1 A First Taste of Design by Contract

1.1 About This Chapter

This chapter provides a quick introduction to design by contract. In doing so, it

- Shows you how contracts can specify the required behavior of a class, and how contracts can check the code at runtime.
- Explains that contracts are built of assertions, which are used to express preconditions, postconditions, and invariants. (Later chapters develop a small set of principles and guidelines to help you write high-quality contracts.)
- Provides an example that is a simplified version of what could be a real software component—a customer manager—but you don't need any experience with components to understand the example.
- Presents some of the benefits of design by contract, a theme that is followed up in more detail in Chapter 8.
- Offers a first taste of design by contract. It'll probably raise many questions in your mind. We haven't attempted to answer them all in Chapter 1. We hope the rest of the book will answer many of them.

Throughout this book we use the Unified Modeling Language (UML) for diagrams that summarize classes and the programming language Eiffel for writing contracts and implementations. (Chapter 11 presents two examples in Java.)

This book does not teach UML, Eiffel, or Java. We just explain the bits we need as we go along. The bibliography lists a small selection of books, papers, and Web sites where you can get more information.

1.2 THE CUSTOMER MANAGER EXAMPLE

The customer manager component explored here is, at its heart, a dictionary (also known as a directory or a look-up table), which is a classic, and basic, data structure. When we begin to develop principles and guidelines for designing with contracts, we deliberately use clean, simple, basic examples so as not to detract from the main message. For instance, Chapter 3 develops a contract for the DICTIONARY class. However, lessons learned from simple examples can be applied to realistic problems, such as customer manager components. We look at more complex examples toward the end of the book.

Let's imagine you are planning to use a software component that will manage all the information about your organization's customers. (The term "component" has many meanings. Components that manage a data resource are only one kind of component.)

An installed customer manager component has exclusive control over all customer objects. Figure 1.1 is a UML-style diagram showing extracts from the CUSTOMER_MANAGER and CUSTOMER types.

There is an association between these types. The black diamond tells us that customer objects are exclusively owned by a customer manager object. The asterisk tells us that a customer manager owns zero, one, or more customer objects.

The box for CUSTOMER_MANAGER lists the features you can use if you are a client of a customer manager object. (A feature is a method you can call or a public attribute you can inspect.) These features allow you to:

- Find out how many customers the manager owns. (The *count* feature might be implemented as a read-only attribute, or it might be made accessible through a *get_count()* method, depending on the implementation language.)
- Ask whether a particular customer id is active.

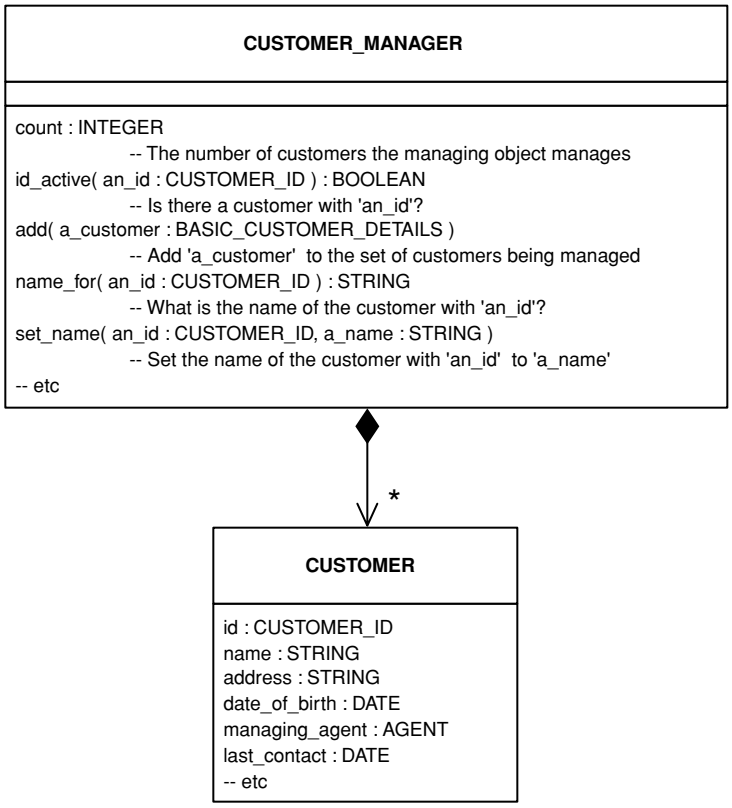

Figure 1.1 The CUSTOMER_MANAGER and CUSTOMER types

- Add a new customer to the customer manager component.
- Ask for the name of a customer whose id you know.
- Give an existing customer a new name.

