Architectural Patterns for Complex Real-Time Systems

Bran Selic
ObjecTime Limited
bran@objectime.com
Overview

♦ Software architectures
  ■ An architectural description language

♦ Architectural design patterns
  ■ Pattern 1: Recursive control
  ■ Pattern 2: Layering

♦ Summary
Software Architectures

♦ General definition [Bass, Clements, Kazman]:

“The structures of a software system, consisting of software components of a system and their externally visible properties and relationships.”

♦ Many different architectures of a software system:
  - module architecture
  - uses (compilation) architecture
  - run-time architecture
  - process architecture
  - physical architecture
  - class (inheritance) architecture
Run-Time Architecture

The run-time organization of significant software components interacting through interfaces, those components being composed of successively smaller components and interfaces.

Key elements:
- significant = abstraction (irrelevant detail omitted)
- organization = components and their structural relationships
- interactions = inter-object (end-to-end) behavior
- interfaces = abstraction (conjunction of structure and behavior)
- architectures occur at different levels of detail
Why Architecture is Important

♦ Enables communication between stakeholders
  ■ exposes how individual requirements are handled

♦ Drives system construction
  ■ decomposition into units of responsibility and parallel development

♦ Determines a system’s capacity for evolutionary growth
The Run-Time Program Structure

♦ Main program components and their relationships

Main program components and their relationships

Ada.Text_IO

player1:Player

partner

start

ball

player2:Player

partner

start

ball

Ada.Text_IO
Basic Run-Time Structural Patterns

♦ Containment:

Composition (existence dependency)
Composition (information hiding)

♦ Peer-to-peer interaction:

♦ Layering:
with Ada.Text_IO; use Ada.Text_IO;

Procedure PingPongGame is
  task type Player is
    entry partner (player: in access Player); -- my opponent’s id
    entry start (server: in Boolean); -- start game
    entry ball; -- accept ball
  task body Player is
    opponent : access Player; -- my opponent’s task id
    begin
      accept other (p) do opponent := p; end other;
      accept start (myServe) do
        if myServe then opponent.ball end if;
      end start;
      loop
        accept ball do Put (“+”); end ball;
        opponent.ball;
      end loop;
    end Player;
  player1 : access Player := new Player;
  player2 : access Player := new Player;
  begin
    player1.partner (player2);
    player2.partner (player1);
    player1.start (True);
    player2.start (False);
  end PingPongGame;
Software architectures

- An architectural description language

Architectural design patterns

- Pattern 1: Recursive control
- Pattern 2: Layering

Summary
UML-RT: An Architecture Description Language

- A formal and executable language specified using UML
  - allows early analysis of high-level models (architectures)
- Designed for complex event-driven real-time systems
- Combines a graphical syntax with a textual syntax
  - graphics for modeling high-level (architectural) aspects
  - text for detail (e.g., Ada, C++, Java)
- Full-cycle language (analysis, design, implementation)
  - single formalism throughout development
  - facilitates iterative and incremental development
- Suitable for full automatic code generation from models
Capsules: Architectural Objects

- A special kind of active object
Capsules: Behavior

♦ Optional hierarchical state machine (signal handler with run-to-completion semantics)

transitionS1toS2:
{int x;
 x = 0;
 p2.send(s1);
 p3.send(s2);
 ...
};
Protocols: Reusable Behavior Patterns

♦ Interaction contracts between capsules
  ■ e.g., operator-assisted call

![Diagram showing interaction contracts between capsules](image)

- Call flow:
  - Caller
  - Operator
  - Callee

- Messages:
  - call
  - ack
  - number
  - transfer
  - call
  - ack
  - talk

- Additional annotations:
  - time

Diagram credits: [OBJECTIME](https://www.objectime.com)
Protocol Specifications

- A collaboration that may be required on multiple occasions and situations

Diagram:
- Operator Assisted Call
- Alice (caller)
- Bob (callee)
- Dexter (operator)
- Significant sequences
- Protocol state machine:
  - Initial
  - Connecting
  - Connected
Protocol Roles

