4 Idioms

"A what?" he said.
"An S.E.P."
"An S...?"
"... E.P."
"And what's that?"

Douglas Adams, Life, the Universe and Everything

Idioms are low-level patterns specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them with the features of the given language.

In this chapter we provide an overview of the use of idioms, show how they can define a programming style, and show where you can find idioms. We refer mainly to other people's work instead of documenting our own idioms. We do however present the Counted Pointer idiom as a complete idiom description.

4.1 Introduction

Idioms represent low-level patterns. In contrast to design patterns, which address general structural principles, idioms describe how to solve implementation-specific problems in a programming language, such as memory management in C++. Idioms can also directly address the concrete implementation of a specific design pattern. We cannot therefore draw a clear line between design patterns and idioms. Idioms can address low-level problems related to the use of a language, such as naming program elements, source text formatting or choosing return values. Such idioms approach or overlap areas that are typically addressed by programming guidelines. To summarize, we can say that idioms demonstrate competent use of programming language features. Idioms can therefore also support the teaching of a programming language.

A programming style is characterized by the way language constructs are used to implement a solution, such as the kind of loop statements used, the naming of program elements, and even the formatting of the source code. Each of these separate aspects can be cast into an idiom, whenever implementation decisions lead to a specific programming style. A collection of such related idioms defines a programming style.

As with all patterns for software architecture, idioms ease communication among developers and speed up software development and maintenance. The collected idioms of your project teams form an intellectual asset of your company.

4.2 What Can Idioms Provide?

Learning a new programming language does not end after you have mastered its syntax. There are always many ways to solve a particular programming problem with a given language. Some might be considered better style or make better use of the available language features. You have to know and understand the little tricks and unspoken rules that will make you productive and your code of high quality.

A single idiom might help you to solve a recurring problem with the programming language you normally use. Examples of such problems are memory management, object creation, naming of methods, source code formatting for readability, efficient use of specific library components and so on.

There are several ways to acquire expertise in solving such problems. One is by reading programs developed by experienced programmers. This makes you think about their style and encourages you to try to reproduce it in your own code. This approach takes a long time, as trying to understand 'foreign' code is not always easy. If a set of idioms are available for you to learn, it is much easier to become productive in a new programming language, because the idioms can teach you how to use the features of a programming language effectively to solve a particular problem.

Because each idiom has a unique name, they provide a vehicle for communication among software developers. A team of experienced engineers who have been working together for some time might share experience by thinking in terms of their own idioms. It may be difficult for a newcomer to such a team to understand and learn these implicit idioms. It is therefore a good idea to make idioms and their use explicit—for example, try to document and name the idioms you use.

In contrast to many design patterns, idioms are less 'portable' between programming languages. For example, the design of Smalltalk's collection classes incorporates many idioms that are specific to the language. They depend on features not present in C++ such as garbage collection or meta-information. An early C++ class library, the NIHCL [GOP90], implemented collection classes for C++ programs by mimicking Smalltalk's collections. For example, every class that has objects stored in collections must inherit from the NIHCL root class `Object`. In addition, memory management relies completely on the programmer, which makes the NIHCL collections much harder to use than Smalltalk's collection classes. Modern C++ class libraries such as Generic++ [SNI94] abandon this approach and implement collection classes differently from NIHCL by using the C++ template mechanism. Such template collections can store any kind of data of a given type, even non-objects.

4.3 Idioms and Style

Experienced programmers apply patterns when doing their work, just as do other experts. A good program written by a single programmer will contain many applications of his set of patterns. Knowing the patterns a programmer uses makes understanding their programs a lot easier.

