Realtime publishers

The Definitive Guide™ To

Quality Application Delivery

sponsored by

MICRO FOCUS®
Leading the Evolution™

Don Jones

Realtime publishers
MICRO FOCUS Leading the Evolution™

Copyright Statement

© 2009 Realtime Publishers. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtime Publishers (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtime Publishers or its web site sponsors. In no event shall Realtime Publishers or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtime Publishers and the Realtime Publishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtime Publishers, please contact us via e-mail at info@realtimepublishers.com.

[**Editor's Note:** This eBook was downloaded from Realtime Nexus—The Digital Library for IT Professionals. All leading technology eBooks and guides from Realtime Publishers can be found at http://nexus.realtimepublishers.com.]

Chapter 6: Design—Building Quality In

At this point, we've *thoroughly* covered the idea that solid, well-written, business-aligned requirements are the first key to delivering a quality application. But requirements in and of themselves can't simply be turned over to a team of programmers. Requirements are missing crucial factors that developers need to get started: *design decisions.* For all the emphasis that I've placed on requirements thus far in this book, I need to acknowledge that requirements are purposely selfish, focusing primarily on the things the business needs. Requirements should deliberately steer clear of technical specifications, leaving that up to a *design document.* The design is intended to map those business requirements to technical specifications.

> **Note**
> I've worked on more than one project where the "business requirements" did, in fact, include technical specifications. The requirements might state that a specific tool set would be used for development, or that GUI applications would use such-and-such a third-party library for charting and graphing. These "requirements" nearly always stemmed from what I call "political motivations" and not actual business requirements. Often driven by upper-level management decisions, these are unfortunate, but their importance to management—perceived or otherwise—makes them requirements nonetheless.

The design has a tremendous impact on quality. Although I feel it's of utmost importance for developers and testers to have access to the original requirements document, a well-written design should remove any real *need* for them to read it regularly.

> **Note**
> Why do developers and testers need access to the business-level requirements at all? Typically, developers and testers are less aware of how the business is run, at a detailed level, than the people who contributed to the requirements. The design tells developers and testers *what* to do; in the event that there's some question the design doesn't answer, the requirements can tell the team *why* they're doing something a particular way, helping them make better independent decisions.

In an article on Builder.au.com, Robert Brogue emphasizes that designs should focus on the *how*, not the *why*:

> …the answer is that you should be focusing on what must be done to complete the solution. For instance, in an e-commerce Web site, you will have to handle users forgetting their passwords. That is the "what must be done" part of the design. Obviously, the person must receive some sort of verification information via e-mail and return it to the site. As to how that is solved, cryptographically sound tokens, hashes, and the like are not a part of a high-level design. They are details best left to the detailed design or to program specifications.
>
> As another example, if the requirement is that the system must have the ability to secure specific content, you need only describe a security mechanism, which has these requirements. You need not indicate the development of a security system based on GUIDs where all items in the system have a unique GUID that can be assigned to a user or group. The introduction of the "How" (using GUIDs) into a design document is an unnecessary addition of information that does not help the architecture or design. However, it solves your competitors need for a way to solve the problem.
>
> The focus of a design document is on "what is needed" to satisfy the requirements (what the user wants and what the environment requires). Any other information in the design document detracts from its primary purpose (Source: http://www.builderau.com.au/strategy/architecture/soa/Application-Design-Writing-design-documents-that-cannot-be-stolen/0,339028264,320282726,00.htm).

In this chapter, we'll focus on writing design documents that not only help developers and testers understand *how* they need to write and test the application but also how they can do so in a quality manner.

Designing *to the Requirements*

Perhaps the most important thing about a design document is that it be strongly tied to the application requirements. As I've said, the design *should* answer every question a developer or tester might have about what it is they're doing, and it should do so in a way that ensures the business requirements will ultimately be met.

Most design authors are aware of these facts, and may even be accustomed to using the requirements document as a kind of checklist. After completing their design, they can go through the requirements and make sure each thing is addressed. However, in my experience, what many design authors aren't accustomed to doing is permanently tying their design to the requirements. Take a look at Figure 6.1, which shows a portion of a sample requirements document.

The CRM application will be used primarily by the remote sales force, but will also be used by HQ and WAN site personnel. Access to this application should be available between the hours of 6am and 10pm CST Monday through Friday. Ninety percent uptime is expected during these hours.

Table 1: Availability Requirements Summary

Application	Required Uptime	Target Uptime	Hours of Operation	Access*
ERP Application	95%	99%	12am – 11pm CST M-Sa	HQ, W
EMAIL	80%	95%	6am – 10pm CST Su-Su	W, H, R
Office Suite	75%	85%	6am – 10pm CST Su-Su	H, R
Project Mgmt. Software	80%	90%	6am – 6pm CST M-F	HQ, W
CRM Application	90%	95%	6am – 10pm CST M-F	HQ, W

* HQ = Headquarters, W = WAN sites, H = Home Users, R = Remote Users (Traveling Users)

Scalability

CompanyX has plans to acquire several new companies over the next few years. The Metaframe environment is viewed as the quickest and most cost effective method to provide application access for acquired companies. It is currently estimated that the number of ERP users will increase from 200 to 400 over the next eighteen months, E-mail users from 300 to 600, Office Suite users from 150 to 300, Project Management software users from 50 to 75, and CRM application users from 50 to 75. It should be noted that many employees would be using multiple applications. The total user base for this solution is expected to grow from 400 to 700, and the maximum concurrent users from 200 to 350. Below is a table that estimates application user growth in three-month increments for the next eighteen months.

Table 2: Growth Projections

Application	Number of Users				
	3 Months	6 Months	12 Months	15 Months	18 Months
ERP Application	200	250	350	350	400
EMAIL	350	400	500	550	600
Office Suite	150	175	225	250	350
Project Mgmt. Software	50	50	65	65	75
CRM Application	50	50	65	65	75
Total	450	500	600	650	700
Maximum Concurrent	200	225	300	325	350

Figure 6.1: Sample requirements document.

Note

Design documents may also be referred to as *specification, functional specifications,* and other terms. I'll use "design document" in this chapter and throughout most of this book.

Now consider the sample shown in Figure 6.2. There's a single significant difference between these requirements in terms of their format and layout—can you spot it?

Realtime publishers

MICRO FOCUS Leading the Evolution™

2.4 The administrator is also responsible for entering customization information into IMS, such as the standard working day for the user organization.

