From Craft to Science: Rules for Software Design -- Part II

by **Koni Buhrer**
Software Engineering Specialist
Rational Software

Developing large software systems is notoriously difficult and unpredictable. Software projects are often canceled, finish late and over budget, or yield low-quality results -- setting software engineering apart from established engineering disciplines. While puzzling at first glance, the shortcomings of software "engineering" are easily explained by the fact that software development is a craft and not an engineering discipline. To become an engineering discipline, software development must undergo a paradigm shift away from trial and error toward a set of first principles.

In this second installment of a two-part series for The Rational Edge, I will outline a set of design rules -- a "universal design pattern" associated with the first principle and axiomatic requirements that I proposed in the first installment. The universal design pattern features four types of design elements that, in combination, can describe an entire software system in a single view. Each type of design element is concerned with one -- and only one -- of the following four aspects of system operation:

- *data structures and primitive operations*
- *external (hardware) interfaces*
- *system algorithms*
- *data flow and sequence of actions*

Why Do We Need Design Rules?

A first principle is an established, fundamental law or widely accepted truth that governs a specific field of science or engineering. For the construction field, for example, the law of gravity is a first principle. In the first part of this article, I proposed that software development, too, has a first principle, namely: "Software runs on and interacts with hardware -- hardware that has only finite speed."

Although first principles are the basis of all scientific reasoning, they provide little practical help to an engineer faced with a design task. They are usually too abstract or too cumbersome to be employed directly. Yet first principles induce axiomatic requirements -- requirements that apply to each and every system that ever has been and ever will be created. For the construction field, for example, "The building shall withstand the force of gravity" is an axiomatic requirement. Obviously, any building of value must satisfy this requirement.

Software development, too, has axiomatic requirements, as we saw in the first installment of this article:

1. The software must obtain input data from one or more external (hardware) interfaces.
2. The software must deliver output data to one or more external (hardware) interfaces.
3. The software must maintain internal data to be used and updated on every execution cycle.
4. The software must transform the input data into the output data (possibly using the internal data).
5. The software must perform the data transformation as quickly as possible.

From their axiomatic requirements, all established engineering disciplines have derived a set of design rules. In construction, for example, a design rule is: "A weight-bearing wall must always be positioned on top of a weight-bearing wall." The purpose of such design rules is to help engineers create architectures that *obviously* satisfy the axiomatic requirements. If an architectural design obeys all the design rules, then the engineer knows, *a priori,* that it satisfies the axiomatic requirements and is therefore consistent with the discipline's first principles. Design rules allow an engineer to easily demonstrate that an architecture is sound *before* beginning construction work.

So, let's find the design rules of software development that are associated with the five axiomatic requirements above.

The Universal Design Pattern

Design rules come in various forms and shapes. They may be very explicit, such as a list of imperative statements of the form, "The software developer shall . . . ." Or they may be more subtle, such as a set of predefined design elements featured in a design language. If a design language features classes, for example, then an implicit design rule would be, "Use classes for software design." Most software design methods provide both a set of predefined design elements and narrative rules about how to use those design elements.

Let's start with the design elements. Note that existing design languages like the Unified Modeling Language (UML) offer little help. The UML, for example, is an *ad hoc* notation system with no underlying first principles or axiomatic requirements. Its design elements were chosen because of their great expressive power and because trial and error had proven their usefulness. These are not the design elements we are looking for. Instead, we need the most restrictive design elements possible -- elements that force software developers to create architectures that *obviously* satisfy our axiomatic requirements.[1] The following four types of design elements fit this description:

- **Data Entity**
  Data entities represent the software system's input data, output data, and internal data.
- **I/O Server**
  I/O servers encapsulate the external (hardware) interfaces with which the software interacts. I/O servers can also be pure input or output servers.
- **Transformation Server**
  Transformation servers perform the transformation from input data to output data, while possibly updating internal data.
- **Data Flow Manager**
  Data flow managers obtain input data from the I/O servers, invoke the transformation servers (which transform input data into output data), and deliver output data to the I/O servers. Data flow managers also own the internal data.

Axiomatic requirements 1 through 4 clearly imply the presence of data entities, I/O servers, and transformation servers -- but why do we need data flow managers? The answer is that data flow managers make the flow of execution explicit. And this, in turn, allows a software developer to show that an architecture satisfies axiomatic requirement 5: "The software perform the data transformation as quickly as possible." Without data flow managers -- say, with the other design elements sending messages to one another -- the flow of execution would quickly become untraceable and execution timing thus unpredictable.

