Patterns of Enterprise Software Architecture

Roberto García
rgarcia@diei.udl.cat

Escola Politècnica Superior (EPS) - Room 3.15

Universitat de Lleida

Contents

Introduction

- **Materials** on this topic: (folder "4 - Software Architecture" in the "Resources" section of the Virtual Campus)
  - **Patterns of Enterprise Software Architecture.zip** (zipped HTML)
    - This presentation.
  - **Patterns of Enterprise Software Architecture.pdf** (PDF)
    - PDF version of this presentation.
  - **Inventory of Patterns of Enterprise Application Architecture.zip** (zipped HTML)
    - Catalog of patterns with code examples.

Introduction

- **Objective: structuring** business applications.
  - Especially those operating through **several layers** of processing.
- **Architecture:** main application parts and how they connect.
- **Architectural patterns:** important decisions on those parts.
- **Design patterns:** help realize that architecture.

Sample business application

- **Online Store (B2C,** business to consumer)
- Features:
  - **Many users.**
  - **Simple business logic:** to capture orders, simple calculations of prices and shipments, shipment notification,...
  - **Generic web interface.**
  - **Database:** orders and inventory.

Sample business application

- **Processing house renting**
- Features:
  - **Few users.**
  - **Complex business logic:** Estimated monthly bills, overdrafts and arrears management, validation of contract data and credit history, etc.
  - **Complex user interface.**
  - **Database:** complex and integrated with external packages (asset valuation, pricing, credit checking...

Sample business application

- **Small company expenses management**
- Features:
  - **Few users.**
  - **Simple business logic.**
  - **Scalable:** future growth.
  - Simple but **integrated** with other applications with complex architecture.

Patterns

- **Pattern:** describes a **common problem** and **guidelines** to solve it.
  - Practical foundations.
  - Same pattern but different specific solutions.

Contents

Layered Architecture

- Common approach to **decompose** a complex problem.
- **Top** layer uses the **services** defined by the **bottom one.**
- **Lower** layer **is unaware** of the top layer.
- Usually, bottom layer **hides** layers below to upper levels.

Layered Architecture

- **Benefits:**
  - Only need to **know 1st layer below, independently** from **others below.**
  - Replace whole layer with **alternate** implementations.
  - **Minimize dependencies** between layers.
  - Layer **reused** by all layers above.

Layered Architecture

- **Disadvantages:**
  - Risk of **cascade changes.**
    - Example: new field in user interface represents change in DB and all intermediate layers.
  - Extra layers may reduce performance.
- **Difficulty:** deciding layers and functionality per layer.

3-Layer Architecture

- **Presentation**: the logic of **user-application interaction.**
  - Ex .: command line, menus, rich client, HTML ...
- **Data Sources:** communication with systems that **provide data.**
  - Ex .: transactions monitor, legacy systems, ... **databases.**
- **Domain Logic:** application-specific **functionality.**
  - Ex.: validating tickets, generating invoices, computing problems,...

3-Layer Architecture

- **Note:** what happens when the user is not a person (web services, batch processes,...)?
  - **Distinction presentation-data** becoming diffuse (both external connections).
  - Distinction (from the point of view of the application):
    - **Interface** when service **provided** to others.
    - **Source** when others services are **consumed.**

Where are layers executed?

- Layers separation useful even only on one computer.
- **Decision between client and server:**
  - **Everything** on the **server:** easy upgrade and maintenance.
    - For instance HTML presentation.
  - Part on the **client:** better responsiveness and offline operation.
    - For instance an installable client application (Applet, App,...).

Contents

Transaction Script

- Simplest domain logic pattern: ***Transaction Script*** [1]
  - Procedure receives input **parameters** from Presentation.
  - **Processes** parameters.
  - **Stores/Retrieves** information from the database.
  - **Invokes** operations on other systems.
  - Finally, **responds** to Presentation with data.

Transaction Script

- **Example** online store: ***Transaction Scripts*** to add product to cart, checkout, check order status,...
- **Advantages:**
  - Simple procedural model.
  - Works well with simple data.
  - Transaction limits are obvious.
- **Disadvantages:** too simple for complex domain logics.
  - Complex domains require objects, the ***Domain Model*** pattern

Domain Model

- Pattern based on the use of **objects.**
- **Advantage:**
  - Suitable for **complex domains**.
- **Disadvantage:**
  - The more complex the domain, more difficult to **map to relational databases**.

