AOT LAB

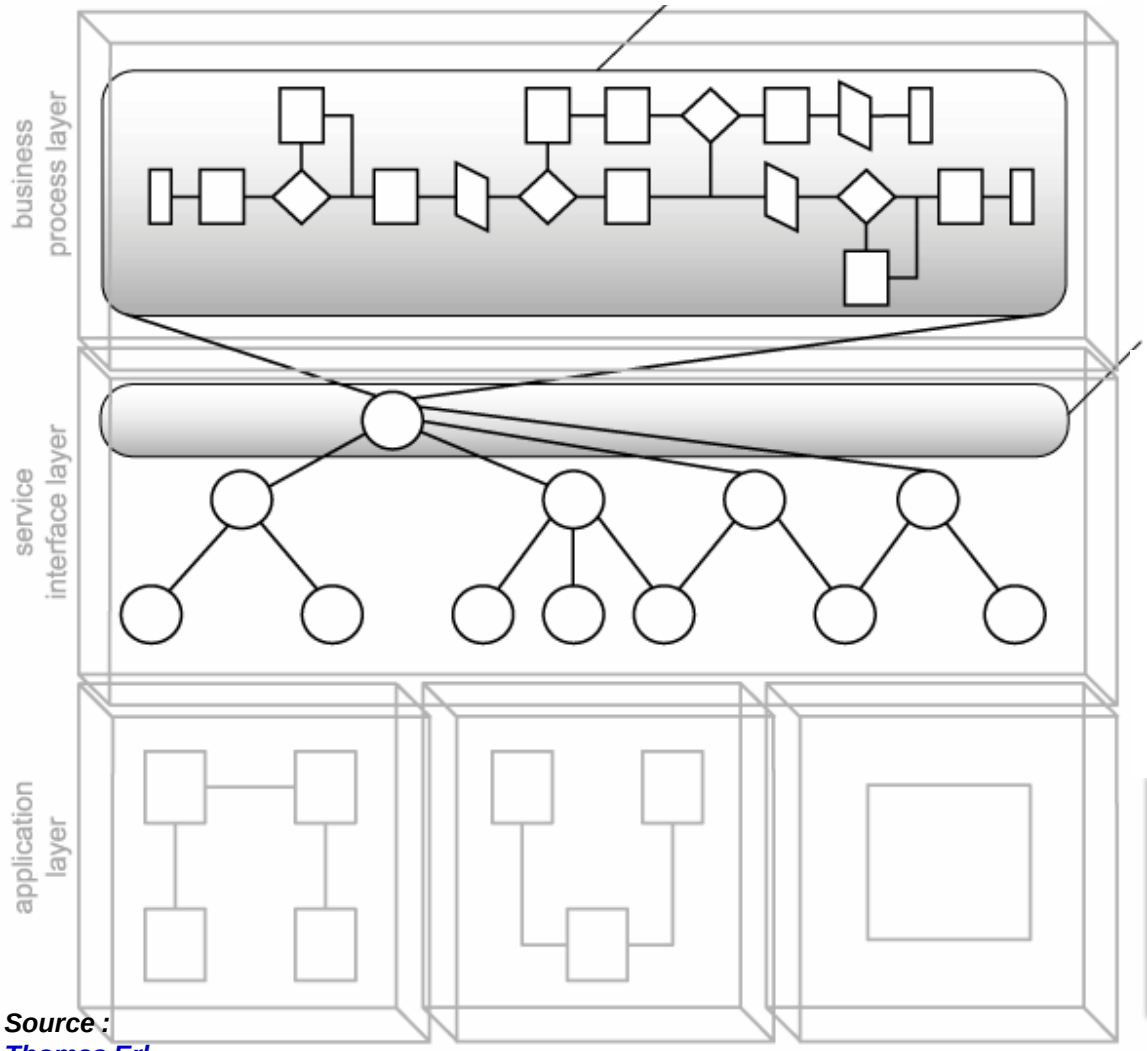
Agent and **O**bject **T**echnology **Lab**
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma

Enterprise Application Integration

Paola Turci

AOT Lab - DII - Università di Parma

Enterprise Applications



- *Business Process Modeling*
  - *UML Activity Diagram*
  - *BPMN*
  - *WSBPEL*
  - *Process Integration Languages*
- *Integration issues and "traditional" approaches*
- *Service-Oriented Architecture*
  - *Service-Oriented paradigm*
  - *Web Services*
    - *Core Web Services Standards*
  - *Semantic Web Services: SAWSDL*
- *Enterprise Applications*
  - *Overview*
  - *Architectural solutions*
    - *Patterns of Enterprise Application Architecture*

Source :
Thomas Erl
Service-Oriented Architecture: Concepts, Technology, and Design

Enterprise Applications

- What characterizes enterprise applications?
- Usually
  - Involve persistent data
    - Older systems used indexed file
    - Modern systems usually use databases, mostly relational databases
  - There is a lot of data
  - Many people access data concurrently
  - There is a lot of user interface screens to handle the amount of data
  - They need to integrate with other enterprise applications scattered around the enterprise or with their business partner

An Example

- A B2C online retailer: People browse and possibly buy.
- Requirements
  - To be able to handle a very high volume of users
  - Reasonably efficient in terms of resources used
  - Scalable; possible to increase the load by adding more hardware.
  - Anyone should be able to access the system easily; a pretty generic Web presentation that can be used with the widest possible range of browsers
- Domain logic: order capturing, some relatively simple pricing and shipping calculations, and shipment notification.
- Data source includes a database for holding orders and perhaps some communication with an inventory system to help with availability and delivery information.

Performance Factors

- *Response time* – time to process a request
- *Responsiveness* – time to acknowledge a request (e.g. a progress bar)
- *Latency* – time required to get any form of response (significant in distributed systems). Important to minimize remote calls.
- *Throughput* – how many things can be done in a given amount of time (e.g. transactions per second)
- *Load* – a statement of how much stress a system is under (e.g. how many users are currently connected to it). Usually a context for some other measurement (e.g. *Load sensitivity* is an expression of how the response time varies with the load).
- *Efficiency* – performance divided by resources
- *Scalability* – a measure of how adding resources (usually hardware) affects performance.

→ *Design decisions do not affect all of these performance factors equally*

Response time & Responsiveness

- Quoted from **J. Nielsen** 's “Response Times: The Three Important Limits” (Excerpt from *Usability Engineering*):
  - The response time guidelines for web-based applications are the same as for all other applications. These guidelines have been the same for 37 years now, so they are also not likely to change with whatever implementation technology comes next.
    - **0.1 second:** Limit for users feeling that they are directly manipulating objects in the UI. For example, this is the limit from the time the user selects a column in a table until that column should highlight or otherwise give feedback that it's selected.
    - **1 second:** A delay of 0.2-1.0 seconds does mean that users notice the delay and thus feel the computer is "working" on the command, as opposed to having the command be a direct effect of the users' actions. Example: If sorting a table according to the selected column can't be done in 0.1 seconds, it certainly has to be done in 1 second, or users will feel that the UI is sluggish and will lose the sense of "flow" in performing their task. For delays of more than 1 second, indicate to the user that the computer is working on the problem, e.g. by changing the shape of the cursor.
    - **10 seconds:** Limit for users keeping their attention on the task. Anything slower than 10 seconds needs a percent-done indicator as well as a clearly signposted way for the user to interrupt the operation. Assume that users will need to reorient themselves when they return to the UI after a delay of more than 10 seconds. Delays of longer than 10 seconds are only acceptable during natural breaks in the user's work, for example when switching tasks.

Significant Issues in the Design of Enterprise Applications

Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

The Architecture

"Architecture" is a term that lots of people try to define, with little agreement. There are two common elements: one is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change. It's also increasingly realized that there isn't just one way to state a system's architecture; rather, there are multiple architectures in a system, and the view of what is architecturally significant is one that can change over a system's lifetime.

The Architecture (II)

- Represents the skeleton of the application
- Choosing the software platform
  - A ***Web-based application***, would be accessed by a Web browser, the data would reside at the Web server
  - A ***database application***, using a relational database. The forms package and the proprietary language to write the application
  - A ***stand alone application***, using a visual object-oriented programming language. User interfaces could be created by using visual construction tools, invoking software functions needed to store, retrieve and manipulate data

Layering (Multi-Tiered Applications)

- To break apart a complicated software system
  - Benefits:
    - A single layer can be understood as a coherent whole without knowing much about the other layers
      - Minimize dependencies between layer
    - Layers can be substituted with alternative implementations
    - Layers make good places for standardization
    - A single layer can be used for many higher-level services
  - Downsides:
    - Extra layers can harm performance. Transformation from one representation to another
    - Layers encapsulate some, but not all, things well. Sometimes cascading changes

→ *The hardest issue is deciding what layers to have and what the responsibility of each layer should be*

Evolution of Layers in EA

- In the '90s client–server systems
  - Two-layer systems: the client held the user interface and other application code; the server was usually a relational database
    - Generally a screen was built by *dragging* controls onto a design area and then using property sheets to connect the controls to the database
- Problems came with domain logic: business rules, validations, calculations, …
  - Usually written on the client, by embedding the logic directly into the UI
    - As the domain logic got more complex, this code became very difficult to work with
    - Simple changes resulted in hunting down similar code in many screens
  - Alternative: put the domain logic in the database as stored procedures
    - Stored procedures gave limited structuring mechanisms
    - SQL as a standard would allow changing the database vendor (few people actually did this). Stored procedures removed that option. since they are all proprietary

Evolution of Layers in EA (II)

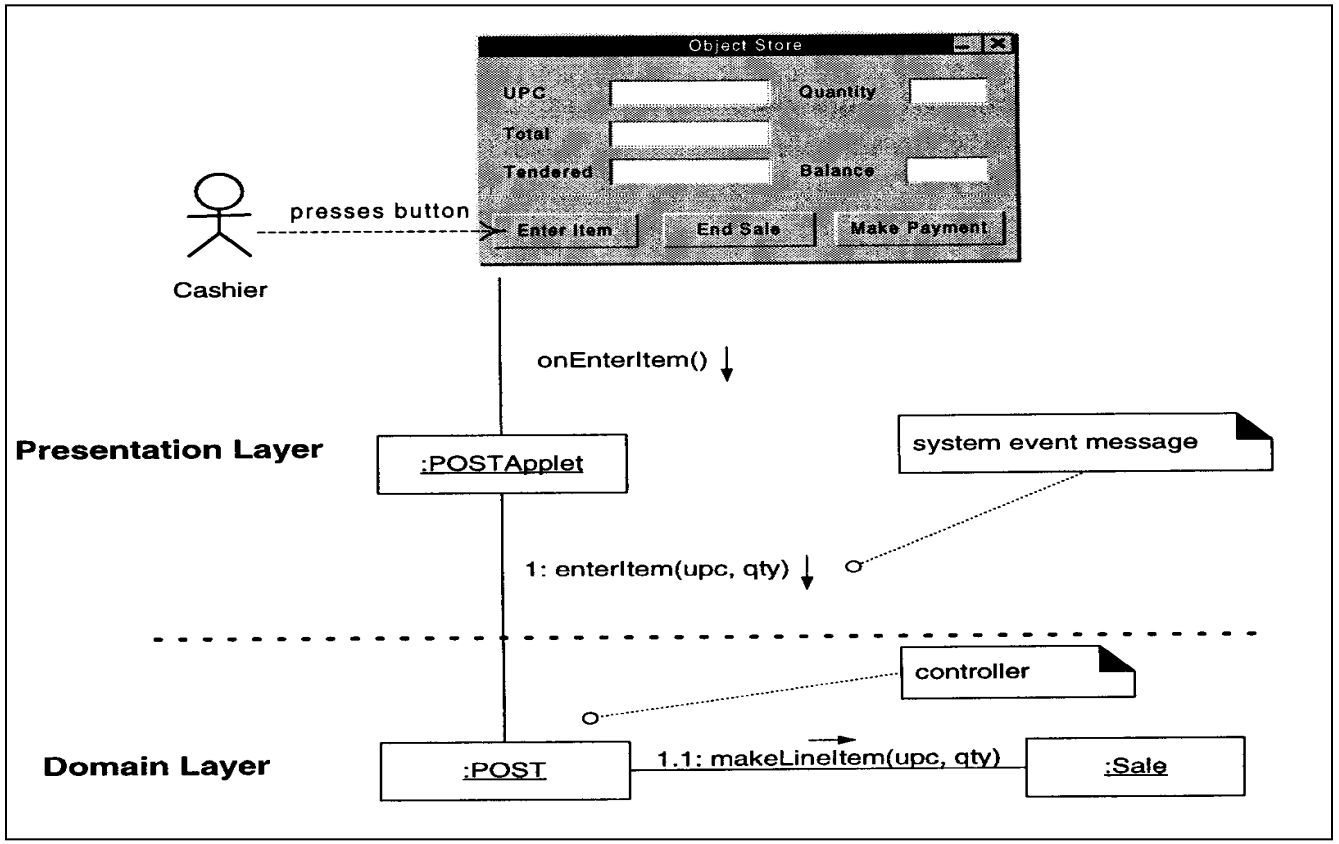
- The object oriented community had an answer to the problem of domain logic:
  - Move to a three-layer system
    - A presentation layer for the UI, a domain layer for the domain logic, and a data source layer
    - Issues
      - Many systems were simple
        - Although the three-layer approach had many benefits, the tooling for client–server was compelling if the problem was simple
      - The client–server tools were difficult, or even impossible, to use in a three-layer configuration
- With the rise of the Web …
  - The tools that appeared to build Web pages were much less tied to SQL and thus more open to a third layer

The Three Principal Layers

- *Presentation logic* - handles the interaction between the user and the software
  - Responsibilities: to display information to the user and to interpret commands from the user invoking the corresponding actions
  - Can be as simple as a command-line or text-based menu system, or a rich-client graphics UI or an HTML-based browser UI
- *Domain logic*, aka business logic
  - Involves calculations based on inputs and stored data, validation of any data, and choosing the right data source logic
- *Data source logic* - communicates with other systems
  - Generally a database, responsible for storing persistent data

Presentation/Domain Layers

- Desirable coupling of presentation to domain layer



The Three Principal Layers (II)

- The domain layer should completely hides the data source from the presentation
- BUT … the presentation quite often accesses the data store directly
  - Less pure, but it tends to work better in practice
    - The presentation may interpret a command from the user, use the data source to pull the relevant data, and then let the domain logic manipulate that data before presenting it
- Asymmetric layering scheme
  - To highlight the distinction between
    - Presentation, an external interface for a service the system offers
    - Data source, an interface to things providing a service used by the system

The Three Principal Layers (III)

- The kind of separation depends on how complex the application is
  - A general advice is to make some kind of separation
    - A simple script (e.g. to pull data from a database and display it in a Web page) may be one procedure. Still it is better to separate the three layers, e.g. by placing the behavior of each layer in separate subroutines
    - As the system gets more complex, break the three layers into separate classes
    - As complexity increased, divide the classes into separate packages
  - A steady rule about dependencies
    - The domain and data source should never be dependent on the presentation
- Separation between layers is useful even if the layers are all running on one physical machine

Choosing Where to Run Layers

- Where to run the presentation depends mostly on the kind of user interface
  - Running code on a client improves responsiveness (rich client)
  - Web interface, all processing is on the server
    - Everything is easy to upgrade and fix
    - No deployment to many desktops and keeping them all in sync with the server
    - No problems of compatibilities with other desktop software
      - Alternative: use of scripting and downloadable applets could reduces the browser compatibility
  - *Web presentation if you can, the rich client if you must*
- The data source generally runs only on servers
  - Exception: duplication of the server functionality onto a powerful client, usually for disconnected operation
- Domain logic: all on the server or all on the client, or split it

Organizing Domain Logic

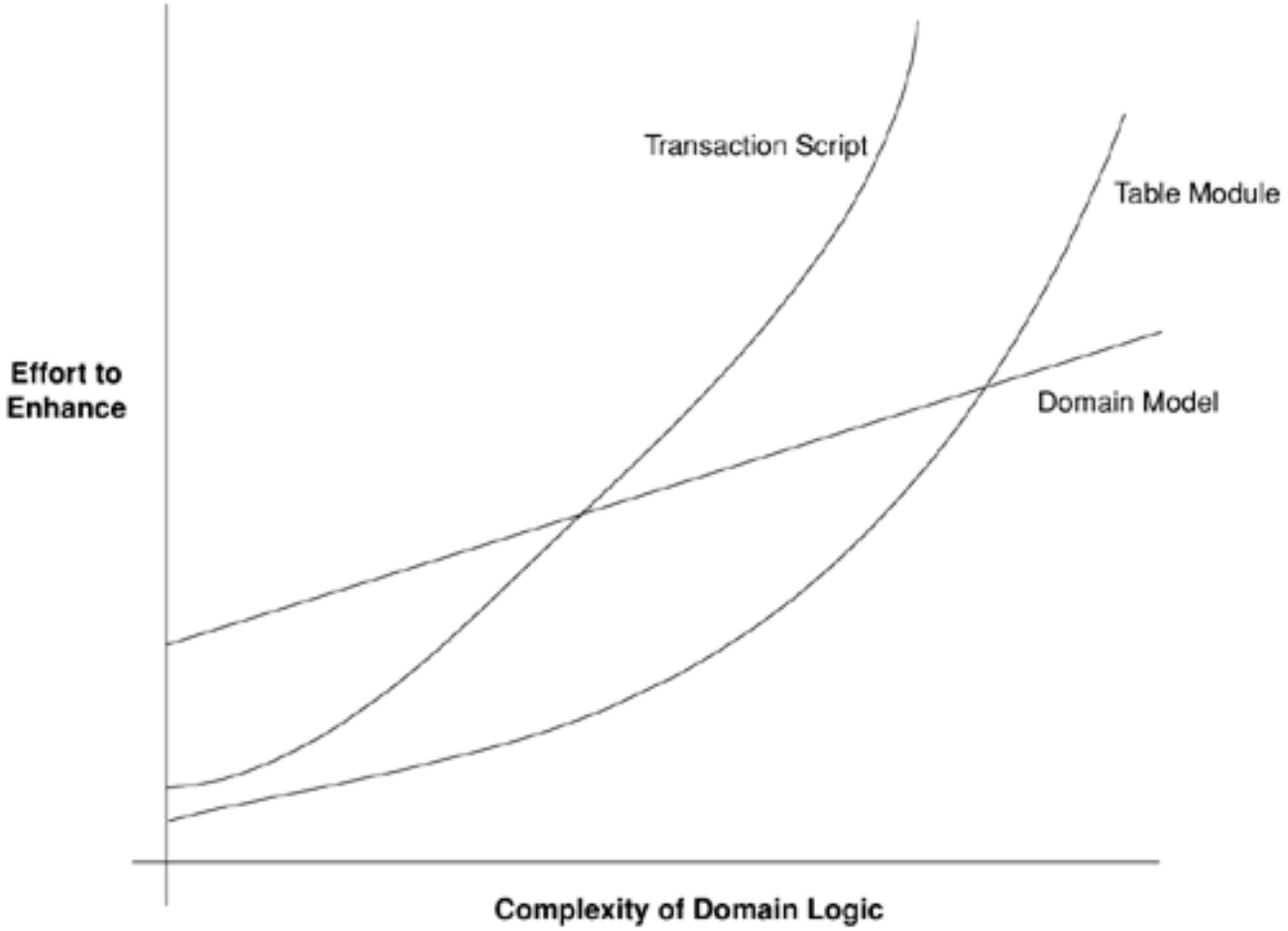
"Domain logic talks to DBMS"