♦ Specifies one party in a protocol

Incoming signals

<table>
<thead>
<tr>
<th>signal</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>caller</td>
</tr>
<tr>
<td>number</td>
<td>caller</td>
</tr>
<tr>
<td>ack</td>
<td>callee</td>
</tr>
</tbody>
</table>

Outgoing signals

<table>
<thead>
<tr>
<th>signal</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>callee</td>
</tr>
<tr>
<td>transfer</td>
<td>caller</td>
</tr>
<tr>
<td>ack</td>
<td>caller</td>
</tr>
</tbody>
</table>

significant sequences

OperatorRole

protocol state machine

initial

connecting

connected
Ports: Boundary Objects

- Fully isolate a capsule’s implementation from its environment (in both directions)

- Each port is typed with a single protocol role
Combining Capsules

◆ Using connectors

Connectors model communication channels
Each connector supports a single protocol
Static typing rules apply (compatible protocols)
Composition: Structural Patterns

♦ The composite is also a first-class object!
Run-time assertion: the complete internal structure of a composite is automatically created (recursively, if necessary) when the capsule is created.

\[
f1 := \text{create}(\text{FaxCall});
\]
Benefits of Run-Time Assertion

♦ **Architectural enforcement:** only explicitly prescribed architectural structures can be instantiated
  - it is not possible to bypass (corrupt) the architecture by low-level programming

♦ **Simplification:** low-level program code that dynamically creates (destroys) components and the connections between them is eliminated
  - in some systems this can be as much as 35% of all code

♦ Major net gain in productivity and reliability
End Ports

♦ Ports directly connected to the state machine

Implementation
End Port

Public End Port

capsule state machine

sender:Fax

receiver:Fax

senderCtrl : Control

receiveCtrl : Control

public End Port

sender:Fax

receiver:Fax

Control

Control

Control

initial

connecting

connected
Software architectures
- An architectural description language

Architectural design patterns
- Pattern 1: Recursive control
- Pattern 2: Layering

Summary
About Design Patterns

- A design pattern is a proven generalized solution to a generalized problem that can be used to derive a specific solution to a specific problem.
- Represent distilled reusable experience.
- Major benefits of using patterns:
  - Simplify and speed-up design
  - Reduce risk
  - Facilitate communications between designers
Software architectures
- An architectural description language

Architectural design patterns
- Pattern 1: Recursive control
- Pattern 2: Layering

Summary
Example System

- A multi-line packet switch that uses the alternating-bit protocol as its link protocol.
Alternating Bit Protocol (1)

♦ A simple one-way point-to-point packet protocol

alternating bit protocol

 Sender

 Receiver

 unpacker

 packetizer

 …etc.

 data(1)

 pktA

 ackA

 data(2)

 pktB

 ackB

 ack

 data(1)

 ack

 data(2)

 ack

 AB protocol
 Alternating Bit Protocol (2)

♦ State machine specification

Sender SM

- AcceptPktA
  - data/pktA
  - ackB/ack
  - timeout/pktB
  - WaitAckA
  - AcceptPktB
- WaitAckA
  - timeout/pktA
  - ackA/ack
  - data/pktB
  - AcceptPktB

Receiver SM

- RcvdPktA
  - pktA/data
  - ack/ackA
  - timeout/ackB
  - WaitPktB
  - RcvdPktB
  - timeout/ackA
  - pktB/data
  - ack/ackB
  - WaitPktA
  - RcvdPktB
  - timeout/ackA
  - pktB/data
  - ack/ackB
  - WaitPktA
Additional Considerations

♦ Support infrastructure

Diagram:
- AB receiver
- AB sender
- AB lines manager
- Operator interface
- DB interface
- System operator
- SWITCH
- DBase

Additional Considerations
**Control**

The set of (additional) mechanisms and actions required to bring a system into the desired operational state and to maintain it in that state in the face of various planned and unplanned disruptions.

