

CONTENTS INCLUDE:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Observer
- Template Method and more...

ABOUT DESIGN PATTERNS

This Design Patterns refcard provides a quick reference to the original 23 Gang of Four design patterns, as listed in the book *Design Patterns: Elements of Reusable Object-Oriented Software*. Each pattern includes class diagrams, explanation, usage information, and a real world example.

- C **Creational Patterns:** Used to construct objects such that they can be decoupled from their implementing system.
- S **Structural Patterns:** Used to form large object structures between many disparate objects.
- B **Behavioral Patterns:** Used to manage algorithms, relationships, and responsibilities between objects.

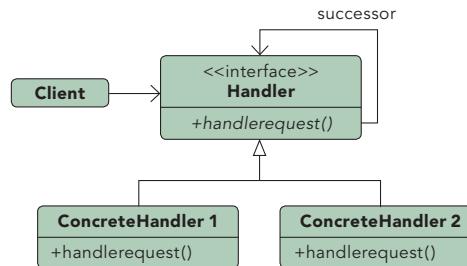
Object Scope: Deals with object relationships that can be changed at runtime.

Class Scope: Deals with class relationships that can be changed at compile time.

C Abstract Factory	S Decorator	C Prototype
S Adapter	S Facade	S Proxy
S Bridge	C Factory Method	B Observer
C Builder	S Flyweight	C Singleton
B Chain of Responsibility	B Interpreter	B State
B Command	B Iterator	B Strategy
S Composite	B Mediator	B Template Method
	B Memento	B Visitor

CHAIN OF RESPONSIBILITY

Object Behavioral

**Purpose**

Gives more than one object an opportunity to handle a request by linking receiving objects together.

Use When

- Multiple objects may handle a request and the handler doesn't have to be a specific object.
- A set of objects should be able to handle a request with the handler determined at runtime.
- A request not being handled is an acceptable potential outcome.

Design Patterns

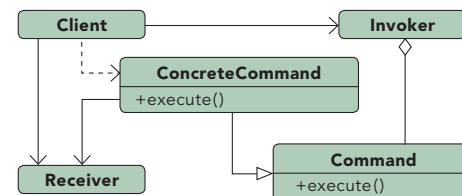
By Jason McDonald

Example

Exception handling in some languages implements this pattern. When an exception is thrown in a method the runtime checks to see if the method has a mechanism to handle the exception or if it should be passed up the call stack. When passed up the call stack the process repeats until code to handle the exception is encountered or until there are no more parent objects to hand the request to.

COMMAND

Object Behavioral

**Purpose**

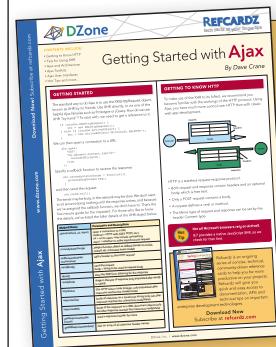
Encapsulates a request allowing it to be treated as an object. This allows the request to be handled in traditionally object based relationships such as queuing and callbacks.

Use When

- You need callback functionality.
- Requests need to be handled at variant times or in variant orders.
- A history of requests is needed.
- The invoker should be decoupled from the object handling the invocation.

Example

Job queues are widely used to facilitate the asynchronous processing of algorithms. By utilizing the command pattern the functionality to be executed can be given to a job queue for processing without any need for the queue to have knowledge of the actual implementation it is invoking. The command object that is enqueued implements its particular algorithm within the confines of the interface the queue is expecting.

**Get More Refcardz**

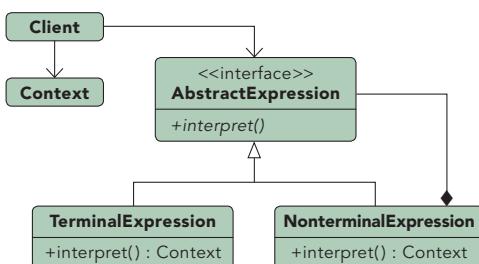
(They're free!)

- Authoritative content
- Designed for developers
- Written by top experts
- Latest tools & technologies
- Hot tips & examples
- Bonus content online
- New issue every 1-2 weeks

Subscribe Now for FREE!
Refcardz.com

INTERPRETER

Class Behavioral



Purpose

Defines a representation for a grammar as well as a mechanism to understand and act upon the grammar.

Use When

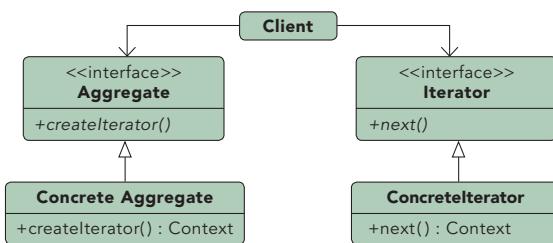
- There is grammar to interpret that can be represented as large syntax trees.
- The grammar is simple.
- Efficiency is not important.
- Decoupling grammar from underlying expressions is desired.

Example

Text based adventures, wildly popular in the 1980's, provide a good example of this. Many had simple commands, such as "step down" that allowed traversal of the game. These commands could be nested such that it altered their meaning. For example, "go in" would result in a different outcome than "go up". By creating a hierarchy of commands based upon the command and the qualifier (non-terminal and terminal expressions) the application could easily map many command variations to a relating tree of actions.

ITERATOR

Object Behavioral



Purpose

Allows for access to the elements of an aggregate object without allowing access to its underlying representation.

Use When

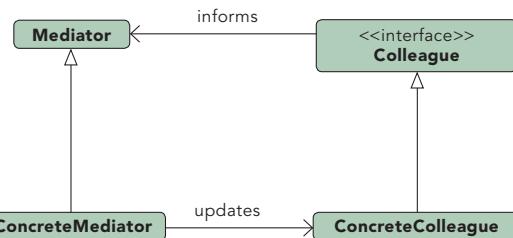
- Access to elements is needed without access to the entire representation.
- Multiple or concurrent traversals of the elements are needed.
- A uniform interface for traversal is needed.
- Subtle differences exist between the implementation details of various iterators.

Example

The Java implementation of the iterator pattern allows users to traverse various types of data sets without worrying about the underlying implementation of the collection. Since clients simply interact with the iterator interface, collections are left to define the appropriate iterator for themselves. Some will allow full access to the underlying data set while others may restrict certain functionalities, such as removing items.

MEDIATOR

Object Behavioral



Purpose

Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.

Use When

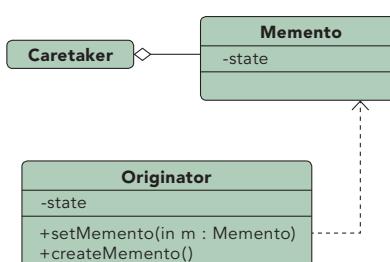
- Communication between sets of objects is well defined and complex.
- Too many relationships exist and common point of control or communication is needed.

Example

Mailing list software keeps track of who is signed up to the mailing list and provides a single point of access through which any one person can communicate with the entire list. Without a mediator implementation a person wanting to send a message to the group would have to constantly keep track of who was signed up and who was not. By implementing the mediator pattern the system is able to receive messages from any point then determine which recipients to forward the message on to, without the sender of the message having to be concerned with the actual recipient list.

MEMENTO

Object Behavioral



Purpose

Allows for capturing and externalizing an object's internal state so that it can be restored later, all without violating encapsulation.

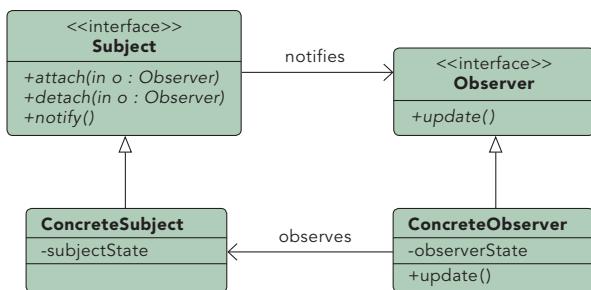
Use When

- The internal state of an object must be saved and restored at a later time.
- Internal state cannot be exposed by interfaces without exposing implementation.
- Encapsulation boundaries must be preserved.

Example

Undo functionality can nicely be implemented using the memento pattern. By serializing and deserializing the state of an object before the change occurs we can preserve a snapshot of it that can later be restored should the user choose to undo the operation.

OBJECT BEHAVIORAL



Purpose

Lets one or more objects be notified of state changes in other objects within the system.

Use When

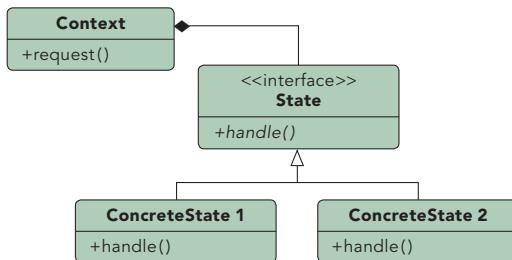
- State changes in one or more objects should trigger behavior in other objects
- Broadcasting capabilities are required.
- An understanding exists that objects will be blind to the expense of notification.

Example

This pattern can be found in almost every GUI environment. When buttons, text, and other fields are placed in applications the application typically registers as a listener for those controls. When a user triggers an event, such as clicking a button, the control iterates through its registered observers and sends a notification to each.

STATE

Object Behavioral



Purpose

Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.

Use When

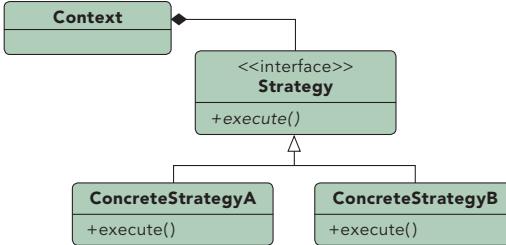
- The behavior of an object should be influenced by its state.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

Example

An email object can have various states, all of which will change how the object handles different functions. If the state is "not sent" then the call to `send()` is going to send the message while a call to `recallMessage()` will either throw an error or do nothing. However, if the state is "sent" then the call to `send()` would either throw an error or do nothing while the call to `recallMessage()` would attempt to send a recall notification to recipients. To avoid conditional statements in most or all methods there would be multiple state objects that handle the implementation with respect to their particular state. The calls within the Email object would then be delegated down to the appropriate state object for handling.

STRATEGY

Object Behavioral



Purpose

Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.

Use When

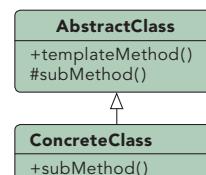
- The only difference between many related classes is their behavior.
- Multiple versions or variations of an algorithm are required.
- Algorithms access or utilize data that calling code shouldn't be exposed to.
- The behavior of a class should be defined at runtime.
- Conditional statements are complex and hard to maintain.

Example

When importing data into a new system different validation algorithms may be run based on the data set. By configuring the import to utilize strategies the conditional logic to determine what validation set to run can be removed and the import can be decoupled from the actual validation code. This will allow us to dynamically call one or more strategies during the import.

TEMPLATE METHOD

Class Behavioral



Purpose

Identifies the framework of an algorithm, allowing implementing classes to define the actual behavior.

Use When

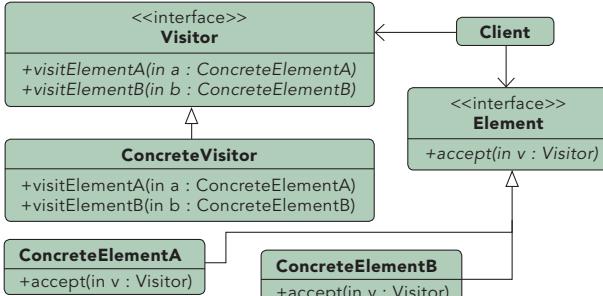
- A single abstract implementation of an algorithm is needed.
- Common behavior among subclasses should be localized to a common class.
- Parent classes should be able to uniformly invoke behavior in their subclasses.
- Most or all subclasses need to implement the behavior.

Example

A parent class, `InstantMessage`, will likely have all the methods required to handle sending a message. However, the actual serialization of the data to send may vary depending on the implementation. A video message and a plain text message will require different algorithms in order to serialize the data correctly. Subclasses of `InstantMessage` can provide their own implementation of the serialization method, allowing the parent class to work with them without understanding their implementation details.

VISITOR

Object Behavioral



```

classDiagram
    class Visitor {
        <<interface>>
        +visitElementA(a : ConcreteElementA)
        +visitElementB(b : ConcreteElementB)
    }
    class ConcreteVisitor {
        +visitElementA(a : ConcreteElementA)
        +visitElementB(b : ConcreteElementB)
    }
    class Element {
        <<interface>>
        +accept(v : Visitor)
    }
    class ConcreteElementA {
        +accept(v : Visitor)
    }
    class ConcreteElementB {
        +accept(v : Visitor)
    }

    Client --> Visitor
    Client --> Element
    ConcreteVisitor --> Element
    ConcreteElementA --> Element
    ConcreteElementB --> Element
  
```

Purpose
Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.

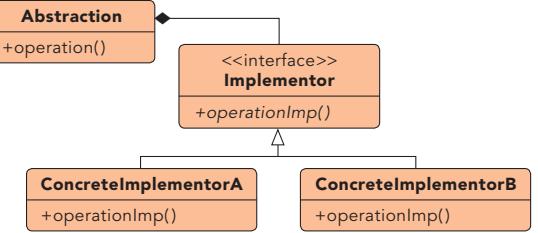
Use When

- An object structure must have many unrelated operations performed upon it.
- The object structure can't change but operations performed on it can.
- Operations must be performed on the concrete classes of an object structure.
- Exposing internal state or operations of the object structure is acceptable.
- Operations should be able to operate on multiple object structures that implement the same interface sets.

Example
Calculating taxes in different regions on sets of invoices would require many different variations of calculation logic. Implementing a visitor allows the logic to be decoupled from the invoices and line items. This allows the hierarchy of items to be visited by calculation code that can then apply the proper rates for the region. Changing regions is as simple as substituting a different visitor.

BRIDGE

Object Structural



```

classDiagram
    class Abstraction {
        <<interface>>
        +operation()
    }
    class Implementor {
        <<interface>>
        +operationImpl()
    }
    class ConcreteImplementorA {
        +operationImpl()
    }
    class ConcreteImplementorB {
        +operationImpl()
    }

    Abstraction <|-- Implementor
    Abstraction <|-- ConcreteImplementorA
    Abstraction <|-- ConcreteImplementorB
  
```

Purpose
Defines an abstract object structure independently of the implementation object structure in order to limit coupling.

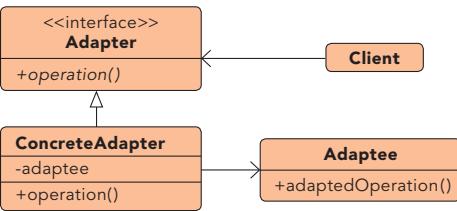
Use When

- Abstractions and implementations should not be bound at compile time.
- Abstractions and implementations should be independently extensible.
- Changes in the implementation of an abstraction should have no impact on clients.
- Implementation details should be hidden from the client.

Example
The Java Virtual Machine (JVM) has its own native set of functions that abstract the use of windowing, system logging, and byte code execution but the actual implementation of these functions is delegated to the operating system the JVM is running on. When an application instructs the JVM to render a window it delegates the rendering call to the concrete implementation of the JVM that knows how to communicate with the operating system in order to render the window.

ADAPTER

Class and Object Structural



```

classDiagram
    class Adapter {
        <<interface>>
        +operation()
    }
    class ConcreteAdapter {
        -adaptee
        +operation()
    }
    class Adaptee {
        +adaptedOperation()
    }

    Client --> Adapter
    Adapter --> Adaptee
  
```

Purpose
Permits classes with disparate interfaces to work together by creating a common object by which they may communicate and interact.

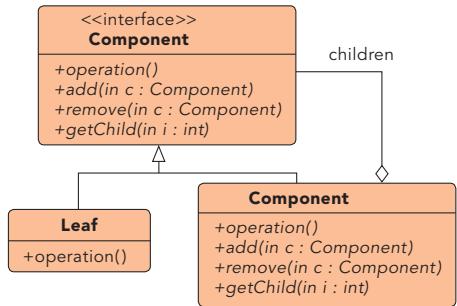
Use When

- A class to be used doesn't meet interface requirements.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

Example
A billing application needs to interface with an HR application in order to exchange employee data, however each has its own interface and implementation for the Employee object. In addition, the SSN is stored in different formats by each system. By creating an adapter we can create a common interface between the two applications that allows them to communicate using their native objects and is able to transform the SSN format in the process.

COMPOSITE

Object Structural



```

classDiagram
    class Component {
        <<interface>>
        +operation()
        +add(c : Component)
        +remove(c : Component)
        +getChild(i : int)
    }
    class Leaf {
        +operation()
    }
    class Composite {
        +operation()
        +add(c : Component)
        +remove(c : Component)
        +getChild(i : int)
    }

    Component <|-- Leaf
    Component <|-- Composite
    Composite "children" --> Component
  
```

Purpose
Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.

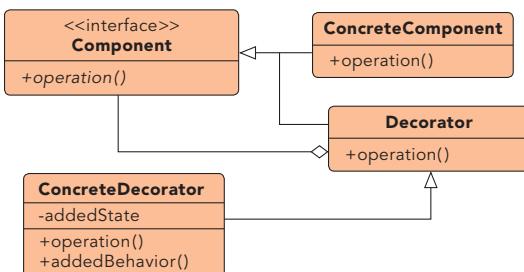
Use When

- Hierarchical representations of objects are needed..
- Objects and compositions of objects should be treated uniformly.

Example
Sometimes the information displayed in a shopping cart is the product of a single item while other times it is an aggregation of multiple items. By implementing items as composites we can treat the aggregates and the items in the same way, allowing us to simply iterate over the tree and invoke functionality on each item. By calling the getCost() method on any given node we would get the cost of that item plus the cost of all child items, allowing items to be uniformly treated whether they were single items or groups of items.

DECORATOR

Object Structural



Purpose

Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.

Use When

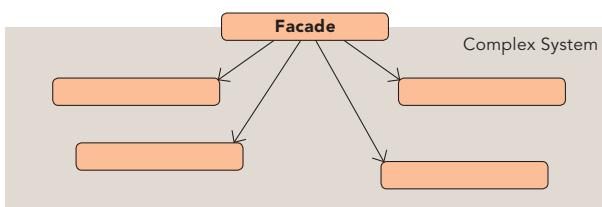
- Object responsibilities and behaviors should be dynamically modifiable.
- Concrete implementations should be decoupled from responsibilities and behaviors.
- Subclassing to achieve modification is impractical or impossible.
- Specific functionality should not reside high in the object hierarchy.
- A lot of little objects surrounding a concrete implementation is acceptable.

Example

Many businesses set up their mail systems to take advantage of decorators. When messages are sent from someone in the company to an external address the mail server decorates the original message with copyright and confidentiality information. As long as the message remains internal the information is not attached. This decoration allows the message itself to remain unchanged until a runtime decision is made to wrap the message with additional information.

FACADE

Object Structural



Purpose

Supplies a single interface to a set of interfaces within a system.

Use When

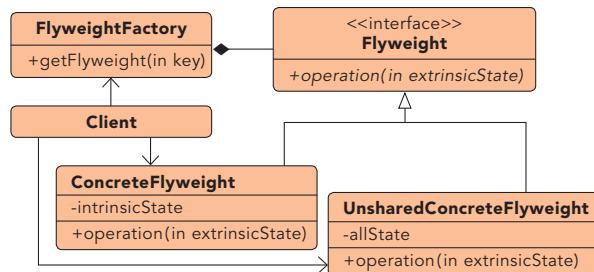
- A simple interface is needed to provide access to a complex system.
- There are many dependencies between system implementations and clients.
- Systems and subsystems should be layered.

Example

By exposing a set of functionalities through a web service the client code needs to only worry about the simple interface being exposed to them and not the complex relationships that may or may not exist behind the web service layer. A single web service call to update a system with new data may actually involve communication with a number of databases and systems, however this detail is hidden due to the implementation of the façade pattern.

FLYWEIGHT

Object Structural



Purpose

Facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient.

Use When

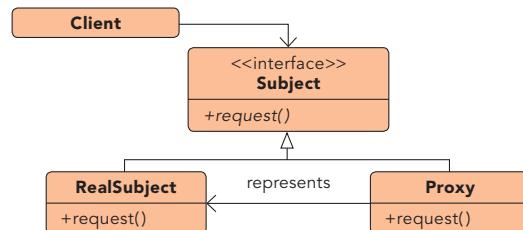
- Many like objects are used and storage cost is high.
- The majority of each object's state can be made extrinsic.
- A few shared objects can replace many unshared ones.
- The identity of each object does not matter.

Example

Systems that allow users to define their own application flows and layouts often have a need to keep track of large numbers of fields, pages, and other items that are almost identical to each other. By making these items into flyweights all instances of each object can share the intrinsic state while keeping the extrinsic state separate. The intrinsic state would store the shared properties, such as how a textbox looks, how much data it can hold, and what events it exposes. The extrinsic state would store the unshared properties, such as where the item belongs, how to react to a user click, and how to handle events.

PROXY

Object Structural



Purpose

Allows for object level access control by acting as a pass through entity or a placeholder object.

Use When

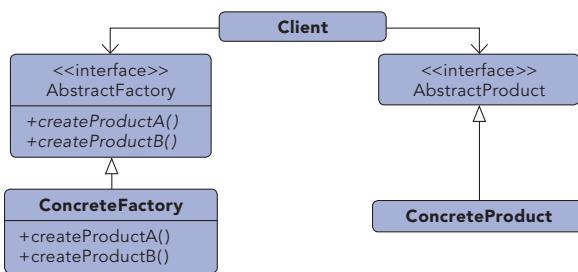
- The object being represented is external to the system.
- Objects need to be created on demand.
- Access control for the original object is required.
- Added functionality is required when an object is accessed.

Example

Ledger applications often provide a way for users to reconcile their bank statements with their ledger data on demand, automating much of the process. The actual operation of communicating with a third party is a relatively expensive operation that should be limited. By using a proxy to represent the communications object we can limit the number of times or the intervals the communication is invoked. In addition, we can wrap the complex instantiation of the communication object inside the proxy class, decoupling calling code from the implementation details.

ABSTRACT FACTORY

Object Creational



Purpose

Provide an interface that delegates creation calls to one or more concrete classes in order to deliver specific objects.

Use When

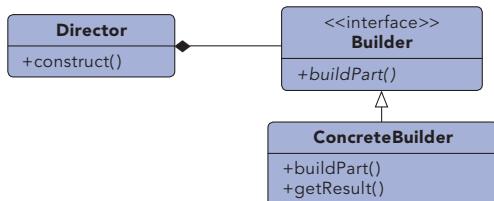
- The creation of objects should be independent of the system utilizing them.
- Systems should be capable of using multiple families of objects.
- Families of objects must be used together.
- Libraries must be published without exposing implementation details.
- Concrete classes should be decoupled from clients.

Example

Email editors will allow for editing in multiple formats including plain text, rich text, and HTML. Depending on the format being used, different objects will need to be created. If the message is plain text then there could be a body object that represented just plain text and an attachment object that simply encrypted the attachment into Base64. If the message is HTML then the body object would represent HTML encoded text and the attachment object would allow for inline representation and a standard attachment. By utilizing an abstract factory for creation we can then ensure that the appropriate object sets are created based upon the style of email that is being sent.

BUILDER

Object Creational



Purpose

Allows for the dynamic creation of objects based upon easily interchangeable algorithms.

Use When

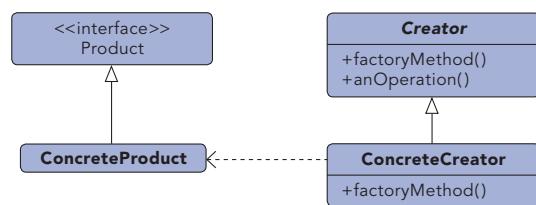
- Object creation algorithms should be decoupled from the system.
- Multiple representations of creation algorithms are required.
- The addition of new creation functionality without changing the core code is necessary.
- Runtime control over the creation process is required.

Example

A file transfer application could possibly use many different protocols to send files and the actual transfer object that will be created will be directly dependent on the chosen protocol. Using a builder we can determine the right builder to use to instantiate the right object. If the setting is FTP then the FTP builder would be used when creating the object.

FACTORY METHOD

Object Creational



Purpose

Exposes a method for creating objects, allowing subclasses to control the actual creation process.

Use When

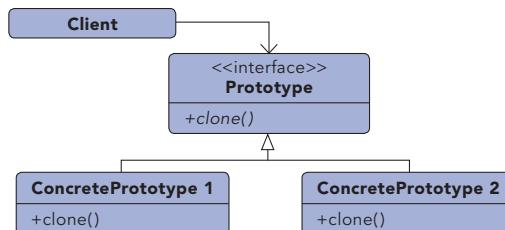
- A class will not know what classes it will be required to create.
- Subclasses may specify what objects should be created.
- Parent classes wish to defer creation to their subclasses.

Example

Many applications have some form of user and group structure for security. When the application needs to create a user it will typically delegate the creation of the user to multiple user implementations. The parent user object will handle most operations for each user but the subclasses will define the factory method that handles the distinctions in the creation of each type of user. A system may have AdminUser and StandardUser objects each of which extend the User object. The AdminUser object may perform some extra tasks to ensure access while the StandardUser may do the same to limit access.

PROTOTYPE

Object Creational



Purpose

Create objects based upon a template of an existing objects through cloning.

Use When

- Composition, creation, and representation of objects should be decoupled from a system.
- Classes to be created are specified at runtime.
- A limited number of state combinations exist in an object.
- Objects or object structures are required that are identical or closely resemble other existing objects or object structures.
- The initial creation of each object is an expensive operation.

Example

Rates processing engines often require the lookup of many different configuration values, making the initialization of the engine a relatively expensive process. When multiple instances of the engine is needed, say for importing data in a multi-threaded manner, the expense of initializing many engines is high. By utilizing the prototype pattern we can ensure that only a single copy of the engine has to be initialized then simply clone the engine to create a duplicate of the already initialized object. The added benefit of this is that the clones can be streamlined to only include relevant data for their situation.

SINGLETON

Object Creational

Singleton
-static uniqueInstance -singletonData
+static instance() +singletonOperation()

Purpose

Ensures that only one instance of a class is allowed within a system.

Use When

- Exactly one instance of a class is required.
- Controlled access to a single object is necessary.

Example

Most languages provide some sort of system or environment object that allows the language to interact with the native operating system. Since the application is physically running on only one operating system there is only ever a need for a single instance of this system object. The singleton pattern would be implemented by the language runtime to ensure that only a single copy of the system object is created and to ensure only appropriate processes are allowed access to it.

ABOUT THE AUTHOR



Jason McDonald

The product of two computer programmers, Jason McDonald wrote his first application in BASIC while still in elementary school and has been heavily involved in software ever since. He began his software career when he found himself automating large portions of one of his first jobs. Finding his true calling, he quit the position and began working as a software engineer for various small companies where he was responsible for all aspects of applications, from initial design to support. He has roughly 11 years of experience in the software industry and many additional years of personal software experience during which he has done everything from coding to architecture to leading and managing teams of engineers. Through his various positions he has been exposed to design patterns and other architectural concepts for years. Jason is the founder of the Charleston SC Java Users Group and is currently working to help found a Charleston chapter of the International Association of Software Architects.

Personal Blog: <http://www.mcdonaldland.info/>

Projects: Charleston SC Java Users Group

RECOMMENDED BOOK



Capturing a wealth of experience about the design of object-oriented software, four top-notch designers present a catalog of simple and succinct solutions to commonly occurring design problems. Previously undocumented, these 23 patterns allow designers to create more flexible, elegant, and ultimately reusable designs without having to rediscover the design solutions themselves.

BUY NOW

books.dzone.com/books/designpatterns

Get More FREE Refcardz. Visit refcardz.com now!

Upcoming Refcardz:

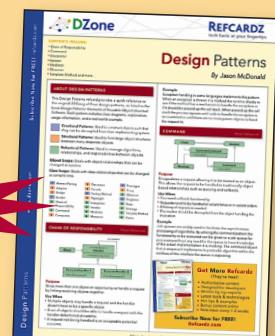
Core Seam
Core CSS: Part III
Hibernate Search
Equinox
EMF
XML
JSP Expression Language
ALM Best Practices
HTML and XHTML

Available:

Essential Ruby	Core CSS: Part I
Essential MySQL	Struts2
JUnit and EasyMock	Core .NET
Getting Started with MyEclipse	Very First Steps in Flex
Spring Annotations	C#
Core Java	Groovy
Core CSS: Part II	NetBeans IDE 6.1 Java Editor
PHP	RSS and Atom
Getting Started with JPA	GlassFish Application Server
JavaServer Faces	Silverlight 2

Visit refcardz.com for a complete listing of available Refcardz.

FREE



Design Patterns
Published June 2008



DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

Copyright © 2008 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher. Reference: *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. Addison-Wesley Professional, November 10, 1994.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513
888.678.0399
919.678.0300
Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-10-3
ISBN-10: 1-934238-10-4

50795

9 781934 238103

\$7.95

SOFTWARE DESIGN

Architectural patterns

Content

- Architectural Patterns
 - Layers
 - Client-Server
 - Event-driven
 - Broker
 - Mediator
 - MVC (and variants)
 - Microkernel
 - Service-based
 - SOA
 - Microservices
 - Space-based (Cloud)

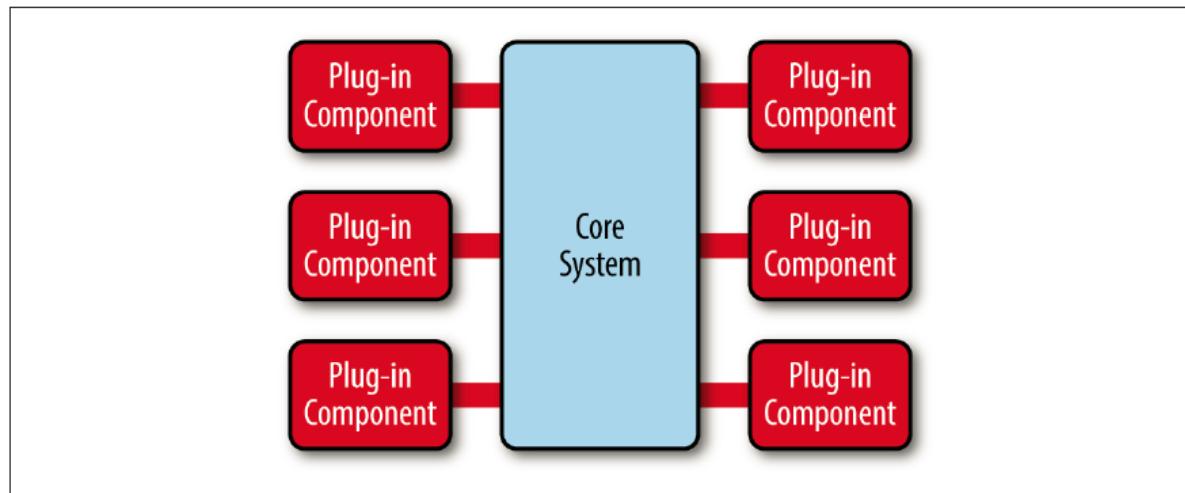
References

- Mark Richards, Software Architecture Patterns, O'Reilly, 2015 [SAP]
- Mark Richards, Microservices vs. Service-Oriented Architecture O'Reilly, 2016
- David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Ed.[Patterson]
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley [Taylor]
- Ian Gorton. 2011. Essential Software Architecture. 2nd Edition, Springer-Verlag [Gorton]
- Microsoft Application Architecture Guide, 2009 [MAAG]
- Armando Fox, David Patterson, and Koushik Sen, SaaS Course Stanford, Spring 2012 [Fox]
- Jacques Roy, SOA and Web Services, IBM
- Mark Bailey, Principles of Service Oriented Architecture, 2008

Microkernel

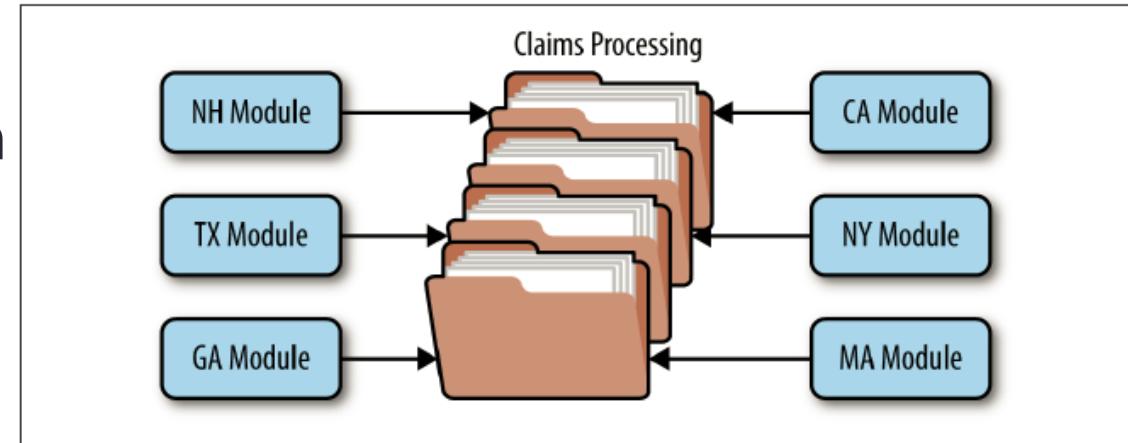
- Aka plug-in architecture
- Natural pattern for implementing product-based applications
- Allows to add additional application features as plug-ins to the core application
- 2 types of components:
 - Core system
 - Plug-in modules

- **The core system** - minimal functionality required to make the system operational.
- **The plug-in modules** - stand-alone, independent components that contain specialized processing, additional features, and custom code.
- Generally, **plug-in modules** should be independent of other plug-in modules.
- **The core system** needs to know about which **plug-in modules** are available and how to get to them => a plug-in registry.
- The registry contains information about each plug-in module (i.e. name, data contract, and remote access protocol details)
- **Plug-in modules** can be connected to the core system via: OSGi (open service gateway initiative), messaging, web services, or even direct point-to-point binding



Considerations

- Ex. Eclipse IDE
- Ex. Insurance application example



<i>Non functional req.</i>	<i>Rating</i>
Overall agility	High
Ease of deployment	High
Testability	High
Performance	Hgh
Scalability	Low
Ease of development	Low

Service based architectures

“1. All teams will henceforth expose their data and functionality through **service interfaces**

2. Teams must **communicate** with each other **through** these **interfaces**

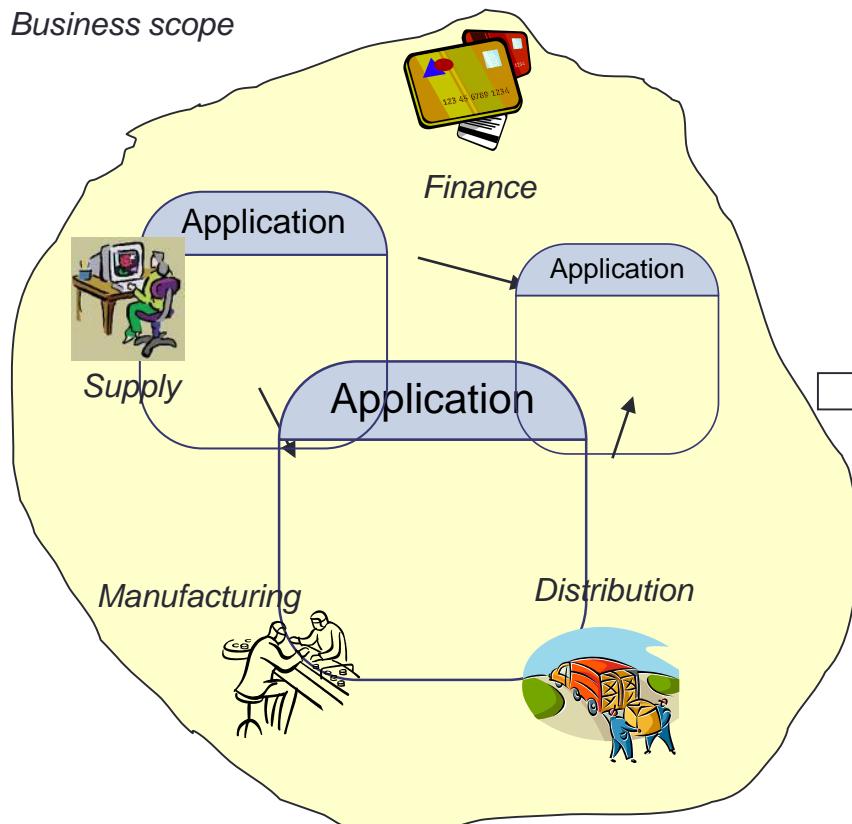
3. There will be **no other form of interprocess communication** allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

4. It **doesn't matter** what [API protocol] **technology** you use.
5. **Service interfaces**, without exception, must be designed from the ground up to be **externalizable**. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6. Anyone who doesn't do this **will be fired**.
7. Thank you; **have a nice day!"**

Amazon CEO

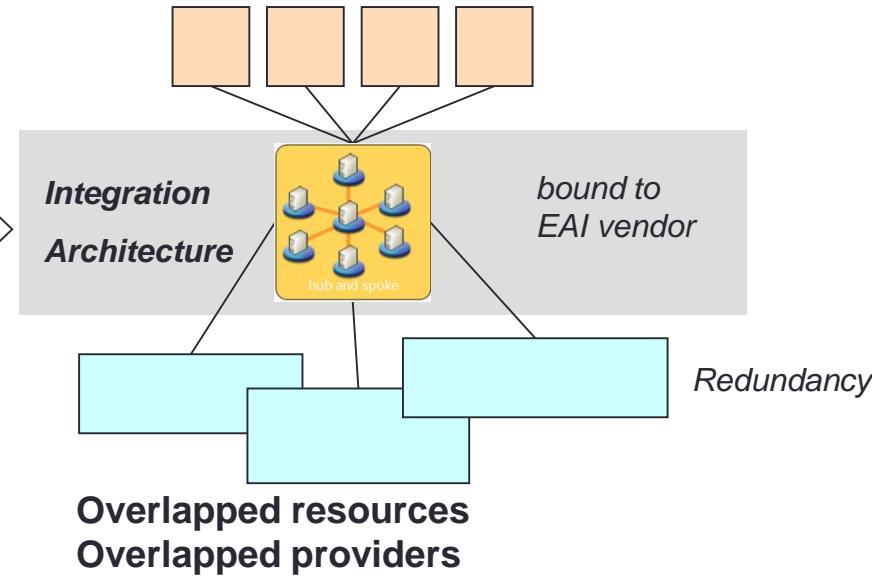
Application Centric

Business scope



Business functionality is duplicated in each application that requires it.

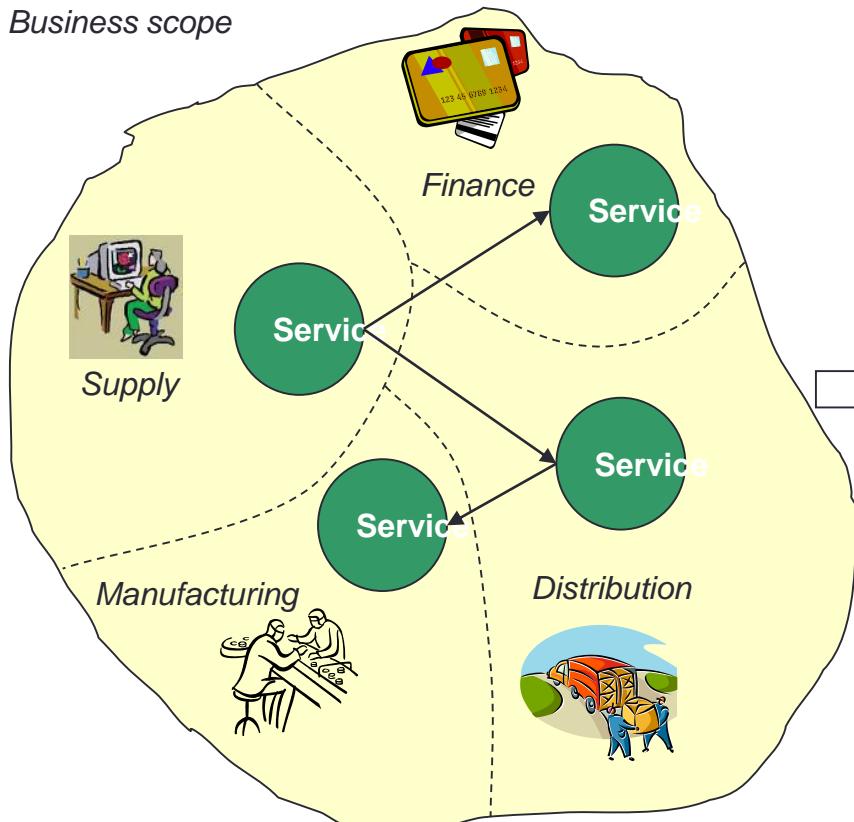
**Narrow Consumers
Limited Business Processes**



EAI ‘leverage’ application silos with the drawback of data and function redundancy.

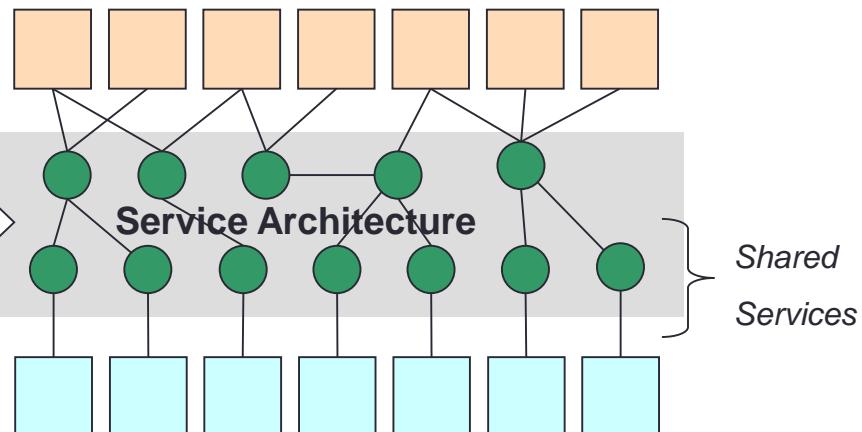
Service Centric

Business scope



SOA structures the business and its systems as a set of capabilities that are offered as Services, organized into a Service Architecture

**Multiple Service Consumers
Multiple Business Processes**

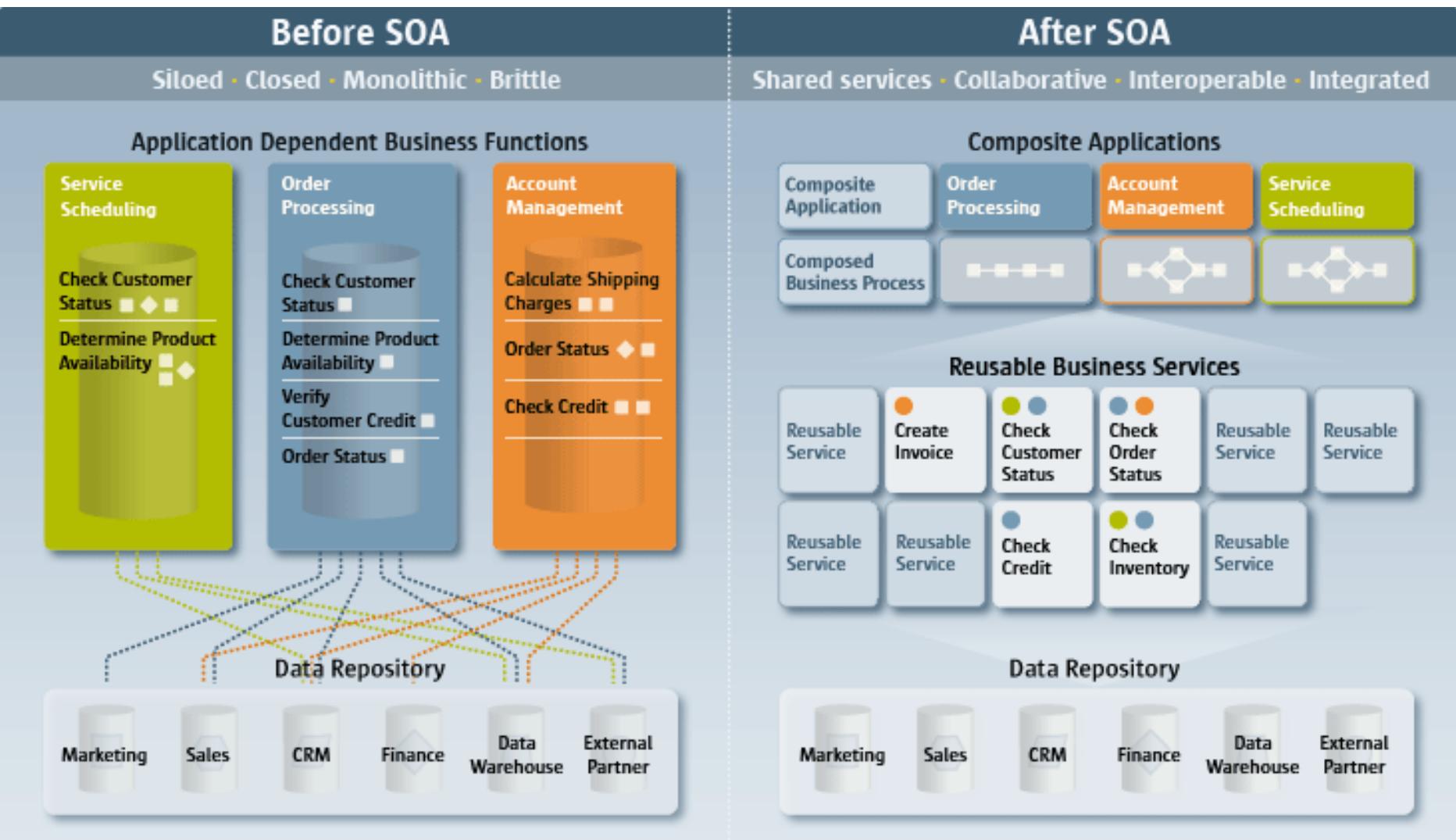


**Multiple Discrete Resources
Multiple Service Providers**

Service virtualizes how that capability is performed, and where and by whom the resources are provided, enabling multiple providers and consumers to participate together in shared business activities.

source: TietoEnator AB,
Kurts Bilder

Before SOA – After SOA

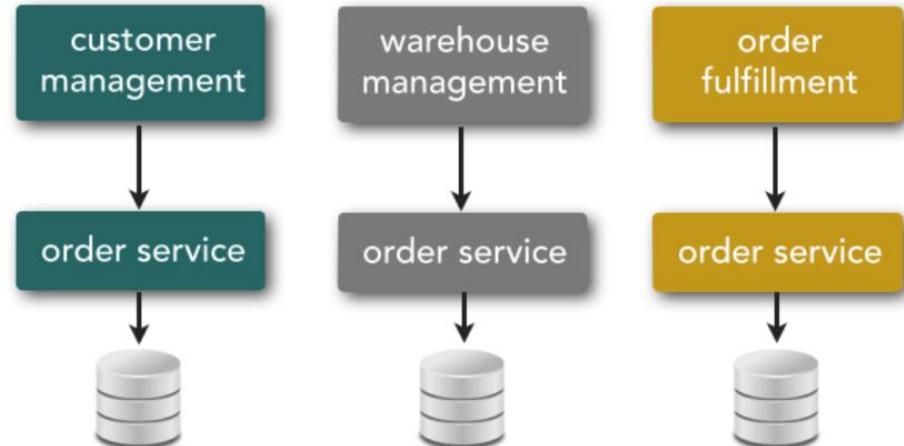


source:IBM

Motivation

Silo-based processing

- Redundant data,
- Needs synchronization
- Replicated processing logic



SOA

- Enterprise-level shared Services
- Synchronization done by service



Principles for identifying services

- A Service should:
 - Represent a tangible business concept.
 - Consist of a series of organization-wide analysis, where a process can be decomposed into several small processes.
 - Identify if any processes can be re-used internally or externally; such processes are strong candidates for Services.
 - Enhance reusability feature, identify possible inputs and possible outputs (should be generic) for these business processes.
 - Identify dependencies among Services and their impact on internal or external to the system.

Design principles [1]

- Services are reusable
 - Business functionalities exposed as services are designed with the intention of reuse whenever and where they are required
- Services share a formal contract
 - Services interact with each through a formal contract which is shared to exchange information and terms of usage
- Services are loosely coupled
 - Services are designed as loosely coupled entities able to interact while maintaining their state of loose coupling.
- Services abstract underlying logic
 - The business logic underpinning a service is kept hidden from the outside world. Only the service description and formal contract are visible for the potential consumers of a service

Design principles [2]

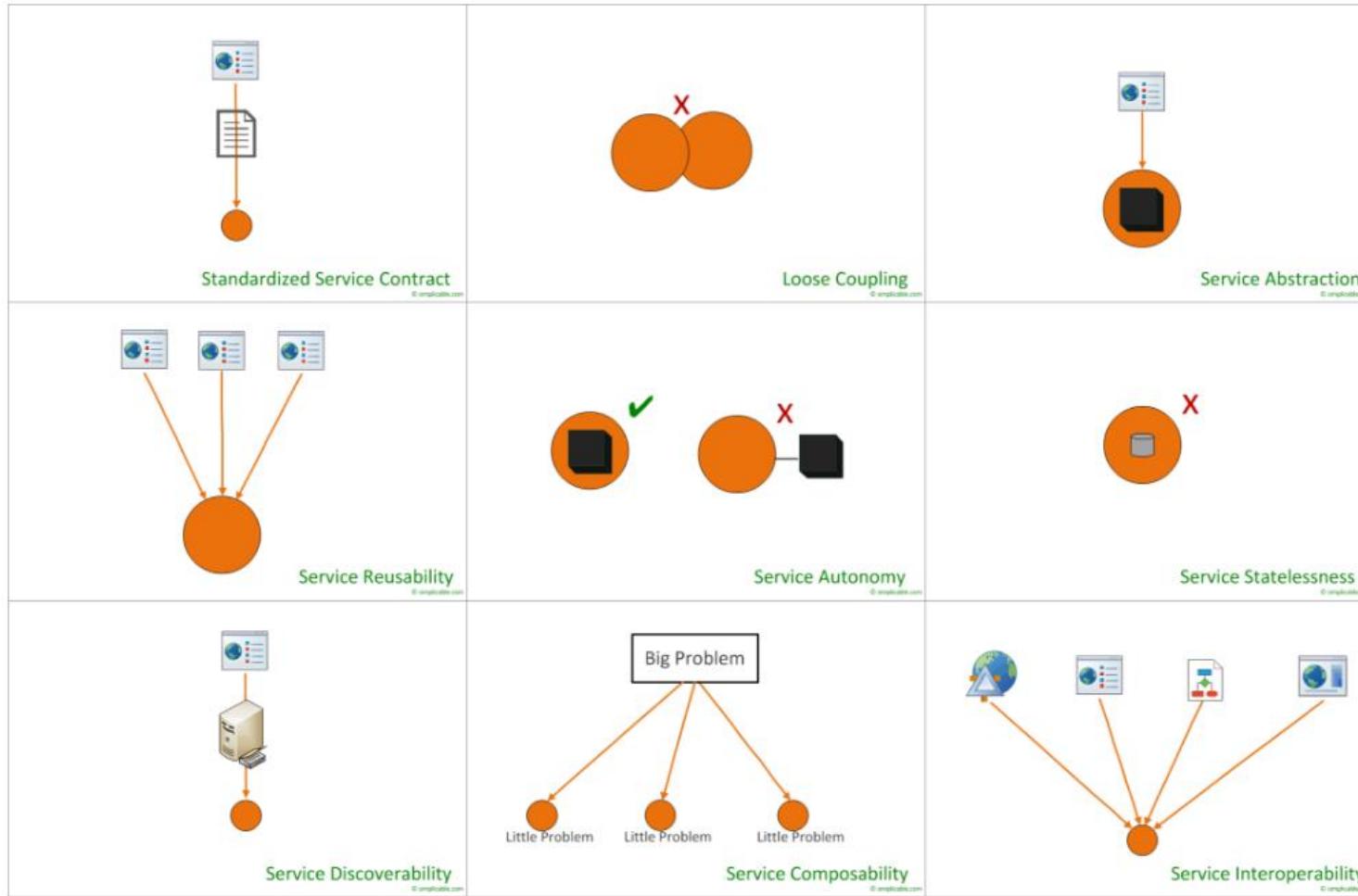
- Services are composable
 - Services may composed of other services. Hence, a service's logic should be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.
- Services are autonomous
 - A service should be independent of any other service
- Services are stateless
 - A service shouldn't required to maintain state information rather it should be designed to maximize statelessness
- Services are discoverable
 - A service should be discoverable through its description, which can be understood by humans and service users. A service can be discovered by the use of a directory provider, or implementation mechanism or hard-coded address

Design Principles [3]

- Services have a network-addressable interface
 - A service should be invoked from the same computer or remotely – through a local interface or Internet
- Services are location transparent
 - A service should be discoverable without the knowledge of its real location. A requestor can dynamically discover the location of a service looking up a registry.
- The core principles are **autonomy, loose coupling, abstraction, formal contract**

Service based architectures

- Rely on services to implement and perform functionality

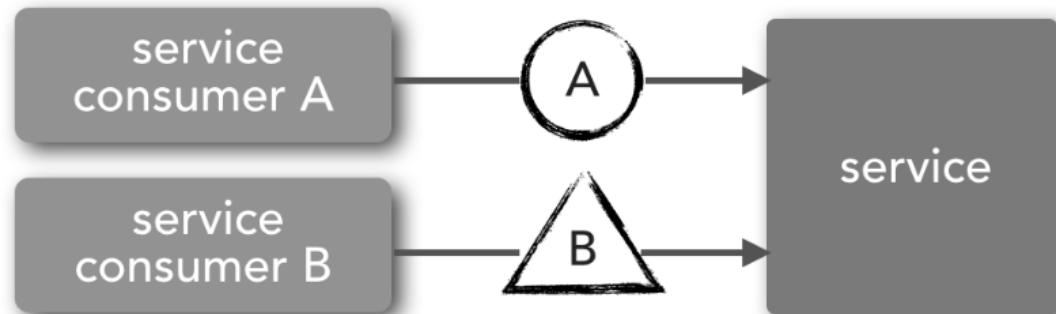


Service-based architectures

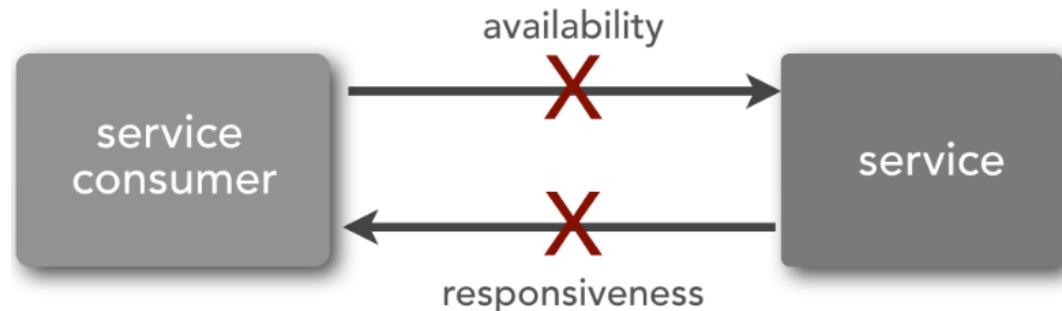
- Distributed, service components are accessed remote by some remote-access protocol:
 - Representational State Transfer (REST), Simple Object Access Protocol (SOAP), Advanced Message Queuing Protocol (AMQP), Java Message Service (JMS), Microsoft Messgae Queuing (MSMQ), Remote Method Invocation (RMI), etc...
- Modularity: self-contained services, designed/developed/tested/deployed independent from other
- Enable easy refactoring: big architectures can be refactored/replaced in smaller pieces
- Increased complexity and cost!

(some of the) Challenges

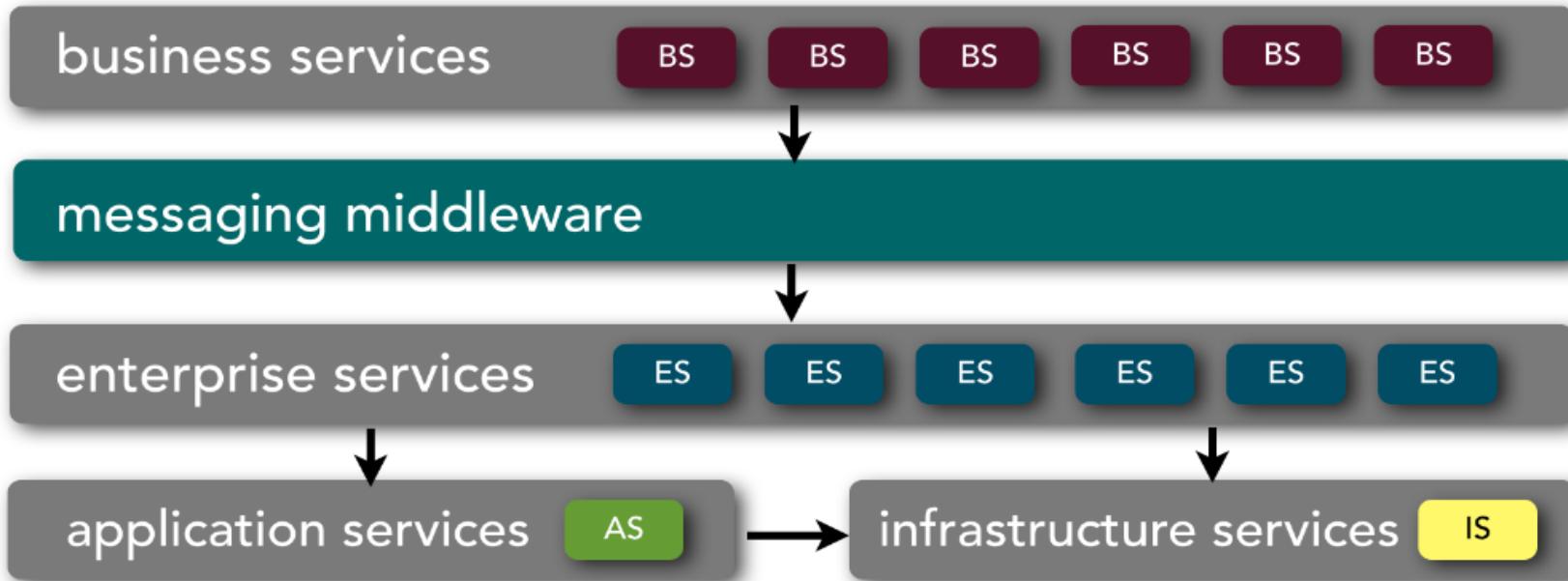
- Service contracts:
 - Agreement between service and consumer specifying the inbound and outbound data along with the contract format (XML, JavaScript Object Notation (JSON), Java object, etc.)



- Availability:
 - ability of a remote service to accept requests in a timely manner



SOA topology



Business services

- abstract, high-level, coarse-grained services that define the core business operations that are performed at the enterprise level.
- typically represented through either XML, Web Services Definition Language (WSDL), or Business Process Execution Language (BPEL).
- Ex. ProcessTrade,

SOA topology [2]

- *Enterprise services*
 - concrete, enterprise-level, coarse-grained services that implement the functionality defined by business services
 - can have a 1-to-1 or 1-to-many relationship with a business service
 - generally shared across the organization
 - Ex. RetrieveCustomer, ValidateOrder, GetInventory,...
- *Application services*
 - fine-grained, application-specific services that are bound to a specific application context
 - Ex. AddVehicle, CalculateAutoQuote,...
- *Infrastructure services*
 - implement nonfunctional tasks such as auditing, security, and logging

What is a Web Service? (W3C definition)

- A Web service is a:
 - software system designed to support
 - interoperable machine-to-machine interaction over a network.
 - has an interface described in a machine-processable format (specifically WSDL).
- Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

What is a Web Service: A Simpler Definition

- A Web Service is a standards-based way for an application to call a function over a network and to do it without having to know:
 - the location where the function will be executed,
 - the platform where the function will be run,
 - the programming language it is written in, or even
 - who built it.

Web Services - SOAP based

1. Discovery

- Where is the service?

2. Description

- What service does it offer?
- How do I use it?

3. Messaging

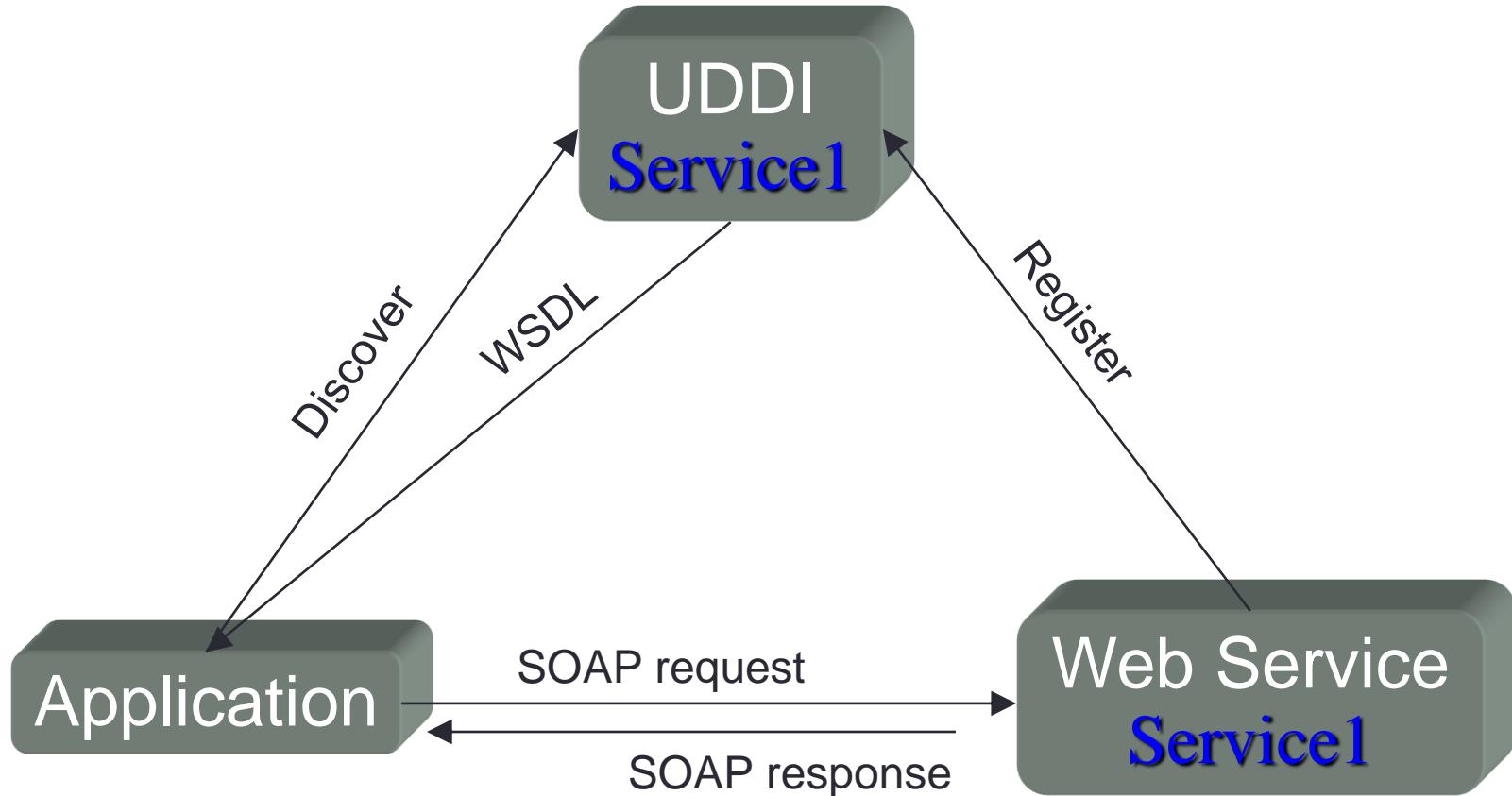
- Let's communicate!



Standard is Key

- WSDL is used to describe the function(s) that an application will be calling documenting in a standard way its entry points, parameters and output
- XML is used to carry the values of parameters and the outputs of the function
- SOAP is used as the messaging protocol that carries content (XML) over a network transport (typically HTTP)
- HTTP is used as the network transport layer

Interaction



WSDL: Web Service Description Language

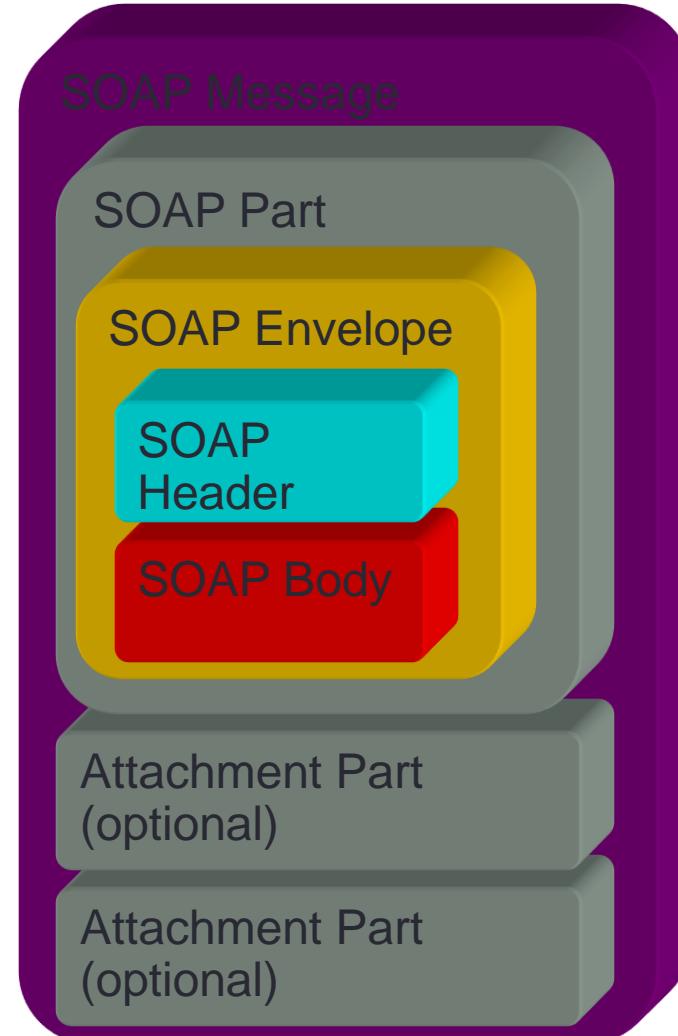
- WSDL (Web Services Description Language) is a public description of the interfaces offered by a web service.
- Expressed in XML
- Describes services as a set of endpoints
 - Document-oriented
 - Procedure-oriented

WSDL Specification

- XML grammar
 - <definitions>: root WSDL element
 - <types>: data types transmitted
(starts with XML Schema specifications)
 - <message>: messages transmitted
 - <portType>: functions supported
 - <binding>: specifics of transmissions
 - <service>: how to access it

SOAP: A Description

- Industry standard message format for sending and receiving data between a web services consumer and a web service provider
- SOAP messages are XML documents which have an envelope and:
 - Header (optional): contains information about the message such as date/time it was sent or security information
 - Body: contains the message itself
- Standard messaging protocol maintained by the W3C [XML Protocol Working Group](#).



SOAP used to stand for Simple Object Access Protocol

SOAP: Request/Response Example

Call to a fictional web service to get details on product with product id=827635

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>827635</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

Possible response to a request for product information:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productName>Toptimate 3-Piece Set</productName>
        <productID>827635</productID>
        <description>3-Piece luggage set. Black Polyester.</description>
        <price>96.50</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

Example

The screenshot illustrates the process of adding a service reference to a Geocoding application.

Geocoder Application Window: A window titled "Geocoder" contains fields for "Address", "Latitude", and "Longitude", along with a "Get Position" button.

Add Service Reference Dialog: This dialog box shows the URL <http://rpc.geocoder.us/dist/eg/clients/GeoCoder.wsdl> highlighted with a red oval. It includes a "Discover" button and a "Go" button.

Service Details: The "Services" section lists "GeoCode_Service". The "Operations" section is empty, with the message "Select a service contract to view its operations."

Summary: A message at the bottom states "1 service(s) found at address 'http://rpc.geocoder.us/dist/eg/clients/GeoCoder.wsdl'".

Advanced Options: An "Advanced..." button is located at the bottom left of the dialog.

Buttons: The dialog features "OK" and "Cancel" buttons at the bottom right.

GetPosition

```
private void btnGetPosition_Click(object sender, EventArgs e)
{
    // Create the client.
    GeocoderService.GeoCode_PortTypeClient Client =
        new GeocoderService.GeoCode_PortTypeClient();
    // Make the call.
    GeocoderService.GeocoderResult[] Result =
        Client.geocode(txtAddress.Text);
    // Check for an error result.
    if (Result != null)
    {
        // Display the results on screen.
        txtLatitude.Text = Result[0].lat.ToString();
        txtLongitude.Text = Result[0].@long.ToString();
    }
    else
    {
        // Display an error result.
        txtLatitude.Text = "Error";
        txtLongitude.Text = "Error";
    }
}
```

Give it a REST

- REST: REpresentation SState TTransfer
- Rest-ful: Follows the REST principles
- Not strictly for web services
- Term used loosely as a method of sending information over HTTP without using a messaging envelope

Representational State Transfer (REST)

- Idea: *Self-contained* requests specify what *resource* to operate on and what to do to it [Roy Fielding's PhD thesis, 2000]
- Wikipedia: “*a post hoc description of the features that made the Web successful*”
- A service (in the SOA sense) whose operations are like this is a RESTful service
- Ideally, RESTful URIs name the operations

REST Principles

- [RP1] The key abstraction of information is a **resource**, named by an URL. Any information that can be named can be a resource.
- [RP2] The **representation** of a resource is a sequence of bytes, plus representation metadata to describe those bytes. The particular form of the representation can be negotiated between REST components.
- [RP3] All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it (**Stateless**).

REST Principles (cont'd)

- [RP4] Components perform only a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST; for instance, all resources exposed via HTTP are expected to support each operation identically (**Uniform interface**).

REST Principles (cont'd)

- [RP5] Idempotent operations and representation metadata are encouraged in support of **caching** and representation reuse.
- [RP6] The presence of intermediaries is promoted. Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the user agent and the origin server (**Links between resources**).

Resources

- XML representation

```
<Person>
  <ID>1</ID>
  <Name>M Vaqqas</Name>
  <Email>m.vaqqas@gmail.com</Email>
  <Country>India</Country>
</Person>
```

- JSON representation

```
{
  "ID": "1",
  "Name": "M Vaqqas",
  "Email": "m.vaqqas@gmail.com",
  "Country": "India"
}
```

Messages – HTTP request



<VERB> is one of the HTTP methods like GET, PUT, POST, DELETE, OPTIONS, etc

<URI> is the URI of the resource on which the operation is going to be performed

<HTTP Version> is the version of HTTP

<Request Header> contains the metadata as a collection of key-value pairs of headers and their values. These settings contain information about the message and its sender like client type, the formats client supports, format type of the message body, cache settings for the response, and a lot more information.

<Request Body> is the actual message content. In a RESTful service, that's where the representations of resources sit in a message.

POST request example

POST http://MyService/Person/

Host: MyService

Content-Type: text/xml; charset=utf-8

Content-Length: 123

<?xml version="1.0" encoding="utf-8"?>

<Person>

<ID>1</ID>

<Name>M Vaqqas</Name>

<Email>m.vaqqas@gmail.com</Email>

<Country>India</Country>

</Person>

HTTP verbs – uniform interface

Method	Operation performed on server	Quality
GET	Read a resource.	Safe
PUT	Insert a new resource or update if the resource already exists.	Idempotent
POST	Insert a new resource. Also can be used to update an existing resource.	N/A
DELETE	Delete a resource .	Idempotent
OPTIONS	List the allowed operations on a resource.	Safe
HEAD	Return only the response headers and no response body.	Safe

Addressing resources

- **URI – well-structured if:**
 - Use plural nouns for naming your resources.
 - Avoid using spaces as they create confusion. Use an _ (underscore) or – (hyphen) instead.
 - A URI is case insensitive.
 - You can have your own conventions, but stay consistent throughout the service.
 - A cool URI never changes; so give some thought before deciding on the URIs for your service. If you need to change the location of a resource, do not discard the old URI. If a request comes for the old URI, use status code 300 and redirect the client to the new location.
 - Avoid verbs for your resource names until your resource is actually an operation or a process

Statelessness

- does not maintain the application state for any client. A request cannot be dependent on a past request and a service treats each request independently.

Example:

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/Persons/2 HTTP/1.1

Links between resources

Example: a Club with links to Persons

```
<Club>
  <Name>Authors Club</Name>
  <Persons>
    <Person>
      <Name>M. Vaqqas</Name>
      <URI>http://MyService/Persons/1</URI>
    </Person>
    <Person>
      <Name>S. Allamaraju</Name>
      <URI>http://MyService/Persons/12</URI>
    </Person>
  </Persons>
</Club>
```

Caching

- can be done on the client, the server, or on any other component between them, such as a proxy server.

Header	Application
Date	Date and time when this representation was generated.
Last Modified	Date and time when the server last modified this representation.
Cache-Control	The HTTP 1.1 header used to control caching.
Expires	Expiration date and time for this representation. To support HTTP 1.0 clients.
Age	Duration passed in seconds since this was fetched from the server. Can be inserted by an intermediary component.

Documenting REST services

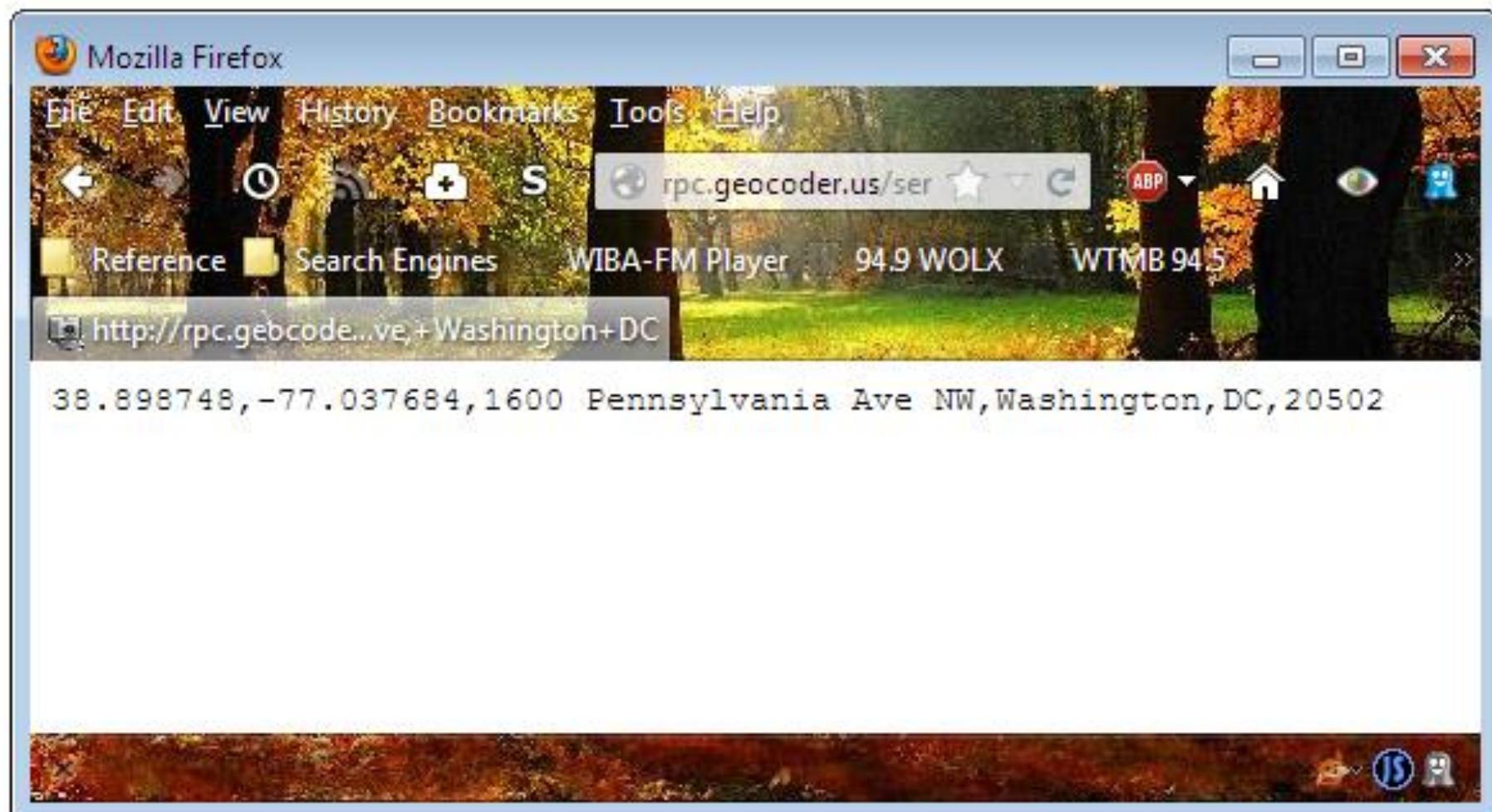
Service Name: MyService

Address: http://MyService/

Resource	Methods	URI	Description
Person	GET, POST, PUT, DELETE	http://MyService/Persons/{PersonID}	Contains information about a person {PersonID} is optional Format: text/xml
Club	GET, POST, PUT	http://MyService/Clubs/{ClubID}	Contains information about a club. A club can be joined by multiple people {ClubID} is optional Format: text/xml
Search	GET	http://MyService/Search?	Search a person or a club Format: text/xml Query Parameters: Name: String, Name of a person or a club Country: String, optional, Name of the country of a person or a club Type: String, optional, Person or Club. If not provided then search will result in both Person and Clubs

Example

<http://rpc.geocoder.us/service/csv?address=1600+Pennsylvania+Ave,+Washington+DC>



SOAP vs. REST

- SOAP
 - SOAP is still offered by some very prominent tech companies for their APIs (Salesforce, Paypal, Docusign).
 - SOAP is good for applications that require *formal contracts* between the API and consumer, since it can enforce the use of formal contracts by using WSDL
 - Additionally, SOAP has built in *WS-Reliable messaging* to increase security in asynchronous execution and processing.
 - SOAP has built-in *stateful operations*. SOAP is designed support conversational state management.
 - Provides support for WS_AtomicTransaction and WS_Security, SOAP can benefit developers when there is a high need for transactional reliability.

SOAP

- Language, platform, and transport independent (REST requires use of HTTP)
- Works well in distributed enterprise environments (REST assumes direct point-to-point communication)
- Standardized
- Provides significant pre-build extensibility in the form of the WS* standards (I.e. WS-Addressing, WS-Policy, WS-Security, WS-Federation, WS-ReliableMessaging, WS-Coordination, WS-AtomicTransaction, and WS-RemotePortlets)
- Built-in error handling
- Automation when used with certain language products

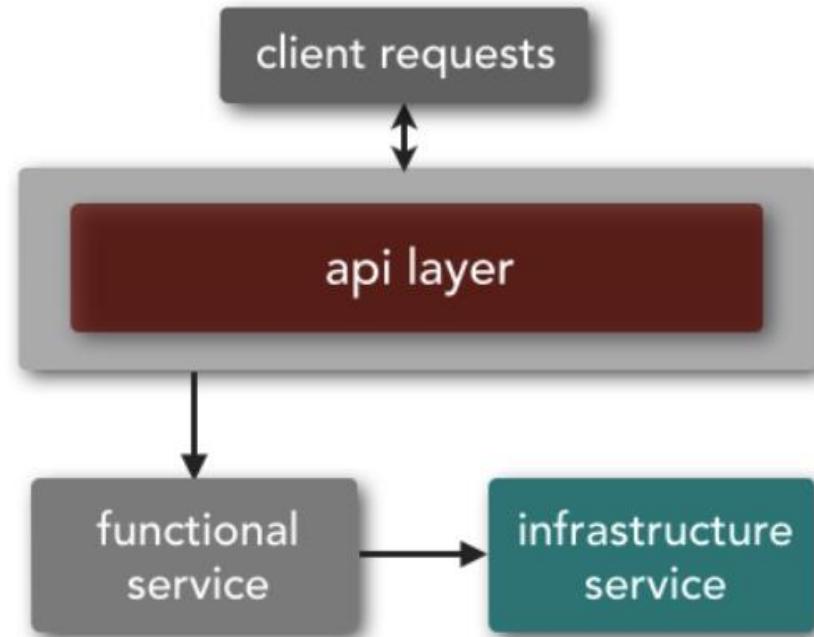
REST

- REST is easy to understand: it uses HTTP and basic CRUD operations, so it is simple to write and document.
- REST also makes efficient use of bandwidth, as it's much less verbose than SOAP. Unlike SOAP, REST is designed to be stateless and REST reads can be cached for better performance and scalability.
- REST supports many data formats, but the predominant use of JSON means better support for browser clients. JSON sets a standardized method for consuming API payloads, so you can take advantage of its connection with JavaScript and the browser.

REST

- No expensive tools require to interact with the Web service
- Smaller learning curve
- Efficient (SOAP uses XML for all messages, REST can use smaller message formats)
- Fast (no extensive processing required)
- Closer to other Web technologies in design philosophy

Microservices topology



Functional services

- support specific business operations or functions
- accessed externally and are generally not shared with any other service

Infrastructure services

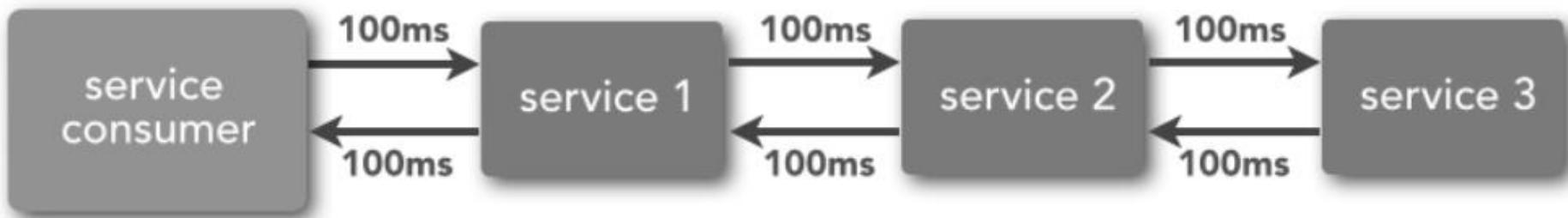
- support nonfunctional tasks such as authentication, authorization, auditing, logging, and monitoring.
- not exposed to the outside world but rather are treated as private shared services only available internally to other services

Granularity

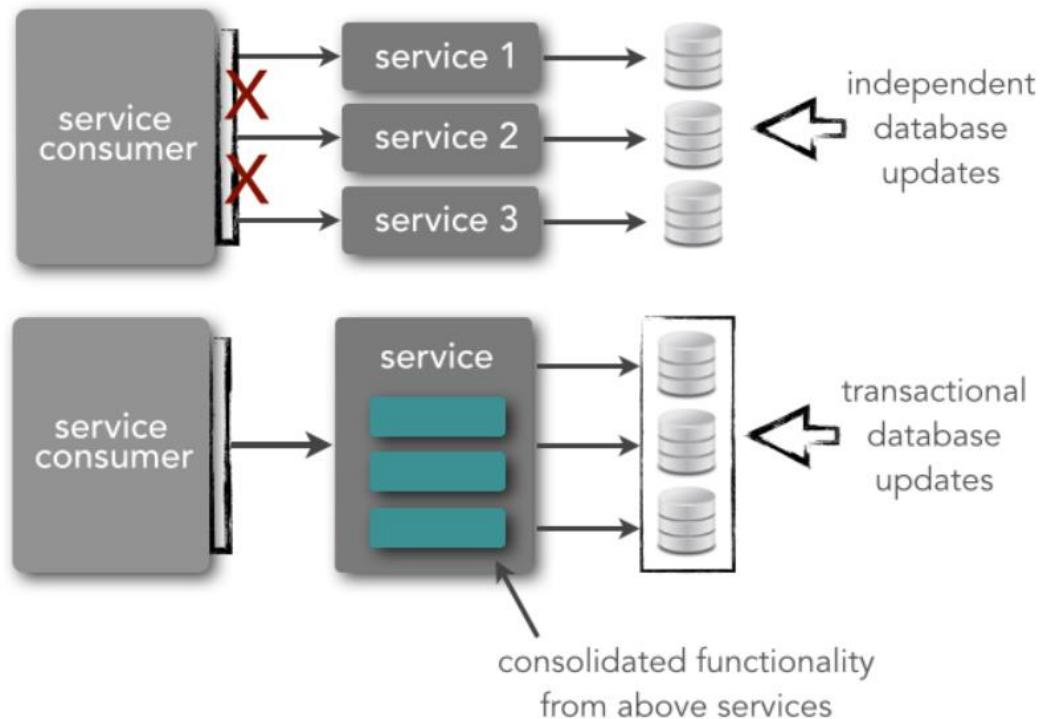
- SOA – [small application services; very large enterprise services]
- Ex. enterprise *Customer* service handles update and retrieval data views, delegating the lower-level getters and setters to application-level services that were not exposed remotely to the enterprise
- Microservices - single-purpose services that do one thing really, really well
- Ex. fine-grained getter and setter services like *GetCustomerAddress*, *GetCustomerName*, *UpdateCustomerName*

Impact of granularity

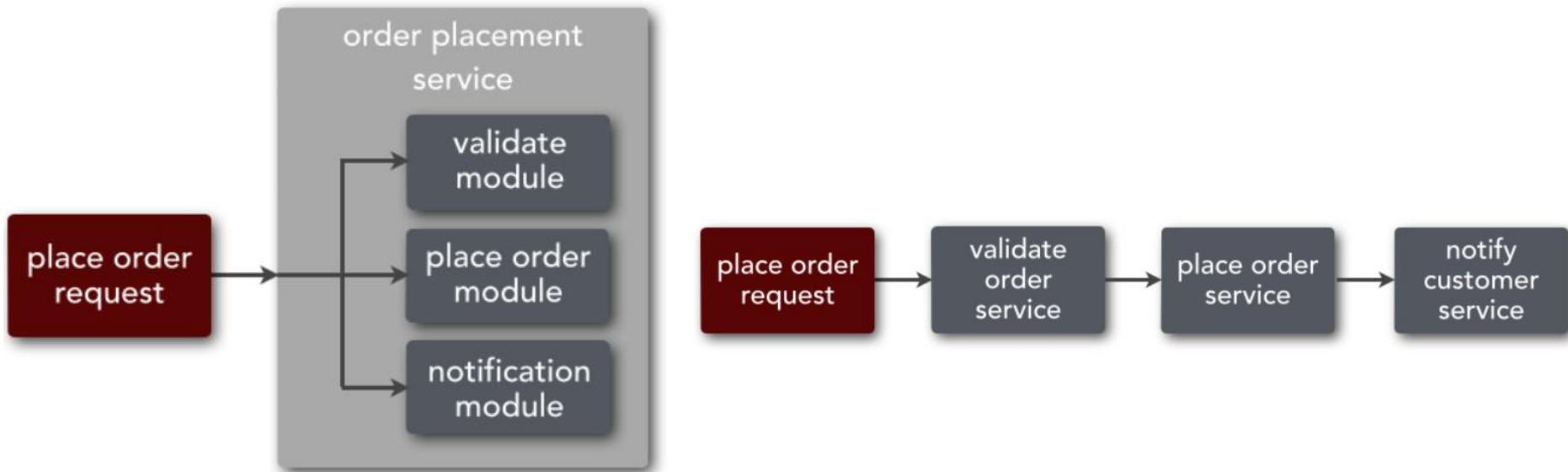
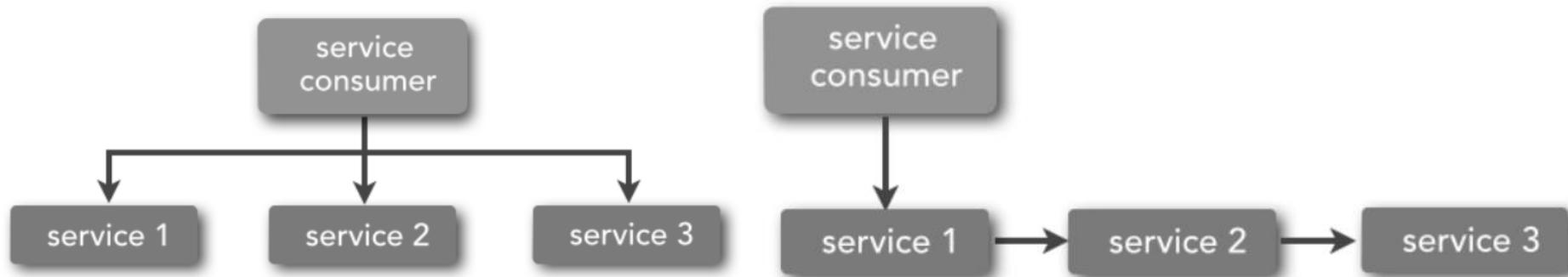
- Performance



- Transaction management

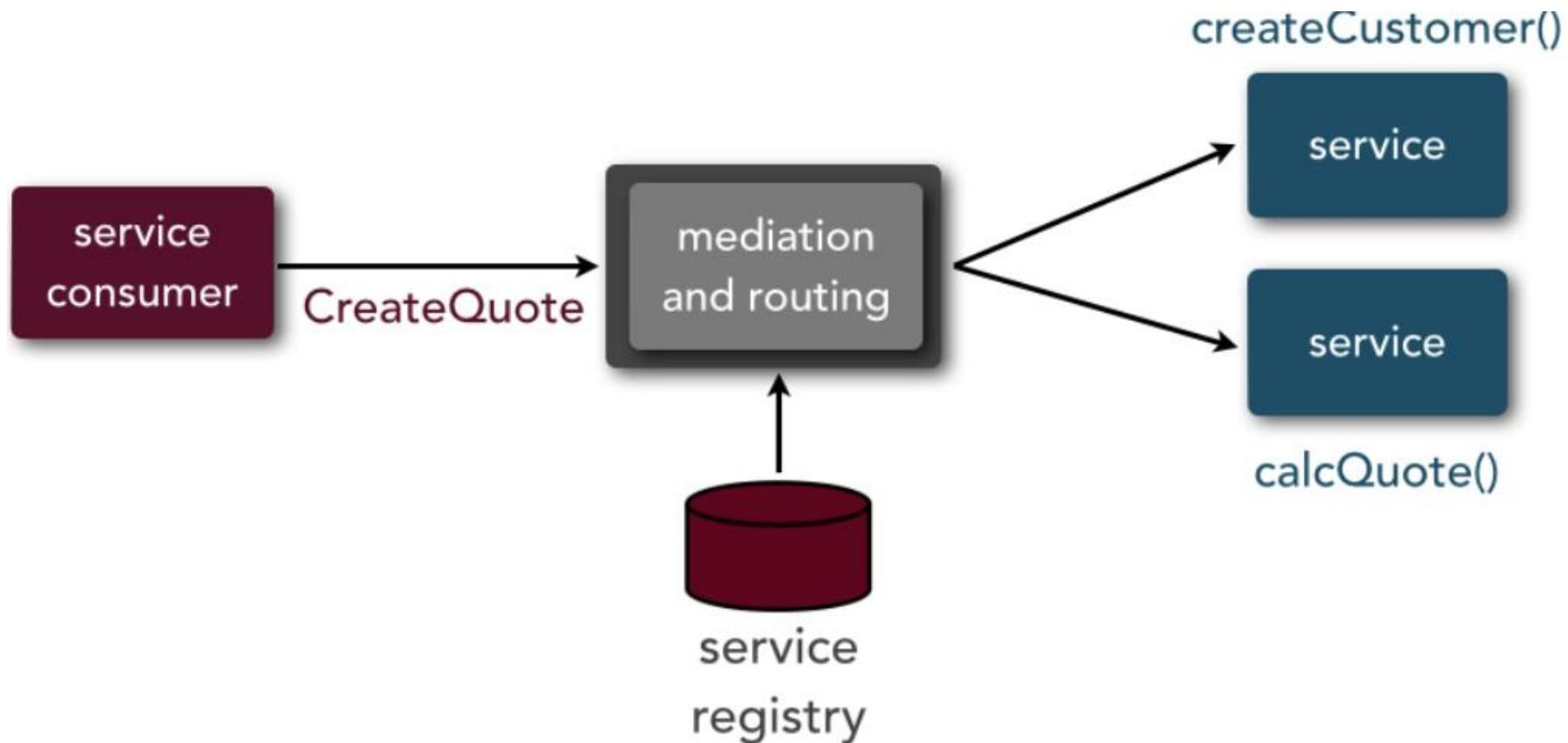


Orchestration and Choreography

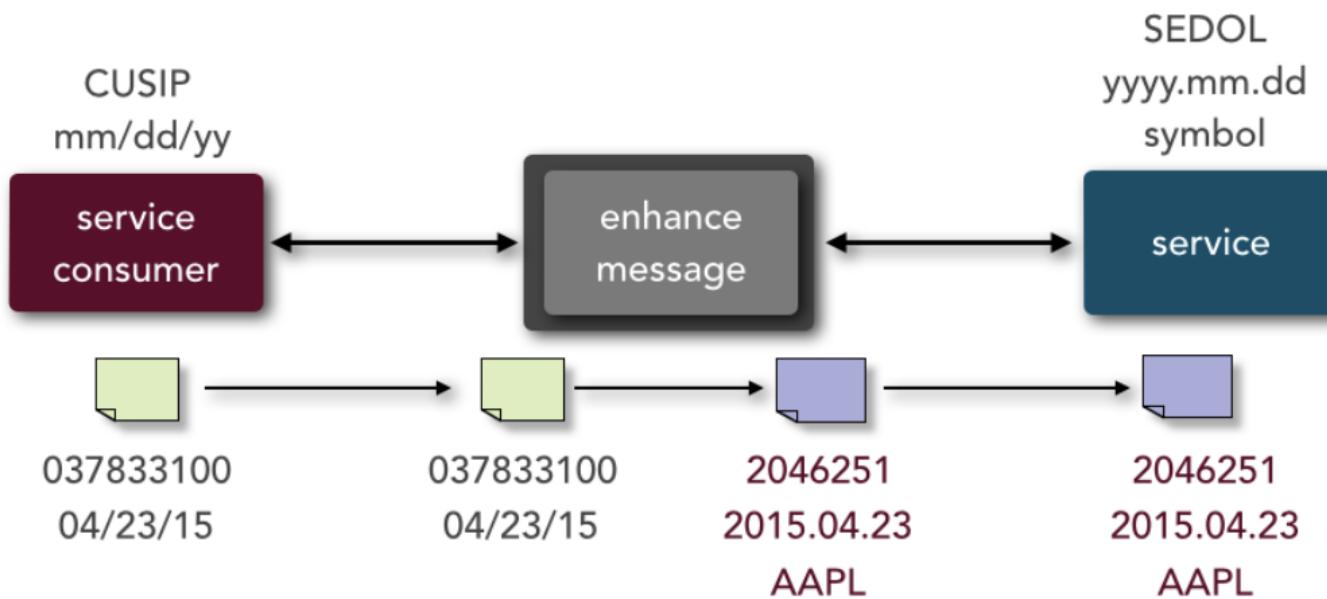


SOA Middleware

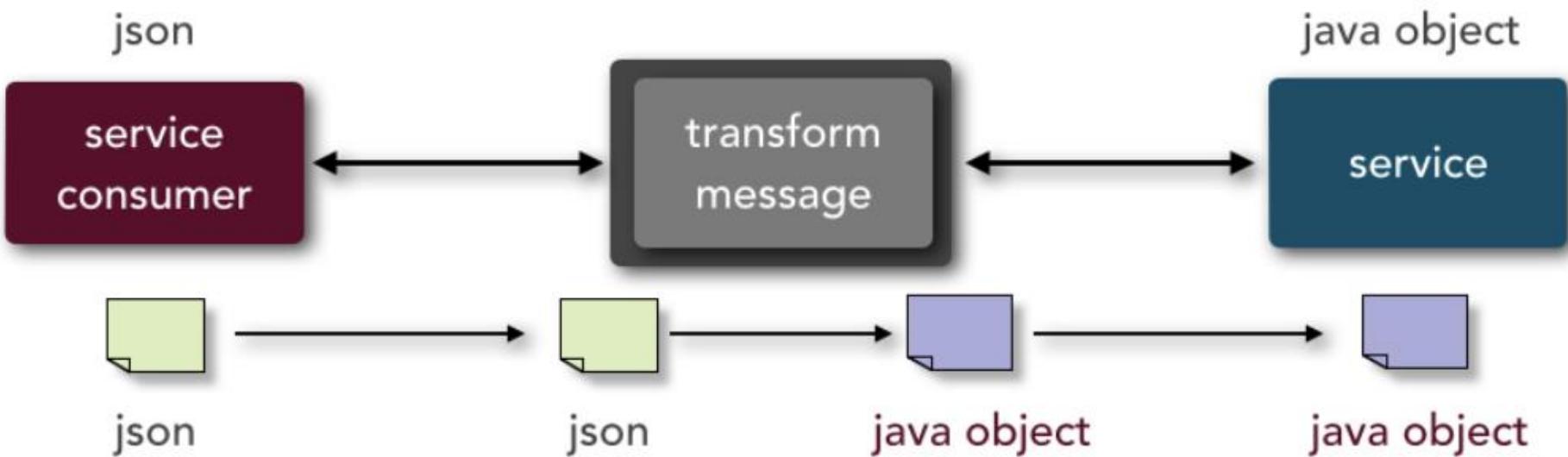
- *Mediation and routing* - locate and invoke a service (or services) based on a specific business or user request



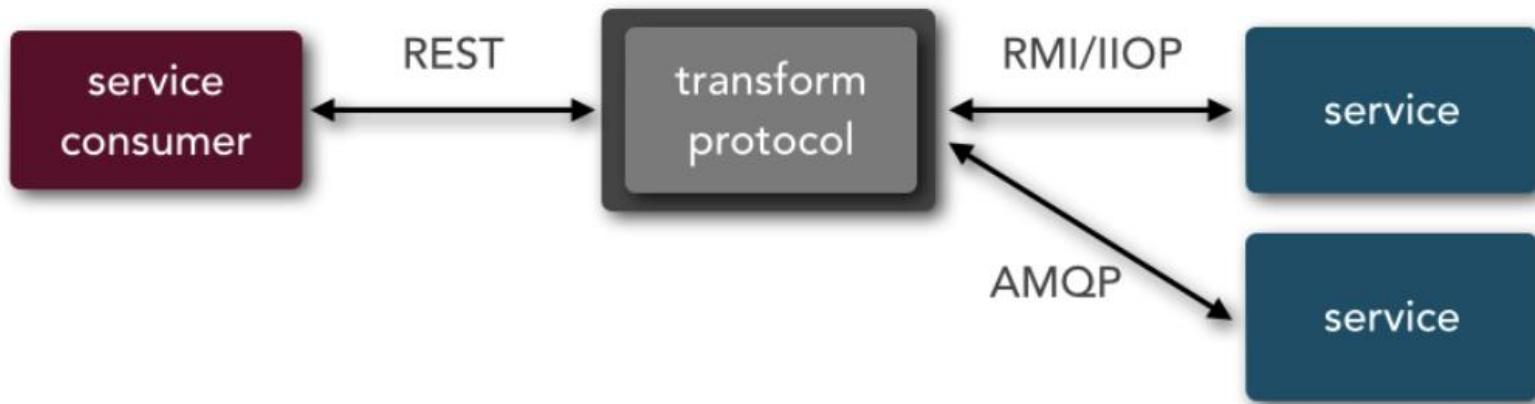
- *Message enhancement* - modify, remove, or augment the data portion of a request before it reaches the service.
- Ex. changing a date format, adding additional derived or calculated values to the request, and performing a database lookup to transform one value into another



- *Message transformation* - modify the format of the data from one type to other.
- Ex. the service consumer is calling a service and sending the data in JSON format, whereas the service requires a Java object

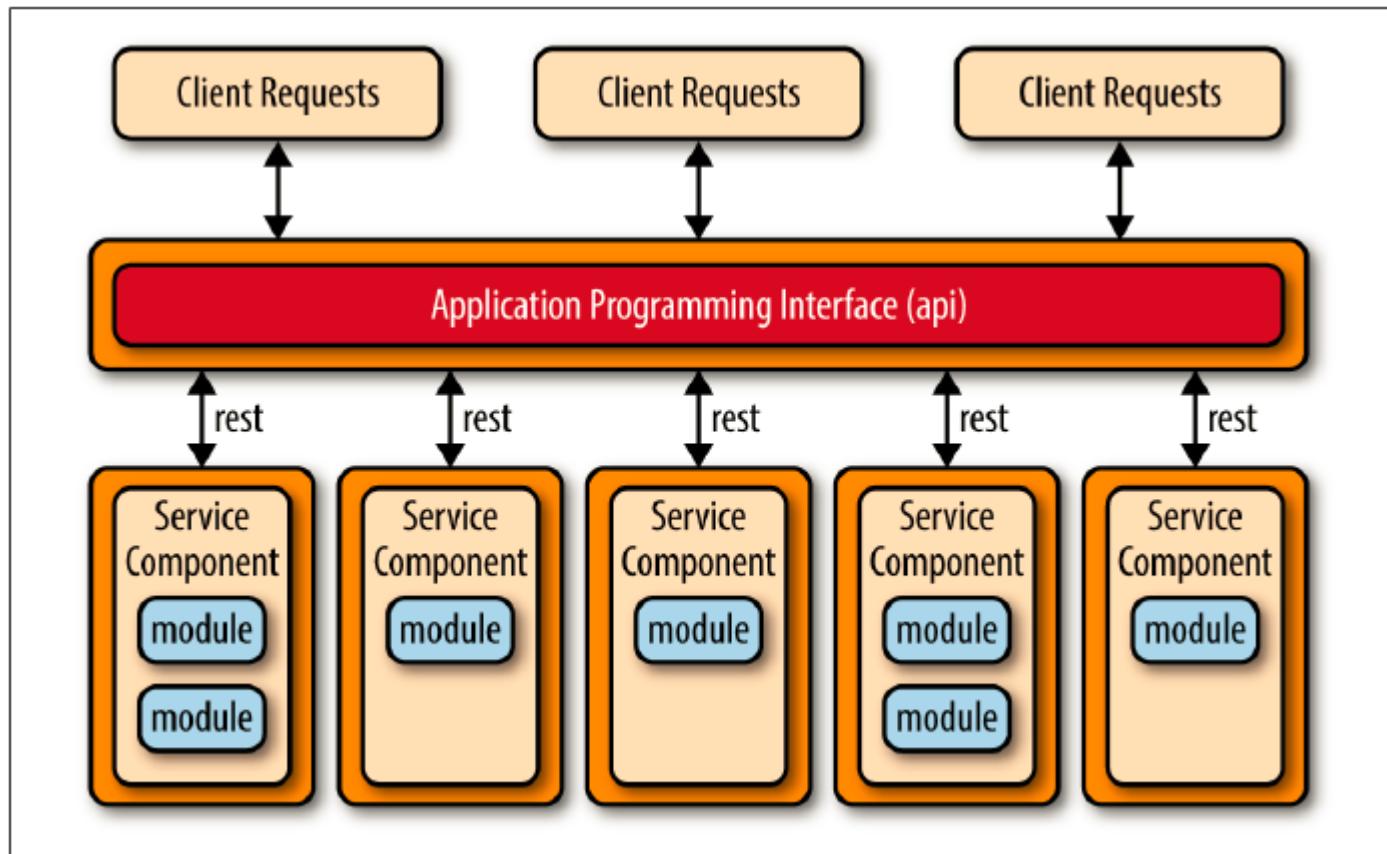


- *Protocol transformation* - have a service consumer call a service with a protocol that differs from what the service is expecting.
- Ex. the service consumer is communicating through REST, but the services invoked require an RMI/IOP connection (e.g., Enterprise JavaBeans 3 [EJB3] bean) and an AMQP connection.



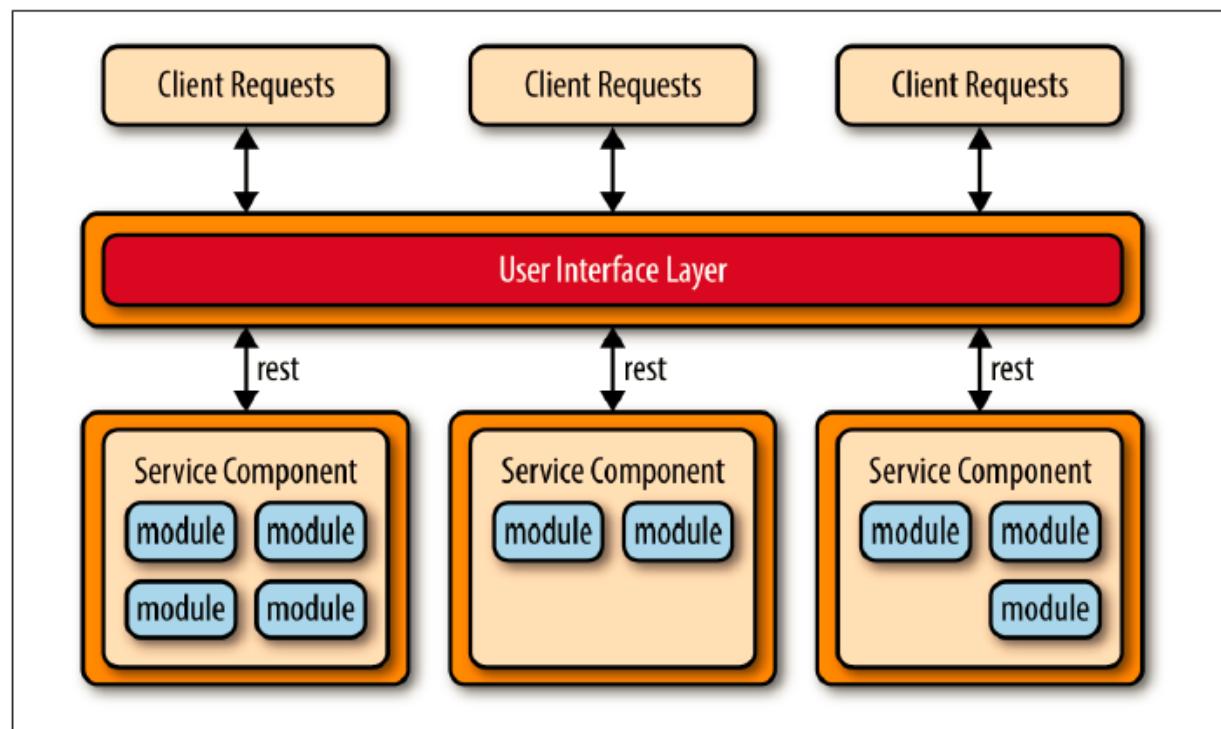
Microservices “API” layer

API REST-based topology - websites that expose small, self-contained individual services through some sort of API



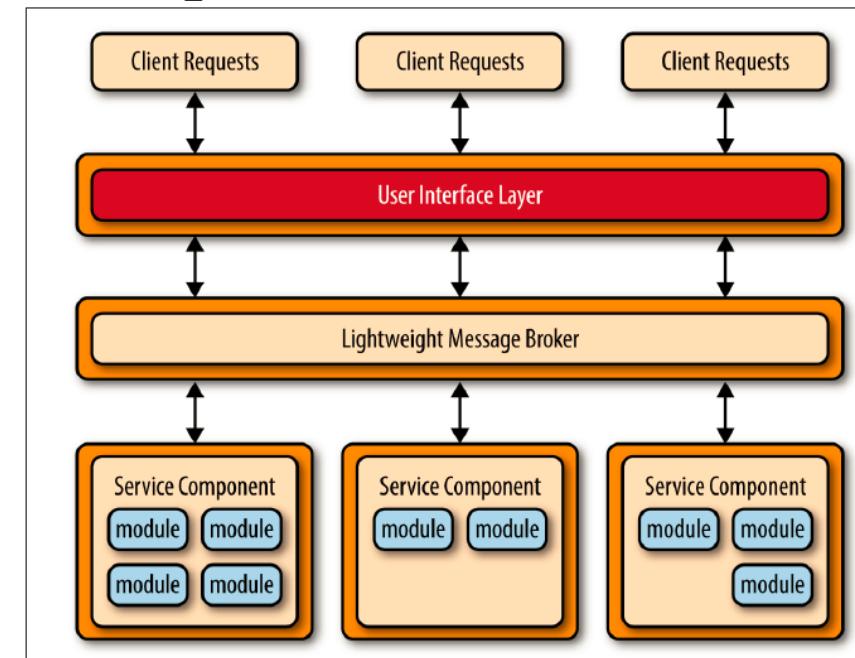
Application REST-based topology – the client requests are received through traditional web-based or fat-client business application screens rather than through a simple API layer

- The service components tend to be larger, more coarse-grained, and represent a small portion of the overall business application rather than fine-grained, single-action services.



Centralized messaging topology - similar to the previous application REST based topology except that instead of using REST for remote access, it uses a lightweight centralized message broker (e.g., ActiveMQ, HornetQ, etc.).

- No orchestration, transformation, or complex routing!
- advanced queuing mechanisms, asynchronous messaging, monitoring, error handling, and better overall load balancing and scalability



SOA vs.

- share-as-much-as-possible
- uses orchestration and choreography
- Uses Message middleware
- Coarse-grained services
- no pre-described limits as to which remote-access protocols can be used

Microservices

- share-as-little-as-possible
- favorizes choreography
- Uses API layer as service access façade
- Fine-grained services
- rely on 2 different remote-access protocols to access services: REST and simple messaging (JMS, MSMQ, AMQP, etc.)

Considerations

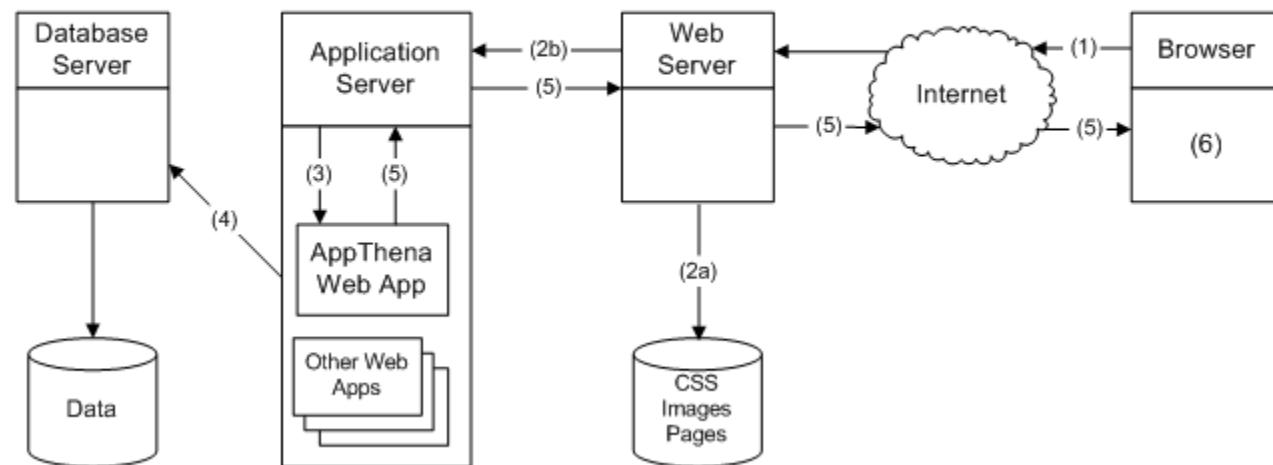
- Microservices
 - more robust,
 - provide better scalability,
 - supports continuous delivery
 - real-time production deployments
- Yet still...
 - Difficult contract creation, maintenance, and government,
 - remote system availability,
 - remote access authentication and authorization

<i>Non functional req.</i>	<i>Rating</i>
Overall agility	High
Ease of deployment	High
Testability	High
Performance	Low
Scalability	High
Ease of development	High

Space-Based Architecture

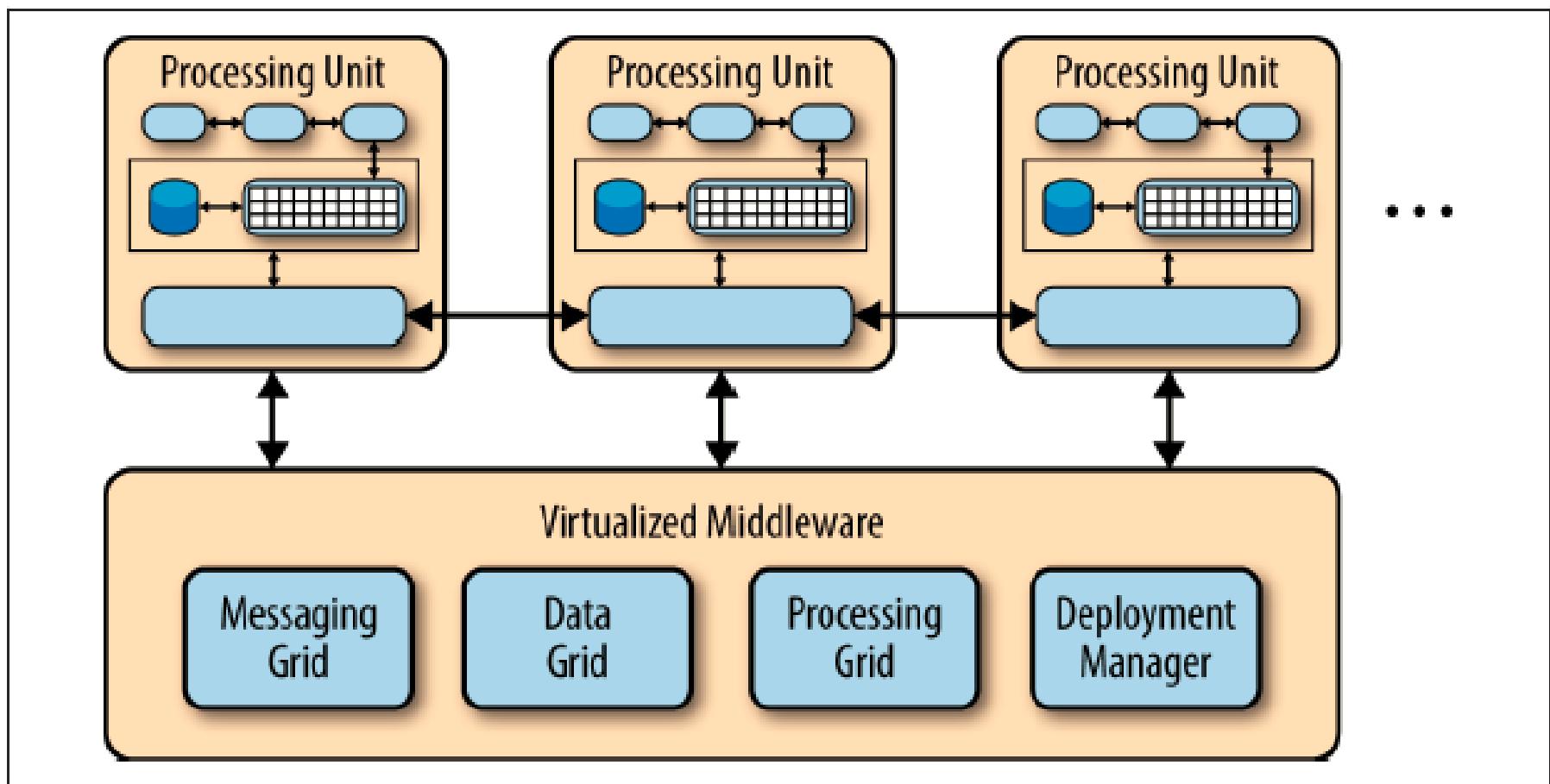
- aka cloud architecture pattern
- designed to address and solve scalability and concurrency issues
- for applications that have variable and unpredictable concurrent user volumes

Web Application Architecture



Description

- relies on the concept of tuple spaces

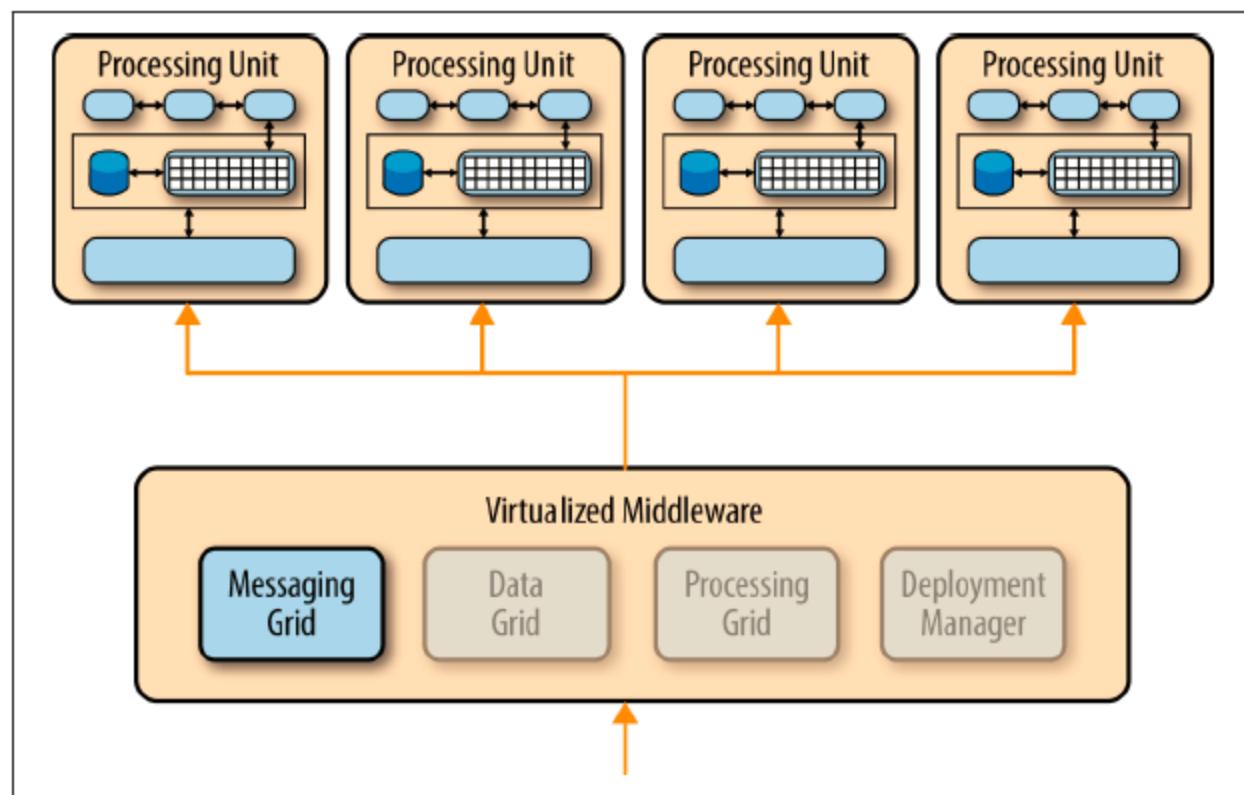


Components

- Processing unit
 - Contains application components (web-based and backend business logic)
 - Contains in-memory data grid and optional asynchronous persistent store for failover
 - Contains replication engine
- Virtualized middleware
 - Handles housekeeping and communications i.e. data synchronization and request handling
 - 4 main components: the messaging grid, the data grid, the processing grid, and the deployment manager.

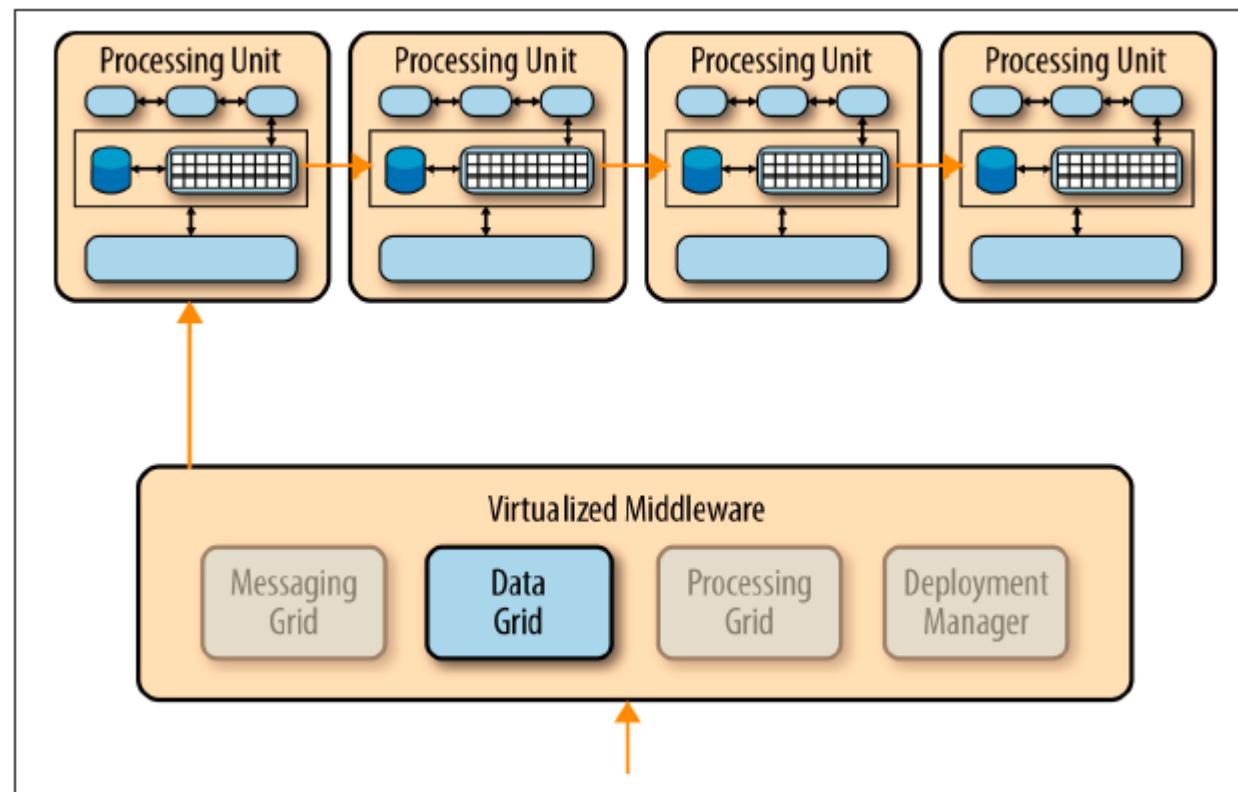
Messaging Grid

- Manages input request and session information
- Determines the available active PU and forwards the request
- Can be a simple round-robin alg. or a complex next-available alg.



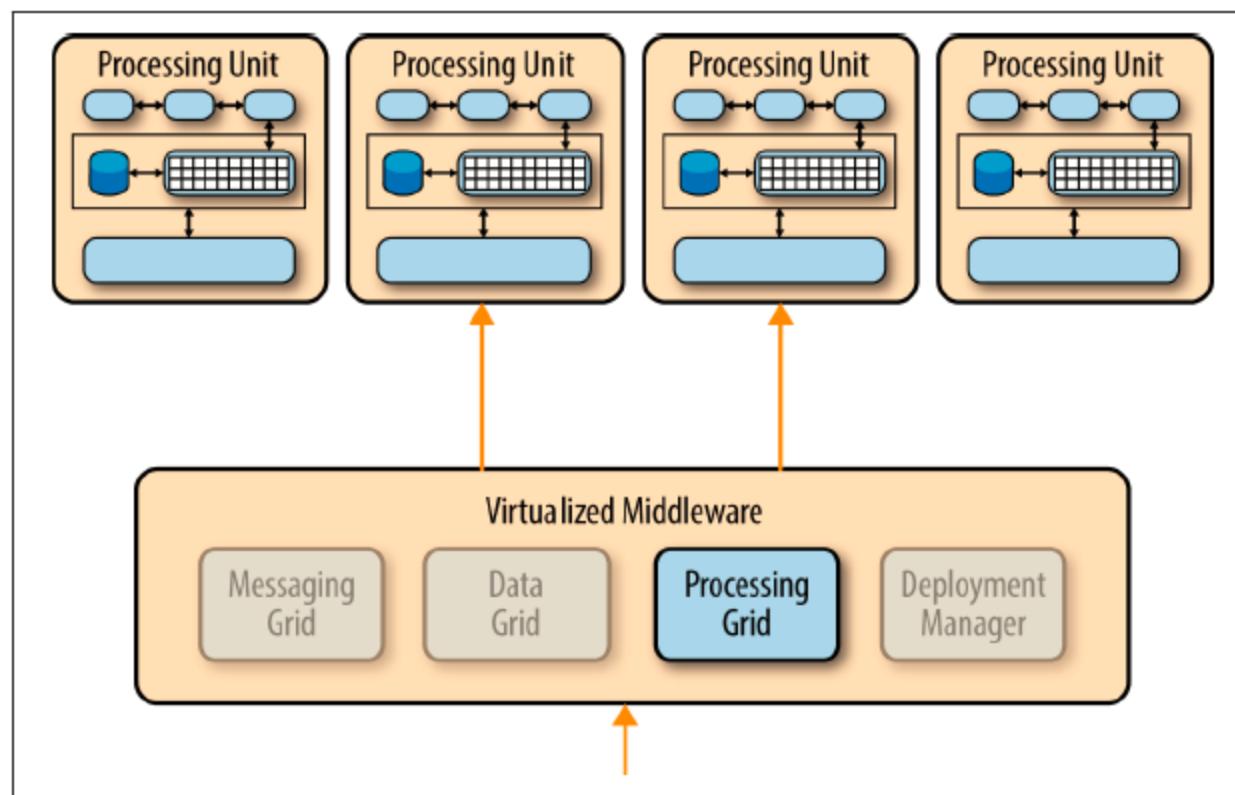
Data Grid

- Interacts with the data replication engine in each PU
- Parallel asynchronous data replication



Processing Grid

- Is optional
- Manages distributed request processing when there are multiple PUs, each handling a portion of the application (i.e. mediates and orchestrates)



Deployment Manager

- manages the dynamic startup and shutdown of PUs based on load conditions
- continually monitors response times and user loads, and starts up new PUs when load increases, and shuts down PUs when the load decreases.

