5.2 Built-in Contract Testing with Programming Languages

It is arguable whether artifacts in standard implementation notations such as Ada, C, C++, Pascal, or Java can be termed components. According to Szyperski's component definition [157, 158] that was established at the 1996 European Conference on Object-Oriented Programming (ECOOP'96), they are not. This defines a component as a unit of composition with contractually specified interfaces and context dependencies only, something that can be deployed independently and is subject to composition by a third party. The term "independently deployable" implies that such components will come in binary executable form, ideally with their own runtime environments if they are not supported by a platform. Implementations are not typically directly executable unless they are interpreted, such as Perl or Python. As said before, implementations always need this last transformation step to be independently deployable.

However, in a model-driven approach, as put forward in this book, an implementation is merely a section or a phase along the abstraction/concretization dimension as displayed in Fig. 5.1. It belongs to a transition between formats that humans can understand easily into formats that are easier for a machine to "understand" and process. The term component is related more to composition of individually solvable and controllable abstractions or building blocks. This terminology is motivated through a typical divide-and-conquer approach that splits a large problem into smaller and more manageable parts, so the term component is related more to the composition/decomposition dimension in Fig. 5.1. If we follow this philosophy, an abstract model (for example, one of the boxes in Fig. 5.1) may denote a component, and what it actually does. We can handle such a component in an abstract way, for example, we can perform some composition and incorporate it into an abstract component framework that is entirely defined in an abstract notation. So, whatever we can do with a concrete representation, i.e., at the code level, can be done in a more abstract representation, i.e., at the model level. Hence, under a model-driven approach all the properties of concrete executable components are meaningless because they are freed from the shackles of their concrete runtime environments. In my opinion, Szyperski's component definition does not explicitly separate between the two dimensions, composition and abstraction. In this respect, any artifact along the abstraction/concretization dimension may be regarded as a component as long as it is identified as a component in the composition/decomposition dimension. Typical implementation notations may therefore well be seen as complying with the component

philosophy, and we can also include typical non-object-oriented implementations, because at the abstract level they can be treated as any other object at that level. In other words, any object technology principle at a high abstraction level can be transformed into non-object technology artifacts on a lower level of abstraction. The following two sections look at how built-in contract testing may be realized through C, C++, and Java implementations.

5.2.1 Procedural Embodiment Under C

The C language belongs to the most widely used implementation technologies, and not only for very technical contexts or system programming for which it has been initially developed. C explicitly supports modular programming through the concept of source code files as modules or components and their separate compilation. It therefore also incorporates everything that is necessary for implementing information hiding, a basic principle of object technology [93]. The module concept manifests itself in a number of ways:

- Modules can be organized hierarchically. This adheres to the principles of composition and explicit dependencies or contracts between the hierarchically organized modules.
- Modules can be reused. This realizes the most rudimentary reuse principles and, in fact, follows Booch's component definition which sees a component as a logically cohesive, loosely coupled module that denotes a single abstraction [20].
- The two previous items lead to the notion of platform independency and abstraction. The module concept essentially hides and encapsulates underlying implementations that are platform-specific, such as system libraries, and other operations closer to the hardware.

C does not explicitly provide typical object-technology properties for implementing built-in contract testing, but since C++ can, and is usually implemented in C, the ideas behind built-in contract testing can well be adapted to C implementations. In C everything locally defined, i.e., through "static," will be accessible only through the procedures in the same scope or file. These can be seen as the attributes of an object in an object-oriented language or the data variables that the module encapsulates. The procedures that are comprised in the module can be seen as the methods of an object, or the module's external interface. The procedures collectively define the contract that the module is providing. The only difference with C++ objects is that their cohesiveness is determined through the data that the object encapsulates, while in a C module cohesiveness is determined through the functionality of the procedures (functional cohesiveness, or functional similarity).