Table Module

- ***Table Module*** designed to operate with pattern ***Record Set.***
  - ***Record Set*** is the result of a DB query.
- Compromise between ***Transaction Script*** and ***Domain Model.***
  - Domain structured with tables (more structured than ***Transaction Script).***
  - Missing inheritance and other OO patterns (less structured than ***Domain Model)***.
- Ideal for technologies like .NET

Remote Interfaces to Domain Logic

- Main kinds of remote interfaces:
  - **HTTP Interface:** through browser or Web service. HTML, JSON, XML,... formats for message content.
    - **Advantages:** almost universal (HTTP and XML are platform independent) and easy deployment.
    - **Disadvantage:** less explicit specification of the interface (XMLSchema, API docs,...).
  - **OO interface:** CORBA, ActiveX, Java RMI, ...
    - **Advantage:** very explicit interface specification (methods, parameters,...).
    - **Disadvantage:** more complex deployment (for instance need configured firewalls)

Contents

Web Presentation

- User interfaces based on **web browsers.**
  - **Advantages:** no software to install, common user interface and universal access.
- **Web Server:** interprets requested URLs and passes control to corresponding web application.

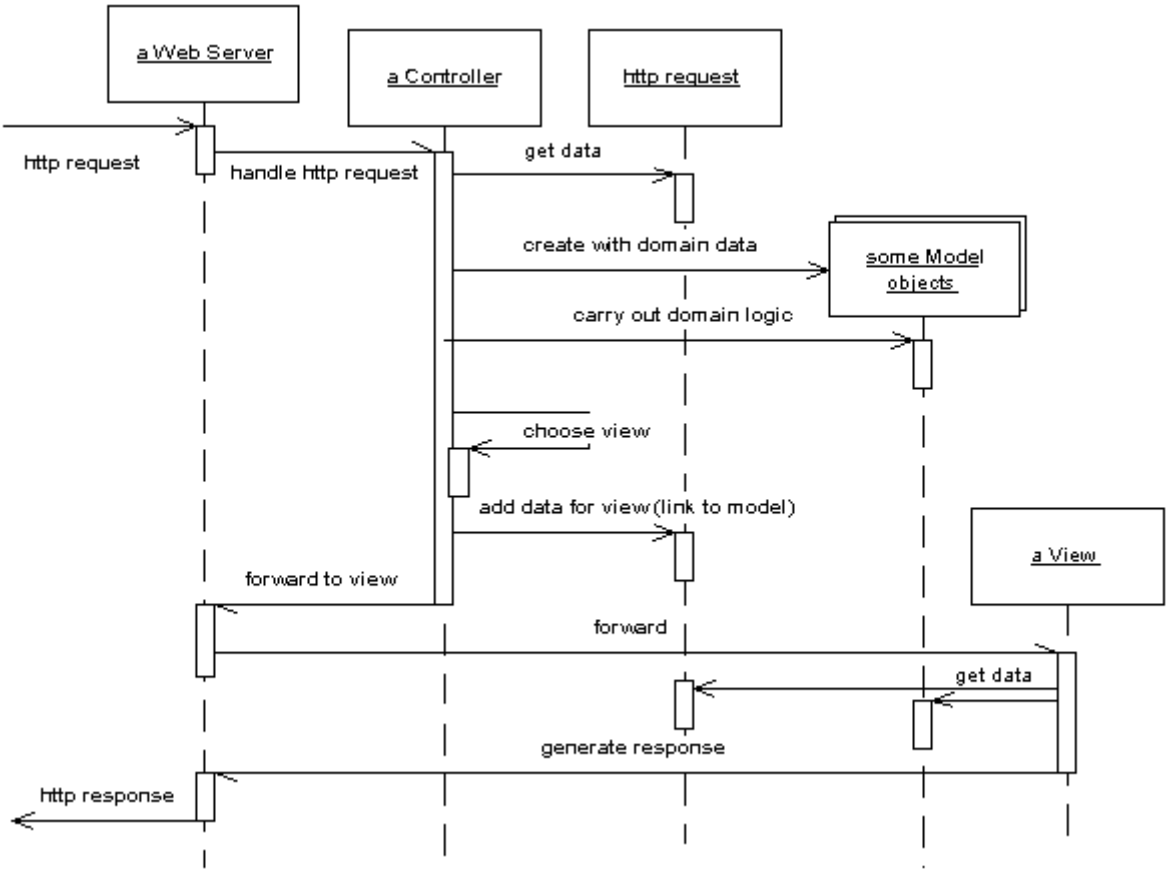
Web Presentation

- Main structural elements:
  - **Script:** program with HTTP processing capabilities, HTML string as output.
    - Ex .: CGIs or Java Servlets.
  - **Server page:** HTML + code template.
    - Ex .: PHP, ASP or JSP.
  - **Usage:** scripts to process requests and server pages to format responses.

