Introduction to design patterns for middleware

Denis Conan

CCN/CSC7321

octobre 2018

Revision : 545
Foreword

The sources of this presentation are:


   ▶ URL of the slides in French:


♦ S. Krakowiak “Chapitre 1 : Introduction à l’intergiciel” dans “Intergiciel et Construction d’Applications Réparties”, 2006,

      http://sardes.inrialpes.fr/ecole/livre/pub/Chapters/Intro/intro.html

♦ S. Krakowiak “Chapitre 2 : Patrons et canevas pour l’intergiciel” dans “Intergiciel et Construction d’Applications Réparties”, 2006,

      http://sardes.inrialpes.fr/ecole/livre/pub/Chapters/Patterns/patterns.html

E. Gamma, R. Helm, R. Johnson, J. Vlissides “Design Patterns : Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994

- Has been translated in French

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal
“Pattern-Oriented Software Architecture : Volume 1, A System of Patterns”, Wiley, 1996


Introduction to design patterns for middleware

Outline

1 Distributed system organisation with a middleware ............................................... 5
2 Design patterns ........................................................................................................ 6
3 Patterns for distributed interaction ........................................................................... 16
4 Patterns for composition ........................................................................................... 31
5 Patterns for coordination ........................................................................................... 41
1 Distributed system organisation with a middleware
2 Design patterns

2.1 Objectives of the pattern orientation .............................................................. 7
2.2 Some design pattern examples for middleware .................................................. 8
2.3 Definition of design patterns ............................................................................ 13
2.4 Writing patterns ............................................................................................... 14
2.5 Classifying patterns ......................................................................................... 15
2.1 Objectives of the pattern orientation

Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.\(^a\)

- Present the design principles of middleware architecture in a systematic way
  - Identify the main design and implementation problems
  - Exhibit the main design solutions relevant to middleware construction
  - Illustrate the patterns in frameworks in the teaching unit

- Well known software design patterns:
  - Factory
  - Singleton
  - Iterator

2.2 Some design pattern examples for middleware

2.2.1 Example 1: A client/server middleware .................................................. 9
2.2.2 Example 2: Integration of legacy applications .......................................... 10
2.2.3 Example 3: Adaptation to client resources .............................................. 11
2.2.4 Example 4: Monitoring and control of networked equipments .................. 12
### 2.2.1 Example 1: A client/server middleware

![Diagram of a client/server middleware]

- **IDL**
- **Pre-compiler**
  - **Client**
    - **Client stub (RPC)**
    - **stub (CORBA)**
    - **proxy (DCOM)**
  - **Session**
  - **Name Server**
    - **server stub (RPC)**
    - **skeleton (CORBA)**
    - **stub (DCOM)**
  - **Server**
    - **Binding object**
    - **Binding Factory**
2.2.2 Example 2: Integration of legacy applications

Diagram:

- Legacy application
- Legacy application
- New component
-Wrapper
- Proprietary interface
- Standard interface
- Inter-applications "exchange bus"
- New component
- Wrapper
- Legacy application
2.2.3 Example 3: Adaptation to client resources
2.2.4 Example 4: Monitoring and control of networked equipments

- **Physical organisation**

- **Logical organisation**

  ![Diagram showing physical and logical organisation of networked equipments]

  - Message bus
2.3 Definition of design patterns

- Definition (not limited to program design)
  - A set of design rules (element definitions, element composition principles, rules of usage) that allow the designer to answer a class of specific needs in a specific environment

- Properties
  - Elaborated from the experience acquired: Class of problems, capture of the solution elements common to those problems
  - Defines design principles, not specific to the implementation
  - Provides an aid to documentation: Common terminology, even formal description ("pattern language")
2.4 Writing patterns

- **Name**: Higher abstraction which conveys the essence of the pattern succinctly
- **Intent**: Short statement stating what the pattern does, its rationale, and the particular design issue or problem addressed
- **Motivation and context**: Scenario illustrating the class of problems addressed; should be as generic as possible
- **Problem**: Requirements, desirable properties of the solution; constraints of the environment
- **Solution**
  - **Structure**: Static aspects, *i.e.* components, relationships; may be depicted in a classes/components diagram
  - **Interactions**: Dynamic aspects, *i.e.* run-time behaviour, life-cycle; may be depicted in a communications/sequence/timing diagram
- **Also known as & related patterns**: Other well-known names & closely related patterns
2.5 Classifying patterns

