**Context**

- Motivation & basic terminology
- Essential concepts for architecture analysis
  - Architectural styles
  - Architecture description
  - (Re)design strategies for (OO) architectures
- Architecture analysis
  - SAAM (Software Architecture Analysis Method)
  - ATAM (Architecture Tradeoff Analysis Method)
- Analysis for rearchitecting legacy systems
Motivation &
basic terminology

Definition of software architecture (I)

Software architecture :=

software components +

the relationships among them

- top-level design of a system
- identification of core subsystems/abstractions
Definition of software architecture (II)

closely related terms/concepts:
- domain-specific software architectures, frameworks, product lines (reuse architectures)
- architectural styles
- design patterns (some of the GoF patterns focus on architecture, e.g., Facade, Mediator)
- component-based software

Benefits of architecture analysis

- Communication among stakeholders based on an explicit description of high-level abstractions of the system under development
- Early design decisions, influenced by driving quality attributes.
- Transferable abstraction of a system; can promote large-scale reuse.
Essential concepts for architecture analysis

- Architectural styles
- Architecture description
- (Re)design strategies for (OO) architectures

Architectural styles
**Architecture description based on architectural styles**

- **Data-centered:**
  - Repository
  - Blackboard

- **Data-flow:**
  - Pipes & filters
  - Batch/sequential

- **Call-and-return:**
  - Top down
  - OO
  - layered

- **Virtual machine:**
  - Interpreter
  - Rule-based

- **Independent components:**
  - Communicating processes
  - Event systems
    - implicit invocation
    - explicit invocation

*) The presentation is based on *Software Architecture in Practice* (Bass et al.; Addison-Wesley, 1998) and *Software Architecture: Perspectives on an Emerging Discipline* (Shaw, Garlan; Prentice Hall, 1996)

---

**Sample architectural style: Data-centered (I)**

Access to shared data represents the core characteristic of data-centered architectures. The data integrability forms the principal goal of such systems.

[Diagram of data flow with Client and Shared Data]
The means of communication between the components distinguishes the subtypes of the data-centered architectural style:

- **Repository: passive data** (see schematic representation of previous slide)
- **Blackboard: active data**
  A blackboard sends notification to subscribers when relevant data change (→ Observer pattern)

+ clients are quite independent of each other
  => clients can be modified without affecting others
  coupling between clients might increase performance but lessen this benefit

+ new clients can be easily added

No rigid separation of styles: When clients are independently executing processes: client/server architectural style
Sample architectural style: Data-flow

The system consists of a series of transformations on successive pieces of (input) data. Reuse and modifiability form the principal goals of such architectures.

- Batch sequential
- Pipe-and-filter (incremental)

Architecture description
Architectural styles are often insufficient to describe a system’s architecture

Architectural styles do not clearly categorize architectures. Thus they do not suffice to describe architectures as a whole.

Consider the following sample cases:

- The layers in a layered architecture might be objects/ADTs.
- Commercial client/server systems with a CORBA-based infrastructure could be described as **layered object-based process systems**, i.e., a hybrid of three styles.

Choose an appropriate mix of various notations + informal description

As a consequence, one has to decide on the appropriate description, which will be a mix of the following principal options:

- schematic representation according to CMU/SEI
- 4+1 View Model of Architecture (Kruchten, 1995)
- **UML:**
  - subsystem/package-diagrams
  - class-diagrams
  - interaction/object-diagrams
  - state-diagrams
- any schematic figures that help; informal text as glue
- source code fragments of coarse-grained components or component interfaces
- formal specifications, eg, with architecture description languages
Sample architectural description of JavaSpaces

JavaSpaces characteristics (from an architectural point of view):

- data-centered, mainly a repository architectural style, sometimes black-board architectural style
- main design goals
  - **extensibility** through loose coupling of distributed processes and distributed Java components
  - **simplicity** from a reuser's point of view

JavaSpaces architecture overview

[Diagram showing interactions between read, write, and take operations within JavaSpaces]
**Characteristics of JavaSpaces (JS)**

- **high-level coordination tool** for gluing processes and components together in a distributed system **without message passing and remote method invocation**
- A space is a shared, network-accessible repository for objects
- Instead of communicating directly, JS apps consist of processes that coordinate by exchanging objects through spaces

---

**Sample JS use scenarios (I)**

*Example isn't another way to teach, it is the only way to teach* (Albert Einstein)

**(1) Space acting as “auction room“:**
Sellers deposit for-sale items with descriptions and asking prices as objects into the space.
Buyers monitor the space for items that interest them. If an item interest them they put bid objects into the space.
Sellers monitor the space for bids.
etc.
Sample JS use scenarios (II)

(2) Compute-intensive jobs
A series of tasks—for example, rendering a frame in a computer animation represents a task—are written into a space.
Participating graphic workstations search the space for rendering tasks. Each one finding tasks to be done, removes it, accomplishes it, writes the result back into the space.

