# **Patterns of Enterprise Application Architecture**

Martin Fowler The Addison-Wesley Signature Series 2003 525 pages Ref: http://nida.se/patterns/poeaa/index.html

## A. Domain Logic Patterns

## 1. Transaction Script (pp 100-115)

Goal: Organizes business logic by procedures where each procedure handles a single request from the presentation.

```
How:
abstract class TransactionScript {
  abstract void run()
}
```

#### 2. Domain Model (pp 116-124)

Goal: An object model of the domain that incorporates both behavior and data.

#### 3. Table Module (pp 125-132)

Goal: A single instance that handles the business logic for all rows in a database table or view.

## 4. Service Layer (pp 133-142)

Goal: Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

## B. Data Source Architectural Patterns

When the database is the master model.

```
CREATE TABLE PERSON (
ID : INTEGER
LAST_NAME : VARCHAR()
..
```

### 5. Table Data Gateway (pp 144-151)

Goal: An object that acts as a Gateway to a database table. One instance handles all the rows in the table.

How:

```
class PersonGateway {
  find(int) : ResultSet
  findWithLastName(String) : ResultSet
  update(id, name)
  insert(id, name)
  delete(id)
}
```

## 6. Row Data Gateway (pp 145-159)

Goal: An object that acts as a Gateway to a single record in a data source. There is one instance per row.

How:

```
class PersonFinder {
  find(id) : Person
  findWithLastName(String) : Person[]
}
class Person {
  int id;
  String lastName;
  insert()
  update()
  delete()
}
```

## 7. Active Record (pp 160-164)

Goal: An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

How:	
class Person {	
int id;	
String lastName;	
<pre>//mapper methods insert(); update(); delete();</pre>	
<pre>//behavioral methods getExemption(); isFlaggedForAudit(); getTaxableEarnings(); }</pre>	

Same as Row Data Gateway, but add behavioral methods. Good if domain logic is not too complex.

## 8. Data Mapper (pp 164-181)

Goal: A layer of Mappers that movers data between objects and a database while keeping them independent of each other and the mapper itself.

How:	
class Person {	abstract class AbstractMapper {
int id;	//mapper methods
String lastName;	public abstract insert();
	public abstract update();
//behavioral methods	<pre>public abstract delete();</pre>
getExemption();	protected find(id);
isFlaggedForAudit();	protected load();
getTaxableEarnings();	}
}	
	class PersonMapper extends AbstractMapper {
	Person person;
	String tablename;
	}

Advantage: object model and database schema may evolve independently.

## C. Object-Relational Behavioral Patterns

## 9. Unit of Work (Transaction) (pp 184-194)

Goal: Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

Unit of works follow the ACID principles:

- <u>A</u>tomicity: everything is committed or everything is rolled back
- <u>C</u>onsistency: the system must be in a consistent, non-corrupt state just before the start of the transaction and right after its completion.
- <u>I</u>solation: the changes are not visible to any other transaction until the transaction successfully commits.
- <u>D</u>urability: after the commit, changes are persistent.

How:



Essential if objects are persisted in a DBMS; useful otherwise.

Instead of updating the DBMS table at each change in the object model (which leads to lots of DB calls), register objects in a cache and perform the change at the commit time.

Advantages:

- Centralize the DB calls in one place, the commit() method
- Undo/Redo easy to implement once we have defined a "unit of work"

## 10. Identity Map (pp 195-199)

Goal: Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks for objects using the map when referring to them.

#### 11. Lazy Load (pp 200-214)

Goal: An object that doesn't contain all the data you need but knows how to get it

## D. Object-Relational Structural Patterns

When the object model is the master model.

#### 12. Identity Field (pp 216-235)

Goal: Saves a database ID in an object to maintain identity between an in-memory object and a database row.



Variants:

- if Person() constructor is public, generates the key and stores in the DB.
- if Person() constructor is private, the find() method looks for a key generated by the DBMS (e.g. Oracle SEQUENCE data type).

Recommendations:

- Prefer meaningless key (a surrogate ID) to meaningful key (eg: a U.S. Security Number).
- Prefer simple key (one column) to compound key (on two or more columns).
- Prefer database-unique key to table-unique key.
- Prefer long integer type to String type.