Considerations

- Good for
 - Smaller web-based apps with variable load (i.e. social media sites, bidding and auction sites, etc)
- Not Good for
 - Traditional large-scale relational database apps with large amount of operational data

<i>Non functional req.</i>	<i>Rating</i>
Overall agility	High
Ease of deployment	High
Testability	Low
Performance	High
Scalability	High
Ease of development	Low

Architectures summary

	Layered	Event-driven	Microkernel	Microservices	Space-based
Overall Agility	↓	↑	↑	↑	↑
Deployment	↓	↑	↑	↑	↑
Testability	↑	↓	↑	↑	↓
Performance	↓	↑	↑	↓	↑
Scalability	↓	↑	↓	↑	↑
Development	↑	↓	↓	↑	↓

SOFTWARE DESIGN

Layers detailed

Content

- Patterns for Enterprise Application Architecture [Fowler]
 - Domain Layer Patterns
 - Transaction Script
 - Domain Model
 - Table Module
 - Active Record
 - Data Source Patterns
 - Row Data Gateway
 - Table Data Gateway
 - Data Mapper
 - Presentation
 - Template View
 - Transform View
 - Page Controller
 - Front Controller

References

- Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]
- Univ. of Aarhus Course Materials
- Univ. of Utrecht Course Materials

Patterns for Enterprise Applications [Fowler]

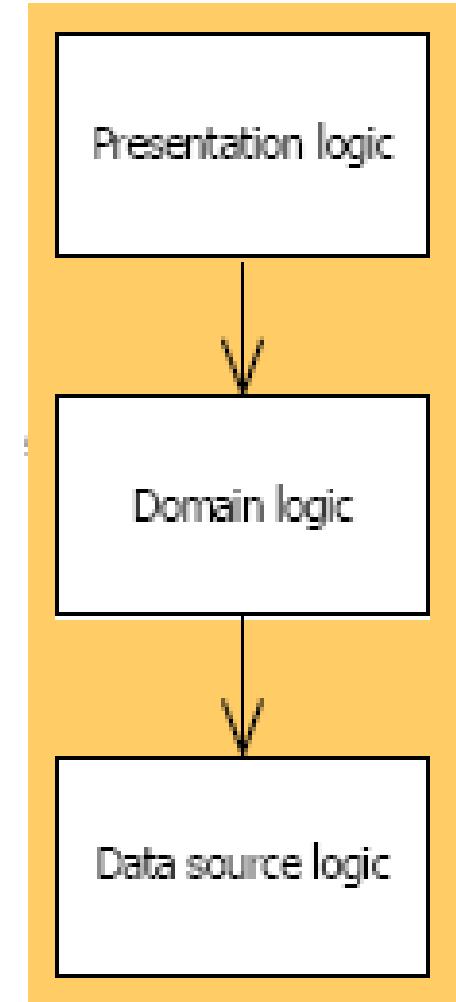
- Persistent data
- Volume of data
- Concurrent access
- Complicated user interface
- Integration with other applications
 - Conceptual dissonance
- Business logic

Enterprise applications

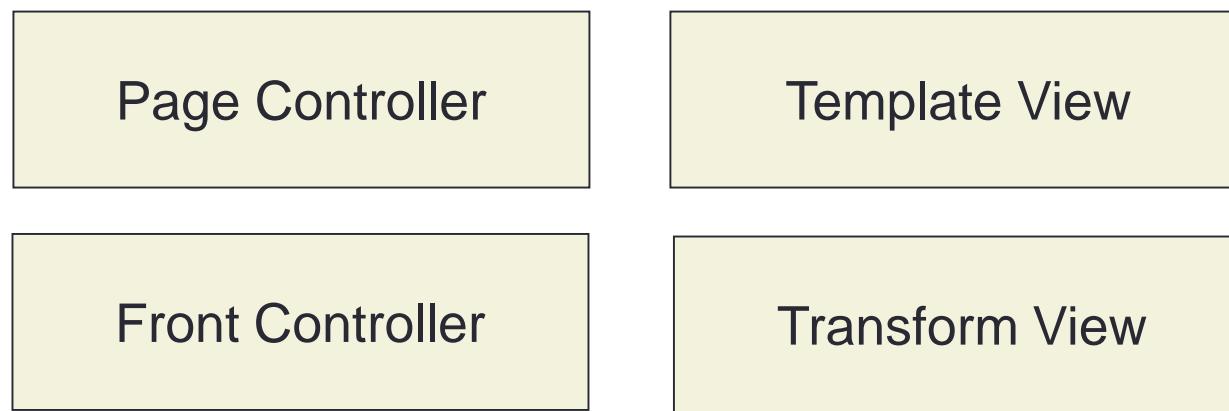
- Example: B2C online retailer
 - High volume of users: scalability
- Example: processing of leasing agreements
 - Complicated business logic
 - Rich-client interface
 - Complicated transaction behavior
- Example: expense tracking for small company

Principal layers

- See pattern Layers in [POSA]
- Here: applied to enterprise applications
- Presentation logic
 - Interaction with user
 - Command-line or rich client or Web interface
- Domain logic
 - Validation of input and calculation of results
- Data source logic
 - Communication with database and other applications



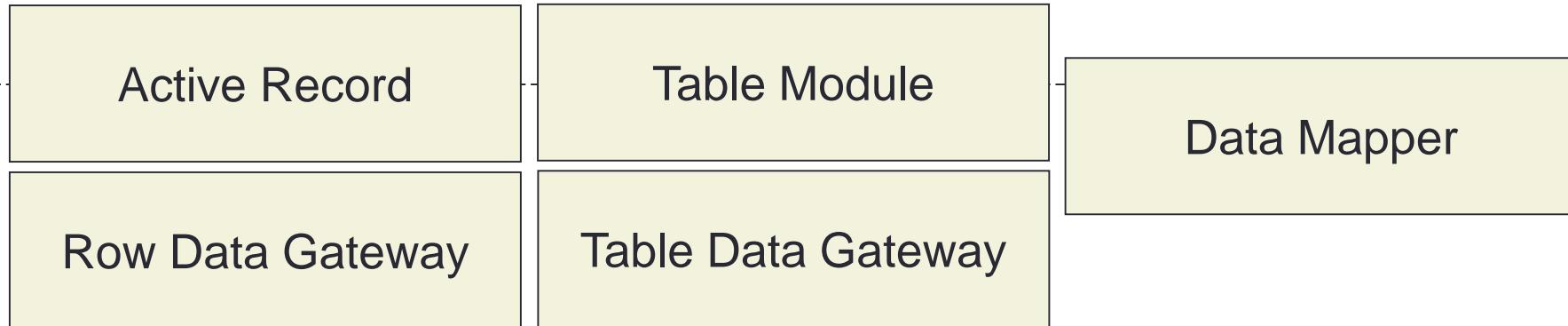
Presentation



Domain



Data Source

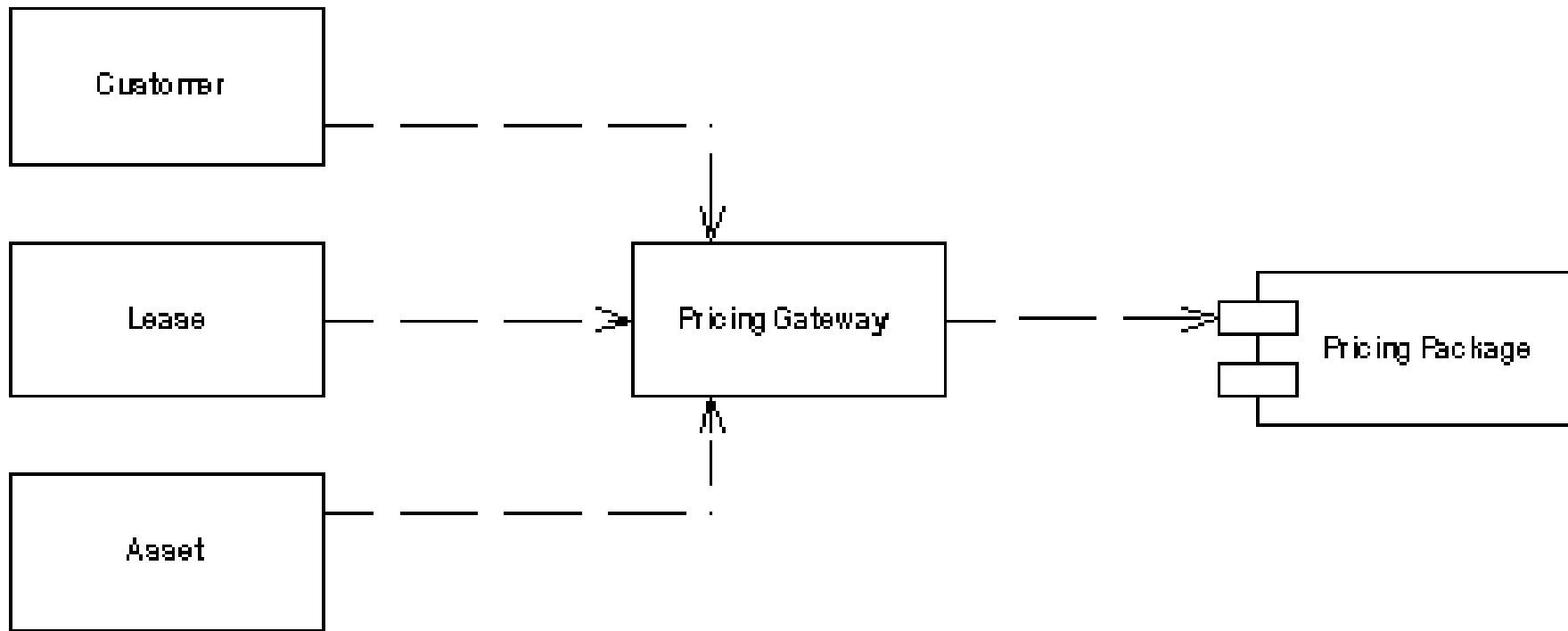


Basic Patterns

- Gateway
- Record Set

Gateway

An object that encapsulates access to an external system or resource



Gateway – How it works

- External resources - each with its own API
- Wraps all the API's into a common Gateway
- Sometimes, better to have 2 objects:
 - Back-end - acts as a minimal overlay to the external resource and does not simplify the API of the external resource at all
 - Front-end - transforms the awkward API into one that's more convenient for your application to use.

Gateway - Benefits

- Easier handling of awkward API's
- Easier to swap out one kind of resource for another
- Easier to test by giving you a clear point to deploy *Service Stubs* (*A stand-in implementation of an external service*)

Gateway - Example

build a gateway to an interface that just sends a message using the message service

```
int send(String messageType, Object[] args);
```

Confirmation message

```
messageType = 'CONFIRM';  
args[0] = id;  
args[1] = amount;  
args[2] = symbol;
```

Better...

```
public void sendConfirmation(String orderID, int amount, String symbol);
```

```
class Order...
    public void confirm() {
        if (isValid()) Environment.getMessageGateway().sendConfirmation(id, amount, symbol);
    }
```

```
class MessageGateway...
    protected static final String CONFIRM = "CNFRM";
    private MessageSender sender;

    public void sendConfirmation(String orderID, int amount, String symbol) {
        Object[] args = new Object[]{orderID, new Integer(amount), symbol};
        send(CONFIRM, args);
    }

    private void send(String msg, Object[] args) {
        int returnCode = doSend(msg, args);
        if (returnCode == MessageSender.NULL_PARAMETER)
            throw new NullPointerException("Null Parameter bassed for msg type: " + msg);
        if (returnCode != MessageSender.SUCCESS)
            throw new IllegalStateException(
                "Unexpected error from messaging system #: " + returnCode);
    }

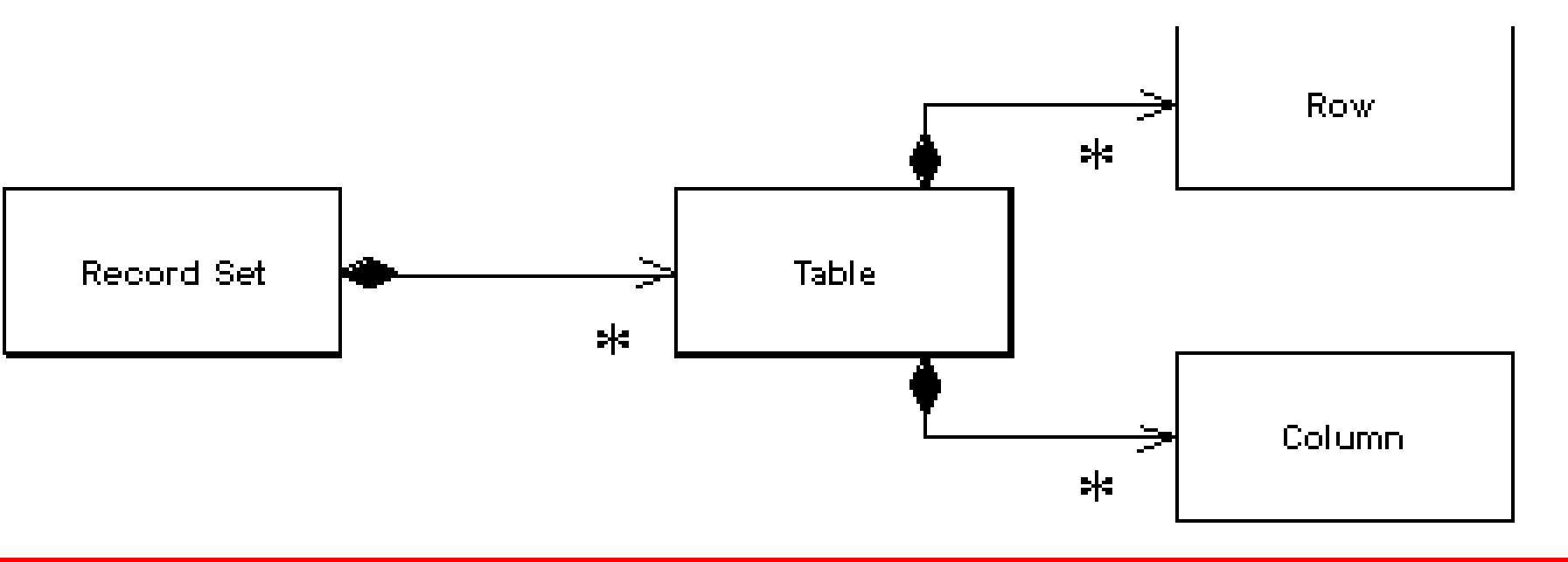
    protected int doSend(String msg, Object[] args) {
        Assert.notNull(sender);
        return sender.send(msg, args);
    }
```

Gateway vs. Façade vs. Adapter

- The *facade* is usually done by the **writer of the service** for general use, while a *Gateway* is written by the **client** for their particular use.
- A facade always implies a different interface to what it's covering, while a *Gateway* may copy the wrapped interface entirely, being used for substitution or testing purposes
- *Adapter* alters an implementation's interface to match another interface which you need to work with.
- With *Gateway* there usually isn't a an existing interface, although you might use an adaptor to map an implementation to a an existing *Gateway* interface. In this case the adaptor is part of the implementation of the *Gateway*

Record Set

An in-memory representation of tabular data



Record Set – how it works

- provides an in memory structure that looks exactly like the result of a SQL query, but can be generated and manipulated by other parts of the system.
- Examples:
 - DataSet of ADO.NET
 - RowSet of JDBC

A disconnected Record Set is one that is separated from its link to the data source

Domain Logic (Layer)

- “... also referred to as *business logic*. ... It involves *calculations* based on inputs and stored data, *validation* of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch ...” [Fowler]

Organizing the Domain Logic

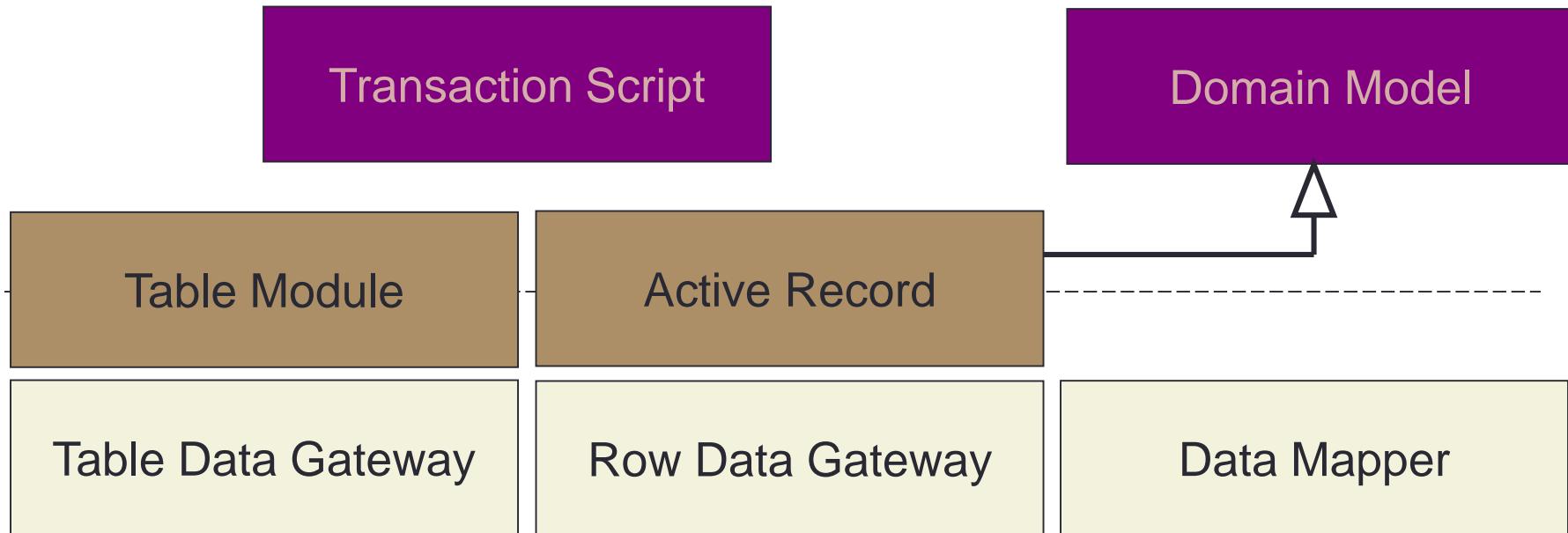
- Key architectural decisions, which influence structure of other layers.
- Pure patterns
 - Transaction Script
 - Domain Model
- Hybrid patterns
 - Active Record
 - Table Module.

Domain Logic Patterns

Presentation



Domain



Data Source

Transaction Script

Fowler: A TS organizes the business logic primarily as a single procedure where each procedure handles a single request from the presentation.

The TS may make calls directly to the DB or through a thin DB wrapper.

Think of a script for: a use case or business transaction.

Transaction Script

- ... is essentially a procedure that takes the
 - input from the presentation,
 - processes it with validations and calculations,
 - stores data in the database,
 - (invokes any operations from other systems, and)
 - replies with more data to the presentation perhaps doing more calculation to help organize and format the reply data.

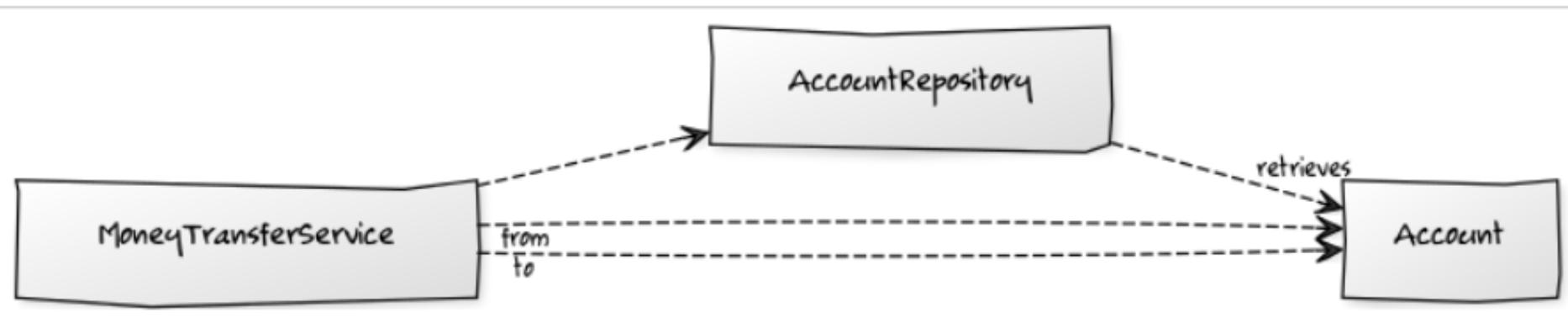
[Fowler]

TS Features

- Business logic is organized by procedures
- Each procedure handles a single transaction
- Transaction: well-defined endpoint
- Must be complete on all-or-nothing basis
- Makes call directly to the database
- May be organized as:
 - a separate class/TS (Command pattern)
 - several TS/class

Example (<http://lorenzo-dee.blogspot.ro/2014/06/quantifying-domain-model-vs-transaction-script.html>)

- Banking application
- Money transfer functionality



```
1 | public interface MoneyTransferService {  
2 |     BankingTransaction transfer(  
3 |         String fromAccountId, String toAccountId, double amount);  
4 | }
```

```
1 public class MoneyTransferServiceTransactionScriptImpl
2     implements MoneyTransferService {
3     private AccountDao accountDao;
4     private BankingTransactionRepository bankingTransactionRepository;
5     ...
6     @Override
7     public BankingTransaction transfer(
8         String fromAccountId, String toAccountId, double amount) {
9         Account fromAccount = accountDao.findById(fromAccountId);
10        Account toAccount = accountDao.findById(toAccountId);
11        ...
12        double newBalance = fromAccount.getBalance() - amount;
13        switch
14        case NEVER:
15            if (
16                th
17            ) 3 } }
18        break;
19    case ALLOWED:
20        if (newBalance < -limit) {
21            throw new DebitException(
22                "Overdraft limit (of " + limit + ") exceeded: " + newBalance);
23        }
24        break;
25    }
26    fromAcc
27    toAccou
28    Banking
29    new
30    banking
31    return
32    }
33 }
```

```
1 // @Entity
2 public class Account {
3     // @Id
4     private String id;
5     private double balance;
6     private OverdraftPolicy overdraftPolicy;
7     ...
8     public String getId() { return id; }
9     public void setId(String id) { this.id = id; }
10    public double getBalance() { return balance; }
11    public void setBalance(double balance) { this.balance = balance; }
12    public OverdraftPolicy getOverdraftPolicy() { return overdraftPolicy; }
13    public void setOverdraftPolicy(OverdraftPolicy overdraftPolicy) {
14        this.overdraftPolicy = overdraftPolicy;
15    }
16 }
```

Analysis

- Strengths
 - Simplicity
- Weaknesses
 - complicated transaction logic
 - duplicated logic

Domain Model (EA Pattern)

Fowler: An object model of the domain that incorporates both behaviour and data.

A DM creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line in an order form.

Domain Model (EA Pattern)

- Realization (via design classes) of UML Domain Model (conceptual classes).
 - E.g. person, book, shopping cart, task, sales line item,
...
- Domain Model classes contain Logic for handling validations and calculations.
 - E.g. a shipment object calculates the shipping charge for a delivery.

DM Features

- Business logic is organized as an OO model of the domain
 - Describes both data and behavior
 - Different from database model
 - Process, multi-valued attributes, inheritance, design patterns
 - Harder to map to the database
- Risk of bloated domain objects

```
1 public class MoneyTransferServiceDomainModelImpl
2     implements MoneyTransferService {
3     private AccountRepository accountRepository;
4     private BankingTransactionRepository bankingTransactionRepository;
5     ...
6     @Override
7     public BankingTransaction transfer(
8         String fromAccountId, String toAccountId, double amount) {
9         Account fromAccount = accountRepository.findById(fromAccountId);
10        Account toAccount = accountRepository.findById(toAccountId);
11        ...
12        fromAccount.debit(amount);
13        toAccount.credit(amount);
14        BankingTransaction moneyTransferTransaction =
15            new MoneyTransfer(1 // @Entity
16                bankingTransactionRep(2
17                    return moneyTransfer(3
18                }
19            }
<
```

```
1 // @Entity
2 public class Account {
3     // @Id
4     private String id;
5     private double balance;
6     private OverdraftPolicy overdraftPolicy;
7     ...
8     public double balance() { return balance; }
9     public void debit(double amount) {
10         this.overdraftPolicy.preDebit(this, amount);
11         this.balance = this.balance - amount;
12         this.overdraftPolicy.postDebit(this, amount);
13     }
14     public void credit(double amount) {
15         this.balance = this.balance + amount;
16     }
17 }
```

```
1 public interface OverdraftPolicy {  
2     void preDebit(Account account, double amount);  
3     void postDebit(Account account, double amount);  
4 }  
  
1 public class NoOverdraftAllowed implements OverdraftPolicy {  
2     public void preDebit(Account account, double amount) {  
3         double newBalance = account.balance() - amount;  
4         if (newBalance < 0) {  
5             throw new DebitException("Insufficient funds");  
6         }  
7     }  
8     public void postDebit(Account account, double amount) {  
9     }  
10 }
```

```
1 public class LimitedOverdraft implements OverdraftPolicy {  
2     private double limit;  
3     . . .  
4     public void preDebit(Account account, double amount) {  
5         double newBalance = account.balance() - amount;  
6         if (newBalance < -limit) {  
7             throw new DebitException(  
8                 "Overdraft limit (of " + limit + ") exceeded: " + newBalance);  
9         }  
10     }  
11     public void postDebit(Account account, double amount) {  
12     }  
13 }
```

Choosing a Domain Logic Pattern

- Which one to choose?
 - Influenced by the complexity of domain logic.
- Application is simple access to data sources
 - Transaction Script, (or Active Record, Table Module)
- Significant amount of business logic
 - Domain Model
- TS is simpler:
 - Easier and quicker to develop and maintain.
 - But can lead to duplication in logic / code.
- DM – difficult access to relational DB
- Which would be easier to refactor?
TS->DM or DM-> TS

Towards Data Source Patterns

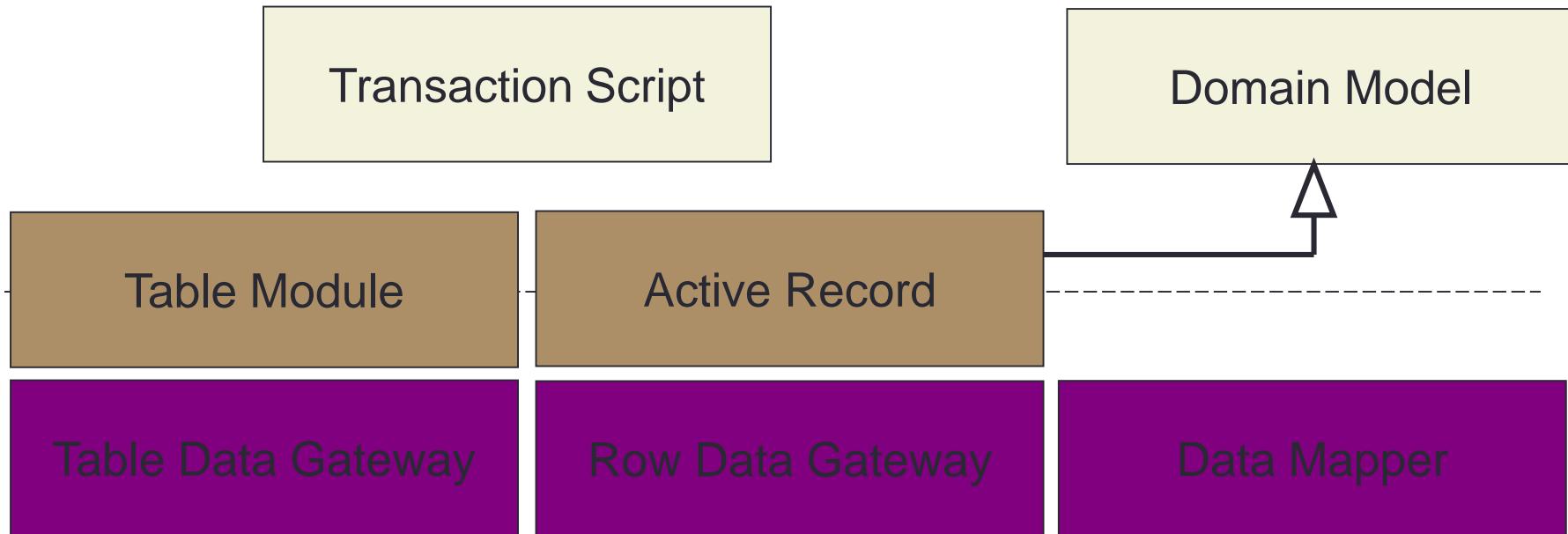
- Hybrid patterns.
 - Active Record
 - Table Module
- Pure patterns.
 - Row Data Gateway,
 - Table Data Gateway,
 - Data Mapper
 - ...

Data Source Patterns

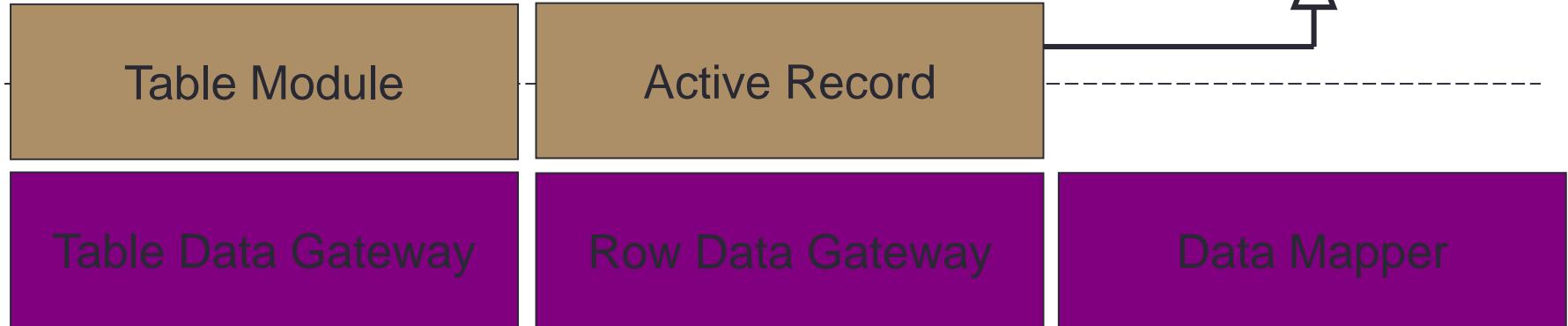
Presentation



Domain



Data Source



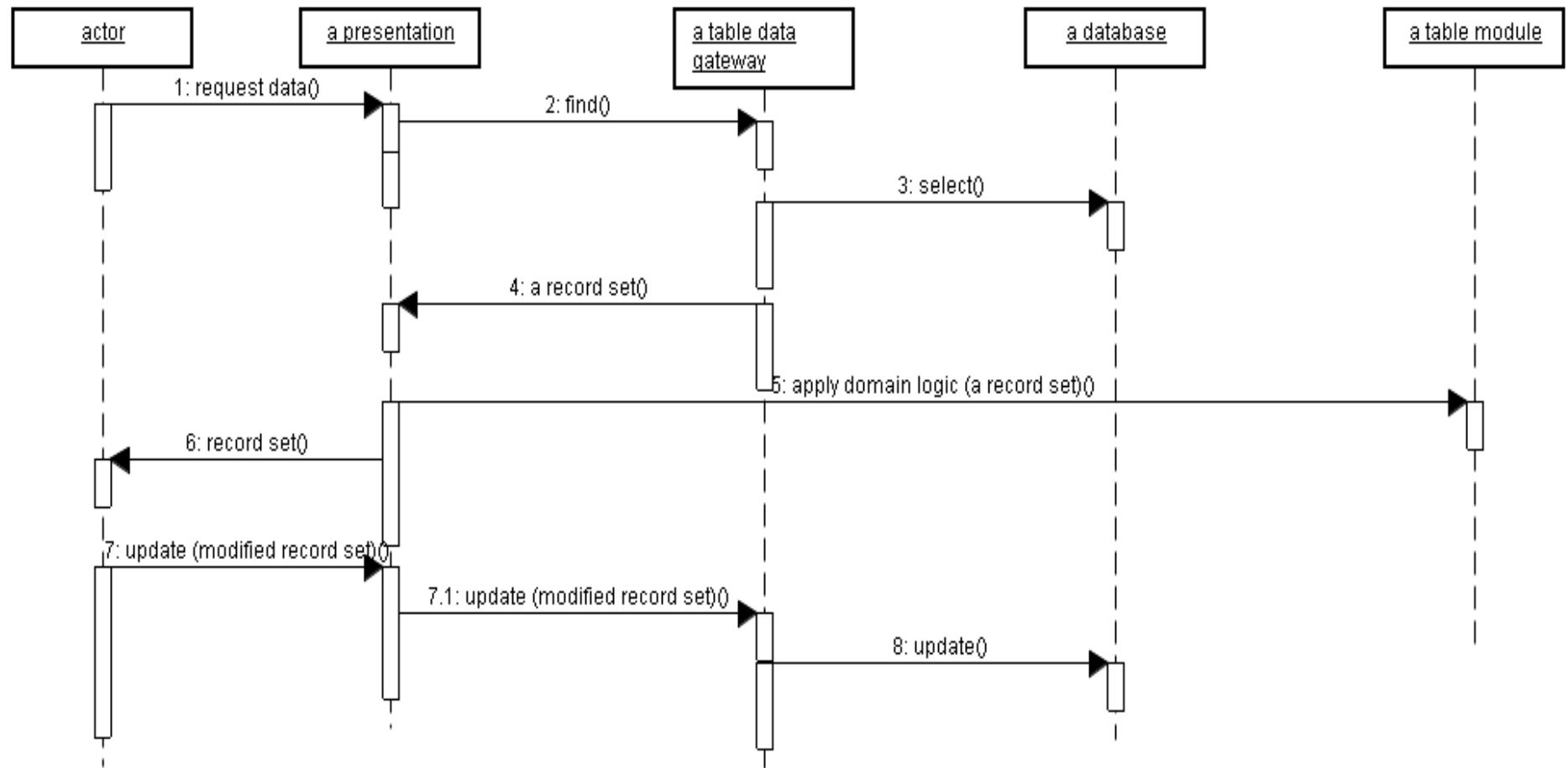
Row Data Gateway

Data Mapper

Table Module

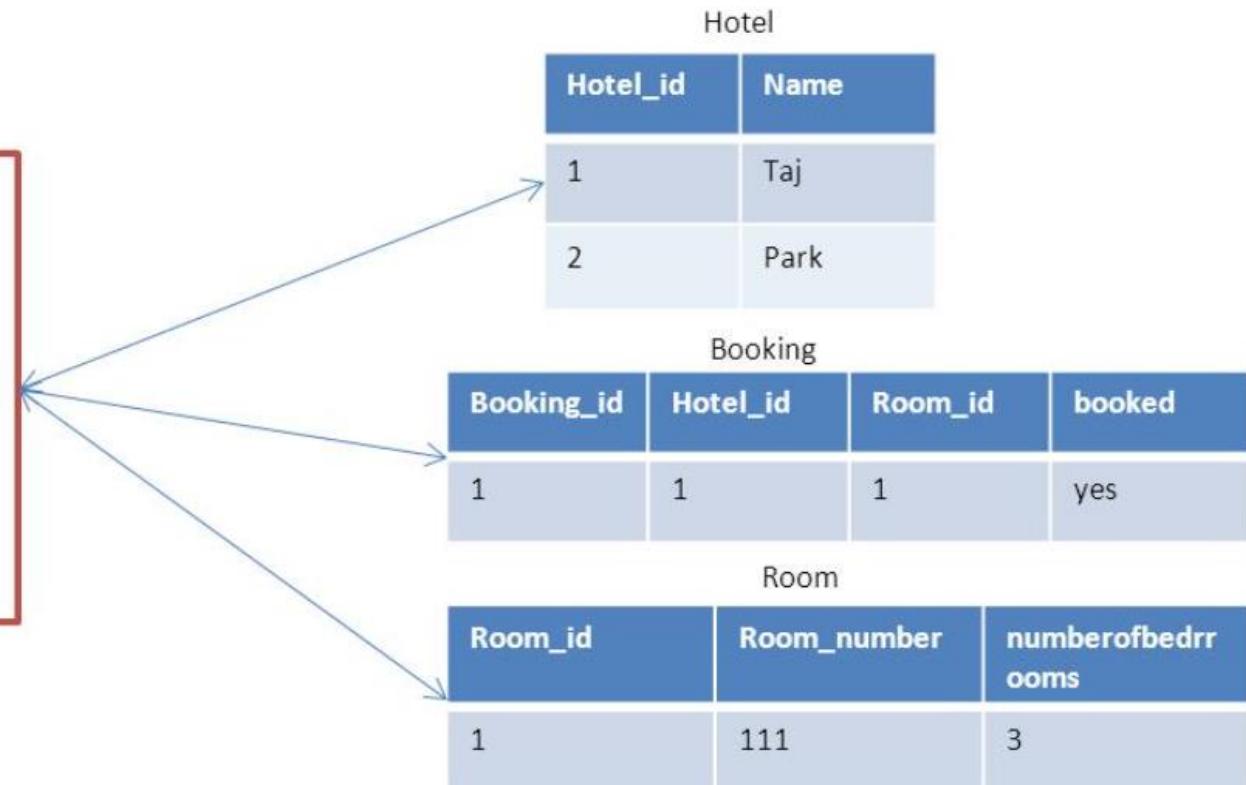
- Provide a single object for all the behavior on a table
- Organizes domain logic with **one class per table**
- *Table Module* has no notion of an identity for the objects that it's working with
⇒ Id references are necessary

Typical interactions for TM

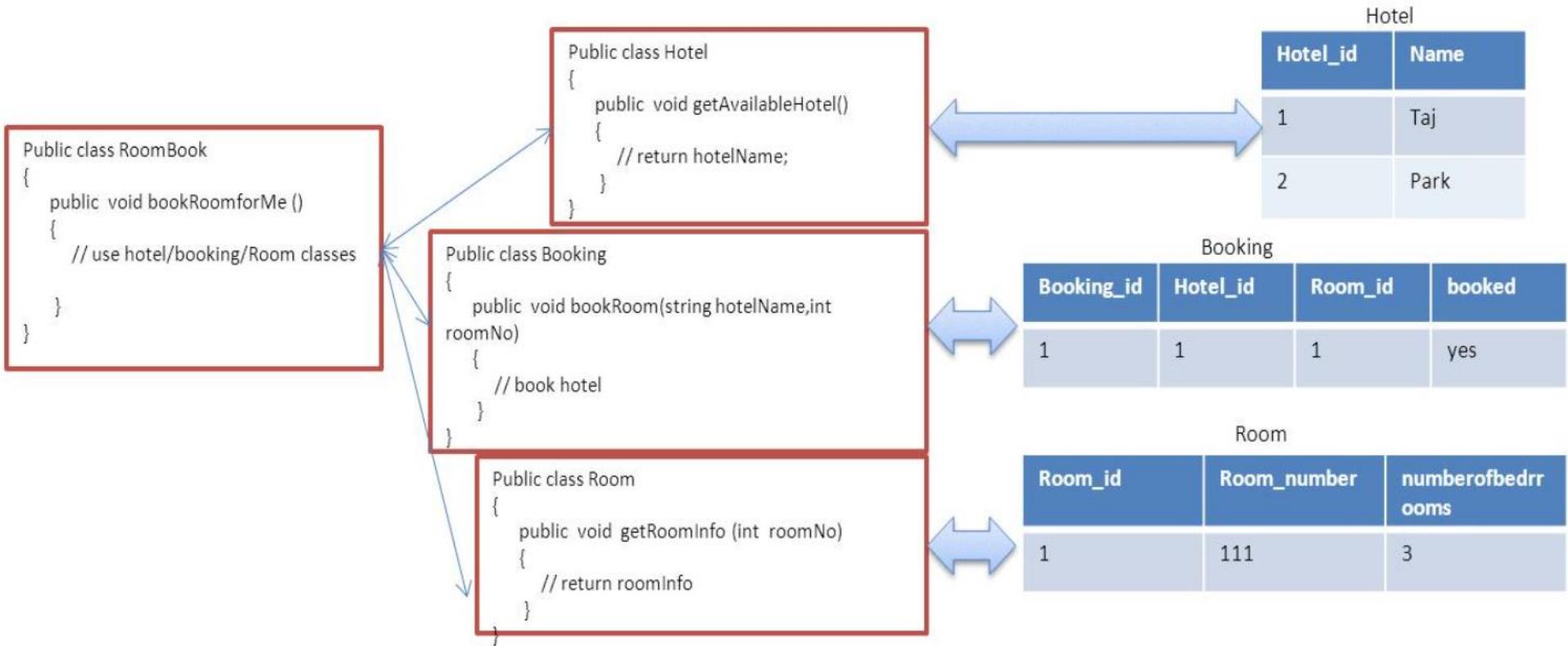


TS vs TM

```
Public class Hotel
{
    public void bookRoom(int roomNo)
    {
        // choose hotel
        //check availability
        //calculate price
        // book the room
        // Commit the Transaction
    }
}
```



TS vs TM



Example - Revenue Recognition (RR)

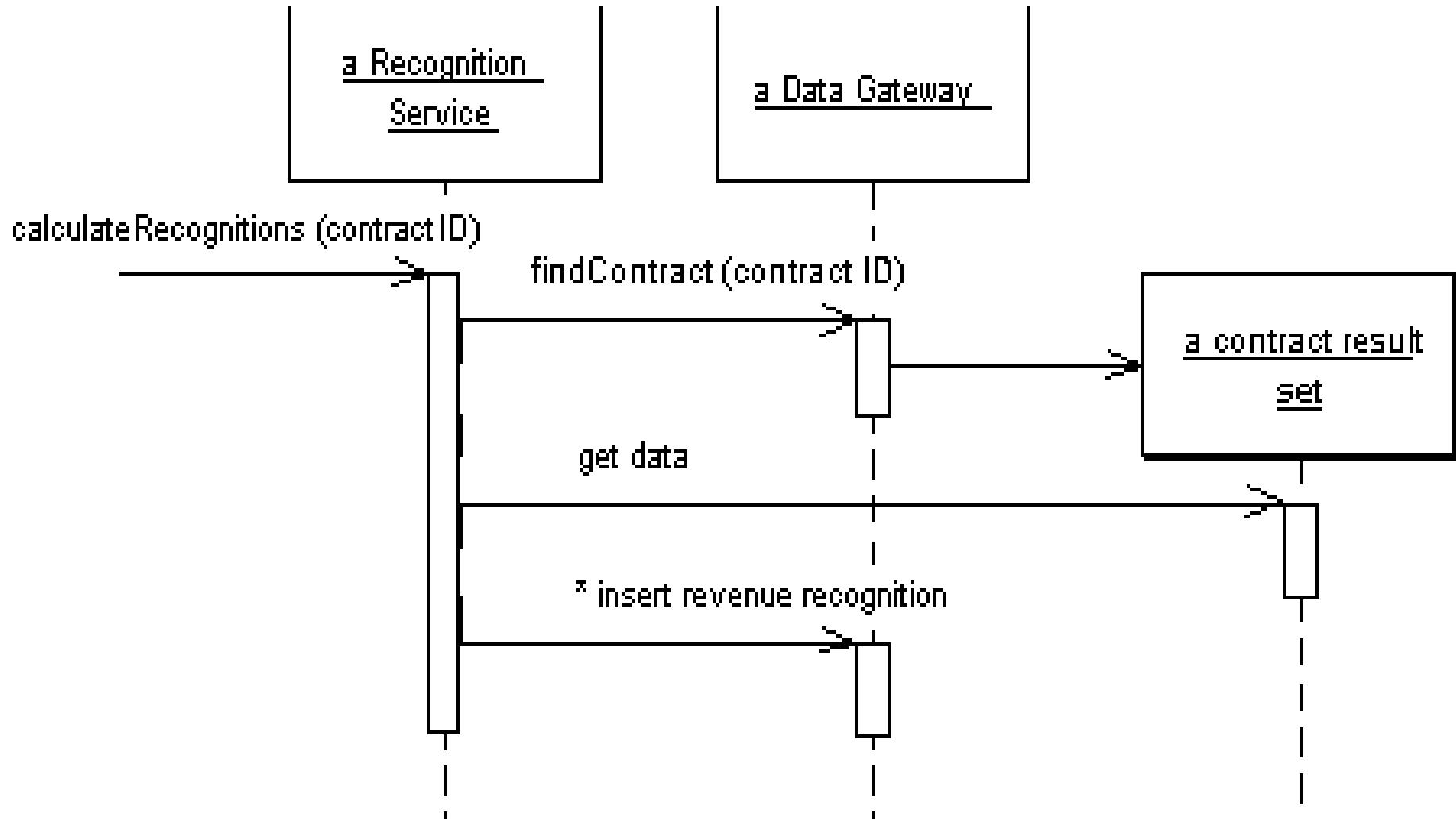
- Revenue recognition is a common problem in business systems.
 - when you can actually count the money you receive on your accounting books.
- E.g. selling a S/W package \$120 today
 - Book \$40 today,
 - \$40 in 30 days,
 - \$40 in 60 days.

[Fowler]

Revenue Recognition concepts

- Product type: description of item to be sold
- Contract: covers only one product.
- Revenue recognition: varies per product type.
- For each product instance: a set of revenue recognition instances

TS: Calculating Revenue Recognitions



Implementation

- Database

```
CREATE TABLE products (ID int primary key,  
name varchar, type varchar)
```

```
CREATE TABLE contracts (ID int primary key,  
product int, revenue decimal, dateSigned  
date)
```

```
CREATE TABLE revenueRecognitions (contract  
int, amount decimal, recognizedOn date,  
PRIMARY KEY (contract, recognizedOn))
```

Implementation

- calculate the amount of recognition due by a particular day:
 - select the appropriate rows in the revenue recognitions table,
 - sum up the amounts.

Gateway class

```
class Gateway...  
    public ResultSet findRecognitionsFor(long contractID,  
                                         MfDate asof) throws SQLException  
{  
    PreparedStatement stmt =  
        db.prepareStatement(findRecognitionsStatement);  
  
    stmt.setLong(1, contractID);  
    stmt.setDate(2, asof.toSqlDate());  
  
    ResultSet result = stmt.executeQuery();  
    return result; }  
  
private static final String findRecognitionsStatement  
= "SELECT amount " + " FROM revenueRecognitions " + "  
WHERE contract = ? AND recognizedOn <= ?";  
private Connection db;
```

RecognitionService class

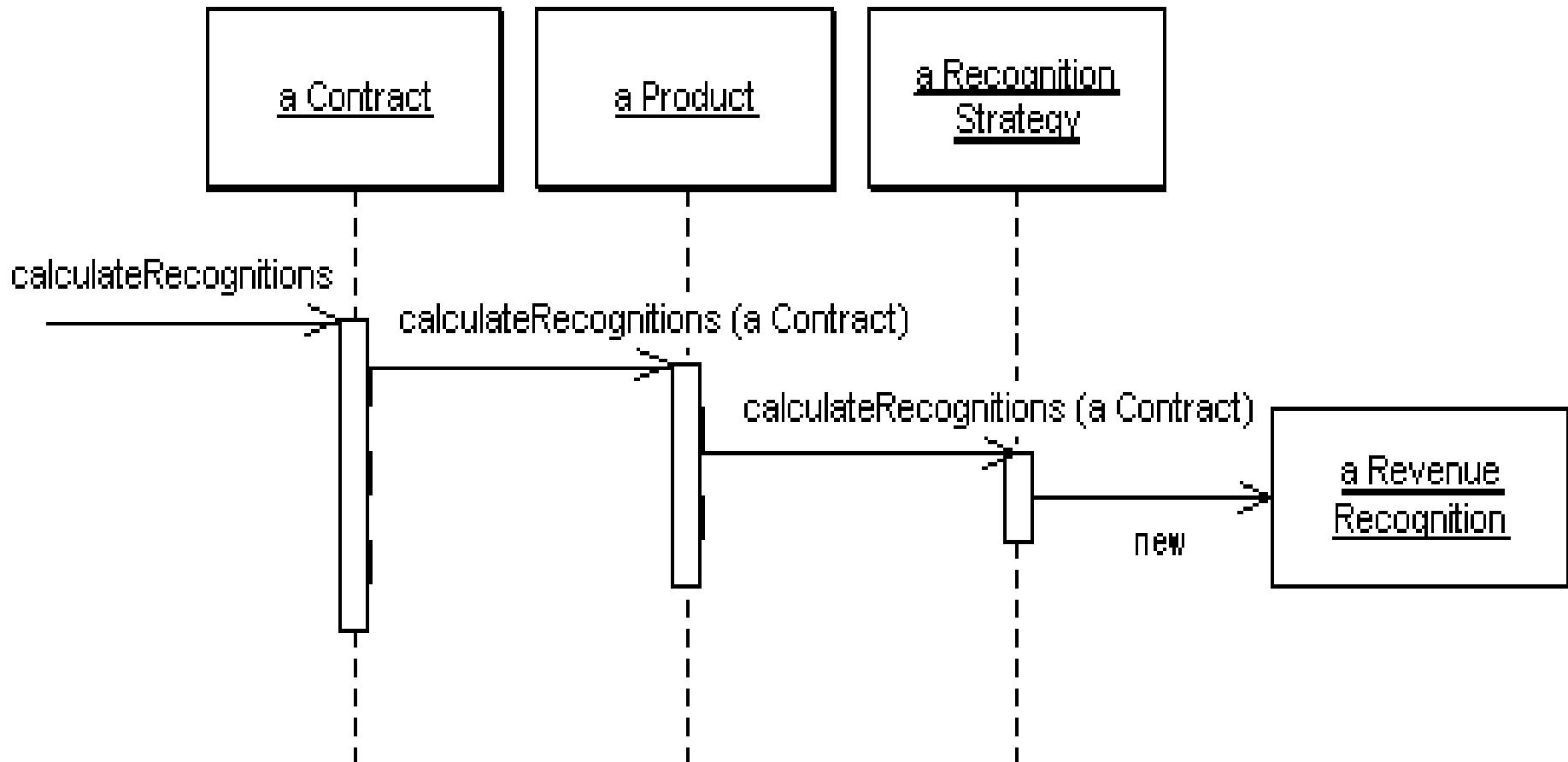
```
class RecognitionService...
public Money recognizedRevenue(long contractNumber,
    MfDate asOf)
{
    Money result = Money.dollars(0);
    try {
        ResultSet rs =
            db.findRecognitionsFor(contractNumber, asOf);
        while (rs.next())
        {
            result =
            result.add(Money.dollars(rs.getBigDecimal("amount")));
        }
        return result;
    }
    catch (SQLException e) {
        throw new ApplicationException (e);  }
}
```

RR Domain Model



if a contract has any revenue recognitions then the contract's revenue should be equal to the sum of the amounts of its revenue recognitions

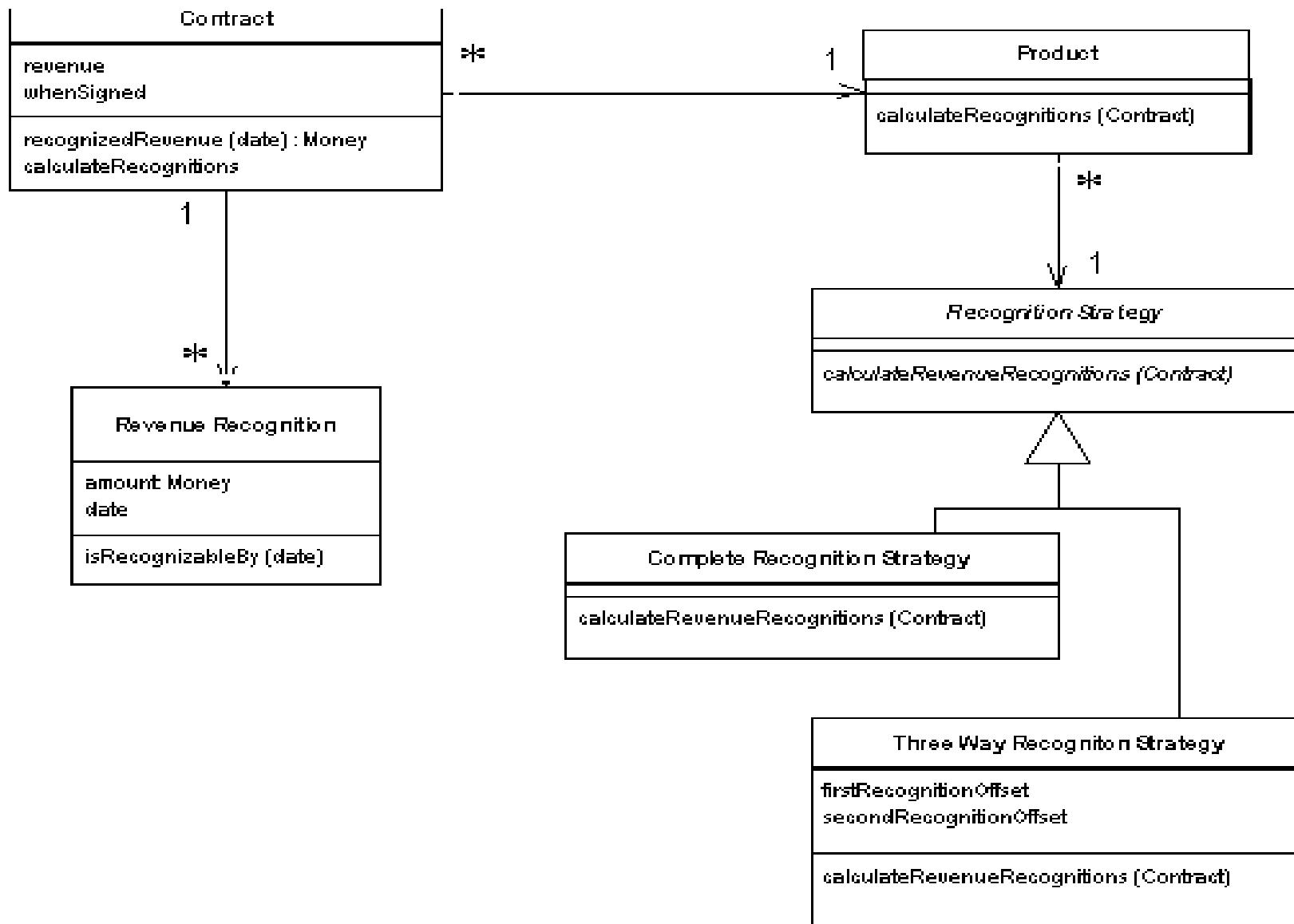
Domain Model: Calculating Revenue Recognitions



Enhancement: e.g. New Revenue Recognition Strategy

- Transaction Script:
 - New conditional, or
 - New subroutine.
- Domain Model:
 - Create new Rev. Recog. Strategy class.

RR updated Domain Model



Implementation

```
class RevenueRecognition...
    private Money amount;
    private MfDate date;
    public RevenueRecognition(Money amount, MfDate
date)
    {
        this.amount = amount;
        this.date = date;
    }
    public Money getAmount()
    {
        return amount;
    }
    boolean isRecognizableBy(MfDate asOf)
    {
        return asOf.after(date) || asOf.equals(date);
    }
```

Contract class

```
class Contract...
    private List revenueRecognitions = new
ArrayList();
    public Money recognizedRevenue(MfDate asOf)
{
    Money result = Money.dollars(0);
    Iterator it =
revenueRecognitions.iterator();
    while (it.hasNext())
    {
        RevenueRecognition r =
(RevenueRecognition) it.next();
        if (r.isRecognizableBy(asOf))
            result = result.add(r.getAmount());
    }
    return result;
}
```

Introducing strategies...

```
class Contract...  
    private Product product;  
    private Money revenue;  
    private MfDate whenSigned;  
    private Long id;  
    public Contract(Product product, Money  
revenue, MfDate whenSigned)  
{  
    this.product = product;  
    this.revenue = revenue;  
    this.whenSigned = whenSigned;  
}
```

Introducing strategies ...

```
class Product...
    private String name;
    private RecognitionStrategy recognitionStrategy;
    public Product(String name, RecognitionStrategy
recognitionStrategy)
    {
        this.name = name;
        this.recognitionStrategy = recognitionStrategy;
    }
    public static Product newWordProcessor(String name)
    {
        return new Product(name, new
CompleteRecognitionStrategy());
    }
    public static Product newSpreadsheet(String name)
    {
        return new Product(name, new
ThreeWayRecognitionStrategy(60, 90));
    }
    public static Product newDatabase(String name)
    {
        return new Product(name, new
ThreeWayRecognitionStrategy(30, 60)); }
```

Introducing strategies ...

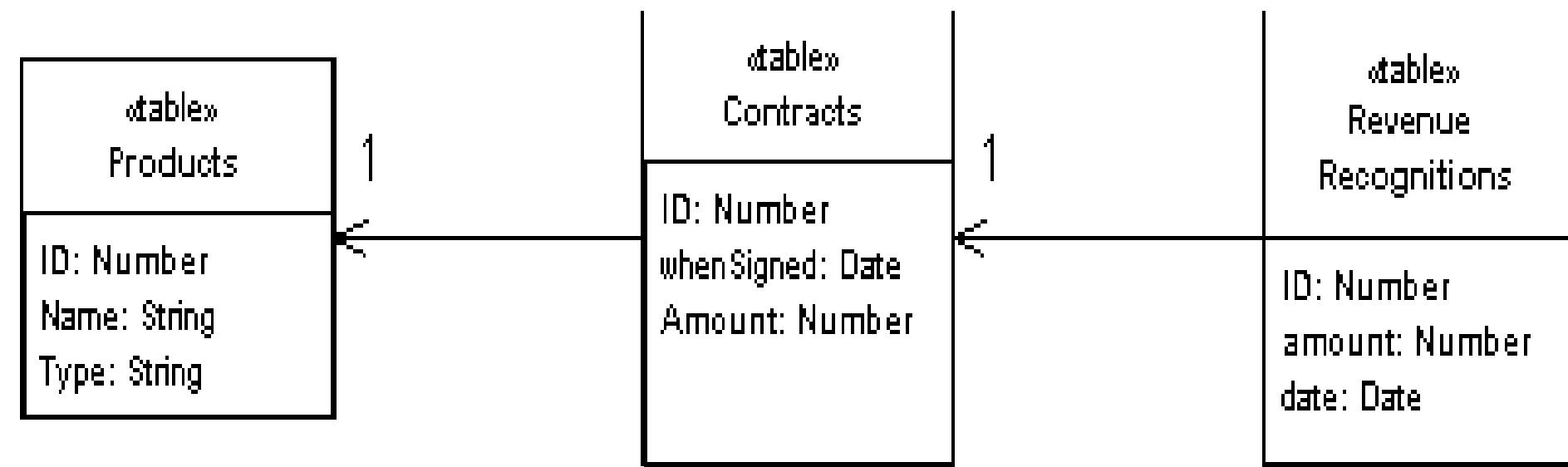
```
class RecognitionStrategy...{  
    abstract void  
    calculateRevenueRecognitions(Contract  
    contract); }  
  
class CompleteRecognitionStrategy...  
    void calculateRevenueRecognitions(Contract  
    contract)  
    {  
        contract.addRevenueRecognition(new  
        RevenueRecognition(contract.getRevenue(),  
        contract.getWhenSigned()));  
    }  
  
class ThreeWayRecognitionStrategy...
```

Introducing strategies

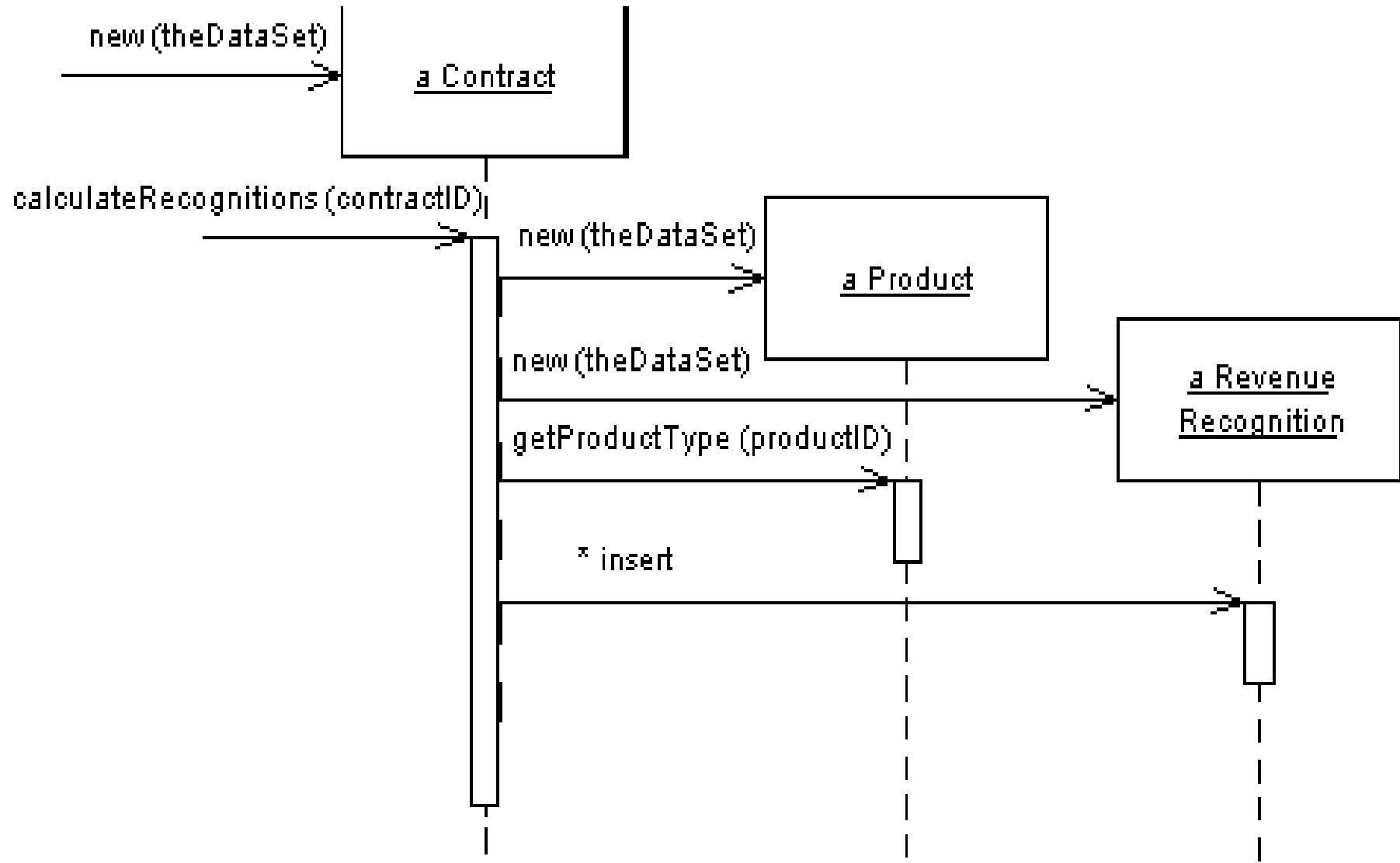
```
class Contract...
    public void calculateRecognitions()
    {
        product.calculateRevenueRecognitions(this
    );
}

class Product...
    void calculateRevenueRecognitions(Contract
contract)
{
    recognitionStrategy.calculateRevenueRecog
nitions(contract);
}
```

TM in the RR example



RR problem with TM



Implementation (C#)

```
class TableModule...  
    protected DataTable table;  
    protected TableModule(DataSet ds,  
String tableName)  
{  
    table = ds.Tables[tableName];  
}
```

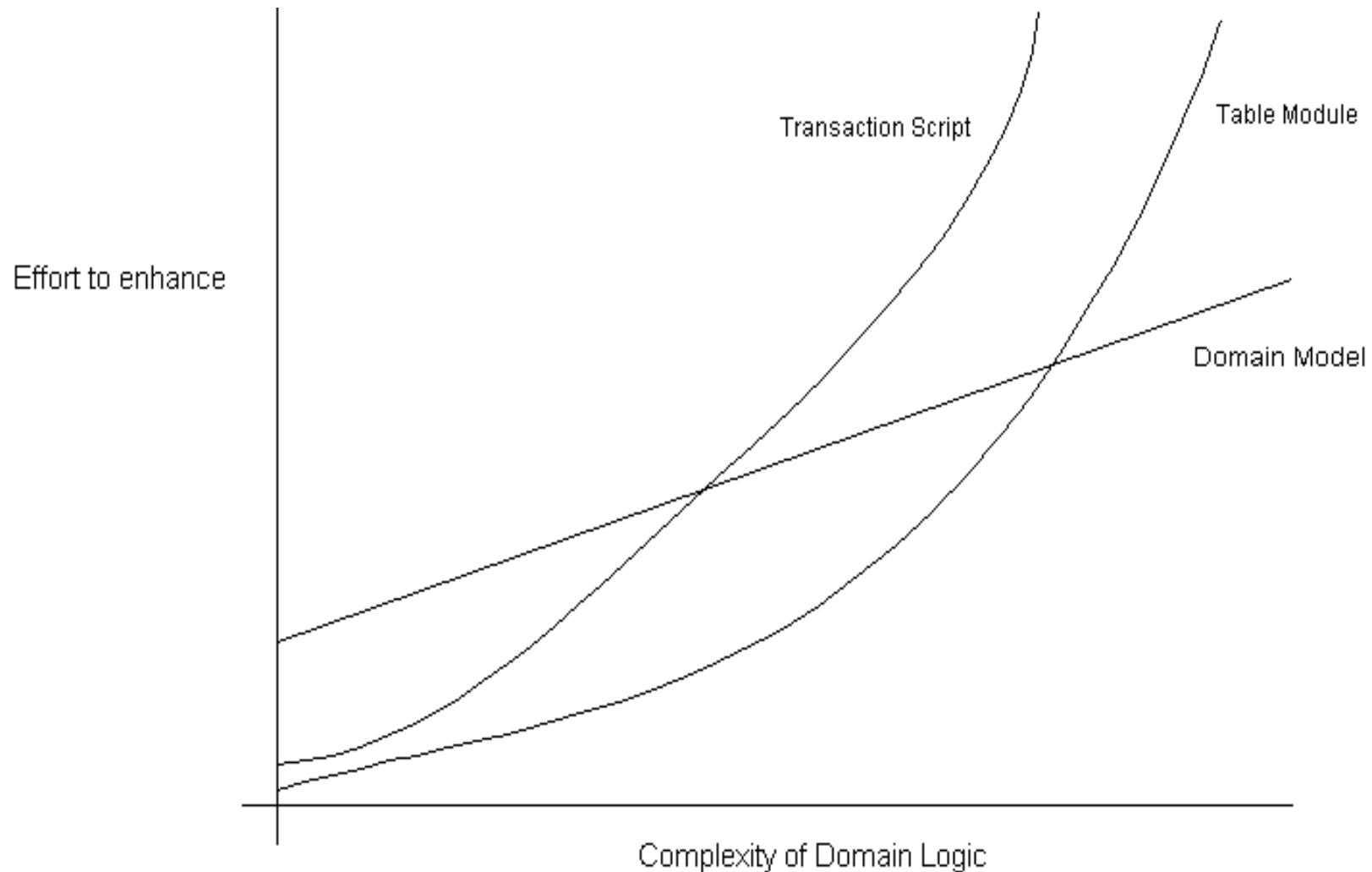
ContractModule subclass

```
class ContractModule...
    public ContractModule (DataSet ds) : base
        (ds, "Contracts") {}
    public DataRow this [long key]
    {
        get
        {
            String filter = String.Format("ID =
{0}", key);
            return table.Select(filter)[0];
        }
    }
contract = new ContractModule(dataset);
```

RevenueRecognition class

```
class RevenueRecognition...
{
    public Decimal RecognizedRevenue (long contractID,
    DateTime asOf)
    {
        String filter = String.Format("ContractID =
{0} AND date <= #{1:d}#", contractID, asOf);
        DataRow[] rows = table.Select(filter);
        Decimal result = 0;
        foreach (DataRow row in rows)
        {
            result += (Decimal) row["amount"];
        }
        return result;
    }
}
```

Which one to use?



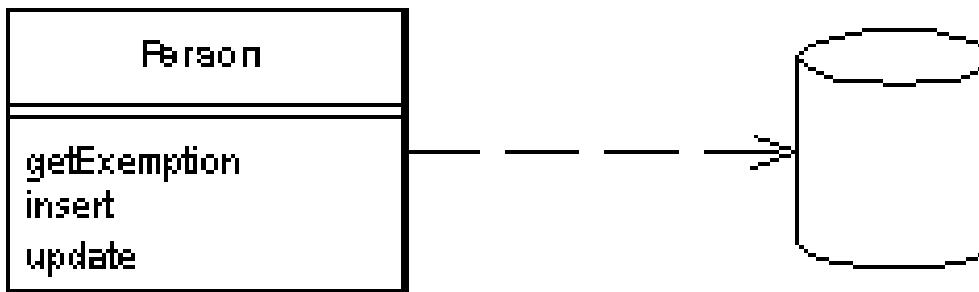
Active Record

Fowler: An object that wraps a record data structure in an external resource, such as a row in a database table, and adds some domain logic to that object.

An AR object carries both data and behaviour.

The essence of an AR is a Domain Model in which the classes match very closely the record structure of the underlying database.

Class operations



- construct an instance of the *Active Record* from a SQL result set row
- construct a new instance for later insertion into the table
- static finder methods to wrap commonly used SQL queries and return *Active Record* objects
- methods to update the database and insert into the database with the data in the *Active Record*
- getting and setting methods for the fields
- methods that implement some pieces of business logic

Implementation

```
class Person...  
private String lastName;  
private String firstName;  
private int numberofDependents;
```

```
create table people (ID int primary key,  
lastname varchar, firstname varchar,  
number_of_dependents int)
```

Find + Load an object

```
class Person...
private final static String findStatementString = "SELECT
id, lastname, firstname, number_of_dependents" + " FROM
people" + " WHERE id = ?";
public static Person find(Long id)
{
    Person result = (Person) Registry.getPerson(id);
    if (result != null) return result;
    PreparedStatement findStatement = null;
    ResultSet rs = null;
    try
    {
        findStatement = DB.prepare(findStatementString);
        findStatement.setLong(1, id.longValue());
        rs = findStatement.executeQuery();
        rs.next();
        result = load(rs);
        return result;
    } catch (SQLException e) { throw new
ApplicationException(e); }
    finally { DB.cleanUp(findStatement, rs); }
}
```

```
public static Person load(ResultSet rs)
throws SQLException
{
    Long id = new Long(rs.getLong(1));
    Person result = (Person)
Registry.getPerson(id);
    if (result != null) return result;
    String lastNameArg = rs.getString(2);
    String firstNameArg = rs.getString(3);
    int numDependentsArg = rs.getInt(4);
    result = new Person(id, lastNameArg,
firstNameArg, numDependentsArg);
    Registry.addPerson(result);
    return result;
}
```

Next Time

- More patterns

SOFTWARE DESIGN

Data Source & Concurrency patterns

Content

- Patterns for Enterprise Application Architecture [Fowler] – continued
 - Data Source Patterns
 - Handling Concurrency

References

- Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]
- Microsoft Application Architecture Guide, 2009 [MAAG]
- SaaS Course Stanford
- Paulo Sousa, Instituto Superior de Engenharia do Porto, Patterns of Enterprise Applications by example

Data Source Patterns

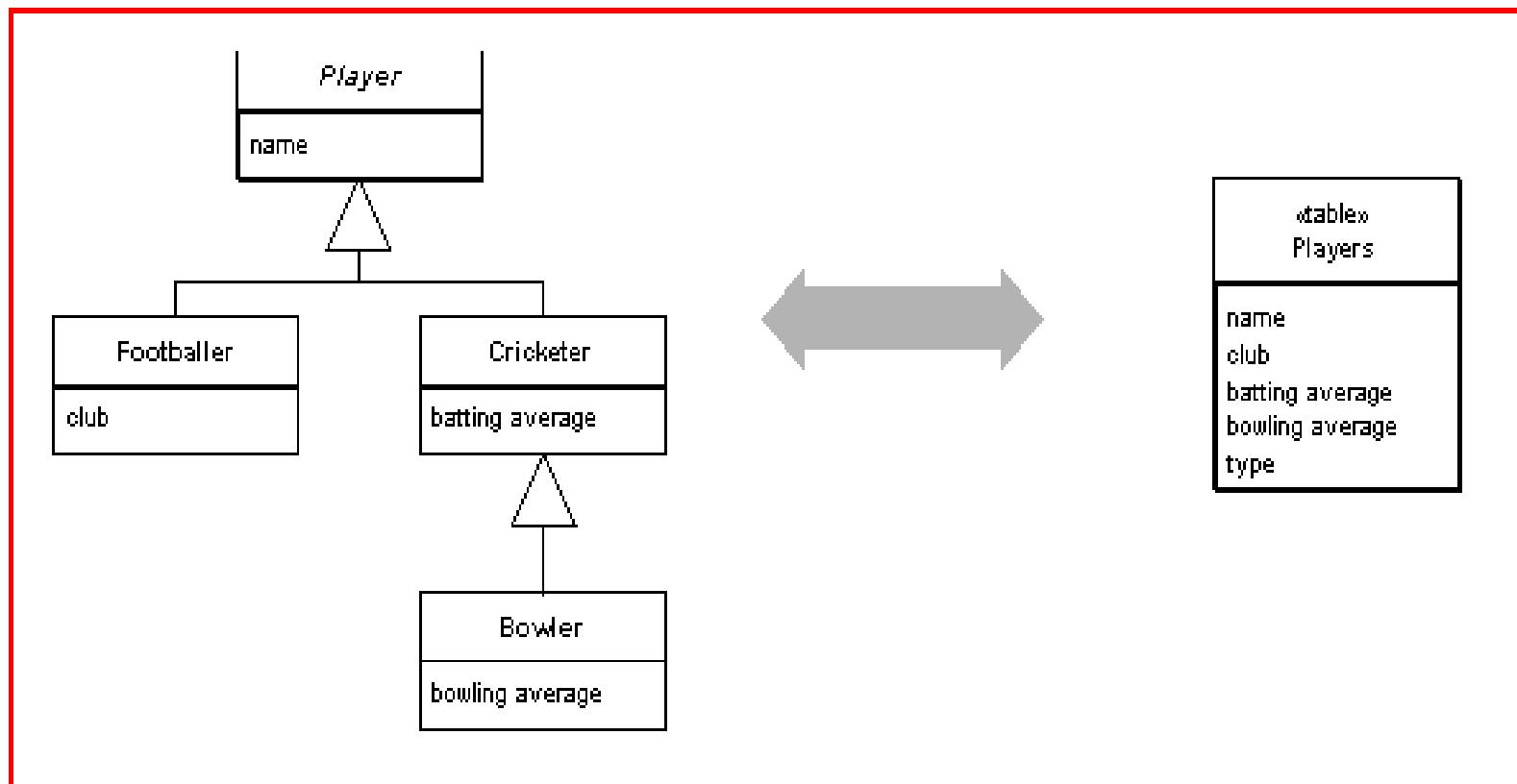
- Hybrid Data Source Pattern (discussed)
 - Active Record
 - Table Module
- Pure Data Source Patterns
 - Gateways
 - Row Data Gateway (RDG)
 - Table Data Gateway (TDG)
 - Data Mapper

Data Source Patterns

- Hide SQL.
- Provide an abstraction for
 - One data row.
 - A collection of data row(s).
- Object Relationships mapping problems
 - Links
 - Inheritance

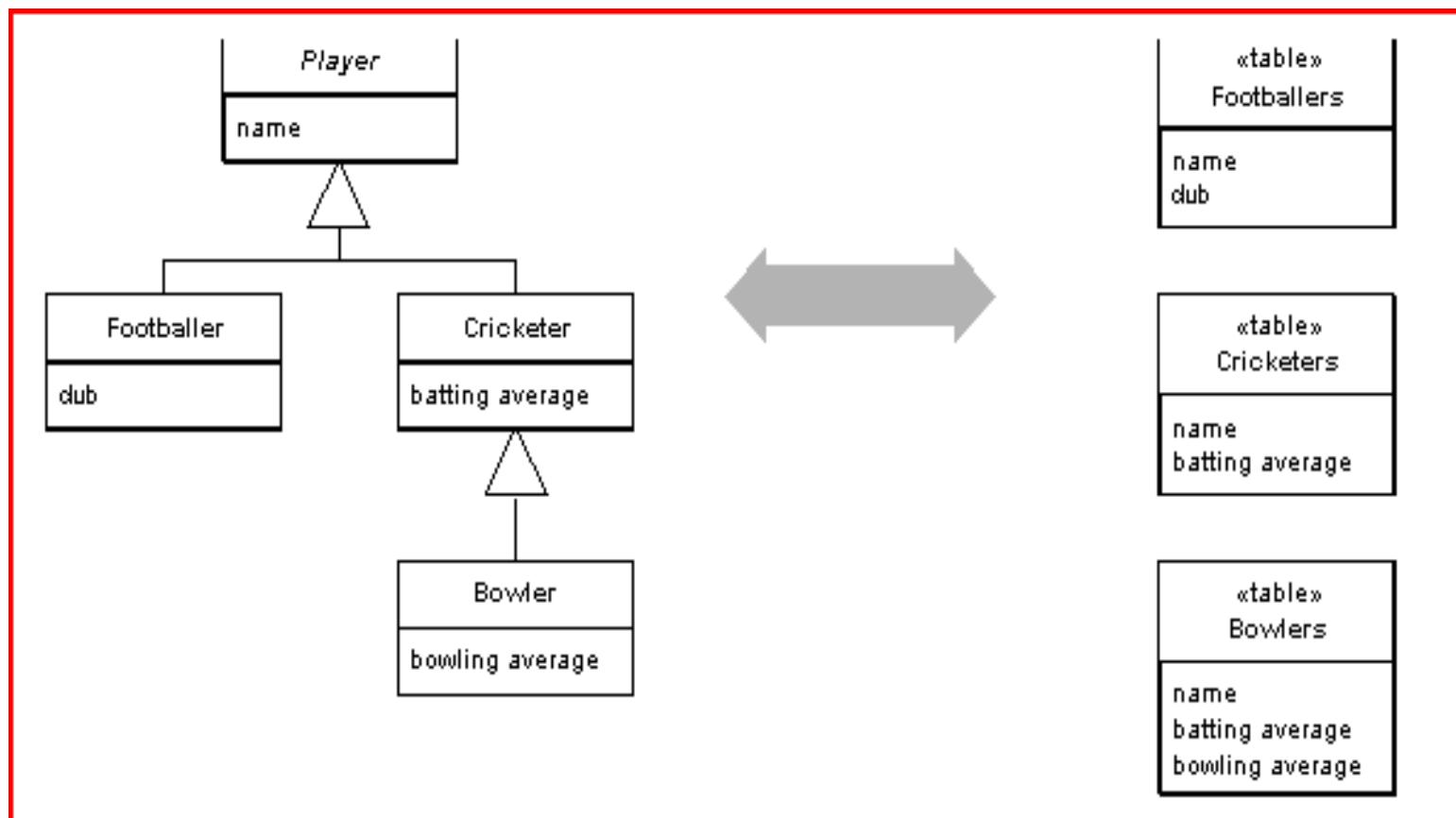
Mapping Inheritance

- Single Table Inheritance



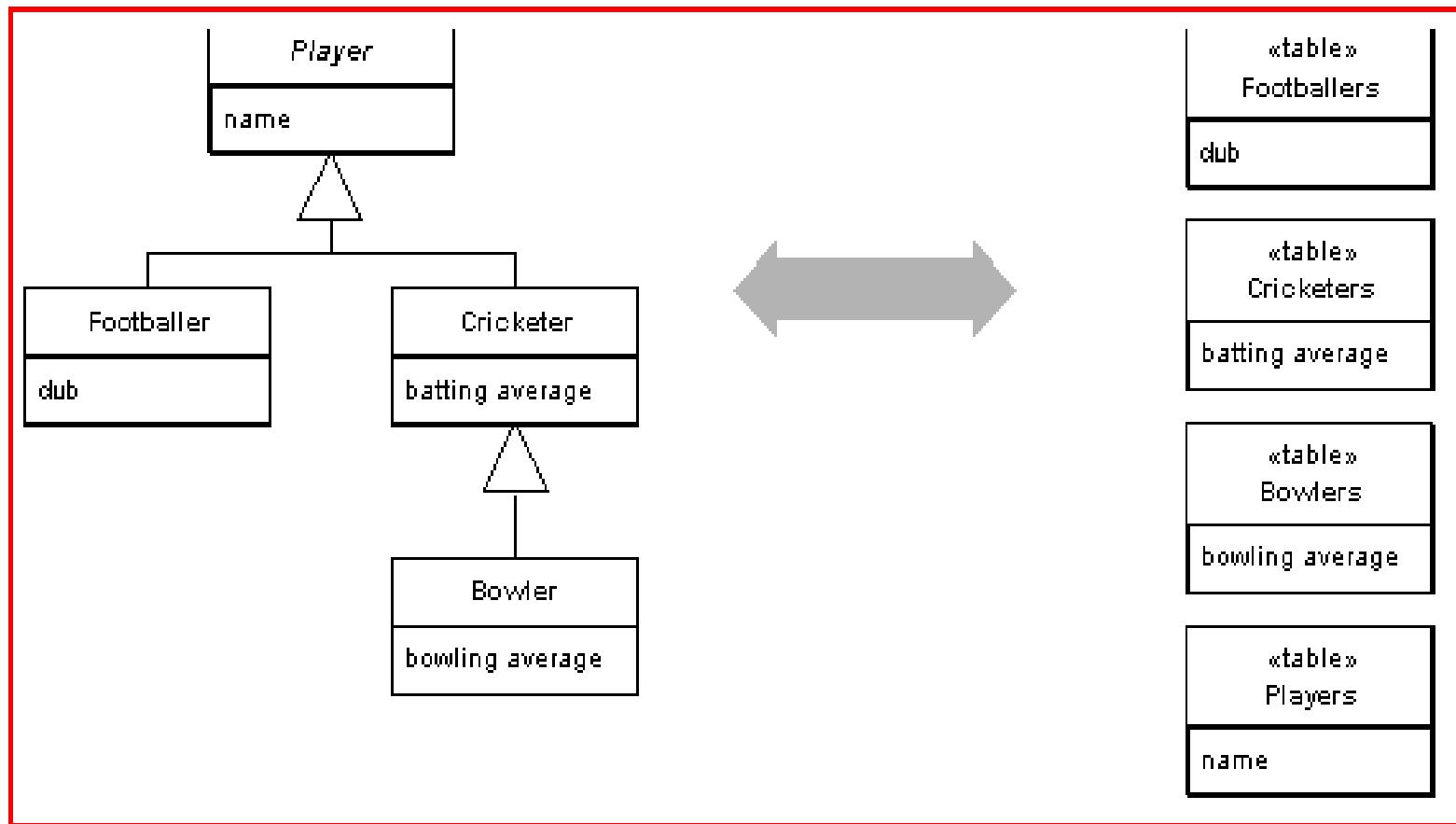
Mapping Inheritance (2)

- Concrete Table Inheritance



Mapping Inheritance (3)

- Class Table Inheritance



Gateways

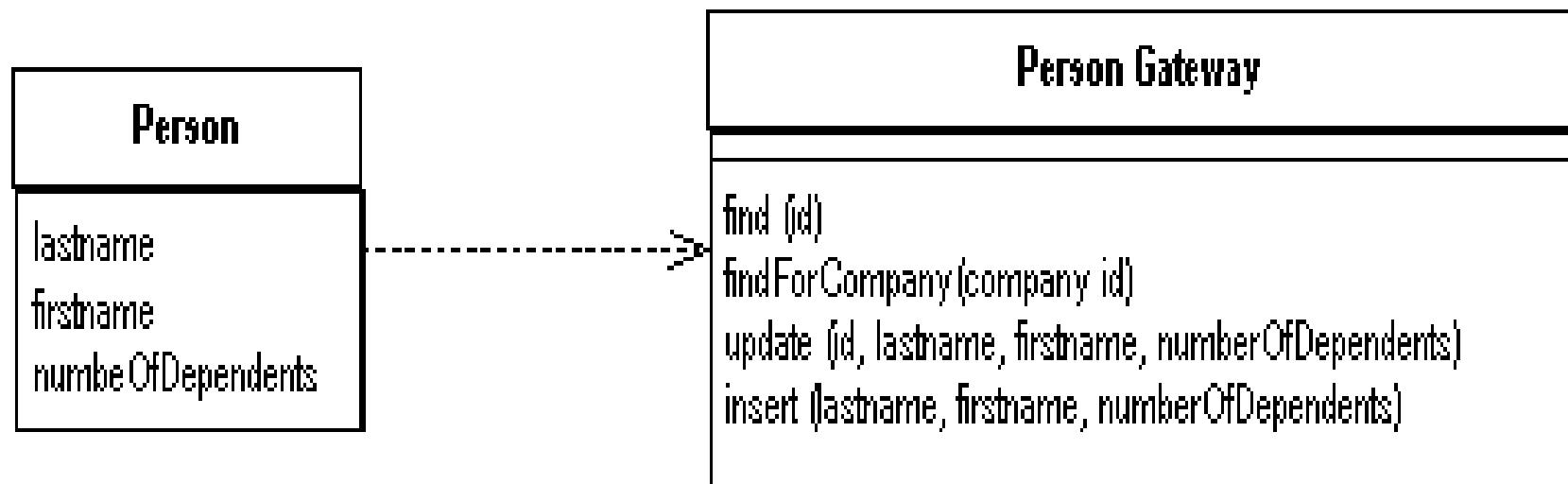
- Definition
 - *An object that encapsulates access to an external system or resource*
- in-memory class which maps exactly to a table in the database => the gateways and the tables are thus *isomorphic*
- contains all the database mapping code for an application, that is all the SQL
- contains no domain logic

Table Data Gateway

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

A TDG hides all the SQL for accessing a single DB table or DB view: selects, updates, deletes.

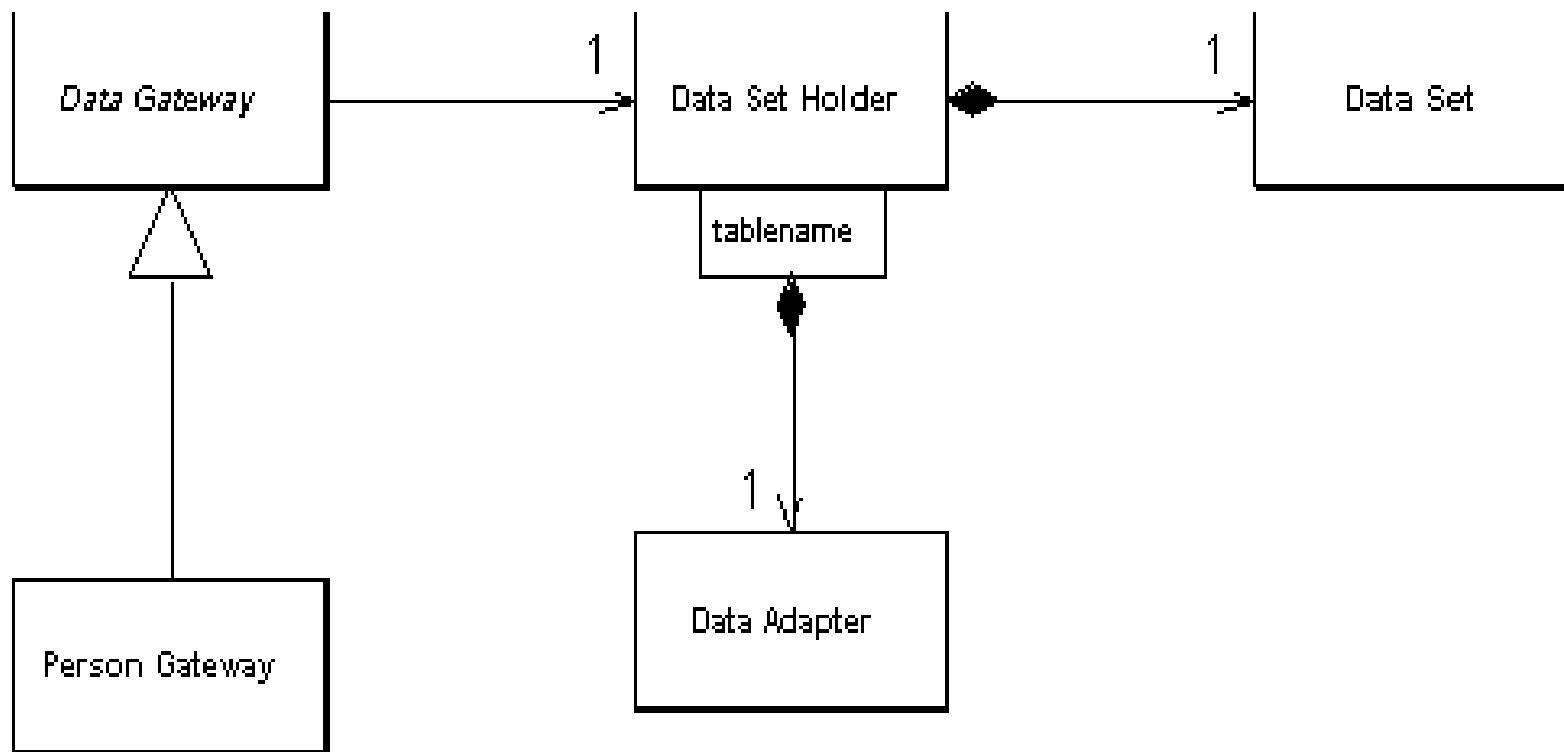
TDG



Features

- No attributes
- CRUD methods
- Challenge: how it returns information from a query ?
 - Data Transfer Object
 - RecordSet
- Goes well with Table Module
- Suitable for Transaction Scripts

Using ADO.NET DataSets



Implementation

```
class DataSetHolder...
    public DataSet Data = new DataSet();
    private Hashtable DataAdapters = new Hashtable();
```

```
class DataGateway...
    public DataSetHolder Holder;
    //public DataSet Data
```

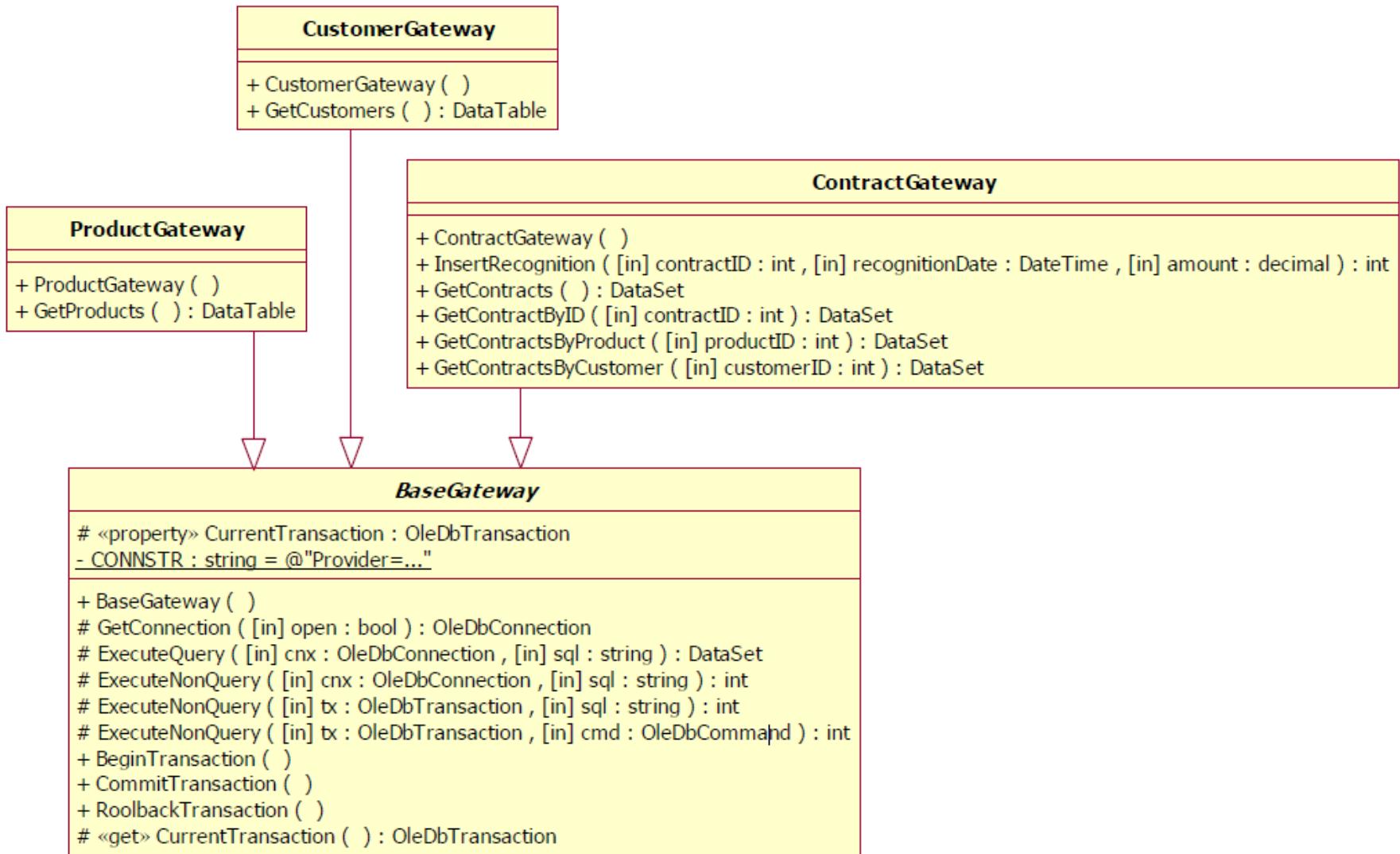
Implementing find behavior

```
class DataGateway...

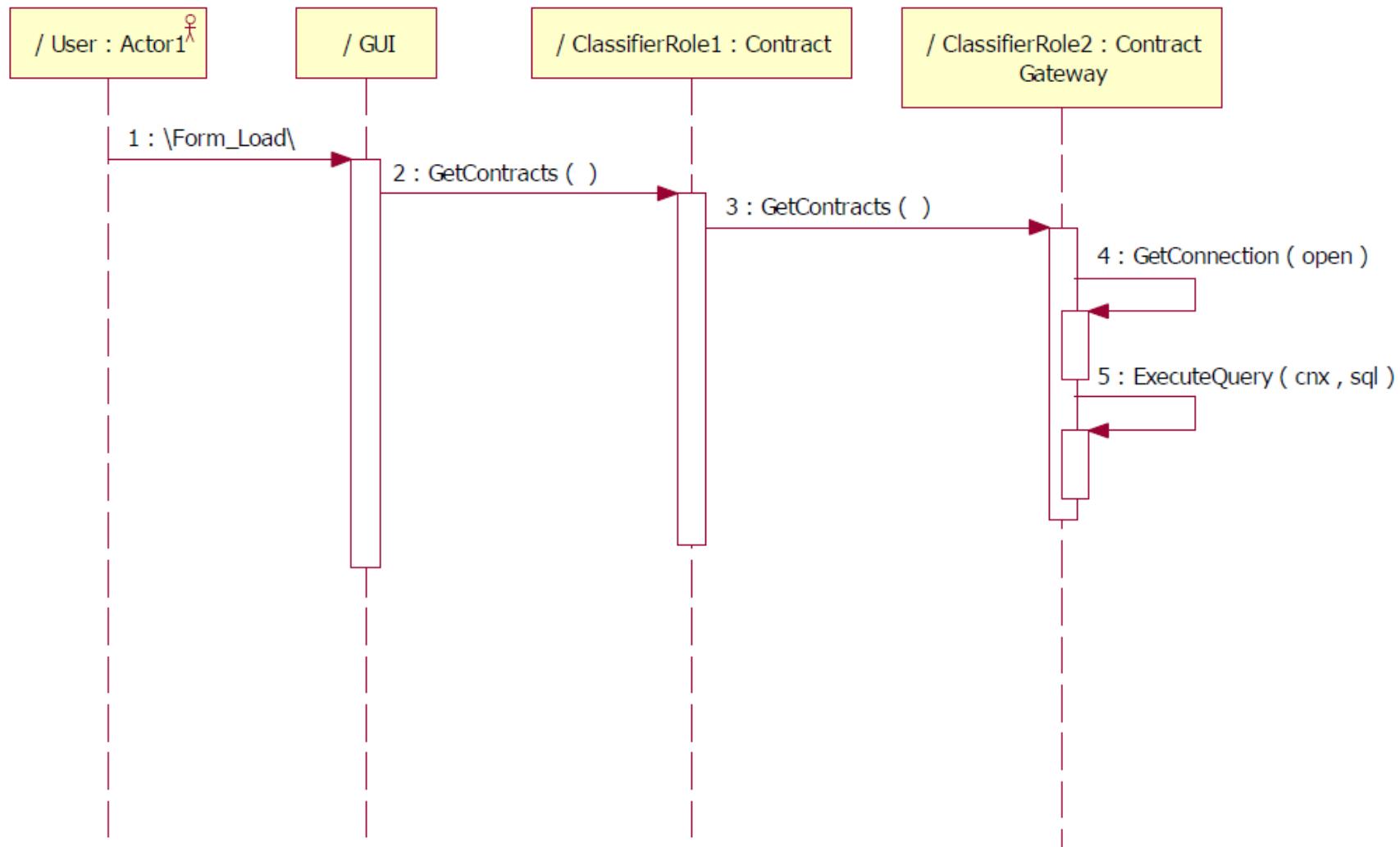
public void LoadAll()
{
    String commandString = String.Format(„Select * from {0}”, TableName);
    Holder.FillData(commandString, TableName);
}

public void LoadWhere(String whereClause)
{
    String commandString =
String.Format(„Select * from {0} where {1}”, TableName, whereClause);
    Holder.FillData(commandString, TableName);
}
```

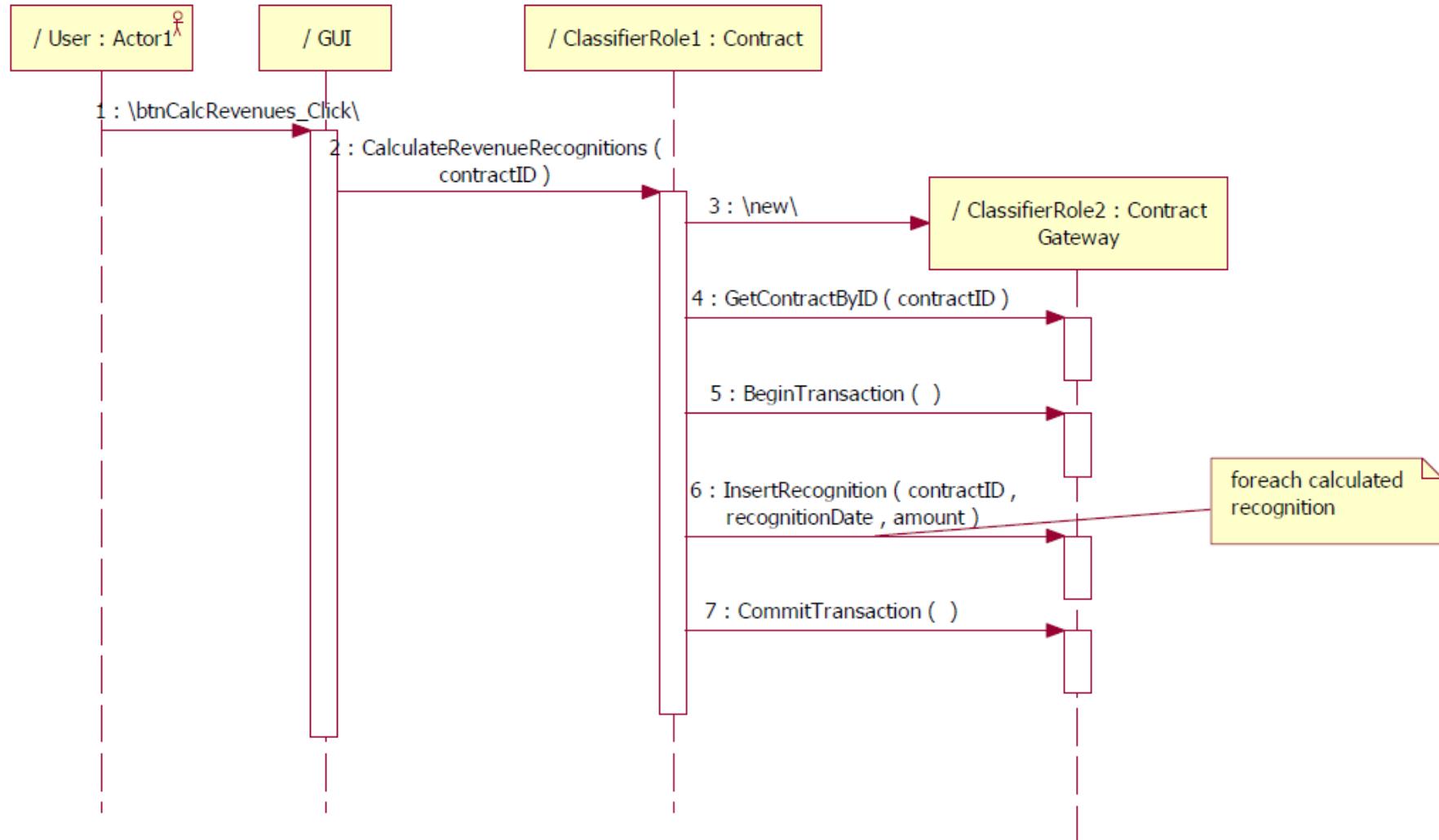
TDG in the Revenue Recognition Pb



TM+TDG Get contracts



TM+TDG Calculate Revenues



Translated in Code

```
public void CalculateRevenueRecognitions(int contractID)
{
    DAL.ContractGateway dal = new DAL.ContractGateway();
    DataSet ds = dal.GetContractByID(contractID);

    string prodType = (string)ds.Tables[0].Rows[0]["type"];
    decimal totalRevenue = (decimal)ds.Tables[0].Rows[0]["revenue"];
    DateTime recDate = (DateTime)ds.Tables[0].Rows[0]["dateSigned"];

    dal.BeginTransaction();
    switch (prodType) {
        case "PT":
            dal.InsertRecognition(contractID, recognitionDate, totalRevenue);
            break;
        case "FC":
            decimal[] allocs = Money.Allocate(totalRevenue, 3);
            dal.InsertRecognition(contractID, recDate, allocs[0]);
            dal.InsertRecognition(contractID, recDate.AddDays(60), allocs[1]);
            dal.InsertRecognition(contractID, recDate.AddDays(90), allocs[2]);
            break;
        case "BD":
            decimal[] allocs = Money.Allocate(totalRevenue, 3);
            dal.InsertRecognition(contractID, recDate, allocs[0]);
            dal.InsertRecognition(contractID, recDate.AddDays(30), allocs[1]);
            dal.InsertRecognition(contractID, recDate.AddDays(60), allocs[2]);
            break;
    }
    dal.CommitTransaction();
}
```

Returns a join of
TContracts and
TProducts

Explicit transaction control
😊/😢

```
public int InsertRecognition(int contractID,
                            DateTime recognitionDate,
                            decimal amount)
{
    OleDbCommand sqlcmd = new OleDbCommand(
        "INSERT INTO TRevenueRecognitions
         (contractID, dateRecognition, amount)
        VALUES (?, ?, ?)",
        CurrentTransation.Connection,
        CurrentTransation
    );

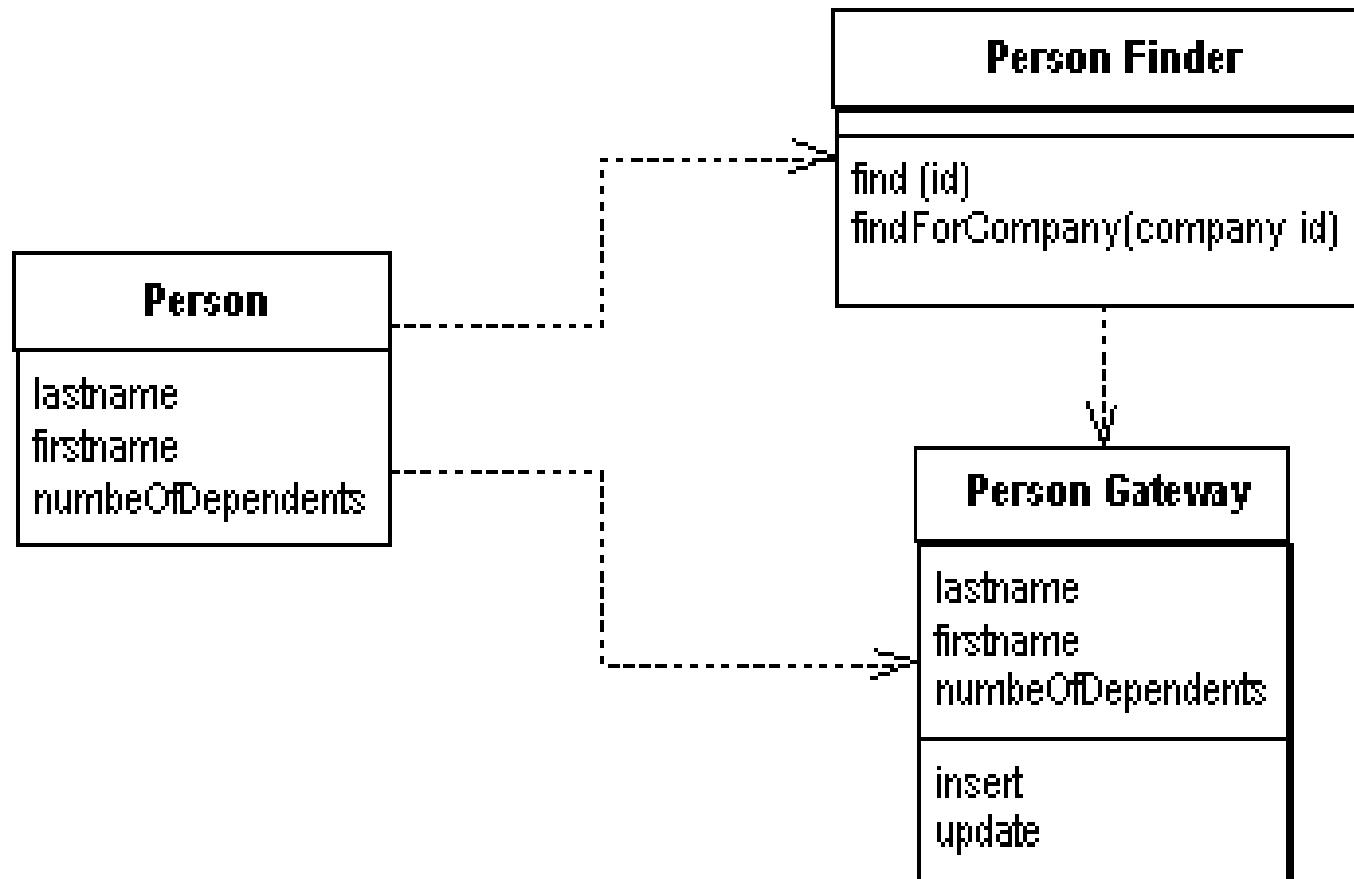
    sqlcmd.Parameters.Add("@cid", contractID);
    sqlcmd.Parameters.Add("@dt", recognitionDate);
    sqlcmd.Parameters.Add("@amt", amount);

    return ExecuteNonQuery(CurrentTransation, sqlcmd);
}
```

Row Data Gateway

- An object that acts as a single record in the data source
 - There is one instance per row
- Fowler RDG combines two roles
 - Class ...Finder with find(id):Gateway method which returns the ‘object’ (i.e. SELECT statements)
 - Class ...Gateway which is the ‘object’

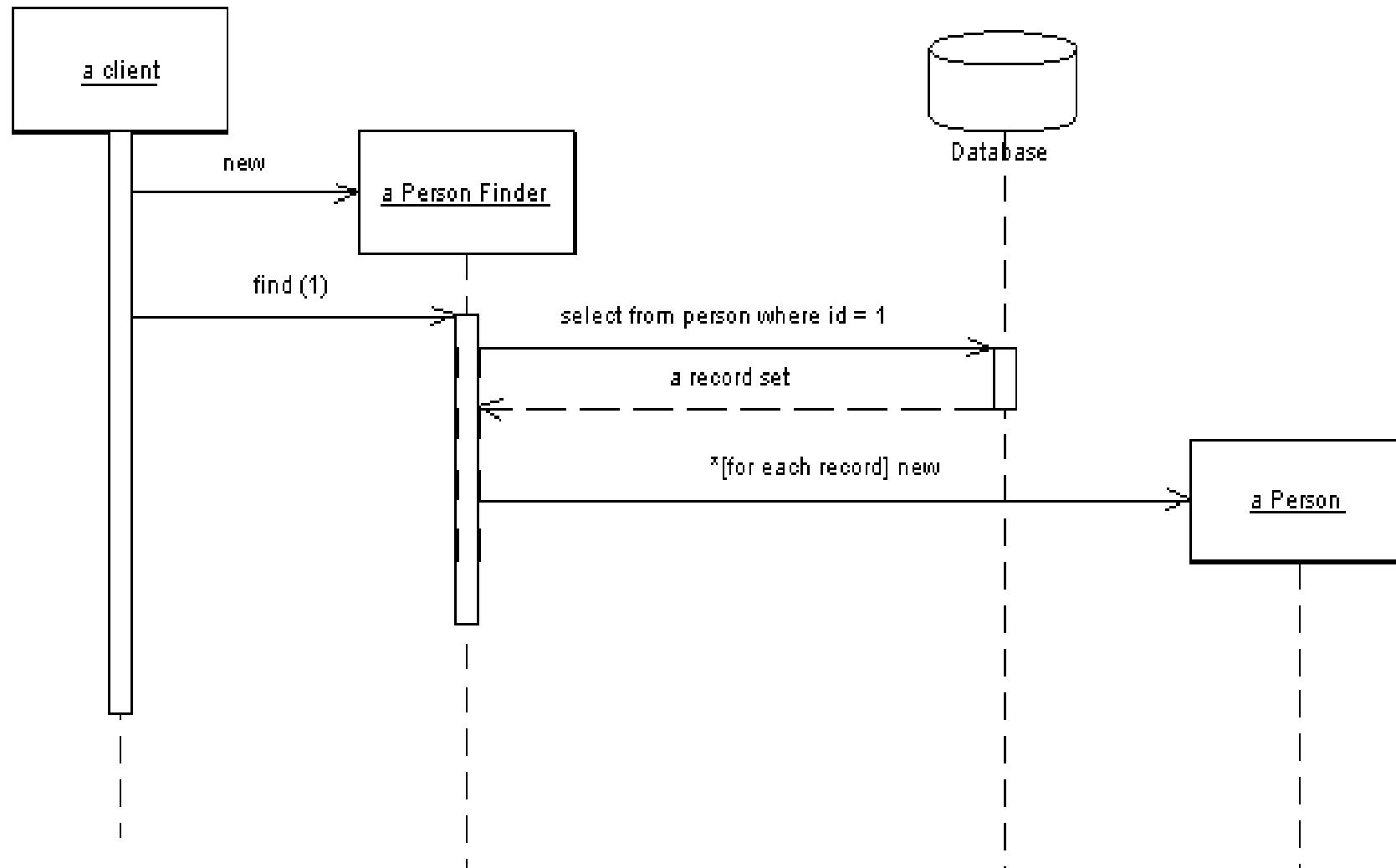
Row Data Gateway



How it works?

- Separate data access code from business logic
- type conversion from the data source types to the in-memory types
- works particularly well for Transaction Scripts
- where to put the find operations that generate the *Row Data* ?
 - separate finder objects
 - each table in a relational database will have:
 - one finder class
 - one gateway class for the results

RDG behavior



Implementation

```
class PersonGateway...
{
    private String lastName;
    private String firstName;
    private int numberDependents;

    public String getLastName()
    { return lastName; }

    public void setLastName(String lastName)
    { this.lastName = lastName; }

    ...

    public void insert() {...}
    public void update() {...}
    public static PersonGateway load(ResultSet rs) {...}

    ...
}
```

Class PersonGateway

```
private static final String updateStatementString = "UPDATE people "
+ " set lastname = ?, firstname = ?, number_of_dependents = ? "
+ " where id = ?";
```

```
public void update() {
    PreparedStatement updateStatement = null;
    try { updateStatement = DB.prepare(updateStatementString);
        updateStatement.setString(1, lastName);
        updateStatement.setString(2, firstName);
        updateStatement.setInt(3, numberOfDependents);
        updateStatement.setInt(4, getID().intValue());
        updateStatement.execute(); }
    catch (Exception e)
        { throw new ApplicationException(e); }
    finally
        {DB.cleanUp(updateStatement); } }
```

...

PersonFinder

```
class PersonFinder...
private final static String findStatementString = "SELECT id, lastname, firstname,
number_of_dependents " + " from people " + " WHERE id = ?";
public PersonGateway find(Long id)
{
    PersonGateway result = (PersonGateway)Registry.getPerson(id);
    if (result != null) return result;
    try
    {
        PreparedStatement findStatement = DB.prepare(findStatementString);
        findStatement.setLong(1, id.longValue());
        ResultSet rs = findStatement.executeQuery();
        rs.next();
        result = PersonGateway.load(rs);
        return result;
    } catch (SQLException e) { throw new ApplicationException(e);
}
```

Using RDG as a holder for the domain object

```
class Person...
    private PersonGateway data;
    public Person(PersonGateway data)
    { this.data = data; }
    public int getNumberOfDependents()
    {
        return data.getNumberOfDependents();
    }
```

Data Mappers

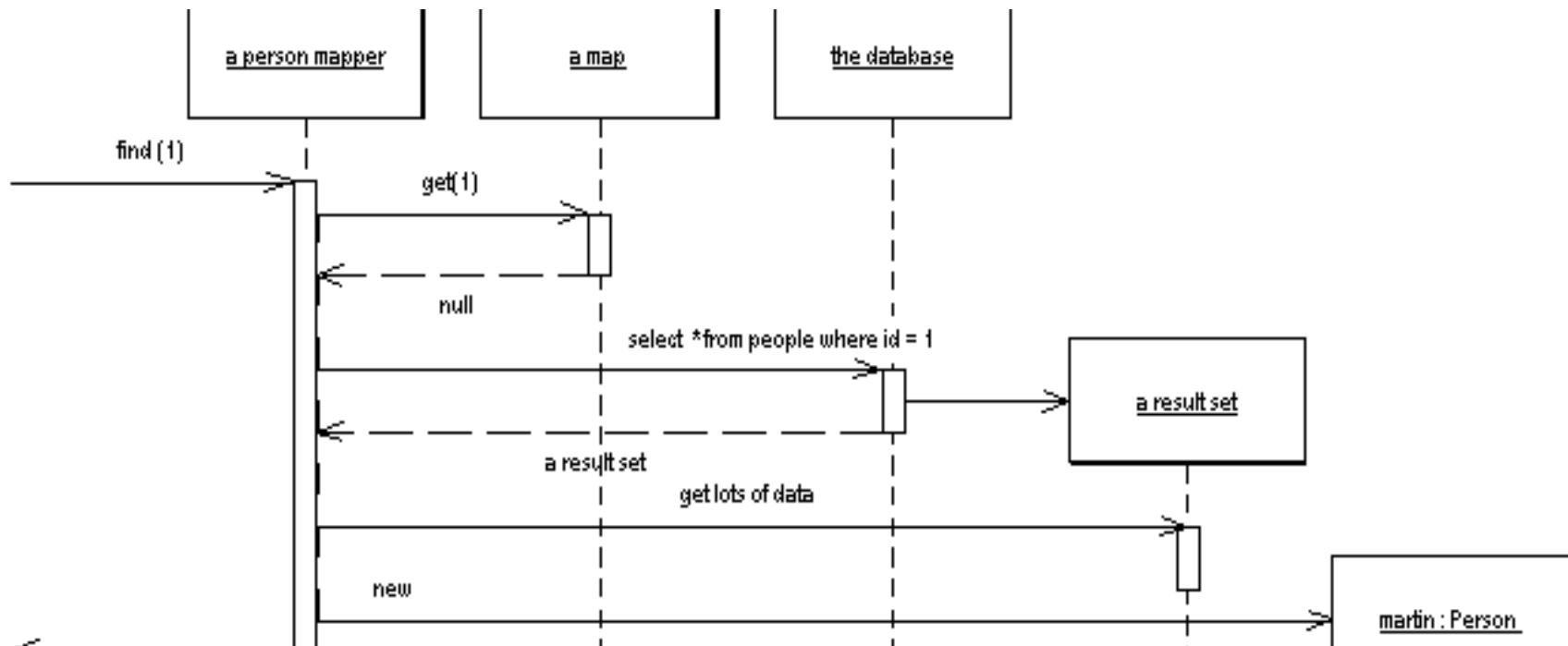
- Acts as an intermediary between Domain Models and the database.
- Allows Domain Models and Data Source classes to be independent of each other



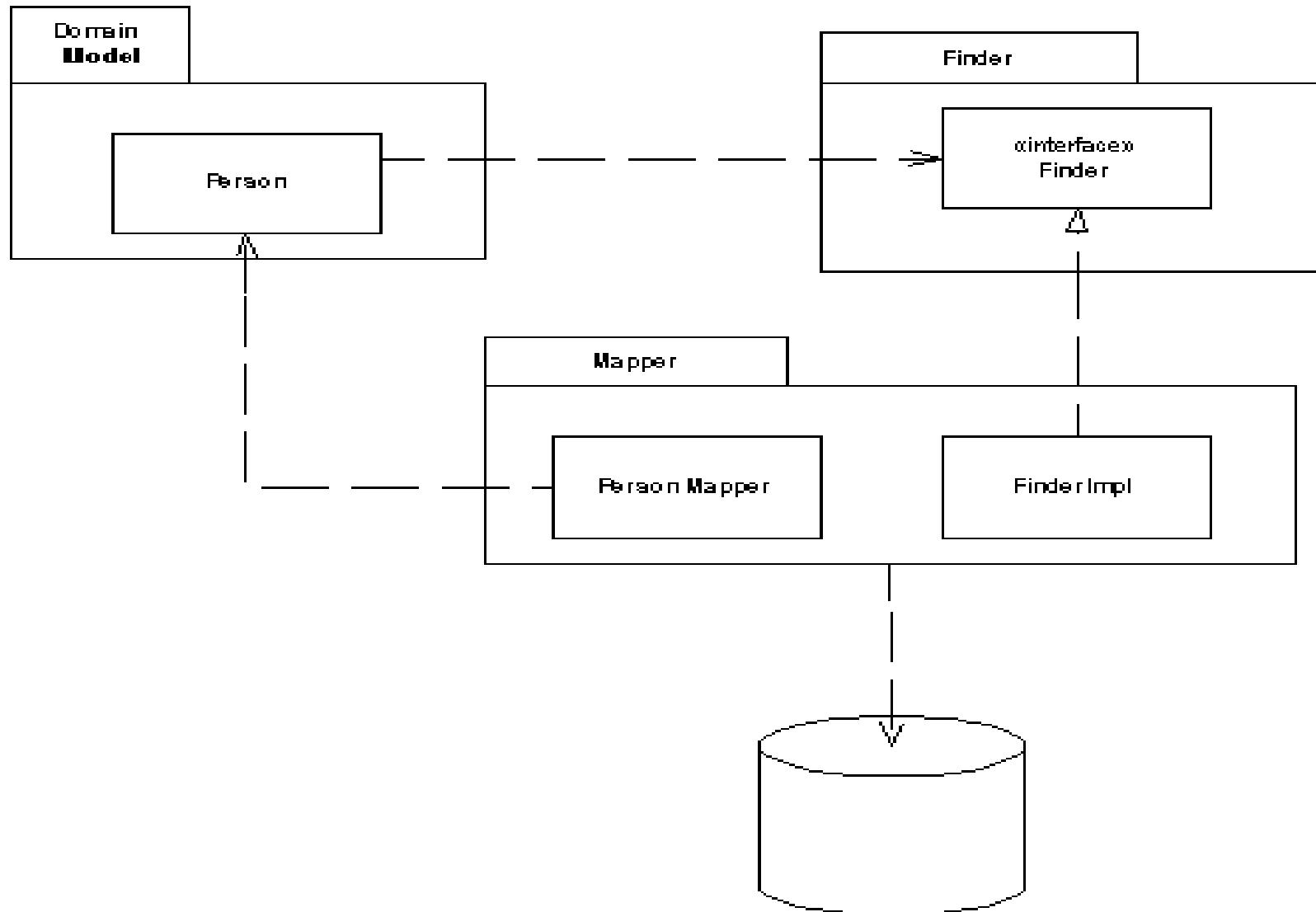
Data Mapper Layer ...

- Can either
 - Access the database itself, or
 - Make use of a Table Data Gateway.
- Does not contain Domain Logic.
- When it uses a TDG, the Data Mapper can be placed in the (lower) Domain layer.