Interfaces of JS key abstractions and their usage

The rest of the architecture description could be a series of documented source code and commented UML class and interaction diagrams that illustrate the simplicity of reusing the JS architecture.

```java
public calls SampleMsg implements Entry { // empty JS interface
    ...
}

// putting an object into a space
SampleMsg msg = new SampleMsg();
JavaSpace space = SpaceAccessor.getSpace();
space.write(msg, ...); // other parameters omitted
```
(Re)design strategies for (OO) architectures

Design strategies should help to improve the quality of a(n OO) software architecture depending on the driving quality attributes

- Check an architecture’s decomposition into modules/components/objects
- Check the variation points of an object-oriented framework architecture
Appropriate modularization in theory (I)

Balance between
- maximizing the cohesion within a component
- minimizing the coupling between components

One module =>
  - coupling = 0
  - cohesion \(\rightarrow\) max

Each operation in a separate module =>
  - coupling \(\rightarrow\) max
  - cohesion = 0

Appropriate modularization in theory (II)

- Balanced distribution of responsibilities between components
- Minimal interface of a component (increases coupling within a component)
  - tight relationship between instance variables and methods
  - no redundant methods/procedures
  - small number of parameters
  - expressive and consistent naming
  - no global data
How can the theory be applied to OO systems?

SAAM focuses on cohesion and coupling from the point of view of relevant quality attributes.

In the following we consider some concepts and rules of thumb for an analysis of

- the coarse-grained design of class hierarchies
- coupling/interaction between components
- cohesion within a component
- evolution of class hierarchies
- a framework’s variation points

Coarse-grained design of class hierarchies (I)

Classes represent the basic units for structuring OO systems. Software architects group them in families, teams and subsystems.

Class families

- based on abstract classes/interfaces
  - Container classes (abstract class Container)
  - GUI components (abstract class Component)
- bottom-up or top-down development of families
- roots of class families should be light-weight, ie, avoid data representations
Class teams
- consist of several cooperating families
  - ET++: Text team consisting of Text, TextView and TextFormatter families
  - Container and Iterator families form a team
- reused as a whole
- abstractly coupled => avoidance of tight coupling

Subsystems
- explicit encapsulation of a class team
  → Facade pattern
  - reduces coupling between specific classes of teams
  - definition of an additional abstraction to make reuse easier for clients
Reducing the coupling between components through mediators

Mediator pattern effect: components can more likely be reused independently of each other:

Ensuring cohesion within a component

In order to test that property asking the following questions is helpful:

- Can a component be
  - understood
  - tested
  without knowing its embedding in a software system?
  Example: modularization of an OO discrete event simulation system

- Does a component have side effects on other components?
Evolution of class hierarchies (II)

Vertical reorganizations:
- moving common properties/protocols/parameter classes up the class hierarchy
- splitting up an overly complex class into a class family
- avoiding the overriding of too many specific methods by adding abstract classes

Horizontal reorganizations:
- splitting up a class with too many responsibilities into a class team/subsystem
- moving features into strategy classes (→ Strategy/Bridge pattern)

Why analyze variation points of frameworks?

How can one assess the quality of an OO framework architecture?
Striving for flexibility for the flexibility’s sake—by incorporating as many patterns as possible—produces an overly complex framework.

- Flexibility of has to be given in the right dosis
- Variation point identification should be an explicit activity in the framework development process and in a framework analysis
How to capture variation points

Variation points can either correspond to
● **functionality** (→ hook methods) or to
● domain abstractions (→ abstract classes/interfaces).

Layout of a (function) variation point card:

<table>
<thead>
<tr>
<th>Variation point name</th>
<th>specify degree of flexibility:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>□ adaptation without restart</td>
</tr>
<tr>
<td></td>
<td>□ adaptation by end user</td>
</tr>
<tr>
<td>general description of semantics</td>
<td></td>
</tr>
<tr>
<td>sketch variation point behavior in at least two specific situations</td>
<td></td>
</tr>
</tbody>
</table>

Variation point description

Sample (function) variation point in a reservation framework architecture:

<table>
<thead>
<tr>
<th>Rate calculation</th>
<th>specify degree of flexibility:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✔ adaptation without restart</td>
</tr>
<tr>
<td></td>
<td>□ adaptation by end user</td>
</tr>
<tr>
<td>hotel system: calculation results from the room rate * number of nights + telephone calls + mini bar consumption</td>
<td></td>
</tr>
<tr>
<td>car rental system: calculation results from the car type rate * number of days + probably rate per mile * (driven miles - free miles) + price for refilling + rate for rented extras such as a cell telephone.</td>
<td></td>
</tr>
</tbody>
</table>
### Transformation heuristics resulting from variation point analysis

<table>
<thead>
<tr>
<th>without restart</th>
<th>by end user</th>
<th>object model transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>additional hook method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Unification pattern)</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>additional hook method in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>separate hook class</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>additional hook method +</td>
</tr>
<tr>
<td></td>
<td></td>
<td>configuration tool</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>additional hook method in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>separate hook class +</td>
</tr>
<tr>
<td></td>
<td></td>
<td>configuration tool</td>
</tr>
</tbody>
</table>

### Sample class hierarchy transformation based on variation point analysis

The rate calculation variation point could imply a horizontal transformation of a class hierarchy from the Unification pattern to the Separation/Bridge pattern and vice versa:

```java
RentalItem
  ... printOffer()
  ... printInvoice()
  ... calcRate(): double

... = calcRate();
... 
```

```java
RentalItem
  ... printOffer()
  ... printInvoice()

... = calculator.calcRate(this);
... 
```

```java
RateCalculator
  ...
  ... calcRate(r: RentalItem): double

... 
```
Hints for finding/analyzing variation points

In practice, domain experts are not used to answering questions regarding generic solutions.