3 Functional Requirements

3.1 Scheduling a meeting

3.1.1 A person calling a meeting (henceforth called the Initiator) will enter information into IMS about the desired meeting, such as but not limited to its proposed purpose, earliest and latest times at which it can usefully be held, the names of desired attendees, and an anticipated duration. IMS shall provide defaults for missing elements.

3.1.2 When the Initiator instructs IMS to schedule the meeting for which this information has been entered, IMS shall obtain from those OLCs that contain schedule data for the desired attendees those attendees' free times during the interval between the earliest and latest times stipulated by the Initiator. IMS shall choose the earliest feasible time for the meeting that meets the constraints specified by the Initiator and the free times returned by the OLCs.

3.1.3 When IMS chooses a time for a meeting, it shall send queries to the OLCs for all the rooms in which the meeting could be held to ascertain which rooms are vacant during the selected time.

3.1.3.1 If no rooms are vacant during the selected time, IMS shall choose the next feasible time.

3.1.3.2 IMS shall choose which room should be the venue for the meeting from the set of rooms free at the selected time by a room-choice algorithm that shall take account of the size of the room given the number of invitees to the meeting, and the convenience of the room for the invitees.

3.1.4 If no feasible time exists for the meeting that meets the Initiator's constraints and the OLC schedule data for the attendees, or no room is free during any of the feasible times, IMS shall present the Initiator with a selection of choices including the following:

(a) Schedule the meeting based on a subset of the originally named attendees.

(b) Put the latest time for the meeting back further into the future.

(c) Abandon the scheduling of this meeting.

3.1.5 If either 3.1.4 (a) or (b) is chosen, IMS shall obtain from the OLC's in question further schedule data and choose the best feasible time as described above. If no feasible time is found again, IMS shall present the Initiator with the

Figure 6.2: Second sample requirements document.

Realtime publishers

MICRO FOCUS Leading the Evolution™

In my view, the first sample may do a good job of spelling out requirements, but it doesn't lend itself to driving quality throughout the application-development process. The second document *does* do a good job, for one simple reason: ***Each requirement is clearly labeled with an identifying number.*** You may think that's pointless, but consider how those identifiers impact the design and the subsequent development and testing:

- The design can clearly map each design decision back to requirements. That way, if there's any trouble or dispute about the design decision later, anyone can clearly trace the requirement that drove it. If a design decision needs to be revisited, that can be done based on the original requirements—without trying to guess what the design author was thinking.
- The design author can easily ensure that each requirement is addressed by simply making sure each identifying number is covered in the design.
- Developers who have a question about the intent of a design decision can easily refer back to the driving requirement. The requirements document is the intent behind the application.
- Rather than testing only for functionality, testers can more easily test to the requirements because each requirement has an identifying number. Testers can, for example, indicate that the test for requirement 3.1.3.1 passed, while the one for 3.1.3.2 failed. This allows testing to generate a concrete quality score: the application meets (for example) 92% of the requirements.

The moral here is that the design *must*—not should, may, or any softer word, but *must*—map each design decision back to the driving requirement. A good, well-labeled requirements document facilitates this.

But wait—what about design decisions that are more arbitrary and don't derive from the requirements? Let's say that the designer needs to make a technology decision that isn't specifically driven by a requirement and doesn't seem to specifically impact the requirements. For example, deciding to use ASP.NET over PHP in a Web application might seem completely arbitrary and detached from the business requirements.

But is it? The choice of programming language affects numerous business issues, such as long-term maintenance of the code, business relationships with companies such as Microsoft, and so forth. In this case, the designer would do well to go back to business decision-makers and present them with choices, document their decision in a revision to the requirements document, and update the design document accordingly.

In my experience, a designer who thinks, "I need to make a decision, but there's nothing in the requirements to drive it" needs to immediately decide, "I need to go back to the decision-makers on this one." Designers who make decisions in a vacuum are opening the door to lowered quality in the application. Even seemingly minor technology choices can and should be driven by business decisions. For example, a designer may have to choose between different types of database drivers—surely the business doesn't care whether we use brand "X" or brand "Y," right? But what *drives* the decision between the two? Is it price? Performance? Corporate relationship? *Those are all business decisions,* and a designer shouldn't be making those on his own.

Earlier, I stated that the design document should address every single requirement; there's a corollary: Every design choice *must* map back to a requirement. If the mapping isn't clear, then it should be stated in the design. For example, rather than this:

> 3.2.1 Based on requirement 7.8.2.1, we will be using ACME brand database drivers rather than the framework's native database drivers.

A clearer design might read something like this:

> 3.2.1 Requirement 7.8.2.1 specifies performance requirements for the application, and requirement 11.12.3 specifies the capacity of the network environment. Based on these requirements, we will use ACME brand database drivers. Their performance profile offers what we need.

The second example isn't much longer, but it spells out that the decision to go with the ACME drivers is based on their performance, and it ties in an additional requirement that helps explain and justify the decision. If there's ever any need to question this design choice, the *why* that drove it will be clear.

Note

It can become awkward to keep track of all the requirements that each decision maps back to. That's where commercial applications come in: You can obtain applications that let you document the requirements and design choices in a sort of database, connecting each design choice to its driving requirements. The application can then produce clearer documents more easily, and lends itself to ongoing work and maintenance.

Designing Quality Points from the Beginning

A well-planned design document can, however, do a lot more than just make sure the initial requirements are met. A good design document can actually include quality checkpoints, metrics, and controls so that the entire development and testing process can continually self-check to make sure they're hitting the requirements. Quality points are especially necessary for some business requirements; not so necessary for others. For example, consider a requirement like this one:

> 5.12.9 The application must provide a means of archiving data that is older than 90 days. Archived data does not need to be immediately available to users in the main portion of the application, but must be available through a separate query/lookup mechanism.

That's a pretty cut-and-dried requirement. Anyone could look at the final application and determine whether that requirement had been met. The design would simply need to focus on *how* that requirement was met: what the user interface would look like, where archived data would be stored, what storage format would be used, how data would be archived, and so forth. But now consider this requirement:

> 7.1.12 The application must be able to retrieve customer data within 10 seconds when the sales agent specifies a customer phone number or customer number.