Built-in Testing Interface

For built-in contract testing, we can implement a testing interface for each server module and a tester component for each client module. A testing inter-

face can be implemented either if a C module encapsulates internal data, such as static variables, or if it will be used to control a module's built-in assertion checking mechanisms, which in fact represent internal state information as well. In C, a testing interface for a module is a collection of additional procedures that are added to the existing functionality. These additional procedures are local to the file of the module and provide an additional access mechanism to that module. Such an organization, internal static variables, and procedures to access these variables, effectively realize an encapsulated entity with class-like properties. The only difference with C++ is that in C we can have only one instance of such a module per process, i.e., per "a.out" file. This limits the value of a testing interface in C because the procedural philosophy of the language adheres to the separation of data and functionality that in effect leads to stateless components. But we can nevertheless easily implement contract testing interfaces in C. For example, Fig. 5.4 illustrates the embodiment of the `testableVendingMachine` component model that is taken from Fig. 4.4 on page 137 (Chap. 4). Here, all additional ≪testing≫ artifacts are hardcoded as specified in the original model.

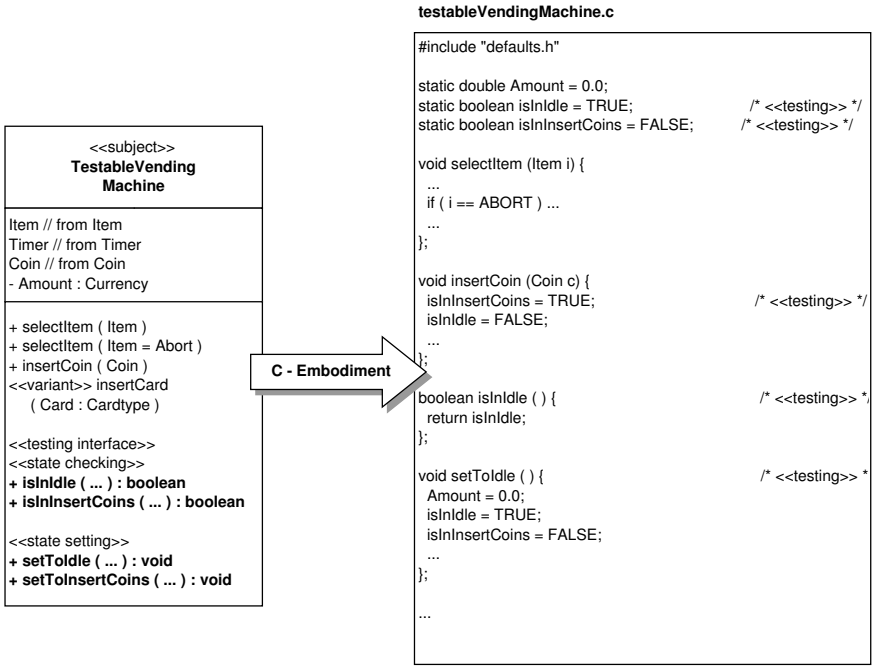

Fig. 5.4. Embodiment of the *testableVendingMachine* model in C

Built-in Tester Component

The built-in contract tester components at the client's side comprise test cases that simulate the client's access to the server module. There is no difference to other implementation technologies that are based on object technology. The value of built-in contract tester components is limited due to the limited number of feasible different usage profiles that a client can present. Since C modules do not typically encapsulate states that have an effect on the different operations, they are also not going to have any effect on the sequence or combination of operation invocations in the modules. All parameters that are provided to such a module are provided from outside its encapsulation boundary. It has no memory. Every operation of a module can therefore be seen as a stand-alone entity without dependencies on any other of the operations, and every operation can be tested in isolation to any other of the operations. Much of the complexity of testing object-oriented systems can be attributed to the interdependencies between individual class operations that are caused through the common data that they access. Under the procedural development paradigm, a unit test of a C module can therefore be regarded as a viable option even if the module will be used in a number of different applications. Other clients will not use such a module much more differently from the way it was initially intended to be used by the provider.