## 13. Foreign Key Mapping (pp 236-247)

Goal: Maps an association between objects to a foreign key reference between tables.

How (with single-value association) :

//classes (the master model)	//the schema generated from the OO model
class Artist {	create table ARTIST (
}	long ID
	)
class Album {	
Artist artist;	create table ALBUM (
}	long ID,
	long ARTIST ID
	)

With a collection of objects, the reference is reversed:

//classes (the master model)	create table TRACK (
class Track{	long ID,
}	long ALBUM_ID
	)
class Album {	
Track[] tracks;	create table ALBUM (
}	long ID,
	)

Limitation: many-to-many association not supported (see Association Table Mapping)

## 14. Association Table Mapping (pp 248-261)

Goal: Saves an association as a table with foreign keys to the tables that are linked by the association.

How:

//classes (the master model)	//the schema generated from the OO model
class Skill {	create table SKILL (
}	long ID (PK)
class Employee {	
//the same skill can be shared	create table EMPLOYEE (
//among employees	long ID (PK)
List <skill> skills;</skill>	
}	
	create table SKILL-EMPLOYEES (
	SKILL ID long,
	EMPLOYEE ID long,
	PK(SKILL ID, EMPLOYEE ID)
	)

No: the table SKILL-EMPLOYEES as no ID, because has no corresponding in-memory object. The PK is the compound of the two FKs.

## 15. Dependent Mapping (pp 248-267)

Goal: Has one class perform the database mapping for a child class.

How:	
//a track cannot exist w/o its album	create table TRACK (
class Track{	long ID,
protected Track(Album owner);	long ALBUM ID,
	PK(ID, ALBUM ID)
	)
class Album {	
Track createTrack() {	create table ALBUM (
new Track(this);	long ID,
}	
}	

## 16. Embedded Value (pp 268-271)

Goal: Maps an object into several fields of another object's table.

How:	
class Order{	CREATE TABLE ORDER (
Address billTo,	BILL_TO_CIVIC_NUMBER,
Address shipTo	BILL_TO_STREET_NAME,
}	BILL_TO_CITY,
	BILL_TO_ZIP_CODE,
class Address {	SHIP_TO_CIVIC_NUMBER,
int civicNumber,	SHIP_TO_STREET_NAME,
String streetName,	SHIP_TO_CITY,
String city,	SHIP_TO_ZIP_CODE,
String ZIPCode	
}	

Advantage: eliminate joint operations in a relational database.

## 17. Serialized LOB (pp 272-277)

Goal: Saves a graph of objects by serializing them into a single large object (LOB), which it stores in a database field.

How:

```
class Customer {

Department[] getDepartments()

}

class Department {

Department getParent()

Department[] getChildren()

}

CREATE TABLE CUSTOMERS (

DEPARTMENTS : BLOB

)
```

Advantage: eliminate a graph of small database rows:

Inconvenient: the department structure only visible in the object side (hidden in the relational database).

## 18. Single Table Inheritance (pp 278-284)

Goal: Represents an inheritance hierarchy of classes as a single table that has columns for all the fields of the various classes.



Strengths:

- No joint in retrieving data (fast).
- Refactoring the class model by moving a field upward or downward in the hierarchy does not affect the database.

Weaknesses:

- Some columns are irrelevant depending PLAYER\_TYPE (wasted space).
- Table may have too many columns; name conflict may occur.

## 19. Class Table Inheritance (pp 285-292)

Goal: Represents an inheritance hierarchy of classes with one table for each class.



Strengths:

- All columns are relevant for each row: no wasted space
- The database model reflects the OO model (easy to understand)

Weaknesses:

- You need to touch multiple tables to load an object: many joints (slow)
- Any refactoring in the OO model affects the database design
- The supertype (PLAYER) may become a bottleneck because they have to be accessed frequently.

## 20. Concrete Table Inheritance (pp 293-301)

Goal: Represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy.

