

The Test Bus Imperative:

Architectures That Support Automated Acceptance Testing

Robert C. Martin

Agile methods—more specifically, test-driven development practices¹—have begun to raise the software industry’s awareness of automated acceptance testing. We now recognize that armies of manual testers slogging through reams of tedious, manually executed test scripts, under schedule pressure at the end of a release, don’t ensure quality.



Many tools can be purchased to help testers write automatic scripts for testing the system through its user interface. Such tools act like robots programmed to behave like testers—they push buttons, select menu items, check boxes, enter data into fields, and inspect the screens. Unfortunately, testing through the UI is slow,

opaque, and dangerous. One of my clients had to run over 10,000 acceptance tests, and running them through the UI took him several days using dozens of machines. Furthermore, the tests were written in a code-like language that made them difficult to understand. Over time, he lost track of what each test was trying to verify: all he knew was that all 10,000 tests had to pass. Even tiny changes to the UI caused dozens, if not hundreds, of tests to fail or become inoperable. In the end, he found it impossible to upgrade and improve his outmoded UI.

Over a century ago, the telephone company solved a similar problem by designing telephone switches with built-in test access. Such

access, called a *test bus*, eventually let the telephone company automatically test every phone line at night and fix failures and degradations long before subscribers learned about them.

Software producers have started adopting a similar strategy, architecting their software systems with built-in test buses. They also use tools such as Fit and FitNesse (www.fitnesses.org) to specify their tests in a simple-to-understand specification language that business people can read.² However, setting up automated acceptance testing requires at least three major architectural separations.

Bypassing the UI

In a software system, a test bus is a set of APIs that provides convenient access to unit and acceptance tests. These tests specify system behavior—unit tests specify module behavior and acceptance tests specify feature behavior.

A test bus’s existence in a system means that the appropriate structures exist in the application to let tests access the modules and subsystems whose behavior and structure they’re specifying. For example, writing tests that bypass the UI and exercise the underlying business rules requires an API that both the UI and tests can use to invoke those business rules. Moreover, that API must be independent of the UI (see figure 1).

Figure 1 clearly shows that the tests are an alternate form of UI. They access the same API that the UI uses, so they can drive the system through all of its revenue-bearing operations and specify the required behaviors in detail.

However, some UIs are very rich, and busi-

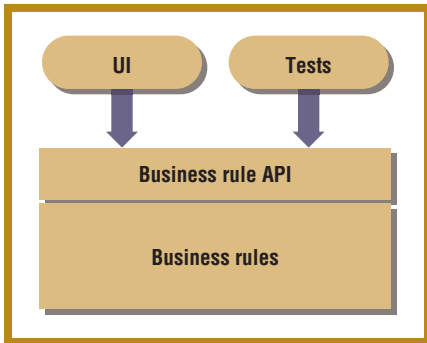


Figure 1. A testable system includes a test bus that can access the API independent of the UI.

ness rules tend to find their way into them. For example, some systems validate data or perform significant calculations in the UI. This is especially true in client-server and Web-based systems. Some common design strategies make such business rules difficult to test. However, if the development team has decided that testability is an architectural imperative, then they must provide test access to those business rules. A common solution is to create another API in the UI that separates the presentation, validation, and calculation rules from the UI's low-level details (see figure 2).

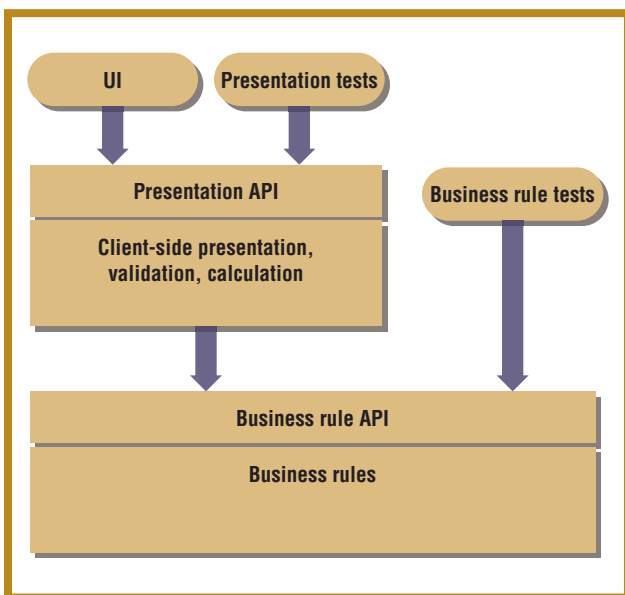


Figure 2. The presentation API separates the presentation, validation, and calculation rules from the UI's low-level details. This helps ensure developers can test all the business rules—even those that extend into the UI.

Clearly, this kind of API constrains the UI design and how designers can use client-side tools such as JavaScript. Designers must find a way to let tests conveniently access these client-side business rules by separating them from the UI so that a test bus API can invoke them. Separating the UI and business rules has long been a goal of good design: “The Model-View Separation principle states that model (domain) objects should not have *direct* knowledge of view (presentation) objects.”³ The test bus imperative turns that good-design goal into a requirement.

Isolating the test bus

The tests written against each API should test only those things governed by that API. Tests written against the presentation API shouldn't test business rules that the business rule API governs. The arguments for this are precisely the same as those for not testing through the UI. If you test the business rules through the presentation API, then the presentation layer becomes difficult to change, because such changes break the tests.