Built-in contract testing may offer only limited gain for the testing of component contracts in an application that is implemented in C. Where it may be much more successfully applied is at the interface between the C implementation and the underlying platform. One of the big challenges in C used to be, and still is, portability, or, in other words, a move of a C implementation from a development platform into a deployment platform. This particular case will be greatly supported by built-in contract tester components that are located at the transition between the user-level application and the underlying support platform that comprises not only the hardware but also the required support libraries and drivers. The built-in contract tester components can be invoked at deployment after the application has been brought to the new platform for the first time. They will contain tests that simulate the application's normal interactions with the underlying support software that is part of and installed on the platform. In this instance, built-in contract testing will alleviate the efforts of assessing whether an application will function properly on a particular platform, and it additionally points out the location of failure. This simplifies problem identification.

The C language lacks proper support for typical object technology properties that are advantageous for the implementation of built-in contract testing, such as dynamic assignment of components and an extension mechanism. This lack of basis technology prohibits dynamic assignment of tester components during runtime as well as dynamic instantiation of testable and non-testable components. In C most built-in testing artifacts will have to be hardcoded, so that they are available at all times in binary representation. This is mainly

the case for the built-in testing interface, since C does not provide extension except through conditional preprocessing at the implementation level. The built-in tester components can be made dynamic to a certain extent, through the use of function pointers, though this is quite awkward compared with the capabilities of modern object-oriented languages and nobody will probably bother to use that. These limitations therefore require a well planned application of built-in contract testing technology in procedural languages such as C. In the next section we will have a look at how modern object-oriented languages support the use of built-in testing technology.

5.2.2 Object-Oriented Embodiment Under C++ and Java

C++ and Java are the most commonly and successfully used embodiment technologies for object-oriented implementations. They both represent success stories with respect to industrial penetration, though it is quite clear that Java is much more modern and represents a much cleaner and simpler way of realizing object-oriented programming than the C successor. C++ is halfway between traditional procedural programming à la C and object-oriented development as it is understood by Java. It is a superset of C, so it is still equipped with the procedural capabilities of its predecessor. Java lacks some of the typical C/C++ features that some programmers regard as cure and others as curse, for example, preprocessor instructions, direct memory access, multiple inheritance of implementation, implicit type conversion, and typical C legacy such as "goto," unions, global variables, and the like. Apart from a number of differences between the two programming languages that are thoroughly discussed in the literature (e.g., [110]), they both come equipped with the right means for implementing built-in contract testing in the way it is described in Chap. 4.

Built-in Testing Interface

Figure 5.5 shows a prototypical C++ implementation of the `TestableVending Machine` component. Figure 5.6 shows a prototypical Java implementation of the `TestableVendingMachine` component. In these implementation examples I have suppressed the variant feature that the model specifies, but I will come to that later on. Both implementations can be derived directly from their respective UML model, because both languages readily support UML's class concept; or, if we look at it the other way, because the programming languages are much older than the modeling notation, UML provides support for specifying classes how the two programming languages view it. For such simple specifications the mapping between model and code is straightforward. Java provides through the concepts of abstract classes and interfaces a much more advanced way of defining prototypes than C++. In my opinion, this leads to a clearer and simpler design in Java, although this is arguable.

