

DiPSUnit: an Extension of the JUnit Test Framework for DiPS

Sam Michiels

Dirk Walravens

Nico Janssens

Pierre Verbaeten

Report CW 333, February 2002



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

DiPSUnit: an Extension of the JUnit Test Framework for DiPS

Sam Michiels
Dirk Walravens
Nico Janssens
Pierre Verbaeten

Report CW 333, February 2002

Department of Computer Science, K.U.Leuven

Abstract

Testing system software (such as protocol stacks or file systems) often is a tedious and error-prone process. The reason for this is that such software is very complex and often not designed to be tested. This paper describes DiPSUnit, an extension of JUnit which allows to test (fine-grained as well as composed) units in a uniform way. Although non-trivial test support is provided, using DiPSUnit keeps testing simple and intuitive.

Keywords : Unit testing, framework, component-oriented software .

DiPSUnit: an Extension of the JUnit Test Framework for DiPS

Sam Michiels, Dirk Walravens, Nico Janssens, Pierre Verbaeten
DistriNet, Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Sam.Michiels,Dirk.Walravens}@cs.kuleuven.ac.be

February 2002

Abstract

Testing system software (such as protocol stacks or file systems) often is a tedious and error-prone process. The reason for this is that such software is very complex and often not designed to be tested. This paper describes DiPSUnit, an extension of JUnit which allows to test (fine-grained as well as composed) units in a uniform way. Although non-trivial test support is provided, using DiPSUnit keeps testing simple and intuitive.

1 Introduction

Testing system software, such as a protocol stack or a file system, is a complex, tedious and error-prone task. The basic problem is that, for performance reasons, system software is often designed as a monolithic block of multi-threaded software. This prevents such software from being tested properly for two reasons.

First, it is very difficult to isolate the basic building blocks (units) in a monolithic piece of software. Even if these units were designed as separate entities, they are not necessarily independent (stand-alone): any unit can create references to other units making them dependent on each other. This makes it impossible to replace a referenced unit without changing any code. This is typically the case for a unit that is tested in isolation. In this case, stubs [2] are often used as a replacement for units that are referenced from within the tested unit. However, to reduce the risk of introducing errors during acceptance testing (when stubs are replaced by the real units), it is essential that the replacement of stubs by the real functionality is transparent.

This research has been carried out in order of Alcatel Bell with financial support of the Flemish institute for the advancement of scientific-technological research in the industry (IWT SCAN # 010319)

The second problem is the concurrency code (such as semaphores), which is introduced in such multi-threaded system software. Most of the time, there is no control mechanism available to manage the threads in the system. Also, concurrency code is often spread out over the functional code. This not only complicates development but also testing of the code, because different aspects (functionality and concurrency) are mixed. It would be an advantage to be able to focus testing on each of those aspects separately.

We present the DiPSUnit test framework, which is an extension of JUnit [5]. It is developed specifically for testing DiPS (Distrinet Protocol Stack) [8] units. DiPS is a framework in the domain of protocol stacks and file systems which offers a number of software engineering characteristics that are very supportive for testing. DiPS is based on stand-alone units which anonymously communicate with each other. Units however can exchange control information, but only via events. DiPSUnit offers a uniform way of testing DiPS units because all units share the same interface. This keeps testing very intuitive and simple. However, the provided support for testing units in isolation in the presence of concurrent behavior is not trivial.

The paper is organized as follows. Section 2 describes the basic concepts of DiPS and discusses the benefits of DiPS towards testing. Section 3 introduces the DiPSUnit test framework. DiPSUnit support is extended in the following sections. Section 4 describes how multi-threaded software can be tested in DiPSUnit. Section 5 describes how units are tested in the presence of external control flows. Section 6 validates DiPSUnit in two case studies. Conclusions are formulated in section 7.

2 The DiPS framework

We present DiPS [8] [10] as a Java component framework based on fine grained independent units that are connected as a pipe-and-filter architecture. The framework supports the development of system software such as protocol stacks or file systems [9]. DiPS implementations of network protocols (such as TCP, UDP, IPv4, IPv6 and Ethernet) have been developed to prove its usefulness. Ongoing research applies DiPS in the context of file systems. Communication between DiPS units is intercepted by the framework. As shown in figure 1, a unit is surrounded by a packet receiver, which accepts incoming packets, and a packet forwarder, which delivers packets to another packet receiver ¹. This allows for units to communicate anonymously, since they have no explicit notion of other units in the system.

A DiPS unit is an object-oriented entity with a very specific (fine grained) responsibility. A distinction has been made between *purely functional units* and *concurrency units*. This separation allows to change the concurrency model independent from the functionality in the system. DiPS units can be grouped together into more coarse grained units (such as a layer in a protocol stack). All

¹We will refer to packet receiver and packet forwarders as entry and exit points of a unit.

