Architectural Analysis

System architecture exists to provide a formal structure to a software system. This is important for four reasons; to provide for future changes, to control complexity, to facilitate training, and to enable reuse. If a system is very simple, will not change, or is only intended to be understood by a few developers, then a formal architecture is not necessary. However, most software systems start out as small, focused systems that become larger and more complex over time. The result of this incremental growth is that sections and components are added without considering the application as a whole, which eventually results in what Brian Foote calls a "big ball of mud" [1]; otherwise known as an non-maintainable legacy application. To avoid this situation, all application developers should at the very least consider how the system will be used, and what future needs may develop.

Using architectural analysis to provide for future changes involves predicting how the system will be used some time in the future. Since future predictions are inherently difficult, the best way to provide for the future is to used a loosely coupled, highly coherent component based architecture with well defined interfaces to the components. All software system can benefit from this approach, which is why it is considered a best practice [2, 3].

Similarly, architectural analysis is used to control complexity by breaking the problem down into smaller sections, and then rebuilding the final solution from each of these smaller solutions. Again, component architecture has been shown to be the best approach to managing the complexity of most business domain challenges. The role of architectural analysis is to discover where the problem can be best divided. For example, in a package delivery system, there is a natural division between identification of packages, the sorting of packages by delivery destination, and the physical transport mechanisms selected based on delivery priority (e.g. air vs. ground).

The architectural analysis can also be useful for training of new developers on the system. A key part of any analysis is to document the analyst's findings; therefore an architectural analysis should result in a formal architecture document (see Appendix for an example document structure). This document then forms the basis for system training, as it contains a full description of all of the critical structural elements of the system.

Finally, reuse is facilitated by a well architected system. Reuse is directly related to the ability to abstract common functionality that can be used by many different systems. For example, transaction management is a core service that is used by virtually all software systems to ensure that information is correctly saved to a persistent store (e.g. database). Since this functionality is very similar from one implementation to the next, it is sensible to abstract the solution elements to a reusable framework; one that can be used in many software systems. This permits testing of the framework independently of the application, reducing errors since frameworks rarely changes once created.

Requirements Driven

All system development is driven by requirements, whether they are documented or not. Even verbal directions count as system requirements, albeit ones having a high tendency to be misunderstood. Architectural analysis uses these requirements to derive a system structure that provides a system form as the developers created code. This system form may be a layered architecture, where all of the system components reside in well defined system layers, or other structures (such filter/pipe architectures for flow-through signal processing). The key to this analysis is to identify the requirements that have the highest impact on overall system structure. These "architectural drivers" are typically the ones that use the highest amount of system functionality, and are the more complex areas of the application. For example, an order management system may have requirements for user security, transaction logging, and trouble ticketing, but the requirements for ordering products is more important to the overall system architecture. Focusing the analysis on these key business flows will permit the rapid identification of architectural drivers and constraints.

Balance of Forces (Drivers and Constraints)

There are two major considerations for software architectures, drivers and constraints. An architectural driver is a specific system need that must be met. For example, a key architectural driver for an order management system is Transaction Management. Therefore any system that is created must have this functionality as a central theme. For many systems another core driver is security, a key functional area that must be considered across the application. Other areas are Data Storage, Communications, System Availability, Work Flow Control, and other specific business needs. Drivers should be identified as early as possible in the development process, because they have such a dramatic impact on the structure of the final software solution.

Constraints are restrictions placed on the developing system. Constraints are often a result of existing software systems that the business upon which the business has become dependent (e.g. legacy systems). They may also result from technology restrictions, such as a particular development language, or computer platforms. Finally, constraints may take the form of business rules that must be met by the developing system, such as Sarbanes-Oxley compliance.

Analysis Patterns

> *A pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing [4].*

Experienced professionals rarely invent a new solution for each occurrence of a problem to be solved. "Experience" is how we perceive our understanding of past solutions. Often this knowledge of previous solutions leads us to develop standard approaches to common problems, such building a barn or sorting a list of values. These recurring solutions are known as "patterns."

A *pattern* is a solution to a specific problem within a defined context:

- Patterns are discovered *not* "invented"
- Patterns are only relevant when applied within a defined problem context, i.e. there are no "generic" patterns in the sense of being universally valid
- A Pattern represents an effective strategy that has been shown to work
- Patterns are identified by name

The concept of a pattern in software development originated in recognizing recurring themes that were often used to solve specific software problems:

- Splitting the interface to a system from the implementation allowed greater flexibility (Bridge Pattern)
- A collection of items can be treated as a unified container, and use the same interface to manipulate the entire tree or any sub-branch identically (Composite)
- A single copy of a design entity (e.g. object) is desired across the system (Singleton)

One of the first books to document these common software themes was "Design Patterns" by Gamma, et. al. [5] These authors described many common patterns that are found during software design and implementation. There are many patterns that apply to analysis as well.