- Three primary patterns or approaches, sorted wrt the complexity of the domain logic:
  - **Transaction Script**
    - Organizes business logic by procedures; each procedure handles a single request from the presentation
  - **Table Module**
    - The domain logic is organized around tables rather than procedures
  - **Domain Model**
    - An object model of the domain that incorporates both behaviour and data
  - *They are not mutually exclusive choices*
  - **Service layer**
    - A common approach is to split the domain layer in two
    - Establishes a set of available operations and coordinates the application's response in each operation; placed over an underlying Domain Model or Table Module

Organizing Domain Logic (II)

- **Object model**
  - Main issue: mapping between in-memory objects and relational database tables
  - OO vs. Relational
    - OO databases improve productivity
    - BUT … most projects do not use OO databases
      - The primary reason against OO DBMS is risk
        - Relational databases are a well-understood and proven technology supported by big vendors.
        - SQL provides a relatively standard interface
    - Sophisticated commercial O/R mapping tools
    - *Hibernate*, open source ORM framework

Making a Choice



- *A sense of the relationships between complexity and effort for different domain logic styles*

Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Managing Third Party APIs

- Third party APIs are a fact of life for software engineers
  - Database engine, middleware engine, class libraries, ...
- *Possible use*
  - Direct call to them from our application code
- *Problems*
  - Our application code becomes more and more polluted with such API calls
  - This is a problem when third party API changes
    - For database it also become a problem when the schema changes
- *Solution*
  - Adding a layer that separates the application business rules from the third party API
    - Such layers can sometimes be purchased (e.g. frameworks implementing JDBC or JDO specifications)
      - Of course they are also third party APIs, therefore the application may need to be insulated even from them

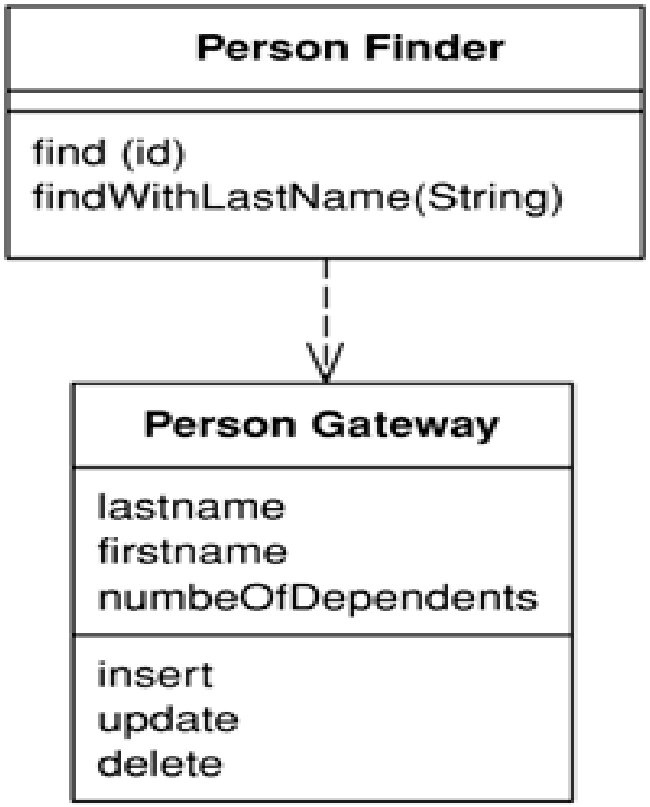
*"**How** the domain logic talks to the DBMS"*

- The choice made here has wide implications on the design and thus it is difficult to refactor
  - Possible solutions to separate a DBMS access from the domain logic
    - *Row Data Gateway,*
    - *Table Data Gateway*
    - *Active Record*
    - *Data Mapper*
    - *Proxy*

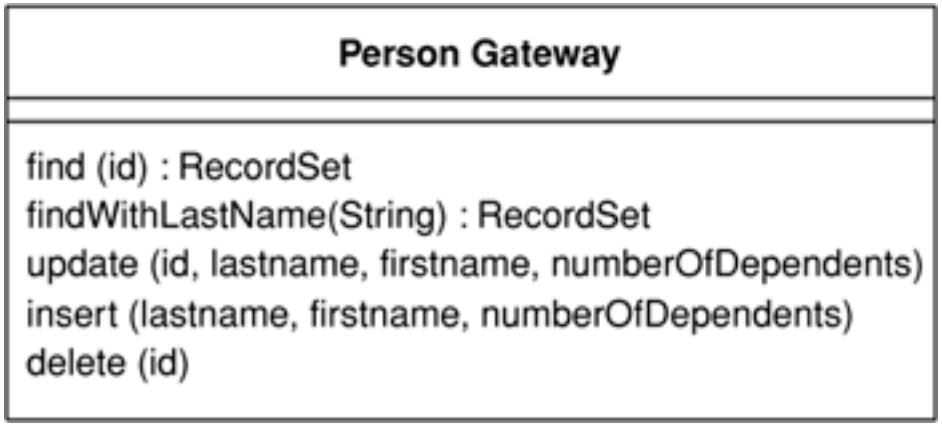
Gateway Pattern

- An object that encapsulates access to an external system or resource
  - Wraps all the special APIs into a class whose interface looks like a regular object
  - Two possible implementations:
    - **Row Data Gateway**
      - An object that acts as a Gateway to a single record in a data source. There is one instance per row
    - **Table Data Gateway**
      - An object that acts as a Gateway to a database table. One instance handles all the rows in the table
      - Provides methods to query the database that return a Record Set (a data structure that consists of a group of database records; It can come as the result of a query to the table)

Gateway Pattern - Examples



- *A Row Data Gateway has one instance per row returned by a query*

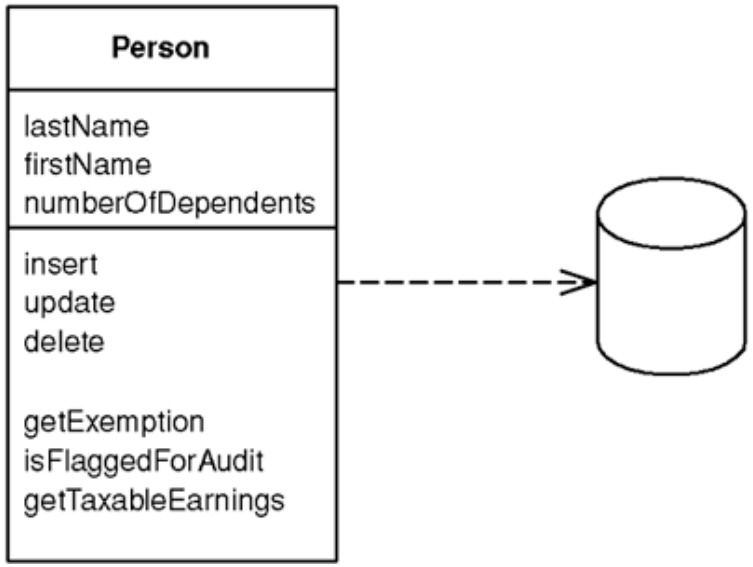


- *A Table Data Gateway has one instance per table. Holds all the SQL for accessing a single table or view*

Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Active Record

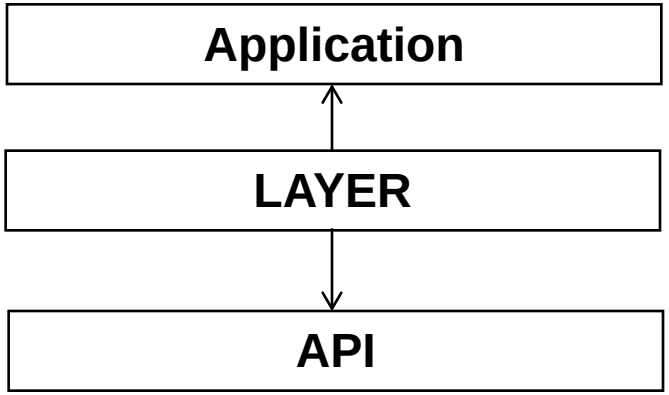
- An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data



- It is a good choice for domain logic that is not too complex
  - The Active Record objects correspond directly to the database tables: an isomorphic schema.
- Weakness: it couples the object design to the database design

Data Mapper Pattern

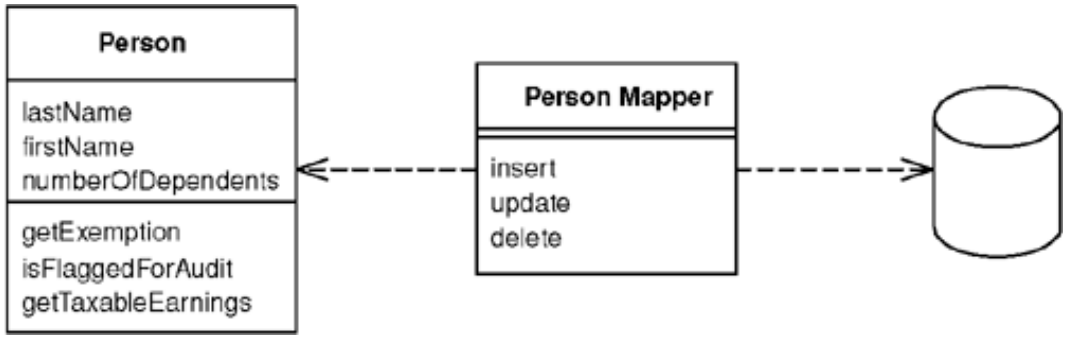
- In order to attain even better insulation, it is necessary to invert the dependency between the application and the layer



  - The application does not depend upon the middle layer
  - The middle layer depends upon the application and the API
  - All knowledge of the mapping between the application and the API is concentrated into the middle layer

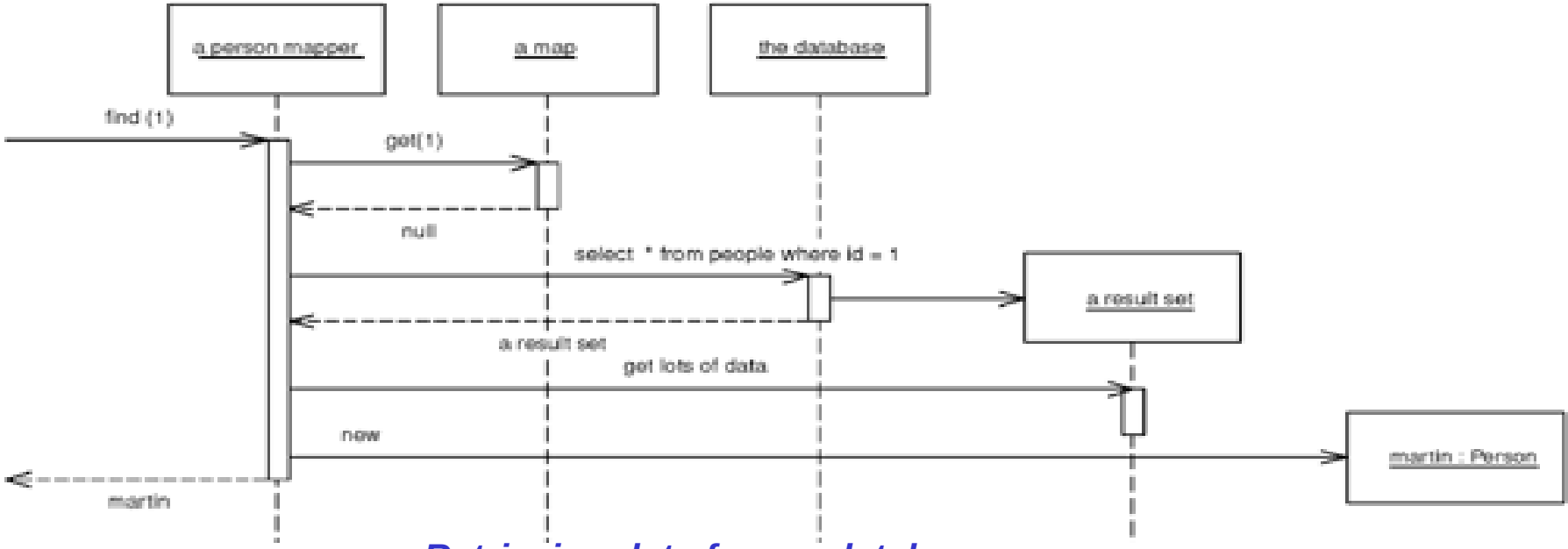
Data Mapper Pattern (II)

- Moves data between objects and a database keeping them independent of each other and the mapper itself

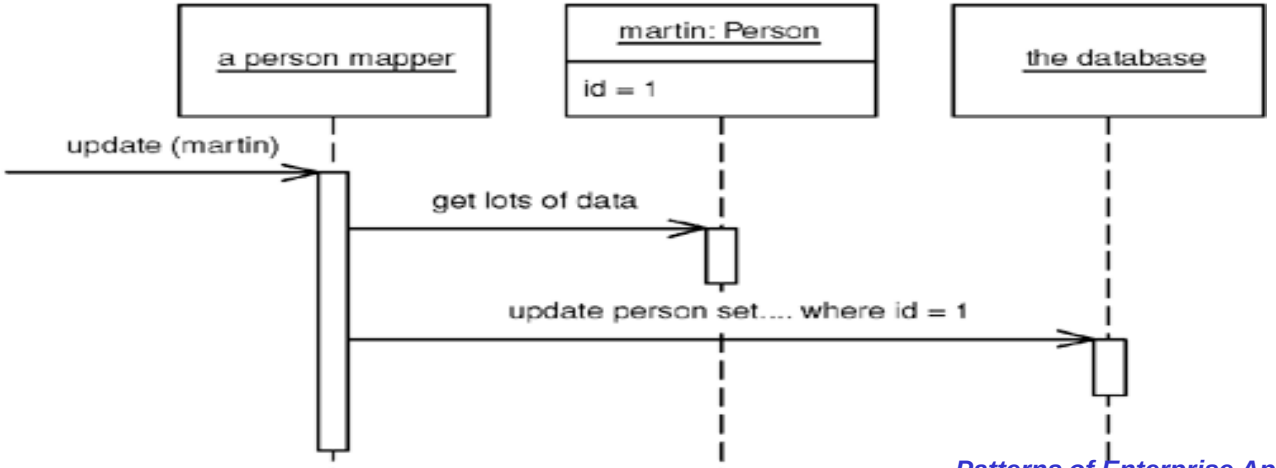


Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Data Mapper Pattern (III)



