Patterns for Software Architecture

Neil Harrison
Avaya Labs
nbharrison@avaya.com
303-538-1541

One of the purposes of architecture is to act as a bridge between the problem space and the solution space. This is both very important and rather difficult. There should be architecture patterns that cover this area.

I have not seen any patterns in this area. Mary Shaw's patterns (PLoPD1, PLoPD2), and the architecture patterns described in Pattern-Oriented Software Architecture come close, but miss the problem-space orientation: the customer view. For example, the Blackboard pattern rests solidly in the solution space: a user would not think of the system in that way. That is even more true of the Pipes and Filters pattern. While these patterns are very useful, they do not address the bridge between the problem and solution spaces.

I have identified five possible patterns that appear to be "bridge" patterns. These are important because they identify how the system works at a very high level to both the user and the implementer. Interestingly, the user and the implementer get different things from the patterns. This is a very powerful characteristic of these patterns. This shows how architecture can be visible to the user, and how the user's perception of the system has a strong impact on the architecture of the system.

These patterns are very preliminary. I hope that the brief descriptions below are sufficient to show the general idea of the patterns. Note also that some of these patterns may actually be two or more patterns; this will become more clear as the patterns mature.

1. Transaction Processing System

To the user, the system handles some sort of transaction, such as banking transactions, record keeping, reservations, etc.

The implementer sees a system that handles independent inputs, one at a time, that are pretty much independent of each other. There may be a database attached. Processing time may be important, but we aren't talking realtime here. We probably don't have to worry about state very much. (If the system handles multiple simultaneous transactions, then there are some locking states to handle.)

2. Translation System

The user inputs data, and gets some other data in return. These include encryption and compression systems, and compilers.

The implementer sees processing streaming input. The way it is processed

may be affected by previous processing. The system may be continuous, such as encryption, or it may be finite, such as a compiler. (This may be a cause to split this into two patterns.) Real time may or may not be important, which may be more motivation to split this into two.

3. Control System

The user wants to fly an airplane, or run the traffic lights. Some users might want to control a computer (operating system). Other systems, like fire alarm systems, are control systems.

There may be two different patterns, life-critical control systems and non-life-critical control systems. The difference is that the life-critical systems must not go down, and have stringent requirements for correct behavior. If the assembly line system crashes, it is merely inconvenient and expensive.

The implementer is concerned with realtime. It may be important that the system stay up all the time, so there may need to be a spare processor strategy. These systems are heavily concerned with state; each probably has multiple state machines.

4. Call Processing (Multiple communication connections)

This may be a special case of a control system, but the communication aspect gives it a different color. The user sees the need for communication, and the implementer sees many protocols and complex interconnection scenarios. Half-Object plus Protocol and Three-Part Call Processing are models.

5. Interactive Systems.

These are web browsers, word processors, games, etc. Imagine the user sitting at a keyboard, typing, and the system displaying something in return (just like I am doing now.)

For the implementer, there is a rather straightforward input and output model. Input is often from a keyboard and mouse. Output is often via a GUI. There may be some state. Performance must not be ignored, but realtime is not important.

One category of interaction systems may deserve its own pattern: simulation systems. This would include interactive games. In these systems, realtime is critical, and graphics are sophisticated.

One powerful aspect of these patterns is that they evoke a set of questions that help us get to important issues of the system. Each pattern has its own set of questions. Note that the questions are relevant for both the user and the implementer. These are still in the embryonic stage, but samples are as follows:

1. Transaction System:

What if the fails? Do we have to back anything out?

2. Translation System:

How much do I need to translate at once?
What to do with the output?

3. Control System:

What happens if the system goes down? How much do we care?
What if control is wrong? What are the implications?
How fast must the response be?
What if the system receives unexpected input?

4. Call Processing:

What types of connections?
What is the capacity of the system?
What kind of reliability and availability does the user expect?

One Rub:

Some systems may be more than one pattern. For example, a communication system (Call Processing) may have an administration subsystem (Interactive System), and a billing subsystem (perhaps a Transaction System). I haven't explored this much, but it seems to me that those would provide natural lines of gross partitioning of the overall system: one should try to make each part as independent of the others as possible.