Ways to overcome this obstacle:
- take a look at maintenance—rank the costs associated with system changes
- investigate scenarios/use cases and check their flexibility requirements
- ask the right people :-) 

Architecture analysis:
- The SAAM
- ATAM
When and Why To Analyze Architecture -1

- Analyzing for system qualities early in the life cycle allows for a comparison of architectural options.
- When building a system
  - Architecture is the earliest artifact where trade-offs are visible.
  - Analysis should be done when deciding on architecture.
  - The reality is that analysis is often done during damage control, later in the project.

When and Why To Analyze Architecture -2.

- When acquiring a system
  - Architectural analysis is useful if the system will have a long lifetime within organization.
  - Analysis provides a mechanism for understanding how the system will evolve.
  - Analysis can also provide insight into other visible qualities.
Qualities Are Too Vague for Analysis

- Is the following system modifiable?
  - Background color of the user interface is changed merely by modifying a resource file.
  - Dozens of components must be changed to accommodate a new data file format.

- A reasonable answer is
  - yes with respect to changing background color
  - no with respect to changing file format

Qualities Are Too Vague for Analysis

- Qualities only have meaning within a context.

- SAAM specifies context through scenarios.
Scenarios

- A scenario is a brief description of a stakeholder’s interaction with a system.
- When creating scenarios, it is important to consider all stakeholders, especially:
  - end users
  - developers
  - maintainers
  - system administrators

Steps of a SAAM Evaluation

- Identify and assemble stakeholders
- Develop and prioritize scenarios
- Describe candidate architecture(s)
- Classify scenarios as direct or indirect
- Perform scenario evaluation
- Reveal scenario interactions
- Generate overall evaluation
### Step 1: Identify and Assemble Stakeholders -1

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>Schedule and budget; usefulness of system; meeting customers’ (or market’s) expectations</td>
</tr>
<tr>
<td>End user</td>
<td>Functionality, usability</td>
</tr>
<tr>
<td>Developer</td>
<td>Clarity and completeness of architecture; high cohesion and limited coupling of parts; clear interaction mechanisms</td>
</tr>
<tr>
<td>Maintainer</td>
<td>Maintainability; ability to locate places of change</td>
</tr>
</tbody>
</table>

### Step 1: Identify and Assemble Stakeholders -2

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>System administrator</td>
<td>Ease in finding sources of operational problems</td>
</tr>
<tr>
<td>Network administrator</td>
<td>Network performance, predictability</td>
</tr>
<tr>
<td>Integrator</td>
<td>Clarity and completeness of architecture; high cohesion and limited coupling of parts; clear interaction mechanisms</td>
</tr>
</tbody>
</table>
Step 1: Identify and Assemble Stakeholders -3.

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tester</td>
<td>Integrated, consistent error-handling; limited component coupling; component cohesion;</td>
</tr>
<tr>
<td>high</td>
<td></td>
</tr>
<tr>
<td>conceptual integrity</td>
<td></td>
</tr>
<tr>
<td>Application builder (if</td>
<td>Architectural clarity, completeness; interaction mechanisms; simple</td>
</tr>
<tr>
<td>product line architecture)</td>
<td>tailoring mechanisms</td>
</tr>
<tr>
<td>Representative of the domain</td>
<td>Interoperability</td>
</tr>
</tbody>
</table>

Step 2: Stakeholders Develop and Prioritize Scenarios

- Scenarios should be typical of the kinds of evolution that the system must support:
  - functionality
  - development activities
  - change activities
- Scenarios also can be chosen to give insight into the system structure.
- Scenarios should represent tasks relevant to all stakeholders.
- Rule of thumb: 10-15 prioritized scenarios
Step 3: Describe Candidate Architectures

- It is frequently necessary to elicit appropriate architectural descriptions.
- Structures chosen to describe the architecture will depend on the type of qualities to be evaluated.
- Code and functional structures are primarily used to evaluate modification scenarios.

Step 4: Classify Scenarios

- There are two classes of scenarios.
  - Direct scenarios are those that can be executed by the system without modification.
  - Indirect scenarios are those that require modifications to the system.
- The classification depends upon both the scenario and the architecture.
- For indirect scenarios we gauge the order of difficulty of each change: e.g. a person-day, person-week, person-month, person-year.
Step 5: Perform Scenario Evaluation

- For each indirect scenario
  - identify the components, data connections, control connections, and interfaces that must be added, deleted, or modified
  - estimate the difficulty of modification
- Difficulty of modification is elicited from the architect and is based on the number of components to be modified and the effect of the modifications.
- A monolithic system will score well on this step, but not on next step.

