An Introduction to Software Architecture and Design
7 January 2018, IEEE Rising Stars

Dr. Mark C. Paulk
Mark.Paulk@utdallas.edu, Mark.Paulk@ieee.org
What Is a Software Architecture?

The set of structures needed to reason about a system
  • which comprise software elements
  • relations among them
  • properties of both

The earliest, most fundamental, hardest-to-change design decisions.

But… why should you care about architecture?
About design?
  • as a student, you probably wrote most of your programs using a “code and fix” approach…
Every System Has an Architecture

Architecture-indifferent design
• opens the door to complexity…

Architecture-focused design

Architecture hoisting
• design the architecture with the intent of guaranteeing a goal or property of the system
• you will either find
  - code that manages the goal or property
  - a deliberate structural constraint (often with reasoning or calculations) that ensures it
Value of Documentation (Kazman 2016)

Case study of ~200K SLOC open source product

Very little architectural documentation

Team reverse-engineered the architecture (2-3 person weeks of effort) and provided the architecture to the developers
- system could be characterized as poor quality architectural design

“Committers” did not need or value the architecture documentation
- system was small enough to keep architectural details in their heads
“Outsiders” were promoted to “committers” more quickly using the architecture documentation

Decentralization occurred
  • developers looked at the documentation rather than asking one of the committers about the architecture

Committers were unwilling to maintain the architecture documentation
  • need to use tools to automatically extract and maintain architectural information
Enabling Quality Attributes

An architecture will inhibit or enable a system’s driving quality attributes.

High performance… modifiability… security… scalability… incremental subsets… reusable…

Poor downstream design or implementation decisions can always undermine an adequate architectural design.
Functional Requirements

State what the system must do, and how it must behave or react to runtime stimuli.
  - are satisfied by assigning an appropriate sequence of responsibilities throughout the design
  - assigning responsibilities to architectural elements is a fundamental architectural design decision

Functionality does not determine architecture.
  - If functionality were the only thing that mattered, you wouldn’t have to divide the system into pieces at all; a single monolithic blob with no internal structure would do just fine.
Quality Attribute

A measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders.

You can think of a quality attribute as measuring the “goodness” of a product along some dimension of interest to a stakeholder.

- A qualification of a functional requirement is an item such as how fast the function must be performed, or how resilient it must be to erroneous input.
- A qualification of the overall product is an item such as the time to deploy the product or a limitation on operational costs.
Common Quality Attributes

Availability
Interoperability
Modifiability
Performance
Security
Testability
Usability

Other quality attributes include portability, scalability, deployability, mobility, safety, ...
Availability

A property of software that it is there and ready to carry out its task when you need it to be.

Builds upon the concept of reliability by adding the notion of recovery

“Availability refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specified time interval.”

Availability is about minimizing service outage time by mitigating faults.
Availability General Scenario

Source
• Internal/external: people, hardware, software, physical infrastructure, physical environment

Stimulus
• Fault: omission, crash, incorrect timing, incorrect response

Artifact
• Processors, communication channels, persistent storage, processes

Environment
• Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response

• Prevent the fault from becoming a failure
• Detect the fault
  - log the fault
  - notify appropriate entities (people or systems)
• Recover from the fault
  - disable source of events causing the fault
  - be temporarily unavailable while repair is being effected
  - fix or mask the fault/failure or contain the damage it causes
  - operate in a degraded mode while repair is being effected
Response Measure

• Time or time interval when the system must be available
• Availability percentage
• Time to detect the fault
• Time to repair the fault
• Time or time interval in which system can be in degraded mode
• Proportion or rate of a certain class of faults that the system prevents, or handles without failing
Tactics

A tactic is a design decision that influences the achievement of a quality attribute response.

The focus of a tactic is on a single quality attribute response.
• Within a tactic, there is no consideration of tradeoffs.
• Tradeoffs must be explicitly considered and controlled by the designer.
• In this respect, tactics differ from architectural patterns, where tradeoffs are built into the pattern.
**Architectural Pattern**

Is a package of design decisions that is found repeatedly in practice

Has known properties that permit reuse

Describes a class of architectures


The Gang of Four (GoF): E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994.
Patterns Establish A Relationship

A context
- A recurring, common situation in the world that gives rise to a problem.

