
Object-Oriented Programming

Object-oriented programming is a set of techniques for creating objects and assembling them into a system that provides a desired functionality. An object is a software model composed of data and operations. An object's operations are used to manipulate its data. Objects may model things such as queues, stacks, windows, or circuit components. When assembled into a system, the objects function as delegators, devices, or models. Messages are passed between the objects to produce the desired results.

The eventual goal of object-oriented programming is to reduce the cost of software development by creating reusable and maintainable code. This is achieved by using three features of object-oriented programming: encapsulation, polymorphism, and inheritance. Encapsulation consists of data abstraction and data hiding. Data abstraction is a process of isolating attributes or qualities of something into an object model. With data hiding an object may contain its own private data and methods, which it uses to perform its operations. By using polymorphism, an object's operation may respond to many different types of data (e.g., graphical and textual). Finally, using inheritance, an object can inherit attributes from other objects. The object may then only need to add the attributes, methods, and data structures that make it different from the object from which it inherited its basic characteristics.

For the design of the object testing framework described in the accompanying article, we used an object-oriented software design methodology called object modeling technique (OMT). This methodology provides a collection of techniques and notation for designing an object-oriented application.

One important aspect of object-oriented design, or any software design, is deciding on who (i.e., module or object) is responsible for doing what. A technique provided in OMT involves using an index card to represent object classes. These cards are called CRC (class, responsibility, and collaboration) cards. The information on one of these cards includes the name of the class being described, a description of the problem the class is supposed to solve, and a list of other classes that provide services needed by the class being defined.

reports back to the host daemon the success or failure of a test start.

The cleaner upper cleans up after the tests have completed. This may include removing temporary files, removing test executables, and so on.

Process Management Subsystem

The two main objects in the process management subsystem are the process controller and `TestEnvironment` objects. The process controller has the overall responsibility to monitor all test-related processes on the SUT. It can register or unregister processes, kill processes, and report process status back to the host daemon.

The `TestEnvironment` class provides the test developer with an application programming interface to the OTE. It provides methods for aborting tests, logging test data and results, checking for exceptions, getting environment variables, and so on. The test developer gets access to these methods through the base `TestCase` class, which has an association with the `TestEnvironment` class.

Creating a test involves writing a class that inherits from either the client `TestCase` or server `TestCase` base classes. The initialization and setup functionality for the test would be included in the test's constructor. The cleanup required when the test is done is included in the destructor. Finally, an implementation for the inherited `run_body()` method is included, which is the test executable that runs the test. The

OTE API is made available through the pointer to the `TestEnvironment` class provided by the base `TestCase` class.

Implementation Approach

Once the design was complete, an initial investigation was made to find an existing system that matched the characteristics of the design. When no system was found, an analysis was done to determine the cost of implementing the new infrastructure.

It quickly became obvious that the transition to the new infrastructure would have to be gradual since we did not want to impact the HP ORB Plus product release cycle. The flexibility provided by an object-oriented system enables gradual migration and evolution through encapsulation, inheritance, and polymorphism. Tests could be isolated from the infrastructure so that new tests could be developed and evolved without modification as the infrastructure evolved. This flexibility fit nicely with the realization that the time to replace the existing infrastructure exceeded an average product life cycle.

Object-oriented encapsulation provided another advantage. Once some basic changes were made to the existing test infrastructure and tests had been converted to the new object-oriented programming model, the existing test infrastructure could be used to simulate some aspects of the new infrastructure. This allowed our system testing efforts to benefit immediately from the features of the new test system.

The development of the current version of the object testing framework has taken place in two steps, which have spanned three releases of the HP ORB Plus product. At each step we have continued to apply the same design principles. This work is summarized in the following sections.

First Step. For the first step, the goal was to consolidate the best practices of three existing test infrastructures into a single infrastructure that simulated as much of the major functionality of the OTE as possible. So as not to impact the ongoing HP ORB Plus software releases, another goal was to minimize changes to existing test code. This resulted in an infrastructure that consisted of a layer of shell scripts on top of two existing test harness tools. This significantly reduced the effort needed to set up, administer, and update the network of systems that were used to system test the HP ORB Plus product, while the tests continued to use existing APIs. It also confirmed that our design was indeed trying to solve the right problems.

Second Step. For this step the goal was to deploy the test developer's API to the OTE. The result was the implementation of the C++ `TestEnvironment` and `TestCase` classes described above.

Additional classes were designed to connect the `TestEnvironment` and `TestCase` classes to the existing infrastructure, but their existence is hidden from the test developer. This provides a stable API without limiting future enhancements to the infrastructure. Once the new infrastructure was deployed, we focused on porting existing tests to it.

This framework has resulted in minimal changes to existing tests and maximum increase in functionality for the tests. Most of the work simply involved taking existing code and