Step 6: Reveal Scenario Interactions

- When multiple indirect scenarios affect the same components, this could indicate a problem.
  - could be good, if scenarios are variants of each other
    - change background color to green
    - change background color to red
  - could be bad, indicating a potentially poor separation of concerns
    - change background color to red
    - port system to a different platform
Step 7: Generate Overall Evaluation

- Not all scenarios are equal.

- The organization must determine which scenarios are most important.

- Then the organization must decide as to whether the design is acceptable “as is” or if it must be modified.

Interaction of SAAM Steps

scenario development architecture description

classification of scenarios ➔ individual evaluation of indirect scenarios ➔ assessment of scenario interaction ➔ overall evaluation
Example: SAAM Applied to Revision Control System

- “WRCS” is a large, commercially-available revision control system.
- No documented system architecture existed prior to the evaluation.
- The purpose of the evaluation was to assess the impact of anticipated future changes.
- Three iterations were required to develop a satisfactory representation, alternating between
  - development of scenarios
  - representation of architecture

Architectural Representation of WRCS
Scenarios Used in WRCS

- User scenarios
  - compare binary file representations
  - configure the product’s toolbar
- Maintainer
  - port to another operating system
  - make minor modifications to the user interface
- Administrator
  - change access permissions for a project
  - integrate with a new development environment

Scenario Classification

- User scenarios
  - compare binary file representations: indirect
  - configure the product’s toolbar: direct
- Maintainer
  - port to another operating system: indirect
  - make minor modifications to the user interface: indirect
- Administrator
  - change access permissions for a project: direct
  - integrate with a new development environment: indirect
Scenario Interactions

- Each indirect scenario necessitated a change in some modules. This can be represented either tabularly or visually.
- The number of scenarios that affected each module can be shown with a table or graphically, with a fish-eye view.
- A fish-eye view uses size to represent areas of interest.

Scenario Interaction Table

<table>
<thead>
<tr>
<th>Module</th>
<th>No. changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>4</td>
</tr>
<tr>
<td>wrcs</td>
<td>7</td>
</tr>
<tr>
<td>diff</td>
<td>1</td>
</tr>
<tr>
<td>bindiff</td>
<td>1</td>
</tr>
<tr>
<td>pvcs2rcs</td>
<td>1</td>
</tr>
<tr>
<td>sccs2rcs</td>
<td>1</td>
</tr>
<tr>
<td>nwcalls</td>
<td>1</td>
</tr>
<tr>
<td>nwspxipx</td>
<td>1</td>
</tr>
<tr>
<td>nwnlm</td>
<td>1</td>
</tr>
<tr>
<td>hook</td>
<td>4</td>
</tr>
<tr>
<td>report</td>
<td>1</td>
</tr>
<tr>
<td>visdiff</td>
<td>3</td>
</tr>
<tr>
<td>ctrls</td>
<td>2</td>
</tr>
</tbody>
</table>
Lessons Learned from WRCS

- Granularity of architectural description
- Interpretation of scenario interactions
Proper Granularity of Architectural Description

- The level of detail of architectural description is determined by the scenarios chosen.

- The next slide shows what an architect thought was an appropriate level of detail.

- Components are annotated with the numbers of indirect scenarios that affect them.

Original Representation of WRCS
The “main” Scenario
Interactions

- Possibilities:
  - Scenarios are all of the same class.
  - Scenarios are of different classes and “main” cannot be subdivided.
  - Scenarios are of different classes, and “main” can be subdivided.

WRCS: What did we learn?

- We identified severe limitations in achieving the desired portability and modifiability. A major system redesign was recommended.
- The evaluation itself obtained mixed results.
  - Senior developers/managers found it important and useful.
  - Developers regarded this as just an academic exercise.
- SAAM allowed insight into capabilities and limitations that weren’t easily achieved otherwise.
- This was accomplished with only scant knowledge of the internal workings of WRCS.
Lessons from SAAM -1

- Direct scenarios provide a
  - first-order differentiation mechanism for competing architectures
  - mechanism for eliciting and understanding structures of architectures (both static and dynamic)

- It is important to have stakeholders present at evaluation meetings.
  - Stakeholders find it to be educational.
  - Architectural evaluators may not have the experience to keep presenters “honest.”

Lessons from SAAM -2.

- SAAM and traditional architectural metrics
  - Coupling and cohesion metrics do not represent different patterns of use.
  - High scenario interaction shows low cohesion.
  - A scenario with widespread hits shows high coupling.
  - Both are tied to the context of use.
  - SAAM provides a means of sharpening the use of coupling and cohesion metrics.
Summary

- A SAAM evaluation produces
  - technical results: provides insight into system capabilities
  - social results
    - forces some documentation of architecture
    - acts as communication vehicle among stakeholders

Architecture analysis: The ATAM
Why Analyze an Architecture?