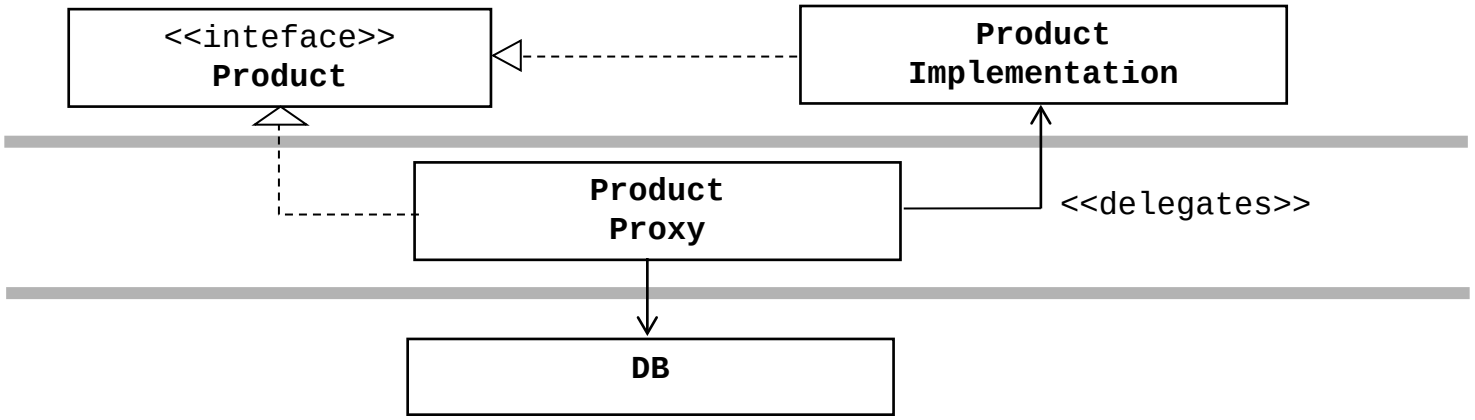
- Retrieving data from a database



- Updating data

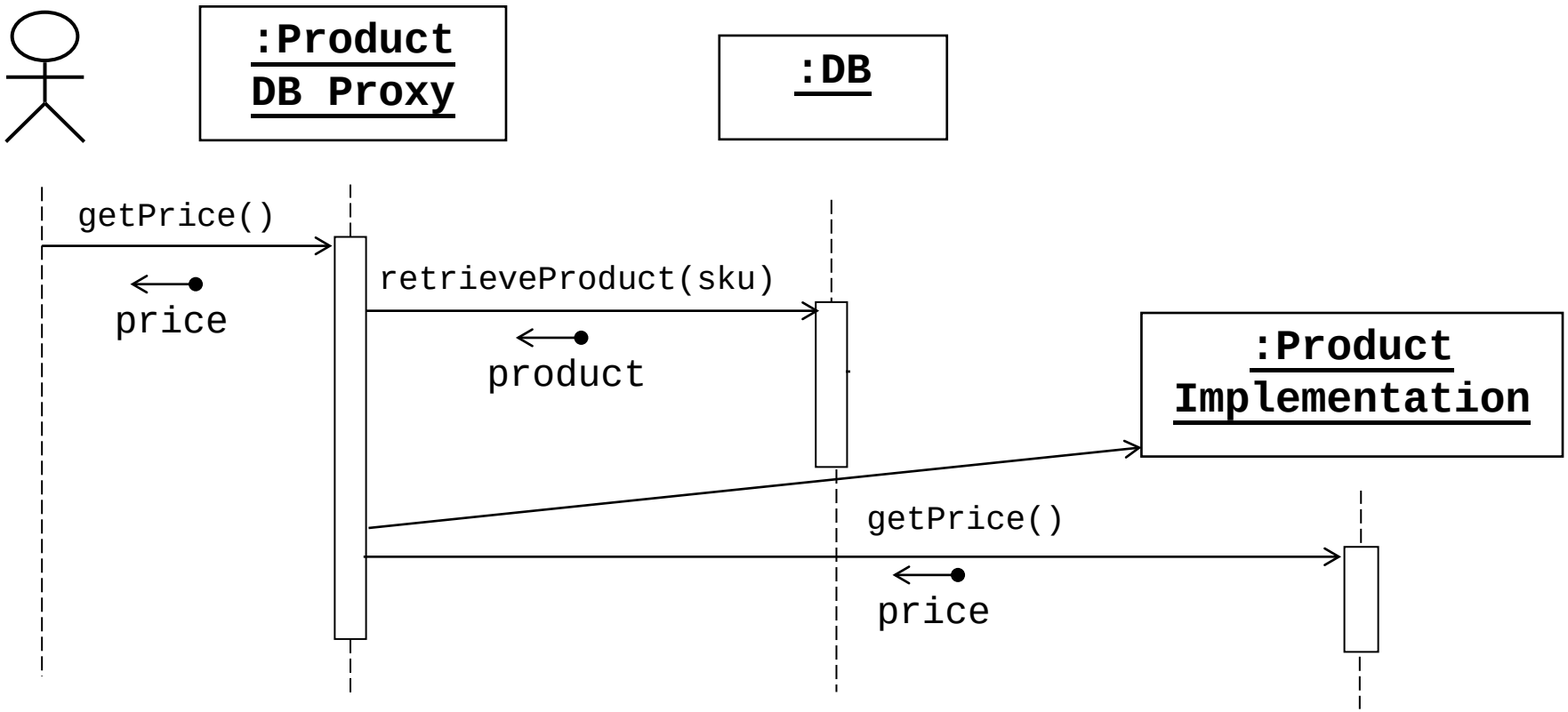
Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

PROXY Pattern



- A proxy is an object that looks like the object that should be in the field, but does not actually contain anything. Only when one of its methods is called does it load the correct object from the database
  - Every object that is to be proxied is split into three parts
- Whenever the API changes, the proxies change; whenever the application changes, the proxies change
  - The proxies are nightmares
    - It is good to know where nightmares live
    - Without the proxies the nightmares would be spread throughout the application code
  - Proxies are a very heavyweight solution
    - Proxies achieve an intense separation between the application and the API

PROXY Pattern – Dynamic Model

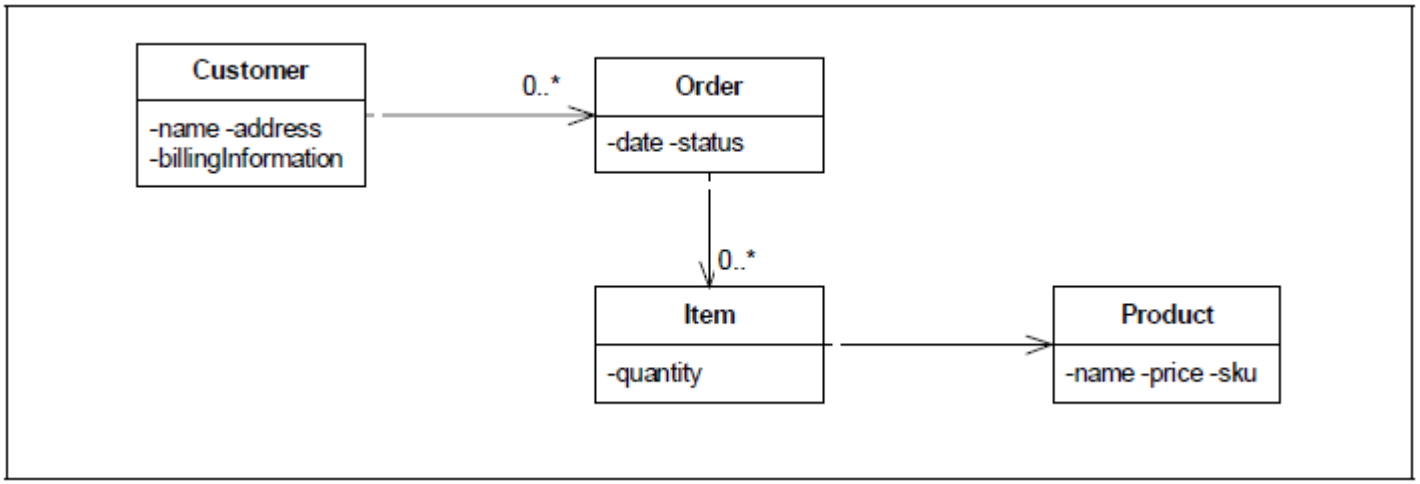


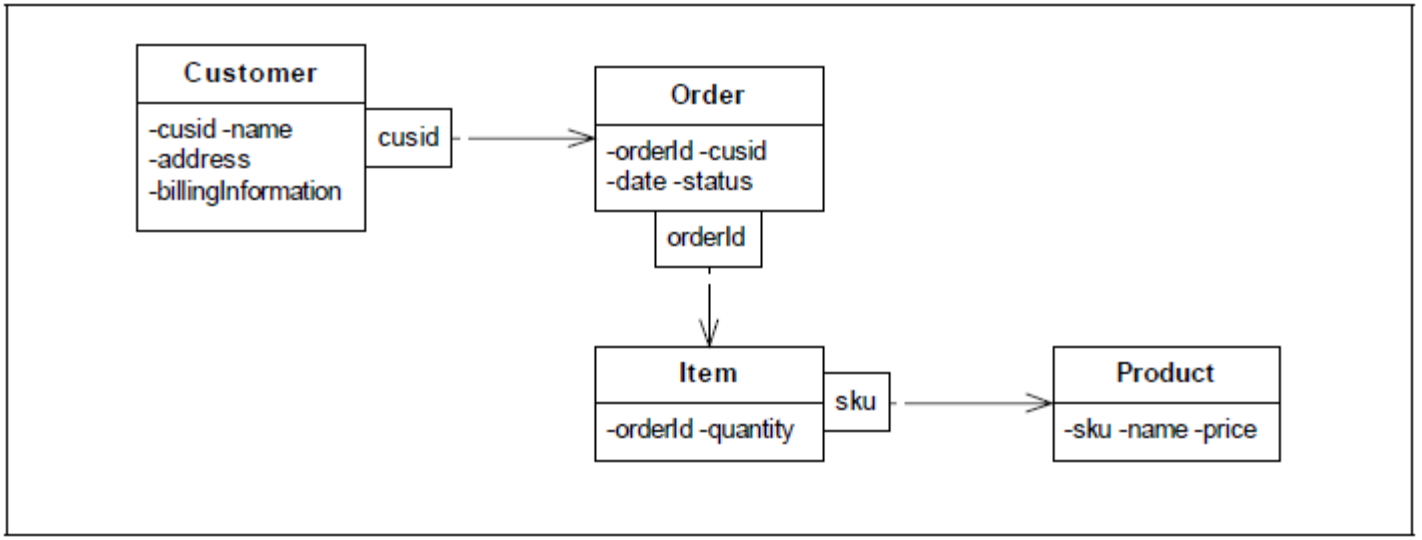
- The client sends the `getPrice` message to what it thinks is a `Product`, but what is really a `Product-DBProxy`. The `ProductDBProxy` fetches the `ProductImplementation` from the database. It then delegates the `getPrice` method to it.

Structural Mapping Issue

- Strictly connected with Data Mapper and Proxy patterns
  - The central issue is the different way in which objects and relations handle links
    - Different representation
      - Objects handle links by storing references that are held by the runtime support
      - Relational databases handle links by forming a key into another table
    - Different way of handling “collections”
      - Objects can use collections to handle multiple references
      - All relation links are single valued
        - Es: an order object has a collection of line item objects that do not need any reference back to the order. In the table structure, the line item must include a foreign key reference to the order since the order cannot have a multivalued field

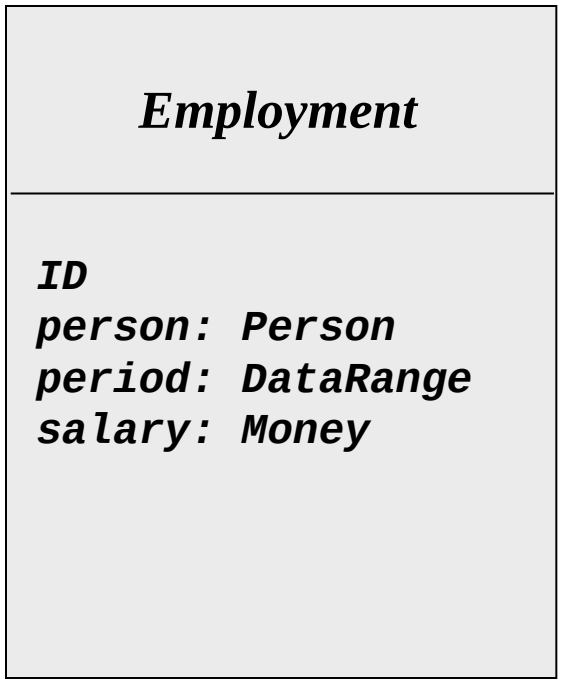
Example

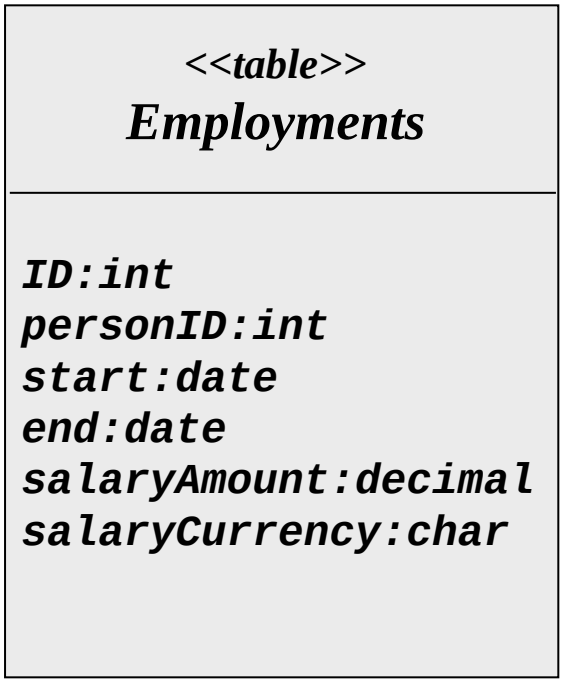

Customer
-name -address
-billingInformation
0..*
Order
-date -status
0..*
Item
-quantity
Product
-name -price -sku


Customer
-cusid -name
-address
-billingInformation
cusid
Order
-orderId -cusid
-date -status
orderId
Item
-orderId -quantity
sku
Product
-sku -name -price

Embedded Value

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





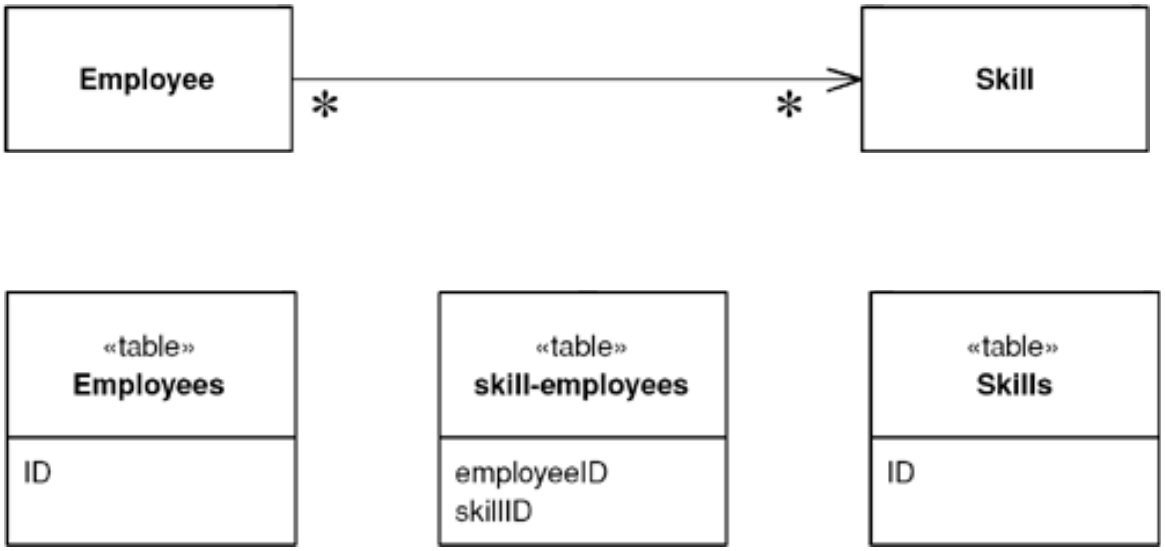
Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Structural Mapping Patterns

- **Identity Field**
  - Saves a database ID field in an object to maintain identity between an in-memory object and a database row
    - Important when you need to write data back to the database
- **Foreign Key Mapping**
  - Useful to map a single-valued field
  - Maps an association between objects to a foreign key reference between tables
    - Small Value Objects (e.g. date ranges and money objects), should not be represented as their own table
- **Dependent Mapping**
  - One class performs the database mapping for a child class
    - One class (the dependent) relies upon another class (the owner) for its database persistence. Each dependent must have one and only one owner
  - Useful in the case of collection objects that are not used outside the scope of the owner

Structural Mapping Patterns (II)

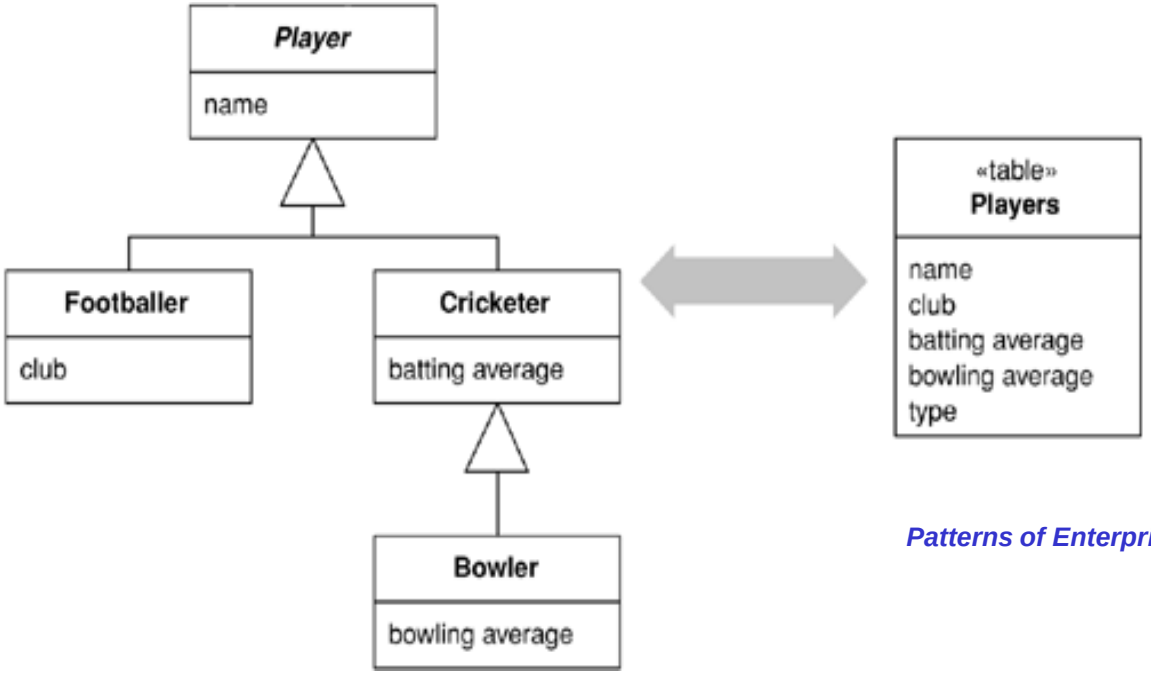
- **Association Table Mapping**
  - Saves a many-to-many association as a table with foreign keys to the tables that are linked by the association



Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Structural Mapping Patterns (III)

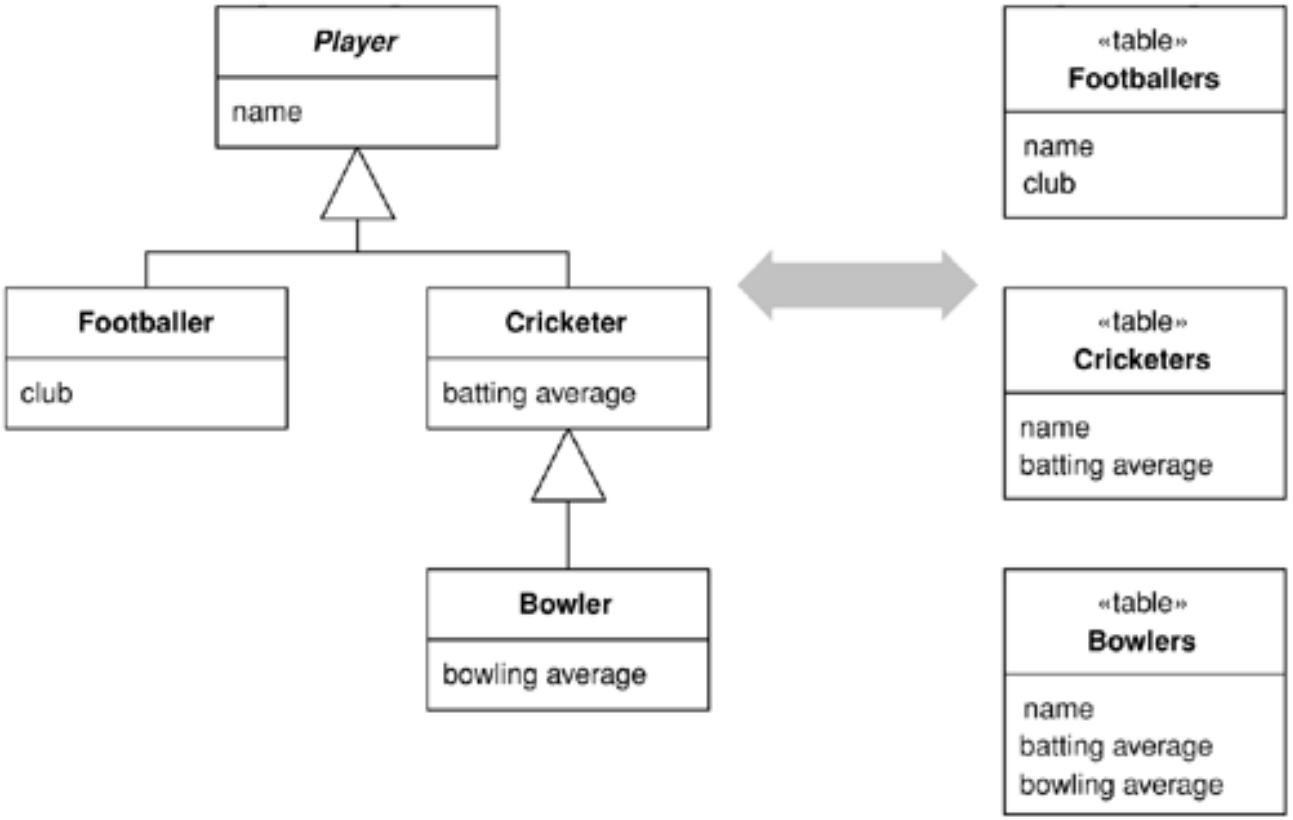
- Inheritance
  - There is no standard way to do inheritance in SQL
  - For any inheritance structure there are basically three options:
    - One table for all the classes in the hierarchy: *Single Table Inheritance*
      - Downside:
        - Wasted space. Each row has to have columns for all possible subtypes and this leads to empty columns (compression of wasted table space)
        - A bottleneck for accesses



Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Structural Mapping Patterns (IV)

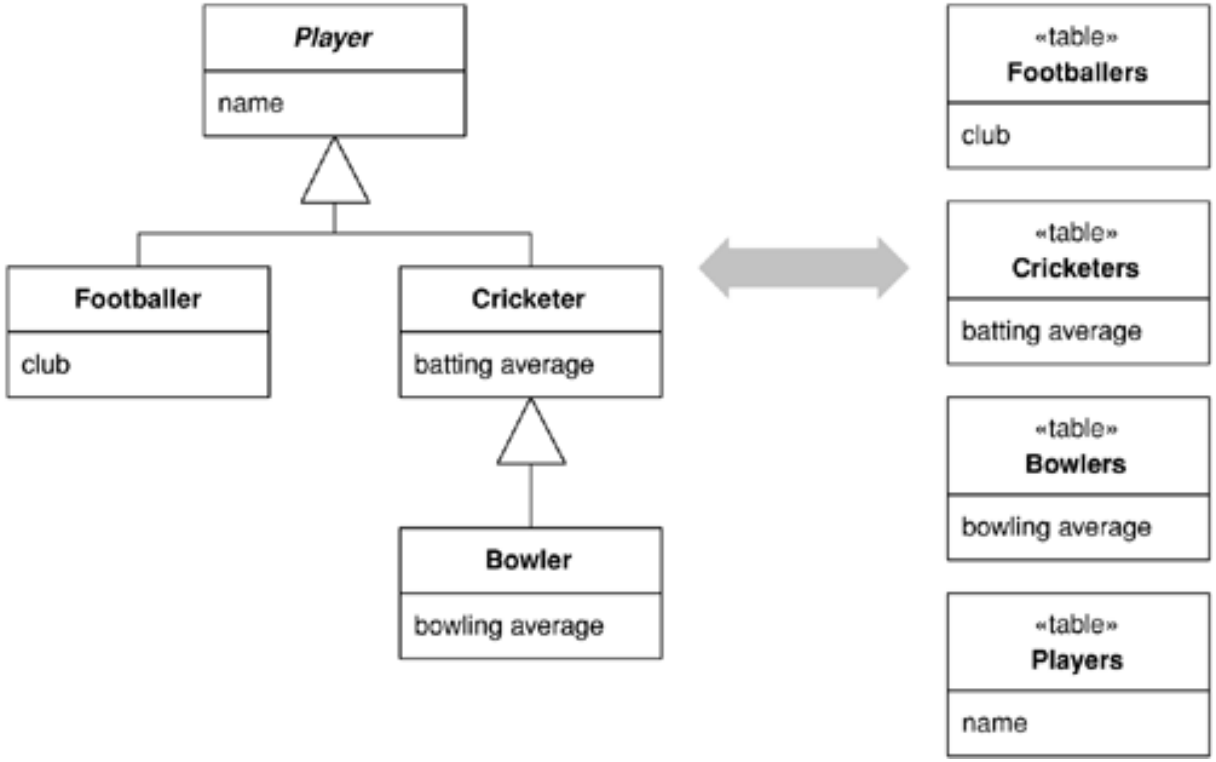
- One table for each concrete class: *Concrete Table Inheritance*
  - It is weak to changes



Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Structural Mapping Patterns (V)

- One table per class in the hierarchy; *Class Table Inheritance*
  - Needs multiple joins to load a single object, which usually reduces performance

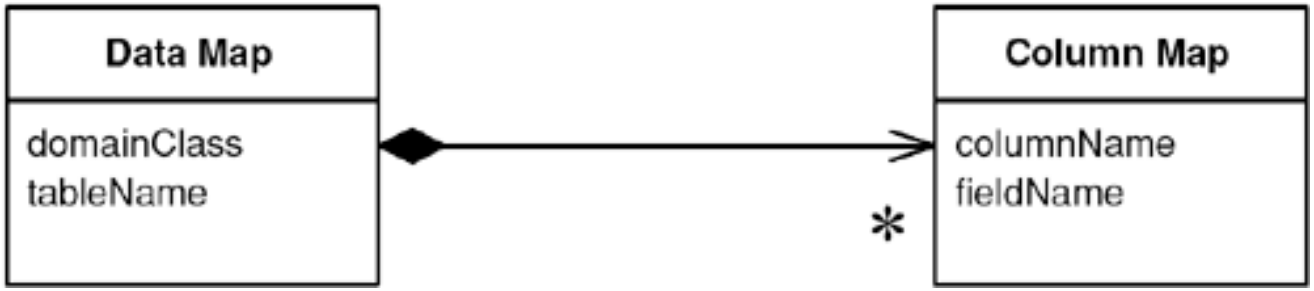


- The trade-offs are all between duplication of data structure and speed of access

Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Metadata Mapping

- Holds details of object-relational mapping in metadata
  - Describes how fields in the database correspond to fields in in-memory objects.



Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Metadata Mapping II

- *Code generation* - a program whose input is the metadata and whose output is the source code of classes that do the mapping
  - Any changes to the mapping require recompiling and redeploying
- *Reflective program* - treats methods and fields as data; the reflective program can read in field and method names from a metadata file and use them to carry out the mapping
  - Suffers in speed
  - It often causes code that is hard to debug
  - Even so, reflection is actually quite appropriate for database mapping