- For software systems this includes:
  - system/component start-up and shut-down
  - failure detection/reporting/recovery
  - system administration, maintenance, and provisioning
  - (on-line) software upgrade
Retrofitting Control Behavior

- JustCreated
- Hardware Audit
- Analysing Failure
- GettingData
- ReadyToGo
- Failed
- AcceptPktA
- AcceptPktB
- WaitAckA
- WaitAckB
Control behavior is often treated in an *ad hoc* manner, since it is not part of the primary system functionality. Typically retrofitted into the framework optimized for the functional behavior, this leads to controllability and stability problems. However, in highly-dependable systems as much as 80% of the system code is dedicated to control behavior!
Some Key Observations

♦ Control predicates function
  ■ before a system can perform its primary function, it first has to reach its operational state

♦ Control behavior is often independent of functional behavior
  ■ the process by which a system reaches its operational state is often the same regardless of the specific functionality of the component
In isolation, the same control behavior appears much simpler.
Basic Design Principles

- Separate control from function
  - separate control components from functional components
  - separate control and functional interfaces
  - imbed functional behavior within control behavior

- Centralize control
  - if possible, focus control in one component
  - place control policies in the control components and control mechanisms inside the controlled components
The Basic Structural Pattern

♦ Set of components that need to be controlled as a unit

Central Controller

Controlled Component 1

Controlled Component N

Control interface

Functional (service) interface
Recursive Application

♦ Hierarchical control
  ■ scales up to arbitrary number of levels
Realization with UML-RT

♦ Composite plays role of centralized controller
Exploiting Inheritance

- Abstract control classes can capture common control behavior and structure
- Different subclasses capture function-specific behavior

```
AbstractController
  ports
  controlPort: CtrlProtocol

Sender
Receiver
```
Software architectures
- An architectural description language

Architectural design patterns
- Pattern 1: Recursive control
- **Pattern 2: Layering**

Summary
An asymmetric relationship with one-way dependencies

Layering is not the same as composition:
- individual layers are separate entities
- e.g., applications do not contain the OS
In complex systems, layering is a complex multidimensional relationship.

- e.g., 7-layer model of Open System Interconnection (OSI)
Implementation Components

- Private sub-components required to realize the functionality offered by component through its public interface.
Interface Types for Layering

- Need to differentiate two interface types:
  - **Usage interface**: implementation-independent interface through which a component provides its services (function and control)
  - **Implementation interface (service access point)**: implementation-specific interface through which a component accesses an external service

- Front-end/back-end views:
Implementation Interfaces

- Implementation interfaces are public interfaces but can be viewed as being in a different “plane” (dimension) from service interfaces.
Implementation interfaces are modeled by implementation end ports that can be connected externally to service ports of other capsules.
Software architectures

- An architectural description language

Architectural design patterns

- Pattern 1: Recursive control
- Pattern 2: Layering

Summary
Software Architectures

♦ Software architecture, including run-time architecture, is crucial:
  ■ as a vehicle for communication between stakeholders
  ■ as a design and implementation framework
  ■ as a framework for system evolution

♦ Programs obscure architecture
  ■ too much detail hides the big picture
  ■ leads to “architectural decay” over time

⇒ We require means to:
  ■ specify architectures clearly
  ■ enforce them through development and evolution
Architectural Description Languages

♦ Requirements for an ADL
  ■ High level of abstraction
  ■ Graphical syntax for ease of understanding
  ■ Describes both structure and behavior
  ■ Can be applied recursively
  ■ Generates verifiable models
    ⇒ formal
    ⇒ executable
  ■ Can be used to automatically generate implementations
    ⇒ can be combined with implementation languages for specifying detail
A good ADL removes incidental complexity allowing us to focus more closely on the difficult problem of architectural design.

The use of well-chosen design patterns can help us immensely in this task.

Many good architectural patterns already exist but new ones are emerging as computing and communications technologies evolve at a rapid pace.

More research is required on understanding how patterns interact with each other (pattern combination).
The task of engineering, including software engineering, is to build systems that will be of use to people. In this activity we should keep in mind that technology is not an end but a means—hence, we should choose technologies based on their suitability to the task at hand rather than personal preference.
Bibliography


♦ Gamma, E, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