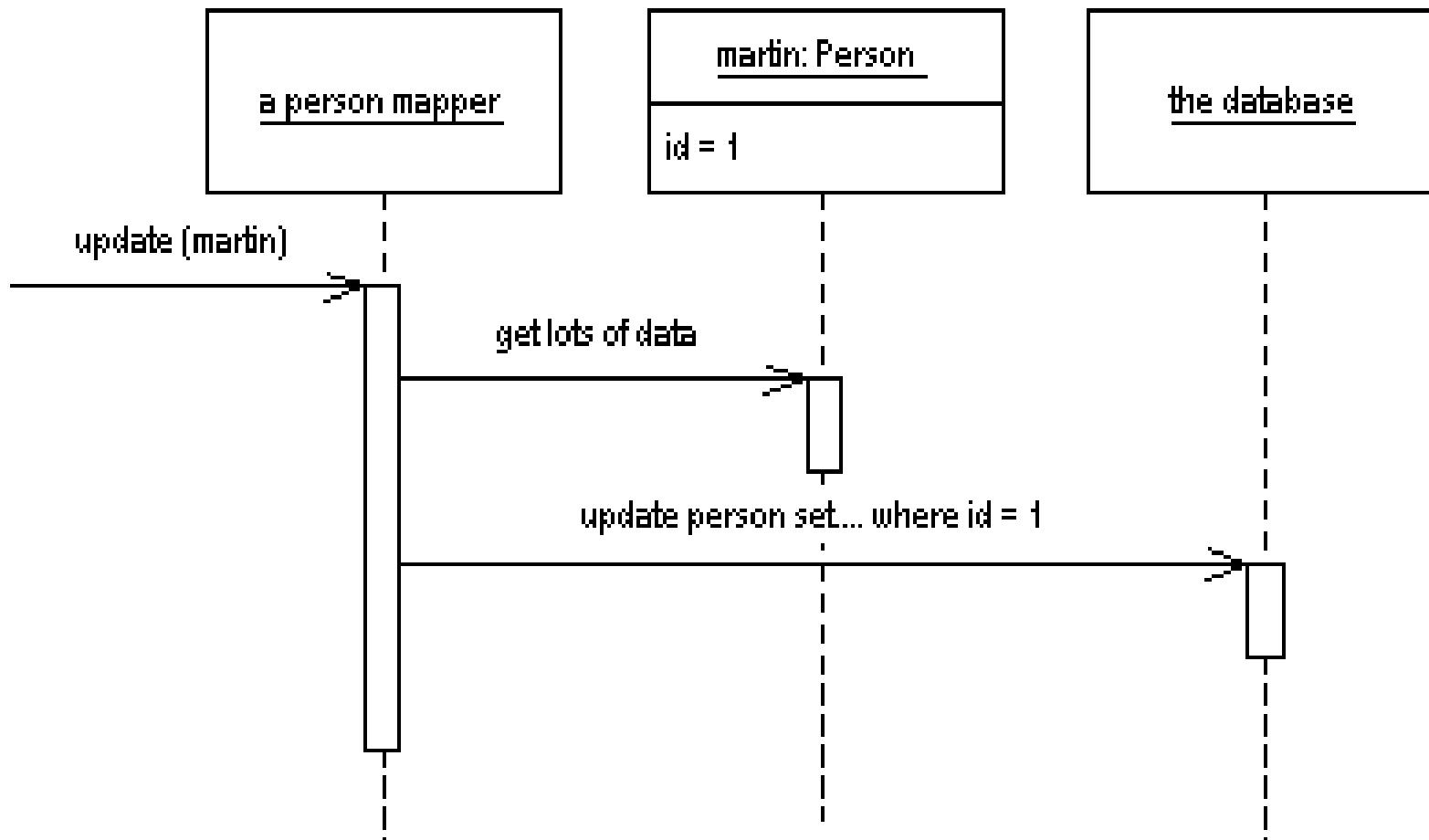
Retrieving data



Finding objects



Updating data



Features

- Independent database schema and object model
- Extra layer
- Makes sense with Domain Model

Implementation

```
class Person...
```

```
    private String name;
```

```
    private int numberOfDependents;
```

```
...
```

```
create table people (ID int primary key, lastname  
        varchar, firstname varchar, number_of_dependents int)
```

AbstractMapper

```
class AbstractMapper...
    protected Map loadedMap = new HashMap();
    abstract protected String findStatement();
    protected DomainObject abstractFind(Long id)
    {
        DomainObject result = (DomainObject) loadedMap.get(id);
        if (result != null) return result;
        PreparedStatement findStatement = null;
        try
        {
            findStatement = DB.prepare(findStatement());
            findStatement.setLong(1, id.longValue());
            ResultSet rs = findStatement.executeQuery();
            rs.next();
            result = load(rs);
            return result;
        } catch (SQLException e)
        {
            throw new ApplicationException(e);
        }
        finally { DB.cleanUp(findStatement); }
    }
```

Load method in AbstractMapper

```
class AbstractMapper...
    protected DomainObject load(ResultSet rs) throws SQLException
    {
        Long id = new Long(rs.getLong(1));
        if (loadedMap.containsKey(id))
            return (DomainObject) loadedMap.get(id);
        DomainObject result = doLoad(id, rs);
        loadedMap.put(id, result);
        return result;
    }
    abstract protected DomainObject doLoad(Long id, ResultSet rs)
    throws SQLException;
```

Mapper class implements finder

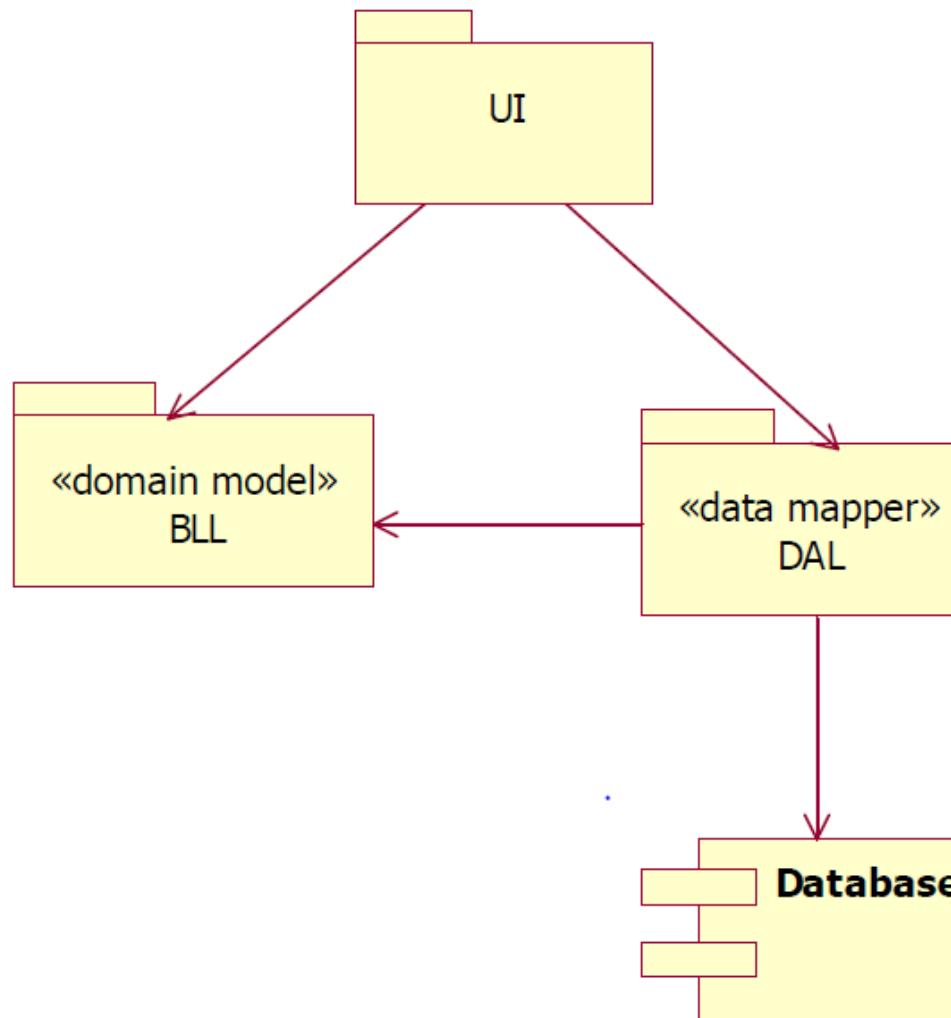
```
class PersonMapper...
    protected String findStatement()
{
    return "SELECT " + COLUMNS + " FROM people" +
" WHERE id = ?";
}
public static final String COLUMNS = " id, lastname,
firstname, number_of_dependents ";
public Person find(Long id)
{
    return (Person) abstractFind(id);
}
```

doLoad in PersonMapper

```
class PersonMapper...
protected DomainObject doLoad(Long id, ResultSet rs)
throws SQLException
{
    String name = rs.getString(2)+” “ +
        rs.getString(3);
    int numDependentsArg = rs.getInt(4);

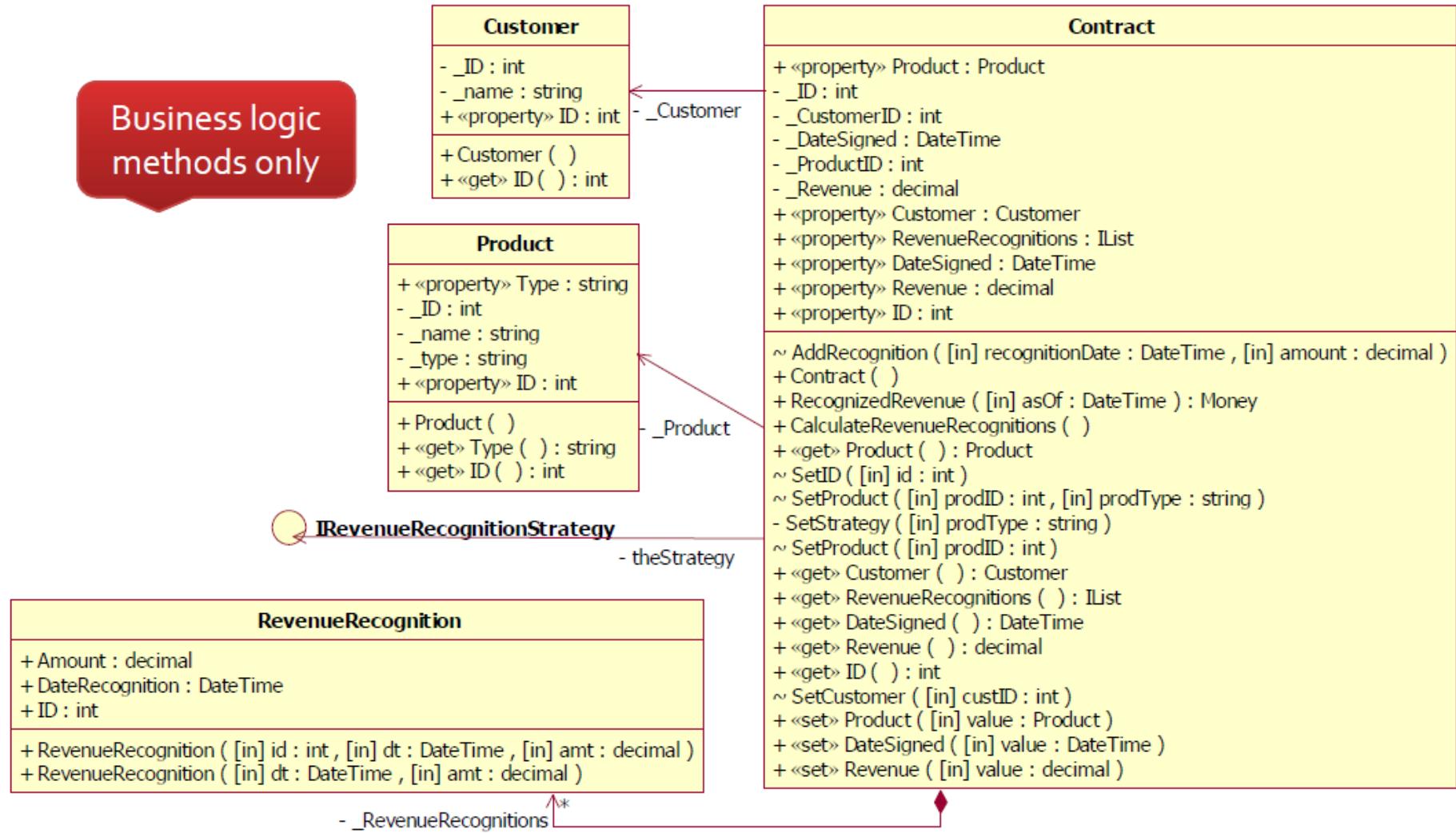
    return new Person(id, name, numDependentsArg);
}
```

Domain Model + Data Mapper

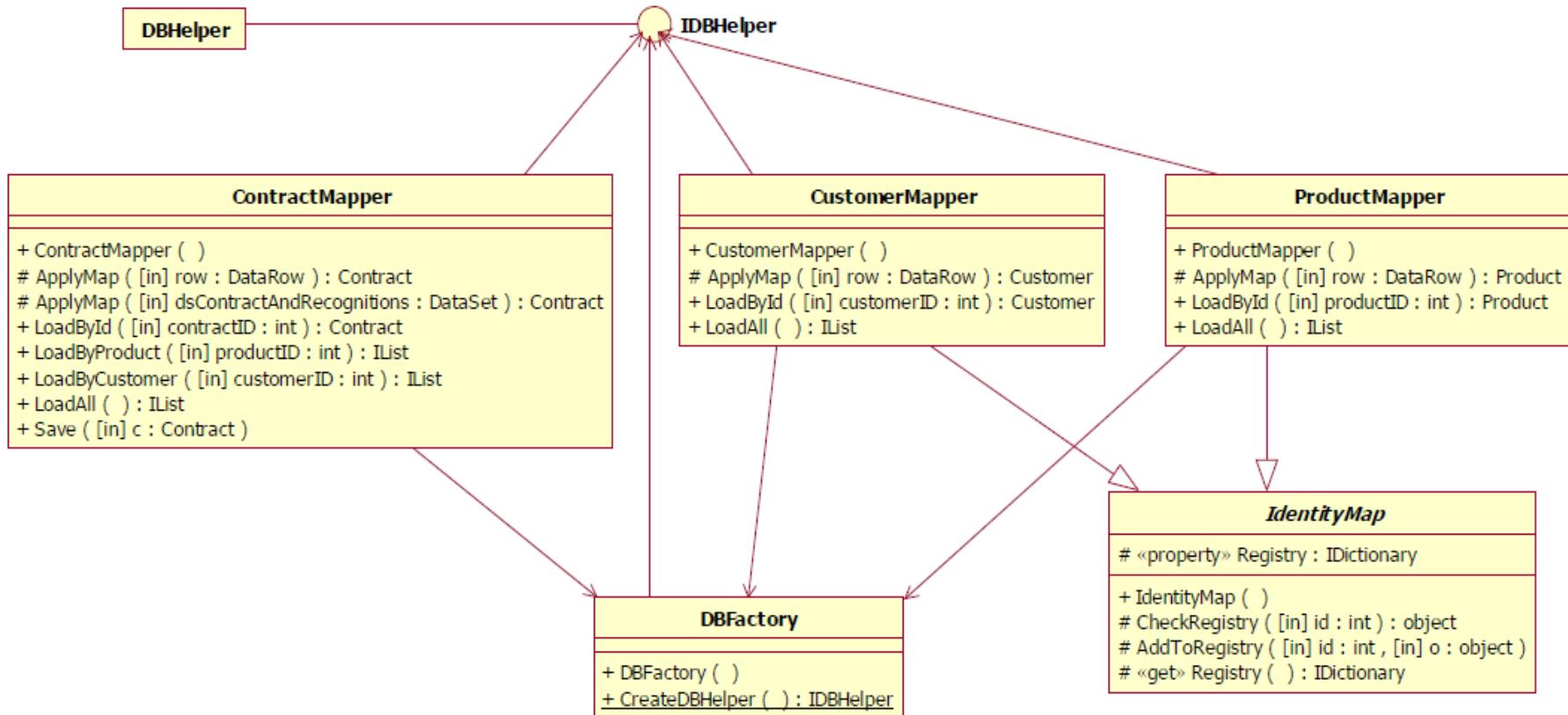


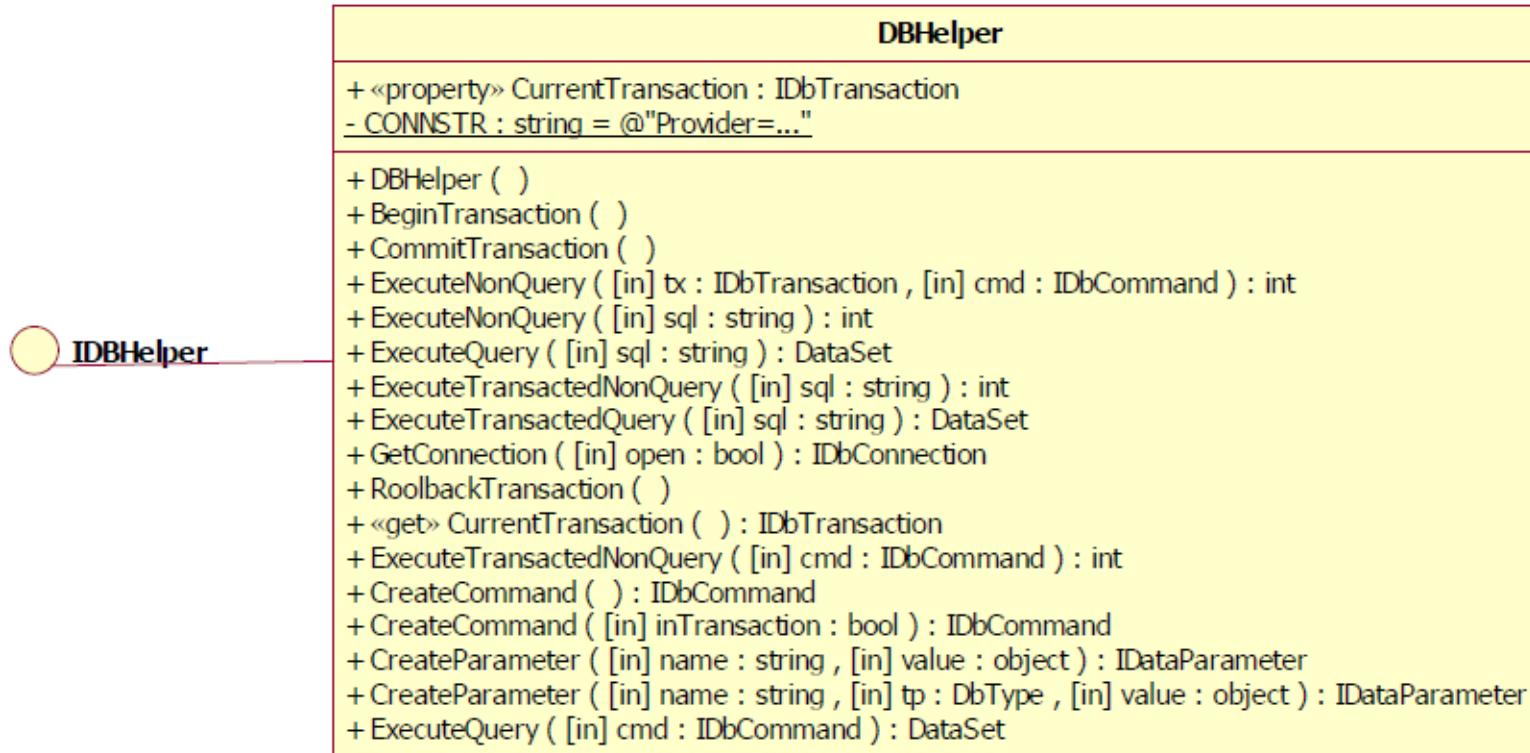
Domain Model

Business logic
methods only

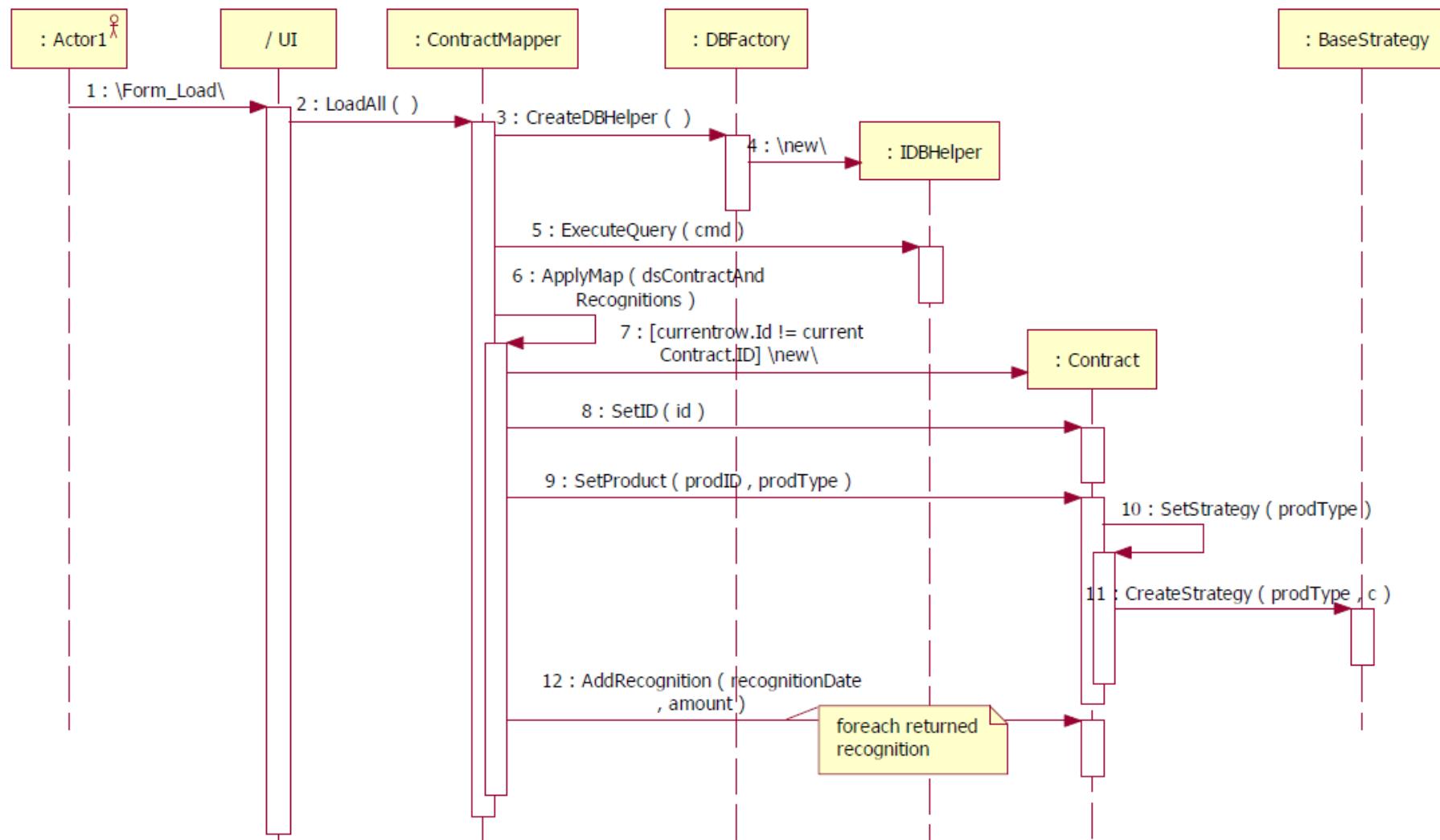


Data Mapper





GetContracts



ContractMapper.applyMap()

```
protected Contract ApplyMap(DataSet dsCNR)
{
    if (dsCNR.Rows.Count < 1)
        return null;

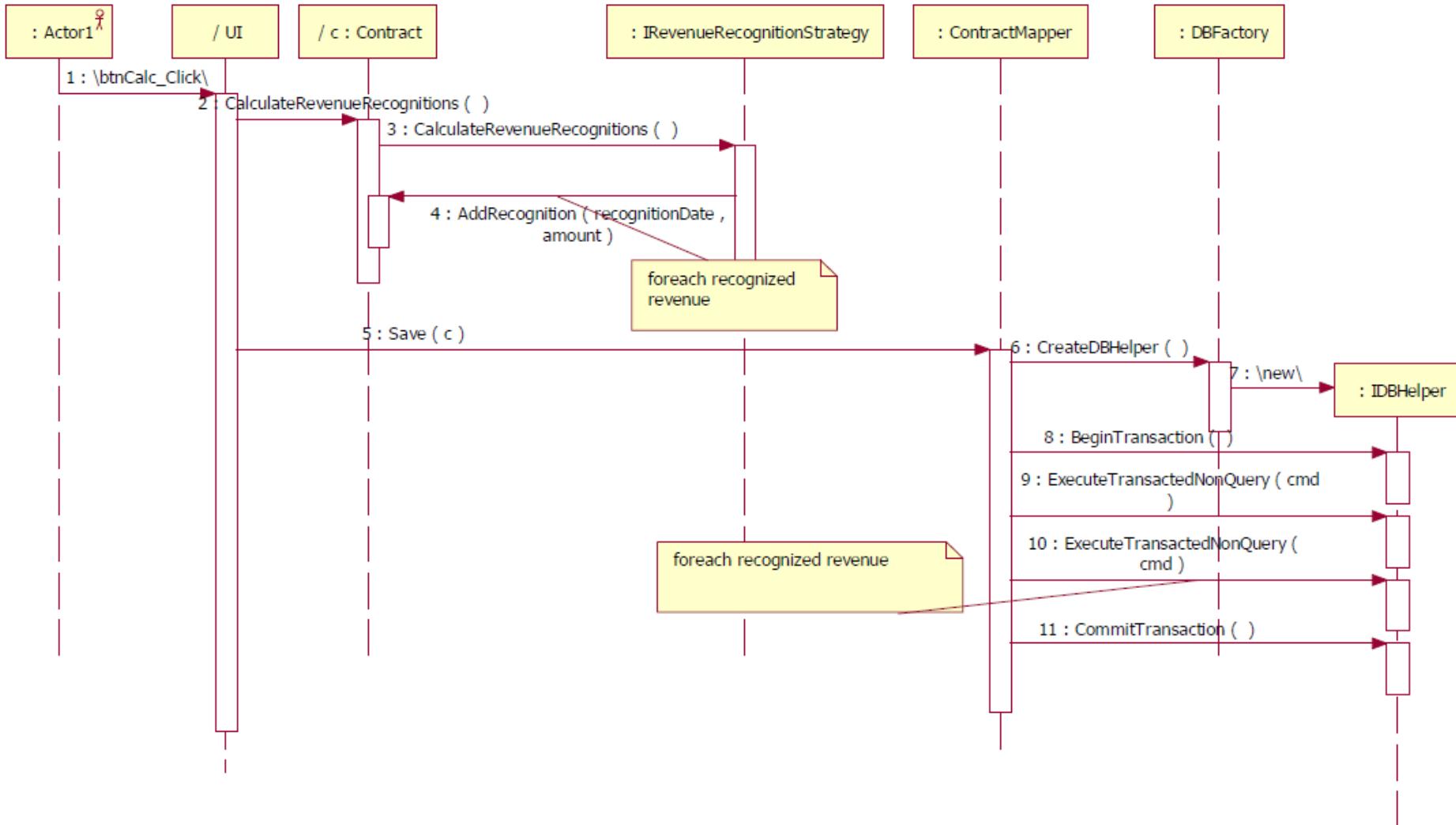
    Contract c = new Contract();

    c.DateSigned = (DateTime)dsCNR.Tables[0].Rows[0]["Datesigned"];
    c.Revenue = (decimal)dsCNR.Tables[0].Rows[0]["Revenue"];
    c.SetID( (int)dsCNR.Tables[0].Rows[0]["ID"] );
    c.SetProduct( (int) dsCNR.Tables[0].Rows[0]["ProductID"],
                  (string) dsCNR.Tables[0].Rows[0]["ProdType"] );
    c.SetCustomer( (int)dsCNR.Tables[0].Rows[0]["CustomerID"] );

    foreach (DataRow r in dsCNR.Tables[0].Rows)
    {
        c.AddRecognition( (DateTime)r["dateRecognition"],
                          (decimal)r["amount"]);
    }

    return c;
}
```

Calculate Revenues



Calculate Revenues

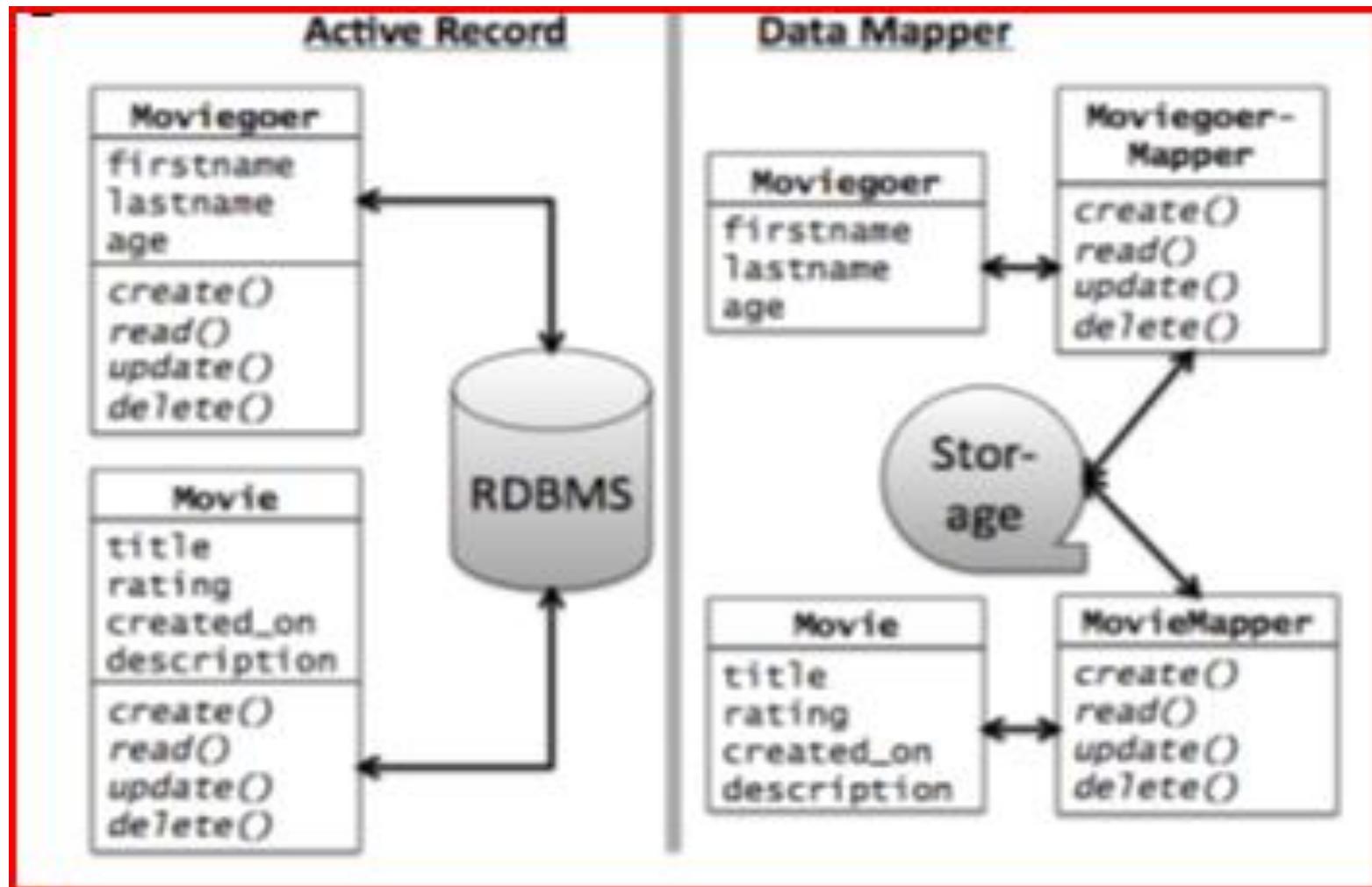
```
public void CalculateRevenueRecognitions()
{
    _RevenueRecognitions.Clear();
    theStrategy.CalculateRevenueRecognitions();
}

// !!! to be used by strategies
internal void AddRecognition(DateTime recognitionDate,
                               decimal amount)
{
    _RevenueRecognitions.Add(
        new RevenueRecognition(recognitionDate, amount)
    );
}
```

Hybrid Data Source Patterns

- Active Record = RDG + Domain Logic.
- Table Module \approx TDG + Domain Logic.
 - TDG like module that processes ResultSets.

Active Record vs. Data Mapper

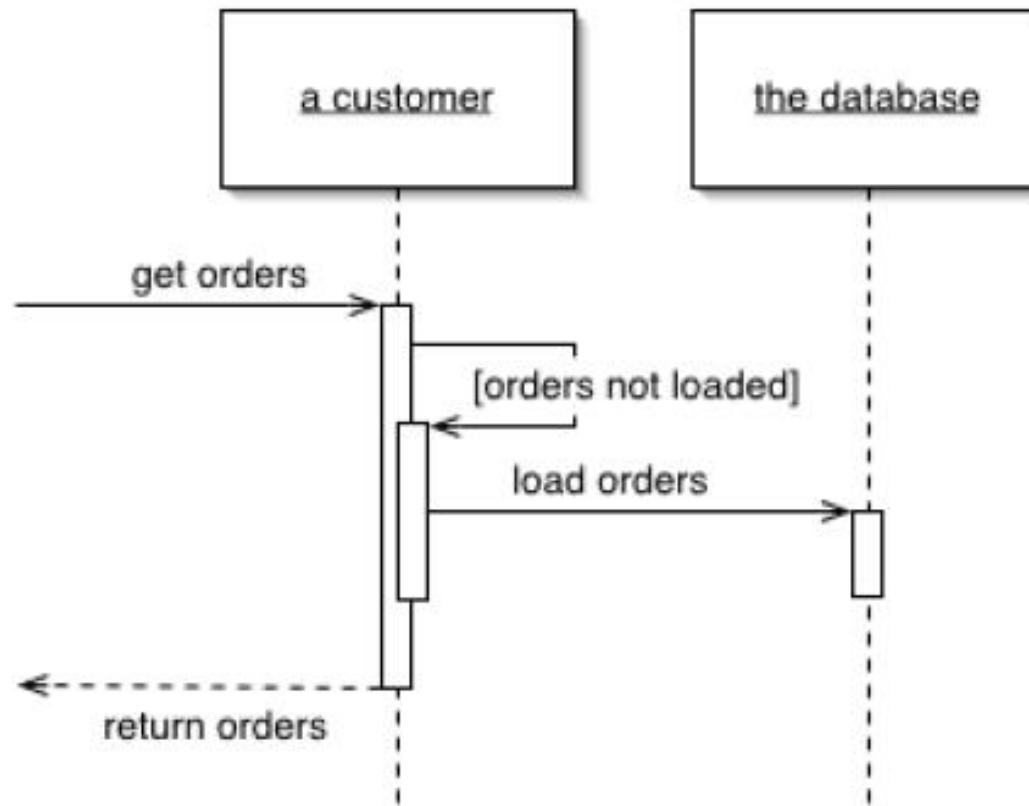


Problems with Domain Model

- Networks of objects
 - E.g. Invoice heading relates to invoice details
 - Invoice details refers to Products
 - Products refers to Suppliers
 - ...
- What to do?
 - Load them all into memory?
 - How to disallow multiple in-memory copies

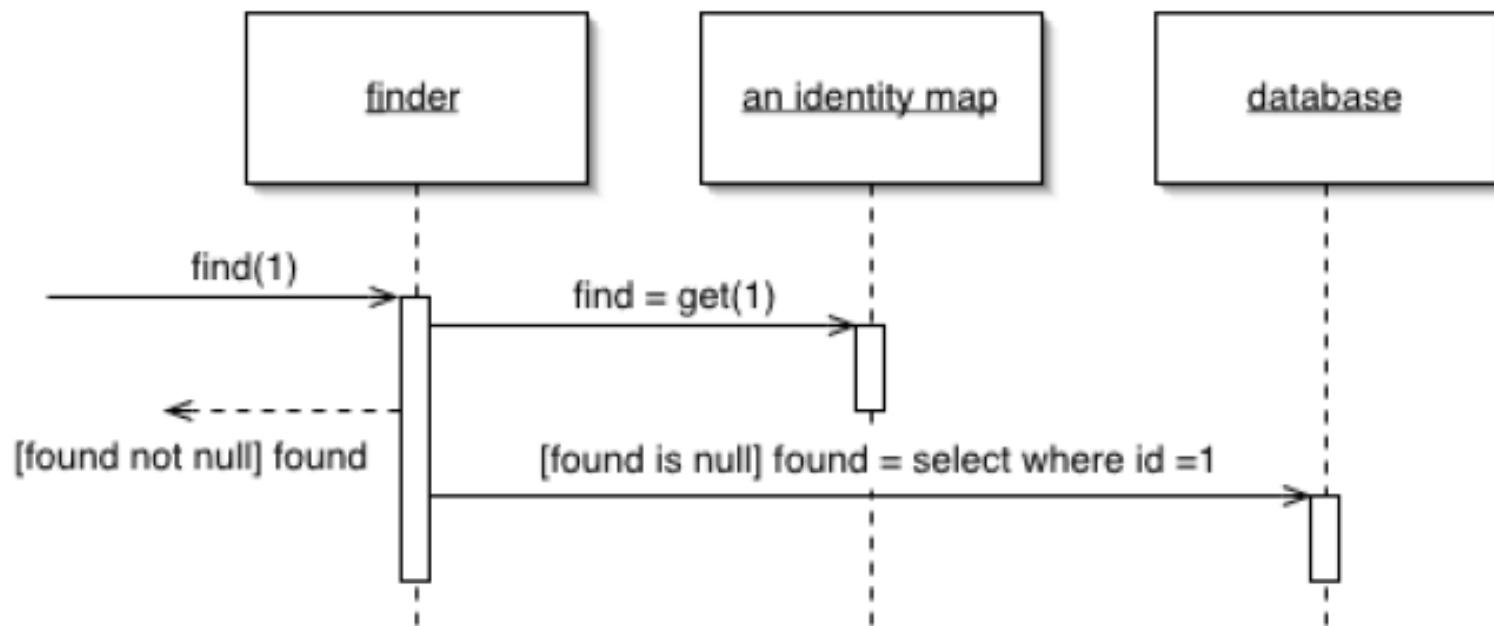
Lazy load

- An object that doesn't contain all of the data you need but knows how to get it.



Identity map

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



How it works

- How many?
 - One map/session
 - One map/table
 - One map/class
 - One map/inheritance tree
- Map key?
 - Primary key in the data base
- Explicit vs. generic
 - `findPerson(1)`
 - `find ("Person", 1)`

Concurrency Patterns

- Multiple processes/threads that manipulate the same data
- A solution -> Transaction managers....
as long as all data manipulation is within a transaction.
- What if data manipulation spans transactions?

Concurrency problems

- lost updates
- inconsistent read
⇒ Correctness failure
- liveness – how much concurrency can the system handle?

Execution contexts

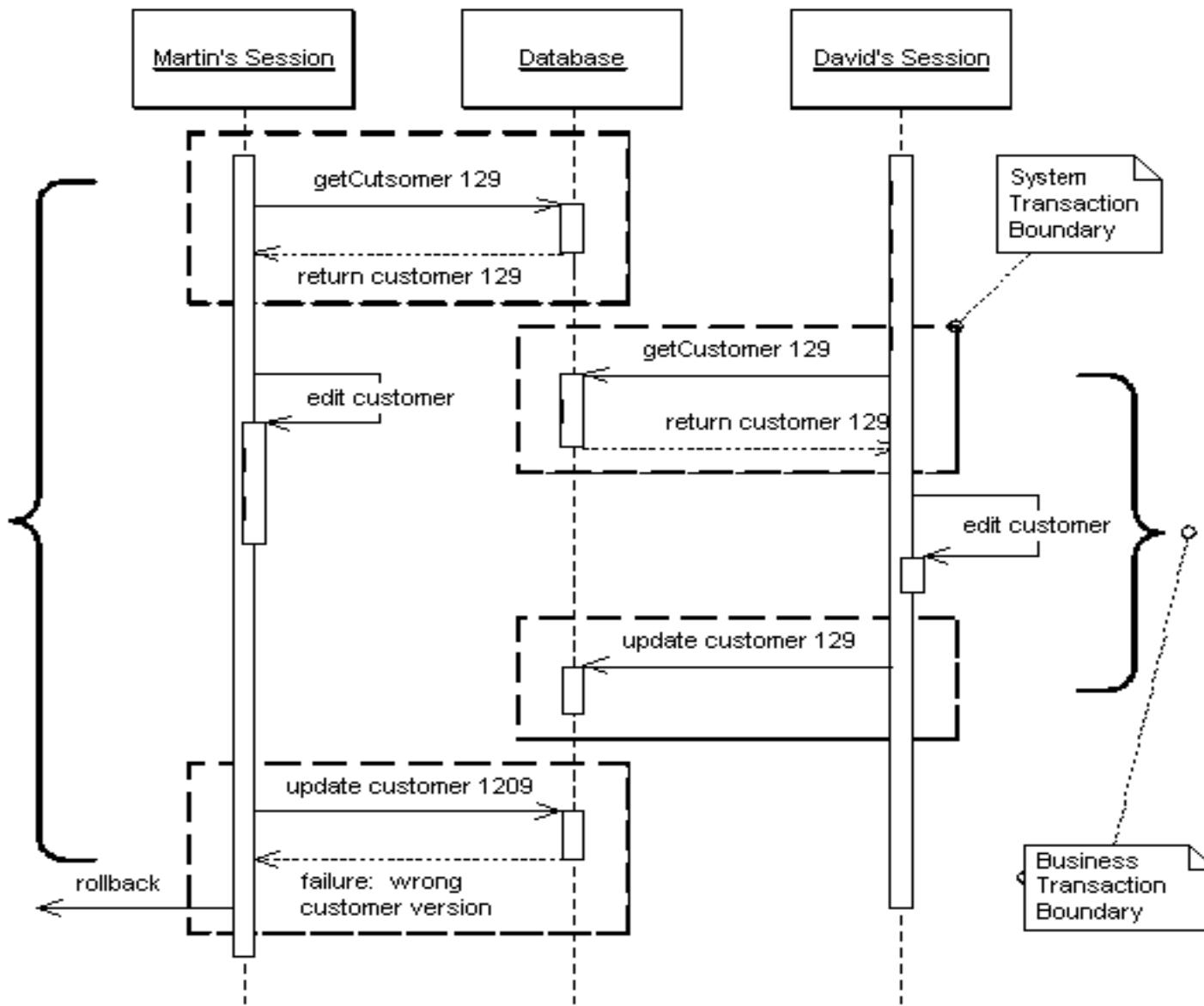
- “A **request** corresponds to a single call from the outside world which the software works on and optionally sends back a response”
- “A **session** is a long running interaction between a client and server.”
- “A **process** is a, usually heavyweight, execution context that provides a lot of isolation for the internal data it works on.”
- “A **thread** is a lighter-weight active agent that's set up so that multiple threads can operate in a single process.”

Solutions

- **isolation:** partition the data so that any piece of data can only be accessed by one active agent.
- **immutable data:** separate the data that cannot be modified.
- **mutable data than cannot be isolated:**
 - Optimistic Concurrency Control
 - Pessimistic Concurrency Control

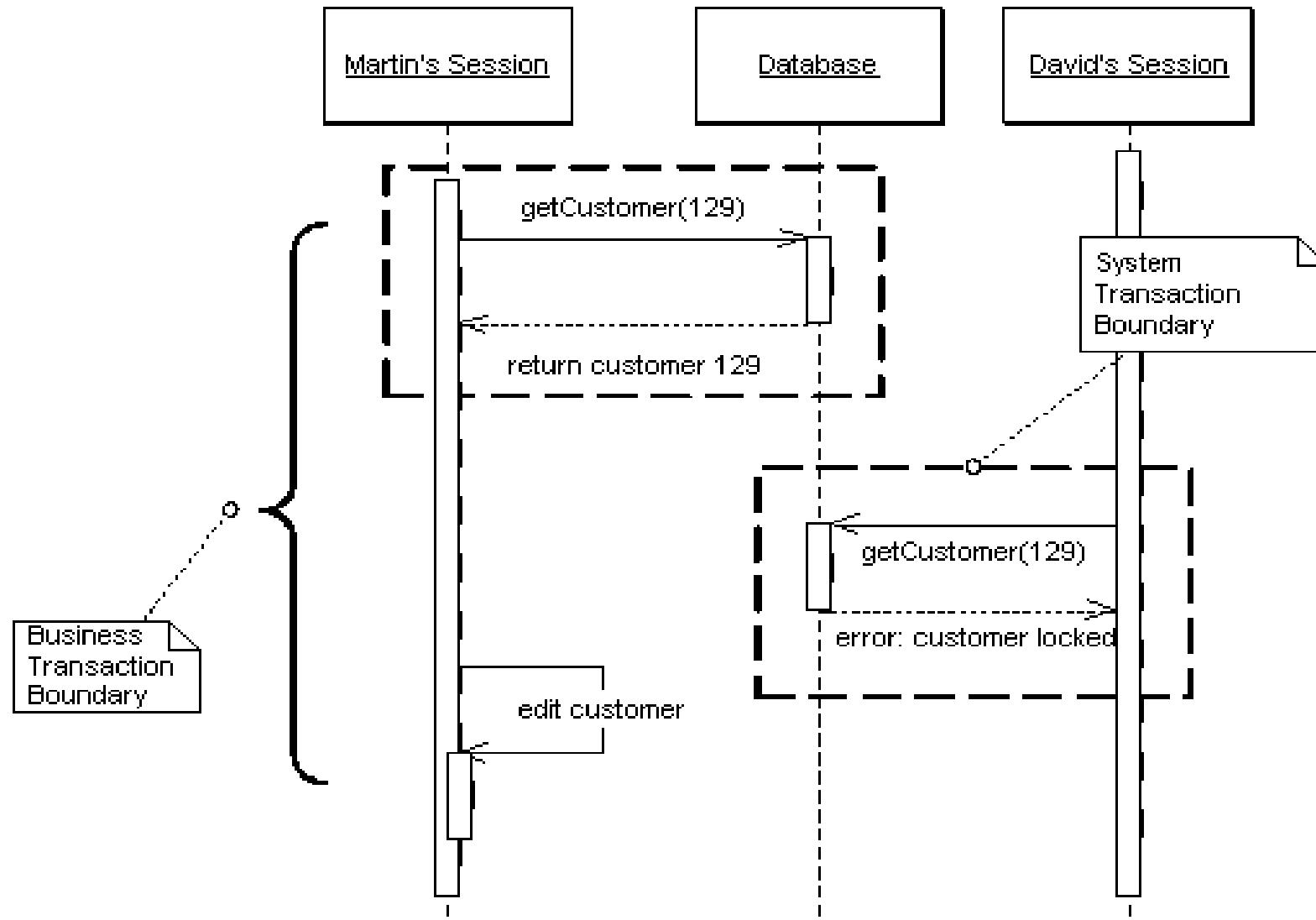
Optimistic Concurrency Control

- Prevent conflicts between concurrent business transactions, by detecting a conflict and rolling back the transaction.
- Conflict detection
- Lock hold during commit
- Supports concurrency
- Low frequency of conflicts
- Used for not critical consequences



Pessimistic Concurrency Control

- Prevent conflicts between concurrent business transactions by allowing only one business transaction to access data at once.
- Conflict prevention
- Lock hold during the entire transaction
- Does not support concurrency
- Used for critical consequences



Preventing inconsistent reads

- Optimistic control
 - Versioning
- Pessimistic control
 - Read ->shared lock
 - Write -> exclusive lock
- Temporal reads
 - Data+time stamps
 - Implies full history storage

Deadlocks

- Pick a victim
- Locks with deadlines
- Preventing:
 - Force to acquire all the necessary locks at the beginning
 - Enforce a strategy to grant locks (ex. Alphabetical order of the files)
- Combine tactics

Business Transactions

- ACID
- Transactional resource (ex. Database)
- Increase throughput -> short transactions
- Transactions mapped on a single request
- Late transactions -> read data first, start transaction for updates
- Transactions spanning several requests -> long transactions
- Lock escalation (row level -> table level)

Application Server Concurrency

- process-per-session
 - Uses a lot of resources
- process-per-request
 - Pooled processes
 - Sequential requests
 - Resources for a request should be released
- thread-per-request
 - More efficient
 - No isolation

How it works

- To implement it you need to:
 - know what type of locks you need,
 - build a lock manager,
 - define procedures for a business transaction to use locks
- Lock types
 - Exclusive write lock
 - Exclusive read lock
 - Read/write lock
 - Read and write locks are mutually exclusive.
 - Concurrent read locks are acceptable

Lock manager

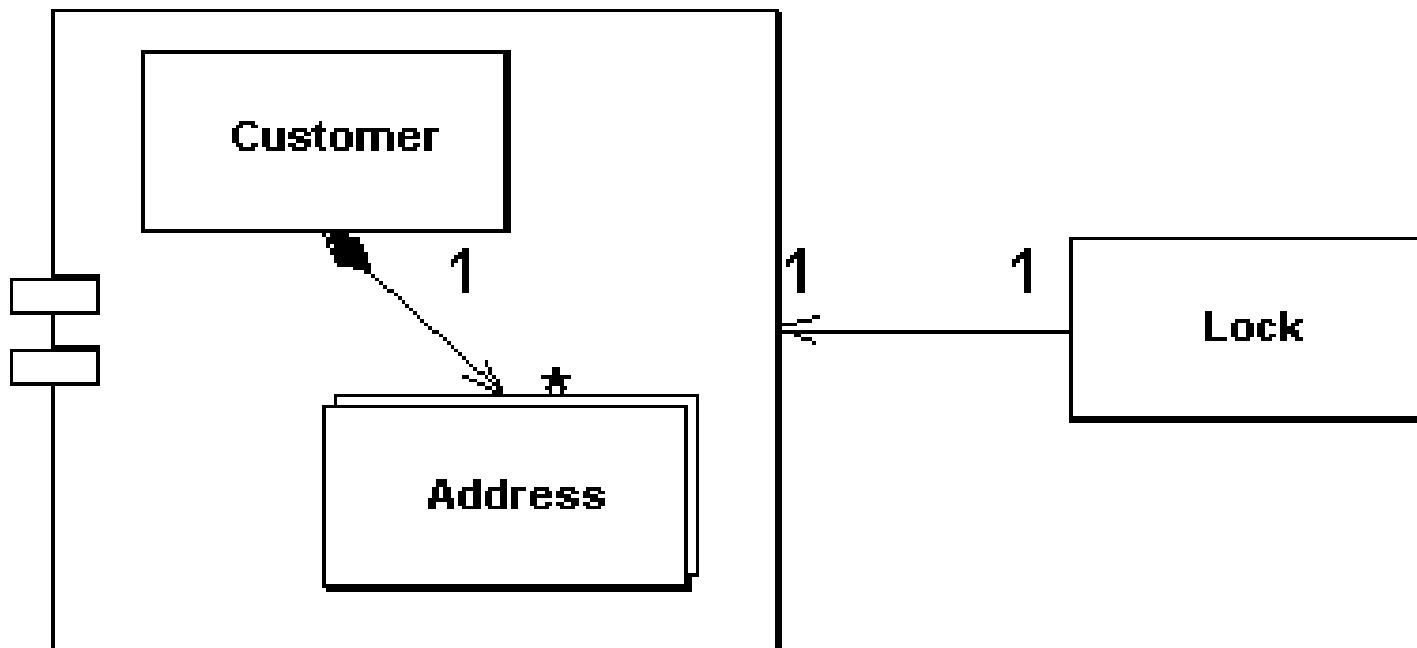
- The lock manager's job is to grant or deny any request by a business transaction to acquire or release a lock
- A table that maps locks to owners
- Protocol of Business transaction to use the lock manager
 - what to lock,
 - when to lock,
 - when to release a lock,
 - how to act when a lock cannot be acquired.

Analysis

- Access to the lock table must be serialized
- Performance bottleneck
- Consider granularity (Coarse grained lock)
- Possible deadlocks
- Lock timeout for lost sessions

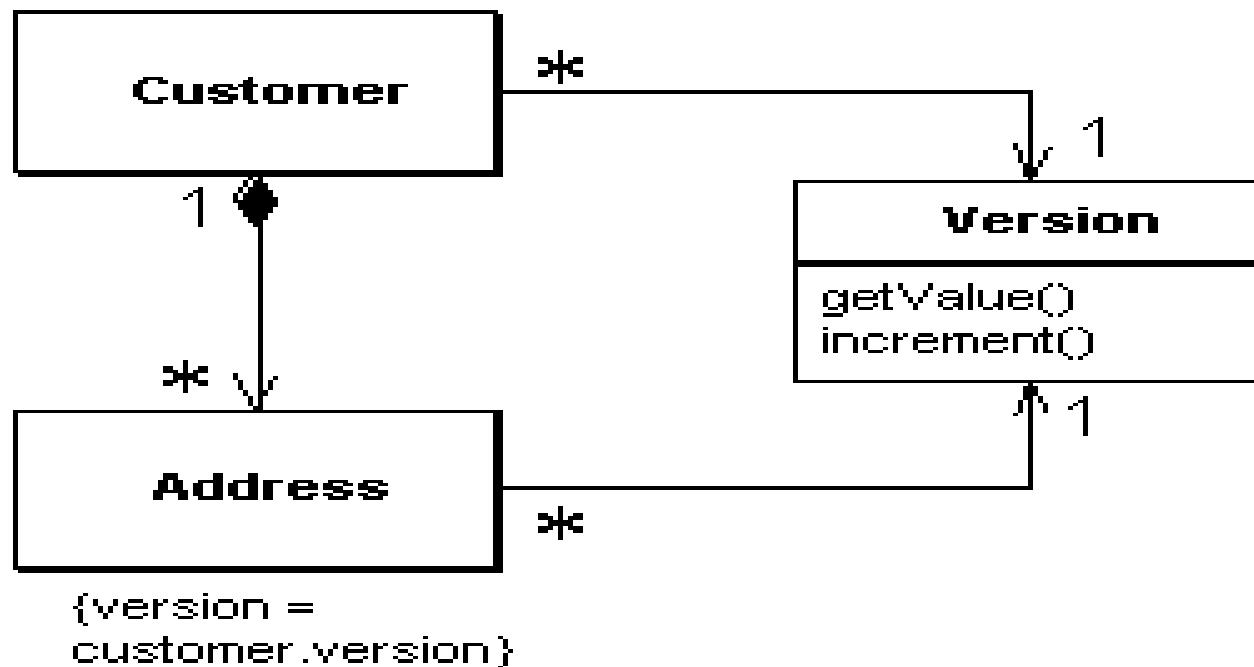
Coarse-Grained Lock

Lock a set of related objects with a single lock



How it works

- create a single point of contention for locking a group of objects
- Optimistic Lock –shared version



Next time

- More patterns ☺

SOFTWARE DESIGN

Presentation and general design patterns

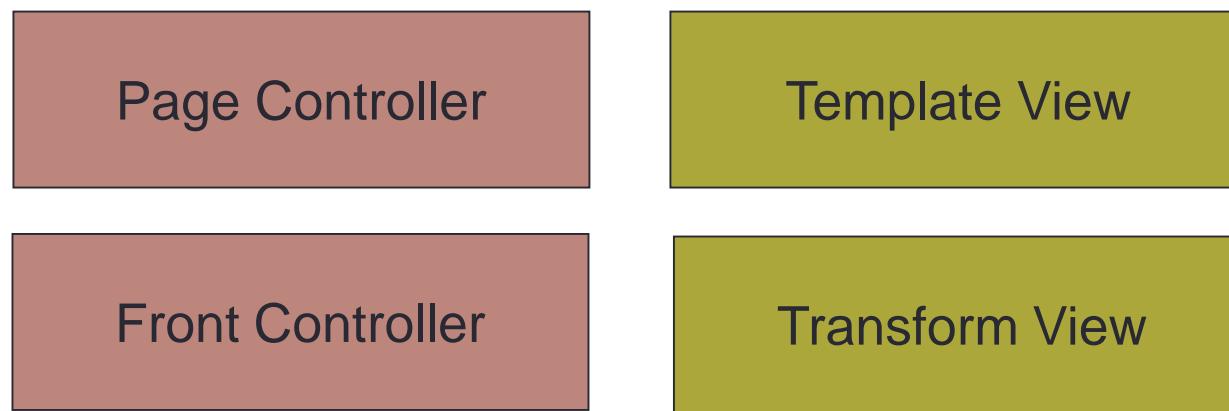
Content

- Presentation patterns
- General DP
 - Introduction
 - Creational
 - Structural
 - Behavioural

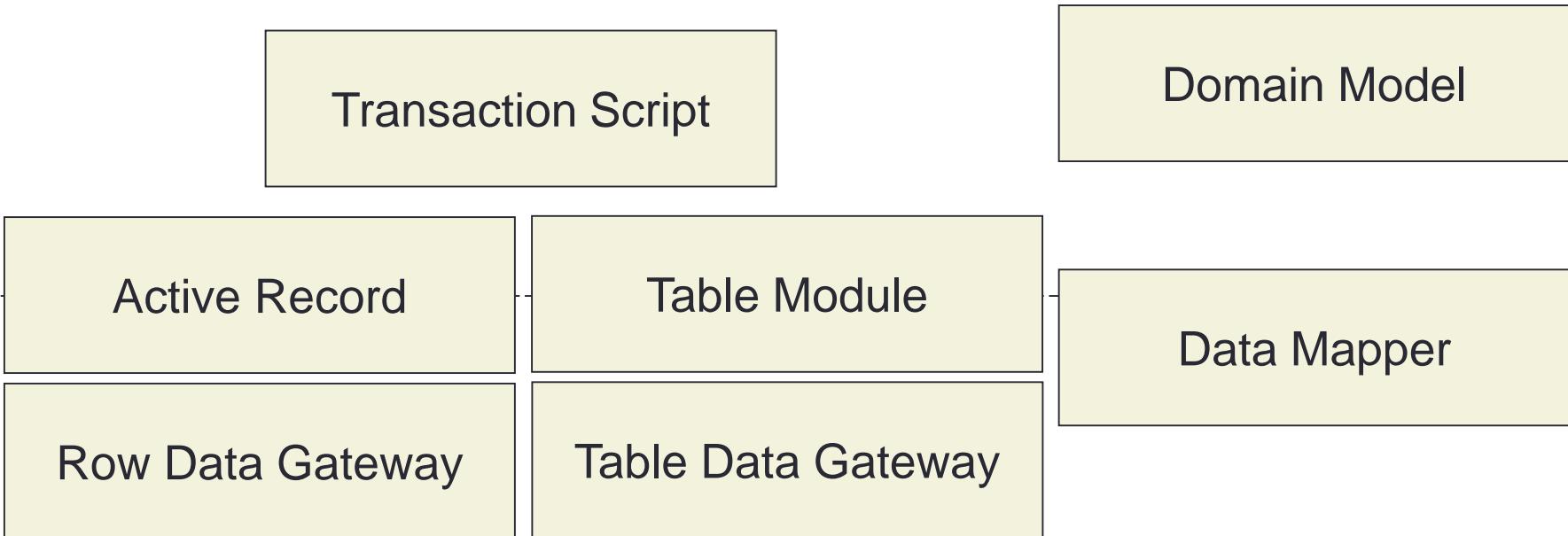
References

- Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]
- Microsoft Application Architecture Guide, 2009 [MAAG]
- SaaS Course Stanford
- Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.
- Univ. of Timisoara Course Materials
- Stuart Thiel, Enterprise Application Design Patterns: Improved and Applied, MSc Thesis, Concordia University Montreal, Quebec, Canada [Thiel]
- Ólafur Andri Ragnarsson, Presentation Layer Design, 2014.

Presentation



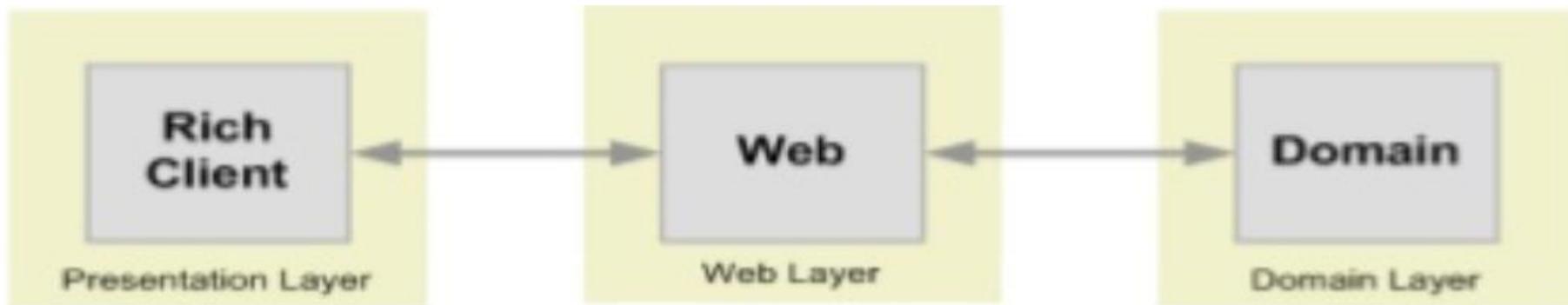
Domain



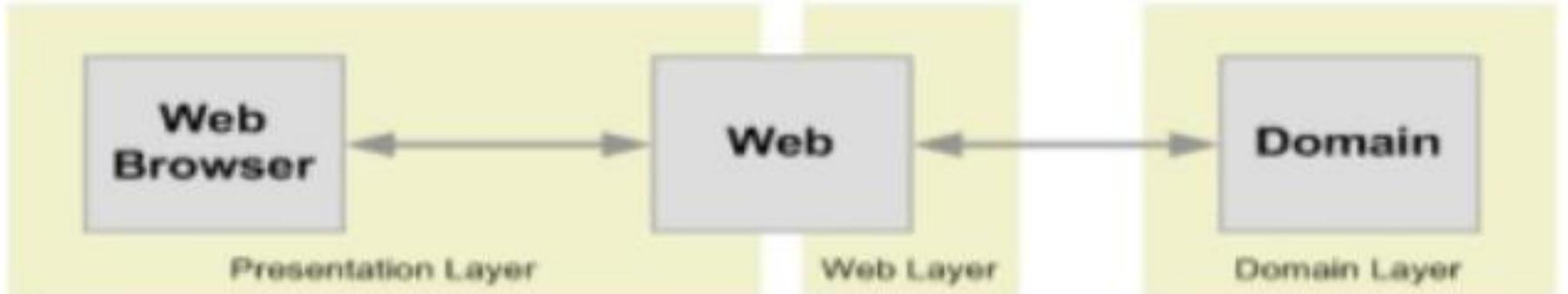
Data Source

Presentation and Web layer

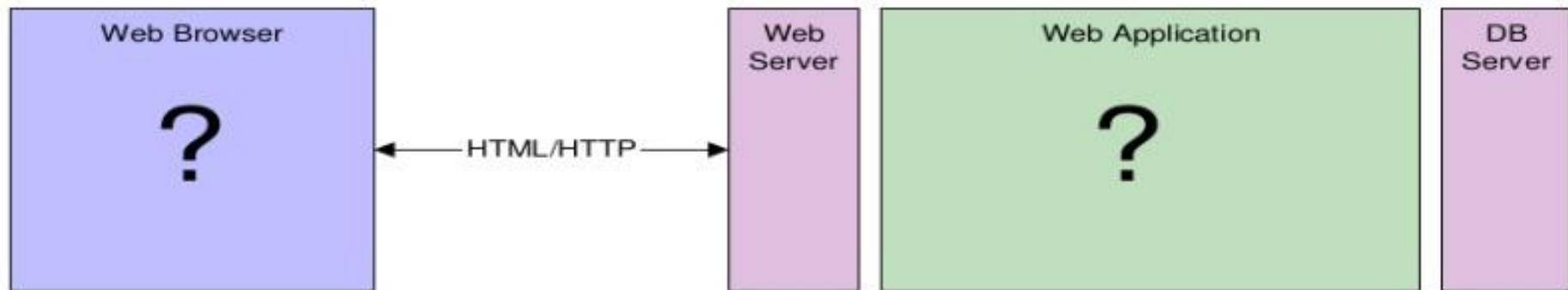
- Desktop and Embedded applications



- Web Browser



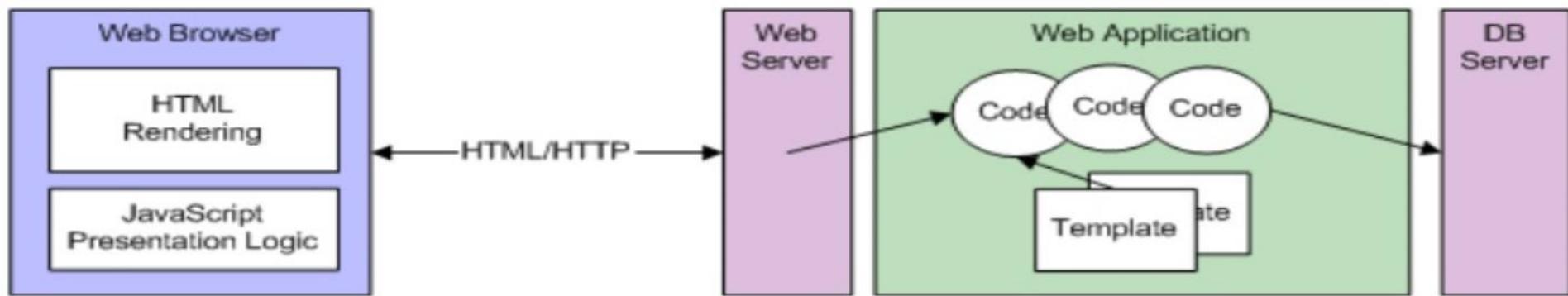
Web Layer Design



- Two approaches
 - Script based – Code using HTML
 - Server Pages based – HTML page with code

Script based

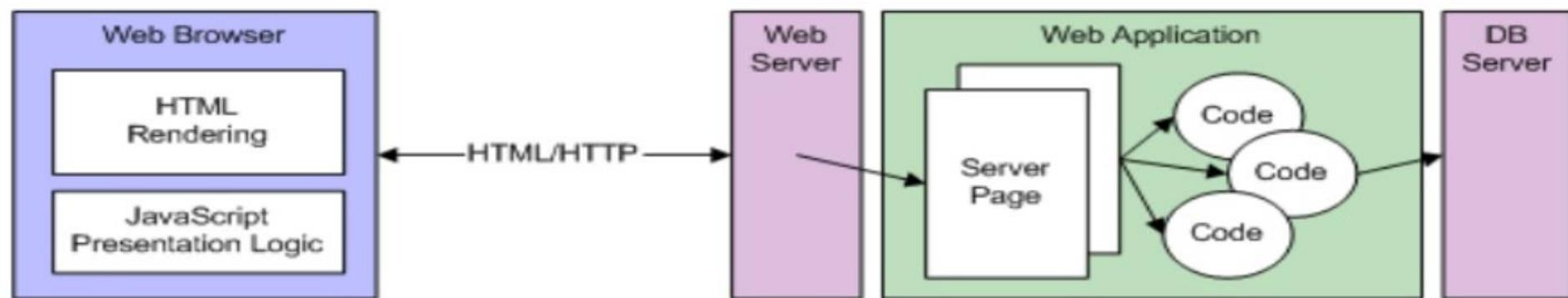
- Useful when the logic and flow are important
 - Request is not easily mapped on a single page



- Examples: CGI, ISAPI, Java Servlets

Server Page Based

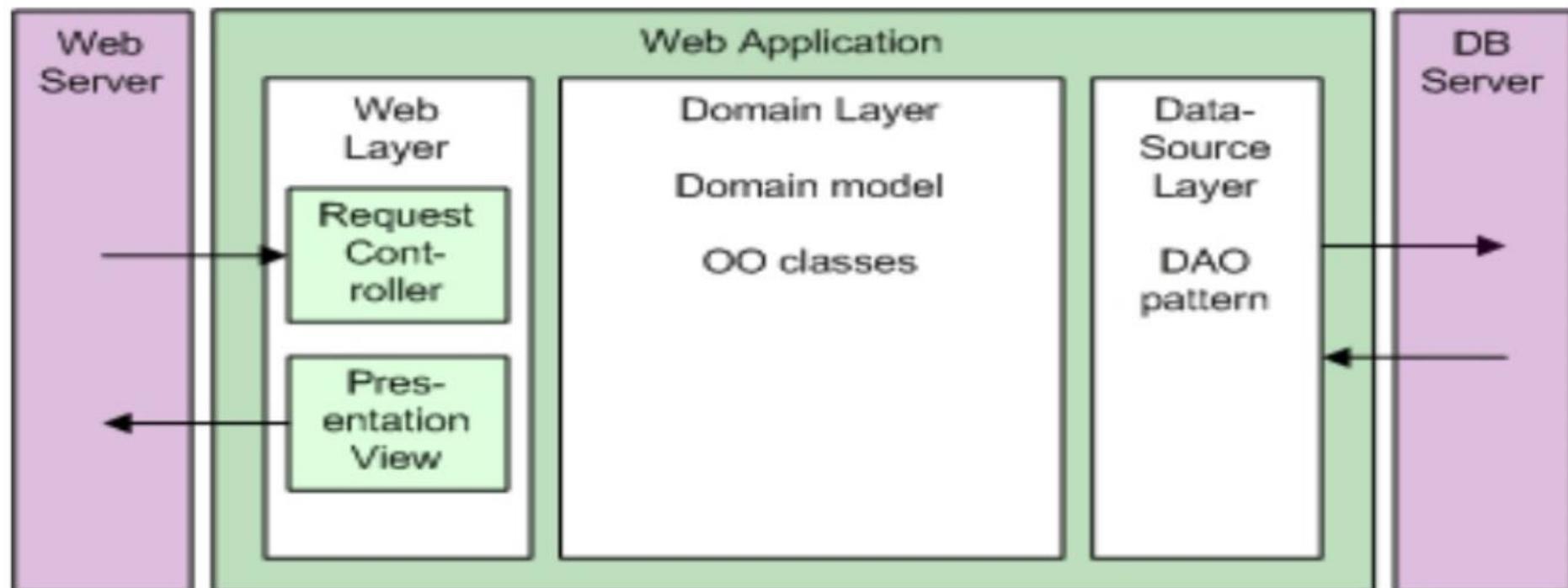
- Useful when there are lots of pages
 - Flow is not as important
 - Each request can be easily mapped to a page



- Examples: PHP, JSP, ASP

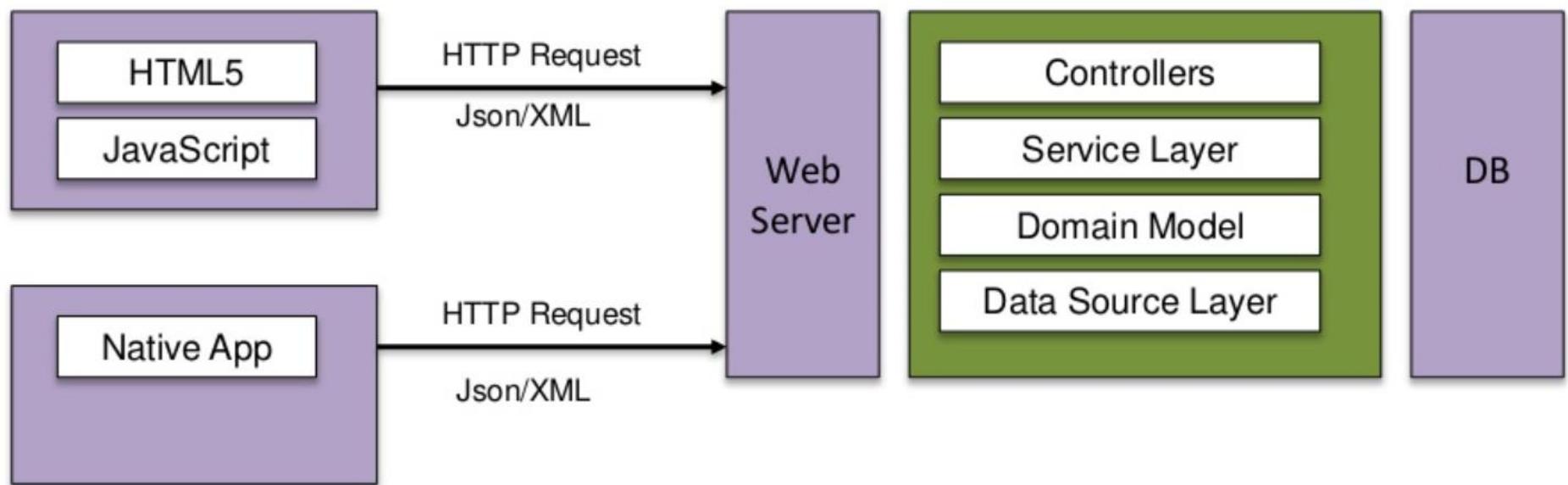
Web layer

- Must handle the request and the response



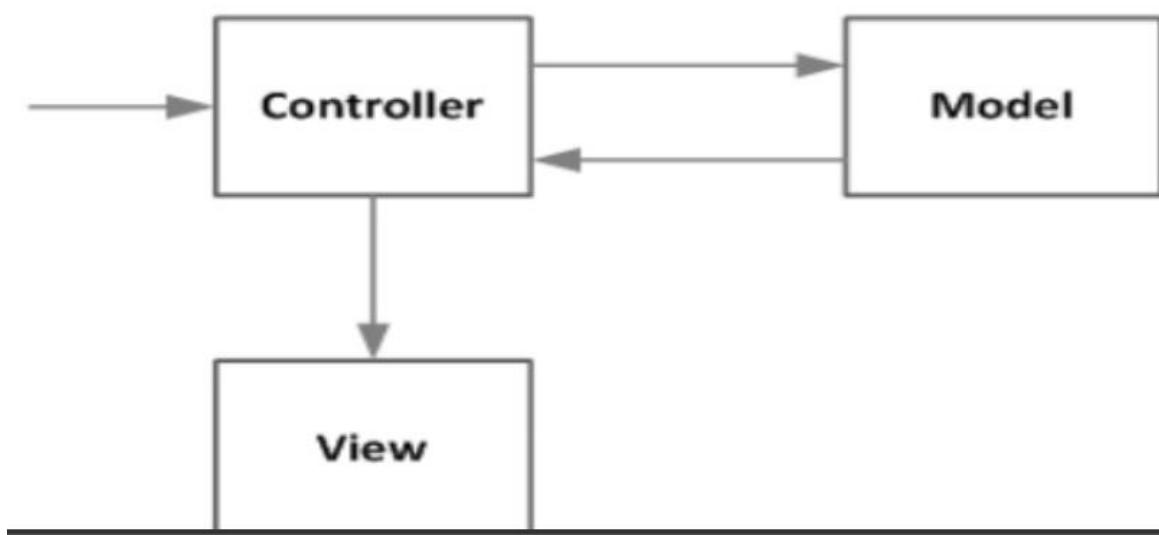
API based

- User Interface is HTML 5 or Native App
- Server side API
- Information format is XML or JSON

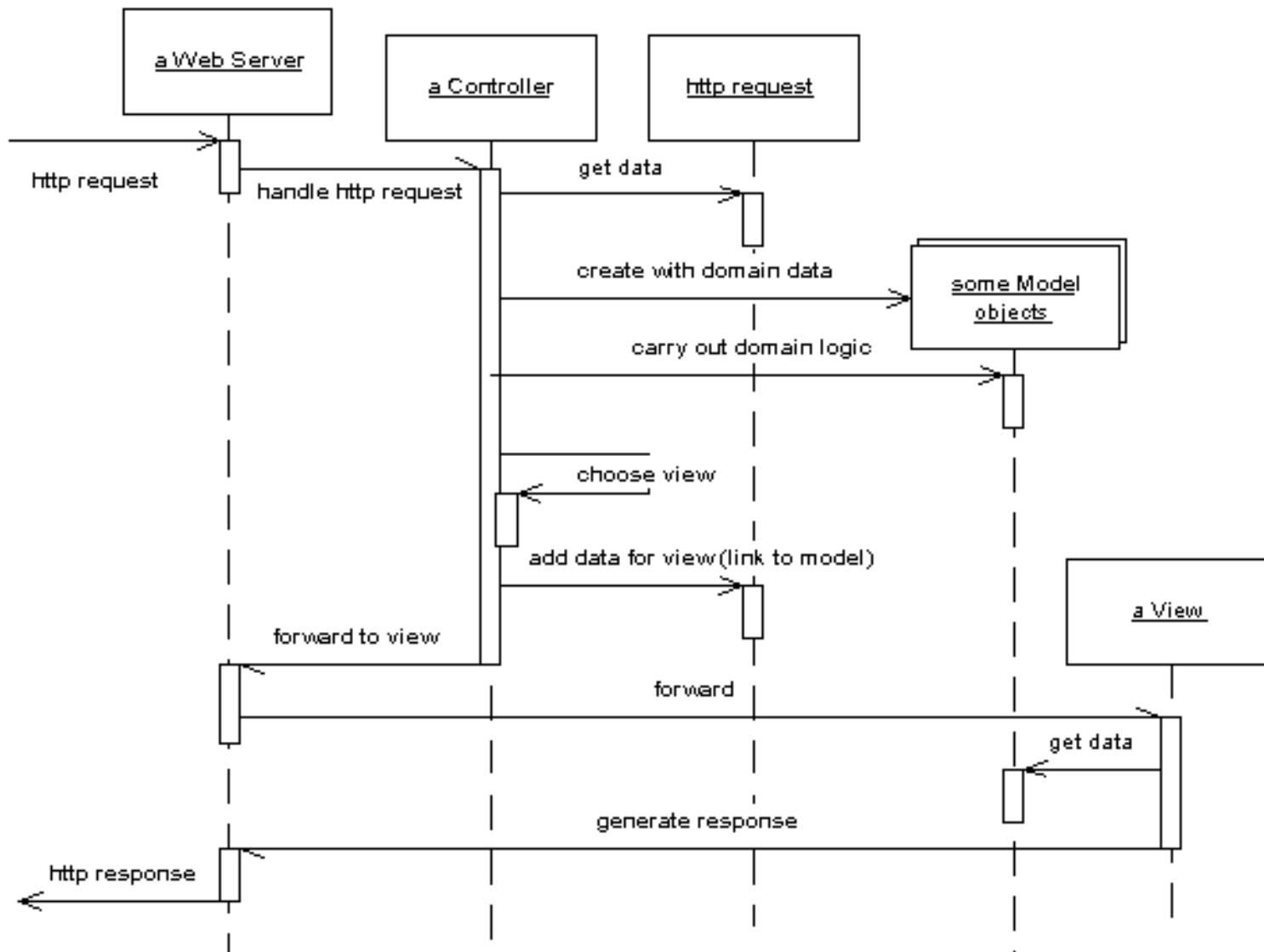


MVC in Web applications

- Input Controller
 - Takes the request
 - Examines the input parameters
 - Calls the Model
 - Handles the response
 - Sends the control to the View for rendering

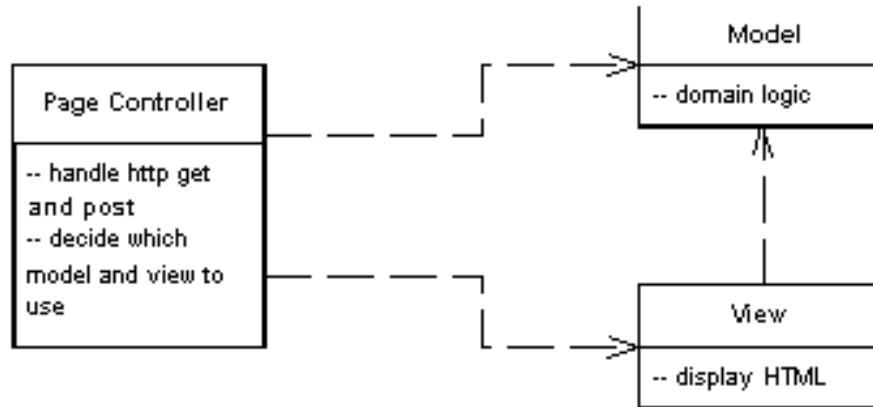


MVC in action

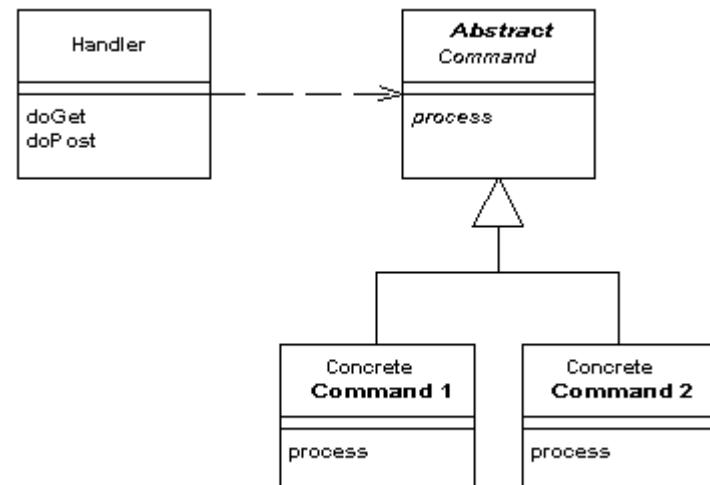


Controllers

- One for each page – Page controller



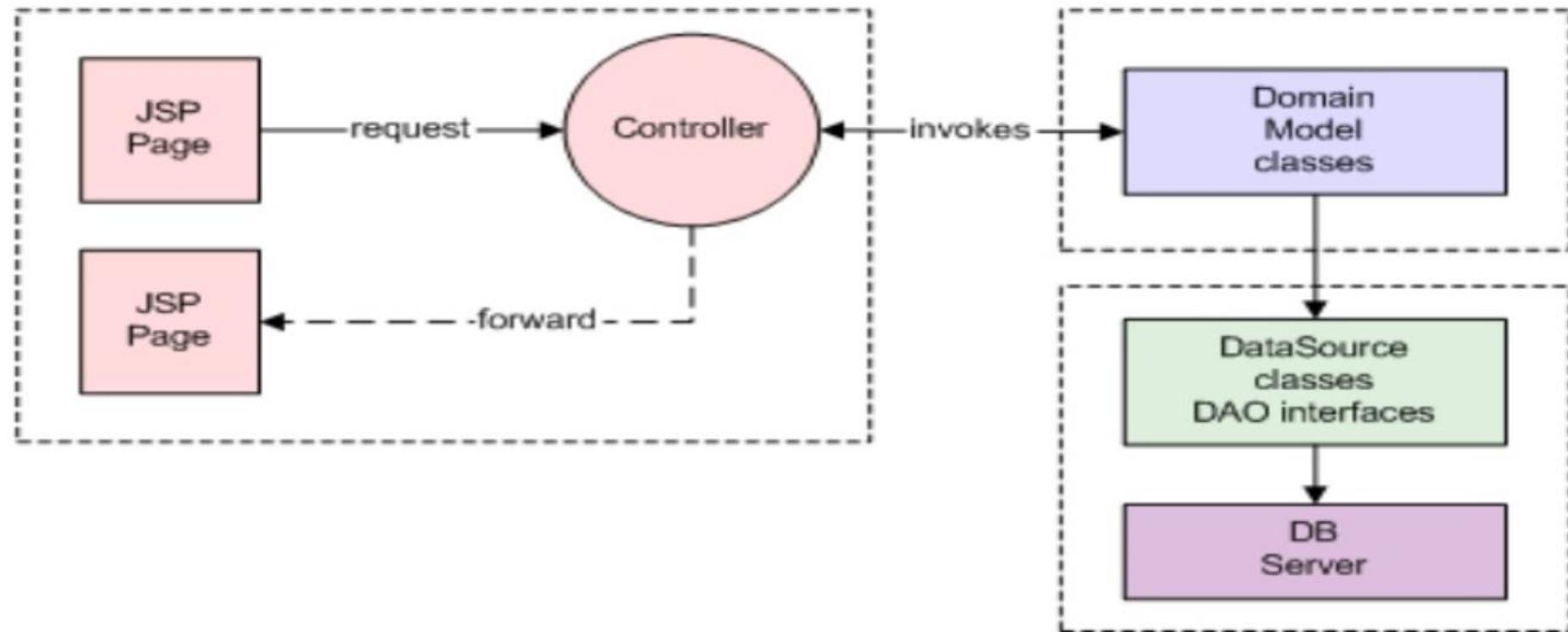
- One per application – Front controller



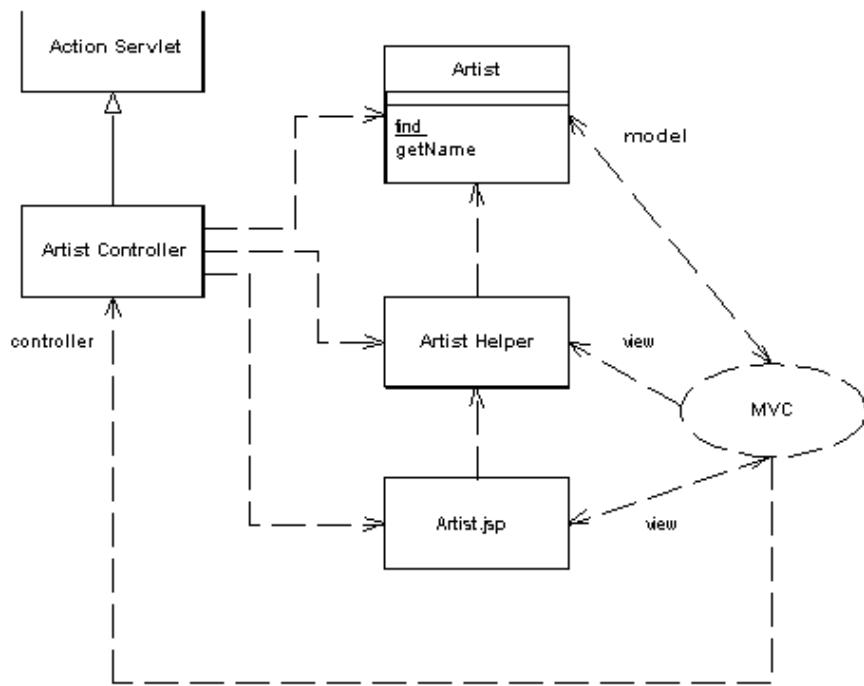
Page Controller

- As Script
 - Servlet or CGI program
 - Applications that need logic and data
- As Server Page
 - ASP, PHP, JSP
 - Use helpers to get data from the model
 - Logic is simple to none
- Basic responsibilities
 - **Decode the URL** and extract all data for the action.
 - **Create and invoke any model objects** to process the data. All relevant data from the HTML request should be passed to the model so that the model objects don't need any connection to the HTML request.
 - **Determine which view** should display the result page and forward the model information to it.

Page Controller



Servlet controller and a JSP view (Java)



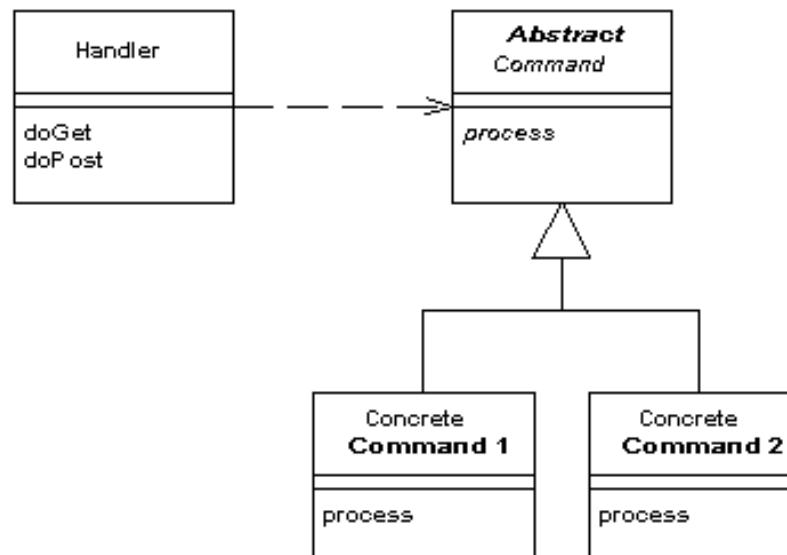
```
class ArtistController...
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        Artist artist = Artist.findNamed(request.getParameter("name"));
        if (artist == null)
            forward("/MissingArtistError.jsp", request, response);
        else {
            request.setAttribute("helper", new ArtistHelper(artist));
            forward("/artist.jsp", request, response);
        }
    }
```

```
<servlet>
<servlet-name>artist</servlet-name>
<servlet-
class>actionController.ArtistController
</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>artist</servlet-name>
<url-pattern>/artist</url-pattern>
</servlet-mapping>
```

Front controller

- One controller handles all requests
- usually structured in two parts:
 - a web handler (rather a class than a server page)
 - a hierarchy of commands (classes)

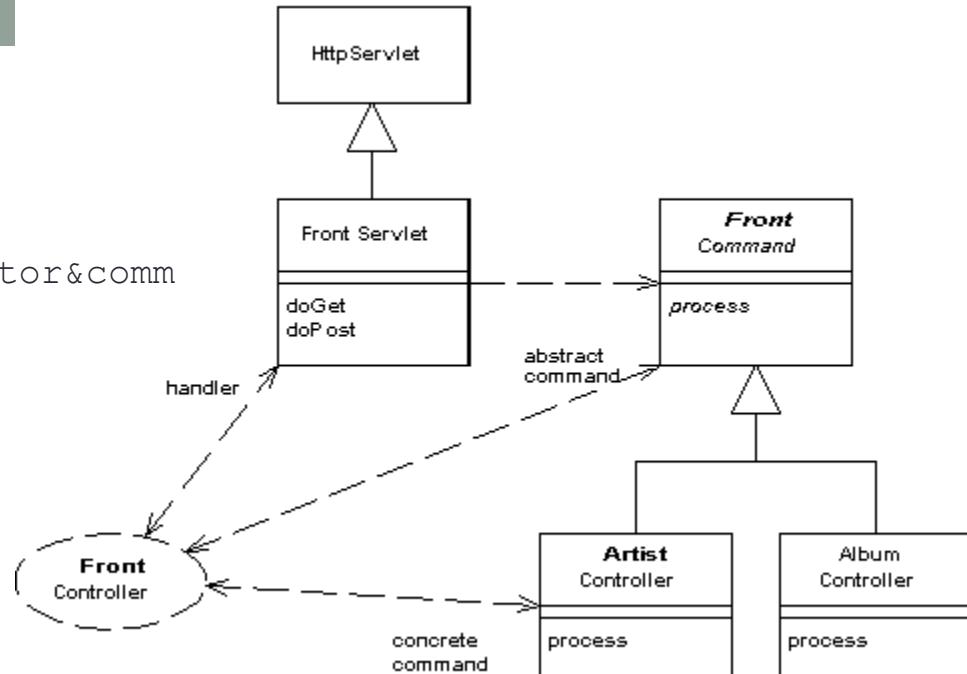


Front controller

- Handler decides what command:
 - **statically**
 - parses the URL and uses conditional logic;
 - advantage of explicit logic,
 - compile time error checking on dispatch,
 - flexibility in URL look-up
 - **dynamically**
 - takes a standard piece of the URL and uses dynamic instantiation to create a command class;
 - allows to add new commands without changing the Web handler;
 - can put the name of the command class into the URL or can use a properties file that binds URLs to command class names.

Example

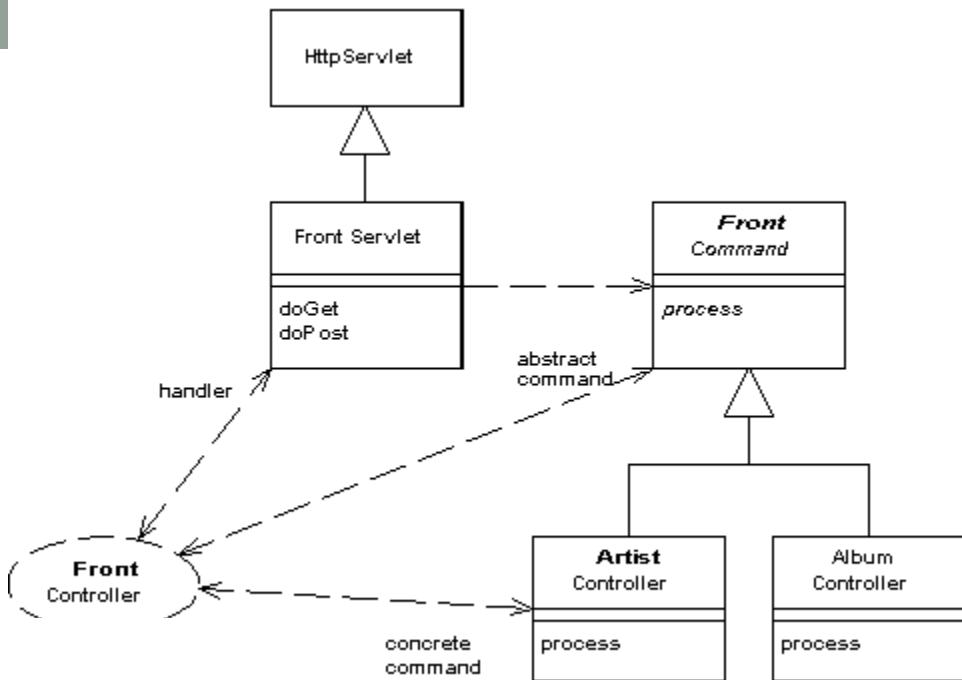
http://localhost:8080/isa/music?name=astor&command=Artist



```
class FrontServlet...
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        FrontCommand command = getCommand(request);
        command.init(getServletContext(), request, response);
        command.process();
    }

    private FrontCommand getCommand(HttpServletRequest request) {
        try {
            return (FrontCommand) getCommandClass(request).newInstance();
        } catch (Exception e) {
            throw new ApplicationException(e);
        }
    }

    private Class getCommandClass(HttpServletRequest request) {
        Class result;
        final String commandClassName =
            "frontController." + (String) request.getParameter("command") + "Command";
        try {
            result = Class.forName(commandClassName);
        } catch (ClassNotFoundException e) {
            result = UnknownCommand.class;
        }
        return result;
    }
```



```

class FrontCommand...
    protected ServletContext context;
    protected HttpServletRequest request;
    protected HttpServletResponse response;

    public void init(ServletContext context,
                     HttpServletRequest request,
                     HttpServletResponse response)
    {
        this.context = context;
        this.request = request;
        this.response = response;
    }

    abstract public void process() throws ServletException, IOException ;

    protected void forward(String target) throws ServletException, IOException
    {
        RequestDispatcher dispatcher = context.getRequestDispatcher(target);
        dispatcher.forward(request, response);
    }
}
  
```

Discussion

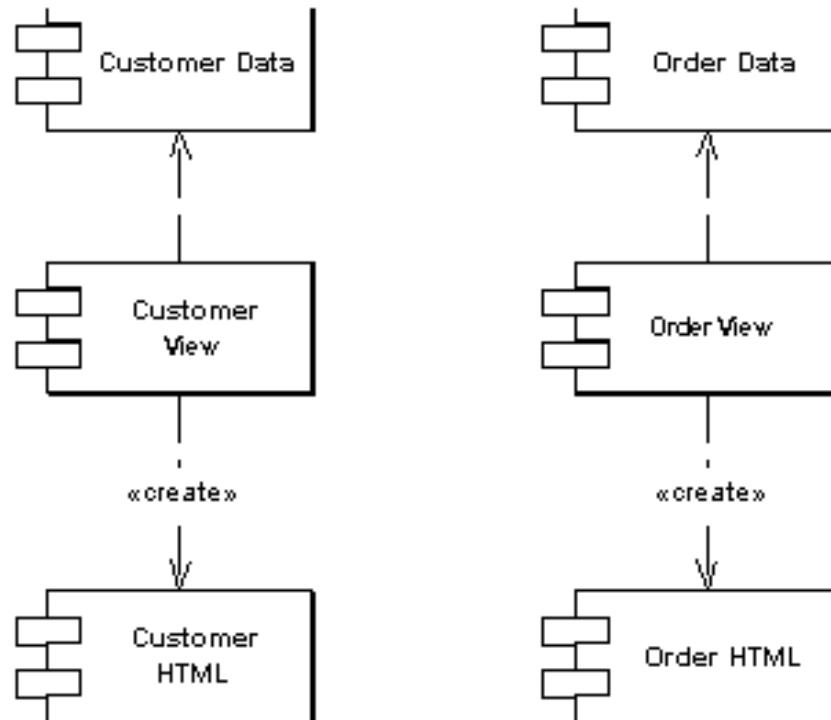
- Only one Front Controller has to be configured into the Web server
- with dynamic commands you can add new commands without changing anything.
- because new command objects are created with each request, its thread safe.
- Re-factor code better in command hierarchy

Discussion

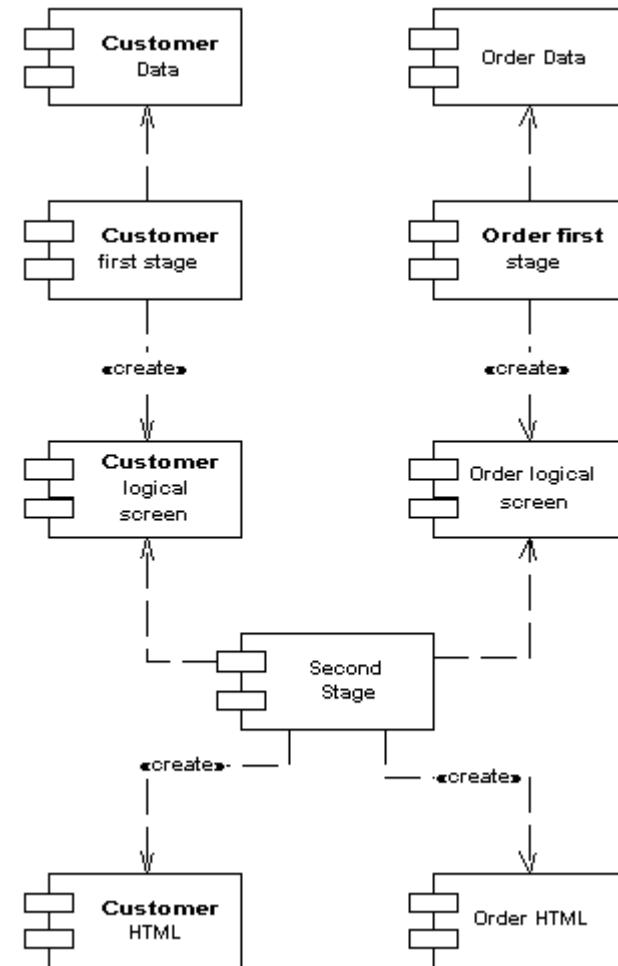
- **Page Controller:**
 - simple controller logic
 - a natural structuring mechanism where particular actions are handled by particular server pages or script classes.
- **Front Controller:**
 - greater complexity;
 - handles duplicated features (i.e. security, internationalization, providing particular views for certain kinds of users) in one place.
 - single point of entry for centralized logic

View

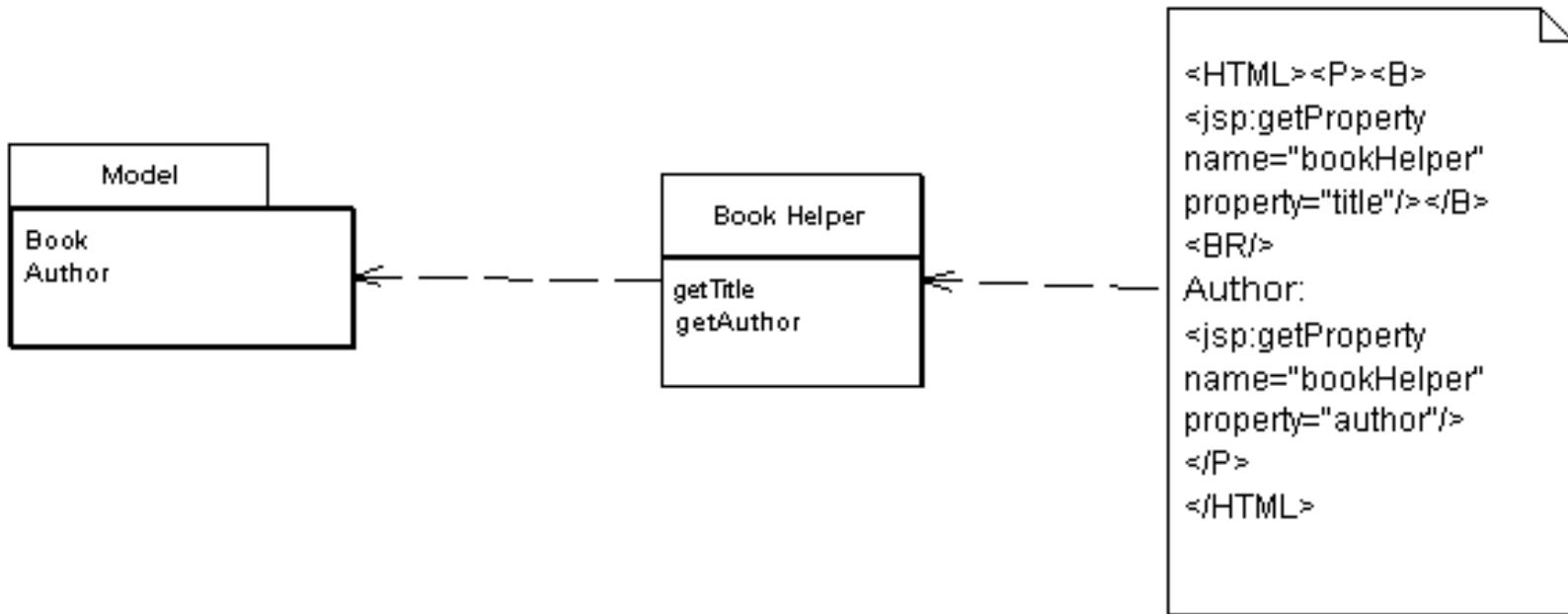
- Single step stage



- Two step stage



Template View



- embed markers into a static HTML page when it's written
- When the page is used to service requests, the markers are replaced by the results of some computation
- **server pages:**
 - ASP, JSP, or PHP.
 - allow to embed arbitrary programming logic, referred to as **scriptlets**, into the page.

Conditional display

```
<IF condition = "$pricedrop > 0.1"> ...show some stuff </IF>
```

Templates become programming languages

⇒ Move the condition to the helper to generate the content

⇒ What if the content should be displayed but in different ways?

- Helper generates the markup
- OR use focused tags:

```
<IF expression = "isHighSelling()"><B></IF><property name =  
"price"/><IF expression = "isHighSelling()"></B></IF>
```

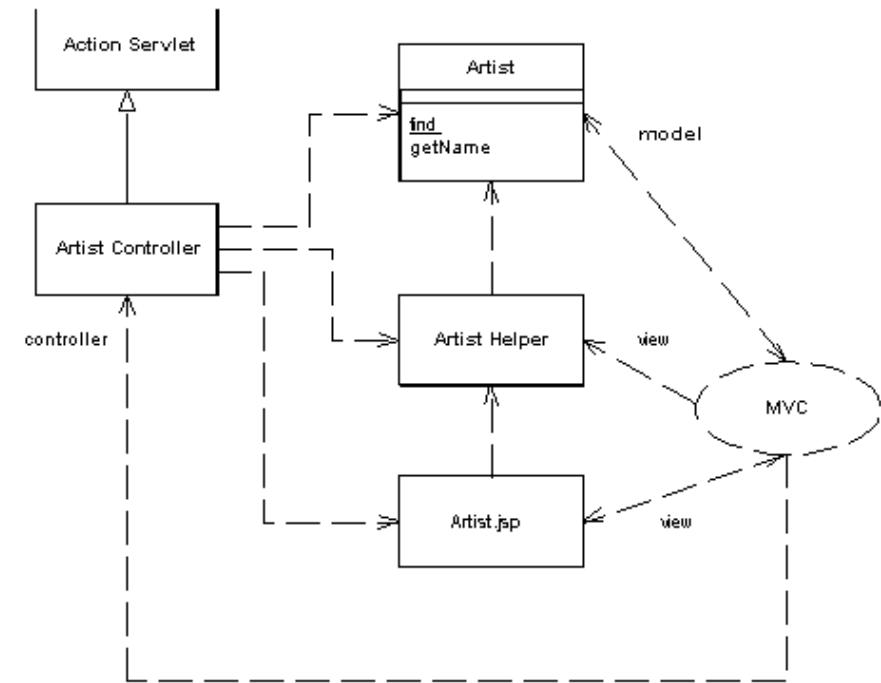
replaced by

```
<highlight condition = "isHighSelling" style =  
"bold"><property name = "price"/></highlight>
```

JSP Template View (see Page Controller)

```
<jsp:useBean id="helper" type="actionController.ArtistHelper" scope="request"/>
```

```
class ArtistHelper...  
    private Artist artist;  
  
    public ArtistHelper(Artist artist) {  
        this.artist = artist;  
    }  
  
    public String getName() {  
        return artist.getName();  
    }
```



```
<B> <%=helper.getName()%></B>
```

```
<B><jsp:getProperty name="helper" property="name"/></B>
```

Show a list of albums for an artist

```
<UL>
<%
    for (Iterator it = helper.getAlbums().iterator(); it.hasNext();) {
        Album album = (Album) it.next();%>
<LI><%=album.getTitle()%></LI>

<%
    }    %>
</UL>

class ArtistHelper...
    public String getAlbumList() {
        StringBuffer result = new StringBuffer();
        result.append("<UL>");
        for (Iterator it = getAlbums().iterator(); it.hasNext();) {
            Album album = (Album) it.next();
            result.append("<LI>");
            result.append(album.getTitle());
            result.append("</LI>");
        }
        result.append("</UL>");
        return result.toString();
    }

    public List getAlbums() {
        return artist.getAlbums();
    }

    <UL><tag:forEach host = "helper" collection = "albums" id = "each">
        <LI><jsp:getProperty name="each" property="title"/></LI>
    </tag:forEach></UL>
```

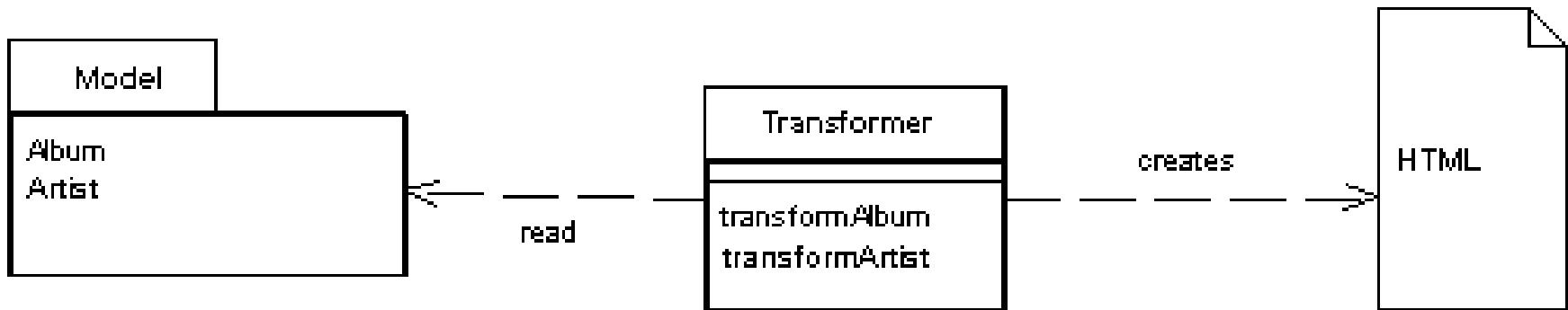


Discussion

- Benefits:
 - Compose the structure of the page based on the template
 - Separate design from code (helper)
- Liabilities
 - common implementations make it too easy to put complicated logic onto the page => hard to maintain
 - Harder to test than Transform View

Transform View

- Input: Model
- Output: HTML



- can be written in any language, yet the dominant choice is XSLT (EXtensible Stylesheet Language Transformation).
- Input: XML
- XML data can be returned as:
 - natural return type
 - output type which can be transformed to XML automatically
 - Data Transfer Object, that can serialize as XML

Translated in code

```
class AlbumCommand...
    public void process() {
        try {
            Album album = Album.findNamed(request.getParameter("name"));
            Assert.notNull(album);
            PrintWriter out = response.getWriter();
            XsltProcessor processor = new SingleStepXsltProcessor("album.xsl");
            out.print(processor.getTransformation(album.toXmlDocument()));
        } catch (Exception e) {
            throw new A
        }
    }

<album>
    <title>Zero Hour</title>
    <artist>Astor Piazzola</artist>
    <trackList>
        <track><title>Tanguedia III</title><time>4:39</time></track>
        <track><title>Milonga del Angel</title><time>6:30</time></track>
        <track><title>Concierto Para Quinteto</title><time>9:00</time></track>
        <track><title>Milonga Loca</title><time>3:05</time></track>
        <track><title>Michelangelo '70</title><time>2:50</time></track>
        <track><title>Contrabajisimo</title><time>10:18</time></track>
        <track><title>Mumuki</title><time>9:32</time></track>
    </trackList>
</album>
```

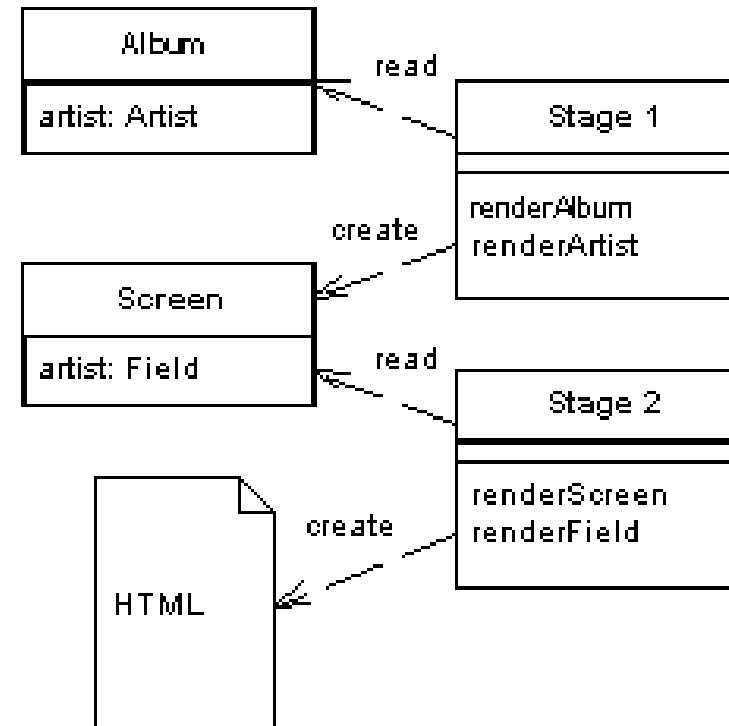
Advantages

- Views are easily testable independent of web servers
- Avoid too much logic in view
- For changing the website look often, one needs to change the transform programs. Using common transforms with XSLT includes, reduce this problem.

Two step view

Multi-page application:

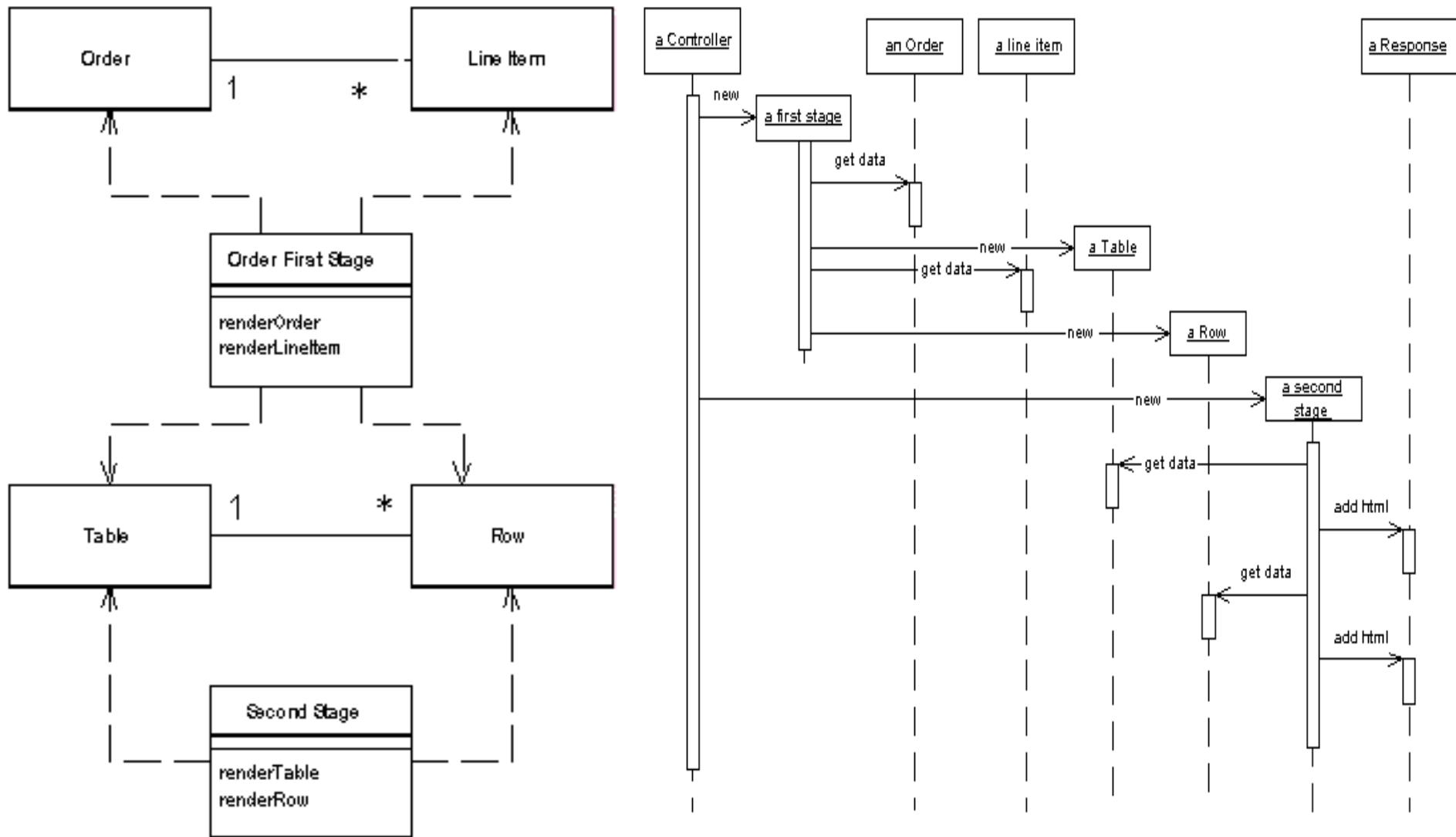
- transforms the model data into a logical presentation without any specific formatting
- converts that logical presentation with the actual formatting needed.



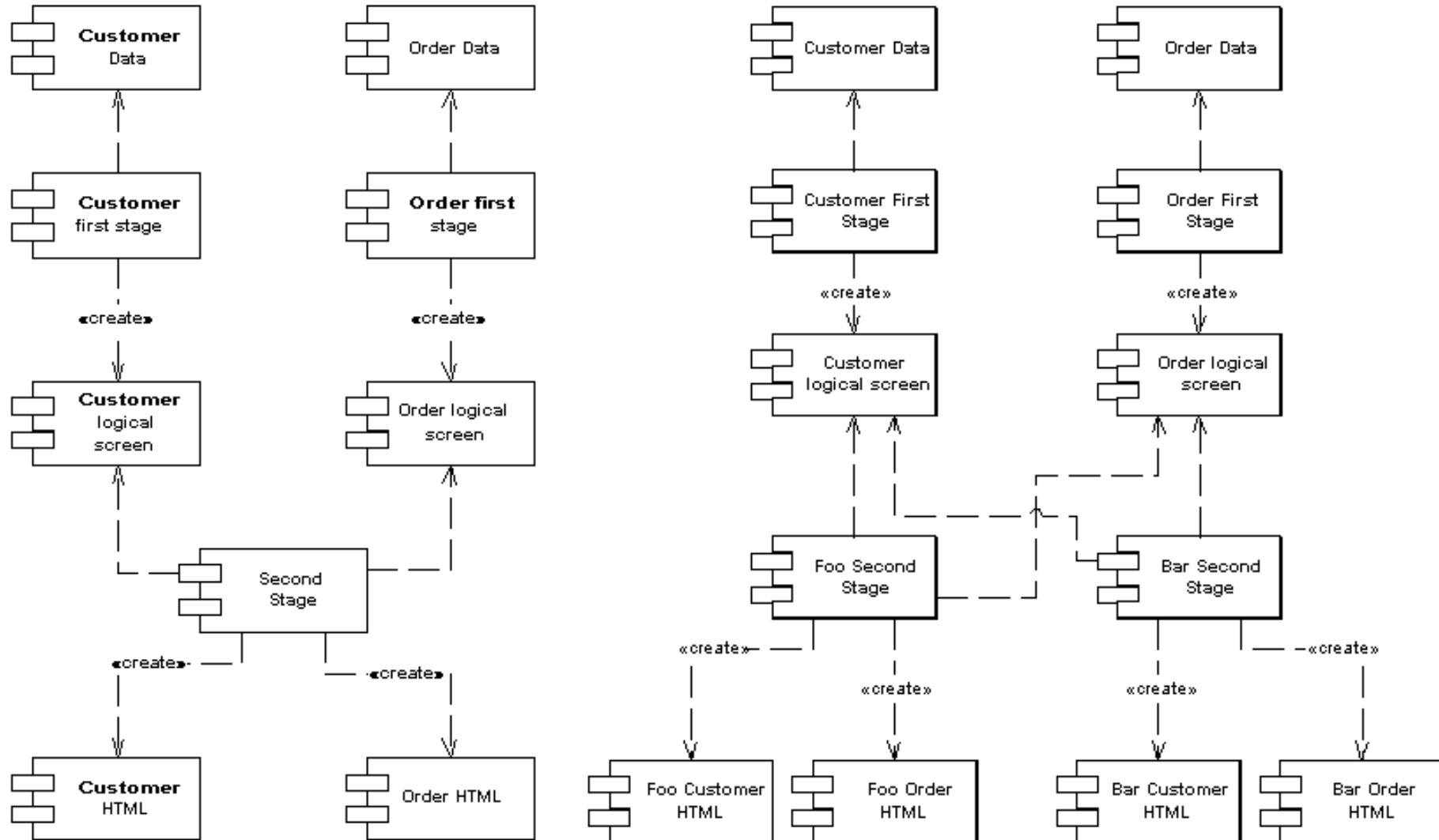
How to do it

- two-step XSLT:
 - domain-oriented XML => presentation-oriented XML,
 - presentation-oriented XML => HTML.
- presentation-oriented structure as a set of classes (table/row class):
 - domain information instantiates T/R classes.
 - renders the T/R classes into HTML
 - each presentation-oriented class generates HTML for itself or
 - having a separate HTML renderer class
- Template View based approach
 - The template system converts the logical tags into HTML.

Example



One vs. two appearances

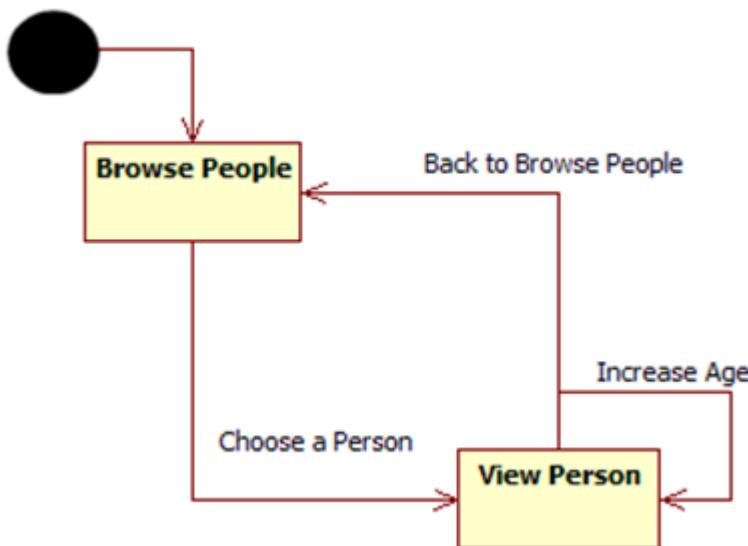


Discussion

- Advantages:
 - Two step view solves the difficulty with Transform view w.r.t. multiple transforms module & global changes.
 - In addition, if the website has multiple appearances/themes, the complexity is higher. With two step view, the issue is resolved and the advantage is compounded with multiple pages/themes.
- Liabilities:
 - It's hard to find enough commonality between the screens to get a simple enough presentation-oriented structure
 - Not for designers/non-programmers. Programmers have to write code for different rendering.
 - Harder programming model to learn
 - Complexity increases if multiple devices have to be supported. Then, the logical structure has to be common for multiple devices too.

Putting it all together [Thiel]

- People management web app
 - Browse people
 - View person
 - Change person data (i.e. increase age)

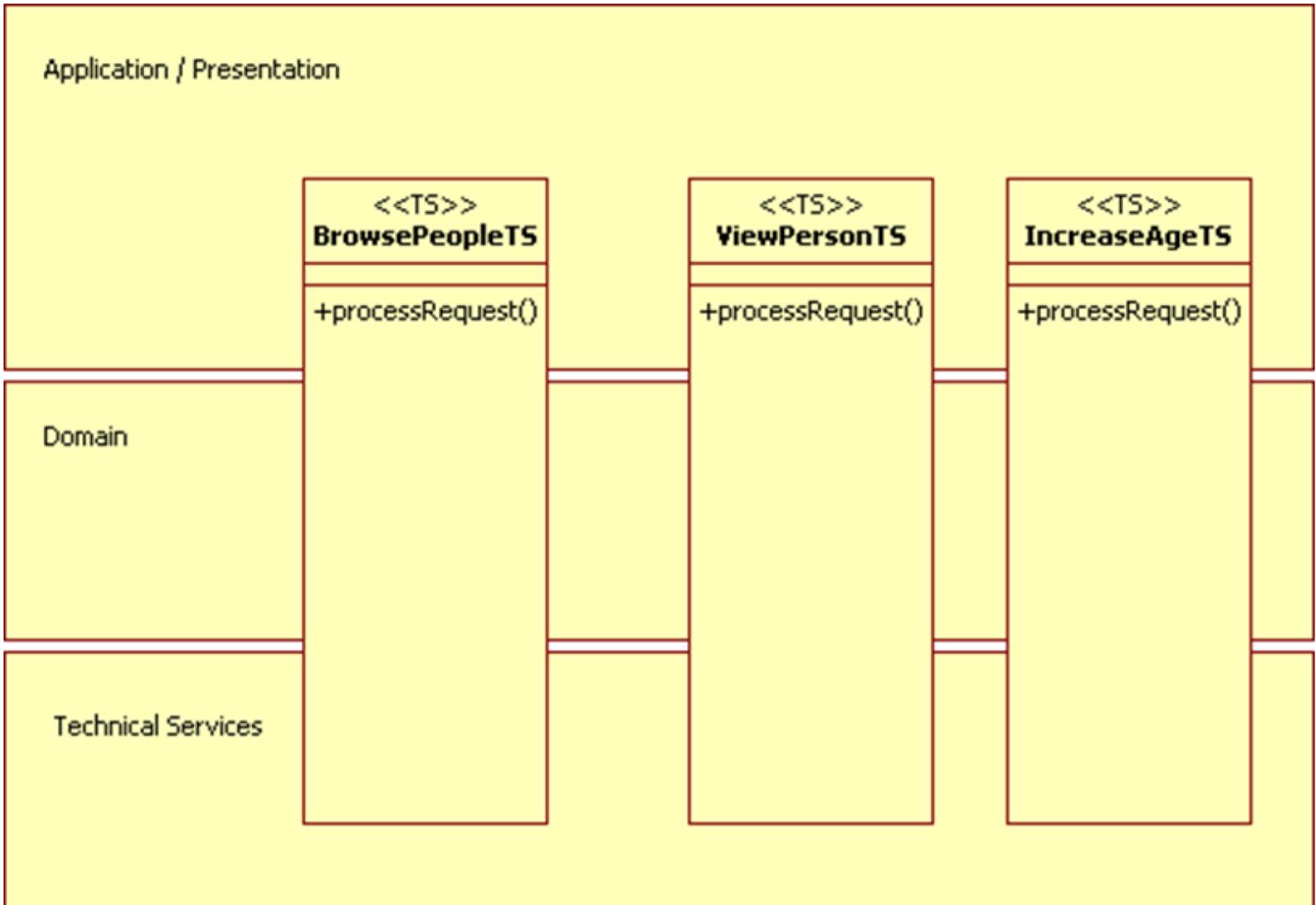


Please choose a person to see their age

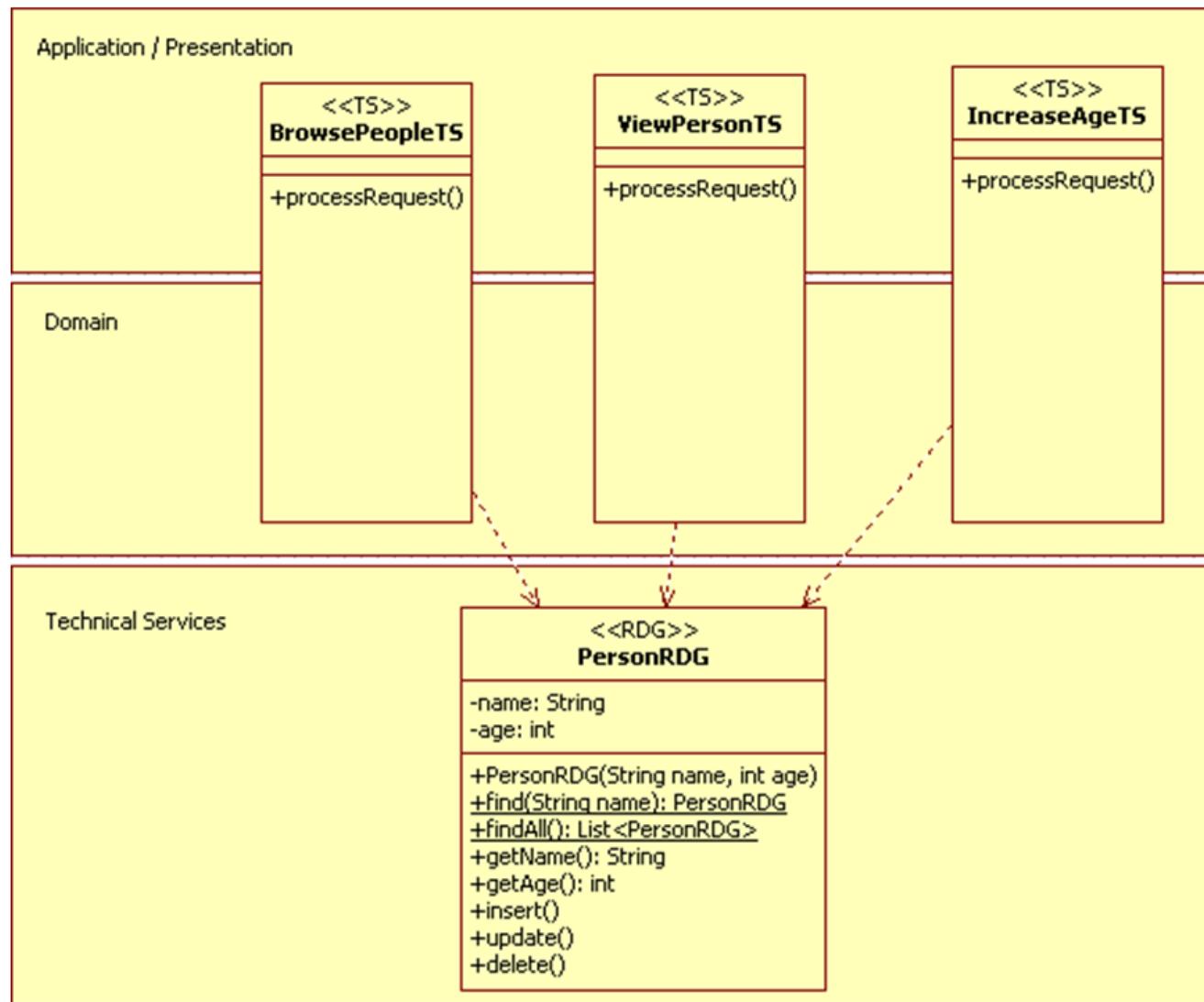
Alice
 Bob
 Chuck
 Dave
 Edith

Choose This Person!

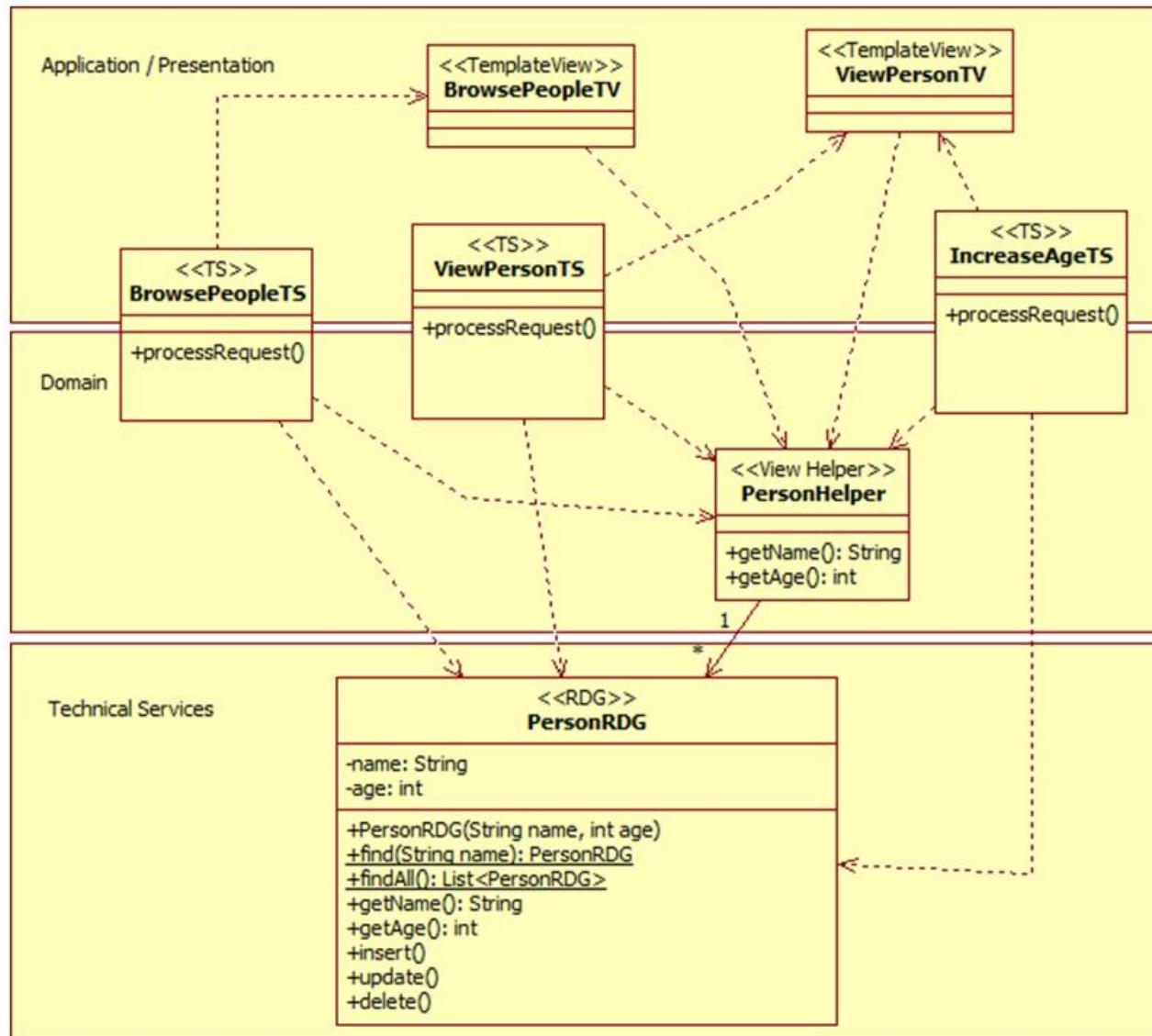
Just TS



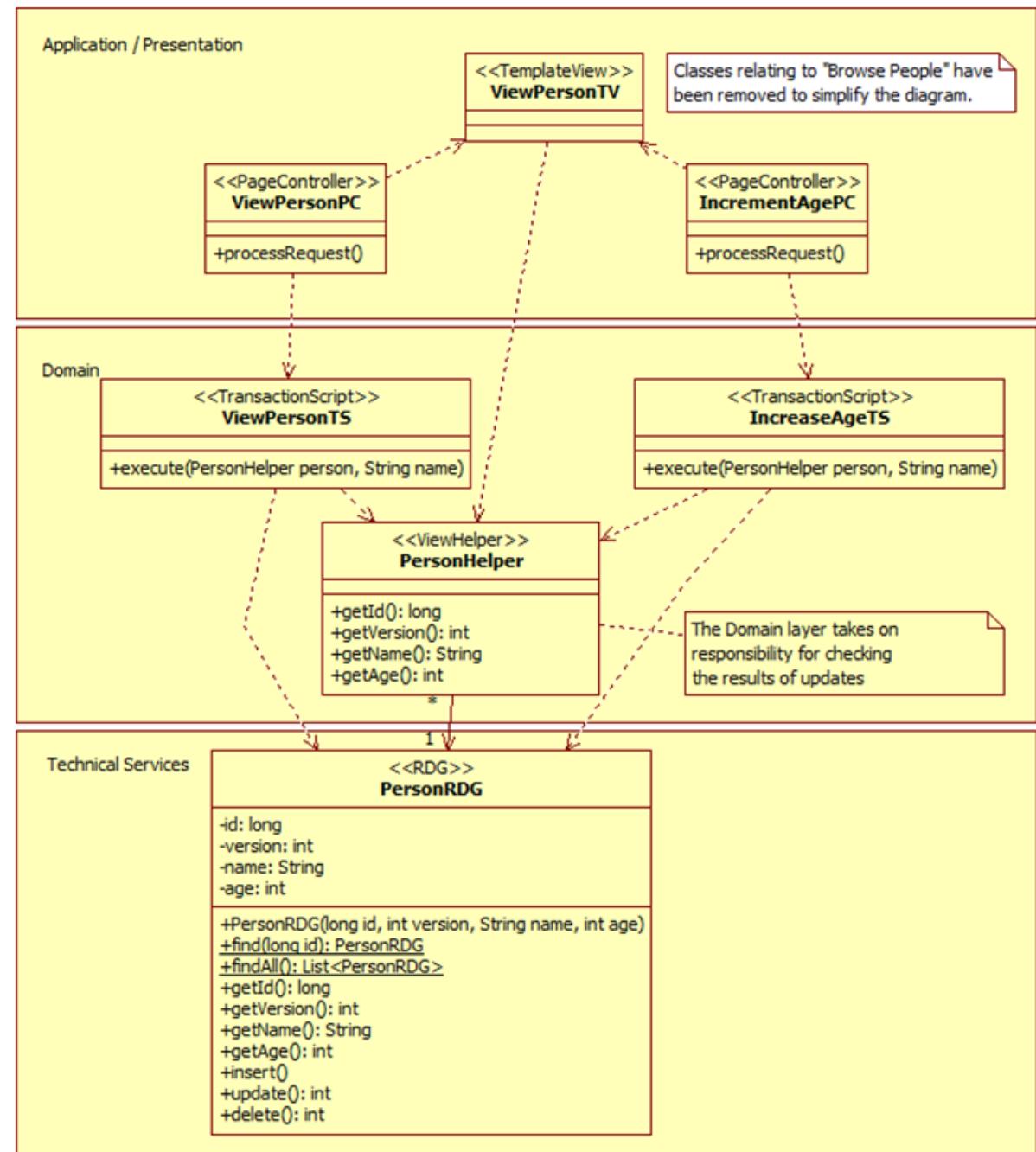
TS + RDG



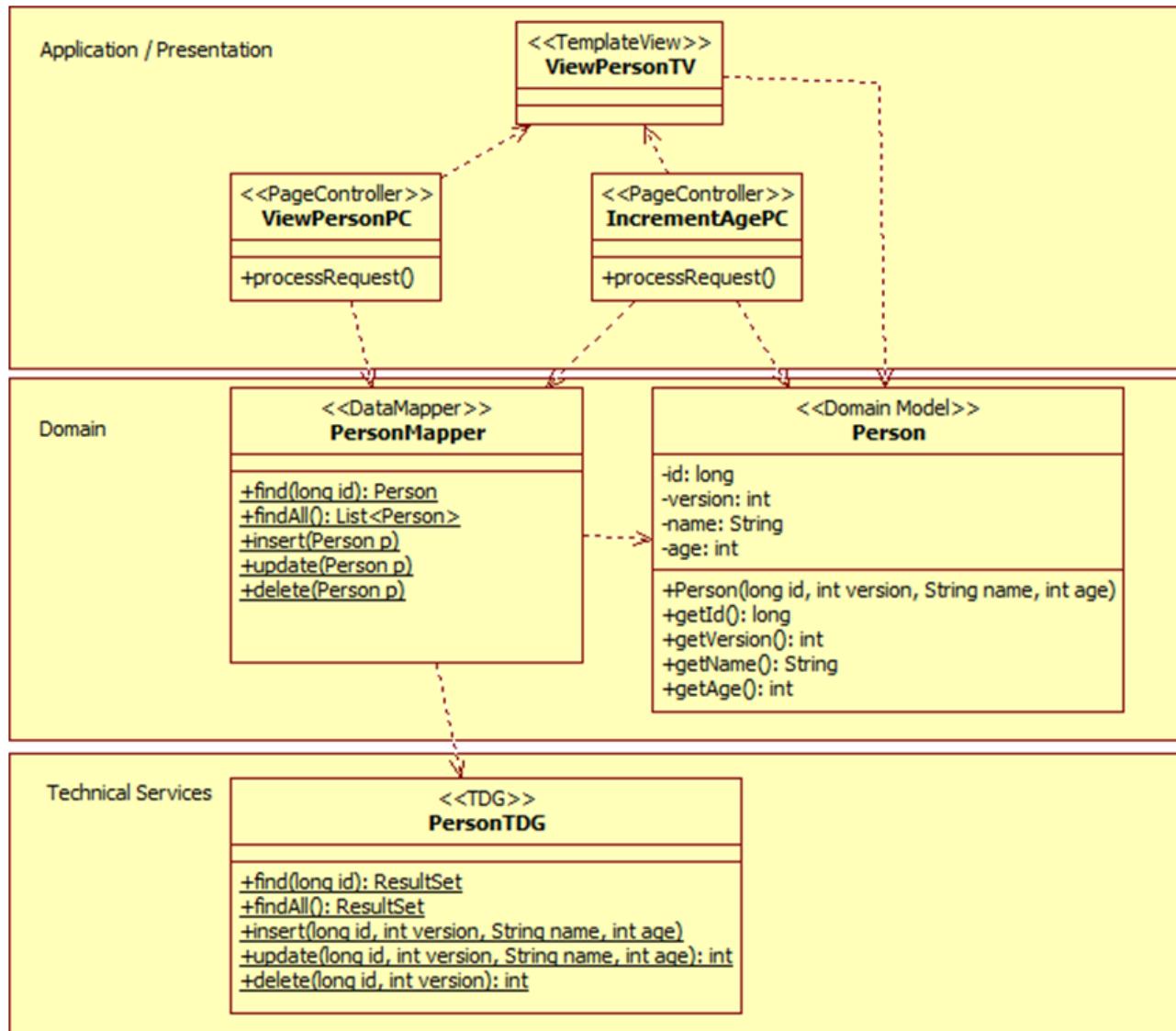
TS + RDG + TV



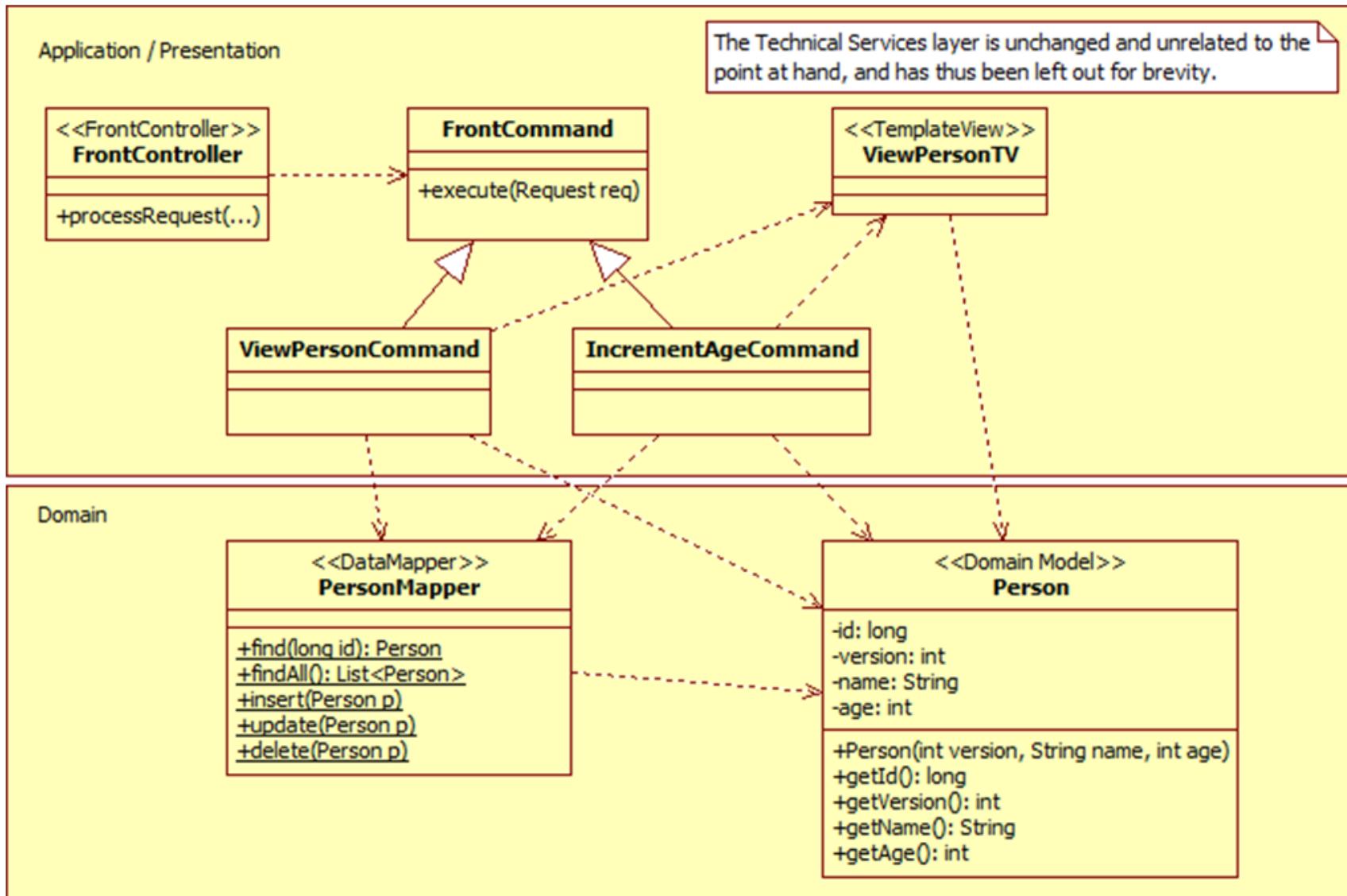
TS+RDG+TV +PC



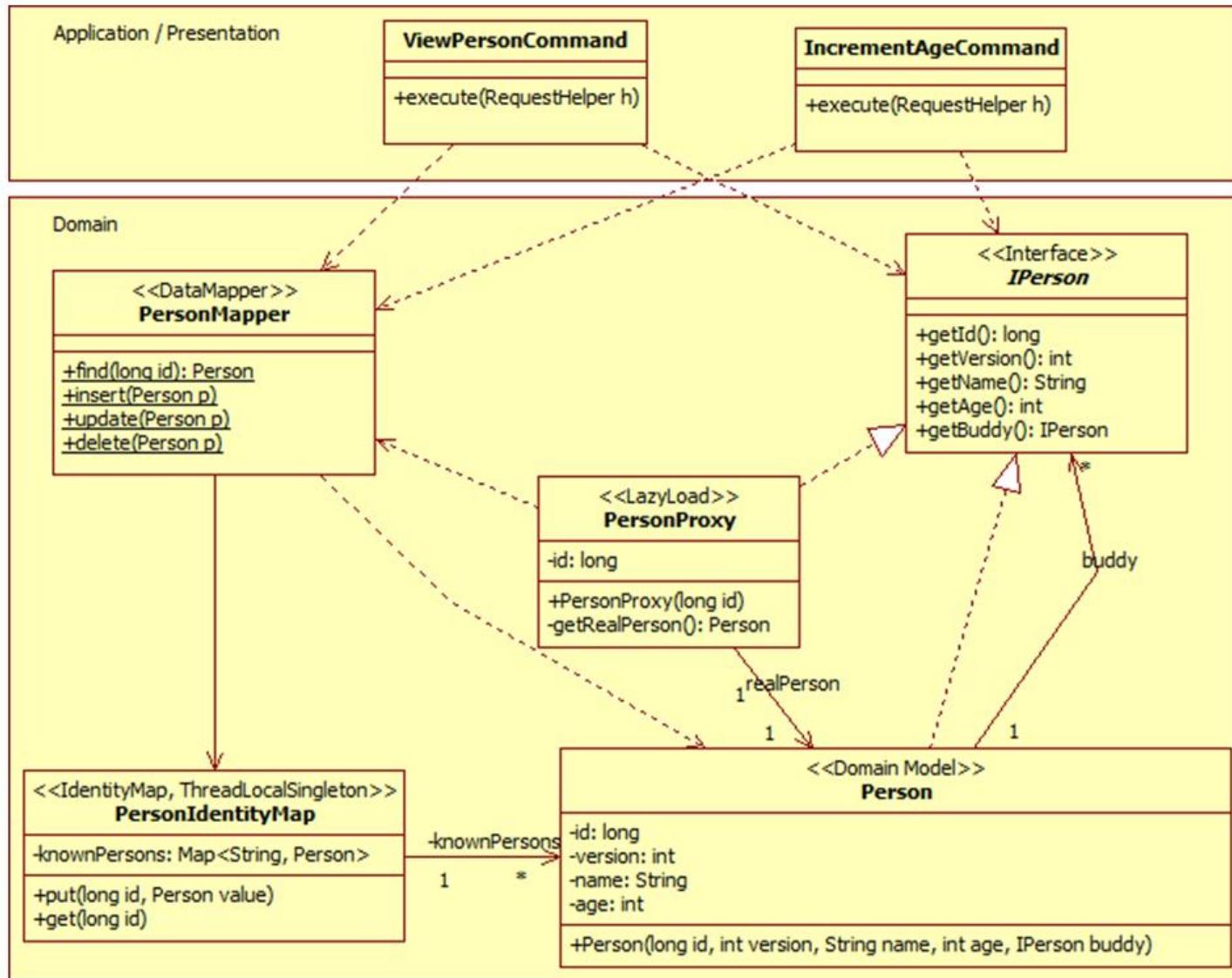
DM+TDG+DMapper+TV+PC



...or DM+...+FC



...+ Lazy load + IM



The Web

§2.1 100,000 feet
 • Client-server (vs. P2P)

§2.2 50,000 feet
 • HTTP & URIs

§2.3 10,000 feet
 • XHTML & CSS

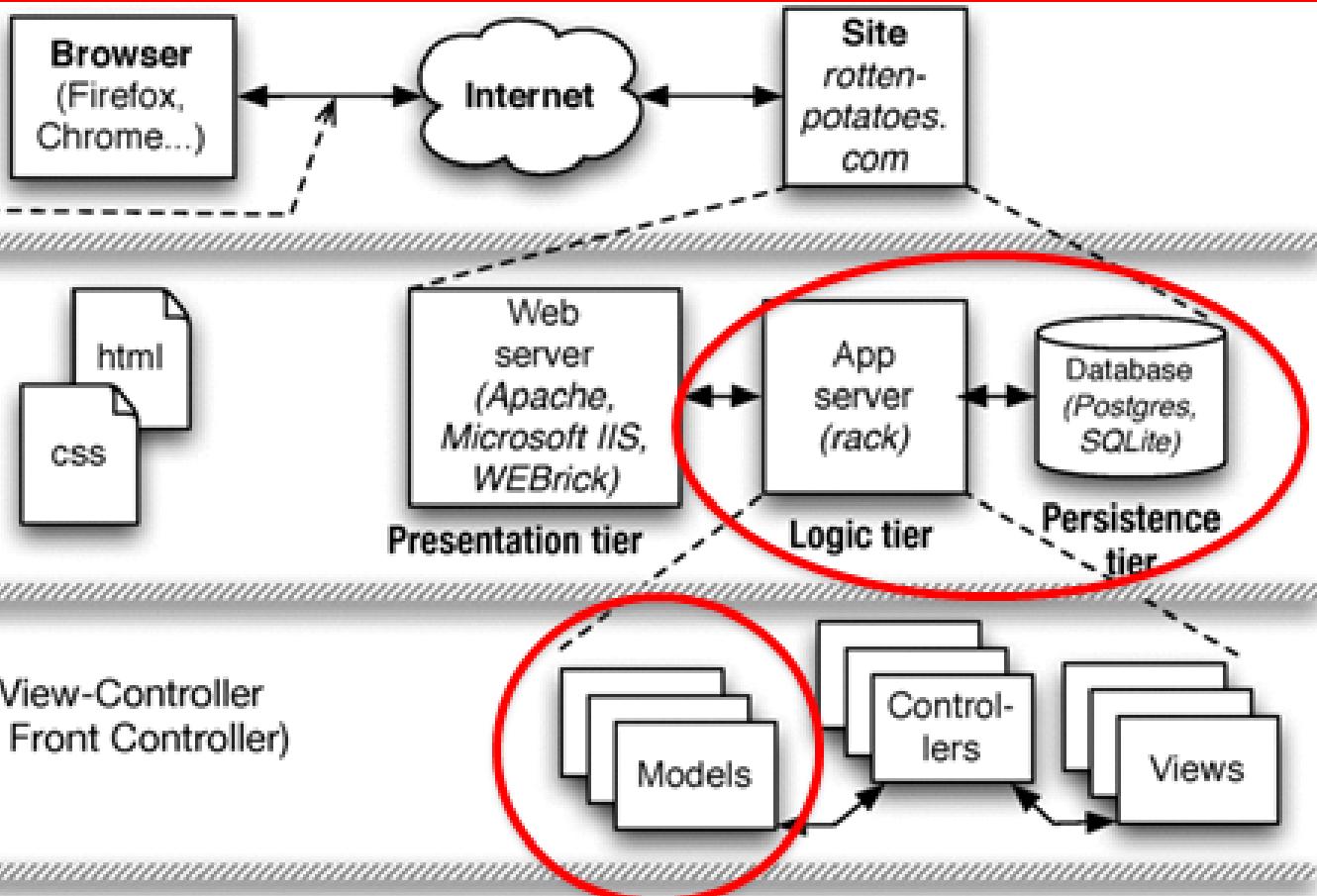
§2.4 5,000 feet
 • 3-tier architecture
 • Horizontal scaling

§2.5 1,000 feet—Model-View-Controller
 (vs. Page Controller, Front Controller)

§2.6 500 feet: Active Record models (vs. Data Mapper)

§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)

§2.8 500 feet: Template View (vs. Transform View)



- Active Record
- REST
- Data Mapper

- Template View
- Transform View

Design Patterns

- Design patterns represent **solutions** to **problems** that arise when developing software within a particular **context**
 - Patterns = **Problem/Solution** pair in **Context**
- Capture static and dynamic structure and collaboration among key participants in software designs
 - key participant – major abstraction that occur in a design problem
 - useful for articulating the **how** and **why** to solve *non-functional forces*.
- Facilitate reuse of successful software architectures and design
 - i.e. the “*design of masters*”

Classification of Design Patterns

- Creational Patterns
 - deal with initializing and configuring classes and objects
 - *how am I going to create my objects?*
- Structural Patterns
 - deal with decoupling the interface and implementation of classes and objects
 - *how classes and objects are composed to build larger structures*
- Behavioral Patterns
 - deal with dynamic interactions among societies of classes and objects
 - *how to manage complex control flows (communications)*

How to select a DP

- Consider how design patterns solve design problems.
- Scan Intent sections.
- Study how patterns interrelate.
- Study patterns of like purpose.
- Examine a cause of redesign.
- Consider what should be variable in your design.