Hibernate Configuration File

File hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD
    3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property name="hibernate.connection.driver_class">
                com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">
                Jdbc:mysql://localhost/bid</property>
        <property name="connection.username">root</property>
        <property name="connection.password"></property>
…
        <!-- List of XML mapping files -->
                <mapping resource="Product.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Product.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
  <class name="test.hibernate.Product  " table="products">
    <id name="id" type="string”> <column name="id" sql-type="char(32)“  not-null="true"/>
    </id>
    <property name="name"> <column name="name" sql-type="char(255)"
      not-null="true"/>
     </property>
    <property name="price"> <column name="price" sql-type="double“  not-null="true"/>
    </property>
    <property name="amount"> <column name="amount" sql-type="integer“
      not-null="true"/>
    </property>
 </class>
</hibernate-mapping>
```

Execution Contexts

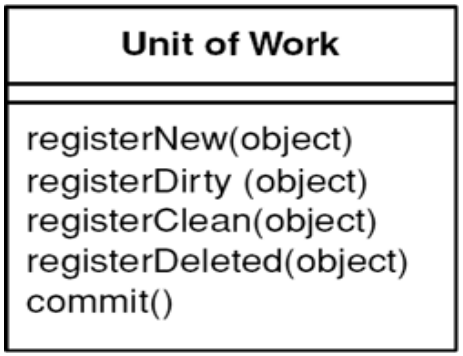
- *Request* - corresponds to a single call from the outside world which the software works on and for which it optionally sends back a response.
- *Session* - a long-running interaction between a client and a server. It may consists of a single request, but more commonly it consists of a series of requests that the user regards as a consistent logical sequence.
- *Transaction* - pulls together several requests that the client wants treated as if they were a single request. It can occur from the application to the database (a system transaction) or from the user to an application (a business transaction).

Stateless/Stateful

- Stateless server is an object that does not retain state between requests
  - Statelessness allows pooling objects so that fewer objects are needed to handle more users
  - Statelessness also fits in well with the Web since HTTP is a stateless protocol
- But … many client interactions are inherently stateful
  - Three possibilities for storing session state
    - **Client Session State**
      - Stores session state on the client; allows the server to be completely stateless
      - Unreasonable with large amount of data
    - **Server Session State**
      - Keeps the state on a server system in memory or in a serialized form
    - **Database Session State**
      - Stores session data as committed data in a database
        - When a request arrives, the server object first pulls the data required from the database. Then it does the work and saves back to the database all the data

The Behavioural Problem

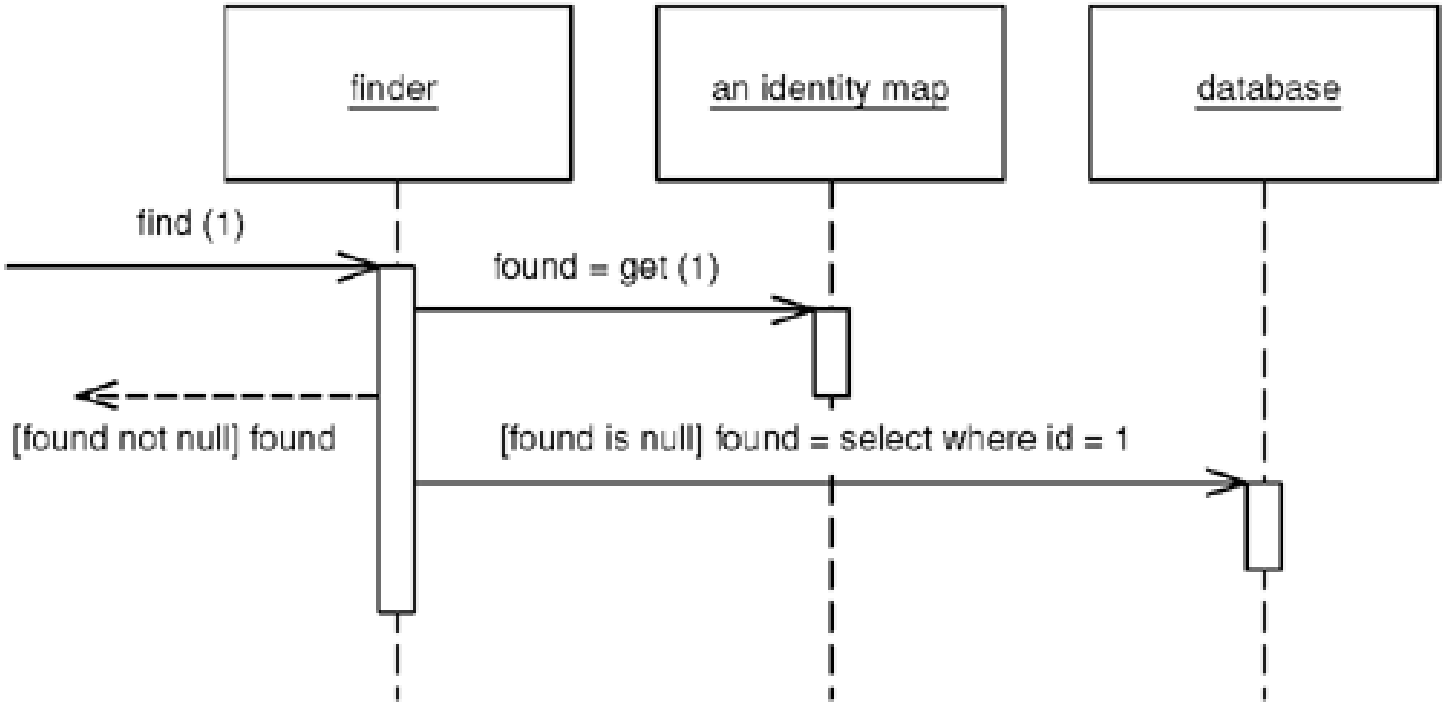
- **Unit of Work**
  - Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems
    - Changing the database with each change to the object model can lead to lots of very small database calls; furthermore it requires to have a transaction open for the whole interaction.



Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

The Behavioural Problem (II)

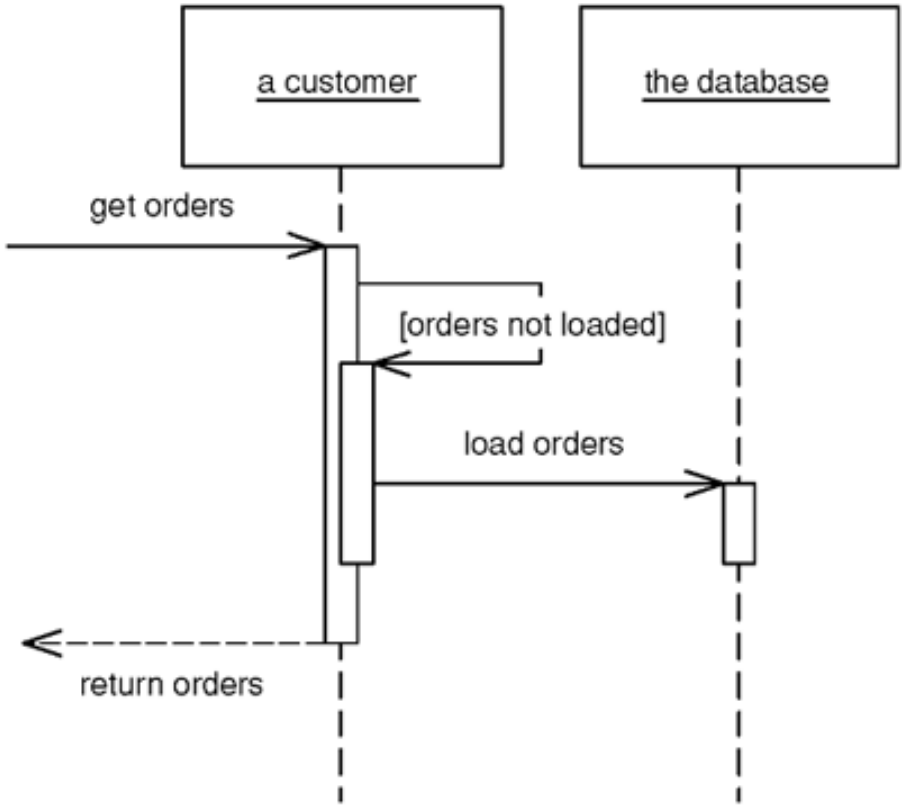
- **Identity Map**
  - Ensures that each object gets loaded only once by keeping a record of every loaded object in a map. Looks up objects using the map when referring to them



Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

The Behavioural Problem (III)

- **Lazy Load**
  - An object that does not contain all of the data you need but knows how to get it
  - Reason:
    - Things are usually arranged so that linked objects are loaded together
    - But with many objects connected together any read of any object can pull an enormous object graph out of the database.
    - To avoid such inefficiencies you need to reduce what you bring back yet still keep the door open to pull back more data if you need it later on



Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

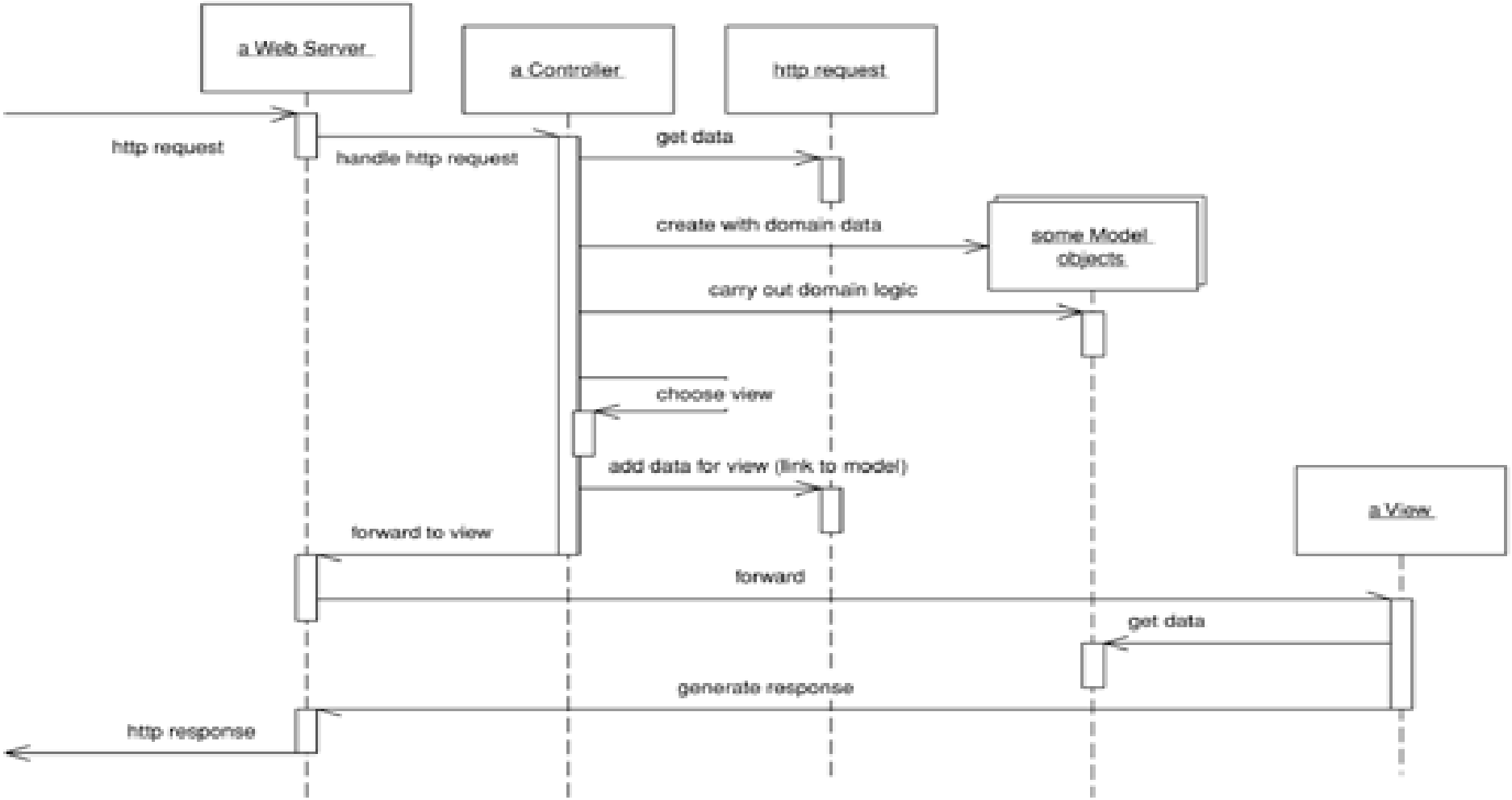
Reading in Data

- Finder methods that wrap SQL select statements
  - e.g. find(id), findForCustomer(customer), …
  - Where to put the finder methods depends on the interfacing pattern used
- Performance issues:
  - It is better to pull back multiple rows at once, i.e. too much data than too little
  - One can use joins to pull multiple tables back with a single query
    - e.g. a Gateway that gets data from multiple joined tables, or a Data Mapper that loads several domain objects with a single call
  - In all cases, it is wise to profile the application with respect to the specific database and data
    - Database systems and application servers often have sophisticated caching schemes

Database Connections

- Database connection object acts as the link between application code and database
  - Connection must be opened before you can execute commands
    - Queries return a *Record Set*
      - Some interfaces provide for disconnected Record Sets, which can be manipulated after the connection is closed
  - Often it is expensive to create a connection, which makes it worthwhile to create a connection pool
    - Developers request a connection from the pool and release it
    - Pooling is often put behind an interface that looks like creating a new connection
  - There are two choices in order to have a connection reference:
    - To pass the connection around as an explicit parameter (heavy solution)
    - To use a *Registry* (or better a thread-scoped Registry); a well-known object that other objects can use to find common objects and services

Web Presentation – Model View Controller

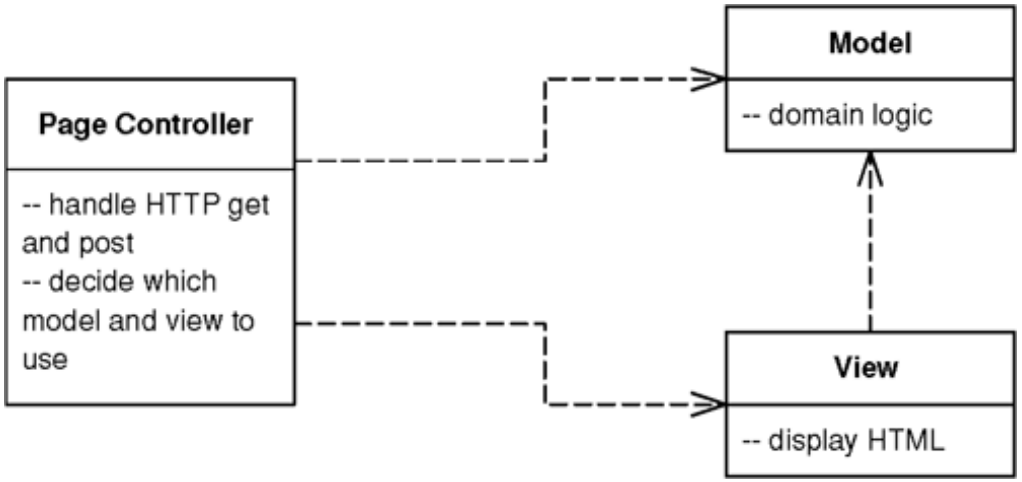


- The controller handles the request, gets the model to do the domain logic, gets the view to create a response based on the model

Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Input Controller Patterns

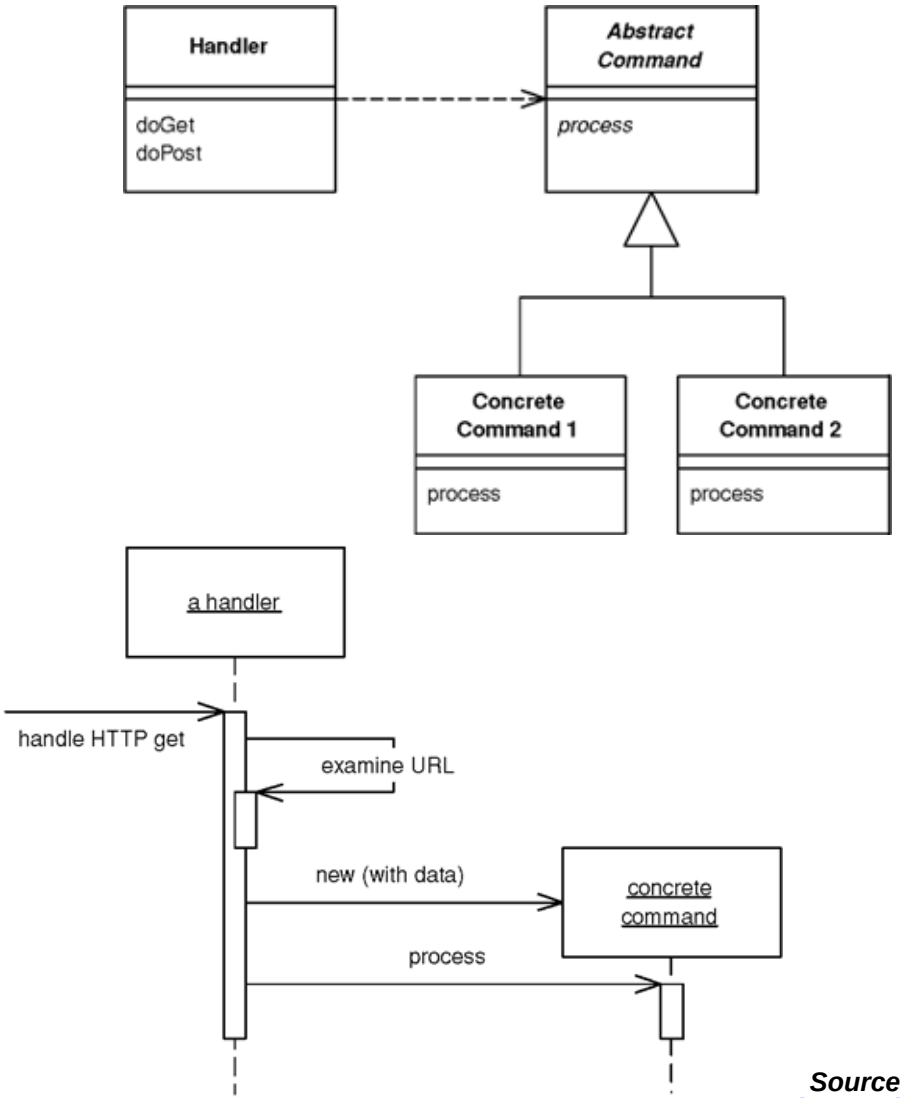
- **Page Controller**
  - An object that handles a request for a specific page or action on a Web site
    - One input controller for each logical page of the Web site
      - May be the page itself, as in server page environments, or a separate object that corresponds to that page
      - Works particularly well if most of the controller logic is pretty simple



Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Input Controller Patterns (II)

- **Front Controller**
  - Handles all requests for a Web site
    - Carries out common behavior: it allows factoring out code that is otherwise duplicated
    - Centralizes all HTTP handling within a single object, avoiding the need to reconfigure the Web server whenever the action structure of the site changes
    - Structured in two parts: a Web handler and a command hierarchy



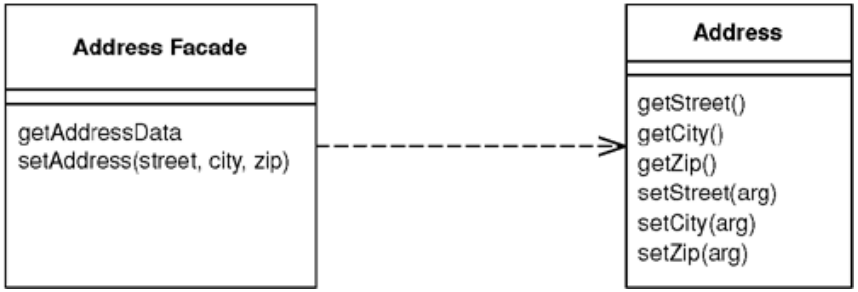
Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

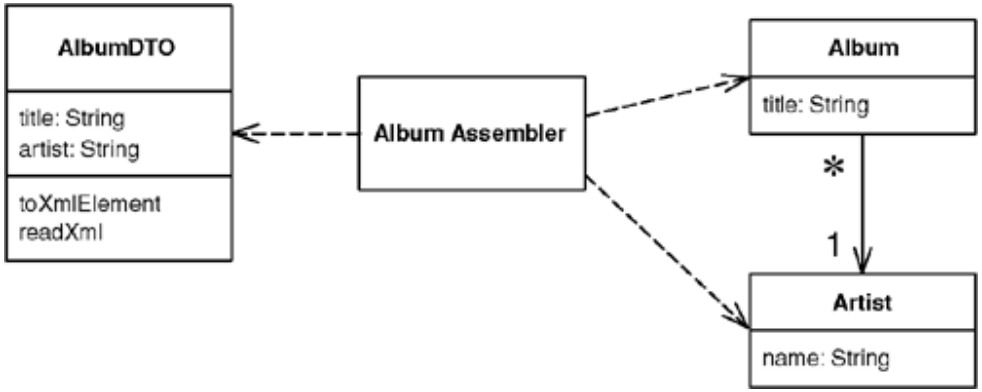
Distributed Objects – Few Remarks

- Remote and local interfaces
  - A procedure call within a process is very, very fast.
  - A procedure call between two separate processes is orders of magnitude slower
    - For processes running on different machines, it is necessary to add another order of magnitude or two, depending on the network topography involved
  - The interface for an object to be used remotely must be different from that for an object used locally within the same process
    - A local interface is best as a fine-grained interface
      - Taken an address class, a good interface will have separate methods for getting the city, getting the state, setting the city, …
      - A fine-grained interface is good because it follows the general OO principle of lots of little pieces that can be combined and overridden in various ways to extend the design into the future
    - In the remote case, the interface should be coarse-grained, designed not for flexibility and extendibility but for minimizing calls
      - In the previous example, we will have get-address details and update-address details

Distributed Objects (II)

- Working with the distribution boundary
  - **Remote Facade**
    - Provides a coarse-grained facade on fine-grained objects to improve efficiency over a network
  - **Data Transfer Object**
    - Carries data between processes in order to reduce the number of method calls
