# 测试总线势在必行

## ——设计支持自动化验收测试的架构

**作者：Robert C. Martin**
**翻译：孙鸣**

敏捷方法，特别是测试驱动开发实践，已经唤醒了软件业对于自动化验收测试的认识。人们现在已经知道，在版本发布冲刺阶段的进度压力下，让大量的测试人员疯狂地手工运行那些枯燥乏味的测试脚本是根本不能保证软件质量的。

市面上可以买到很多工具，这些工具可以帮助测试者编写通过用户界面来进行系统测试的自动化脚本。这样的工具就像一个机器人，它们被编程用来模拟测试人员—它们揿下按钮，选择菜单项，勾上复选框，在文本框里输入数据，然后观察屏幕。糟糕的是，通过 UI 进行的测试运行缓慢，晦涩难懂并且充满危险。我的一个客户必须得运行超过 10,000 个的验收测试，如果通过 UI 去运行这些测试，即使用上好几十台机器，也会花去他好几天的时间。而且，测试是用一种和码字类似的语言写出来的，使得测试难以理解。随着时间的推移，他都记不起来每个测试到底要验证什么了，他所知道的就是所有这 10,000 个测试必须要通过。界面上每个微小的改变都会造多则百个，少则几十个测试的失败或者无法运行。最后，他发现根本不可能去升级或者改进已经过时了的界面。

大概一个世纪以前，电话公司设计了包含内置测试装置的电话交换机，解决了类似的问题。这样的装置，又称作测试总线，能够让电话公司在晚间自动测试每条电话线，这样远在用户发现问题前，他们就把故障和对服务质量的影响给消除了。

软件开发商也开始采用类似的策略，用内置的测试总线来构架软件系统。他们使用类似 Fit 以及 FitNesse（www.fitnese.org）这样的工具，用一种业务人员易于理解的说明性语言来描述测试。不过，要想构建出这样的自动化验收测试，在架构设计层面，至少要做到三个主要的隔离。

## 绕过 UI

在软件系统中，测试总线指的是一组 API，通过这组 API 能够方便地编写单元测试和验收测试。这些测试描述了系统行为。单元测试描述了模块行为，而验收测试描述了功能特性行为。

测试总线的存在意味着系统中存在一些相应的结构，能够让测试去访问一些承载它们所描述行为的模块和子系统。例如，想要编写绕过 UI 去操作底层业务规则的测试，就得有这样一个 API，UI 和测试都可以使用这个 API 调用业务规则。此外，这个 API 必须还要独立于 UI（见图 1）。