That seems pretty cut and dried but what can a designer specify that will *guarantee* that level of performance? So many additional factors come into play: The condition of the production network (which is something that the requirements should outline in a description of the operating environment), the condition of the database, and so forth. This is a very difficult requirement to actually create a design for, since you're going to need developers to specifically test their compliance to the requirement as they develop. This is exactly where a quality point can be added, in a design statement like the following:

> Requirement 7.1.12 specifies maximum response times for data retrieval. Assuming that the production network and slow client computers may add up to 3 seconds' delay, developers must ensure that the code provides a response within 7 seconds of receiving a request. This must be tested against Test Database 3-B using Test Scenario 3. Response times greater than 5 seconds must be reviewed by development managers.

This sets out a clear design goal, and provides instructions to testers and developers. It also promises to provide realistic test data and a realistic test scenario. Having something like this in the design documents puts everyone on notice that there's a firm performance commitment. Quality checkpoints like this can be built into numerous aspects of the application – not just for performance-related issues. Consider these example design statements that serve as quality checkpoints:

- **Usability:** "Each dialog box or screen of the application must be tested for compliance with the company's accessibility guidelines. The application must be completely operable without the use of a mouse or other pointing device."
- **Compatibility:** "The application must run on all operating systems (OSs) specified in Requirement 23.4.2, and when running must consume less than 800MB of memory in order to conform to Requirement 23.4.8."
- **Security:** "The application may not use dynamically constructed SQL queries. All parameterized queries must be built as stored procedures."

When mapped to a corresponding business requirement, these design goals provide specific points that can be unambiguously tested to ensure that requirements are being met. That last one is a pretty straightforward design decision but it's something that a tester couldn't normally verify when looking at compiled code. Having a requirement like that cues testers to the fact that they need to verify this by performing a code review, for example.

Designing for Code Quality

You might think that "code quality" is something best left to developers. Although it's certainly true that you can't get high-quality code without committed developers, it's not true that amazing developers can *always* turn out quality code entirely on their own: They need a good design to start with. There are a number of things a good design can contribute to help improve code quality:

- Provide pseudo-code for critical algorithms and processes. Doing so makes key components of business logic part of the design; it's up to developers to implement those in whatever technologies they're working in.
- Specify standards for checked-in code. Doing so allows the entire project to insist that all code be of high quality. Standards might include coding conventions such as variable naming, construct formatting, and so forth, and it might also include standards around unit testing and other quality measures.
- Your design document should go as deep as specifying functional units of code, such as programming objects, modules, and so forth. It should also specify unit tests for these functional units so that developers can know when their code is ready to be checked in.

Specifying and managing components and their interactions right in the design is a major contributor to quality. For example, Figure 6.3 shows the relationships between four components and a component that they all depend upon.

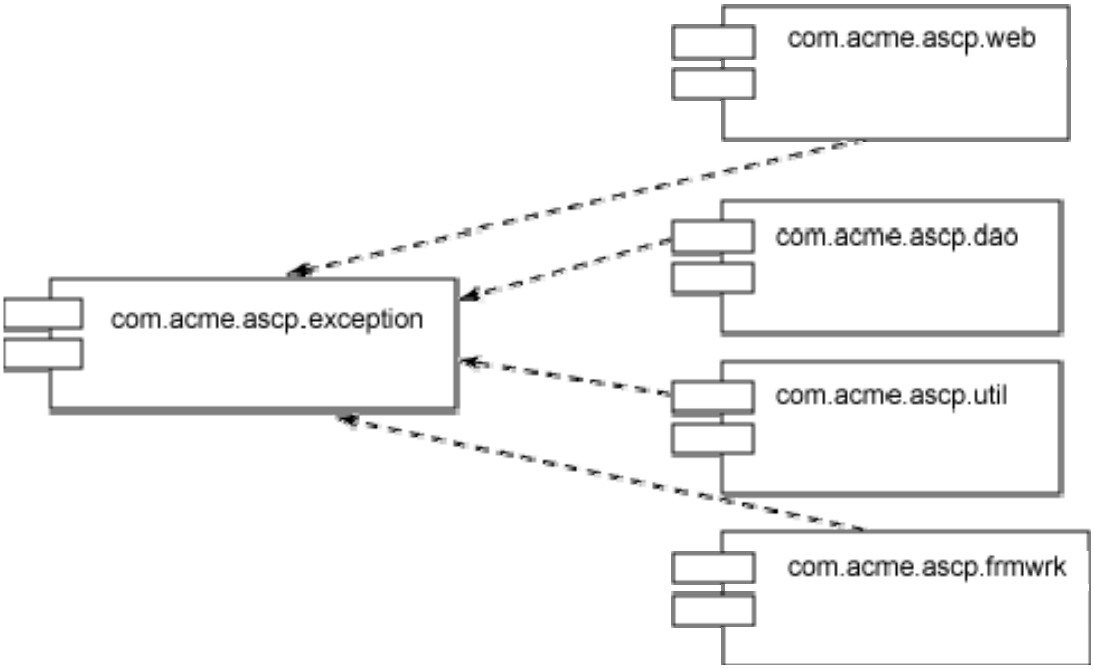

Figure 6.3: Relationships between components.

By documenting these components, you not only help modularize the overall application, but you also help manage ongoing change to the application. Knowing that changes to the "Exception" class may require review and reprogramming in its four dependent classes is important information. Too often, code quality is diminished by a "minor change" to a piece of code, without sufficient insight into how that change may cascade into major problems elsewhere in the application.

The relationships between units of code are referred to as *coupling;* Figure 6.3 is an illustration of *afferent coupling*. Specifically, the "exception" class has an afferent coupling of four, indicating that four components depend upon it. Components with a high afferent coupling are extremely critical to the application; identifying this fact in the design can help ensure that extra attention and care is given to this central piece of code.

By explicitly designing components and their couplings, a design can help avoid *entropy*, which is the unexpected and unplanned-for coupling between components. Figure 6.4 shows an example: a third-party billing component, on the left, was intended to support a custom billing component, which is on the right in the middle. However, during the development process, other components—DAO and WEB—took dependencies on the third-party component. Entropy can be damaging to the overall application because it complicates changes to components; in this example, swapping out the third-party billing component might be problematic, even if the original in-house billing component was carefully designed to support replacement of its dependency.

