

A COLLEAGUE DESCRIBED HIS FRUSTRATION IN TRYING TO HELP A pair of French tourists driving through Lisbon. The tourists stopped their car and asked Carlos for directions. Carlos knew the place they wanted to go—it was only a few streets away—but he had suddenly forgotten how to say “left” and “right” in French. He gesticulated and hoped they understood. In telling me the story, he lamented, “Only after they drove away did I think, ‘Why didn’t I draw a picture?!?’”

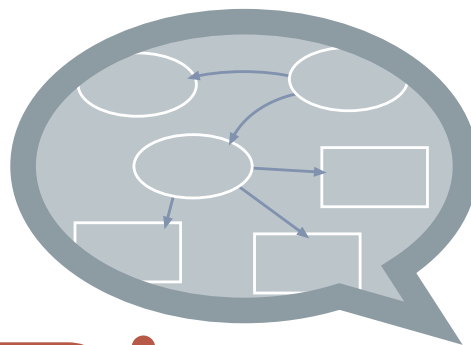
Maps are a universal way of describing an area. You use them to plan your route and find your destination. Just as it’s a good idea to have a map when traveling, it’s a good idea to have a picture of the software you’re testing.

Software Maps

Imagine that it’s time to design tests for the latest version of the software. You have been given a fifty-three-page specification that appears to be mostly boilerplate. It even still has sections that read “[fill this in with your data structure].”

It took the efforts of your manager and three VPs to make the developer deliver it, so you know it’s as much information as you’re going to get, at least in writing. You’re supposed to decide how you’ll test the software from the specification so that when the software arrives next week, already four weeks late, you’ll be able to hit the ground running.

The trouble is that you can’t make heads or tails of the spec. In between hand-waving generalities (“We’ll employ an n-tier architecture”) are minute details (“We’ll use the `obj.foo1()` method, not `obj.foo()`”). Neither type of information



A Picture's Worth a Thousand Words

How to create and use diagrams and maps to improve your testing by *Elisabeth Hendrickson*

QUICK LOOK

- Creating diagrams to visualize the software under test
- Using diagrams for communication and planning
- Uncovering test cases by visualizing the architecture

is helping you. The specification isn't in your language.

Sound familiar?

Just as Carlos should have sketched a quick map to give the tourists directions, we should sketch a map of the software so we can understand what we're supposed to test. Pictures pack a great deal of information into a small space. They help us see connections that mere words cannot. As with driving directions, when trying to understand software, showing is better than telling.

I protest! Why should I spend time on reverse engineering?

Sometimes when I suggest to testers that they create maps of the software, they protest that I am asking them to do Development's job. I see their point. In order to create maps, you end up reverse engineering the software.

However, as much as we might wish that someone would give us all the information we'd like, that rarely happens. If you are in an organization where you can count on comprehensive, up-to-date project artifacts that tell you everything you might want to know in order to test the software, count your lucky stars. You're one of the few. I believe more organizations are doing more documentation now than a few years ago, but most organizations still have out-of-date or incomplete specifications.

Besides, the time we take to understand the software is never wasted. It improves our tests and may also improve the organization's understanding of the software. That said, allow me to describe how to create these maps, how to use them, and the benefits you're likely to receive from having spent time on them.

Styles of maps

There are numerous types of maps and styles of notations that you might try. UML, Unified Modeling Language, is gaining widespread acceptance as a de facto standard. However, long before UML, the industry had flow charts, state charts, data flow diagrams, entity-relationship diagrams, and a host of other ways to represent various views of the system under development.

We have many different kinds of diagrams, because any system can be seen from a variety of perspectives. The user sees something far different from what the database designer sees. It's valuable to draw diagrams of the system from various perspectives to gain a better under-

standing of the whole.

The diagrams in this article are represented in a UML-ish notation, but do not adhere rigidly to the UML specification. My goal here is to get you interested in diagramming your system and using those diagrams to improve your testing, rather than in teaching a particular notation. In general, I think it's a good idea to be aware of different notations, but I caution you not to be obsessed with making your diagram "right." I think it's much more important to ensure that your pictures are useful than to worry about getting the notation right.