Web Presentation

- Apply ***Model View Controler*** pattern:
  - **Controller** receives HTTP request.
  - Redirect to appropriate **Model** object.
  - Model object queries **data sources** and processes the response.
  - Returns to a **controller** that selects the appropriate view.
  - **View** receives the response using HTTP session object and displays it.

Web Presentation



View Patterns

- Three patterns for views:
  - ***Transform View, Template View*** and ***Two Step View***.
- Number of stages:
  - **One stage** for ***Transform View*** and ***Template View.***
  - ***Two stages*** when applying **Two Step View** to either ***Transform View View*** or ***Template View***.

Template View

- Page structure defines **presentation**, markers to include executable code for **dynamic content**.
- Usual pattern for **server pages.**
- **Advantage:** easy development.
- **Disadvantage:** messier code, mixed with template, harder to maintain.

Transform View

- Based on **transformation** language (eg. XSLT for XML) data.
  - Controller selects and applies the transformation to XML generated from the model.
- **Advantage:** flexible, adaptable to different presentation formats or devices.
- **Disadvantage:** complex.

Number of stages of the View pattern

- 2nd decision: **number** of **stages** of the view pattern (1 or 2).
- **1 stage:** a view component for each screen, Template or Transform View.
- **2 stages**: ***Two Steps View*** pattern, first step produces logical display and second one final display, for instance HTML presentation.

Two Seps View

- Features of the ***Two Steps View***:
  - One first step for each display.
  - Only **one** second step for the **entire application**.
  - **Advantages:** decisions about HTML to generate centralised in just one place, **facilitating global changes**.
    - Appropriate when different screens share basic design.
  - **Disadvantages**: introduces more complexity.

Controller Pattern

- ***Page Contraller***: a ***Controller*** object for each **page** of the website, or even for every **action** (button, link, ...).
- ***Front Controller***: **isolates** the responsibilities **to process and serve** HTTP **requests** and **to decide** what to do with them.
  - Single front object receives all requests and creates separate secondary objects for processing.
  - **Advantages**: centralize requests management.

Contents

Mapping to Relational DB

- **Problem:** store objects in relational databases.
  - Quite different **data structures.**
  - Objects in **memory,** coordinate with **disk** (database).
- **Alternative:** Object DBs.
  - **Risky**: not as mature as relational DB technology.

RDB Mapping Architectural Patterns

- For mapping to relational database:
  - ***Row Data Gateway***
  - ***Table Data Gateway***
  - ***Active Record***
  - ***Data Mapper***

Gateway Pattern

- ***Gateway:*** basic pattern for ***Row Data Gateway*** and ***Table Data Gateway.***
  - **In memory** class maps to DB table.
  - A **field** per DB **column.**
  - **Note**: patterns based on ***Gateway*** contain all the **mapping code** but not domain logic code.

Patterns based on Gateway

- Choice between ***Row Data Gateway*** and ***Table Data Gateway***.
- If technology plaform supports data recovery pattern ***Record Set***, then it is easier to use ***Table Data Gateway***.
  - For instance in Microsoft platforms like .NET.
- Otherwise, it is usually simpler to use ***Row Data Gateway.***
  - An object for each record in the data source.

Active Record pattern

- Appropriate when domain model **similar** to database schema.
- Combine the ***Gateway*** and the **domain object** in one class.
  - Class includes access to DB and business logic code.
  - **Disadvantage:** it is advisable to keep these responsibilities separate.

Data Mapper pattern

- More complex pattern, but also more **flexible.**
- **Advantage:** domain objects **independent** from DB schema.
- **Disadvantage**: added complexity.
  - However, for complex mappings, possible to **reuse existing** Object-Relational mapping **tools**.

