SENG 401 Notes

Brian Pho

April 17, 2018

Contents

Chapter 1

Introduction

- Software Architecture
  - about communication, what the parts are, how the parts fit together
  - structure, communication, non-functional requirements
  - the underlying structure of things
  - all architecture is design but not all design is architecture.
- Low coupling, high cohesion
- Challenges: complexity, conformity, changeability
- Architectural Pattern: a pattern of structural organization
  - Data flow systems
  - Call-and-return systems
  - Independent components
  - Virtual machines
  - Data-centered systems
- View: a representation of a system from a single perspective
- Viewpoint: specification for constructing/using a view
- Requirements = What?
- Design = How?
- Architecture is the intersection of requirements and design
- Architecture Benefits
  - Analytical capability
  - Structural visibility
  - Establish systems discipline

  - Maintain conceptual integrity

- As both the scale and complexity of the software systems increase, algorithms and data structures become less important than getting the right structure for the system

Chapter 2

4 View Model

Notation and diagrams are not as strict as UML.

2.1 Conceptual View

- The primary engineering concerns in this view are to address how the system fulfills the requirements
- Main concern: functional requirements
- Input: Functional requirements, system boundary, functional constraints (redundancy, delay, user types, etc.)
- Output: Components/Connectors, Protocols
- Components have connectors
- Conceptual view doesn't need to be very detailed
- Shows a high level overview
- It is the diagram with the small black boxes for data in/out, rectangle boxes for components, and connectors
- Connectors only go between components, not between the outside world and the component

2.2 Module View

- Module: umbrella term
- Functionality in conceptual view was logical. In module view, the implementation constraints are also accounted for
- Map a component, port, connector to a module
- Subsystem: larger modules, runs on its own node

2.2.1 Layer

  - Layer: partially ordered hierarchy
  - Based on communication, not organization/functionality
  - Too bulky = Not modular
  - Don't have cross dependencies
  - Visibility: layers can only see/depend within current layer and the next lower layer
  - Volatility: upper layers affected by requirements changes, lower layers affected by environment changes
  - Generality: more abstract model elements in lower layers
  - Avoid circular dependency

- Tier vs layer: physical separation of components (execution/deployment view) vs logical separation of components (module/conceptual view)
- Partitioning of components and connectors into modules (horizontal) and layers (vertical) is the main purpose of the module view
- It is also used to prepare and guide implementation

2.3 Execution View

- Addresses dynamic structural issues
- How to allocate functional components to runtime entities?
- Shows how modules are mapped onto the hardware of the system
- Focus on non-functional requirements
  - Performance
  - Scalability
  - Throughput
  - Distribution
  - Communication
- Multiprocessing: multiple CPUs executing concurrently
- Multitasking: one CPU executing multiple tasks
- Process vs Thread
- Inter-Process Communication (IPC): exchange of data between processes
- Remote Procedure Call (RPC): calling the procedure of another machine
- Provides better traceability among requirements, external factors, and execution environment

Chapter 3

Architectural Patterns

An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. Examples:

- Layers: different levels of abstraction
- Model-View-Controller (MVC): Model (business rules and underlying data), View (how information is displayed), Controllers (process user input)
- Pipes and filters: data is processed in streams that flow through pipes from filter to filter
- Blackboard: independent specialized applications

Patterns

- Data flow systems
  - Batch
  - Pipes and Filters
  - UNIX piping commands
- Call-and-return systems
  - Client-Server
  - Peer-to-Peer
  - Layers
- Independent components
  - Event Driven (GUI)
  - Communicating processes
- Virtual machines
  - Interpreters
  - Rule-based Systems

- Data-centered systems
  - Database systems
  - Blackboards

Notes

- Architectural Patterns are for the conceptual view and module view
- Design Patterns are for the module view and code view

Chapter 4

Components and Frameworks

- Opportunistic vs systematic reuse: accidental vs purposefully building for reuse
- Best Practices: an organized and documented set of principles, methods, and processes that increase quality and productivity of software development
- Best Practices
  - Develop Iteratively: deliver in successive series of releases
  - Manage Requirements:
  - Use Components and Architectures:
  - Model Visually:
  - Continuously Verify Quality:
  - Manage Change:
- Reuse mechanisms
  - Design Patterns: reuse of standard designs
  - Function Libraries: set of already existing functions
  - Class Libraries: unit of abstraction
  - Components: an encapsulated software unit with specified interfaces
  - Frameworks: skeleton of an application
- Framework
  - A horizontal framework provides general application that a large number of applications can use
  - A vertical framework is more specialized but still needs some slots to be filled
  - White box: inheritance based, override to customize, internal structure of classes exposed
  - Black box: composition based, configure to customize, only interfaces exposed
- Advantages
  - A lot of work is done for you

  - Shielded from technology changes
  - Greater consistency across applications

- Disadvantages
  - Tight coupling between application and framework
  - Extra learning curve
  - Framework may not provide every thing needed

Chapter 5

Midterm

- Know definitions
- Stateful vs stateless server
- Layer vs tier
- Up to components and framework slide set
- Framework vs library (reverse control, you call the library, the framework calls your code)
- Pros and cons of frameworks

Chapter 6

Industry Workshop: Best Practices

- DRY: Don't repeat yourself
- Before changing any code, write tests
- Refactoring: no changing functionality but changing the implementation
- Code smells
  - Long parameter list (can break function down)
  - Long method (doesn't fit on screen)
  - Switch statement
  - Primitive obsession
  - Shotgun surgery (changing code requires changing it in multiple spots)
- SOLID
  - Single responsibility
  - Open/closed principle
  - Liskov substitution principle
  - Interface segregation
  - Dependency inversion
- YAGNI: You ain't gonna need it

Chapter 7

Service Oriented Architecture (SOA)

- Application vs Service: applications are duplicated each time it's required, service is provided where needed
- A service provides a discrete business function that operates on data
- Service oriented architecture is an architectural style
- Service's job is to ensure that the business functionality is applied consistently, returns predictable results, and operates within the quality of service required
- Three roles:
  - Service Provider: allows access to services
  - Service Requestor: is responsible for discovering a service by searching through service descriptions
  - Service Broker: hosts a registry of service descriptions. Links requestor to a service provider
- SOA Characteristics
  - Loosely coupled: minimize dependencies
  - Contractual: adhere to agreement
  - Autonomous: control the business logic they encapsulate
  - Abstract: hides the business logic
  - Reusable: divides business logic
  - Composable: facilitate the assembly of composite services
  - Stateless: minimize retained information
  - Discoverable: self-described

Chapter 8

RESTful API

- REST: Representational State Transfer
- Design criteria
- URI: resource address
- Stateless
- Safety: the request doesn't change server state
- Idempotence: executing the same operation multiple times is the same as executing it once
- REST Principles:
    1. The key abstraction of information is a resource, named by a URI. Any information that can be named can be a resource.
    2. All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.
    3. The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes. The particular form of the representation can be negotiated between REST components.
    4. Components perform only a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST; for instance, all resources exposed via HTTP are expected to support each operation identically
    5. Idempotent operations and representation metadata are encouraged in support of caching and representation reuse.
    6. The presence of intermediaries is promoted. Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the user agent and the origin server.