Patterns are expressed in many forms. Software patterns often take the following form:

- **A Name** – describes the intent of the pattern
- **Problem Statement** – concise description of the problem
- **Motivation/Forces** – description of driving forces that require resolution
- **Solution** – instructions for solving the stated problem
- **Discussion** – relates the current pattern to other related patterns
- **Known Implementations** – examples of existing adaptations
- **Related Patterns** – previously defined similar patterns

Analysis models for a particular domain can also be captured as patterns. Martin Fowler is credited with coining the term *analytical patterns,* which are similar to design based patterns, but are focused on the organization and capture of problem specific information rather than as instructions to for solving a problem [6]. A recently published work by Arlow and Neustadt, has presented a refinement and novel extension of Fowler's original work [7]. These authors take a somewhat different approach by introducing the concepts of *archetypes* and *archetype patterns*. Their basic idea is that there are business archetypes -- universal concepts that occur consistently in multiple business domains. Based on this idea, the authors propose a catalog of business domain archetype patterns. For example, the first archetype they identify is a **Party** (Figure 1). Common to virtually all businesses, a Party represents an identifiable entity that may have legal status, such as a customer, supplier, or client. Parties may enter into **PartyRelationships** with specific

roles and responsibilities for each Party; for example, there is a **PartyRelationship** between a buyer and a seller of goods.