It may be difficult to follow a consistent style, however, even for an experienced programmer. If programmers who use different styles form a team, they should agree on a single coding style for their programs. For example, consider the following sections of C/C++ code, which both implement a string copy function for 'C-style' strings:

```
void strcopyKR(char *d, const char *s) {
    while (*d++=*s++);
}

void strcopyPascal(char d[], const char s[]){
    int i ;
    for (i = 0; s[i] != '\0'; i = i + 1)
    {
        d[i] = s[i];
    }
    d[i] = '\0'; /* always assign 0 character */
} /* END of strcopyPascal */
```

Both functions achieve the same result—they copy characters from string `s` to string `d` until a character with the value zero is reached. A compiler might even be able to create identical optimized machine code from both examples. The function `strcopyKR()` uses pointers as synonyms for array parameters, in the terse C style in the tradition of Kernighan and Ritchie [KR88]. The `strcopyPascal()` function might have been written by a programmer with a background in a language such as Pascal, where pointers are intended for use with linked data structures. Both implementations follow their own style. Which version you prefer, or what your own version would look like, depends on your experience, background, taste and many other factors. A program that uses a mixture of both styles might be much harder to understand and maintain than a program that uses one style consistently. It is a prerequisite that we can understand the

style of the program, such as the strange looking `while` loop in `strcopyKR()`.

Corporate style guides are one approach to achieving a consistent style throughout programs developed by teams. Unfortunately many of them use dictatorial rules such as 'all comments must start on a separate line'. This means that they are not in pattern form—they give solutions or rules without stating the problem. Another shortcoming of such style guides is that they seldom give concrete advice to a programmer about how to solve frequently-occurring coding problems.

We think that style guides that contain collected idioms work better. They not only give the rules, but also provide insight into the problem solved by a rule. They name the idioms and thus allow them to be communicated. For example, it is easier to say and memorize 'you should use an Intention Revealing Selector here' [Bec97] than 'apply rule §7-42 and change your method name accordingly'. However, not many such style guides exist yet. A further problem is that idioms from conflicting styles do not mix well if applied carelessly to a program.

Here is an example of a style guide idiom from Kent Beck's *Smalltalk Best Practice Patterns* [Bec97]:

Name Indented Control Flow

Problem How do you indent messages?

Solution Put zero or one argument messages on the same lines as their receiver.

```
foo isNil
2 + 3
a < b ifTrue:[...]
```

Put the keyword/argument pairs of messages with two or more keywords each on its own line, indented one tab.

```
a < b
    ifTrue:[...]
    ifFalse:[...]
```

❑

Different sets of idioms may be appropriate for different domains. For example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding. In some domains, such as real-

time systems, a more 'efficient' style that does not use dynamic binding is required. A single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains. A style guide cannot and should not cover a variety of styles.

A coherent set of idioms leads to a consistent style in your programs. Such a single style will speed up development, because you do not have to spend a lot of time thinking about the simple problems covered by your set of idioms, like how to format a block of code. In addition a consistent style also helps during program evolution or maintenance, because it makes programs a lot easier to understand.

4.4 Where Can You Find Idioms?

It is beyond the scope of this book to cover a programming style for a programming language—such styles and idioms could easily fill an entire book by themselves. We suggest that you look at any good language introduction to make a start on collecting a set of idioms to use. As an exercise in documenting your own patterns, you can try to rephrase the guidelines given in such books to correspond to a pattern template. This will help you to understand when to apply the rules, so that you can easily determine which problem a guideline solves.

Some design patterns that address programming problems in a more general way can also provide a source of idioms. If you look at these patterns from the perspective of a specific programming language, you can find embedded idioms. For example, the Singleton design pattern [GHJV95] provides two idioms specific to Smalltalk and C++:

Name Singleton (C++)

Problem You want to implement the Singleton design pattern [GHJV95] in C++, to ensure that exactly one instance of a class exists at run-time.

Solution Make the constructor of the class private. Declare a static member variable `theInstance` that refers to the single existing instance of the

class. Initialize this pointer to zero in the class implementation file. Define a public static member function getInstance() that returns the value of theInstance. The first time getInstance() is called, it will create the single instance with new and assign its address to theInstance.

Example

```
class Singleton {
static Singleton *theInstance;
Singleton();
public:
static Singleton *getInstance() {
        if (! theInstance)
            theInstance = new Singleton;
        return theInstance;
    }
};
//...
Singleton* Singleton::theInstance = 0;
```

❑

The corresponding Smalltalk version of Singleton solves the same problem, but the solution is different because Smalltalk's language concepts are completely distinct from C++:

Name Singleton (Smalltalk)

Problem You want to implement the Singleton design pattern [GHJV95] in Smalltalk, to ensure that exactly one instance of a class exists at run-time.

