Current Best Practices in Software Architecture

Session 3: Designing Software Architectures

Paul Clements
Software Engineering Institute
Carnegie Mellon University
24 August 2005
Review: From Session 1

The rise of software architecture has resulted from two trends:
- Recognition of the importance of quality attributes
- The development of very large and very complex systems

Large-scale design decisions cannot be made by programmers.
- Have limited visibility and short-term perspectives
- Trained in technology solutions to specific problems.

A software architecture
- Exists to achieve a system’s quality attributes
- Exists to allow parallel development by distributed teams (a special kind of quality attribute)
- Involves decomposing a whole into parts
- Involves system-wide design decisions such as
  - How the parts work together to achieve the system’s function and goals
Review: From Session 1

Software architecture is the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationships among them.

Architecture is important because:

- Serves as a Communication Vehicle among Stakeholders
- Architecture Constrains the Implementation
- The Development Project is Organized Around Architectural Elements
- Architecture Permits/Precludes Achievement of Quality Attributes
- Architecture is Key to Managing Change
- Architecture is Basis for Incremental Development
- Architecture is a Reusable Model

Architectures are created by engineering its structures.
Review: From Session 1

Module
- Uses
- Decomposition
- Class/Generalization
- Layers

Component-and-Connector
- Client-Server
- Concurrency
- Shared-Data
- Process

Allocation
- Work Assignment
- Deployment
- Implementation
Review: From Session 2

Each structure provides the architect with an engineering handle on some aspect of the system. Architects choose the structures they need to engineer based on the important quality attribute drivers.

Architectures are documenting by capturing views: A view is a representation of a set of architectural elements and the relations associated with them.
Review: From Session 2

We need help capturing and expressing quality attributes. Quality attribute scenarios help.

Quality attributes come from stakeholders. Use a Quality Attribute Workshop to elicit them.

Other influences on the architecture are at work also:
• Developing organization
• Technical environment
• Architect’s experience

The architect must recognize and capture these.

Organizations must recognize that an architecture can influence these very factors: An Architecture Business Cycle exists.
Architecture Business Cycle (ABC)
Creating the Architecture

How does the architect create an architecture? (Multiple choice):

a. By re-using approaches from other architectures
b. By inventing new approaches out of thin air
c. By magic
Creating the architecture

Architects primarily work by using previously-tried solutions

• Large scale: Patterns and styles
• Small scale: Tactics

Styles, patterns, and tactics represent conceptual tools in the architect’s “tool bag.”

Professional architects always keep their tool bag up to date.
Patterns and styles

The modern term is “patterns” but early papers on software architecture wrote about “software architecture styles.”

Styles in architecture were analogous to styles in houses:
- Victorian (multi-story, lots of frilly wood decorations, tall windows, basically square footprint…)
- Colonial (brick front, pillars or columns, usually symmetrical front…)
- Ranch (single-story, sprawling, not very decorated…)
Patterns and styles

Authors such as Shaw and Garlan wrote “style catalogs”

Independent component patterns
- communicating-processes
- event systems
  - implicit invocation
  - explicit invocation

Data flow patterns
- batch sequential
- pipe-and-filter
- layers

Data-centered patterns
- blackboard
- repository

Virtual machine patterns
- interpreters
- rule-based systems

Call-return patterns
- main program and subroutine
- object oriented
Then, the design patterns community arrived.

Architectural styles were clearly just patterns, whose scope of design was the whole system – that is, whose scope was the architecture.

Now, *architectural patterns* is the term in use.

There are books of architectural patterns, e.g.,
Architectural patterns

These are broadly-scoped solutions to previously encountered problems.

An architectural pattern
  • is found repeatedly in practice
  • is a package of design decisions
  • has known properties that permit reuse
  • describes a class of architectures
Architectural patterns

A pattern is determined and described by
- a set of element types
  - for example, data repositories, processes, and objects
- a set of interaction mechanisms or connectors
  - for example, subroutine calls, events, and pipes
- a topological layout of the components
- a set of semantic constraints covering topology, element behavior, and interaction mechanisms
Architectural patterns

These are widely known and include many familiar design approaches:
• Layered
• Pipe-and-filter
• Client-server
  - Thin client
  - Thick client
  - Asynchronous
  - Synchronous
  - N-tier client-server
  - Etc.
• Peer-to-peer
• Agent-based systems
• Service-oriented architectures
• Etc.