Looking at the Parts

The first kind of diagram I usually draw shows the pieces and parts of the system and how they connect. I draw a picture that includes interfaces, processes, files, databases, and connections. In UML, this is called a Deployment Diagram. I've heard others call them architectural diagrams, block diagrams, and parts maps. Whatever you call them, the point of these pictures is to lay out all the pieces and parts of the system. For the purposes of this article, I'm referring to these diagrams as "parts maps."

I start with what I know. For example, if the specification talks about a client piece,

you can be pretty sure there's a server piece somewhere. So my picture will start out looking like Figure 1.

Pretty simple, but already tests are popping to mind. What if the Server goes down? What happens if the network fails? What if the Client is dialing in over a 28.8 modem?

Then I go through the specification and other resources looking for anything I can add to my picture. Maybe the specification talks about a database. Maybe there's a configuration file that determines how the server behaves. I'll add them in, giving me a diagram like the one in Figure 2.

After scouring all the documentation and experimenting with the software under test, my map is usually much more detailed. For example, a typical n-tier Web application might look something like Figure 3.

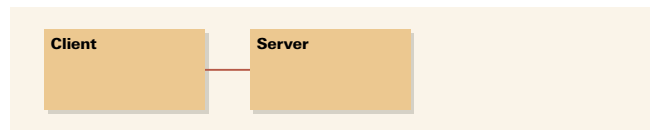


Figure 1: Simple parts map

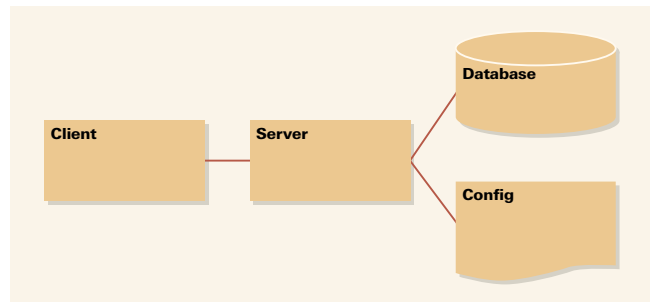


Figure 2: Expanded parts map

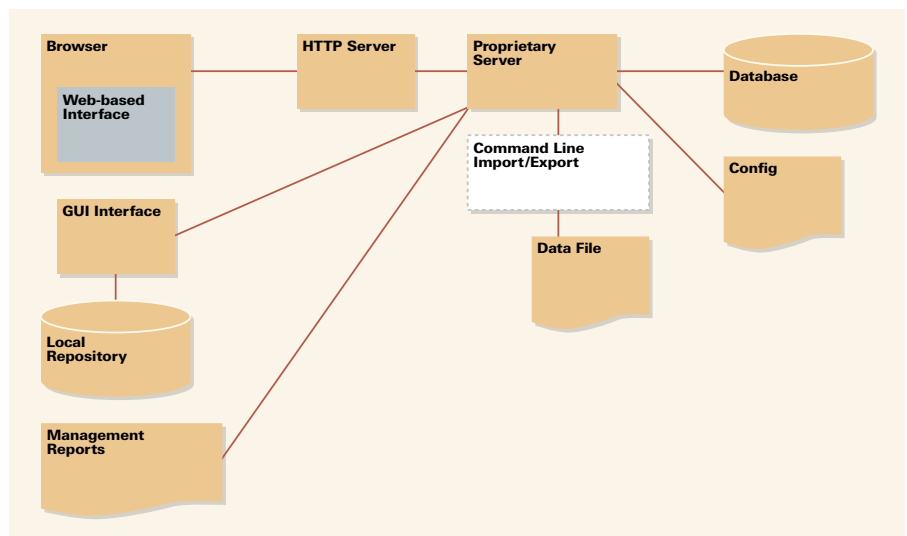


Figure 3: Parts map for a typical n-tier Web application

CLIENT OS	SERVER OS	DATABASE
Windows 98	Sun Solaris	Sybase
Windows ME	Microsoft Windows XP Professional	MS SQL Server
XP	Microsoft Windows 2000 Advanced Server	Oracle

Figure 4: Sample configuration table

Using your parts map

Your parts map is helpful in numerous testing activities, including interviewing developers, planning the test lab, and planning the tests.