- All design involves tradeoffs.
- A software architecture is the earliest life-cycle artifact that embodies significant design decisions: choices and tradeoffs.
The ATAM

- We have been developing the Architecture Tradeoff Analysis Method (ATAM) for over two years.

- The purpose of ATAM is: to assess the consequences of architectural decision alternatives in light of quality attribute requirements.

Purpose of ATAM - 1

- We need a method in which the right questions are asked early to:
  - Discover risks -- alternatives that might create future problems in some quality attribute
  - Discover sensitivity points -- alternatives for which a slight change makes a significant difference in some quality attribute
  - Discover tradeoffs -- decisions affecting more than one quality attribute
Purpose of ATAM - 2.

- The purpose of an ATAM is NOT to provide precise analyses . . . the purpose IS to discover risks created by architectural decisions.
- We want to find *trends*: correlation between architectural decisions and predictions of system properties.
- Discovered risks can then be made the focus of mitigation activities: e.g. further design, further analysis, prototyping.

ATAM Benefits

- There are a number of benefits from performing ATAM analyses:
  - Clarified quality attribute requirements
  - Improved architecture documentation
  - Documented basis for architectural decisions
  - Identified risks early in the life-cycle
  - Increased communication among stakeholders
- The results are improved architectures.
Outline

- Why analyze an architecture?
- ATAM Steps
- An example
- Summary and Status

ATAM Steps

1. Present ATAM
2. Present business drivers
3. Present architecture
4. Identify architectural styles
5. Generate quality attribute utility tree
6. Elicit and analyze architectural styles
7. Generate seed scenarios
8. Brainstorm and prioritize scenarios
9. Map scenarios onto styles
10. Present out-brief and/or write report
1. Present ATAM

Evaluation Team presents an overview of ATAM including:
- ATAM steps in brief
- techniques
  - utility tree generation
  - style-based elicitation/analysis
  - scenario brainstorming/mapping
- outputs
  - scenarios
  - architectural styles
  - quality attribute questions
  - risks and “non-risks
  - utility tree

2. Present Business Drivers

ATAM customer representative describes the system’s business drivers including:
- business context for the system
- high-level functional requirements
- high-level quality attribute requirements
  - architectural drivers: quality attributes that “shape” the architecture
  - critical requirements: quality attributes most central to the system’s success
3. Present Architecture

- Architect presents an overview of the architecture including:
  - technical constraints such as an OS, hardware, or middle-ware prescribed for use
  - other systems with which the system must interact
  - architectural approaches used to meet quality attribute requirements
- Evaluation team begins probing for:
  - risks
  - architectural styles

4. Identify Architectural Styles

- High-level overview of architecture is completed by itemizing architectural styles found in the architecture.
- Examples:
  - client-server
  - 3-tier
  - pipeline
  - publish-subscribe
5. Generate Quality Attribute Utility Tree

- Identify, prioritize and refine the most important quality attribute goals by building a utility tree.
  - a utility tree is an AHP (analytic hierarchy process)-like model of the “driving” attribute-specific requirements
  - typically performance, modifiability, security, and availability are the high-level nodes
  - scenarios are leaves of utility tree
- Output: a prioritization of specific quality attribute requirements.

### Utility Tree Construction -1

```
Utility
  Performance       Modifiability      Availability
    New sensors     New middleware    Change Web UI
        Live upgrade     Quick restart after disk failure     Survive a single network failure
```
### Utility Tree Construction -2

#### Utility

- **Performance**
  - (0.2) New sensors
  - (0.2) Live upgrade

- **Modifiability**
  - (0.5) New middleware
  - (0.1) Quick restart after disk failure

- **Availability**
  - (0.3) Change Web UI
  - (0.7) Survive a single network failure

---

### 6. Elicit and Analyze Architecture Styles

- Evaluation Team probes architectural styles from the point of view of specific quality attributes to identify risks.
  - Identify the styles which pertain to the highest priority quality attribute requirements
  - Generate quality-attribute specific questions for highest priority quality attribute requirement
  - Ask quality-attribute specific questions
  - Identify and record risks and non-risks
Concurrent Pipelines Style

Quality Attribute Questions

- Quality attribute questions probe styles to elicit architectural decisions which bear on quality attribute requirements.
- Performance
  - How are priorities assigned to processes?
  - What are the message arrival rates?
- Modifiability
  - Are there any places where layers/facades are circumvented?
  - What components rely on detailed knowledge of message formats?
While risks are potentially problematic architectural decisions, …

Non-risks are good decisions relying on implicit assumptions.

Risk and non-risk constituents

- architectural decision
- quality attribute requirement
- rationale

Sensitivity points are candidate risks and risks are candidate tradeoff points.

Example risk

- Rules for writing business logic modules in the second tier of your 3-tier style are not clearly articulated. This could result in replication of functionality thereby compromising modifiability of the third tier.