Solution Override the class method new to raise an error. Add a class variable TheInstance that holds the single instance. Implement a class method getInstance that returns TheInstance. The first time getInstance is called, it will create the single instance with super new and assign it to TheInstance.

Example

```
new
    self error: 'cannot create new object'

getInstance
    TheInstance isNil ifTrue: [TheInstance := super new].
    ^ TheInstance
```

❑

Idioms that form several different coding styles in C++ can be found for example in Coplien's *Advanced C++* [Cope92], Barton and Neckman's *Scientific and Engineering C++* [BN94] and Meyers' *Effective C++* [Mey92].

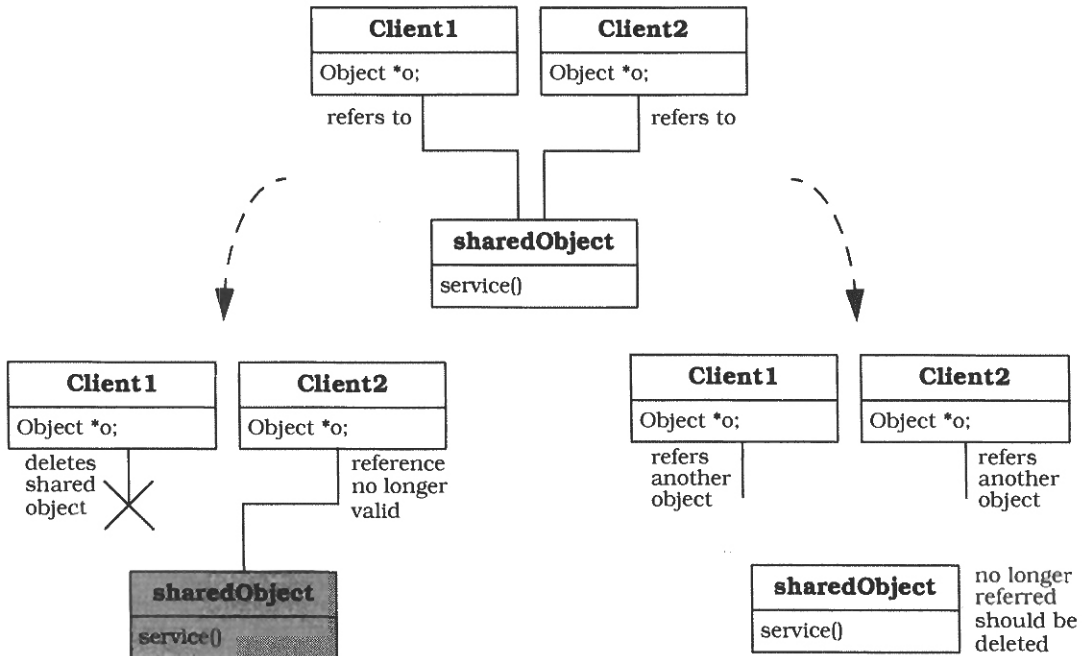
You can find a good collection of Smalltalk programming wisdom in the idioms presented in Kent Beck's columns in the *Smalltalk Report*. His collection of *Smalltalk Best Practice Patterns* is about to be published as a book [Bec96]. Beck defines a programming style with his coding patterns that is consistent with the Smalltalk class library, so you can treat this pattern collection as a Smalltalk style guide. Many of his patterns build on each other, so that in addition to being a style guide, his collection can be considered a pattern language.

You can also look at your own program code, or the code of your colleagues, read it and extract the patterns that have been used. You can use such 'pattern mining' to build a style guide for your programming language that becomes an intellectual asset of your team. By giving a name to each idiom, your style guide provides a language for communication between your developers. It can also provide a teaching aid for new developers who join your team.