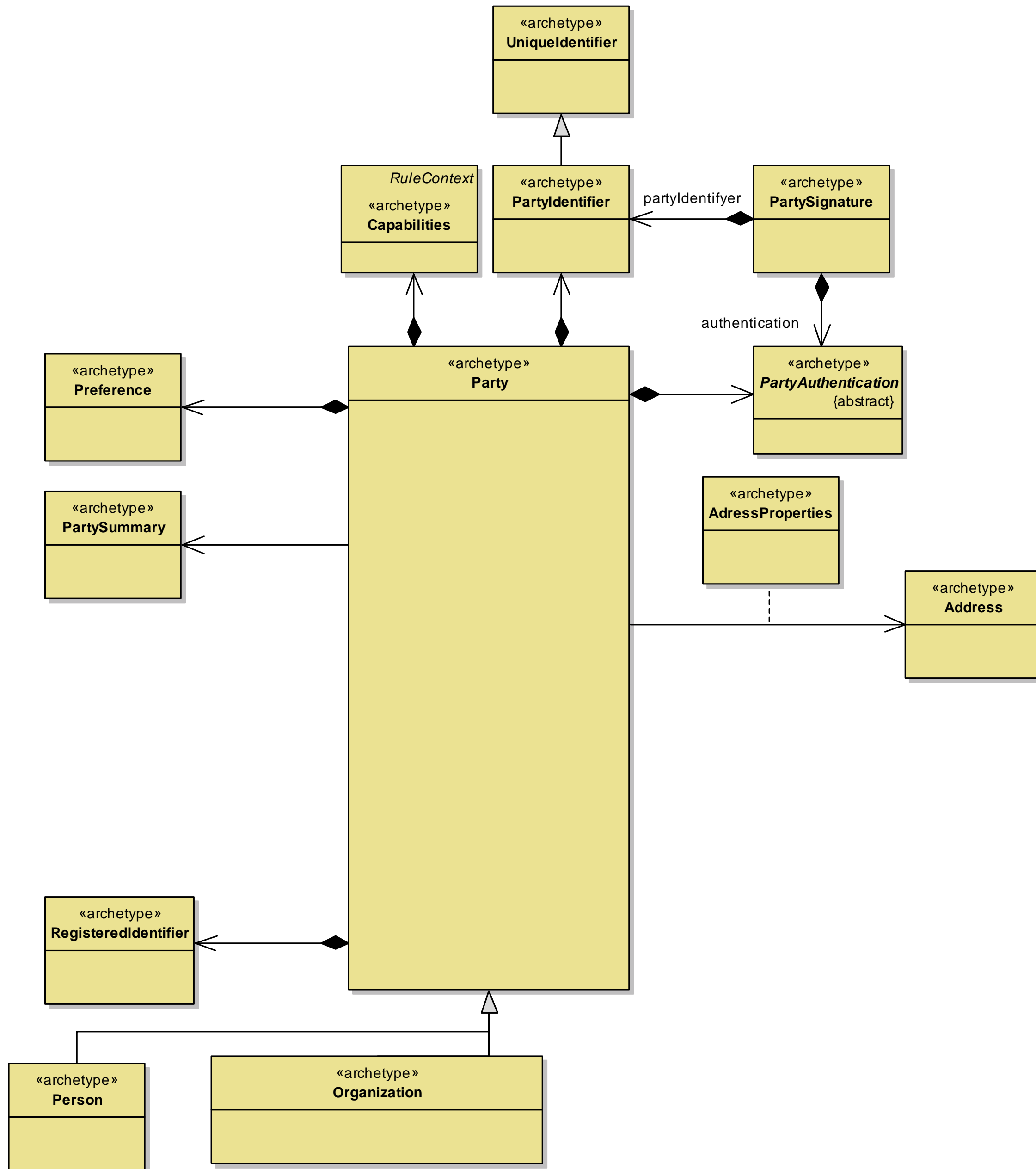

Figure 1. Archetype pattern for Party

To support the diversity found in business environments, the authors also provide mechanisms to easily modify or extend the archetypes and archetype patterns. They describe a "principle of variation," which holds that different business require related, but slightly altered, versions of the same domain elements. They divide variations into three extension categories:

1. **Archetype variation.** This involves eliminating archetype options (attributes or methods) that are not necessary to the development model.
2. **Archetype pattern variation.** This refers to including/excluding optional archetypes while maintaining critical dependencies (e.g., a PartyRelationship always requires a Party).
3. **Pleomorphism.** This refers to the wholesale modification of part of a pattern while retaining the general theme. Usually this means making an abstract concept more specific (e.g., Product becomes UniqueProduct or IdenticalProduct).

Together, these extension mechanisms and the archetype pattern catalog provide a significant advancement in analysts' ability to approach complex, but common, business problems. Because the archetype pattern catalog is based on business domains -- versus business rules, which are much more variable -- the patterns are quite stable and require very little modification to be highly useful. Thus, an Analysis Pattern represents a description of a particular problem domain rather than a description of solutions to that problem. The elements and behavior of the system are captured so that different solutions can be considered and checked against the desired behavior of the to-be-constructed system.

In addition to these more formal approaches, there are a number of more familiar analytical patterns that are used everyday, even if they are not recognized as such. For example, writers often use the pattern of outlining to model story plot and character development prior to or during a book's creation. Libraries use the analytical pattern of cataloging and indexing information to model the organization and rapid location of information. Even the creative process itself can be modeled using "mind-mapping" dependencies to model the problem of free-form conceptual association (Figure 2) [8].

Figure 2.   Mind-Mapping for Puzzles (Buzan)

Patterns have proven a very effective way to capture domain knowledge for both the software problem and solution domain in the form of analysis and design patterns. At the time of this books' writing design patterns predominate over analysis patterns. However,

I expect that problem domain analysis will eventually be supported by a diverse collection of analysis patterns. With the help of a catalog of analysis patterns a modeler will be better able to rapidly create complete and accurate models, with less chance of error and oversight.

Example System: Communications – Real Time Control

Communications systems often require real-time response with very high reliability and message fidelity. Examples of Communication Systems are:

- Satellite
- Command and Control
- Telephony
- Cable/IP

Recurring Themes (Drivers/Constraints):

- Message Security (Encryption)
- Synchronization (Synchronous/Asynchronous Transfer)
- High Available (Fail-over Control)
- High Message Fidelity (Error Detection/Resolution)

Example System: Financial – Transaction Fidelity

Financial reporting places a high value on speed and accuracy. A misplaced decimal point or a dropped digit in a transaction can have far reaching and often disastrous results! Examples of Financial Systems include:

- Banking
- Brokerage
- Insurance
- Accounting

Recurring Themes (Drivers/Constraints):

- Transaction Management (Transaction Control)
- Reporting (Filter)
- Transaction Tracking (Auditing)
- Logging (Tracking)

Example System: Ordering and Inventory – Materials Control

Ordering and Inventory control systems are very frequently seen when dealing with materials based businesses (e.g. Land's End, WalMart, Grocery)

Examples of Ordering Systems:

- Online Transaction Processing (OLTP)
- Purchasing and Procurement
- Wholesale/Retail
- Manufacturing Materials Acquisition

Examples of Inventory Systems:

- Warehouse Management
- Flow-Through Provisioning (e.g. Cellular Telephony)
- Retail Materials Management (e.g. On-Hand Inventory)
- JIT (just-in-time) Shipping

Recurring Themes (Drivers/Constraints):

- Order Creation
- Order Fulfillment
- Materials Management
- Scheduling
- Reporting

Example System: Shipping and Transport – Scheduling

Businesses and organizations are often faced with the problem of maintaining a constant flow of materials or people from one location to another. These businesses include:

Materials transport:

- Package Delivery
- Ground Trucking
- Ocean Shipping
- Air Transport

Personnel transport:

- US Military
- Forestry Service (fire suppression)
- Marketing and Sales Groups
- Consulting Organizations.

Recurring Themes (Drivers/Constraints):

- Tracking (Material or Personnel)
- Pickup/Delivery
- Flow-Through Processing
- Coordination/Synchronization

Aesthetics

In his work on building patterns, Christopher Alexander describes a good pattern as one that has a “quality without a name.” [4] He was referring to a highly sought after feeling of “wholeness” in a building or project. A good structural architecture captures the eye and mind of the viewer simply by its very existence. Similarly, a good software architecture pattern should resolve the forces of the problem context in an elegant, concise manner.