Example non-risk

- Assuming message arrival rates of once per second, a processing time of less than 30 ms, and the existence of one higher priority process, a 1 second soft deadline seems reasonable.
Scenarios are example stimuli used to
  - Represent stakeholders’ interests
  - Understand quality attribute requirements
Seed scenarios are sample scenarios
Scenarios are specific
  - anticipated uses of (use cases),
  - anticipated changes to (growth scenarios), or
  - unanticipated stresses (exploratory) to the system.

Stakeholders generate scenarios using a brainstorming process.

Each stakeholder is allocated a number of votes roughly equal to 0.3 x #scenarios

Prioritized scenarios are compared with the utility tree and differences are reconciled.
Example Scenarios

- Use case
  - Remote user comes via the web to access report database.

- Growth scenario
  - Add a new data server to reduce latency by 50%.

- Exploratory scenario
  - Half of the servers go down during operation.

=> Scenarios should be as specific as possible.

9. Map Scenarios onto Styles

- Identify styles and components within styles impacted by each scenario.

- Continue identifying risks and non-risks.

- Continue annotating architectural information.
10. Present Out-Brief/Write Report

- Recapitulate steps of ATAM
- Present ATAM outputs
  - styles
  - scenarios
  - questions
  - utility tree
  - risks
  - non-risks
- Offer recommendations

Outline

- Why analyze an architecture?
- ATAM Steps
  - An example
- Summary and Status
2. Present Business Drivers

- A distributed battlefield management system (BMS)
  - One mobile central commander node
  - A set of mobile fighter nodes under commander
  - Information from many sources/sensors
  - Messages of different types (maps, orders)

- Stakeholders wanted to understand how the system would perform and adapt to changes

3. Present Architecture

- Physical view: “customer-providers”, where the commander node is the customer and the fighter nodes are providers.
- Detailed information also collected for concurrency and code views.
We elicited information on the architectural approaches with respect to modifiability, availability, and performance.

- For availability, a backup commander scheme was described.
- For modifiability, standard subsystem organizational patterns were described.
- For performance, an independent communicating components style was described.

### 5. Generate Quality Attribute Utility Tree

Utility

- **Performance**
  - Ballistics kernel computation
  - New message formats

- **Modifiability**
  - New message data types

- **Availability**
  - Server failure

Weights:
- Performance: 0.3
- Modifiability: 0.2
- Availability: 0.5
6. Elicit and Analyze Architecture Styles

- The repair time for the system is the time to turn the backup into the commander node.

- Communication between the commander node and the backup keeps the backup “in sync”.

Availability Analysis - 1

- \( Q_A \) = the fraction of time the system is working
- The system is considered to be working if there is a working commander node and one or more fighter nodes.
- When the commander node fails the system has failed.
- Provisions have been made in the BMS architecture to turn a designated fighter (backup) node into a commander node.
Availability Analysis - 2

- Availability can be seen as:
  \[ Q_A = h(\lambda_c, \lambda_b, \mu_c, \mu_b) \]
  where \( \lambda_c \) = failure rate of the commander
  \( \lambda_b \) = failure rate of the backup
  \( \mu_c \) = repair rate of the commander
  \( \mu_b \) = repair rate of the backup
- Problem! The backup has no backup, i.e. in the BMS architecture, \( \mu_b = 0 \)
- We discovered this problem via qualitative analysis questions that focused on failure and repair rates.

Availability Analysis - 3

- Hence, two well-aimed hits (or hardware failures) disable the entire system!
- The solution was to turn more fighter nodes into potential backups.
- Alternatives could be:
  - Acknowledging backups \((n)\)
  - Passive backups \((m)\)
  - Passive backups \((m) + update\)
Availability: Sensitivity/Risk Identification

- The availability of the system can now be seen as:

\[ Q_A = j(n, m) \]

- \( n \) and \( m \) are architectural availability sensitivity points
- Since availability is a key attribute for the battle management mission, some choices of \( n \) and \( m \) present availability risks

7. Generate Seed Scenarios

- Initial set of seed scenarios were too general
  - “System fails”

- Seed scenarios were later refined
  - “Command node is destroyed and the Backup node takes over as the Commander node”
46 scenarios were collected, covering modifiability, scalability, availability, performance, portability.

Examples:
- Modifiability: map data formats change
- Performance: the number of simultaneous missions doubles
- Availability: the commander is disable by a direct hit

Scenario Prioritization

- The stakeholders suggested groupings of scenarios.
- The stakeholders used preference-voting to prioritize scenarios.
- The result was 15 high priority scenarios.
9. Map Scenarios onto Styles

- The architects mapped each of the high-priority scenarios onto the BMS architecture.

- During this stage we:
  - Gathered attribute-specific information qualitative attribute questions
  - Clarified our understanding of the architecture and the scenarios
  - Documented the answers

Performance Analysis - 1

- We discovered a performance problem via a qualitative attribute questions that asks about the relative speeds of communication and processing.

