Name: Architecture Achilles Heel Analysis

Elisabeth Hendrickson, Grant Larsen

Objective: Identify areas for bug hunting – relative to the architecture

Problem: How do you use the architecture to identify areas for bug hunting?

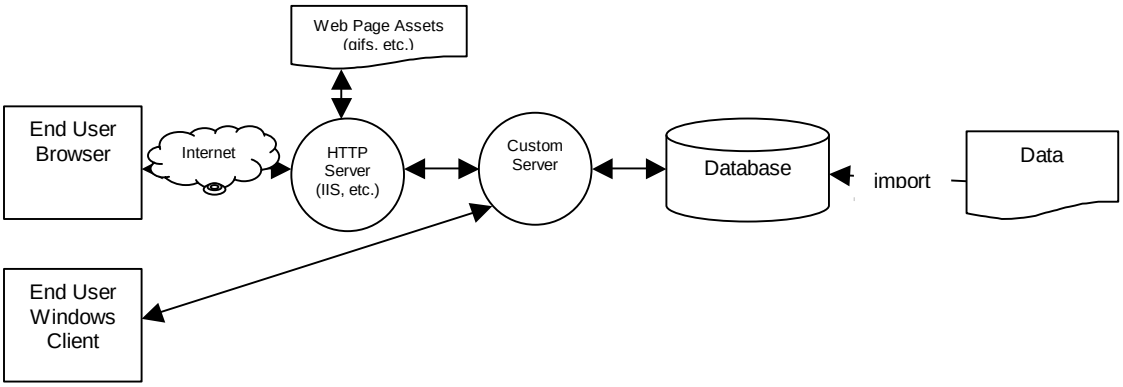
Context: You are a tester and want to make best use of the testing time. Your goal is to find high severity bugs. You have a physical, functional, or dynamic map of the system in some form (or can create one with *Architecture Reverse Engineering*). You know what characteristics of the system are most important (*i.e.* is reliability more important than performance?). You may or may not be doing formal test planning.

Forces:

- You may not be able to answer all your own questions about the architecture.
- Getting information about the architecture or expected results may require additional effort.
- The goals of the architecture may not have been articulated or may be unknown.

Solution:

An architectural diagram is usually a collection of elements of the system, including executables, data, etc., and connections between those elements. The architecture may specify the connections between the elements in greater or lesser detail. Here's an example deployment architecture diagram. This technique works for deployment, physical, functional, or dynamic architecture maps.



In this example, we have two user interfaces, a Web-based interface and a Windows client. The Web based interface connects over the Internet to an HTTP server that in turn connects to the custom server process that reads and writes data to the database. The Web version relies on web site assets (gifs, etc.). The Windows client connects directly to this custom server. Data from the development environment is imported into the database with a command line import utility.

In analyzing the architecture, walk through each item and connection in the diagram. Consider tests and questions for each of the items in the diagram.

Connections

- Disconnect the network
- Change the configuration of the connection (for example, insert a firewall with maximum paranoid settings or put the two connected elements in different NT domains)
- Slow the connection down (28.8 modem over Internet or VPN)
- Speed the connection up.
- Does it matter if the connection breaks for a short duration or a long duration? (In other words, are there timeout weaknesses?)

User Interface

- Input extreme amounts of data to overrun the buffer
- Insert illegal characters
- Use data with international and special characters
- Use reserved words
- Enter zero length strings
- Enter 0, negatives, and huge values wherever numbers are accepted (watch for boundaries at 215=32768, 216=65536, and 231=2147483648).
- Make the system output huge amounts of data, special characters, reserved words.
- Make the result of a calculation 0, negative, or huge.

Server Processes

- Stop the process while it's in the middle of reading or writing data (emulates server down).
- Consider what happens to the clients waiting on the data?
- What happens when the server process comes back up? Are transactions rolled back?

Databases

- Make the transaction log absurdly small & fill it up.
- Bring down the database server.
- Insert tons of data so every table has huge numbers of rows.
- In each case, watch how it affects functionality and performance on the server and clients.

Multiple Users and Interfaces

- Two users doing the same thing at the same time—both trying to save changes to the same record at the same time, for example.
- Two users doing the same thing at the same time in different interfaces (Web and Windows)

Data Files

- Write a file to a full disk full
- Write a file to an area with no write permissions
- Use UNC path names
- Use path names with spaces
- Use long path names (not 8.3)
- File exists if it shouldn't or doesn't exist if it should
- Read/write to a file locked by another process.
- Read a file with no read permissions
- Read a file in the wrong format
- Read in a huge file
- Read a 0 length file
- Read a file with line feed v. carriage return chars at end of line (often a UNIX v. NT problem).

Big System Questions

Walk through the architectural diagram asking questions. A list of sample questions follows. Depending on your architecture, you may have additional or different questions.

- For each executable in the architecture, what happens if that executable is not running?
- For each data storage mechanism (whether a relational database, file in the file system, or other persistent mechanism), what if the data is corrupted? Deleted? Lacks integrity or conflicts with other data in the system?
- For each connection, what if the connection breaks? Does it matter if the connection breaks for a short duration or a long duration? (In other words, are there timeout weaknesses?)
- Are there multiple connections going into or out of an element? Does this open up the possibility of resource conflicts or deadlocks?
- Are there shared resources where there may be performance bottlenecks?

- Do the elements of the architecture have states? Is there any possibility that those states might conflict—for example, could one part of the system try to start up a process while another part is trying to shut it down?

As you answer the questions mentally or verbally, write down possible risks, weaknesses, or tests that occur to you. These become areas of the system on which to focus your energy. Alternatively, you could use the architectural diagram while exploring the system hands-on.

Where you are unable to answer your own questions, seek the assistance of others who can help you: programmers, architect(s), other testers, technical support personnel, etc.

Rationale:
Understanding the big picture of the system under test can help testers focus their energy appropriately. Without this understanding it's very easy to spend large amounts of time investigating relatively minor bugs or risks.

Resulting Context:
You now know more about the architecture and implications of various design decisions and can use that information to guide your test effort. If the architectural goals are still unclear, perhaps an *Explicit Architectural Goal Articulation* is in order.

Additional Benefits:
- Increases communication between different groups within the project.
- Finds potential architectural flaws earlier.
- Aligns architecture with its goals.
- Provides additional fodder for test planning for current and future releases.

Liabilities:
- You may need to spend a large amount of time understanding the system before you can productively seek out bugs.
- Other members of the team or organization may need to invest more time answering questions to contribute to the test effort than anticipated.

Note that the last two liabilities may actually be good for the project or software in the long run.
- Some members of the team may feel threatened during this process.
- Information uncovered during this process may throw the project into chaos, at least temporarily.

Related Patterns:
- Architecture Reverse Engineering
- Explicit Architectural Goal Articulation