Observe:
• No “universal” list
• Patterns can be combined: e.g., layered client-server
• Patterns can be specialized
• Choice of patterns to use is not random!
Architectural patterns

These are widely known and include many familiar design approaches:
- Layered
- Pipe-and-filter
- Client-server
  - Thin client
  - Thick client
  - Asynchronous
  - Synchronous
  - N-tier client-server
  - Etc.
- Peer-to-peer
- Agent-based systems
- Service-oriented architectures
- Etc.

A pattern is determined by
- a set of element types
- a set of interaction mechanisms or connectors
- a topological layout of the components
- a set of semantic constraints for topology, element behavior, and interaction mechanisms

In addition, a pattern is described by
- when and why to use it
Patterns are coarse-grained solutions

While there are dozens (hundreds?) of patterns, there are thousands of design problems.

Expecting a complete list of patterns is not realistic.

What if we can’t find a pattern to solve our problem?
Tactics

An architectural tactic is a fine-grained design approach used to achieve a quality attribute response.

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

Stimulus: Fault occurs

Tactics to control Availability

Response: Fault masked or Repair made
Availability Tactics – 1

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
Availability Tactics – 2

Fault recovery

• voting: when processes take equivalent input and compute output values that are sent to a voter

• active redundancy: when redundant components are used to respond to events in parallel

• passive redundancy: when a primary component responds to events and informs standby components of the state updates they must make

• spare: when a standby computing platform is configured to replace failed components
Availability Tactics – 3

Fault recovery and reintroduction

- shadow operation: running a previously failed component in “shadow mode” before it is returned to service
- state resynchronization: saving a state periodically and then using it to resynchronize failed components
- checkpoint/rollback: recording a consistent state that is created periodically or in response to specific events
Availability Tactics – 4

Fault prevention

• removal from service: removing a system component from operation so it can undergo some procedure that will help it avoid failure in the future (e.g., rebooting a component prevents failures caused by memory leaks)

• transactions: the bundling of several sequential steps such that the entire bundle can be undone at once
  - prevents data from being affected if one step in a process fails
  - prevents simultaneous access to data by concurrent threads

• process monitor: Monitoring processes are used to monitor critical components, remove them from service. and re-instantiate new processes in their place.
Summary of Availability Tactics

Fault Detection
- Ping/Echo
- Heartbeat
- Exception

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

Fault Recovery and Reintroduction
- Shadow
- State Resynchronization
- Rollback

Fault Prevention
- Removal From Service
- Transactions
- Process Monitor

Fault masked or repair made
Tactics for Modifiability

Stimulus: Change arrives

Tactics to control Modifiability

Response: Changes made, tested, and deployed within time and budget
Summary of Modifiability Tactics

Stimulus: Change arrives

Response: Changes made, tested, and deployed within time and budget

Modifiability

- Localization of Changes
  - Semantic coherence
  - Anticipate expected changes
  - Generalize module
  - Limit possible options
  - Abstract common services

- Prevention of Ripple Effect
  - Hide information
  - Maintain existing interface
  - Restrict communication paths
  - Use an intermediary

- Defer Binding Time
  - Runtime registration
  - Configuration files
  - Polymorphism
  - Component replacement
  - Adherence to defined protocols
Tactics for Performance

Stimulus: Events arrive

Response: Response generated within time constraints

Performance

Resource demand
- Increase computation efficiency
- Reduce computational overhead
- Manage event rate
- Control freq. Of sampling

Resource management
- Introduce concurrency
- Maintain multiple copies
- Increase available resources

Resource arbitration
- Scheduling policy
Tactics for Security

Stimulus: Attack

- Resisting Attacks
  - Authenticate users
  - Authorize users
  - Maintain data confidentiality
  - Maintain integrity
  - Limit exposure
  - Limit access

- Detecting Attacks
  - Intrusion detection

- Recovering from an attack

Security

Response: System detects, resists, or recovers from attacks

- Identification
  - See “Availability”
  - Audit trail
Tactics for Testability

Stimulus: Completion of an increment

Testability

- Manage Input/Output
  - Record/playback
  - Separate interface from implementation
  - Specialized access routines/interfaces

- Internal monitoring
  - Built-in monitors

Response: Faults detected
Tactics for other QAs

Tactics exist for other QA’s as well.