Patterns applied to the system should share the same qualities as a model (see **Set B, Part 1. System Modeling with UML**) be clear, concise, complete and most importantly, *relevant*; a pattern designed for one context will likely not satisfy another. All patterns should resolve the forces of the problem context and be presented as:

Concise – Extraneous information should be removed or cross-referenced
Complete – All necessary information should be included
Correct – No incorrect information should be presented (mis-information, non-information)

Patterns should be carefully considered for their affect on one another. A pattern applied for persistence may conflict with another attempting to limit external communication dependencies. In this manner the competing forces of the architecture are *balanced*.

Documentation and Reuse

As noted above, all patterns (analysis or implementation) are named and detailed. Documentation of patterns should follow a fairly standard form:

The **Pattern Name** which allows for design at a high level of abstraction and clearly states the purpose of the pattern

The **Problem** for which the pattern is a solution. This includes the specific context and forces that must be balanced by the solution.

The **Solution** which describes the elements, relationships, and collaborations of the solution pattern

The **Consequences** which notes the benefits, costs, and trade-offs that must be considered when applying the pattern.

In addition there may be a section describing the **Implementation** of a pattern (more typical for Design patterns than Analysis patterns) into a specific design language, and a description of other **Known Uses** of the pattern.

For documenting the architectural analysis, it is a good idea to follow a reusable form, such as the one presented in the Appendix. This structure covers all of the important features of a software system, such as the drivers/constrains, patterns in use, reusable frameworks, core communication connectors, and other valuable system information.

Assignment

Consider the typical problem of commuting to work. What patterns are present that solve the problem of moving from home to work? What are the "architectural" drivers that must be resolved in order to find a solution to the problem? How will these patterns work to solve the problem of a commuter who needs to travel by air, train or boat? Does a home office worker have any overlap with these patterns?

References

1. Foote, B. and J. Yoder, *Big Ball of Mud*, in *Pattern Languages of Program Design (4)*, N. Harrison, B. Foote, and H. Rohnert, Editors. 2000, Addison-Wesley: Boston.
2. Kruchten, P., *The Rational Unified Process, And Introduction*. Second Edition ed. 2000, Boston: Addison-Wesley.
3. IBM-Rational, *Rational Unified Process*. 2000, IBM Rational Software Corporation.
4. Alexander, C., *A Timeless Way of Building*. 1979: Christopher Alexander.
5. Gamma, E., et al., *Design Patterns*. 1995, Reading, Massachusetts: Addison-Wesley.
6. Fowler, M., *Analysis Patterns: Reusable Object Models*. 1997, Menlo Park, CA: Addison Wesley.
7. Arlow, J. and I. Neustadt, *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*. 2003, Boston, MA: Adison-Wesley.
8. Buzan, T., *How to Mind Map: Make the Most of Your Mind and Learn to Create, Organize and Plan*. 2003, London, UK: Thorsons Publishing.

Appendix – Example Software Architecture Document Structure

1. Introduction
    - 1.1. Purpose
    - 1.2. Scope
    - 1.3. Definitions, Acronyms, and Abbreviations
    - 1.4. Overview
2. Architectural Drivers and Constraints
    - 2.1. Drivers
        - «Driver 1»
        - «Driver 2»
    - 2.2. Constraints
        - «Constraint 1»
        - «Constraint 2»
3. Use-Case View
    - 3.1. Use-Cases
4. Logical View
    - 4.1. Project Overview
    - 4.2. System Layers
        - Presentation Layer
        - WebService Layer (optional)
        - Business Logic Layer
        - Network Resource Layer
        - Data Access Layer
    - 4.3. Frameworks
        - «Framework Component #1»

- «Framework Component #2»
- 4.4. Services
  - WebServices (optional)
  - Security
  - Logging
  - Validation
  - Scheduler Service
  - Tunables (System Properties)
- 4.5. External Interfaces
  - «External Interface 1»
  - «External Interface 1»
- 4.6. Components
  - Component: «component #1»
  - Component: «component #2»
- 4.7. Sub-Systems
  - «SubSystem #1»
  - «SubSystem #2»

5. Process View
   - 5.1. Use Case Realizations – «use case #1»
   - 5.2. Use Case Realizations – «use case #2»
   - 5.3. Container Processes (optional)
6. Implementation View
   - 6.1. Connectors
   - 6.2. Package Structure (Namespaces)
   - 6.3. Code Configuration Control
     - Baseline
     - Branching
     - Merging
7. Data View
   - 7.1. Data Access Mechanism(s)
   - 7.2. Data Structure
   - 7.3. Logging and Reporting