Interviewing Developers Sketching a parts map gives you a basis for discussing the software that will be delivered to you. Now interview the developer:

- Is my diagram correct, at least as far as it goes?
- What's missing?
- What parts of the software are you responsible for?
- Who is responsible for the other pieces?
- How does this part of the software talk to that part?

The answers developers provide will lead to improvements and additions in the diagram, as well as a better overall understanding of the software you're testing.

Planning the Test Lab The parts map also allows you to think about configuration testing. Will all the processes be on one machine, or will you have to test in different server farm configurations? Will you have to test different combinations of operating systems and supporting software such as HTTP servers? You can create configuration tables with each element of the deployment diagram, as shown in Figure 4.

Then you can discuss whether you need to test every combination (in Figure 4, twenty-seven configurations), or whether it is sufficient to test every Client OS, Server OS, and Database without worrying about combinations. (The answer depends on how your software works. The point isn't to come up with an answer independently, but to be able to frame the questions.)

You can also discuss how you'll set up the test environment:

- How many servers will I need to set this up in the lab?
- Do you have a server running now in development that I could see?
- What operating systems does the client support?

In other words, the parts map allows you to lay out all the parts of the system on the table so you can analyze the relationships between them and determine the best testing strategy.

Planning Tests In addition to thinking about configurations for testing, your parts map helps you think about types of tests.

Anywhere you see a process, such as a server, consider these tests:

- Stop the process, preferably while it's in the middle of reading or writing data.
- Shut down the machine running the process.
- Run another program on the machine to hog all the CPU and memory.

Anywhere you see a file, consider making that file

- in a path with a very long name (>255 chars)
- on a hard drive with little or no disk space
- on a write-protected location such as a locked diskette
- with no read or write permissions
- present if it shouldn't be
- missing if it should be there

- locked by another process
- in the wrong format
- huge
- zero-length
- corrupted

For all network connections, consider these tests:

- Disconnect the network.
- Insert a firewall with maximum paranoid settings.
- Slow the connection down (for example, by using a 28.8 modem to connect).

Your diagram will also show you various interfaces into the system. When working with software with a graphical user interface (GUI), it's easy to fixate on that one entry point into the system. There may be other interfaces, such as command line utilities to import and export data. If you discover multiple paths for data to enter the system, find out what happens when bad data sneaks in through a back door. Find ways to do the following:

- Enter out-of-bounds or incorrect data.
- View invalid data through all the interfaces.

Consider the case where an import utility allows bad data into the system that the GUI rejects. When you run reports, how is that invalid data displayed? What happens if you edit the invalid data in the GUI?

A note on expected results: Many of the tests that I suggest here are negative tests. I expect them to fail. For example, if you delete a required file, I don't expect the software to proceed merrily as though nothing is amiss. I'm looking for the software to handle error conditions gracefully. If the user sees an error message, that error message should contain information that tells the user what's happening and why, and preferably what they can do about it.

The Software in Action

Having drawn a parts map, we have a picture of all the parts of the software. We don't have a picture of how those parts behave. We don't know what a

transaction with the server looks like. We don't know if there are modes we need to worry about in the user interface. If we want to understand both what's in it and how it works, we really need to draw pictures of the software from both perspectives: parts and behavior.

Flow charts

Flow charts show the logical steps that a program goes through, including decision points and activities. You can represent flow charts in a variety of ways. In UML, these are activity diagrams. Figure 5 shows a flow chart for a typical installer.

More tests start popping to mind. What if I cancel while the installer is copying files? Can the installer back up to the previous step? What if the "copy files" activity fails? Whenever you have a flow chart, look for tests that cause the software to

- skip a step
- return to a previous step
- take steps out of order
- exit in the middle of each step

Having a picture of the logical progression of the software makes it easier to think about what paths we might want to take through the software, as well as what illegal paths we might want to force.

State diagrams

Another way to diagram the behavior of the software is with a state diagram. The bubbles represent states; the arrows are the events that cause transitions. Figure 6 depicts a state diagram for a process that periodically connects to a remote server to download system updates.

This diagram shows four states. At any given time, the system is idle, checking to see if there's an Internet connection available, getting updates from the server, or incorporating the updates it downloaded. Transitions prompt the system to move from state to state. For example, the system moves from Idle to Check Network either when enough time has elapsed, or when the user chooses to check for updates. The system moves from Check Network to Get Updates if it determines

that an Internet connection is available, or back to Idle if no Internet connection is present.