Counted Pointer

The *Counted Pointer* idiom [Cope92] makes memory management of dynamically-allocated shared objects in C++ easier. It introduces a reference counter to a body class that is updated by handle objects. Clients access body class objects only through handles via the overloaded `operator->()`.

Example When using C++ for object-oriented development, memory management is an important issue. Whenever an object is shared by clients, each of which holds a reference to it, two situations exist that are likely to cause problems: a client may delete the object while another client still holds a reference to it, or all clients may 'forget' their references without the object being deleted.



Context Memory management of dynamically allocated instances of a class.

Problem In every object-oriented C++ program you have to pass objects as parameters of functions. It is typical to use pointers or references to objects as parameters. This allows you to exploit polymorphism. However, passing object references around freely can lead to the situations shown in the diagram above—you do not know if references are still valid, or even still needed.

One approach to the problems arising from the use of pointers and references is to avoid them completely and pass objects by value, as is normally done with integers. C++ allows you to create programs that do this, and the compiler will automatically destroy value objects that go out of scope.

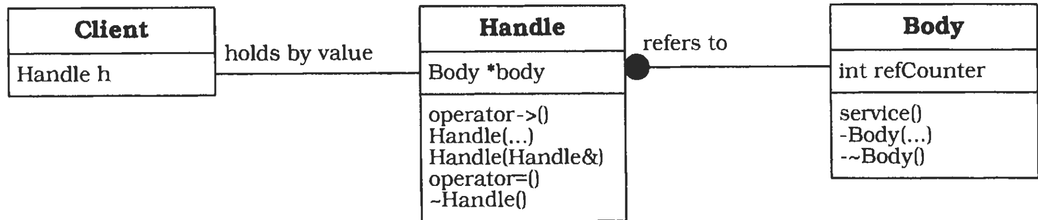
This solution does not work well for all kinds of program, however, for three reasons. Firstly, if the objects you pass by value are large, copying them each time they are used is expensive in run-time and memory consumption. Secondly, you might want to create dynamic structures of objects, such as trees or directed graphs, which is almost impossible to do in C++ using value objects alone. Lastly, you may want to share an object deliberately, for example by storing it in several collections.

If you have to deal with references or pointers to dynamically allocated objects of a class, you may need to address the following *forces*:

- Passing objects by value is inappropriate for a class.
- Several clients may need to share the same object.
- You want to avoid 'dangling' references—references to an object that has been deleted.
- If a shared object is no longer needed, it should be destroyed to conserve memory and release other resources it has acquired.
- Your solution should not require too much additional code within each client.

Solution The *Counted Pointer* idiom eases memory management of shared objects by introducing reference counting. The class of the shared objects, called *Body*, is extended with a reference counter. To keep track of references used, a second class *Handle* is the only class

allowed to hold references to Body objects. All Handle objects are passed by value throughout the program, and therefore are allocated and destroyed automatically. The Handle class takes care of the Body object's reference counter. By overloading `operator->()` in the Handle class, its objects can be used syntactically as if they were pointers to Body objects.



See the Variants section for a variation of this solution that applies when Body objects are only shared for performance reasons.

Implementation To implement the Counted Pointer idiom, carry out the following steps:

1 Make the constructors and destructor of the Body class private (or protected) to prohibit its uncontrolled instantiation and deletion.

2 Make the Handle class a friend to the Body class, and thus provide the Handle class with access to Body's internals.

3 Extend the Body class with a reference counter.

4 Add a single data member to the Handle class that points to the Body object.

5 Implement the Handle class' copy constructor and its assignment operator by copying the Body object pointer and incrementing the reference counter of the shared Body object. Implement the destructor of the Handle class to decrement the reference counter and to delete the Body object when the counter reaches zero.

6 Implement the arrow operator of the Handle class as follows:

```
Body * operator->() const { return body; }
```

and make it a public member function.

7 Extend the Handle class with one or several constructors that create the initial Body instance to which it refers. Each of these constructors initializes the reference counter to one.