- Architectural: Large scale, structural organisation, subsystems and relationships between them
- Design: Small scale, commonly recurring structure within a particular context
- Idioms: Language specific, how to implement a particular aspect in a given language
- And many more: Software process, requirement elicitation, analysis, etc.
3 Patterns for distributed interaction

3.1 Asynchronous call, synchronous call, buffered message ........................................ 17
3.2 Call-back and Inversion of control ........................................................................... 18
3.3 Reflection : Observe and act on its own state and behaviour .................................. 19
3.4 Factory : Entity creation ......................................................................................... 21
3.5 Proxy : Representative for remote access ............................................................... 23
3.6 Wrapper or Adapter : Interface transformation ......................................................... 25
3.7 Interceptor : Adaptable service provision ............................................................... 27
3.8 Similarities and differences between the previous patterns ................................. 29
3.1 Asynchronous call, synchronous call, buffered message

Asynchronous event (push)  Synchronous call  Buffered messages (pull)
3.2 Call-back and Inversion of control

**Synchronous call with callback**
A callback is first registered and later called asynchronously.

**Inversion of control**
The control flow is no more under the responsibility of the application but controlled by the framework.

The service request for A is triggered from the outside through B, which controls A.
3.3 Reflection: Observe and act on its own state and behaviour

- **Context**: Support different types of variations/adaptations of an application
- **Problem**: Particular variations must be hidden to the client
- **Solution**
  - Make the system self-aware
    - Select aspects of its structure and behaviour accessible for adaptation
      - Objectify/reify information about properties and variant aspects of the application’s structure, behaviour, and state into a set of meta-objects
  - Split the architecture into two major parts
    - Meta-level: Self-representation of the system in meta-objects
      - Type structures, algorithms, or even function call mechanisms
    - Base level: Application logic
      - Uses the meta-objects to remain independent of those aspects that change
  - An interface is specified for manipulating the meta-objects
    - Meta-Object Protocol responsible for performing changes
Architecture principle

[Diagram of distributed interaction]

Introduction to design patterns for middleware
3.4 Factory : Entity creation

■ Context : Applications organised as a set of distributed entities

■ Problem
  ♦ Dynamically create multiple instances of an entity type
  ♦ Desirable properties
    ▶ Instances should be parameterised
    ▶ Evolution should be easy, i.e. no hard-coded decisions
  ♦ Constraints : Distributed environment, i.e. no single address space

■ Solution
  ♦ Abstract factory : Defines a generic interface and organisation for creating entities; the actual creation is deferred to concrete factories that actually implement the creation methods
  ♦ A further degree of flexibility is achieved by using Factory Factory, that is the creation mechanism itself is parameterised
3.4.1 Sequence diagram of Factory

- **c:** Client
  - Request for creation
  - Request for removal

- **s:** Servant
  - Create
  - Return servant’s reference

- **f:** Factory
  - Create
  - Remove (optional)

- **ff:** FactoryFactory
  - Create (optional)
3.5 Proxy : Representative for remote access