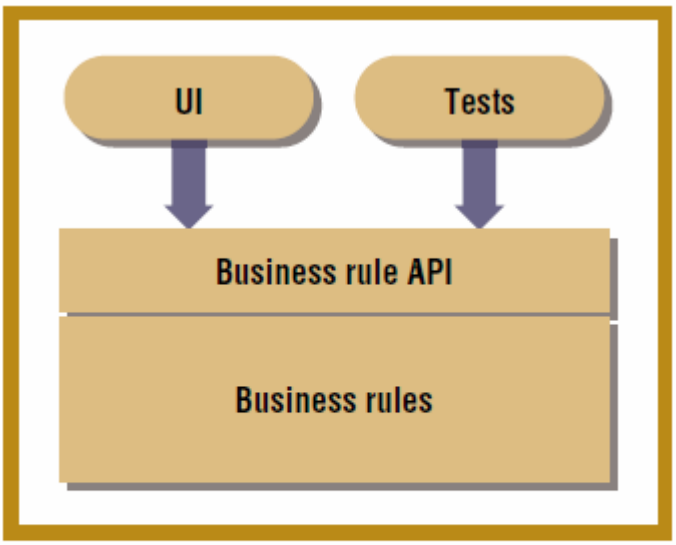


**Figure 1. A testable system includes a test bus that can access the API independent of the UI.**

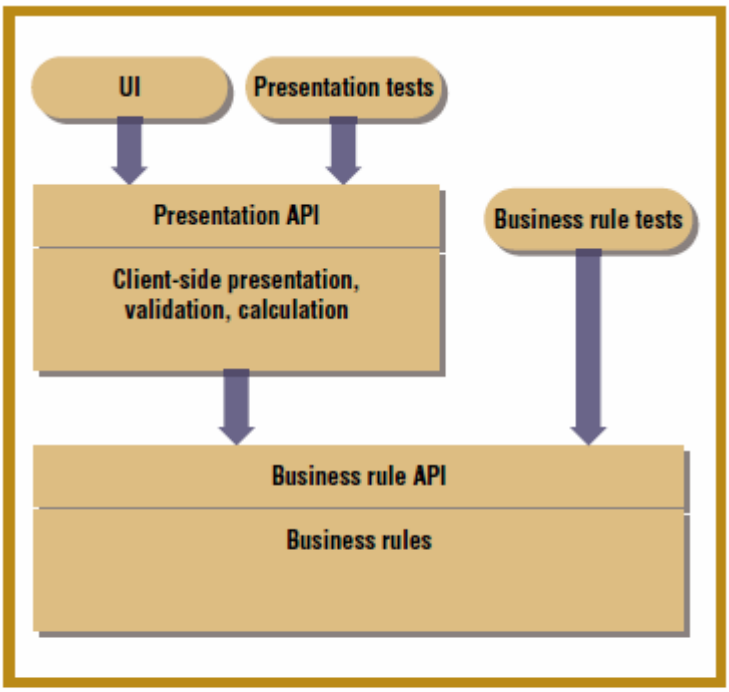


**Figure 2. The presentation API separates the presentation, validation, and calculation rules from the UI's low-level details. This helps ensure developers can test all the business rules—even those that extend into the UI.**

在图 1 中，我们可以清晰地看到，测试和 UI 是可以互换的。它们使用同样的 API，因此，测试可以通过所有的真实操作去驱动系统并指明所要求的行为细节。

然而，有些 UI 富含逻辑，很容易将业务规则包含进去。例如，有些系统会在 UI 中校验数据或者执行一些重要计算。在客户-服务器以及基于 WEB 的系统中，这种情况尤其普遍。此时，采用一般的设计策略往往会使得业务规则难以测试。不过，如果开发团队把可测试性作为一种强制的架构要求，那么他们就必须得提供能够让测试访问业务规则的机制。一种常见的解决方案就是在 UI 中创建另一个 API 来把校验、计算规则与 UI 的底层细节隔离（见图 2）

显然，这种 API 会约束 UI 的设计方式以及设计者使用客户端工具（例如，JavaScript）的方式。设计者必须找到一种使得测试可以方便地访问客户端业务规则的方法，这种方法就是：把客户端业务规则和 UI 进行隔离，使得测试总线 API 可以调用这些业务规则。分离 UI 和业务规则一直以来都是一个好的设计目标：“模型—视图隔离原则告诉我们，模型（领域）对象不应当和视图对象有直接的依赖关系”。强制的测试总线要求会把好的设计目标变成一个必须要实现的需求。

## 隔离测试总线

基于每个 API 写的测试应当仅仅测试该 API 所涵盖的东西。针对 Presentation API 编写的测试不应该直接测试业务规则 API 涵盖的内容。原因和不要通过 UI 去测试业务逻辑一样。如果你通过 Presentation API 去测试业务规则，那么 Presentation 层就会变得难以改变，因为这些改变会破坏测试。

这里，我们再一次看到，把测试总线当成一种强制要求会迫使我们达成好的设计目标。在子系统和层级间进行解耦一直以来都是良好设计的特征，不过当我们用自动化测试来明确系统的行为时，这种解耦就变成了一个需求，而不仅仅只是一个设计目标（请参见另外一篇对此有着深刻见解的文章）。

当然，有些端到端的测试也是必要的。这些测试描述了那些分离的系统模块是如何协作的。然而，绝大多数的测试都必须针对它们对应的子系统去写。

另外，一个自动化测试环境只有在方便运行的前提下才能说是有用的。如果运行一个自动化套件需要花去整整 8 天，那么开发人员肯定不会经常运行它。测试运行得越不频繁，两次运行之间累积的缺陷就会越多。那么开发人员就需要更多漫长的测试运行才能验证他们修改了哪些缺陷。不过，如果运行整个测试套件需要的时间不足一小时，那么一天就可以运行好几次测试。因为开发人员可以很快地发现缺陷并且修复它们，所以，系统中的缺陷也就不那么容易累积。

通过被隔离的 API（而不是 UI）运行这些测试极大地加快了测试的运行速度，使得测试可以在每次构建后都运行，一天能运行很多次。

## 隔离数据库

数据库是测试运行慢的另外一个原因。在大型数据库上的操作会比较慢，而同样的操作在小型数据库上就可能快很多。这里有一个简单的加速测试的策略，就是在一系列小型的、预先准备好的数据库上运行测试。测试系统只在每次测试前创建这些数据库，测试运行完毕即删除它。

