DiPS: Filling the Gap between System Software and Testing

Sam Michiels, Dirk Walravens, Nico Janssens, Pierre Verbaeten
DistriNet, Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium,
+32 16 327640
{Sam.Michiels, Dirk.Walravens}@cs.kuleuven.ac.be

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 presents DiPS, a component framework, which forces to develop *testable* software, and DiPSUnit, a JUnit extension, to test DiPS units in a uniform way. Although non-trivial test support is provided, using DiPSUnit keeps testing simple and intuitive thanks to the DiPS approach.

Keywords
Testing, framework, component software engineering

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 because of two reasons: first, it is very difficult to isolate the basic building blocks as stand-alone units that are independent from each other. Second, concurrency code, which is introduced in such multi-threaded system software, often crosscuts the code [4].

This paper presents DiPS (Distrinet Protocol Stack) [6], a component framework we have built to support protocol stack development. DiPS forces to deploy four design principles, which are important prerequisites to develop adaptable and *testable* software. As proof of concept, we have developed the DiPSUnit test framework, which is an extension of JUnit [3], specifically to test DiPS units.

The rest of this paper is organised as follows. Section 2 presents four essential chararcteristics of *testable* software. Sections 3 and 4 present DiPS and DiPSUnit, two frameworks we have developed to proof our ideas about software development and testing. Conclusions and some open points of discussion are formulated in section 5.

2 DESIGN FOR TESTING

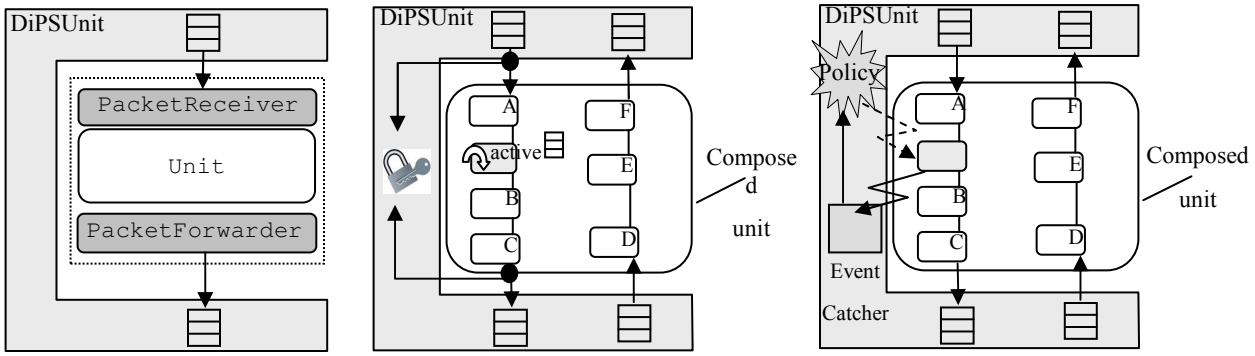
XP could be the victim of its own success: testing complex (system) software in XP could lead to *test hell*, where test code becomes so complex and unmanageable that it needs testing… One way to deal with this is to design *testable* software, i.e. software that is designed such that it can be tested easily. We distinguish four essential characteristics of so-called *testable* software:

- ***Modularity***: it is important that the design reflects fine-grained (singular) units as separate entities, to allow unit testing in isolation [5]. A collection of units can be grouped together into a *composed unit*, which is treated the same way as a singular unit.
- ***Independent units***: to allow transparent substitution of units and to reduce the risk of unexpected side effects (when units are replaced) during acceptance testing, it is essential that units are independent from each other.
- ***Separation of concurrency from functionality***: traditional system software is difficult to develop, understand, maintain, adapt or test. One of the major reasons is that concurrency code crosscuts the functional code [4]. Separating concurrency from the rest of the code facilitates development and testing because programmers can concentrate on one aspect at a time. A protocol stack developer should concentrate on creating a header parser or a packet fragmenter without being distracted by non-functional aspects such as concurrency (parallellism). Because of this separation, unit tests can be done *first* in a single-threaded context, and multi-threading can be added *later*, without changing any unit code.
- ***Uniform unit interface***: reduce the unit's interface and share the same interface type as much as possible. This facilitates reuse and raises the software's level of abstraction. This feature, combined with modularity (i.e. composed unit is again a unit), allows a uniform testing approach for both singular units and composed units.