Concurrence

- **Load and save** objects to/from the database.
- Track **changes** and guarantee **persistence.**
- **Concurrency**: avoid changing objects in DB while loaded.
- ***Unit of Work:*** **register** loaded and modified objects. **Manage updates** to the database.
  - Without this pattern, domain layer has to decide when to read and write.
  - ***Unit of Work*** extracts this behavior to a specific object.

Data reading

- Mapping classes encapsulate SQL for object-relational mapping.
  - For instance: search methods encapsulate SQL "SELECT" command.
- ***Identity Map*** pattern controls what is read.
- If ***Unit of Work*** implemented, ***Identity Map*** maintained in that object.

Mapping Data Structures

- **Problems mapping** relationalal DB **data** to domain model objects.
  - **Links:** objects use references while relational DBs use foreign keys.
  - **Collections:** objects implement them but DBs restricted to each field one value.
- ***Identity Field***: special field in the object keeping its identify, helps mapping between between objects and relational DB keys.

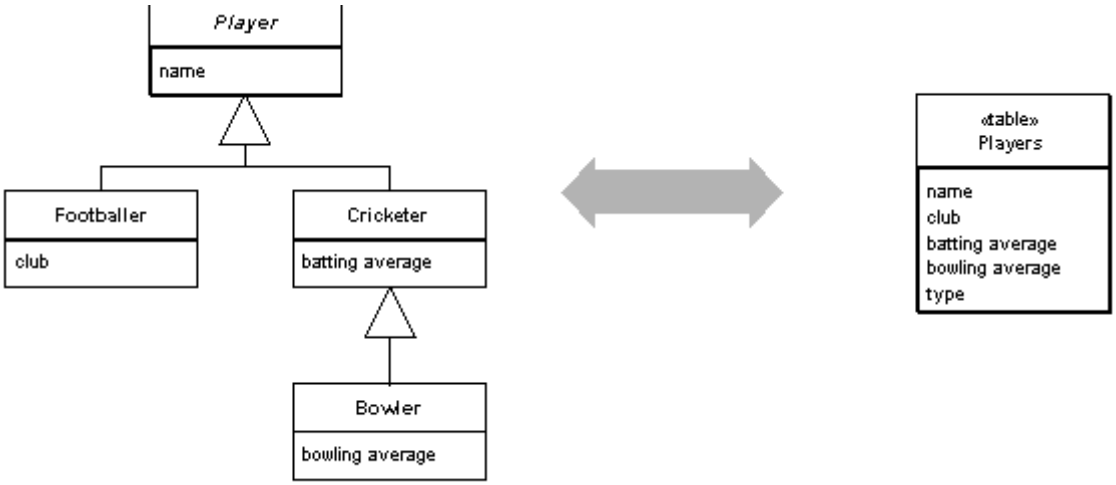
Mapping Data Structures

- When **reading foreign key**, if not yet in the ***Identity Map***, load object from DB using key.
- When **storing object**, references are replaced by the referenced entity ***Identity Field***.
- When **reading a collection**, retrieve rows linked to the identifier of the object keeping the collection.
- When **saving a collection**, store each object in it with foreign key pointing to the object keeping the collection.

Inheritance

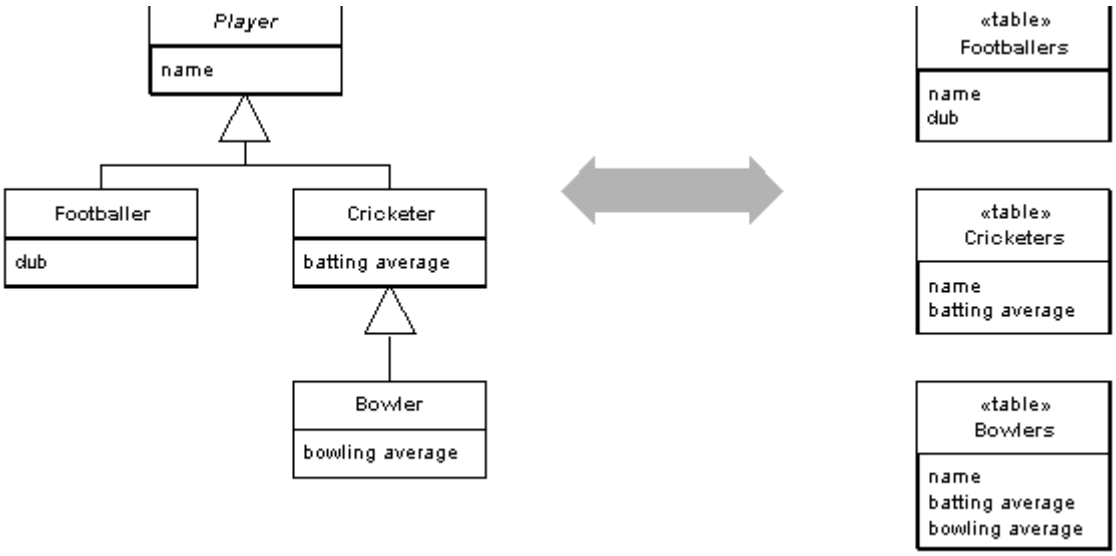
- Commonly, **relational databases without inheritance** mechanism.
- **Mapping** required to store it.
- 3 Options:

Inheritance



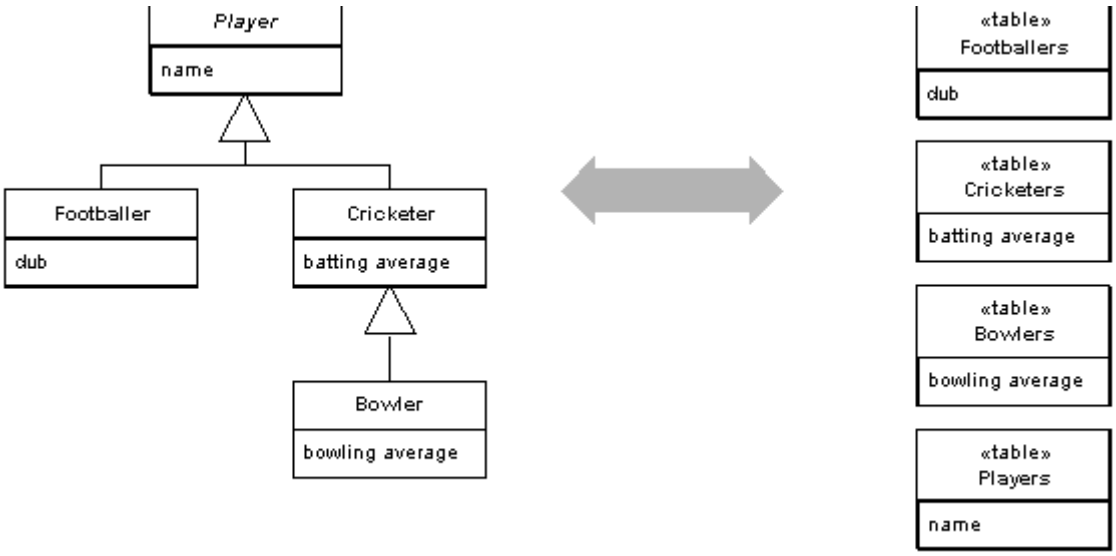
Single Table Inheritance, use a table to store all classes in a hierarchy

Inheritance



Concrete Table Inheritance, a table for each concrete class in the hierarchy, including inherited fields

Inheritance



Class Table Inheritance, use one table per class in the hierarchy, fields stored just in the table corresponding to the class defining it

Inheritance

- Compromise between duplication/flexibility and access speed.
- ***Class Table Inheritance:*** no duplication, very flexible/extensible, but inefficient, requires many JOINS.
- ***Concrete Table Inheritance:*** some duplication and less flexible because domain changes affect many tables, but more efficient, less JOINS.
- ***Single Table Inheritance:*** a lot of duplication and table affected by each domain change, but very efficient, without JOINS.

Mapping Metadata

- ***Metadata Mapping*** facilitates the reuse of mapping tools.
- Metadata file or annotations inform the mapping process.
    - For instance: how BD columns correspond to object fields.
    - Facilitate the generation of the mapping code.
- Common used in commercial tools.

<Field name = "customer" targetclass = "Customer" dbColumn = "custID" targetTable = "customers" lowerbound = "1" upperbound = "1" setter = "loadCustomer" />

Contents

Concurrence

- **Multiple processes** or threads manipulate the **same data**
- Within a **transaction**, Request/Response, avoid concurrency problems.
- **Problem**: interaction among different transactions.
- **Offline concurrency:** concurrency control spanning multiple transactions, interaction involving multiple Request/Response.