The box for CUSTOMER lists some of the attributes of a customer (we do not need to explore the details of the methods).

The customer manager component has exclusive rights to modify customer objects, and clients of the component do not access customer objects directly. Instead, we provide some simpler types to allow clients to communicate with the

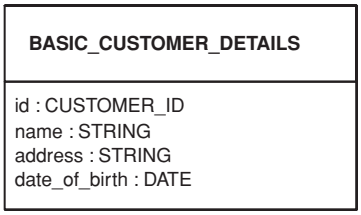

Figure 1.2 The BASIC_CUSTOMER_DETAILS type

customer manager component about customer objects. Figure 1.2 shows just one of these simpler types, the BASIC_CUSTOMER_DETAILS type. This type allows a client to pass in the basic information needed to create a new customer object, for instance.

The basic details of a customer are an id (given to him or her by the bank), a name, an address, and a date of birth.

In Eiffel, attributes can be public, but then they are read-only. So the design in Figure 1.2 is also the Eiffel programmer's view. If we were programming in Java, for example, we would program it with the attributes and methods shown in Figure 1.3 (in the code, we'd make the Java attributes private, but we haven't shown that detail on the UML-style diagram).

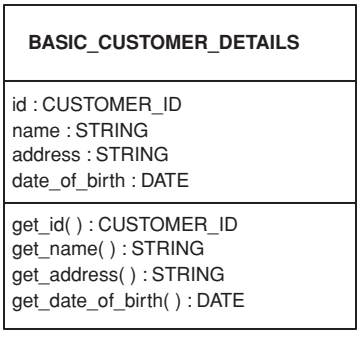

Figure 1.3 A Java design for the BASIC_CUSTOMER_DETAILS type

In Eiffel syntax, the name of a customer would be given by *a_customer.name*. In Java syntax, it would be given by *a_customer.get_name()*.

The customer manager exclusively owns the customer objects, and only the customer manager can change the information about a customer, so there are no set methods in the BASIC_CUSTOMER_DETAILS class. If you do succeed in changing the state of a basic customer details object, this has no effect on the corresponding customer object that the customer manager owns. You have to call a method on the customer manager component in order to change anything about a customer object.

1.3 SOME QUESTIONS

Look again at the list of methods that a client of a customer manager can call.

count : INTEGER
    -- The number of customers the managing object manages
id_active(an_id : CUSTOMER_ID) : BOOLEAN
    -- Is there a customer with 'an_id'?
add(a_customer : BASIC_CUSTOMER_DETAILS)
    -- Add 'a_customer' to the set of customers being managed
name_for(an_id : CUSTOMER_ID) : STRING
    -- What is the name of the customer with 'an_id'?
set_name(an_id : CUSTOMER_ID, a_name : STRING)
    -- Set the name of the customer with 'an_id' to 'a_name'

As you design your client of the customer manager, here are some questions you might want answered.

- How do I make an id active? By adding a customer with that id?
- If I add a customer whose details match an existing customer's basic details, what happens?
- What do I get if I ask for the name of a customer whose id is not active?
- What happens if I set the name for a customer whose id is not active?

In a component world, the person who developed the component might work for a company on the other side of the globe—and might be sleeping peacefully when you want your questions answered! The component must come with documentation that answers your questions. In the next section, we show how a contract can provide the answers to your questions. In later chapters, we'll see how to write good contracts.

1.4 A CONTRACT FOR CUSTOMER_MANAGER

The contract for the customer manager component contains smaller contracts, one for each of the features of the component. (Reminder: For now, a feature is a method or a public, read-only attribute. Sometimes we'll be lazy and talk of "calling a feature," even when we're actually inspecting an attribute.)

We'll explore these smaller contracts one by one. We hope you will concentrate on understanding each contract—don't worry yet about how to write a contract. And keep in mind that we are working on just a fragment of a component (so, for example, we will not discuss how the component is initialized).

By the end of this section, you'll know roughly what a contract is because you'll have seen an example: a contract is a collection of assertions (things that ought to be true) that describe precisely what each feature of the component does and doesn't do.