You may notice some strong similarities between the flow chart described earlier and the state diagram here. That's to be expected. Both the flow chart and state diagram describe the behavior of the system. Typically, state diagrams refer to internal states that are invisible to the user, while flow charts describe behavior the user instigates. However, these are not hard-and-fast distinctions. You could describe the state diagram above using flow chart symbols.

Identifying States

When creating parts maps, you can see files or processes running; creating state diagrams may require more guesswork on your part. States are more difficult to see. However, there are usually clues.

When reviewing the specification, look for keywords such as "state," "mode," and "transition." If the documentation contains a phrase like, "...when in Edit Mode, you can change..." you now know that there's an Edit Mode. That's probably a state. The next question is, how do you get into that state? What state was it in before? What states are available after? By asking these questions as you explore the system, you build up a state model fairly quickly.

Also look for functions that are available only after performing certain actions. For example, if you can only change a diagram in a document after double-clicking it, you've identified two different states: Edit Text and Edit Graphics.

You can identify distinct states in the software under test by looking at the be-

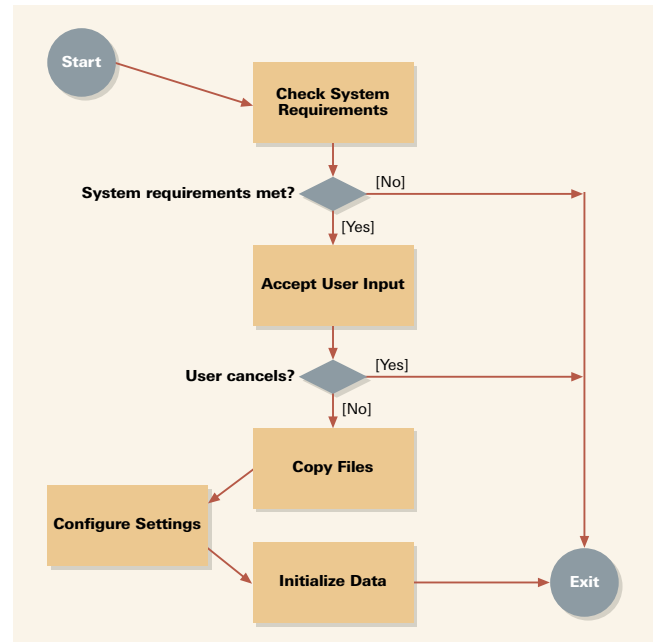


Figure 5: Flow chart for a typical installer

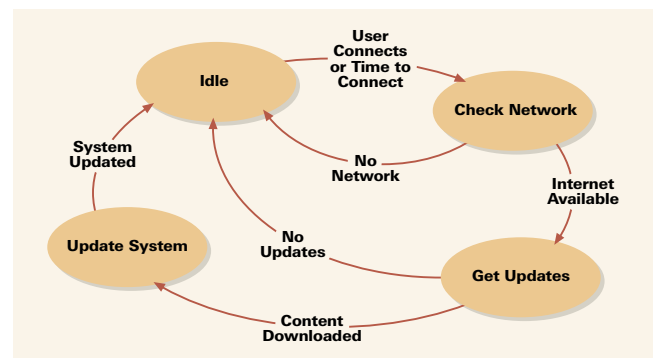


Figure 6: Sample state diagram

havior of the software. My colleague, Alan Jorgensen, explains that you should look for situations in which

- you can do things you couldn't do before
- you can't do things you could do before
- your actions have different results now than they did before

For example, when the software is starting up, it performs initialization tasks that it doesn't perform at any other time. You would represent that period of time as an initialization state.

Using a State Diagram Once I have drawn a state diagram, I have even more things to test. What if an event

occurs at an unexpected time? What if the resources (files, etc.) that the state needs aren't available? When analyzing a state model to design tests, I look for opportunities to do certain things:

- Apply every controllable event to every state. For example, I might try to force an update when the system is already in the middle of updating.
- Force exits from every state and check that the program behaves sensibly. What happens if I shut down the software as it's starting up, or close windows when it's least expected?
- Make two machines interact in various combinations of states. If two different parts of the system rely on shared resources, there may be conflicts.