Concurrency Problems

- Problems to avoid:
    - **Lost updates**: User A editing, then user B too. B ends and then A, overwriting B's version.
    - **Inconsistent readings**: data readings correct separately but not correct if simultaneously.
- Avoid them using **concurrency control mechanisms**.

Execution Contexts

- **Interaction contexts:**
  - **Request**: from outside the system, optionally processed to compute a response.
    - Processing in the server and client on hold, waiting for response.
  - **Session**: set of related client-server interactions over time.
    - Commonly request/responses between log in and log out.

Execution Contexts

- **Processing context**:
  - **Process:** heavy weight execution context, isolates data.
  - **Thread**: lightweight execution context, one process multiple threads. They share memory so potential concurrency problems.
- **DB Access context**:
  - **Transaction**: coordinate requests accessing same data to avoid concurrency issues.

Isolation

- **Source** of concurrency problems: more than one process or thread **accessing** the same data.
- **Isolation**: mechanism to avoid these problems. **Coordinate access**.
  - Examples: operating system memory access mechanisms or file locks.

Concurrency Control

- **Shared data**: no isolation so coordinate concurrent access.
- Measures of concurrency control: **optimistic or pessimistic**.
- **Scenario**: 2 users simultaneously edit a file.

Concurrency Control

- **Optimistic locking**: focus on detecting conflict.
  1. Both user copy and edit file freely.
  2. Concurrency control when 2n user saves the file.
  3. Detect conflict between both changes.
  4. Tell 2nd user file cannot be saved. Options: discard, merge, new file,...

Concurrency Control

- **Pessimistic locking**: focus on preventing conflict.
  1. 1st user starts edition and file is blocked.
  2. No one else can edit till file is saved and lock released.
- **Prefer Optimistic**: when conflicts rare or consequences not critical, higher degree of concurrency.
- **Prefer Pessimist**: when big risk of conflicts or they are critical.

Inconsistent readings

- **Avoid** inconsistent readings, **Pessimistic Lock**:
  - Reading requires **read lock** (which is shared by readers)
  - Writing requires **write lock** (exclusive, just one writer).
    - Many users can read concurrently, if someone reading no one can write.
    - If requesting writer gets write block, no one else read or write.

Deadlock

- Pessimistic locking problem: **deadlocks**.
  - Example: 2 users editing different files, to complete their work they need to edit the files blocked by the other user.
- Options to **avoid** deadlocks:
  - **Detect** deadlock and break it by selecting a victim, release all his locks.
  - Locks with **expiry** time.
  - All **locks obtained at the beginning** of user work session. **Order** block are provided, for instance alphabetical.

Transactions

- **Transaction**: main **concurrency control** tool in enterprise applications.
- Bounded **sequence of actions**, start and end points.
  - Applied to DBs, message queues, printers, etc.
- Better performance when **transactions short** span only one request.

Transactions

- Properties:
  - **Atomicity**: complete all actions or undo them, **all or nothing**.
  - **Consistency**: resource in consistent state at the beginning and also at the end.
  - **Isolation:** result not visible externally until transaction completed successfully.
  - **Persistence**: successful transaction results are stored.

Business Transactions and System Transactions

- **System Transaction**: SQL commands to open and close transaction.
- **Business Transaction**: at a higher level. For instance online banking:
  1. Select account.
  2. Establish bills payment.
  3. Click "OK" button.

Business Transactions

- How to avoid **concurrency** problems in business transactions?
- **Not possible** to fit a business transaction within a system one.
- **Problem**: they span multiple request/responses.
  - Time between requests (user interaction) can lock the system.

Business Transactions

- **Atomicity and persistence:** apply changes just when user agrees.
  - Keep track of changes: for instance with ***Domain Model*** and ***Unit of Work***.
- **Consistency:** application responsible for implementing and maintaining business rules.
- **Isolation**: each session controls it does not overwrite others.

Offline Concurrency Control

- Concurrency patterns for transactions spannig more than one request.
- ***Optimistic Offline Lock***: optimistic pattern for business transactions.
  - Easier to implement and more efficient, but detects conflict at the end, changes may be lost.
- ***Pessimistic Offline Lock***: prevents conflict but more complex to implement and less response capacity.

Concurrency in Application Server

- Application server must manage multiple **simultaneous requests**.
- Simple solution: one **process per session**.
  - **Advantage**: each process isolated from the rest.
  - **Disadvantage**: resources consumption, each process dedicated memory.