A problem
- outlines the problem and its variants
- describes any complementary or opposing forces
- includes quality attributes that must be met

A solution
- appropriately abstracted
- describes the architectural structures that solve the problem
- how to balance the forces at work
Discovering Patterns

Patterns are by definition found in practice
  • one does not invent them
  • one discovers them

Patterns spontaneously emerge in reaction to environmental conditions
  • as long as conditions change, new patterns will emerge
Some Architectural Patterns

Module patterns
• layered

Component-and-connector patterns
• broker
• model-view-controller
• pipe-and-filter
• client-server
• peer-to-peer
• service-oriented architecture
• publish-subscribe
• shared-data

Allocation patterns
• map-reduce
• multi-tier
Architectural Patterns Overview

Layered pattern
• defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional allowed-to-use relation among the layers.
  - usually shown graphically by stacking boxes representing layers on tope of each other

Broker pattern
• defines a runtime component, called a broker, that mediates the communication between a number of clients and servers
Model-view-controller pattern
• breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view
  - model is a representation of the application data or state, and it contains (or provides an interface to) application logic
  - view is a user interface component that either produces a representation of the model for the user or allows for some form of user input or both
  - controller manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view

Pipe-and-filter pattern
• transforms data from a system’s external inputs to its external outputs through a series of transformations performed by its filters, which are connected by pipes
Client-server pattern
• clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests

Peer-to-peer pattern
• computation is achieved by cooperating peers that request service from, and provide services to, one another across a network

Service-oriented-architecture (SOA) pattern
• computation is achieved by a set of cooperating components that provide and/or consume services over a network
  - the computation is often described using a workflow language
Publish-subscribe pattern
• components publish and subscribe to events
  - when an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers

Shared-data pattern
• communication between data accessors is mediated by a shared data store
  - control may be initiated by the data accessors or the data store
  - data is made persistent by the data store
Map-reduce pattern

- provides a framework for analyzing a large distributed set of data that will execute in parallel on a set of processors
  - parallelization allows for low latency and high availability
  - the map performs the extract and transform portions of the analysis
  - the reduce performs the loading of the results
  - extract-transform-load is sometimes used to describe the functions of the map and reduce

Multi-tier pattern

- the execution structures of many systems are organized as a set of logical groupings of components termed tiers
  - tiers may be based on a variety of criteria, such as the type of component, sharing the same execution environment, or having the same runtime purpose
Tactics vs Patterns

Tactics typically use just a single structure or computational mechanism.
• meant to address a single architectural force (quality attribute)
• the “building blocks” of design
• comparable to atoms

Patterns typically combine multiple design decisions into a package.
• are constructed from several different tactics
• patterns package tactics
• comparable to molecules
Relationships Between Tactics and Patterns

Tactics are the “building blocks” of design from which architectural patterns are created.

Most patterns are constructed from several different tactics.
- these tactics might all serve a common purpose
- they are often chosen to promote different quality attributes

A pattern is a general solution.

A documented pattern is underspecified with respect to applying it in a specific situation.
Two Perspectives

To make a pattern work...

Inherent quality attribute tradeoffs that the pattern makes
- Patterns exist to achieve certain quality attributes, and we need to compare the ones they promote (and the ones they diminish) with our needs.

Other quality attributes that the pattern isn’t directly concerned with
- but it affects...
- which are important in our application
Decided to employ ping/echo to detect failed components →

Security
• How to prevent a ping flood attack?

Performance
• How to ensure that the performance overhead of ping/echo is small?

Modifiability
• How to add ping/echo to the existing architecture?
Focus on Performance Tradeoff
Architecture in an Agile Context

The best teams may be self-organizing, but the best architectures still require technical skill, deep experience, and deep knowledge.

A focus on early and continuous release of software, where “working” is measured in terms of customer-facing features, leaves little time for addressing the kinds of cross-cutting concerns and infrastructure critical to a high-quality large-scale system.

The issue is not agile vs architecture but how to best blend agile and architecture…
Building the Foundation
Guidelines for Agile Architecture
(Booch)

All good software-intensive architectures are agile.
- a successful architecture is resilient and loosely coupled
- composed of a core set of well-reasoned design decisions
- contains some “wiggle room” that allows modifications to be made and refactorings to be done

An effective agile process will allow the architecture to grow incrementally as the system is developed and matures.
- decomposability
- separation of concerns
- near-independence of the parts

The architecture should be visible and self-evident in the code
- make the design patterns, cross-cutting concerns, and other important decisions obvious, well communicated, and defended
- may, in turn, require documentation
- “socialize” the architecture

33
Architectures Are Abstractions

Cannot be seen in the low-level implementation details

Tools aggregate abstractions
• not a panacea
• no programming language construct for layer or connector or …

Architecture reconstruction is an interpretive, interactive, iterative process

Workbench – open, integration framework
Software Architecture References


Recommended

• P.H. Feiler and D.P. Gluck, Model-Based Engineering with AADL, 2012.
• E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1994.
Questions and Answers