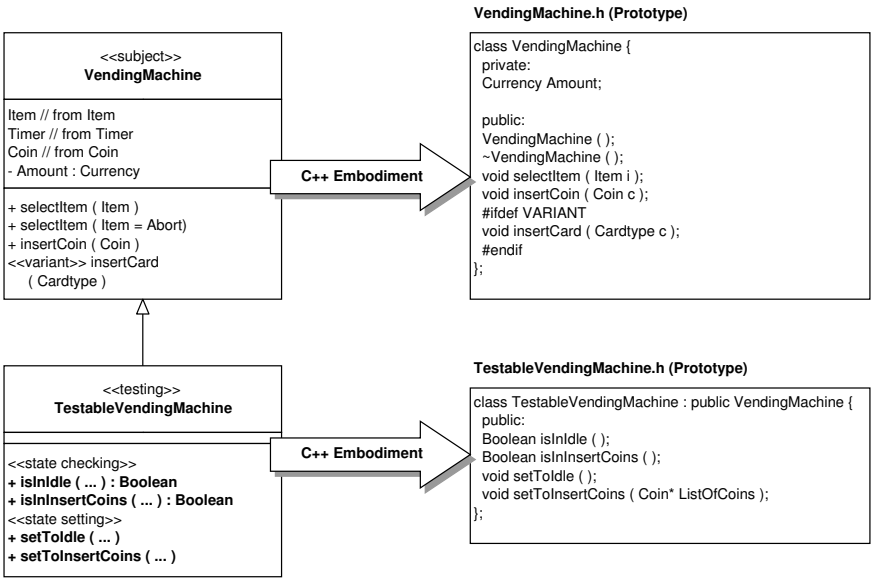

Fig. 5.5. Embodiment of the *testableVendingMachine* model in C++

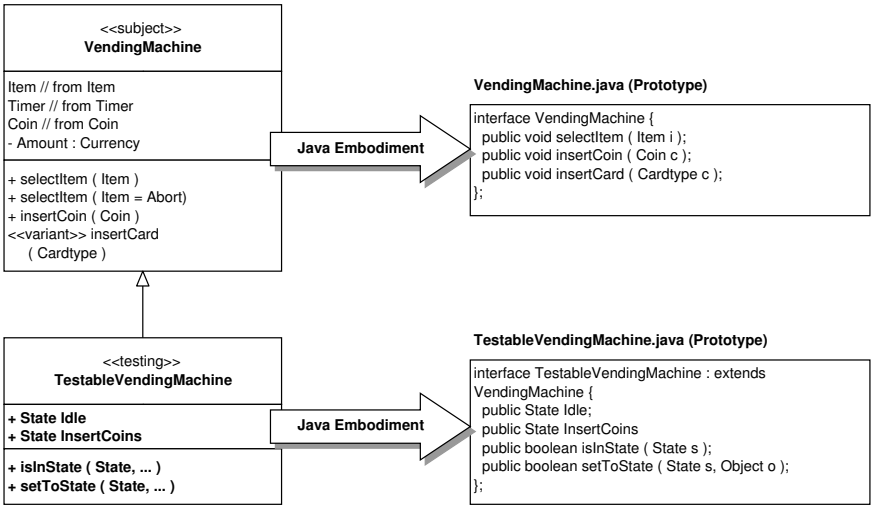

Fig. 5.6. Embodiment of the *testableVendingMachine* model in Java

In the C example I have incorporated the variation point for dealing with the optional functionality of a vending machine that also supports credit card billing. This can be done in the same way for C++, because the language provides preprocessor instructions. For the Java example it does not work like that because Java does not provide a preprocessor that incorporates functionality according to optional design decisions. If we would like to implement this in Java, or implement it differently in C++, we have to change the model, and this represents a refinement and a translation step. Without the two steps we would implicitly assume design decisions that will never appear anywhere in the documentation of the system. For such a simple system as that we are dealing with, this might be okay. For larger systems, however, it is essential that such design decisions be documented well. I will briefly explain how the refinement of the models and the translation into Java code can be carried out.

The existing model in Fig. 5.5 maps only directly to C or C++. For a different Java implementation we will have to refine the model according to the limitations of Java in the way I have explained in Chap. 2. This refined Java implementation-specific model is displayed in Fig. 5.7. The variation point is realized as an extension to the original model, `ExtendedVendingMachine`, which can be extended through the testing interface, turning it into a `testableVendingMachine`. The translation step for this example is depicted in Fig. 5.8.

Built-in Tester Component

Table 5.1 repeats the specification of the `VendingMachineTester` component from Chap. 3. Initially, this test may be performed at the user interface of the vending machine, and it would be carried out through a real human tester who is performing the task described in the table. Alternatively, we can have the `VendingMachineTester` component access the `VendingMachine` component directly. So, the test will actually be applied like a simulation of a real user who performs transactions on the vending machine. Embodiment is essentially concerned with turning the test specification in Table 5.1 into source code, for example, in Java. This is laid out for the first test case in Table 5.1 in the following Java source code example. A prerequisite for executing this test is that the precondition, `ItemX == Empty` or `ItemX != Empty`, holds. The preconditions can be set through a stub that emulates the `Dispenser` component, or they can be set through the testing interface of that component if no stubs are used. This requires that the dispenser and the display component also provide testing interfaces according to the built-in contract testing paradigm through which a client tester can also set and check initial and final states. The organization of this testing system is represented by the containment hierarchy in Fig. 5.9.