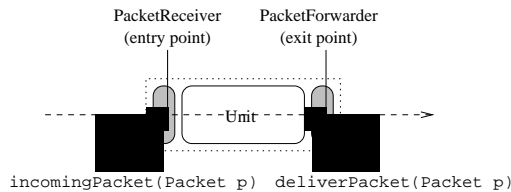


Figure 1: DiPS unit with one entry, one exit point

units process an explicit semantic entity (`Packet`), which represents a network packet or an I/O request packet. These packets can enter and leave a DiPS unit through one or more entry and exit points.

Next to the data (packet) flow there is a control flow that allows inter-unit control communication via events. The difference with the data flow is that instead of data packets, control information is transported. For example, a timeout signal that triggers a retransmit unit, or an ARP request that is sent whenever a network address cannot be translated into a hardware address. Registering listeners and dispatching of events is supported within the DiPS framework.

DiPS allows systems to be configured based on a set of high-level requirements and preferences (e.g. *reliable* and *secure* transport over a *network*). The DiPS `Composer` matches these requirements and preferences to unit descriptions, which describe the provided functionality of a unit. This matching process results in one or more valid structure configurations (a stack of protocol layers that adhere to the set of requirements) [3]. Finally a DiPS `Builder` translates such structure configuration into a running protocol stack.

The software design principles behind DiPS are based on the Law of Demeter (LoD) [7]. Thanks to these design principles, we believe that the DiPS approach can offer several benefits towards testing. We show how each of the five main principles of the LoD is supported in DiPS and how it affects testing:

- *Coupling control.* The LoD effectively reduces the number of methods a programmer can call and therefore facilitates reuse and raises the software's level of abstraction. DiPS units offer a *well-defined narrow* entry interface (`incomingPacket(Packet p)`). This allows easy development of specific unit tests: writing DiPS unit test cases is as easy as providing the unit with specific test packets and checking the resulting packet(s) leaving the unit.
- *Information hiding.* In general, the LoD prevents a method from directly retrieving a subpart of an object which is directly accessible. The object should offer methods that hide how that subpart can be reached. DiPS restricts units to *only contact each other anonymously* via the DiPS framework by using one specific method call (`deliverPacket(packet)`) (see figure 1). Therefore, DiPS units certainly cannot retrieve subparts from other units. Moreover, since DiPS units are independent, the system structure can be adapted for testing purposes without changing any code. This allows specific test units to be inserted at any given point

within the system structure. For example, to test how TCP connection establishment deals with communication delays or packet loss, specific manipulating units (providing delay, removal) are introduced in the system structure to manipulate specific packets. The alternative is to use network monitor tools that can intercept specific packets from the physical network, which is a less flexible solution.

- *Information restriction.* The LoD restricts the use of message sends (method calls). Message sends are used in DiPS only to forward a packet after it has been processed in a unit. DiPS doesn't force a developer to only send the packet after all processing is done, but it is a logical assumption in a pipe-and-filter architecture (such as DiPS).
- *Localization of information.* The LoD focuses on localizing class information. A programmer who studies a method should only be aware of types which are very closely related to the class to which the method is attached. The use of *independent (stand-alone) units* within the DiPS architecture allows for units to be tested in isolation (one only has to deal with control events, which are uniformly delivered by the framework).
- *Structural induction.* This principle states that the meaning of a system (as a whole of units) is a function of the meanings of its immediate constituents (i.e. the units it is composed of). DiPS offers *composition and builder support* which focuses on composing systems based on application specific requirements. Those requirements are matched by the DiPS composer to unit properties which allows it to calculate one or more unit configurations that adhere to the set of requirements [3]. Although this principle does not directly affect testing, it can be applied to translate high-level test descriptions into test instances. However, currently we do not provide this support.

3 DiPS unit testing

This section presents the DiPSUnit framework as a unit test extension for DiPS. DiPSUnit provides a uniform way of testing both singular and composed units (with one or more entry and exit points). It has been developed specifically for testing DiPS units, but we will present some characteristics that are interesting in general. There is a need for infrastructure support that helps developers to setup a test. As already mentioned, a DiPS unit can have multiple entry and exit points. This complicates a unit's plug in because every entry and every exit has to be connected with the test framework. Experience has learned that plugging in a unit with two entries and two exits (for example a protocol layer as shown in figure 2) would already be a great burden to the test developer.