To catalog tactics for a QA.
1. Begin with a *general scenario* for the QA of interest.
2. Capture stimulus and the response
3. Capture the broad approaches
4. Fill in specific design approaches for each
Tools – and how to use them

Tactics round out an architect’s bag of tools.
• Patterns are the large-grained solution tools.
• Tactics fill in the gaps.

But tools aren’t enough. An architect – like a carpenter -- has to know how to use the tools to build something.

Architecture – like carpentry – is more than a matter of bringing some tool out of the bag and using it on the problem.
• A hammer is not the best tool for cleaning glass.

A method for using the tools would be very helpful.
Attribute-Driven Design (ADD) Method

ADD is a step-by-step method for systematically producing the first architectural designs for a system.

ADD results
- Overall structuring decisions
- Interconnection and coordination mechanisms
- Application of patterns and tactics to specific parts of architecture
- Explicit achievement of quality attribute requirements
- NOT detailed interfaces

ADD requires as input:
- Quality attribute requirements
- Functional requirements
- Constraints
Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we’ve chosen to decompose.
Step 5: Instantiate architectural elements and allocate functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose
Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we’ve chosen to decompose.
Step 5: Instantiate architectural elements and allocate functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose
Step 2: Choose Part of the System to Decompose – 1

ADD is a decomposition method:

• Just starting out? Then the “part” is the whole system
• Otherwise, choose a part identified from an earlier iteration

All required inputs *for the part you choose to decompose* should be available. They include

• functional requirements
• quality attribute requirements
• constraints
Step 2: Choose Part of the System to Decompose – 2

How to choose? It might depend on

- Risk. Design the high-risk pieces first.
- Progress and hand-off. Design the low-risk (i.e., simple) pieces quickly, to begin implementation.
- Importance. Design the important pieces (in terms of business context) first.
- Depth first. Choose a part of the system and “drive” its design to completion
- Breadth first. Make sure there are no major unknowns lurking at the high levels.
- Prototype building. Design enough (and in the right areas) to build a prototype early on.
Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we’ve chosen to decompose.
Step 5: Instantiate architectural elements and allocate functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose
Step 3: Prioritize requirements and identify architectural drivers

Some requirements are more influential than others in the architecture and the decomposition of each module.

Influential requirements can be
- functional (e.g., training crews in flight simulator)
- quality attribute related (e.g., high security)
- business oriented (e.g., product line)

Architectural drivers are the combination of functional, quality attribute, and business requirements that “shape” the architecture or the particular module under consideration.
Step 3: Prioritize requirements and identify architectural drivers

To identify the key architectural drivers

- Locate the quality attribute scenarios that reflect the highest priority business goals relative to the module.
- Locate the quality attribute scenarios that have the most impact on the decomposition of the module.

Try to keep the number of architectural drivers to five or less.

Prioritize the architectural drivers.
Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we’ve chosen to decompose.
Step 5: Instantiate architectural elements and allocate functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we’ve chosen to decompose.

The goal of this step is to establish an overall architectural approach that satisfies the architectural drivers.

- Start by trying to apply an architectural pattern.
  - E.g. client-server
- If necessary, apply a combination of patterns.
  - E.g., layered client-server
- If necessary, augment the pattern(s) with tactics.
  - E.g., layered client-server with ping-echo interaction
Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we’ve chosen to decompose.
Step 5: Instantiate architectural elements and allocate functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose
Step 5: Instantiate architectural elements and allocate functionality

Patterns define the types of elements but not a specific number.

