Architecture-Centric Software Development

Claudio Riva
Principal Scientist

Nokia Research Center
Software Engineering

• The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. IEEE Definition

• An engineering discipline that is concerned with all aspects of software production:
  • Software requirements
  • Software design
  • Software construction
  • Software testing
  • Software maintenance

• The term became popular after the 1968 NATO conference
Software Failures

  - Replacement for the antiquate FBI’s IT infrastructure
  - Cost: $389 million - $500 million
  - Never delivered. Continuous changing requirements. Lost: $100-$200 million
  - Never delivered. $2.6 billion spent. Cumulative cost for U.S. airlines: $50 billion
  - Never delivered. Cost: $600 million.
- Arianne 5 (1996)
  - Arithmetic overflow due to 64-bit to 16-bit conversion

Only in U.S. yearly cost for failed or troubled software is about $60-$70 billion
Why Software Fails?

- Unclear and changing requirements
- Slipping deadlines
- Bad estimate of resources
- Misunderstood business needs
- Quality
- System goes out of date when delivered
- Cost and schedule overrun
- Use of immature technology
- Inability to handle the project complexity
- Poor project management
- Pressure because of time-to-market
- Poor communication among customers, developers and users

Software failures are caused by a combination of technical, management, business and human problems.
Brief History of SE: No Silver Bullet

- **1950’s**: Hardware engineering
- **1960’s**: Software crafting
- **1970’s**: Code and fix
- **1980’s**: Formal methods
- **1990’s**: Agile methods
- **2000’s**: Model-driven Development
- **2010’s**: Software Services

- **1950’s**: Structured design
- **1960’s**: Information hiding
- **1970’s**: Modularity
- **1980’s**: ADT
- **1990’s**: UML
- **2000’s**: Agile Manifesto

- **1950’s**: Waterfall
- **1960’s**: CASE tools
- **1970’s**: Object-orientation
- **1980’s**: Software architectures
- **1990’s**: Software patterns
- **2000’s**: COTS

- **1950’s**: Maturity models
- **1960’s**: Domain Engineering
- **1970’s**: Design patterns
- **1980’s**: Software frameworks
- **1990’s**: Open Source
- **2000’s**: System Engineering

- **1950’s**: Agile methods
- **1960’s**: Domain Engineering
- **1970’s**: Product families
- **1980’s**: System Engineering
- **1990’s**: Agile
- **2000’s**: System Engineering

- **1950’s**: Agile
- **1960’s**: Software architectures
- **1970’s**: Model-driven Development
- **1980’s**: Agile
- **1990’s**: Software frameworks
- **2000’s**: Agile

- **1950’s**: Open Source
- **1960’s**: Design
- **1970’s**: Patterns
- **1980’s**: Software
- **1990’s**: Agile
- **2000’s**: Agility

- **1950’s**: UML
- **1960’s**: Design
- **1970’s**: Patterns
- **1980’s**: Frameworks
- **1990’s**: Agility
- **2000’s**: Global integration

- **1950’s**: Development
- **1960’s**: Crafting
- **1970’s**: Formality
- **1980’s**: Productivity
- **1990’s**: Scalability
- **2000’s**: Agility
Essential Difficulties of Software

• Complexity
  • Huge number of states
  • Scaling-up software happens by adding elements and states
  • Nonlinear increasing with size
  • Human-added complexity (e.g. Y2K)

• Changeability
  • Continuously evolving to stay competitive
  • Software is malleable (easy to change)
  • Software is expected to be easy to change

• Invisibility
  • Software is invisible
  • Several diagrams or views can be created
  • Software programs are rich in details
Forces in software development

• Functionality
• Compatibility
• Realiability/Availability
• Security
• Cost/Schedule
• Performance
• Scalability
• Technology
• Fault tolerance
The importance of Software Architecture

- It can give clarity to the software assets
- It can guide the development
- It can give responsibility to people

- Successful projects have clear and robust architecture

- Examples from OSS:
  - Apache: pipe and filter architecture (functionality is added through modules)
  - Wordpress: pipe and filter (plugins)
  - Gallery2: pluggable architecture
What is a Software Architecture?

- Software architectures is what software architects do

Neo: Who are you?
The Architect: I am the Architect. I created the Matrix.