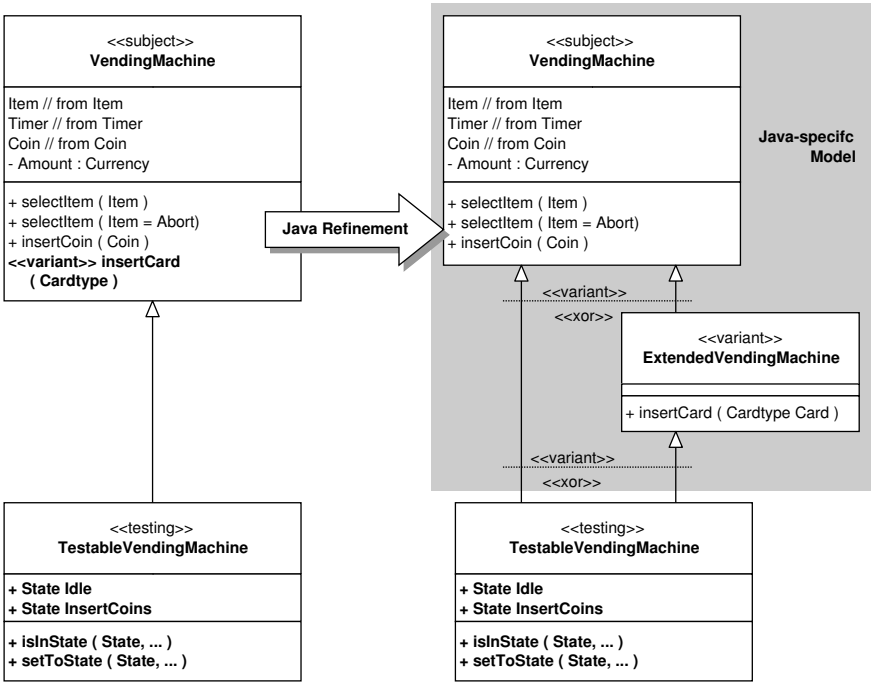

Fig. 5.7. Refinement of the *VendingMachine* model for a Java implementation

The following source code example represents a feasible Java implementation for the `VendingMachineTester` component specified in Fig. 5.9 and in Table 5.1. Here, I give only an excerpt of the full tester component:

```
class VendingMachineTester {

  // configuration interface
  private object Dispenser;          // test bed
  private object Display;            // test bed
  private object TVM;                // tested component

  public void setDispenser (object testableDispenser) {
    Dipenser = testableDispenser;
  }

  public void setDisplay (object testableDisplay) {
    Display = testableDisplay;
  }

  void setTestableVendingMachine (object tvm) {
```