- A layered pattern doesn’t tell you how many layers
- A pipe-and-filter pattern doesn’t tell you how many pipes and filters
- A shared data pattern doesn’t tell you how many data repositories and data accessors

The architect now has to apply the chosen pattern(s) to define a new set of elements that conform to it.

Functionality is allocated to the instantiated elements.
Step 5: Instantiate architectural elements and allocate functionality

The responsibilities of each module type must be documented:

- This usually requires the refinement of the parent module’s responsibilities and the reallocation of its responsibilities to the child modules.

Note: This is the step that “creates” new elements.

These elements might need to be further refined – that is, decomposed and given sub-structure – during the next iteration of the method.
Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we’ve chosen to decompose.
Step 5: Instantiate architectural elements and allocate functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality

These are bookkeeping and consolidation steps.

We must “hook together” designs of different parts of the system.

We must make sure that no requirements have “fallen through the cracks”.
Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we’ve chosen to decompose.
Step 5: Instantiate architectural elements and allocate functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose
Step 8: Define interfaces for instantiated elements

The interface for each instantiated element is identified.

Interfaces consist of

- the services and properties that a element requires and produces
  - identified during the allocation of functionality
- the data and control flow information needed by each element as defined by the architectural pattern

At this point, interfaces need not be as detailed as a signature, but they document what elements need, what they can use, and on what they can depend.
Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that
    satisfies the architectural drivers associated with the part of
    the system we’ve chosen to decompose.
Step 5: Instantiate architectural elements and allocate
    functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them
    constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system
    you wish to decompose
Step 9: Verify and refine requirements and make them constraints for instantiated elements

Each child element has responsibilities that are derived partially from the decomposition of requirements of the child’s parent.

Those responsibilities must be translated into requirements that are derived and refined from the parent’s requirements.

For example, a use case that initializes the whole system can be decomposed into use cases that initialize the subsystems.
Attribute-Driven Design (ADD) Steps

Step 1: Confirm there is sufficient requirements information
Step 2: Choose part of the system to decompose
Step 3: Prioritize requirements and identify architectural drivers
Step 4: Choose design concept – patterns, styles, tactics -- that satisfies the architectural drivers associated with the part of the system we’ve chosen to decompose.
Step 5: Instantiate architectural elements and allocate functionality
Step 6: Merge designs completed thus far
Step 7: Allocate remaining functionality
Step 8: Define interfaces for instantiated elements
Step 9: Verify and refine requirements and make them constraints for instantiated elements
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose
Step 10: Repeat steps 2 through 9 for the next part of the system you wish to decompose

After each iteration, we have:

- A set of elements that decomposes an element we started the iteration with
- Each element will have
  - a collection of responsibilities
  - an interface
  - quality and functional requirements that pertain to it
  - constraints

Now we have the input for the next iteration of decomposition.
ADD: Summary

ADD is a general-purpose architecture design method.

As you can see, it
• Relies heavily on patterns and tactics
• Relies heavily on quality attribute requirements
• Results in a fully-justified architecture

We haven’t discussed architecture documentation yet, but the architect needs to document the selection and instantiation of patterns as he/she goes along.

More on that topic later.
A Picture of Architecture-Based Development -1

Development organizations who use architecture as a fundamental part of their way of doing business often define an architecture-based development process.

This seminar series will illuminate the usual parts of that process.

Typically, the first few steps are
• Analyze the business case
• Understand the architecturally significant requirements
• Create an architecture to satisfy those requirements
A Picture of Architecture-Based Development -2

We now have tools in hand to carry out these steps.

- Architecture Business Cycle (ABC) – helps us identify business case factors that will shape the architecture

- Quality Attribute Workshop (QAW) – first way to engage the stakeholders.

- QA scenarios – the way to capture QA requirements.

- ADD – a method to design an architecture to meet its functional and QA requirements.
Source of material so far

Software Architecture in Practice

Len Bass
Paul Clements
Rick Kazman

Addison Wesley 2003
Next time

Evaluating software architectures:

*How do we know that our architecture is the right one for the job?*

*Date: Watch the Web site*