- Architects are the most experienced developers in the team
  - Who are very aware of what’s going on in the project
  - Who tackle important issues before they become a serious problem
  - Who have broad view on the design from the code to the requirements
  - Who can communicate with programmers, testers, designers, management
  - Who can facilitate communication and minimize their need

- Architecture: shared understanding of the important matters of the system design
Software Architecture and Design Decisions

• Documents the most **important design decisions** about the system
  • The architects consider them important
  • Everyone should be aware of them and respect them

• A software architecture is a **shared mental model** of:
  • Terminology and concepts
  • Structure of the system from different viewpoints (functional, physical, run-time, organizational)
  • Development practice enabling evolution without breaking integrity
  • Responsibility to maintain the system healthy

• Successful projects usually have a **clear and robust** software architecture
Description of a Software Architecture

“The description of the software architecture should communicate the essential decisions that have been taken in the design of the software system”

• Categories of design decisions:
  • Concepts: the way we think of a system, its architectural style
  • Architecturally significant requirements: the major concerns that should be addressed by a proper software architecture
  • Structure: components and their relationships
  • Texture: architecturally significant design decisions at the implementation level, such as design patterns, coding conventions and policies.

• Multiple architectural views are typically used to communicate the different aspects of an architecture (like the “4+1” model).
Software Architecture as a Bridge

It allows the designers to reason about the ability of the system to satisfy certain requirements.