Software Mapping Tips

Start simple You also don't have to diagram the whole system the first time you sit down to make a map. Choose one small area to begin with. You can always expand your map later. If you attempt to map too much at once, you're likely to become overwhelmed.

Collaborate A picture created by several knowledgeable people is more likely to be correct than a picture created by one. Also, in attempting to reconcile multiple perspectives, you may find problems even before you begin using your map to test the actual software.

Discover hidden details with operating system tools Simple PC utilities such as InCtrl or Process Explorer can help you find files your software uses. You can use utilities provided by your operating system to see a listing of all the processes running (for example, Task Manager on Windows 2000 or ps on Unix). There are numerous utilities available, many part of your operating system or free for the downloading, that will help you see what's happening on your computer. Take advantage of these to help inform your mapping efforts.

The Power of Visualization

How much benefit could you possibly get from taking the time to describe the software you're testing in pictures? Quite a bit, especially if it's the first time

anyone has attempted to visualize the entire system.

Pictures facilitate understanding One organization struggling with intermittent bugs used a picture of interacting state models to understand why the bugs might be occurring. Before drawing a map, everyone on the team was baffled. After they saw a picture showing two states interacting, it became obvious where the bugs came from.

In another case, a skeptical participant in a workshop questioned the value of these diagrams. She presented a problem to be diagrammed, thinking the diagram would be useless in finding a bug she had in mind. In a few seconds she told the facilitator, "Never mind. You can stop drawing. There's the bug right there." The bug that had taken weeks of effort to isolate and fix was patently obvious to her on the diagram. She'd thought it was a subtle bug no one could have been expected to find. Seeing the bug so quickly on the diagram convinced her that if the team had diagrammed the logic flow, they would have found the bug before they shipped the software.

Pictures facilitate communication In another organization, the testers were having a difficult time communicating with the developers. The project was well underway, but the testers couldn't figure out how to test the software. One session with a large pad of flipchart paper and some pens enabled the testers and developers to share enough information that the testers could start their test design process. In fact, the testers started their test design process then and there—and tested the design before they even had code, much to the delight of the developers.

Managers understand pictures At one company, the testers used a deployment diagram to enable upper management to see the impossibility of testing all possible deployment platform combinations. Management quickly revised their "must-have" list of supported platforms. At another company, the test managers used a parts map to help the executives see how changes in the architecture would affect testing. The executives stopped asking, "Why will it take so long to test?" and instead started asking, "What are the most strategic changes we can make?"

Go Forth and Draw!

All you need to get started is half an hour, a sheet of paper, and a writing implement. You don't need modeling software or specialized skills to make this work. Although design tools such as Rational Rose or Visio are likely to help your diagramming efforts, they are not necessary—especially for a first attempt. So what are you waiting for? Unless you already have a map of the software you're testing, it's in your best interest to create one. And even if you do have a map of the software, it's a good idea to create your own to ensure you really understand what it is that you're testing.

STQE

Elisabeth Hendrickson (esh@qualitytree.com) is an independent consultant who specializes in software quality, testing, and management. She has more than twelve years of experience working with leading software companies and is the founder of Quality Tree Software, Inc. An award-winning author, Elisabeth has published more than twenty articles and is a frequently invited speaker at major software quality and software management conferences.

ACKNOWLEDGMENTS: I am grateful to the participants of the LAWST 13 and Sim-LAWST 1 meetings in which the participants shared their experiences drawing various types of diagrams. The participants of LAWST 13 were: III, Chris Agruss, Sue Bartlett, Hans Buwalda, Anne Dawson, Marge Farrell, David Gelperin, Elisabeth Hendrickson, Doug Hoffman, Mark Johnson, Karen Johnson, Cem Kaner, Brian Lawrence, Hung Nguyen, Bret Pettichord, Harry Robinson, and Melora Svoboda. The participants of Sim-LAWST 1 were: III, Sue Bartlett, Paul Downes, Marge Farrell, Rochelle Grober, Sam Guckenheimer, Elisabeth Hendrickson, Doug Hoffman, Brian Lawrence, Serge Lucio, Frank McGrath, Bret Pettichord, Melora Svoboda, Andy Tinkham, Jo Webb, and Melissa Wibom.

STQE magazine is produced by STQE Publishing, a division of Software Quality Engineering.