Realtime publishers
MICRO FOCUS Leading the Evolution™

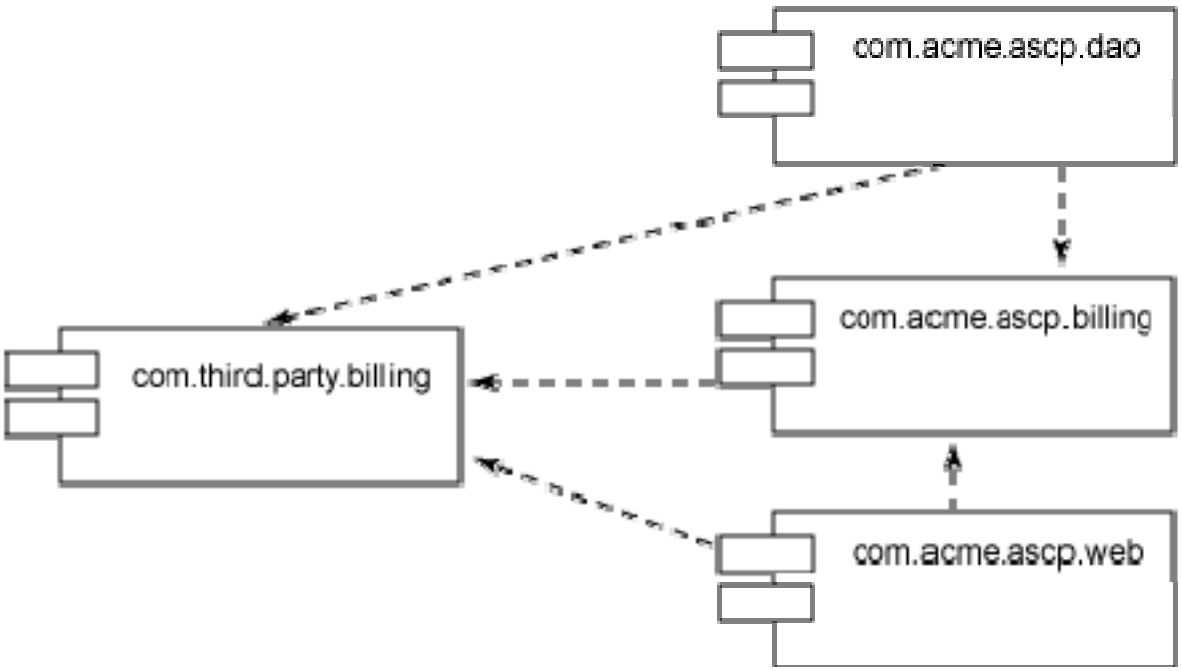

Figure 6.4: Diagramming entropy in code coupling.

Figure 6.4 illustrates another design concept that can lead to quality code: *flexibility.* Using third-party components, such as this billing component, is a common practice—why re-invent the wheel? However, by writing your own interface component—sometimes called a *wrapper*—you achieve a few benefits. First, your wrapper—the "ACME Billing" component in Figure 6.4—can be written for the specific needs of your application, making things easier on other developers on the team. Second, provided that only your wrapper depends on the third-party component, the third-party component can be swapped out more easily.

> **Note**
> Again, commercial software can help keep track of all the code units and their relationships, and can provide statistics on coupling.

A final way that the design can help produce quality code is to help diagram *efferent* coupling, which is essentially the number of dependencies a component has. Consider Figure 6.5, in which the DAO component has an efferent coupling of three—meaning it depends upon three other components.

Realtime publishers
MICRO FOCUS Leading the Evolution™

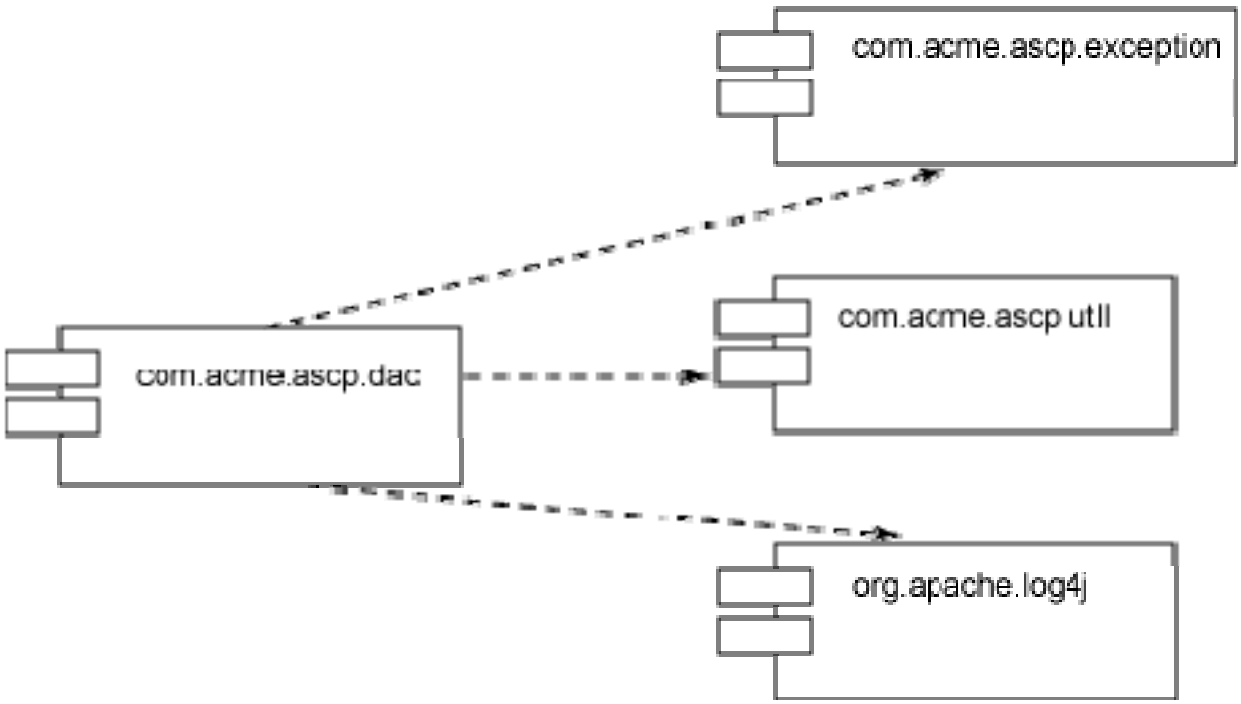

Figure 6.5: Examining efferent coupling.

Keeping track of all these relationships is an important part of the design, and it can really help drive quality coding by setting boundaries for developers, componentizing code to make unit testing more effective, and insulating components of code from one another to reduce cascades of changes and defects.

> **Cross Reference**
> I'm also the author of *The Definitive Guide to Building Quality Code* (Realtime Publishers), which goes into more depth—from a developer perspective—on code quality. Get it free from www.realtimenexus.com.