3 THE DIPS FRAMEWORK

DiPS is a Java component framework based on 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. Communication between DiPS units is intercepted by the framework. This forces units to communicate anonymously (***independent units***), 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 (***modularity***), such as a packet



Consistent test approach in DiPSUnit: a singular DiPS unit, a composed unit with internal concurrency and a composed unit which sends (and receives) control events

fragmenter or a header parser. DiPS units can be grouped together into composed units (such as a protocol layer in a protocol stack). A distinction has been made between *purely functional units* and *concurrency units*. This ***separation*** allows the concurrency model to change, independent from the functionality in the system.

All DiPS units process (only) packets, which are delivered via a ***uniform unit interface***. These packets can enter and leave a DiPS unit through one or more entry and exit points. A singular DiPS unit (such as a fragmenter or an encryption unit) with one entry and one exit (`PacketReceiver` and `PacketForwarder`) is shown in detail in the left figure. Next to the data (packet) flow there is a control flow that allows anonymous inter-unit control communication via *DiPS events*.

4 PROOF OF CONCEPT: DIPS UNIT TESTING

Thanks to DiPS, the DiPSUnit framework can provide a uniform way to test singular units as well as composed units (see figure). This keeps testing very intuitive and simple. However, the provided support for testing units in isolation in the presence of concurrent behavior and external control events is not trivial.

The concurrency (*active*) unit in the middle figure has an internal thread and a buffer to store incoming packets. This decouples the packet flow in two parallel flows. When such a unit is present within a composed unit, a test must be suspended until all packets have arrived or until a timeout occurs (to avoid being suspended forever in case of an error). DiPSUnit offers a monitor that blocks until all packets are processed (even in the context of internal packet removal/creation).

DiPS units can exchange control information by using DiPS events (right figure). To test a unit in isolation, all control flows must be intercepted. Although the stub mechanism is simple, transparently introducing stubs is not always trivial. DiPSUnit offers support to uniformly deal with external events. A test developer can describe how to respond to a given event by creating a Policy (which acts as a stub). The substitution of control flow functionality is transparent for the code under test. This reduces the risk of introducing errors when stubs are replaced by the actual functionality during acceptance testing.

For a detailed description of DiPSUnit we refer to [7] [8].

5 CONCLUSIONS

The combination of JUnit, DiPS and DiPSUnit seems very promising. JUnit offers the basic infrastructure to develop test cases and test suites. DiPS facilitates unit testing because it forces to create modularized architectures and because it allows units to be replaced without changing any code. Thanks to this support, DiPSUnit can consistently test DiPS units, from fine-grained to composed unit level. However, developing test cases is still intuitive and simple.

Relevant points of discussion are:

- Software engineering techniques, such as design patterns [2] and refactoring [1], do help in creating 'good' software, and the xUnit test framework helps in testing software. However, we claim that infrastructure support is required (such as a component framework) to *force* design techniques to be applied.
- What other software design principles facilitate or hinder testing?

ACKNOWLEDGEMENTS

This research has been carried out in order of Alcatel Bell with financial support of IWT (project SCAN #010319).

REFERENCES

1. M. Fowler. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
2. E. Gamma, e.a., *Design patterns: elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
3. E. Gamma, K. Beck, *Test infected: Programmers love writing tests*, http://www.junit.org/, 1998.
4. G. Kiczales, e.a., *Aspect-Oriented Programming*, In proceedings of ECOOP'97, 1997.
5. T. Mackinnon, e.a., *Endo-Testing: Unit Testing with Mock Objects*, XP2000, June 2000.

6. F. Matthijs, *Component Framework Technology for Protocol Stacks*, Ph.D. thesis, K.U.Leuven, 1999. (Available at http://www.cs.kuleuven.ac.be/~samm/netwg/dips/)

7. S. Michiels, D. Walravens, e.a., *DiPSUnit: an Extension of the Junit Test Framework for DiPS*, Tech. Report CW-333, K.U.Leuven, Dept. Comp. Science, 2002.

8. S. Michiels, D. Walravens, e.a., *DiPSUnit: A JUnit Extension for the DiPS Framework*, To appear as experience report in XP2002. K.U.Leuven, Dept. Comp. Science, 2002.