    - An assembler may be used on the server side to transfer data between the DTO and any domain objects
    - Usually responsible for serializing itself into some format that will go over the net





Source :
Fowler Martin
Patterns of Enterprise Application Architecture
Addison Wesley

Interfaces for Distribution

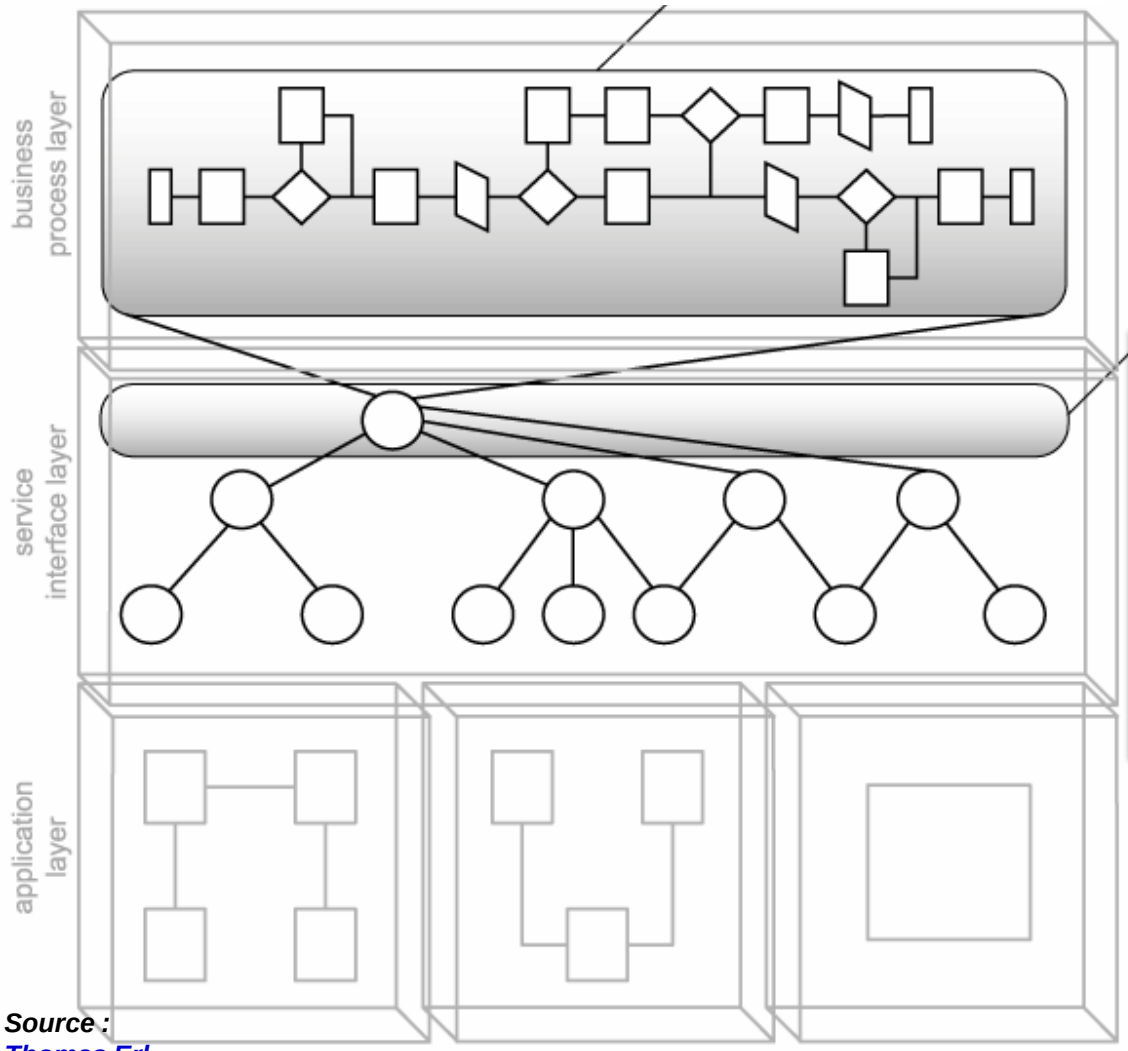
- Traditionally, interfaces for distributed components have been based on synchronous RPCs (global procedures or methods on objects); in recent times on “message passing”
- In the last years, interfaces based on XML over HTTP have emerged (e.g. SOAP)
  - Can be a synchronous RPC-based or a message-based approach (inherently asynchronous)
  - XML is a common format with parsers available in many platforms
  - But … moving all the transferred data into XML structures and strings can add a considerable burden
- Web services are about application integration (interoperability) rather than application construction
  - The approach is not to break up a single application into Web services that talk to each other
  - Rather, build applications and expose various parts of it as Web services, treating those Web services as Remote Facades

Enterprise Application Integration

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Enterprise Application Integration



- *Business Process Modelling*
  - *UML Activity Diagram*
  - *BPMN*
  - *WSBPEL*
  - *Process Integration Languages*
- *Integration issues and "traditional" approaches*
- *Service-Oriented Architecture*
  - *Service-Oriented paradigm*
  - *Web Services*
    - *Core Web Services Standards*
  - *Semantic Web Services: SAWSDL*
- *Enterprise Applications*
  - *Overview*
  - *Architectural solutions*
    - *Patterns of Enterprise Application Architecture*

Source :
Thomas Erl
Service-Oriented Architecture: Concepts, Technology, and Design

Enterprise Integration

- Users want to execute business functions that span multiple applications
- Enterprise integration is the task of making disparate applications work together
  - To support common business processes and data sharing across applications
- Challenges:
  - Applications can run on multiple computers, which may represent multiple platforms, and may be geographically dispersed
    - Networks are slow and unreliable
  - Applications may run outside of the enterprise by business partners or customers
    - Applications are quite dissimilar
    - Changes are inevitable

Enterprise Integration (II)

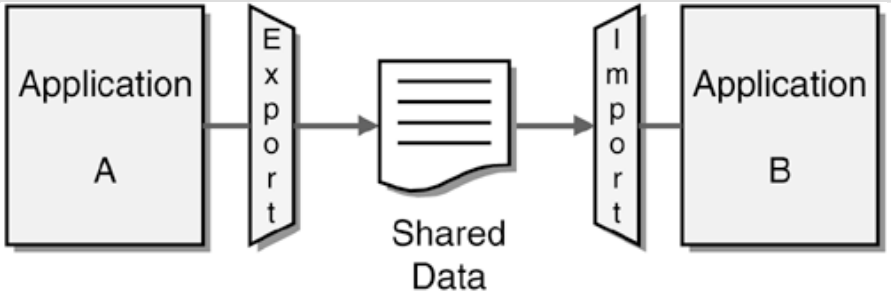
- Criteria that should be considered when choosing and designing an integration approach
  - Application coupling — minimize dependencies
  - Intrusiveness — when integrating, minimize both changes to the application and the amount of integration code needed
  - Technology selection — choose thoroughly the suitable tools
  - Data format — agreement on the format of the data exchanged. Possibly an intermediate translator can unify applications that insist on different data formats
  - Data timeliness — minimize the length of time of data exchange
  - Data or functionality — Many integration solutions allow applications to share not only data but functionality as well
  - Remote Communication — The choice is between synchronous or asynchronous
  - Reliability — remote applications may not be running or the network may be temporarily unavailable

Enterprise Integration (III)

- No one integration approach addresses all criteria equally well
- Four main integration styles; the order reflects an increasing order of sophistication, but also increasing complexity:
  - File Transfer — multiple applications share some files
  - Shared Database — multiple applications share the same database schema, located in a single physical database
  - Remote Procedure Invocation — applications expose some of their functionality so that they can be invoked remotely
    - The communication is synchronous
  - Messaging — applications publish messages by using a common messaging system
    - Agreement on the message format. The communication is asynchronous
- Each style has its advantages and disadvantages

→ choose the best style for a particular integration opportunity

File Transfer (70s)



- The simplest approach for data sharing
  - Good physical decoupling
    - Applications need to agree on the filename and location
  - Language and system independent
    - An important decision is what format to use
      - The current method is to use XML
      - Readers, writers, and transformation tools have built up around formats
  - No extra tools or integration packages are needed, but … developers have to do a lot of the work themselves
- Drawbacks
  - A certain amount of effort is required to produce and process files
    - Usually, regular business cycles drive the decision: nightly, weekly, quarterly, …
  - Systems can get out of synchronization
    - Timing of when it will be written and read, and who will delete the file
  - Possible semantic dissonance

Source :
G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

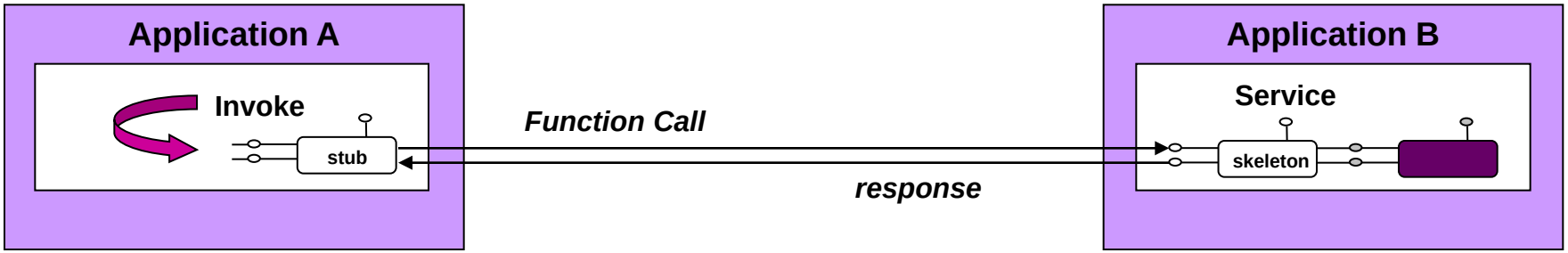
Shared DataBase (80s)

- All applications access a common database
  - Made easier by the widespread use of SQL-based relational databases
  - Consistent data
- Drawbacks
  - Integration of data instead of business functions
    - Unencapsulated data makes more difficult to maintain a family of integrated applications
      - Changes in any application may trigger a change in the database
    - Organizations using shared database are often very reluctant to change the database, which means that the application development work is much less responsive to the changing needs of the business
  - Difficult to find a common representation
    - A unified schema that should meet the needs of multiple applications
  - Database may become a performance bottleneck

Remote Procedure Call (90s)

- The most common model for communications in distributed applications
  - The basic model for RPC was described by Birrell & Nelson in 1980
    - Based on work done at Xerox PARC
  - Each application seen as a large-scale object or component with encapsulated data
- **Goal**
  - Make RPC look as much like local PC as possible
    - Details of remote communication largely hidden from programmer
- There are 3 components on each side:
  - A user program (client or server)
  - A set of *stub* procedures
  - RPC runtime support

Remote Procedure Call (II)



- An ***RPC*** mechanism allows a client to invoke the methods of a remote service using a familiar local procedure call paradigm
  - *On the client*, the RPC infrastructure
    - Converts the local procedure call arguments into some standard request representation
    - Communicates the request to the remote service
    - Converts any response back into procedure call return values
  - *On the server*, the RPC infrastructure
    - Converts incoming requests into local procedure calls
    - Converts the result of local procedure calls into responses
    - Communicates responses to the client
- XML based RPC mechanisms use XML for the representation of RPC requests and responses
- RPC relies on a stub compiler to automatically produce client/server stubs

Remote Procedure Call (III)

- Advantages
  - Data exchanged only as needed
  - Integration of business function
- Weaknesses
  - Performance and reliability
    - Latency
    - Lack of control over other systems
  - Applications tightly coupled as a local method call

Remarks about Coupling

- Coupling is a measure of dependency between applications
  - Technology Dependency
  - Location Dependency
  - Temporal Dependency
  - Data Format Dependency
- Coupling is not inherently good or bad
- Tightly Coupled Systems
  - Make many assumptions about each other
  - Well suited for internal communication inside of an application
  - Well suited for “near” communication where we have control over both sides of the interaction
  - Generally more efficient, easier to develop and debug

Simple EAI Example

- *Requirements*: to build an online banking system that allows customers to deposit money into their account from another bank. To perform this function, the front-end Web application has to be integrated with the back-end financial system that manages fund transfers