Designing for Performance

In almost a decade of experience, I could count on one hand the number of projects I've worked on that included design specifications for application performance. Yet performance is often one of the top two or three things that lead to a perception of poor application quality! Even giants like Microsoft have run afoul of performance: When they first released Windows Vista, they provided hardware requirements that were, in reality, a bit too generous. The result was users trying to run the brand-new operating system on barely-suitable hardware—and a lasting perception of poor quality that the company spend tens of millions fighting.

Realtime publishers
MICRO FOCUS Leading the Evolution™

In my experience, designers often ignore performance for three reasons: First, it's difficult to design for. Second, they feel that *eventually*, hardware and other environment factors will catch up. Third—and most important—is that their requirements didn't specify any performance metrics. Hopefully in the previous chapter I put sufficient emphasis on performance as a part of the requirements document. The requirements should also specify the environment that the application will run in, so that eliminates the latter two excuses and leaves us with "difficult to design for."

First, I'll acknowledge that unless the requirements *states* performance goals, then there's little a designer can do to help. You can always design for better performance, but at a cost in time and effort; a designer relies on clear requirements to help draw the line between performance and cost. When I'm tasked with designing an application, I look for performance requirements first. If I don't, I won't proceed until the requirements are modified to include performance!

Once the requirements are clear on the level of performance required *and* the environment in which that performance must be achieved, the design can take over. There are numerous techniques and considerations a designer has to help design for a desired level of performance:

- Scaling up on larger systems, or scaling out across many systems
- Selecting communications components, such as networking stacks or database drivers
- Managing concurrency in data access
- Specifying caching to reduce communications requirements
- Improving algorithms to require less processing
- Leveraging multi-tier architecture; for example, designing database queries to best leverage the capabilities of the database back-end
- Replicating data to reduce communications bottlenecks
- Partitioning data to reduce communications bottlenecks and to increase scalability
- Careful examination of blocking operations – those which necessarily suspend processing while waiting on a dependent process, and which can create a perception of poor performance simply because the end-user can't see the background processing in action

So here's how the design needs to lay things out: The requirements must specify some desired level of performance. The designer must estimate any performance overhead from out-of-control components such as the environment (client computers, network, and so forth), creating for themselves a performance cushion, which is deducted from the desired performance metric. The remaining room in the performance metric becomes a testing goal, from unit testing through final acceptance testing. The design *specifically* needs to indicate how that performance testing will occur: In what conditions, using what data, and under what load—something we'll discuss more in the second half of this chapter.

Designing for Maintenance

In addition to working as a developer, I also have a background in systems administration. One of the most common laments of administrators is that *nobody designs software to be maintained.* Unfortunately, software that can't be maintained is often perceived as poor quality software, but administrators aren't entirely correct in that nobody *designs* for maintenance; in fact, few requirements documents I've read specify any *requirements* for maintenance.

However, assuming that you're working with a good set of requirements that *do* include maintenance criteria, a designer can have a significant contribution to the long-term maintainability of the application. I'm not referring to maintainability of the code, by the way; while that's obviously important, too, I'm focusing on the maintenance tasks that keep the application up and running smoothly over the long haul.

Maintenance tasks can include:

- Archiving data
- Deleting old data
- Backing up and restoring data
- Managing user access and permissions within the application
- Monitoring the application's health and performance
- Managing the application's configuration and operating parameters
- Maintaining primarily static data such as lookup tables
- Deploying the application (and subsequent patches or updates)

Some estimates suggest that application maintenance consumes 50 to 70 percent of IT budgets; my experience supports that. In other words, most administrators spend most of their time maintaining applications. The problem is that writing administrative interfaces and maintenance code is, frankly, boring and tedious. Developers don't like to do it, and so they *won't* do it unless the application design includes clear specifications.

Here are some ideas for design specifications that address application maintenance:

- Don't wait until the application is finished to code maintenance capabilities. As each unit of code is written, its corresponding maintenance interfaces need to be built at the same time. This allows testers to test the entire application end-to-end.
- Never forget to include performance monitoring instrumentation. In Windows, for example, this is commonly done through performance counters or through Windows Management Instrumentation. Proper monitoring capabilities within the application can help tremendously when it comes to performance tuning, since you're able to "peek inside" the application for key performance metrics.
- Don't assume that a particular form of administration is the universal best: Graphical user interfaces, scripts, and command-line tools all have their place. Ideally, code maintenance capabilities in a form that can be leveraged by various interface types. For example, a set of maintenance code classes could be implemented in command-line tools, which lend themselves to automation, or in a GUI, which is often easier for less-experienced technicians.
- Design a complete disaster recovery plan for the application. Even if you can do so using capabilities which already exist in your platform—such as using a database server's native backup and restore capabilities—your design should clearly indicate how it will be handled.

A designer should try and *foresee* maintenance issues, even if they're not addressed in the requirements. If you're building an application that stores data, for example, you can anticipate that data will accumulate over time. How will that be handled? Will it affect performance? Is there a requirement to archive older data to near-line or off-line storage?

> **Caution**
>
> A set of requirements that doesn't address maintenance is a red flag for designers. Go back with a list of questions, and ask that the requirements be revised to indicate how the business would like those questions addressed. Questions might include, "how will this data be maintained over the years?" and "how will administrators monitor the performance of this application?" Requirements authors may overlook these issues, but a smart designer will help them realize the importance of addressing them.

Designing for Users

In the previous chapter, I described how a requirements document may include suggestions or directives for designing user interfaces. Figure 6.6 shows an example sketch that a requirements contributor might provide.