The architecture exposes certain properties while hiding others.
Where is the Model?

```java
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

Describe/draw a model that captures the design of this Java code (5 min)

The UML class diagram used during design

![UML Diagram](image)
Some issues with UML

- The UML language is not precise
- Some of the UML concepts have no direct mappings to the implementation
- UML lacks some concepts that are necessary to model the source code
Abstraction Levels

Requirements

Architecture

Views

Views

Model

Design and Language Concepts

Code

Architectural Views

Views

Abstraction

Implementation

Class

Method

Attribute

Function

has_method

has_class

inherit

has_attribute

access

invocation
Multiple Architectural Views

- The 4+1 view model (P. Kruchten)
  - Logical view
  - Process view
  - Development view
  - Physical view
- The Siemens views (C. Hofmeister et al.)
  - Conceptual Architecture view
  - Module Architecture view
  - Execution Architecture view
  - Code Architecture view
- The DSA views (P. Clements et al.)
  - Module viewtype: decomposition, uses, generalization, layered
  - C&C viewtype: pipe & filter, Shared-date, publish-subscribe, client-server, peer-to-peer, communicating processes
  - Allocation viewtype: deployment, implementation, work assignment
My Architectural Views

- **Component view**: describing the major components, their interfaces and their logical relationships.
- **Task/Process view**: describing the task allocation of the architectural entities and showing the inter task communications.
- **Development view**: describing the organisation of the source code files and their relationships (for example, include dependencies).
- **Deployment view**: describing the physical location of components in the processing units.
- **Feature view**: describing the run-time implementation of a feature at a high level of abstraction.
- **Organizational view**: describing the organization of the development activities (projects, programs, sites).
ADLs and MDA

- Architecture Description Languages (ADLs)
  - Standard notations for describing architectures (components, connections, properties and behavior)
  - No universal agreement on ADL (not even UML)
  - Lack of tool support and industrial acceptance

- Model Driven Architecture (MDA)
  - “The code is the model”
  - Functionality is defined using the Platform-Independent Model (PIM)
  - Platform specific issues are modeled in the Platform Definition Model (PDM)
  - Executable code is generated by transforming the PIM given a PDM (ex. CORBA, .NET, Web, etc.)
  - Models are created using UML, XML, MOF and other standards
  - MDA heavily relies on tools (vendor lock-in) and industry standards
  - Standardized by OMG and supported by IT industries
What are the Architecturally Significant Requirements?

- ASRs are requirements that
  - play an important role in the execution of the system
  - have a major impact in the design of the system and its components
- ASRs need to be addressed by a properly designed architecture
  - If ignored, the system may suffer major deficiencies
- ASRs represent the rationale behind certain design decisions
  - Architects must document them

- Examples: mobile phones have strict real-time requirements on the communication with the base stations
What are the Architectural Concepts?

- Every software system is built according to its own architectural style.
- **Architectural style:** the types of concepts and rules that are permitted for the construction of the system.
- **Architectural concepts:** the basic building blocks with well defined behavior, properties and relationships.
  - Invented by the architects.
  - Properly designed to facilitate the implementation of the requirements.
  - Expensive to change!!
What is the Texture?

- Textures are **recurring microstructures** used to build software components
  - May have a significant impact on the system
  - May be hard to modify later during development and maintenance
- Possible textures:
  - Policies
  - Coding and naming conventions
  - Design patterns
  - Standards
What is the Structure?

• The structure documents how the software is partitioned into components or units at different stages of the software life-cycle (run-time, design-time, build-time, etc...)

• Software system can be partitioned in:
  Set of tasks and processes (at run-time)
  Set of components, packages, and subsystems (at design-time)
  Set of executables and libraries (at build-time)
  Set of code files (at implementation-time)
  Set of configuration files (at configuration-time)
Example: the Reconstructed Logical View
Desired, Concrete and Real Architectures

- **Requirements**
- **Desired Architecture**
- **Concrete Architecture**
- **Real Architecture**

Time

Code
The Concrete Architecture (the reality)

NPF Component View

Apps

Core

Other

Protocol

HW
Loosing an Architecture

• (Initial) architecture not documented
  • Decisions are made, rationale not recorded
  • Lots of details: big picture is missing.
  • Architectural knowledge is distributed among people, projects and sites
• Documentation is outdated
  • Rapidly changing requirements
  • Original developers have left
  • Maintaining documentation low priority (empowered architects are needed)
• Lack of proper tool support
  • Lack of proper formalism (like ADL)
  • Architecture description is maintained with informal notations (graphical or textual) without a systematic approach for analysis, controlled modifications, and efficient maintenance.
  • The mental models of the architecture do not reflect the reality
  • Architecture reconstruction is conducted manually
Architectural Decay, Erosion and Drift

• Symptoms of architecture decay:
  • Performance problems but no idea what to fix
  • Features are added slower and with difficulties
  • Organization issues (who owns what?)
  • Increasing resistance to change

• Architecture erosion: violations of the architecture during the incremental changes.

• Architecture drift: lack of coherence and clarity of form. The architecture is more obscure and not coherent with the intended design.