Therefore, the DiPSUnit framework provides generic infrastructure support to plug in a unit (regardless of its number of entries or exits). First, a so-called `PacketContainer` is provided which manages packet buffers. The packet container is initialized by dynamically extending it with a packet buffer for each entry and exit point of a tested unit. An entry buffer is used to store specific

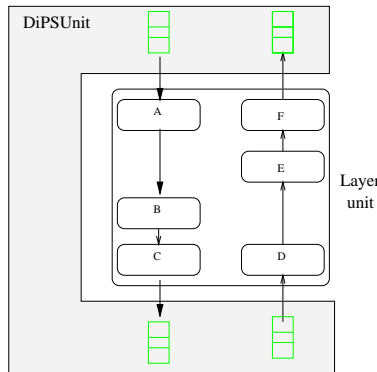


Figure 2: DiPSUnit test architecture

test packets created at test execution time. An exit buffer collects result packets that are received from the tested unit during a test run. There is a semantic difference between entry and exit buffers because they are linked differently with the tested unit (as we will describe later). Figure 2 shows the DiPSUnit framework and a set of DiPS units grouped into a layer (which again is a DiPS unit). For clarity reasons, the figure shows the packet container as 4 queues instead of 1 logical entity. Unit details such as entry and exit points are also removed.

Second, the `Linker` allows to link a unit with the DiPSUnit framework. This means that each entry buffer in the packet container is connected to a unit's specific entry point and each exit buffer is connected with an exit point. Connecting an exit buffer is done by introducing a specific packet receiver that puts incoming packets from the unit's exit point into an exit buffer. This is done completely transparent for a test developer.

The current version of DiPSUnit hides how a unit is linked with the framework, and therefore also how test packets are delivered to the tested unit. A test developer herself can however influence how test packets are delivered. For example, a non-trivial forwarding policy would alternate between entry buffers to simulate a send-receive scenario (such as a three-way handshake protocol for TCP connection establishment). For the tests we have written until now, the framework linking support is sufficient. However, we plan to extend the `Linker` so that it can interpret a high level description of a test configuration and use this to build a test case. Such a description can also describe how a unit must be linked with the framework. The linker then becomes an equivalent service for tests compared to what the DiPS builder is for protocol stacks. A next step could be to also describe how test packets are initialized. A test developer then no longer has to write any test code, but he can simply describe what has to be done.

To illustrate how all this support is provided in DiPSUnit, we summarize how it can be used. Unit testing is typically a three step process: a setup phase, a test and assertion phase and a tear down phase [5]. A DiPS unit test case translates these steps into the following scenario (as summarized below). First, in the setup phase, a packet container is created and initialized by creating a

packet buffer for each entry and for each exit of the tested unit. The unit is linked with the DiPSUnit framework by connecting each unit entry point with an entry buffer and each exit point with an exit buffer. This part of the setup process is test case specific. Second, to run a test case, specific test packets are created and added to specific entry buffers in the packet container. After that, the test case is started, i.e. the packets in the entry buffer(s) are delivered to the unit. After the test, the packet(s) in the packet containers' exit buffer(s) are compared to the expected result packet(s). Third, in the tear down phase, test packets are disposed by resetting the packet container.

1. *setup-phase* :
 - (a) create and initialize a packet container
 - (b) link the unit to be tested with the framework
2. *test- and assertion-phase* :
 - (a) fill specific buffers in the packet container with prepared test packets
 - (b) deliver the packets to the unit
 - (c) check assertions on the output packet(s)
3. *tear down-phase* : clear the packet container

The advantage of using DiPSUnit is the uniform test approach for both basic units (with one entry and one exit point) and composed units (with several entry and exit points). It makes no difference for the framework whether one unit or a flow of units² is tested, since the whole is being treated as one black-box unit.

The framework as it is described here is very intuitive and supports testing basic unit flows. However, it is far from sufficient to test complex systems such as a protocol stack or a file system. Important issues, such as concurrency support and external control events are not supported. We will show that by extending DiPSUnit in those two directions, we end up with a powerful yet simple framework that encourages independent unit testing. Throughout the remainder of this paper, we will gradually extend the test scenario above.

4 Dealing with concurrency

4.1 Context

Multi-threading is a powerful but error-prone programming technique. Multi-threading (or concurrency) is often introduced to support performance in heavy loaded systems (such as the protocol stack and the file system in a popular web server [4]). However, testing such concurrent systems is often very complicated, for two reasons. First, the lack of concurrency control that is available for the

²If it is a synchronous flow.

(test) developer. For example, it is interesting to limit concurrency in a test case to one thread at a time instead of testing the complete concurrent system at once. Second, multi-threading code is often spread out over the code (Kiczales e.a. [6] say concurrency code *cross-cuts* the functional code). Languages, such as Java, support threading in such a way that programmers stop thinking about the reasons and the consequences of using it. It's just too easy to introduce such a complex aspect as threads. This unintelligent use of threads clearly complicates testing, maintaining, debugging and even understanding the code.

