Software Architecture

David Garlan
Carnegie Mellon University

NASA Fault Management Workshop
New Orleans
April 2012
About me

■ **Professor of Computer Science**
  - At Carnegie Mellon University since 1990
  - Before then in industry (test and measurement)

■ **Research interests**
  - Software architecture tools and techniques
  - Self-healing and self-adaptive systems

■ **Connection with NASA**
  - Engagement since 2004
  - Sabbatical at JPL summer of 2006
  - On-going education offerings for several NASA Centers
This Talk

- What is Software Architecture?
  - Why is it important?
  - What are key principles and concepts of software architecture?
  - How can formal “architectural thinking” yield systems that better satisfy their requirements?

- Prospects for improving Fault Management through architectural design
  - How do these ideas relate to the themes of this workshop?
Examples of Software Architecture Descriptions

Figure 2. Display PostScript interpreter components.
Figure 2. Abstraction layering.

FIGURE 7. Flight Computer Operating System (The FCOS dispatcher coordinates and controls all work performed by the on-board computers.)

Software Architecture in Context

1950
- Macros
- Programming-any-which-way

1960
- Subroutines
- Separate compilation
- Programming-in-the-small

1970
- NATO SE conference
- Information hiding
- Programming-in-the-large

1980
- Inheritance
- Abstract data types
- Object-oriented Patterns
- Packages
- Pipes and filters

1990
- Software architecture
- Component-based Systems
- Integrated product lines
- Abstract architectures

2000
- 1980
- 1990
- 2010
- Vanishing system boundaries
- Democratization of Internet

2010
- Cloud computing arch
- Service-oriented arch
- Model-driven development
- Component-based Systems
- Integrated product lines

4/14/2012
Gašlan
The Promise

- **Software Architecture as critical element of an effective engineering discipline**
  - from ad hoc definition to codified principles

- **Develop systems “architecturally”**
  - improve system quality through conceptual integrity and coherence
  - support trade-off analysis & appropriate selection of architectural approaches
  - assure that the system will have desired properties by design
  - manage essential complexity; hide accidental complexity
The Big Problem

Requirements

???

Implementations

How to bridge the gap between requirements and solutions?
The Role of Software Architecture

- High level of system design
- System-level abstractions
- Satisfy high priority requirements
What is Software Architecture?

The software architecture of a computing system is the **set of structures** needed to reason about the system, which comprise software **elements**, **relations** among them and **properties** of both.

Documenting Software Architecture: Views and Beyond, 2nd Ed., Clements et al. 2010.
Issues Addressed by Architectural Design

- **Structure**: decomposition of a system into interacting components
  - assignment of function to components
  - selection of component interaction/coordination mechanisms

- **Quality attributes**: emergent system properties
  - performance, reliability, security, evolvability, testability, cost of maintenance
  - tradeoffs

- **Design principles**: conceptual integrity
  - vocabulary and rules for system composition
  - “load-bearing walls”
  - use of codified design idioms, styles, and tactics
Example: Google

Quality Attributes
- Performance
- Cost
- Availability

Source: “The Google File System”
Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Example: Google

Principles of Architectural Design

- Understand architectural drivers
  - functional; quality attributes (QA); constraints

- Identify relevant architectural approaches
  - Styles, idioms, patterns, tactics

- Understand how those approaches impact achievement of quality attributes
  - Consider tradeoffs in achieving multiple QAs

- Select the set of approaches that are optimal for the particular system
Example QA: Availability Tactics

Availability

Fault Detection
• Ping/Echo
• Heartbeat
• Exception

Fault Recovery Preparation and Repair
• Voting
• Active Redundancy
• Passive Redundancy
• Spare

Fault Recovery and Reintroduction
• Shadow
• State re-synch
• Rollback

Fault Prevention
• Removal from Service
• Transactions
• Process Monitor
This Workshop

- What architectural approaches are currently used for FM today?
- What factors influence that decision?
- What are the tradeoffs in picking one FM architecture over another?
- What can we learn by looking at positive and negative experiences of prior FM architectures?
- How can we address future challenges in FM through better understanding of architectural principles?
The End
Supplementary Slides

- Architecture Drivers
- Quality Attributes
- Styles and Tactics
- Architecture tradeoff analysis
Architectural Drivers – 1

Architectural drivers are requirements that shape the software architecture.
Functional Requirements – what the system must do.
- In architectural design we are concerned with high level function not implementation details.

Constraints – design decisions already made for the designers.
- Business/organizational (e.g., schedule)
- Technical (e.g., required use of legacy platform)

Quality Attributes – characteristics the system must possess in addition to the functionality.
Quality Attributes

- Example QAs
  - availability
  - modifiability
  - performance
  - security

- Important notes
  - There is no standard taxonomy/definitions of QA
  - Each QA has multiple aspects
  - System-level QA requirements may induce functional requirements on a subsystem

4/14/2012
Garlan
Definition: Availability is concerned with system failure and its associated consequences. A system failure occurs when a system no longer delivers a service that is consistent with its specification.
Example: Availability – 2

- Areas of concern include
  - preventing catastrophic system failures
  - detecting system failures
  - recovering successfully from system failures
  - the amount of time needed to recover from system failures
  - the frequency of system failures
  - degraded modes of operation due to system failures
Styles and Tactics

- Architectural design can be improved by reusing prior architectural approaches that have well-understood properties.
- Two of the most common forms of reuse are:
  - **Styles**: general families of systems based on overall compositional structure.
  - **Tactics**: techniques for improving quality attributes.
A (Partial) Catalogue of Styles

- **Data flow**
  - batch sequential
  - pipes and filters
  - process control

- **Call-return**
  - main program-subroutine
  - object-oriented
  - component-based
  - peer-to-peer
  - service-oriented
  - N-tiered

- **Event-based**
  - asynchronous messaging
  - publish-subscribe
  - implicit invocation
  - data-triggered

- **Data-centered**
  - repository
  - blackboard
  - shared variable
A tactic is a design decision that refines a high level style and is influential in the control of a quality attribute response.
Example QA: Availability Tactics

Availability

- Fault Detection
  - Ping/Echo
  - Heartbeat
  - Exception

- Fault Recovery Preparation and Repair
  - Voting
  - Active Redundancy
  - Passive Redundancy
  - Spare

- Fault Recovery and Reintroduction
  - Shadow
  - State re-synch
  - Rollback

- Fault Prevention
  - Removal from Service
  - Transactions
  - Process Monitor
Availability Tactics – 2

- Fault detection
  - ping/echo: when one component issues a ping and expects to receive an echo within a predefined time from another component
  - heartbeat: when one component issues a message periodically while another listens for it
  - exceptions: using exception mechanisms to raise faults when an error occurs
Architecture Tradeoff Analysis

- A tactic is usually selected because it will improve a particular QA
- But at the same time it will have an impact on other QA or other aspects of the same QA
- Example: Detection Tactics for Availability
  - Performance: number of messages, timeliness of detection
  - Testability: complexity, non-determinism
  - Modifiability: distributed/localized responsibility