Requirements

System Design
(Architecture, High-level Design)

Module Design
(Program Design, Detailed Design)

Implementation of Units (classes, procedures, functions)

Unit testing

Module Testing
(Integration testing of units)

System Testing
(Integration testing of modules)

Acceptance Test
(Release testing)

Validation
Verify System Design
Verify Module Design
Verify Implementation

Validate Requirements, Verify Specification

Maintenance

Project Management, Software Quality Assurance (SQA), Supporting Tools, Education
Why should we design a system?

Requirements

Why not?

Implementation

Carol the customer

Harry the hacker
Constructing a building...

I need a tower, with a big clock...

The king's requirements

Construction
Constructing a building...

The king's requirements

Architecture

Ulla

Construction
Constructing software...

**Software is different**

- No physical natural order of construction (e.g. start with the foundation of the house)
- Software is not tangible
Constructing software...
Why design and document software architectures?

**Communication between stakeholders**
A high-level presentation of the system.
Use for understanding, negotiation and communication.

**Early design decisions**
Profound effect on the systems quality attributes, e.g. performance, availability, maintainability etc.

**Large-scale reuse**
If similar system have common requirements, modules can be identified and reused. (Bass et.al., 2003)
Analyze and Synthesis a system (decompose and compose)

Requirements

System Design
  (Architecture, High-level Design)
  Divide into "virtual" modules
  Design each module

Module Design
  (Program Design, Detailed Design)

Implementation
  of Units (classes, procedures, functions)

Imagine a "virtual" System

a "concrete" System

Acceptance Test
  (Release testing)

System Testing
  (Integration testing of modules)

Module Testing
  (Integration testing of units)

Unit testing
Analyze and Synthesis a system (decompose and compose)

- Requirements
- System Design (Architecture, High-level Design)
- Module Design (Program Design, Detailed Design)
- Implementation of Units (classes, procedures, functions)
- Unit testing
- Module Testing (Integration testing of units)
- System Testing (Integration testing of modules)
- Acceptance Test (Release testing)

Imagine a "virtual" System
- Design each module

a "concrete" System

Design is an iterative process!
- Throw away Prototyping
- Evolutionary Prototyping
Box-and-line diagrams...

- **Encryption / Decryption**
- **Logging**
- **Packet Handler**
- **Session Handler**
- **Identification & Authentication**
- **User Database**

Module, Subsystem, Element, Entity, Component... (many names)

Relationship, shows data and/or control flow

Interface
Architectural views

Implementation (code) view

Execution view

Deployment view

Cryptographic Module

On different machines?

Packages, components, artifacts

Components, connectors, sub-systems (box-and-line)

One machine? Different CPUs?
Well-known Diagrams of UML in architecture

- **UML 2.5 Diagram**
  - **Structure Diagram**
    - **Class Diagram**
    - **Object Diagram**
  - **Behavior Diagram**
    - **Package Diagram**
    - **Component Diagram**
    - **Deployment Diagram**
    - **Sequence Diagram**
    - **Use-Case Diagram**
    - **State Machine Diagram**
    - **Interaction Diagram**
Implementation view with packages

A developer’s perspective:
1. What are we going to develop?
2. Where is the code?

Packages can be used to give an overall structure to other things than code, e.g., Use-cases and Classes.

Package
- Organize work
- Compile together
- Name space

DIAGRAM:
- GUI
- Transaction manager
- Encryption/decryption
- Storage manager

Dependency arrows indicate relationships between the components.
Component diagram with interfaces

Dictionary

Older notation:

Alternative notation:

<<component>>
Subsystem with components

<<subsystem>> word-book

Dictionary

Search engine

port

delegation connector
Artifacts

Physical code, file, or library

The artifact implements the component
Deployment view in UML

Node, physical hardware

<<client>>
<<artifact>>
clientCrypto.jar

<<protocol>>
TCP/IP

<<use>>
<<server>>
<<artifact>>
serverCrypto.jar

Communication path
Coupling - dependency between modules

- **Uncoupled** - no dependencies
- **Loosely coupled** - few dependencies
- **Highly coupled** - many dependencies

What do we want?

- **Low coupling. Why?**
  - Replaceable
  - Enable changes
  - Testable - isolate faults
  - Understandable
Cohesion - relation between internal parts of the module

**What do we want?**

High cohesion. Why?

- More understandable
- Easier to maintain

---

**Low cohesion** - the parts e.g. functions have less or nothing in common.

**Medium cohesion** - some logically related function. E.g. IO related functions.

**High cohesion** - does only what it is designed for.
Several factors - sometimes overlap

Non-functional requirements...

Portability

Maintainability

Modifiability

Reliability

Availability

Testability

Performance

Usability

Safety

Scalability
Performance - timing

**Timing**
- Throughput
- Response time (interactive system)

Can our architecture be parallelized?

Scale up...

Scale out...
Security

**Confidentiality**
- Only authorized users can read the information
- E.g. Military

**Integrity**
- Only authorized users can modify, edit or delete data.
- E.g. bank systems

**Availability**
- Right information is available at the right time
- Important for everyone
Safety - absence of critical faults

How can we validate that a safety critical system is correct?

- Formal validation?
- Testing?

Design so that all safety critical operations are located in one or few modules / subsystems.
Modifiability - cost of change

What can change?
- Platform?
- Function?
- Protocols?
- Environment?