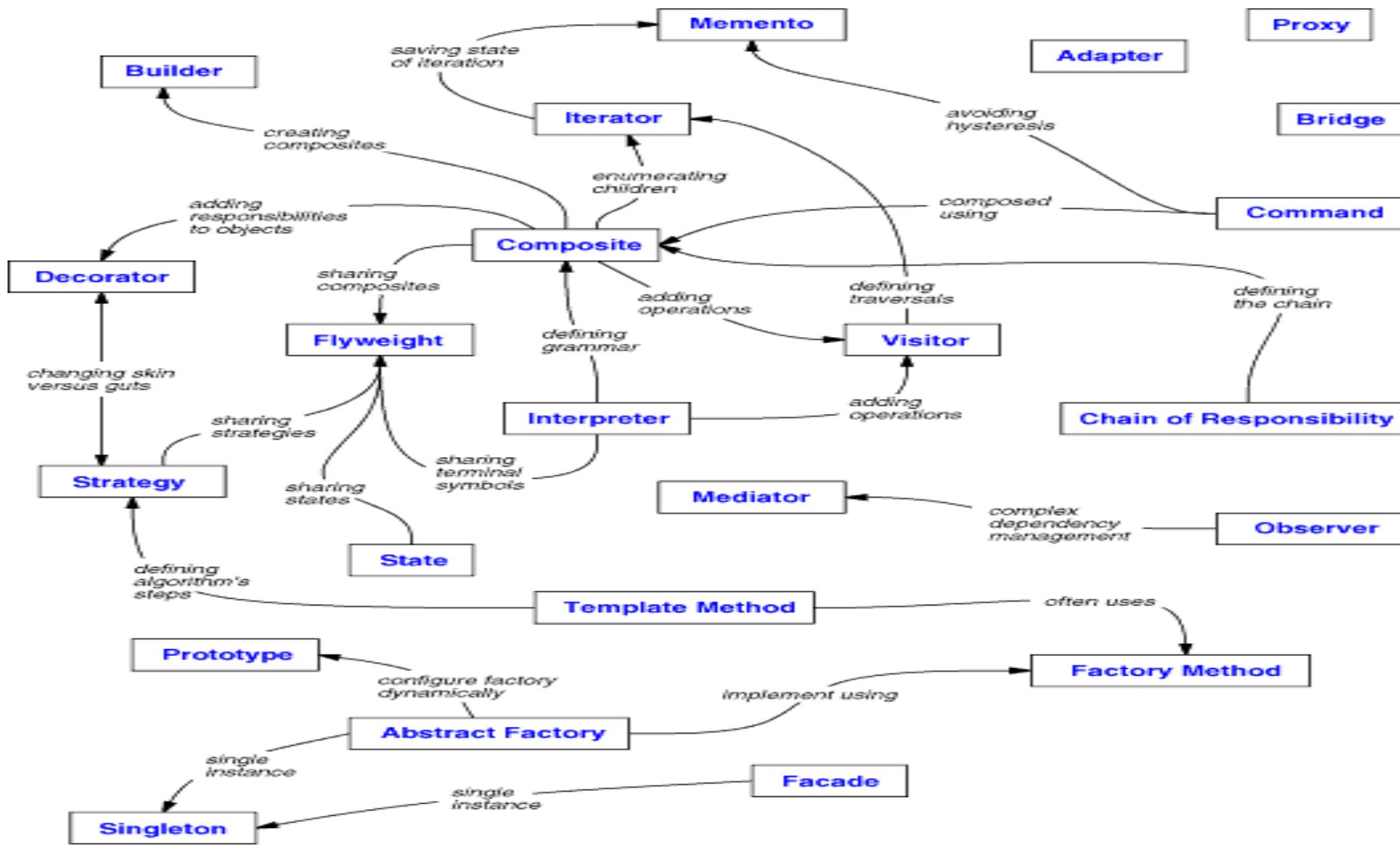
Drawbacks of Design Patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from patterns overload
- Integrating patterns into a software development process is a human-intensive activity

Design Pattern Space

		Purpose		
Scope	Class	Creational	Structural	Behavioral
		Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Singleton Prototype	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Command Chain of Responsibility Strategy Visitor Iterator Mediator Memento Observer State

DP Relationships



GoF Design Principle no. 1

Program to an interface, not an implementation

- Use interfaces to define common interfaces
 - and/or abstract classes
- Declare variables to be instances of the abstract class
 - not instances of particular concrete classes
- Use *Creational patterns*
 - to associate interfaces with implementations

Benefits

Greatly reduces the implementation dependencies

Client objects remain unaware of the classes that **implement** the objects they use.

Clients know only about the abstract classes (or interfaces) that define the interface.

Class Inheritance vs. Composition

- Mechanisms of reuse
 - White-box vs. Black-box
- **Class Inheritance**
 - easy to use; easy to modify
 - implementation being reused;
 - language-supported
 - static bound \Rightarrow can't change at run-time;
 - mixture of physical data representation \Rightarrow breaks information hiding
 - change in parent \Rightarrow change in subclass
- **Object Composition**
 - objects are accessed solely through their interfaces
 - no break of information hiding
 - any object can be replaced by another at runtime
 - as long as they are the same type

Design Principle No.2

- ***Favor composition over class inheritance!***
- Keeps classes focused on one task
- Inheritance and Composition Work Together!
 - ideally no need to create new components to achieve reuse
 - this is rarely the case!
 - reuse by inheritance makes it easier to make new components modifying old components
- Tendency to overuse inheritance as code-reuse technique
- Designs – more reusable by depending more on object composition

Creational DP

- Abstract instantiation process
- System independent of how objects are created, composed, and represented
- **Class** creational patterns use inheritance to vary class instantiation
- **Object** creational patterns delegate instantiation to another object
- Focus on defining small behaviors and combining into more complex ones

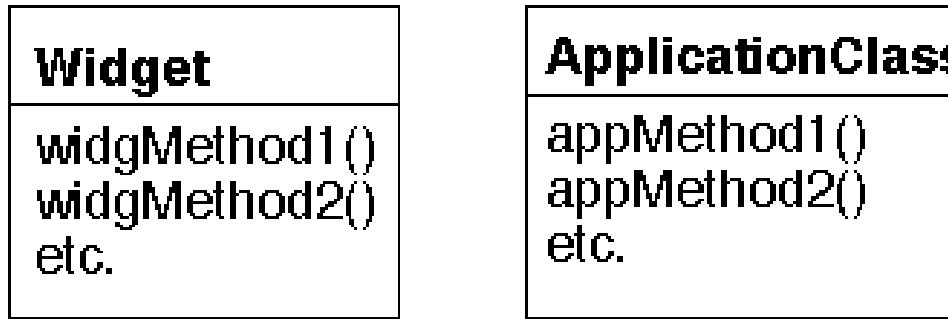
Recurring Themes

- Encapsulate knowledge about which concrete classes the system uses
- Hide how instances of these classes are created and put together

Advantages

- Flexibility in what gets created
 - Who created it
 - How it was created and *when*
- Configure system with “product” objects that vary in structure and functionality
- Configuration can be static or dynamic
- Create standard or uniform structure

Let's start simple...



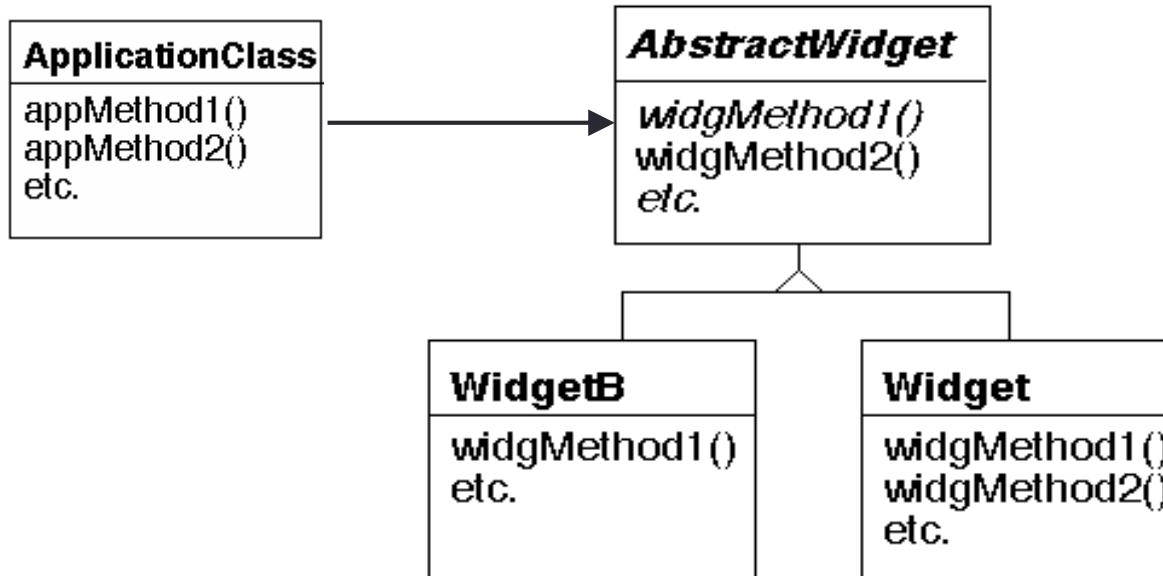
```
class ApplicationClass {  
    public appMethod1() { ..  
        Widget w = new Widget();....  
    }...}
```

- We can modify the internal **Widget** code without modifying **ApplicationClass**

Problems with Changes

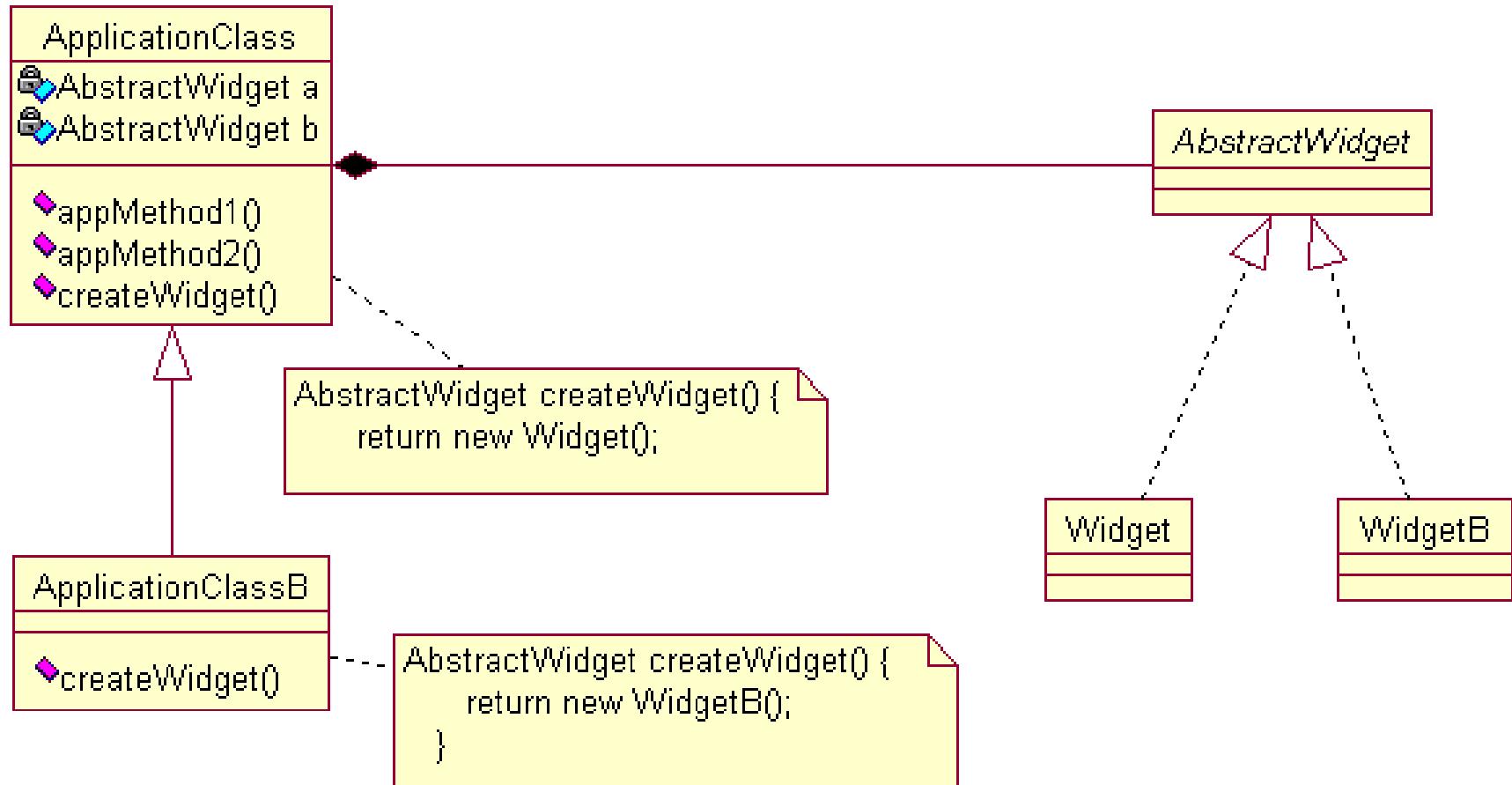
- What happens when we discover a new widget and would like to use in the **ApplicationClass**?
- Multiple coupling between **Widget** and **ApplicationClass**
 - **ApplicationClass** knows the interface of **Widget**
 - **ApplicationClass** explicitly uses the **Widget** type
 - hard to change because Widget is a concrete class
 - **ApplicationClass** explicitly creates new Widgets in many places
 - if we want to use the new **Widget** instead of the initial one, changes are spread all over the code

Apply “Program to an Interface”



- **ApplicationClass** depends now on an (abstract) interface
- But we still have hard coded which widget to create

Use a Factory Method

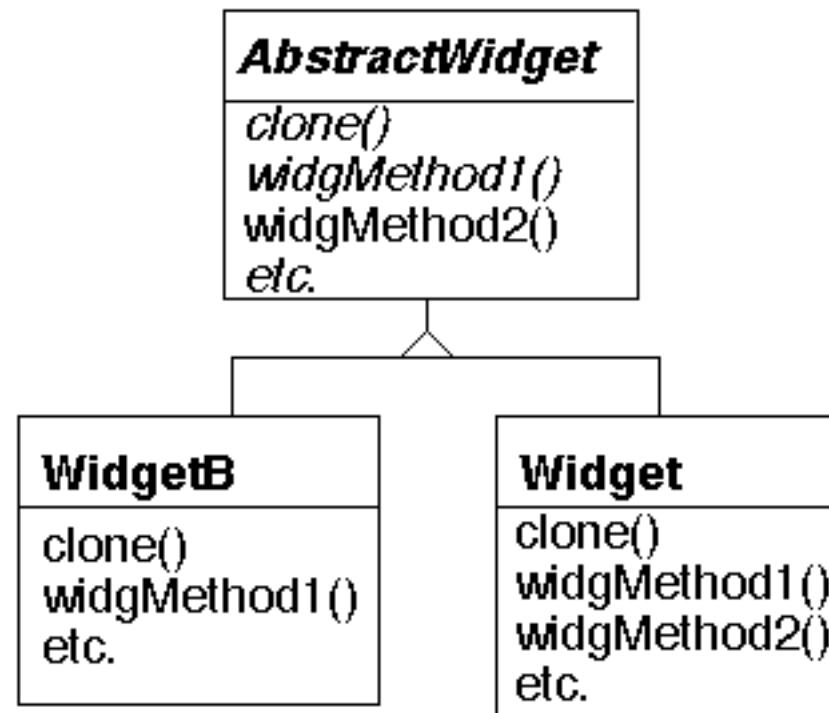


Evaluation of Factory Method Solution

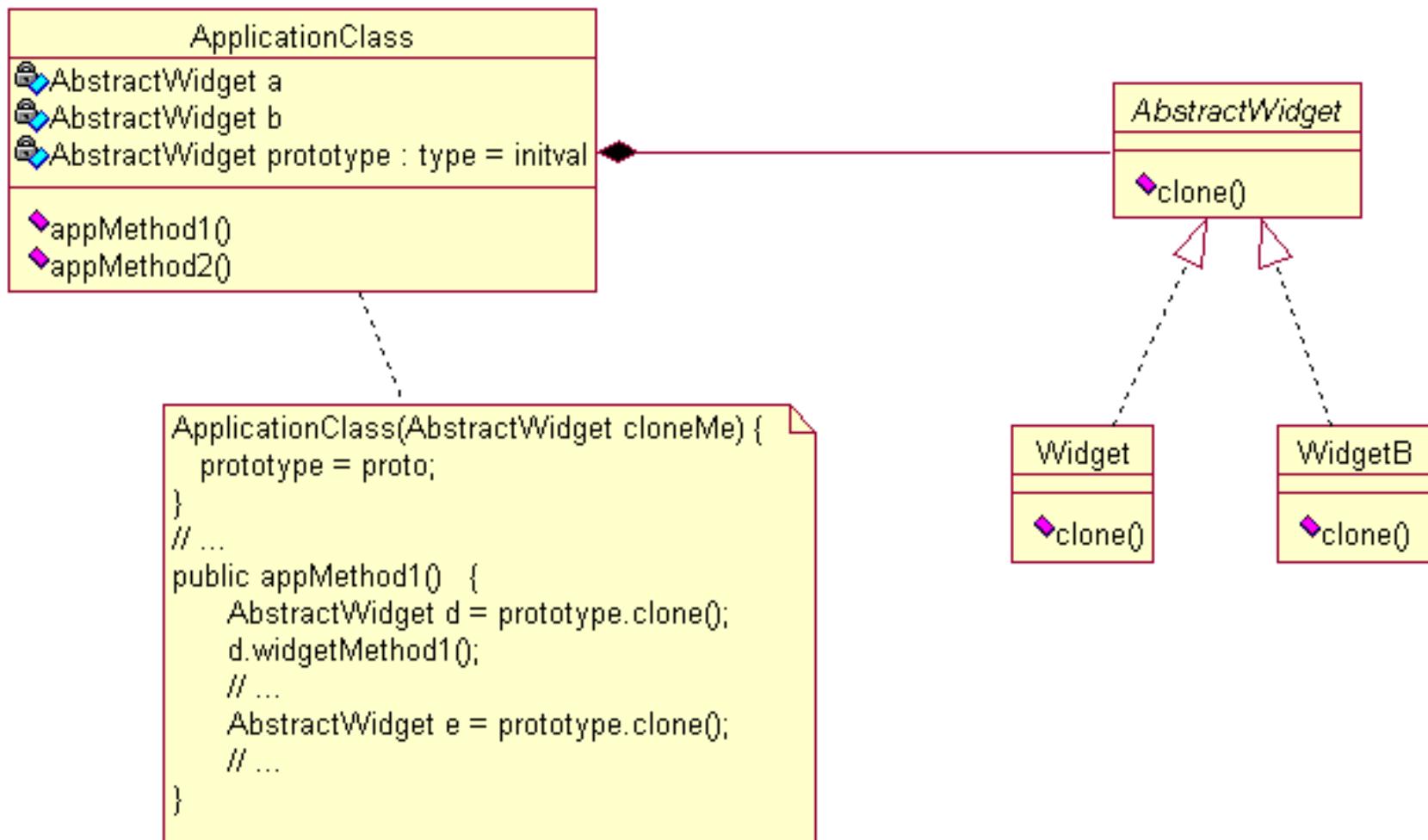
- Explicit creation of **Widget** objects is not anymore dispersed
 - easier to change
- Functional methods in **ApplicationClass** are decoupled from various concrete implementations of widgets
- Avoid ugly code duplication in **ApplicationClassB**
 - subclasses reuse the functional methods, just implementing the concrete *Factory Method* needed
- Disadvantages
 - create a subclass only to override the factory-method
 - can't change the **Widget** at run-time

Solution 2: Clone a Prototype

- Provide the **Widgets** with a clone method
 - make a copy of an existing Widget object



Using the Clone



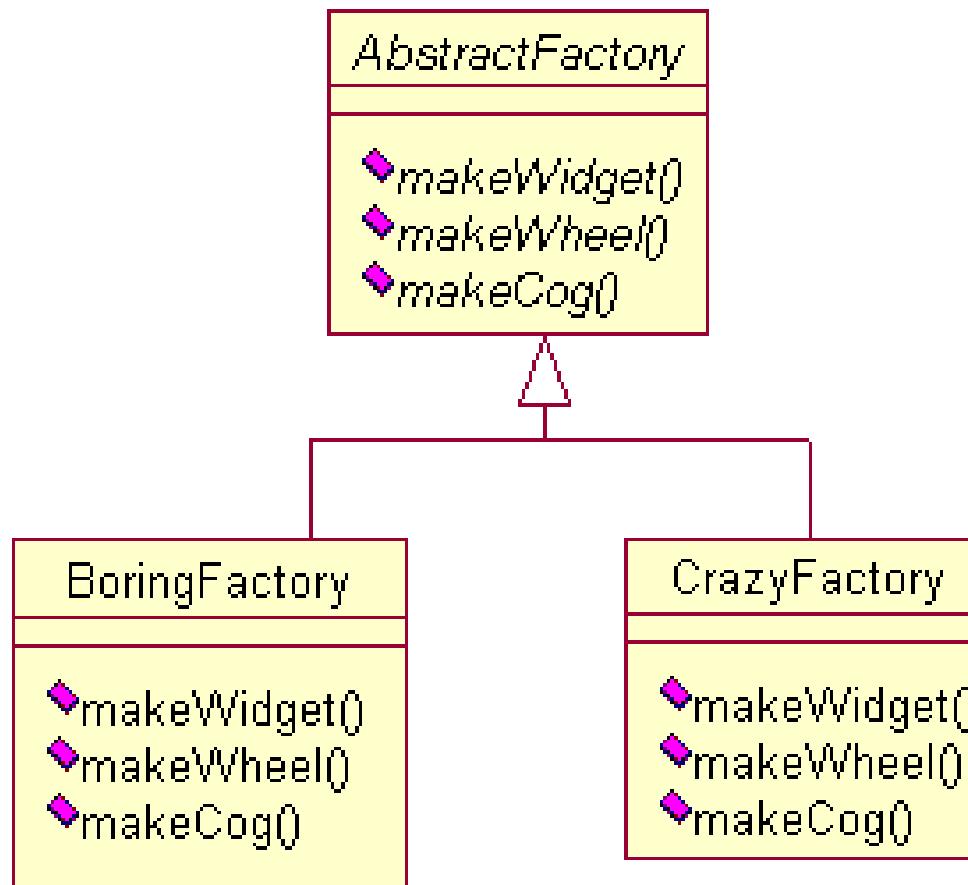
Advantages

- Classes to instantiate may be specified dynamically
 - client can install and remove prototypes at run-time
- We avoided subclassing of **ApplicationClass**
 - Remember: *Favor Composition over Inheritance!*
- Totally hides concrete product classes from clients
 - Reduces implementation dependencies

More Changes

- What if **ApplicationClass** uses other "products" too...
 - e.g. Wheels, etc.
- Each one of these stays for an object family
 - i.e. all of these have subclasses
- Assume that there are restrictions on what type of Widget can be used with which type of Wheel or Cog
- Factory Methods or Prototypes can handle each type of product but it gets hard to insure the wrong types of items are not used together

Solution: Create an Abstract Factory



Next Time

- Design Patterns Continued

SOFTWARE DESIGN

Presentation patterns and Midterm review

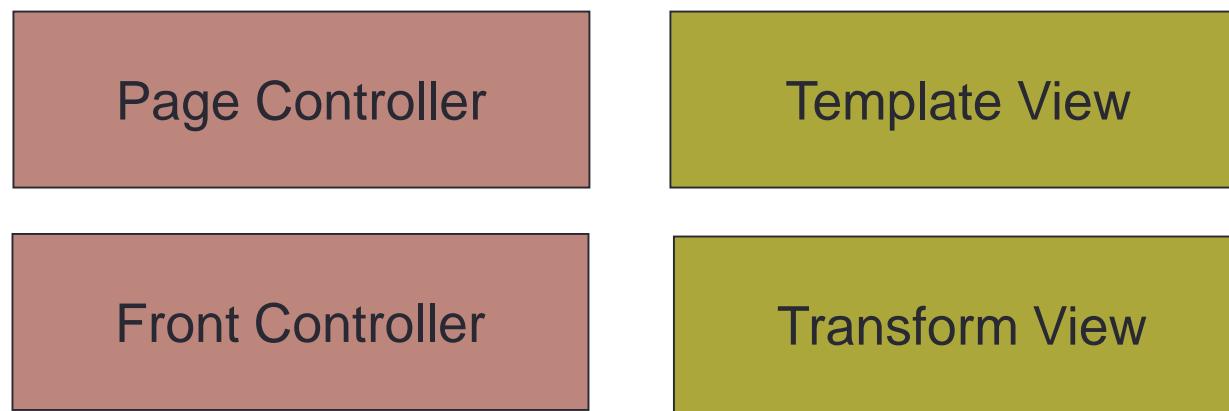
Content

- Presentation patterns
- MIDTERM REVIEW (April 11)

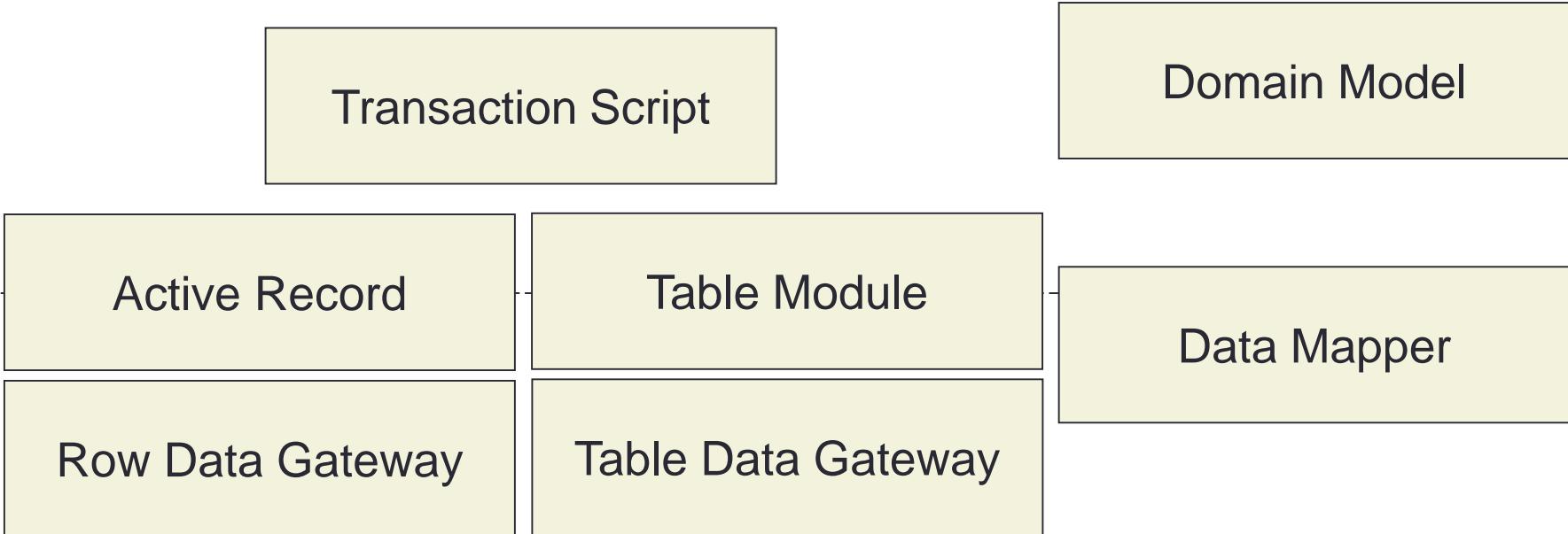
References

- Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]
- Microsoft Application Architecture Guide, 2009 [MAAG]
- SaaS Course Stanford
- Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.
- Univ. of Timisoara Course Materials
- Stuart Thiel, Enterprise Application Design Patterns: Improved and Applied, MSc Thesis, Concordia University Montreal, Quebec, Canada [Thiel]
- Ólafur Andri Ragnarsson, Presentation Layer Design, 2014.

Presentation



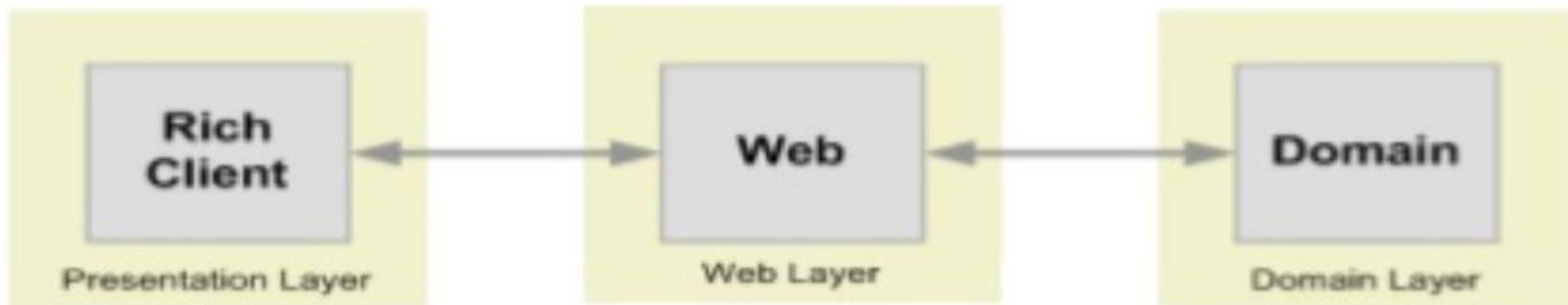
Domain



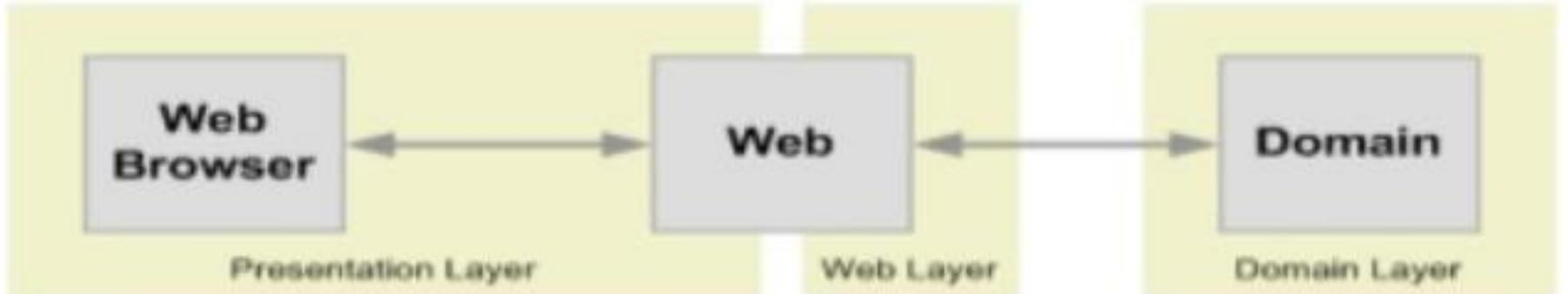
Data Source

Presentation and Web layer

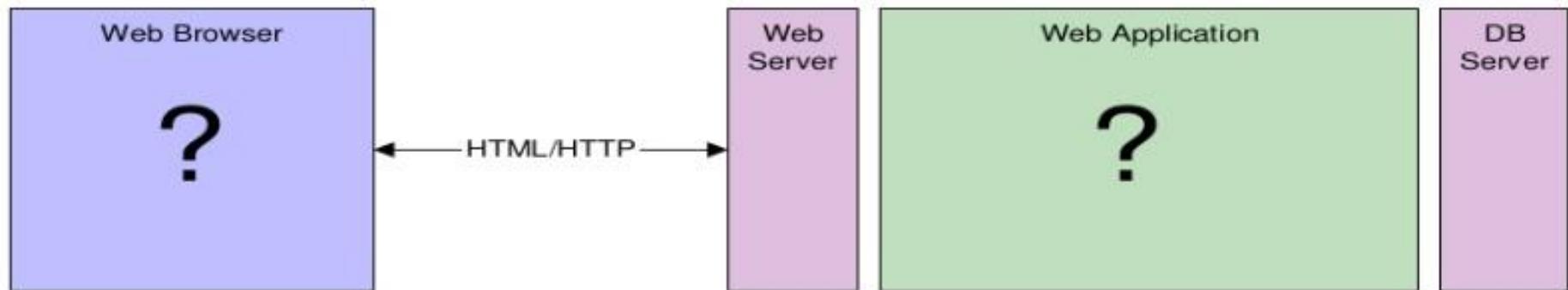
- Desktop and Embedded applications



- Web Browser



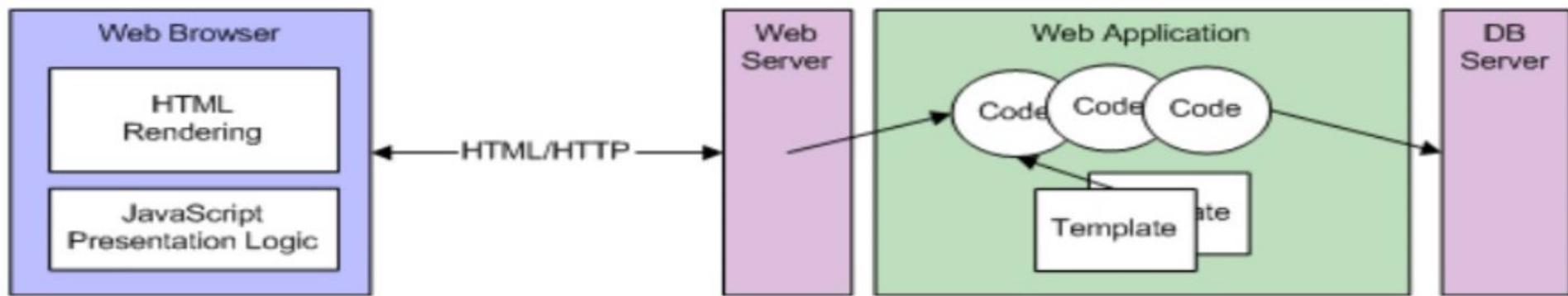
Web Layer Design



- Two approaches
 - Script based – Code using HTML
 - Server Pages based – HTML page with code

Script based

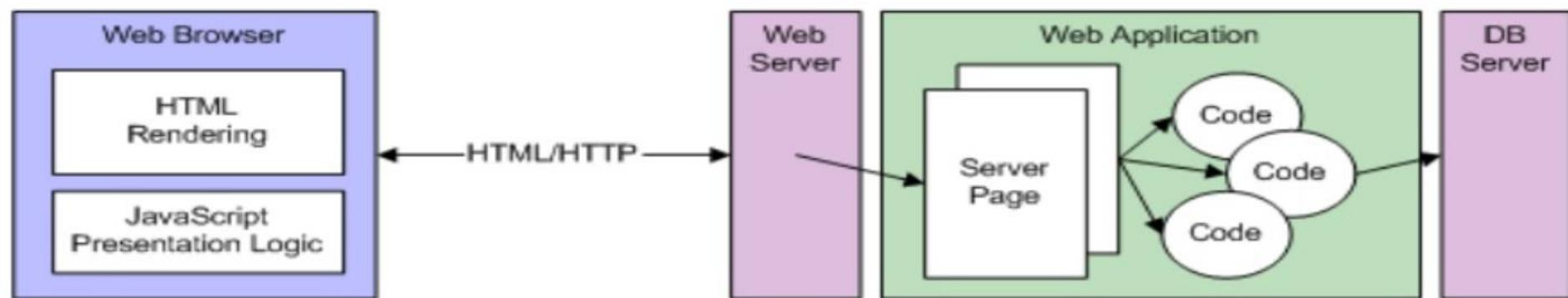
- Useful when the logic and flow are important
 - Request is not easily mapped on a single page



- Examples: CGI, ISAPI, Java Servlets

Server Page Based

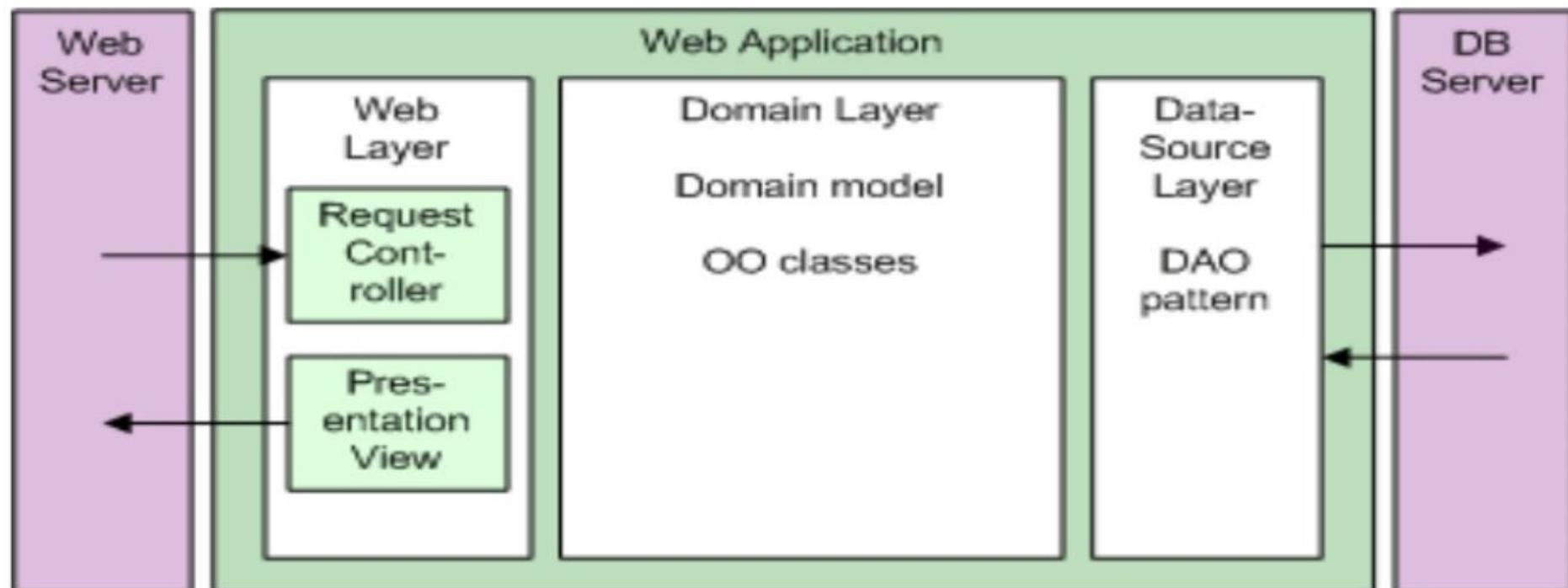
- Useful when there are lots of pages
 - Flow is not as important
 - Each request can be easily mapped to a page



- Examples: PHP, JSP, ASP

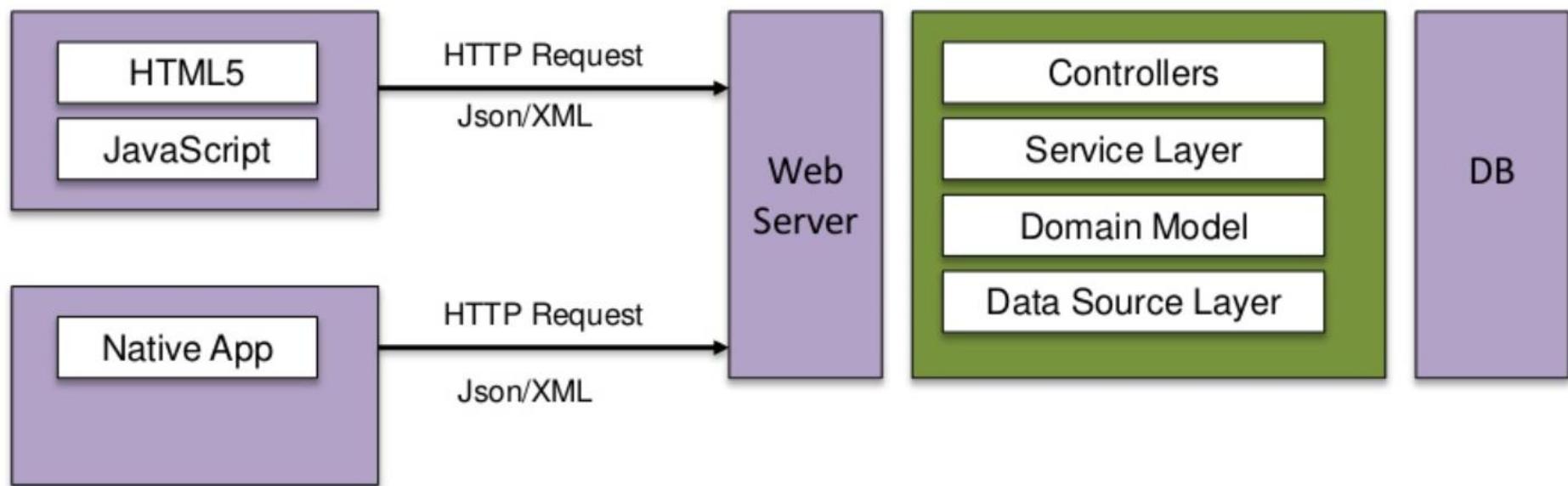
Web layer

- Must handle the request and the response



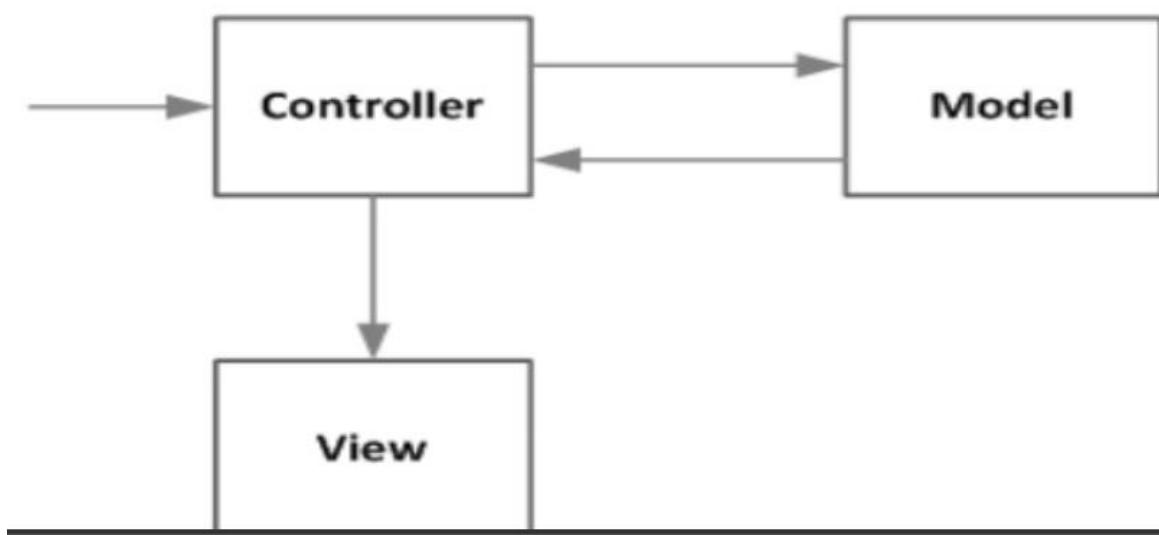
API based

- User Interface is HTML 5 or Native App
- Server side API
- Information format is XML or JSON

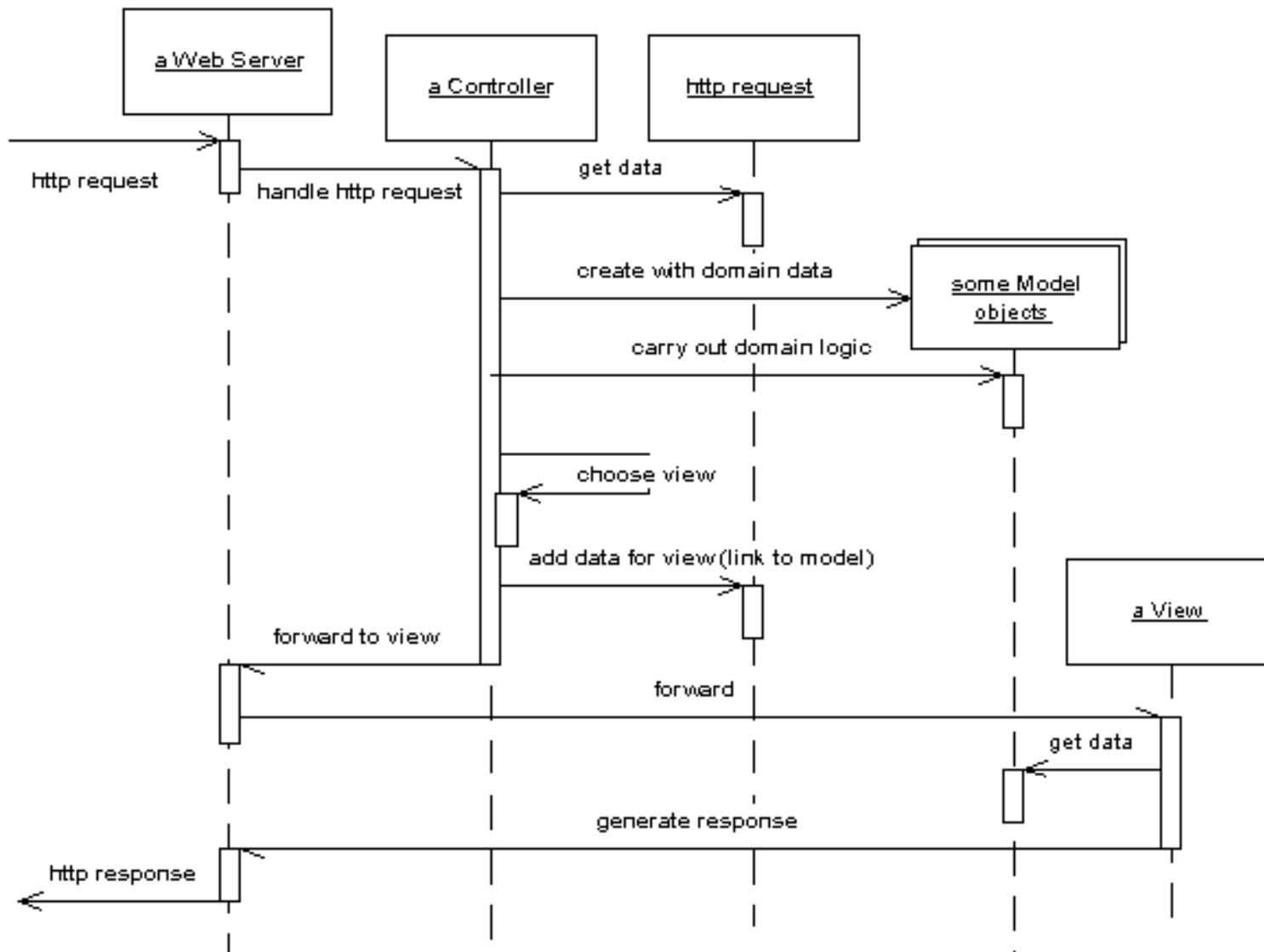


MVC in Web applications

- Input Controller
 - Takes the request
 - Examines the input parameters
 - Calls the Model
 - Handles the response
 - Sends the control to the View for rendering

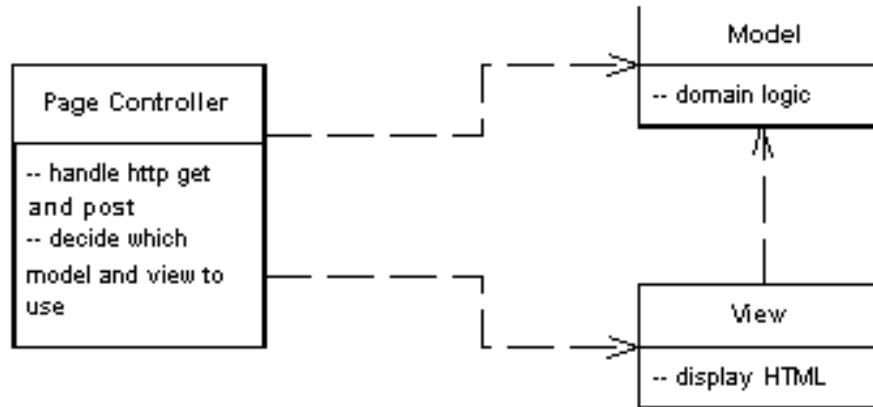


MVC in action

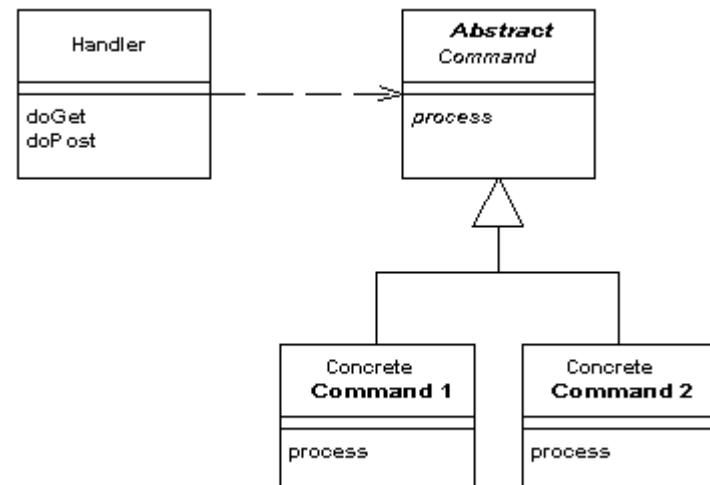


Controllers

- One for each page – Page controller



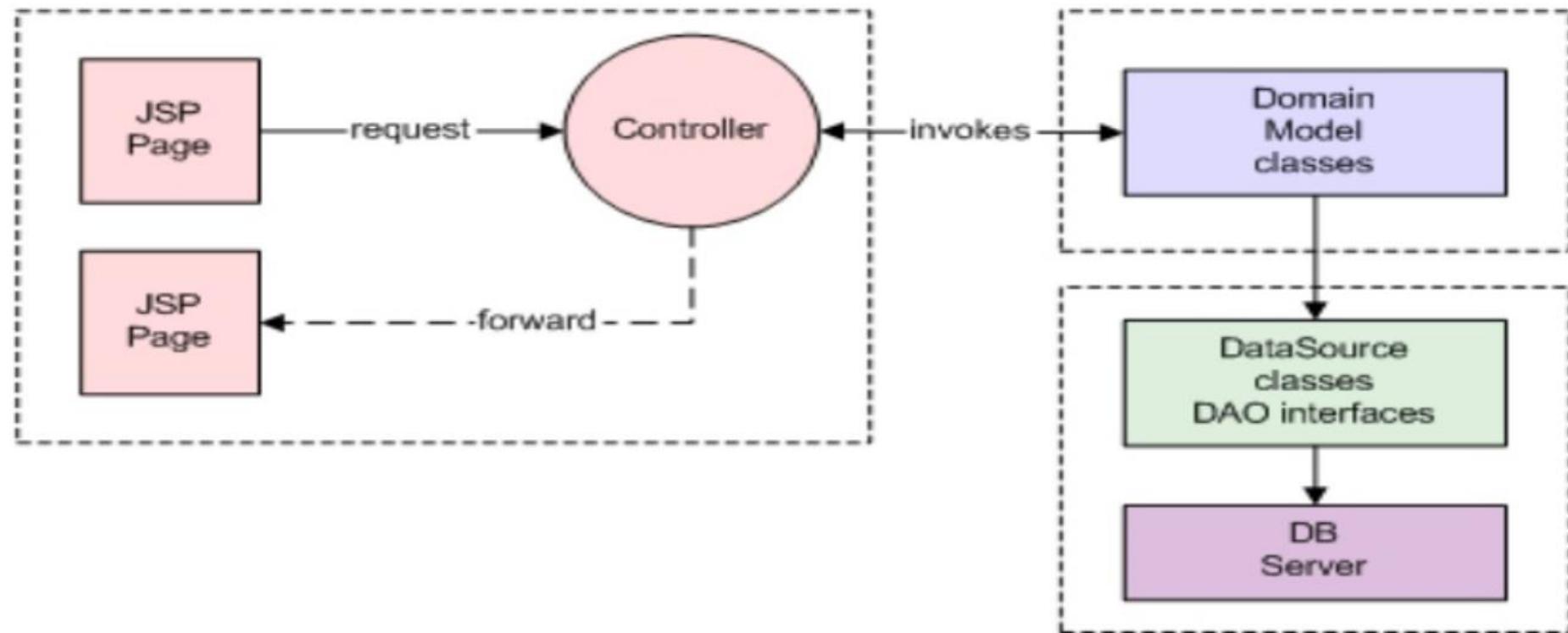
- One per application – Front controller



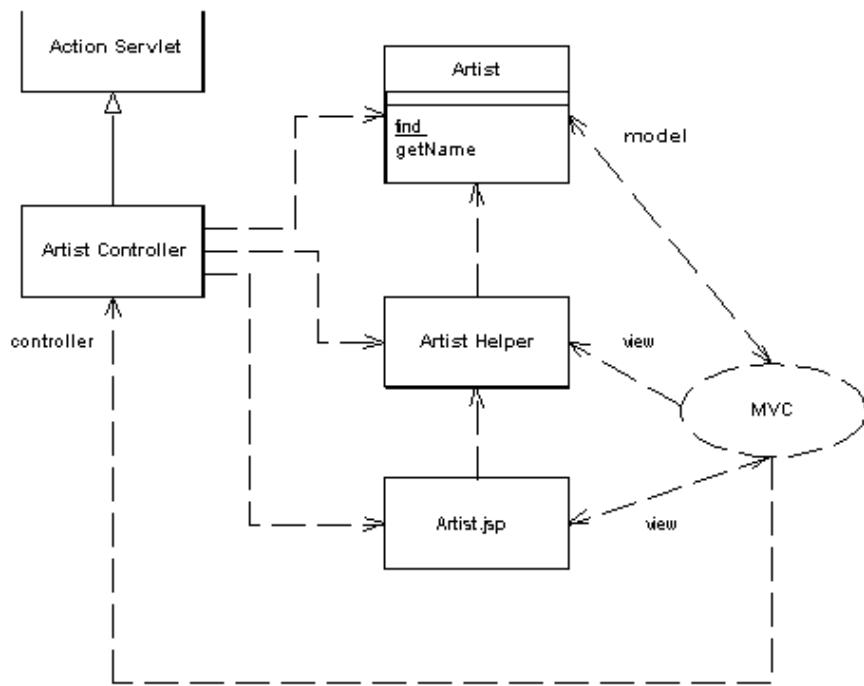
Page Controller

- As Script
 - Servlet or CGI program
 - Applications that need logic and data
- As Server Page
 - ASP, PHP, JSP
 - Use helpers to get data from the model
 - Logic is simple to none
- Basic responsibilities
 - **Decode the URL** and extract all data for the action.
 - **Create and invoke any model objects** to process the data. All relevant data from the HTML request should be passed to the model so that the model objects don't need any connection to the HTML request.
 - **Determine which view** should display the result page and forward the model information to it.

Page Controller



Servlet controller and a JSP view (Java)



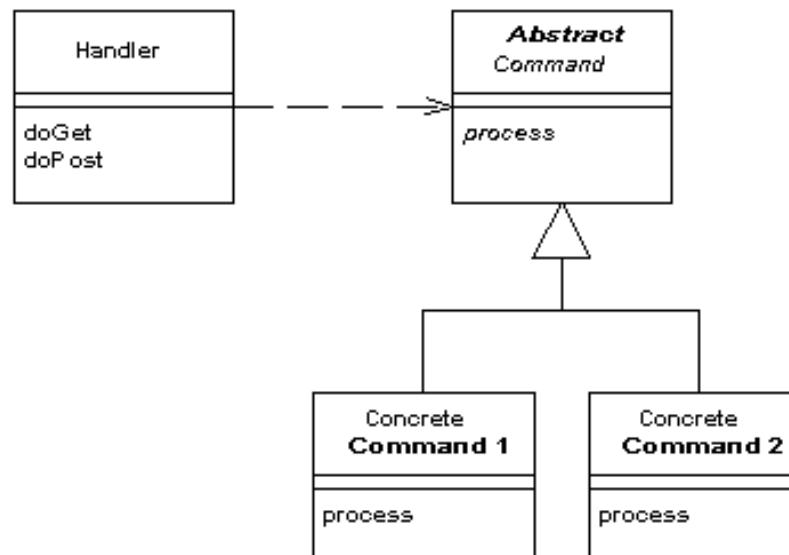
```
class ArtistController...
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        Artist artist = Artist.findNamed(request.getParameter("name"));
        if (artist == null)
            forward("/MissingArtistError.jsp", request, response);
        else {
            request.setAttribute("helper", new ArtistHelper(artist));
            forward("/artist.jsp", request, response);
        }
    }
```

```
<servlet>
<servlet-name>artist</servlet-name>
<servlet-
class>actionController.ArtistController
</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>artist</servlet-name>
<url-pattern>/artist</url-pattern>
</servlet-mapping>
```

Front controller

- One controller handles all requests
- usually structured in two parts:
 - a web handler (rather a class than a server page)
 - a hierarchy of commands (classes)

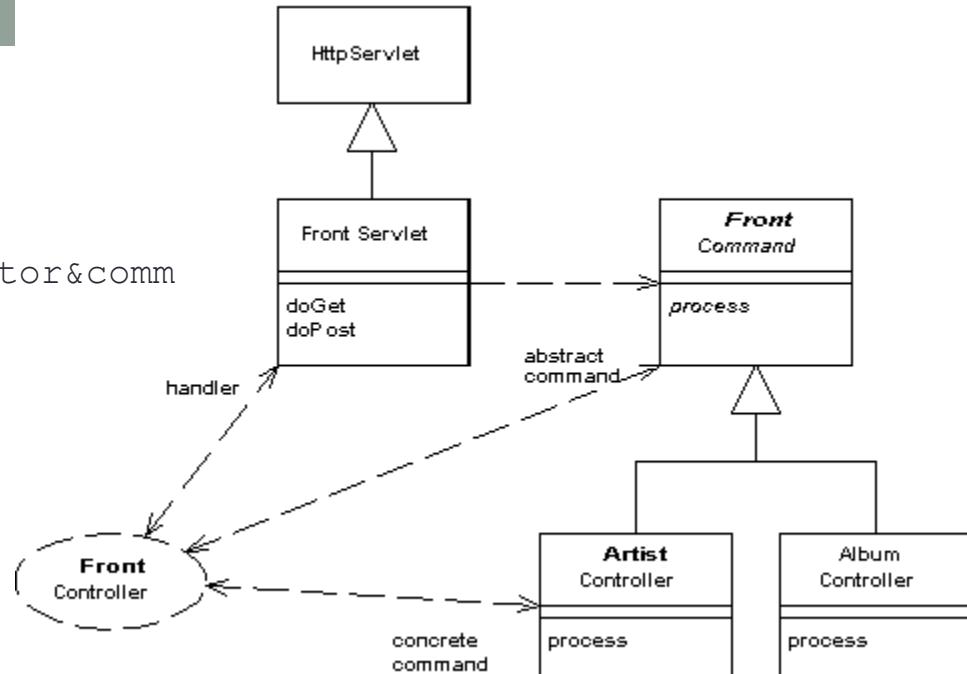


Front controller

- Handler decides what command:
 - **statically**
 - parses the URL and uses conditional logic;
 - advantage of explicit logic,
 - compile time error checking on dispatch,
 - flexibility in URL look-up
 - **dynamically**
 - takes a standard piece of the URL and uses dynamic instantiation to create a command class;
 - allows to add new commands without changing the Web handler;
 - can put the name of the command class into the URL or can use a properties file that binds URLs to command class names.

Example

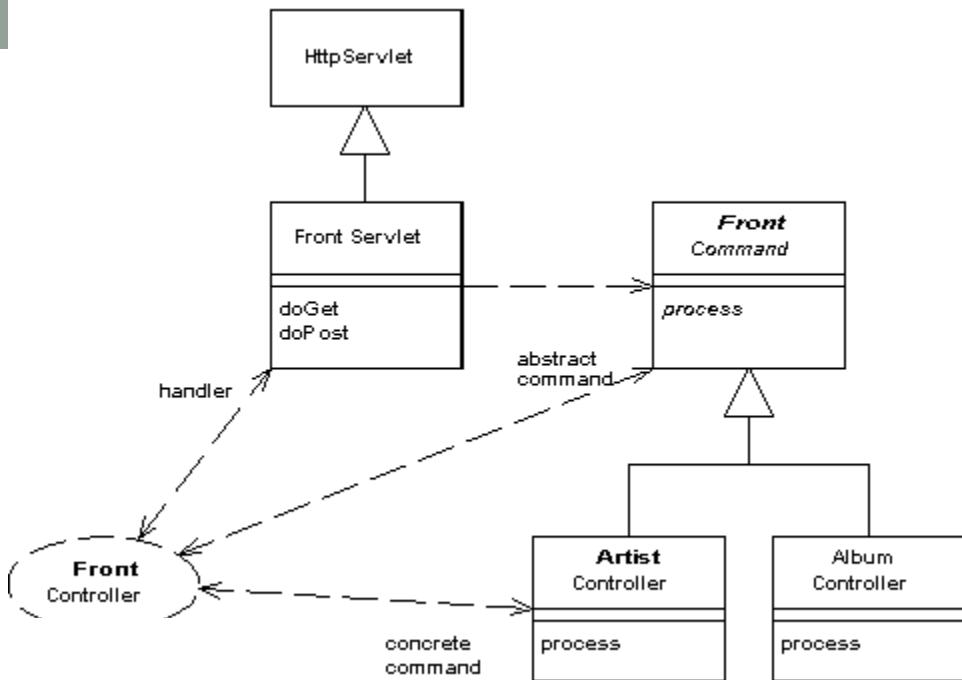
http://localhost:8080/isa/music?name=astor&command=Artist



```
class FrontServlet...
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        FrontCommand command = getCommand(request);
        command.init(getServletContext(), request, response);
        command.process();
    }

    private FrontCommand getCommand(HttpServletRequest request) {
        try {
            return (FrontCommand) getCommandClass(request).newInstance();
        } catch (Exception e) {
            throw new ApplicationException(e);
        }
    }

    private Class getCommandClass(HttpServletRequest request) {
        Class result;
        final String commandClassName =
            "frontController." + (String) request.getParameter("command") + "Command";
        try {
            result = Class.forName(commandClassName);
        } catch (ClassNotFoundException e) {
            result = UnknownCommand.class;
        }
        return result;
    }
```



```

class FrontCommand...
    protected ServletContext context;
    protected HttpServletRequest request;
    protected HttpServletResponse response;

    public void init(ServletContext context,
                      HttpServletRequest request,
                      HttpServletResponse response)
    {
        this.context = context;
        this.request = request;
        this.response = response;
    }

    abstract public void process() throws ServletException, IOException ;

    protected void forward(String target) throws ServletException, IOException
    {
        RequestDispatcher dispatcher = context.getRequestDispatcher(target);
        dispatcher.forward(request, response);
    }
}
  
```

Discussion

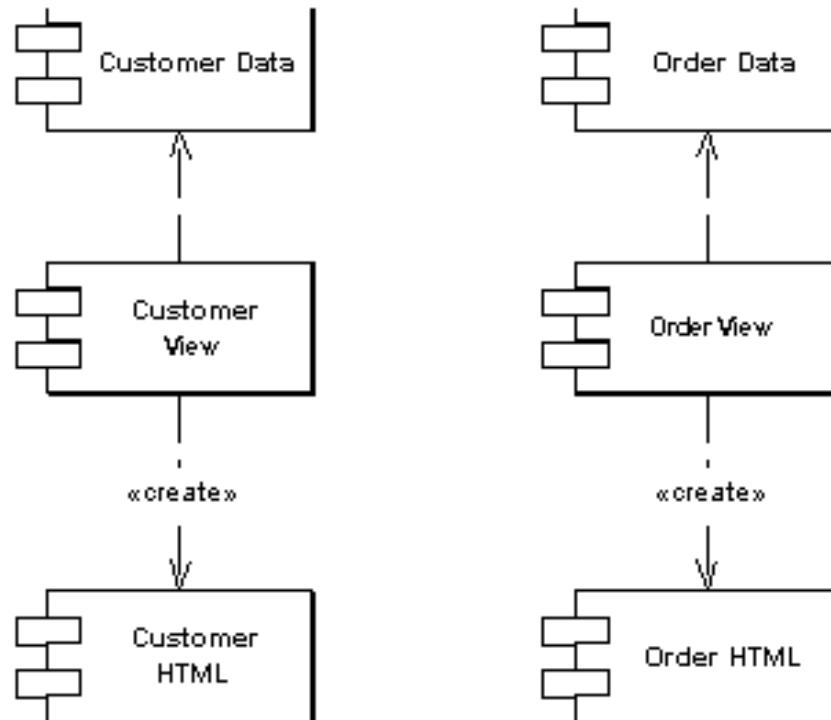
- Only one Front Controller has to be configured into the Web server
- with dynamic commands you can add new commands without changing anything.
- because new command objects are created with each request, its thread safe.
- Re-factor code better in command hierarchy

Discussion

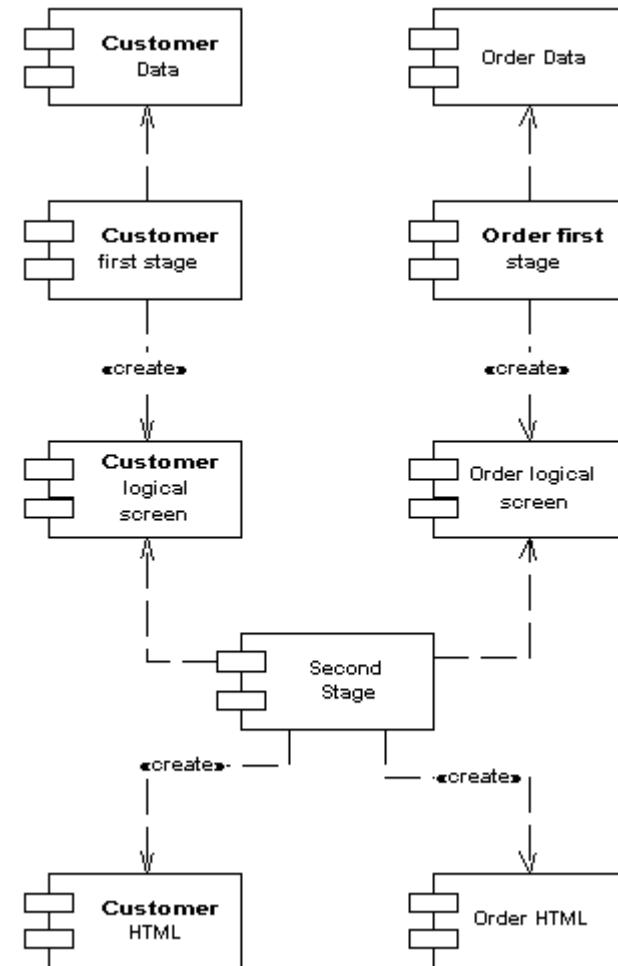
- **Page Controller:**
 - simple controller logic
 - a natural structuring mechanism where particular actions are handled by particular server pages or script classes.
- **Front Controller:**
 - greater complexity;
 - handles duplicated features (i.e. security, internationalization, providing particular views for certain kinds of users) in one place.
 - single point of entry for centralized logic

View

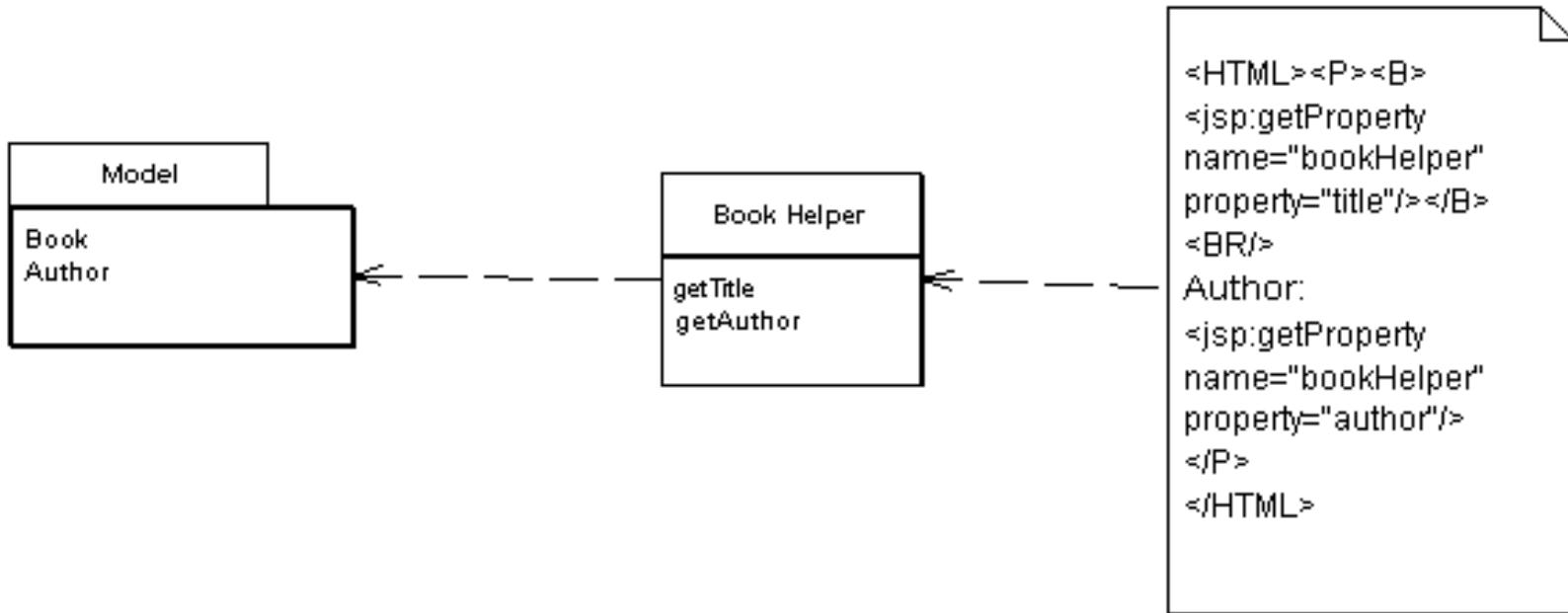
- Single step stage



- Two step stage



Template View



- embed markers into a static HTML page when it's written
- When the page is used to service requests, the markers are replaced by the results of some computation
- **server pages:**
 - ASP, JSP, or PHP.
 - allow to embed arbitrary programming logic, referred to as **scriptlets**, into the page.

Conditional display

```
<IF condition = "$pricedrop > 0.1"> ...show some stuff </IF>
```

Templates become programming languages

⇒ Move the condition to the helper to generate the content

⇒ What if the content should be displayed but in different ways?

- Helper generates the markup
- OR use focused tags:

```
<IF expression = "isHighSelling()"><B></IF><property name =  
"price"/><IF expression = "isHighSelling()"></B></IF>
```

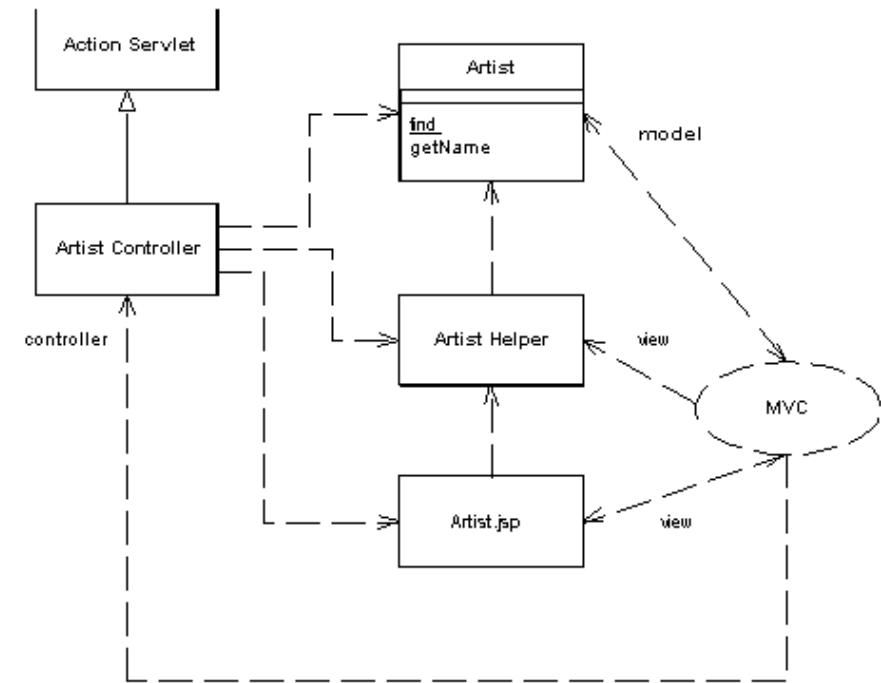
replaced by

```
<highlight condition = "isHighSelling" style =  
"bold"><property name = "price"/></highlight>
```

JSP Template View (see Page Controller)

```
<jsp:useBean id="helper" type="actionController.ArtistHelper" scope="request"/>
```

```
class ArtistHelper...  
    private Artist artist;  
  
    public ArtistHelper(Artist artist) {  
        this.artist = artist;  
    }  
  
    public String getName() {  
        return artist.getName();  
    }
```



```
<B> <%=helper.getName () %></B>
```

```
<B><jsp:getProperty name="helper" property="name"/></B>
```

Show a list of albums for an artist

```
<UL>
<%
    for (Iterator it = helper.getAlbums().iterator(); it.hasNext();) {
        Album album = (Album) it.next();%>
<LI><%=album.getTitle()%></LI>

<%
    }    %>
</UL>

class ArtistHelper...
    public String getAlbumList() {
        StringBuffer result = new StringBuffer();
        result.append("<UL>");
        for (Iterator it = getAlbums().iterator(); it.hasNext();) {
            Album album = (Album) it.next();
            result.append("<LI>");
            result.append(album.getTitle());
            result.append("</LI>");
        }
        result.append("</UL>");
        return result.toString();
    }

    public List getAlbums() {
        return artist.getAlbums();
    }

    <UL><tag:forEach host = "helper" collection = "albums" id = "each">
        <LI><jsp:getProperty name="each" property="title"/></LI>
    </tag:forEach></UL>
```

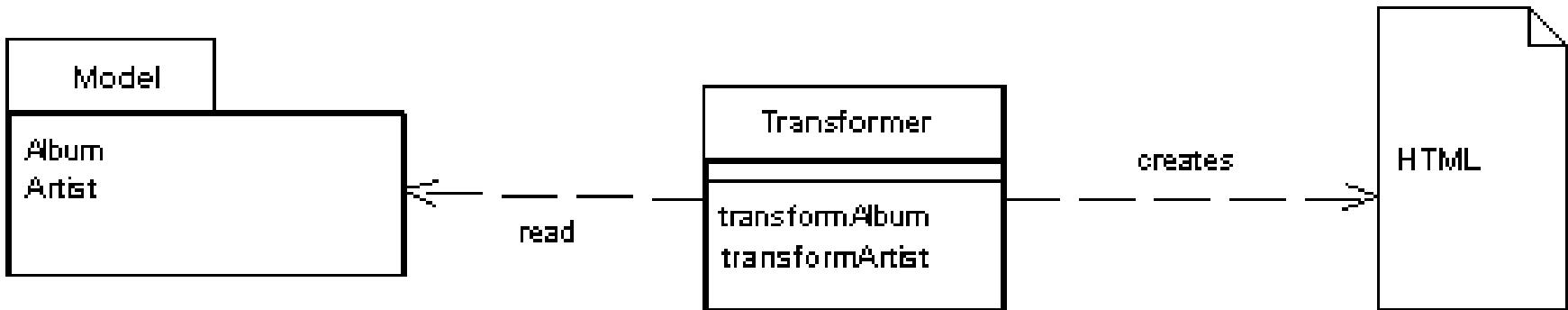


Discussion

- Benefits:
 - Compose the structure of the page based on the template
 - Separate design from code (helper)
- Liabilities
 - common implementations make it too easy to put complicated logic onto the page => hard to maintain
 - Harder to test than Transform View

Transform View

- Input: Model
- Output: HTML



- can be written in any language, yet the dominant choice is XSLT (EXtensible Stylesheet Language Transformation).
- Input: XML
- XML data can be returned as:
 - natural return type
 - output type which can be transformed to XML automatically
 - Data Transfer Object, that can serialize as XML

Translated in code

```
class AlbumCommand...
    public void process() {
        try {
            Album album = Album.findNamed(request.getParameter("name"));
            Assert.notNull(album);
            PrintWriter out = response.getWriter();
            XsltProcessor processor = new SingleStepXsltProcessor("album.xsl");
            out.print(processor.getTransformation(album.toXmlDocument()));
        } catch (Exception e) {
            throw new A
        }
    }

<album>
    <title>Zero Hour</title>
    <artist>Astor Piazzola</artist>
    <trackList>
        <track><title>Tanguedia III</title><time>4:39</time></track>
        <track><title>Milonga del Angel</title><time>6:30</time></track>
        <track><title>Concierto Para Quinteto</title><time>9:00</time></track>
        <track><title>Milonga Loca</title><time>3:05</time></track>
        <track><title>Michelangelo '70</title><time>2:50</time></track>
        <track><title>Contrabajisimo</title><time>10:18</time></track>
        <track><title>Mumuki</title><time>9:32</time></track>
    </trackList>
</album>
```

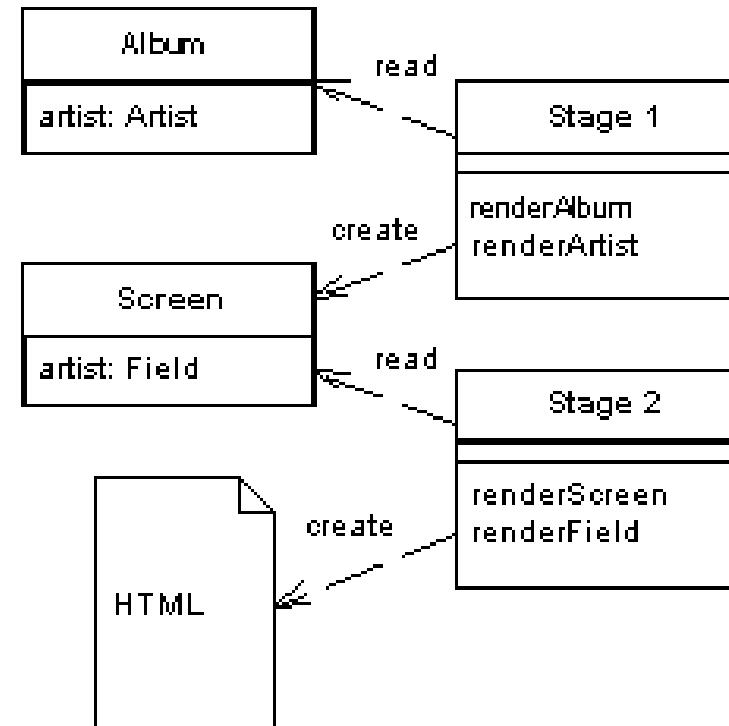
Advantages

- Views are easily testable independent of web servers
- Avoid too much logic in view
- For changing the website look often, one needs to change the transform programs. Using common transforms with XSLT includes, reduce this problem.

Two step view

Multi-page application:

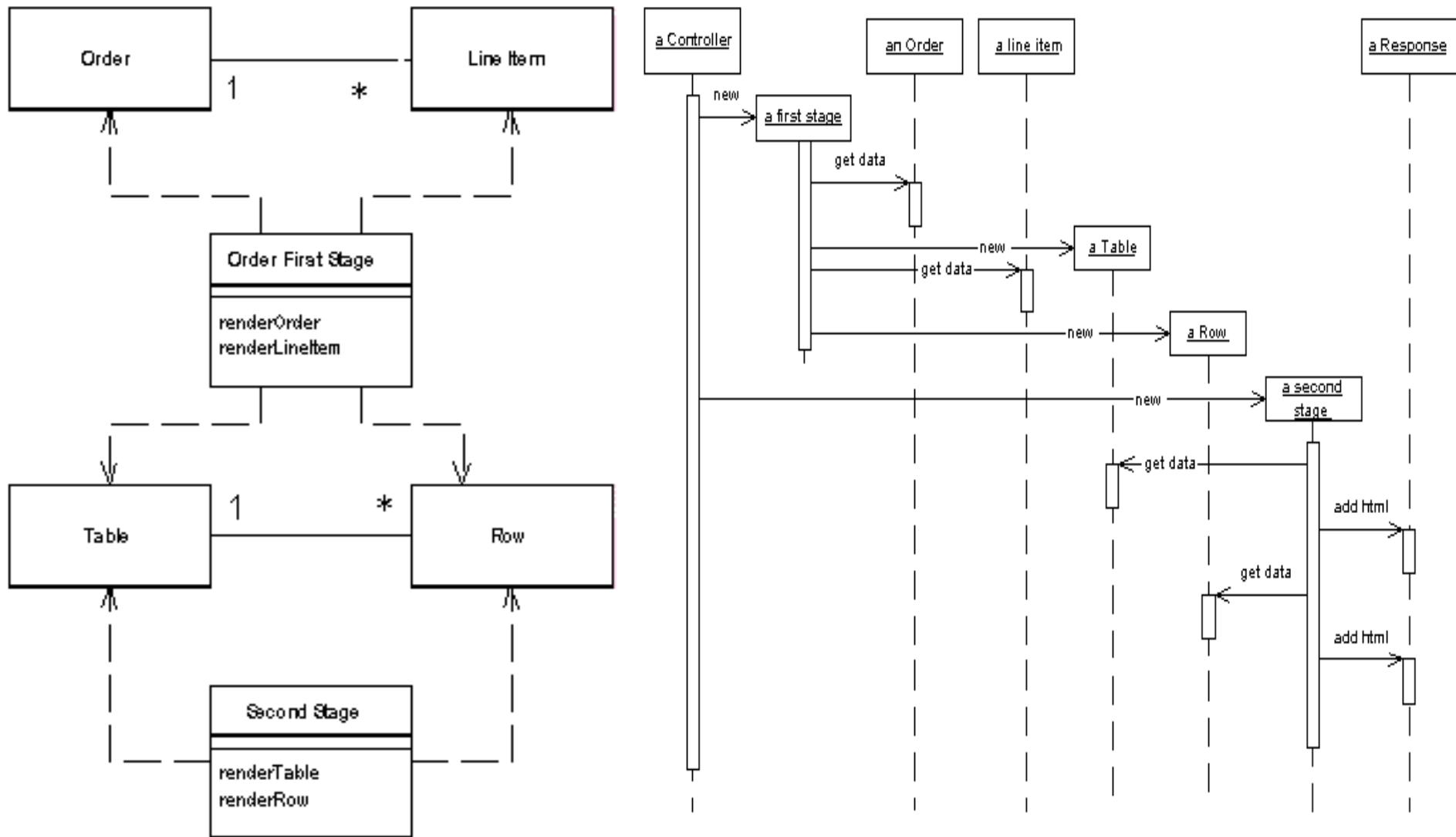
- transforms the model data into a logical presentation without any specific formatting
- converts that logical presentation with the actual formatting needed.



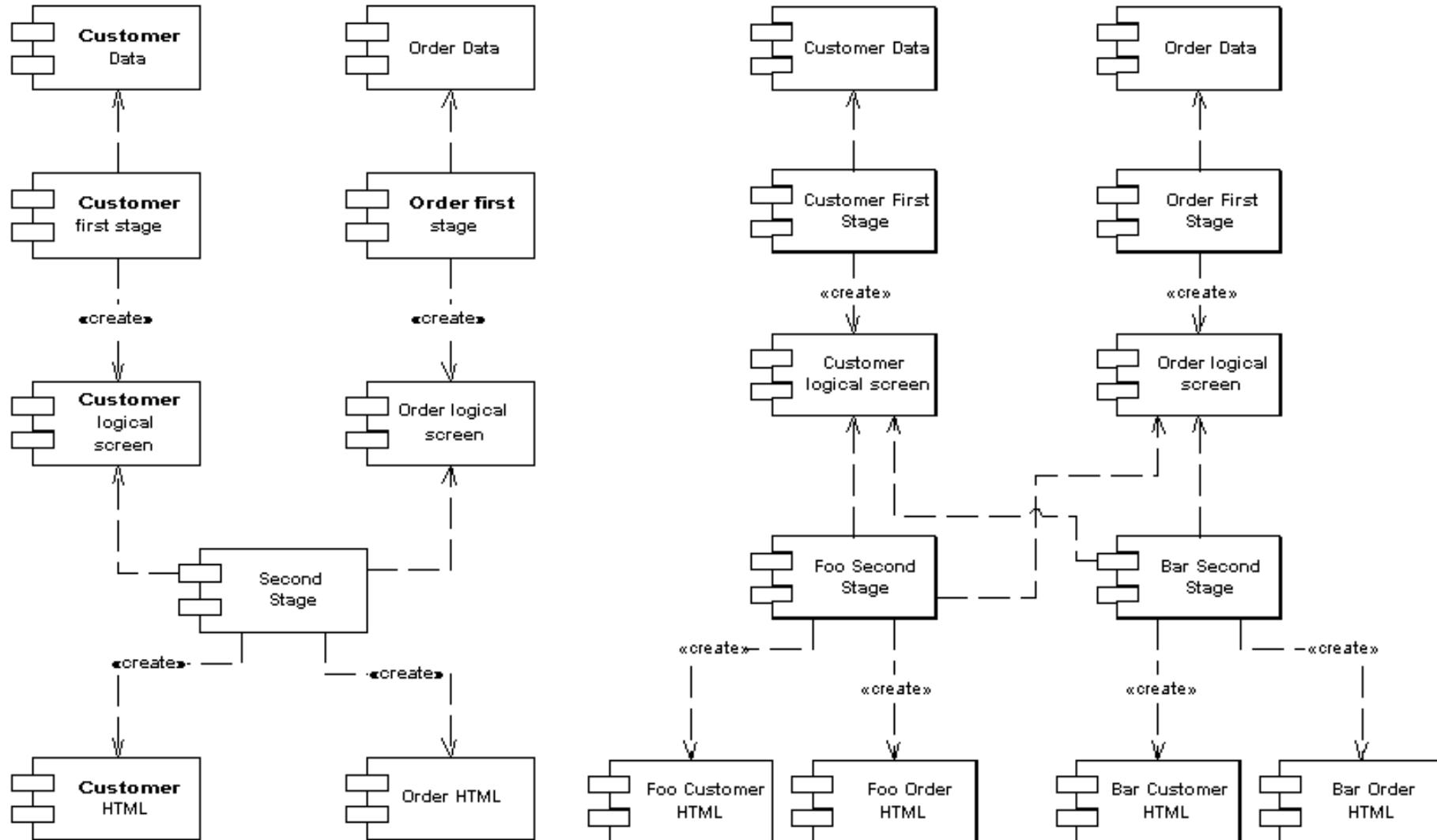
How to do it

- two-step XSLT:
 - domain-oriented XML => presentation-oriented XML,
 - presentation-oriented XML => HTML.
- presentation-oriented structure as a set of classes (table/row class):
 - domain information instantiates T/R classes.
 - renders the T/R classes into HTML
 - each presentation-oriented class generates HTML for itself or
 - having a separate HTML renderer class
- Template View based approach
 - The template system converts the logical tags into HTML.

Example



One vs. two appearances

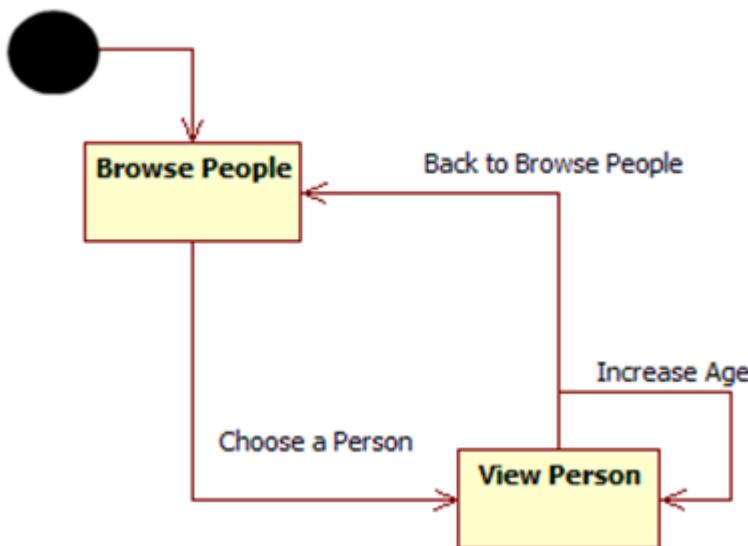


Discussion

- Advantages:
 - Two step view solves the difficulty with Transform view w.r.t. multiple transforms module & global changes.
 - In addition, if the website has multiple appearances/themes, the complexity is higher. With two step view, the issue is resolved and the advantage is compounded with multiple pages/themes.
- Liabilities:
 - It's hard to find enough commonality between the screens to get a simple enough presentation-oriented structure
 - Not for designers/non-programmers. Programmers have to write code for different rendering.
 - Harder programming model to learn
 - Complexity increases if multiple devices have to be supported. Then, the logical structure has to be common for multiple devices too.

Putting it all together [Thiel]

- People management web app
 - Browse people
 - View person
 - Change person data (i.e. increase age)

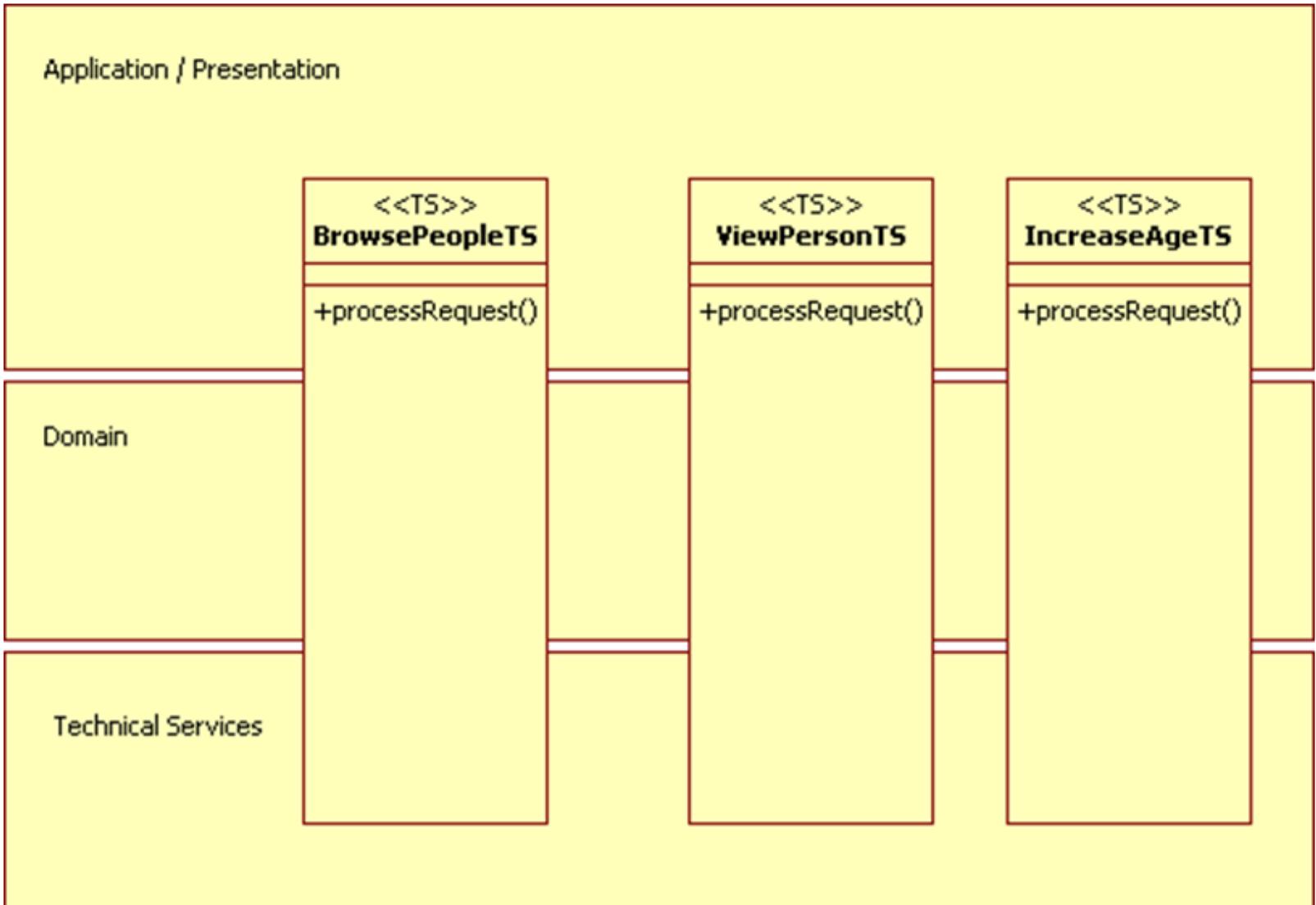


Please choose a person to see their age

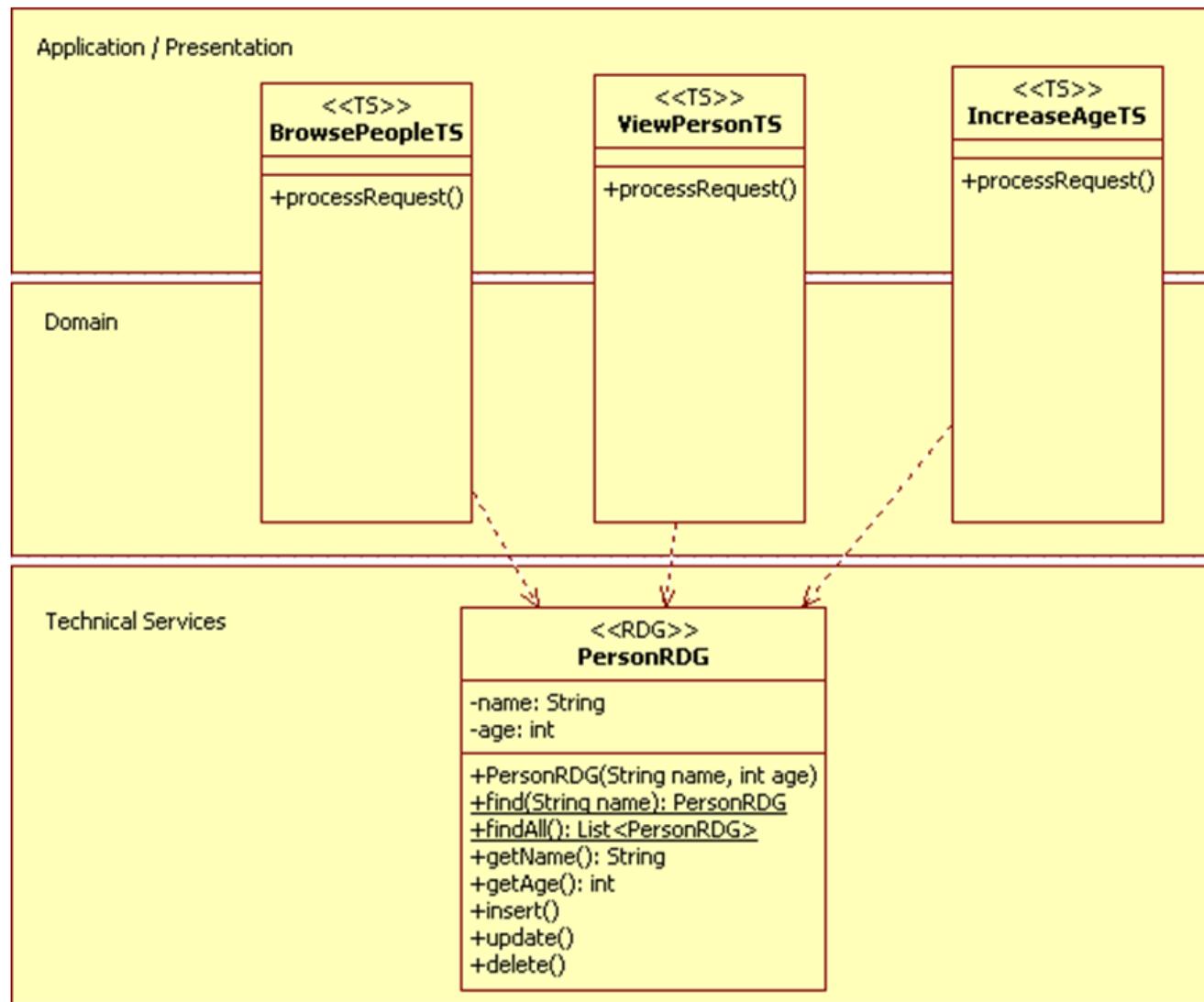
Alice
 Bob
 Chuck
 Dave
 Edith

Choose This Person!

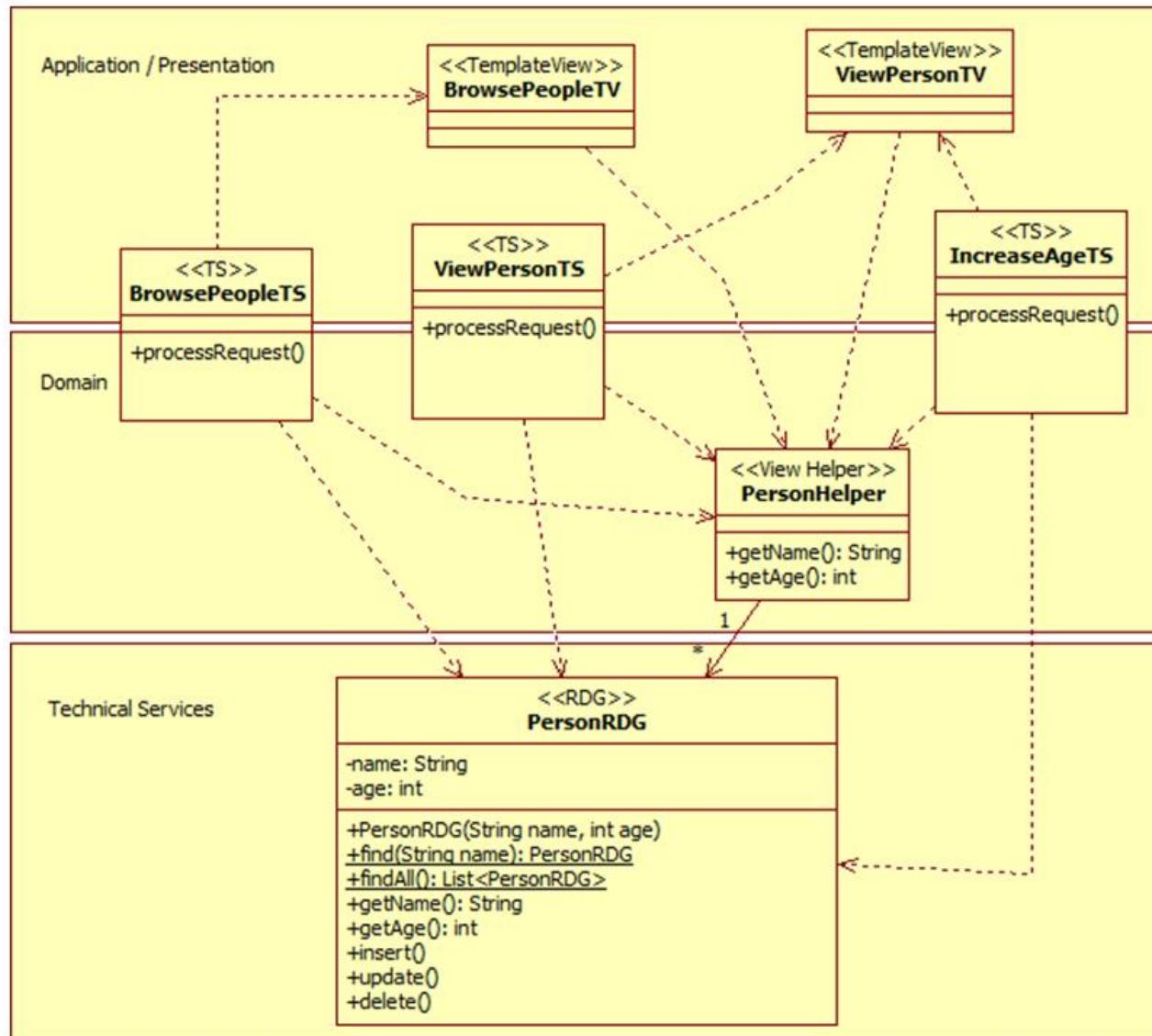
Just TS



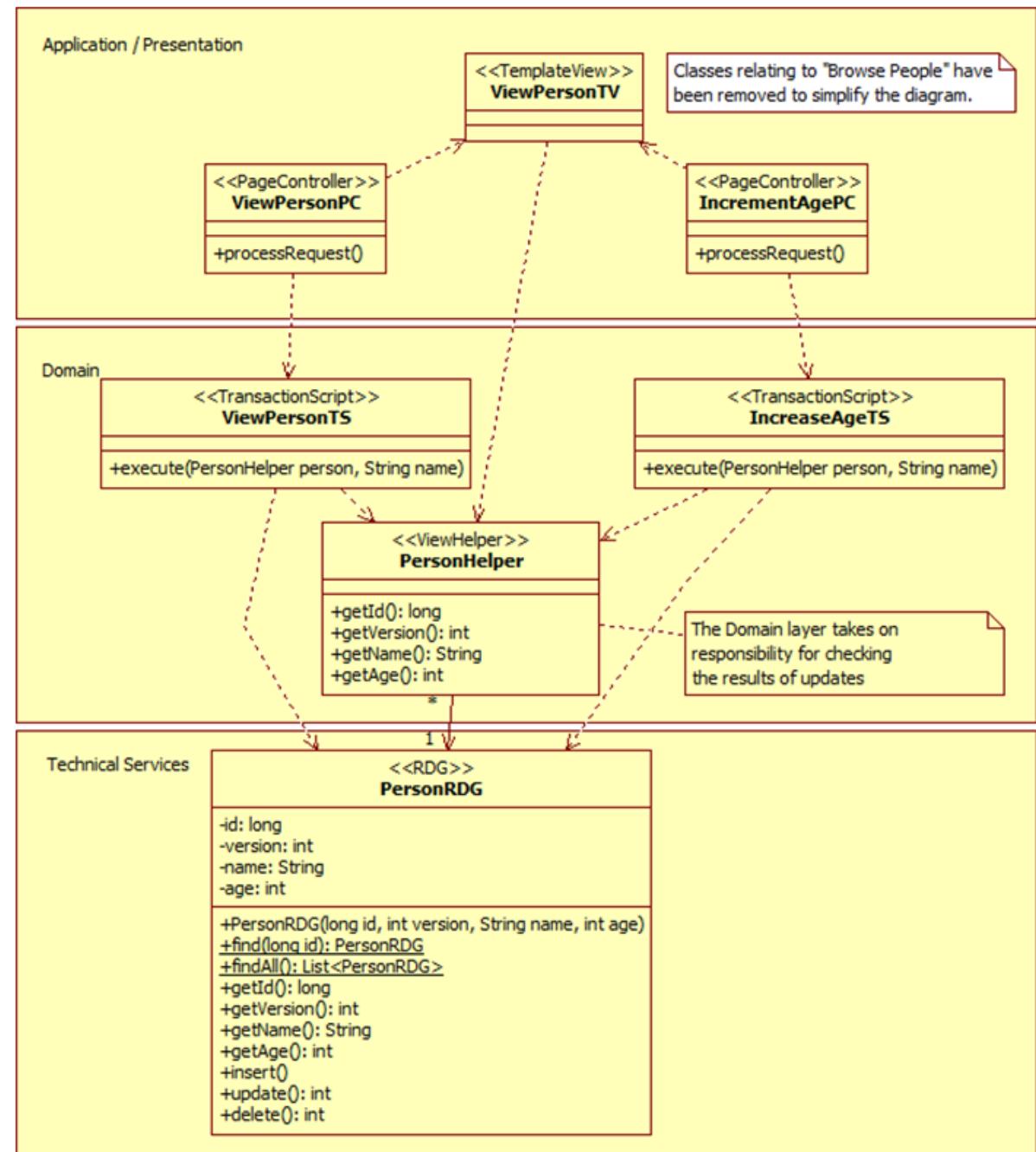
TS + RDG



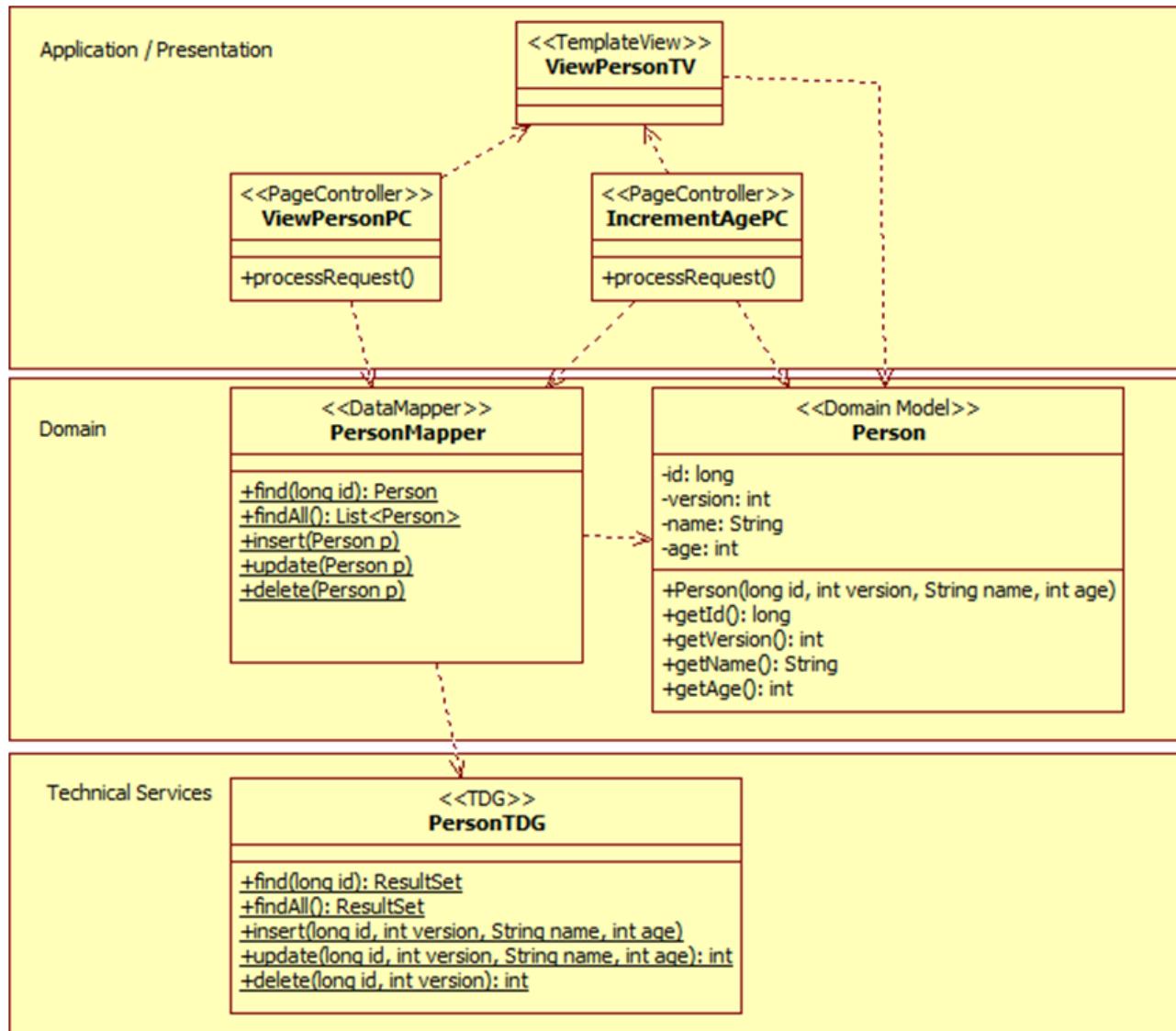
TS + RDG + TV



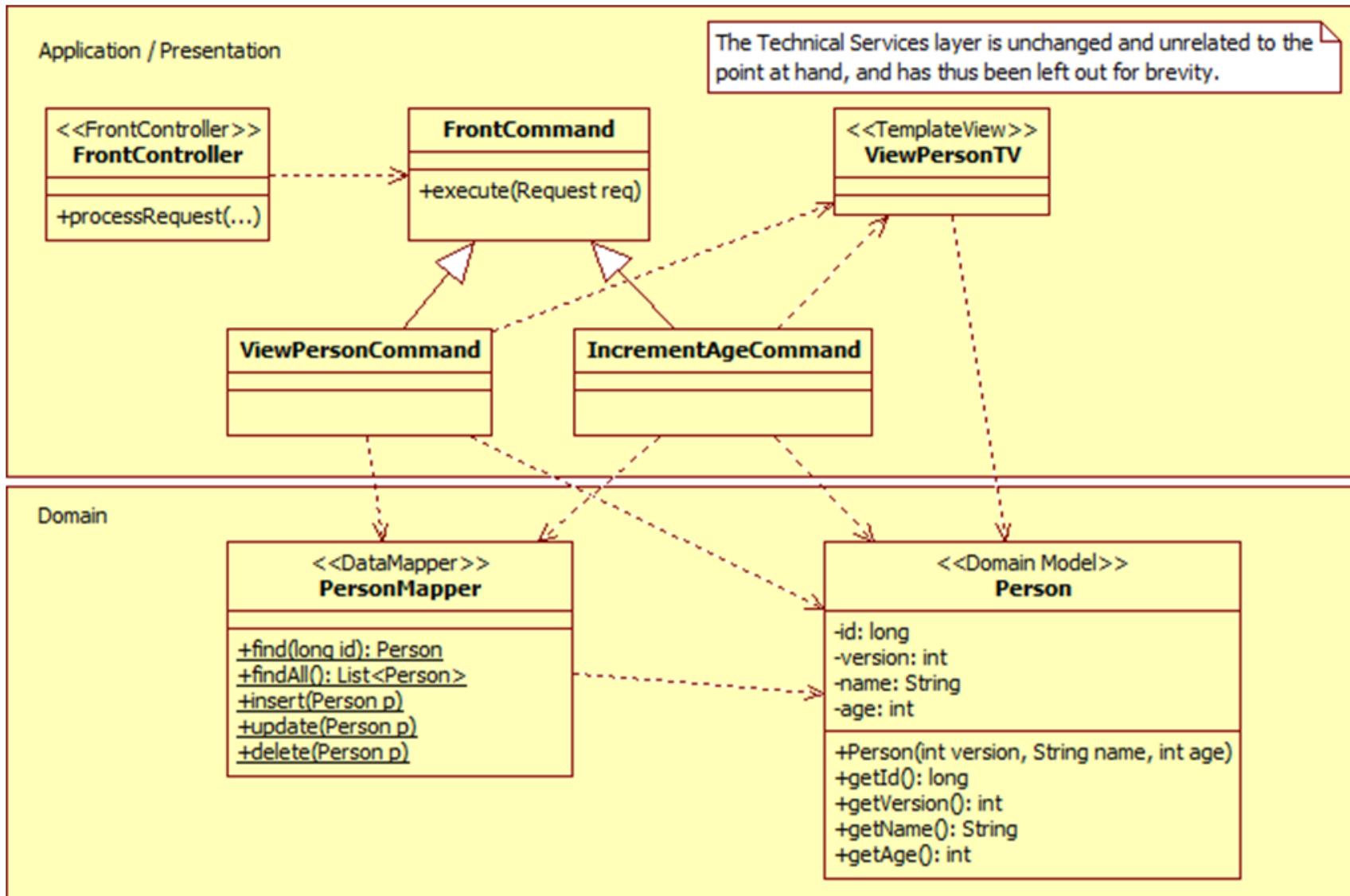
TS+RDG+TV +PC



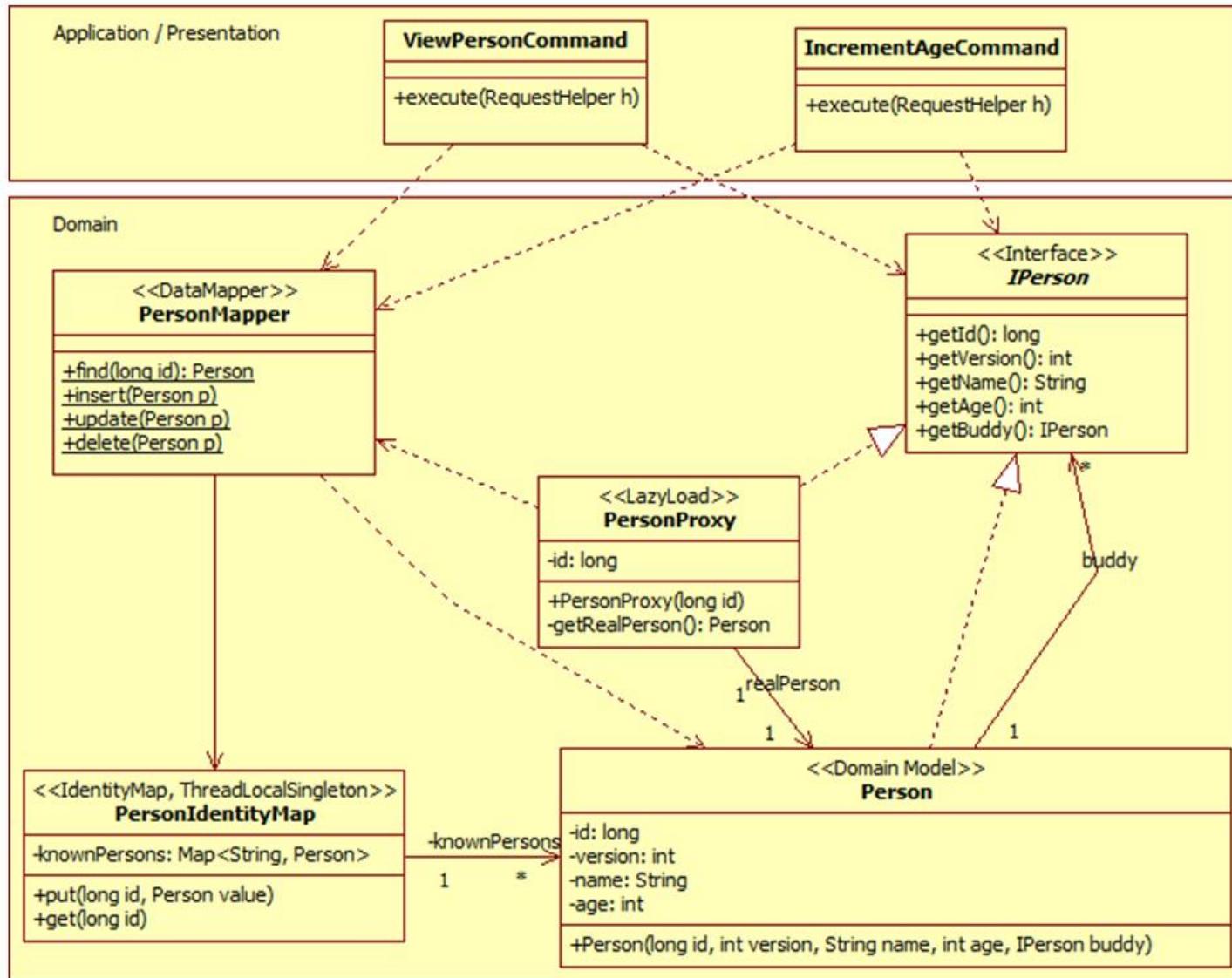
DM+TDG+DMapper+TV+PC



...or DM+...+FC



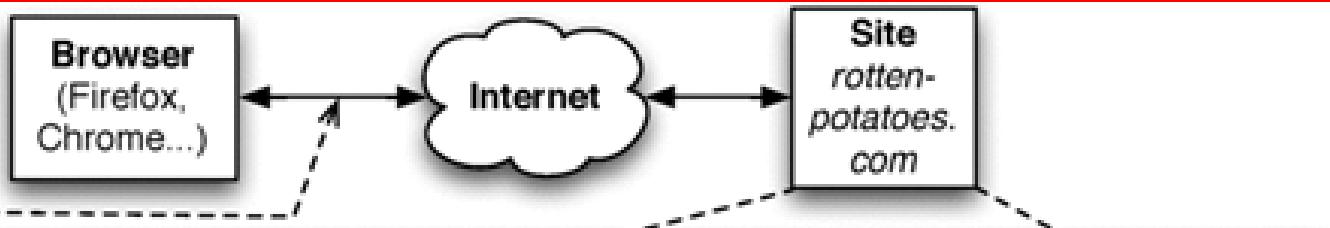
...+ Lazy load + IM



The Web

§2.1 100,000 feet

- Client-server (vs. P2P)



§2.2 50,000 feet

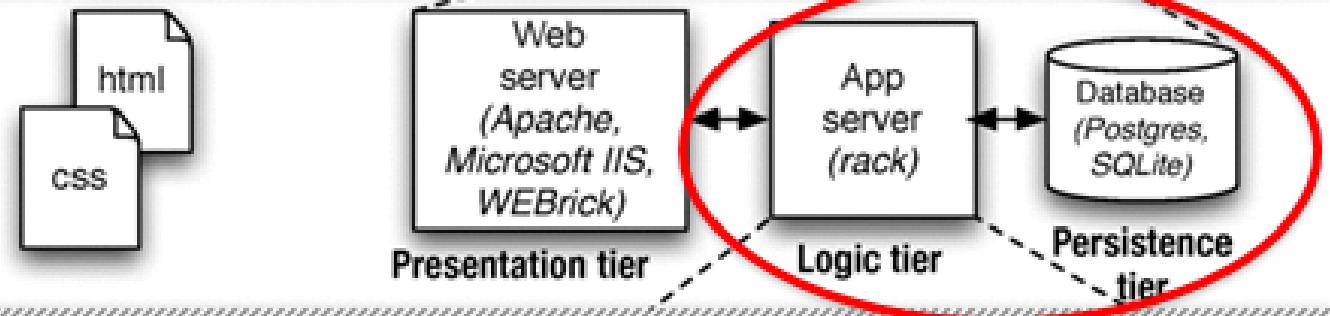
- HTTP & URIs

§2.3 10,000 feet

- XHTML & CSS

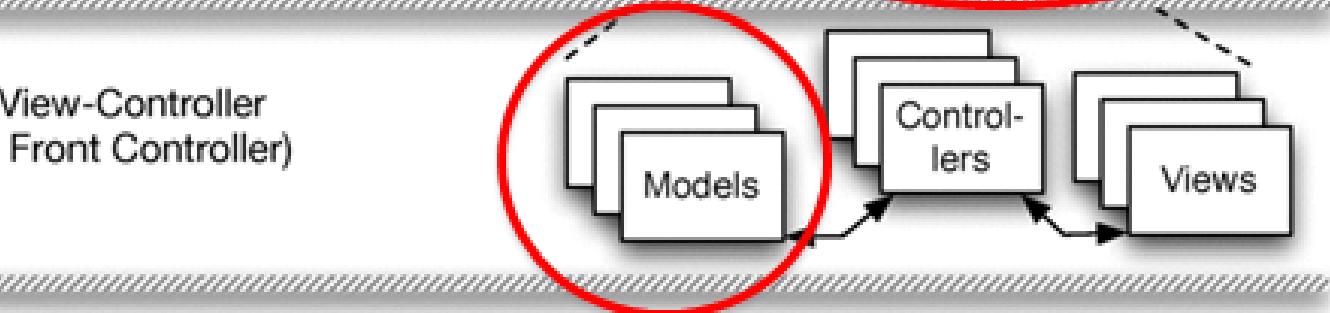
§2.4 5,000 feet

- 3-tier architecture
- Horizontal scaling



§2.5 1,000 feet—Model-View-Controller

(vs. Page Controller, Front Controller)



§2.6 500 feet: Active Record models (vs. Data Mapper)

- Active Record
- REST
- Template View

§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)

- Data Mapper

- Transform View

§2.8 500 feet: Template View (vs. Transform View)

MIDTERM HERE

- MIDTERM REVIEW START

Architectural Patterns

- Layers (and instances)
- Broker (and alternatives)
- Model-View-Controller (and variants)
- Service-based
- References
 - David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Ed.[Patterson]
 - Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley [Taylor]
 - F. Buschmann et. al, PATTERN-ORIENTED SOFTWARE ARCHITECTURE: A System of Patterns, Wiley&Sons, 2001.[POSA]
 - Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]
 - Mark Richards, Software Architecture Patterns, O'Reilly, 2015 [SAP]
 - Mark Richards, Microservices vs. Service-Oriented Architecture O'Reilly, 2016

Types of Questions

- Given a specification
 - ⇒ Analyze different AP for a solution and justify the most appropriate one
- Given an architecture
 - ⇒ Identify the used APs and analyze their consequences.
 - ⇒ Describe how different APs are related
- DON'T!!
 - Enumerate all the AP you know/copy

Patterns for Enterprise Application Architecture

- Domain Layer Patterns
 - Transaction Script
 - Domain Model
 - Table Module
 - Active Record
- Data Source Patterns
 - Row Data Gateway
 - Table Data Gateway
 - Data Mapper
- Presentation Patterns
 - Template and Transform View
 - Front and Page Controller
- Concurrency Patterns (optimistic, pessimistic)
- References
 - Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]

Types of Questions

- Given a specification
⇒ Analyze different design approaches and justify the most appropriate one
- Explain how different pattern relate to each other
 - Ex. Gateway/Façade/Adapter
- Discuss different design approaches to address different issues (ex. data access/mapping inheritance).
- Refer to the asked question, don't present general topics.