Architecture Reconstruction

• What is it?
  Architecture reconstruction concerns with the task of **recovering the past design decisions** that have been taken during the development of a software system.

• Why?
  • Quality assurance upon intake, acquisition, merger
  • The architecture is unclear and not well documented
  • Original developers have left and documentation is outdated
  • The size and complexity of the system
  • Supporting Software Evolution
    • Conducting *informed* software modifications

• What for?
  • To increase our comprehension of a software system by creating mental models of its *as-implemented* architecture.
Architecture Recovery Objectives

• To extract the concrete architecture from the implementation (as opposed to the intended architecture).
• To identify the past design decisions (reverse architecting) and the new ones originated from the evolution (arch discovery).
• To analyse the dependencies among the major artefacts of the architecture (such as components, tasks, files).
• To re-document the architecture and to communicate it to the developers.
• To develop a repeatable process applicable at each build of the system.
• To gather the information for architectural conformance checking.
Architecture Reconstruction & Conformance Checking

Stakeholders

Problem Elicitation

Concept Determination

Problem Statement

Library of viewpoints

Data flow

View definition

Actor in

Phase 1: Process design

Phase 2: View recovery

Phase 3: Result interpretation

Target Viewpoints

Presentation

Hypothetical Views

Source Viewpoints

Mapping Rules

Target Viewpoints

Source Viewpoints

Knowledge Inference

Data Gathering

Artifacts

Architectural Views

Hypothetical Views

Architecture Conformance Checking

Architecture Assessment

Re-documentation

System Experts

Stakeholders
Definitions

- **System**: a collection of components organized to accomplish a specific function or set of functions.
- **System stakeholder**: an individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system.
- **Architecture**: the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
- **View**: a representation of a whole system from the perspective of a related set of concerns.
- **Viewpoint**: a specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.
View Types and Mappings

Target Viewpoint

Linked with

Mapping rules

specifies

Hypothetical View

Derived from

Target View

Map

Source Viewpoint

specifies

Source View

specifies

specifies
Activity 2: Concept Determination

Phase 1: Process design
- Problem Elicitation
- Source Viewpoints
- Library of viewpoints
- Data flow
- View definition
- Actor in

Phase 2: View recovery
- Target Viewpoints
- Knowledge Inference
- Data Gathering
- System Experts

Phase 3: Result interpretation
- Architectural Views
- Hypothetical Views
- Hypothetical Views
- Target Viewpoints
- Architecture Conformance Checking
- Architecture Assessment
- Re-documentation

Stakeholders
- Problem Statement
- Concept Determination

Artifacts

Source Viewpoints
- Mapping Rules
- Target Viewpoints

© 2005 Nokia Tampere Uni 23.10.2007 claudio.riva@nokia.com
Phase 1 – An MSC drawn during a session

My System API

My System

My Server

Application

Client

Server

Sys1 API

Sw Bus

Bus Manager

Server

Proxy Manager

System 2 Proxy

function call

::Connect

Sys1.Open()

Sys2.Send()

return

Send_msg()

Message

Recv_Msg

Response

Response

Response

Response

Response
Phase 1 - Examples of high level concepts

• Components:
  • Subsystems, applications, servers, classes, software busses, tasks, processes, queues.

• Communication:
  • Function calls, RPCs, pipes, sockets, software busses, shared memory.

• Textures (micro-architectures and design patterns) should also be considered at this stage because they hide interaction patterns that are architecturally significant for the reconstruction.
Example of Architectural concepts
Example of Communication

Application Service Server

Register/UnRegister

SendMessageToServer

SubscribeEvents

Software Bus
Architectural Concepts

Source Code Concepts

Architecturally Relevant Concepts
Target Viewpoints

Component Viewpoint

Organizational Viewpoint
The FAMIX core model
Activity 3: Data Gathering

**Stakeholders**
- Problem Elicitation
- Concept Determination
- Problem Statement
- Library of viewpoints
- Data flow
- View definition
- Actor in

**Target Viewpoints**
- Presentation
- Knowledge Inference
- Source Viewpoints
- Hypothetical Views
- Source Viewpoints
- Mapping Rules

**Architecture Views**
- Hypothetical Views
- Architecture Conformance Checking
- Architecture Assessment
- Re-documentation

**System Experts**
- Data Gathering
- Artifacts

**Phase 1: Process design**
- Phase 2: View recovery
- Phase 3: Result interpretation
Phase 2 - Extraction strategies

1. Extracting the typical programming languages constructs from code
   • Schema already exist for typical OO languages (e.g. FAMIX)
   • There are plenty of front-ends that can analyse the source code
     • Columbus, SourceNavigator

2. Extracting domain specific information from code and other artefacts
   • The schema is defined in the concept determination
   • Code pattern detectors and ad-hoc analysers (e.g. Perl, grep)

• The two strategies are usually combined in order to extract the complete source code model of a system.
Activity 3: Extraction

- Python script with a list of regular expressions that detect code patterns:
  - Component definition
  - Message definition
  - File system structure (directories and files)
  - Function definitions
  - Identifiers of the components
  - Interface of the servers
  - Messages and events
  - Function calls

- Eg: `/^.*send_message\s*\(\s*\(\S+\)\s*,.+,.+\)/`
The unstructured RSF format

- This format is used to describe a plain graph without any hierarchy. The format is a text file with a list of tuples with the following syntax:

  `<verb> <subject> <object>`

  where `verb` can be any arc defined in the Rigi domain or the built-in verb type. `Subject` and `object` are respectively the source and destination of the arc. The nodes must have a unique name.

