What went wrong.

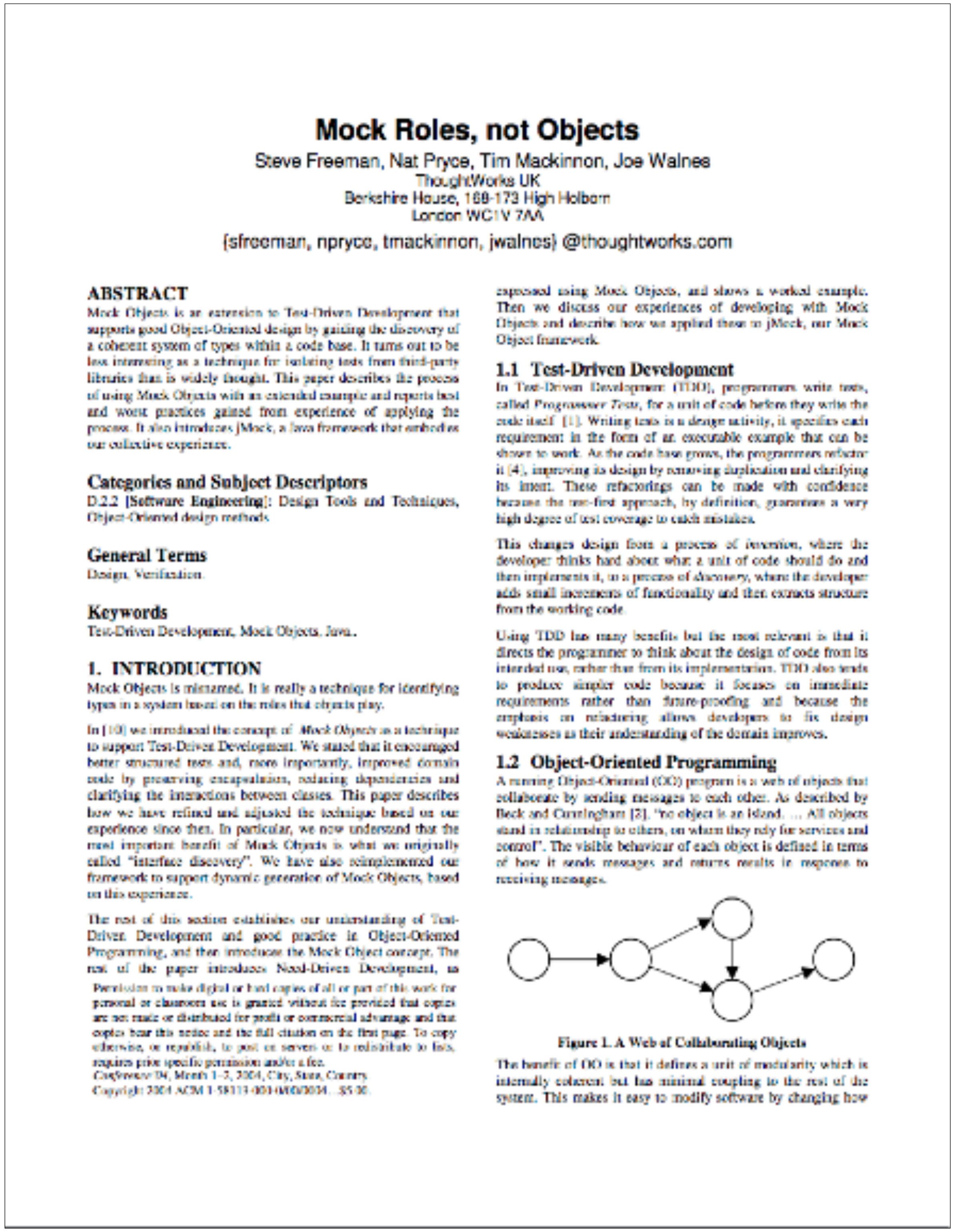
Mocking is *supposed* to be a technique for **(1) Object Oriented Design**, in the context of **(2) TDD**, for exploratory design and discovery.

Mocking is **not** just a tool for test isolation.

What went wrong.

Mocking is *supposed* to be a technique for (✗) **Object Oriented Design**, in the context of (✗) **TDD**, for exploratory design and discovery.

✗ Mocking is **not** just a tool for test isolation.



Mock Roles, not Objects

Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes
ThoughtWorks UK
Berkshire House, 168-173 High Holborn
London WC1V 7AA
{sfreeman, npryce, tmackinnon, jwalnes} @thoughtworks.com

ABSTRACT
Mock Objects is an extension to Test-Driven Development that supports good Object-Oriented design by guiding the discovery of a coherent system of types within a code base. It turns out to be less interesting as a technique for isolating tests from third-party libraries than is widely thought. This paper describes the process of using Mock Objects with an extended example and reports best and worst practices gained from experience of applying the process. It also introduces jMock, a Java framework that embodies our collective experience.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques, Object-Oriented design methods.

General Terms
Design, Verification.

Keywords
Test-Driven Development, Mock Objects, Java..

1. INTRODUCTION
Mock Objects is misnamed. It is really a technique for identifying types in a system based on the roles that objects play.

In [10] we introduced the concept of *Mock Objects* as a technique to support Test-Driven Development. We stated that it encouraged better structured tests and, more importantly, improved domain code by preserving encapsulation, reducing dependencies and clarifying the interactions between classes. This paper describes how we have refined and adjusted the technique based on our experience since then. In particular, we now understand that the most important benefit of Mock Objects is what we originally called "interface discovery". We have also reimplemented our framework to support dynamic generation of Mock Objects, based on this experience.

The rest of this section establishes our understanding of Test-Driven Development and good practice in Object-Oriented Programming, and then introduces the Mock Object concept. The rest of the paper introduces Need-Driven Development, as expressed using Mock Objects, and shows a worked example. Then we discuss our experiences of developing with Mock Objects and describe how we applied these to jMock, our Mock Object framework.

1.1 Test-Driven Development
In Test-Driven Development (TDD), programmers write tests, called *Programmer Tests*, for a unit of code before they write the code itself [1]. Writing tests is a design activity, it specifies each requirement in the form of an executable example that can be shown to work. As the code base grows, the programmers refactor it [4], improving its design by removing duplication and clarifying its intent. These refactorings can be made with confidence because the test-first approach, by definition, guarantees a very high degree of test coverage to catch mistakes.

This changes design from a process of *invention*, where the developer thinks hard about what a unit of code should do and then implements it, to a process of *discovery*, where the developer adds small increments of functionality and then extracts structure from the working code.

Using TDD has many benefits but the most relevant is that it directs the programmer to think about the design of code from its intended use, rather than from its implementation. TDD also tends to produce simpler code because it focuses on immediate requirements rather than future-proofing and because the emphasis on refactoring allows developers to fix design weaknesses as their understanding of the domain improves.

1.2 Object-Oriented Programming
A running Object-Oriented (OO) program is a web of objects that collaborate by sending messages to each other. As described by Beck and Cunningham [2], "no object is an island. … All objects stand in relationship to others, on whom they rely for services and control". The visible behaviour of each object is defined in terms of how it sends messages and returns results in response to receiving messages.

Figure 1. A Web of Collaborating Objects

The benefit of OO is that it defines a unit of modularity which is internally coherent but has minimal coupling to the rest of the system. This makes it easy to modify software by changing how

"[Dependency Injection] … is a virtue."