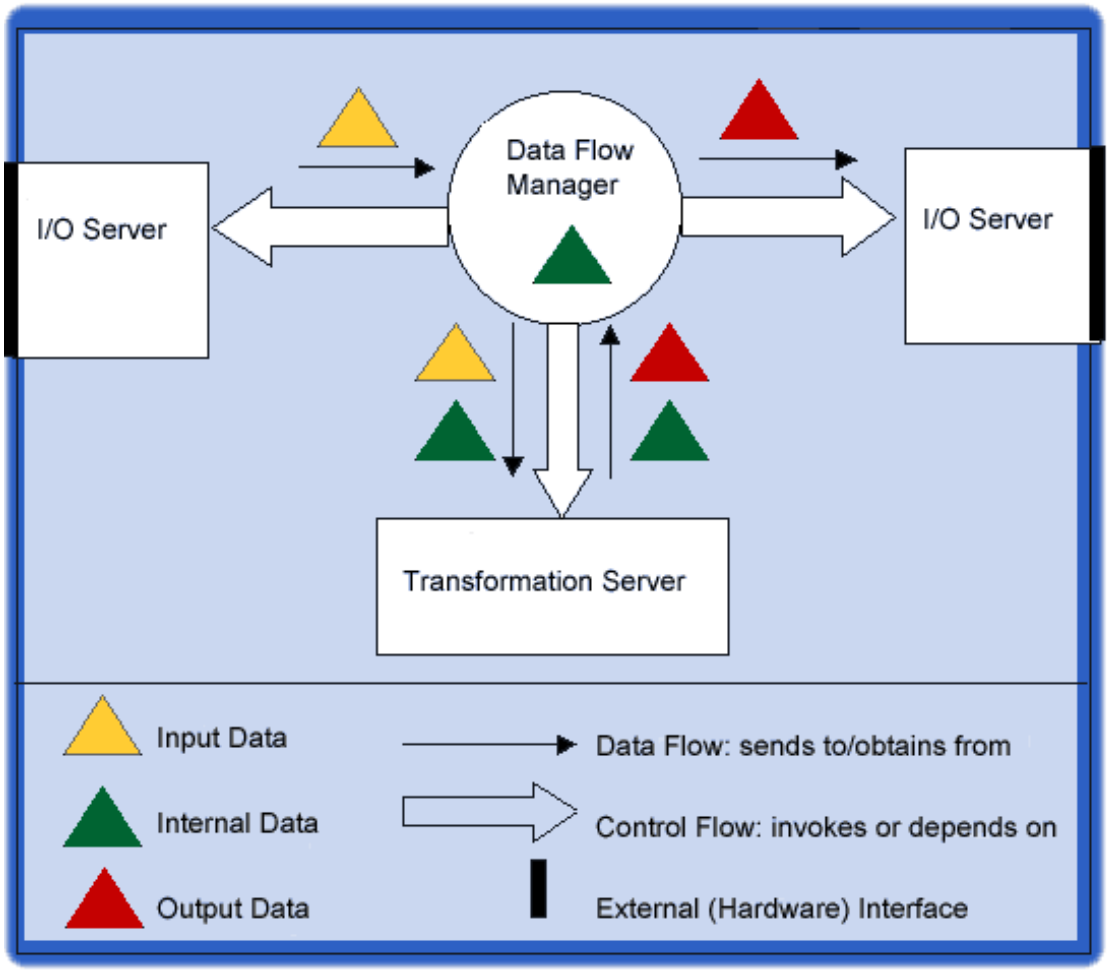

Figure 1: A Simple Architectural Design

Figure 1 is an example of a very simple architectural design using the design elements introduced above. A more complicated design might have multiple data flow managers, multiple I/O servers, and multiple transformation servers. Each data flow manager might obtain input from several I/O servers at different points and deliver output to different I/O servers. Furthermore, there might be internal databases or abstract services represented by I/O servers.

Each type of design element has a distinct nature and a distinct responsibility within a software system. The properties and responsibilities overlap very little, and the design elements of the universal design pattern complement one another nicely. Below is a list of narrative rules that describe in more detail how the design elements of the universal design pattern should be used and implemented.

Data Entity

Each data entity is a passive object of a stateless class. Data entities may be constructed from more basic data entities through aggregation and inheritance. The primitive operations of a data entity are implemented as methods of its class.

The primitive operations of a data entity never invoke operations of a transformation server or an I/O server.