Subjects examples

In an airplane, there are many sensors: speed, altitude, cabin pressure, fuel level, etc. The monitoring system performs different checks on the sensor data. If a problem is noticed, the system either shows a warning to the pilot (e.g. low on fuel), or in a dangerous situation may react automatically (e.g. by dropping oxygen masks).

Based on a layered architecture:

- Propose a Transaction Script/Domain Model/Table Module-based solution.
- Highlight any other patterns that are included in your design.
- Discuss at least 2 advantages and 2 disadvantages of your solution

- Given an architecture:
 - What patterns do you identify?
 - Complete the architecture with a presentation/business logic/data source layer based on the patterns you know.
 - Transform the solution in a Transaction Script/Domain Model/Table Module-based solution (it depends on the architecture of the initial solution).

MIDTERM HERE

- MIDTERM REVIEW END

SOFTWARE DESIGN

Design Patterns 2

Content

- Design Patterns
 - Creational Patterns
 - Structural Patterns
 - Adapter
 - Bridge
 - Composite
 - Decorator
 - Proxy
 - Behavioral Patterns

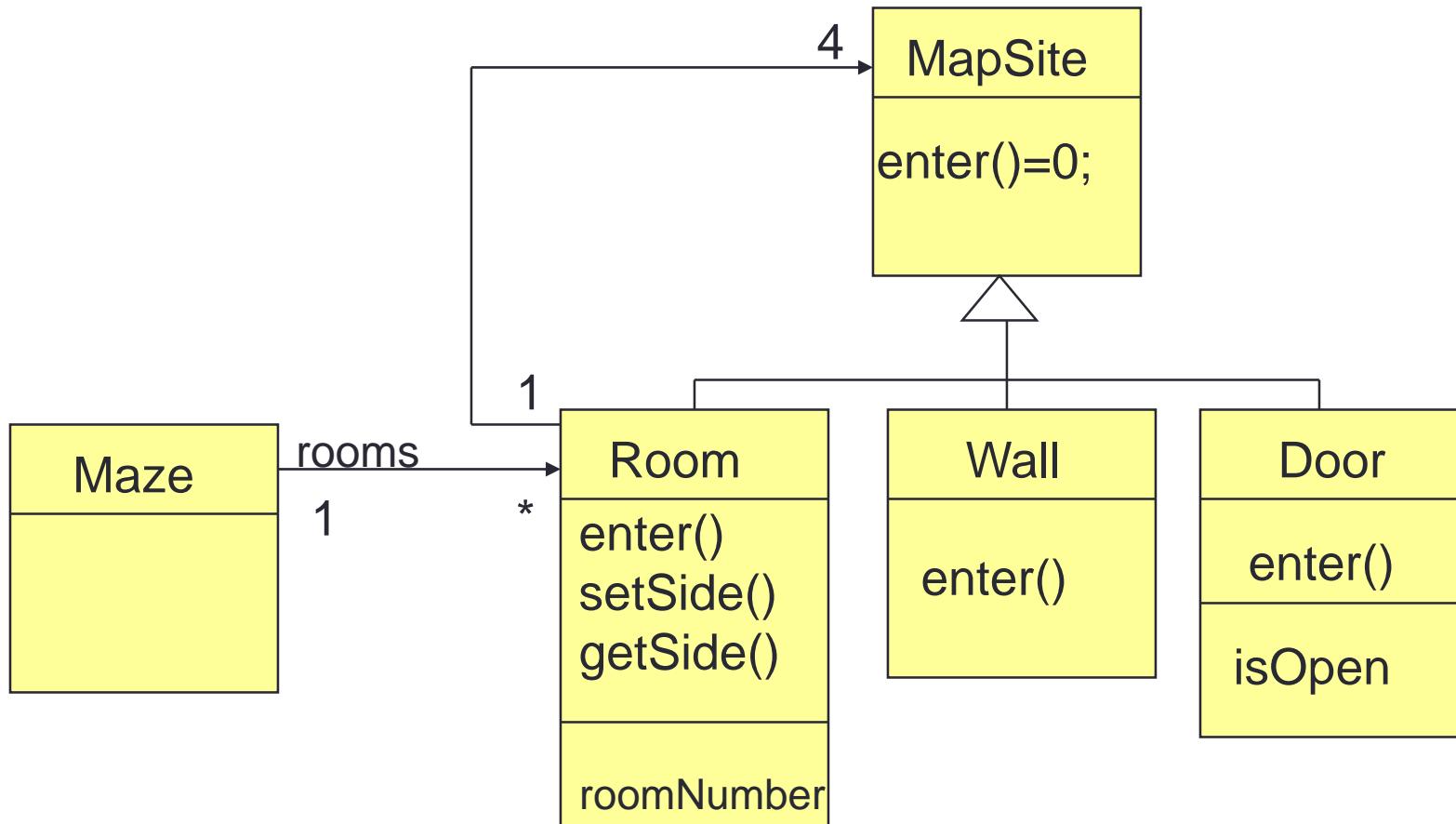
References

- Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.
- Univ. of Timisoara Course materials

Creational DP in Action

- Maze Game

Class Diagram for the Maze



Common abstract class for all Maze Components

```
enum Direction {North, South, East, West};  
  
class MapSite {  
public:  
    virtual void enter() = 0;  
};
```

- Meaning of enter() depends on *what* you are entering.
 - room → location changes
 - door → if door is open go in; else hurt your nose

Components of the maze – Maze

```
class Maze {  
public:  
    void addRoom(Room*);  
    Room * roomNo(int) const;  
private:  
};
```

A maze is a collection of rooms. Maze can find a particular room given the room number.

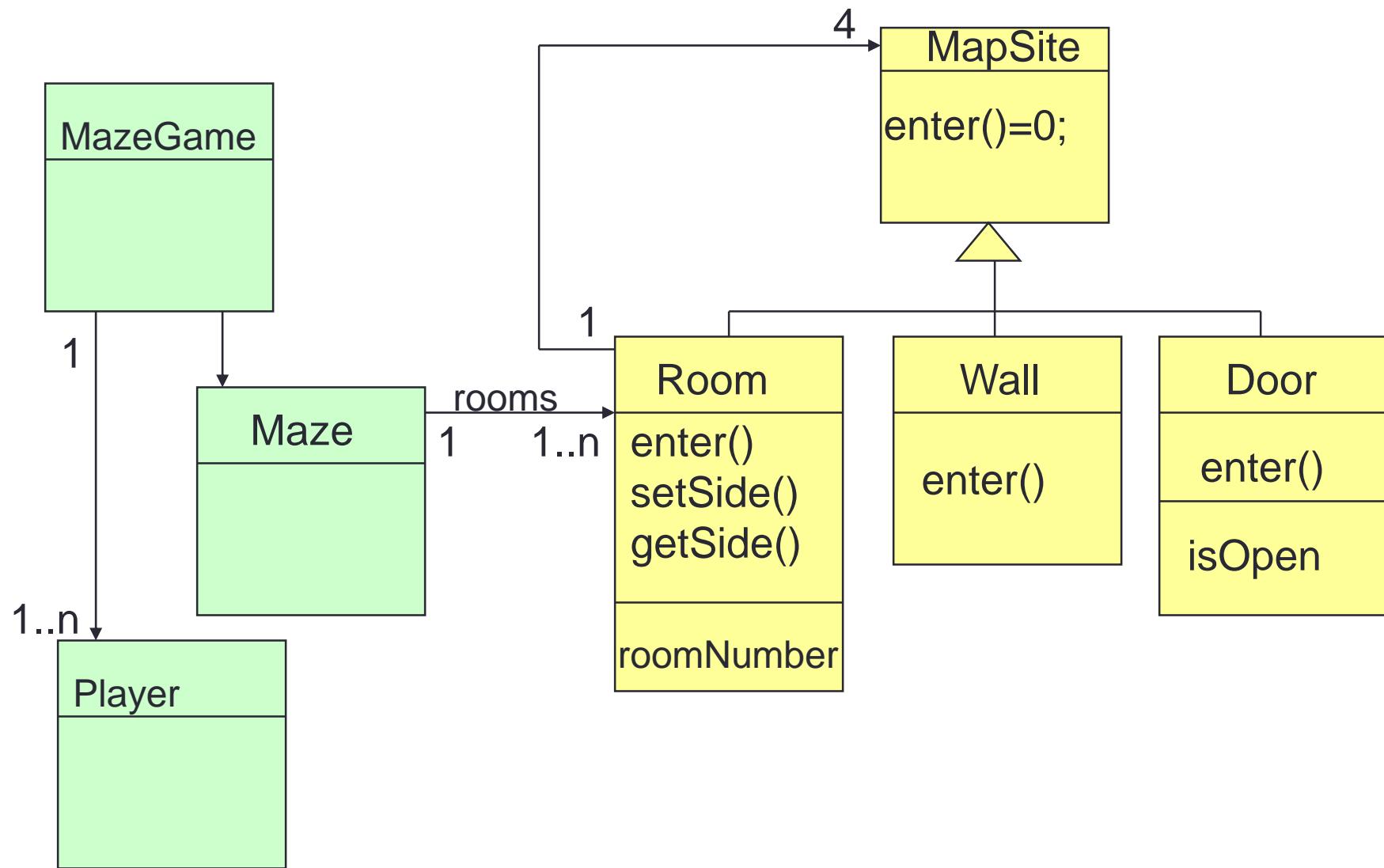
roomNo() could do a lookup using a linear search or a hash table or a simple array.

Components of the maze – Wall & Door & Room

```
class Room : public MapSite {  
public:  
    Room(int roomNo);  
    MapSite* getSide(Direction) const;  
    void setSide(Direction, MapSite*);  
  
    void enter();  
private:  
    MapSite* sides[4];  
    int roomNumber;  
}
```

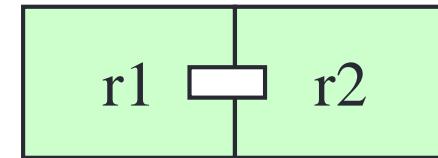
```
class Wall : public MapSite {  
public:  
    Wall();  
    virtual void enter();  
};  
  
class Door : public MapSite {  
public:  
    Door(Room* = 0, Room* = 0);  
    virtual void enter();  
    Room* otherSideFrom(Room*);  
private:  
    Room* room1;  
    Room* room2;  
    bool isOpen;  
};
```

We want to play a game!



Creating the Maze

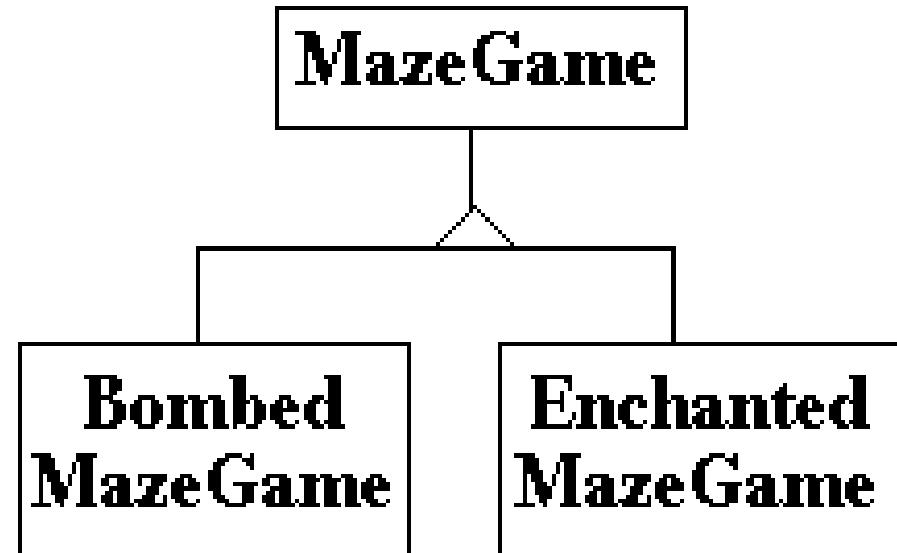
```
Maze* MazeGame::createMaze() {  
    Maze* aMaze = new Maze;  
    Room* r1 = new Room(1);  
    Room* r2 = new Room(2);  
    Door* theDoor = new Door(r1, r2);  
    aMaze->addRoom(r1);  
    aMaze->addRoom(r2);  
  
    r1->setSide(North, new Wall); r1->setSide(East, theDoor);  
    r1->setSide(South, new Wall); r1->setSide(West, new Wall);  
  
    r2->setSide(North, new Wall); r2->setSide(East, new Wall);  
    r2->setSide(South, new Wall); r2->setSide(West, theDoor);  
}
```



- The problem is **inflexibility**
 - hard-coding of maze layout
- Pattern can make game creation more flexible... *not* smaller!

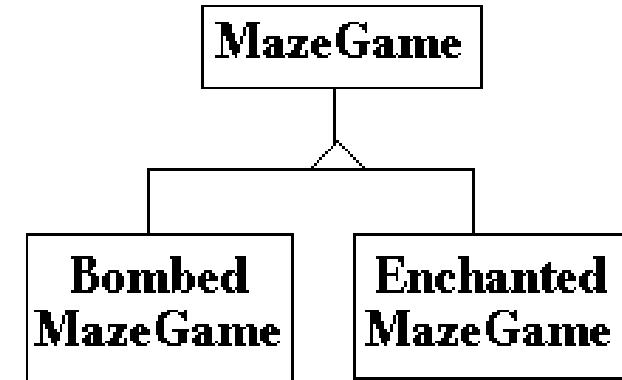
We want Flexibility in Maze Creation

- Be able to vary the kinds of mazes
 - Rooms with bombs
 - Walls that have been bombed
 - Enchanted rooms
 - Need a spell to enter the door!



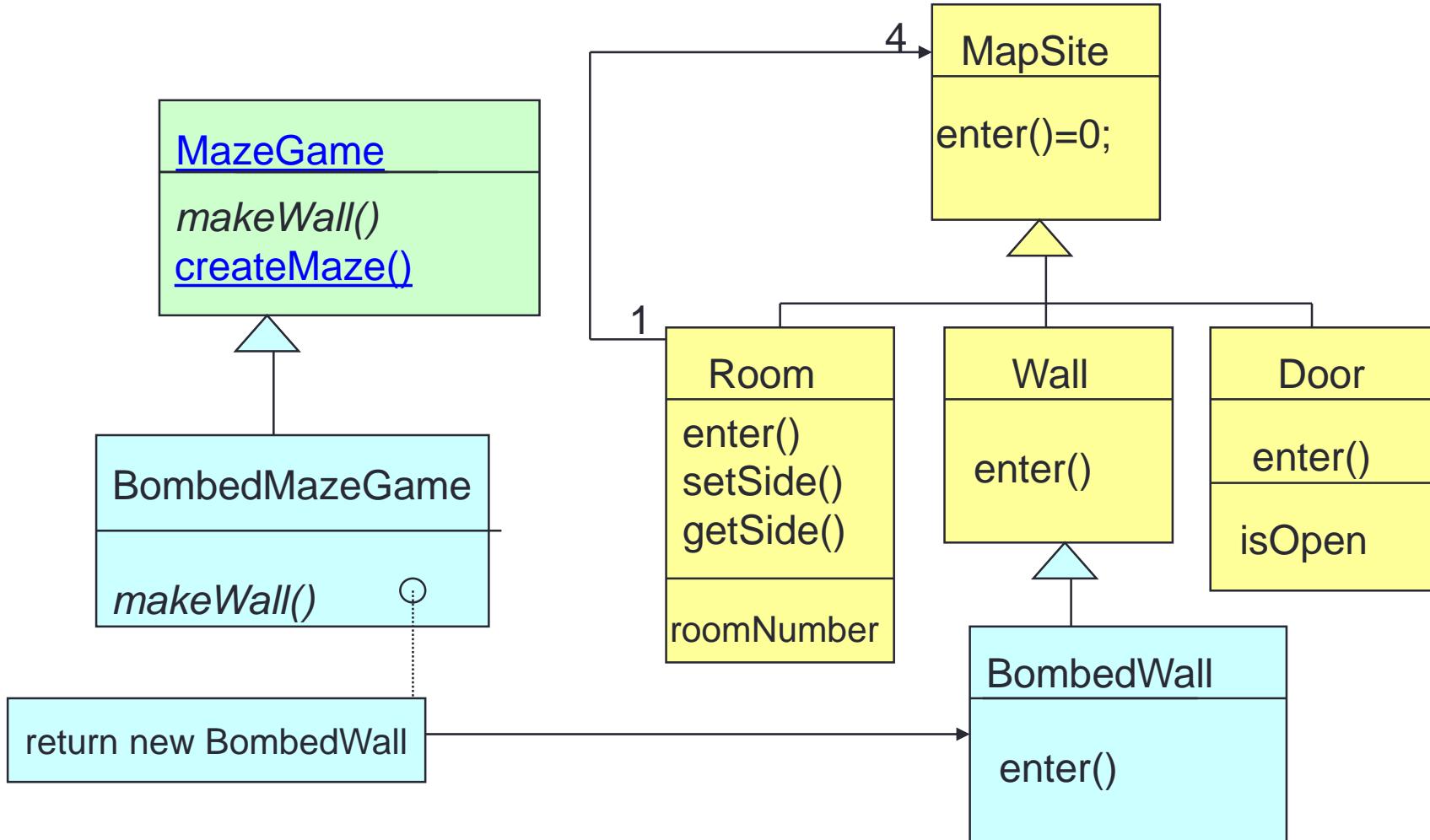
Idea 1: Subclass **MazeGame**, override **createMaze**

```
Maze* BombedMazeGame::createMaze() {  
    Maze* aMaze = new Maze;  
    Room* r1 = new RoomWithABomb(1);  
    Room* r2 = new RoomWithABomb(2);  
    Door* theDoor = new Door(r1, r2);  
    aMaze->addRoom(r1);  
    aMaze->addRoom(r2);  
  
    r1->setSide(North, new BombedWall);  
    r1->setSide(East, theDoor);  
    r1->setSide(South, new BombedWall);  
    r1->setSide(West, new BombedWall);  
    // etc...etc...  
}
```



- Lots of code duplication... :((

Idea 2: Use a Factory Method

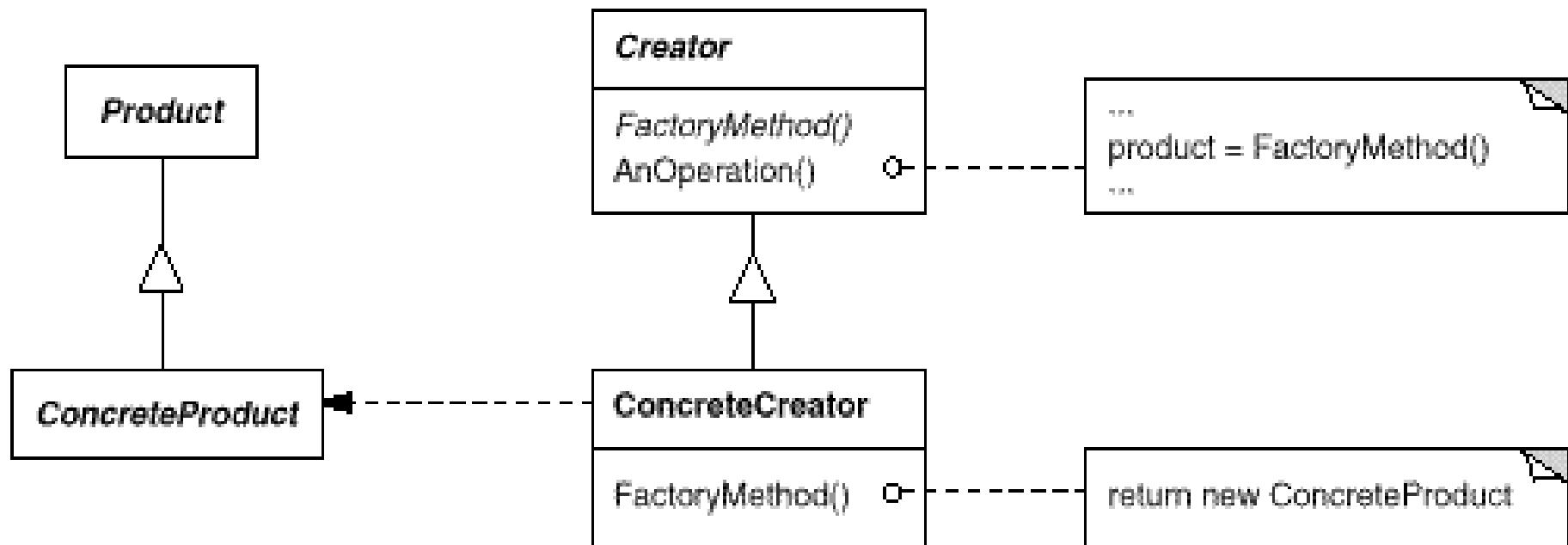


FACTORY METHOD

Basic Aspects

- Intent
 - Define an interface for creating an object, but let subclasses decide which class to instantiate.
 - Factory Method lets a class defer instantiation to subclasses
- Also Known As
 - Virtual Constructor
- Applicability
 - A class can't anticipate the class of objects it must create
 - A class wants its subclasses to specify the objects it creates
 - Classes delegate responsibility to one of several helper subclasses

Structure



Participants & Collaborations

- Product
 - defines the interface of objects that will be created by the FM
 - Concrete Product implements the interface
- Creator
 - declares the FM, which returns a product of type Product.
 - may define a default implementation of the FM
- ConcreteCreator
 - overrides FM to provide an instance of ConcreteProduct

Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct

Consequences

- Eliminate binding of application specific classes into your code.
 - creational code only deals with the Product interface
- Provide hooks for subclassing
 - subclasses can change this way the product that is created
- Clients might have to subclass the Creator just to create a particular ConcreteProduct object.

Implementation Issues

- Varieties of Factory Methods
 - Creator class is **abstract**
 - does not provide an implementation for the FM it declares
 - requires subclasses
 - Creator is a **concrete** class
 - provides default implementation
 - FM used for flexibility
 - Create objects in a separate operation so that subclasses can override it
- Parametrization of Factory Methods
 - A variation on the pattern lets the factory method create multiple kinds of products
 - a *parameter* identifies the type of Product to create
 - all created objects share the Product interface

Parameterizing the Factory

```
class Creator {  
public:  
    virtual Product * create(productId) ;  
};  
  
Product* Creator::create(ProductId id) {  
    if (id == MINE) return new MyProduct;  
    if (id == YOURS) return new YourProduct;  
}  
  
Product * MyCreator::create(ProductId id) {  
    if (id == MINE) return new YourProduct;  
    if (id == YOURS) return new MyProduct;  
    if (id == THEIRS) return TheirProduct;  
    return Creator::create(id); // called if others fail  
}
```

selectively *extend* or *change* products that get created

Idea 3: Factory Method in Product

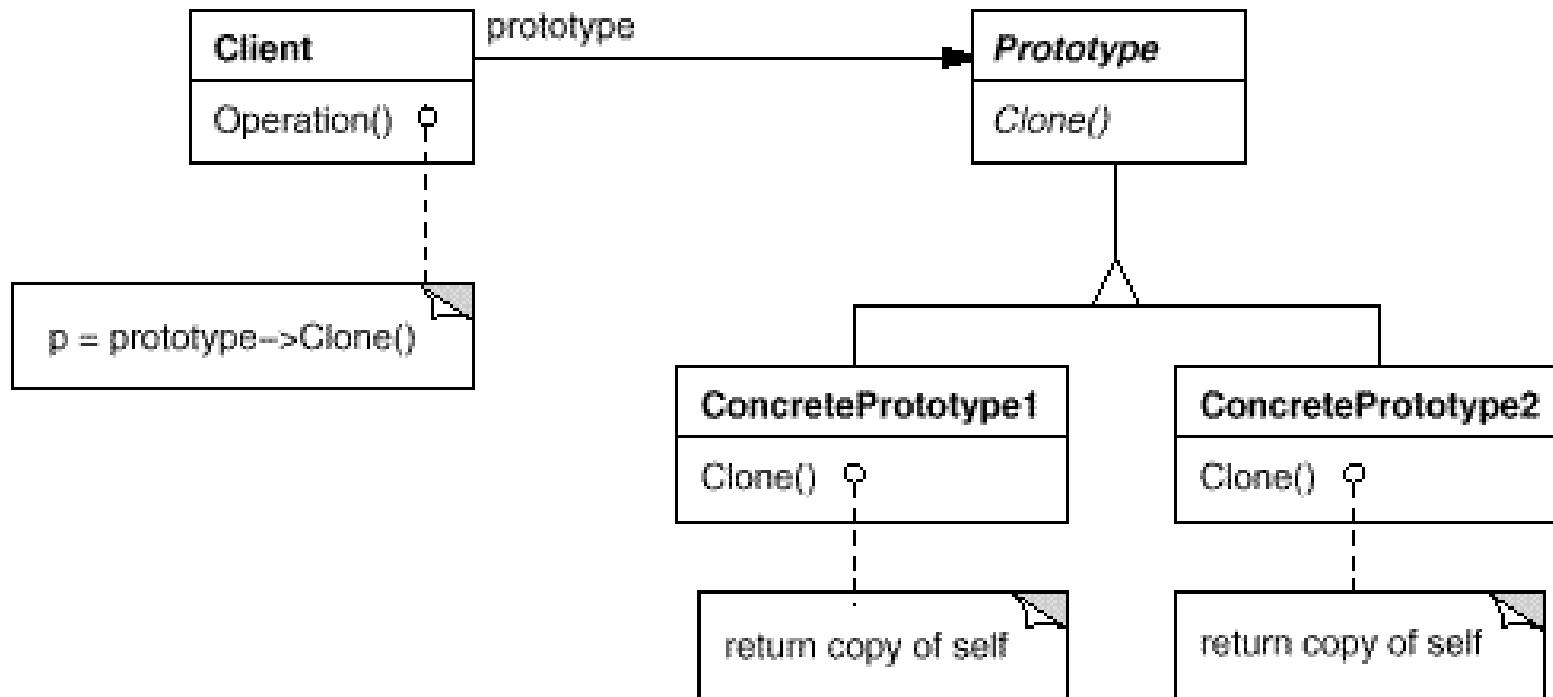
- Make the product responsible for creating itself
 - e.g. let the Door know how to construct an instance of it rather than the MazeGame
- The client of the product needs a reference to the "creator"
 - specified in the constructor
- [Sample Code](#)

THE PROTOTYPE PATTERN

Basic Aspects

- Intent
 - Specify the kinds of objects to create using a prototypical instance
 - Create new objects by copying this prototype
- Applicability
 - when a client class should be independent of how its products are created, composed, and represented **and**
 - when the classes to instantiate are *specified at run-time*

Structure



Participants & Collaborations

- Prototype
 - declares an interface for cloning itself.
- ConcretePrototype
 - implements an operation for cloning itself.
- Client
 - creates a new object by asking a prototype to clone itself.
- *A client asks a prototype to clone itself.*
- *The client class must initialize itself in the constructor*
 - *with the proper concrete prototype.*

Consequences

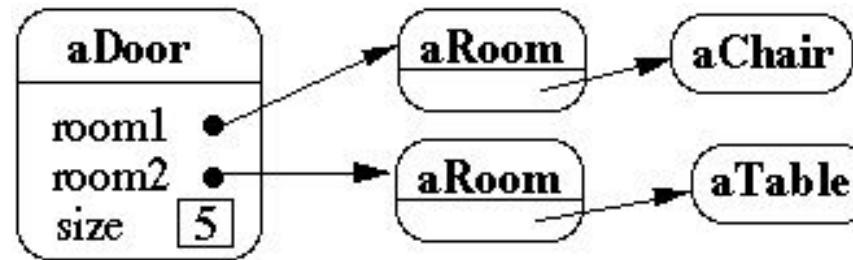
- Adding and removing products at run-time
- Reduced subclassing
 - avoid parallel hierarchy for creators
- Specifying new objects by varying **values of prototypes**
 - client exhibits new behavior by delegation to prototype
- Each subclass of Prototype must implement **clone**
 - difficult when classes already exist or
 - internal objects don't support copying or have circular references

Implementation Issues

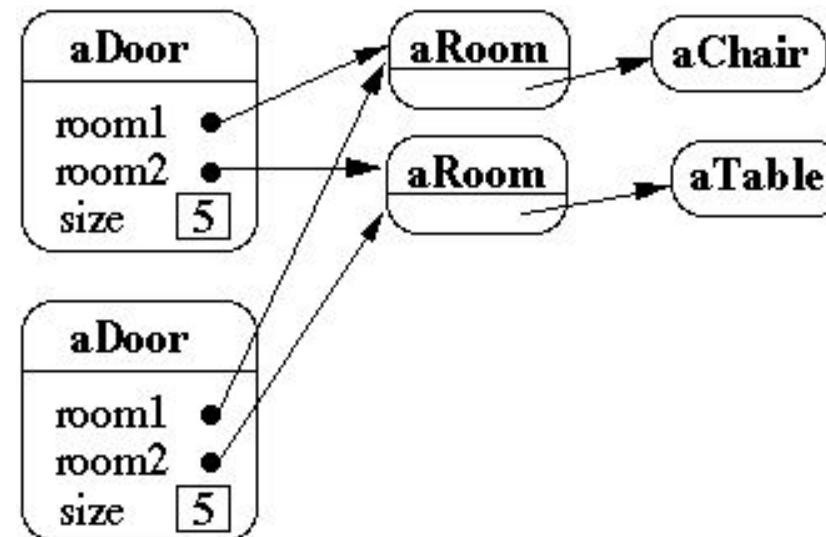
- Using a Prototype manager
 - number of prototypes isn't fixed
 - keep a registry → **prototype manager**
 - clients instead of knowing the prototype know a manager
- Initializing clones
 - heterogeneity of initialization methods
 - write an **Initialize** method
- Implementing the **clone** operation
 - shallow vs. deep copy

Shallow Copy vs. Deep Copy

Original

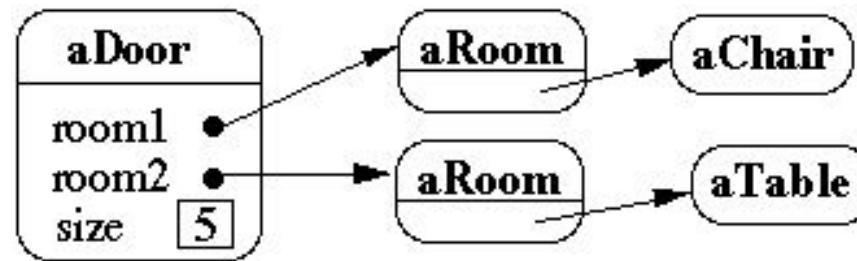


Shallow Copy

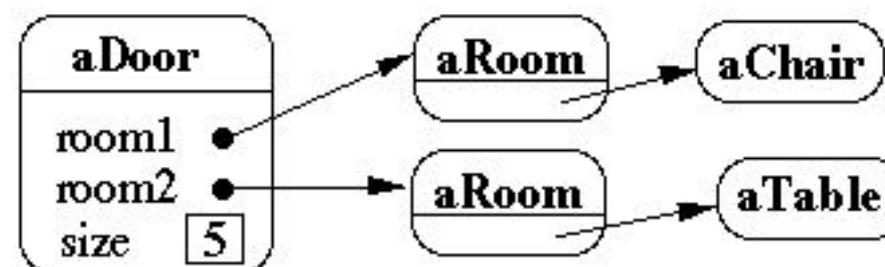
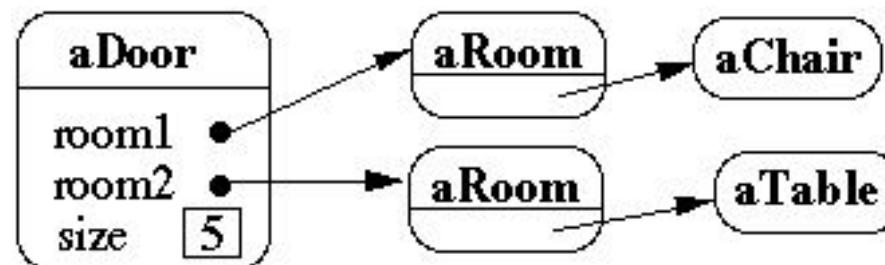


Shallow Copy vs. Deep Copy (2)

Original



Deep Copy



Cloning in C++ – Copy Constructors

```
class Door {  
public:  
    Door();  
    Door( const Door& );  
    virtual Door* clone() const;  
    virtual void Initialize( Room*, Room* );  
private:  
    Room* room1; Room* room2;  
};  
  
//Copy constructor  
Door::Door ( const Door& other )  {  
    room1 = other.room1; room2 = other.room2;  
}  
  
Door* Door::clone() {  
    return new Door( *this );  
}
```

Cloning in Java – Object clone()

```
protected Object clone() throws  
CloneNotSupportedException
```

- Creates a clone of the object.
 - allocate a new instance and,
 - place a *bitwise clone* of the current object in the new object.

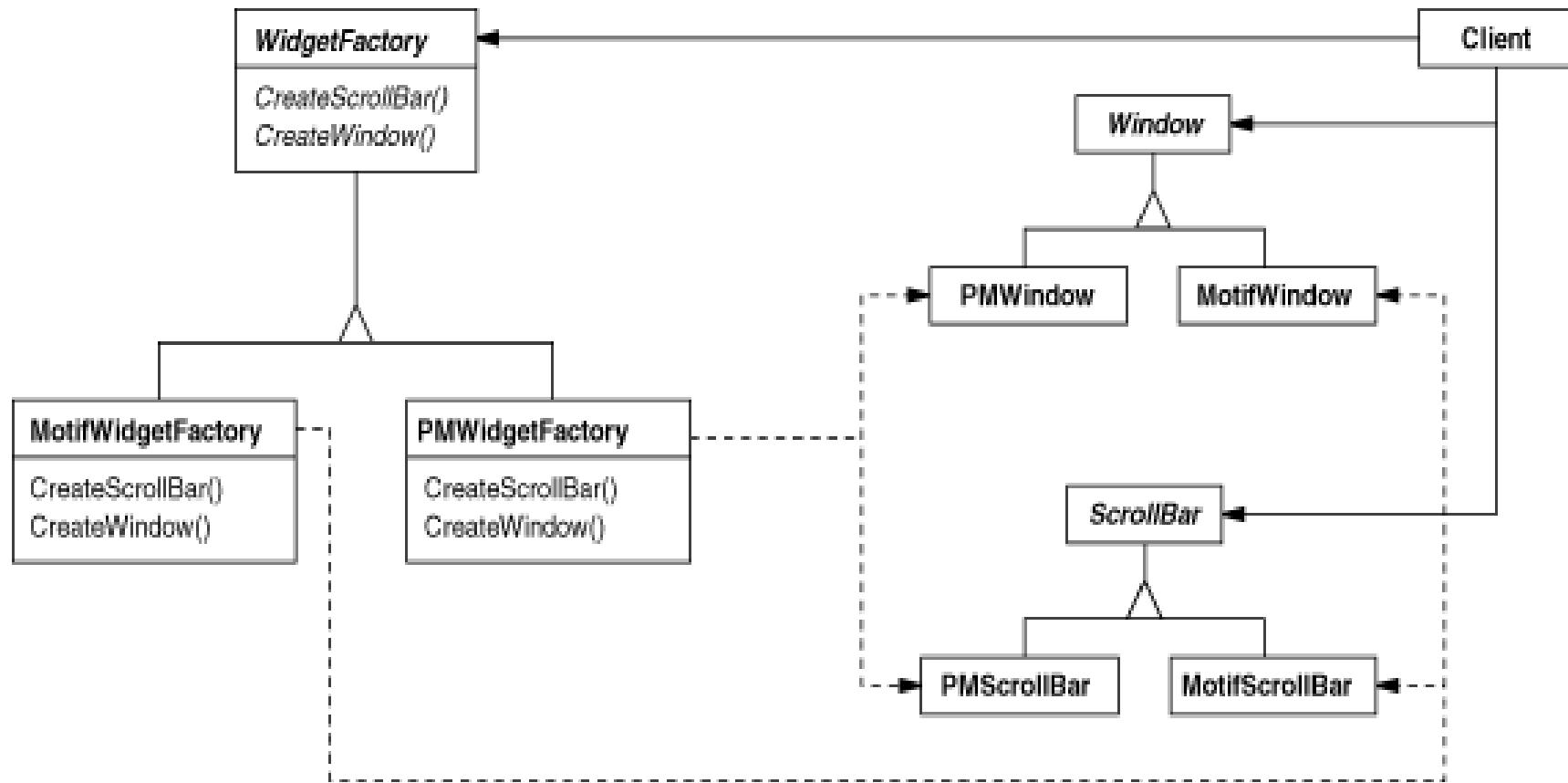
```
class Door implements Cloneable {  
    public void Initialize( Room a, Room b) {  
        room1 = a; room2 = b;  
    }  
  
    public Object clone() throws  
CloneNotSupportedException {  
        return super.clone();  
    }  
    Room room1, room2;  
}
```

Solving the Maze Problem

- [See code....](#)

ABSTRACT FACTORY

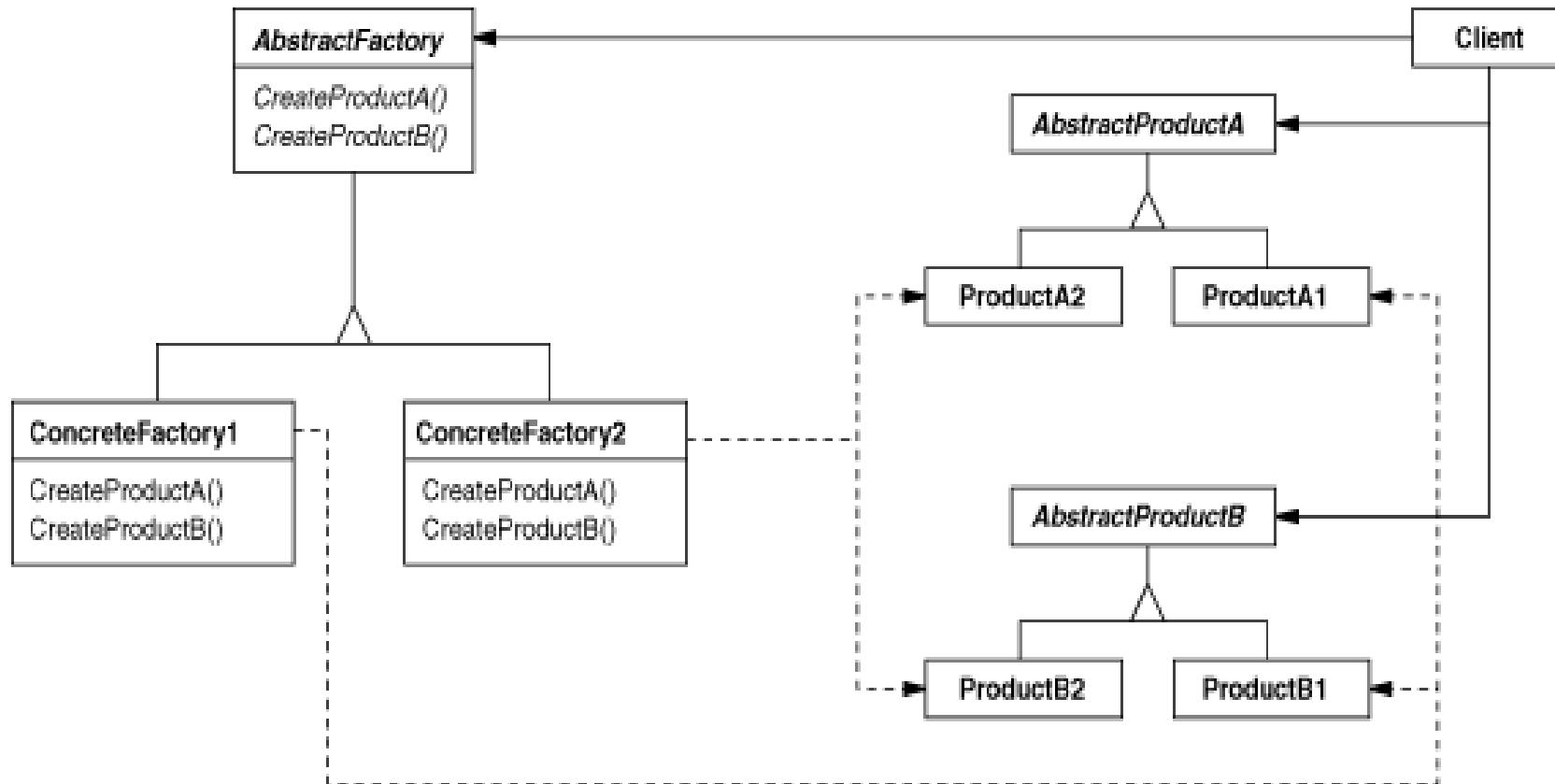
Introductive Example



Basic Aspects

- Intent
 - Provide an interface for creating **families of related or dependent objects** without specifying their concrete classes
- Applicability
 - System should be independent of how its products are created, composed and represented
 - System should be configured with one of multiple families of products
 - Need to **enforce** that a family of product objects is used together

Structure



Participants & Collaborations

- Abstract Factory
 - declares an interface for operations to create abstract products
- ConcreteFactory
 - implements the operations to create products
- AbstractProduct
 - declares an interface for a type of product objects
- ConcreteProduct
 - declares an interface for a type of product objects
- Client
 - uses only interfaces decl. by AbstractFactory and AbstractProduct
- *A single instance of a ConcreteFactory created.*
 - *create products having a particular implementation*

Consequences

- Isolation of concrete classes
 - appear in **ConcreteFactories** not in client's code
- Exchanging of product families becomes easy
 - a **ConcreteFactory** appears only in one place
- Promotes consistency among products
 - all products in a family change **at once**, and change **together**
- Supporting new kinds of products is difficult
 - requires a change in the interface of AbstractFactory
 - ... and consequently all subclasses

Implementation Issues

- Factories as Singletons
 - to assure that only one ConcreteFactory per product family is created
- Creating the Products
 - collection of *Factory Methods*
 - can be also implemented using *Prototype*
 - define a prototypical instance for each product in ConcreteFactor
- Defining Extensible Factories
 - a single factory method with parameters
 - more flexible, less safe!

Creating Products

- ...using own factory methods
- ... using product's factory methods
 - subclass just provides the concrete products in the constructor
 - spares the reimplementation of FM's in subclasses

Solving the Maze problem....

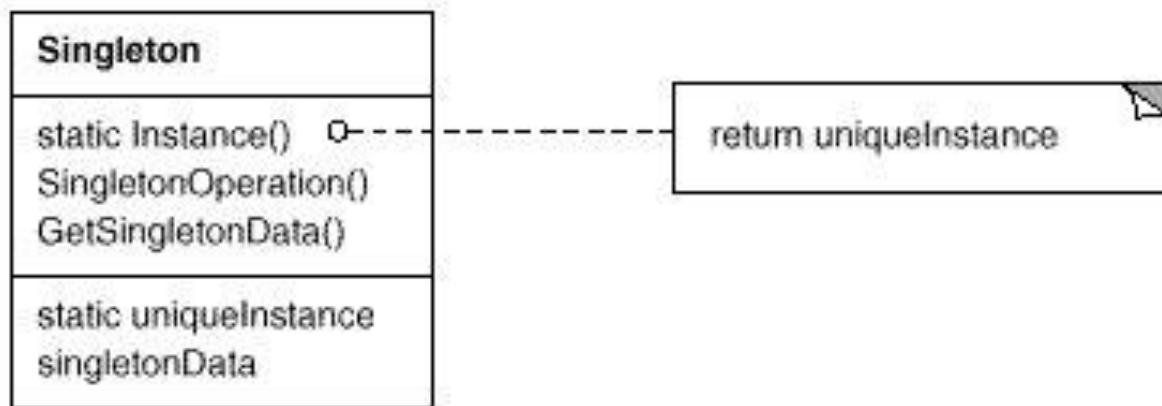
- see [code](#)...

SINGLETON

Basics

- Intent
 - Ensure a class has only one instance and provide a global point of access to it
- Applicability
 - want exactly one instance of a class
 - accessible to clients from one point
 - want the instance to be extensible
 - can also allow a countable number of instances
 - improvement over global namespace

Structure of the Pattern



Put constructor in private/protected data section

Participants and Collaborations

- Singleton
 - defines an **Instance** method that becomes the single "gate" by which clients can access its unique instance.
 - **Instance** is a class method (static member function in C++)
 - may be responsible for creating its own unique instance
 - *Clients access Singleton instances solely through the **Instance** method*

Consequences

- Controlled access to sole instance
- Permits refinement of operations and representation
- Permits a variable (but precise) number of instances
- Reduced global name space

Making a single MazeFactory

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};

MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

Structural Patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- Structural **class** patterns use **inheritance** to compose interfaces or implementations.
- Rather than composing interfaces or implementations, structural **object** patterns describe ways to **compose** objects to realize new functionality.

Paying by PayPal

```
<?php  
class PayPal {  
    public function construct() {  
        // Your Code here //  
    }  
    public function sendPayment($amount) {  
        // Paying via Paypal //  
        echo "Paying via PayPal: ". $amount;  
    }  
}  
  
$paypal = new PayPal();  
$paypal->sendPayment('2629');
```

Changes

- sendPayment -> payAmount
- COMMIT TO AN INTERFACE, NOT AN IMPLEMENTATION!

```
class PayPal {  
    public function construct() {  
        // Your Code here //  
    }  
    public function sendPayment($amount) {  
        // Paying via Paypal //  
        echo "Paying via PayPal: ". $amount;  
    }  
}
```

```
// Simple Interface for each Adapter we create
interface paymentAdapter {
    public function pay($amount);
}

class paypalAdapter implements paymentAdapter {
    private $paypal;
    public function construct(PayPal $paypal) {
        $this->paypal = $paypal;
    }
    public function pay($amount) {
        $this->paypal->sendPayment($amount);
    }
}
```

```
// Client Code
$paypal = new paypalAdapter(new PayPal());
$paypal->pay('2629');
```

```
class paypalAdapter implements paymentAdapter {  
    private $paypal;  
    public function construct(PayPal $paypal) {  
        $this->paypal = $paypal;  
    }  
    public function pay($amount) {  
        $this->paypal->payAmount($amount);  
    }  
}
```

Adapter Pattern

- Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- Also known as

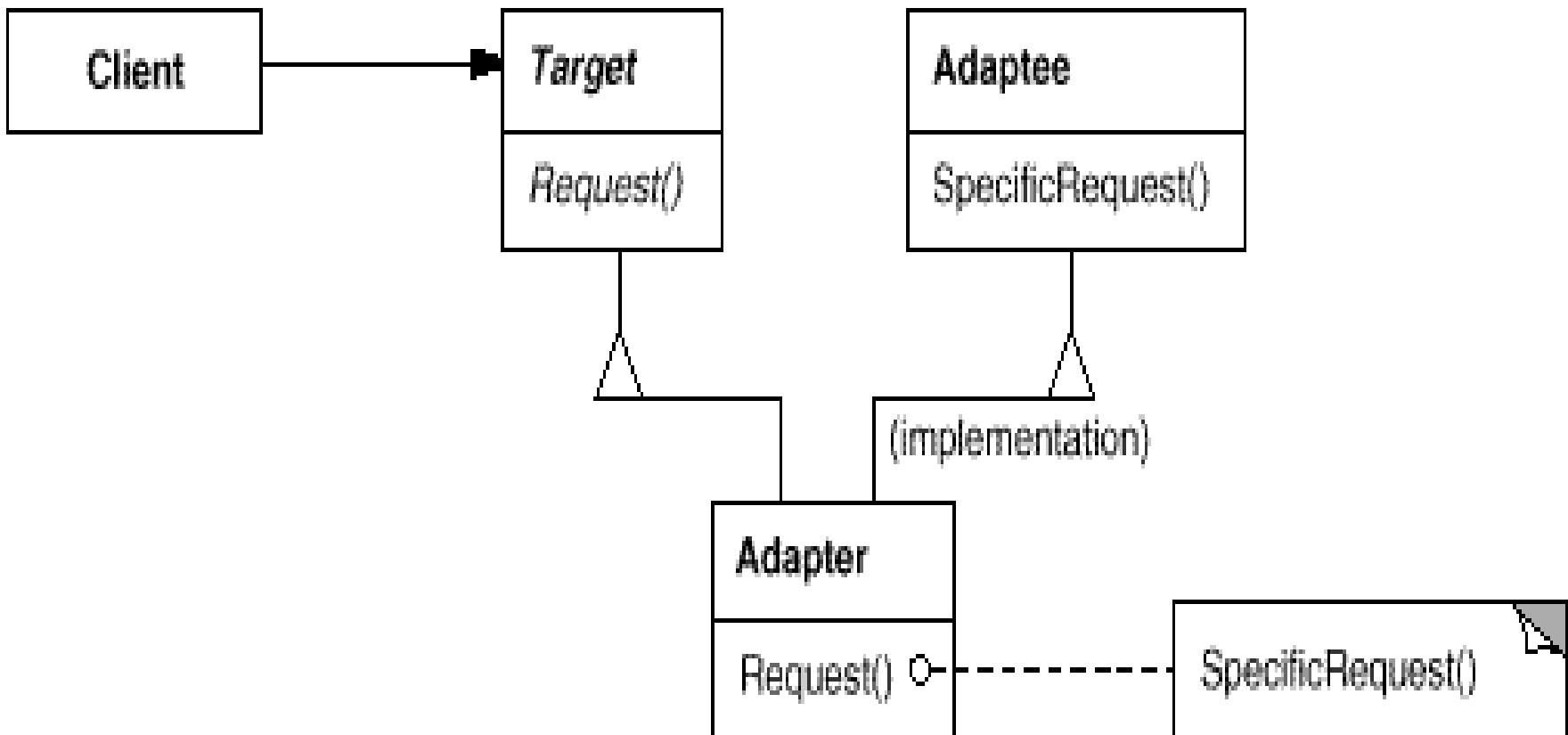
Wrapper

Adapter Pattern

- Applicability
 - you want to use an existing class, and its interface does not match the one you need.
 - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
 - (*object adapter only*) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

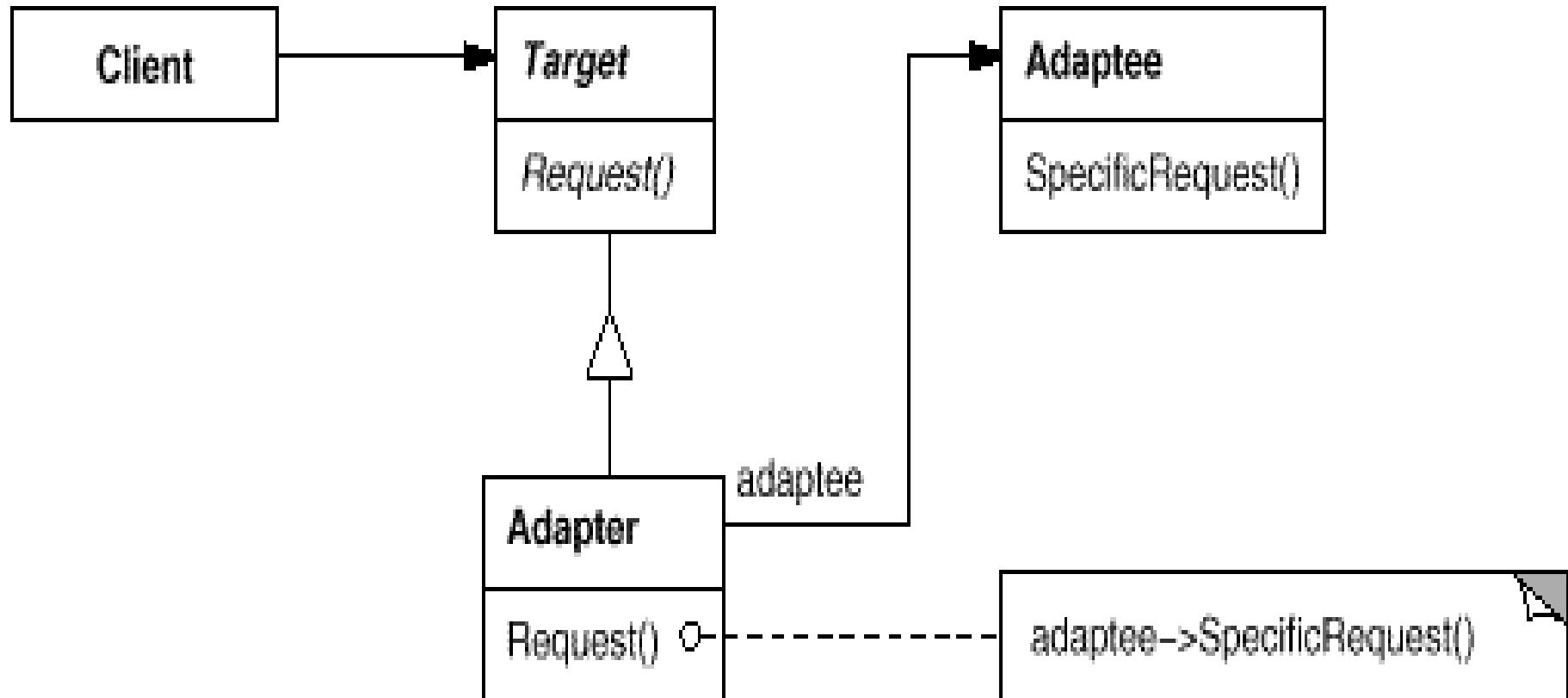
Adapter Pattern Structure

- Class Adapter



Adapter Pattern Structure

- Object Adapter



Adapter Pattern - Participants

- **Target**
 - defines the domain-specific interface that Client uses.
- **Client**
 - collaborates with objects conforming to the Target interface.
- **Adaptee**
 - defines an existing interface that needs adapting.
- **Adapter**
 - adapts the interface of Adaptee to the Target interface.

Adapter Pattern Consequences

- Class Adapter
 - adapts Adaptee to Target by committing to a concrete Adapter class
=> a class adapter won't work when we want to adapt a class *and* all its subclasses.
 - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
 - introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

```
class Adapter extends Adaptee implements Target {  
    public void request() {  
        specificRequest()  
    }  
}
```

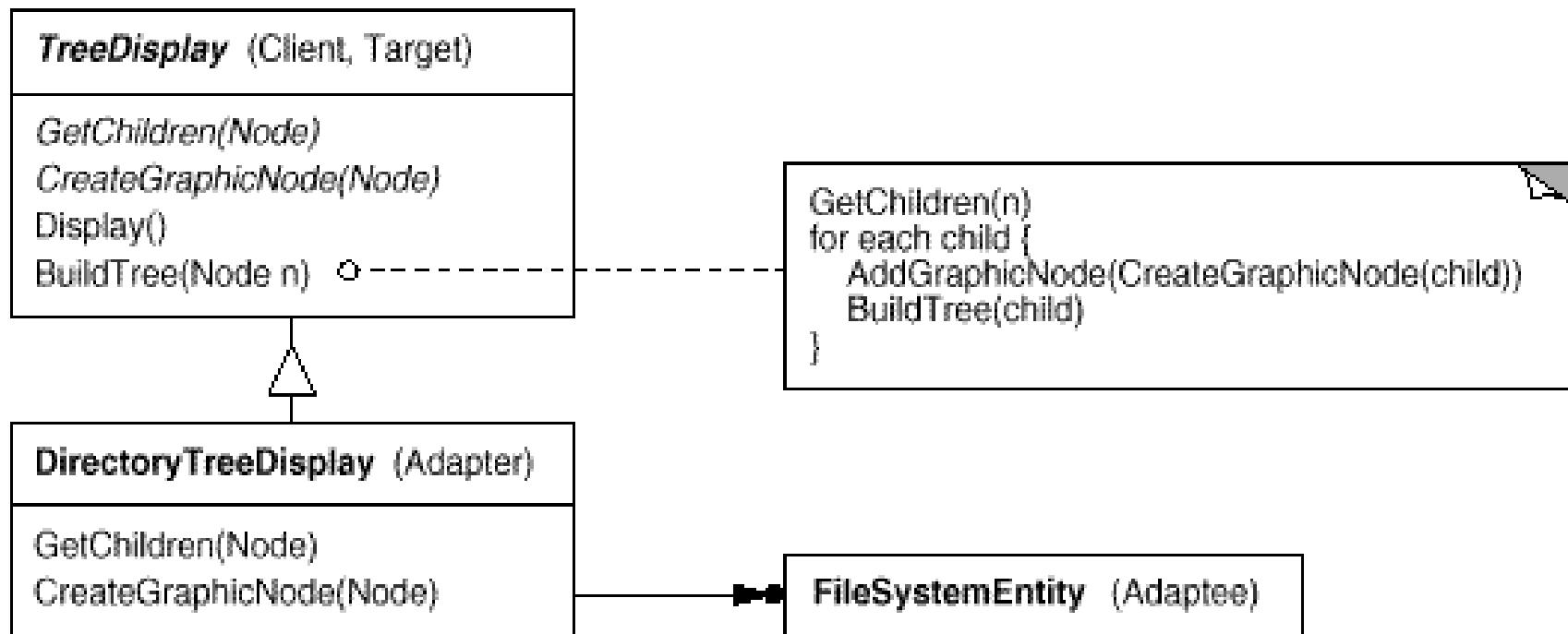
Adapter Pattern Consequences

- Object Adapter
 - lets a single Adapter work with many Adaptees - the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
 - makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

```
class Adapter implements Target {  
    Adaptee adapted;  
    public void request() {  
        adapted.specificRequest();  
    }  
    ... }
```

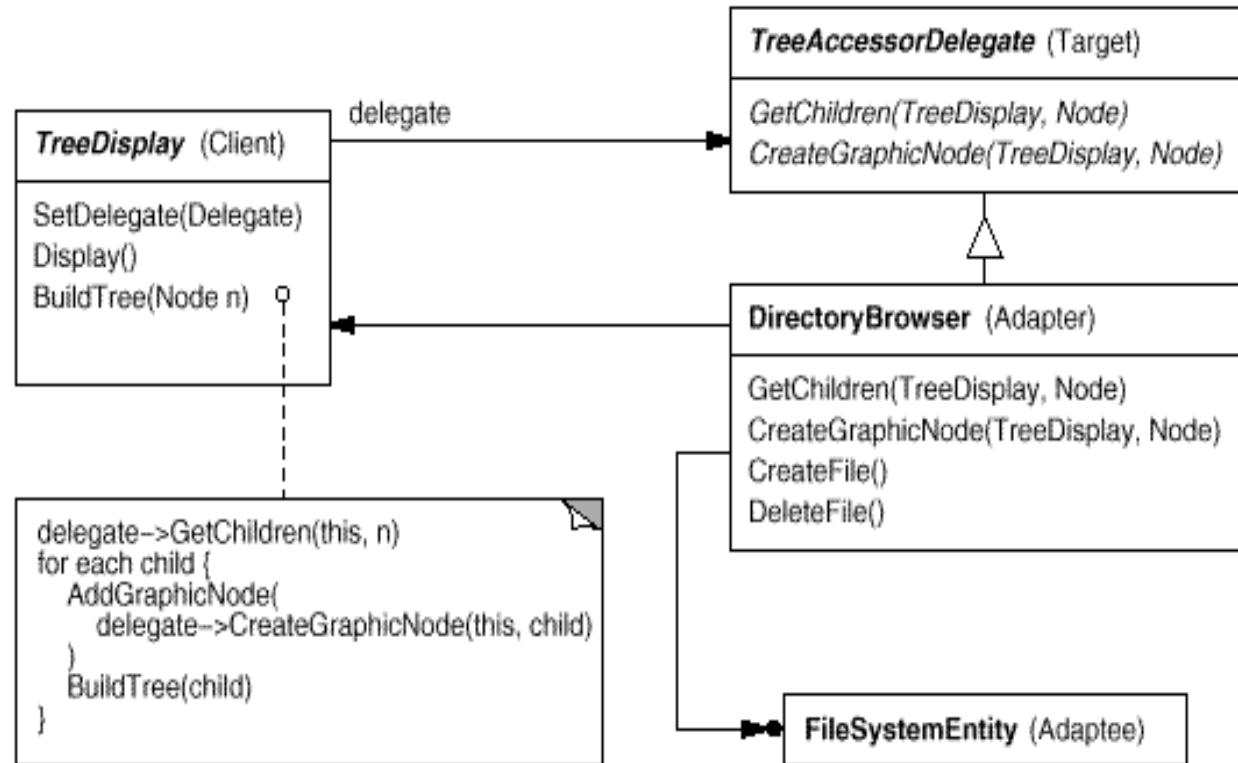
Other issues

- *How much adapting does Adapter do?*
- *Pluggable adapters.*
 - Using abstract operations



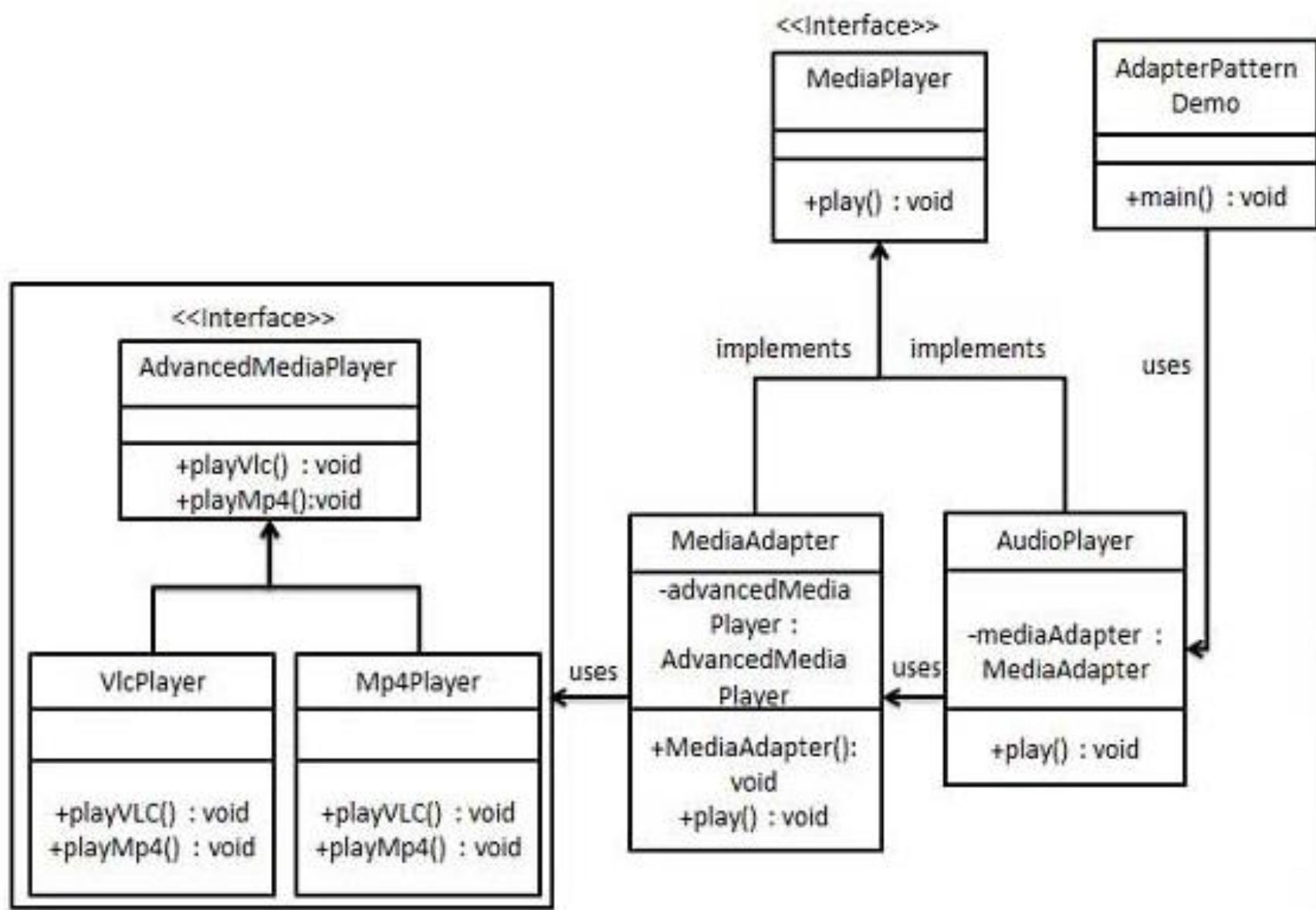
Pluggable adapters continued

- Using delegate objects



- Parameterized adapters

Another (bad!) Example

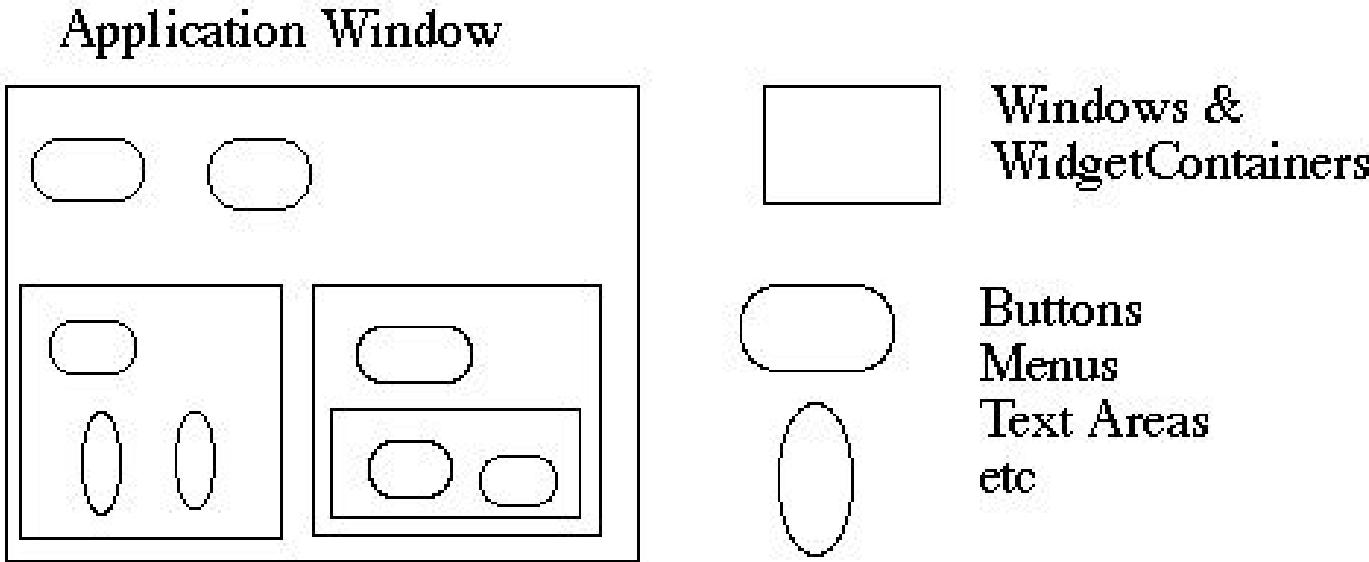


Principles Violations

- Interface Segregation
- “Commit to the interface, not the implementation”
- How would you design it?

COMPOSITE PATTERN

Motivation



- GUI Windows and GUI elements
 - How does the window hold and deal with the different items it has to manage?
 - Widgets are different from WidgetContainers

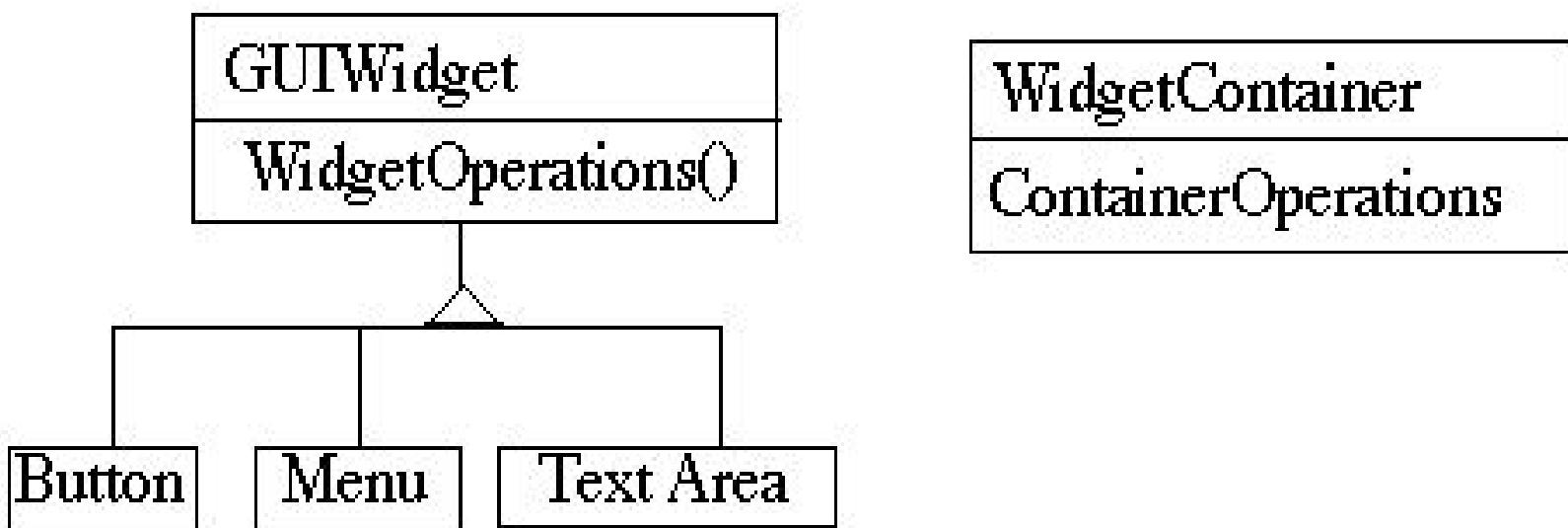
Implementation Ideas

- Nightmare Implementation

- for each operation deal with each category of objects individually
- no uniformity and no hiding of complexity
- a lot of code duplication

- Program to an Interface

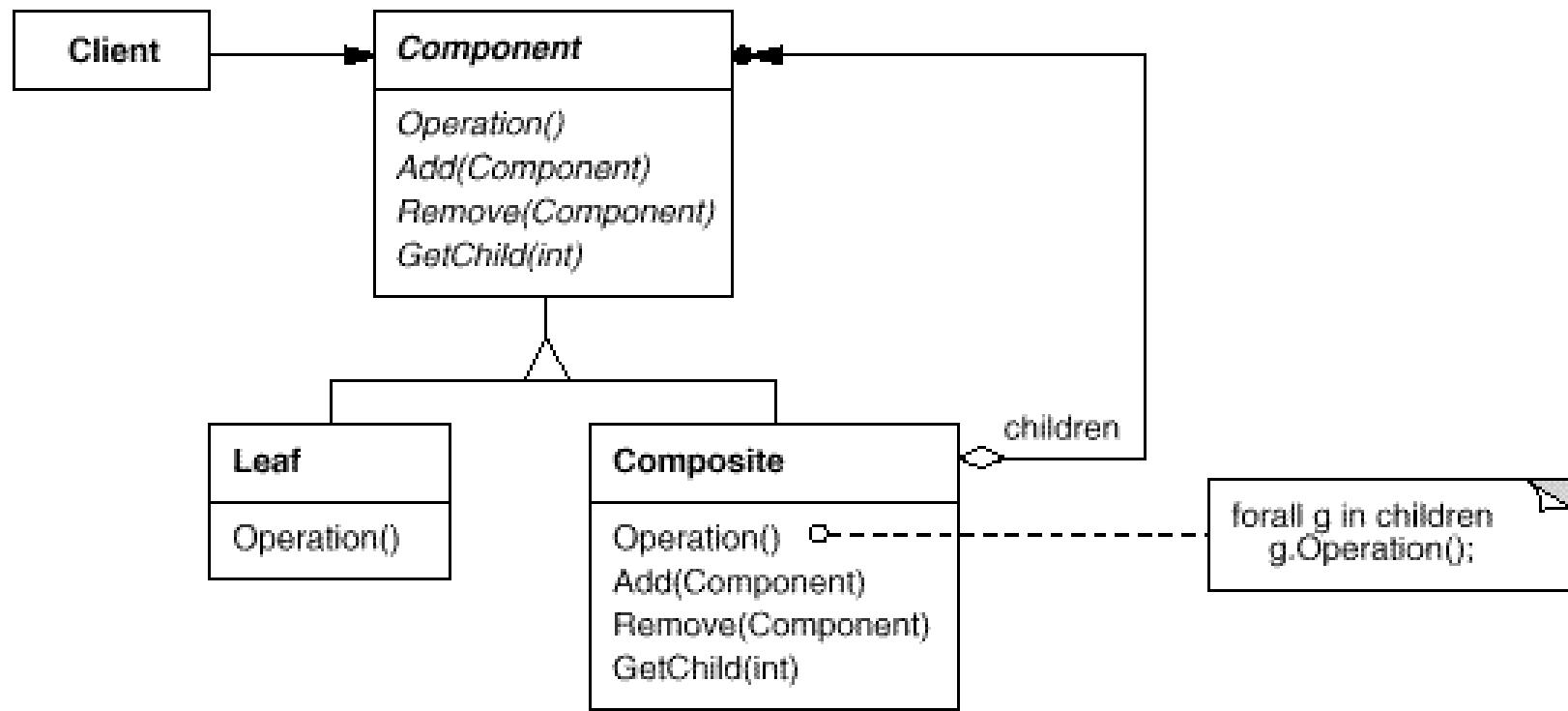
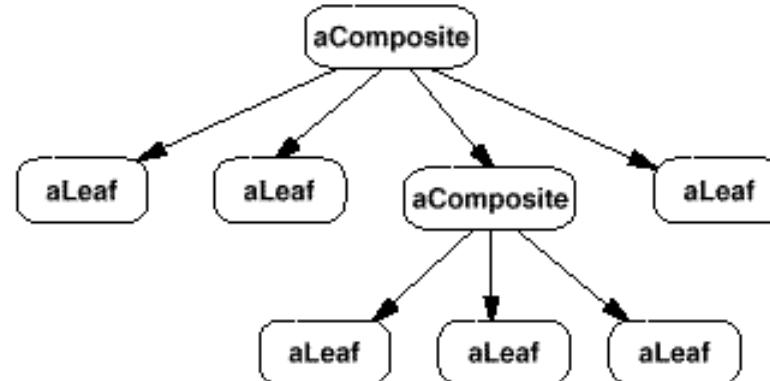
- uniform dealing with widget operations
- but still containers are treated different



Basic Aspects of Composite Pattern

- Intent
 - Treat individual objects and compositions of these object **uniformly**
 - Compose objects into tree-structures to represent recursive aggregations
- Applicability
 - represent part-whole hierarchies of objects
 - be able to ignore the difference between compositions of objects and individual objects

Structure



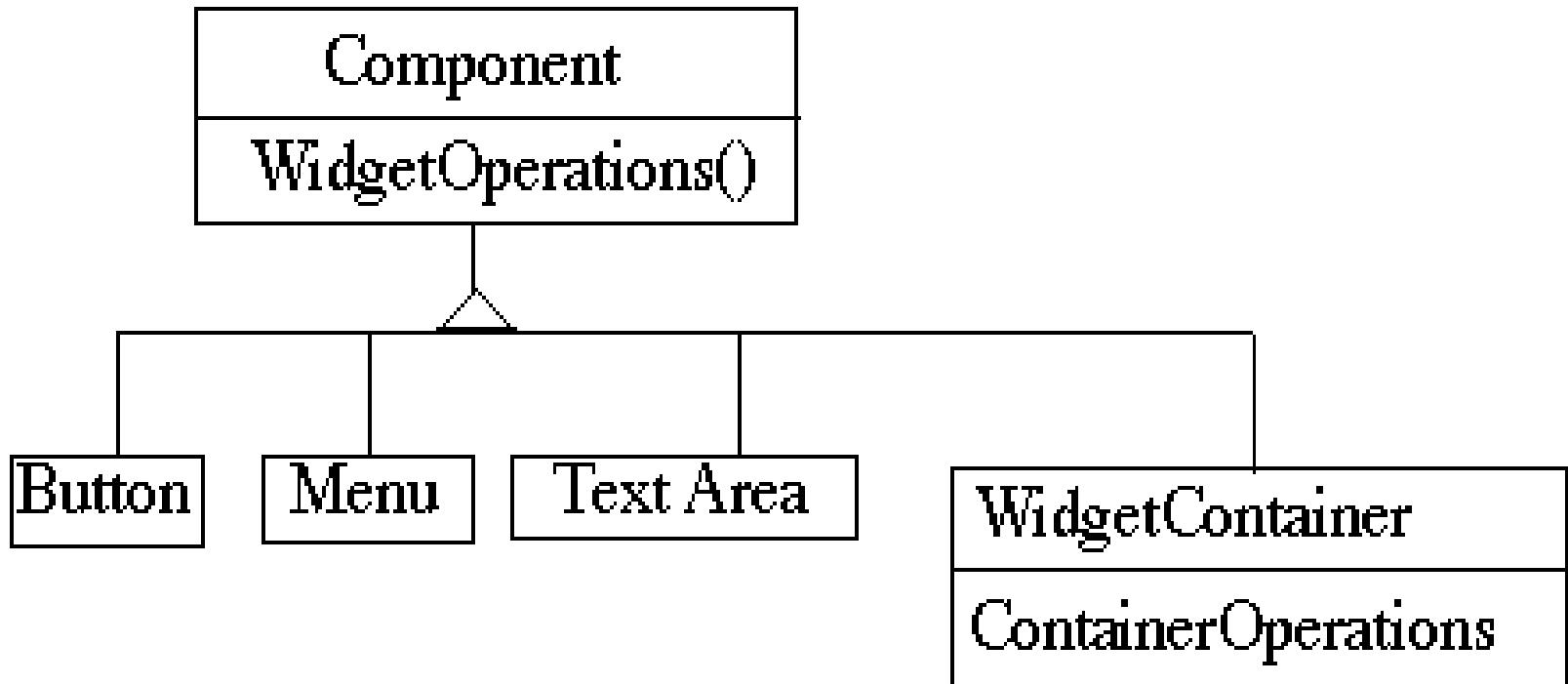
Participants & Collaborations

- Component
 - declares interface for objects in the composition
 - implements default behavior for components when possible
- Composite
 - defines behavior for components having children
 - stores child components
 - implement child-specific operations
- Leaf
 - defines behavior for primitive objects in the composition
- Client
 - manipulates objects in the composition through the Component interface

Consequences

- Defines uniform class hierarchies
 - recursive composition of objects
- Make clients simple
 - don't know whether dealing with a leaf or a composite
 - simplifies code because it avoids to deal in a different manner with each class
- Easier to extend
 - easy to add new Composite or Leaf classes
- **Design excessively general**
 - **type checks needed to restrict the types admitted in a particular composite structure**

Applying Composite to Widget Problem



- See [code](#)
 - Component implements default behavior when possible
 - Button, Menu, etc override Component methods when needed
 - WidgetContainer will have to override all widget operations

Where to place Container operations ?

- adding, deleting, managing components in composite
 - should they be placed in Component or in Composite?
- Pro-**Transparency** Approach
 - Declaring them in the Component gives all subclasses the same interface
 - All subclasses can be treated alike.
 - costs safety
 - clients may do stupid things like adding objects to leaves
 - [getComposite\(\)](#) to improve safety.
- Pro-**Safety** Approach
 - Declaring them in Composite is safer
 - Adding or removing widgets to non-WidgetContainers is an error

Other Implementation Issues

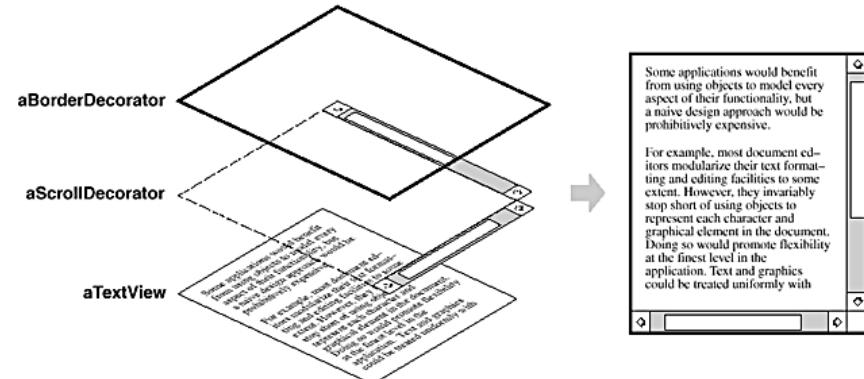
- Explicit container references
 - simplifies traversal
 - place it in Component
 - the consistency issue
 - change container reference **only** when add or remove component
- Component Ordering
 - consider using Iterator
- Who should delete components?
 - Composite should delete its children
- Caching to improve performance
 - cache information about components in containers

DECORATOR PATTERN

Changing the skin of an object

Motivation

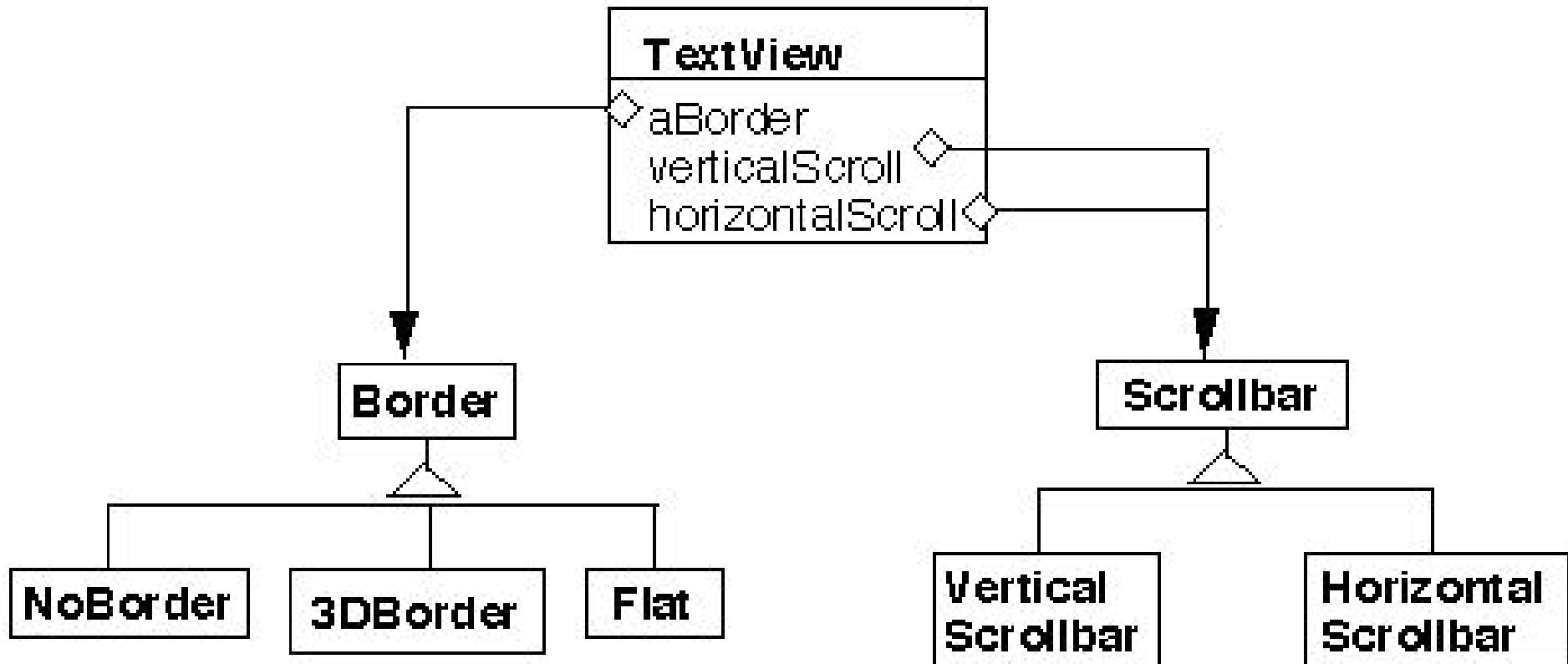
- A TextView has 2 features:
 - borders:
 - 3 options: none, flat, 3D
 - scroll-bars:
 - 4 options: none, side, bottom, both
- Inheritance => How many Classes?
 - $3 \times 4 = \text{12} !!!$
 - e.g. TextView, TextViewWithNoBorder&SideScrollbar,
TextViewWithNoBorder&BottomScrollbar,
TextViewWithNoBorder&Bottom&SideScrollbar,
TextViewWith3DBorder, TextViewWith3DBorder&SideScrollbar,
TextViewWith3DBorder&BottomScrollbar,
TextViewWith3DBorder&Bottom&SideScrollbar,



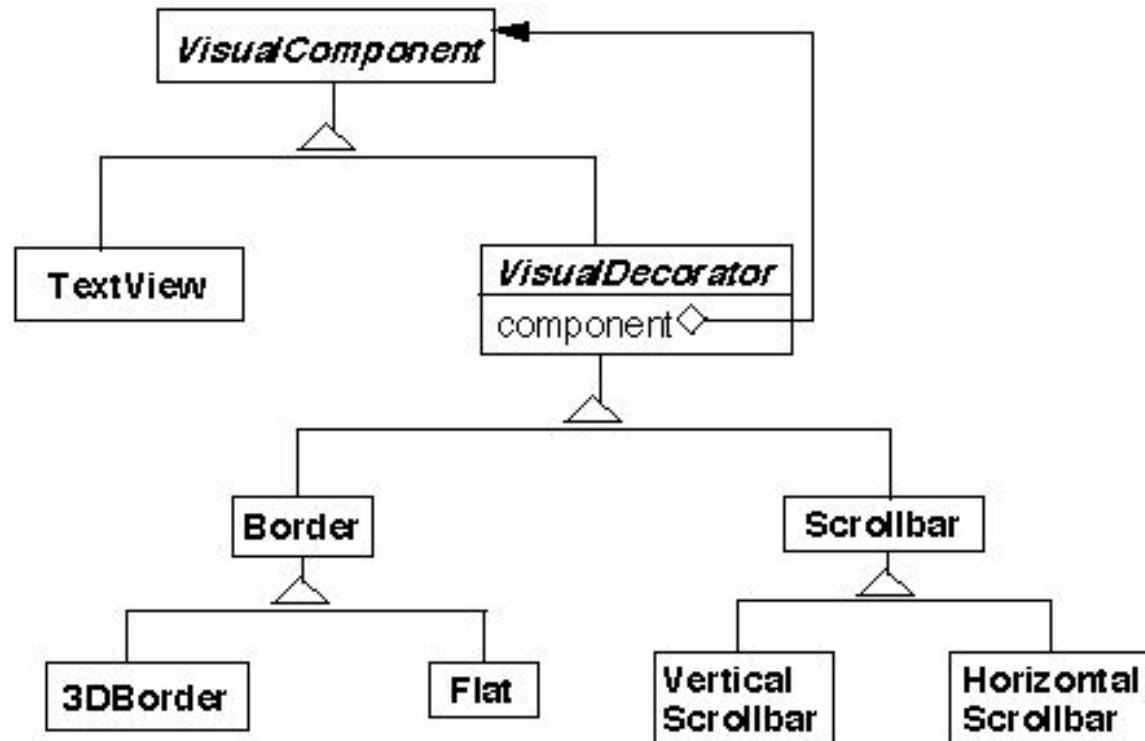
Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

Solution 1: Use Object Composition



Solution 2: Change the Skin, not the Guts!

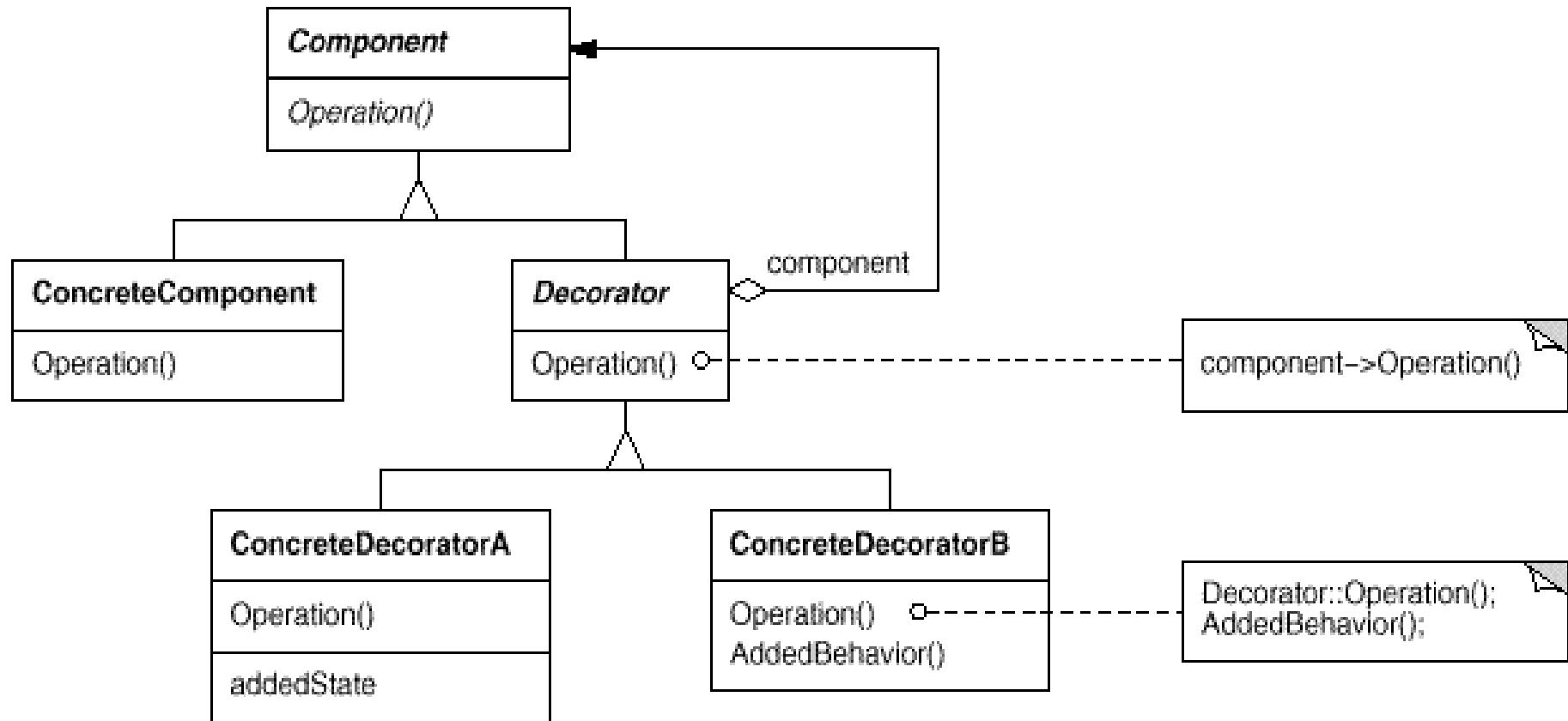


- *TextView* has **no** borders or scrollbars!
- Add borders and scrollbars **on top of** a *TextView*

Basic Aspects

- Intent
 - *Add responsibilities to a particular object rather than its class*
 - Attach additional responsibilities to an object dynamically.
 - Provide a flexible alternative to subclassing
- Also Known As
 - Wrapper
- Applicability
 - Add responsibilities to objects **transparently** and **dynamically**
 - i.e. without affecting other objects
 - Extension by subclassing is impractical
 - may lead to too many subclasses

Structure



Participants & Collaborations

- Component
 - defines the interface for objects that can have responsibilities added dynamically
- ConcreteComponent
 - the "bases" object to which additional responsibilities can be added
- Decorator
 - defines an interface conformant to Component's interface
 - for transparency
 - maintains a reference to a Component object
- ConcreteDecorator
 - adds responsibilities to the component

Consequences

- More flexibility than static inheritance
 - allows to mix and match responsibilities
 - allows to apply a property twice
- Avoid feature-laden classes high-up in the hierarchy
 - "*pay-as-you-go*" approach
 - easy to define new types of decorations
- Lots of little objects
 - easy to customize, but hard to learn and debug
- A decorator and its component aren't identical
 - checking object identification can cause problems
 - e.g. **if (aComponent instanceof TextView)**

Implementation Issues

- Keep Decorators lightweight
 - Don't put data members in Component
 - use it for shaping the interface
- Omitting the abstract Decorator class
 - if only one decoration is needed
 - subclasses may pay for what they don't need

CHANGING THE GUTS

Changing the "Guts" of an Object ...

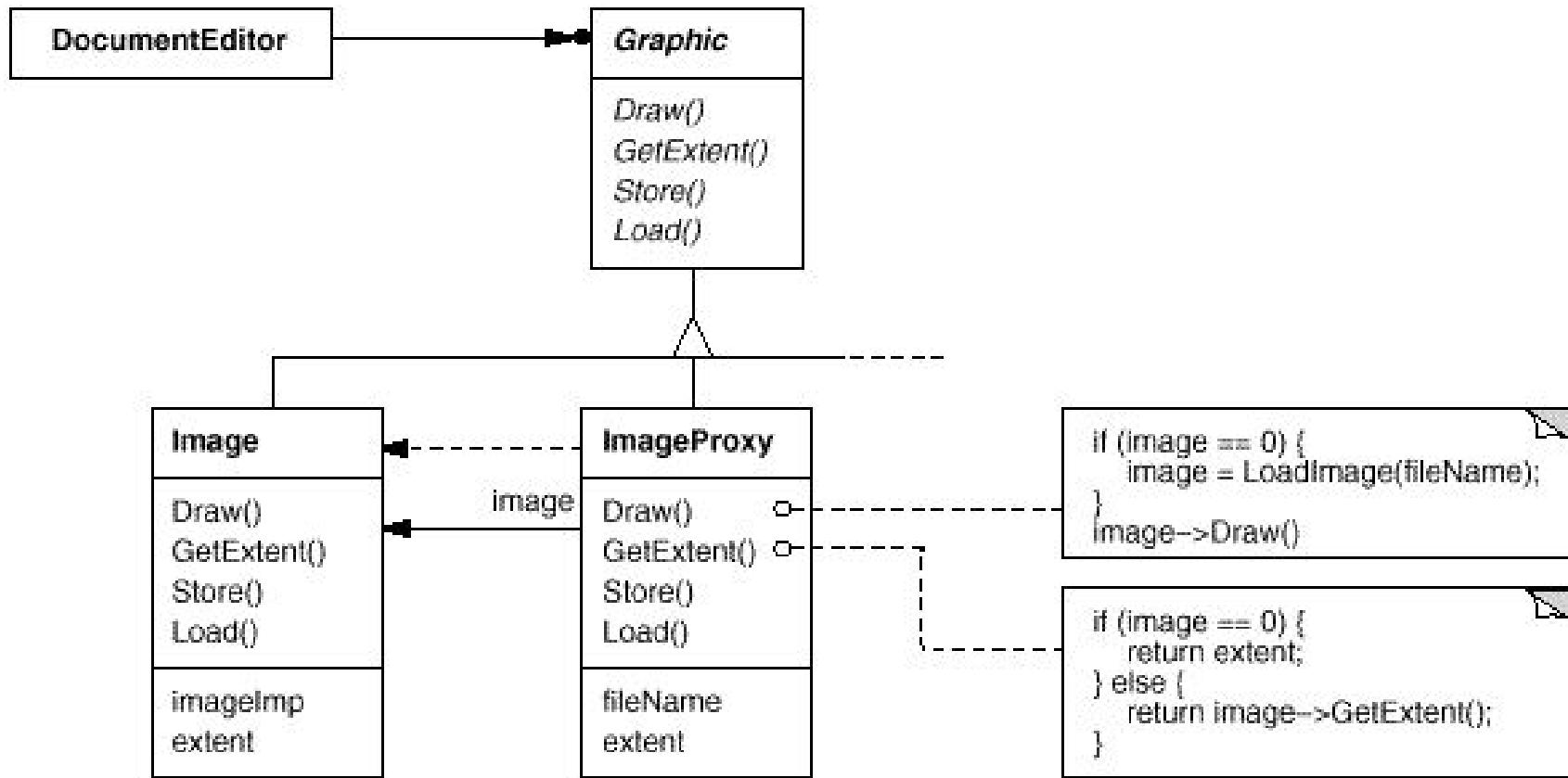
- Optimize
 - use an alternative algorithm to implement behavior (**Strategy**)
- Alter
 - change behavior when object's state changes (**State**)
- Control
 - "shield" the implementation from direct access (**Proxy**)
- Decouple
 - let abstraction and implementation vary independently (**Bridge**)

PROXY PATTERN

Loading "Heavy" Objects

- Document Editor that can embed multimedia objects
 - MM objects are expensive to create \Rightarrow opening of document slow
 - avoid creating expensive objects
 - they are not all necessary as they are not all visible at the same time
- Creating each expensive object **on demand !**
 - i.e. when image has to be displayed
- What should we put instead?
 - hide the fact that we are "lazy"!
 - don't complicate the document editor!

Idea: Use a Placeholder!



- create only when needed for drawing
- keeps information about the dimensions (extent)

Basic Aspects

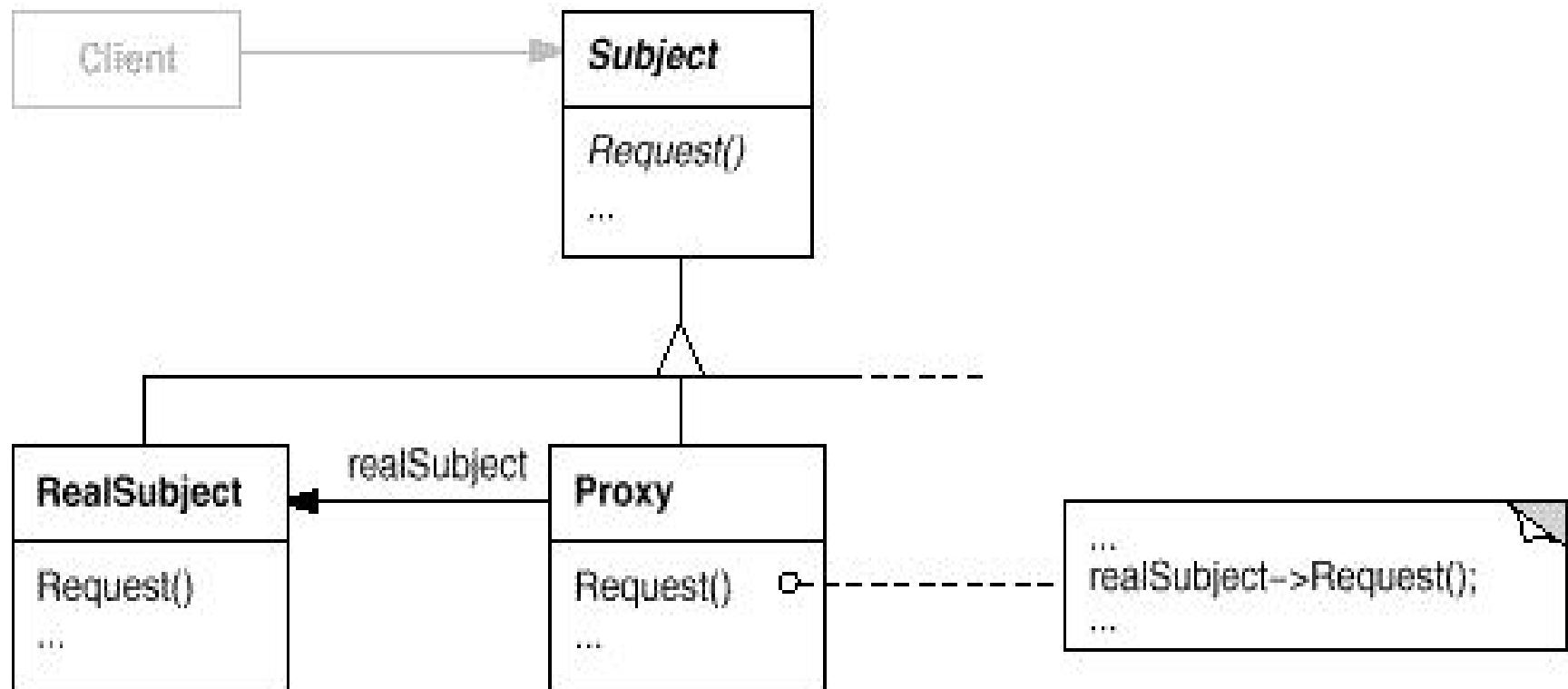
- Definition

proxy (n. pl prox-ies) The agency for a person who acts as a substitute for another person, authority to act for another

- Intent

- provide a surrogate or placeholder for another object to control access to it
- Applicability: whenever there is a need for a more *flexible* or *sophisticated* reference to an object than a simple pointer
 - remote proxy ... if real object is “**far away**”
 - virtual proxy ... if real object is “**expensive**”
 - protection proxy ... if real object is “**vulnerable**”
 - enhancement proxies (**smart pointers**)
 - prevent accidental delete of objects (counts references)

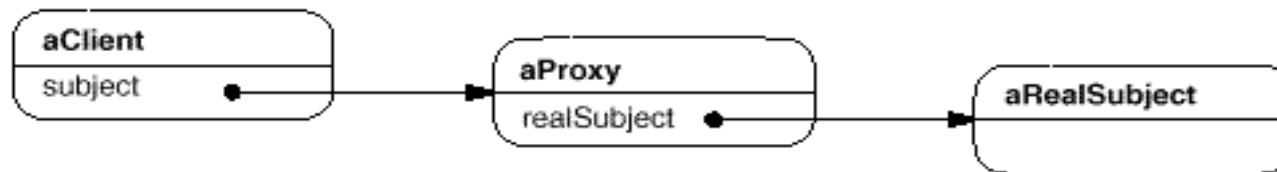
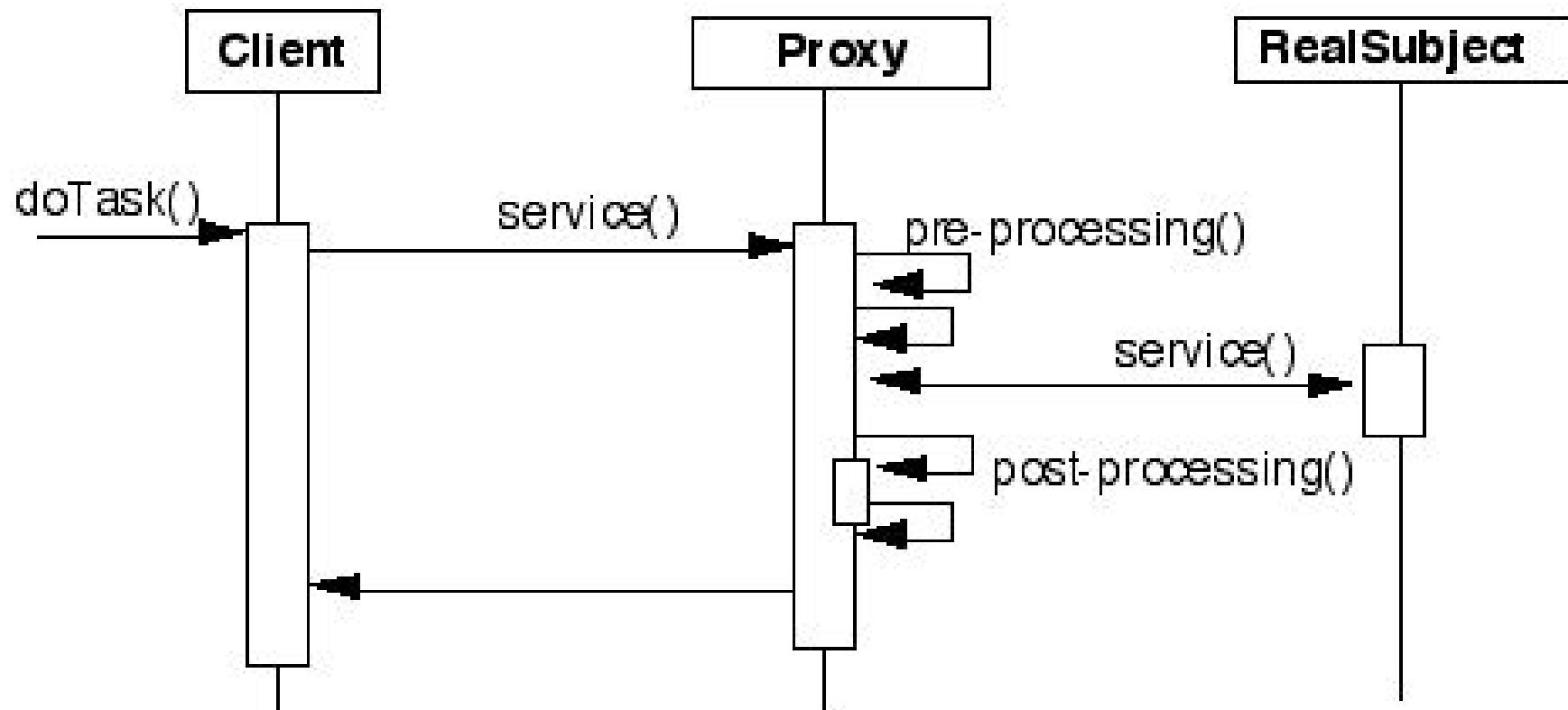
Structure



Participants

- Proxy
 - maintains a reference that lets the proxy access the real subject.
 - provides an interface identical to Subject's
 - so that proxy can be substituted for the real subject
 - controls access to the real subject
 - may be responsible for creating or deleting it
- Subject
 - defines the common interface for RealSubject and Proxy
- RealSubject
 - defines the real object that the proxy holds place for

Collaborations

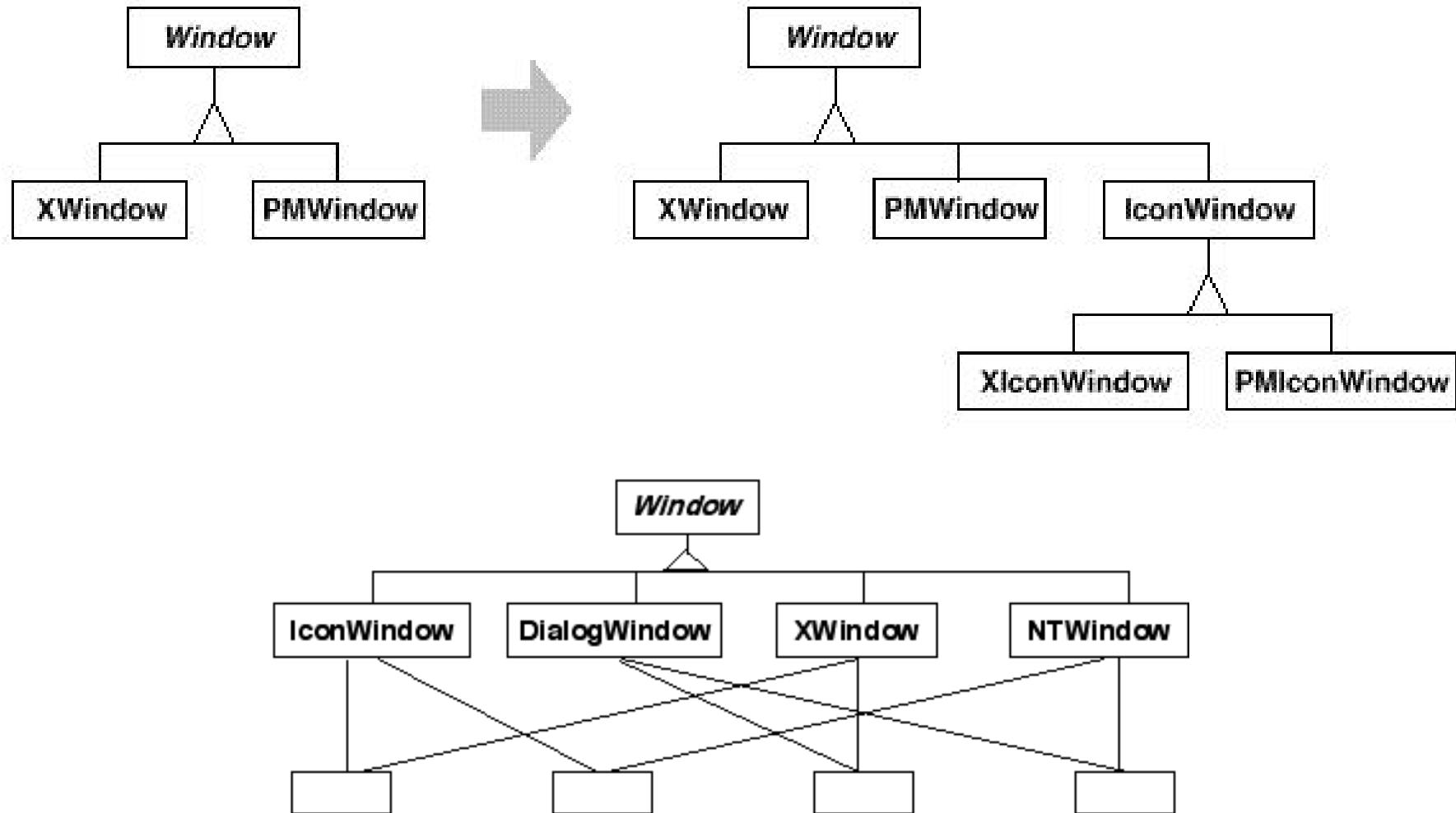


Consequences

- Proxies introduce a level of indirection
 - used differently depending on the kind of proxy:
 - hide different address space (remote p.)
 - creation on demand (virtual p.)
 - allow additional housekeeping activities (protection, smart pointers)

BRIDGE PATTERN

Inheritance that Leads to Explosion!



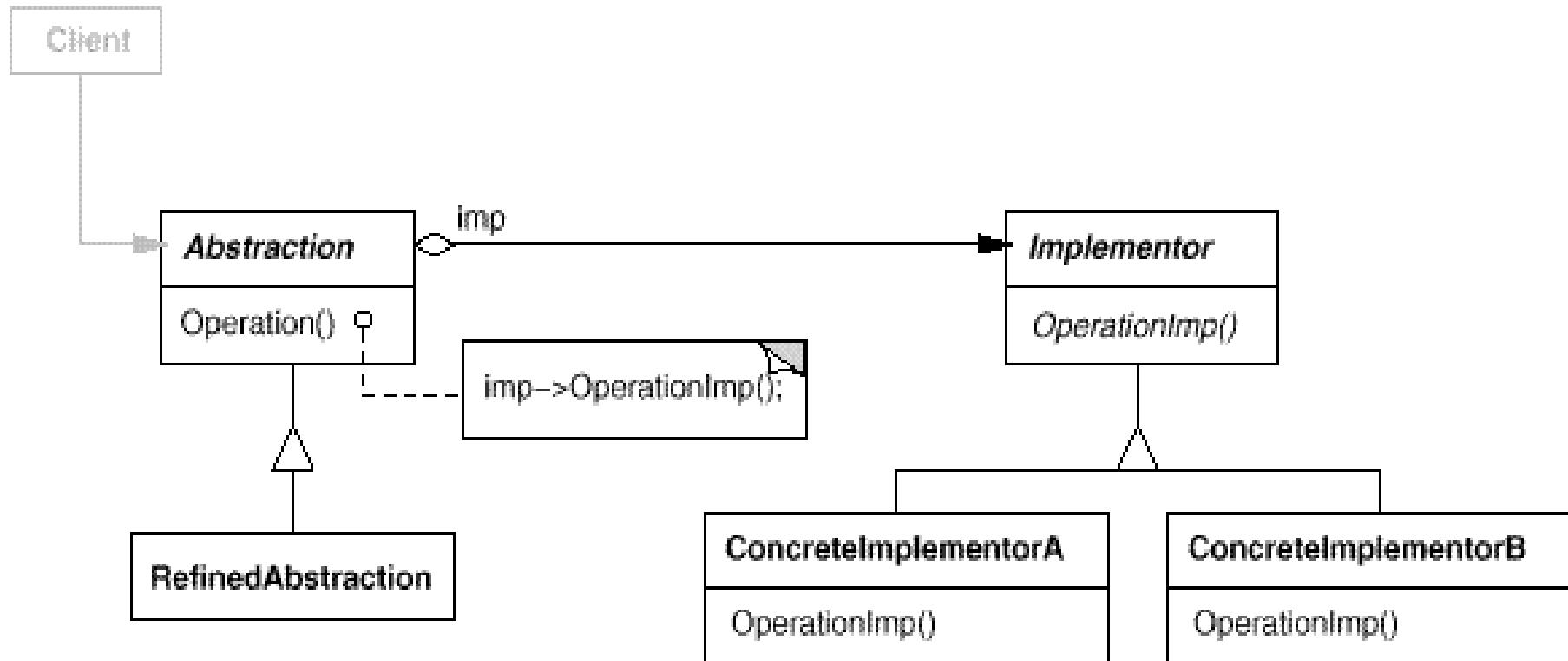
Basic Aspects of Bridge Pattern

- Intent
 - decouple an abstraction from its implementation
 - allow implementation to vary independently from its abstraction
 - abstraction defines and implements the interface
 - all operations in abstraction call methods from its implementation obj.
- In the Bridge pattern ...
 - ... an abstraction can use different implementations
 - ... an implementation can be used in different abstractions

Applicability

- Avoid permanent binding btw. an abstraction and its implementation
- Abstractions and their implementations should be *independently extensible* by subclassing
- Hide the implementation of an abstraction completely from clients
 - their code should not have to be recompiled when impl. changes
- Share an implementation among multiple objects
 - and this fact should be hidden from the client

Structure



Participants

- Abstraction
 - defines the abstraction's interface
 - maintains a reference to an object of type Implementor
- Implementor
 - defines the interface for implementation classes
 - does not necessarily correspond to the Abstraction's interface
 - Implementor contains primitive operations,
 - Abstraction defines the higher-level operations based on these primitives
- RefinedAbstraction
 - extends the interface defines by Abstraction
- ConcreteImplementer
 - implements the Implementor interface, defining a concrete impl.

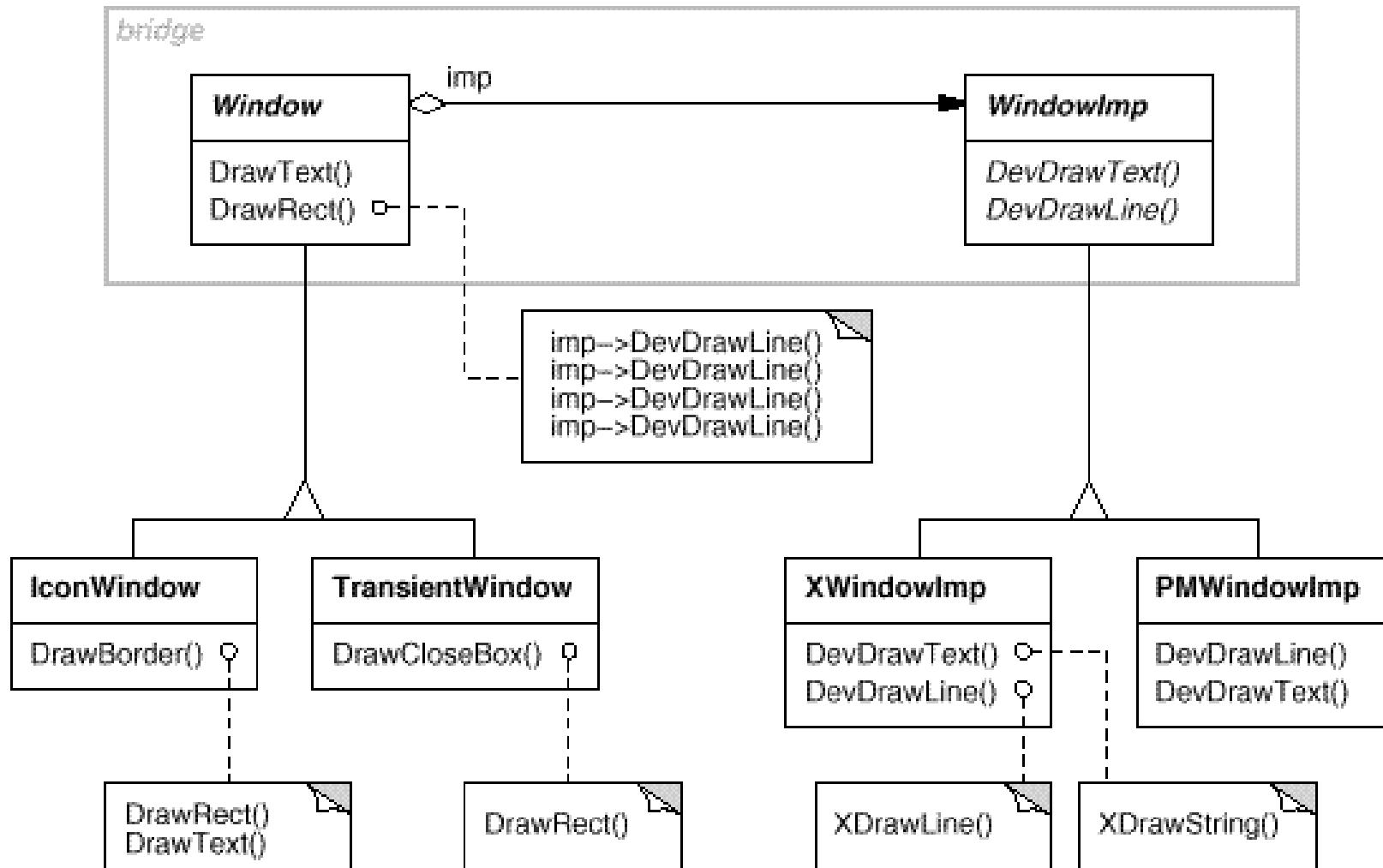
Consequences

- Decoupling interface and implementation
 - implem. **configurable** and **changeable** at run-time
 - reduce compile-time dependencies
 - implement. changes do not require Abstraction to recompile
- Improved extensibility
 - extend by subclassing independently Abstractions and Implementations
- Hiding implementation details from clients
 - shield clients from implementations details
 - e.g. sharing implementor objects together with reference counting

Implementation

- Only one Implementor
 - not necessary to create an abstract implementor class
 - degenerate, but useful due to decoupling
- Which Implementor should I use ?
 - Variant 1: let Abstraction know all concrete implem. and choose
 - Variant 2: choose initially default implem. and change later
 - Variant 3: use an Abstract Factory
 - no coupling btw. Abstraction and concrete implem. classes

Windows Example Revisited



Adapter vs. Bridge

- Common features
 - promote flexibility by providing a level of indirection to another object.
 - involve forwarding requests to this object from an interface other than its own.
- Differences
 - Adapter **resolves incompatibilities** between two existing interfaces. It doesn't focus on how those interfaces are implemented, nor does it consider how they might evolve independently
 - Bridge **links an abstraction and its implementations**. It provides a stable interface to clients even as it lets you vary the classes that implement it.
 - **The Adapter pattern makes things work after they're designed; Bridge makes them work before they are.**

Next time

- Even more patterns...

SOFTWARE DESIGN

Design Patterns 3

Content

- Design Patterns
 - Creational Patterns
 - Structural Patterns
 - Behavioral Patterns
 - Chain of Responsibility
 - Command
 - Observer
 - State
 - Strategy

References

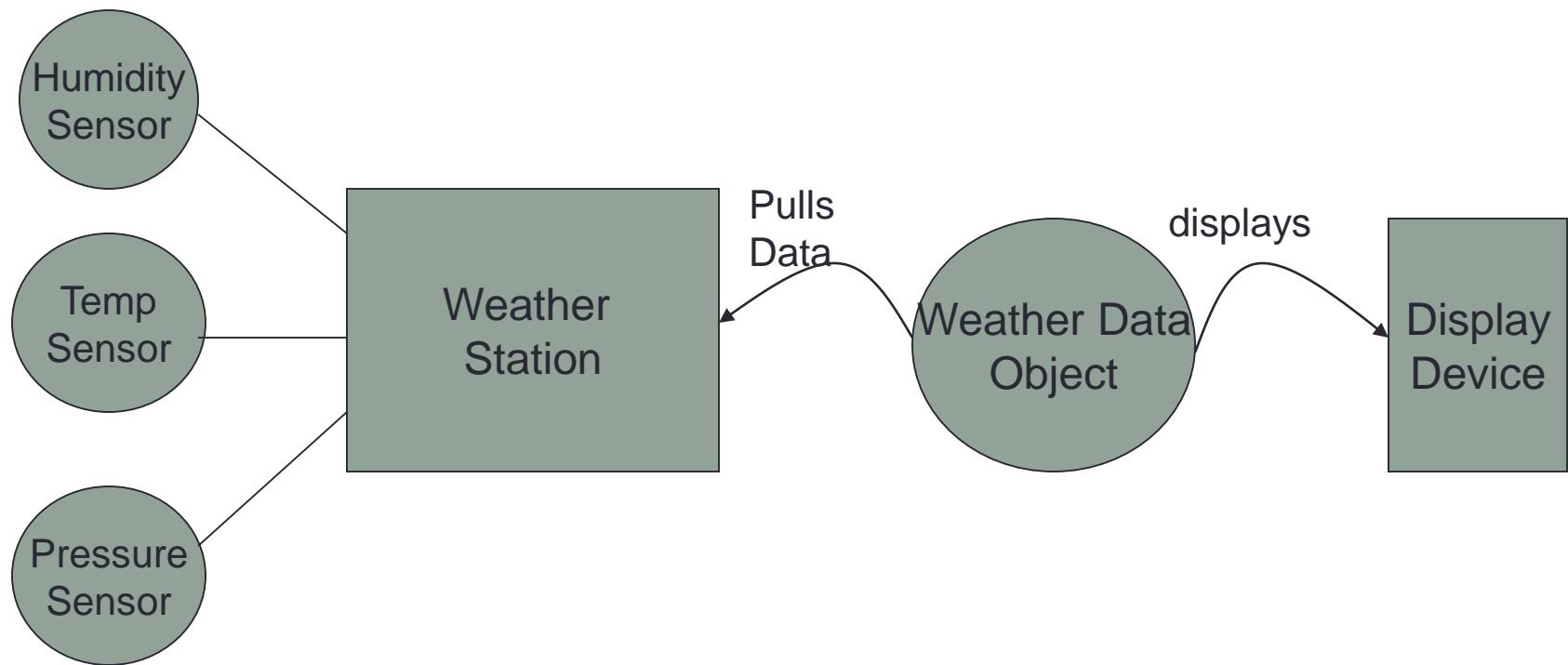
- Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.
- Univ. of Timisoara Course materials

Behavioral patterns

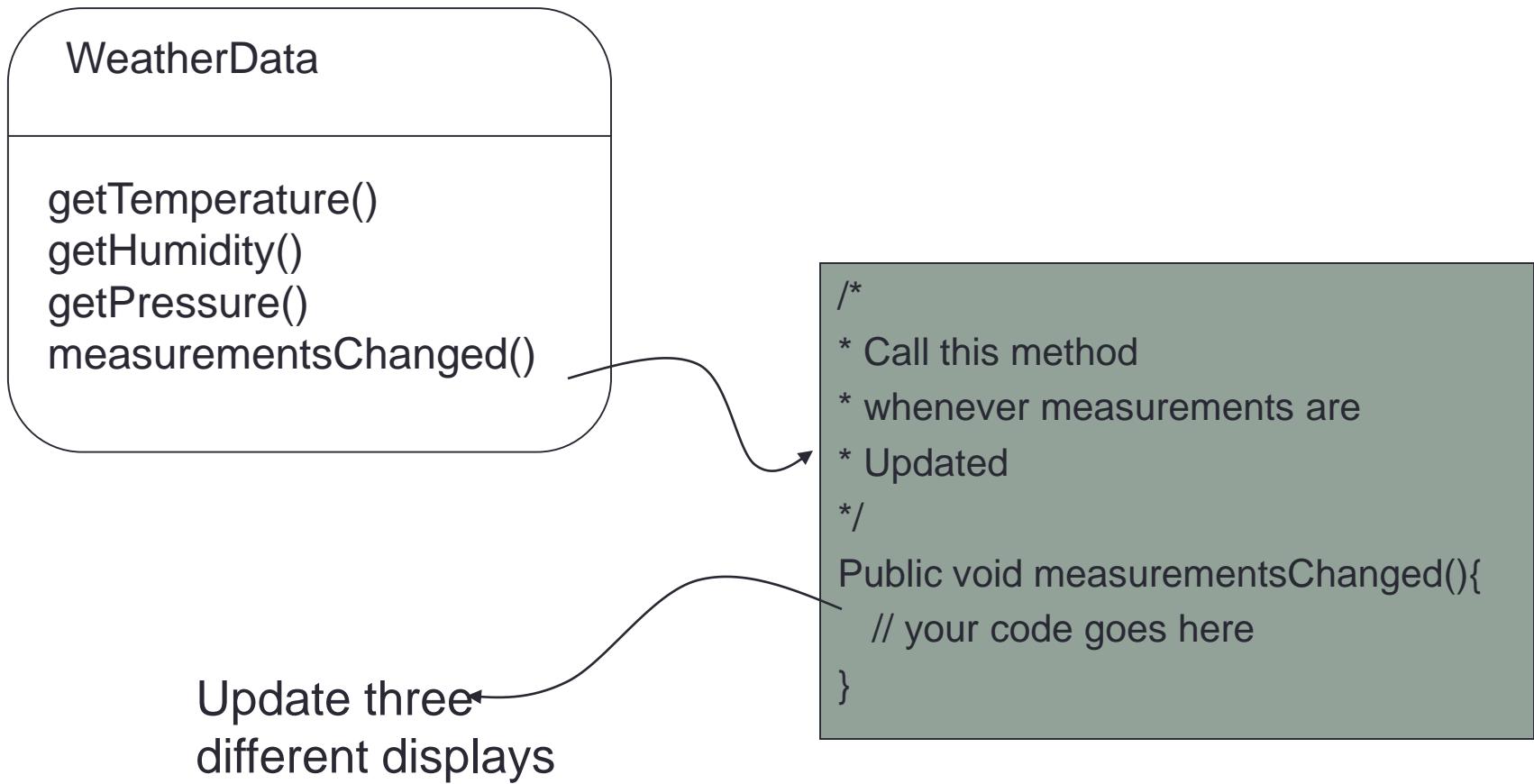
- Behavioral patterns are concerned with **algorithms** and the **assignment of responsibilities** between objects.
- Behavioral patterns describe not just patterns of objects or classes but also the **patterns of communication** between them.
- Behavioral **class** patterns use inheritance to distribute behavior between classes.
- Behavioral **object** patterns use object composition rather than inheritance.

OBSERVER PATTERN

Weather Monitoring Application



What needs to be done?