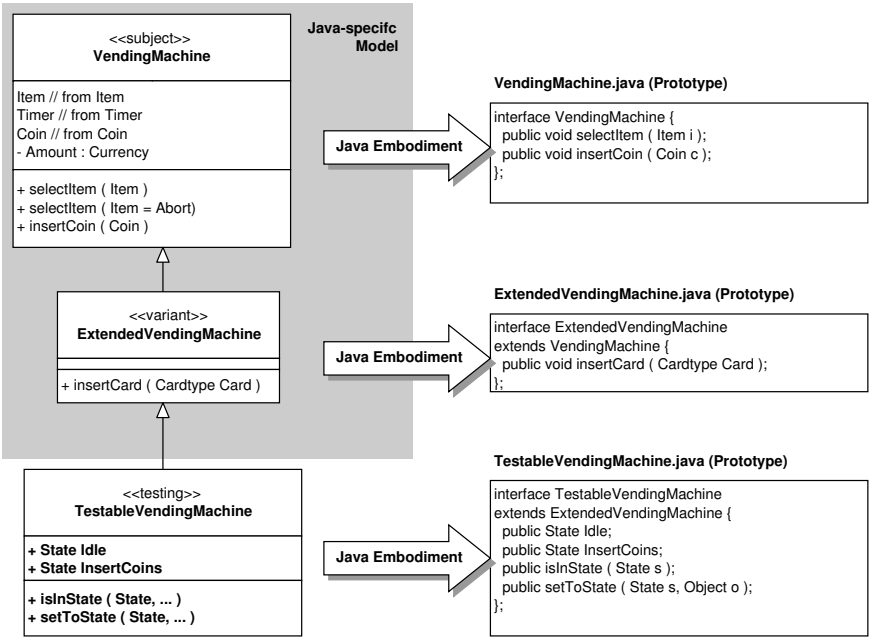

Fig. 5.8. Translation of the Java-specific model of the *testableVendingMachine* into a Java implementation

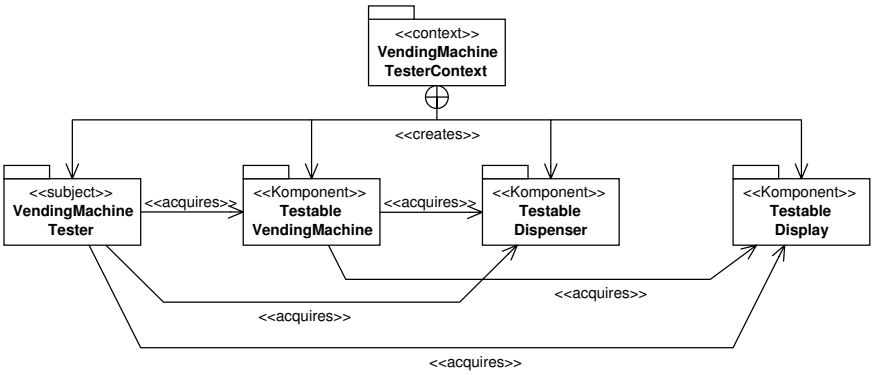

Fig. 5.9. Containment hierarchy of a testing system for the *VendingMachine* component

```
  TVM = tvm;
}

// start test

public boolean startTest () {
  if (false == startTest11 ()) return false;
  if (false == startTest12 ()) return false;
  if (false == startTest13 ()) return false;
  if (false == startTest14 ()) return false;
  if (false == startTest15 ()) return false;
  if (false == startTest21 ()) return false;
  if (false == startTest22 ()) return false;
  if (false == startTest23 ()) return false;
  ...
  return true;
}

// test cases

public boolean startTest11 () {   // TEST 1.1
  Dispenser.setTo (Item1, empty); // set precondition
  TVM.setTo(idle);                // set init state
  try {                           // expect exception
    SelectItem (Item1);           // call transaction
  } catch (DispenserItemEmptyException e) {
    if (TVM.isIn(idle) && Display.isIn(Empty))
      return true;                // check final state
    else                          // and postcondition
      return false;
  }
  return false;
}

public boolean startTest12 () {   // TEST 1.2
  Dispenser.setTo (Item2, empty); // set precondition
  TVM.setTo(idle);                // set init state
  try {                           // expect exception
    SelectItem (Item2);           // call transaction
  } catch (DispenserItemEmptyException e) {
    if (TVM.isIn(idle) && Display.isIn(Empty))
      return true;                // check final sate
    else                          // and postcondition
      return false;
  }
```

```
    return false;
  }

  public boolean startTest13 () {   // TEST 1.3
    Dispenser.setTo (Item2, empty); // set precondition
    TVM.setTo(idle);                // set init state
    try {                           // expect exception
      SelectItem (Item2);           // call transaction
    } catch (DispenserItemEmptyException e) {
      if (TVM.isIn(idle) && Display.isIn(Empty))
        return true;                // check final state
      else                          // and postcondition
        return false;
    }
    return false;
  }
  ...
  public boolean startTest31 () {    // TEST 3.1
    Dispenser.setTo(Item1, notEmpty);// set precondition
    TVM.setTo(idle);                 // set init state
    TVM.insertCoin (0.1);            // call transaction
    if (TVM.isIn(insertCoins) &&     // check final state
        Display.isIn(0.1))           // and postcondition
      return true;
    else
      return false;
  }
  ...
}
```

If our implementation technology is restricted to a single programming language environment, we are actually done with the embodiment step when we have implemented the testing interface and the tester component according to each identified component contract in our component framework. The remaining work is to then integrate our individual components according to the mechanisms of the programming language.