- The problem uncovered was: the nodes in the BMS architecture communicated via slow modems.
Performance Analysis -2

- End-to-end latency calculations showed that the overall latency was highly sensitive to the number and size of transmitted messages.

- Communication load came from:
  - The normal operations communication overhead
  - The number of backups (both acknowledging and passive)

Performance: Sensitivity/Risk Identification

- Thus, system performance can be characterized as:
  - \( Q_P = k(n, m, CO) \)

- Communications overhead was a constant.

- \( n \) and \( m \) are architectural performance sensitivity points.
Tradeoff Identification

- Increasing the number of backups increases availability, but also increases average latency (because these backups must be kept up-to-date by the commander).

- Hence, the number of active and passive backups (n and m) is a tradeoff point in the BMS architecture.

- The designers had *not* been aware of the tradeoff inherent in their design.

10. Present Out-Brief/Write Report

- Presentation and written report detailed the potential modifiability, performance, and availability problems, and ...

- delineated new architecture options and their costs:
  - Acknowledging backups
  - Passive backups
  - Passive backups + updates
Results of the BMS ATAM

- Greatly improved architectural documentation
- Stakeholder buy-in
- Discovery of missing performance and availability requirements
- Highlighting of a previously unknown tradeoff point in the architecture
- Delineation of recommendations to mitigate the risks of this tradeoff

Outline

- Why analyze an architecture?
- ATAM Steps
- An example
  - Summary and Status
Summary - 1

- ATAM is a method for evaluating an architecture with respect to multiple quality attributes.

- It is an effective risk mitigation strategy to avoid the disastrous consequences of a poor architecture. ATAM:
  - can be done early
  - requires stakeholder participation

- The key to the method is looking for *trends*, not in making precise analyses.

Summary - 2

- ATAM relies critically on
  - Clearly-articulated quality attribute requirements
  - Active stakeholder participation
  - Active participation by the architect
  - Familiarity with architectural styles and analytic models
Analysis for rearchitecting legacy systems

Which systems get reengineered?

Modifyability

maintain          enhance

discard          reengineer

Value
Rearchitecting for improving the modifyability of an architecture

- Getting rid of replicated code/functionality increases the modifyability  
  Analyse existing source code in order to find replicated code fragments that vary a bit. Replicated functionality is more difficult to find, as the corresponding code fragments are often quite different.

- In most cases these variations don’t allow simple packaging into procedures with some parameters  
  => small frameworks are helpful

Case study: replicated code in C/S systems

[Diagram showing the replication of code in a C/S system with repeated RPC calls and ListHandl.]
A framelet
- does not assume main control of an application,
- is small in size (< 10 classes/components),
- has a clearly defined simple interface.

The term framelet just makes explicit that it is a small framework.
Thus, framework construction principles can be applied.
Sample framelet components

List-Handling (GUI-Bean)

parameters: dialog, class of a list item
reuse effect: about 200 LOC (at hundreds of spots)
reuse effort in a Beans environment: 3–4 minutes
**RPC (Non-GUI-Bean)**

**parameter:**
RPC name
(RP parameters are assembled automatically)

**reuse effect:**
about 300 LOC (at hundreds of spots)

---

**Reflection as more flexible alternative to abstract classes & interfaces**

Reflection can be useful to **automate component configuration**:

- assumes only that components support reflection, eg, checking the methods of a component
- semantics have to be defined on the domain level eg, naming conventions
- coupling is based on these semantics

Components of a framelet as well as different framelets can be coupled via reflection.
Coupling semantics defined separately

The simplest way to define coupling semantics is a naming convention.

- exchange values of instance variables with the same name and compatible types
- domain-specific conventions

Object

programming language

Experience with framelet-based rearchitecting

- significant effort required for the identification of replicated functionality
- good news: that effort pays off as legacy systems get streamlined
- framework technology useful to define small components with variation points
  => hands-on experience with frameworks
- design and implementation of framelets implies a small component library with highly reusable assets that might also be adaptable for other applications
Where Have We Been?

Hopefully, by now, you are convinced that:
- it is a good idea to have a software architecture
- this architecture should be documented
- this architecture should be communicated to the system’s stakeholders
- this architecture should be analyzed
To get insights into a system's qualities that could not be achieved via design inspections, code inspections,...

We can analyze to understand:
- how to rearchitect a system
- the eventual modifiability of a system
- where architectural tradeoffs occur

Architecture analysis is easy to do.
Architecture analysis is (relatively) cheap.
It can be done early, when discovered problems are still fixable at a reasonable cost.
It improves the quality of an architecture as well as the stakeholders’ understanding and confidence.
It is a way of enhancing team communication.
It is an effective risk mitigation technique.
Appendix A

Overview of architectural styles

Data-centered:
- Repository
- Blackboard

Data-flow:
- Pipes & filters
- Batch/sequential

Call-and-return:
- Top down
- OO
- layered

Virtual machine:
- Interpreter
- Rule-based

Independent components:
- Communicating processes
- Event systems
  - implicit invocation
  - explicit invocation

*) The presentation is based on Software Architecture in Practice (Bass et al.; Addison-Wesley, 1998) and Software Architecture: Perspectives on an Emerging Discipline (Shaw, Garlan; Prentice Hall, 1996)
Access to shared data represents the core characteristic of data-centered architectures. The data integrability forms the principal goal of such systems.