When can change?
- Source code?
- Compiler option?
- Library?
- Setup config?
- At runtime?

Portability
- OS (UNIX, Windows, Real-time OS, Mac,...)
- Memory consumption
- CPU (Big-endian?)
- Computation power

Maintainability
- Low coupling
- Consistent with code
- Enough details
Usability - How easy is it and what support exists to perform a task

- Easy to learn system features
  - E.g. a word-processor or app

- Using the system efficiently
  - E.g. Latex, or UNIX shells and pipes
Testability

**At least 40% of the cost of well-engineered system is due to testing**
*(Bass et. al., 2003)*

What about cohesion and coupling?
Some Business Qualities

**Time-to-market**

*Reuse component and use commercial-off-the-self (COTS) products*

**Cost-and-benefits**

*Use technology that the organization knows*
Architecture Styles / Patterns

Example of styles and patterns

- Client-Server
- Layering
- Pipes-and-filters
- Service-oriented
- Model-View-Control (MVC)
- Repository
- Peer-to-Peer

Discussed today
1. Client-Server

The clients need to be aware of the server.

Clients initiate communication
1. Client-Server

The clients need to be aware of the server.

Clients initiate communication
1. Client-Server

**Two-Tier, Thin-client**
- Client
  - Presentation layer
- Server
  - Business Layer
  - Data management

- Heavy load on server
- Significant network traffic

**Two-Tier, Fat-client**
- Client
  - Presentation layer
- Server
  - Business Layer
  - Data management

**Three-Tier**
- Client
  - Presentation layer
- Middle-ware
  - Business Layer
- Server
  - Data management

- + Distribute workload on clients
- - System management problem, update software on clients

- + Map each layer on separate hardware
- + Possibility for load-balancing
2. Layers

In a “pure” layered model, only the immediate below layer can be accessed.

Layer bridging – can access lower than the closest one.
2. Layers

**Pros**
- Easy reuse of layers
- Support for standardization
- Dependencies are kept local - modification local to a layer
- Supports incremental development and testing

**Cons**
- Could give performance penalties
- Layer bridging looses modularity
3. Pipes and Filters

Example: UNIX Shell

```
ls -R | grep "html$" | sort
```

Example: A Compiler

lexer → parser → semantic analysis → Intermediate Code Generation → Optimization → Code Generation
Case: SOA and Amazon

**Before 2001…**

Two-tier architecture

- Customer web clients
- Webserver + DBMS

**Problems**

- Scaling the DBMS
- Too complex software to maintain and develop

**After 2001…**

- Customer web clients
- Web servers
- Services
- Partner Companies (1 million)

**Key Success Factors**

- Data encapsulated with business logic.
- No data sharing between services
- Independent dev teams for each service
- Developers have operational responsibility (you build, you run)

CTO Werner Vogels blog
www.allthingsdistributed.com
**Write from the point of view of the readers...**

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Use of the architect document</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements engineers</td>
<td>Negotiate and make tradeoffs among requirements</td>
</tr>
<tr>
<td>Architects/Designers</td>
<td>Resolve quality issues (e.g. performance, maintainability etc.)</td>
</tr>
<tr>
<td>Architects/Designers, Designers</td>
<td>A tool to structure and analyze the system</td>
</tr>
<tr>
<td>Developers</td>
<td>Design modules according to interfaces</td>
</tr>
<tr>
<td>Testers and Integrators</td>
<td>Get better understanding of the general product</td>
</tr>
<tr>
<td>Managers</td>
<td>Specify black-box behavior for system testing</td>
</tr>
<tr>
<td>New software engineers, Quality</td>
<td>Create teams that can work in parallel with e.g. different modules. Plan and allocate resources.</td>
</tr>
<tr>
<td>assurance team</td>
<td>To get a quick view of what the system is doing</td>
</tr>
<tr>
<td></td>
<td>Make sure that implementation corresponds to architecture.</td>
</tr>
</tbody>
</table>
When to document?

- Initial design
- Design iterations
- After implementation (consistent with code?)

Time

- Requirements
- Design
- Implementation
What to document?

**System Overview**

A brief description / overview of the system, stating
- Who the users are
- The main requirements / constraints including important quality factors
- Any important background information

Give the reader a mental model of the system!
What to document?

- System Overview
- Different structural Views
- Mapping between views
- Behavior
- Rationale

**Implementation view**

**Execution view**

**Deployment view**

Cryptographic Module

Server

Client

On different machines?
What to document?

For each view

- Give an overview of the view
- Create a view diagram
- Describe each element
- Describe each relation between elements
- Describe interfaces

System Overview

Different structural Views

Mapping between views

Behavior

Rationale
What to document?

**Mapping between views**

- **Describe how the view relates (all views describe the same system – what is common)**
- **Central to decrease confusion and increase understanding**
- **Especially important if not one-to-one mapping**
What to document?

**Behavior**

- Views state structural information
- Behavior for elements or between elements can be described using e.g., texts, sequence diagrams and state machines.

![Diagram showing an Order, orderLine, booking, and customer: Customer.](image)
What to document?

Why the architecture is the way it is
- Rationales for views, interfaces, etc.
- Architecture implication due to certain requirements
- Expected effect when changing requirements or adding new ones
- Constraints for the developer when implementing the solution
- Design alternatives that were rejected and the rational for doing so.

In general
- Why a decision was made
- What the implication is to change it
Linköping University
expanding reality
www.liu.se