It is important to provide concurrency support to the programmer, both at testing as at development time. The explicit concurrency support in DiPS (see also the *active* concurrency unit in figure 3 with a buffer and a thread inside) has advantages towards testing. It allows concurrency code to be added to the system without changing the code. This also allows to control the concurrency model. For example, DiPS allows to change the concurrency model from completely synchronous to asynchronous, or some hybrid model.

This section shows how the DiPSUnit framework provides support to test multi-threaded software. We stress that although the DiPS architecture helps to manage concurrency, not all concurrency issues are solved. For example, concurrent data access testing is not yet supported. We focus on testing multi-threaded systems as a whole.

4.2 Monitor support

It should be clear that some mechanism is needed which allows for the test to be *suspended* until all packets have arrived at the unit's exit(s) (i.e. until they are all available in the packet container). But things are not as simple as they seem. For instance, we should avoid our test to be suspended indefinitely when packets do get lost. Therefore it should be possible to specify some kind of timeout-value after which the test will continue. Altogether, a test should be resumed after all packets have arrived at the unit's exit(s) OR when a timeout occurs.

To this purpose the test framework offers a **Monitor**. A test developer trying to deal with concurrency issues should simply ask for such a monitor to be present. DiPSUnit provides the monitor with the packet-container it should keep an eye on and a timeout value after which the test should continue. After handing over the packets to the unit being tested, the developer calls the monitor asking the current test to be suspended. The monitor will then send a wake-up signal to the test as soon as all packets have arrived in the container or when the timeout-value expires.

There is still one issue to solve though. To be able to know when all packets have arrived at the exit(s), we also need to be able to specify the number of packets we're expecting at each of the exit(s). Some DiPS units may introduce new packets (such as a fragmenter unit) to, or remove packets (such as a reassembly unit or a header constructor that drops packets with erroneous headers) from the data flow. Therefore, one cannot assume the number of packets at the exit(s)

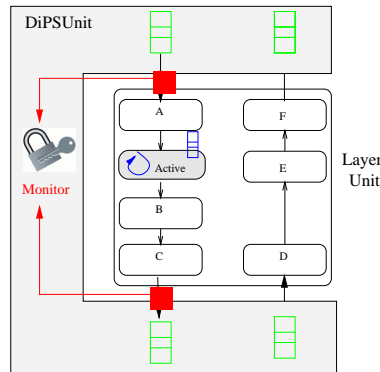


Figure 3: Monitor support

to be equal to the number of packets presented at the input(s).

We extend DiPSUnit as follows. First, the packet container allows to specify the number of packets to be expected at each exit. The `Monitor` will use this number to determine whether all packets have arrived at a certain exit, or not. Second, when the monitor object is called, it creates a specific packet receiver for each exit point of the unit. Such a packet receiver not only puts incoming packets in an exit buffer, it also signals the monitor when a packet is received. When the total number of expected packets (which can be asked from the container) equals the total number of received signals by the monitor, the suspended test is resumed (unless a timeout has already occurred).

Figure 3 shows DiPSUnit, extended with the monitor. The figure shows a down-going path with an active (concurrency) unit. The monitor controls the down-going path. It blocks the test until all expected packets have arrived at the lower exit point. The test case scenario for such concurrent data flows looks like this :

1. *setup-phase* :

- (a) create and initialize a packet container
- (b) *ask for a monitor to be present*
- (c) *specify the number of expected packets at the different exits*
- (d) link the unit to be tested with the framework

2. *test- and assertion-phase* :

- (a) fill specific buffers in the packet container with prepared test packets
- (b) deliver the packets to the unit
- (c) *ask the monitor to be suspended until timeout expires OR the number of expected packets has been reached*
- (d) check assertions on the output packet(s)

3. *tear down-phase* :

- (a) reset the packet container
- (b) *dispose the monitor*

As one might notice, despite the added complexity of multiple threads being present in the unit and the multiple inputs and outputs, the actual test case stays simple and easy to understand. Remark that the use of the monitor is purely optional and only needed if concurrency is present.

5 Control flow stubs

5.1 Context

We already mentioned that DiPS units are independent from other unit instances. Because of this independence, the units can easily be replaced by dummy implementations at testing time. Such dummies can for example prepare necessary information for the tested unit (this is done by attaching this information to the `Packet` that is being processed).

To test a unit in isolation, all control flows must be intercepted. This is typically done by using stubs [2] or mock objects [11] that simulate the behavior of external units that are contacted by the tested unit. In the case of DiPS, every control event is caught by a dummy unit, acting as a stub.