```
  type Applet Class
  type HelloWorld Class
  type Graphics Class
  type HelloWorld.paint(Graphics) Method
  type Graphics.drawString(String,int,int) Method
  inherit HelloWorld Applet
  has_method HelloWorld HelloWorld.paint(Graphics)
  has_method Graphics Graphics.drawString(String,int,int)
  invocation HelloWorld.paint(Graphics)
  Graphics.drawString(String,int,int)
```

The definition of a node

The definition of an arc
Activity 3: Knowledge Inference

Phase 1: Process design
- Problem Elicitation
- Concept Determination
- Library of viewpoints
- Data flow
- View definition
- Actor in

Phase 2: View recovery
- Source Viewpoints
- Target Viewpoints
- Knowledge Inference
- Presentation
- Hypothetical Views
- Target Views
- Source Viewpoints
- Mapping Rules
- Target Viewpoints
- Data Gathering

Phase 3: Result interpretation
- Architecture Views
- Hypothetical Views
- Architecture Conformance Checking
- Architecture Assessment
- Re-documentation
- Stakeholders

System Experts

Stakeholders
The goals of abstraction

• Abstraction is the process of enriching the model with domain specific information that will lead to a high-level view of the system with the following goals in mind:
  • Information hiding, encapsulation, compactness
  • Easy navigation of the model
  • Architecturally relevance
• The main goal with abstraction is to create a model that is architecturally significant for the developers:
  • Showing important dependencies
  • Hiding details
  • Showing the “big picture” of the system
• The major issues with abstraction are:
  • Selecting the correct level of abstraction for showing the data
  • Selecting the appropriate views
  • Classifying and grouping the concepts in a correct way
Rigi will take care of maintaining the correct composite arcs
Grouping the classes

- Grouping the methods and attributes of a class in a new class node
- High level dependencies are maintained by Rigi
Case 1: wrong selection of the abstraction level

- The system is based on CORBA
- The model on the left has been generated with the FAMIX schema
- All the nodes seem to be loosely coupled but...
  ...the problem is that this model shows only the low-level dependencies (e.g. function calls) that are not relevant for this case!!
Case 1: wrong selection of the abstraction level

- High level dependencies are hidden by the CORBA communication infrastructure (ORB)
- The solution has been to:
  - Restrict the analysis to the Implementation classes
  - Collapse the Impl, Stub and Skel in one single entity

The graph shows the real dependencies among the CORBA classes
Manual preparation of the map

- A table that maps source elements to target elements
  - Manually prepared with the help of the experts
  - Based on the output of linker, configuration files or other sources
- An example from the Relflexion models:

```
[ file=*pager.* mapTo=Pager ]
[ file=vm_map.* mapTo=VirtAddressMaint ]
[ file=vm_fault.c mapTo=KernelFaultHdler ]
[ dir=[un]fs mapTo=FileSystem ]
[ dir=sparc/mem.* mapTo=Memory ]
[ file=pmap.* mapTo=HardwareTrans ]
[ file=vm_pageout.c mapTo=VMPolicy ]
```
Binary Relational Algebra

- Binary relational algebra provides us a set of axioms to manipulate the relational data and to formally specify the graph transformations.

- We rely on Tarski Relational Algebra:
  - Operators: $\cup, \cap, \setminus, -1, \circ, \text{id, } R^+, R^*$
  - Important operators: lifting, transitive closure

- Usage:
  - Transform architectures (e.g. for abstraction)
  - Analyze architectures (e.g. conformance checking)

- Some advantages are (over other approaches like SQL, Prolog):
  - Formal specification of transformation rules
  - Support for the transitive closure
  - Scalable

- Approaches:
  - Relational Partition Algebra (Philips)
  - Grok (University of Waterloo)
  - Algebra + Python scripts (Nokia)
Algebraic operations (1)

R1 = \{(a,b), (a,c), (d,e)\}
R2 = \{(a,d), (c,b), (a,c)\}

• Union: \(R_1 \cup R_2 = R_1 + R_2\) = \{(a,d), (c,b), (a,b), (a,c), (d,e)\}
• Intersection: \(R_1 \cap R_2 = \{(a,c)\}\)
• Difference: \(R_1 - R_2 = \{(a,b), (d,e)\}\)
• Transposition: \(\sim R_1 = \{(b,a), (c,a), (d,e)\}\)
Algebraic Operations (3)

\[ R_1 = \{(a,d), (c,b), (a,c)\} \]
\[ R_2 = \{(a,b), (a,c), (d,e)\} \]

• Relational Composition: \( R_1 \ast R_2 = \{(a,e)\} \)

\[ \begin{array}{cc}
  a & d \\
  c & b \\
  a & c \\
\end{array} \ast
\begin{array}{cc}
  a & b \\
  a & c \\
  d & e \\
\end{array}
= \begin{array}{cc}
  a & e \\
\end{array} \]

• Relational Power: \( R^n = R \ast R \ast \ldots \ast R \) \( n \) times
Algebraic operations (4)

\[ R_1 = \{(a,c), (c,d), (a,b), (b,f), (b,e), (e,g)\} \]

- Transitive closure: \( R^+ = \bigcup_{i=1}^{\infty} \{(a,e)\} \)
- Reflexive transitive closure: \( R^* = \bigcup_{i=1}^{\infty} \bigcup_{R^i\cap R = \emptyset}^{R^i} \{(a,e), (a,a), (b,b), (c,c), (d,d), (e,e), (f,f) \} \)
Lifting (1)

\[ C = \{(A,B), (A,D), (B,C), (B,e), (C,f), (C,g), (D,h), (D,i)\} \]

\[ R = \{(e,f), (h,g), (i,h), (f,i)\} \]

- Reflexive lifting: \( R \uparrow C = C^+ \ast R \ast (\sim C)^+ = D \ast R \ast A \)
- Lifting: \( R \uparrow C = C^+ \ast R \ast (\sim C)^+ - I_R - C^+ - (\sim C)^+ = D \ast R \ast A - I - D - A \)
Activity 4: Knowledge Inference

\[
\begin{align*}
\text{implement} &= \text{compDefineRequest} + \text{compDefineEvent} \\
\text{subscrEvent} &= \text{compContainFile} \circ \text{fileSubscrEvent} \circ \text{compDefineEvent}^{-1} \\
\text{message} &= \text{compContainFile} \circ (\text{fileSendMessage} \circ \text{compDefineRequest}^{-1} + \\
&\quad \text{fileSendDynMessage} \circ \text{compDefineServerID}^{-1})
\end{align*}
\]

Define the containment relation

\[
\begin{align*}
\text{compContain} &= \text{pkgContainPkg} + \text{pkgContainComp} + \text{platformContainPkg} \\
\text{orgContain} &= \text{siteContainFactory} + \text{factoryContainPrj} + \text{prjContainCompp} \\
\text{taskContain} &= \text{taskContainComp}
\end{align*}
\]

Calculate the high-level dependencies

\[
\begin{align*}
\text{compDependency} &= \text{use} \uparrow \text{compContain} \\
\text{orgDependency} &= \text{use} \uparrow \text{orgContain} \\
\text{taskDependency} &= \text{use} \uparrow \text{taskContain}
\end{align*}
\]

Define the architectural views as hierarchical graphs

\[
\begin{align*}
V_{\text{Component}} &= G(T, \text{use} + \text{compDependency}, \text{compContain}) \\
V_{\text{Organization}} &= G(T, \text{use} + \text{orgDependency}, \text{orgContain}) \\
V_{\text{Task}} &= G(T, \text{use} + \text{taskDependency}, \text{taskContain})
\end{align*}
\]
Target Viewpoints

Component Viewpoint

Organizational Viewpoint
Presentation

Phase 1: Process design

- Problem Elicitation
- Concept Determination
- Library of viewpoints
- Data flow
- View definition
- Actor in

Phase 2: View recovery

- Target Viewpoints
- Hypothetical Views
- Source Viewpoints
- Mapping Rules
- Target Viewpoints
- Knowledge Inference
- Data Gathering
- Artifacts

Phase 3: Result interpretation

- Architectural Views
- Hypothetical Views
- Architecture Conformance Checking
- Architecture Assessment
- Re-documentation

Stakeholders

System Experts

Artifacts

60 © 2005 Nokia Tampere Uni 23.10.2007 claudio.riva@nokia.com
Presentation Requirements

• Show multiple structures
• Show relationships between structures
• Multiple levels of abstraction
  • Zoom in, zoom out
• Visual as well as textual information
  • Graph visualization
• Browsing and searching
Presentation

Geographical view

Organizational view
Hyperlinked web pages

Logical View

1. Select the logical view: w36_006 and press Go.

2. Select the query type:

3. Please select the element and press Go.

You can also browse the tree menu or the hierarchical view.

Summary of the dependencies:

The clients and the suppliers for Component:

<table>
<thead>
<tr>
<th>Component type</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>C# Configuration</td>
</tr>
<tr>
<td></td>
<td>C# Configuration</td>
</tr>
<tr>
<td></td>
<td>C# Configuration</td>
</tr>
<tr>
<td></td>
<td>Application</td>
</tr>
<tr>
<td></td>
<td>Application</td>
</tr>
</tbody>
</table>

© 2005 Nokia Tampere Uni 23.10.2007 claudio.riva@nokia.com
Activity 5: Hyperlinked Web Pages

Summary of the dependencies:
The clients and the suppliers for Energy Management Server by component type:

Clients:

<table>
<thead>
<tr>
<th>Component type</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>A Application</td>
</tr>
<tr>
<td></td>
<td>B Application</td>
</tr>
<tr>
<td></td>
<td>C Application</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>Operating System</td>
</tr>
<tr>
<td>HW_Driver</td>
<td>A HW Driver</td>
</tr>
<tr>
<td>Server</td>
<td>A Server</td>
</tr>
<tr>
<td></td>
<td>B Server</td>
</tr>
<tr>
<td></td>
<td>C Server</td>
</tr>
</tbody>
</table>

Suppliers:

<table>
<thead>
<tr>
<th>Component type</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol</td>
<td>A Protocol</td>
</tr>
<tr>
<td></td>
<td>B Protocol</td>
</tr>
<tr>
<td>HW_Driver</td>
<td>A HW Driver</td>
</tr>
<tr>
<td>Server</td>
<td>A Server</td>
</tr>
<tr>
<td></td>
<td>B Server</td>
</tr>
</tbody>
</table>

Detailed information about the dependencies of Energy Management Server:

- Dependencies by interface type for Energy Management Server
- Source files in Energy Management Server
- Source file dependencies for Energy Management Server
- Interface provided by Energy Management Server
The Portable Bookshelf showing Linux FS

http://swag.uwaterloo.ca/pbs/
UML diagrams in Rational Rose
Graph based visualization
Feature view

MSC_COMMANDS
FILE
CANVAS
DIALOGS
DATA_STRUCTURES
EXECUTION
GUI

msc_env_get
exec_initialMsc
TREE_CREATE
HASH_CREATE
guisel_initBindings
file_loadUseCase
exec_Command
ADD_PARTICIPANTS
ADD_MESSAGES
gui_initialDraw
exec_getVisibleParticipants
tree_children
draw_participants
gui_scaleCanvasesToFit
exec_getVisibleMessages
tree_children
draw_messages
exec_filterMessagesToDisplay
FILTERING_MESSAGES
tree_children
exec_getVisibleParticipants
tree_children
FILTERING_MESSAGES
exec_getVisibleParticipants
Architecture Conformance Checking

Stakeholders
- Problem Elicitation
- Concept Determination

Problem Statement

Library of viewpoints
- Data flow
- View definition
- Actor in

Phase 1: Process design
Phase 2: View recovery
Phase 3: Result interpretation

Source Views
- Source Viewpoints
- Mapping Rules
- Target Viewpoints

Target Views
- Presentation
- Knowledge Inference
- Data Gathering

Architecture Views
- Hypothetical Views
- Architecture Conformance Checking
- Architecture Assessment
- Re-documentation

Stakeholders
- System Experts

Artifacts

Phase 2: View recovery

Phase 3: Result interpretation
Activity 6: Layer Conformance Checking
Activity 6: Layer Conformance Checking

Calculate the layer violations

\[
\text{permittedDependency} = \text{permitMessage} + \text{permitSubscrEvent} + \text{permitInstatiate} + \text{permitInvocation}
\]

\[
\text{layerDependency} = \text{compDependency} \uparrow \text{contain}
\]

\[
\text{layerViolation} = \text{permittedDependency} - \text{layerDependency}
\]

Calculate the component violations

\[
\text{compViolation} = (\text{layerViolation} \downarrow \text{contain}) \cap \text{compDependency}
\]

Calculate the message violations

\[
\text{allowedMessage} = (\text{permitMessage} \downarrow \text{contain}) \cap \text{message}
\]

\[
\text{forbiddenMessage} = \text{message} - \text{allowedMessage}
\]
Architecture Reconstruction and Conformance Checking

- Architecture Reconstruction
  - Recovering the concrete architecture
  - Manual, semi-automatic or automatic recovery
- Architecture Conformance Checking
  - Comparing the **concrete** vs the **desired** architecture
  - Enforcing certain **architectural rules**

150 subsystems
1000 components
15,000 dependencies
100 Architecture Rules

3700 violations
650 layering violations

Software compiles
Products are delivered
**Integrity problems?**
Architecture Reconstruction Maturity Levels

1. **No architecture.** No control of the architecture. Architecture is only in the minds of the designers. No overall picture of the system. Gurus.

2. **Responsibilities.** Architects have been nominated. Architects’s main job is to preserve architectural integrity (not programming, not designing). Architects are empowered (Approve/reject features)

3. **Described.** It is possible to recover an architecture description from the implementation. The description is updated and reliable.

4. **Refined.** The reconstructed models are aligned with the intended design. The reconstructed models are the arch documentation.

5. **Managed.** The evolution of the architecture is automatically checked against the architectural rules. Break rules => no release (like compilation, errors => no executable)

6. **Optimized.** Architecture can be extended while preserving certain quality attributes. Possibility to configure new features.
What tools for recovery?

- Extraction:
  - grep
  - Perl, Python
  - Source Navigator, Columbus
- Abstraction:
  - Relational Algebra + Python
  - Grok (University of Waterloo)
- Presentation:
  - Graphs in Rigi
  - Rational Rose
  - Hypertext (HTML)
- Formats
  - RSF
  - GXL
Constraints in Mobile Phones Software Development

**Variance factors:**
- handset category
- telecom protocols
- User interfaces
- Operating systems
- customer/country customization

**Development process** is:
- Geographically distributed
- Concurrent
- > 1000 developers
- Integration is critical (testing)
- Feature interaction

**Highly dynamic domain:**
- Fast product development
- Continuously adding new features
- Fast changing requirements

**Hardware constraints:**
- Orders take long time