Sample Code Applying the Counted Pointer idiom results in the following C++ code:

```
class Body {
public:
// methods providing the bodies functionality to the world
    void service() ;
    // further functionality...
private:
    friend class Handle;
    // parameters of constructor as required
    Body(/*...*/) { /* ... */ }
    ~Body() { /* ... */ }
    int refCounter;
};

class Handle {
public:
    // use Body's constructor parameters
    Handle(/*...*/) {
        body = new Body(/*...*/);
        body->refCounter = 1;
    }
    Handle(const Handle &h) {
        body = h.body;
        body->refCounter++;
    }
    Handle & operator=(const Handle &h) {
        h.body->refCounter++;
        if (--body->refCounter) <= 0)
            delete body;
        body= h.body;
        return *this;
    }
    ~Handle() {
        if (--body->refCounter <= 0)
            delete body;
    }
    Body* operator->() { return body; }
private:
    Body *body;
};
```

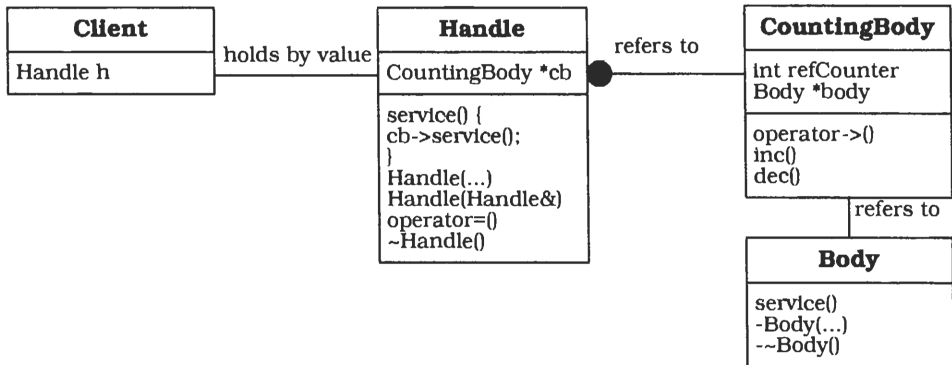
```
// example use of handles ...
Handle h(/* some parameter */);
// create a handle and also a new body instance
{   Handle g(h); // create just a new handle
    h->service(); g->service();
} // g goes out of scope and is automatically deleted

h->service(); // still possible
// after h goes out of scope the body instance is
// automatically deleted.
```
❑

Variants A common application of reference counting, similar to Counted Pointer, is used for performance improvement with large Body objects. [Cope92] names this variant the *Reference Counting Idiom* or *Counted Body* in [Cope94a]. In this variant a client has the illusion of using its own Body object, even if it is shared with other clients. Whenever an operation is likely to change the shared Body object, the Handle creates a new Body instance and uses this copy for all further processing. To achieve this functionality it is not sufficient to just overload `operator->()`. Instead, the interface of the Body class is duplicated by the Handle class. Each method in the Handle class delegates execution to the Body instance to which it refers. Methods that would change the Body object create a new copy of it if other clients share this Body object.

See Also Bjarne Stroustrup [Str91] discusses several ways of extending the Handle class. The Handle can be implemented as a template if the Body class, passed as a template parameter, cooperates with the Handle template class—for example, if the Body class provides the Handle class access to the reference counter.

The solution provided by the Counted Pointer idiom has the drawback that you need to change the Body class to introduce the reference counter. Coplien and Koenig give two ways to avoid this change.

James Coplien [Cope92] presents the Counted Pointer idiom and several variations. In cases where the Body class is not intended to have derived classes, it is possible to embed it in the Handle class. Another variation, shown in the diagram that follows, is to wrap existing classes with a reference counter class. This wrapper class then forms the Body class of the Counted Pointer idiom. This solution requires an additional level of indirection when clients access the Body object.



Andrew Koenig gives a further variation of the theme that allows you to add reference counting to classes without changing them [Koe95]. He defines a separate abstraction for use counts. Then the Handle holds two pointers: one to the body object, the other to the use-count object. The use-count class can be used to implement handles for a variety of body classes. The Handle objects of this solution require twice the space of the other Counted Pointer variants, but the access is as direct as with a change to the Body class.