Problem specification

- WeatherData class has three getter methods
- measurementsChanged() method called whenever there is a change
- Three display methods needs to be supported: current conditions, weather statistics and simple forecast
- System should be expandable

First cut at implementation

```
public class WeatherData {  
  
    public void measurementsChanged(){  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        currentConditionsDisplay.update (temp, humidity, pressure);  
        statisticsDisplay.update (temp, humidity, pressure);  
        forecastDisplay.update (temp, humidity, pressure);  
    }  
    // other methods  
}
```

Area of change which can be managed better by encapsulation

By coding to concrete implementations there is no way to add additional display elements without making code change

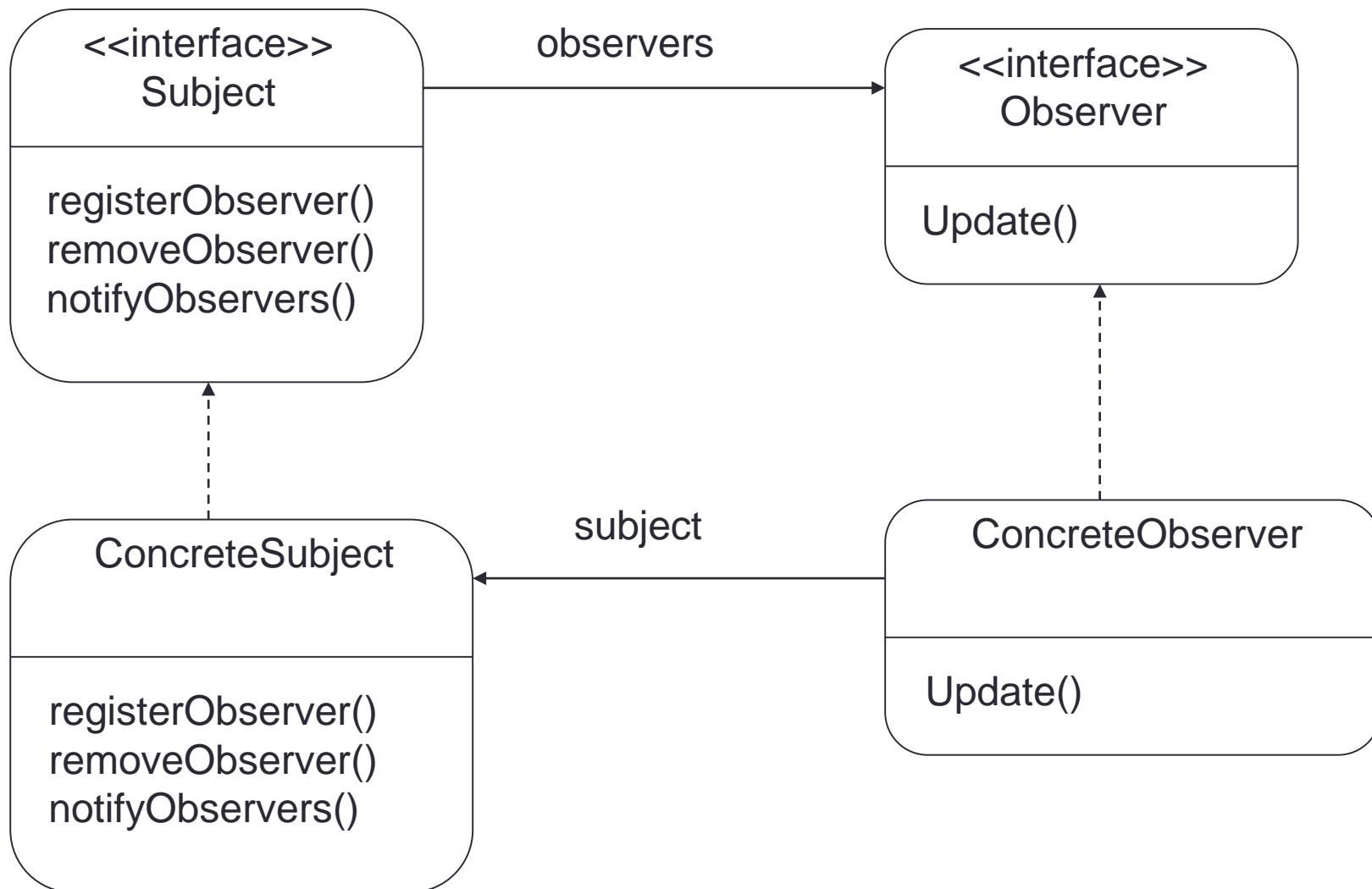
Basis for observer pattern

- Fashioned after the publish/subscribe model
- Works off similar to any subscription model
 - Buying newspaper
 - Magazines
 - List servers

Observer Pattern Defined

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

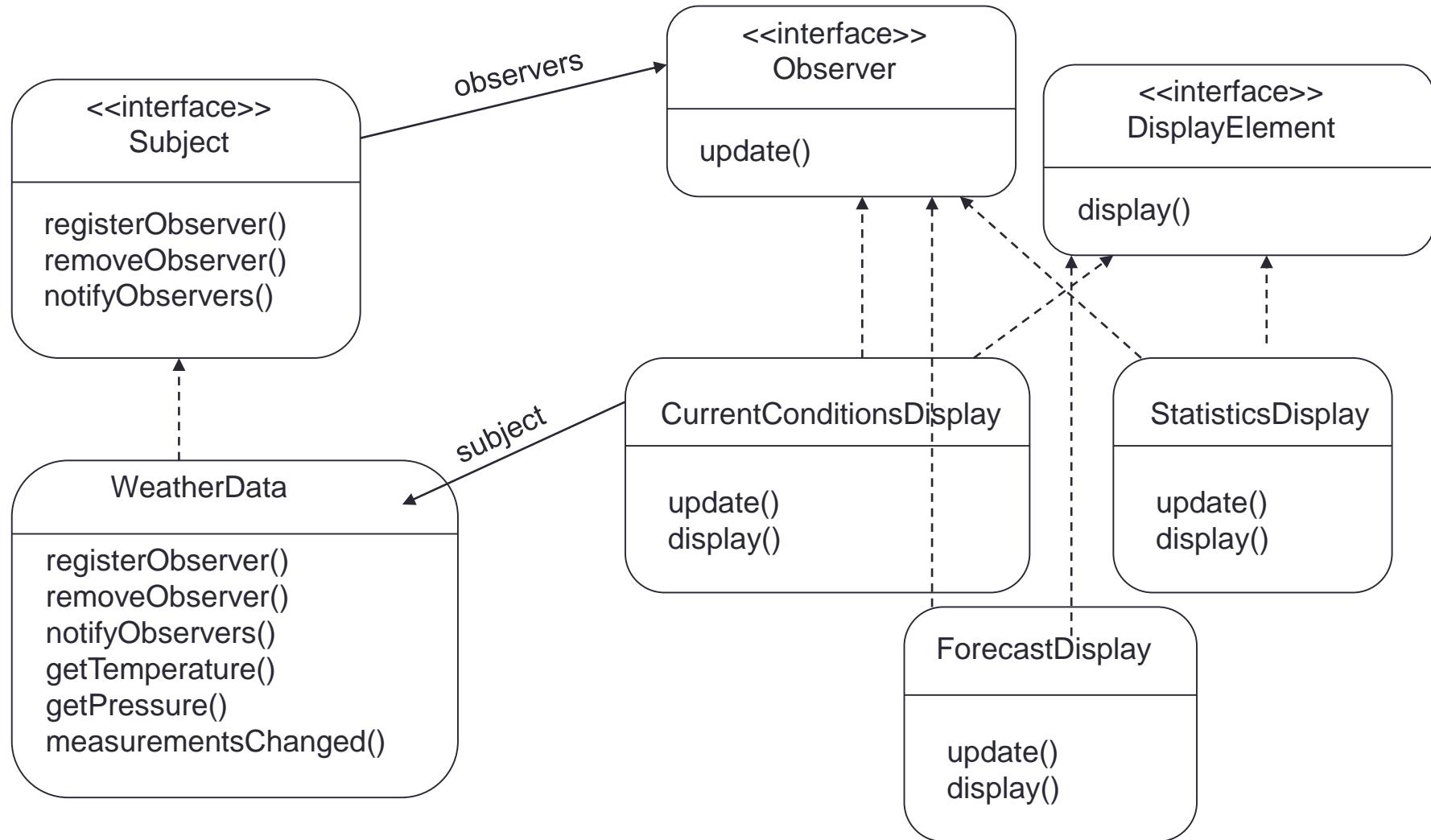
Observer Pattern – Class diagram



Observer pattern – power of loose coupling

- The only thing that the subject knows about an observer is that it implements an interface
- Observers can be added at any time and subject need not be modified to add observers
- Subjects and observers can be reused or modified without impacting the other [as long as they honor the interface commitments]

Observer Pattern – Weather data



Weather data interfaces

```
public interface Subject {  
    public void registerObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
}  
  
public interface Observer {  
    public void update(float temp, float humidity, float  
        pressure);  
}  
  
public interface DisplayElement {  
    public void display();  
}
```

Implementing subject interface

```
public class WeatherData implements Subject {  
    private ArrayList observers;  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherData() {  
        observers = new ArrayList();  
    }
```

Register and unregister

```
public void registerObserver(Observer o) {  
    observers.add(o);  
}  
  
public void removeObserver(Observer o) {  
    int i = observers.indexOf(o);  
    if (i >= 0) {  
        observers.remove(i);  
    }  
}
```

Notify methods

```
public void notifyObservers() {  
    for (int i = 0; i < observers.size(); i++) {  
        Observer observer = (Observer)observers.get(i);  
        observer.update(temperature, humidity,  
pressure);  
    }  
}  
  
public void measurementsChanged() {  
    notifyObservers();  
}
```

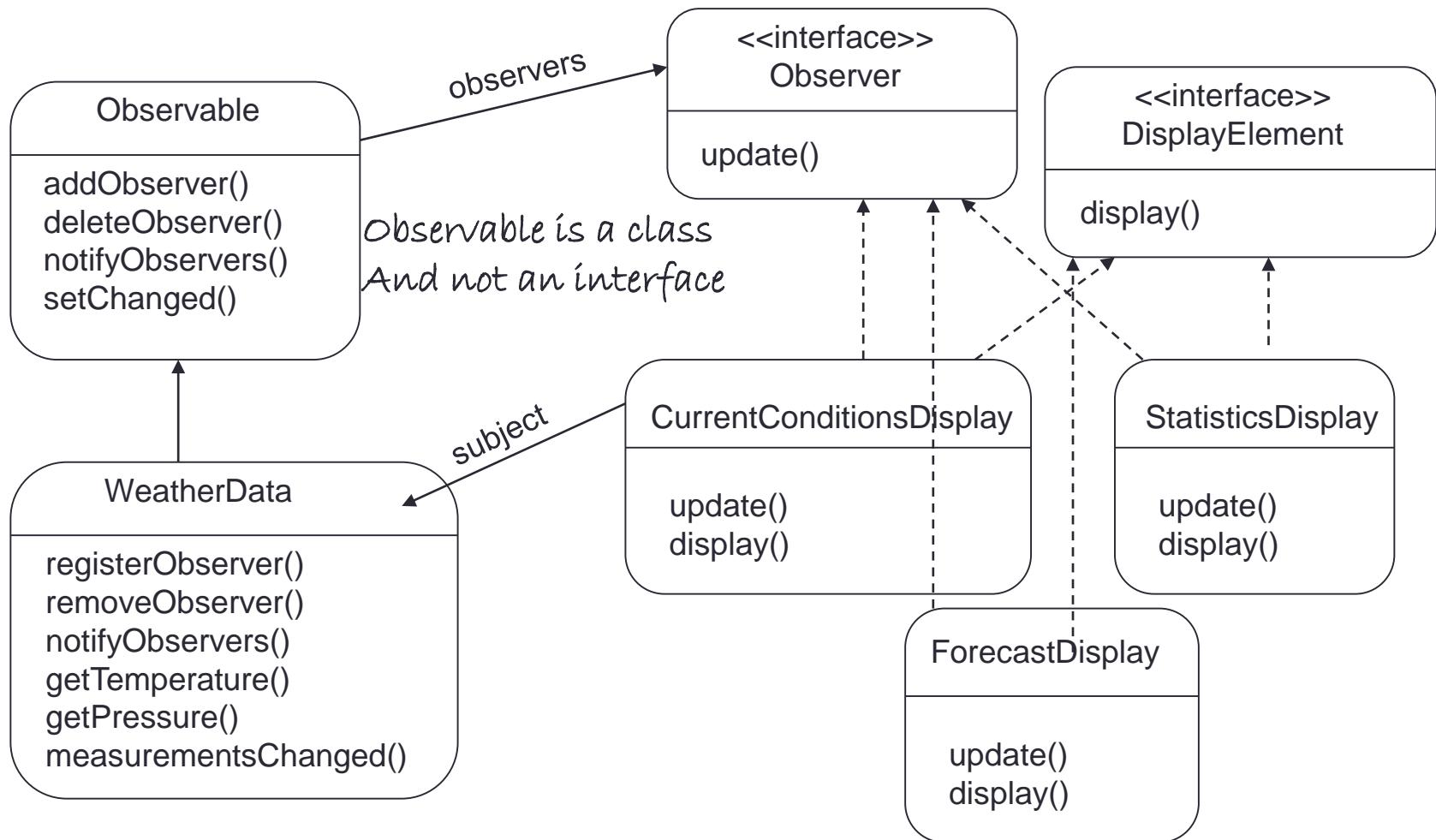
OBSERVER PATTERN

More analysis

Push or pull

- The notification approach used so far pushes all the state to all the observers
- One can also just send a notification that some thing has changed and let the observers pull the state information
- Java observer pattern support has built in support for both push and pull in notification
 - **java.util.Observable**
 - **java.util.Observer**

Java Observer Pattern – Weather data



Problems with Java implementation

- Observable is a class
 - You have to subclass it
 - You cannot add observable behavior to an existing class that already extends another superclass
 - *You have to program to an implementation – not interface*
- Observable protects crucial methods
 - Methods such as setChanged() are protected and not accessible unless one subclasses Observable.
 - *You cannot favor composition over inheritance.*
- You may have to roll your own observer interface if Java utilities don't work for your application

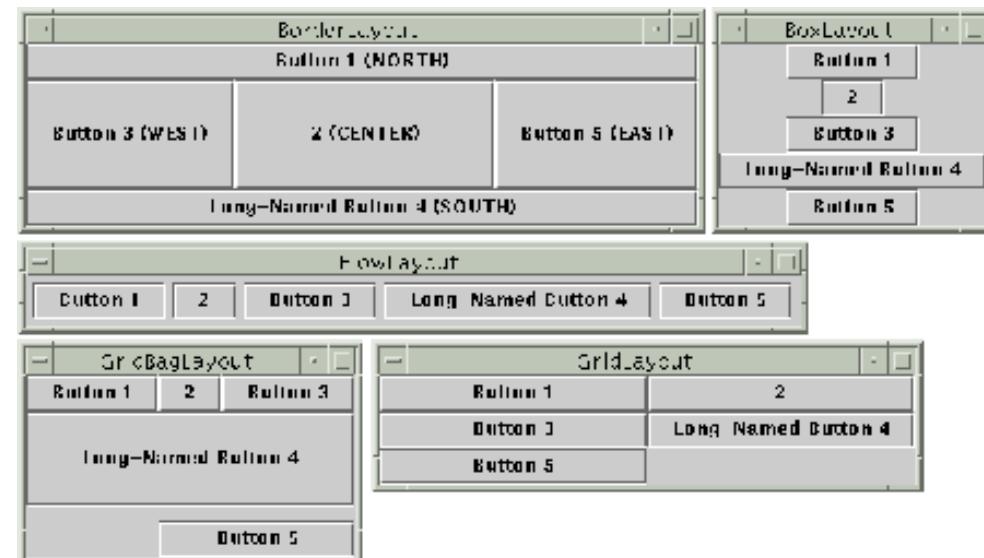
Changing the "Guts" of an Object ...

- Optimize
 - use an alternative algorithm to implement behavior (**Strategy**)
- Alter
 - change behavior when object's state changes (**State**)
- Control
 - "shield" the implementation from direct access (**Proxy**)
- Decouple
 - let abstraction and implementation vary independently (**Bridge**)

STRATEGY PATTERN

Java Layout Managers

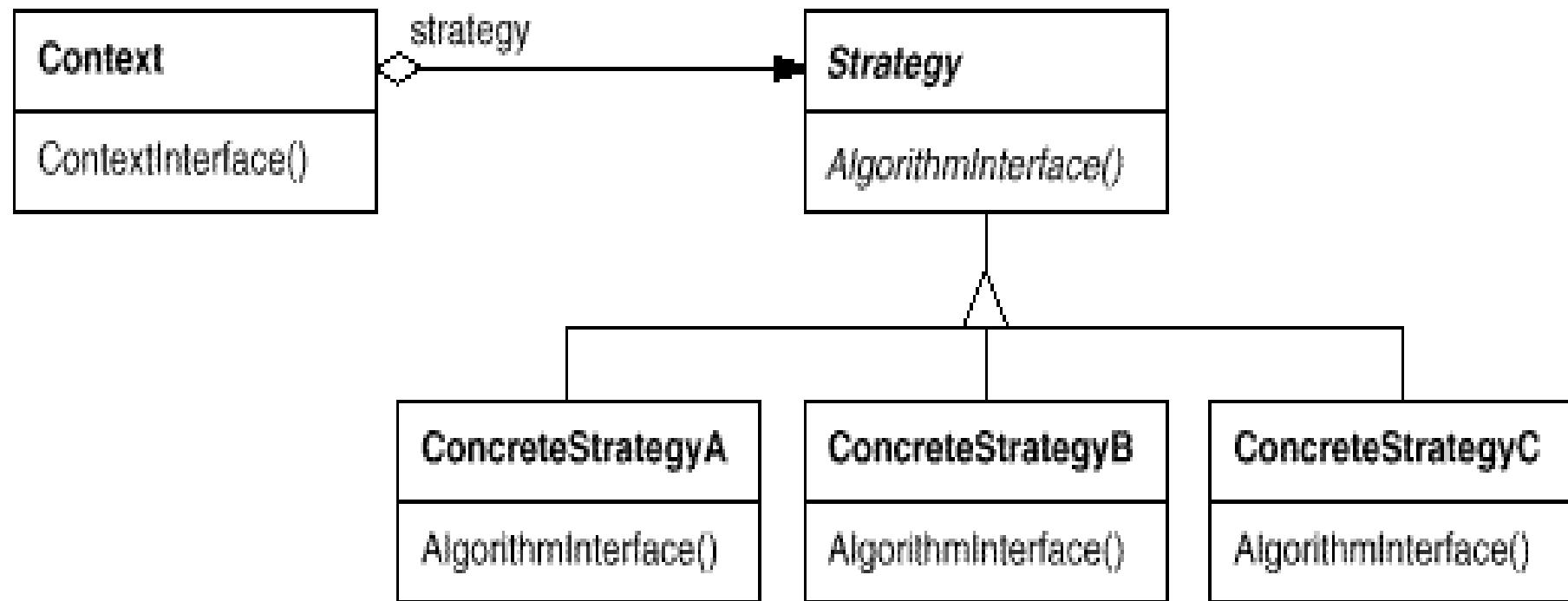
- GUI container classes in Java
 - frames, dialogs, applets (top-level)
 - panels (intermediate)
- Each container class has a layout manager
 - determine the size and position of components
 - 20 types of layouts
 - ~40 container-types
 - imagine to combine them freely by inheritance
- Consider also sorting...
 - open-ended number of sorting criteria



Basic Aspects

- Intent
 - Define a family of algorithms, encapsulate each one, and make them interchangeable
 - Let the algorithm vary independently from clients that use it
- Applicability
 - You need different variants of an algorithm
 - An algorithm uses data that clients shouldn't know about
 - avoid exposing complex, algorithm-specific data structures
 - Many related classes differ only in their behavior
 - configure a class with a particular behavior

Structure



Participants

- Strategy
 - declares an interface common to all supported algorithms.
 - Context uses this interface to call the algorithm defined by a ConcreteStrategy

- ConcreteStrategy
 - implements the algorithm using the Strategy interface

- Context
 - configured with a ConcreteStrategy object
 - may define an interface that lets Strategy objects to access its data

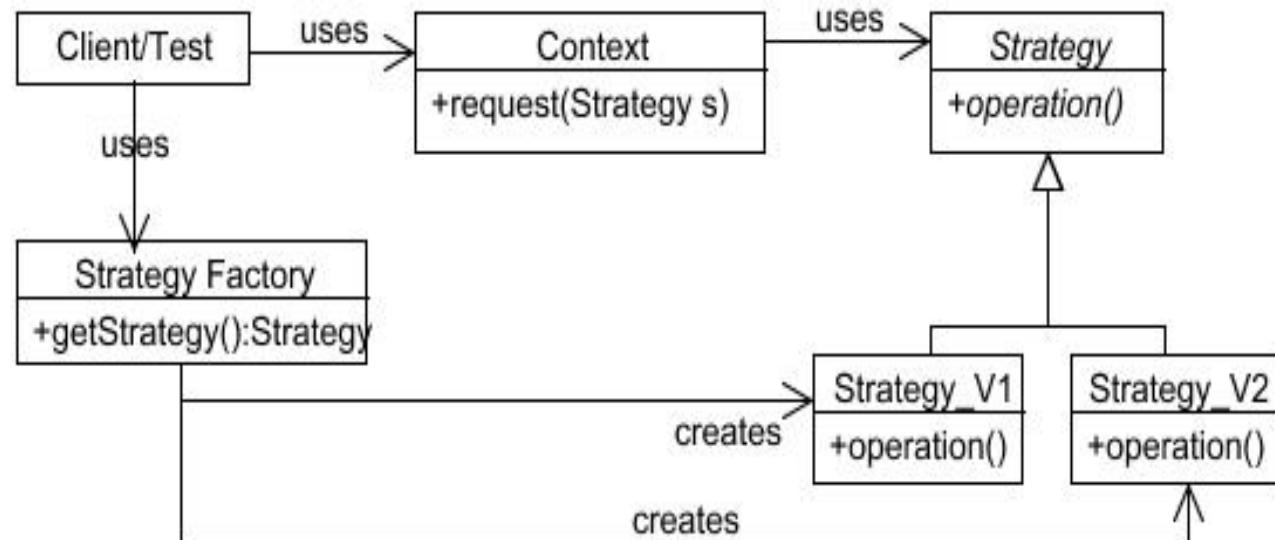
Consequences

- Families of related algorithms
 - usually provide different implementations of the same behavior
 - choice decided by time vs. space trade-offs
- Alternative to subclassing
 - see examples with layout managers (Strategy solution, [here](#))
 - We still subclass the strategies...
- Eliminates conditional statements
 - many conditional statements → "invitation" to apply Strategy!

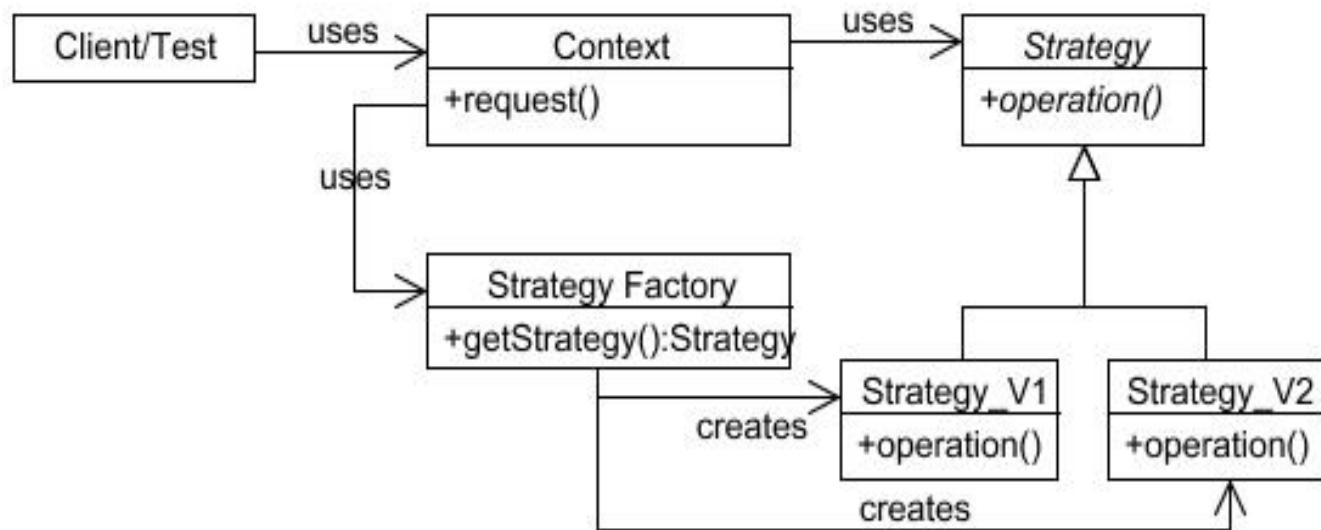
Issues

Who chooses the strategy?

a. client



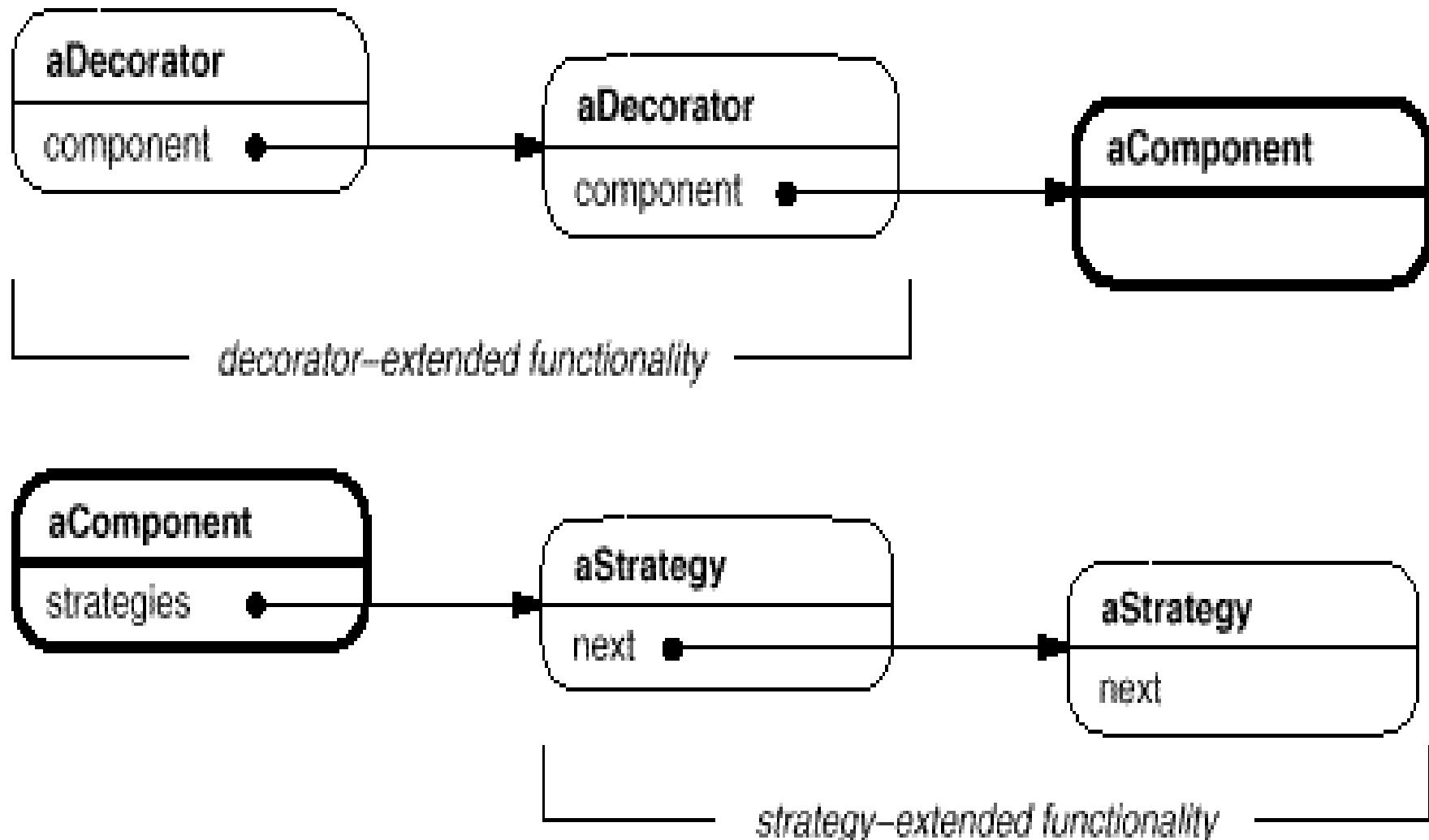
b. context



Implementation

- How does data flow between Context and Strategies?
 - **Approach 1:** take data to the strategy
 - decoupled, but might be inefficient
 - **Approach 2:** pass Context itself and let strategies take data
 - Context must provide a more comprehensive access to its data thus, more coupled
 - In Java strategy hierarchy might be **inner classes**
- Making Strategy object optional
 - provide Context with default behavior
 - if default used no need to create Strategy object
 - don't have to deal with Strategy unless you don't like the default behavior

Decorator vs. Strategy



STATE PATTERN

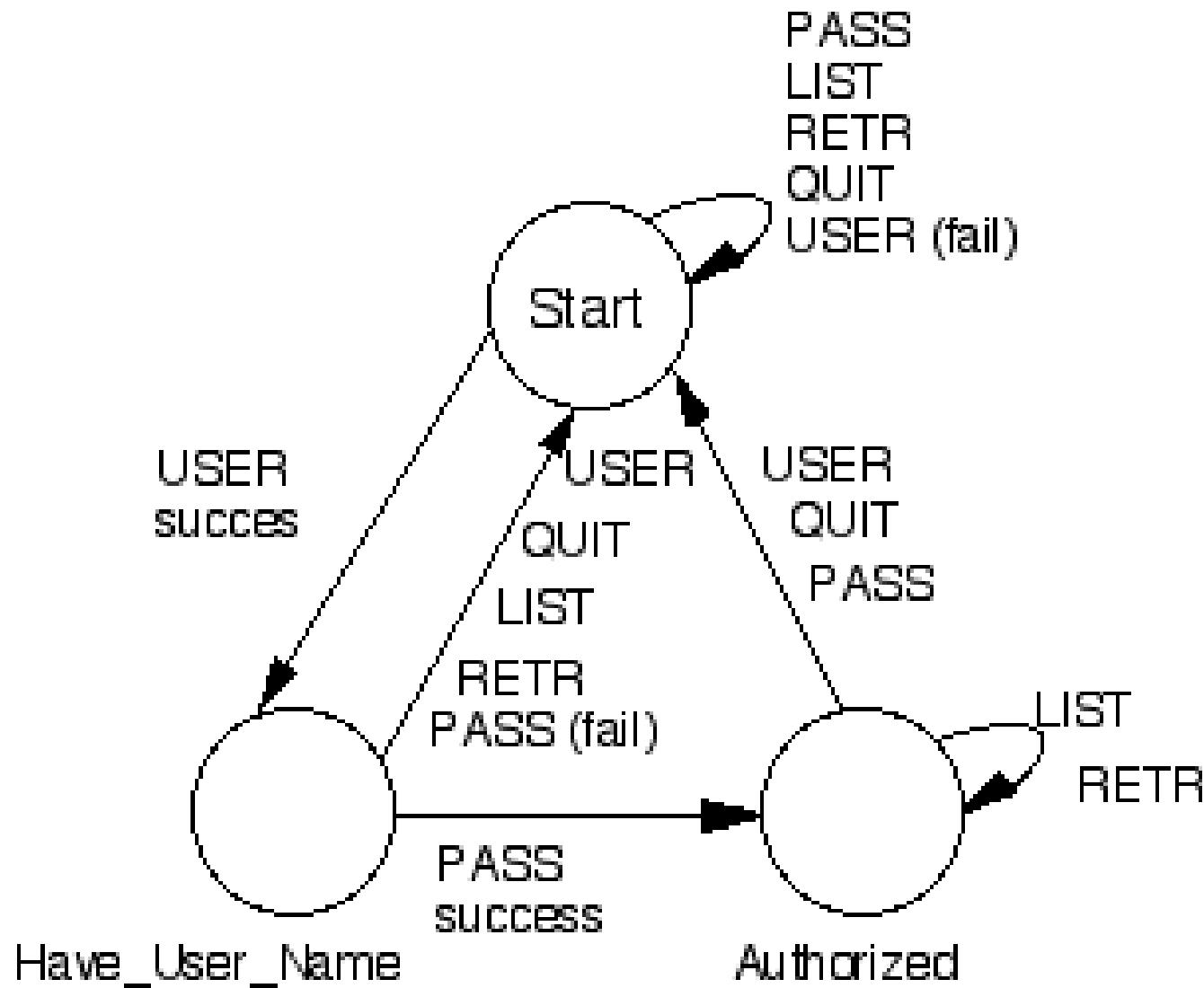
Example: SPOP

- SPOP = Simple Post Office Protocol
 - used to download emails from server
- SPOP supports the following commands:
 - USER <username>
 - PASS <password>
 - LIST
 - RETR <message number>
 - QUIT
- USER & PASS commands
 - USER with a username must come first
 - PASS with a password or QUIT must come after USER
 - If the username and password are valid, the user can use other commands

SPOP (contd.)

- LIST command
 - Arguments: a message-number (optional)
 - Returns: size of message in octets
 - if message number, returns the size of that message
 - otherwise return size of all mail messages in the mail-box
- RETR command
 - Arguments: a message number
 - Returns: the mail message indicated by that number
- QUIT command
 - Arguments: none
 - updates mailbox to reflect transactions taken during the transaction state, the logs user out
 - if session ends by any method except the QUIT command, the updates are not done

SPOP States



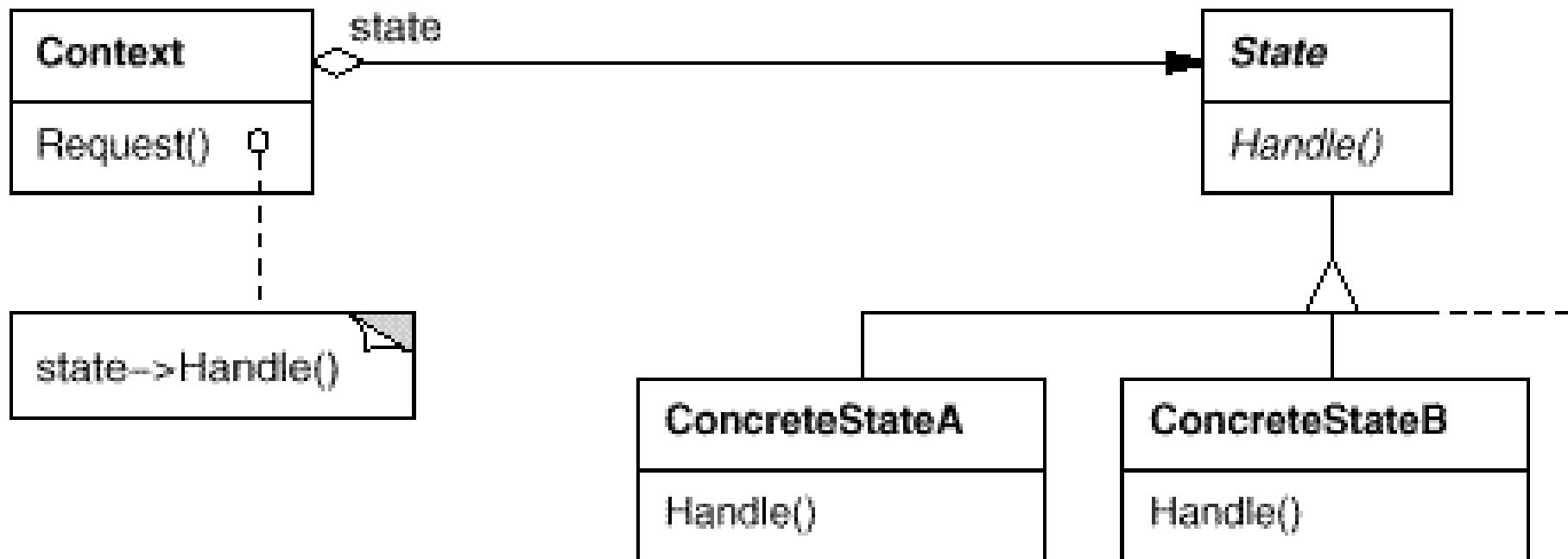
The "Dear, Old" Switches in Action

- ...as you see in this [code](#)
 - long functions
 - complex switches
 - same switches occur repeatedly in different functions
- Think about adding a new state to the protocol...
 - changes all the code
 - not Open-Closed
- Why?
 - object's behavior depends on its state

Basic Aspects of State Pattern

- Intent
 - allow an object to alter its behavior when its internal state changes
 - object will appear to change its class
- Applicability
 - object's behavior depends on its state
 - it must change behavior at run-time depending on that state
 - operations with multipart conditional statements depending on the object's state
 - state represented by one or more enumerated constants
 - several operations with the same (or similar) conditional structure

Structure



Participants

- Context
 - defines the interface of interest for clients
 - maintains an instance of ConcreteState subclass
- State
 - defines an interface for encapsulating the behavior associated with a particular state of the Context
- ConcreteState
 - each subclass implements a behavior associated with a state of the Context

Collaborations

- Context delegates state-specific requests to the State objects
 - the Context may pass itself to the State object
 - if the State needs to access it in order to accomplish the request
- State transitions are managed either by Context or by State
 - see discussion on the coming slides
- Clients interact exclusively with Context
 - but they might configure contexts with states
 - e.g initial state

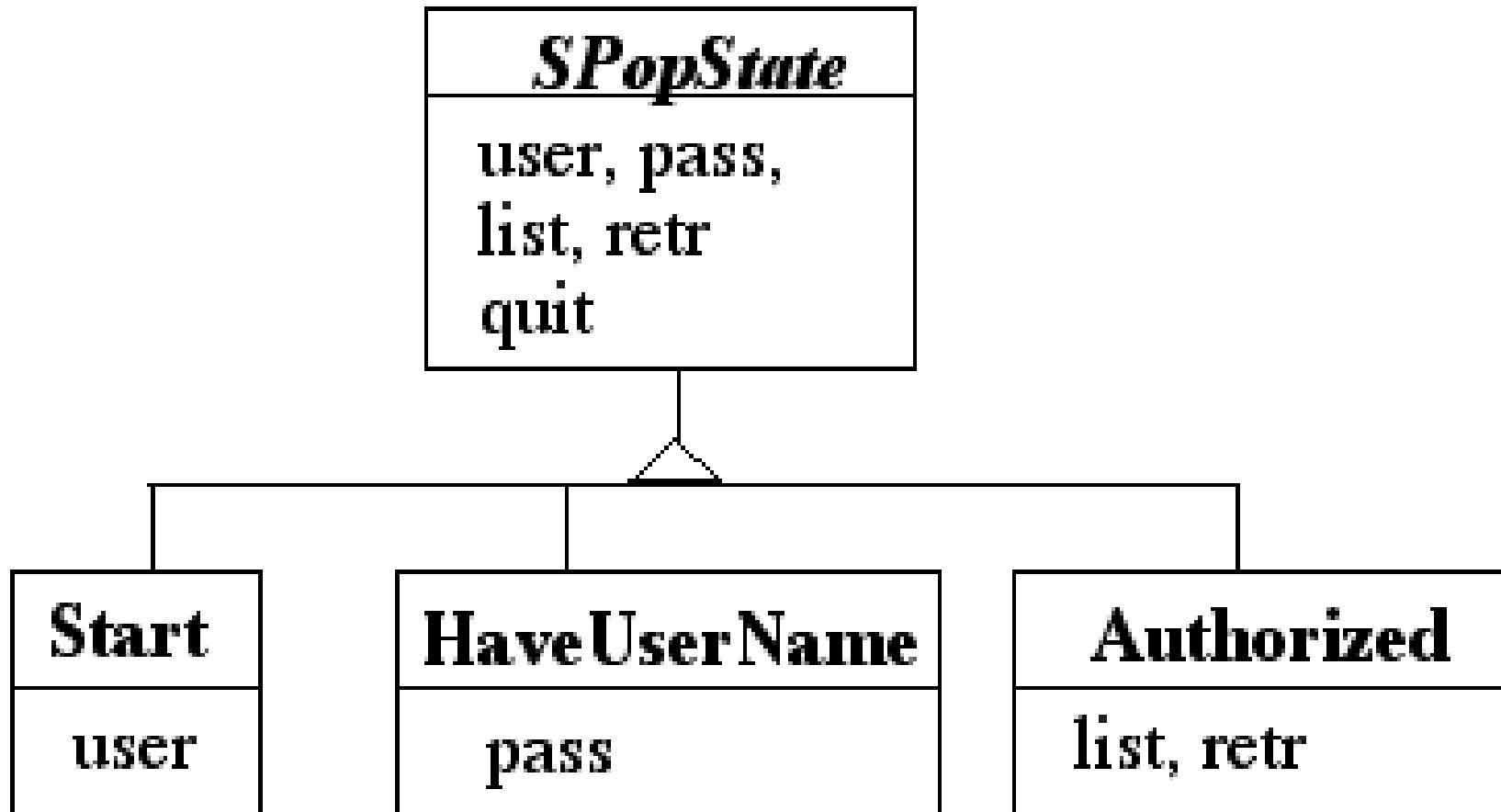
Consequences

- **Localizes** state-specific behavior and **partitions** behavior for different states
 - put all behavior associated with a state in a state-object
 - easy to add new states and transitions
 - behavior spread among several State subclasses
 - number of classes increases, less compact than a single class
 - good if many states...
- Makes state transitions explicit
 - not only a change of an internal value
 - **states receive a full-object status!**
 - protects Context from inconsistent internal states

Applying State to SPOP

```
class SPop {  
    private SPopState state = new Start();  
    public void user( String userName ) {  
        state = state.user( userName );  
    }  
    public void pass( String password ) {  
        state = state.pass( password );  
    }  
    public void list( int messageNumber ) {  
        state = state.list( messageNumber );  
    }  
    // . . .  
}
```

SPOP States



How much State in the State?

- Let's identify the roles...
 - **SPop** is the Context
 - **SPopState** is the abstract State
 - **Start**, **HaveUserName** are ConcreteStates
- All the state and *real* behavior is in SPopState and subclasses
 - this is an extreme example
- In general Context has data and methods
 - besides State & State methods
 - this data will not change states
- Only some aspects of the Context will alter its behavior

Who defines the State transition?

○ The Context if ...

- ...states will be **reused** in different state machines with different transitions
- ... the criteria for changing states are fixed
- SPOP [Example](#)

○ The States

- More flexible to let State subclasses specify the next state
- as we have seen [before](#)

Sharing State Objects

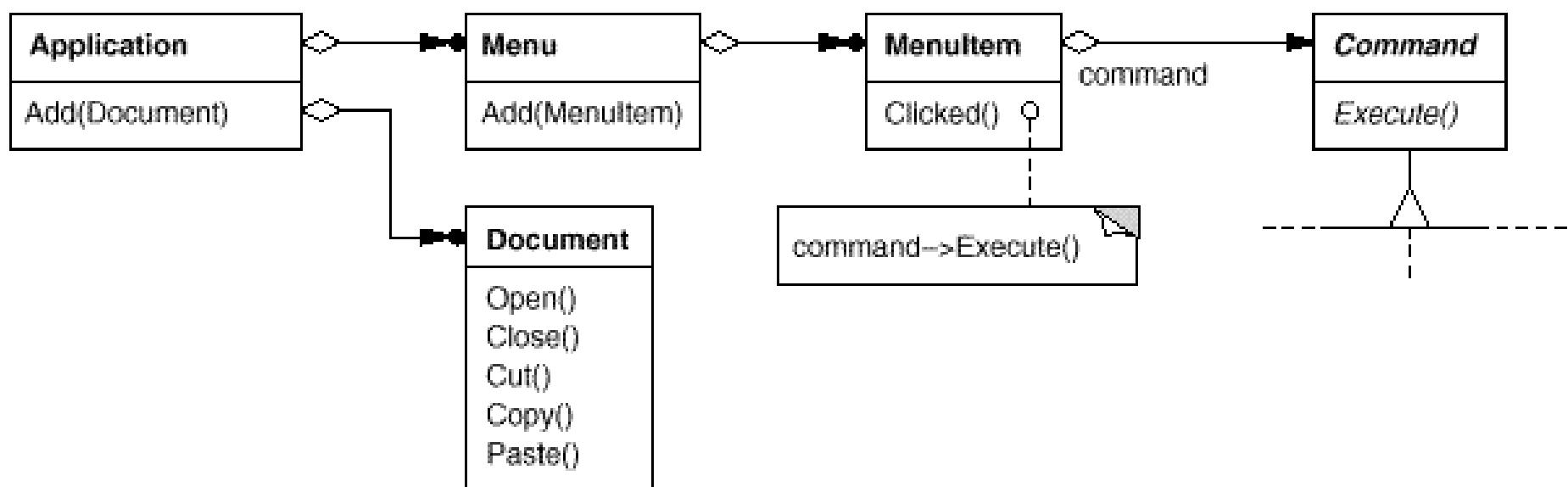
- Multiple contexts can use the same state object
 - if the state object has no instance variables
- State object has no instance variables if the object ...
 - ... has **no need** for instance variables
 - ... stores its instance variables elsewhere
 - Variant 1 – Context stores them and passes them to states
 - Variant 2 – Context stores them and State gets them

State versus Strategy

- Rate of Change
 - Strategy
 - Context object usually contains one of several possible **ConcreteStrategy** objects
 - State
 - Context object often changes its **ConcreteState** object over its lifetime
- Visibility of Change
 - Strategy
 - All **ConcreteStrategy** do the same thing, but differently
 - Clients do not see any difference in behavior in the Context
 - State
 - **ConcreteState** acts differently
 - Clients see different behavior in the Context

COMMAND PATTERN

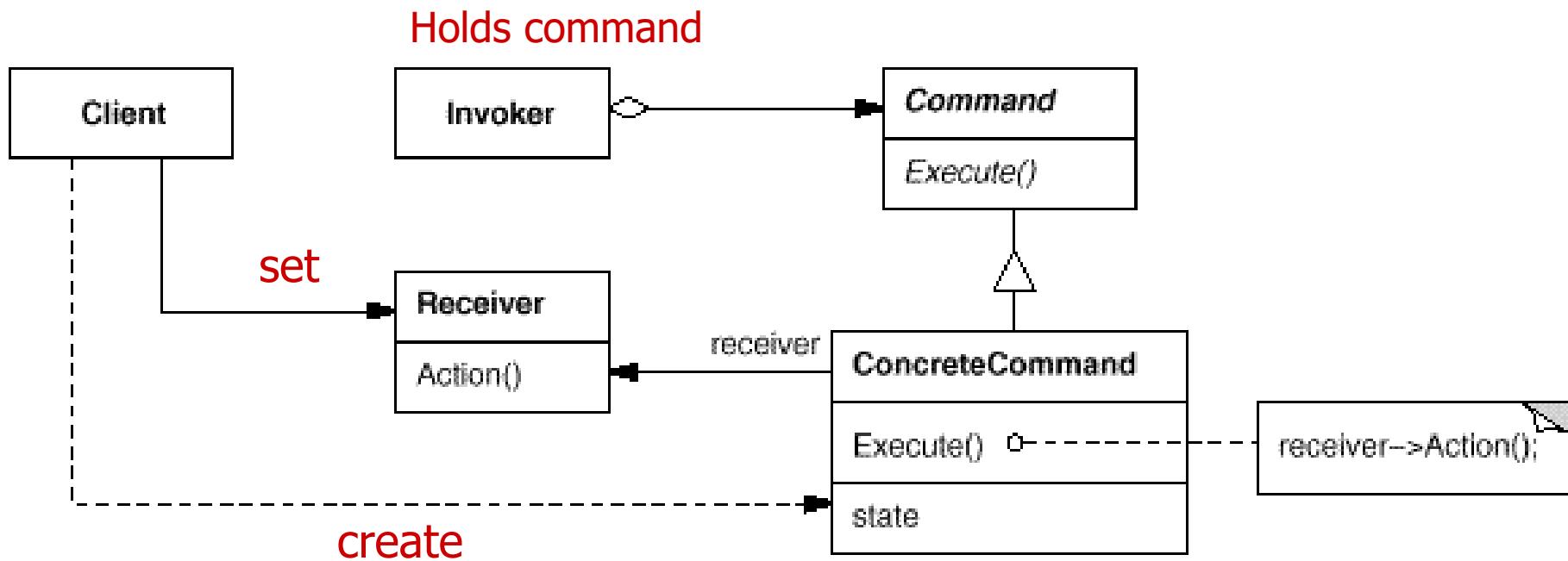
Menu Items Use Commands



Basic Aspects

- Intent
 - Encapsulate requests as objects, letting you to:
 - parameterize clients with different requests
 - queue or log requests
 - support undoable operations
- Applicability
 - Parameterize objects
 - replacement for callbacks
 - Specify, queue, and execute requests at different times
 - Support undo
 - recover from crashes → needs undo operations in interface
 - Support for logging changes
 - recover from crashes → needs load/store operations in interface
 - Model transactions
 - structure systems around high-level operations built on primitive ones
 - common interface ⇒ invoke all transaction same way

Structure

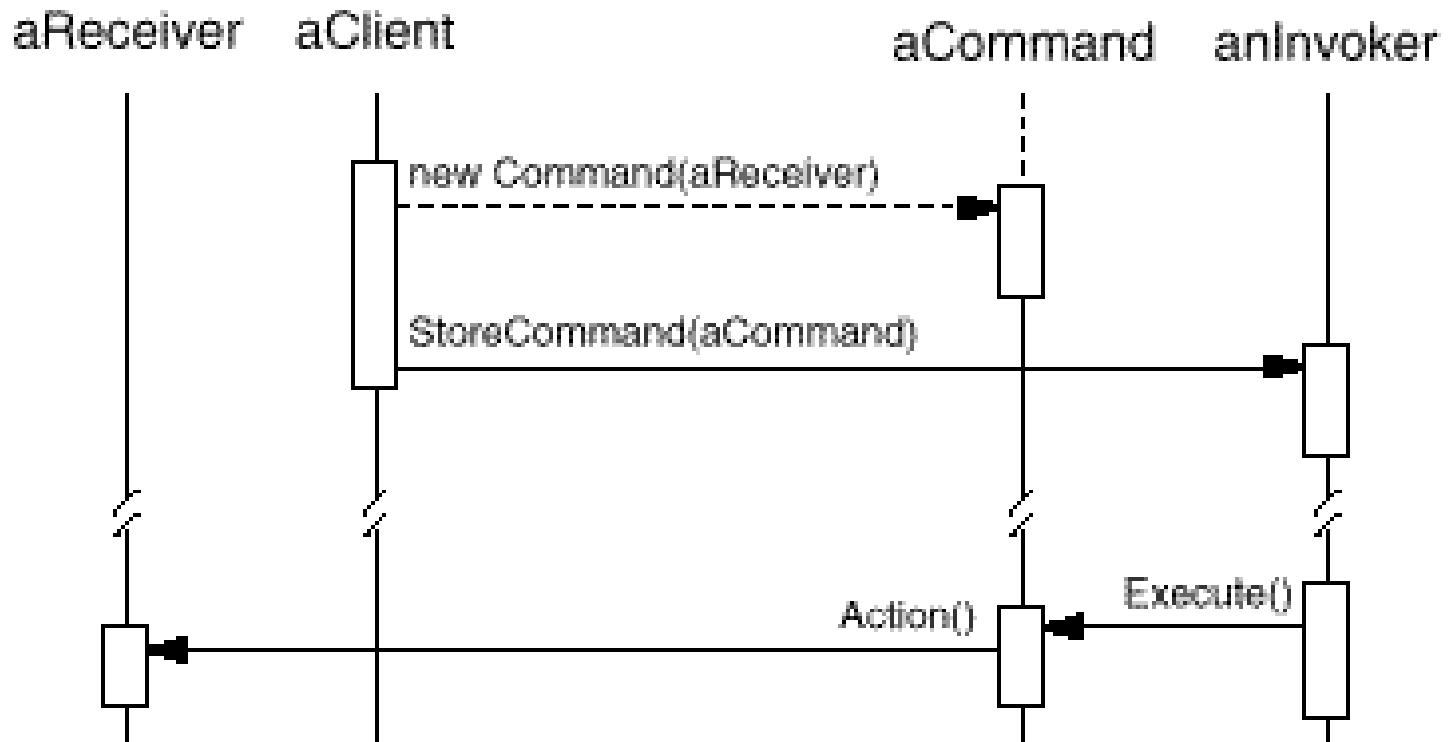


Transforms: `concreteReceiver.action()` in `command.execute()`

Participants

- Command
 - declares the interface for executing the operation
- ConcreteCommand
 - binds a request with a concrete action
- Invoker
 - asks the command to carry out the request
- Receiver
 - knows how to perform the operations associated with carrying out a request.
- Client
 - creates a ConcreteCommand and sets its receiver

Collaborations

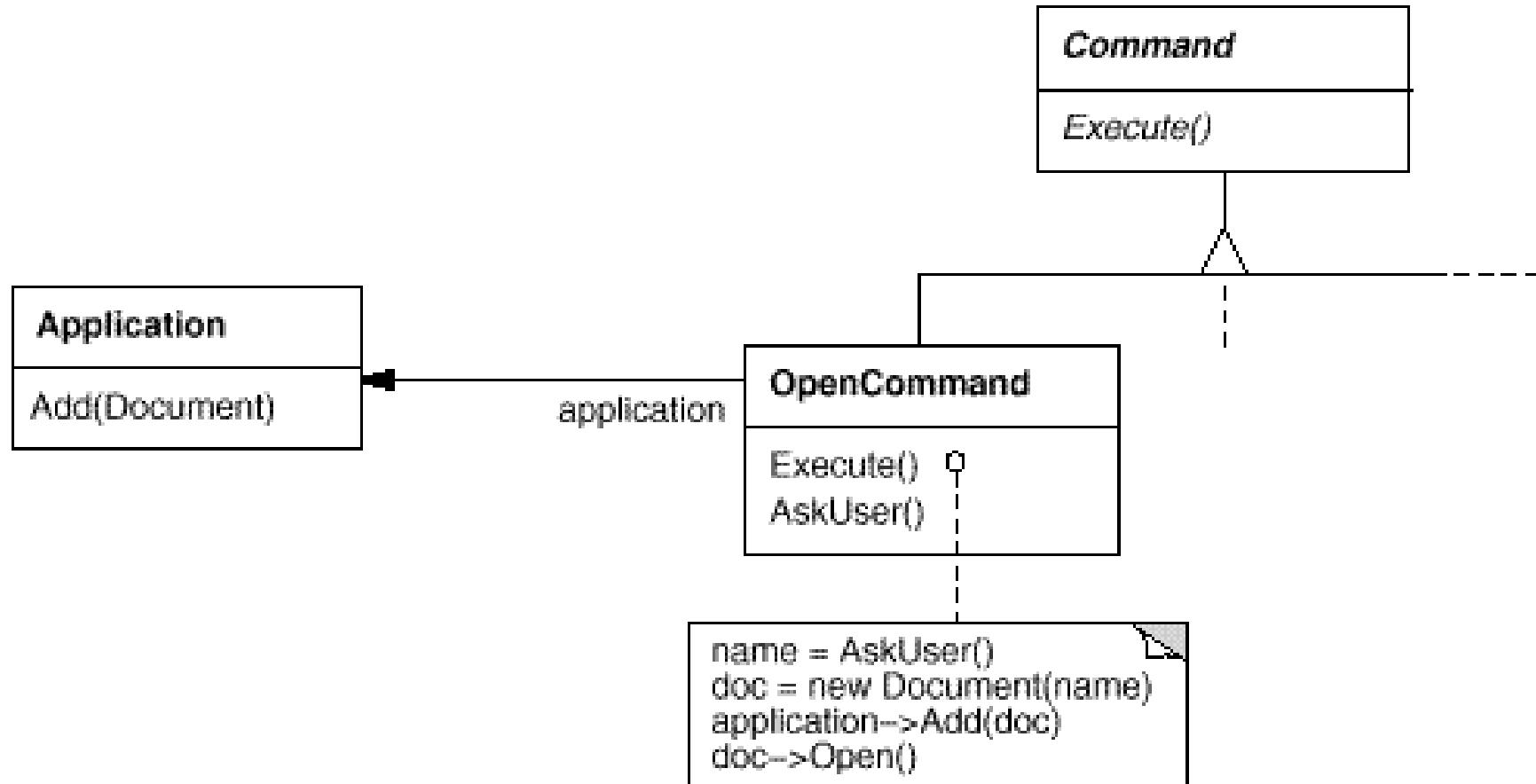


- Client → ConcreteCommand
 - creates and specifies receiver
- Invoker → ConcreteCommand
- ConcreteCommand → Receiver

Consequences

- Decouples Invoker from Receiver
- Commands are **first-class objects**
 - can be manipulated and **extended**
- Composite Commands
 - see also *Composite* pattern
- Easy to add new commands
 - Invoker does not change
 - it is Open-Closed
- Potential for an excessive number of command classes

Example: Open Document



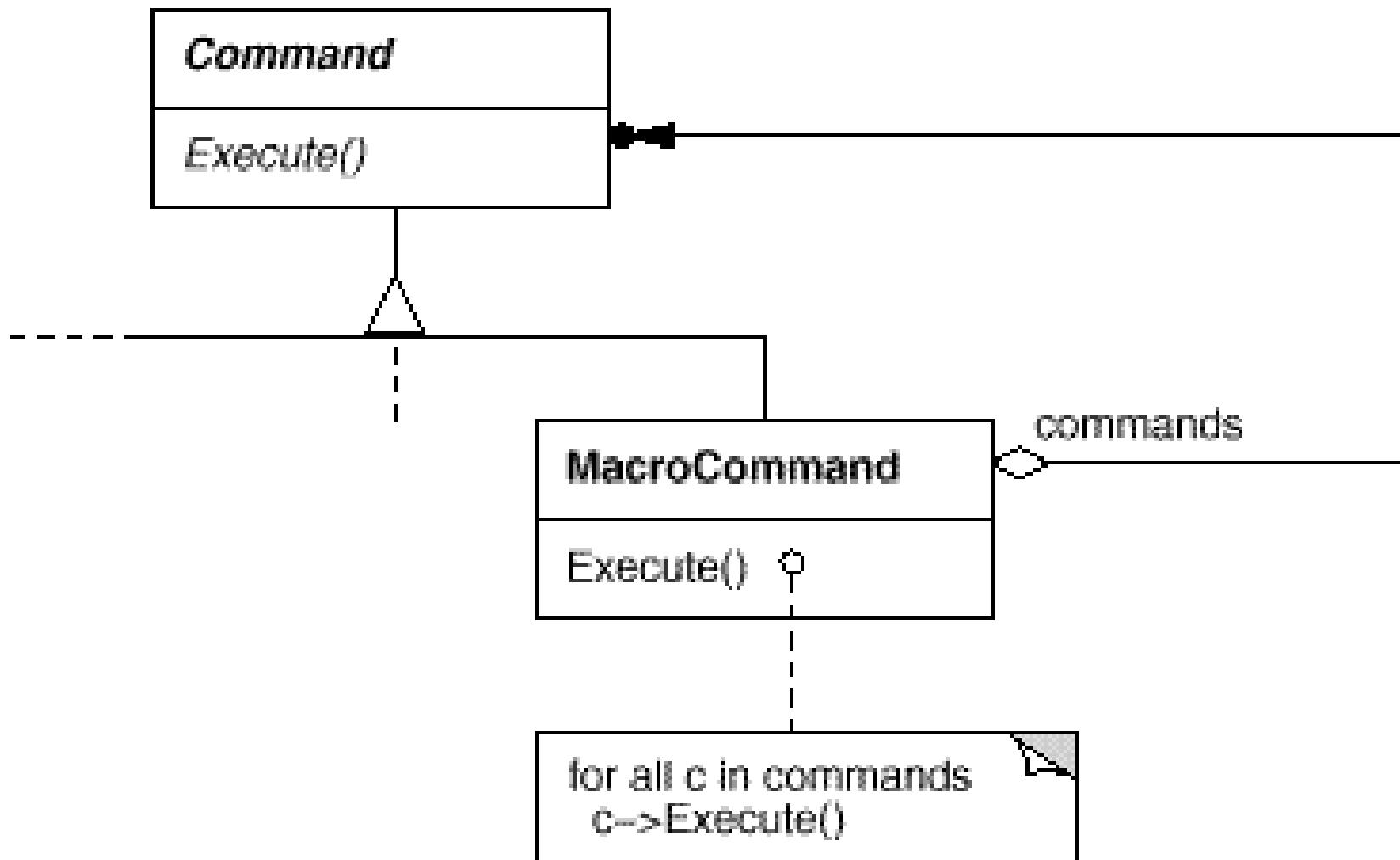
Intelligence of Command objects

- "Dumb"
 - delegate everything to Receiver
 - used just to decouple Sender from Receiver
- "Genius"
 - does everything itself without delegating at all
 - useful if no receiver exists
 - let ConcreteCommand be independent of further classes
- "Smart"
 - find receiver dynamically

Undoable Commands

- Need to store additional state to reverse execution
 - receiver object
 - parameters of the operation performed on receiver
 - original values in receiver that may change due to request
 - receiver must provide operations that makes possible for command object to return it to its prior state
- History list
 - sequence of commands that have been executed
 - used as LIFO with reverse-execution \Rightarrow undo
 - used as FIFO with execution \Rightarrow redo
 - Commands may need to be copied
 - when state of commands change by execution

Composed Commands



CHAIN OF RESPONSIBILITY PATTERN

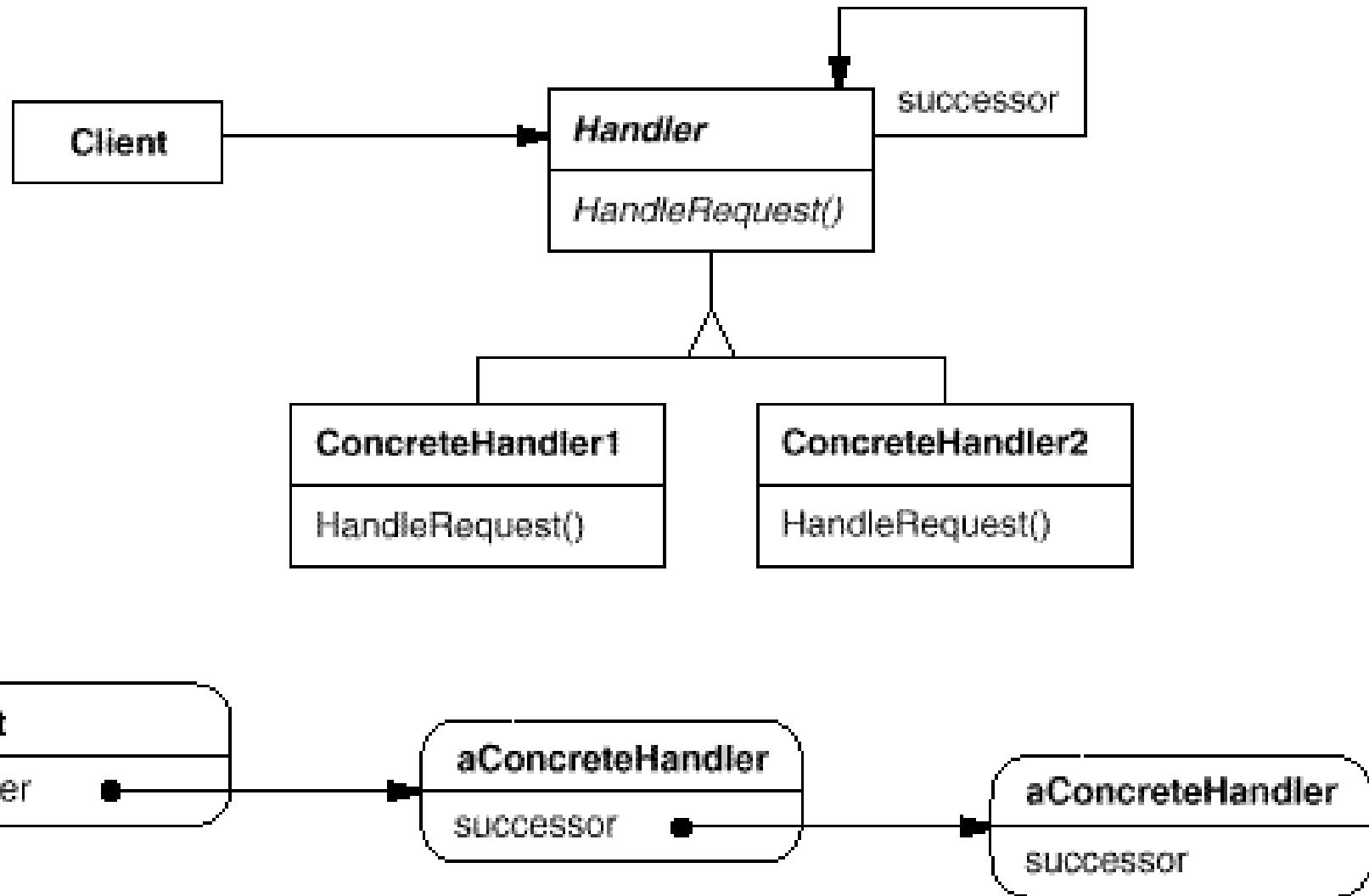
Basic Aspects

- Intent
 - Decouple sender of request from its receiver
 - by giving more than one object a chance to handle the request
 - Put receivers in a chain and pass the request along the chain
 - until an object handles it
- Motivation
 - context-sensitive help
 - a help request is handled by one of several UI objects
 - Which one?
 - depends on the context
 - The object that initiates the request does not know the object that will eventually provide the help

When to Use?

- Applicability
 - more than one object may handle a request
 - and handler isn't known a priori
 - set of objects that can handle the request should be dynamically specifiable
 - send a request to several objects without specifying the receiver

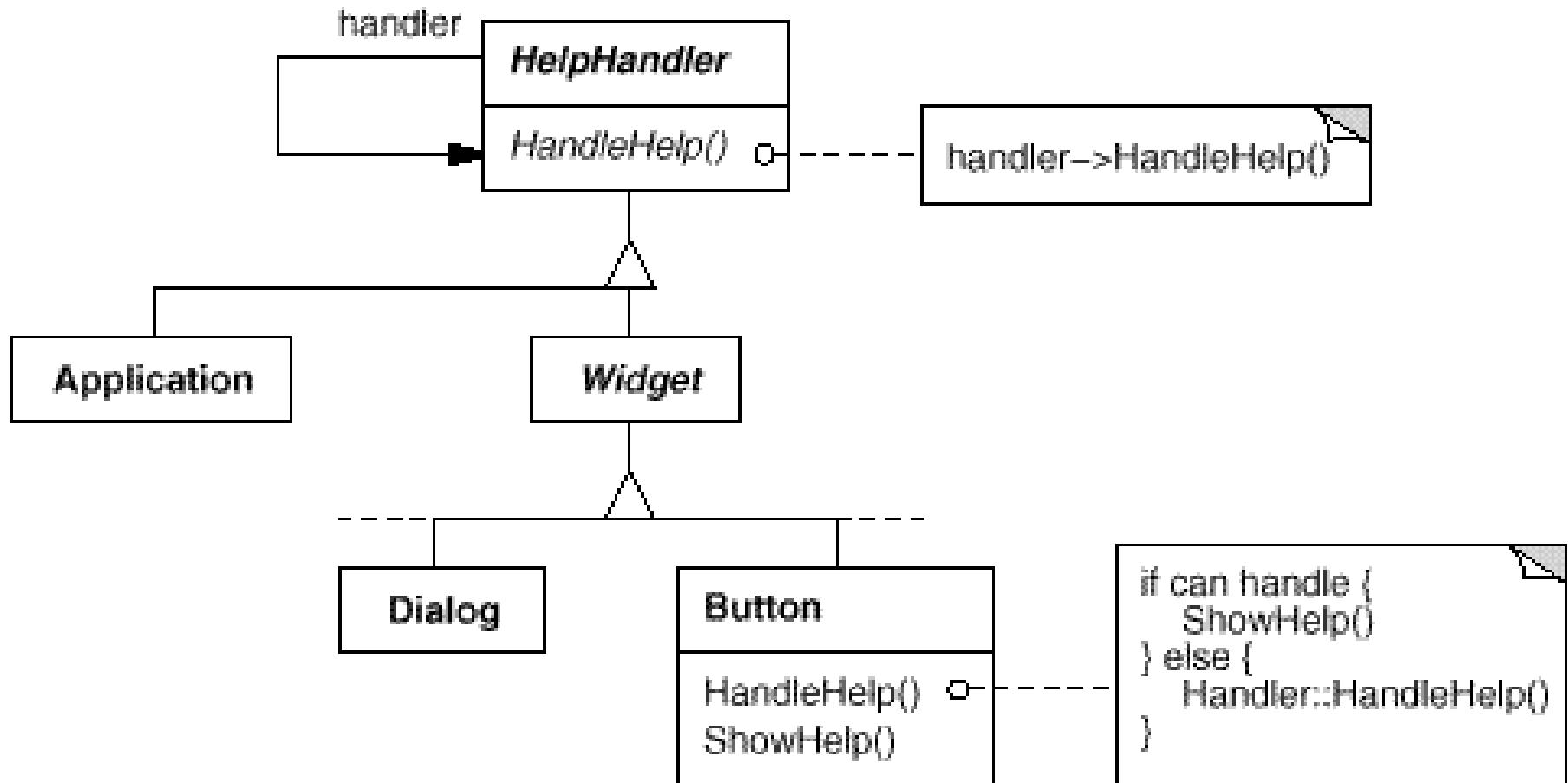
Structure



Participants & Collaborations

- Handler
 - defines the interface for handling requests
 - may implement the successor link
- ConcreteHandler
 - either handles the request it is responsible for ...
 - ... or it forwards the request to its successor
- Client
 - initiates the request to a ConcreteHandler object in the chain

The Context-Help System

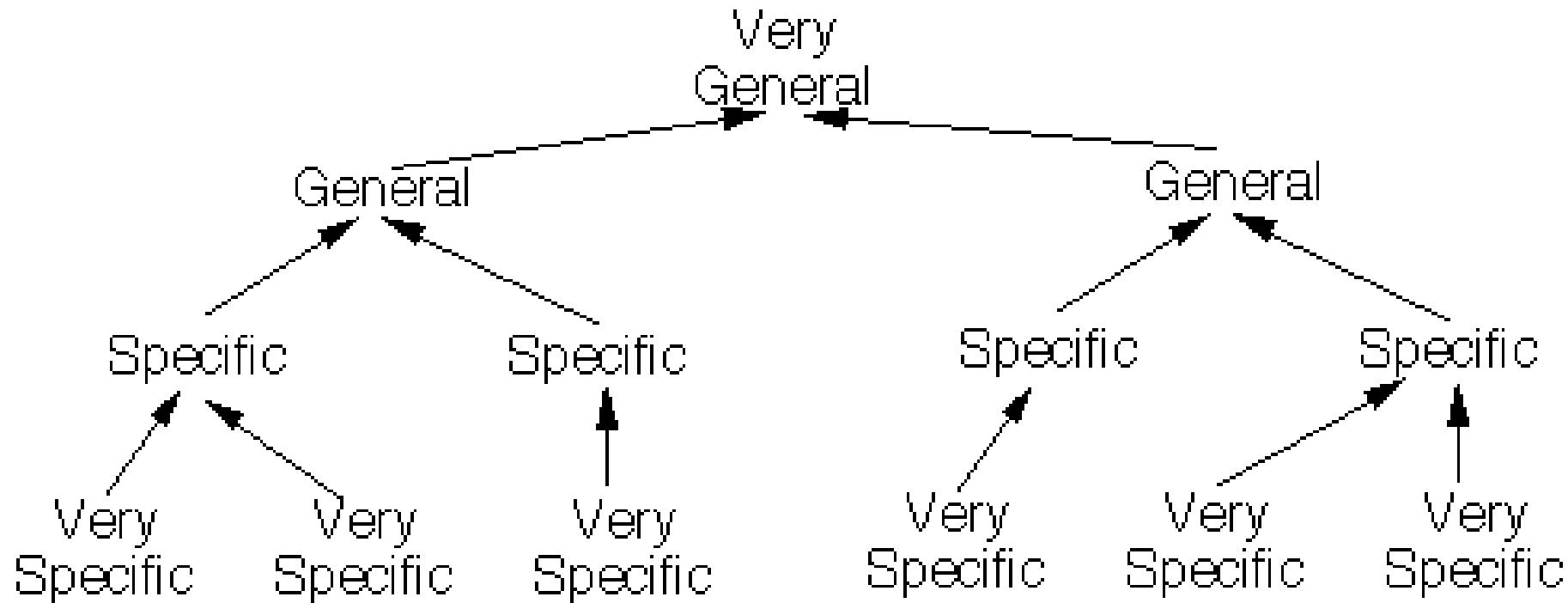


Consequences

- Reduced Coupling
 - frees the client (sender) from knowing who will handle its request
 - sender and receiver don't know each other
 - instead of sender knowing all potential receivers, just keep a single reference to next handler in chain.
 - simplify object interconnections
- Flexibility in assigning responsibilities to objects
 - responsibilities can be added or changed
 - chain can be modified at run-time
- Requests can go unhandled
 - chain may be configured improperly

How to Design Chains of Commands?

- Like the military
 - a request is made
 - it goes up the chain of command until someone has the authority to answer the request



Implementing the Successor Chain

- Define new link
 - Give each handler a link to its successor
- Use existing links
 - concrete handlers may already have pointers to their successors
 - parent references in a part-whole hierarchy
 - can define a part's successor
 - spares work and space ...
 - ... but it must reflect the chain of responsibilities that is needed

Representing Multiple Requests using One Chain

- Each request is hard-coded
 - convenient and safe
 - not flexible
 - limited to the fixed set of requests defined by handler
- Unique handler with parameters
 - more flexible
 - but it requires conditional statements for dispatching request
 - less type-safe to pass parameters
- Unique handler with Request object parameter
 - subclasses extend rather than overwrite the handler method

Decorator vs. Chain of Responsibility

Chain of Responsibility	Decorator
Comparable to “event-oriented” architecture	Comparable to layered architecture (layers of an onion)
The "filter" objects are of equal rank	A "core" object is assumed, all "layer" objects are optional
User views the chain as a "launch and leave" pipeline	User views the decorated object as an enhanced object
A request is routinely forwarded until a single filter object handles it. many (or all) filter objects could contrib. to each request's handling.	A layer object always performs pre or post processing as the request is delegated.
All the handlers are peers (like nodes in a linked list) – "end of list" condition handling is required.	All the layer objects ultimately delegate to a single core object - "end of list" condition handling is not required.

SOFTWARE DESIGN

Service oriented architectures

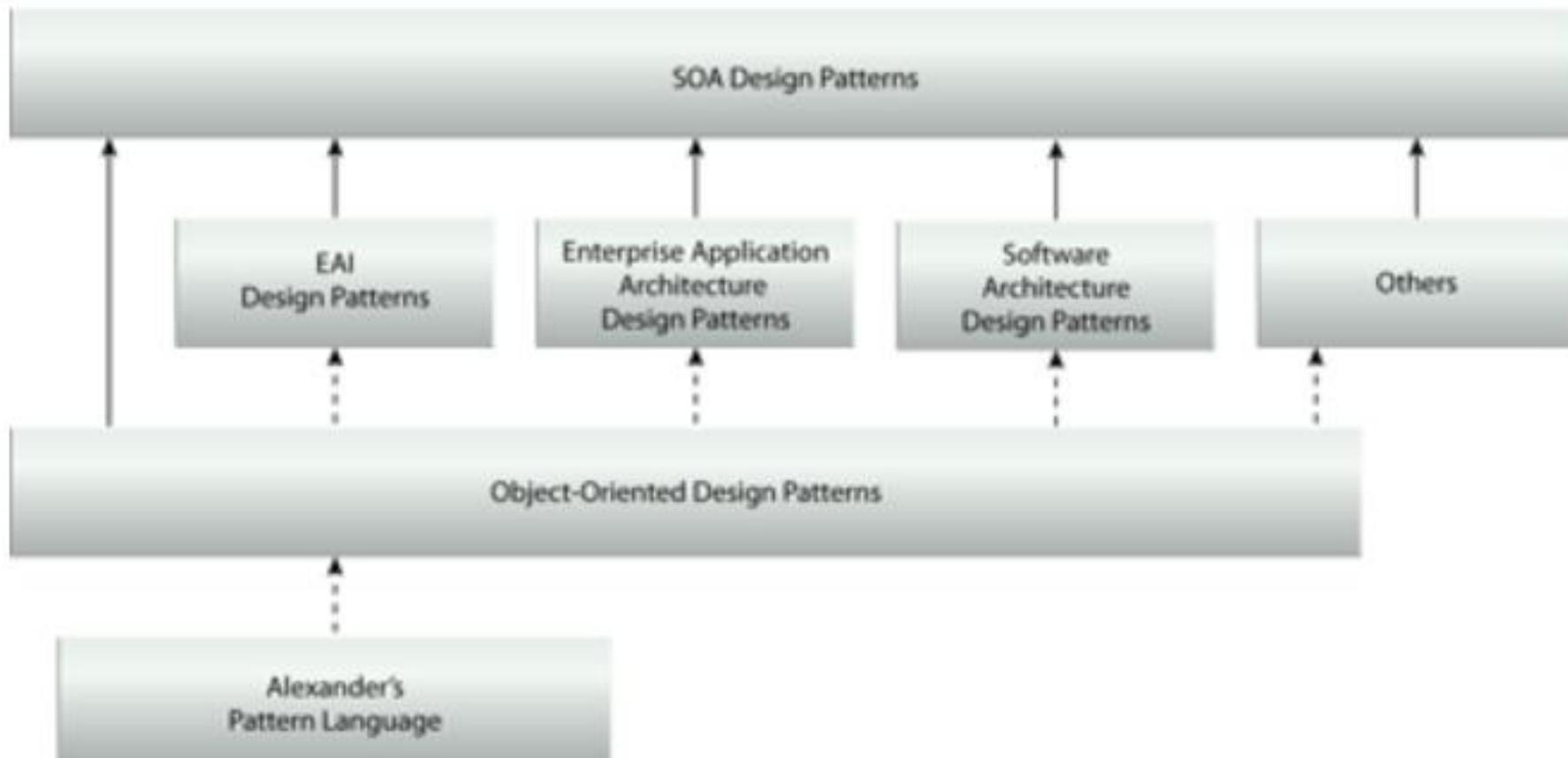
References

- Erl, Thomas. Service-Oriented Architecture: Analysis and Design for Services and Microservices. Pearson Education. 2016
- Erl, Thomas. SOA Design Patterns, Prentice Hall, 2009.
- <http://soapatterns.org>
- Mark Richards, Software Architecture Patterns, O'Reilly, 2015
- Mark Richards, Microservices vs. Service-Oriented Architecture O'Reilly, 2016

Content

- Service Oriented Architecture Process
- Case-Study
- Service oriented patterns examples

Overview

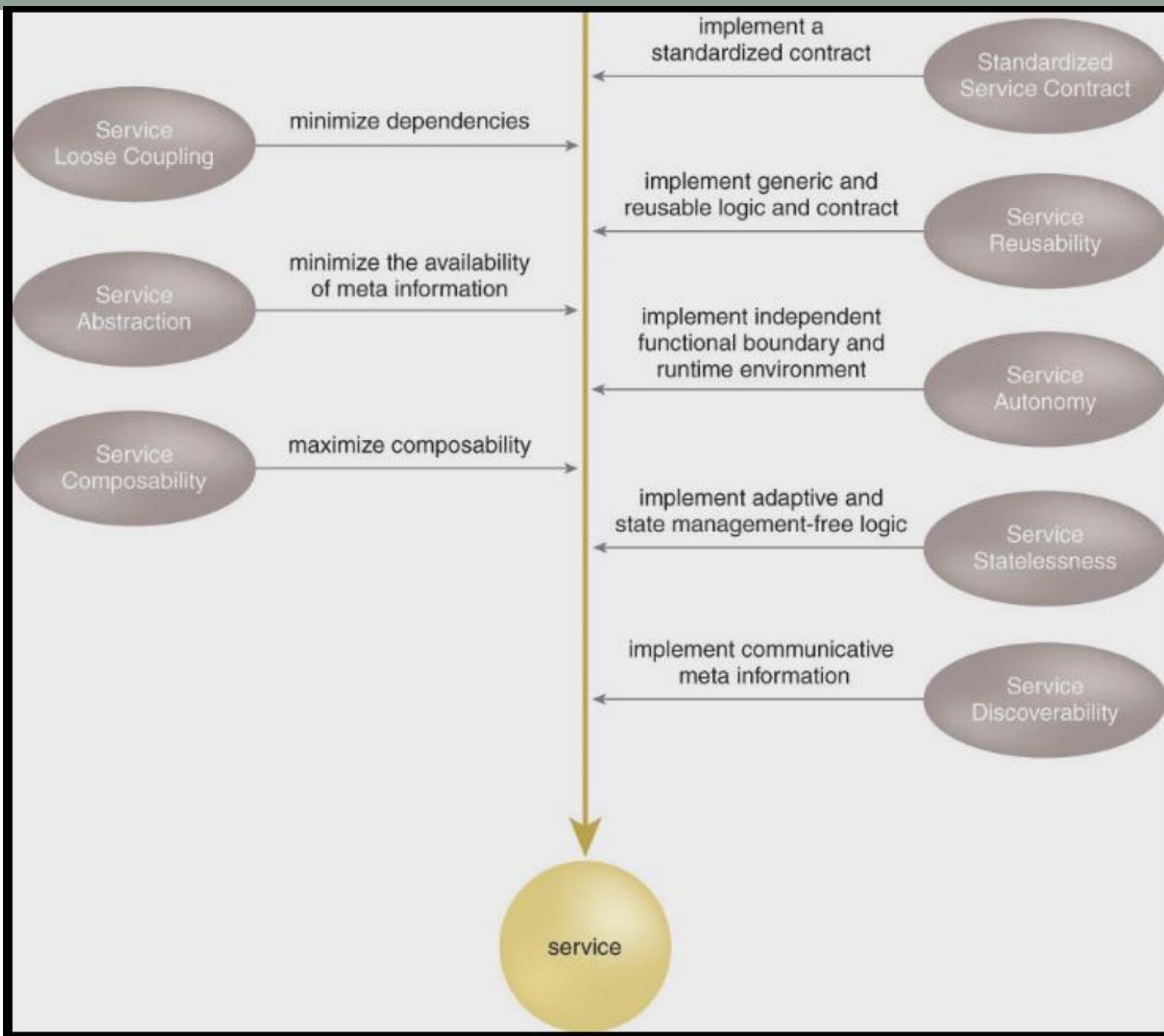


Service Design principles

- Service Abstraction – Service contracts only contain essential information and information about services is limited to what is published in service contracts.
- Service Autonomy – Services exercise a high level of control over their underlying runtime execution environment.
- Service Statelessness – Services minimize resource consumption by deferring the management of state information when necessary.
- Service Discoverability – Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted.

Service Design principles

- Service Loose Coupling – Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment.
- Service Reusability– Services contain and express agnostic logic and can be positioned as reusable enterprise resources.
- Service Composability – Services are effective composition participants, regardless of the size and complexity of the composition.
- Standardized Service Contract – Services within the same service inventory are in compliance with the same contract design standards.



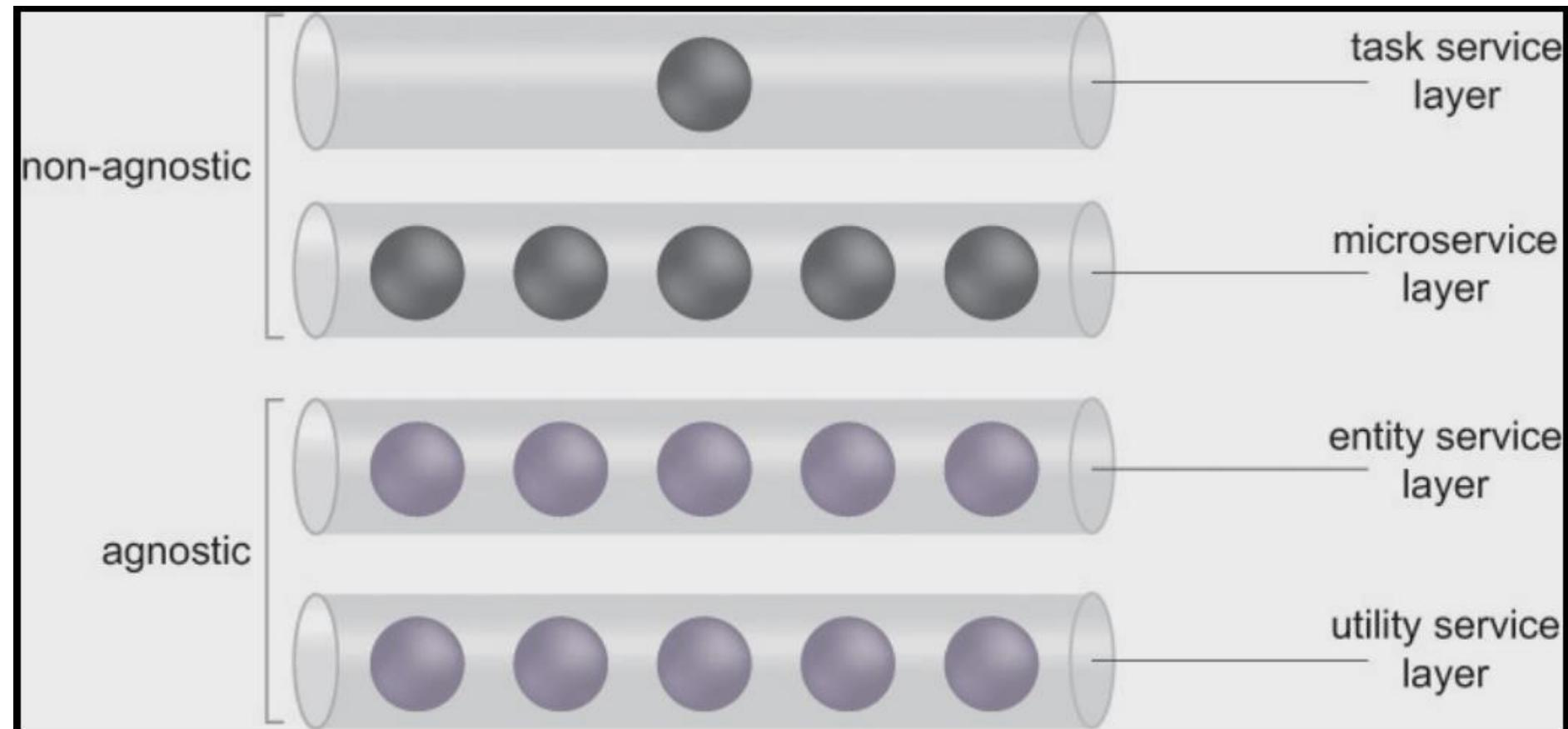
Service models

- Task Service
 - a **non-agnostic functional context**
 - **single-purpose**, parent business process logic.
 - encapsulates the **composition logic** required to compose several other services to complete its task.
- Microservice
 - a **non-agnostic service**
 - **small functional scope**
 - logic with **specific processing** and implementation requirements.
 - typically **not reusable** but can have intra-solution reuse potential.
 - The nature of the logic may vary.

Service models

- Entity Service
 - a **reusable service** with an **agnostic functional context**
 - associated with one or more related **business entities** (such as invoice, customer, or claim).
- Utility Service
 - a **reusable service** with an **agnostic functional context** as well,
 - not derived from business analysis specifications and models.
 - encapsulates **low-level technology-centric functions**, such as notification, logging, and security processing.

Service layers



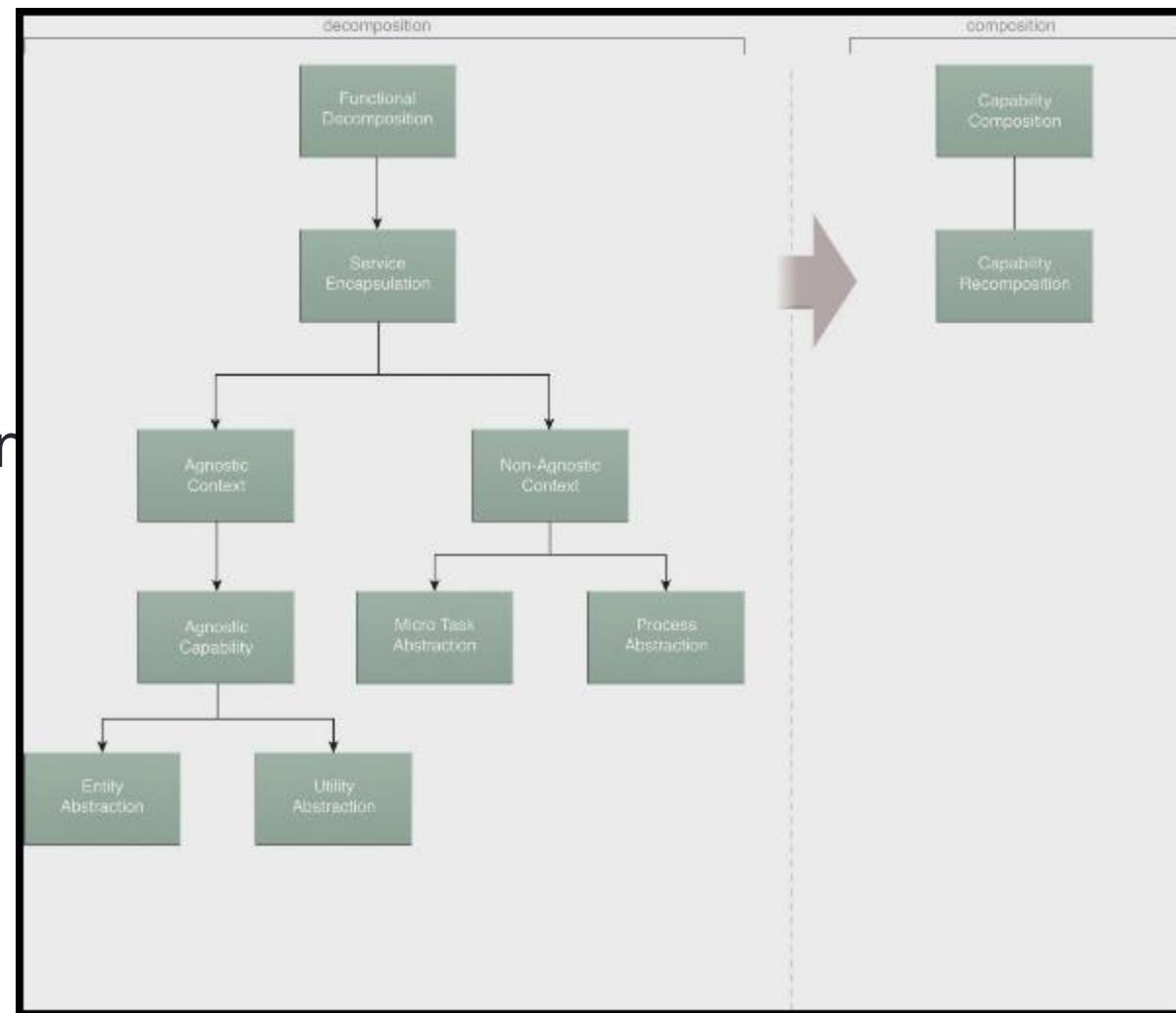
Process

- Break down the business problem
⇒ Service design

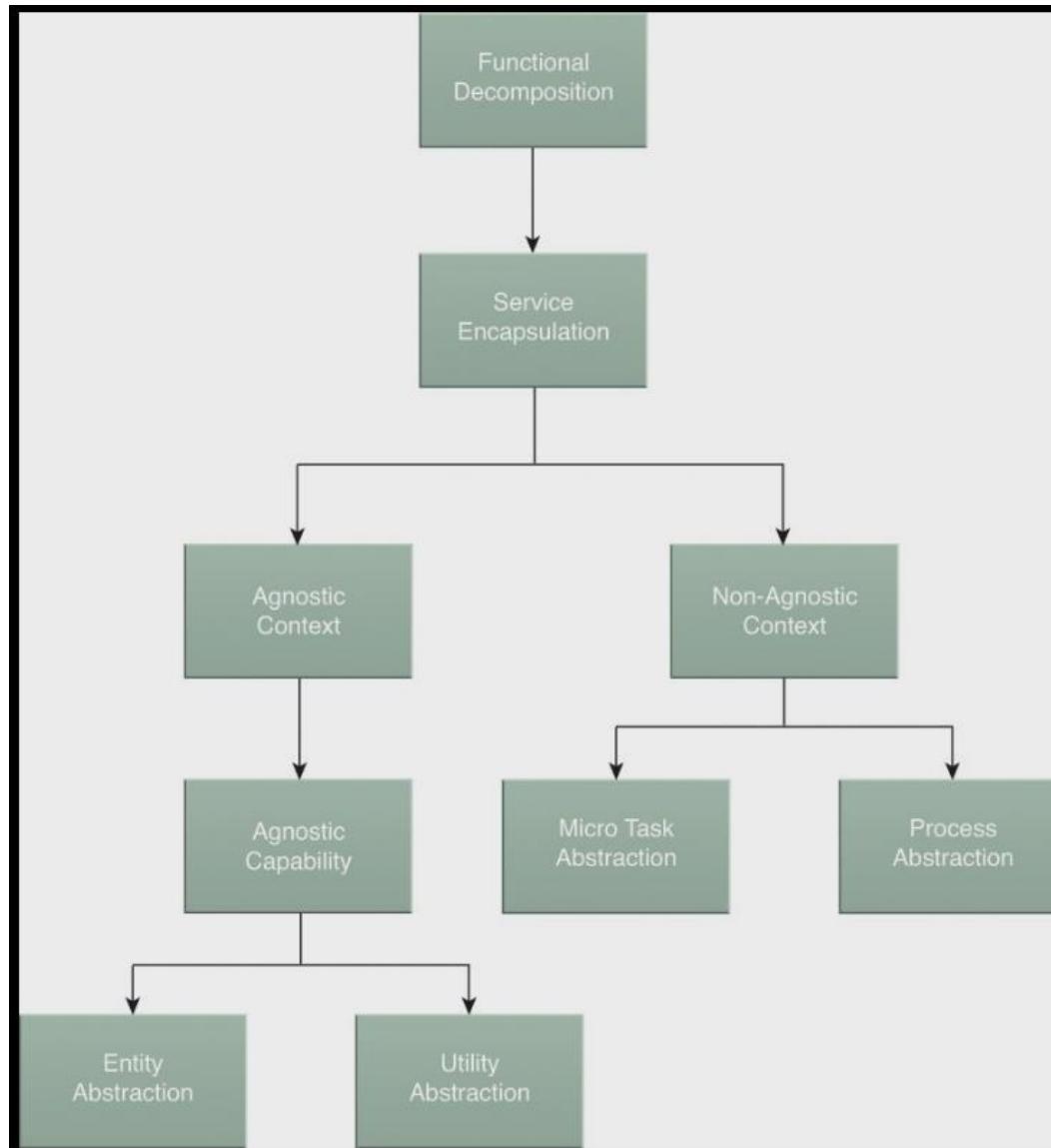
- Build up the Service-oriented solution

⇒ Solution

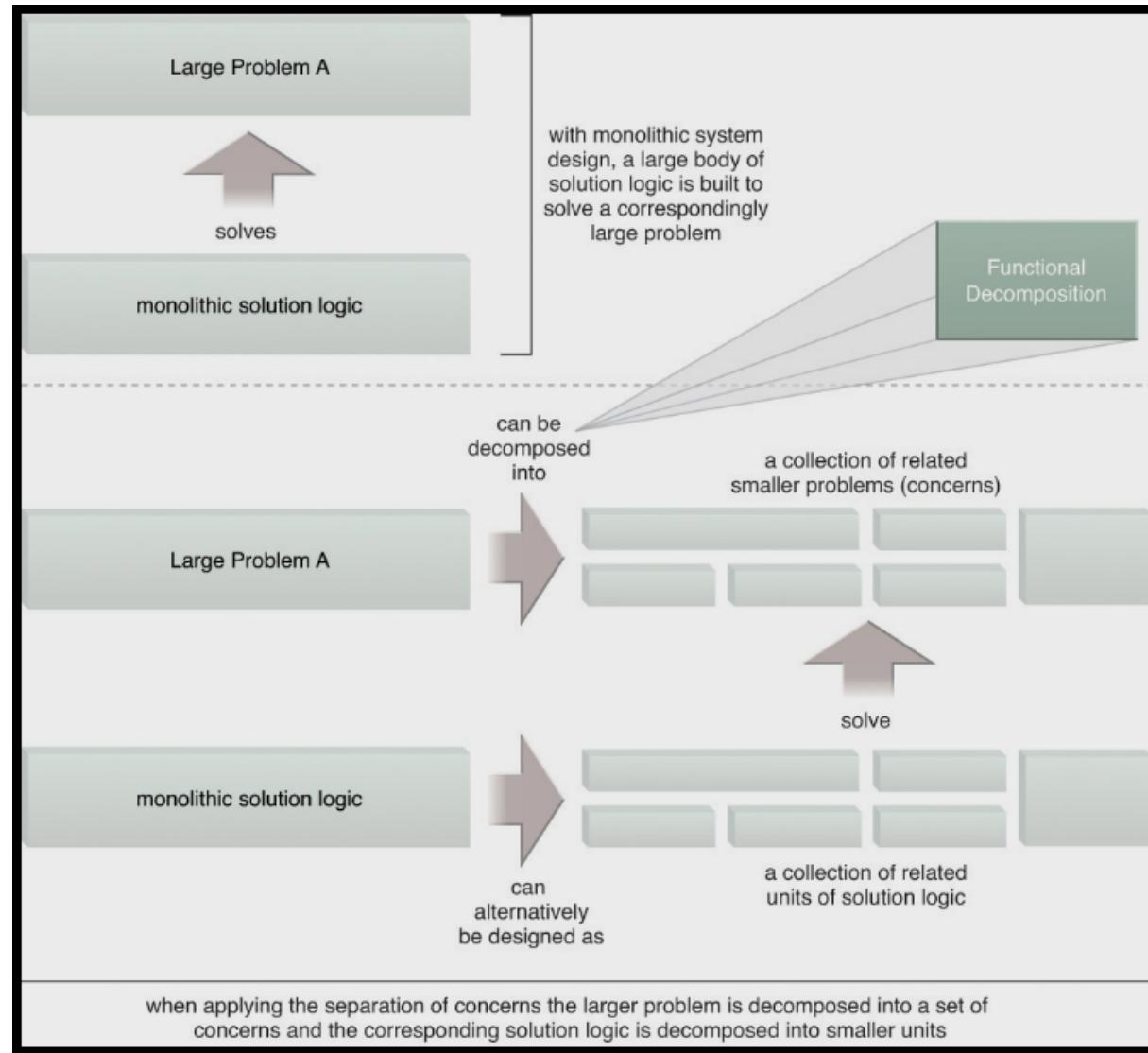
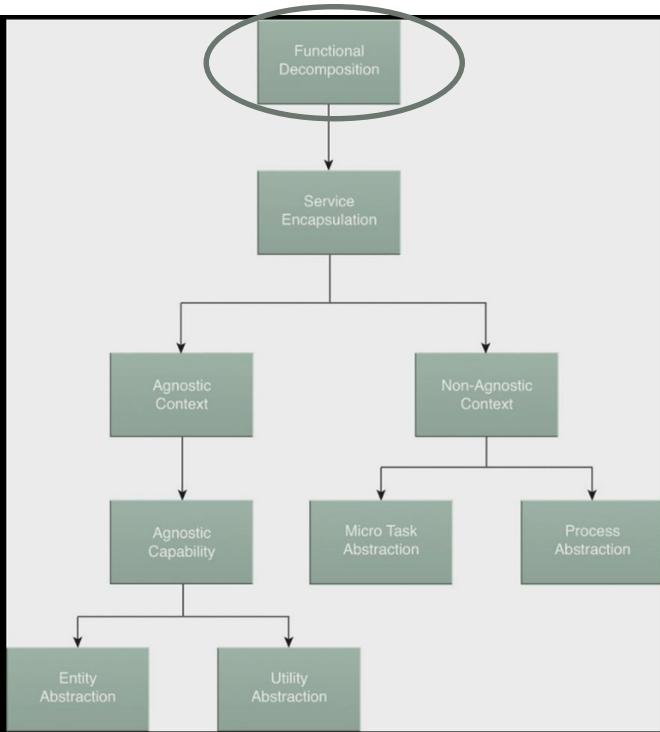
- Service layers



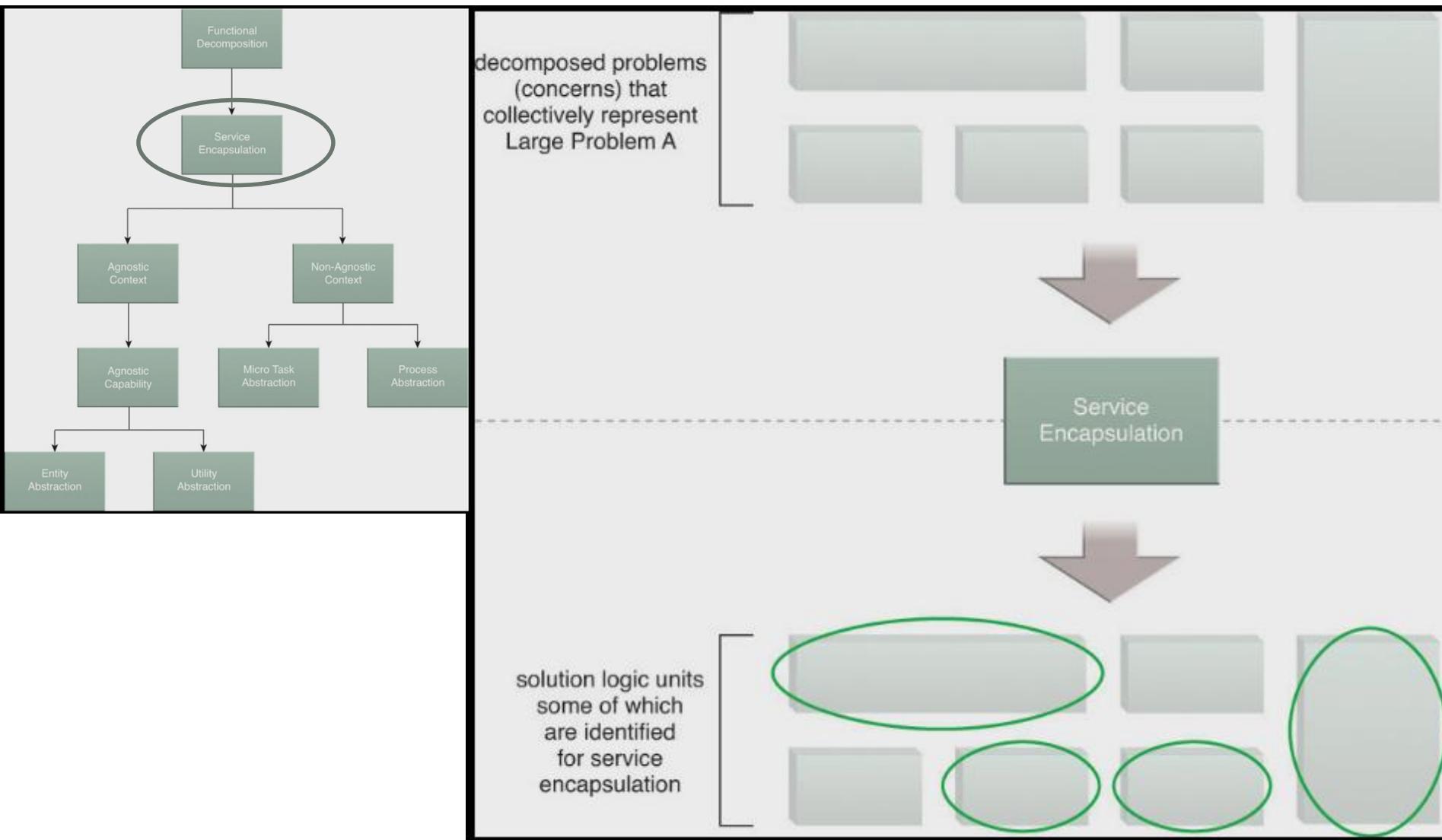
Break down the business problem



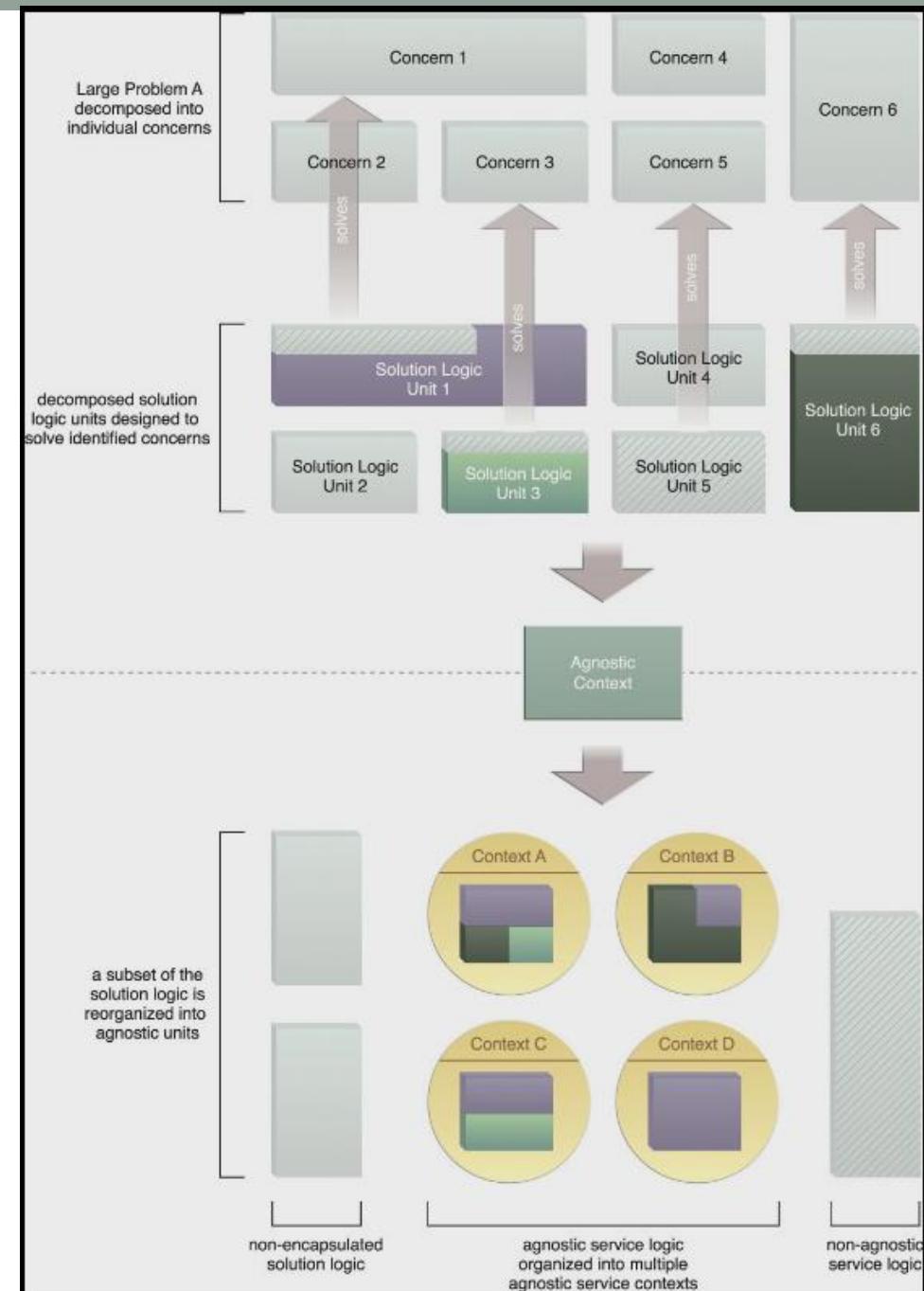
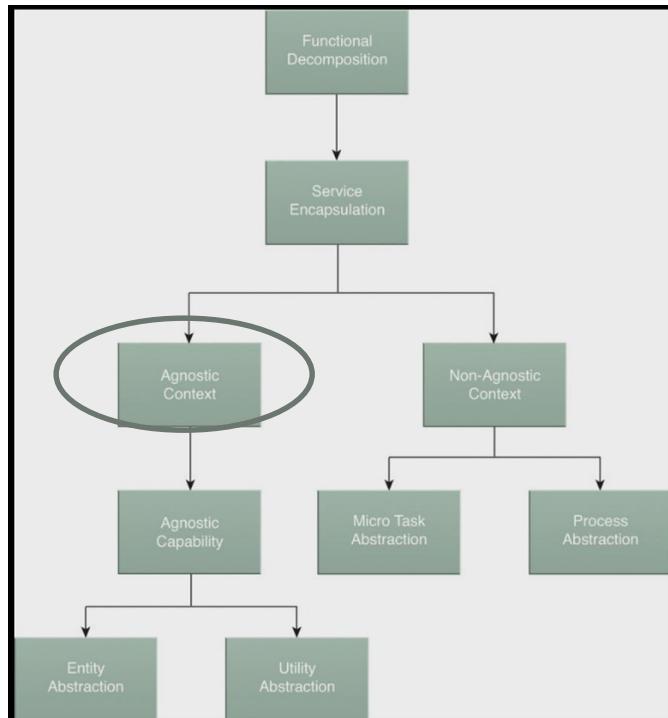
Functional Decomposition – separation of concerns



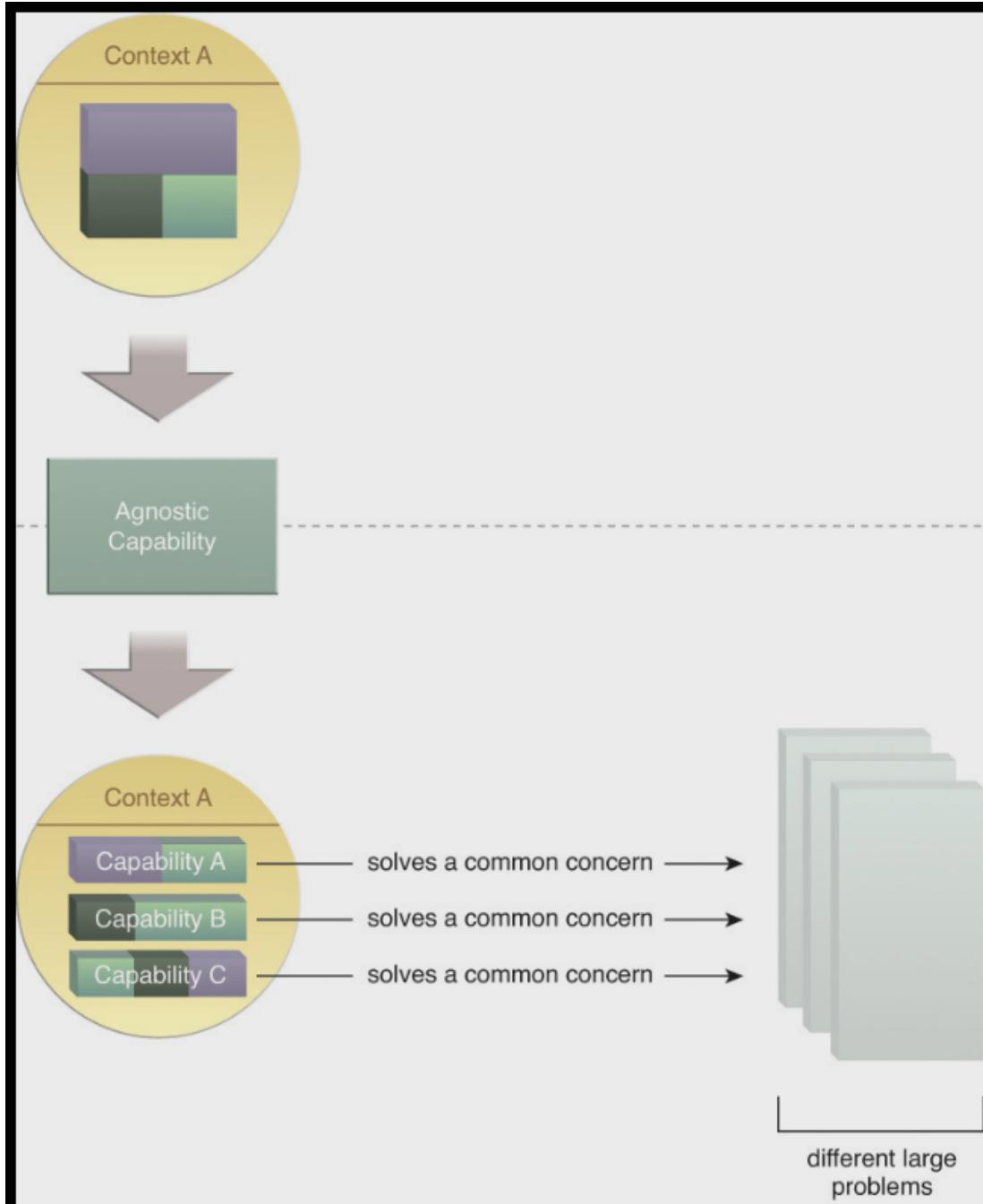
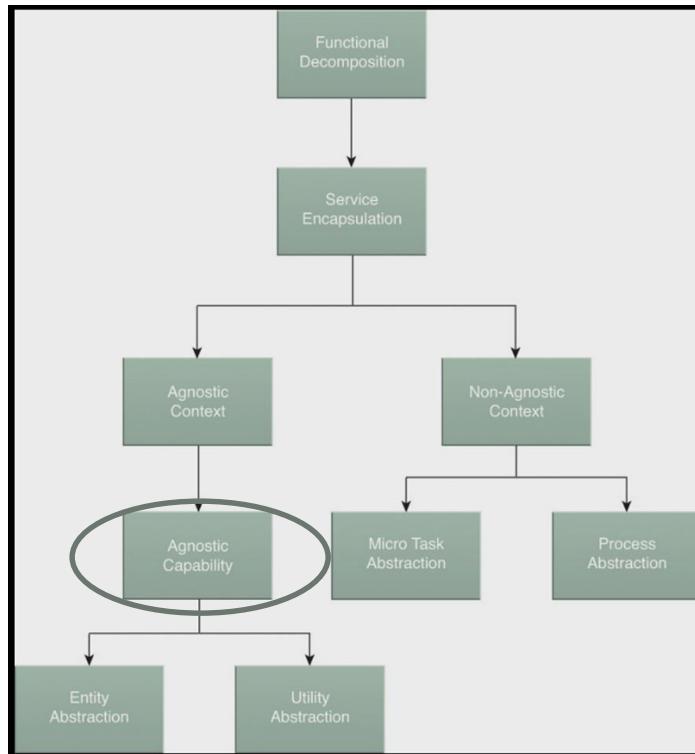
Service Encapsulation



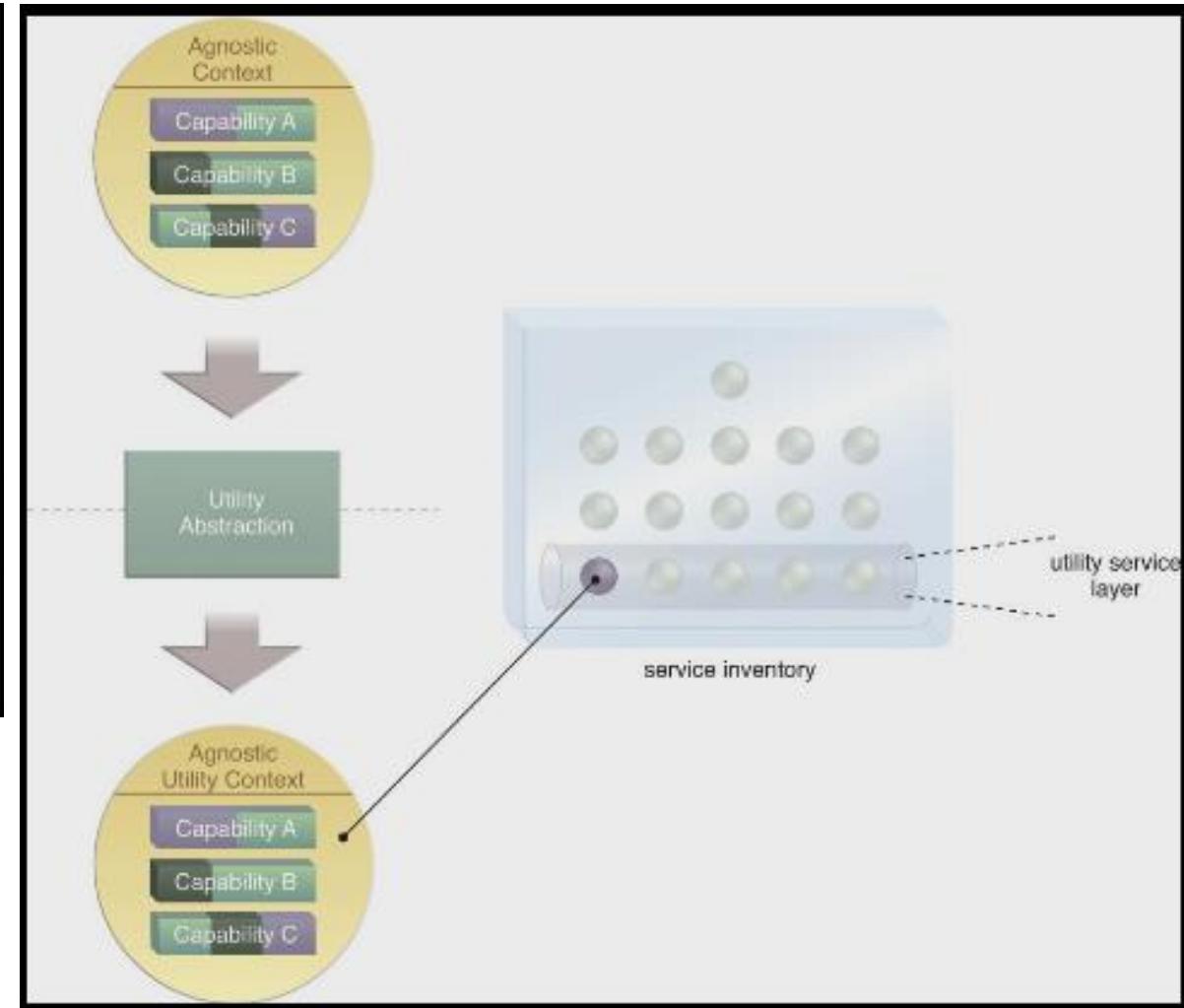
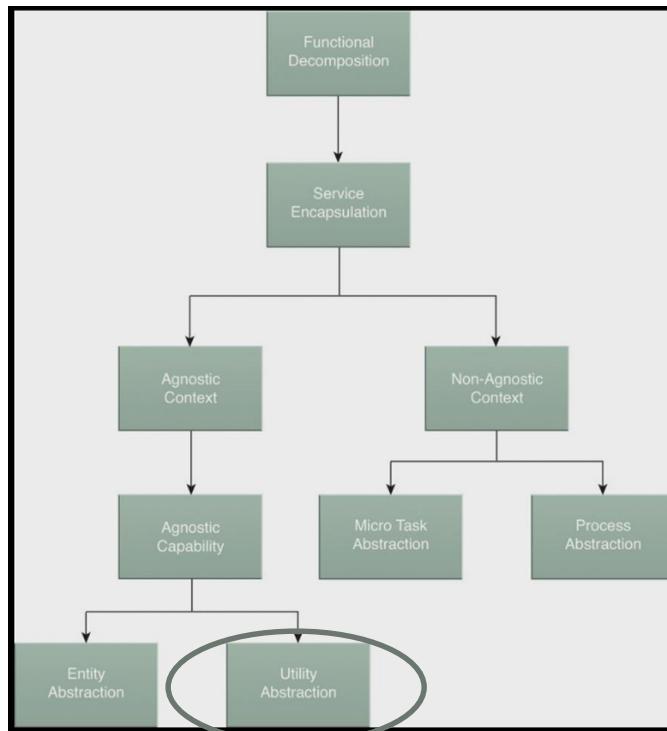
Agnostic context



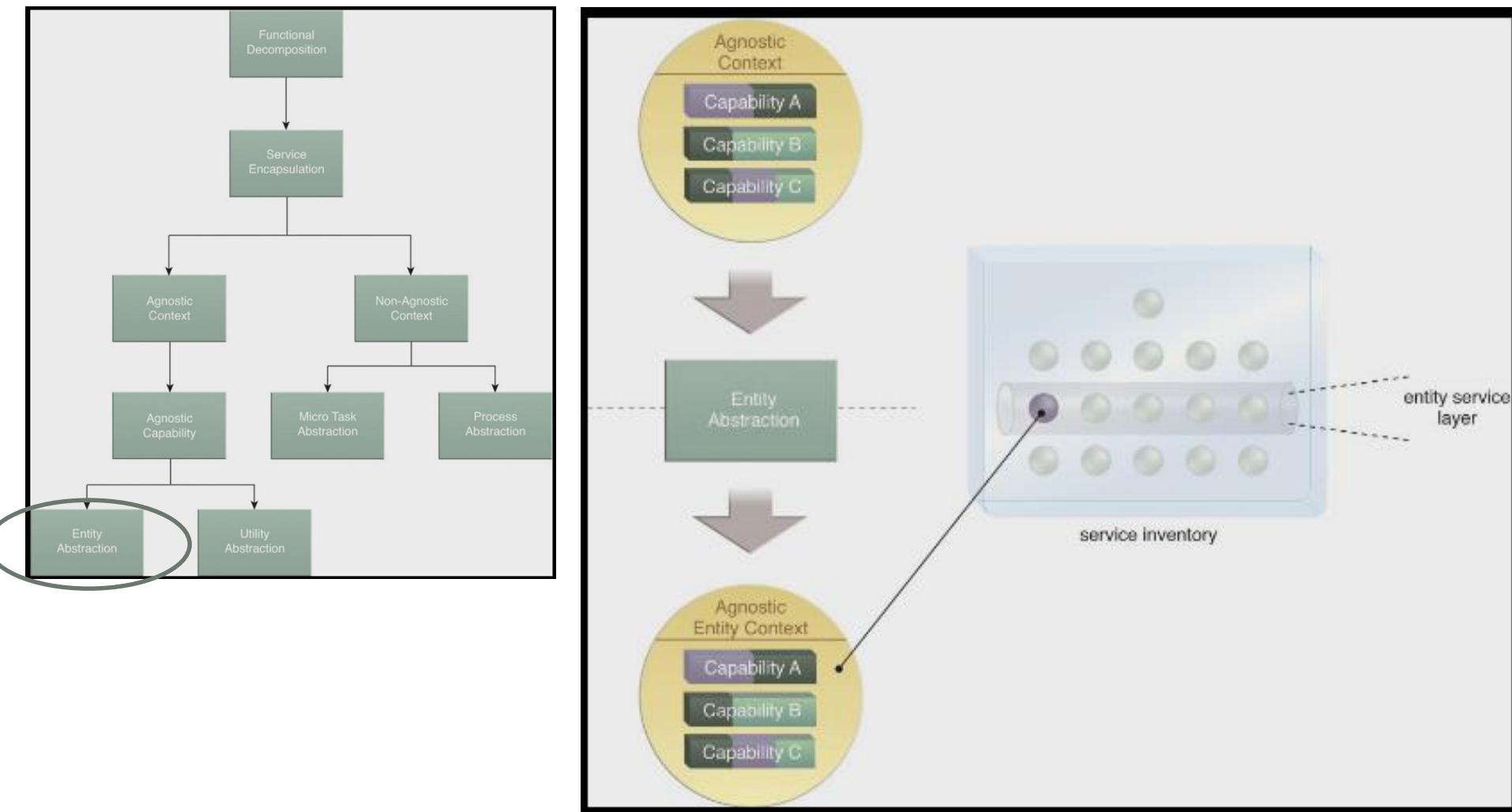
Agnostic capability



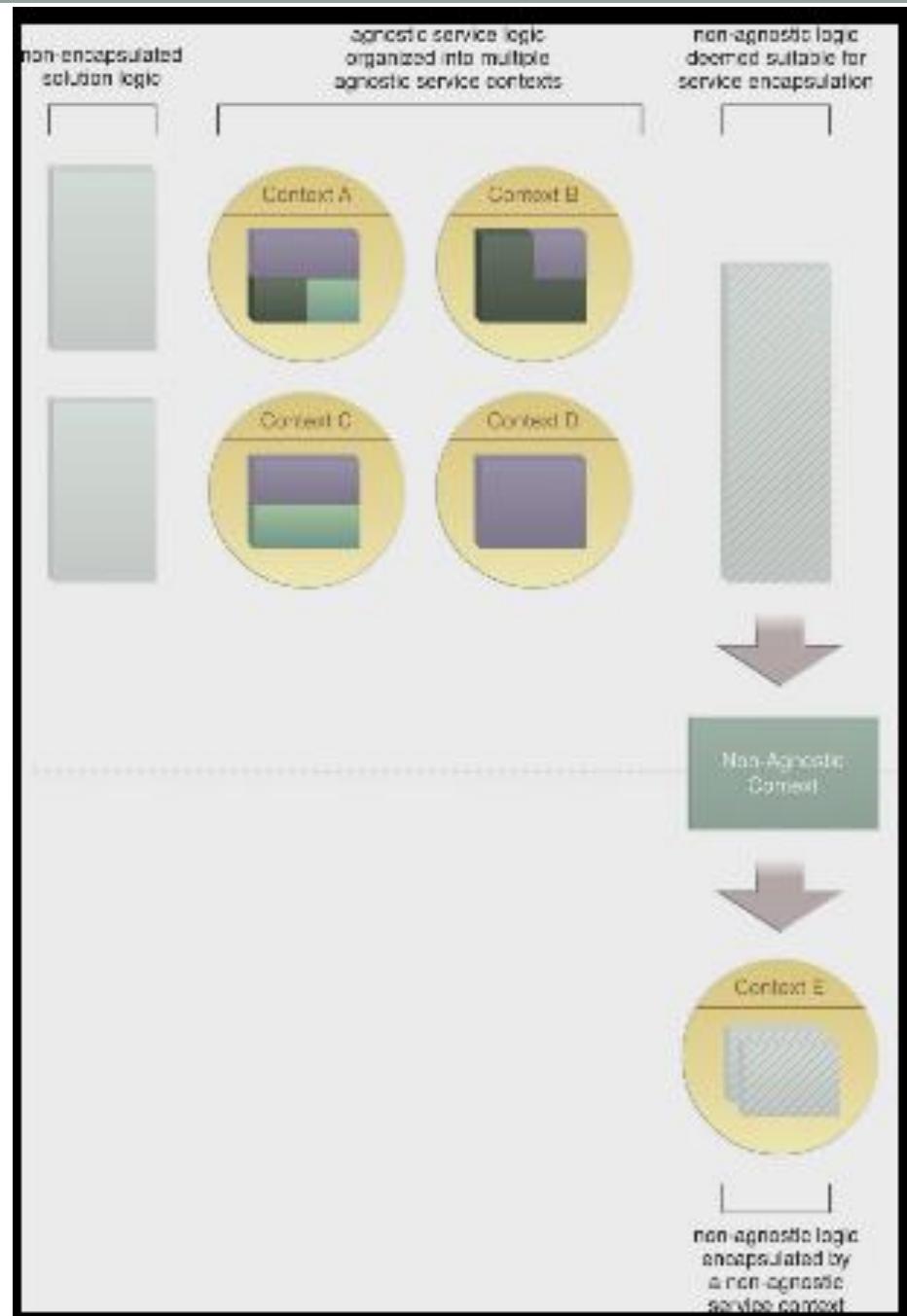
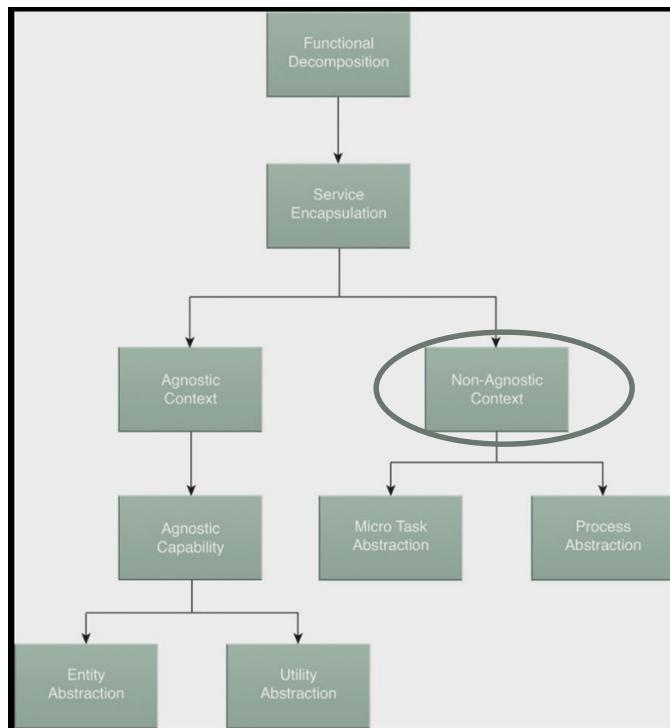
Utility Abstraction



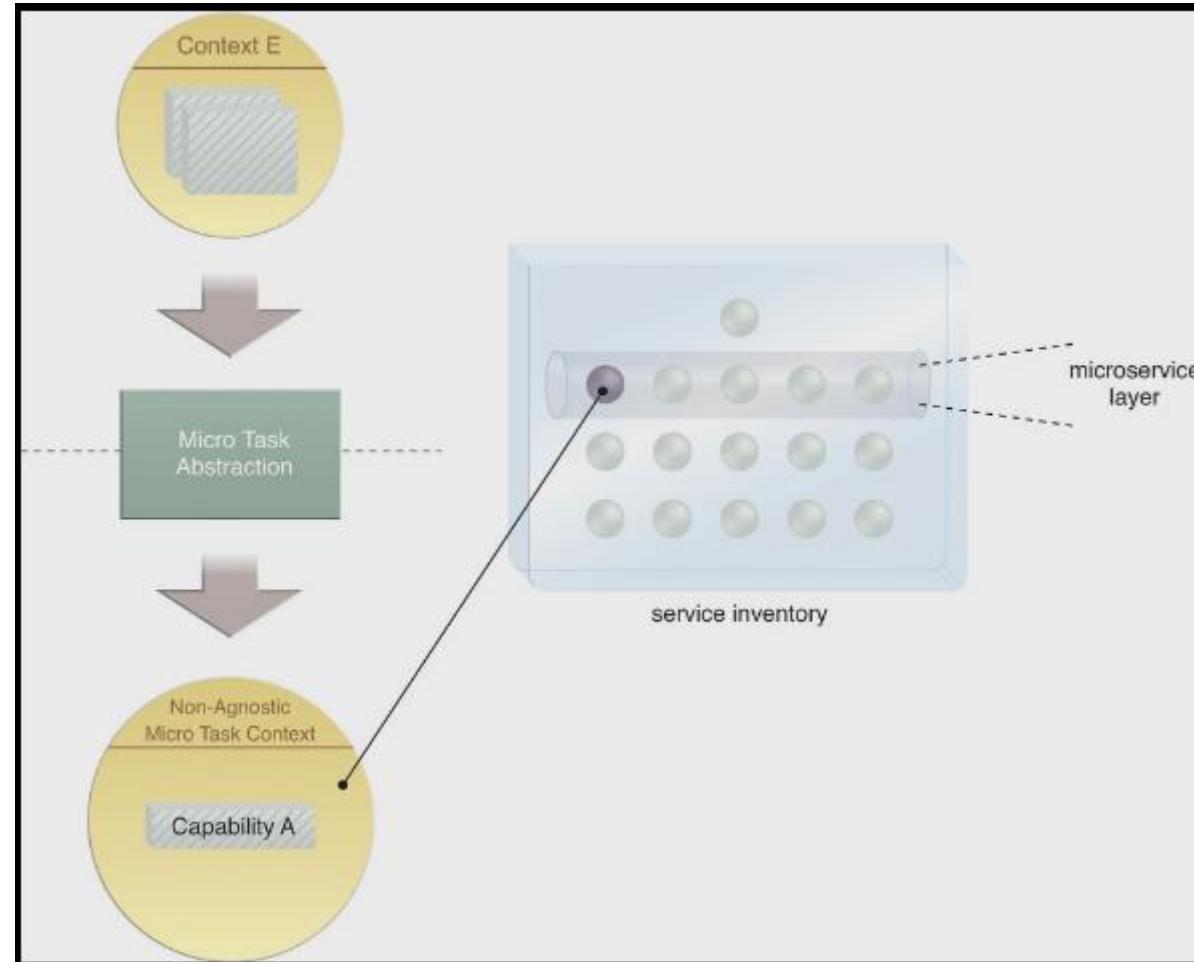
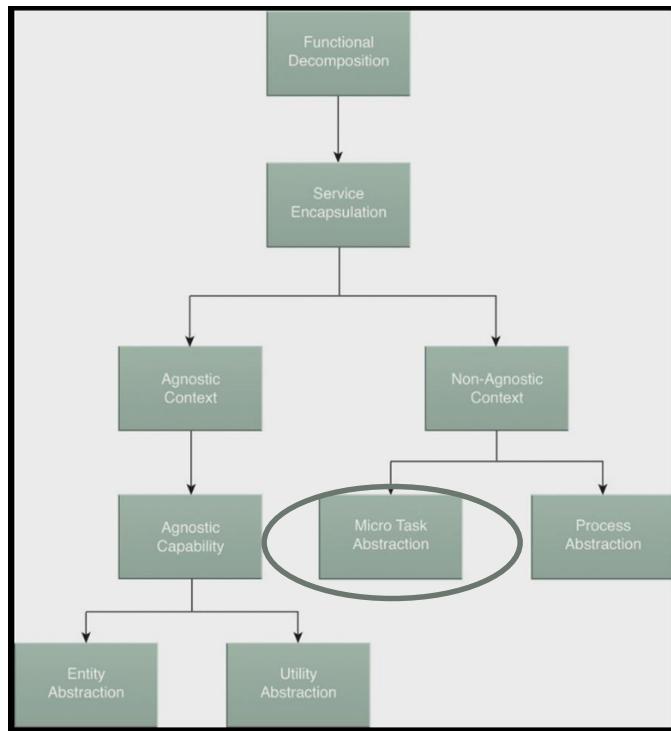
Entity Abstraction



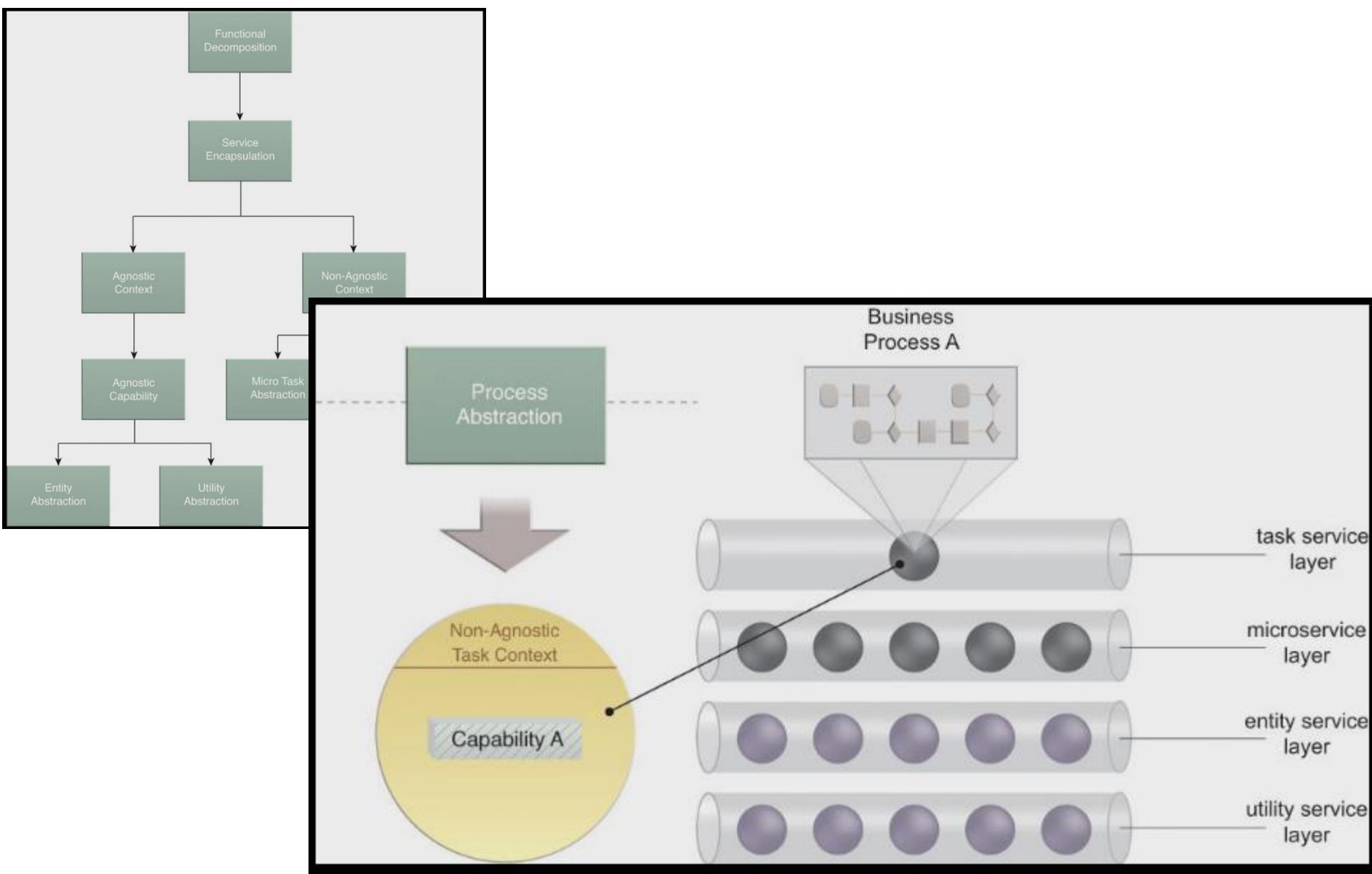
Non-agnostic context



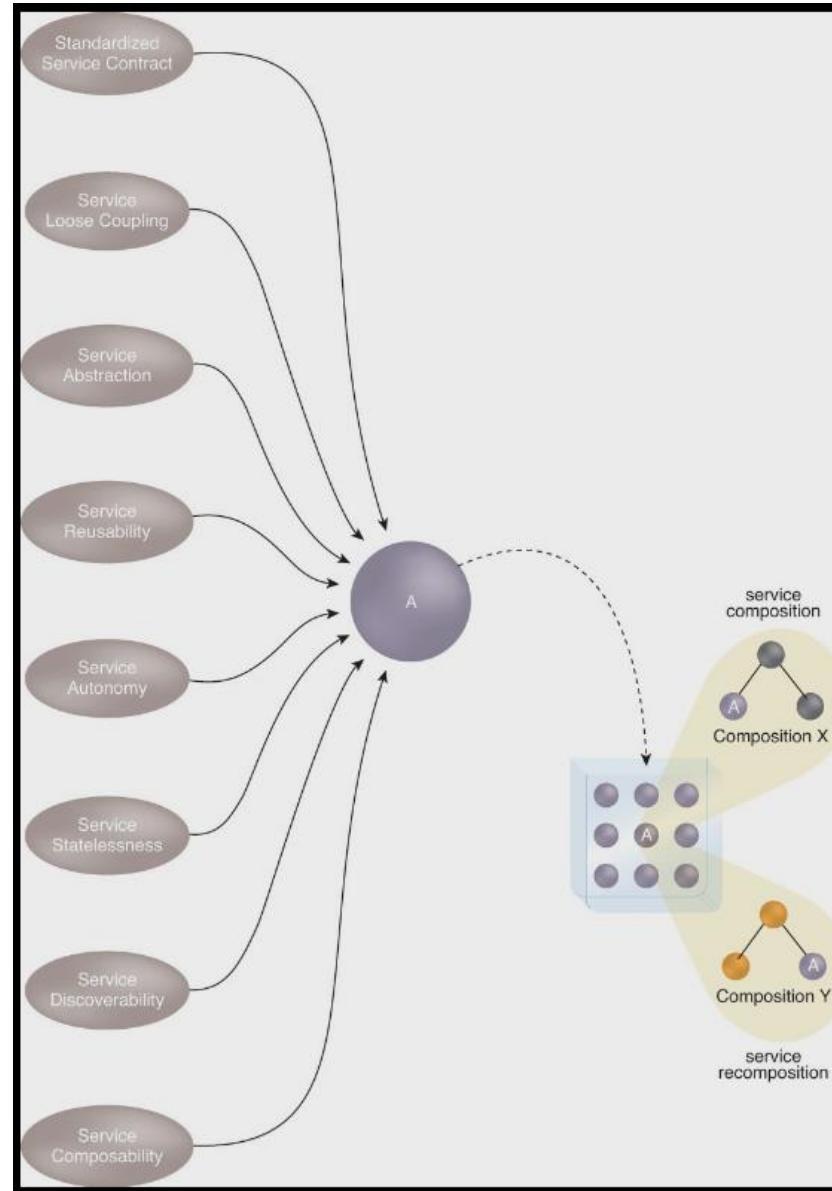
Micro Task abstraction and microservices



Process abstraction and Task services

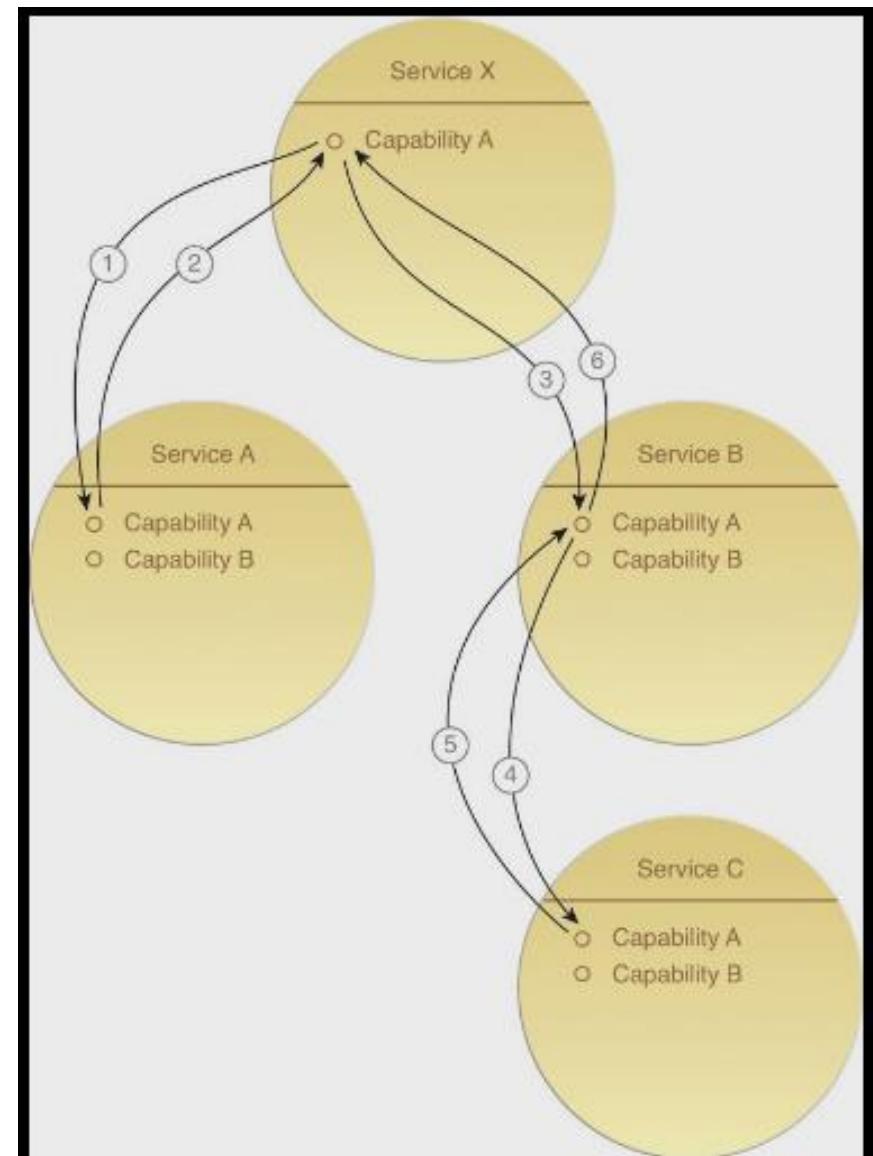


Building up the service-based solution

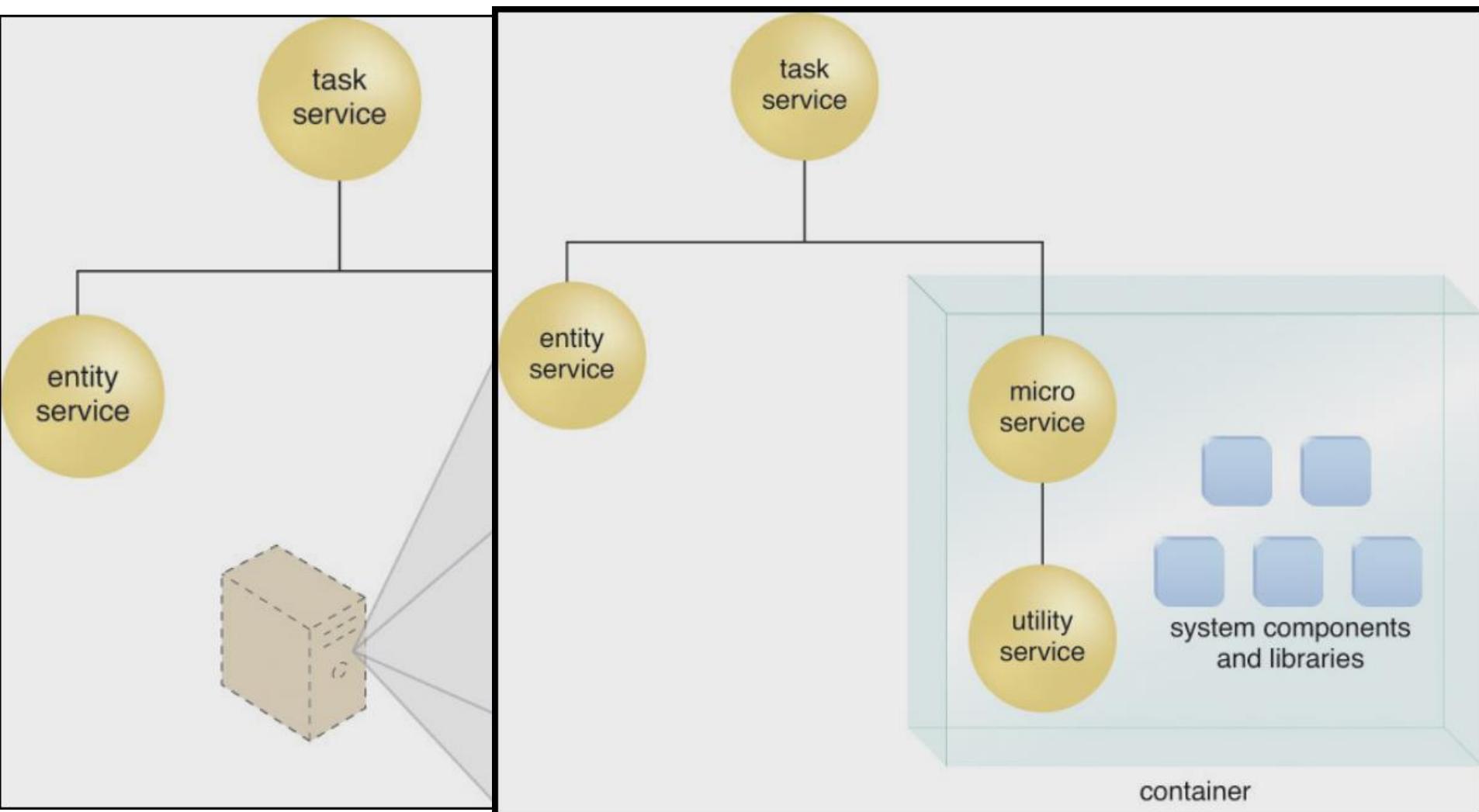


Capability composition

- Candidate service capabilities are sequenced together in order to assemble the decomposed service logic into a specific service composition that is capable of solving a specific larger problem
- Much of the logic that determines which service capabilities to invoke and in which order they are to be composed will usually reside within the task service.