I/O Server
Externally, I/O servers are passive system elements. They do not dispatch input data or act on other system elements. I/O servers have a fairly uniform external interface, based solely on the operations *Get*, *Put*, and *Wait* (and any combinations thereof). Internally, most I/O servers have an independent thread of control and maintain global state, to respond to hardware events and client (data flow manager) requests.

I/O servers have the ability to buffer both input and output data. When a data flow manager invokes an I/O server operation, the I/O server typically retrieves input data from a buffer and stores output data into a buffer. However, the algorithms by which an I/O server manages its data buffers must be simple.

An I/O server never invokes transformation server operations directly.

Transformation Server
Transformation servers are passive and stateless (and thus re-entrant) system elements. Each transformation server implements a data transformation operation, which is governed by a sequential, deterministic algorithm. The algorithm may be arbitrarily complex, but it must depend only on the arguments of the data transformation operation. Together the transformation servers implement all the algorithmic aspects of a software system.

When a data flow manager invokes a transformation operation, all input data is explicitly passed in, all output data is explicitly passed out, and any internal data is explicitly passed in and out. A transformation server may update internal data only through the arguments of its transformation operation, but it may not maintain any global state.

A transformation server never obtains input data directly from, or delivers output data directly to, an I/O server.

Data Flow Manager
The data flow managers are the active elements of a system; each one represents an independent thread of control. Data flow managers are the means by which a software developer should implement concurrency within a system (except for hardware drivers). Data flow managers act on I/O servers and transformation servers by invoking their operations. A data flow manager performs all its actions as quickly as possible. Note that for a data flow manager, "waiting for input" constitutes an action.[2]

Data flow managers ensure that data flow and control flow always go hand-in-hand. Data is passed along and eventually passed back whenever a data flow manager invokes the operation of an I/O server or a transformation server. A data flow manager always *obtains* input data and *delivers* output data; no data is ever *sent to* or *taken from* a data flow manager. As a result, data flow managers de-couple the processing elements (I/O servers and transformation servers) of a system and make data flow and control flow predictable and traceable.

Divide and Conquer

The universal design pattern has one key characteristic that sets it apart from any other design method or modeling language: it strictly divides operational concerns among its four types of design elements. The universal design pattern thus eliminates any need for different *views* of a design; with the universal design pattern, all aspects of a software system can be represented in a single view. But in that view, each type of design element is concerned with one -- and only one -- aspect of system operation:

- Data entities are concerned solely with *data structures* and primitive operations on those data structures. A data entity is an entirely passive object -- pure data[3] -- oblivious to system algorithms, external interfaces, or sequence of actions.
- I/O servers are concerned solely with encapsulating and servicing *external (hardware) interfaces*. An I/O server may be concerned with I/O timing, but not with scheduling or triggering other system actions. I/O servers do not perform any algorithmic tasks with respect to the input or output data they handle.
- Transformation servers are concerned solely with *system algorithms*. A transformation server never has to worry about where data is coming from or going to. All operations of a transformation server are sequential and deterministic. Transformation servers are not concerned with data representation, external interfaces, or sequence of actions.
- Data flow managers are concerned solely with *data flow and sequence of actions*. Data flow managers know what actions the system needs to perform and in what sequence, but they are not concerned with the details of those actions -- the algorithms and external (hardware) interfaces. While data flow managers own the internal data, they are not concerned with their representation.

Other design approaches typically yield design elements that are concerned with multiple aspects of system operation and thus have a high level of internal complexity. For example, a design based on identifying objects in the problem space often yields some design elements that are concerned with all four aspects of system operation: data structures, external interfaces, algorithms, and system actions.

Tying It All Together

You may wonder how the universal design pattern relates to established design languages like UML, the object-oriented paradigm, or modern programming languages. Am I proposing that we should turn our back on object orientation and revive data-flow modeling? Or that we should abandon UML and component-based approaches? No, not at all. We just need to use these approaches more judiciously.

Remember, each type of design element of the universal design pattern is concerned with only one specific aspect of system operation. Therefore, it is perfectly reasonable to use different design languages or modeling tools to describe each type of design element. A particular design language or modeling tool may be well suited to express some aspects of system operation, but not all of them. Similarly, a different programming language may be most appropriate for implementing each type of design element.

- Data entities can be modeled very well with object-oriented design languages such as UML. All these design languages are perfectly capable of representing classes, primitive operations, and relationships between classes. Data entities are best implemented in an object-oriented programming language.
- I/O servers are best modeled with skeletal, high-level language code. Except for UML/RT, no design language or visual modeling tool even comes close to adequately describing the low-level, hardware-related issues with which an I/O server must deal.