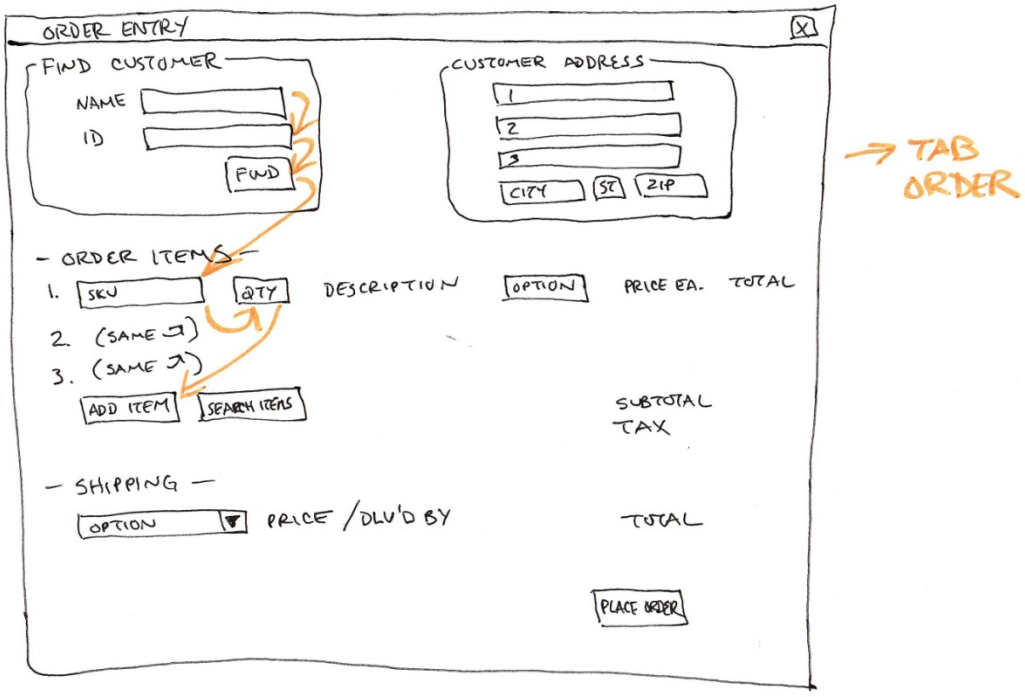

Figure 6.6: User interface sketch.

A designer needs to formalize these types of sketches, and in fact provide details on *every single aspect* of the application's user interface. *Do not* leave UI design entirely to developers to create ad-hoc: Developers shouldn't be doing anything that isn't in the design, and developers should only be *implementing* the design, not designing new things.

Generally, UI sketches shouldn't be treated as the final word; many times, the contributors of such sketches seek only to illustrate workflow and layout, and may not understand all the background requirements that an application has. So consider the spirit of such sketches. For example, Figure 6.7 shows an in-progress, formal UI design that captures the tab order from Figure 6.6. That tab order was clearly important to the person who made the sketch, so it's important to make sure that is communicated through to developers if it's to be in the final application.

Realtime publishers
MICRO FOCUS Leading the Evolution™

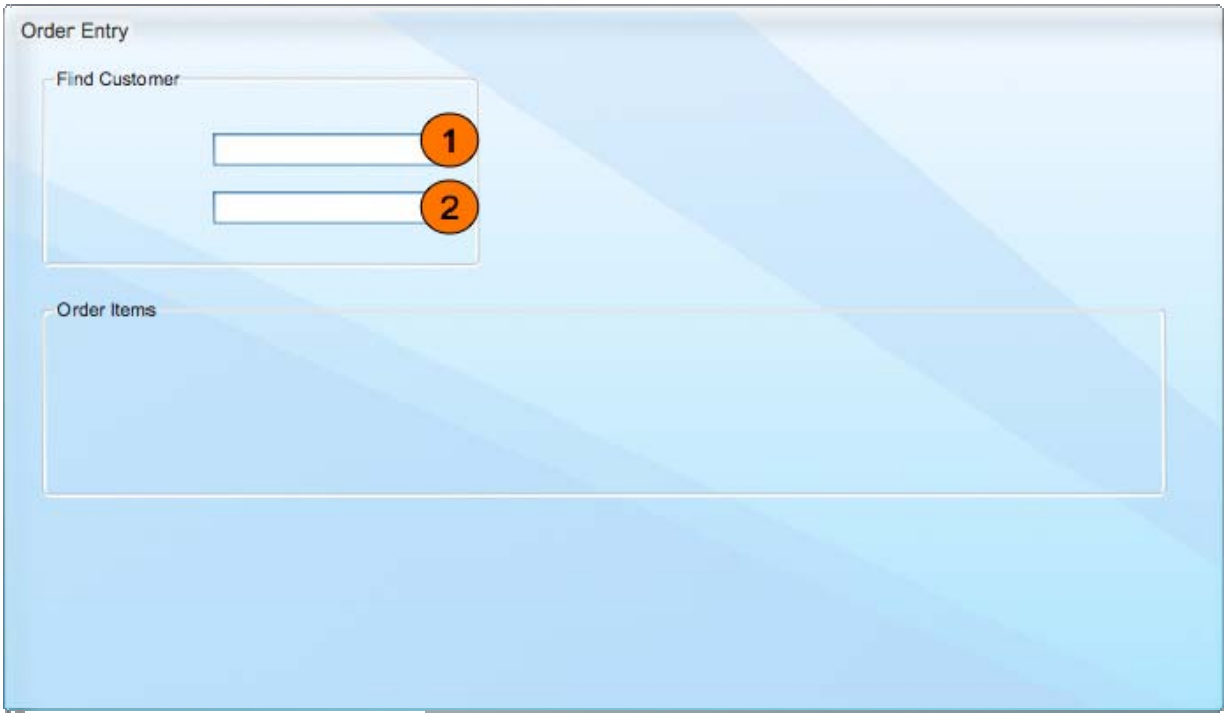

Figure 6.7: In-progress user interface design derived from a requirements sketch.

UI design is another are where requirements authors may not think of all the things that a designer needs to know, so the designer can help improve the application quality by going back, asking questions, and seeking expansion of the requirements. Don't assume that just because something isn't in the requirements that it isn't needed; the requirements authors may simply have no realized they were overlooking something. These often-overlooked considerations may include:

- Globalization—Will the user interface need to support different languages? That not only affects the UI layout, since some languages require more physical space on the screen, but will also affect many other application design decisions.
- Accessibility—Will the application need to be used by users who have physical challenges, such as poor eyesight or difficulty using traditional pointing devices like a mouse?
- Usability—What's the exact workflow that users will need to follow in order to obtain maximum productivity? A designer will rarely know this, and shouldn't guess, but it's a key to making an application that's perceived as being of high quality.

Realtime publishers
MICRO FOCUS Leading the Evolution™

Designing Testing

Finally, we come to one of the most important aspects of a high-quality application design: How will the application be tested?

Test Case Development

How will the application eventually be used? The various scenarios you can think of—order entry by a phone agent, data export by a manager, and so forth—are referred to as *use cases*. Each major use case needs to be tested, typically in discrete tests known as *test cases*. **Test cases must derive directly from the requirements,** and must be clearly defined in the application design.

Figure 6.8 shows an example test case for performance testing. This is a simple chart, which includes a reference to a more detailed description of how each test is to be performed. This is obviously a simplified example, suitable for summary reporting and keeping track of various tests.