The means of communication between the components distinguishes the subtypes of the data-centered architectural style:

- **Repository: passive data** (see schematic representation of previous slide)
- **Blackboard: active data**
  A blackboard sends notification to subscribers when relevant data change (→ Observer pattern)
Data-centered (III)

- clients are quite independent of each other
  => clients can be modified without affecting others
  coupling between clients might increase performance but
  lessen this benefit

- new clients can be easily added

No rigid separation of styles: When clients are independently
executing processes: client/server architectural style

Data-flow

The system consists of a series of transformations on successive
pieces of (input) data. Reuse and modifiability form the principal
goals of such architectures.

Tape → Validate → Sort → Report → Page

Legend:

- data flow

- process
Data-flow substyles

- **Batch sequential** (→ sample on previous slide)
  - Components (= processing steps) are independent programs
  - Each step runs to completion before the next step starts, i.e., each batch of data is transmitted as a whole between steps

- **Pipe-and-filter** (→ UNIX pipes & filters)
  - Incremental transformation of data based on streams
  - Filters are stream transducers and use little contextual information and retain no state information between instantiations
  - Pipes are stateless and just move data between filters

Pros and cons of pipes-and-filters

- No complex component interactions to manage
- Filters are black boxes
- Pipes and filters can be hierarchically composed

- Batch mentality => hardly suitable for interactive applications
- Filter ordering can be difficult; filters cannot interact cooperatively to solve a problem
- Performance is often poor
  - Parsing/unparsing overhead due to lowest common denominator data representation
- Filters which require all input for output production have to create unlimited buffers
Virtual machine (I)

Virtual machines simulate some functionality that is not native to the hardware/software on which it is implemented. This supports achieving the quality attribute of portability.

Examples:
- interpreters
- command language processors
- rule-based systems

Virtual machine (II)

Schematic representation:

- Inputs
  - Data (program state)
  - Interpretation Engine
- Program being interpreted
  - Internal state
  - Program instructions
  - Selected instruction
  - Selected data
- Outputs
  - Data updates
  - Selected data
  - State data
Call-and-return architectures rely on the well-known abstraction of procedures/functions/methods. Shaw and Garlan discern between the following substyles:

- main-program-and-subroutine style
- remote-procedure-call systems also belong to this category but are decomposed in parts that live on computers connected via a network
- object-oriented or abstract-data-type style
- layered style

Layered style

Components belong to layers. In pure layered systems each level should communicate only with its immediate neighbors.

Each successive layer is built on its predecessor, hiding the lower layer and providing some services that the upper layers make use of. Upper layers often form virtual machines.
Event systems

Publish/subscribe (observer) pattern: Components can register an interest in notifications.

Example: coupling between JavaBeans

Heterogeneous styles (I)

Example: event system + layered style
Heterogeneous styles (II)

In general, the presented architectural styles do not clearly categorize architectures. Styles exist as cognitive aids and communication cues.

- The data-centered style, composed out of thread-independent clients is like an independent component architecture.
- The layers in a layered architecture might be objects/ADTs.
- The components in a pipe-and-filter architecture are usually independently operating processes and thus also correspond to an independent component architecture.
- Commercial client/server systems with a CORBA-based infrastructure could be described as layered object-based process systems, i.e., a hybrid of three styles.

Appendix B—Bibliography (I)

Fayad M., Schmidt D., Johnson R. (1999) Implementing Application Frameworks: Object-Oriented Frameworks at Work, Wiley

comprehensive architecture descriptions of real-world software systems:
Wirnh N, Gutknecht J. (1993) Project Oberon—The Design of an Operating System and Compiler, Addison-Wesley
Appendix B—Bibliography (II)

Bibliography on Software Architecture Analysis (http://www.fit.ac.jp/~zhao/pub/sa.html), maintained by Jianjun Zhao

This is the bibliography on software architecture analysis, with special emphasis on architectural-level understanding, testing, debugging, reverse engineering, re-engineering, maintenance, and complexity measurement.

R. Balzer, "Instrumenting, Monitoring and Debugging Software Architectures."


Appendix B—Bibliography (III)


Appendix B—Bibliography (IV)


Other Links on Software Architecture

Bibliographies:

- SEI Bibliography on Software Architecture: http://www.sei.cmu.edu/architecture/bibpart1.html

Others:


Appendix B—Bibliography (V)


J. Zhao, "Using Dependence Analysis to Support Software Architecture Understanding," in M. Li (Ed.), "New Technologies on Computer Software...


Other Links on Software Architecture

Bibliographies:

- SEI Bibliography on Software Architecture: http://www.sei.cmu.edu/architecture/bibpart1.html

Others:


Maintained by Jianjun Zhao (zhao@cs.fit.ac.jp)