Here again we see how the test bus imperative enforces good design goals. Decoupling subsystems and layers has

always been a hallmark of good design, but when automated tests are used to specify the system, this decoupling becomes a requirement, not just a goal. (A particularly insightful discussion of this appears elsewhere.⁴)

Some end-to-end tests are certainly necessary. Such tests describe how separate system modules interplay. However, the vast majority of the tests must be written against their respective subsystems.

Additionally, an automated test environment is useful only to the extent that it's

convenient to run. If running the automated suite takes eight days, developers won't run it very often. The less often it runs, the more defects will accumulate between runs. Then the developers will need to execute more long runs to verify that they've fixed those defects. However, if running the entire suite of tests takes less than an hour, the tests can run several times a day. Defects don't accumulate in the system because developers quickly find and repair them.

Running these tests through isolated APIs (rather than the UI) drastically increases the test runs' speed and makes it feasible to run the tests at every build, several times per day.

Separating the database

The database is another cause of slow tests. Operations on large databases can be slow, whereas the same operations on small databases can be much faster. One simple strategy to speed testing is to run the tests on a series of small canned databases. The test system creates these databases just before the tests run and deletes them just after.

Even small databases are marshaling data on and off rotating magnetic disks. These operations are orders of magnitude slower than equivalent operations in RAM. So, another strategy to keep the tests running quickly is to separate the database from the application and replace it with a database that exists solely in RAM.

Figure 3 shows that we can achieve this separation by creating another API that the business rules can use and that either a real database or RAM database can implement.

Speed isn't the only benefit of such a separation. Running test cases in RAM lets the tests completely control the database's content. Tests begin with blank databases and invoke special setup functions that put the RAM database into a known state. Moreover, if tests run on a RAM database, the changes they make to that data aren't persisted. This makes the tests both independent and repeatable.

Test independence and repeatability let us quickly isolate and diagnose prob-

lems. Furthermore, when we can conveniently run any individual test or group of tests over and over, in any order, it enhances the diagnoses.

When tests depend on each other, we must run the entire suite every time we want to run a particular test. Moreover, if a test fails, the downstream tests necessarily fail, greatly impeding problem isolation and diagnosis.

Once again, we see the relationship between good design principles and the test bus imperative. Separating the application from the database has long been a principle of good design: According to Martin Fowler, “It is wise to separate SQL access from domain logic and place it in separate classes.”⁵ Furthermore, Ivar Jacobson says, “To make the design minimally affected by the DBMS, as few parts of our system as possible should know about the DBMS’s interface.”⁶ Indeed, many frameworks and layers have been written to assist in this separation. The need for test speed, repeatability, and independence makes this separation an essential architectural requirement.

Separation in the small

In a truly test-driven environment, the need for separation extends below the major subsystems to the modules, classes, and methods. As business analysts, quality assurance professionals, and testers write acceptance tests, and developers write unit tests, the need becomes acute to separate the software into units that those tests can independently access and operate. This greatly increases the need for good object-oriented design skills among the architects, designers, and developers. Object-oriented design provides the tools, principles, and patterns that enable the kinds of separation that automated testing requires.

The software industry’s increased awareness of automated testing’s benefits and potentials is good news for an industry that for decades has been beset by quality and productivity issues. In light of the fact that many of our client industries such as telephony, electronics,

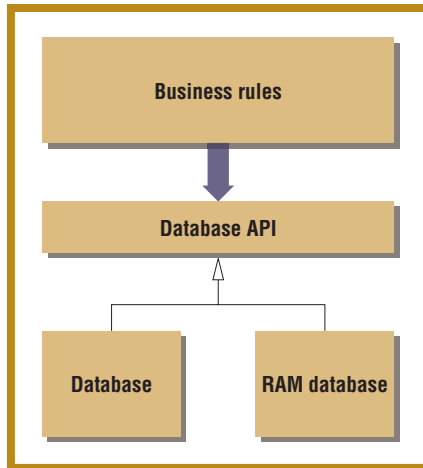



Figure 3. Creating another API that the business rules can use and that a real database or RAM database can implement separates the database from the application. We can then replace the database with one that exists solely in RAM to keep the tests running quickly. (The smaller arrow represents a UML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

and manufacturing have been designing test access into their products as a standard part of their process for over a century, it’s ironic that we’ve come to this awareness so late.

More ironic still is the fact that when we make testing an architectural imperative, it forces us to follow the principles of good design and decouple our systems. 

References

1. K. Beck, *Test Driven Development*, Addison-Wesley, 2003.
2. R. Mugridge and W. Cunningham, *Fit for Developing Software*, Addison-Wesley, 2005.
3. C. Larman, *Applying UML and Patterns*, Prentice Hall, 2002, p. 472.
4. M. Feathers, *The Humble Dialog Box*, 2002, www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf.
5. M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003, p. 34.
6. I. Jacobson, *Object Oriented Software Engineering*, Addison-Wesley, 1992, p. 276.

Robert C. Martin (Uncle Bob) is the founder, CEO, and president of Object Mentor. Contact him at unclebob@objectmentor.com.

How to Reach Us

Writers

For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access www.computer.org/software/author.htm.

Letters to the Editor

Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

On the Web

Access www.computer.org/software for information about *IEEE Software*.

Subscribe

Visit www.computer.org/subscribe.

Subscription Change of Address

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

Membership Change of Address

Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

Missing or Damaged Copies

If you are missing an issue or you received a damaged copy, contact help@computer.org.

Reprints of Articles

For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

Reprint Permission

To obtain permission to reprint an article, contact the Intellectual Property Rights Office at copyrights@ieee.org.