Adding a New Customer Here is the contract for the feature you use to add a customer to a customer manager component. It's followed by an explanation.

```
add( a_customer : BASIC_CUSTOMER_DETAILS )
        -- Add 'a_customer' to the set of customers
    require
        id_not_already_active:
            not id_active( a_customer.id )
    ensure
        count_increased:
            count = old count + 1
```

```
customer_id_now_active:
    id_active( a_customer.id )
```

The first line is a *signature*, which names the feature and lists its arguments (here we have just one argument, *a_customer*). The second line is a comment, which informally describes the feature.

The third line contains the keyword ***require***, which introduces a *precondition*. A precondition is a condition that a client must be sure is true; otherwise, it is not legal for the client to call the feature (we'll explore later what happens if a client makes an illegal call). On the next line, the identifier *id_not_already_active* is a tag, or label, chosen by the programmer to improve readability (and to help with debugging—as we will see later). The precondition's assertion is in the next line, and says:

```
not id_active( a_customer.id )
```

This asserts that, for it to be legal to call the *add* feature to add *a_customer*, it must *not* be true that the *id* of *a_customer* is an active id (we will soon know what makes an id active). The assertion is in Eiffel code so that it can be evaluated at runtime to check whether the client is keeping to its side of the contract.

The keyword ***ensure*** introduces a *postcondition*. A postcondition is a condition that should become true when the feature is executed (otherwise there is a bug in the code that implements the *add* feature).

In this example, the postcondition contains two assertions, each with its own tag. The first asserts that adding a customer increases the *count* by one. The expression ***old*** *count* is the value of the *count* before the feature was called. The = operator is the equality test, not the assignment operator. Asserting that the *count* now is what it was before, plus one, asserts that it has been increased by one.

The second assertion of the postcondition asserts that the *id* of the customer passed as argument is now active. This assertion, taken together with the precondition, says that you cannot add a customer if that customer's *id* is already active, and adding a customer makes that customer's *id* active.

Now we know the answers to the first two questions we asked earlier.

- How do I make an id active? By adding a customer with that id?

  Yes.

- If I add a customer whose details match an existing customer's details, what happens?

  You are not allowed to add a customer whose id equals the id of a customer already owned by the manager. If you keep to this rule, there are no further constraints on whether the name, address, and date of birth can be the same as those of an existing customer.

Setting the Name of a Customer Here is the contract for the feature you use to set the name of a customer.

```
set_name( an_id : CUSTOMER_ID; a_name : STRING )
        -- Set the name of the customer with 'an_id' to 'a_name'
    require
        id_active:
            id_active( an_id )
    ensure
        name_set:
            name_for( an_id ).is_equal( a_name )
```

The signature tells us that the feature takes a CUSTOMER_ID and a STRING as arguments.

The precondition (introduced by ***require***) makes it illegal to try to set the name of a customer with *an_id* that is not active. In other words, it is illegal to try to change the name of a customer that has not been added to the customer manager.

The postcondition (introduced by ***ensure***) asserts that if you now look up the name for the customer with *an_id,* you'll get back a name that is equal to

a_name, the second argument to the *set_name* feature. In other words, if you set the name of a customer, that's the name the customer now has.

A small detail: Because strings are objects, we assert that we get back a string that *is_equal* to the name we passed in as argument. One string *is_equal* to another if they both contain the same characters in the same order. If we'd used the = operator, we would be asserting that we actually get back the string object we passed in. That would be overspecification.

Asking for the Name of a Customer Here is the contract for the query feature *name_for*.

```
name_for( an_id : CUSTOMER_ID ) : STRING
        -- The name of the customer with 'an_id'
    require
        id_active:
            id_active( an_id )
```

This time, there is only a precondition, which states that you must not ask for the name for a customer whose id is not active. Why is there no postcondition? Because you are told the value of this query feature in the postcondition of the *set_name* feature, the feature that defines the value that *name_for* should have.

Now you know the answers to the third and fourth of your questions.

- What do I get if I ask for the name of a customer whose id is not active?

  It is illegal to ask for the name of a customer whose id is not active.

- What happens if I set the name for a customer whose id is not active?

  It is illegal to attempt to set the name of a customer whose id is not active.

1.5 THE STORY SO FAR

Here, in one place, is the contract on the CUSTOMER_MANAGER component, followed by some discussion. There is a new part to the contract, an invariant.