Although the stub mechanism is simple, transparently introducing stubs is not always trivial. Often, references to units are created internally what makes it impossible to replace these units with stub implementations without changing any code. Therefore, all code must use interfaces instead of instances directly and all instances must be created externally and handed over to the unit [7].

DiPS solves this problem because of the following reasons. First, DiPS prevents units from having direct references to each other. This allows any unit to be replaced by a stub without any problem. Second, DiPS inherently forces to split up functionality in fine-grained units, which all process the same generic packet abstraction. Therefore, all stubs share the same interface. Third, control information can only be exchanged via events which allows easy integration of stubs by replacing an event handler with a stub.

5.2 Control flow policies

DiPSUnit offers infrastructure support to uniformly deal with external events. The test developer only provides what should be done in case a certain event is received. Installing the stubs and connecting them with the tested unit is done by DiPSUnit itself. This section describes how this support is provided.

To allow a test developer to easily describe how to respond to a given event, a `Policy` is introduced. Such a `Policy` can, for example, be a simulation of the

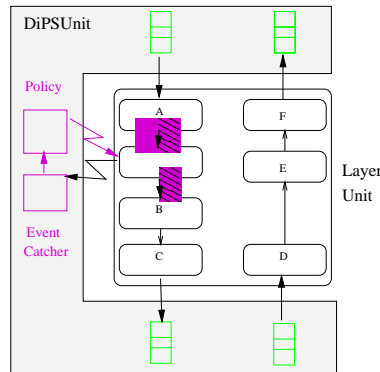


Figure 4: Control flow stub support

actual functionality, but it is also interesting to provide a policy that simulates an error. For example, not sending a reply event while it is expected, delaying the reply or producing a wrong answer deliberately. Separation of policies from actual test cases, allows for a generic policy to be easily reused. A policy, for example, that silently removes incoming triggers is generic enough to test packet loss in different cases. The same is true for a *delay* and a *duplicate* policy.

All policies are registered in the DiPSUnit `EventManager` class, together with the event type that triggers each of them. This is done at test initialization time by a specific test case. The `EventManager` is actually the link between a test case and the DiPSUnit framework: a test case registers its policies in the `EventManager`, DiPSUnit looks up the associated `Policy` for each event that is received (as is explained further).

Control events are actually caught by an `EventCatcher` object. An `EventCatcher` is created for each event type that is registered in the `EventManager`. The default `EventCatcher` that is offered by DiPSUnit simply delegates all events to the `EventManager` to look up the associated `Policy` object. However, a non-trivial `EventCatcher` can be introduced (for example, an `EventCatcher` that collects a number of events before it calls the `EventManager`). Registering an `EventCatcher` as event listener is done transparently in DiPSUnit, by using event support that is offered in DiPS.

Figure 4 shows the case where a unit emits a control event which is caught by the `EventCatcher` and delegated to a specific `Policy`. As shown, a `Policy` object can send control events back to a unit, if necessary.

In the following scenario, we summarize how DiPSUnit is used to support control events. The concurrency steps, introduced in the previous section have been omitted to focus on control flow stubs. Remark that the extra steps for control flow stub support are again optional. When no events are triggered within the tested unit, the scenario falls back to the original scenario described in section 3. This keeps the complexity of the test environment minimal.

1. *setup-phase* :

- (a) create and initialize a packet container
 - (b) *register event catchers and associated policies*
 - (c) link the unit to be tested with the framework
2. *test- and assertion-phase* :
- (a) fill specific buffers in the packet container with prepared test packets
 - (b) deliver the packets to the unit
 - (c) check assertions on the output packet(s)
3. *tear down-phase* :
- (a) clear the packet container
 - (b) *reset the event manager*

The combination of DiPS and DiPSUnit has several advantages. The substitution of control flow functionality by stubs is transparent for the actual tested code. This reduces the risk of replacing stubs by the actual functionality during acceptance testing because no code changes have occurred. DiPS units (and therefore also stubs) are typically very fine grained functional elements. This decreases the development cost of a unit (or a stub) and it encourages a very iterative test process (*test a little, code a little* [1]). DiPSUnit can build on the event catching and throwing support provided in DiPS. This keeps DiPSUnit simple, although it provides non-trivial support to a test developer.

6 Validation/application: integration in JUnit

To validate our ideas on testing for the DiPS framework, we implemented them using the *JUnit testing framework* [5]. JUnit is a well known Java-based unit test framework which originated from within the XP(Extreme Programming)-approach [1]. The JUnit framework consists of a few base-classes which allow a developer to quickly write specific tests, group them in larger test suites and run them in batch or from within a simple GUI (see also the dark shaded box in figure 5). For a more detailed explanation of the JUnit-framework, we refer to [5].

