Methodical Design of Software Architecture Using an Architecture Design Assistant (ArchE)

Felix Bachmann and Mark Klein
Software Engineering Institute

Sponsored by the U.S. Department of Defense
© 2005 by Carnegie Mellon University

Outline
Motivation
Principles
ArchE
Example
The Key Question

How do we systematically move from a set of requirements to a software architecture that satisfies those requirements?

The Problem

Designing is very knowledge intensive:
- The required expertise rarely resides in one place/person
- It’s unclear how/what knowledge should drive design

Knowledge requirements:
- Domain
- Quality attribute (e.g. performance, security, modifiability)
- Architectural design
- Design methodology
- ....
Our Goals

Goal: To methodically design software architectures so that they predictably meet quality attribute requirements.

Sub-goals:
- Determine/discover fundamental design principles
- Operationalize principles via method(s) (“Attribute Driven Design”)
- Investigate techniques and build prototypes for automated support (ArchE)

Outline

Motivation

Principles

ArchE

Example
Types of Requirements

Constraints – pre-specified design decisions
Features – what functions add value to the user (e.g., what the system does)
Quality Attribute – how well the system does by various measures (e.g., how timely, secure, modifiable it is)

What type of requirements drive architectural design?

Answer: Functional requirements are least important for architecture design – quality requirements and constraints are most important

Here’s some evidence:
If the only concern is functionality then a monolithic system would suffice.

However is it quite common to see:
• Redundancy structures for reliability
• Concurrency structures for performance
• Layers for modifiability
What does an architect/ArchE need to know to methodically design?

Knowledge requirements

• Quality knowledge – how to achieve required qualities in an architecture design
• Architecture design process – how to get an architecture from requirements

Our approach:

• Precisely define quality attribute requirements in terms of scenarios.
• Exploit the “structure” of quality attribute models to define the structure of well-formed architectures.
• Define transformations between architecture models, quality attribute models, quality attribute scenarios and quality attribute measures.

We have a common form for specification of quality requirements

We use quality attribute general scenarios, which are system independent, to guide the specification of quality attribute requirements.

We characterize quality attribute requirements for a specific system by a collection of concrete quality attribute scenarios. These are instances of general scenarios.

We use general scenario generation tables to construct well-formed general scenarios for each attribute.
General Scenarios

General scenarios have six parts. The “values” for each part define a vocabulary for articulating quality attribute requirements. The parts are:

- Stimulus
- Source of stimulus
- Environment in which the stimulus arrives
- Artifact influenced by the stimulus
- Response of the system to the stimulus
- Response measures

Availability Scenario Generation Table

<table>
<thead>
<tr>
<th>Source of stimulus:</th>
<th>Stimulus:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Internal to the system</td>
<td>✓ Unanticipated event</td>
</tr>
<tr>
<td>✓ External to the system</td>
<td>• Update to a data store</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment:</th>
<th>Artifact:</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Normal operation</td>
<td>✓ Process</td>
</tr>
<tr>
<td>• Degraded mode</td>
<td>• Persistent storage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response:</th>
<th>Response measures:</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ record it</td>
<td>✓ Availability percentage</td>
</tr>
<tr>
<td>✓ notify parties</td>
<td>• Time range in which the system can be in degraded mode</td>
</tr>
<tr>
<td>✓ operate in normal or degraded mode</td>
<td></td>
</tr>
</tbody>
</table>

Example Scenario:

“An unanticipated message is received by a system process during normal operation. The process has to record it, inform the appropriate parties and continue to operate in normal mode without any downtime.”
What does it mean to satisfy a quality attribute requirement?

A quality attribute requirement defines a region within the set of quality attribute measures. An architecture can be interpreted in terms of quality attribute model which, in turn, can be evaluated to determine which quality attribute value the software architecture will achieve for a particular stimulus.

If evaluated value is inside the region defined by the requirement, the requirement is satisfied.

Quality Attribute Models

Scenario includes:
- Stimulus (arrival rate)
- Response (Latency)

Latency = F(
- Arrival rate,
- Computational requirements,
- Priorities of tasks,
- ...)

Bound by scenario
Bound by either architecture or constraints
Parameters define architectural tactics

Scenario includes:
- Stimulus (arrival rate)
- Latency

Latency = F(Arrival rate, Computational requirements, Priorities of tasks, …)

Bound by scenario
Bound by architecture or constraints.

Tactics are designed to adjust the parameters.

Can work backwards – determine which values of parameters will satisfy latency, with given arrival rate, and then ask whether these values are architecturally achievable using tactics.