Reference Number	Test Case	Test Type	Specific Test Conditions	Expected Results	PASS/FAIL	Date Tested	Initials	Comments
1	Test batch load daily weblog file load into ABC	ETL Load	Weblog and Search should be of typical size of those available from Monday - Friday	Processing should take less than 6 hours per batch (1/2 day of files)	Pass	12/30/2004	AB	
2	Test batch load of Data Mart	ETL Load	Process a single batch containing weblog Fact data representative of in size of those available from Monday - Friday	Processing should take less than 6 hours per batch (1/2 day of files)	Pass	12/30/2004	AB	
3	Test incremental load of SYSTEM reference data	ETL Load	Process the complete set of daily reference data files from SYSTEM	Processing should take less than 2 hours per daily set of SYSTEM reference data	Pass	12/30/2004	AB	
4	ABC ETL Load Process	ETL Load	Process a single days worth of reference and weblogs data from ABC	Processing should take less than 24 hours elapsed time and no individual segment (running in parallel) should take more than 6 hours	Pass	12/30/2004	AB	
5	Test report generation - Customer Reports	Reports	Initiate Customer Report generation through daemon process	At least 12,000 Customer Reports should be generated on a nightly (3 pm - 3 am) basis	Fail	1/4/2004	AB	Investigating issue. Customer reports terminated abnormally.
6	Test report generation - Heading Reports	Reports	Initiate Heading Report generation through daemon process	At least 2,000 Heading Reports should be generated on a nightly (3 pm - 3 am) basis	Pass	1/4/2004	AB	

Figure 6.8: Example test case summary.

Realtime publishers
MICRO FOCUS Leading the Evolution™

Going into more detail can be accomplished in several ways. One way is to identify various scenarios. In an example from it.toolbox.com, one *scenario type list* identifies various business scenarios for bank account scenarios. Figure 6.9 shows a portion of the list.

BUSINESS SCENARIO TYPE LIST	
Business Scenario Type Identifier	**Business Scenario Type**
T1	IRA Savings Account - depositor is 70 1/2 years old or older
T2	IRA Savings Account - depositor is less than 70 1/2 years old
T3	IRA Timed Deposit - depositor is 70 1/2 years old or older
T4	IRA Timed Deposit - depositor is 59 1/2 years old - term deposit within 91 days through 10 years
T5	IRA Timed Deposit - depositor is less than 59 1/2 years old - term deposit within 91 days through one year
T6	IRA Timed Deposit - depositor is less than 59 1/2 years old - term deposit > one year < or = two years
T7	IRA Timed Deposit - depositor is less than 59 1/2 years old - term deposit > two years < or = 10 years

Figure 6.9: Business scenario types.

Notice the type identifiers T1 through T7. The designer can now specify test cases that relate to these various real-world scenarios.

Test designs can become quite complex—more complicated, in some cases, than the rest of the application design simply because test cases need to not only cover the *proper* operation of the application, but also *improper* use. Commercial test management applications can help in a few ways:

- Managing test cases and their relationship to the original requirements
- Managing test data associated with each test case
- Automating some types of tests
- Tracking tests and their results

Designing Test Data

Poor test data is, in my experience, a leading contributor to poor-quality applications. Developers instinctively seem to avoid "bad" data when performing unit tests, and without good test data even testers won't be putting the application through its paces.

There are three ways to obtain test data:

- Make it up. This isn't the preferred approach, as you tend to get fairly simplistic data that doesn't reflect real-world operating conditions. As soon as real-world data hits the application, you're likely to find defects.
- Generate it. Sites like GenerateData.com, as well as many commercial testing tools, can generate various kinds of data. Good generators are an ideal source for test data, and they have the advantage of producing data that is fictional – meaning real customer data (for example), with all of its privacy concerns, doesn't become an issue.
- Borrow it. Use real-world production data taken from another system. This is the best and preferred source of good test data, since it reflects exactly what the final application will deal with. However, real data can carry security and privacy concerns. A medical billing application, for example, may not be able to use borrowed data due to privacy laws.

Tip

One trick is to borrow production data but then scramble sensitive data. The name "Connie Smith" might be scrambled to "Ninoce Htmis," for example, or identifying numbers rearranged. This technique helps provide test data that is real-world in nature, but which may be considered less sensitive.

Commercial applications exist to help manage test data sets. After designing and creating the appropriate sets of test data, the application can ensure that the data is available to anyone who needs it. The application may assist in populating databases and other data stores with test data sets on demand, and the application may help manage the security of sensitive test data – such as that borrowed from a production environment. An application may even provide auditing capabilities, so that all access to test data can be tracked; this is often useful when dealing with sensitive real-world test data that is subject to privacy or security laws.

The bottom line is that the application design needs to clearly indicate what test data will be used, when it will be used, how it will be managed, and so forth.

Test Environment Preparation

The designer also needs to specify what the test environment will look like—again, referring back to the requirements for information on what the application's production environment will look like. Specifying detailed test environments is easier these days, with the broad availability of virtualization software. Virtual machines can run nearly any operating system that the application needs to run on, and virtual machines can have arbitrary restrictions places on their memory usage, processor capability, network bandwidth, and other resources—allowing a bank of virtual machines to behave in much the same way that real, in-production computers will behave. Virtual machines also have advantages like the ability to "roll back" to a given state, allowing for easier reset-and-re-run test cycles.

The test environment may also need external systems such as load generators, test control systems, and so forth; the application's test design should specify the availability and use of these resources.

Test Execution

Many organizations feel that automated testing is the key to better software quality. Certainly it is *a* key—but *only* if combined with well-designed test plans, realistic test data, and properly-configured test environments. Automation can improve consistency and speed of testing, but it *cannot improve a bad or missing test design.*

> **Note**
>
> Let me emphasize this: Automated testing tools are wonderful at removing tedium, ensuring that tests are completed consistently every time, and often at improving the speed with which tests can be run. That is all. Automation in and of itself *contributes very little* to software quality – automation simply makes it a bit easier to achieve whatever quality you have set yourself up to have.

Even in an environment using automated testing, there's still value in manual testing. An automated test can't try different things, can't act on hunches, and so forth; it's just dumb repetition. If the automated tests don't cover a particular scenario, then it won't be tested. That's why manual tests can be so valuable: If nothing else, they help identify additional scenarios that can be set up for automated testing!