Capability composition and microservices

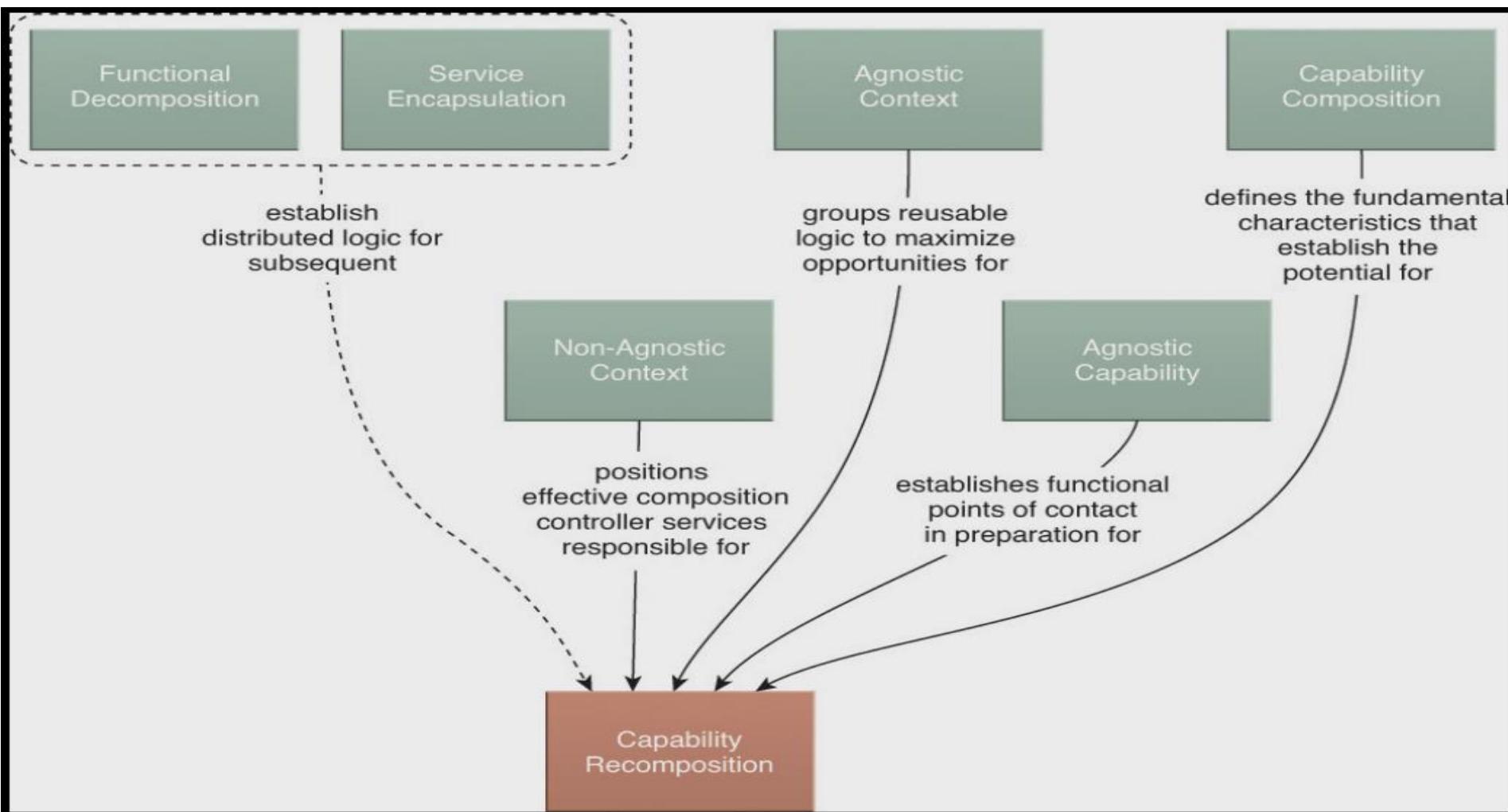


Variations

- Services packaged in the same deployment bundle may be able to **communicate in-process or out-of-process**.
- The **microservice** in the preceding scenarios may compose the utility service to **access an underlying resource** or it may disregard the Service Loose Coupling principle and access the underlying resource directly.
- **Multiple deployment bundles** can be located on the same virtual server, as long as respective **autonomy requirements** can be fulfilled.
- In previous Figure, the **container is located on a physical server**, but it can also be located on a virtual server.
- **A container can host multiple deployment bundles**, which may be desirable if communication between services and resources in the respective bundles is required.

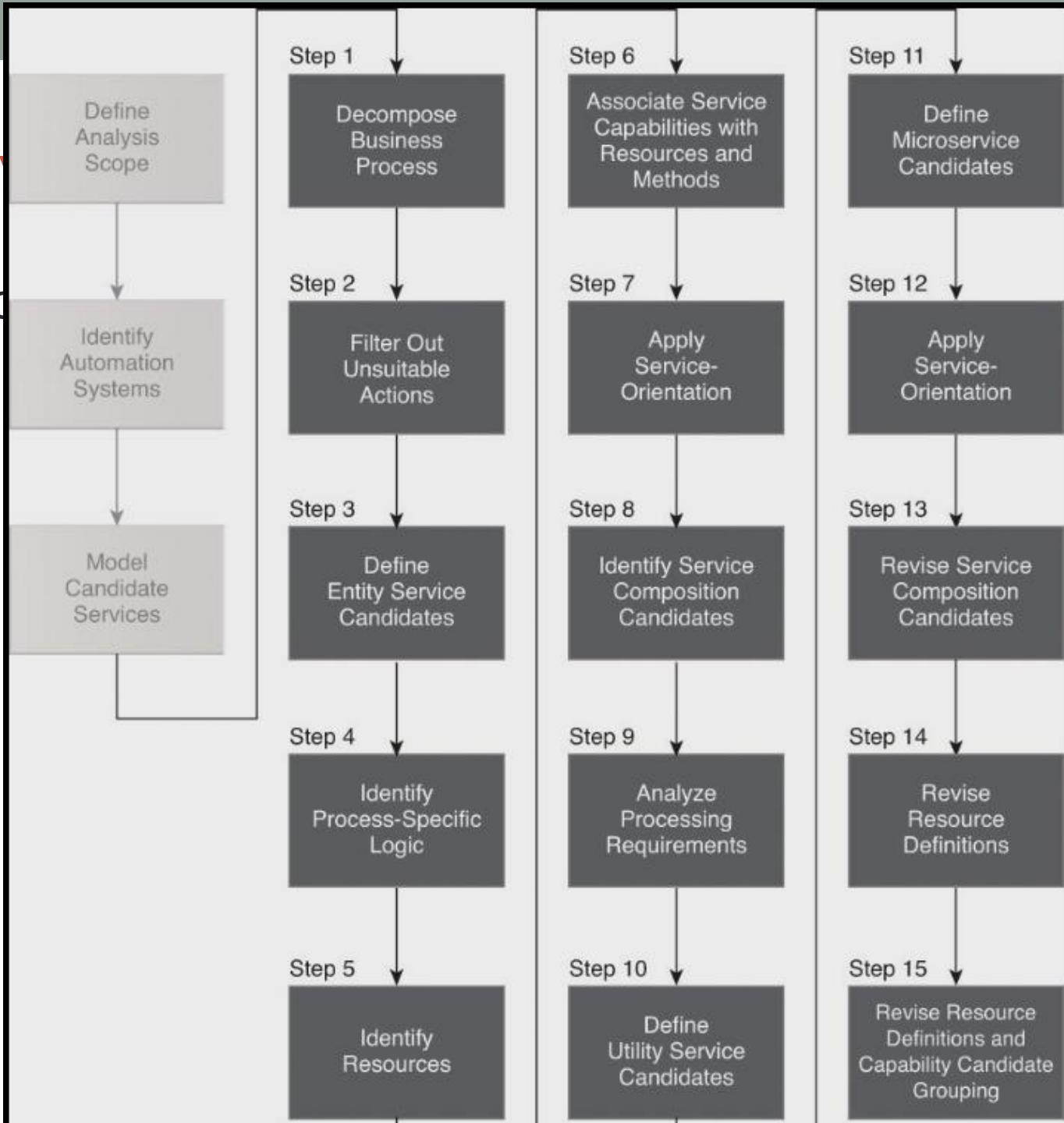
Capability recomposition

- addresses the recurring involvement of a service via the repeated composition of a service capability.



REST services

- incorporation of REST services

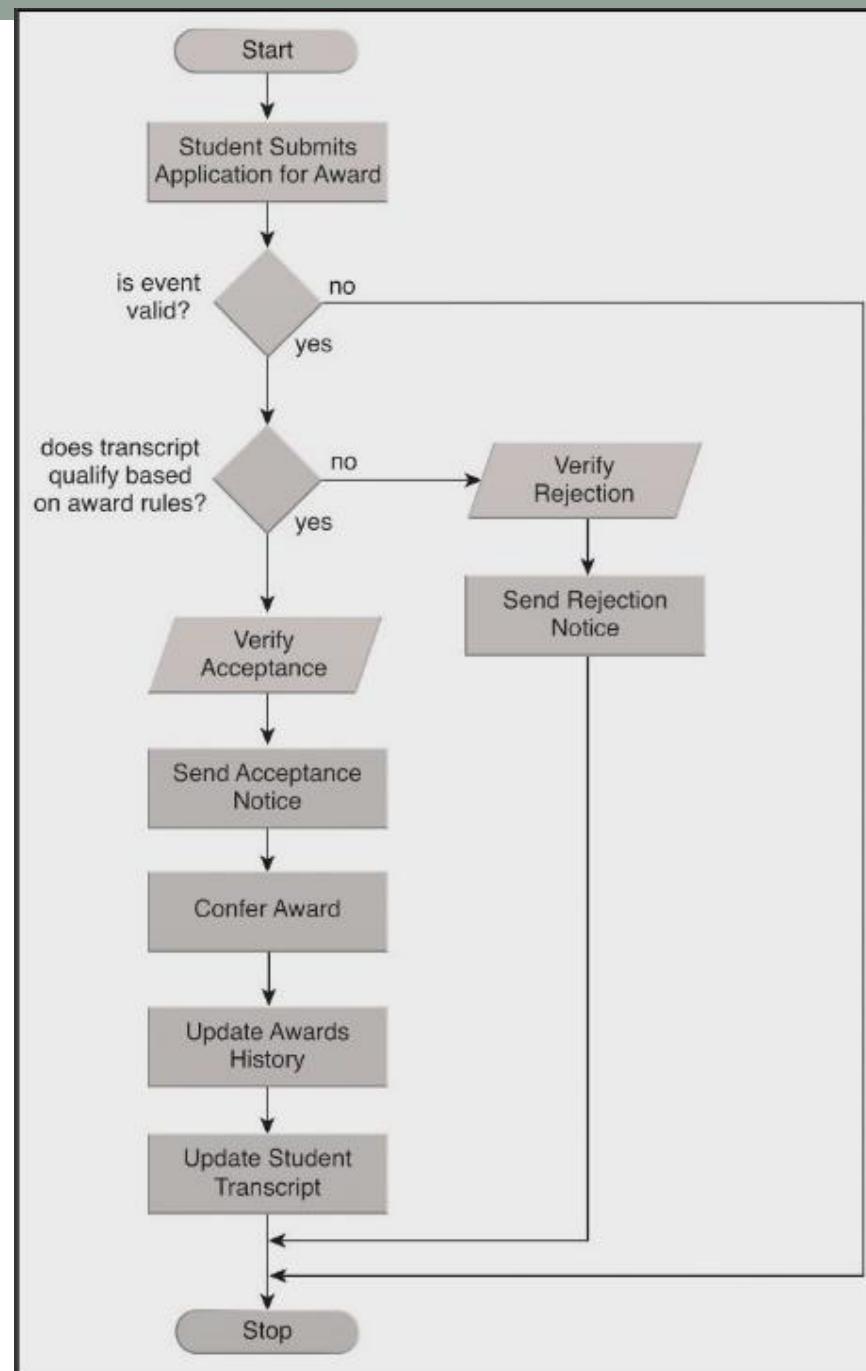


Case study – Midwest University Assoc. (MUA)

- six remote locations along with its main campus that employ more than 6,000 faculty and staff.
 - each program within the university has an independent IT staff and budget to support systems management. The remote campuses also have their own IT departments.
 - various automated solutions for common processes, such as student enrollment, course cataloging, accounting, financials, as well as grading and reporting.
 - primary system for record keeping is an IBM mainframe that is reconciled every night with a batch feed from the individual remote locations.
 - different schools themselves employ a variety of technologies and platforms.
-
- After a careful assessment of the existing infrastructure, it is decided to re-engineer several IT systems to a service-oriented architecture that will preserve legacy assets, simplify integration between various internal and external systems, and improve channel experience for both the students and staff. The enterprise architecture group at MUA has proposed a phased adoption of SOA via the use of REST services that can be leveraged across schools and from remote locations.

MUA – solution considerations

- focus on **entity services** that track the information assets of the various campuses. This initial set of services is to be deployed on the main campus first, so that IT staff can monitor maintenance requirements. Individual campuses are then to build solutions based on the same centralized service inventory.
- Solutions that introduce new **task services** will be allocated to virtual machines in the main campus to allow them to be moved to independent hardware and onto dedicated server farms, if the need arises in the future.
- The team begins with a REST service modeling process for the Student Achievement Award Conferral business process.
 - this business process logic represents the procedures followed for the assessment, conference, and rejection of individual achievement award applications submitted by students.
 - An application that is approved results in the conferral of the achievement award and a notification of the conferral to the student. An application that is rejected results in a notification of the rejection to the student.



Step 1: Decompose Business Process (into Granular Actions)

- Initiate Conferral Application
- Get Event Details
- Verify Event Details
- If Event is Invalid or Ineligible for Award, End Process
- Get Award Details
- Get Student Transcript
- Verify Student Transcript Qualifies for Award Based on Award Conferral Rules
- If Student Transcript Does Not Qualify, Initiate Rejection
- Manually Verify Rejection
- Send Rejection Notice
- Manually Verify Acceptance
- Send Acceptance Notice
- Confer Award
- Record Award Conferral in Student Transcript
- Record Award Conferral in Awards Database
- Print Hard Copy of Award Conferral Record
- File Hard Copy of Award Conferral Record

Step 2: Filter out unsuitable actions

- Initiate Conferral Application
- Get Event Details
- Verify Event Details
- If Event is Invalid or Ineligible for Award, End Process
- Get Award Details
- Get Student Transcript
- Verify Student Transcript Qualifies for Award Based on Award Conferral Rules
- If Student Transcript Does Not Qualify, Initiate Rejection
- Manually Verify Rejection
- Send Rejection Notice
- Manually Verify Acceptance
- Send Acceptance Notice
- Confer Award
- Record Award Conferral in Student Transcript
- Record Award Conferral in Awards Database
- Print Hard Copy of Award Conferral Record
- File Hard Copy of Award Conferral Record

Step 3: Define Entity service candidates

- Which service capability candidates defined so far are closely related to each other?

=> group capability candidates based on common functional contexts

- Are identified service capability candidates business-centric or utility-centric?

=> the organization of service candidates within logical service layers (entity and utility)

- What types of functional service contexts are suitable, given the overarching business context of the service inventory?

=> establish functional service boundaries

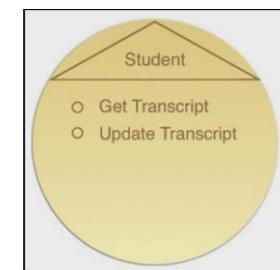
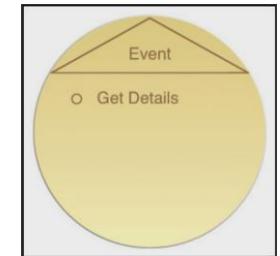
Analysis results

NON-AGNOSTIC

- Initiate Conferral Application
- Get Event Details
- Verify Event Details
- If Event is Invalid or Ineligible for Award, End Process
- Get Award Details
- Get Student Transcript
- Verify Student Transcript Qualifies for Award Based on Award Conferral Rules
- If Student Transcript Does Not Qualify, Initiate Rejection
- Send Rejection Notice
- Send Acceptance Notice
- Record Award Conferral in Student Transcript
- Record Award Conferral in Awards Database
- Print Hard Copy of Award Conferral Record

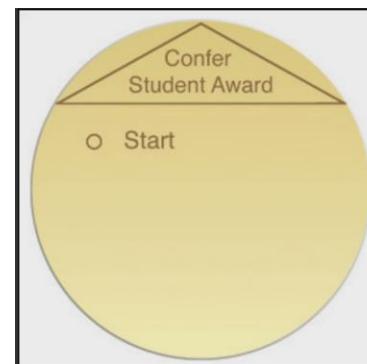
Agnostic actions -> entity service candidates

- Get Event Details
- Get Award Details
- Get Student Transcript
- Send Rejection Notice
- Send Acceptance Notice
- Record Award Conferral in Student Transcript
- Record Award Conferral in Awards Database
- Print Hard Copy of Award Conferral Record



Step 4: identify process-specific logic

- Non-agnostic capabilities
- Grouped in a task service/consumer service (composition controller)
 - Initiate Conferral Application
 - Verify Event Details
 - If Event is Invalid or Ineligible for Award, End Process
 - Verify Student Transcript Qualifies for Award Based on Award Conferral Rules
 - If Student Transcript Does Not Qualify, Initiate Rejection



Step 5: identify resources

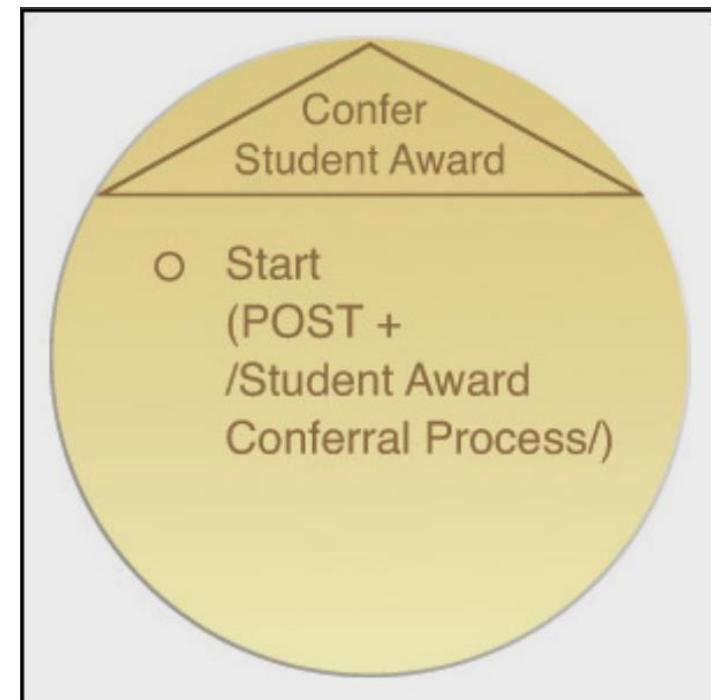
- Agnostic (multi-purpose)
 - can be incorporated into agnostic service and capability candidates without limitation.
 - likely to be shared and reused more frequently
- Non-agnostic (single-purpose)
- Case-study
 - /Process/ -> /Student Award Conferral Process/
 - /Application/ -> /Conferral Application/
 - /Event/
 - /Award/
 - /Student Transcript/
 - /Notice Sender/
 - /Printer/

Resource mapping

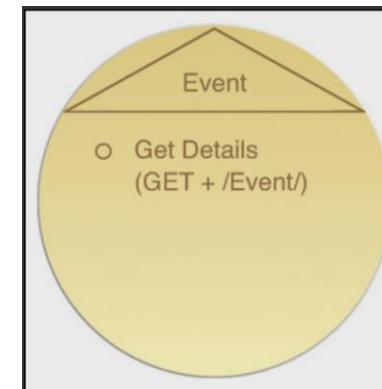
Entity	Resource
Event	/Event/
Award	/Award/
Student	/Student Transcript/

Step 6: Associate Service Capabilities with Resources and Methods

- Confer Student Award Service Candidate (Task)
 - /Application/ resource
 - POST method to forward the application to a resource named after the business process itself



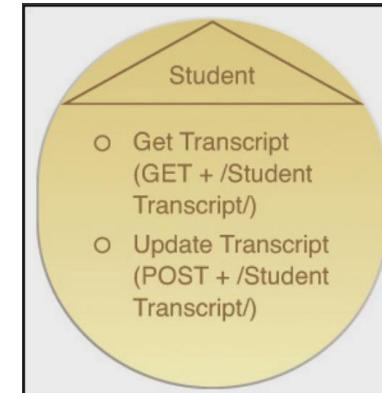
- Event service candidate (Entity)



- Award service candidate (Entity)



- Student service candidate (Entity)

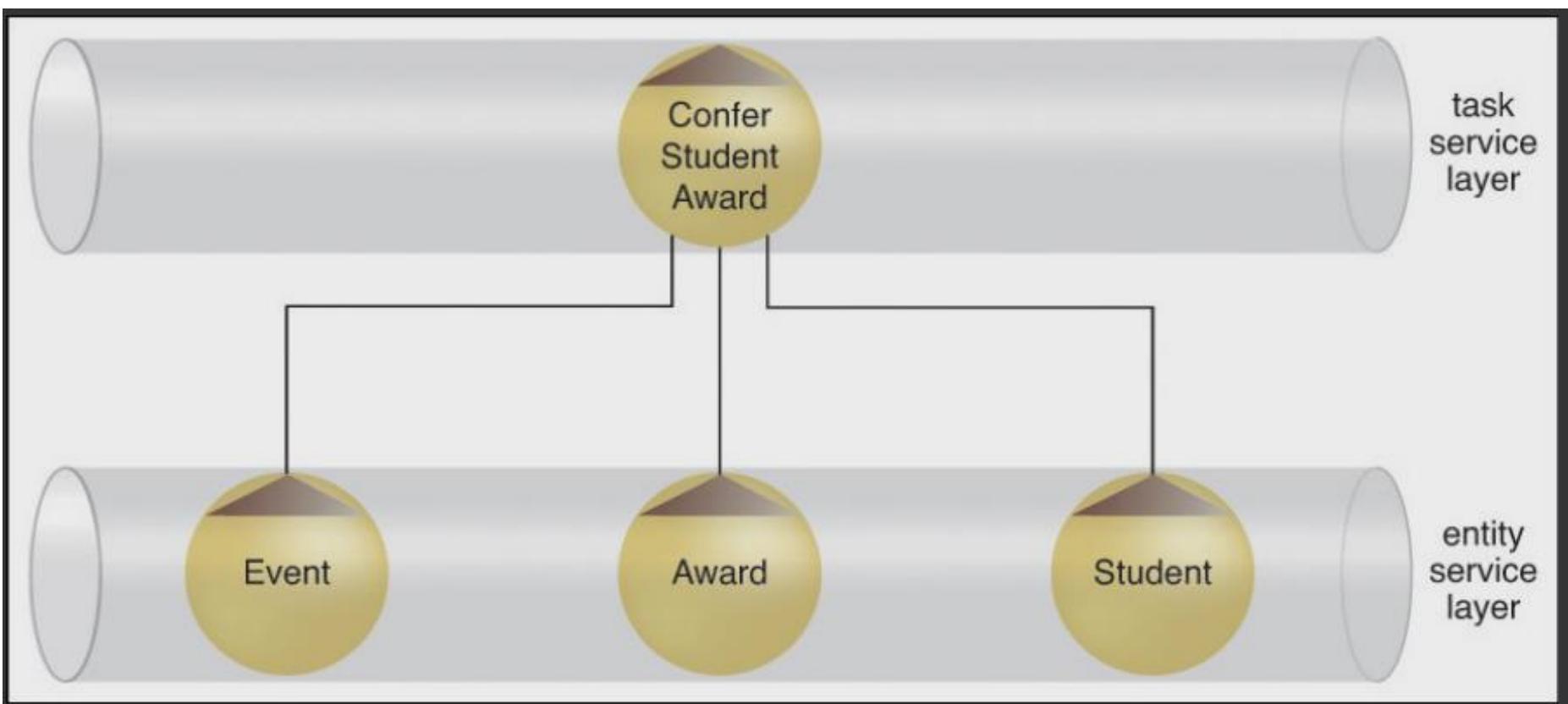


Step 7: Apply service orientation

- Ex. a given set of resources is related to data provided by a large legacy system => impacts functional service boundaries by the extent to which the Service Autonomy (297) principle can be applied.

Step 8: identify service composition candidates

- Identify most common service capability interactions based on success and failure scenarios



Step 9: Analyze processing requirements

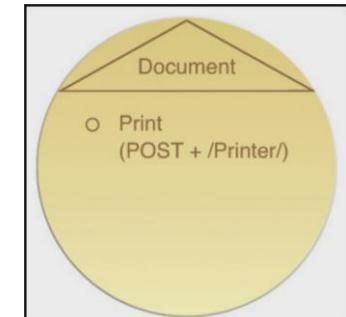
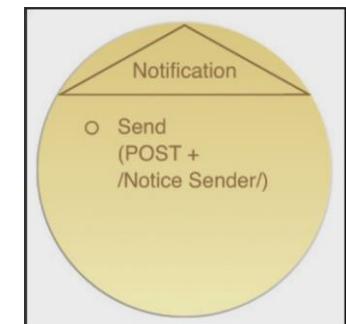
- Which of the resources identified so far can be considered utility-centric?
- Can actions performed on business-centric resources be considered utility-centric (such as reporting actions)?
- What underlying application logic needs to be executed in order to process the actions and/or resources encompassed by a service capability candidate?
- Does any required application logic already exist?
- Does any required application logic span application boundaries? (In other words, is more than one system required to complete the action?)

Case Study

- Utility centric functions
 - Send Rejection Notice
 - Send Acceptance Notice
 - Print Hard Copy of Award Conferral Record
 - Utility centric resources
 - /Notice Sender/
 - /Printer/
 - External Processes
 - Rules utility service (usually overloaded)
- => Verify Student Transcript Qualifies for Award Based on Award Conferral Rules -> microservice

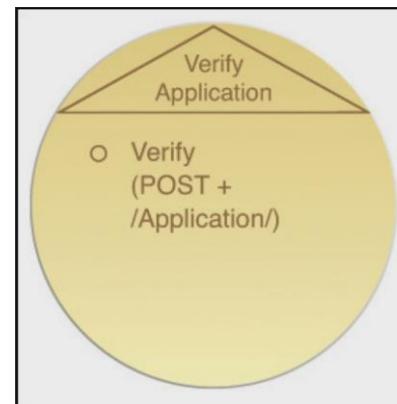
Step 10: Define Utility Service Candidates (and Associate Resources and Methods)

- group utility-centric processing steps according to pre-defined contexts.
 - Association with a specific legacy system
 - Association with one or more solution components
 - Logical grouping according to type of function
- Notification Service candidate
 - Send Rejection Notice
 - Send Acceptance Notice
 - /Notice sender/
- Document service candidate
 - Print Hard Copy of Award Conferral Record
 - /Printer/



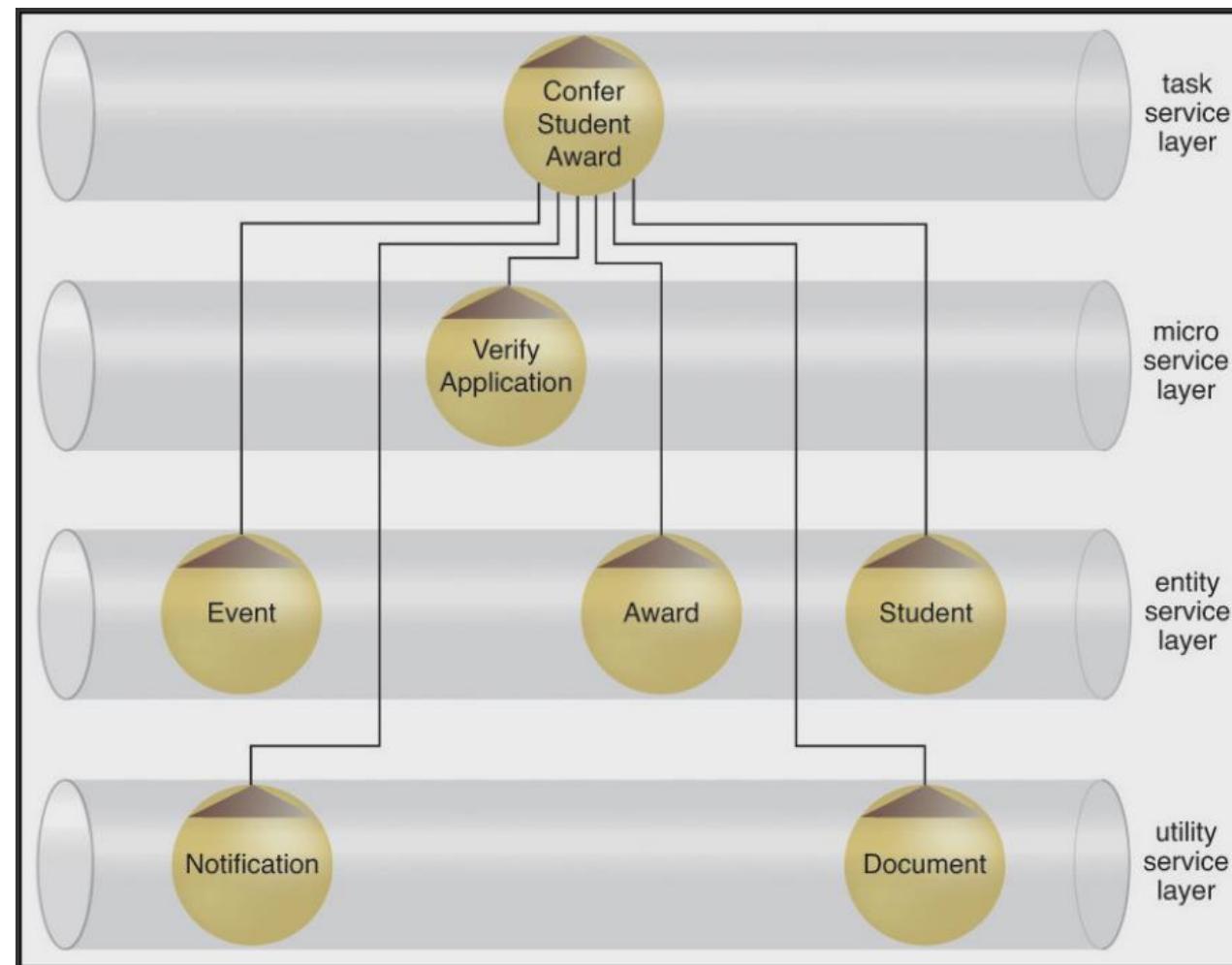
Step 11: Define Microservice Candidates (and Associate Resources and Methods)

- introduce a highly independent and autonomous service implementation architecture that can be suitable for units of logic with particular processing demands.
- considerations can include:
 - Increased autonomy requirements
 - Specific runtime performance requirements
 - Specific runtime reliability or failover requirements
 - Specific service versioning and deployment requirements



Step 12: Apply service-orientation

Step 13: Revise candidate service compositions



Additional considerations

- Uniform Contract Modeling
 - standardizing a number of aspects pertaining to service capability representation, data representation, message exchange, and message processing.
- REST Service Inventory Modeling
 - A service inventory is a collection of services that are independently owned, governed, and standardized.
- Business information:
 - Understanding the **types of information and documents** that will need to be exchanged and processed can help define necessary media types.
 - Understanding the **service models (entity, utility, task, etc.)** in use by service candidates can help determine which available methods should be supported.
 - Understanding **policies and rules** that are required to regulate certain types of interaction can help determine when certain methods should not be used, or help define special features that may be required by some methods.
 - Understanding how service capability candidates may need to be **composed** can help determine suitable methods.
 - Understanding certain **quality-of-service requirements** (especially in relation to reliability, security, transactions, etc.) can help determine the need to support special features of methods, and may further help identify the need to issue a set of pre-defined messages

REST Constraints and Uniform Contract Modeling

- Stateless – From the data exchange requirements we are able to model between service candidates, can we determine whether services will be able to remain stateless between requests?
- Cache – Are we able to identify any request messages with responses that can be cached and returned for subsequent requests instead of needing to be processed redundantly?
- Uniform Contract – Can all methods and media types we are associating with the uniform contract during this stage be genuinely reused by service candidates?
- Layered System – Do we know enough about the underlying technology architecture to determine whether services and their consumers can tell the difference between communicating directly or communicating via intermediary middleware?

Resources vs. Entities

- “things” that need to be accessed and processed by service consumers.
- Differences:
 - Entities are business-centric and are derived from enterprise business models, such as entity relationship diagrams, logical data models, and ontologies.
 - Resources can be business-centric or non-business-centric. A resource is any given “thing” associated with the business automation logic enabled by the service inventory.
 - Entities are commonly limited to business artifacts and documents, such as invoices, claims, customers, etc.
 - Some entities are more coarse-grained than others. Some entities can encapsulate others. For example, an invoice entity may encapsulate an invoice detail entity.
 - Resources can also vary in granularity, but are often fine-grained. It is less common to have formally defined coarse-grained resources that encapsulate fine-grained resources.
 - All entities can relate to or be based on resources. Not all resources can be associated with entities because some resources are non-business-centric.

SOA Patterns

- Requirement – A requirement is a concise, single-sentence statement that presents the fundamental requirement addressed by the pattern in the form of a question. Every pattern description begins with this statement.
- Icon – Each pattern description is accompanied by an icon image that acts as a visual identifier. The icons are displayed together with the requirement statements in each pattern profile.
- Problem – The issue causing a problem and the effects of the problem. It is this problem for which the pattern is expected to provide a solution.
- Solution – This represents the design solution proposed by the pattern to solve the problem and fulfill the requirement.
- Application – This part is dedicated to describing how the pattern can be applied. It can include guidelines, implementation details, and sometimes even a suggested process.
- Impacts – This part highlights common consequences, costs, and requirements associated with the application of a pattern and may also provide alternatives that can be considered.
- Principles – References to related service-orientation principles.
- Architecture – References to related SOA architecture types.

Agnostic capability

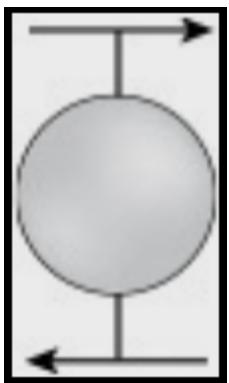
How can multi-purpose service logic be made effectively consumable and composable?



Problem	Service capabilities derived from specific concerns may not be useful to multiple service consumers, thereby reducing the reusability potential of the agnostic service.
Solution	Agnostic service logic is partitioned into a set of well-defined capabilities that address common concerns not specific to any one problem. Through subsequent analysis, the agnostic context of capabilities is further refined.
Application	Service capabilities are defined and iteratively refined through proven analysis and modeling processes.
Impacts	The definition of each service capability requires extra up-front analysis and design effort.
Principles	Standardized Service Contract (291), Service Reusability (295), Service Composability (302)
Architecture	Service

Agnostic context

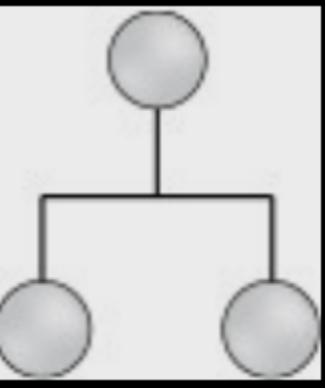
How can multi-purpose service logic be positioned as an effective enterprise resource?



Problem	Multi-purpose logic grouped together with single purpose logic results in programs with little or no reuse potential that introduce waste and redundancy into an enterprise.
Solution	Isolate logic that is not specific to one purpose into separate services with distinct agnostic contexts.
Application	Agnostic service contexts are defined by carrying out service-oriented analysis and service modeling processes.
Impacts	This pattern positions reusable solution logic at an enterprise level, potentially bringing with it increased design complexity and enterprise governance issues.
Principles	Service Reusability (295)
Architecture	Service

Capability composition

How can a service capability solve a problem that requires logic outside of the service boundary?

	Problem A capability may not be able to fulfill its processing requirements without adding logic that resides outside of its service's functional context, thereby compromising the integrity of the service context and risking service denormalization.
Solution	When requiring access to logic that falls outside of a service's boundary, capability logic within the service is designed to compose one or more capabilities in other services.
Application	The functionality encapsulated by a capability includes logic that can invoke other capabilities from other services.
Impacts	Carrying out composition logic requires external invocation, which adds performance overhead and decreases service autonomy.
Principles	All
Architecture	Inventory, Composition, Service

Entity abstraction

How can agnostic business logic be separated, reused, and governed independently?



Problem	Bundling both process-agnostic and process-specific business logic into the same service eventually results in the creation of redundant agnostic business logic across multiple services.
Solution	An agnostic business service layer can be established, dedicated to services that base their functional context on existing business entities.
Application	Entity service contexts are derived from business entity models and then establish a logical layer that is modeled during the analysis phase.
Impacts	The core, business-centric nature of the services introduced by this pattern require extra modeling and design attention and their governance requirements can impose dramatic organizational changes.
Principles	Service Loose Coupling (293), Service Abstraction (294), Service Reusability (295), Service Composability (302)
Architecture	Inventory, Composition, Service

Utility abstraction

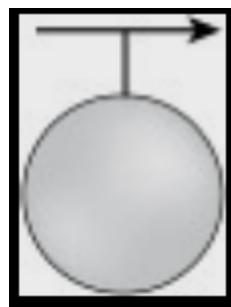
How can common non-business centric logic be separated, reused, and independently governed?



Problem	When non-business centric processing logic is packaged together with business-specific logic, it results in the redundant implementation of common utility functions across different services.
Solution	A service layer dedicated to utility processing is established, providing reusable utility services for use by other services in the inventory.
Application	The utility service model is incorporated into analysis and design processes in support of utility logic abstraction, and further steps are taken to define balanced service contexts.
Impacts	When utility logic is distributed across multiple services it can increase the size, complexity, and performance demands of compositions.
Principles	Service Loose Coupling (293), Service Abstraction (294), Service Reusability (295), Service Composability (302)
Architecture	Inventory, Composition, Service

Non-agnostic context

How can single-purpose service logic be positioned as an effective enterprise resource?



Problem	Non-agnostic logic that is not service-oriented can inhibit the effectiveness of service compositions that utilize agnostic services.
Solution	Non-agnostic solution logic suitable for service encapsulation can be located within services that reside as official members of a service inventory.
Application	A single-purpose functional service context is defined.
Impacts	Although they are not expected to provide reuse potential, non-agnostic services are still subject to the rigor of service-orientation.
Principles	Standardized Service Contract (291), Service Composability (302)
Architecture	Service

Conclusions

- detailed step-by-step process for modeling REST service candidates.
- Extended SOA patterns catalog at <http://soapatterns.org>
- Next time:
 - Class design principles

SOFTWARE DESIGN

Class Design Principles

Content

- SOLID Class Design Principles
- General Responsibility Assignment Principles (GRASP)

References

- Robert Martin
<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- [SOLID ebook](#)
- GRASP
 - Craig Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Ed, Addison Wesley, 2004 – Chapters 17, 18.
- Courses
 - B. Meyer (ETH Zurich)
 - R. Marinescu (Univ. Timisoara)
- <http://www.vogella.com/tutorials/DependencyInjection/article.html>

Martin's signs of poor design

- **Rigidity**
 - code difficult to change
- **Fragility**
 - code breaks in unexpected places
 - even small changes can cause cascading breaks
- **Immobility**
 - code is so tangled that it's impossible to **reuse** anything
- **Viscosity**
 - design viscosity:
 - much easier to hack than to preserve original design
 - “*easy to do the wrong thing, but hard to do the right thing*” (R.Martin)
 - environment viscosity
 - Long compile times
 - Long versioning control times...

Causes of Poor Design

- **Changing Requirements**
 - is inevitable
 - both better designs and poor designs have to face the changes;
 - good designs are stable
- **Dependency Management**
 - the issue of coupling and cohesion
 - It can be controlled!
 - create *dependency firewalls*

Class Design Principles

- Single Responsibility
- Open-Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

SOLID

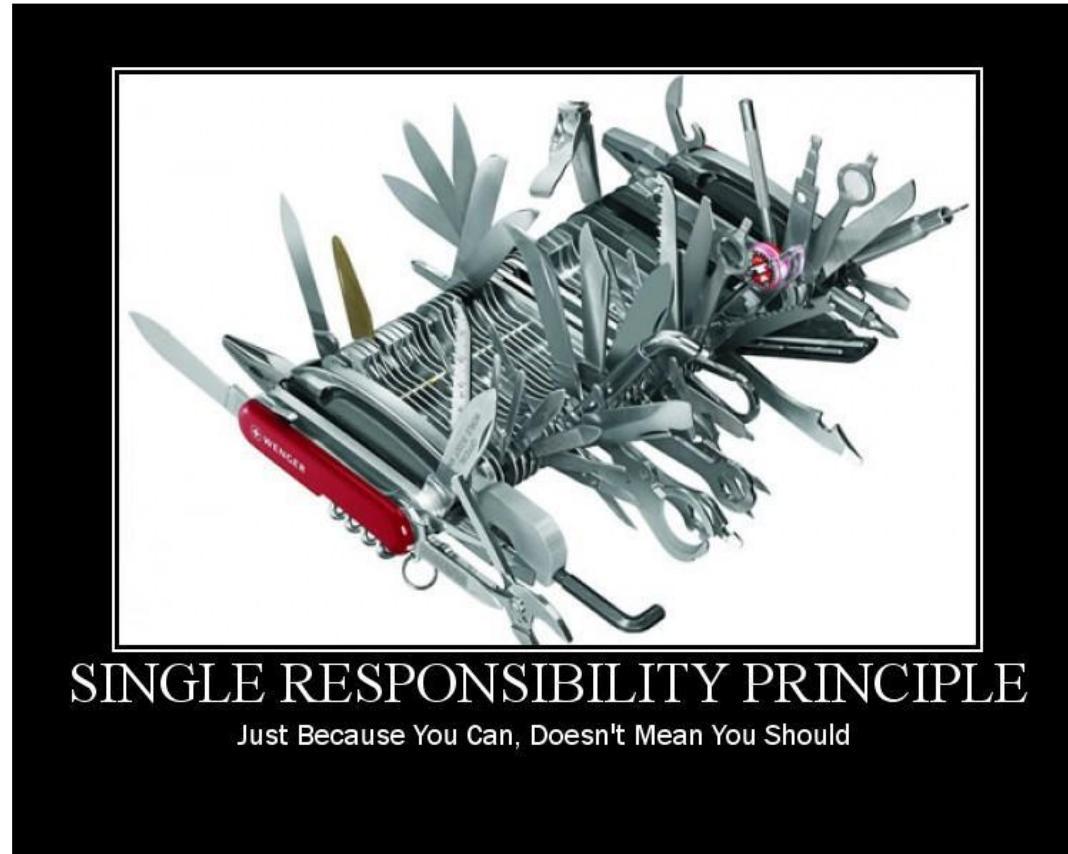


SOLID

Software Development is not a Jenga game

Single Responsibility

THERE SHOULD NEVER BE MORE THAN ONE REASON
FOR A CLASS TO CHANGE



Example

```
1: public abstract class BankAccount
2: {
3:     double Balance { get; }
4:     void Deposit(double amount) {}
5:     void Withdraw(double amount) {}
6:     void AddInterest(double amount) {}
7:     void Transfer(double amount, IBankAccount toAccount) {}
8: }
```

```
1: public abstract class BankAccount
2: {
3:     double Balance { get; }
4:     void Deposit(double amount);
5:     void Withdraw(double amount);
6:     void Transfer(double amount, IBankAccount toAccount);
7: }
```

```
8:
9: public class CheckingAccount : BankAccount
10: {
11: }
```

```
12:
13: public class SavingsAccount : BankAccount
14: {
15:     public void AddInterest(double amount);
16: }
```

Anti-example: Active Record

```
/* Active Record */
public class Student {
    private final UUID guid;
    private int studentID;
    private String name;
    private char grade;
    public Student() {
        this.guid = UUID.randomUUID();
    }
    public Student(UUID guid) {
        this.guid = guid;
    }
}
```

//Getters and Setters

```
public int getStudentId() {
    return studentID;
}
public void setStudentId(int studentID) {
    this.studentID = studentID;
}
```

//Domain Logic Methods

```
public boolean passes() {
    return grade != 'F';
}
public boolean onProbation() { return grade < 'D'; }
```

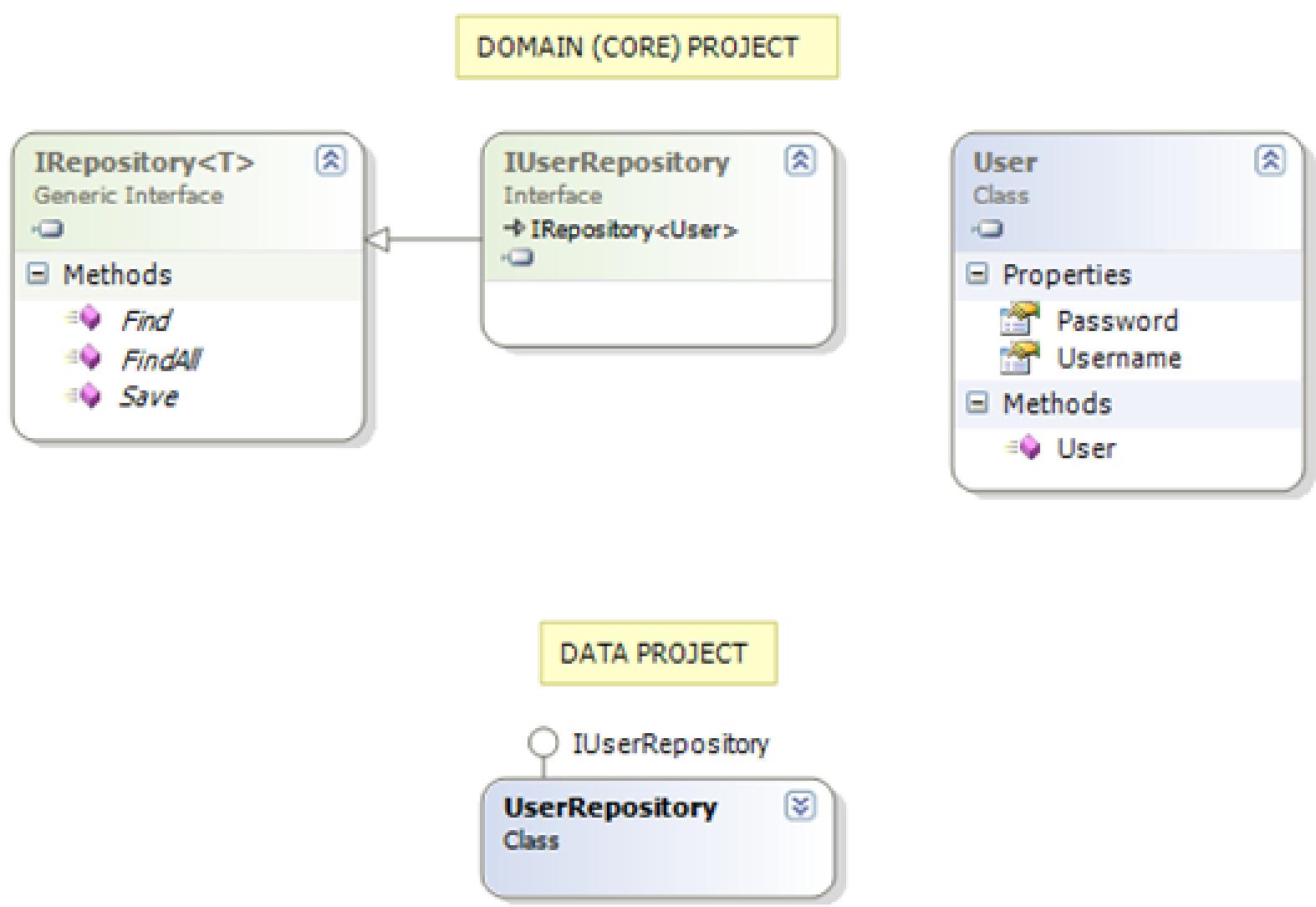
//SQL Operation Methods

```
public synchronized static Student findByGuid(UUID uniqueID)
throws ActiveRecordException {
    try {
        Connection db = /* You need an object which
is responsible for opening and closing connections */
        String statement = "SELECT 'guid', 'grade', 'studentID',
'name' FROM 'students' where 'guid'=?";
        PreparedStatement dbStatement =
db.prepareStatement(statement);
        dbStatement.setString(1, uniqueID.toString());
        ResultSet rs = dbStatement.executeQuery();
        while(rs.next()) {
            UUID guid = UUID.fromString(rs.getString("guid"));
            String name = rs.getString("name");
            char grade = rs.getString("grade").charAt(0);
            int studentID = rs.getInt("studentID");

            Student student = new Student(guid);
            student.name = name;
            student.grade = grade;

            student.studentID = studentID;
            return student;
        }
        return null;
    } catch (SQLException e) {
```

Refactor



Swiss army knives vs regular cutlery



```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void btnBrowse_Click(object sender, EventArgs e)
    {
        openFileDialog1.Filter = "XML Document (*.xml)| *.xml|All Files (*.*)| *.*";
        var result = openFileDialog1.ShowDialog();
        if (result == DialogResult.OK)
        {
            txtFileName.Text = openFileDialog1.FileName;
            btnLoad.Enabled = true;
        }
    }

    private void btnLoad_Click(object sender, EventArgs e)
    {
        listView1.Items.Clear();
        var fileName = txtFileName.Text;
        using (var fs = new FileStream(fileName, FileMode.Open))
        {
            var reader = XmlReader.Create(fs);
            while (reader.Read())
            {
                if(reader.Name != "product") continue;
                var id = reader.GetAttribute("id");
                var name = reader.GetAttribute("name");
                var unitPrice = reader.GetAttribute("unitPrice");
                var discontinued = reader.GetAttribute("discontinued");
                var item = new ListViewItem(
                    new string[]{id, name, unitPrice, discontinued});
            }
        }
    }
}
```

What's wrong with this?

```
<?xml version="1.0" encoding="utf-8" ?>
<products>
    <product id="1" name="IPod Nano" unitPrice="129.55" discontinued="false"/>
    <product id="2" name="IPod Touch" unitPrice="259.10" discontinued="false"/>
    <product id="3" name="IPod" unitPrice="78.95" discontinued="true"/>
</products>
```

Responsibilities

- Storing the model => Product class
- Loading data from the data source => Loader
- Mapping a single XML node to a product => Mapper
- Return a list of instances of type Product => ProductRepository
- Displaying data, delegating requests triggered by the user to a *controller* or *presenter* => MVC/MVP

Refactored design



Open-Closed Principle (OCP)

*Modules should be written so they can be extended without requiring them to be modified (**open** for extension but **closed** for modification).*



Example

```
public class ProductFilter
{
    public IEnumerable<Product> ByColor(IList<Product> products, ProductColor productColor)
    {
        foreach (var product in products)
        {
            if (product.Color == productColor)
                yield return product;
        }
    }
}
```

User: “We need to also be able to filter by size.” Developer: “Just size alone or color and size? “

User: “Umm probably both.”

```
public class ProductFilter
{
    public IEnumerable<Product> ByColor(IList<Product> products, ProductColor productColor)
    {
        foreach (var product in products)
        {
            if (product.Color == productColor)
                yield return product;
        }
    }

    public IEnumerable<Product> ByColorAndSize(IList<Product> products,
                                                ProductColor productColor,
                                                ProductSize productSize)
    {
        foreach (var product in products)
        {
            if ((product.Color == productColor) &&
                (product.Size == productSize))
                yield return product;
        }
    }

    public IEnumerable<Product> BySize(IList<Product> products,
                                         ProductSize productSize)
    {
        foreach (var product in products)
        {
            if ((product.Size == productSize))
                yield return product;
        }
    }
}
```

CLOSED?

- Every time a user asks for new criteria to filter a product, do we have to modify the ProductFilter class?

OPEN?

- Every time a user asks for new criteria to filter a product, can we extend the behavior of the ProductFilter class to support the new criteria, without opening up the class file again and modifying it?

Solution?

- Commit to an interface, not an implementation => Strategy pattern

```
public abstract class ProductFilterSpecification
{
    public IEnumerable<Product> Filter(IList<Product> products)
    {
        return ApplyFilter(products);
    }

    protected abstract IEnumerable<Product> ApplyFilter(IList<Product> products);
}

public IEnumerable<Product> By(IList<Product> products, ProductFilterSpecification filterSpecification)
{
    return filterSpecification.Filter(products);
}
```

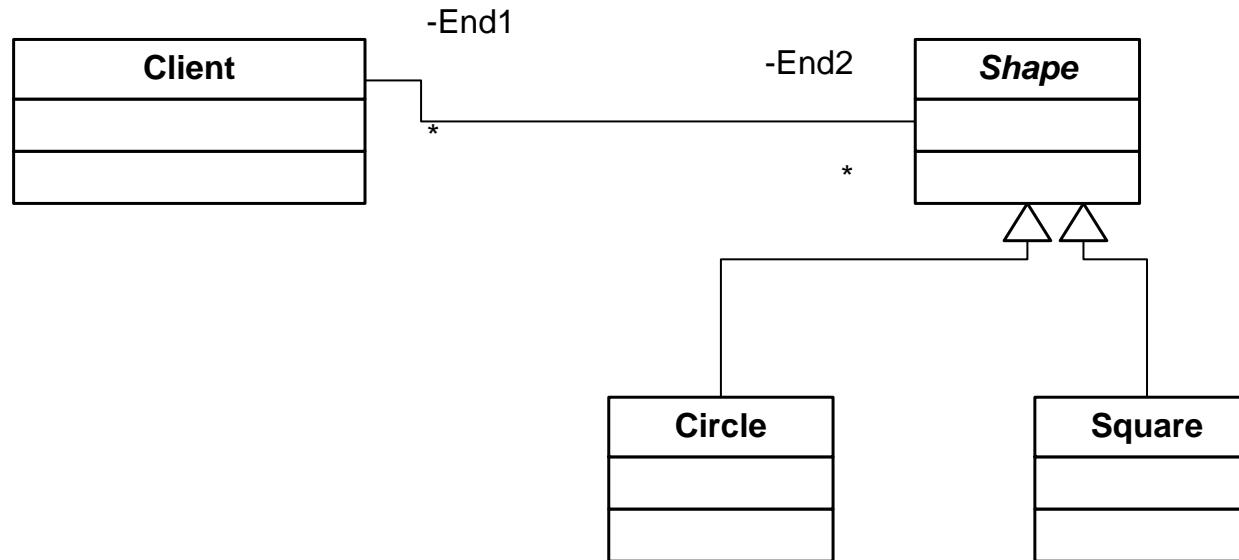
```
public class ColorFilterSpecification : ProductFilterSpecification
{
    private readonly ProductColor productColor;

    public ColorFilterSpecification(ProductColor productColor)
    {
        this.productColor = productColor;
    }

    protected override IEnumerable<Product> ApplyFilter(IList<Product> products)
    {
        foreach (var product in products)
        {
            if (product.Color == productColor)
                yield return product;
        }
    }
}
```

Techniques

- Dynamic polymorphism



- Static polymorphism
 - Templates, generics

Strategic closure

- Closure not *complete* but *strategic*
- Use abstraction to gain explicit closure
 - provide class methods which can be dynamically invoked to determine *general* policy decisions
 - design using abstract ancestor classes
- Use "Data-Driven" approach to achieve closure
 - place volatile policy decisions in a separate location
 - e.g. a configuration file or a separate object
 - minimizes future change locations

OCP Heuristics (I)

- Make all member variables private.
- Changes to public data are always at risk to “open” the module
 - They may have a rippling effect requiring changes at many unexpected locations;
 - Errors can be difficult to completely find and fix.

OCP Heuristics (II)

- RTTI is dangerous.
- RTTI is ugly and dangerous
 - If a module tries to dynamically cast a base class pointer to several derived classes, any time you extend the inheritance hierarchy, you need to change the module
 - recognize them by type **switch** or **if-else-if** structures
- Not all these situations violate OCP all the time
 - when used only as a "filter"

RTTI Example

```
//RTTI breaks OCP
public abstract class Shape
{ }

public class Square extends Shape
{
    private Point topLeftCorner;
    private double side;
    public DrawSquare() {//code here}
}

public class Circle extends Shape
{
    private Point center;
    private double radius;
    public DrawCircle() {//code here}
}
```

```
public class Client
{
    public DrawShapes(Collection
shapes)
{
    Iterator i = shapes.iterator();
    while(i.hasNext())
    {
        try
        {
            Circle c = (Circle)i.next();
            c.DrawCircle();
        }
        catch(...)
        try
        {
            Square s = (Square)i.next();
            s.DrawSquare();
        }
        catch(...)

    }
}
}
```

RTTI continued

```
//RTTI does not break OCP
public abstract class Shape
{
    public abstract Draw();
}

public class Square extends Shape
{
    private Point topLeftCorner;
    private double side;
    public void Draw() { //code here}
}
```

```
public class Circle extends Shape
{
    private Point center;
    private double radius;
    public void Draw() { //code here}
}

public class Client
{
    public DrawSquares(Collection
        shapes)  {
        Iterator i = shapes.iterator();
        while(i.hasNext()) {
            try
            {
                Square c = (Square)i.next();
                c.Draw();
            }
            catch(...)
            }
        }
}
```

Liskov Substitution Principle (LSP)

Functions that use references to base classes must be able to use objects of derived classes without knowing it.



LSP Violation

```
class Rectangle
```

```
{
```

```
    private:
```

```
        double width;
```

```
        double height;
```

```
    public:
```

```
        void setWidth(double w)...
```

```
        void setHeight(double h)...
```

```
}
```

```
class Square inherits Rectangle ?
```

IS-A relationship refers to BEHAVIOR

- Override `setWidth()` and `setHeight()`

- Duplicate code

- Problem - Static binding (C++)

```
void g(Rectangle& r)
```

```
{
```

```
    r.setWidth(4);
```

```
    r.setHeight(5);
```

```
}
```

Problem continued

- Dynamic binding

```
class Rectangle
{
    private:
        double width;
        double height;
    public:
        virtual void setWidth(double w)...
        virtual void setHeight(double h)...
}
void g(Rectangle& r)
{
    r.setWidth(4);
    r.setHeight(5);
    assert(r.getWidth()*r.getHeight()==20);
}
```

Design by Contract [Meyer]

- Basic notation: (P , Q : assertions, i.e. properties of the state of the computation. A : instructions).

$$\{P\} A \{Q\}$$

- Total correctness: Any execution of A started in a state satisfying P will terminate in a state satisfying Q .

Design by contract

1. Preconditions of the derived class method are no stronger than the base class method.
2. Postconditions of the derived class method are no weaker than the base class method.

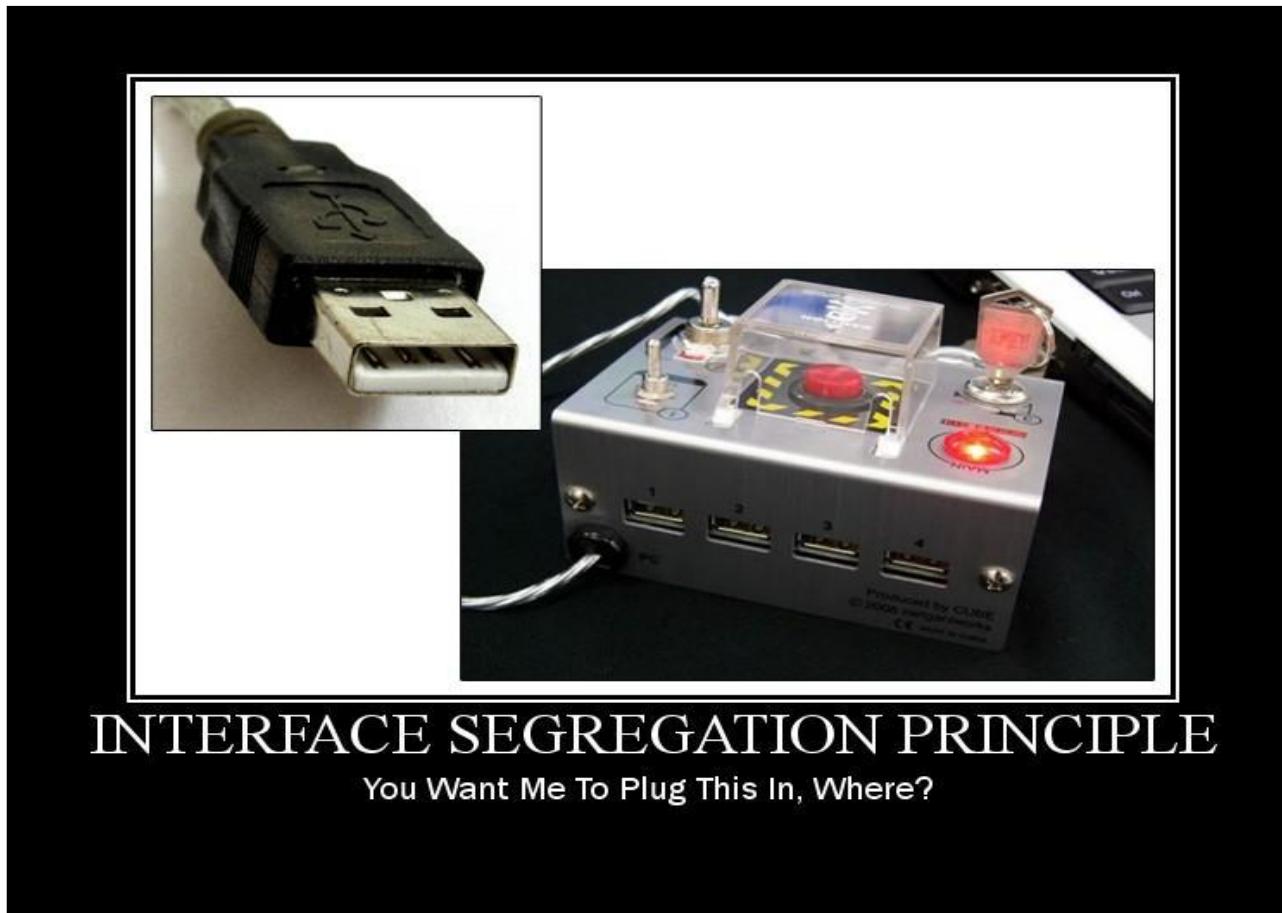
LSP Heuristics

It is illegal for a derived class, to override a base-class method with a NOP method

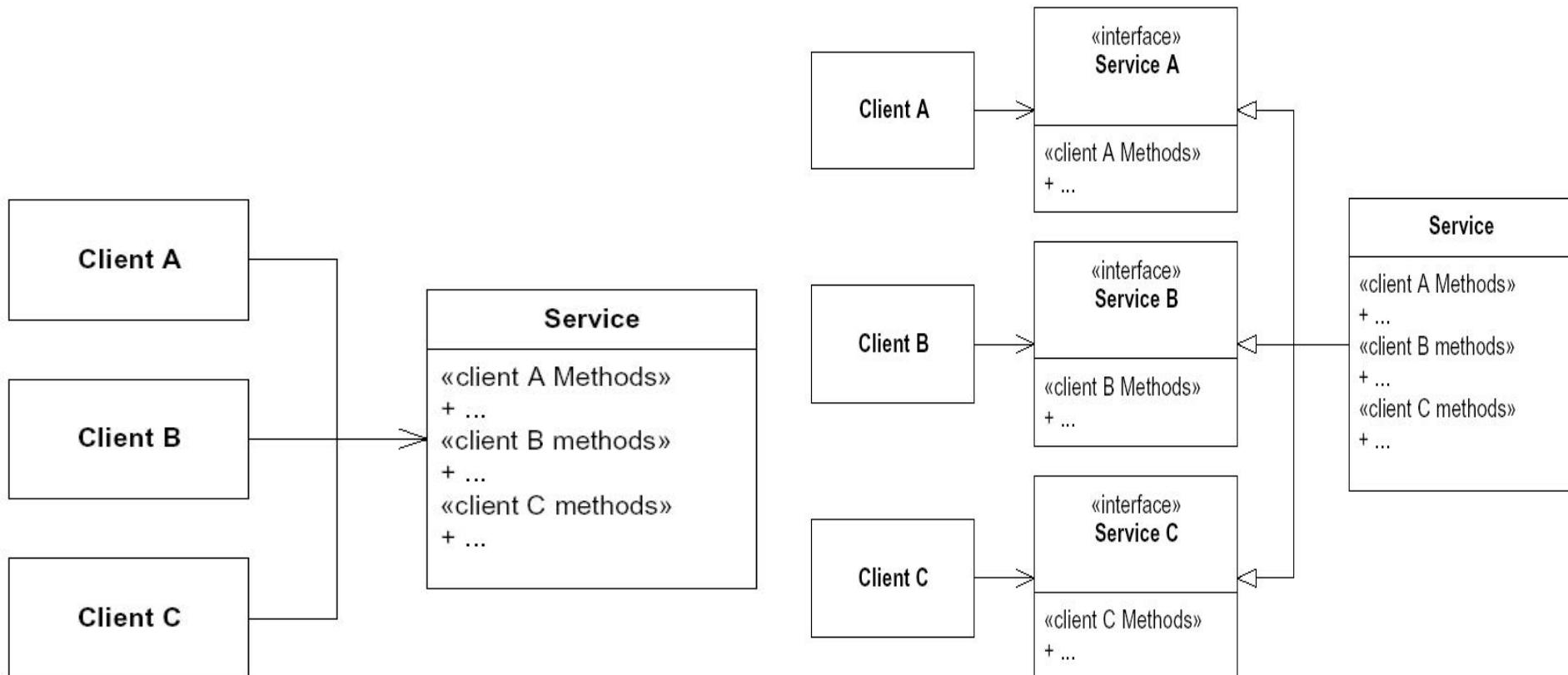
- NOP = a method that does nothing
- Solution 1: Inverse Inheritance Relation
 - if the initial base-class has only additional behavior
- Solution 2: Extract Common Base-Class
 - if both initial and derived classes have different behaviors

Interface Segregation Principle (ISP)

Clients should not be forced to depend upon interfaces that they don't use.

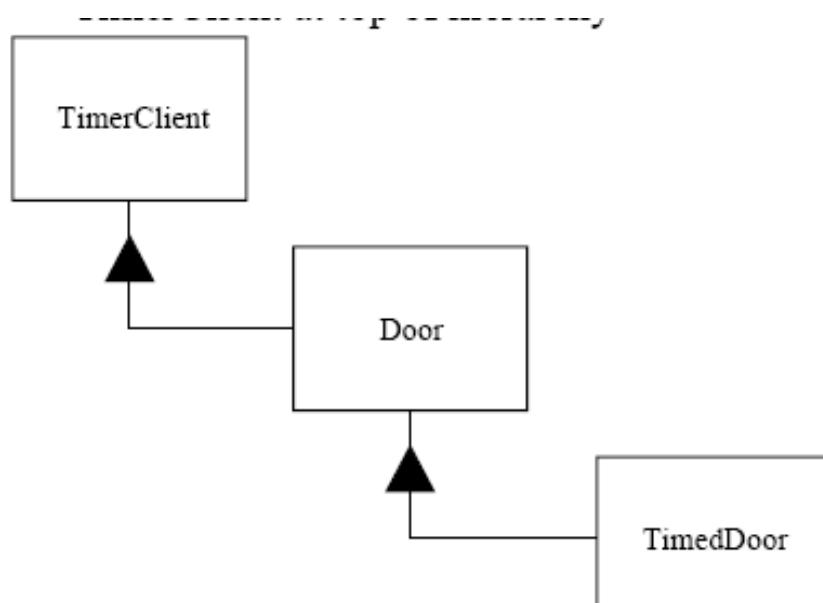


ISP

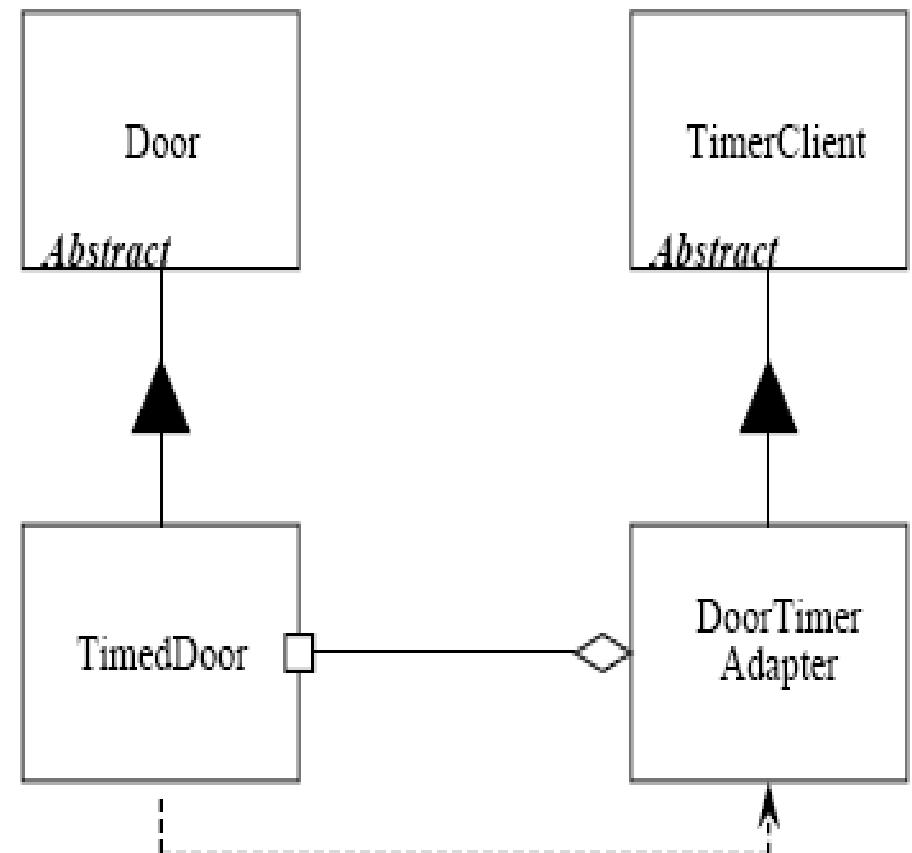


ISP Example

Non ISP conformant
design

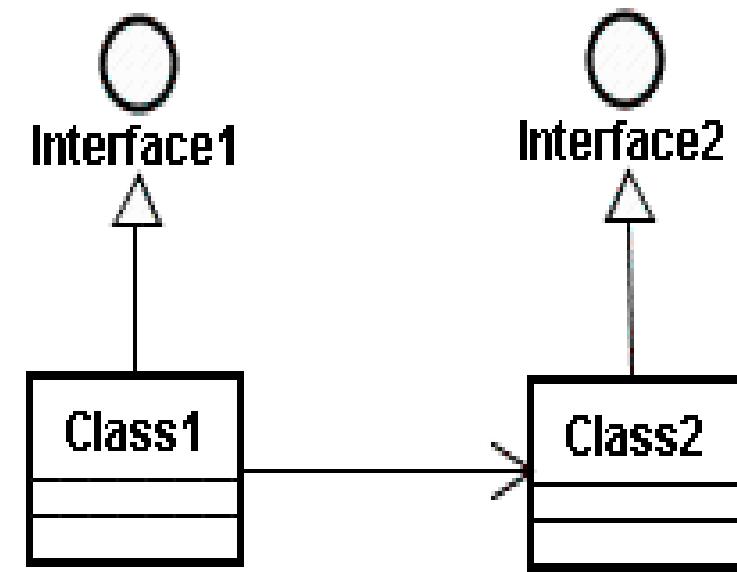
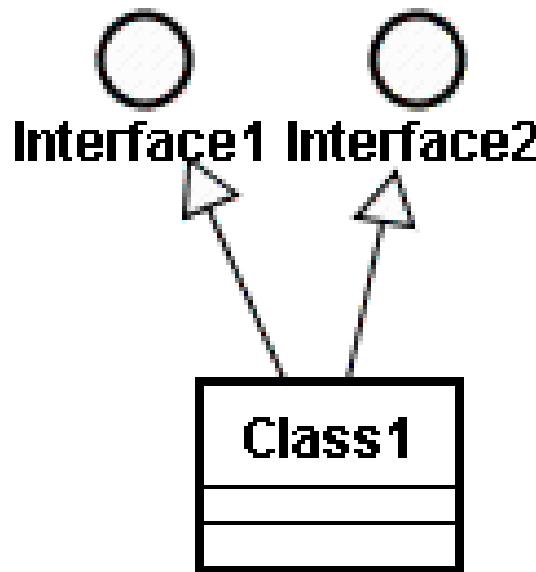


ISP conformant design



ISP Example

- Separation thru Multiple Inheritance vs. separation thru delegation



Dependency Inversion Principle (DIP)

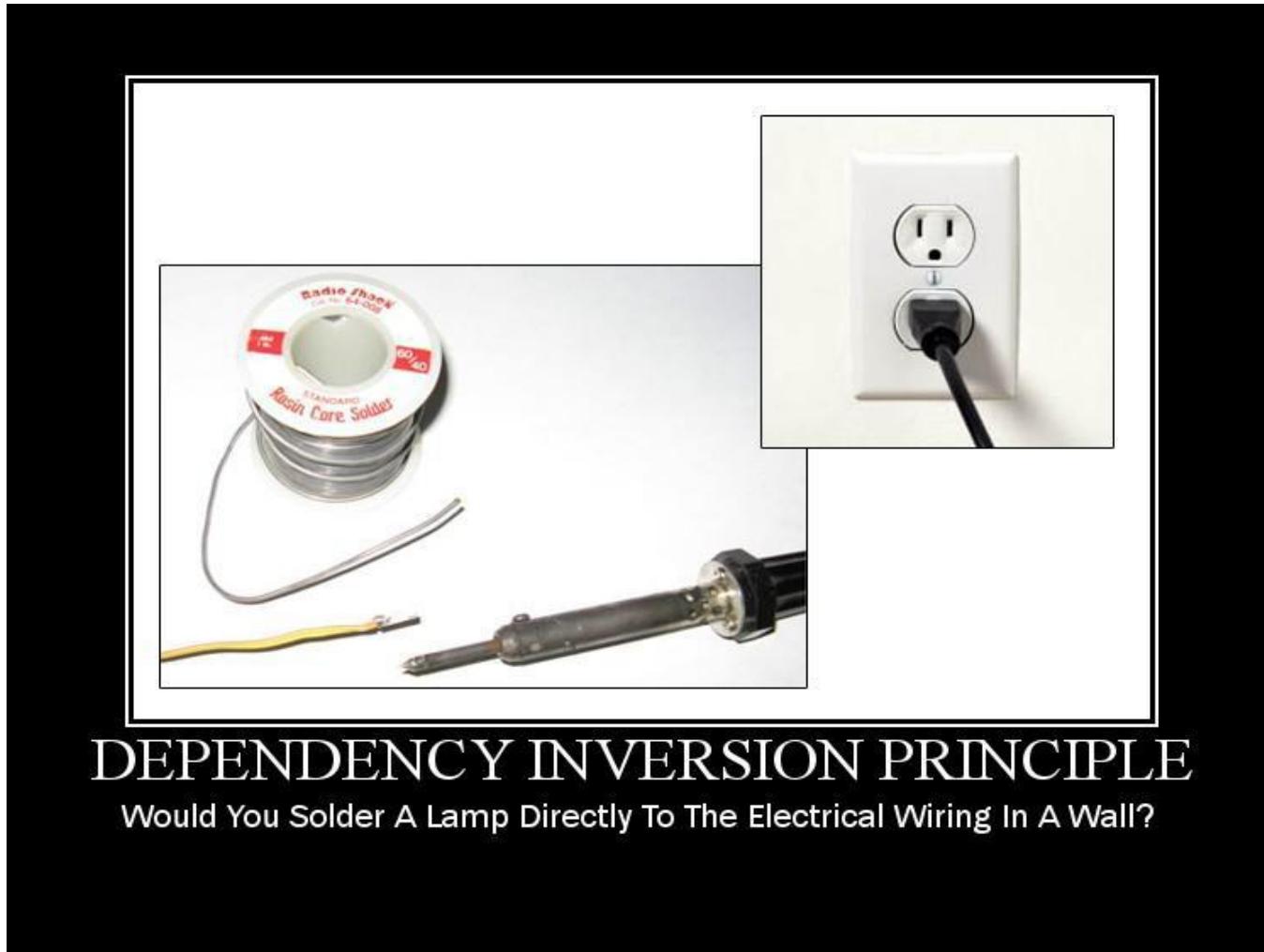
I.High-level modules should *not* depend on low-level modules.

Both should depend on abstractions.

II.Abstractions should not depend on details. Details should depend on abstractions.

R. Martin

DIP

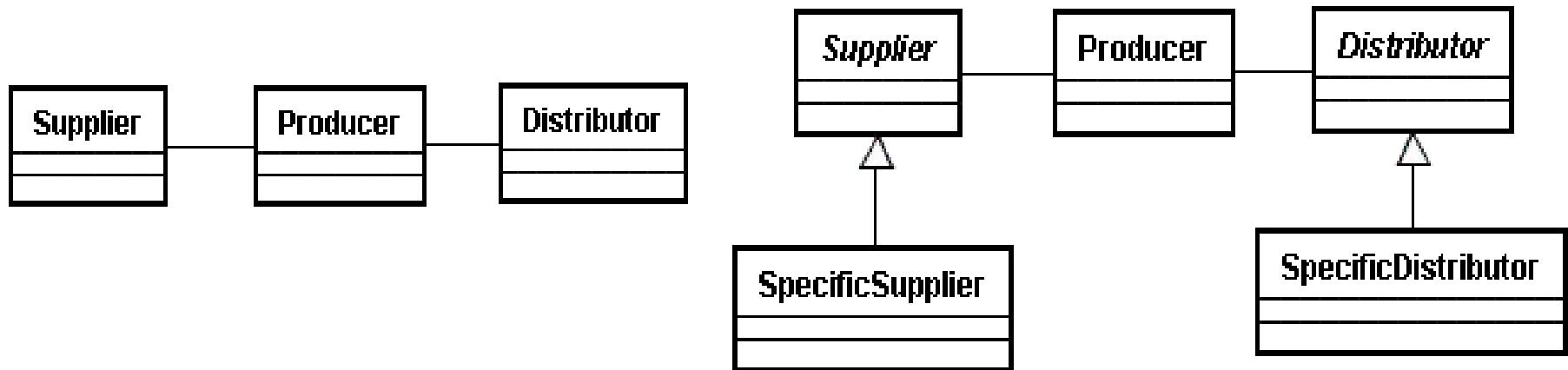


DEPENDENCY INVERSION PRINCIPLE

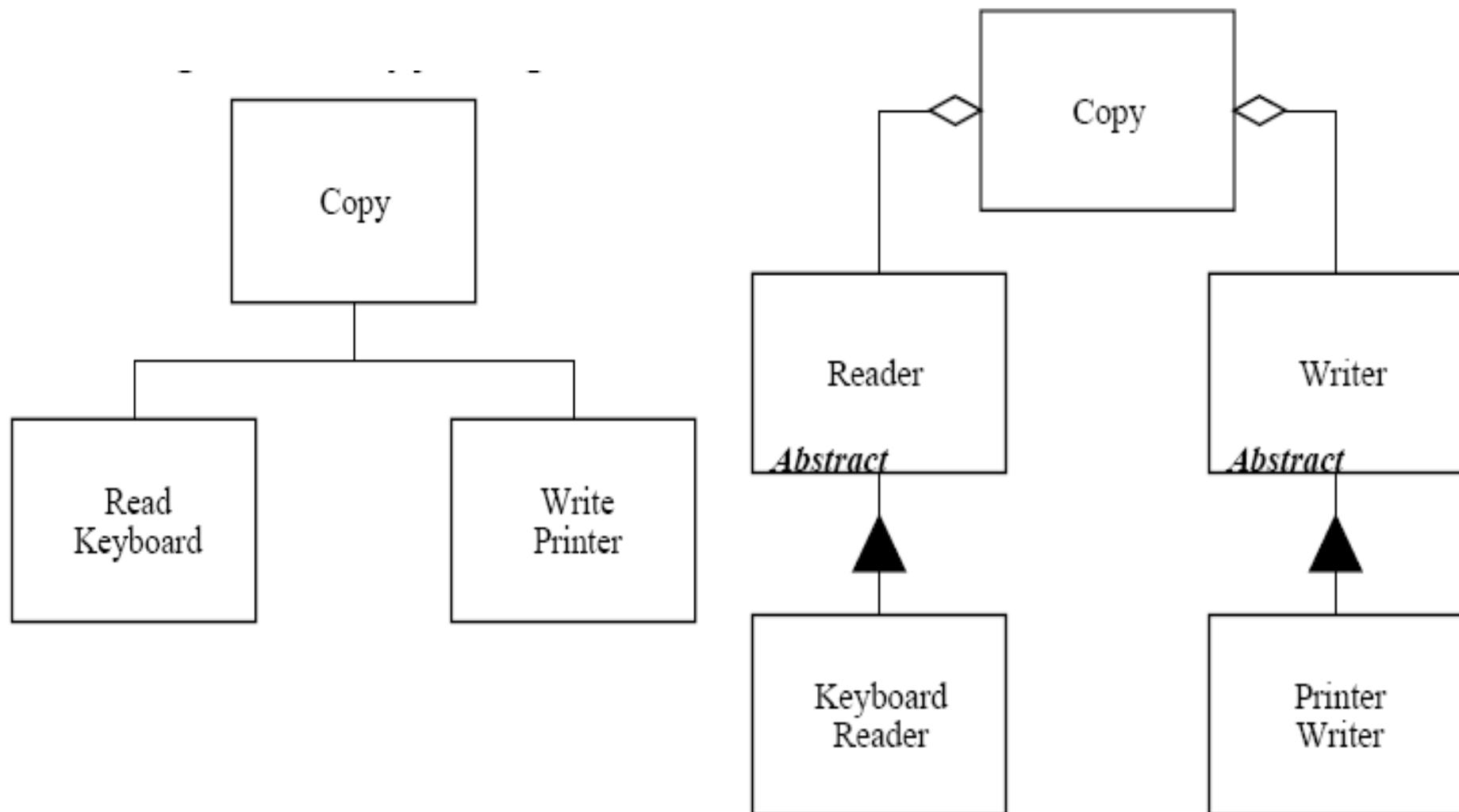
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

DIP

- A base class in an inheritance hierarchy should not know any of its subclasses
- Modules with detailed implementations are not depended upon, but depend themselves upon abstractions
- OCP states the goal; DIP states the mechanism;
- LSP is the insurance for DIP



DIP Example



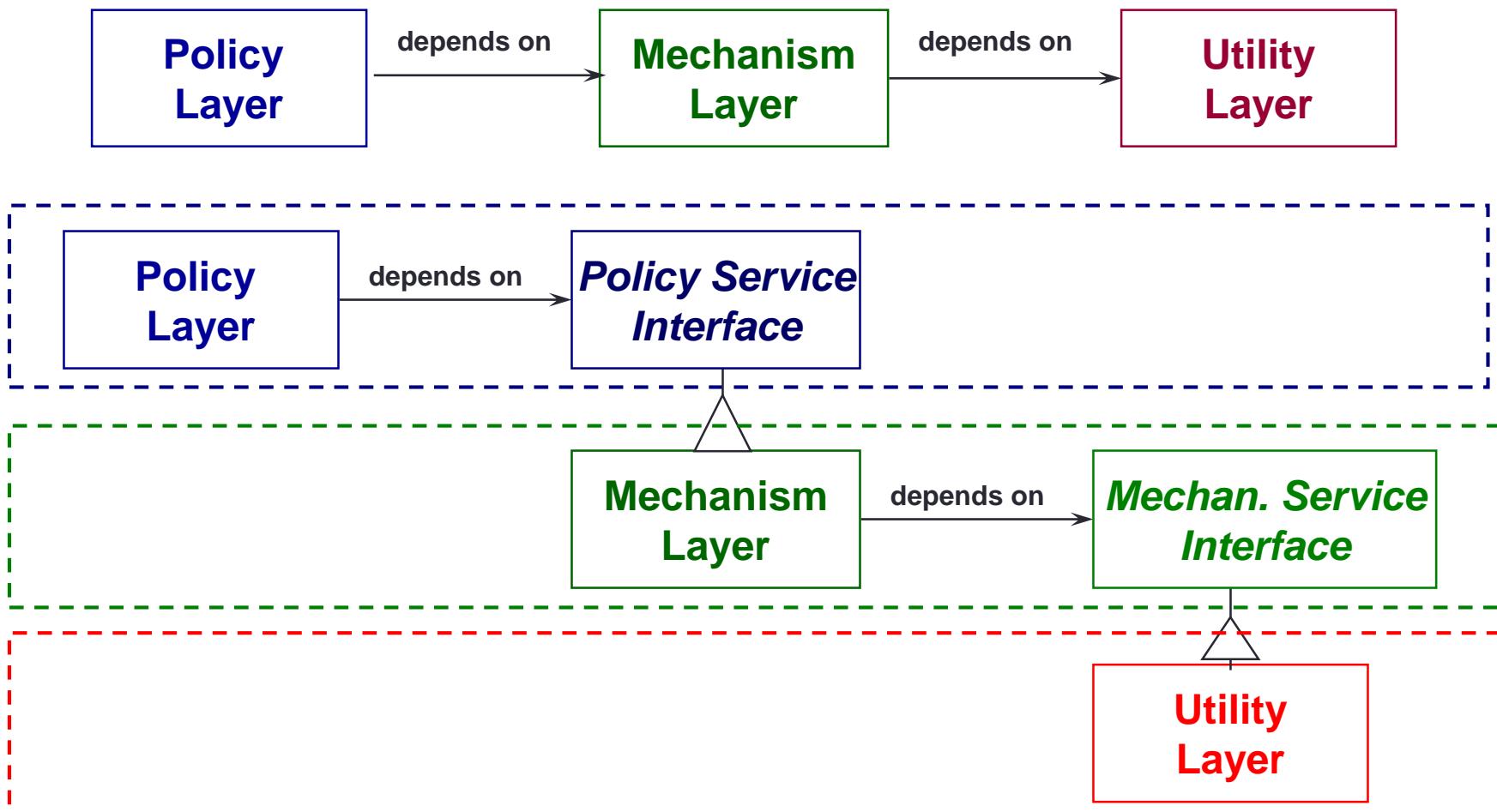
DIP Heuristics (I)

- Design to an interface, not an implementation!
- **Abstract classes/interfaces:**
 - tend to change less frequently
 - abstractions are ‘hinge points’ where it is easier to extend/modify
 - shouldn’t have to modify classes/interfaces that represent the abstraction (OCP)
- **Exceptions**
 - Some classes are very unlikely to change;
 - therefore little benefit to inserting abstraction layer
 - Example: String class
 - In cases like this can use concrete class directly

DIP Heuristics (II)

- Avoid Transitive Dependencies
- Avoid structures in which higher-level layers depend on lower-level abstractions:
 - In example below, Policy layer is ultimately dependant on Utility layer.
- Use inheritance and abstract ancestor classes to effectively eliminate transitive dependencies:

DIP



GRASP

- General Responsibility Assignment Software Patterns
- OO system = objects sending messages to other objects to complete operations.
- Issues:
 - Responsibilities assigned to objects
 - Interaction ways between objects
- To arrive at a good object-oriented design we use principles applied during the creation of interaction diagrams and/or responsibility assignment: e.g. GRASP patterns.

Responsibilities

- Knowing responsibilities:
 - knowing about private encapsulated data;
 - knowing about related objects;
 - knowing about things it can derive or calculate;
- Doing responsibilities:
 - doing something itself;
 - initiating action in other objects;
 - controlling and coordinating activities in other objects;
- A responsibility is not the same as a method, but methods are implemented to fulfil responsibilities.

GRASP: General Principles in Assigning Responsibilities

- From Craig Larman's 9 patterns:
 - **Expert**
 - **Creator**
 - **Controller**
 - Low Coupling
 - High Cohesion
 - Polymorphism
 - Pure Fabrication
 - Indirection
 - **Don't Talk to Strangers (Law of Demeter)**

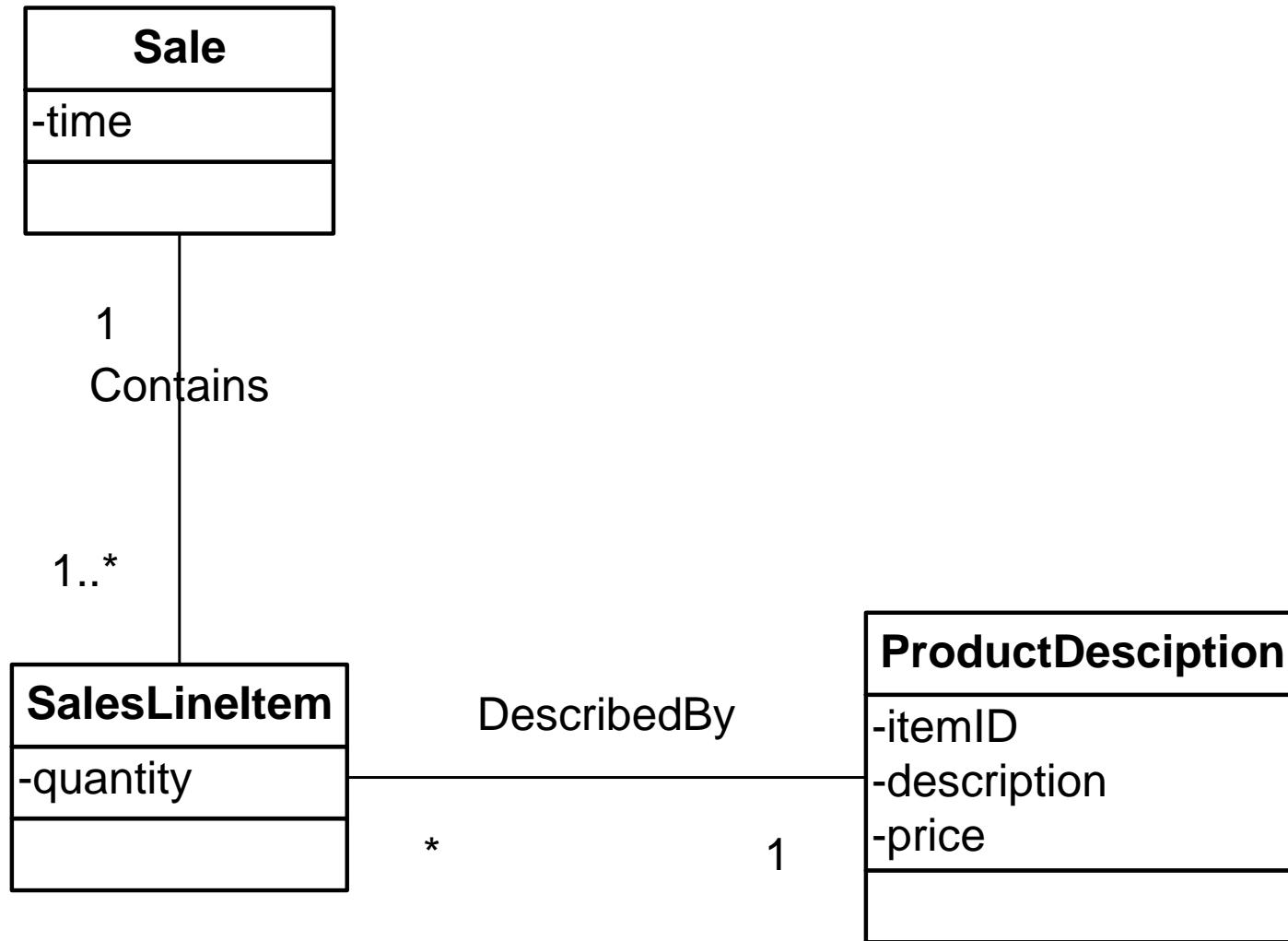
Case Study – POS System

A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

Models

- Domain Model
 - Sale
 - SaleLineItem
 - Payment
 - Product
 - Register
- Design Model
 - ?

Design Model



Expert

Problem: What is the most basic principle by which responsibilities are assigned in object-oriented design?

Solution: Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility

Example: In the POS application, some class needs to know the grand total of a sale.

- A general advice is to start assigning responsibilities by clearly stating the responsibility.
- Here: "Who should be responsible for knowing the grand total of the sale?"
- By Expert, we should look for that class of objects that has the information needed to determine the total.
- By the Expert pattern, Sale is the information expert for the responsibility under consideration.

Expert – Sale class



Sale

-time

+getTotal()

SalesLineItem

-quantity

+getSubtotal()

ProductDescription

-itemID

-description

-price

+getPrice()

Conclusion

- Discussion
 - Expert is used more than any other pattern in the assignment of responsibilities.
 - Whenever information is spread across different objects, they will need to interact via messages in order to share the work.
- Benefits
 - Information encapsulation is maintained since objects use their own information to fulfill tasks => supports low coupling
 - Behavior is distributed across the classes that have the required information => more cohesive "lightweight" class definitions

Creator

Problem: Who should be responsible for creating a new instance of some class?

Solution: Assign class B the responsibility to create an instance of class A if one of the following is true:

B contains A objects

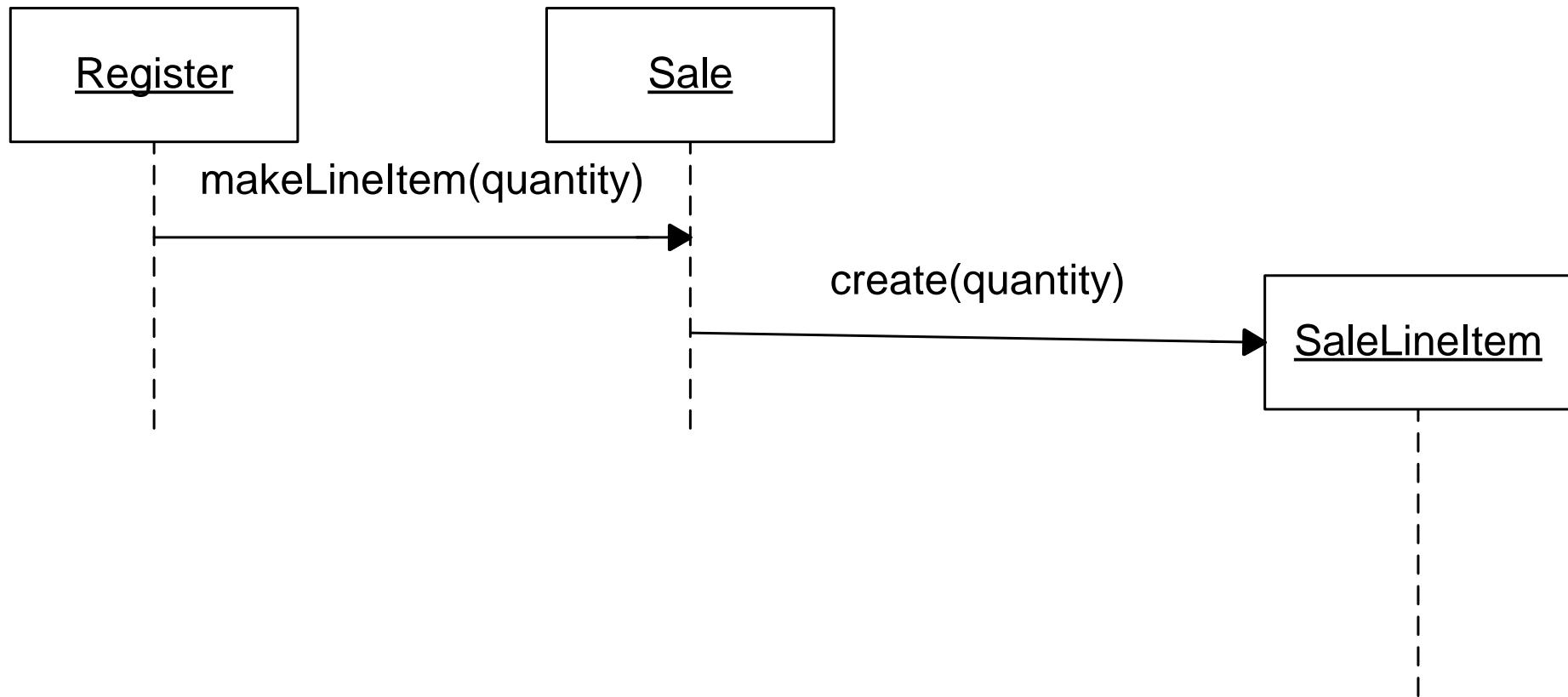
B closely uses A objects

B has the initialising data that will be passed to A when it is created.

Example: In the POS application who should be responsible for creating a SalesLineItem?

- By Creator we should look for a class that contains/closely uses/has initialization data of SalesLineItem instances.
- By Creator, Sale is a good candidate to have the responsibility of creating SalesLineItem.

Creator



Conclusion

- Discussion:
 - Creator guides assigning responsibilities related to the creation of objects.
 - Sometimes a creator is found by looking for the class that has the initialising data that will be used during creation.

For example, who should be responsible to create a Payment instance ?

- Benefits:
 - Low Coupling is supported
- Contraindications:
 - Complex creation procedures
 - Solution ?

Controller

Problem: Who should be responsible for handling a system event?

- A system event is a high level event generated by an external actor.
- They are associated with system operations.
- A Controller is a non-user interface object responsible for handling a system event.
- A controller defines the method for the system event

Solution: Assign the responsibility for handling a system event message to a class representing one of the following choices:

- Represents the overall "system" (facade controller);
- Represents the overall business or organisation (facade controller);
- Represents something in the real-world that is active (for example, the role of a person) that might be involved in the task (role controller);
- Represents an artificial handler of all system events of a use-case, usually named (use-case controller)

Controller

Example: In the point of sale application the current system operations have been identified as:

- endSale()
- enterItem()
- makePayment()

- Who should be the controller for the system events such as enterItem and endSale?
- By the Controller pattern, here are the choices:
 - represents the overall "system": POST
 - represents the overall business or organisation: Store
 - somebody/something actively involved: Cashier
 - an artificial handler: BuyItemsHandler

Conclusions

Discussion:

- Normally, a controller object delegates to other objects the work that needs to be done to fulfil the responsibilities that it has been assigned.
- A facade controller usually handles all the system events of a system.
- Facade controllers are only suitable when there are only a few system events.
- Use case handler controller handles the system events of one use case.
- There are as many use case handler controllers as there are use cases.
- A use case controller is an alternative to consider when placing the responsibilities in any of the other choices of controller, leads to designs with low cohesion or high coupling (usually because of bloated controllers).
- External interfacing objects (for example window objects) and the presentation layer should not have responsibility for fulfilling system events: they delegate to a controller.

Conclusions

Benefits:

- Increased potential for reusable components:
 - it ensures that business or domain processes are handled by the layer of domain objects rather than by the interface layer.
 - the application is not bound to a particular interface.
- Reason about the state of the use case:
 - As all the system events belonging to a particular use case are assigned to a single class, it is easier to control the sequence of events that may be imposed by a use case (e.g. MakePayment cannot occur until EndSale has occurred).

Law of Demeter

- Weak Form

Inside of a method M of a class C, data can be accessed in and messages can be sent to only the following objects:

- **this** and **super**
 - **data members** of class C
 - **parameters** of the method M
 - **object created** within M
 - by calling directly a constructor
 - by calling a method that creates the object
 - **global variables**
- Strong Form:

In addition to the Weak Form, you are not allowed to access directly inherited members

LoD Example

- Any methods of an object should call only methods belonging to:

```
class Demeter {  
private:  
    A *a;  
public:  
    // ...  
    void example(B& b);  
void Demeter::example(B& b) {  
    C *c;  
    c = func();  
    b.invert();  
    a = new A();  
    a->setActive();  
    c->print();  
}
```

passed parameters

created objects

directly held component objects

LoD Counter Example

```
class Course
{
    Instructor boring = new Instructor();
    int pay = 5;
    public Instructor getInstructor() { return boring; }
    public Instructor getNewInstructor() {return new Instructor(); }
    public int getPay() {return pay; }
}

class C {
    Course test = new Course();
    public void badM() { test.getInstructor().fire(); }
    public void goodM() { test.getNewInstructor().hire(); }
    public int goodOrBadM?() { return test.getpay() + 10; }
}
```

LoD good example

```
class Course {  
    Instructor boring = new Instructor();  
    int pay = 5;  
    public Instructor fireInstructor(){ boring.fired(); }  
    public Instructor getNewInstructor() { return new Instructor();}  
    public int getPay() { return pay ; }  
}  
  
class C {  
    Course test = new Course();  
    public void reformedBadM() {test.fireInstructor();}  
    public void goodM() {test.getNewInstructor().hired();}  
    public int goodOrBadM() { return test.getpay() + 10; }  
}
```

LoD for children

- You can play *with yourself.*
- You can play *with your own toys*
- You can play *with toys that were given to you.*
- You can play *with toys you've made yourself.*

LoD Benefits

- Coupling Control
 - reduces data coupling
- Information hiding
 - prevents from retrieving subparts of an object
- Information restriction
 - restricts the use of methods that provide information
- Few Interfaces
 - restricts the classes that can be used in a method
- Explicit Interfaces
 - states explicitly which classes can be used in a method

Acceptable LoD Violations

- If optimization requires violation
 - Speed or memory restrictions
- If module accessed is a fully stabilized “Black Box”
 - No changes to interface can reasonably be expected due to extensive testing, usage, etc.
- Otherwise, do not violate this law!
 - Long-term costs will be very prohibitive

Wrap-up

- Principles for good class design related to
 - Inheritance (LSP)
 - Modifiability (O-CP)
 - Assigning Responsibilities (SRP, GRASP)
 - Publishing (ISP)
 - Dependency (DIP)

SOFTWARE DESIGN

Package design principles, Software design metrics

Content

- Package Design
 - Cohesion Principles
 - Coupling Principles
- Software design metrics

References

- David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Ed.[Patterson]
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley [Taylor]
- Gillibrand, David, Liu, Kecheng. Quality Metric for Object-Oriented Design. *Journal of Object-Oriented Programming*. Jan 1998.
- Dynamic Metrics for Object Oriented Designs, Sherif Yacoub, Tom Robinson, and Hany H. Ammar, West Virginia University, 2010
- S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- ETHZ course materials
- Univ. of Aarhus Course Materials
- Univ. of Utrecht Course Materials

High-level Design

- Dealing with *large-scale systems*
 - team of developers, rather than an individual
- Classes are a valuable but not sufficient mechanism
 - too *fine-grained* for organizing a large scale design
 - need mechanism that impose a higher level of order

Packages

- a logical grouping of declarations that can be imported in other programs
- containers for a group of classes (UML)
 - reason at a higher-level of abstraction

Issues of High-Level Design

Goal

- *partition* the classes in an application according to some *criteria* and then *allocate* those partitions to packages

Issues

- What are the best partitioning criteria?
- What principles govern the design of packages?
 - *creation* and *dependencies* between packages
- Design packages first? Or classes first?
 - i.e. *top-down* vs. *bottom-up approach*

Approach

- Define principles that govern package design
 - the creation and interrelationship and use of packages

Principles of OO High-Level Design

- Cohesion Principles
 - Reuse/Release Equivalency Principle (REP)
 - Common Reuse Principle (CRP)
 - Common Closure Principle (CCP)
- Coupling Principles
 - Acyclic Dependencies Principle (ADP)
 - Stable Dependencies Principle (SDP)
 - Stable Abstractions Principle (SAP)

What is really Reusability ?

- Does copy-paste mean reusability?
 - Disadvantage: **You own that copy!**
 - you must change it, fix bugs.
 - eventually the code diverges
 - Maintenance is a nightmare
- Martin's Definition:
 - *I reuse code if, and only if, I never need to look at the source-code*
 - treat reused code like a *product* ⇒ don't have to maintain it
- Clients (re-users) may decide on an appropriate time to use a newer version of a component release

Reuse/Release Equivalency Principle (REP)

- *The granule of reuse is the granule of release. Only components that are released through a tracking system can be efficiently reused.* [R. Martin]
- *Either all the classes in a package are reusable or none of it is!* [R. Martin]

What does this mean?

- Reused code = product
 - Released, named and maintained by the producer.
- Programmer = client
 - Doesn't have to maintain reused code
 - Doesn't have to name reused code
 - May choose to use an older release

The Common Reuse Principle

All classes in a package [library] should be reused together. If you reuse one of the classes in the package, you reuse them all. [R.Martin]

If I depend on a package, I want to depend on every class in that package! [R.Martin]

What does this mean?

- Criteria for grouping classes in a package:
 - Classes that tend to be reused together.
- Packages have physical representations (shared libraries, DLLs, assembly)
 - Changing just one class in the package -> re-release the package -> revalidate the application that uses the package.

Common Closure Principle (CCP)

The classes in a package should be closed against the same kinds of changes.

A change that affects a package affects all the classes in that package

[R. Martin]

What does this mean?

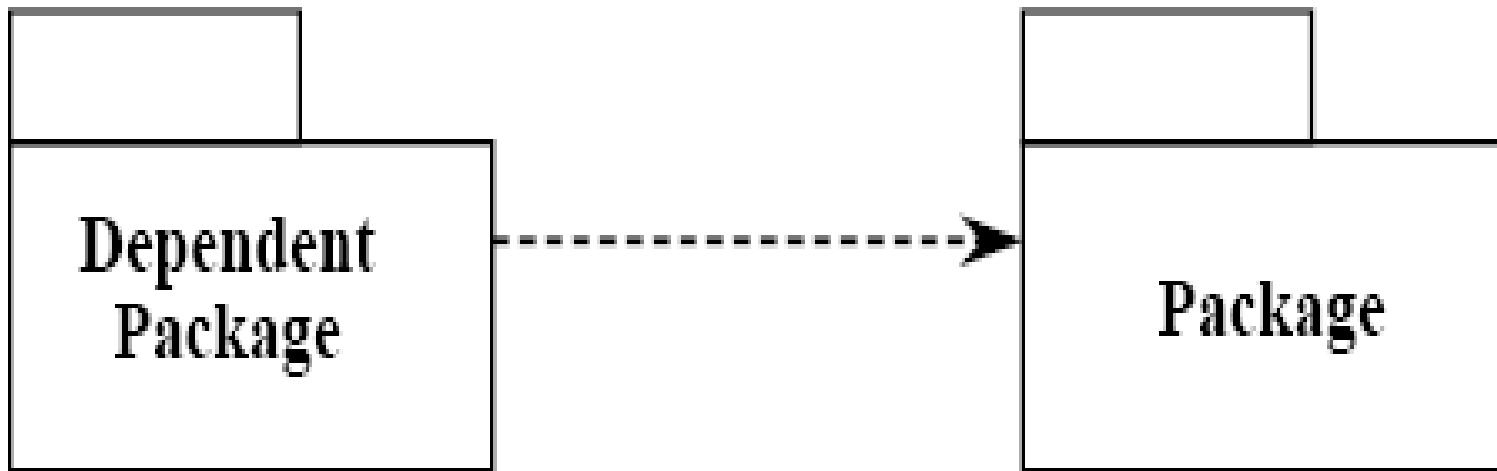
- Another criteria of grouping classes:
 - Maintainability!
 - Classes that tend to change together for the same reasons
 - Classes highly dependent
- Related to OCP
 - How?

Reuse vs. Maintenance

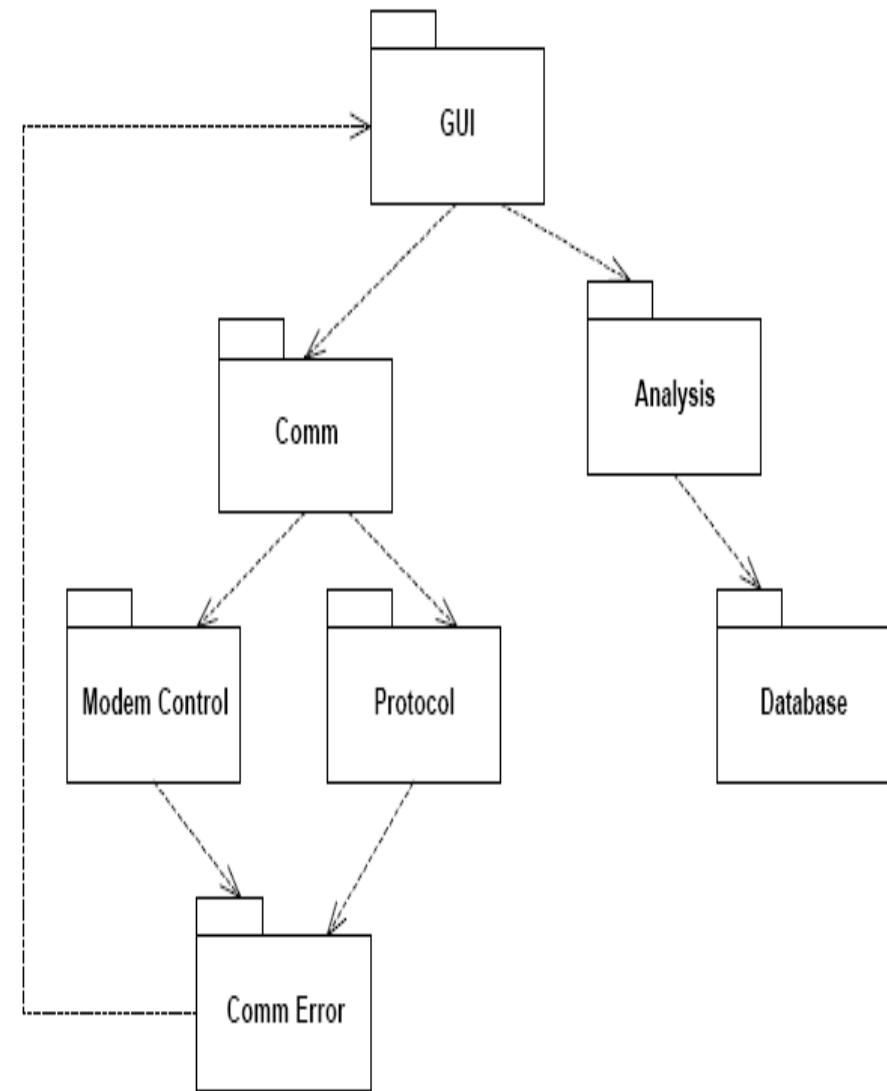
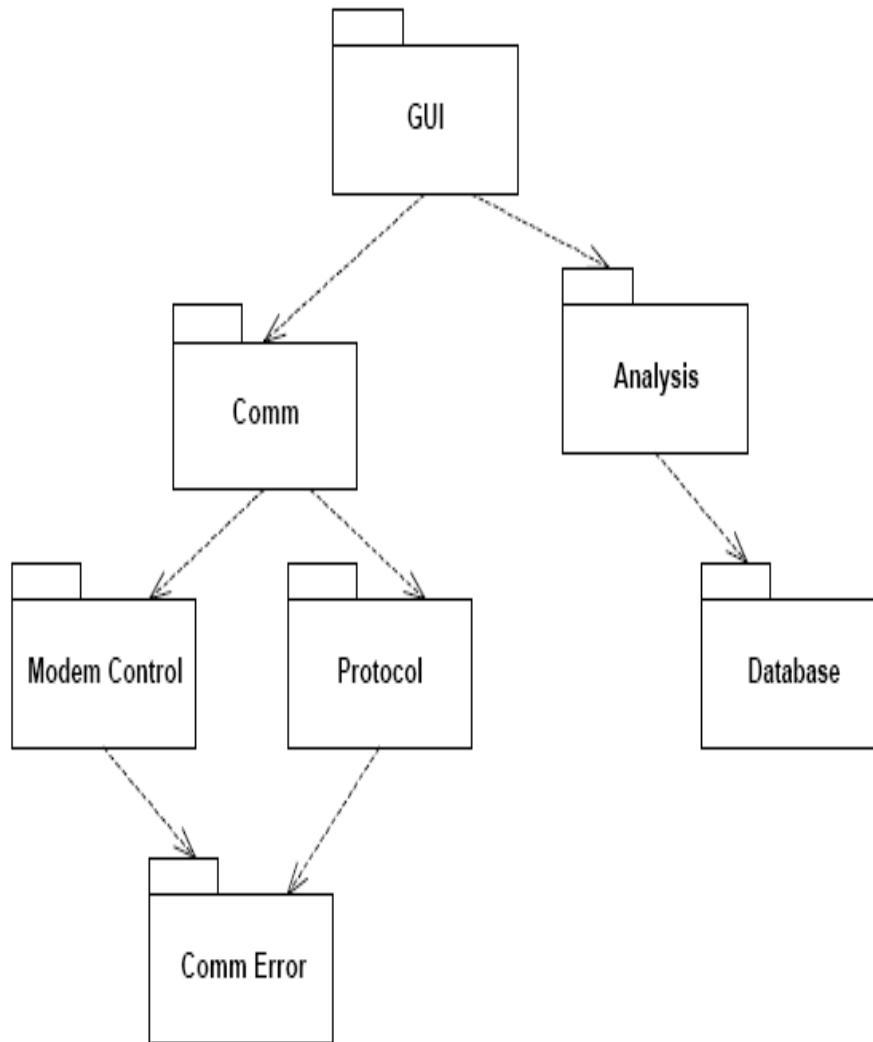
- REP and CRP makes life easier for **reuser**
 - packages very small
- CCP makes life easier for **maintainer**
 - large packages
- **Packages are not fixed in stone**
 - early in project focus on CCP
 - later when architecture stabilizes: focus on REP and CRP

Acyclic Dependencies Principles (ADP)

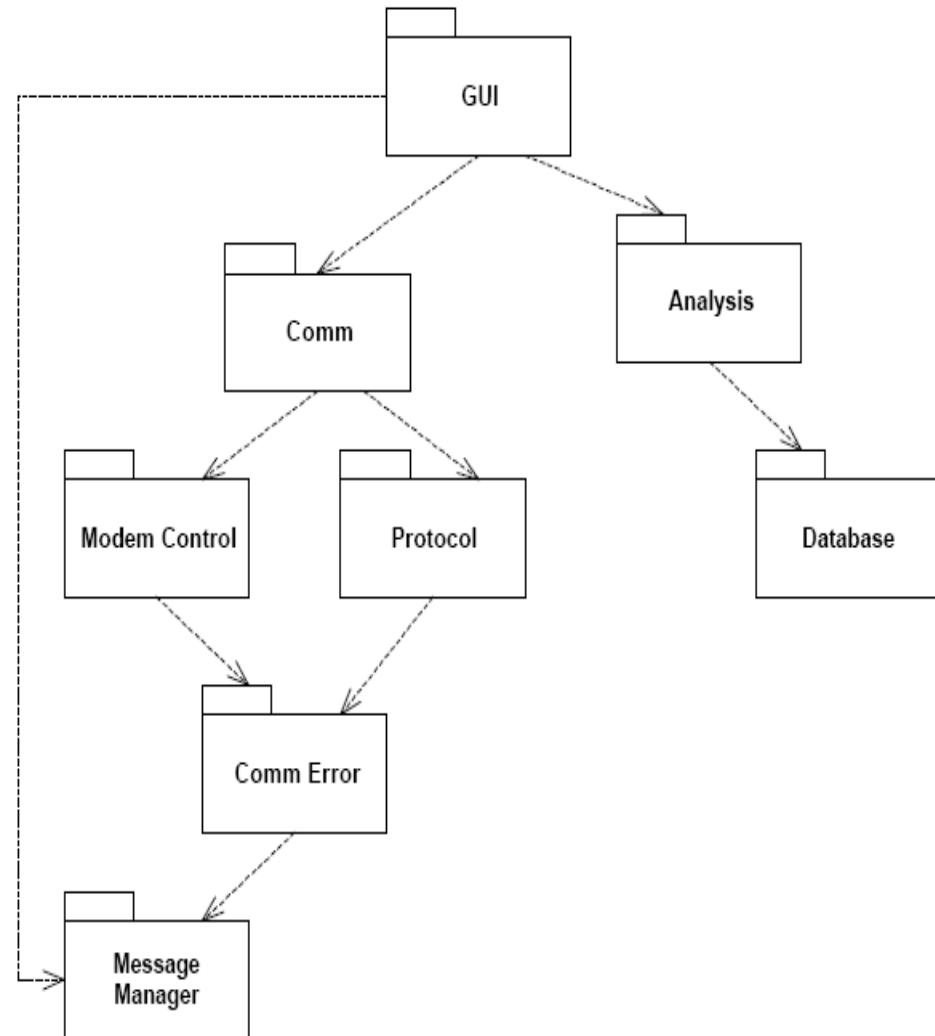
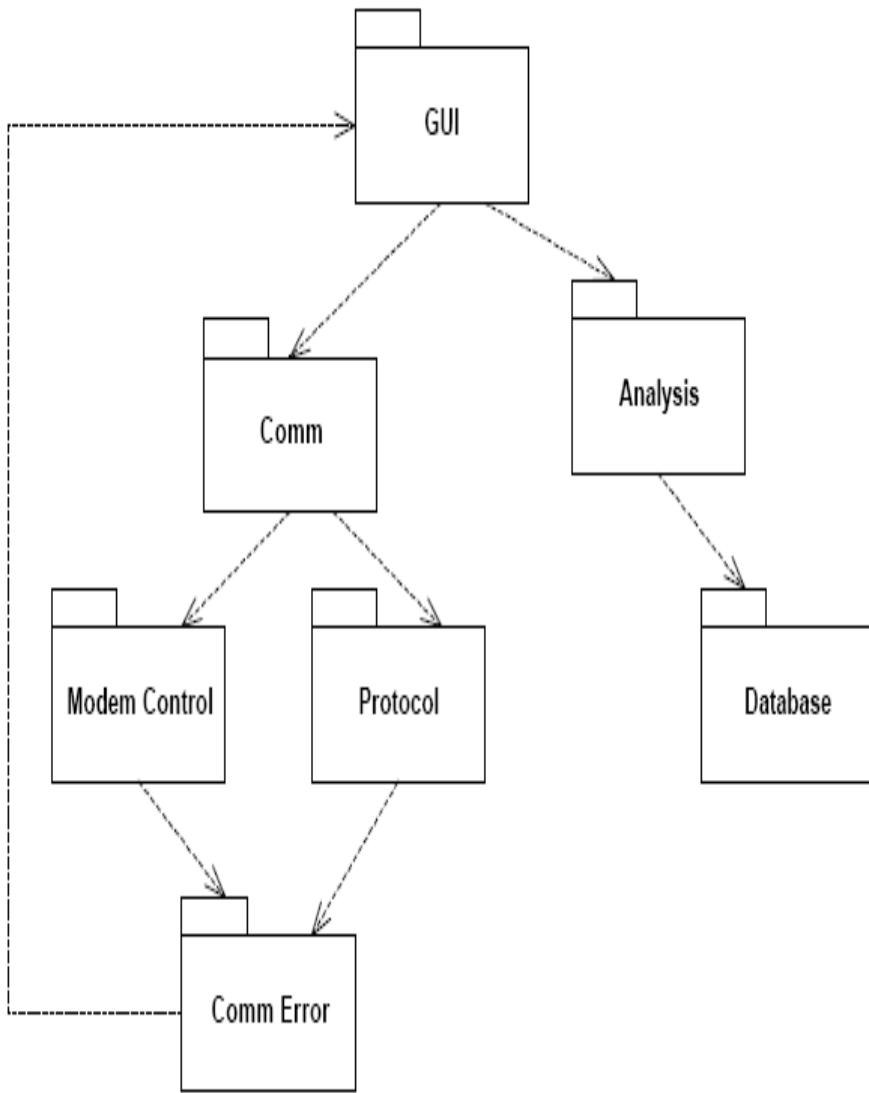
The dependency structure for released component must be a Directed Acyclic Graph (DAG). There can be no cycles. [R. Martin]



Dependency Graphs

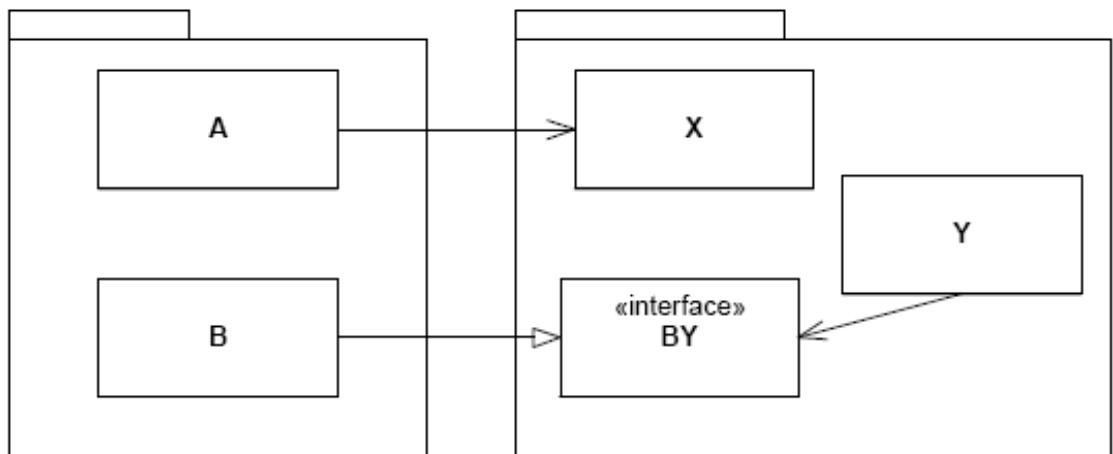
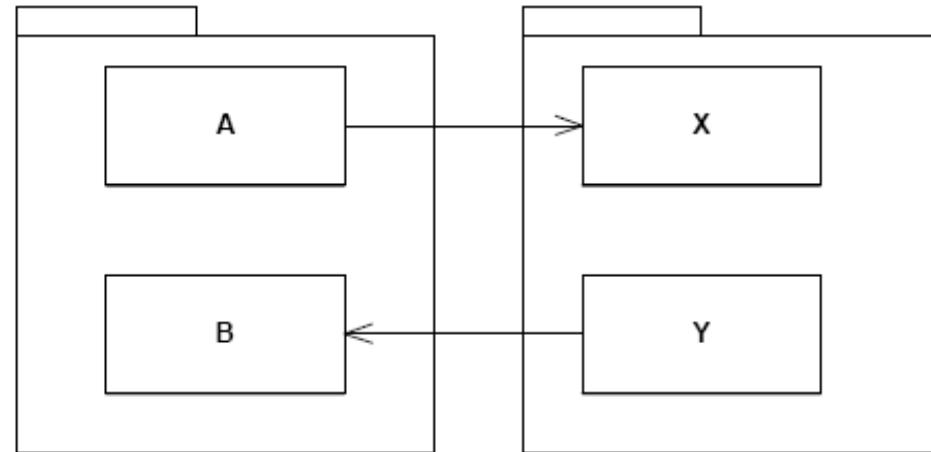


Breaking the Cycle



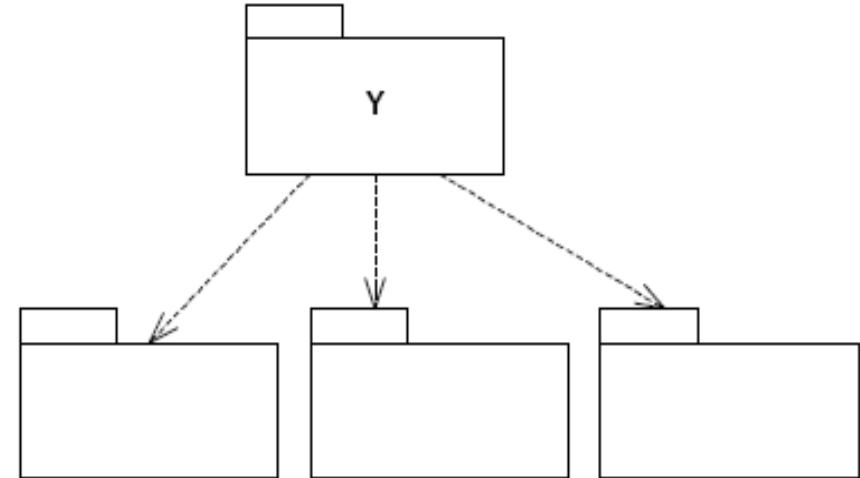
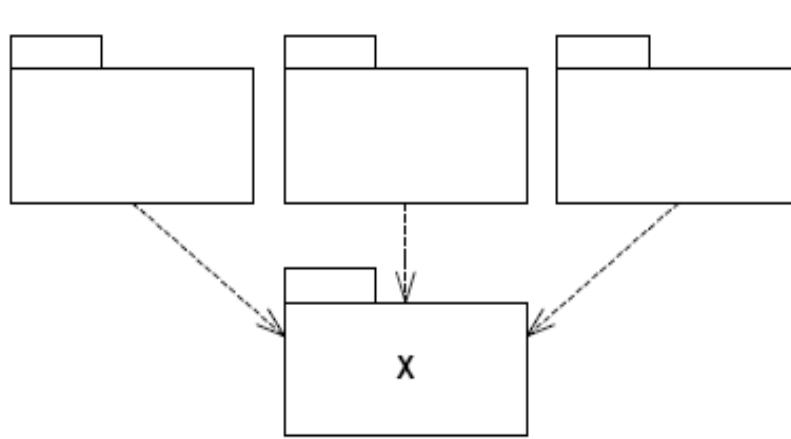
Breaking the Cycle

- DIP + ISP



Stability

- Stability is related to the amount of work in order to make a change.