Devising a source code implementation is always the first step if our deployment environment is a component platform or some middleware. If we target a component platform as the final embodiment infrastructure, we have to add platform-specific code to our source code development that is based on a single language. The component platform provides a binding to a particular language. This binding is similar to invoking library operations that the language provides. I will give some more details on how built-in contract testing may be realized on these component platforms in the subsequent sections.

Table 5.1. Behavioral tests for the *VendingMachine* component

No	Initial State	Precondition	Transition	Postcondition	Final State
1.1	idle	Item1 ==Empty	SelectItem (Item1)	Display (Empty)	idle
1.2	idle	Item2 ==Empty	SelectItem (Item2)	Display (Empty)	idle
1.3	idle	Item3 ==Empty	SelectItem (Item3)	Display (Empty)	idle
1.4	idle	Item4 ==Empty	SelectItem (Item4)	Display (Empty)	idle
1.5	idle	Item5 ==Empty	SelectItem (Item5)	Display (Empty)	idle
...	...	...	...	...	...
2.1	idle	Item1 !=Empty	SelectItem (Item1)	Display (Item1.Price)	idle
2.2	idle	Item2 !=Empty	SelectItem (Item2)	Display (Item2.Price)	idle
2.3	idle	Item3 !=Empty	SelectItem (Item3)	Display (Item3.Price)	idle
2.4	idle	Item4 !=Empty	SelectItem (Item4)	Display (Item4.Price)	idle
2.5	idle	Item5 !=Empty	SelectItem (Item5)	Display (Item5.Price)	idle
...	...	...	...	...	...
3.1	idle		InsertCoin (10ct)	Display (0.10)	insert Coins
3.2	idle		InsertCoin (20ct)	Display (0.20)	insert Coins
3.3	idle		InsertCoin (50ct)	Display (0.50)	insert Coins
3.4	idle		InsertCoin (1EUR)	Display (1.00)	insert Coins
3.5	idle		InsertCoin (2EUR)	Display (2.00)	insert Coins
4.1			Perform tests 6.1 to 6.3		
4.2	insertCoins		abort ()	CashUnit.dispense () == 0.80EUR	idle
5.1			Perform tests 6.1 to 6.3 and wait for some time		
5.2	insertCoins		Timeout ()	CashUnit.dispense () == 0.80EUR	idle
6.1			Perform test 3.1		
6.2	insertCoins		InsertCoin (20ct)	Display(0.30)	insert Coins
6.3	insertCoins		InsertCoin (50ct)	Display(0.80)	insert Coins
7.1			Perform test 3.1		
7.2	insertCoins	0.10 < Item1.Price	SelectItem (Item1)	Display(0.10)	insert Coins
...	...	...	...	...	...

Before that, we will have a brief look at how third-party components for which we do not own the source code can be augmented with built-in contract testing functionality. In the previous paragraphs I have concentrated mainly on how we can add testability features to our own classes for which the code is readily available. In the next paragraphs I will introduce a way to augment existing third-party Java classes with testing interfaces according to the built-in contract testing philosophy. This can be done through the BIT/J Library.