- Context : A client needs access to the services by some entity (the “servant”)
- Problem
  - Define an access mechanism that does not involve
    - Hard-coding the location of the servant into the client code
    - Deep knowledge of the communication protocols by the client
  - Desirable properties
    - Access should be efficient at run-time and secure
    - Programming should be simple : No difference between local and remote access
  - Constraints : Distributed environment (no single address space)
- Solutions
  - Use a local representative of the server on the client side that isolates the client from the communication system and the servant
  - Keep the same interface for the representative as for the servant
  - Define a uniform proxy structure to facilitate automatic generation
3.5.1 Sequence diagram of Proxy

```
<table>
<thead>
<tr>
<th>c:Client</th>
<th>p:Proxy</th>
<th>s:Servant</th>
</tr>
</thead>
<tbody>
<tr>
<td>service request</td>
<td>pre-processing e.g., marshalling</td>
<td>post-processing e.g., unmarshalling</td>
</tr>
<tr>
<td>service request</td>
<td>result</td>
<td>result</td>
</tr>
<tr>
<td>Interface I</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
3.6 Wrapper or Adapter: Interface transformation

- Context: Clients requesting services; servers providing services; services defined by interfaces
- Problem
  - Reuse an existing server by modifying either its interface or some of its functions in order to satisfy the needs of a client (or class of clients)
  - Desirable properties: Should be run-time efficient; should be adaptable because the needs may change and may not be anticipated; should be itself reusable (generic)
- Solutions
  - The wrapper screens the server by intercepting method calls to its interface
  - Each call is prefixed by a prologue and followed by an epilogue in the wrapper
  - The parameters and results may need to be converted
3.6.1 Sequence diagram of Wrapper/Adapter
3.7 Interceptor: Adaptable service provision

- Context: Service provision (in a general setting)
  - Client-server, peer-to-peer, high-level to low-level
  - May be uni- or bi-directional, synchronous or asynchronous

- Problem
  - Transform the service (adding new treatments), by different means
    - Interposing a new layer of processing (like wrapper)
    - Changing the destination (may be conditional)
  - Constraints: Services may be added/removed dynamically

- Solutions
  - Create interposition entities (statically or dynamically). These entities
    - Intercept calls (and/or return statements) and insert specific processing, that may be based on content analysis
    - May redirect call to a different target
    - May use call-backs
3.7.1 Sequence diagram of Interceptor
3.8 Similarities and differences between the previous patterns

- **Wrapper Vs. Proxy**
  - Wrapper and Proxy have a similar structure
    - Proxy preserves the interfaces
    - Vs. Wrapper transforms the interface
  - Proxy often (not always) involves remote access
    - Vs. Wrapper is usually on-site

- **Wrapper Vs. Interceptor**
  - Wrapper and Interceptor have a similar function which is behavioural reflection
    - Wrapper transforms the interface
    - Vs. Interceptor transforms the functionality (may completely screen servant)

- **Reflection Vs. Interceptor**
  - Interceptor provides a means to implement reflective mechanisms
    - Not the only way to implement reflection (others = language, byte code transformation, etc.)
Interceptor exposes only part of the state of the base level

Reflection can define a type of interception mechanism in the form of a meta-object protocol
4 Patterns for composition

4.1 Principle of de/composition in distribution.......................... 32
4.2 Contract : Qualified required/offered interfaces ...................... 33
4.3 Layer or Abstract machine or Protocol stack : Vertical decomposition ........ 34
4.4 Multi-tier architecture : Horizontal decomposition ...................... 35
4.5 Component/Container : Contract + Factory + Interceptor + extra-functionalities 38
4.6 Composite with sharing : Component + Vertical decomposition + Sharing ...... 39
4.1 Principle of de/composition in distribution

■ Objective

♦ Ease the design
  ▶ Show the design approach through the means of the structure
  ▶ Show off the interfaces and the dependencies

♦ Ease the evolution
  ▶ Apply the encapsulation principle
  ▶ Standardise the exchanges

■ Examples

♦ Multi-level structure
  ▶ “Vertical” decomposition : e.g., Layer
    Vs. “horizontal” decomposition : e.g. Multi-tier
    Vs. both of them : e.g. Middle-tier/Component

♦ Leverage the concept of Contract
  ▶ From “simple” interfaces to
    Offered/server, required/client, and internal and external interfaces
§ Four levels of contract

1. Syntactic contract: Types of operations, verified statically
2. Behavioural contract: Dynamic behaviour (semantics) of operations, assertion-based
3. Synchronisation contract: Interactions between operations, synchronisation
4. Quality of service contract: extra-functional aspects such as performance, availability, security
4.3 Layer or Abstract machine or Protocol stack: Vertical decomposition

- Context: Complex “local” system design
- Problem: Define different levels of abstraction/refinement
- Solution: Vertical decomposition with levels, and upper and lower interfaces
4.4 Multi-tier architecture: Horizontal decomposition

- **Context**: Complex distributed system; incremental upgrade
- **Problem**: Evolution of the client and the server sides, load-balancing, scalability
- **Solution**: Horizontal decomposition into *tiers*, separation of system functionalities

![Diagram of multi-tier architecture](image)

(a) Application management
(b) User interface
(c) Data management
4.4.1 Focus on presentation tier : The MVC pattern

- **Context**: Management of the client view or user interface
- **Problem**: Confusion in the roles of objects prevents evolution.
- **Solution**: Separate the data (Model), the HMI on screen (View) and the control logic (Controller) which is the glue between the two
- **Proposed in 1978-79 by Trygve Reenskaug et al. from XEROX PARC for the Smalltalk language**
4.4.2 MVC pattern vs 3-tier architecture

- **MVC pattern**
  - Focus on the presentation layer to improve code evolutivity
  - Triangular architecture: The view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.

- **vs 3-tier architecture style**
  - Focus on the distribution of the architecture to favor scalability
  - Linear architecture: The presentation tier never communicates directly with the data tier. Communication goes through the middle tier.
4.5 Component/Container: Contract + Factory + Interceptor + extra-functionalities

- **Context**: Distributed application accessing extra-functional services
- **Problem**: Control life-cycle; separate business(extra-functional) parts
- **Solution**:
  - ♦ Contract to make explicit server and client interfaces
  - ♦ Container that implement Factory + Interceptor to manage extra-functional services

**Diagram**:

```
Client Component
Access to extra-functional services: transaction, security, etc.

Server interface
service request

Interception
life-cycle

Configuration

Component

Interceptor

Container/Membrane
```

**Legend**:
- **Client**: Access to extra-functional services
- **Server interface**: Service request
- **Interceptor**: Interception
- **Configuration**: Configuration
- **Component**: Component
- **Container/Membrane**: Container/Membrane
- **Interception**: Interception
- **life-cycle**: Life cycle
- **Access to extra-functional services**: Transaction, security, etc.
4.6 Composite with sharing: Component + Vertical decomposition + Sharing

- **Context**
  - Part-whole hierarchies of components

- **Problem**
  - Make the client simple
    - Ignore the difference between composite entities and individual components
  - A component can have more than one parent
  - Make it easier to add new kinds of components
  - Make the design overly general

- **Solution**
  - Abstract component entity which represents both a primitive or a composite
  - Control the content of composite components
  - Extend the reference/naming system to explicitly express sharing
4.6.1 Example of the Fractal Component Model

5 Patterns for coordination

5.1 Naming : White pages service ................................................................. 42
5.2 Trading : Yellow pages service ............................................................. 43
5.3 Publish/subscribe or Observer or Event channel : Change-propagation mechanism44
5.4 Pipes and filters : Structure for processing streams of data ....................... 46
5.1 Naming : White pages service

■ Context : clients and servers distributed over the network
■ Problem
  ♦ Obtain a (distributed) reference to an entity
  ♦ Only the logical name is known by the client
■ Solution
  ♦ The server registers its reference under a logical name to a name server
  ♦ The name server has a “well-known” reference
  ♦ The client retrieves the server’s reference by providing the logical name
  ♦ Logical names are organised as a hierarchy

Client → service request → Server

Client resolve() Server register()

NamingContext

NameServerA

NamingContext

NameServerB
5.2 Trading : Yellow pages service

- Context: clients and servers distributed over the network
- Problem
  - Obtain a (distributed) reference to an entity
  - Only a property characterising the server is known by the client: Service name...
- Solution
  - The client specifies its requests by providing properties of the required service
  - The trader answers by giving a set of server's references matching the client's query
5.3 Publish/subscribe or Observer or Event channel:
Change-propagation mechanism

- **Context**
  - Keep the state of cooperating components synchronised

- **Problem**
  - Be notified about state changes in a particular entity
  - Number and identities of dependent entities not known *a priori*
  - Explicit polling not feasible or not efficient
  - Notifiers and notified entities not tightly coupled

- **Solution**
  - Notifier also called publisher or subject: Maintains a registry of subscribers
  - Notified entities also called subscribers or observers: Subscribe to notification
  - Push model (publisher sends all changes)
    - Vs. pull model (publisher sends nature of data change and subscriber gets retrieves data)
5.3.1 Example of OMG CORBA Event channel
5.4 Pipes and filters: Structure for processing streams of data

- **Context**: Distributed application processing data streams
- **Problem**
  - Flexibility by reordering/recombining processing steps
  - Small processing steps are easier to reuse in a different setting
  - Non-adjacent steps do not share information
- **Solution**
  - Each processing step is encapsulated in a filter component
  - Data is passed through pipes between adjacent filters
  - Filters are the processing units of the pipeline
    - Consume data incrementally to achieve low latency and enable parallelism
  - Push mode Vs. pull mode Vs. active mode (pull + push)