  Ada would be the programming language of choice for both design and implementation.
- To describe the architecture of a transformation server, flow charts may be perfectly adequate. After all, a transformation server operation is nothing more than an algorithm. Any means for describing an algorithm is therefore suitable for describing the architecture of a transformation server. Transformation servers are best implemented in a structured programming language.
- The data flow and object interaction aspects of data flow managers can be modeled with almost any design language. The sequence of actions performed by a data flow manager is best modeled with skeletal, high-level language code. Any structured programming language is suitable for implementing a data flow manager.

It would be nice to have a design language or modeling tool that could visually describe the architecture of an entire system -- including all types of design elements -- in a uniform way. Although such a design language or modeling tool does not currently exist, there is an implementation language that maps surprisingly well to the elements of the universal design pattern: Ada95.[4] The popular object-oriented language C++, on the other hand, performs rather poorly when it comes to implementing data flow managers and I/O servers.

The universal design pattern is clearly inspired by real-time system issues. However, sequential batch applications also appear in the universal design pattern as an interesting special case. A sequential batch application can be modeled with one transformation server and any number of data entities, but without I/O servers or data flow managers. When we implement a simple *program*, we are really implementing a data transformation. The program takes some input data (input files), produces some output data (output files), and computes output data from input data according to a deterministic algorithm. And that's exactly what a transformation server does. Of course, I/O servers, and possibly data flow managers, are also present when we execute the simple program. They are not explicitly represented in the design though, but hidden within the operating system.

And that should explain why design approaches that are so successful for sequential applications -- object orientation and flow charts, for example -- perform so poorly for real-time systems. In truth, they can work well for real-time systems if they are applied only to the *right* types of design elements, namely transformation servers and data entities. Object orientation and flow charts are inadequate, however, for designing data flow managers and I/O servers.

Conclusion

Identifying and understanding the first principle of software development allowed us to define a set of universal design rules -- the universal design pattern -- upon which all software architecture should be based. Using the universal design pattern, a software developer should be able to demonstrate that a software architecture is sound without performing any tests. This makes software design and implementation a predictable, repeatable engineering activity.

Specifically, we have learned that:

- The hardware environment of a software system is not a constraint, but rather a primary driving force of software architecture and design. It is impossible to create quality architectures without taking the hardware environment into account at the earliest design stages. Hardware interactions are not a deployment issue and cannot be deferred or ignored during architectural design.
- At a high level, every (every!) software system can be modeled with four types of design elements that represent different aspects of system operation: *data structures and primitive operations*, *external (hardware) interfaces*, *system algorithms*, and *data flow and sequence of actions*.
- None of the design languages and modeling tools currently in use is adequate for developing and representing an entire software system. There is, however, a programming language powerful and versatile enough to implement all elements and aspects of a software system: Ada.

The universal design pattern is truly *universa*l. It applies to software of all

domains, and it can be used no matter what design method or modeling language the software developer otherwise employs. Studying the universal design pattern conveys deep insights into the very nature of software and software design. The universal design pattern has so many interesting properties that it is impossible to address them all in one article. In future issues of *The Rational Edge*, I will tell you more about these properties.

---

[1]Restricting the freedom of a software developer would be a frightening idea if software development were to remain an art or craft -- nobody wants to constrain an artist. But if software development is to become an engineering discipline, then restricting the freedom of the developer is a necessity.

[2]This may seem odd; let me elaborate. In many software systems that process real-time data, an input server sends a signal to a data processing element when input data arrives and thus invokes the data processing element. The universal design pattern does not allow servers to be pro-active in this way. A data flow manager always takes the lead by requesting input data from the server. If the data flow manager wants to wait for the data to arrive, then waiting for data becomes its action. At a very low level, of course, an I/O server thread may send a signal to a data flow manager thread to implement the conclusion of the "waiting for input" action, but we should not concern ourselves with such implementation details in the architectural design.

[3]The famous exception to this rule is "files." A file object has a hidden, embedded hardware interface. It is acceptable to use such data entities in sequential applications. But in real-time systems, data entities with hidden, embedded hardware interfaces must be strictly avoided.

[4]In fact, Ada maps so well that I can't believe it is pure coincidence. I'd like to ask the Ada designers if they had the universal design pattern in mind when they created the Ada language.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!