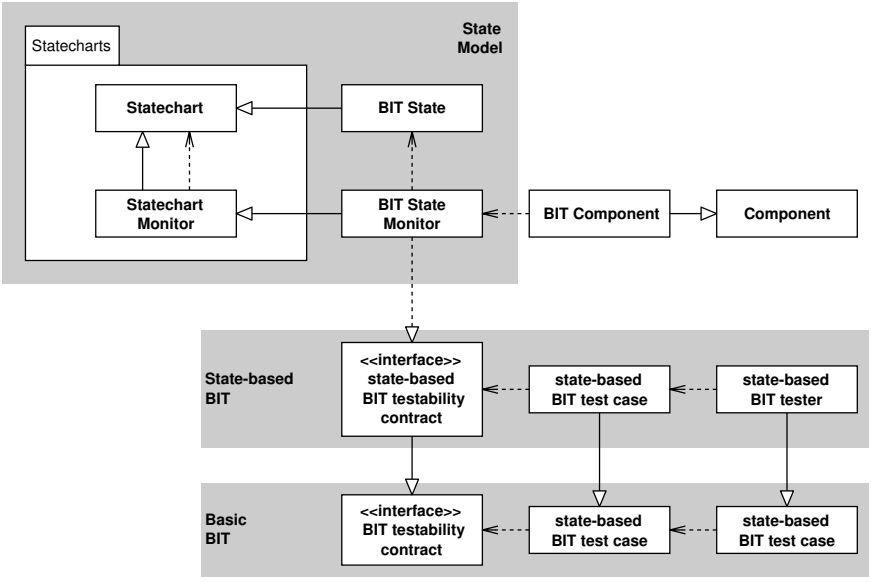

Fig. 5.10. Structural model of the BIT/J Library from the University of Pau, Laboratory of Computer Science (LIUPPA). BIT stands for Built-In Testing

BIT/J Library Java Support Framework for Built-in Contract Testing

Commercial third-party (off-the-shelf) components (COTS) cannot typically be augmented with additional built-in contract testing interfaces that provide a client of a server component with introspection capabilities for improved testability and observability [74]. Unless COTS vendors follow the built-in contract testing philosophy and incorporate testing interfaces into their products right from the beginning, such components can be tested only through their normal provided interfaces. Basically, this comes down to the traditional way of testing objects or components.

However, modern object-oriented implementation technologies such as Java do provide mechanisms that enable internal access to an encapsulated

third-party component at a binary level. The BIT/J Library and tool suite developed by members of the Laboratory of Computer Science at the University of Pau, France (LIUPPA), is one such instance that is capable of implementing external access to existing third-party Java components. It is a free tool that has been developed as part of the Component+ project [38], and it is available through the LIUPPA Web site [13]. The BIT/J Library is based on the idea of incorporating the behavioral model of a third-party component, as it is defined through the specification of the component, directly into that component. It uses mainly the Java reflection mechanism to gain access to and retrieve internal information from a COTS component. Figure 5.10 displays the structural model of the BIT/J Library. Contract testing based on this library is initially concerned only with `BIT testability contract`, `BIT test case`, and `BIT tester` indicated through the shaded box labeled "Basic BIT" in Fig. 5.10. These are the three most fundamental extensions that any testing environment using BIT/J will require. The `BIT testability contract` represents the initial testing interface that copes with the assessment of results, execution environment, and faults. It is fully specified in [7]. The `BIT tester` comprises `BIT test case`s that access this interface for retrieving testability information from a component. State-based testing according to a component's behavioral model is added through their respective state-based versions indicated through the shaded box that is labeled "State-based BIT" in Fig. 5.10. These concepts add state-based testing interfaces as they are required for state information setup and retrieving according to the definition of the built-in contract testing technology, and an execution environment for the specified state model of a component, the `BIT state monitor`. The state model implementation is based on Harel's statecharts formalism [85] that is also adopted by the OMG in UML state diagrams. The statechart runtime environment is added through classes in the shaded box called "State Model" in Fig. 5.10. BIT/J essentially adds an executable state machine plus a number of access interfaces to a Java component. Such an augmented component is termed `BIT component` in Fig. 5.10. A more detailed specification of BIT/J is available through [7, 13]. So far, this type of built-in contract testing support library is available only for Java components. Other implementation technologies do not yet provide the proper support for realizing similar access mechanisms.

http://www.springer.com/978-3-540-20864-8

Component-Based Software Testing with UML
Gross, H.-G.
2005, XVIII, 316 p., Hardcover
ISBN: 978-3-540-20864-8