How:



Strengths:

- Each table is self-contained and has no irrelevant fields (no wasted space)
- There are no joins to do when reading the data (fast)

Weaknesses:

- PKs can be difficult to handle.
- You can't enforce database relationship to abstract classes.
- If superclass field changes, you need to change each table that has this field because the superclass field is duplicated across the tables.
- A find on the superclass forces you to query all the tables (e.g. Select \* from <tables> where name equals "Smith").

## 21. Inheritance Mappers (pp 302-304)

Goal: A structure to organize database mappers that handle inheritance hierarchies.

## E. Object-Relational Metadata Mapping Patterns

## 22. Metadata Mapping (pp 306-304)

Goal: Holds details of object-relational mapping in metadata.

How:	
class DataMap {	class ColumnMap {
Class domainClass,	String fieldName,
String tableName,	String columnName,
ColumnMap[] columns,	}
}	

## 23. Query Object (pp 316-321)

Goal: An object that represents a database query.

How:	
class Query {	class Criterion {
Criterion[] criteria; //linked by AND	String operator = "<",
	String field = "salary",
addCriterion(criterion);	Object value = 50;
}	}

How (most sophisticated):

class Query {	class SimpleCriterion -> ICriterion {
ICriterion criterion;	String operator = "<",
	String field = "salary",
	Object value = 50;
}	}
<pre>interface ICriterion{ }</pre>	<pre>class CompoundCriterion -&gt; ICriterion {     booleanOperator {OR, AND, XOR}     leftOperand: ICriterion,     rightOperand: ICriterion, }</pre>

Advantage: hides SQL queries, changing column names does not affect Java code.

# 24. Repository (pp 322-327)

Goal: Mediates between the domain and the data mapping layers using a collection-like interface for accessing domain objects.

How:

abstract class Repository { protected List matching(Criteria);	class RelationalStrategy extends Repository { }
}	class InMemoryStrategy extends Repository { }

## F. Web Presentation Patterns

## 25. Model View Controller (pp 330-332)

Goal: Splits user interface interaction into three distinct roles.

Advantages:

- Different concerns. Often people prefer one area to another and they specialize in one side of the line.
- Multiple presentations (rich client, web browser, command-line interface) for the same model.
- Nonvisual objects easier to test than visual ones; test domain logic w/o awkward GUI scripting tools.

### 26. Page Controller (pp 333-343)

Goal: An object that handles a request for a specific page or action on a Web site.

How:

client (web browser)	class PageController { //handle HTTP get and post	class Model { }
	<pre>//decide which model and //view to use }</pre>	class View { //generate HTML }

#### 27. Front Controller (pp 344-349)

Goal: A controller that handles all requests for a Web site.

How:	
class Handler {	interface ICommand {
doGet();	process()
doPost();	}
}	
	class Command -> ICommand {
	}

## 28. Template View (pp 350-360)

Goal: Renders information into HTML by embedding markers in an HTML page.

#### 29. Transform View (pp 361-364)

Goal: A view that processes domain data element by element and transforms it into HTML.

## 30. Two Step View (pp 365-378)

Goal: Turns domain data into HTML in two steps: first by forming some kind of logical page, then rendering the logical page into HTML.

### 31. Application Controller (pp 379-386)

Goal: A centralized point for handling screen navigation and the flow of an application.

### G. Distribution Patterns

#### 32. Remote Facade (pp 388-400)

Goal: Provides a coarse-grained facade on fine-grained objects to improve efficiency over a network.

## 33. Data Transfer Object (pp 401-413)

Goal: An object that carries data between processes in order to reduce the number of method calls.

## H. Offline Concurrency Patterns

### 34. Optimistic Offline Lock (pp 416-425)

Goal: Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction.

Optimistic assumes that the chances of conflict is low: it is not likely that multiple users to work with the same data in the same time.

```
How:
```

WriteTransaction tx = session.createWriteTransaction();

try {

data.write(tx, newvalue); //throw exception if data is being written by another user tx.commit();

} catch (DbException ex) {

tx.rollback(); //data set back to oldvalue

## 35. Pessimistic Offline Lock (pp 426-437)

Goal: Prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data.