- Hard real-time OS
- Memory limitations
- Energy consumption
Software Product Families

- A product family is a collection of products that share:
  - Common requirements
  - Features
  - Architectural concepts
  - Code, typically, in the form of components

- Product family approach allows to deliver customized software systems for the various products

- Software product families are rarely created right away but they emerge in a mature domain where long-term investments become possible.

- Several approaches:
  - Copy/paste
  - Configurations
  - Component-based
  - Platform
  - Feature-based
Nokia Mobile Product Families

Series 30
- Video Calling, SMS, Picture messaging, built-in games, time, date, calculator, FM Radio

Series 40
- Video calling, Push to Talk, Email, PIM, Push email, Calendar, Camera, Music Player, Video streaming, Multimedia messaging, Rich applications, EDGE, WCDMA, CDMA, GPS, Bluetooth, Themes, Active Idle

Series 60
- Full Internet experience, Rich graphics and media, Location and Navigation, Office applications, Advanced Device Management, Secure platforms, Mobile TV

Maemo

Price

Time/Features
The Evolution of Nokia Mobile Phone Platforms

- **S30 (CUI)**
- **S40 (ISA)**
- **S60 (Symbian)**

- 1 MLOC
- 4-5 MLOC
- 10 MLOC
- 20 MLOC
- 30 MLOC

# lines (SLOC)

- # components

Continuing Change and Growth

Increasing Complexity

Effort
Product family architecture

- Overview Architecture Management
- xxx Family Architecture Management
- xxx Lead Product Architecture Management
- xxx Copy Product A Architecture Management

Reference Architecture

Family Architecture
- Platform Areas
  - Components
- Application Areas

Lead Product Architecture
- Platform Areas
  - Component
- Application Areas

Copy Product Architecture

Inherited by

Instantiated to (Application areas and features)

Copied and adapted to (Application areas and features)

Copied / configured to (platform areas)
Architecture Evolution vs. One-Shot Development

SW Quantity

Initial architecture design

One-shot development

Architecture maintenance

Architecture re-engineering

Architecture evolution

d = Architecture documentation/model

x = documented target

a = Architecture assessment

= Non release-quality SW

= Release-quality SW
Product family issues

- **Increasing bureaucracy**: new procedures, new managers, flexibility decreases
- **Slow process of change**: long approval process for changes. Changes than might break the architecture are suspended.
- **Over-designed platform**: too long time spent in designing complex framework that are difficult to instantiate.
- **Spaghetti dependency**: interfaces change without notice and long queues for change requests.
- **Feature reallocation**: features are consolidated in the platform
- **Cross-family reuse**: architectural mismatches when reusing code among different product families.
- **Introduction of new requirements**: risk that a new requirement can break the architecture
Lessons to remember

• Software architecture is about the important stuff, whatever that is for you

• Software architectures are a tool for mastering complexity of software development
  • Bridge between requirements and code
  • High-level understanding of system design (Simplification of the reality)
  • Document the high-level constraints of the system
  • Support reuses of software
  • Architecture description provides a partial blueprint for development (showing major components, dependencies, concepts and relationships)
  • Support the evolution by revealing the conflicts between desired vs concrete design
  • Architecture conformance checking
  • Simplify management of the software project
Future Trends in Mobile Devices

- Mobile services
- Evolution towards standardized platforms and APIs
- Higher level of abstraction
  - Software run-times: Flash, Python, Browser, Browser run-times
- Open Source Software
Open Source Software Development

• Open source software development
  • Quality of OSS software is increasing (huge testing base)
  • Pluggable and robust software architectures
  • Development scalable to the planetary level

• Many companies are considering how to integrate and use OSS
  • Improve time-to-market
  • Reduce costs
  • Involvement with the developers’ community

• Challenges
  • Program comprehension
    • Understanding/integrating/maintaining external code not written by us
    • Architecture re-documentation (e.g. functional and logical dependencies, interfaces)
    • Unclear modifications damage the system
  • Customization & Integration
    • Forking/merging the development (crucial to OSS)
    • Integrating all the different components and testing the integration
    • Switching to new alternatives
    • Coping with abandoned projects
  • Quality assessment
    • Architecture conformance (intended vs implemented architecture)
    • Stability, level of active involvement and aging of OSS development
The evolution of Linux main distributions

Code base: 230 MLOC
57% of C code and 17% C++
From 3.0 to 3.1 it doubled the size
Number of packages is doubling every year
Release frequency of stable Debian releases has been slowing down
The super-linear growth of Linux kernel
Conclusions

• Building software is a hard job. No silver bullet
• Software architectures play an important role:
  • for coping with complexity
  • for supporting software evolution
• Matrix-like architects do not exist.