Concurrency in Application Server

- Therefore, limit number of processes: **process pool**.
  - They **take turns** to attend waiting requests (queue).
  - Common option in old application servers.
- **Multithreaded** solution: multiple threads, threads pool, one thread per request.
  - Improves performance but should take care of **isolation** problems. Example: Tomcat.

Contents

Distributed Applications

- Applications with **distributed objects**, remote communication.
  - Objects from application design become separate components at implementation.
  - Each component executed on **separate processing nodes**.

Local and Remote Interfaces

- Distributed objects: maximize distribution degree but with **efficiency problems**.
  - A call between machines many orders of magnitude slower than within the same one (network delay).
- Differentiate **local** and **remote interfaces** for callable objects.
  - **Local interface**: for local calls, many methods with simple functionality. No problem if many calls needed.
    - Example: methods to set parts of an address (street, number,...).

Local and Remote Interfaces

  - **Remote interface**: for remote calls, methods provide complex functionality, minimize number of calls.
    - Example: one method to set address. Less calls required, more efficient but less inflexible.
- **Recommendation**: minimize remote calls, avoid distributed objects.
- **Alternative**: horizontal scalability, many computer with application replicas.
  - All application objects in a single machine.
  - Execute multiple copies of the application on different machines.

Distribution Scenarios

- However, some remote calls **unavoidable**:
  - Between client and server, two different machines.
  - Between application server and database.
    - Usually separate machines to improve performance and share data (avoid synchronization issues).
    - Leverage SQL Remote Interface, well prepared to minimize issues due to remote calls.

Distributed Objects Patterns

- ***Remote Facade***: groups methods to be called remotely, remote interface. Build on top of simpler local methods.
- ***Data Transfer Object***: for remote calls that receive or return objects, simplified version of local object that reduces amount of data sent.

Contents

Putting into Practice

- Guidelines about which patterns to use in developing a business application.
  - Options layer by layer.
  - Options depending on the technological platform.

Domain Layer

- Main options: ***Transaction Script, Table Module*** and ***Domain Model.***
- ***Transaction Script:*** simple applications, procedural model.
- ***Domain Model:*** complex or easier to extend domain logic. Difficulty of mapping to relational databases.
- ***Table Module:*** intermediate option, for more complex domain logics (not as extensible) while keeping connection to DB quite direct.

Data Layer

- Depends on choice in domain layer:
- ***Transaction Script***:
  - DB connection with ***Row Data Gateway*** or ***Table Data Gateway*** if platform implements ***Record Set***.
  - Minimises concurrency problems, script often corresponds to system transaction.
  - Exception: load data to edit it and then save. Solve conflicts applying ***Optimistic Offline Lock***.

Data Layer

- ***Table Module:*** best choice if good ***Record Set*** implementation.
  - Mapping with ***Table Data Gateway***.
  - ***Record Set*** usually incorporates ***Unit of Work*** for concurrency control.

Data Layer

- ***Domain Model:*** depending on domain complexity:
  - Simple model and stable (no need extensible), with available ***Active Record*** implementation: ***Table Data Gateway*** or ***Row Data Gateway***.
  - More complex logic or extensibility required: ***Data Mapper***, reusing existing mapping tools[one].
  - ***Unit of Work*** for concurrency control.

[1] http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

Presentation Layer

- Rich client-side Web application (e.g. AngularJS) or Server-side Web application (e.g. SpringMVC).
  - Server-side for simple HTML applications (server templates, e.g. JSP, server sends HTML). Client-side for more interactive apps (server as API sends data, e.g. JSON).
- ***Model View Controller***: used in both cases. Model in domain layer, controller and view in presentation layer.

Presentation Layer

- Controller:
  - ***Page Controller***: for simple document-oriented sites, mixing static and dynamic pages.
  - ***Front Controller***: for more complex and interactive interfaces, both client or server-side.

Presentation Layer

- View:
  - ***Template View***: common option, simplicity, server pages but also client-side.
  - ***Transform View***: usual when XML input, allows multiple output formats.
  - ***Two Step View***: unusual, for very complex views with different output formats.

Reference

- This presentation is based on the book:

  Martin Fowler and David Rice:
  Patterns of Enterprise Application Architecture.
  Addison-Wesley, 2003. ISBN 0321127420