May also weaken constraints or requirements using tactics.

What are architectural tactics?

For the six quality attributes – availability, modifiability, performance, security, testability, usability - we have enumerated a collection of “tactics”

Formal definition: An architectural tactic is a means of satisfying a quality attribute response measure by manipulating some aspect of a quality attribute model through architectural design decisions.
An architectural tactic moves from one architecture to another where a parameter of the quality attribute model moves in a known direction.

Performance Tactics

- Performance Demand
  - Increase Computational Efficiency
  - Reduce Computational Overhead
  - Manage Event Rate
  - Control Frequency of Sampling

- Resource Management
  - Introduce Concurrency
  - Maintain Multiple Copies
  - Increase Available Resources

- Resource Arbitration
  - Scheduling Policy
  - Synchronization Policy

Events arrive

Response generated within time constraints
ArchE – Architectural Expert

ArchE is a tool intended to complement an architect during the design process.

Our vision is that
• The architect has domain knowledge and an understanding of what is feasible
• ArchE has knowledge of quality attributes and their relation to design

ArchE is emerging work at the SEI.
ArchE vis a vis any particular quality attribute

Quality attribute theories are created and change over time

We want ArchE infrastructure to be independent of any particular quality attribute
  • ArchE is modular with respect to quality attributes that are included
  • We use term “reasoning framework” to describe how quality attribute knowledge is encapsulated in ArchE.
  • We view reasoning frameworks as “plug-ins”

Process of using ArchE (current version)

Architect: provide scenarios and features to ArchE

ArchE: generates initial architecture based on reasoning frameworks and scenarios

ArchE: presents list of possible tactics to improve architecture to architect

Architect: choose tactic to apply

ArchE: apply tactic and generate new list of possible tactics
ArchE Uses Tactics to Move Architecture in the Design Space

An architectural tactic moves from one architecture to another where a parameter of the quality attribute model moves in a known direction.

Outline

Motivation

Principles

ArchE

Example
Initial Functions for Sensor Demo

- Receive data from sensor
- Correct for environmental factors
- Identify differences from past readings
- Determine sensor status
- Determine warning status of type 2
- Inform rest of system of any alerts

Key:
- Responsibility
- Data flow

Sensor input goes down two paths:
- One determines whether sensor is operational
- One calculates values and potential alerts based on values

Functions as they are entered into ArchE

1. Receive data from sensor (Receive)
2. Correct for environmental factors (Correct)
3. Determine sensor status (Status)
4. Identify warning conditions (Detect)
   4.1 Identify differences from past readings (Diff)
   4.2 Determine warning status
      4.2.1 Determine warning status of type 1 (Type 1)
      4.2.2 Determine warning status of type 2 (Type 2)
5. Inform rest of the system of any errors (Inform)
   (Demo step 1)
Scenarios for the Sample Problem

Modifiability
1. Replace sensor without change to functionality within 4 person days
2. Add new warning status without impacting existing warning statuses within 2.5 person days

Performance
1. Determine sensor status within 250 ms after receiving sensor input. Sensor input arrives every 500ms
2. Determine differences from past readings within 1250ms after receiving input. Input arrives every 1600ms.
3. Inform the rest of system of any alerts within 350ms after the arrival of alert status. Alert status arrives every 350ms.
   (Demo step 2)

Relate Scenarios to Responsibilities

Responsibilities and relations among responsibilities carry parameters.

Scenarios are not yet related to responsibilities.

Costs, execution times, and dependency are not yet assigned

Thus, there is not enough information for ArchE to determine whether the scenarios can be met.
Initial Architecture for Impact Analysis

If no assignment of responsibilities to modules then assign each responsibility from initial set to its own module.

(Demo step 3)

Retrieve parameters from architect.
- Cost of change of responsibility
- Probability of change propagating

Parameterized Values of Impact Analysis Model

(Demo step 4)

Key:
- $\circ$ Cost of changing responsibility
- $\triangleleft$ Probability of propagation
Scenarios for the Sample Problem

Modifiability
1. Replace sensor without change to functionality within 4 person days
2. Add new warning status without impacting existing warning statuses within 2.5 person days

Performance
1. Determine sensor status within 250 ms after receiving sensor input. Sensor input arrives every 500 ms
2. Determine differences from past readings within 1250 ms after receiving input. Input arrives every 1250 ms.
3. Inform the rest of system of any alerts within 350 ms after the arrival of alert status. Alert status arrives every 350 ms.