The invariant asserts that *count* is always zero or greater. An *invariant* property of a component is always true (more precisely, it is true whenever you can call a feature on the component).

```
component CUSTOMER_MANAGER

    count : INTEGER
            -- The number of customers

    id_active( an_id : CUSTOMER_ID ) : BOOLEAN
            -- Is there a customer with 'an_id'?

    add( a_customer : BASIC_CUSTOMER_DETAILS )
            -- Add 'a_customer' to the set of customers
        require
            id_not_already_active:
                not id_active( a_customer.id )
        ensure
            count_increased:
                count = old count + 1
            customer_id_now_active:
                id_active( a_customer.id )

    name_for( an_id : CUSTOMER_ID ) : STRING
            -- The name of the customer with 'an_id'
        require
            id_active:
                id_active( an_id )

    set_name( an_id : CUSTOMER_ID; a_name : STRING )
            -- Set the name of the customer with 'an_id' to 'a_name'
        require
            id_active:
                id_active( an_id )
        ensure
            name_set:
                name_for( an_id ).is_equal( a_name )
...
```

```
invariant
    count_never_negative:
        count >= 0
end
```

The contract is definitely unfinished. It still leaves many questions unanswered, but we hope you can see that a contract *can* answer many of the questions that a client programmer might ask.

The syntax of the component contract is that of the programming language Eiffel (except for the first line—Eiffel doesn't have special syntax to distinguish a component from a class).

The features *count* and *id_active* have neither preconditions nor postconditions. They don't have preconditions because it is always legal to call them. They don't have postconditions because their values are defined elsewhere. Specifically, the postcondition on the *add_customer* feature defines that *add_customer* both increments the *count* and makes *id_active* for the id of the added customer.

Notice how different parts of the component and its contract cannot be discussed in isolation. For example, the name returned by the *name_for* query feature is defined in the postcondition of the *set_name* feature. We cannot define the *name_for* some id without discussing the feature that sets it. Conversely, we cannot define what name is set for a customer by *set_name* without discussing the *name_for* query that tells us the name for the particular customer.

That's why we say that the component has a single contract (which, in turn, is made up of the individual contracts on the individual features).

1.6 RUNTIME CHECKING

So far, we have concentrated on the role that contracts play in specifying the behavior of a component. In this role, contracts are intended to be read by people. But contracts can also be checked at runtime. In this section, we'll see one example of what happens if you write careful contracts and then accidentally introduce bugs into the code. You might not be convinced by the example

because it is so simple. More powerful examples of the benefits appear in later chapters.

Suppose, for example, that we got confused about the precise meaning of the *add* feature and believed that we could use it to change the details associated with an existing customer. Suppose we called *add* with a customer details argument whose *id* was already active. In a programming environment that understands contracts, we would be told something like the following (we'll assume the CUSTOMER_MANAGER component has been implemented by a class of the same name):

Stopped in **object** *[0xE96978]*
Class: *CUSTOMER_MANAGER*
Feature: *add*
Problem: *Precondition violated*
Tag: *id_not_already_active*
Arguments:
a_customer: BASIC_CUSTOMER_DETAILS [0xE9697C]
Call stack:
CUSTOMER_MANAGER add
was called by *CUSTOMER_MANAGER_UIF change_customer*

This is the level of detail provided by the Eiffel development environment supplied by Interactive Software Engineering, Inc. Other environments provide similar detail (including other Eiffel environments and environments that add design by contract facilities to other programming languages, such as Java and C++). Working through this wealth of debugging information line by line, we can tell

1. That the application has stopped in some object (we could open the object with an object browser and examine its attributes).
2. That this object is of the class CUSTOMER_MANAGER.
3. That a problem arose when that class's *add* feature was called.
4. That the problem was that some part of the precondition on *add* was violated.
5. That if a precondition is violated, it means some client called the *add* feature when it was not legal to do so. Specifically, it was the part of the precondition with the *id_not_already_active* tag that was violated.

6. Which BASIC_CUSTOMER_DETAILS object was passed as an argument to the call.
7. The sequence of calls that led up to the problem: A *change_customer* feature in a *CUSTOMER_MANAGER_UIF* class (the user interface to the customer manager application) called the *add* feature in the CUSTOMER_MANAGER class.

Putting all this information together leads us to conclude that the *change_customer* feature in CUSTOMER_MANAGER_UIF class is the cause of the problem—it called *add* with a BASIC_CUSTOMER_DETAILS object whose id was already active, and the contract says that's illegal. In other words, the *change_customer* feature contains a bug, which we must find and fix.