Within JUnit, everything centers around the `TestCase` class. It offers methods to prepare the fixture of the test (the setup-phase of a unit-test), to run several test methods and to clean-up temporary objects afterwards (the tear down phase). We extended this `TestCase` class and created the `DiPSTestCase`. The latter provides a test-developer with a packet container, linking support, monitoring support and an event manager. The extended JUnit, which we call *DiPSUnit* (it's specifically tailored to meet the needs of the DiPS-framework), is shown in figure 5.

In the following paragraphs we will elaborate on two specific test cases. We will start from a simple scenario which will gradually be extended to a more complex

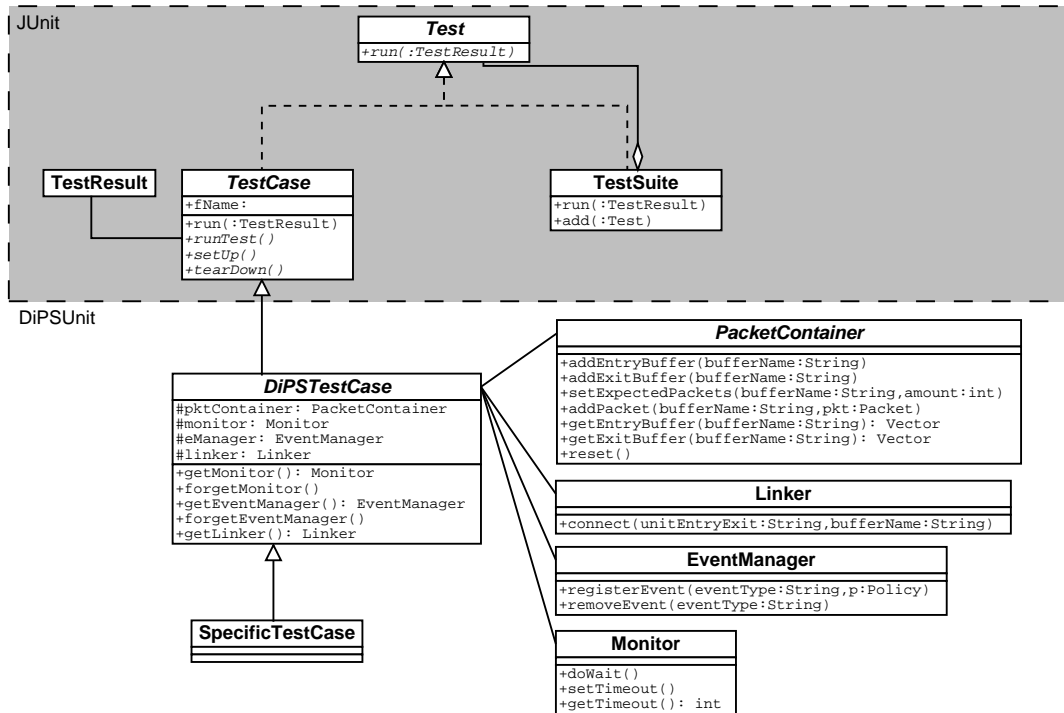


Figure 5: UML model of DiPSUnit

case which includes previously mentioned issues as concurrency and external communication. We will show that, despite the continuous growth in complexity, the development of unit tests stays uniform and simple to understand.

6.1 A simple test case

As an example, let us write a simple unit test for an Ethernet header-constructor. To this purpose we inherit from `DiPSTestCase`. We will go over each of the three steps found in every test case : setup, testing, tear down.

The following two steps will define the entire `setUp()` phase of our test : First thing that needs to be done is to initialize the `PacketContainer` with the necessary buffers. Since we're planning on testing a simple unit with only one entry and one exit, an equal number of buffers will suffice. As one might notice from the code below, the packet container allows to dynamically add any number of entry/exit buffers (`addEntryBuffer(...)`, `addExitBuffer(...)`). Each of these buffers can be given a name to enable the developer to separate them from one another.

In a second step, we create the unit to be tested, in this case an `EthernetHeaderConstructor` and link it into the framework. The linking is done by means of a request to the framework for an appropriate `Linker`

(`getLinker(testedUnit)`). This linker is used to connect specific entry/exit points of a unit with specific buffers in the packet container.