```
String hostName = "finance.bank.com";
int port = 80;

IPHostEntry hostInfo = Dns.GetHostByName(hostName);
IPAddress address = hostInfo.AddressList[0];
IPEndPoint endpoint = new IPEndPoint(address, port);
//connect the two systems through the TCP/IP protocol
Socket socket = new Socket(address.AddressFamily,
                           SocketType.Stream, ProtocolType.Tcp);
socket.Connect(endpoint);
byte[] amount = BitConverter.GetBytes(1000);
byte[] name = Encoding.ASCII.GetBytes("Joe"); //only the person's name is required
int bytesSent = socket.Send(amount);
bytesSent += socket.Send(name);
socket.Close();
```

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Simple EAI Example (II)

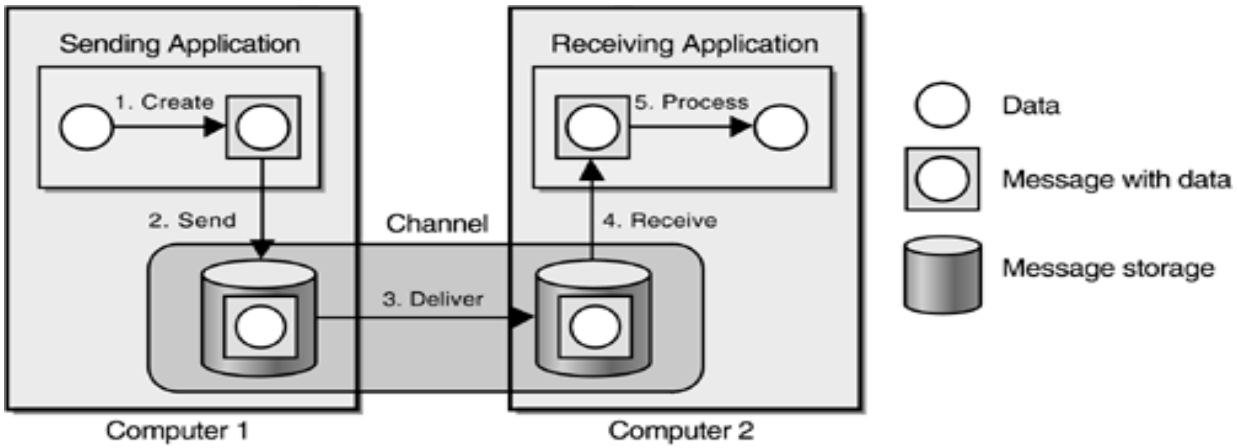
- Dependencies:
  - Platform technology — internal representations of numbers and objects
  - Location — hardcoded machine addresses
  - Time — all components have to be available at the same time
  - Data format — the list of parameters and their types must match

Source :

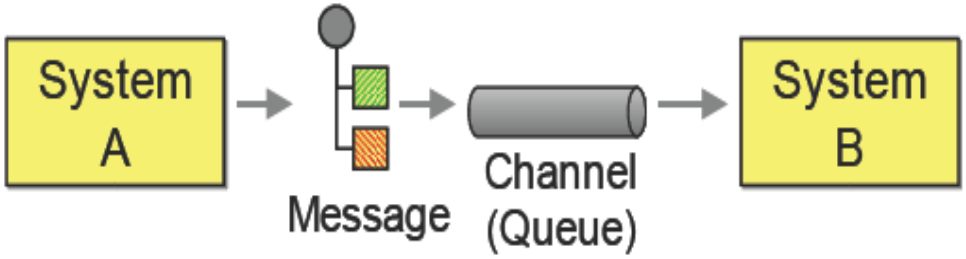
G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Messaging Architectures

- Messaging is a technology that enables high-speed, asynchronous, program-to-program communication with reliable delivery
- The message is some sort of data structure
  - Can be interpreted as data, as the description of a command to be invoked on the receiver, or as the description of an event that occurred in the sender
  - Contains two parts, a header and a body
    - The header contains meta-information about the message, used by the messaging system
    - The body contains the application data being transmitted
- Messaging capabilities are typically provided by a separate software system called a messaging system or message-oriented middleware (MOM).



Message-Oriented Communication



- Systems communicate via *Channels,* aka queues
  - Channels are logical pathways and have logical (location-independent) addresses
- Data is encapsulated in messages in a technology neutral format, e.g. XML
- The sending application places a message into the Channel and goes on to other work ("fire-and-forget")
- The Channel queues the data until the receiving application is ready to consume it (FIFO)

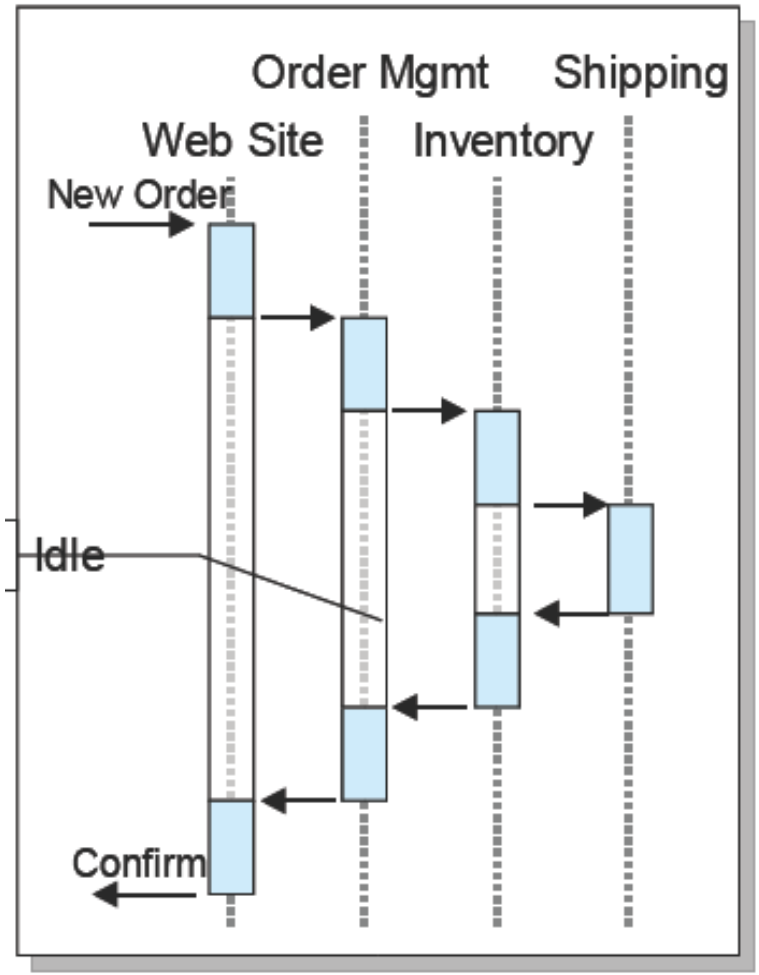
Source :
G. Hohpe
Enterprise Integration Patterns
2004 JavaOne[SM] Conference

Message-Oriented Communication (II)

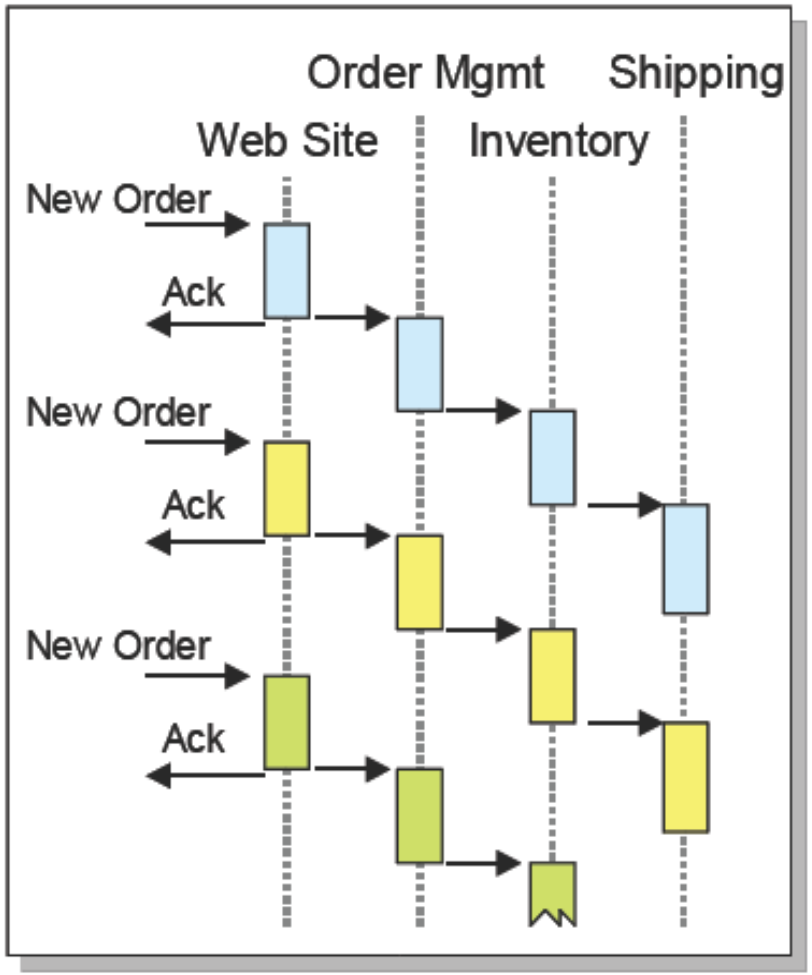
- Provides loose coupling and reliability
  - Channels are separate from applications
  - Channels are asynchronous and reliable (queues)
  - Use a common data representation, e.g. XML
  - Messages are self-contained
  - Removes location dependencies
  - Removes temporal dependencies
  - Removes technology dependencies
  - Removes data format dependencies

Source :
G. Hohpe
Enterprise Integration Patterns
2004 JavaOne^{SM} Conference

Thinking Asynchronously



Synchronous
(Call Stack)



Asynchronous
(Pipeline)

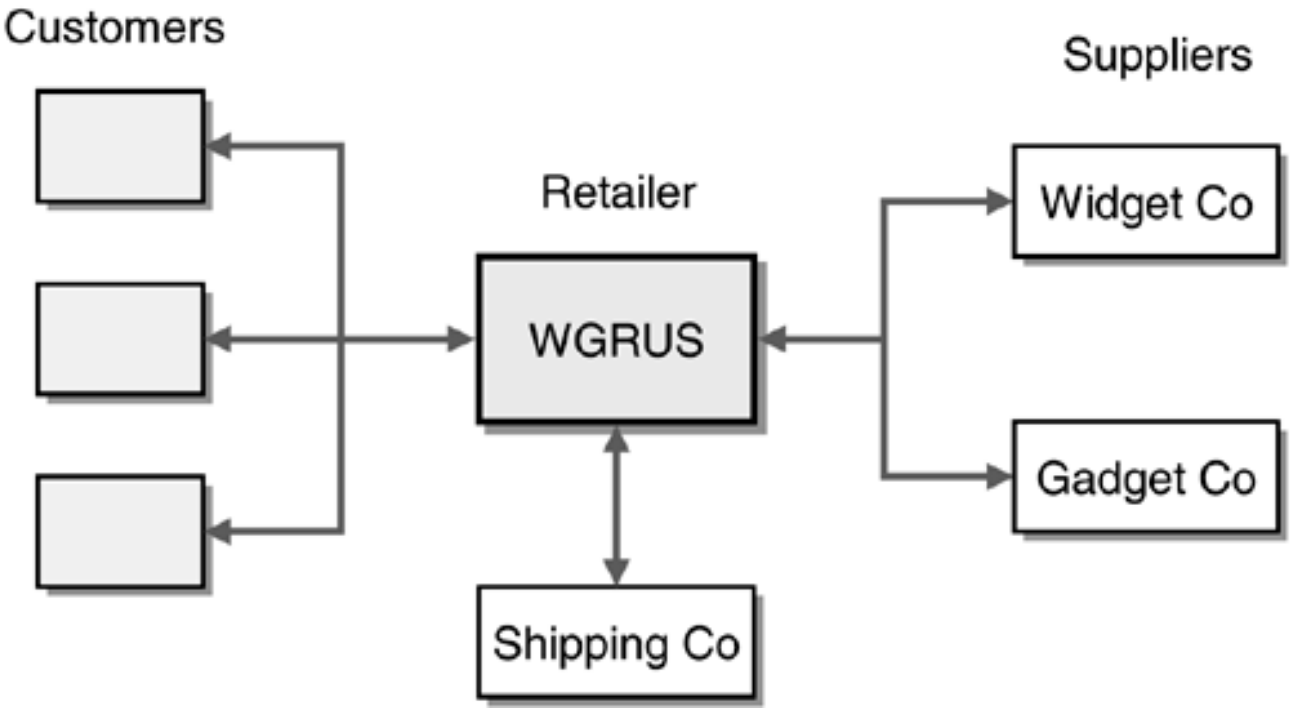
Source :
G. Hohpe
Enterprise Integration Patterns
2004 JavaOne℠ Conference

Fewer Assumptions...

- This means that more is left to do for the developer
  - Correlation of related messages (e.g. request and response)
  - Maintaining state
  - Determining order of events
    - To re-establish the message sequence
  - Complex programming model
    - Figuring out what to do next …

Source :
G. Hohpe
Enterprise Integration Patterns
2004 JavaOne℠ Conference

Widgets & Gadgets 'R Us: An Example



An online retailer that buys widgets and gadgets from manufacturers and resells them to customers

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

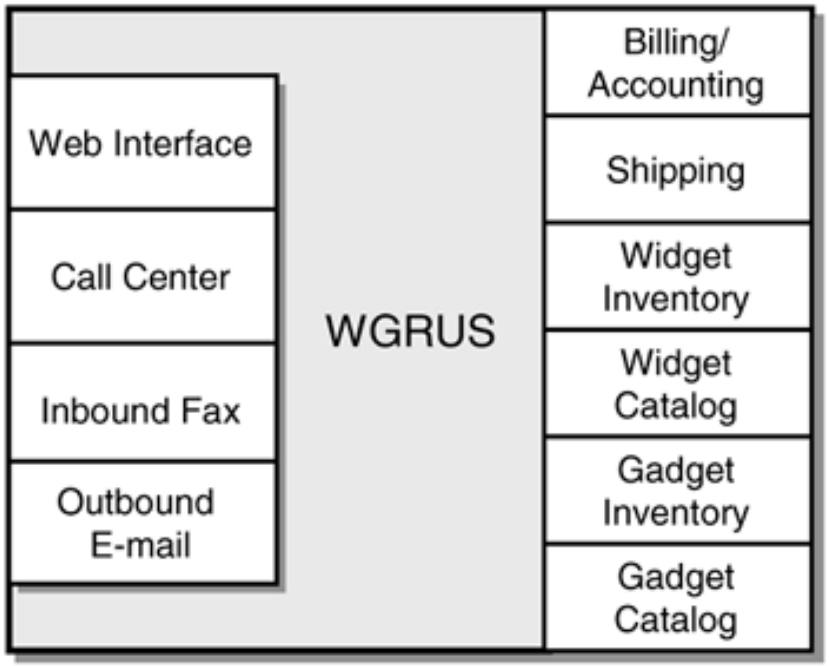
Widgets & Gadgets 'R Us (II)

- Requirements:
  - Take Orders: customers can place orders via Web, phone, or fax
  - Process Orders: processing an order involves multiple steps: verifying inventory, shipping the goods, and invoicing the customer
  - Check Status: customers can check the order status
  - New Catalog: suppliers update their catalog periodically. WGRUS needs to update pricing and availability based on the new catalogs
  - Change Address: customers can use a Web front-end to change their billing and shipping address
  - Announcements: customers can subscribe to selective announcements from WGRUS
  - Testing and Monitoring: operations staff needs to be able to monitor all individual components and the message flow between them

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Widgets & Gadgets 'R Us (III)



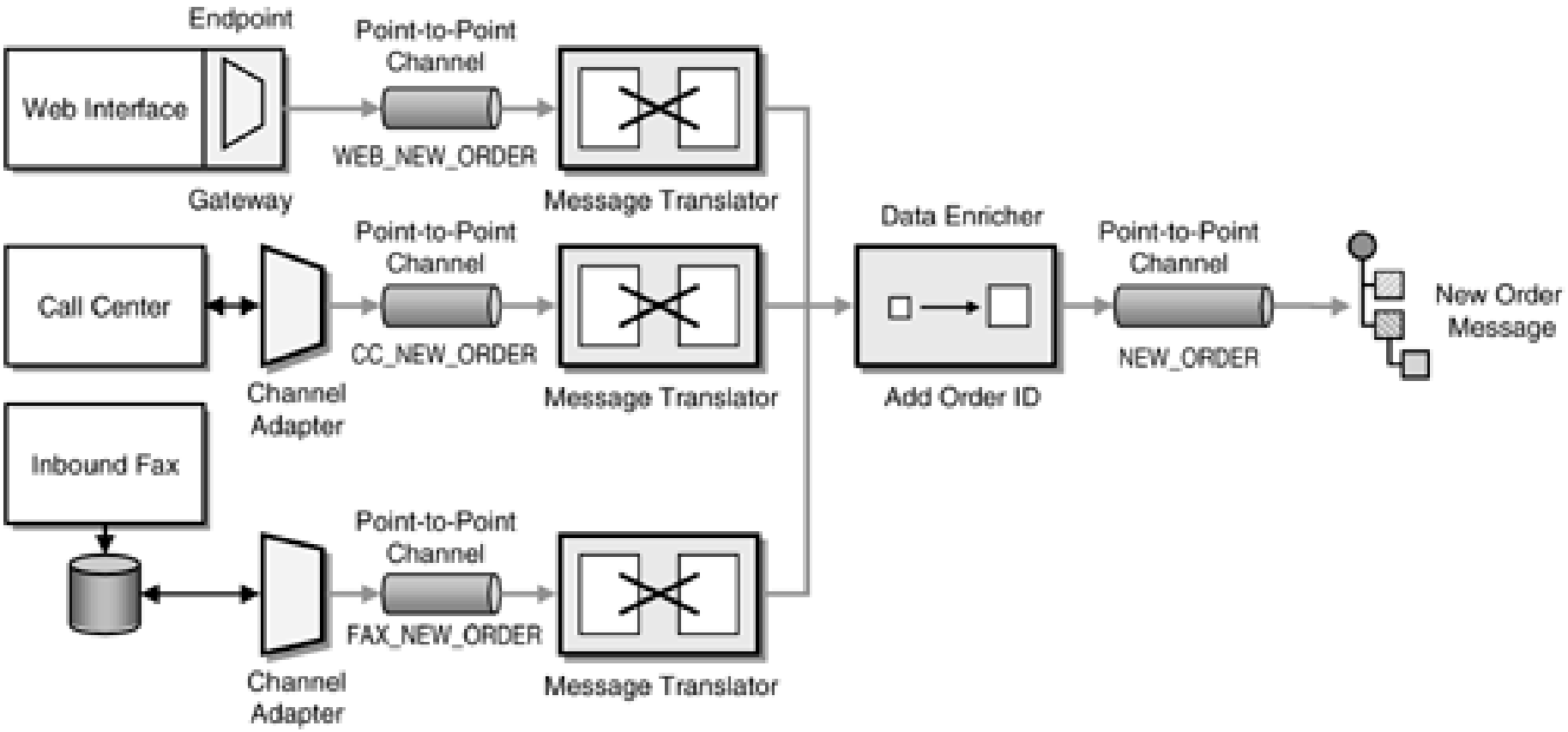
WGRUS IT Infrastructure

- The processing of orders is a typical implementation of a distributed business process

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Taking Orders

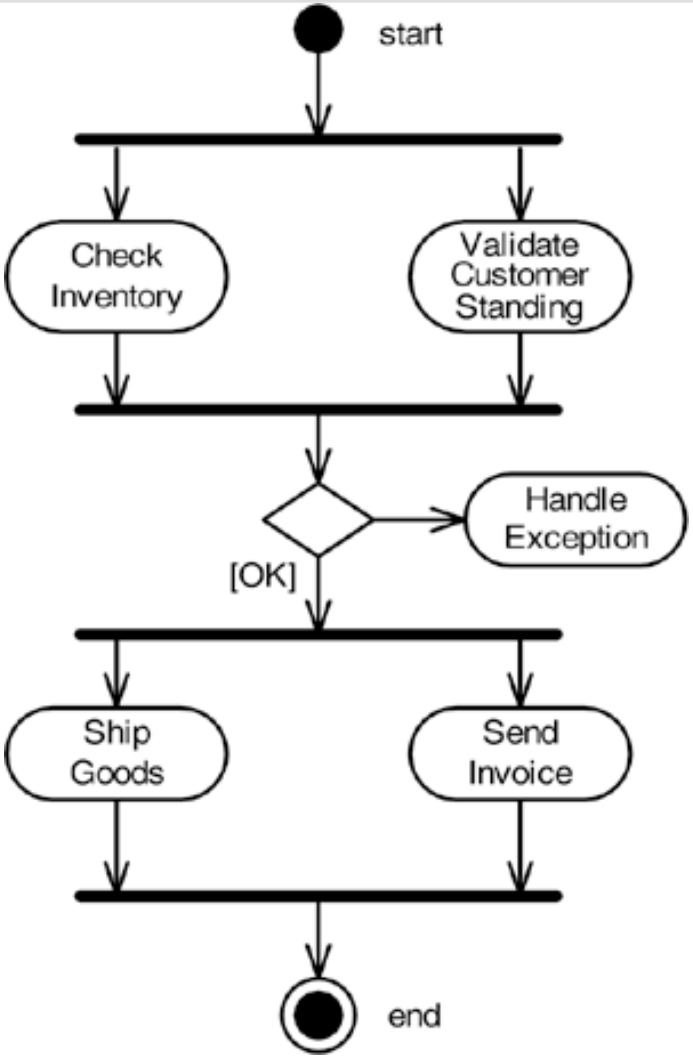