The actual execution of tests should be part of the test design. In the case of automated tests, detailed descriptions of what is being tested, what inputs will be provided, and what outputs are expected, should be documented in the design. Manual tests should have at least the same level of detail, although some application designers I've worked with tend to think that human testers can be set off on their own to "beat up on" the application. My experience is that "beating up" on an application does a poor job of revealing significant defects and doesn't contribute much to application quality; the best use of manual testing is to actually use the application just as any end-user would. Often, manual testing is the only way to test complex software and processes that involve actions and evaluations that automated testing can't accommodate.

Test Results Analysis

Analyzing test results is a skill in and of itself. Some simpler tests are easy to analyze, such as pass/fail tests that check for proper operation by feeding specific input into an application and then checking for specific output. Such *deterministic* tests are often the most easily automated, and they can be very useful in testing the various paths through an application.

Other tests can be more complex. For example, Figure 6.10 shows an example result from a performance test of a Web application.

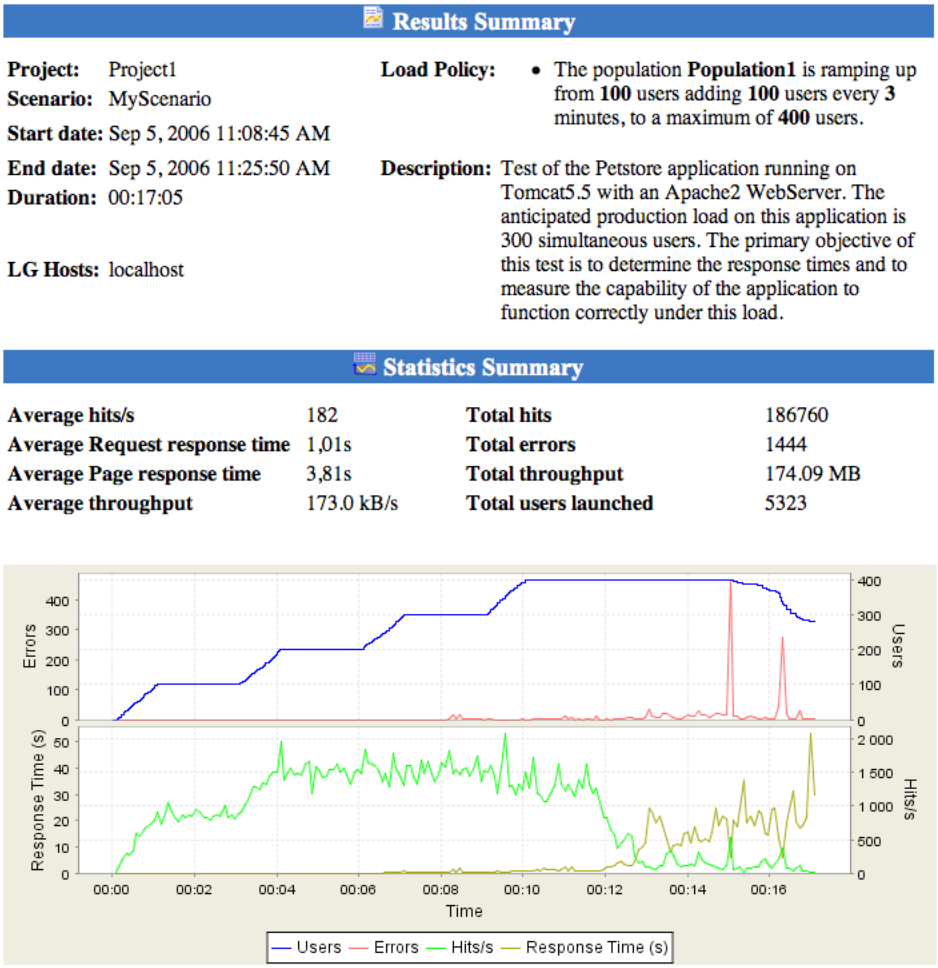

Figure 6.10: Example test results from a performance test.

This isn't a direct pass/fail result, although if it can be related back to a requirement that specifies a maximum response time, then these results *can* be interpreted as pass/fail. However, the results themselves are useful to developers and development managers, because in the event that performance is not acceptable, this test helps provide the detail needed to determine where the performance bottleneck lies. This test, for example, revealed an increase in application errors toward the end of the test, resulting in a marked climb in response times. That's something that bears further investigation in the code.

Realtime publishers
MICRO FOCUS Leading the Evolution™

Management Reporting

Knowing the results of tests is critical, because it helps manage the overall application lifecycle. An application that is only successfully completing 10% of its tests isn't close to release, and the business can make appropriate decisions about the project—perhaps extending timelines, if needed, adding more resources to the project, and so forth.

Ideally, detailed reports will be available for development managers. These might include specifics on which code units passed and failed which individual tests, allowing the development manager to continually balance resources across the project as needed. Other, higher-level reports might show the application's state of compliance with requirements—something that should be easy if the test cases relate back to those requirements (and something that test management applications can help keep track of). While lower-level reports typically need a good amount of detail, higher-level reports may be expressed as a list of requirements and a pass/fail indicator.

With the right management reports, development managers and higher-level managers can keep track of the project. They can see the application's current level of quality at a glance, and they can more easily measure the balance between obtaining more quality and controlling project expenses and timelines. Because these types of management reports must roll up a great deal of test data, many development teams rely on test management applications that can store the results of discrete tests, map those back to requirements, and automatically generate management reports, "dashboard" displays, and other higher-level reporting.

Summary

A solid, high-quality application design is the first step in translating business requirements into a functional, high-quality application. A great design should live little, if anything, to the imagination: Developers should be able to use a good design as a clear set of "marching orders" that help them produce the exact application that the business needs. A good design will clearly relate each design decision back to the original application requirements. That's a carefully-planned flow of information, and by following a few simple rules designers can ensure that they're producing a high-quality design:

- Nothing goes into the design unless it derives from the requirements.
- Everything a developer will need to know or guess must go into the design.
- If something needs to go into the design but isn't addressed in the requirements, go back and get it into the requirements first.
- If something isn't in the requirements, it's because the requirements authors overlooked it—not because they didn't feel a need for it. If, for example, the business doesn't need performance monitoring in the application, modify the requirements to simply state that.

In the next chapter, we'll look at things developers can do, while implementing the design, to bring more quality to the application project.

Download Additional eBooks from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this eBook to be informative, we encourage you to download more of our industry-leading technology eBooks and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.

Realtime publishers

MICRO FOCUS Leading the Evolution™