Pessimistic assumes that the chances of conflict is low: it is likely that multiple users to work with the same data in the same time.

Different strategies:

- Exclusive write lock: no more than one write transaction in the same time. Acceptable for non-critical operations.
- Exclusive read-lock: no more than one transaction (read or write) in the same time.
- Read/write lock: a write lock blocks either write or read transactions; concurrent read locks are acceptable (the best strategy, but the most complicated to implement).

How:

try {
 //throw ConcurrencyException if Data.class locked by another user
 WriteTransaction tx = session.createWriteTransaction(data.class);
 data.write(tx, newvalue);
 tx.commit();
} catch (ConcurrencyException ex) {
 //report exception
}

### 36. Coarse-Grained Lock (pp 438-448)

Goal: Locks a set of related objects with a single lock.

How:	
<pre>class Version {     int ID;     static Version create(); //increment id</pre>	class Customer { Version version; Address[] addresses;
<pre>static Version load(id); }</pre>	<pre>} class Address {   Customer owner;   Version version = owner version;</pre>
	}

### 37. Implicit Lock (pp 449-453)

Goal: Allows framework or layer supertype code to acquire offline locks.

How:

class BusinessTransaction {	HibernateTransaction {
HibernateTransaction tx_m;	
	}
BusinessTransaction() {	
<pre>tx_m = new HibernateTransaction();</pre>	
}	
<pre>void commit() {</pre>	
tx_m.commit();	
}	
}	

## I. Session State Patterns

## 38. Client Session State (pp 456-461)

Goal: Stores session state on the client.

## 39. Database Session State (pp 462-464)

Goal: Stores session data as committed data in the database.

### J. Base Patterns

#### 40. Gateway (pp 466-472)

Goal: An object that encapsulates access to an external system or resource.

#### 41. Mapper (pp 473-474)

Goal: An object that sets up a communication between two independent objects.

#### 42. Layer Supertype (p 475)

Goal: A type that acts as the supertype of all types in its layer.

#### 43. Separate interface (pp 476-479)

Goal: Defines an interface in a separate package from its implementation.

#### 44. Registry (pp 480-485)

Goal: A well-known object that other objects can use to find common objects and services.

#### 45. Value Object (pp 486-487)

Goal: A small simple object, like money or a date range, whose equality isn't based on identity.

How:

```
class Address {
    //fields: civic number, street name, etc..
    public boolean equals(Object other) { //overrides java.lang.Object
    return (other instanceof Address) && (equals(Address)other);
    }
    public boolean equals(Address addr) {
        compare fields..
    }
    public int hash() { //everytime we define equals(), we must redefine hash()
    }
}
```

#### 46. Money (pp 488-495)

Goal: Represents a monetary value

```
How:
```

```
class Money {
 //amount of cents, or the smaller base unit, enough to store 92 223 720 G$...
 long amount m;
 Currency currency;
 //amount in dollars
 Money(long amount, currency) {
  amount m = amount * centFactor();
 }
 Money(double amount, currency) {
  amount m = Math.round(amount * centFactor());
 }
 Money(double amount) {
  this(amount, Currency.USD);
 }
//compareTo
public int compareTo(Object other) {
  return compareTo(Money)other);
 }
 public int compareTo(Money other) {
  assertSameCurrencyAs(other);
  if (amount < other.amount) return -1;
  else if (amount == other.amount) return 0;
  else return 1;
 }
public boolean greaterThan(Moner other) { return (compareTo(other) > 0); }
//define equals() & hashCode()
//monetary operations
 Money add(Moner other) {
  assertSameCurrency(other);
  return new Money(amount + other.amount);
 }
 Money multiply(double factor, int roundingMode) { //to solve Foemmel's Conundrum
```

### 47. Special Case (pp 496-498)

Goal: A subclass that provides special behavior for particular class.

#### 48. Plugin (pp 499-503)

Goal: Links classes during configuration rather than compilation.

## 49. Service Stub (pp 504-507)

Goal: Removes dependence upon problematic services during testing.

## 50. Record Set (pp 508-510)

Goal: An in-memory representation of tabular data.

Missing: nullable, indexes