A message-oriented middleware solution to streamline the order entry process (pipes+filters)

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Processing Orders

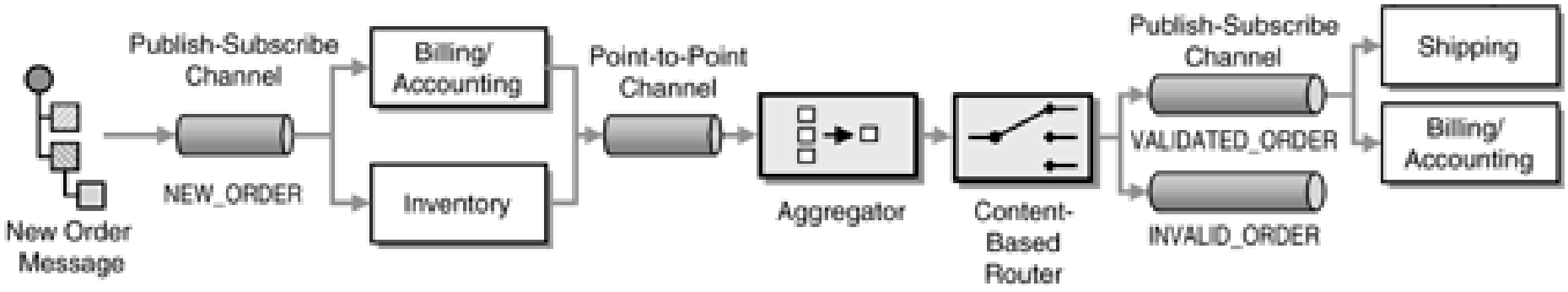


Activity Diagram for Order Processing

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Processing Orders (II)

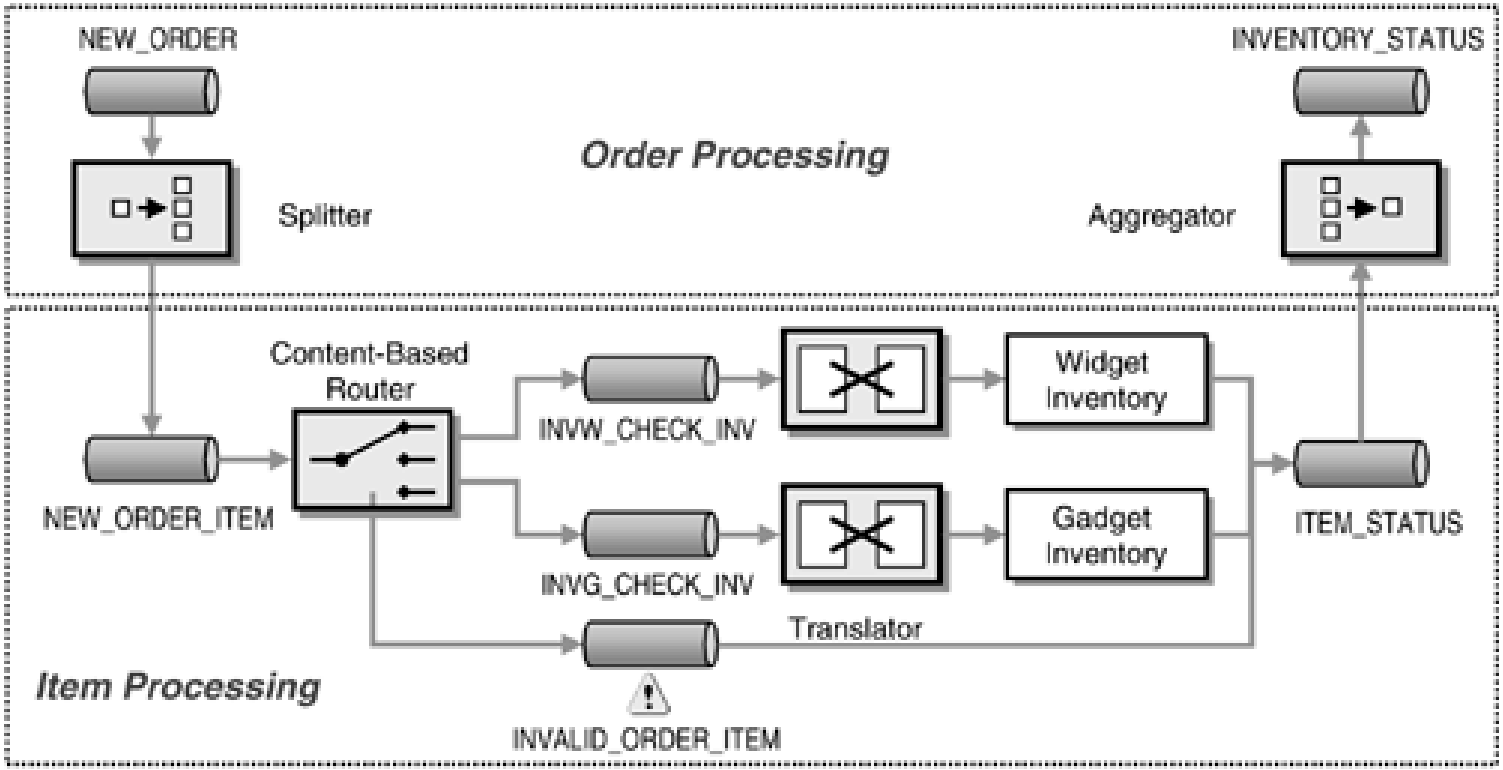


- A *Publish-Subscribe Channel* is used to implement the fork action and an *Aggregator* to implement the join action
- *Content-Based Router* is a component that consumes a message and publishes it, unmodified, to a choice of other channels based on rules coded inside the router

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Processing Orders – Inventory Function

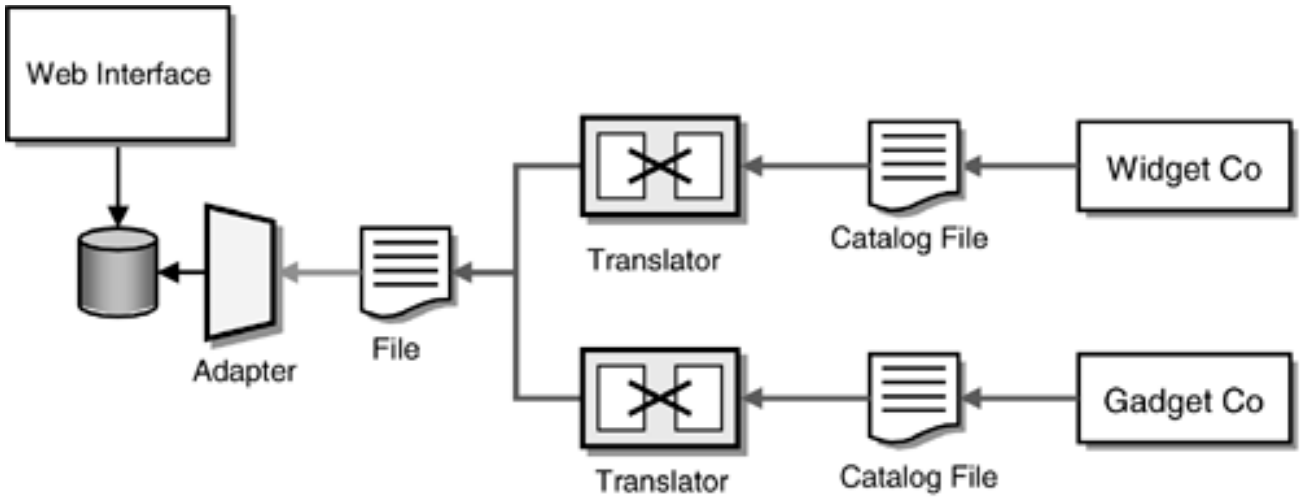


- Processing Order Items Individually
  - A *Splitter*, breaks a single message into multiple individual messages
  - An *Aggregator* concatenates all Order Item messages (with the same order ID) back into a single Order message

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

New Catalog

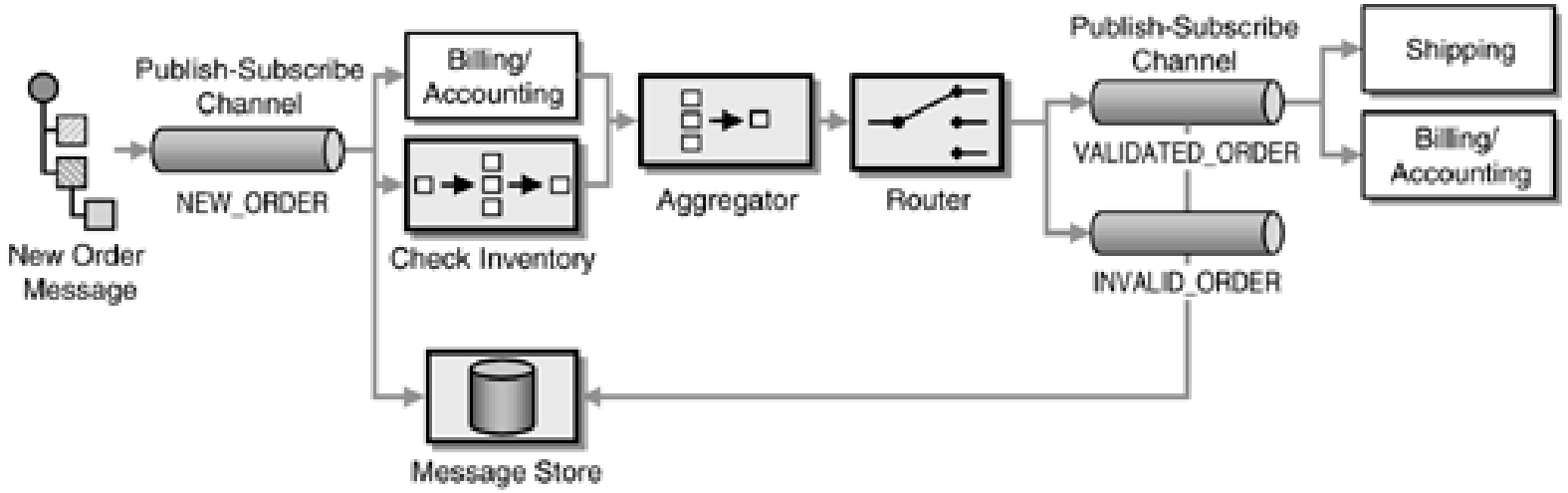


- Both suppliers update their product catalog once every three months
  - It makes relatively little sense to create a real-time messaging infrastructure to propagate catalog changes
- The choice is to use *File Transfer* integration to move catalog data from suppliers to WGRUS
  - Another advantage is that files are easily and efficiently transported across public networks using FTP or similar protocols
  - A database Channel Adapter is used to update the data in the relational database

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

... adding a Message Store

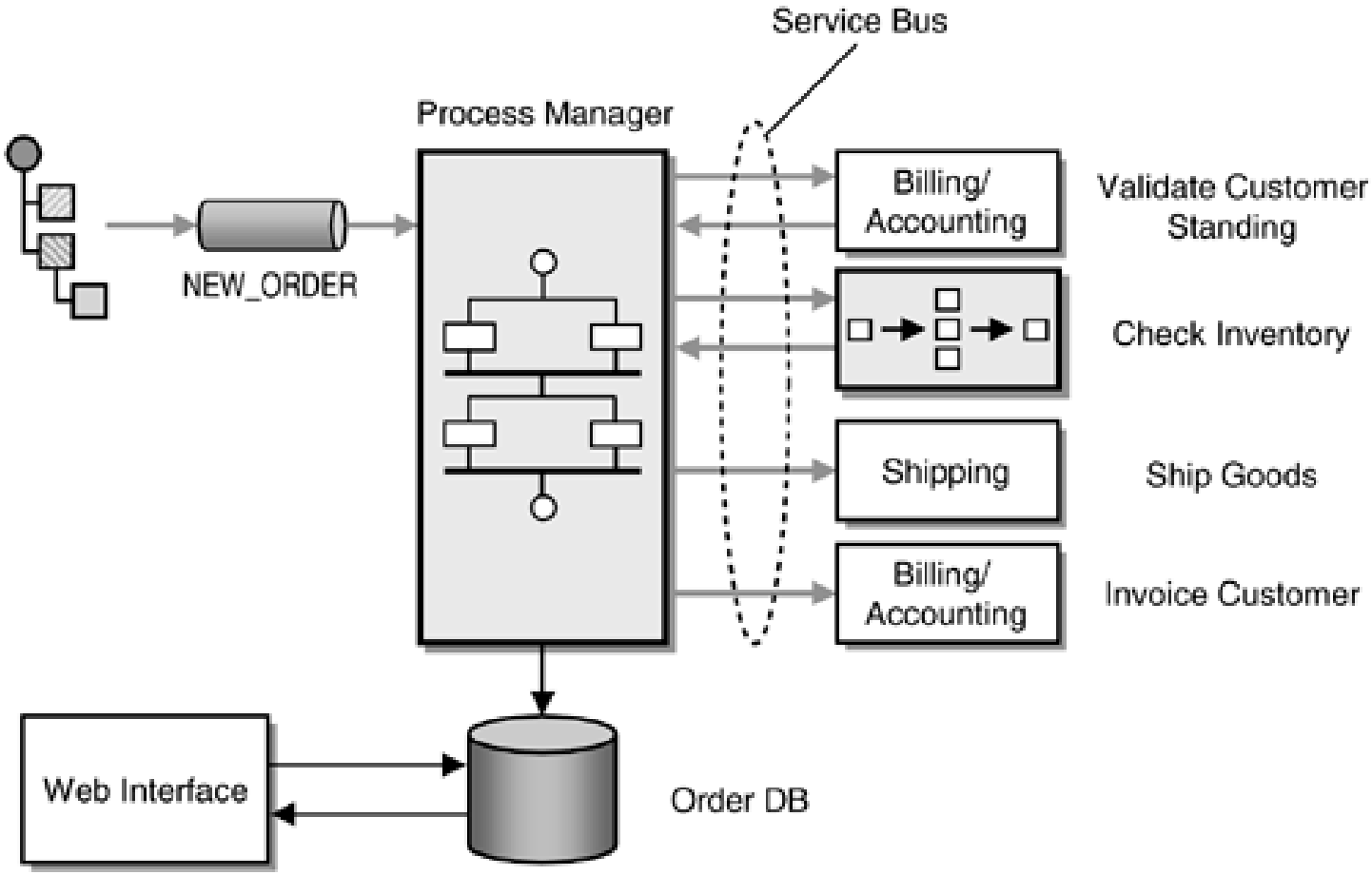


- To find out the status of an order in a sequence of steps, it could be useful to know the "last" message related to the order
  - One of the advantages of a Publish-Subscribe Channel is that additional subscribers can added without disturbing the flow of messages
    - e.g. to listen to new and validated orders and store them in a Message Store
    - The Message Store database can be queried for the status of an order

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Process Manager



- Individual systems are turned into shared business functions that can be accessed by other components as services, thus increasing reuse and simplifying maintenance

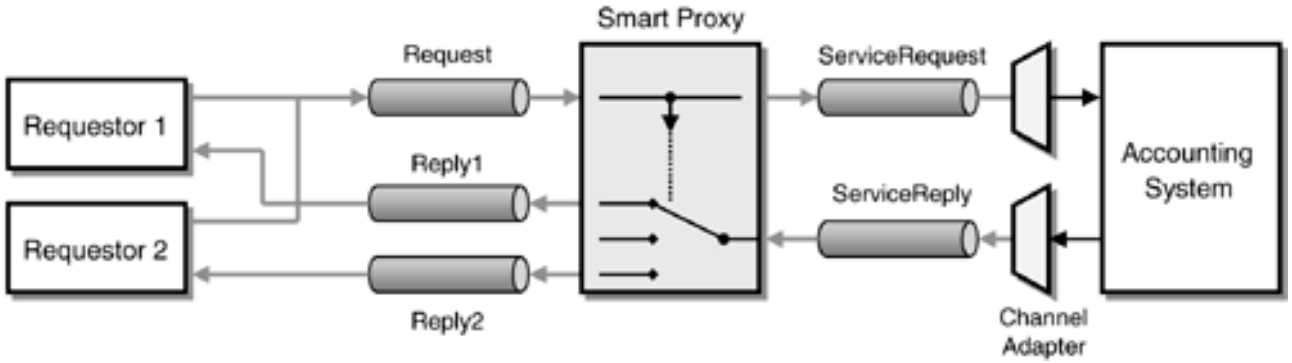
Source :
G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Process Manager (II)

- The services can be wired together via a message flow **or** orchestrated via a Process Manager
  - The *Process Manager* provides two main functions:
    - Persistent store, to store data between messages (inside a "process instance")
    - Keeping track of progress and determining the next step (by using a workflow)
- The Web interface queries the status of an order (checking status is a synchronous process) accessing the order database directly
  - A form of Shared Database

- To turn the WGRUS IT infrastructure into an SOA it is necessary to add facilities to look up ("*discover*") a service from a central registry
  - In order to participate in this SOA, each service would have to provide additional functions
    - e.g. each service would have to expose an interface contract that describes the functions provided by the service
    - Each request-reply service also needs to support the concept of a Return Address, which allows the caller (the service consumer) to specify the channel where the service should send the reply message
      - Important to allow the service to be reused in different contexts

Smart Proxy to Interface Legacy System



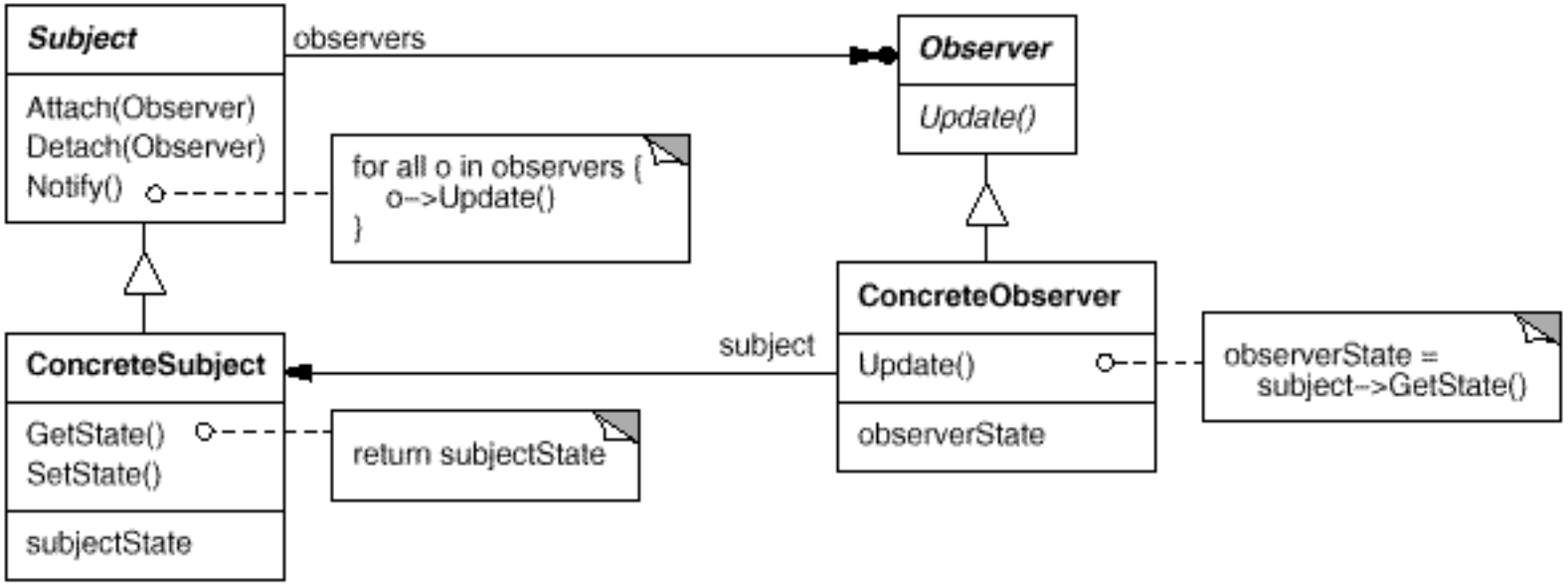
- The *Smart Proxy* enhances the basic system service with additional capability so that it can participate in an SOA
  - It intercepts both request and reply messages to and from the basic service

Source :

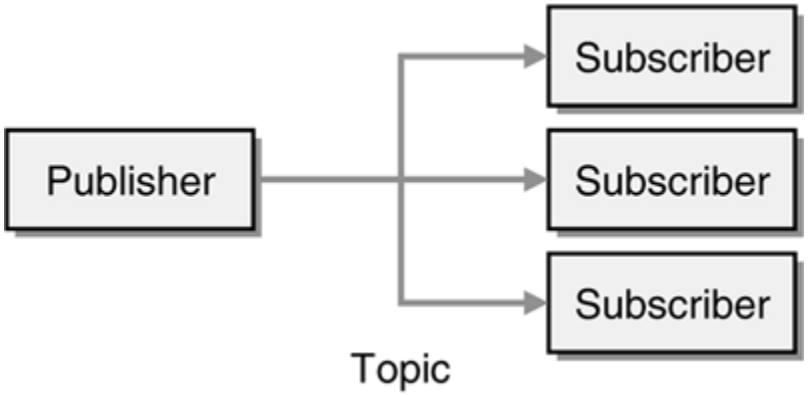
G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley

Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



Publish-Subscribe Channel



- **Distributed Observer Pattern**
  - A ***Publish-Subscribe Channel*** implements the Observer pattern in a distributed environment. The pattern is implemented in three steps.
    - The messaging system administrator creates a Publish-Subscribe Channel (represented in Java applications as a JMS Topic)
    - The application acting as the subject creates a TopicPublisher (a type of MessageProducer) to send messages on the channel
    - Each of the applications acting as an observer creates a TopicSubscriber (a type of MessageConsumer) to receive messages on the channel (analogous to calling the Attach(Observer) method in the Observer pattern)

Source :

G. Hohpe, B. Woolf
Enterprise Integration Patterns
Addison Wesley