Developers who make good use of contracts come to expect this amount of help when something goes wrong. They don't spend long hours hunting for the causes of a runtime error. They put their effort into writing contracts instead. And, in return, they get a second benefit, trustworthy documentation, which is the subject of the next section.

Just before we leave this section, though, we should explain that you might want contracts checked while you are developing an application, but you might not want production code to be slowed down by all this checking. In a programming environment that supports design by contract, you can turn contract-checking on and off. You can turn it on in some classes and off in others. And you can turn it on at different levels, such as checking only preconditions, rather than full checking of preconditions, postconditions, and invariants.

1.7 TRUSTWORTHY DOCUMENTATION

In the previous section, we saw that contracts can be checked at runtime, delivering valuable debugging information. We took an example of a faulty caller who broke a precondition. What if the component itself contains a bug? Then a postcondition (or the invariant) will evaluate to false, highlighting the problem and its cause.

For example, suppose that the developer of the component forgot to increment the *count* in the feature to *add* a new customer. During testing, the *add* feature is

called. Its postcondition is evaluated. And up comes a message, like the one in the previous section, telling the developer that the false postcondition was the one with the tag *count_increased* in the *add* feature in the CUSTOMER_MANAGER class.

If an application survives execution without any false preconditions, postconditions, or invariants, we can be sure that the code is doing what the contract says.

Now turn that around. By checking that the code is doing what the contract says, the runtime system is also checking that the documentation accurately says what the code does. If one assertion in the contract did not correctly describe what the code did, that assertion would evaluate to false on some test.

In other words, if you document your classes using contracts, you get documentation that everyone can trust to be telling the truth. There's a novelty!

Further, the documentation sets out clearly the rights and obligations of each party to the contract. In any one call to a feature, one object will be playing the role of client, or caller. The other will be playing the role of supplier.

A client calling a feature on a supplier has an *obligation* to fulfill the precondition. A client who calls a feature when its precondition is false has broken the client's side of the contract. The supplier's called feature has an *obligation* to terminate with the postcondition true. A feature that leaves its postcondition false has broken the supplier's side of the contract.

In return for these obligations, both parties obtain rights. The client has the *right* to expect that the called feature makes its postcondition true. If the postcondition is false, this is not the fault of the client. The supplier has the *right* to expect that the precondition is true. If the feature is called with the precondition false, this is not the fault of the supplier, which is then not obliged to meet its postcondition.

There is more on this theme in Chapter 8.

1.8 SUMMARY

Using a very simple example of (part of) a customer manager component, we've introduced the idea that a program can contain assertions. These assertions can be used to write preconditions, postconditions, and invariants.

A precondition specifies the circumstances under which it is valid to call a feature. A postcondition specifies the effect of calling a feature. An invariant specifies unchanging properties of the objects of a class.

Assertions can be checked at runtime to help us test and debug the implementation. As an important by-product of this checking, the documentation accurately describes what the code actually does.

Assertion-checking is built into Eiffel. For other languages such as Java and C++ you can obtain preprocessors that generate the code to turn your assertions into runtime checks, thereby getting the same benefits: runtime checking and trustworthy documentation. (In Chapter 11, we present examples that use the iContract tool developed by Reto Kramer, available from Trusted Systems.)

1.9 AN AIDE MEMOIRE

Contracts are "specs 'n' checks."

1.10 THINGS TO DO

In later chapters, we invite you to download (or rewrite) code and try using contracts for yourself, but this chapter did not present a complete example. However, we do have some suggestions for things you might do.

1. Start work on obtaining a programming environment in which you can try out contracts (download and install an Eiffel compiler, or seek a design by contract preprocessor for another language). This book makes a whole lot more sense if you play with contracts.

2. Choose an API (such as those you find in class libraries) and ask yourself what questions you cannot answer with certainty from the signatures and comments in the API. Where do you have to go for the answers? Would you like to get future libraries from a supplier who writes contracts? Would you pay extra for such libraries?
3. Browse the Web for information on design by contract. Quite a lot is out there. (See the bibliography for some Web sites that might help.)

We've deliberately built in lots of repetition of the concepts throughout the book. If you don't get an idea straight off, don't worry. It'll almost certainly come around again soon, and maybe its explanation will click then.