即便是小型数据库，也需要通过磁头的旋转从磁盘上读写数据。这些磁盘操作和 RAM 上的同样操作相比，慢得不是一个数量级。所以，另外一个加速测试运行的策略就是从应用中将数据库分离出去，在测试中用一个只在 RAM 中存在的数据库替换它。

图 3 展示了我们可以创建另外一个 API 来达到这个目标，业务规则可以使用这个 API，无论是真实的数据库还是 RAM 中的数据库都来实现这个 API。

速度不是这种隔离带来的唯一好处。在 RAM 中运行测试用例能够让测试完全控制数据库中的内容。测试首先使用一个空白的数据库，调用特定的启动函数将 RAM 数据库设置到一个已知的状态。而且，如果测试在一个 RAM 数据库上运行，那么它们对数据的修改是不会持久化的，这使得测试既是独立的又是可重复运行的。

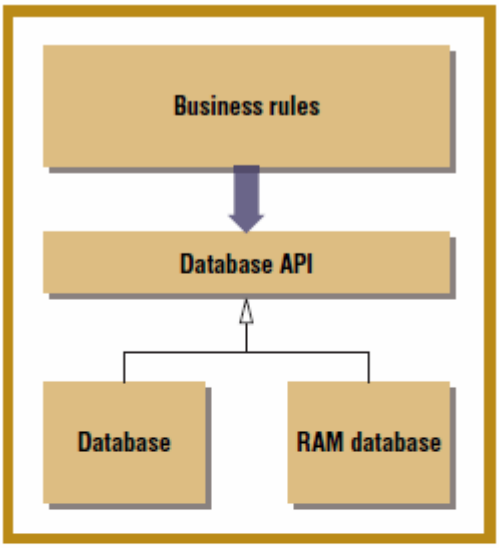


**Figure 3. Creating another API that the business rules can use and that a real database or RAM database can implement separates the database from the application. We can then replace the database with one that exists solely in RAM to keep the tests running quickly. (The smaller arrow represents a UML inheritance relationship.)**

测试的独立性和可重复性使得我们可以快速隔离并诊断问题。此外，当我们可以很方便地一遍遍以任何顺序运行单独的测试或者一组测试时，问题的诊断也会加速。

当测试相互依赖时，每次我们想要运行某个特定的测试，我们都必须运行整个测试套件。而且，一旦一个测试失败，它下面的测试也必定要失败，这些都阻碍了问题的隔离和诊断。

再一次，我们看到了好的设计原则和强制测试总线之间的关系。将数据库和应用隔离一直以来都是良好设计的一个原则：按照 Martin Fowler 的说法，“将 SQL 的存取从领域逻辑中隔离出来放到另外的单独类中是明智的。”此外，Ivar Jacobson 说，“要让 DBMS 对设计的影响最小化，那么系统中应当知道 DBMS 接口的部分越少越好。事实上，很多框架和分层系统都是按照这个分离原则来编写的。对测试速度，测试可重复性以及测试独立性的要求使得这个分离成为了一个至关重要的架构需求。

## 小尺度上的隔离

在一个真实的测试驱动环境中，对隔离的要求会延伸到更低的层次，从主要的子系统到模块，到类，甚至到方法。作为编写验收测试的业务分析师，质量保证专家，和测试人员，以及编写单元测试的开发人员，将软件隔离成测试可以独立访问和操作的单元，已经成为一个急迫的需求。而这又进一步要求架构师，设计人员以及开发人员具有更高、更好的面向对象设计技能。面向对象的设计为我们提供了工具，原则以及模式，使得以上提到的自动化测试需要的那些隔离能够达成。

软件工业界日益认识到自动化测试的好处和潜力，这对十多年来一直被质量和生产力问题困扰的这个行业来说是个好消息。

我们的客户所处的行业（比如：电话，电子，以及制造）在一个多世纪以来都将产品中包含测试装置作为其过程的一个标准组成部分，和他们相比，我们这么晚才认识到这个事实实在是有点讽刺意味。

更有讽刺意味的是，当我们把可测试性作为一个强制的架构设计要求时，它会迫使我们去遵循好的设计原则并降低我们系统的耦合。