }
    public float get() { ... }
    public String gets() { ... }
};
```

*prompter*

Together, the classes *displayer* and *prompter* define a rudimentary interface which is sufficient to take care of many of the interactions between the user and the system.

**Events** To define the interactions with the user (and their possible consequences) we will employ *events*, that is instances of realizations of the abstract event class, defined below.

```

abstract class event {
    public void dependent(event e) { ... }
    public void process() { ... }
    public void operator(); // abstract method

    private event[] dep;
};

```

event

Since a simple event (for example, the modification of a value) may result in a series of events (needed to keep the system in a consistent state), an event object maintains a set of dependent events, which may be activated using the *process* method. Further, each class derived from *event* is assumed to define the application operator, that is the actual actions resulting from activating the event.

The first realization of the abstract event class is the *update* event class, which corresponds to retrieving a new temperature value from the user.

```

class update extends event {

    public update(thermometer th, prompter p) {
        _th = th; _p = p;
    }

    void operator()() {
        _th.set( _p.get() );
        process();
    }

    thermometer _th;
    prompter _p;
};

```

update

An update involves a thermometer and a prompter, which are stored when creating the update event object. Activating an update event instance results in retrieving a value from the prompter, setting the thermometer to this value and activating the dependent events.

In a similar way, we define the second realization of the abstract event class, the *show* event class, which corresponds to displaying the value of a thermometer.

```

class show extends event {

    public show(thermometer th, displayer d) {
        _th = th; _d = d;
    }

    public void operator() {
        _d.put( _th.get() );
        process();
    }
}

```

show

```

thermometer _th;
displayer _d;
};

```

Activating a show event instance results in retrieving a value from the thermometer, putting that value on display and activating the events associated with this event.

**The installation** The next step we must take is to install the application, that is to create the objects comprising the functional behavior of the system, the user interface objects and (finally) the various event objects.

```

thermometer c = new centigrade();
thermometer f = new fahrenheit();

displayer cd = new displayer("centigrade");
displayer fd = new displayer("fahrenheit");

prompter cp = new prompter("enter centigrade value");
prompter fp = new prompter("enter fahrenheit value");

show sc = new show(c,cd);
show sf = new show(f,fd);

update uc = new update(c,cp);
update uf = new update(f,fp);

```

Having created the objects, we are almost done. The most important and perhaps difficult part is to define the appropriate dependencies between the respective event objects.

```

uc.dependent(sc);
uc.dependent(sf);
uf.dependent(sc);
uf.dependent(sf);

```

As shown above, we declare the event of showing the value of the centigrade thermometer (and also of the fahrenheit thermometer) to be dependent upon the event of updating the value of the centigrade thermometer. And we repeat this declaration for the event of updating the value of the fahrenheit thermometer.

We may now allow the user the choice between updating the centigrade or fahrenheit thermometer temperature value, for example by inserting these events in a menu, as indicated below

```

menu.insert(uc);
menu.insert(uf);

```