At this point, setup is complete. As one might notice, we have not yet constructed any test-packets. The reason for this will be explained later. Since the unit we're about to test, does not contain any threading and since it does send control events, no monitor or event manager are needed.

```
protected void setUp() throws Exception{

    // create the PacketContainer and add all the buffers
    pktContainer = new PacketContainer();
    pktContainer.addEntryBuffer("packets_to_below");
    pktContainer.addExitBuffer("packets_from_above");

    // Create the unit to be tested
    testedUnit = new EthernetHeaderConstructor();

    // link the unit into the testing-framework
    linker = getLinker(testedUnit);
    linker.connect("entry", "packets_to_below");
    linker.connect("exit", "packets_from_above");
}
```

Now that we have linked our unit into the test framework, we are ready to construct test packets which will be sent through the unit in the `test()` phase. One might argue that creating test packets is something to be done in the setup-phase. The argument against goes as follows : *JUnit* allows a test programmer to group several test methods for a specific unit within one single `TestCase`. It also specifies that both `setUp()` and `tearDown()` are executed for each test method present in the test case. In the case of *DiPSUnit* (which is an extension of *JUnit*), the typical difference between each test method lies in the fact that different test packets are sent through the unit.

To resume our example: after the test packets have been created, we add them to an entry buffer of the `pktContainer`. Since we will not be dealing with a multi-threaded unit, all that is left to do is send the packet through the unit by flushing the entry buffer (`flushEntryBuffer(...)`). Finally, we check some assertions on the output (`assertEquals(...)`).

```
public void test...{
    // create a test packet
    Packet testPkt = new Packet(.....);

    // add it to the an input buffer of the packet container
    pktContainer.addPacket("packets_to_below", pkt);

    // flush the entry buffer = send packet through the unit
```

```

    pktContainer.flushEntryBuffer("packets_to_below");

    // check assertions on the output
    assertEquals(getExitBuffer("packets_from_above").size(), 1);
    assert(...);
}

```

After the test has finished, we're left with the `tearDown()` phase. In this case, this is as simple as removing all test packets from the packet container (`pktContainer.reset()`).

```

protected void tearDown() throws Exception {
    // reset the packet container
    pktContainer.reset();
}

```

6.2 A complex test case

We will now complicate our example by embedding the header-constructor into a larger unit : the complete `EthernetLayer` (which is again a DiPS unit). First of all, this `EthernetLayer` has twice as many entries and exits (one entry/exit towards the layers above and one entry/exit towards the network card(s) below). Similar to the simple test case, all entries and exits are added to the `PacketContainer` and the unit is linked to `DiPSUnit`.

What really complicates testing are the following two problems : First of all, Ethernet might call in the help of external units to map for example an IP-address to an Ethernet-address. In the previous example, the test case provided the Ethernet header constructor unit with a source and a destination Ethernet-addresses. However, in real life Ethernet will contact another protocol (for example Address Resolution Protocol (ARP)) to do this. Since our test case focuses on testing the functionality of Ethernet only, we need to replace ARP with a *stub*. As mentioned in section 5, such control communication in DiPS occurs by means of events in DiPS which are registered in the `EventManager` together with a policy `NormalARPRequestReplyPolicy` that acts as a stub.

The second problem to deal with is the presence of concurrency within Ethernet. The `EthernetLayer` contains threading to avoid it from being blocked while some IP-address is being resolved to an Ethernet-address by ARP. For performance reasons, Ethernet must be able to process packets while waiting for an ARP reply. Because of this concurrency, we are obliged to use the `DiPSUnit` monitor (`getMonitor()`).

If we compare the `setUp()` code below with the simple test case of the `EthernetHeaderConstructor` unit, we see two differences: the registration of an event and a `Policy` at the `EventManager` and the request to use a `Monitor`.

```

public void setUp() throws Exception{

```

```

// create the PacketContainer and add all the buffers
pktContainer = new PacketContainer();
pktContainer.addEntryBuffer("packets_to_below");
pktContainer.addEntryBuffer("packets_to_above");
pktContainer.addExitBuffer("packets_from_above");
pktContainer.addExitBuffer("packets_from_below");

// Create the unit to be tested
testedUnit = new EthernetLayer();

// register an event type and policy with the event manager
getEventManager().registerEvent( "ARPRequestEvent",
                                new NormalARPRequestReplyPolicy() );

// ask for a monitor to deal with concurrency
Monitor monitor = getMonitor();

// link the unit into the testing-framework
linker = getLinker(testedUnit);
linker.connect("UpperEntry", "packets_to_below");
linker.connect("LowerExit", "packets_from_above");
linker.connect("LowerEntry", "packets_to_above");
linker.connect("UpperExit", "packets_from_below");

}