ArchE Proposes Possible Tactics

For modifiability ArchE can propose tactics like:
- Localization
- Encapsulation
- wrappers

We choose “localize”
Result after tactic “localize”

ArchE creates new responsibility and makes it dependent on affected modules

Architect must give new responsibility meaningful name and adjust dependencies, costs, probabilities, and names of affected responsibilities.

Demo step 5

Result after changing dependencies and choosing encapsulation

3.97 Days for Scenario 1

Key:
- Responsibility with cost factor
- Dependency with probability
Scenarios for the Sample Problem

Modifiability
1. Replace sensor without change to functionality within 3 person days
2. Add new warning status without impacting existing warning statuses within 2 person days

Performance
1. Determine sensor status within 250 ms after receiving sensor input. Sensor input arrives every 500ms
2. Determine differences from past readings within 1250ms after receiving input. Input arrives every 1600ms.
3. Inform the rest of system of any alerts within 350ms after the arrival of alert status. Alert status arrives every 350ms.

Responsibility
\[\text{Scenario 1 – Period (500) and Deadline (250)}\]
\[\text{Scenario 2 – Period (350) and Deadline (350)}\]
\[\text{Scenario 3 – Period (1600) and Deadline (1250)}\]

Create a task for each scenario.
Assign deadline monotonic priorities to the tasks
Evaluate Model

Scenario 1 – Period (500) and Deadline (250)
- Status
  - Latency = 150

Scenario 2 – Period (350) and Deadline (350)
- Inform
  - Latency = INF !

Scenario 3 – Period (1600) and Deadline (1250)
- Receive
  - Correct
  - Detect
  - Latency = INF !

Total utilization > 1.0 and deadlines are violated!
Tactic: Try reducing execution times of several responsibilities

(Demo step 6)

Applying Tactics -1

Scenario 1 – Period (500) and Deadline (250)
- Status
  - Latency = 130

Scenario 2 – Period (350) and Deadline (350)
- Inform
  - Latency = 360 !

Scenario 3 – Period (1600) and Deadline (1250)
- Receive
  - Correct
  - Detect
  - Latency = 2375 !

Total utilization < 1.0 but deadlines are still violated!
Tactic: Try reducing execution time of Status.

(Demo step 7)
Applying Tactics -2

Scenario 1 – Period (500) and Deadline (250)

Latency = 120

Inform

Scenario 2 – Period (350) and Deadline (350)

Inform

Latency = 345

Inform

Latency = 1980!

Receive  Detect

Scenario 3 – Period (1600) and Deadline (1250)

Receive

Detect

One deadline is still violated!

Tactic: Try increasing period of Inform.

(Demo step 8)

Applying Tactics -3

Scenario 1 – Period (500) and Deadline (250)

Latency = 120

Status

Scenario 2 – Period (385) and Deadline (350)

Inform

Latency = 345

Inform

Latency = 1410!

Receive  Detect

Scenario 3 – Period (1600) and Deadline (1250)

Receive

Detect

One deadline is still violated!

Tactic: Try increasing period of Status.
Applying Tactics -4

Responsibility (n is exec time)

Scenario 1 – Period (550) and Deadline (250)

Scenario 2 – Period (385) and Deadline (350)

Scenario 3 – Period (1600) and Deadline (1250)

Latency = 120
Latency = 345
Latency = 1065

*** Success ***

(Demo step 8)

Status

Applying ArchE to realistic examples
- ArchE has demonstrated that methodical design with predictable results is possible for small systems.
- We are looking for collaborators to help us with the extension of PAD and ArchE.

Extensions to ArchE that are underway
- Input constraints
- ArchE proposes patterns as well as tactics
- Variability reasoning framework
- Extension of performance reasoning framework
Future Work - 1
Make searching more efficient
• Patterns presented to architect as well as tactics
• Tradeoffs managed in a better fashion
• Better initial guess at architecture
• More sophisticated search
• Learning based on past choices

Future Work - 2
Make more and better reasoning frameworks
• More depth in current reasoning frameworks
• Add reasoning frameworks for other attributes (e.g., variability, security, dependability)
• Develop domain specific language for specification of reasoning frameworks
• Make ArchE more realistic
  - Apply to more sophisticated problems
  - Improve the user interface
More Information

Three SEI technical reports available on our web site:
1. *Illuminating the fundamental contributors to software architecture quality.* CMU/SEI-2002-TR-025
2. *Deriving architectural tactics: A step toward methodical architectural design* CMU/SEI-2003-TR-004

Lists of general scenarios and tactics are available in second edition of
*Software Architecture in Practice*