

# Implementation Architecture

John Reekie  
University of Technology, Sydney

Contributors  
Rohan McAdam, Honeywell Inc.

Terms of Use: Creative Commons Attribution-ShareAlike 2.0  
<http://creativecommons.org/licenses/by-sa/2.0/>

---

---

---

---

---

---

---

---

## A view focussing on...

### ◆ Build-time structure

- Implementation modules
- Off-the-shelf technology
- Build-time configuration

### ◆ Detailed activity

- Call sequences



The implementation architecture focuses on the "built-time" structure of the system.

---

---

---

---

---

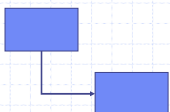
---

---

---

## Conceptual vs implementation

	<b>Conceptual Architecture</b>	<b>Implementation Architecture</b>
<i>Component</i>	Domain-level responsibilities	Implementation module
<i>Connector</i>	Flow of information	"Uses" relationship
<i>View</i>	Single view	Split views



---

---

---

---

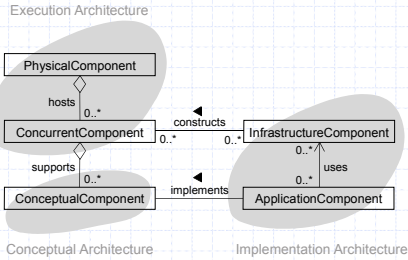
---

---

---

---

# Component relationships



This is a simplified and idealized (but useful) diagram.

---

---

---

---

---

---

---

---

# Application components

◆ Contain a chunk of application-specific functionality

- One-to-one mapping to conceptual components
- Implemented as packages, libraries, a set of files
- May be packaged as "off-the-shelf" binaries



---

---

---

---

---

---

---

---

# Infrastructure components

◆ Support the application components

- Do not appear in conceptual architecture
- Can be implemented in the same ways



---

---

---

---

---

---

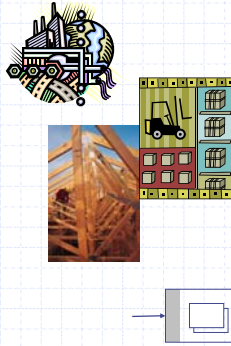
---

---

# Containers

◆ A kind of infrastructure component, that provides an "environment" for other components

- Frameworks
- Plug-in architectures
- Operating systems



---

---

---

---

---

---

---

---

# Connector styles

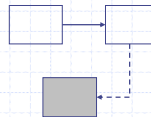
◆ A connector is a "uses" relationship

→ API call

↔ Callback

- - - Network protocol

⋯ Signal



---

---

---

---

---

---

---

---

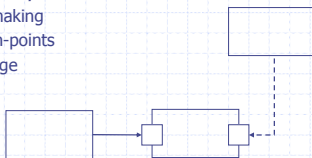
# Interfaces

◆ Interfaces clarify:

- What services are provided
- How a component is used

◆ Interfaces are relatively stable

- Facilitate decision-making
- Act as crystallization-points
- Mark points of change



---

---

---

---

---

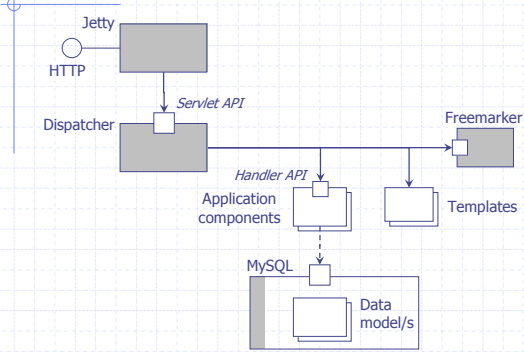
---

---

---



# Simple "three-tier" web app



---

---

---

---

---

---

---

---

# Prototyping

- ◆ In engineering, prototypes are an indispensable tool:
  - Study or verify the properties of useful structures
  - Study or verify the properties of an incomplete implementation
- ◆ The product:
  - Knowledge that helps to *predict* the behavior (performance, robustness) of the complete implementation
  - Knowledge that reduces the risk of construction

Why don't we do more prototyping in software "engineering"?!

---

---

---

---

---

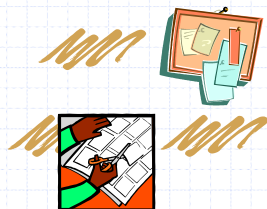
---

---

---

# Paper prototypes

- ◆ Low-cost exploration of user interaction
  - Focuses on the essential nature of the interaction
  - One person is user, the other "plays" computer
  - Gathers knowledge
  - Cheap and fast to "rebuild" interfaces



---

---

---

---

---

---

---

---

## Technical prototypes

- ◆ A “throw-away”
  - The prototype code is not part of the delivered system
  - No concern for irrelevant quality attributes (performance, robustness)
- ◆ Purely to gain knowledge (or confidence):
  - Test a new version of a commercial component
  - Verify that a set of components work together
  - Examine performance trade-offs
  - Verify that a proposed architecture is sound

---

---

---

---

---

---

---

---

## Simulations

- ◆ Often based on abstract or mathematical models of phenomena of interest
- ◆ Not built with actual system components
- ◆ Example: queuing model to analyse network traffic

---

---

---

---

---

---

---

---

## Executable prototypes

- ◆ The skeleton of the system
  - Key architectural infrastructure in place
  - Key functionality paths demonstrated
- ◆ The prototype *is* the system
  - Evolves into the delivered system as functionality is added
  - Production-quality code and configuration management at all times

---

---

---

---

---

---

---

---

## The re-usability candle

- ◆ The Optimistic end
  - "Build it and they will come"
  - Object-oriented reusability efforts are often based on this (often dubious) premise
- ◆ The Opportunistic end
  - "We came, we saw, we re-used"
  - If you need something, see if there is something that meets your needs (more or less)

---

---

---

---

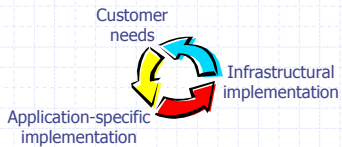
---

---

---

---

## An architectural perspective



- ◆ Re-use of larger components achieves greater "pay-off"
- ◆ Significant (useful) components evolve over time
- ◆ Components are stabilized by their interfaces (which also evolve, but much more slowly)

---

---

---

---

---

---

---

---

## Object-oriented frameworks

- ◆ One of the more successful types of object-oriented re-use
  - "Don't call us, we'll call you"
  - Example: user interface frameworks (MFC, Swing)
  - Example: simulation frameworks (Ptolemy)
- ◆ A framework can be viewed as a component
  - New behaviour implemented by sub-classing
  - Highly customisable to a particular use context

---

---

---

---

---

---

---

---

# Component frameworks

- ◆ Define a "component" as an entity with defined (provided and required) interfaces
- ◆ Provide services to the component
- ◆ Expect certain behaviour of the component

Conceptual component  
=  
Implementation component  
=  
Concurrent component

Examples: CORBA, COM

---

---

---

---

---

---

---

---

# That's all folks!

- ◆ Questions or comments?



---

---

---

---

---

---

---

---