```

If we take a look at the actual `test(...)` phase, it is clear that it is practically identical to the test method for the `EthernetHeaderConstructor` unit³. Data in the test packet might be different, but this is irrelevant for the example. Since the monitor needs to know the number of expected packets in each of the exit buffers, we have to specify this by calling `setExpectedPackets(...)`. By calling the `monitor.doWait()` method, the current thread (the one in which the test is running) is suspended until all expected packets have arrived or until a timeout has occurred (whichever happens first).

```

public void test...{
    // create a test packet
    Packet testPkt = new Packet(...);

    // add it to the an input buffer of the packet container
    pktContainer.addPacket("packets_to_below", pkt);
    pktContainer.setExpectedPackets("packets_from_above", 1);

    // flush the entry buffer = send packet through the unit
    pktContainer.flushEntryBuffer("packets_to_below");
}

```

³Of course more complex tests can be defined.

```

    // go to sleep
    monitor.doWait();

    // check assertions on the output
    assertEquals(getExitBuffer("packets_from_above").size(), 1);
    assert(...);
}

```

Also the `tearDown()` phase is similar compared to the tear down phase for the simple test case. The only additions are the call to `forgetMonitor()` and `forgetEventManager()`. The first method call informs the framework that the `Monitor` is no longer needed. The second call tells the framework we are no longer interested in the previously registered events.

```

protected void tearDown() throws Exception {
    // reset the packet container
    pktContainer.reset();
    forgetMonitor();
    forgetEventManager();
}

```

This example shows that a test case doesn't need to become increasingly complex for complex units. *DiPSUnit* allows to deal with issues such as concurrency and external communication, while still providing a test developer with a uniform way of testing and keeping it simple and intuitive to write a test.

7 Conclusion

The DiPS framework facilitates executing tests because of its forced modularized architecture with independent units. The anonymous communication model and the event based control flow allows to test units in isolation without changing unit code. This is essential in a test-first development strategy.

Another advantage is the possibility to introduce manipulating test units in the system. To test a network protocol, it is essential to test how this protocol behaves in the context of network issues such as transmission delay, packet loss, or data corruption. For example, a test for fragmentation/reassembly functionality in the IP layer is able to introduce specific manipulators *close to* the fragmenter and/or reassembly units (for instance a manipulator that removes one of the fragments). The integration of manipulator units into the system structure is much more intuitive and logical than manipulating the raw data flow on the network.

Especially for more complex software systems, such as protocol stacks, support for concurrency is needed. One of the key elements in DiPS is to separate functional code from (non-functional) concurrency code. This allows to localize

active (threaded) spots in the architecture and even to replace an asynchronous architecture into a synchronous one and vice versa.

DiPSUnit supports to uniformly test protocol stacks from fine grained unit level, to composed unit level (i.e. testing layers in isolation) and even acceptance level (testing a complete protocol stack). However, developing test cases is still intuitive and simple. Our experience is that DiPSUnit (therefore) encourages a test-first development approach.

References

- [1] Kent Beck, *Extreme programming explained: embrace change*, Addison Wesley, 1999.
- [2] Robert V. Binder, *Testing object-oriented systems: models, patterns and tools*, Addison Wesley, 1999.
- [3] Ioana Şora, Frank Matthijs, Yolande Berbers, and Pierre Verbaeten, *Automatic composition of systems from components with anonymous dependencies specified by semantic-unaware properties*, Proceedings of Technology of Object-Oriented Languages and Systems Eastern Europe (TOOLSEE) 2001, IEEE Computer Society Press, March 2002, To appear.
- [4] Google, <http://www.google.com/press/highlights.html>, Google's Technical Highlights, 2001.
- [5] E. Gamma K. Beck, *Test infected: Programmers love writing tests*, Tech. report, <http://www.junit.org/>, 1998.
- [6] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, *Aspect-oriented programming*, ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland (Mehmet Akşit and Satoshi Matsuoka, eds.), vol. 1241, Springer-Verlag, New York, NY, 1997, pp. 220–242.
- [7] Karl J. Lieberherr and Ian Holland, *Assuring good style for object-oriented programs*, IEEE-*software* (1989), 38–48.
- [8] Frank Matthijs, *Component framework technology for protocol stacks*, Ph.D. thesis, Katholieke Universiteit Leuven, Dec 1999, Available at <http://www.cs.kuleuven.ac.be/~samm/netwg/dips/index.html>.
- [9] S. Michiels, P. Kenens, F. Matthijs, D. Walravens, and P. Verbaeten, *Component framework support for developing device drivers*, Proceedings of International Conference on Software, Telecommunications and Computer Networks (SoftCOM2000) 1 (2000), 117–126.
- [10] S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten, *DiPS: A Unifying Approach for Developing System Software*, Position paper in Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), K.U.Leuven, Dept. of Computer Science, May 2001.

- [11] P. Craig T. Mackinnon, S. Freeman, *Endo-testing: Unit testing with mock objects*, eXtreme Programming and Flexible Processes in Software Engineering XP2000, Connextra Ltd., London, England, June 2000.