Stability defined by:

- Responsibility
- Independence

Stability metrics

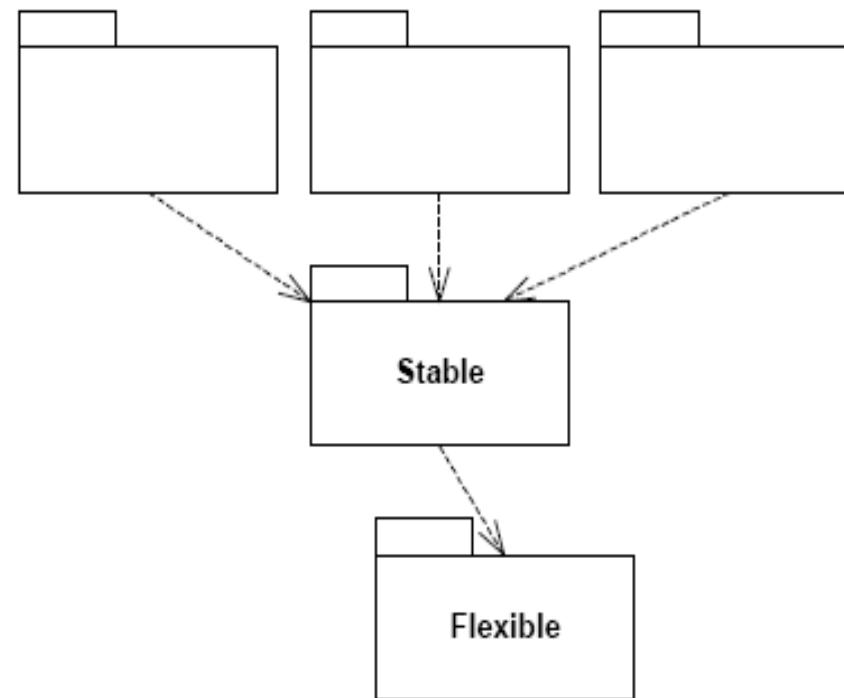
- C_a – Afferent coupling (incoming dependencies)
 - How responsible am I?
- C_e – Efferent coupling (outgoing dependencies)
 - How dependant am I?
- Instability I
 $I = C_e/(C_a+C_e)$

Example for X:

$C_a = 3, C_e = 0 \Rightarrow I = 0$ (very stable)

Stable Dependency Principle (SDP)

- Depend in the direction of stability.
- What does this mean?
 - Depend upon packages whose I is lower than yours.
- Counter-example



Where to Put High-Level Design?

- High-level architecture and design decisions don't change often
 - shouldn't be volatile \Rightarrow place them in stable packages
 - design becomes hard to change \Rightarrow *inflexible design*
- How can a totally stable package ($I = 0$) be flexible enough to withstand change?
 - improve it without modifying it...
- Answer: ***The Open-Closed Principle***
 - classes that can be extended without modifying them \Rightarrow **Abstract Classes**

Stable Abstractions Principle (SAP)

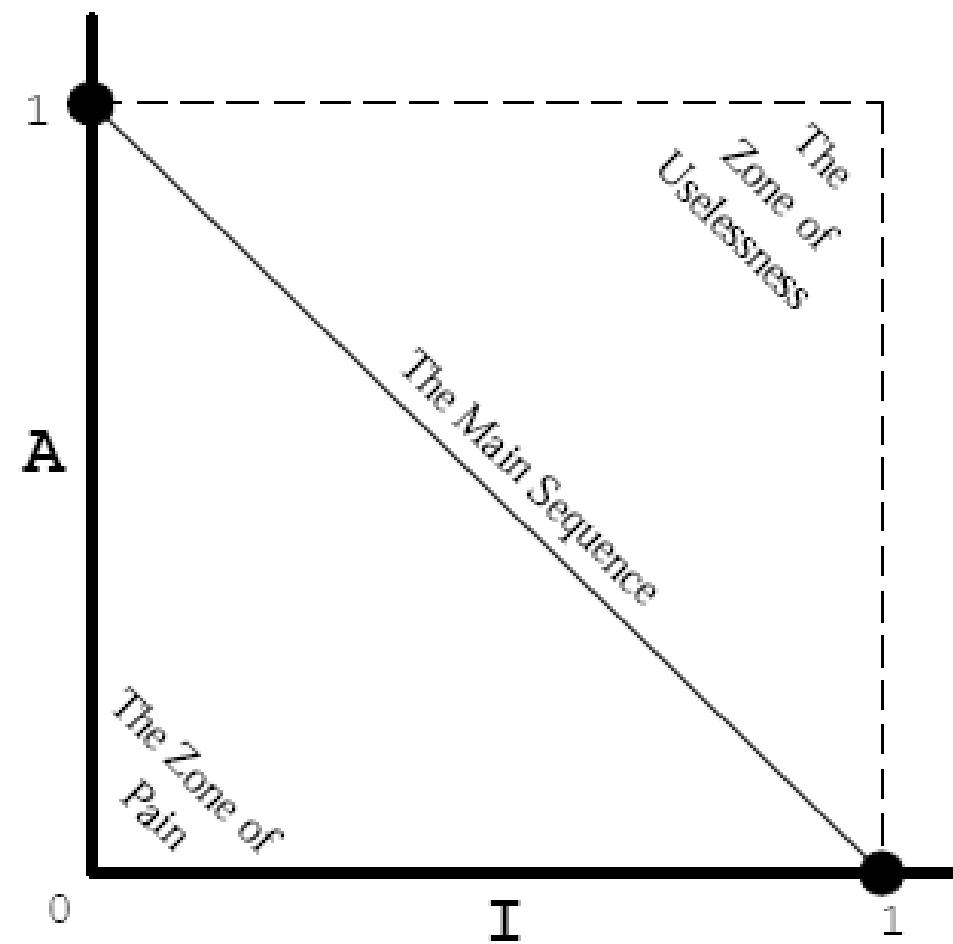
- Stable packages should be abstract packages.
- What does this mean?
 - Stable packages should be on the bottom of the design (depended upon)
 - Flexible packages should be on top of the design (dependent)
 - OCP => Stable packages should be highly abstract

Abstractness metrics

- N_c = number of classes in the package
- N_a = number of abstract classes in the package
- $A = N_a/N_c$ (Abstractness)
- Example:
 - $N_a = 0 \Rightarrow A = 0$
- What about hybrid classes?

The Main Sequence

- I should increase as A decreases



The Main Sequence

- Zone of Pain
 - highly stable and concrete \Rightarrow rigid
 - famous examples:
 - database-schemas (volatile and highly depended-upon)
 - concrete utility libraries (instable but non-volatile)
- Zone of Uselessness
 - instable and abstract \Rightarrow useless
 - no one depends on those classes
- Main Sequence
 - maximizes the distance between the zones we want to avoid
 - depicts the balance between abstractness and stability.

Object-Oriented Design Metrics

- Appropriate measures
 - Classes
 - Modularity
 - Encapsulation
 - Inheritance
 - Abstraction
- Indicate reusability, maintainability, testability

Chidamber and Kemerer (CK) metrics

- WMC—Weighted Methods per Class
 - an indicator of the amount of effort required to implement and test a class
- DIT—Depth of Inheritance Tree
 - Large values imply more design complexity, but also more reuse.
- NOC—Number of Children
 - large values imply that the abstraction dilution, the increased need for testing and more reuse
- CBO—Coupling Between Object Classes
 - Large values imply more complexity, reduced maintainability and reduced reusability
- RFC—Response for Class
 - RFC increases => complexity and need for testing increases
- LCOM—Lack of Cohesion on Methods
 - LCOM increases => complexity and design difficulty increases

Weighted Methods Per Class (WMC)

- Sum of the complexity of each method contained in the class.
- Method complexity: (e.g. cyclomatic complexity)
 - When method complexity assumed to be 1, WMC = number of methods in class

Coupling between objects (CBO)

- Number of other classes to which a class is coupled, i.e., suppliers of a class.
- Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.
- The uses relationship can go either way: both uses and used-by relationships are taken into account, but only once.

Lack of cohesion (LCOM)

- LCOM measures the dissimilarity of methods in a class by instance variable or attributes.
- *Functional cohesion* - the design unit (module) performs a single well-defined function or achieves a single goal.
- *Sequential cohesion* - the design unit performs more than one function, but these functions occur in an order prescribed by the specification, i.e., they are strongly related.
- *Communication cohesion* - a design unit performs multiple functions, but all are targeted on the same data.

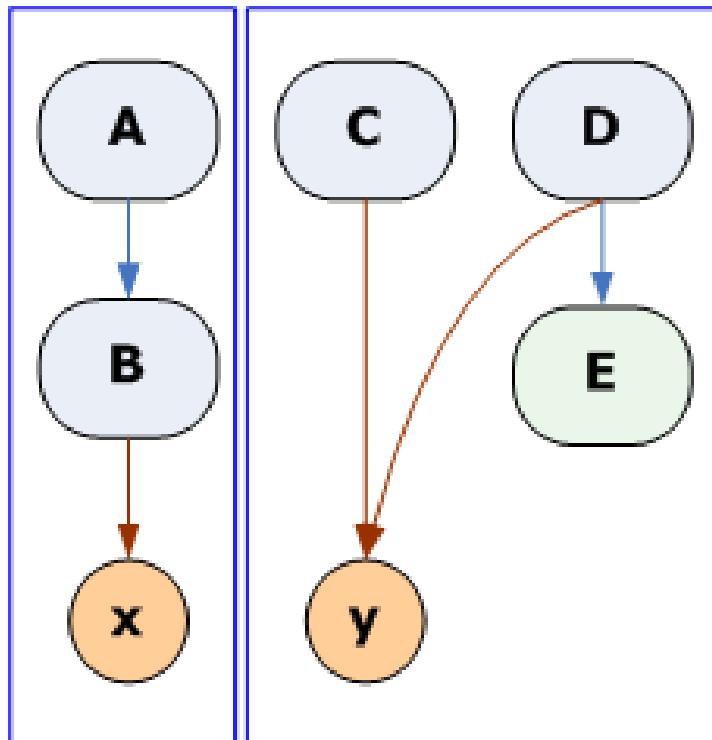
LCOM cont'd

- *Procedural cohesion* - a design unit performs multiple functions that are procedurally related. The code in each module represents a single piece of functionality defining a control sequence of activities.
- *Temporal cohesion* - a design unit performs more than one function, and they are related only by the fact that they must occur within the same time span (ex. a design that combines all data initialization into one unit and performs all initialization at the same time even though it may be defined and utilized in other design units).
- *Logical cohesion* - a design unit that performs a series of similar functions (ex. the Java class `java.lang.Math`)

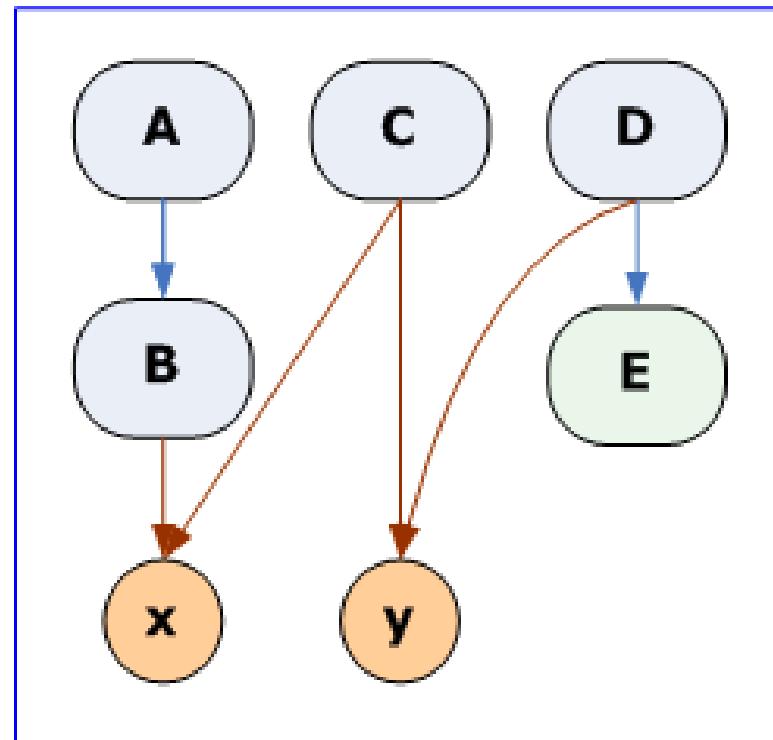
LCOM4

- LCOM4 measures the number of "*connected components*" in a class.
- A connected component is a set of related methods (and class-level variables).
- Methods a and b are related if:
 - they both access the same class-level variable, or
 - a calls b, or b calls a.
- There should be only one such a component in each class. If there are 2 or more components, the class should be split into so many smaller classes.

LCOM4



$\text{LCOM4} = 2$



$\text{LCOM4} = 1$

Response for a Class (RFC)

- The RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class.
- Includes all methods accessible within the class hierarchy.

CK metrics discussion

- high values of the CK OOD metrics correlate with:
 - Lower productivity
 - High effort to reuse classes
 - High effort to design classes
 - Difficulties in implementing classes
 - Number of maintenance changes
 - Number of faulty classes
 - Faults
 - User reported problems

Engineering rules

Any class that meets at least two of the following needs to be considered for refactoring:

- Response for class > 100
- Coupling between objects > 5
- Response for class > 5 * the number of methods in the class
- Weighted methods per class > 100
- Number of methods > 40

Metrics for Object Oriented Design (MOOD)

- Method Inheritance Factor
- Attribute Inheritance Factor
- Coupling Factor
- Clustering Factor
- Polymorphism Factor
- Method Hiding Factor
- Attribute Hiding Factor
- Reuse Factor

Method Hiding Factor

- $M_d(C_i) = M_v(C_i) + M_h(C_i)$

Where

- $M_d(C_i)$ - number of methods defined in class C_i
- $M_v(C_i)$ - number of visible methods defined in class C_i
- $M_h(C_i)$ - number of hidden methods defined in class C_i

$$\text{MHF} = \frac{\sum_{i=0}^{TC} M_h(Ci)}{\sum_{i=0}^{TC} M_d(Ci)}$$

Attribute Hiding Factor

- $A_d(C_i) = A_v(C_i) + A_h(C_i)$

Where

- $A_d(C_i)$ - number of attributes defined in class C_i
- $A_v(C_i)$ - number of visible attributes defined in class C_i
- $A_h(C_i)$ - number of hidden attributes defined in class C_i

$$AHF = \frac{\sum_{i=0}^{TC} A_h(Ci)}{\sum_{i=0}^{TC} A_d(Ci)}$$

MHF/AHF heuristics

- measures how variables and methods are encapsulated in a class. Visibility is counted with respect to other classes.
- the number of visible methods is a measure of the class functionality. MHF will decrease if overall functionality will increase.
- a high MHF indicates very less functionality and that the design includes a high proportion of specialized methods that are not available for reuse

Inheritance

- Base class (has no ancestors)

$$is_base(C_b) = \begin{cases} 1 & \text{iff } \forall j \in [1, TC], \neg \exists C_j : C_b \rightarrow C_j \\ 0 & \text{otherwise} \end{cases}$$

- Leaf class (has no descendants)

$$is_leaf(C_l) = \begin{cases} 1 & \text{iff } \forall j \in [1, TC], \neg \exists C_j : C_j \rightarrow C_l \\ 0 & \text{otherwise} \end{cases}$$

Definitions

- **Children Count**, $CC(C_i)$ - number of children of C_i ,
- **Descendants Count**, $DC(C_i)$ - number of descendants of C_i
- **Parents Count**, $PC(C_i)$ - number of parents of C_i (notes: if $PC(C_i)=0$ then C_i is a base class; if $PC(C_i)>1$ we have multiple inheritance).
- **Ancestors Count**, $AC(C_i)$ - number of ancestors of C_i .
- $M_d(C_i)$ - number of Methods defined in C_i
- $M_n(C_i)$ - number of New Methods defined in C_i
- $M_i(C_i)$ - number of Inherited Methods defined in C_i (not overridden)
- $M_o(C_i)$ - number of Overridden Methods defined in C_i (inherited methods that are redefined)
- $M_a(C_i)$ - number of Available Methods defined in C_i

Derived relations

- Can be used for attributes too

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

$$\begin{aligned}M_a(C_i) &= M_d(C_i) + M_i(C_i) \\&= M_n(C_i) + \sum_{j=1}^{PC(Ci)} M_a(C_j)\end{aligned}$$

System Metrics

- Total number of Methods Defined

$$TM_d = TM_n + TM_o = \sum_{k=1}^{TC} M_d(C_k)$$

- Total number of New Methods defined

$$TM_n = \sum_{k=1}^{TC} M_n(C_k)$$

- Total number of Methods Overridden

$$TM_o = \sum_{k=1}^{TC} M_o(C_k)$$

- Total number of Methods Inherited

$$\begin{aligned} TM_i &= \sum_{k=1}^{TC} M_i(C_k) = \sum_{k=1}^{TC} [M_n(C_k) * DC(C_k) - M_o(C_k)] \\ &= \sum_{k=1}^{TC} [(M_d(C_k) - M_o(C_k)) * DC(C_k) - M_o(C_k)] \end{aligned}$$

System Metrics cont'd

- Total number of Methods Available

$$\begin{aligned}
 TM_a &= TM_d + TM_i = \sum_{k=1}^{TC} M_a(C_k) \\
 &= \sum_{k=1}^{TC} [(M_d(C_k) - M_o(C_k)) * [1 + DC(C_k)]] \\
 &= \sum_{k=1}^{TC} [M_n(C_k) * [1 + DC(C_k)]]
 \end{aligned}$$

- Total Length of Inheritance Chain

$$TLIC = \sum_{i=1}^{TC} PC(C_i) = \sum_{i=1}^{TC} CC(C_i)$$

- Total Number of Inheritance Paths

$$TNIP =$$

$$TLIC - TC - \sum_{i=1}^{TC} [is_base(C_i) + is_leaf(C_i)]$$

MOOD

- Method Inheritance Factor

$$MIF = \frac{TM_i}{TM_a}$$

- Attribute Inheritance Factor

$$AIF = \frac{TA_i}{TA_a}$$

- Coupling Factor

$$COF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is_client(C_i, C_j) \right]}{TC^2 - TC}$$

where

$$is_client(C_c, C_s) = \begin{cases} 1 & \text{iff } C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 & \text{otherwise} \end{cases}$$

MOOD

- **Clustering Factor** (as defined by client-supplier and inheritance relations)

$$\text{CLF} = \frac{\text{TCC}}{\text{TC}}$$

- **Polymorphism Factor**

$$PF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{DC(C_i)} M_o(C_j) \right]}{\sum_{i=1}^{TC} [M_d(C_i) * DC(C_i)]}$$

- **Reuse Factor**

$$RF = \frac{\sum_{i=1}^{TC} \text{in_library}(C_i)}{TC} + \\ + \frac{MIF * \sum_{i=1}^{TC} [1 - \text{in_library}(C_i)]}{TC}$$

MOOD heuristics

MOOD METRIC	LL	INT	UL
Method Inheritance Factor		x	
Attribute Inheritance Factor		x	
Coupling Factor			x
Clustering Factor	x		
Polymorphism Factor		x	
Method Hiding Factor	x		
Attribute Hiding Factor	x		
Reuse Factor	x		

Dynamic Metrics for Object Oriented Designs

- metrics for dynamic complexity and object coupling based on execution scenarios
- Definitions and Terminology
 - o_i : is an instance of a class (an object)
 - O : is the set of objects collaborating during the execution of a specific scenario
 - $|Z|$: The number of elements in set Z
 - $\{\}$: A representation of a set of elements

Definitions

- A **scenario** x from a set of scenarios X , is a sequence of interactions between objects stimulated by input data or event
- The **probability of a scenario** is the execution frequency of that scenario with respect to all other scenarios of the application and is denoted as " PS_x "
- A **scenario profile** for an application is the set of probabilities of execution of each specific scenario " PS_x "
- $M_x(o_i, o_j)$ is set of **messages** sent from object o_i to object o_j during the execution of scenario x . A message is defined as a request that one object makes to another to perform a service.

Export Object Coupling $EOC_x(o_i, o_j)$

- $EOC_x(o_i, o_j)$, the export coupling for object o_i with respect to object o_j is the count of the number of messages sent from o_i to o_j during the execution of a specific scenario x .

$$EOC_x(o_i, o_j) = |\{M_x(o_i, o_j) \mid o_i, o_j \in O \wedge o_i \neq o_j\}|$$

- High EOC => low maintainability, understandability, reusability. Source for error propagation

Object Request for Service OQFS

- total number of messages sent by the object o_i to all other objects in the design.

$$OQFS_i(o_i) = \left| \bigcup_{j=1}^K M_i(o_i, o_j) \mid o_i, o_j \in O \wedge o_i \neq o_j \right|$$

Import Object Coupling; $IOC_x(o_i, o_j)$

- $IOC_x(o_i, o_j)$, the import coupling for object o_i with respect to object o_j , is the count of the number of messages received by object o_i and was sent by object o_j during the execution of the scenario x .

$$IOC_x(o_i, o_j) = |\{M_x(o_j, o_i) \mid o_i, o_j \in O, o_i \neq o_j\}|$$

- High IOC => likely to be reused with the objects using its services; faults in objects can be very visible due to frequent requests.

Object Response for Service (OPFS)

- total number of messages sent to the object from all other objects in the application during the execution of a specific scenario x .

$$OPFS_x(o_i) = \left| \bigcup_{j=1}^K M_x(o_i, o_j) \mid o_i, o_j \in O \wedge o_i \neq o_j \right|$$

Incorporating Scenarios profiles

- apply the metric to objects for a given scenario,
- average the measurements weighted by the probability of executing the scenario.

$$IOC(o_i, o_j) = \sum_{x=1}^{|X|} PS_x \times IOC_x(o_i, o_j)$$

$$OPFS(o_i) = \sum_{x=1}^{|X|} PS_x \times OPFS_x(o_i)$$

$$EOC(o_i, o_t) = \sum_{x=1}^{|X|} PS_x \times EOC_x(o_i, o_t)$$

$$OQFS(o_i) = \sum_{x=1}^{|X|} PS_x \times OQFS_x(o_i)$$

Properties and assumptions

- EOC and IOC are greater than or equal zero
- $EOC_x(o_1, o_2) = IOC_x(o_2, o_1)$
- The mechanism by which a message is sent from one object to another is not of concern in defining the metric.
- Access to an object attributes and methods are both treated as messages
- A fixed number of objects during execution of a scenarios is assumed

Conclusions

- Dynamic metrics can be used to measure the actual run-time properties of an application as compared to the expected properties measured by static metrics.
- The proposed metrics do not reflect the complexity or coupling to abstract classes. It is applicable to concrete classes from which objects are executing at run-time.

SOFTWARE DESIGN

Software Quality Attributes

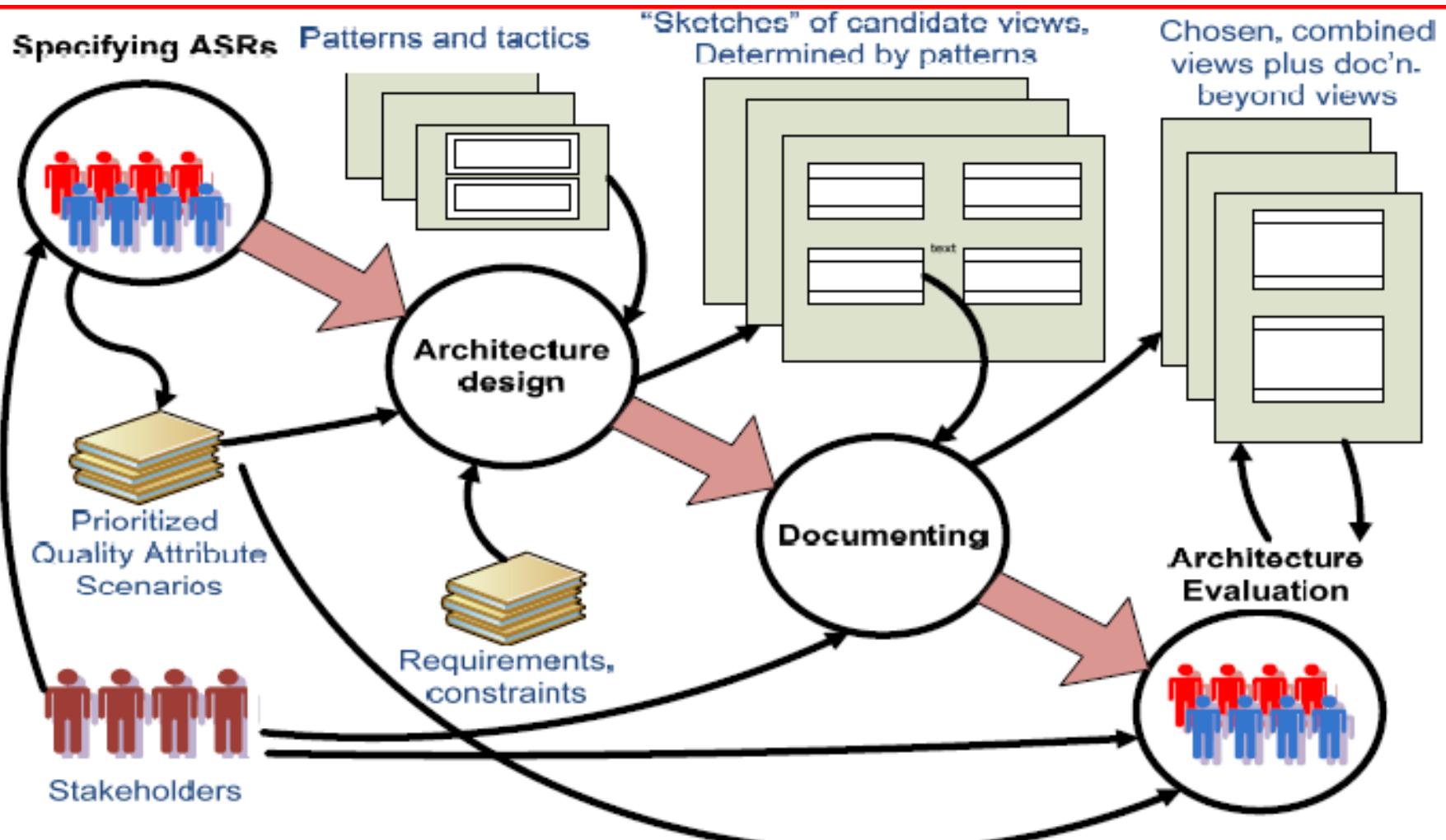
Content

- Requirements Engineering
- Achieving Quality Attributes
- Attribute-Driven Design (ADD)

References

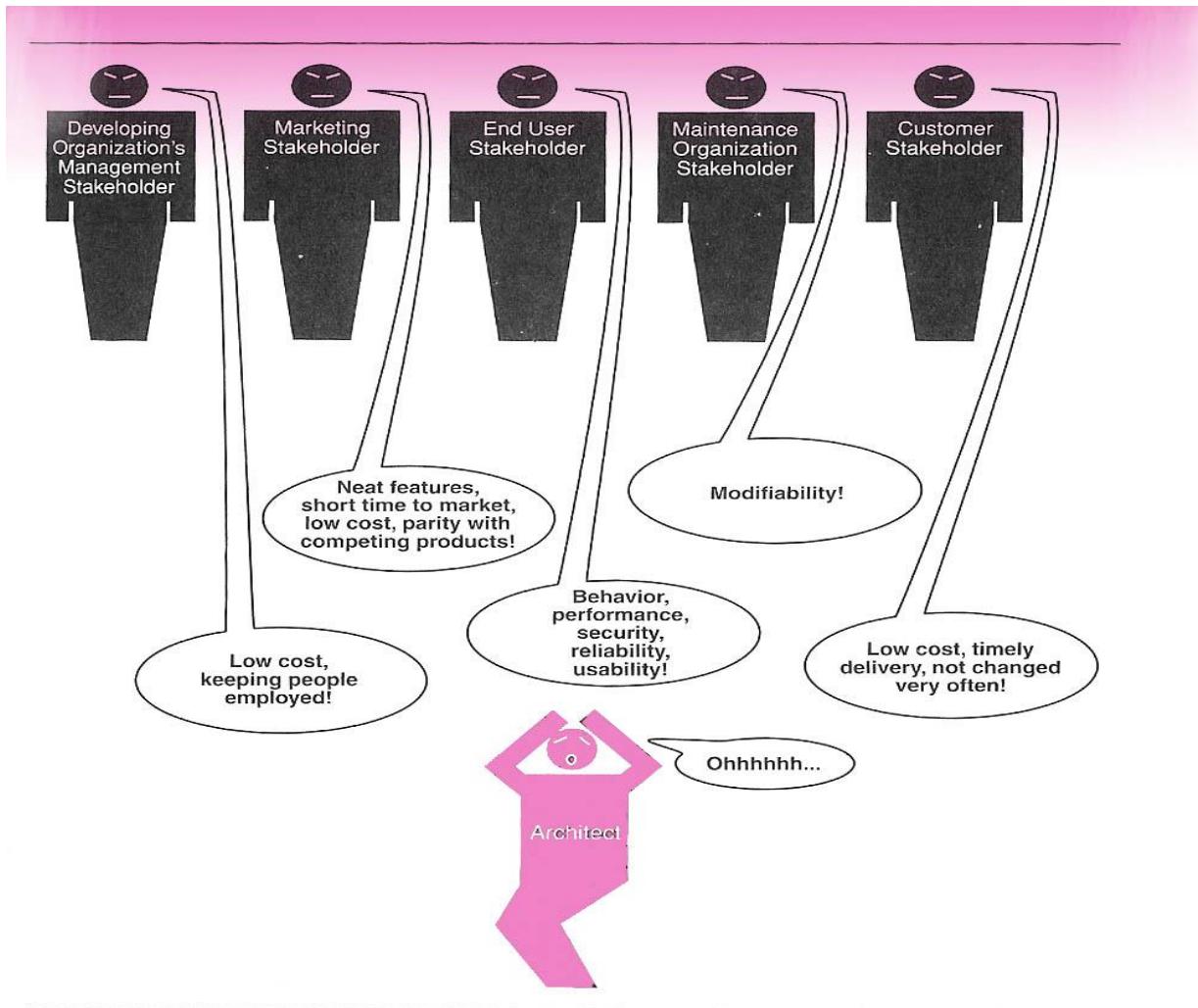
- Len Bass, Paul Clements, Rick Kazman, **Software Architecture in Practice, Second Edition**, Addison Wesley, 2003, ISBN: 0-321-15495-9
- Felix Bachmann, Len Bass, Mark Klein, **Deriving Architectural Tactics: A Step Toward Methodical Architectural Design**,
- TECHNICAL REPORT CMU/SEI-2003-TR-004
- IBM Rational
- Microsoft MSF
- <http://www.cs.uu.nl/wiki/bin/view/Swa/CourseLiterature>

Software Architecture Process



Adapted from Hofmeister et al., 2006.

Stakeholders



- Developing Organization Management
- Marketing
- End User
- Maintenance Organization
- Customer

FIGURE 1.2 Influence of stakeholders on the architect

Requirements elicitation

- Requirements elicitation = the activities involved in discovering the requirements of the system
- Techniques:
 - **Asking:** interview, (structured) brainstorm, questionnaire
 - **Task analysis**
 - **Scenario-based analysis:** ‘think aloud’, use case analysis
 - **Ethnography:** active observation
 - **Form and document analysis**
 - Start from **existing system**
 - **Prototyping**
 - **Own insight**
- General advice: use more than one technique!

Requirements specifications

- Requirements specification = rigorous modeling of requirements, to provide formal definitions for various aspects of the system
- Requirements specification document should be
 - **as precise** as possible: starting point for architecture/design
 - **as readable** as possible: understandable for the user
- Other preferred properties:
 - Correct
 - Unambiguous
 - Complete
 - Consistent
 - Ranked for importance
 - Verifiable
 - Modifiable
 - Traceable

Specification techniques

- Specification techniques:
 - Entity-Relationship (E-R) modeling
 - Structured Analysis and Design Technique (SADT)
 - Finite State Machines
 - Use cases
 - UML

Requirements validation

- Requirements validation = checking the requirements document for consistency, completeness and accuracy
- Validation of requirements needs user interaction
- Techniques:
 - Reviews (reading, checklists, discussion)
 - Prototyping
 - Animation

Use-cases

- Technique for specifying **functional** requirements (What should the system do?)
- A **use case** captures a **contract** between the stakeholders of a system about its behavior
- The use case describes the system's behavior **under various conditions** as the system responds to a request from one of the stakeholders, called the primary actor
- Use cases are fundamentally a text form describing use scenarios

Basic use-case format

- Use case: <use case goal>
- Level: <one of: summary level, user-goal level, subfunction>
- Primary actor: <a role name for the actor who initiates the use case>
- Description: <the steps of the **main success scenario** from trigger to goal delivery and any cleanup after>
- Extension: <alternate scenarios of success or failure>

Use-cases: best practices

- A use case is a prose essay
- Make the use cases easy to read
- Sentence form for the scenario: active voice, present tense, describing an actor successfully achieving a goal
- Include sub-use cases where appropriate
- Keep the GUI out
- Use UML use case diagrams to visualize relations between actors and use cases or among use cases. Use text to specify use cases themselves!
- It is hard, and important, to keep track of the various use cases

Functional requirements

- Pitfalls:
 - Undefined or inconsistent system boundary (scope!)
 - System point of view instead of actor centered
 - Inconsistent actor names
 - Spiderweb actor-to-use case relations
 - Long, too many, confusing use case specifications, incomprehensible to customer
- Beware of:
 - ‘Shopping cart’ mentality
 - The ‘all requirements are equal’ fallacy
 - Stakeholders who won’t read use case descriptions because they find them too technical or too complicated

Architectural drivers

- Functionality is the ability of the system to do the work for which it was intended
- Functionality may be achieved through the use of any number of possible structures
=> functionality is largely independent of structure
- Architecture constrains the mapping of functionality on various structures if **quality attributes** are important

Architectural drivers [2]

- Get the functionality right and then accommodate nonfunctional requirements – NOT POSSIBLE!!!
- Non-functional requirements must be taken into account EARLY ON!!!
- There are two broad categories of non-functional requirements (Design-time and Run-time)

Software Architecture and Quality Attributes

- Quality isn't something that can be added to a software intensive system after development finishes
- Quality concerns need to be addressed during all phases of the software development.
- **BUSINESS GOALS** determine the qualities attributes, which are over and above of system's functionality!!!

Quality requirements objectives

- Input for architecture definition
- Driving architecture evaluation
- Communicating architectural parts and requirements
- Finding missing requirements
- Guiding the testing process

Quality requirements: best practices

- Not all quality attributes are equally important: prioritize!
- Make the quality requirements measurable
 - Not: ‘The system should perform well’ but ‘The response time in interactive use is less than 200 ms’
- Link the quality requirements to use cases
 - Example: ‘The ATM validates the PIN within 1 second’

Change scenarios

- Some quality requirements do not concern functionality but other aspects of the system. These cannot be linked to any use case
 - Mainly attributes from Maintainability and Portability, e.g. Changeability and Adaptability
- Link these quality requirements to specific change scenarios
 - **Not** ‘The system should be very adaptable’ **but** ‘The software can be installed on all Windows and Unix platforms without change to the source code’
 - **Not** ‘The system should be changeable’ **but** ‘Functionality can be added to the ATM within one month that makes it possible for users to transfer money from savings to checking account’

Constraints

- Functional and quality requirements specify the goal, constraints limit the (architecture) solution space
- Stakeholders should therefore not only specify requirements, but also constraints
- Possible constraint categories:
 - Technical, e.g. platform, reuse of existing systems and components, use of standards
 - Financial, e.g. budget
 - Organizational, e.g. processes, availability of customer
 - Time, e.g. deadline

Achieving quality

- Once determined, the quality requirements provide guidance for **architectural decisions**
- An architectural decision that influences the qualities of the product is sometimes called a **tactic**
- Mutually connected tactics are bundled together into **architectural patterns**: schemas for the structural organization of entire systems

Types of requirements (FURPS model)

- **Functionality**
 - feature sets
 - capabilities
 - security
- **Usability**
 - human factors
 - aesthetics
 - consistency in the user interface
 - online and context-sensitive help
 - wizards and agents
 - user documentation
 - training materials
- **Reliability**
 - frequency and severity of failure
 - recoverability
 - predictability
 - accuracy
 - mean time between failure

Types of requirements [2]

- **Performance**
 - speed
 - efficiency
 - availability
 - accuracy
 - response time
 - recovery time
 - resource usage
- **Supportability**
 - testability
 - extensibility
 - adaptability
 - maintainability
 - compatibility
 - configurability
 - installability
 - localizability (internationalization)

Quality attributes

- Business quality:
 - Time to market – “Time”
 - Cost and benefit – “Economy”
 - Projected lifetime – “Form”
 - Target market – “Function”
 - Rollout schedule – “Time”
 - Integration with legacy – “Time”

Quality attributes

- Architectural quality
 - Conceptual integrity
 - Correctness and completeness
 - Buildability

Quality Attributes

- System quality
 - Availability
 - Performance
 - Security
 - Modifiability
 - Testability
 - Usability

System Quality attributes

Are defined in terms of scenarios

- **source of stimulus** [the entity (human or another system) that generated the stimulus or event.] **who?**
- **stimulus** [a condition that determines a reaction of the system.] **what?**
- **environment** [the current condition of the system when the stimulus arrives.] **when?**
- **artifact** [is a component that reacts to the stimulus. It may be the whole system or some pieces of it.] **where?**
- **response** [the activity determined by the arrival of the stimulus.] **which?**
- **response measure** [the quantifiable indication of the response.] **how?**

General Scenario

76 Part Two Creating an Architecture

4—Understanding Quality Attributes

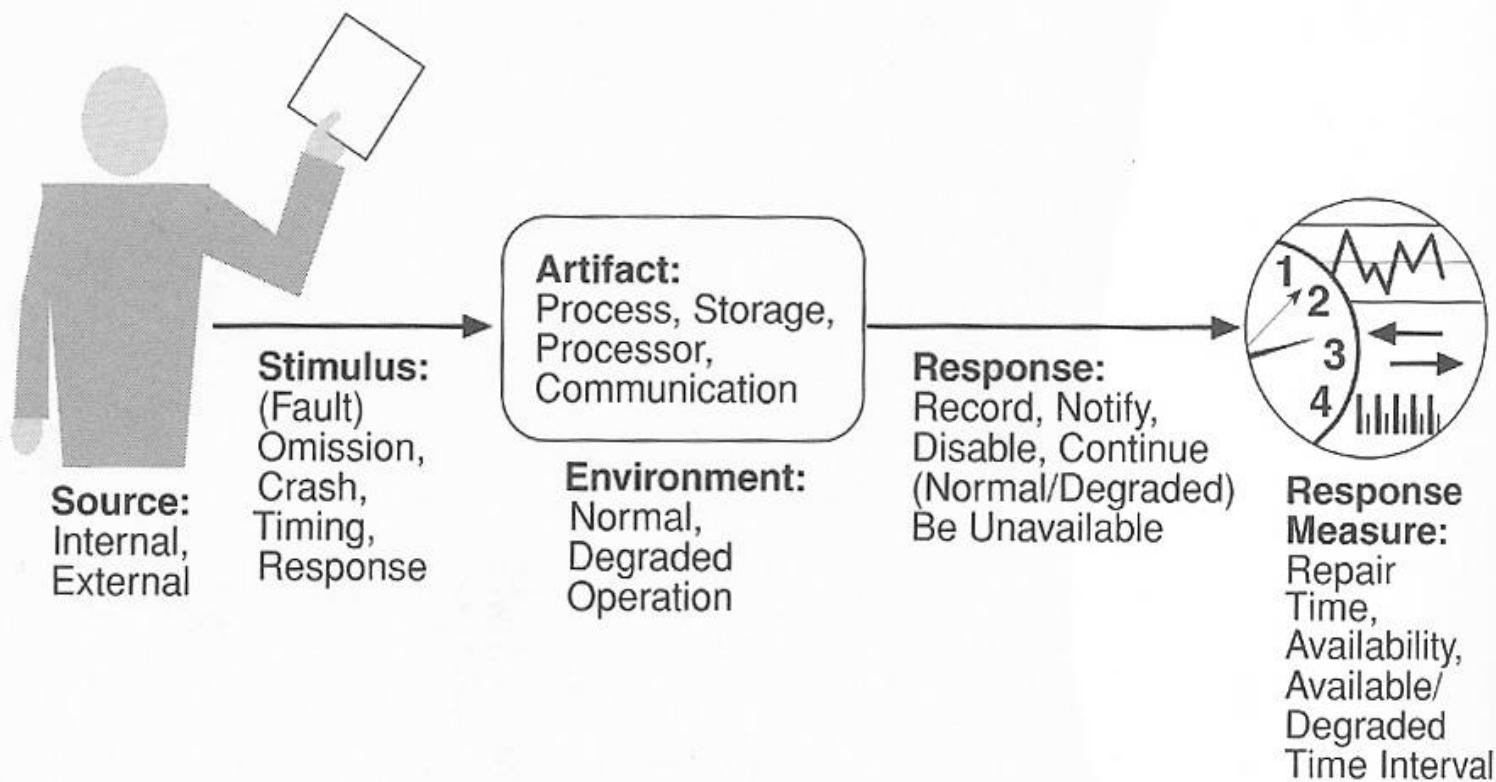
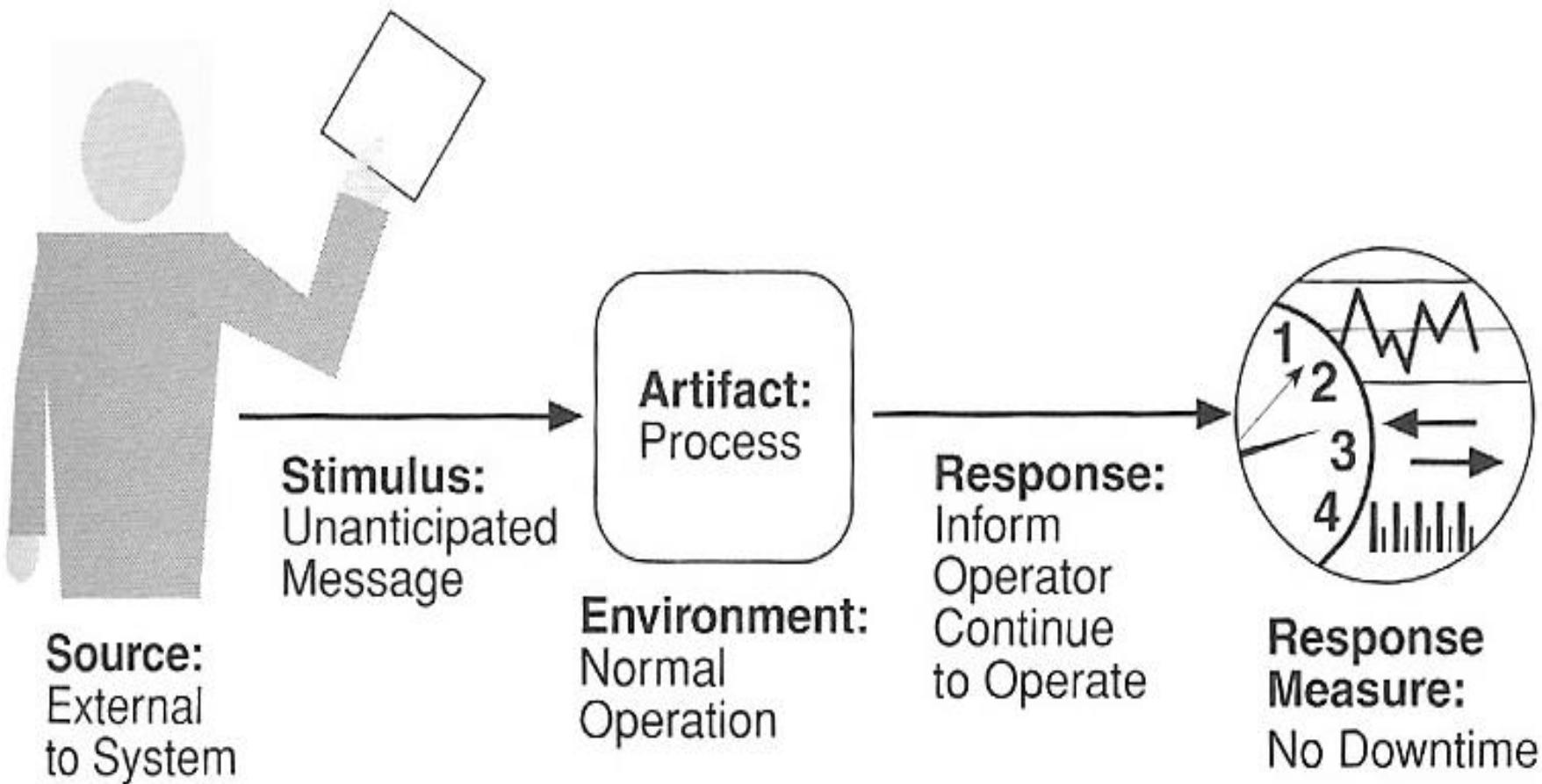


FIGURE 4.2 Availability general scenarios

Concrete Scenario



Example

A large number of requests periodically arrives on an individual data entity attribute from a user interface at the system under normal condition. The system has to transfer the data within a certain amount of time without generating too many network calls

Elements	Refined General Scenarios
Source	User interface
Stimulus	A large number of requests on data entity attribute periodically arrives
Environment	Normal condition
Artifact	System
Response	Transfer the data
Response Measure	Within 10 ms?

Availability

- typically defined as the probability of a system to be operational when needed in terms of

mean time to failure / (mean time to failure + mean time to repair)

Availability scenario

- **Source of stimulus:** internal or external
- **Stimulus:**
 - omission: a component fails to respond to an input.
 - crash: a component repeatedly suffers omission faults.
 - timing: a component responds, but the response is early or late.
 - response: a component responds with an incorrect value.
- **Artifact:** the resource that is required to be available (i.e. processor, communication channel, process, or storage device).

Availability Scenario

- **Environment:** defines the state of the system when the fault or failure occurred: normal, degraded
- **Response:** logging, notification, switching to backup, restart, shutdown
- **Response measure:**
 - the availability percentage,
 - a time for repair,
 - certain times during which the system must be available,
 - the duration for which the system must be available.

POS – Quality Attribute Scenario 1

- **Scenario(s)**: The barcode scanner fails; failure is detected, signalled to user at terminal; continue in degraded mode
- **Stimulus Source** : Internal to system
- **Stimulus**: Fails
- **Environment**: Normal operation
- **Artefact (If Known)**: Barcode scanner
- **Response**: Failure detected, shown to user, continue to operate
- **Response Measure**: No downtime, React in 2 seconds

POS – Quality Attribute Scenario 2

- **Scenario(s)**: The inventory system fails and the failure is detected. The system continues to operate and queue inventory requests internally; issue requests when inventory system is running again
- **Stimulus Source** : Internal to system
- **Stimulus**: Fails
- **Environment**: Normal operation
- **Artefact (If Known)**: Inventory system
- **Response**: Failure detected, operates in degraded mode, queues requests, detects when inventory system is up again
- **Response Measure**: Degraded mode is entered for maximum one hour

Tactics to achieve availability

- for fault detection
- for fault recovery
- for fault prevention

Tactics for Fault Detection

- Ping/echo
 - Signal is issued, response is waited for
 - Estimates round-trip time and rate of package loss
- Heartbeat
 - Periodic signal is broadcasted
- Exception handling

Tactics for Fault Recovery

- Voting
 - run the same algorithm on different processors.
"majority rules" algorithm uncovers any deviant behavior in the processors.
- Active redundancy
 - Set up redundant components and keep them synchronized
- Passive redundancy
 - have only one component respond to events, but have that component inform redundant components about the changes

Tactics for Fault Recovery [2]

- Spare
 - a standby spare computing platform is prepared to replace many different failed components. Backup + logs needed.
- Shadow operation
 - A previously failed component may be run in "shadow mode" for a short time to make sure that it mimics the behavior of the working component before it is restored to service.

Tactics for Fault Prevention

- Removal from service
 - removal of a component of the system from operation in order to update it and avoid potential failures.
 - Automatic
 - Manual
- Transactions
 - set of operations where either all or none are executed successfully.
- Process monitor
 - If a fault is detected in a process, an automated monitoring process can delete the failed process and create a new instance of it, initializing it to some appropriate state as in the spare tactic

Performance

- **Performance** refers to the time it takes the system to respond to an event. The event can be fired by:
 - a user,
 - another system,
 - the system itself.

Performance scenario

- **Source of stimulus:** The stimuli arrive either from external (possibly multiple) or internal sources.
- **Stimulus:** The stimuli are the event arrivals. The arrival pattern can be characterized as periodic, stochastic, or sporadic.
 - **Periodic** means that the events arrive in regular intervals of time
 - **Stochastic** means that the arrival of events is based on some probabilistic distribution
 - **Sporadic** means that the events arrive rather randomly.
- **Artifact.** The artifact is always the system's service, which has to respond to the event.
- **Environment.** The system can be in various operational modes, such as normal, emergency, or overload. The response varies depending on the current state of the system.

Performance scenario

- **Response.** The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode). The response of the system can be characterized by:
 - **latency** (the time between the arrival of the stimulus and the system's response to it),
 - **deadlines** in processing (a specific action should take place before another),
 - **throughput** of the system (e.g., the number of transactions the system can process in a second),
 - **jitter** of the response (the variation in latency),
 - **number of events not processed** because the system was too busy to respond,
 - **lost data** because the system was too busy.
- **Response measure.** Response measures include the time it takes to process the arriving events (latency, or deadlines by which the events must be processed), variations in this time (jitter), the number of events that can be processed within a particular time interval (throughput), and the characterization of the events that cannot be processed (miss rate, data loss).

Scenario profile for performance

Quality Factor	Scenario description
Initialization	The system Must perform all initialization activities within 10 minutes.
Latency	The system shall Run simulations with no instantaneous lags greater than five seconds, no average lags greater than three seconds.
Capacity	The system shall be able to provide run-time simulation with debug enabled.
Latency	A sensor shall finish data collection within 30 seconds of simulation termination.
Throughput	The system shall finish data collection request from three network sensors within 10 seconds.

POS Case Study

- **Scenario(s)**: The POS system scans a new item, item is looked up, total price updated within two seconds
- **Stimulus Source** : End user
- **Stimulus**: Scan item, fixed time between events for limited time period
- **Environment**: Development time
- **Artefact (If Known)**: POS system
- **Response**: Item is looked up, total price updated
- **Response Measure**: Within two seconds

Throughput

- Measure of the amount of work an application must perform in unit time
 - Transactions per second (TPS)
 - Messages per second (MPS)
- Is required throughput:
 - Average?
 - Peak?
- Many system have low average but high peak throughput requirements

Response time

- Measure of the latency an application exhibits in processing a request
- Usually measured in (milli)seconds
- Often an important metric for users
- Is required response time:
 - Guaranteed?
 - Average?
- E.g. 95% of responses in sub-4 seconds, and all within 10 seconds

Deadlines

- ‘something must be completed before some specified time’
 - Payroll system must complete by 2am so that electronic transfers can be sent to bank
 - Weekly accounting run must complete by 6am Monday so that figures are available to management
- Deadlines often associated with batch jobs in IT systems.

Attention!

- What is a
 - Transaction?
 - Message?
 - Request?
- All are application specific measures.
- System must achieve 100 mps throughput
 - BAD!!
- System must achieve 100 mps peak throughput for *PaymentReceived* messages
 - GOOD!!!

Factors affecting performance

- Resource Consumption
- Blocked time
 - Contention for resources
 - Availability of resources
 - Dependency on other computation

Tactics to achieve performance

- resource demand,
- resource management
- resource arbitration.

Resource demand

- **Increase computational efficiency.**
 - Improving the algorithms,
 - Trading one resource for another.
- **Reduce computational overhead.**
 - Use local class vs. RMI
 - Use of intermediaries => the classic modifiability/performance tradeoff.

Resource demand - reduce the number of processed events

- **Manage event rate.**
 - Reduce the sampling frequency at which environmental variables are monitored
- **Control frequency of sampling.**
 - If there is no control over the arrival of externally generated events, queued requests can be sampled at a lower frequency, possibly resulting in the loss of requests.

Resource demand – control the use of resources

- **Bound execution times.**
 - Place a limit on how much execution time is used to respond to an event.
- **Bound queue sizes.**
 - This controls the maximum number of queued arrivals and consequently the resources used to process the arrivals.

Resource management

- **Introduce concurrency.**

- process different streams of events on different threads
- create additional threads to process different sets of activities.
- appropriately allocate the threads to resources (load balancing)

- **Maintain multiple copies of either data or computations.**

- clients in a client-server pattern are replicas of the computation.
- caching is a tactic in which data is replicated,
- keeping the copies consistent and synchronized becomes a responsibility that the system must assume.

- **Increase available resources.**

- faster processors, additional processors, additional memory, faster networks.

Resource arbitration - Scheduling

- **First In, First Out (FIFO).**
 - queues that treat all requests equally (all have the same priority). Requests are ordered by time of arrival.
- **Fixed priority.**
 - assign to each resource requester a priority, and treat the requests issued by high-priority requesters first.
 - Strategies:
 - **semantic importance.** Priority is assigned based on some domain characteristic.
 - **deadline monotonic.** This is a static strategy that assigns higher priority to requesters with shorter deadlines.
 - **rate monotonic.** This is a static strategy for periodic requesters; higher priority is assigned to requesters with shorter periods.

Resource scheduling

- **Dynamic priority.**
 - ordering according to a criterion and then, at every assignment possibility, assign the resource to the next request in that order.
 - earliest deadline: the priority is assigned to the pending request with the earliest deadline.
- **Static scheduling.**
 - determine the sequence of assignment offline.

Security

- Security is a measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users.
- An attempt to breach security is called an attack

Security features

- **Nonrepudiation** is the property that a transaction (access to or modification of data or services) cannot be denied by any of the parties to it.
- **Confidentiality** is the property that data or services are protected from unauthorized access.
- **Integrity** is the property that data or services are being delivered as intended.
- **Assurance** is the property that the parties to a transaction are who they purport to be.
- **Availability** is the property that the system will be available for legitimate use.
- **Auditing** is the property that the system tracks activities within it at levels sufficient to reconstruct them.

Security scenario

- **Source** - Individual or system
 - that is correctly identified, identified incorrectly, of unknown identity
 - who is internal/external, authorized/not authorized
 - with access to limited resources, vast resources
- **Stimulus** - Tries to
 - display data, change/delete data, access system services, reduce availability to system services
- **Artifact** - System services; data within system
- **Environment** -
 - online or offline,
 - connected or disconnected,
 - firewalled or open

Security scenario continued

- **Response**
 - Authenticates user; hides identity of the user; blocks access to data and/or services; allows access to data and/or services; grants or withdraws permission to access data and/or services; records access/modifications or attempts to access/modify data/services by identity; stores data in an unreadable format; recognizes an unexplainable high demand for services, and informs a user or another system, and restricts availability of services
- **Response Measure**
 - Time/effort/resources required to circumvent security measures with probability of success; probability of detecting attack; probability of identifying individual responsible for attack or access/modification of data and/or services; percentage of services still available under denial-of-services attack; restore data/services; extent to which data/services damaged and/or legitimate access denied

Security scenarios

Series No.	Security Requirements	Security properties
SR1	A system shall accept online payments for the services, which means the transactions between the system and financial institutes must be protected.	Private communication/information protection- Defense in depth
SR2	A system provides secured storage to customers' credit details and other information.	Data protection
SR3	A system shall be able to identify different users and verify their access privileges according to their account types.	User identification Access verification
SR4	A system shall be able to detect and prevent Denial Of Service (DOS) attacks. The system shall be able to run reliably most of the time.	Reducing exposure to attack / Error prevention & handling
SR5	A system is an evolving system that shall be easily modifiable to introduce changes in the security policy and other security checks.	Encapsulation of Security policy/ Initializaiton process

Dealing with Security Risks

Vulnerability Category	Potential Problem Due to Bad Design
Input / Data Validation	Insertion of malicious strings in user interface elements or public APIs. These attacks include command execution, cross - site scripting (XSS), SQL injection, and buffer overflow. Results can range from information disclosure to elevation of privilege and arbitrary code execution.
Authentication	Identity spoofing, password cracking, elevation of privileges, and unauthorized access.
Authorization	Access to confidential or restricted data, data tampering, and execution of unauthorized operations.
Configuration Management	Unauthorized access to administration interfaces, ability to update configuration data, and unauthorized access to user accounts and account profiles.
Sensitive Data	Confidential information disclosure and data tampering.
Cryptography	Access to confidential data or account credentials, or both.
Exception Management	Denial of service and disclosure of sensitive system-level details.
Auditing and Logging	Failure to spot the signs of intrusion, inability to prove a user's actions, and difficulties in problem diagnosis.

Principles for Security Strategies

Category	Guidelines
Input / Data Validation	Do not trust input; consider centralized input validation. Do not rely on client-side validation. Be careful with canonicalization issues. Constrain, reject, and sanitize input. Validate for type, length, format, and range.
Authentication	Use strong passwords. Support password expiration periods and account disablement. Do not store credentials (use one-way hashes with salt). Encrypt communication channels to protect authentication tokens.
Authorization	Use least privileged accounts. Consider authorization granularity. Enforce separation of privileges. Restrict user access to system-level resources.
Configuration Management	Use least privileged process and service accounts. Do not store credentials in clear text. Use strong authentication and authorization on administration interfaces. Do not use the Local Security Authority (LSA). Secure the communication channel for remote administration.

Principles for Security Strategies

Sensitive Data	Avoid storing secrets. Encrypt sensitive data over the wire. Secure the communication channel. Provide strong access controls for sensitive data stores.
Cryptography	Do not develop your own. Use proven and tested platform features. Keep unencrypted data close to the algorithm. Use the right algorithm and key size. Avoid key management (use DPAPI). Cycle your keys periodically. Store keys in a restricted location.
Exception Management	Use structured exception handling. Do not reveal sensitive application implementation details. Do not log private data such as passwords. Consider a centralized exception management framework.
Auditing and Logging	Identify malicious behavior. Know what good traffic looks like. Audit and log activity through all of the application tiers. Secure access to log files. Back up and regularly analyze log files.

Security tactics

- Resisting attacks
- Detecting attacks
- Recovering from attacks

Resisting attacks

- Authenticate users.
 - Ensure that a user or remote computer is actually who it purports to be. Passwords, one-time passwords, digital certificates, and biometric identifications provide authentication.
- Authorize users.
 - ensure that an authenticated user has the rights to access and modify either data or services. This is usually managed by providing some access control patterns within a system. Access control can be by user or by user class. Classes of users can be defined by user groups, by user roles, or by lists of individuals.
- Maintain data confidentiality.
 - encryption to data and to communication links. The link can be implemented by a virtual private network (VPN) or by a Secure Sockets Layer (SSL) for a Web-based link. Encryption can be symmetric (both parties use the same key) or asymmetric (public and private keys).

Resisting attacks

- Maintain integrity.
 - Data should be delivered as intended. It can have redundant information encoded in it, such as checksums or hash results, which can be encrypted either along with or independently from the original data.
- Limit exposure.
 - The architect can design the allocation of services to hosts so that limited services are available on each host.
- Limit access.
 - Firewalls restrict access based on message source or destination port. It is not always possible to limit access to known sources. One configuration used in this case is the so-called demilitarized zone (DMZ).

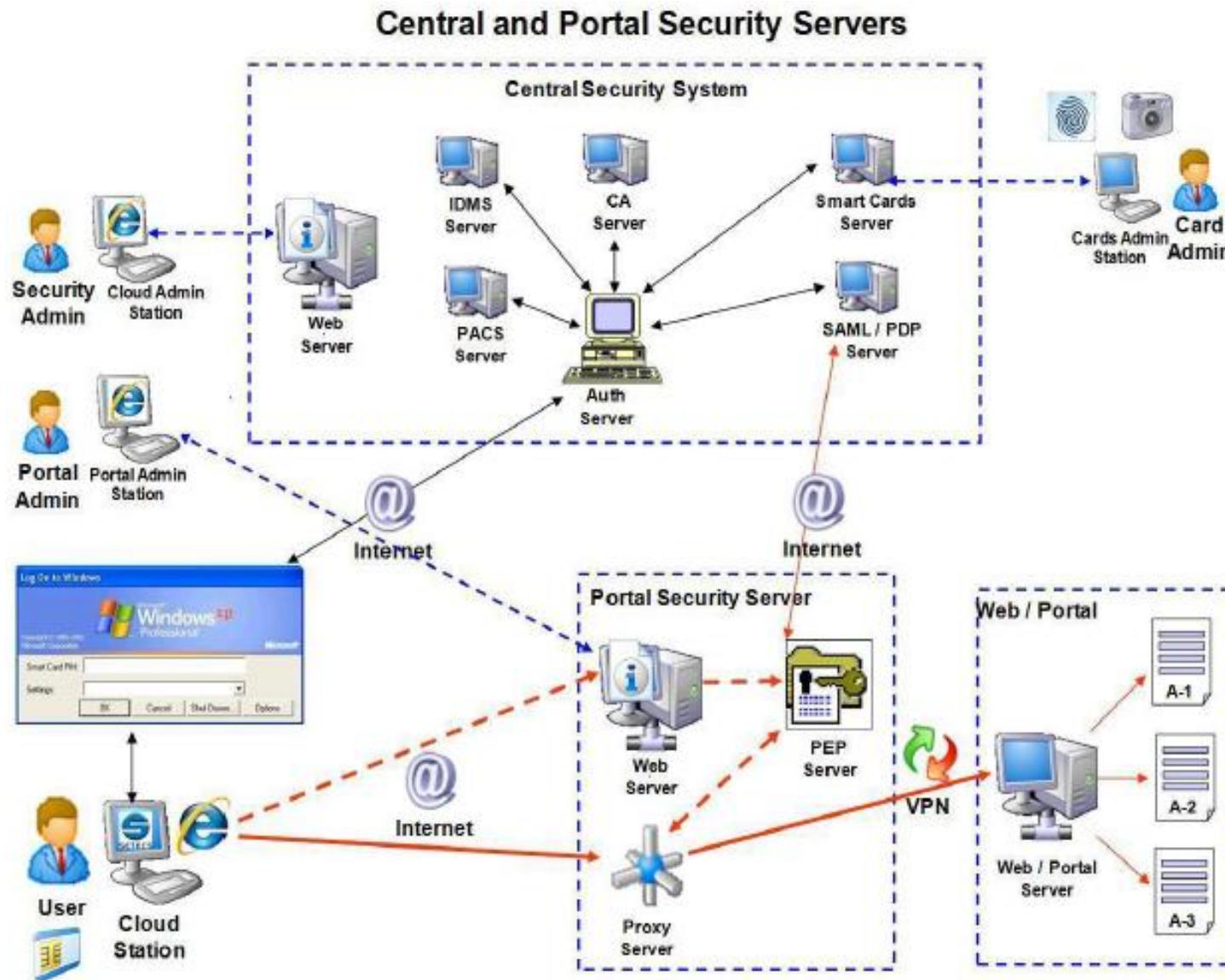
Security tactics

- Detecting attacks
 - Intrusion detection system
- Recovering from attacks
 - restoring state (availability)
 - attacker identification (nonrepudiation)

Cloud-based security vulnerabilities

- Identified by CSA, NIST and ENISA
- Data Privacy and Reliability
- Data Integrity
- Authentication and Authorization
 - authentication frameworks like OpenID, SAML, Shibboleth

Cloud Security Infrastructure example



Other Quality attributes

- Modifiability
 - **Source:** developer, administrator, user
 - **Stimulus:** add/delete/modify function or quality
 - **Artifact:** UI, platform, environment
 - **Environment:** design, compile, build, run
 - **Response:** make change and test it
 - **Measure:** effort, time, cost

Testability

- **Source:** developer, tester, user
- **Stimulus:** milestone completed
- **Artifact:** design, code component, system
- **Environment:** design, development, compile, deployment, run
- **Response:** can be controlled and observed
- **Measure:** coverage, probability, time

Example testability scenario

- **Source:** Unit tester
- **Stimulus:** Performs unit test
- **Artifact:** Component of the system
- **Environment:** At the completion of the component
- **Response:** Component has interface for controlling behavior, and output of the component is observable
- **Response Measure:** Path coverage of 85% is achieved within 3 hours

Usability

- **Source:** end user
- **Stimulus:** wish to learn/use/minimize errors/adapt/feel comfortable
- **Artifact:** system
- **Environment:** configuration or runtime
- **Response:** provide ability or anticipate
- **Measure:** task time, number of errors, user satisfaction, efficiency

Lessons learned

- Requirements engineering is important and not trivial.
Involves:
 - Elicitation
 - Specification
 - Validation
- Architecture is driven by requirements:
 - Functional
 - Non-functional (quality attributes)
- Quality attributes
 - Defined as scenarios
 - Achieved using appropriate tactics

Design trade-offs

- QAs are rarely orthogonal
 - They interact, affect each other
 - highly secure system may be difficult to integrate
 - highly available application may trade-off lower performance for greater availability
 - high performance application may be tied to a given platform, and hence not be easily portable
- Architects must create solutions that makes sensible design compromises
 - Not possible to fully satisfy all competing requirements
 - Must satisfy all stakeholder needs
 - This is the difficult bit!

SOFTWARE DESIGN REVIEW

Summary of topics

- Introduction
- Architectural Styles and Patterns
- Design Patterns
- Class&Package Design Principles
- Software Quality Attributes

Introduction

- OOP review
 - Classes
 - Objects
 - Relationships: inheritance, composition
 - Abstract classes, Interfaces
 - Polymorphism
 - Nested classes

Architectural Patterns

- Layers
 - Client-Server
- Event-driven
 - Broker
 - Mediator
- MVC (and variants)
- Service-based
 - SOA
 - Microservices
- Space-based (Cloud)

References

- David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Ed.[Patterson]
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley [Taylor]
- F. Buschmann et. al, PATTERN-ORIENTED SOFTWARE ARCHITECTURE: A System of Patterns, Wiley&Sons, 2001.[POSA]
- Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]
- Mark Richards, Software Architecture Patterns, O'Reilly, 2015 [SAP]
- Mark Richards, Microservices vs. Service-Oriented Architecture O'Reilly, 2016
- Jacques Roy, SOA and Web Services, IBM
- Mark Bailey, Principles of Service Oriented Architecture, 2008

Types of Questions

- Given a specification
 - ⇒ Analyze different AP for a solution and justify the most appropriate one
- Given an architecture
 - ⇒ Identify the used APs and analyze their consequences.
 - ⇒ Describe how different APs are related
- DON'T!!
 - Enumerate all the AP you know/copy

Patterns for Enterprise Application Architecture

- Domain Layer Patterns
 - Transaction Script
 - Domain Model
 - Table Module
 - Active Record
- Data Source Patterns
 - Row Data Gateway
 - Table Data Gateway
 - Data Mapper
- Presentation Patterns
 - Template and Transform View
 - Front and Page Controller
- Concurrency Patterns (optimistic, pessimistic)
- References
 - Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]

Types of Questions

- Given a specification
⇒ Analyze different design approaches and justify the most appropriate one
- Explain how different patterns relate to each other
 - Ex. Gateway/Façade/Adapter
- Discuss different design approaches to address different issues (ex. data access/mapping inheritance).
- Refer to the asked question, don't present general topics.

Design Patterns

- Creational
- Structural
- Behavioral
- Service-oriented
- including what you learned in previous courses!
- References
 - Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2
 - Erl, Thomas. Service-Oriented Architecture: Analysis and Design for Services and Microservices. Pearson Education. 2016
 - Erl, Thomas. SOA Design Patterns, Prentice Hall, 2009.
 - <http://soapatterns.org>

Types of Questions

- Given a specification and some quality criteria
 - ⇒ Identify applicable design patterns (there is no unique solution) and analyze them (pros and cons for each)
 - ⇒ Identify ways in which patterns can be combined to obtain a more complex behavior
- Explain how different pattern relate to each other (common/different aspects)
- Given a certain design
 - ⇒ Recognize the applied design patterns. Analyze.
- Given a specification of a business process, identify appropriate services that can be used to compose a service-based solution

Class & Package Principles

- SOLID
- GRASP
- Package-related Cohesion Principles
- Package-related Coupling Principles
- References
 - Robert Martin Articles
 - Solid eBook
http://cdn.cloudfiles.mosso.com/c82752/pablos_solid_ebook.pdf
 - Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd Ed, Addison Wesley, 2004 – Chapters 17, 18.

Types of Questions

- In a given design recognize what principles were respected and what not
- In a given design identify the DP that are compliant with design principles
- Show relationships between principles and DP (how DP support certain principles)
- DON'T!
Enumerate all the DP you know/copy or all the principles you know/copy.

Software Quality attributes

- Representing functional requirements (UC diagrams)
- Representing nonfunctional requirements (Scenarios)
- Tactics to address different scenarios
- References
 - Len Bass, Paul Clements, Rick Kazman, **Software Architecture in Practice, Second Edition**, Addison Wesley, 2003, ISBN: 0-321-15495-9
 - <http://www.cs.uu.nl/wiki/bin/view/Swa/CourseLiterature>
 - Felix Bachmann, Len Bass, Mark Klein, **Deriving Architectural Tactics: A Step Toward Methodical Architectural Design**,
 - TECHNICAL REPORT CMU/SEI-2003-TR-004, IBM Rational, Microsoft MSF

Types of Questions

- Given a NL specification
 - ⇒ Identify and represent the functional req
 - ⇒ Identify and represent the nonfunctional req
 - ⇒ Recommend appropriate tactics ADAPTED to the problem. JUSTIFY!
- DON'T!!
Enumerate all the QA and tactics you know/copy

Example 1

- A Security Central Office (SCO) monitors the security systems of several clients. The security system of a client is of one of several possible types. There can be different types of contracts between the clients and the SCO. Depending on the contract there is a different insurance type attached to the contract. You have to design a management system for SCO. The system should allow the following operations: CRUD operations for clients, CRUD operations for the security system of a client, monitor a security system of a client, react to an event, CRUD operations for the contract between clients and SCO.
- Propose an architecture for the system described above. Highlight the strong and weak points of your architecture.
- Sketch the UML diagram of the system including appropriate design pattern(s).

Example 2

In a web-based class management system, the instructor can see the list of classes he taught in the last 10 semesters, can select a class and see the students enrolled in it, and he can set the grade of each enrolled student if the gradebook is not closed. When a grade is changed the student gets a notification by e-mail.

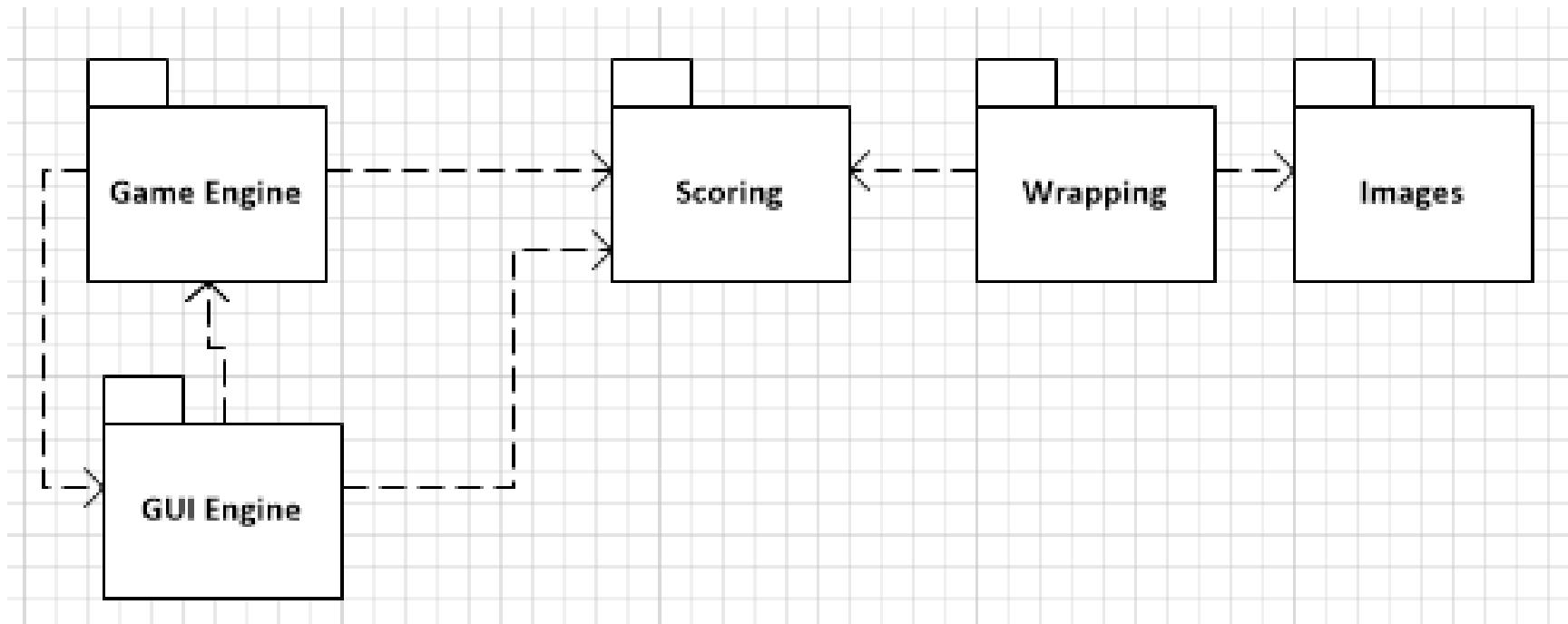
- Based on a layered architecture propose a solution covering all the 3 layers. Highlight any other patterns that are included in your design. Discuss at least 2 advantages and 2 disadvantages of your solution

Example 3

- State for the three cases below if they satisfy the Liskov Substitution Principle and explain why:
 - We have a List class, and we have a subclass of List, OrderedList, which provides the same interface but returns an ordered list
 - We have a List class, and an ArrayList subclass, which uses an array in its implementation in order to store the list elements
 - We have a List class, and a BoundedList subclass which only holds Lists of size n

Example 4

Given the following package design show how the package design principles are respected or not



Exam format

- Theoretical exercises (4)
 - As discussed above
 - Graded by Mihaela Dinsoreanu
- Problem
 - Based on the lab assignments
 - Graded by Teaching Assistant
- 3 hours
- **ANY DETECTED ATTEMPT OF CHEATING WILL BE PUNISHED ACCORDING TO THE TUCN REGULATIONS!**

- Questions?
- Thank you for your participation!
- Good luck with your exams!

Software Design

Prof. Mihaela Dinsoreanu

Contact: room D01, Baritiu 26-28

E-mail: mihaela.dinsoreanu@cs.utcluj.ro

Housekeeping Details

- Time & Location:
 - See Schedule on www.ac.utcluj.ro
- Prerequisites:
 - Programming Techniques Course
 - Software Engineering Course
- Grading:
 - Project 20%
 - Lab 20%
 - Final Exam 60%

Housekeeping continued..

- TA's

ing. Timotei Dolean – 30231, 30236

ing. Grigore Vlad – 30232

ing. Lucian Braescu - 30233

ing. Flaviu Zapca – 30234

ing. Paul Stanescu - 30235

ing. Titus Giuroiu – 30237, 30432, 30434

ing. Andrei Corovei – 30238

ing. Vasile Cimpean – 30239

ing. Alexandru Cosma - 30431

ing. Tudor Vlad – 30433

dr.inf. Camelia Chira – CSC

Housekeeping continued..

- Lab sessions are compulsory
 - Maximum 3 absences allowed (should be caught up)
 - Only one assignment/week can be presented
 - **Attend the lab sessions only with your group**
- 2 types of projects
 - Common (1 semester)
 - Research-oriented (can be continued as DS projects and Diploma projects)
 - Data/knowledge fusion
 - Knowledge representation (adaptive)
 - Knowledge extraction (Data mining, text mining, time series analysis, etc)
 - Machine learning and reasoning
- Course files
<http://users.utcluj.ro/~dinso/PS2017...>to be set up

References

- Software Architectures
 - Ian Gorton, Essential Software Architecture, Springer, second ed. 2011.
 - Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley.
 - Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice, 3rd edition, 2013.
 - David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Ed.
 - Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sornmerlad, and Michael Stal. 2001. *Pattern-oriented system architecture, volume 1: A system of patterns*. Hoboken, NJ: John Wiley & Sons. [POSA book]
 - Fowler Martin, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002
- Design Patterns
 - E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. AddisonWesley, 1995. [GoF]
 - Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition), Prentice Hall, 2004, ISBN: 0131489062
- Courses
 - B. Meyer (ETH Zurich)
 - G. Kaiser (Columbia Univ. NY)
 - I. Crnkovic (Sweden)
 - (Univ. of Copenhagen)
 - R. Marinescu (Univ. Timisoara)
 - SaaS (Stanford)

Course Content [Tentative]

1. Introduction, Basic Concepts Review (Classes, objects, relationships, polymorphism)
2. Architectural patterns Structural, Distributed
3. Patterns for Enterprise Application Architecture: Intro, Business Logic
4. Patterns for Enterprise Application Architecture: Data Source, Hybrid
5. Patterns for Enterprise Application Architecture: Presentation, General
6. Creational DP
7. **Midterm????**
8. Structural DP
9. Behavioral DP
10. Class Design principles (SOLID)
11. Package Design Principles, Basic software design metrics
12. Architecting for quality attributes
13. Service oriented architectures (SOAP, REST)
14. Architectures Evaluation and Review

Why study Software Architecture?

PROS of a Career as a Software Architect

Very rapid job growth is expected for software developers (22% increase in jobs between 2012 and 2022)*

High salary potential (median \$102,880 for systems software developers as of May of 2014)*

Opportunity to work remotely*

Creativity is a valued asset*

CONS of a Career as a Software Architect

Long hours are often necessary*

Some companies outsource their software architecture work*

Software needs to be updated consistently to keep up with technological developments *

Some positions may require a master's degree*

*Source: *U.S. Bureau of Labor Statistics*

Objectives

After completing this course, you should be able to:

- **Identify** the most relevant functional and non-functional **requirements** of a software system and to document them
- Generate **architectural alternatives** for a problem and select among them
- **Design and motivate software architectures** for (large scale) software systems
- **Recognize and apply** major **software architectural styles**, design patterns, and frameworks
- **Describe a software architecture** using various documentation approaches and architectural description languages

Software Design Techniques

What are Software Design Techniques?

- SD Techniques provide a set of practices for analysing, decomposing, and modularising software system architectures
- characterized by structuring the system architecture on the basis of its *components* rather than the *actions* it performs.

Learning SD Techniques

Rules



Junior Developer

- algorithms, data structures and programming languages
- write programs, although not always good ones

Principles



Senior Developer

- software design & programming paradigms with pros and cons
- importance of cohesion, coupling, information hiding, dependency management etc.

Patterns of solutions



Technical Architect

- design models
- understand how design solutions interact and can be integrated

Where do you stand?

You know the Rules

- 1-2 OO programming languages (Java, C++, C#)
- some experience in writing programs (< 10 KLOC)

You heard about Principles

- "Open-Closed"; "Liskov Substitution Principle" etc.
- Maybe (in)voluntary applied some of them

You aim to become "design masters"

- writing good software is not some "magic"
- not exclusively tailored for geniuses, gurus

What do you need?

Knowledge

- Is obtained by reading book or attending lectures – terminology, concepts, principles, methods, and theories

Understanding

- is obtained from using your knowledge for activities in the course, e.g., practical or theoretical exercises, assignments, projects, discussions

Skills

- are obtained by actively using knowledge and understanding of the subject matter – working hard is one secret of becoming skillful

Today's outline

- Object Oriented Concepts Review
 - Classes
 - Objects
 - Relationships: inheritance, composition
 - Abstract classes, Interfaces
 - Polymorphism
 - Nested classes

What is a class?

- A type that encapsulates
 - State (Attributes)
 - Constructors
 - Behavior (Methods)
- Class candidates:

- Person



- Mihaela



- AddAccount



How to declare a class (java)

```
class ClassName [extends ParentName implements  
InterfaceName(s)]  
{  
    [modifier(s)] type attribute1;  
    ...  
    [modifier(s)] return_type method1(param_list)  
    {//method body here}  
}
```

Access modifiers: public, protected, private

“Mutability” modifier: final

“Scope” modifier: static

Access modifiers in Java

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

The Person class

```
class Person {  
    private int birthYear;  
    private String firstName, lastName;  
    private boolean employed;  
    private int nrOfLegs;  
  
    //constructor(s)  
    //setters & getters  
}
```

What should we make static?

What should we make final?

The Person class

```
//code here  
class Person {  
    static  
    private final int birthYear;  
    private String firstName, lastName;  
    private boolean employed;  
    private static int nrOfLegs;  
  
    final  
    //constructor(s)  
    //setters & getters  
    ..  
}
```

Overloading methods

Define in a class methods with the same name and different:

- Number of parameters
- Type of parameters
- Return type

```
class Person {  
    public int calculateAge() {  
        return Date.currentYear() – birthYear;}  
  
    public int calculateAge (int year ) {  
        return year – birthYear;}  
  
    public float calculateAge() {  
        return Date.currentYear() – birthYear;}  
}
```

What is an object?

A specific entity of the type defined by the class.

⇒ Has specific values for the attributes

me is an object of type Person.

```
Person me = new Person();
```

me.firstName = “Mihaela”

me.lastName = “Dinsoreanu”

me.employed = true

me.numberOfLegs = ??

How to create objects?

Using constructors

```
public Person (String fN, String IN, int bY, boolean e)  
{  
    firstName = fN;  
    ...  
}
```

```
Person me = new Person();
```

```
Person me = new Person ("Mihaela", "Dinsoreanu", 1970,  
true);
```

No-arg constructor

- What if we don't declare any constructor?
 - Objects cannot be created
 - A default no-arg constructor is used
 - The default no-arg constructor calls the superclass no-arg constructor

How to use objects?

- Call public methods to query the object (getters)

```
String name = me.getfirstName() + " " + me.getlastName();
```

```
int birthY = me.getbirthYear();
```

```
int age = me.calculateAge();
```

...

How to use objects? (2)

- Call public methods to set attribute values (setters)

```
me.setfirstName(fN);
```

```
me.setlastName(lN);
```

```
me.setbirthYear(bY);
```

```
me.setAge(int age);
```

...

- How are parameters passed?

- By value !

- Primitive type?

- Reference?

```
public class Person {
    final int birthYear;
    String firstName, lastName;
    boolean employed;
    static int nrOfLegs;
    public Person (String fN, String lN, int bY, boolean e)
    {
        firstName = fN;
        lastName = lN;
        employed = e;
        birthYear = bY;
    }

    public void setFirstName(String n)
    {
        firstName = n;
    }

    public void setLastName(String n)
    {
        lastName = n;
    }

    public int calculateAge (int year ) {
        return year - birthYear;}
    public String toString(){
        return "The person "+firstName+ ' '+lastName + " is " +calculateAge(2016)+" years old!";
    }
    public static void display(Person p){
        System.out.println(p);
        p.setFirstName("Vasile");
        System.out.println("inside display "+p);
    }

    public static void main(String args[])
    {
        Person me = new Person ("Mihaela", "Dinsoreanu", 1970, true);
        display(me);
        System.out.println("outside display "+me);
    }
}
```

Output - CMSC (run) ×

run:
The person Mihaela Dinsoreanu is 46 years old!
inside display The person Vasile Dinsoreanu is 46 years old!
outside display The person Vasile Dinsoreanu is 46 years old!
BUILD SUCCESSFUL (total time: 0 seconds)

What about class methods (static)

Can be called with the class name => no instance(object) is necessary

- ⇒ Instance methods can access instance and class attributes directly
- ⇒ Class methods can access ONLY class attributes directly!!

```
class Person {  
    public static int getNrofLegs() {  
        System.out.println(firstName+“ has “+nrOfLegs+” legs.” );  
        return nrOfLegs;  
    }  
}
```

What is inheritance?

- The way to reuse CLASSES to create more specific classes
- Represents the IS-A relationship
- The attributes and methods of the superclass are inherited in the subclass
- FINAL classes cannot be subclassed!
- Examples:
 - Student IS-A Person
 - Dog IS-A Animal
 - Truck IS-A Vehicle
 - Square IS-A Rectangle

Inheritance Example

```
class Student extends Person {  
    ...  
    private String school;  
    private double GPA;  
    ....  
  
    public Student () {};  
    public void setSchool(String newSchool){  
        school = newSchool;  
    }  
...}
```

Or...

```
class Female extends Person {  
}
```

```
class Male extends Person {  
}
```

??

Overriding methods

- Change the inherited code of the method
- The method signature DOESN'T change!
- Can all methods be overridden?
 - FINAL methods cannot!

```
class Student extends Person {
```

```
...
```

```
    public String toString() {
        return "This is student "+ firstName + " "
+lastName;
    }
}
```

What is Composition?

- The way to reuse OBJECTS in order to create more complex objects.
- Represents HAS-A relationship
- Examples:
 - House HAS-A Door
 - Vehicle HAS-A Engine
 - Person HAS-A Heart

```
class Person {  
    private Heart heart;  
    private String firstName, lastName;  
    ...  
    Person (Heart h, String fN, ...)  
}  
  
class Heart {  
    private double pulse;  
    private double weight;  
    ...  
}
```

Abstract classes

- Have at least one abstract method
- May have only abstract methods
- Cannot be instantiated
- How are they useful?

- Example

```
abstract class Person {
```

```
...
```

```
    public abstract String getStatus();
```

```
}
```

Implementing abstract classes

```
class Student extends Person {
```

```
...
```

```
    public String getStatus() {
        return "This is the Student "+firstName+
" "+lastName+" having the GPA "+GPA);
    }
```

```
Person s = new Person();
```

```
Person s = new Student();
```

```
Student s = new Person();
```

```
Student s = new Student();
```

Interfaces

- ONLY abstract methods
- No attributes
- A class can implement several interfaces
- A class can extend just one superclass

More ...

- Inherit a superclass
- Implement an interface
- Can it be defined inside another class?
- Should it always have a name?

Inheriting a superclass

- What is inherited?
 - Attributes
 - Methods
 - Constructors
- What can you do in a subclass?
 - use the inherited fields and methods directly
 - declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
 - declare new fields in the subclass that are not in the superclass.
 - write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it. (NOT FINAL!!!)

[2]

- write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- declare new methods in the subclass that are not in the superclass.
- write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Defining a Method with the Same Signature as a Superclass's Method

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

```
class Base {  
    private int i;  
    public int getI() {return i;}  
    public void setI(int j) {i = j;}  
}
```

```
public class Test extends Base {  
    public static void main(String args[]) {  
        Test t = new Test();  
        t.setI(5);  
        System.out.println(i);  
        System.out.println(t.getI());  
    } }
```

Implement an interface

You have to implement all the methods defined in all the interfaces

```
interface Enjoyable {  
    public boolean enjoy();}
```

```
class Person implements Enjoyable {
```

...

```
        public boolean enjoy() { if (age<80) return  
true;}}
```

```
Enjoyable you = new Enjoyable();
```

```
Enjoyable you = new Person();
```

```
Person you = new Enjoyable();
```

Polymorphism

The possibility to consider an instance as having different types. NOT ANY TYPE!!!!

```
String display(Person p) {  
    System.out.println(p);  
}
```

```
Person me, you;  
me = new Person();  
you = new Student();
```

display(me); => “me@32342323”

display(you); => “This is student

Nested classes

- Static
- Non-static (inner)

```
class Person {  
    static class PersonParser {...}  
    class Address{  
        String street;  
        Street city;..  
    }  
....  
}
```

Local classes

Defined within a method of the outer class

```
class Person {..  
    public void validateZipCode(String zC) {  
        ..  
        class ZipCode {  
            String validZC;  
            ZipCode(String z) {....}  
            String getCode() {return validZC;}  
        }  
        ZipCode z = new ZipCode(zC);  
        z.getCode();  
    }
```

Anonymous classes

- declare and instantiate a class at the same time

```
interface Enjoyable { public String enjoy();}  
class Person {  
    public void beHappy() {  
        Enjoyable happy = new Enjoyable() {  
            public String enjoy() {  
                return "It's a sunny day, I am happy!";  
            }  
            System.out.println(happy.enjoy());  
        }  
    }  
}
```

Lambda expressions

- express instances of single-method classes more compactly
- Use Case for Lambda Expressions

You manage a collection of person objects, want to select the objects that match certain criteria and perform some action on the resulted selection.

Example [Oracle tutorial]

```
public class Person {  
    public enum Sex { MALE, FEMALE }  
    String name;  
    LocalDate birthday;  
    Sex gender;  
    String emailAddress;  
    public int getAge() { // ... }  
    public void printPerson() { // ... } }
```

List<Person>

Create methods that search for members that match one characteristic

```
public static void printPersonsOlderThan(List<Person>  
roster, int age)  
{  
    for (Person p : roster) {  
        if (p.getAge() >= age) {  
            p.printPerson(); } } }
```

Create more generalized search methods

```
public static void printPersonsWithinAgeRange(  
List<Person> roster, int low, int high) {  
    for (Person p : roster) {  
        if (low <= p.getAge() && p.getAge() < high) {  
            p.printPerson();  
        }  
    }  
}
```

Specify search criteria code in a local class

```
public static void printPersons(List<Person> roster,  
CheckPerson tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson(); } } }
```

```
interface CheckPerson { boolean test(Person p); }
```

```
class CheckPersonEligibleForSelectiveService
implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25; } }

//method call
printPersons(roster, new
CheckPersonEligibleForSelectiveService());
```

Specify search criteria code in an Anonymous Class

```
printPersons( roster, new CheckPerson() {  
    public boolean test(Person p) {  
        return p.getGender() ==  
Person.Sex.MALE  
            && p.getAge() >= 18 &&  
p.getAge() <= 25; } } );
```

Specify search criteria code with a Lambda Expression

```
printPersons( roster, (Person p) -> p.getGender() ==
```

```
Person.Sex.MALE && p.getAge() >= 18 && p.getAge()
```

```
<= 25 );
```

Use standard functional interfaces with Lambda Expressions

```
interface Predicate<T> { boolean test(T t); }
```

```
interface Predicate<Person> { boolean test(Person t); }
```

```
public static void printPersonsWithPredicate( List<Person>
roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson(); } } }
```

When to use nested classes

- Local class: if you need to create more than one instance of a class, access its constructor, or introduce a new, named type
- Anonymous class: if you need to declare fields or additional methods.
- Lambda expression:
 - if you are encapsulating a single unit of behavior that you want to pass to other code.
 - if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).

Wrap-up

- Our objective is to develop GOOD software design
- EACH PROBLEM HAS SEVERAL SOLUTIONS!
 - Design
 - Technology
 - Code
- Basic OO concepts overview

SOFTWARE DESIGN

Architectural patterns

Content

- Architectural Patterns
 - Layers
 - Client-Server
 - Event-driven
 - Broker
 - Mediator
 - MVC (and variants)
 - Service-based
 - SOA
 - Microservices
 - Space-based (Cloud)

References

- Mark Richards, Software Architecture Patterns, O'Reilly, 2015 [SAP]
- Mark Richards, Microservices vs. Service-Oriented Architecture O'Reilly, 2016
- David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Ed.[Patterson]
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley [Taylor]
- F. Buschmann et. al, PATTERN-ORIENTED SOFTWARE ARCHITECTURE: A System of Patterns, Wiley&Sons, 2001.[POSA]
- Martin Fowler et. al, Patterns of Enterprise Application Architecture, Addison Wesley, 2003 [Fowler]
- O. Shelest, MVC, MVP, MVVM design patterns,
http://www.codeproject.com/KB/architecture/MVC_MVP_MVVM_design.aspx
- <http://www.tinmegali.com/en/model-view-presenter-mvp-in-android-part-2/>
- Univ. of Aarhus Course Materials
- Univ. of Utrecht Course Materials

Architectural Patterns

- Definition
 - An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- [POSA, vol.1]
- Different books present different taxonomies of patterns

Structural AP

- From Mud to Structure
- Direct mapping Requirements -> Architecture?
- Problems: non-functional qualities like portability, stability, maintainability, understandability...

Layers

- Definition
 - The **Layers** architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.
- Example
 - Networking: OSI 7-Layer Model

Layers AP

- Addressed problems
 - High-level and low-level operations
 - High-level operations rely on low-level ones

⇒ Vertical structuring
- Several operations on the same level on abstraction but highly independent

⇒ Horizontal structuring

Constraints

- Late source code **changes should not ripple through the system.**
- **Interfaces should be stable** and may even be prescribed by a standards body.
- **Parts of the system should be exchangeable.**
- It may be necessary to **build other systems** at a later date **with the same low-level issues** as the system you are currently designing.

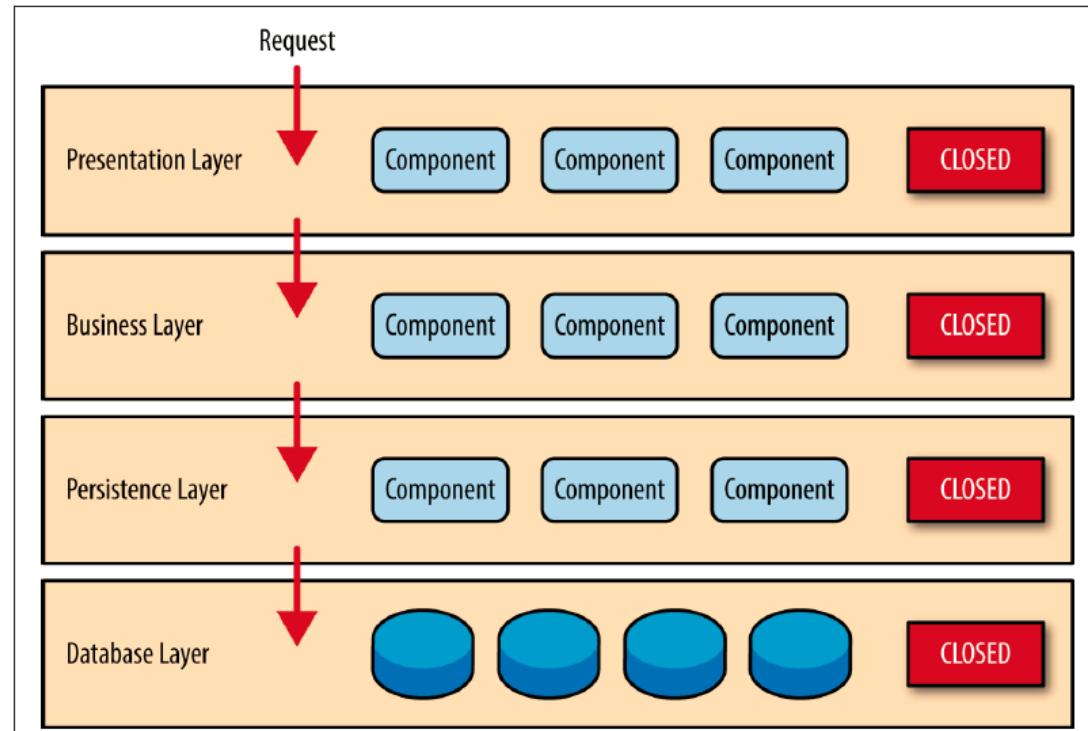
Constraints [2]

- **Similar responsibilities should be grouped** to help understandability and maintainability.
- There is **no 'standard' component granularity**.
- Complex components need further **decomposition**.
- The system will be built by a team of programmers, and work has to be subdivided along clear boundaries.

Key concepts

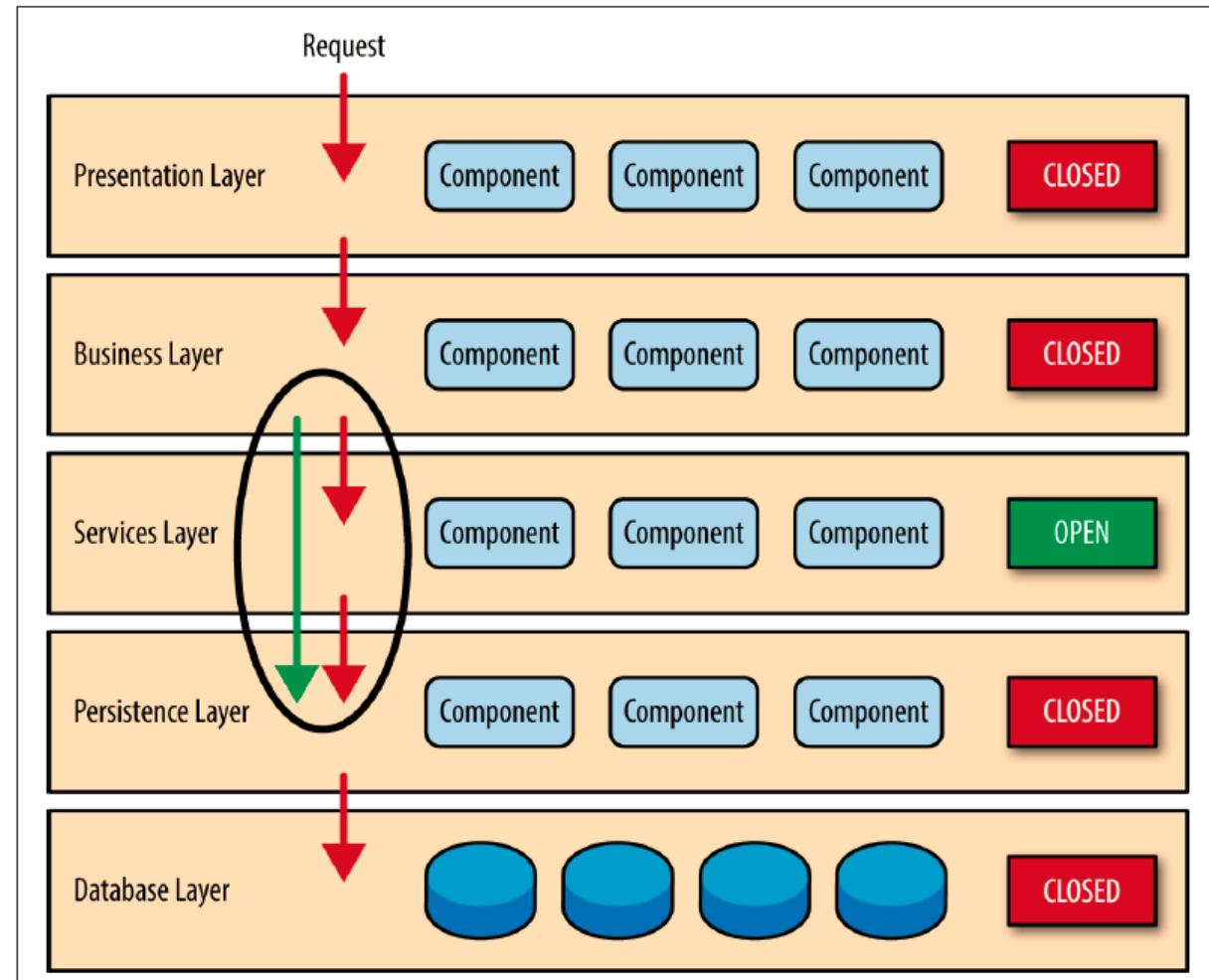
- Closed layers
- Layers of isolation:

changes made in one layer of the architecture don't impact components in other layers



Variants

- Relaxed layered system



Example

Customer Screen:

- Java Server Faces
- ASP (MS)

Customer Delegate:

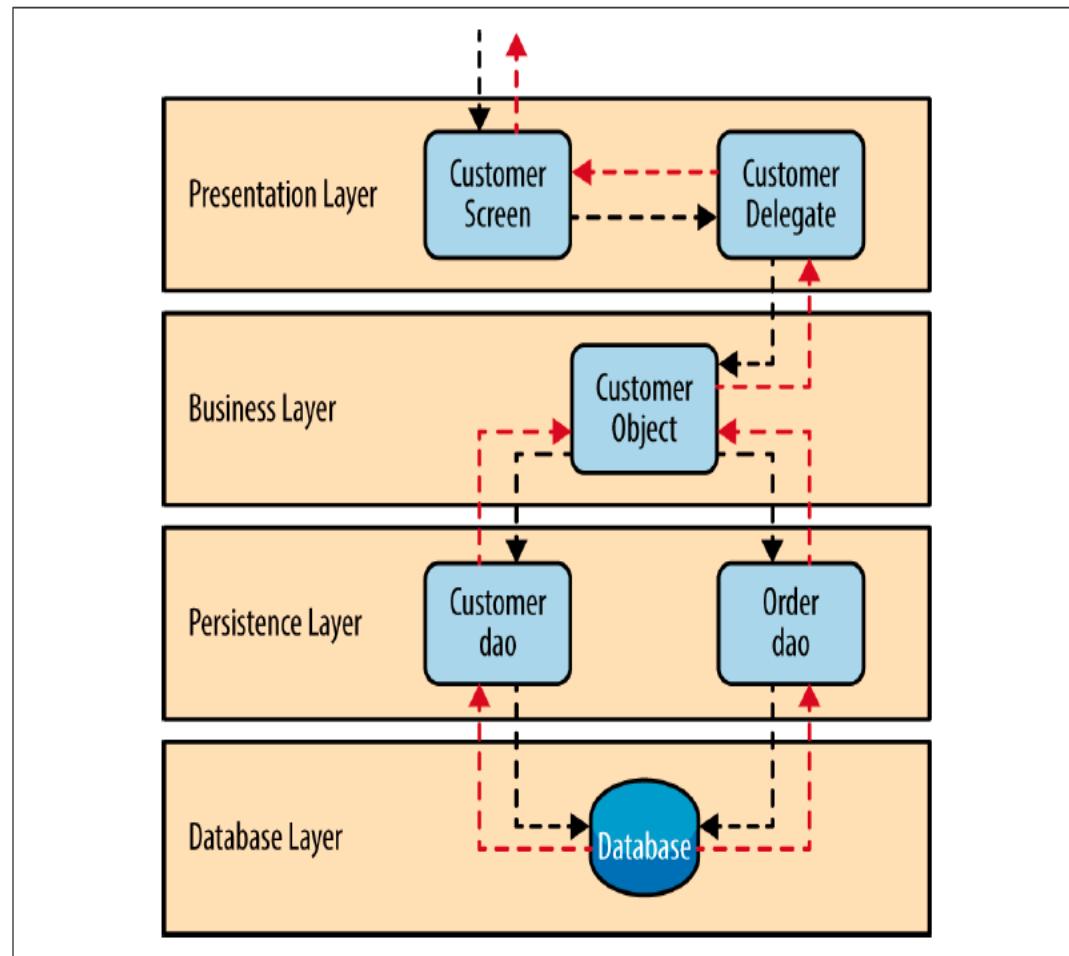
- managed bean component

Customer Object:

- Local Spring Bean
- Remote EJB3 component
- C# (MS)

DAOs:

- POJOs
- MyBatis XML Mapper files
- Objects encapsulating raw JDBC calls or Hibernate queries
- ADO (MS)



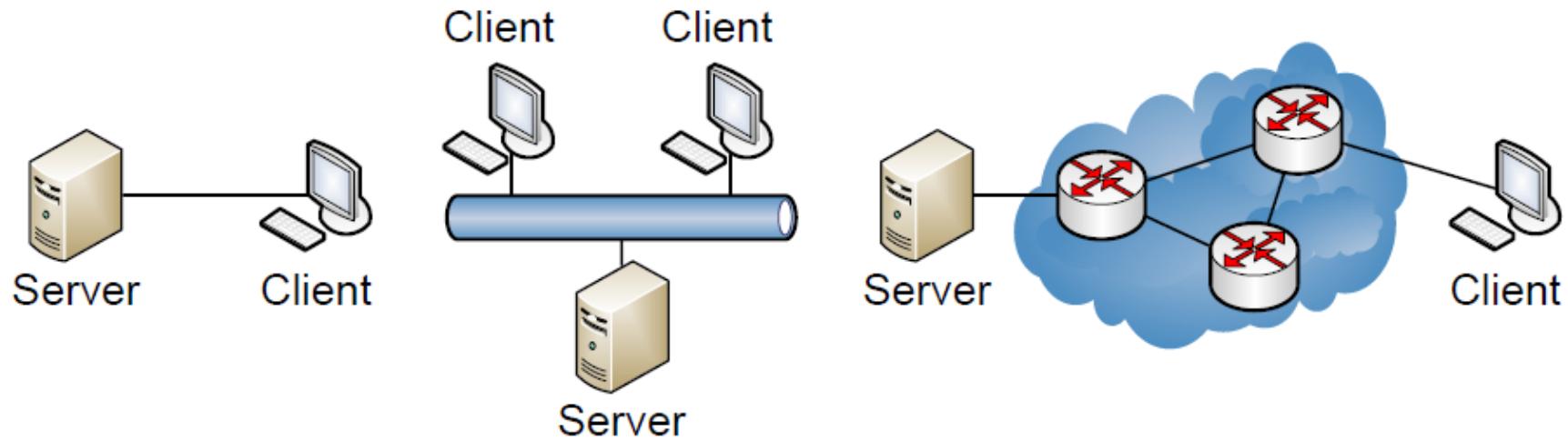
Considerations

- Sink-hole anti-pattern
 - requests flow through multiple layers of the architecture as simple pass-through processing with little or no logic performed within each layer.

<i>Non functional req.</i>	<i>Rating</i>
Overall agility	Low
Ease of deployment	Low
Testability	High
Performance	Low
Scalability	Low
Ease of development	High

Client-Server

- Components are clients and servers
- Servers do not know number or identities of clients
- Clients know server's identity



Types of servers and clients

- Servers

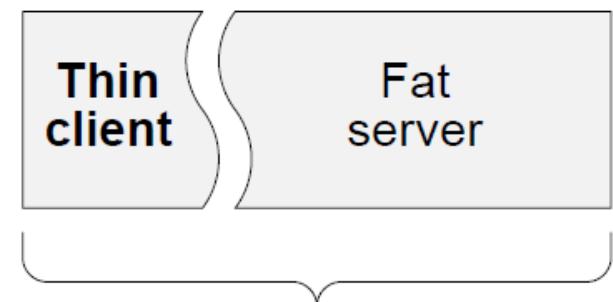
- Iterative (UDP-based servers, ex. Internet services like echo, daytime)
- Concurrent (TCP-based servers, ex. HTTP, telnet, and FTP)
 - Thread-per-client
 - Thread pool

- Clients

- Thin
- Fat



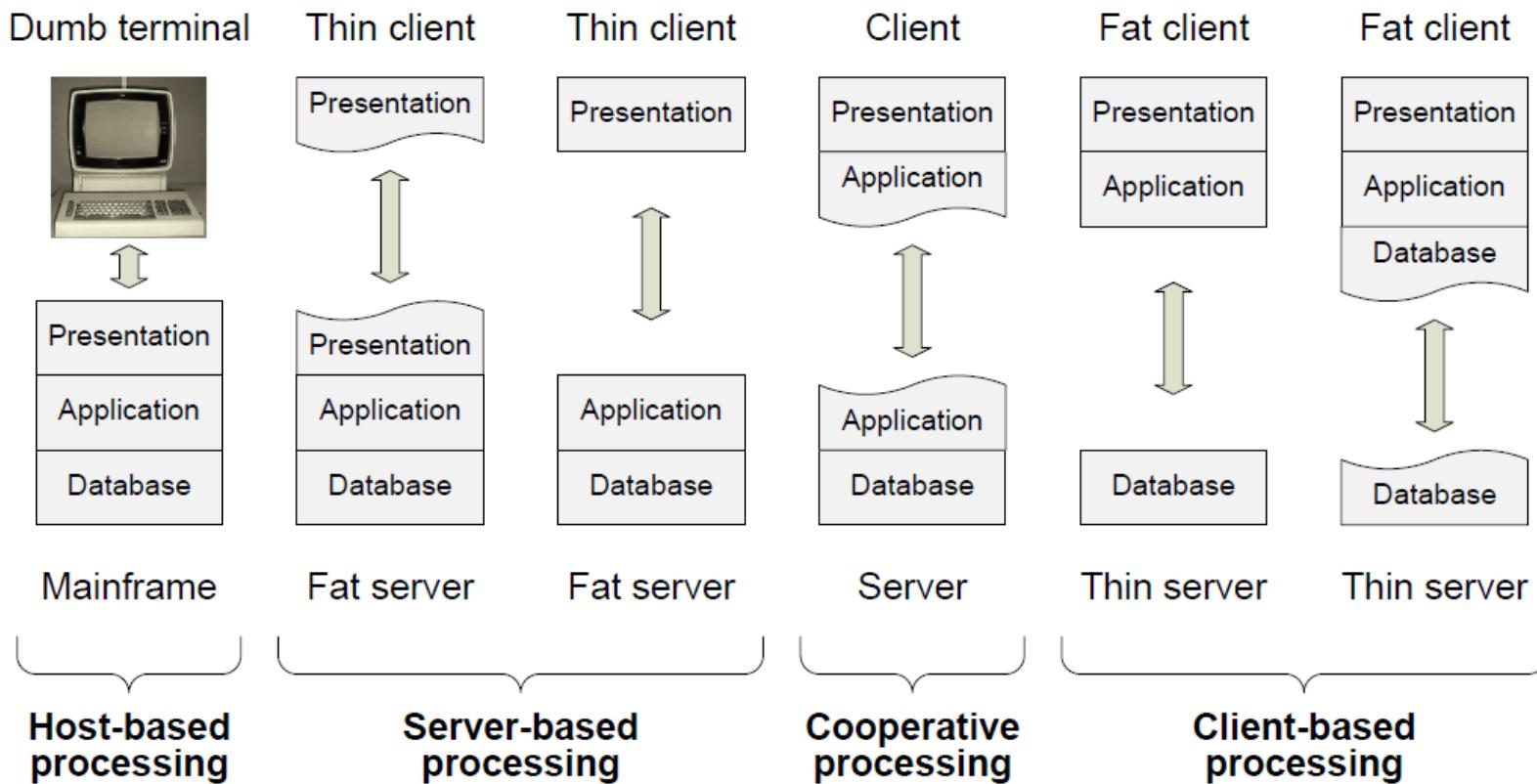
Functionality &
processing load



Functionality &
processing load

Layers vs. tiers

- Layers – logical (ex. presentation, business logic, data access)
- Tiers – physical

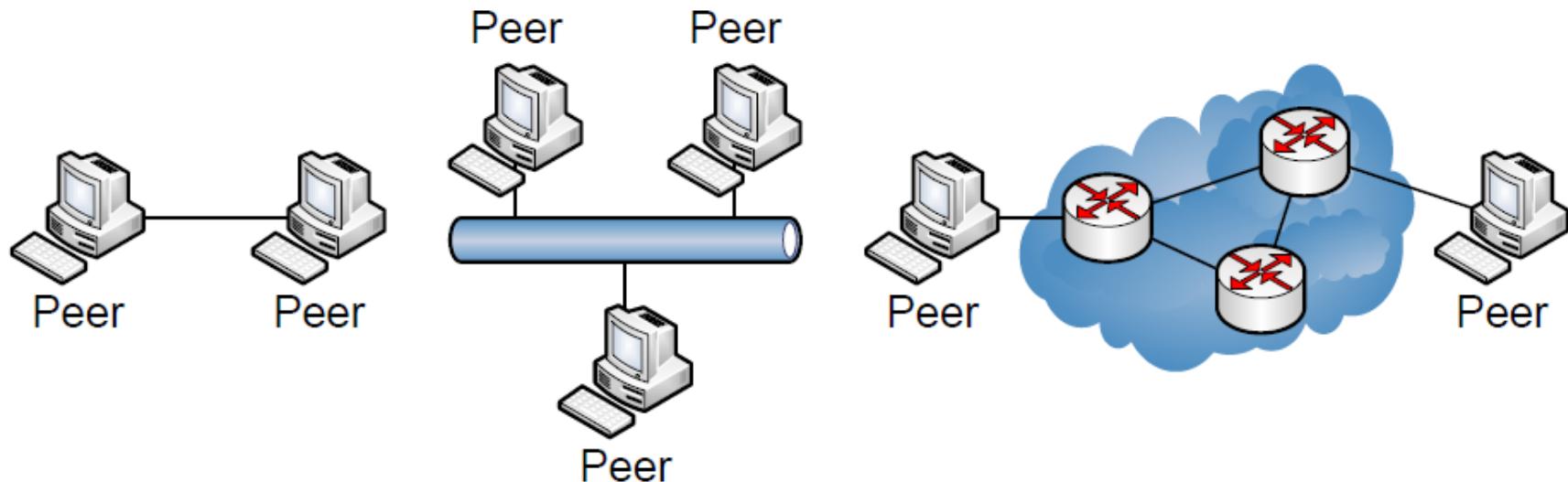


Considerations

Quality Attribute	Issues
Availability	Servers in each tier can be replicated, so that if one fails, others remain available. Overall the application will provide a lower quality of service until the failed server is restored.
Failure handling	If a client is communicating with a server that fails, most web and application servers implement transparent failover. This means a client request is, without its knowledge, redirected to a live replica server that can satisfy the request.
Modifiability	Separation of concerns enhances modifiability, as the presentation, business and data management logic are all clearly encapsulated. Each can have its internal logic modified in many cases without changes rippling into other tiers.
Performance	This architecture has proven high performance. Key issues to consider are the amount of concurrent threads supported in each server, the speed of connections between tiers and the amount of data that is transferred. As always with distributed systems, it makes sense to minimize the calls needed between tiers to fulfill each request.
Scalability	As servers in each tier can be replicated, and multiple server instances run on the same or different servers, the architecture scales out and up well. In practice, the data management tier often becomes a bottleneck on the capacity of a system.

Peer-to-Peer

- State and behavior are distributed among peers which can act as either clients or servers.
- Peers: independent components, having their own state and control thread.
- Connectors: Network protocols, often custom.
- Data Elements: Network messages



Peer-to-Peer [2]

- Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically
- Supports decentralized computing with flow of control and resources distributed among peers.
- Highly robust in the face of failure of any given node.
- Scalable in terms of access to resources and computing power.
- Drawbacks:
 - Poor security
 - Nodes with shared resources have poor performance

Event-driven (distributed) architectures

- (Distributed) asynchronous architecture pattern
- Highly scalable applications
- Highly adaptable by integrating highly decoupled, single-purpose event processing components that asynchronously receive and process events
- 2 main topologies
 - Broker
 - Mediator

Broker

- Definition
 - The Broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication.
- Example
 - SOA

Broker

- Context
 - The environment is a distributed and possibly heterogeneous system with independent, cooperating components.
- Problems
 - System = set of decoupled and inter-operating components
 - Inter-process communication
 - Services for adding, removing, exchanging, activating and locating components are also needed.

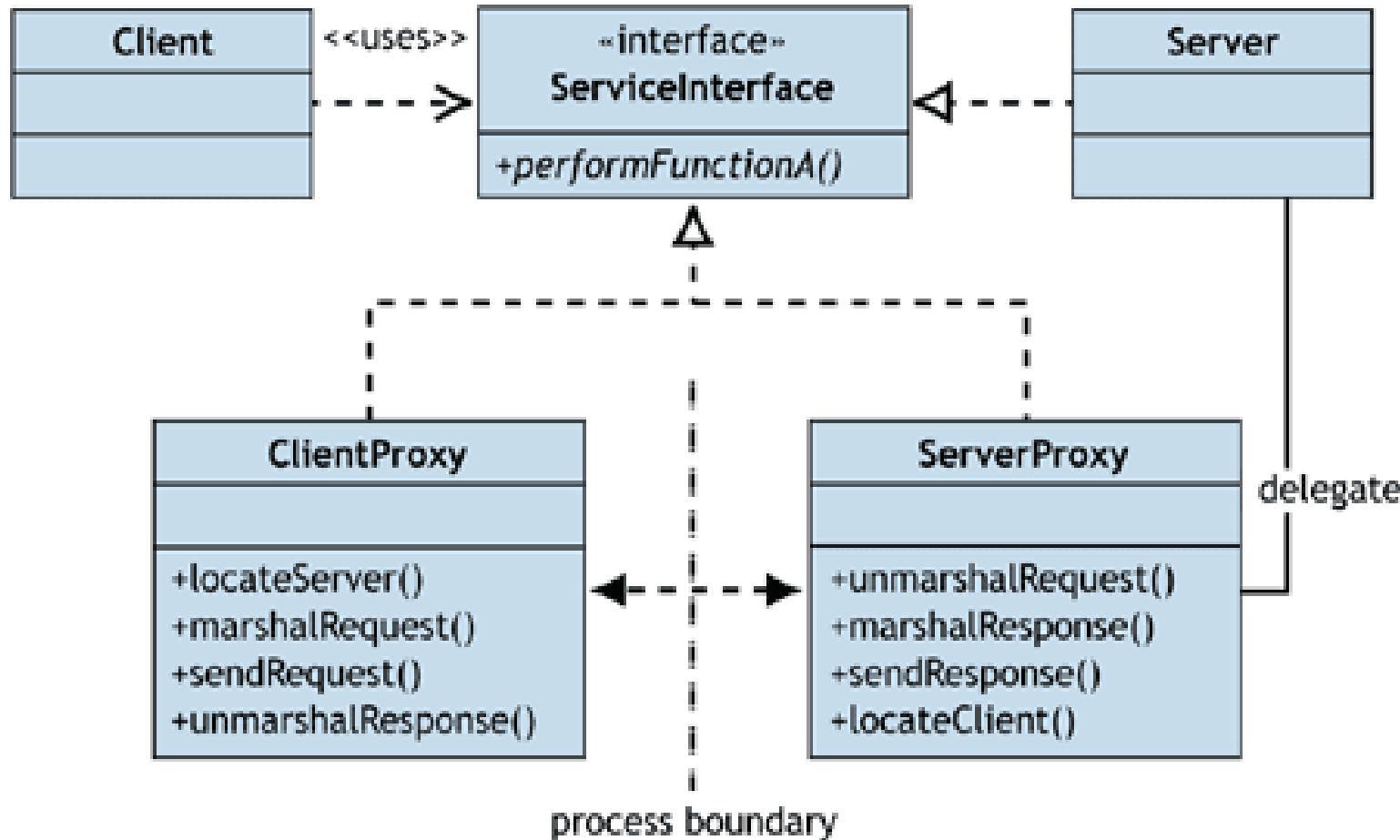
Forces

- Components should be able to access services provided by others through remote, location-transparent service invocations.
- You need to exchange, add, or remove components at run-time.
- The architecture should hide system- and implementation-specific details from the users of components and services.

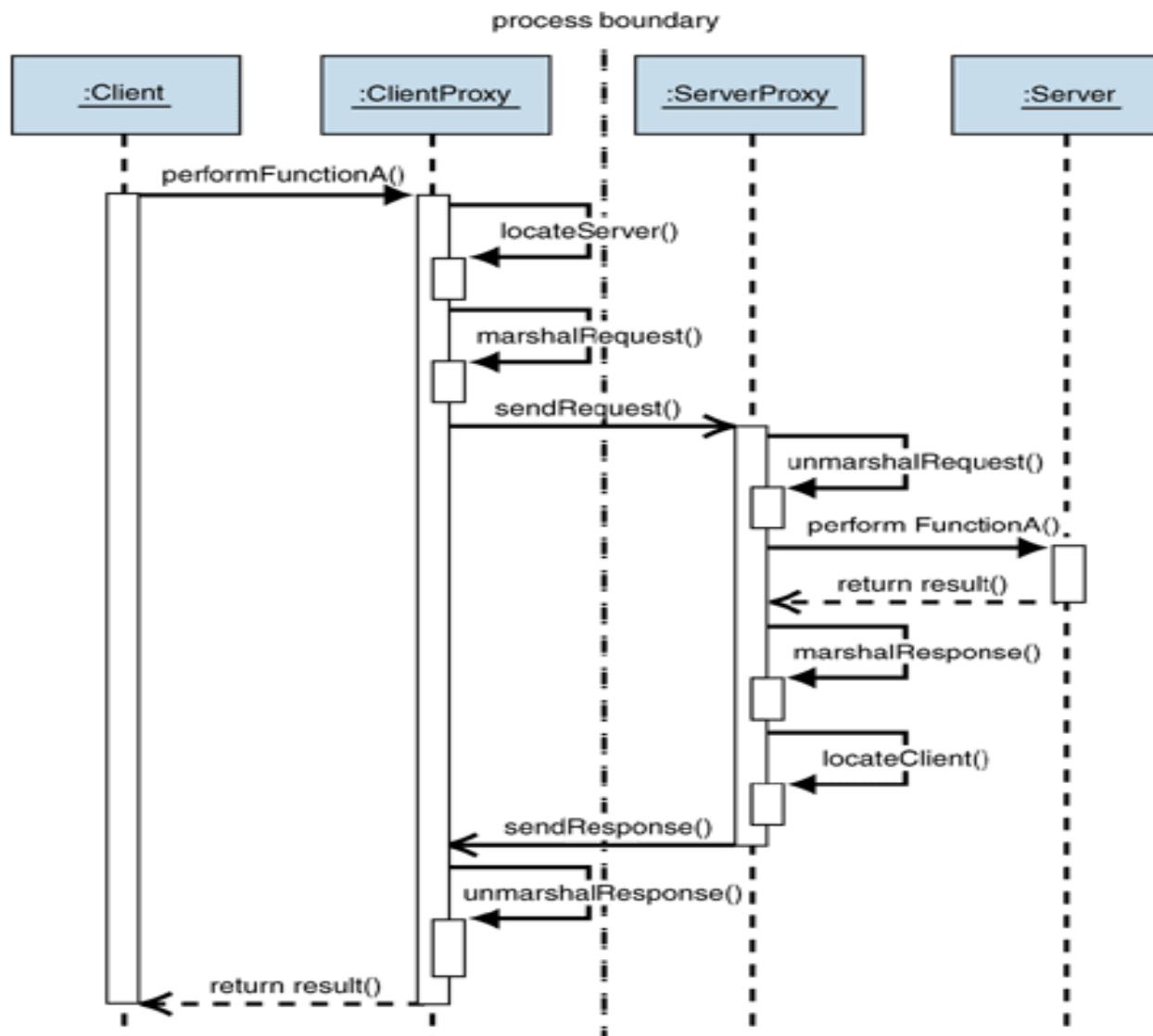
Non-distributed system [MSDN]



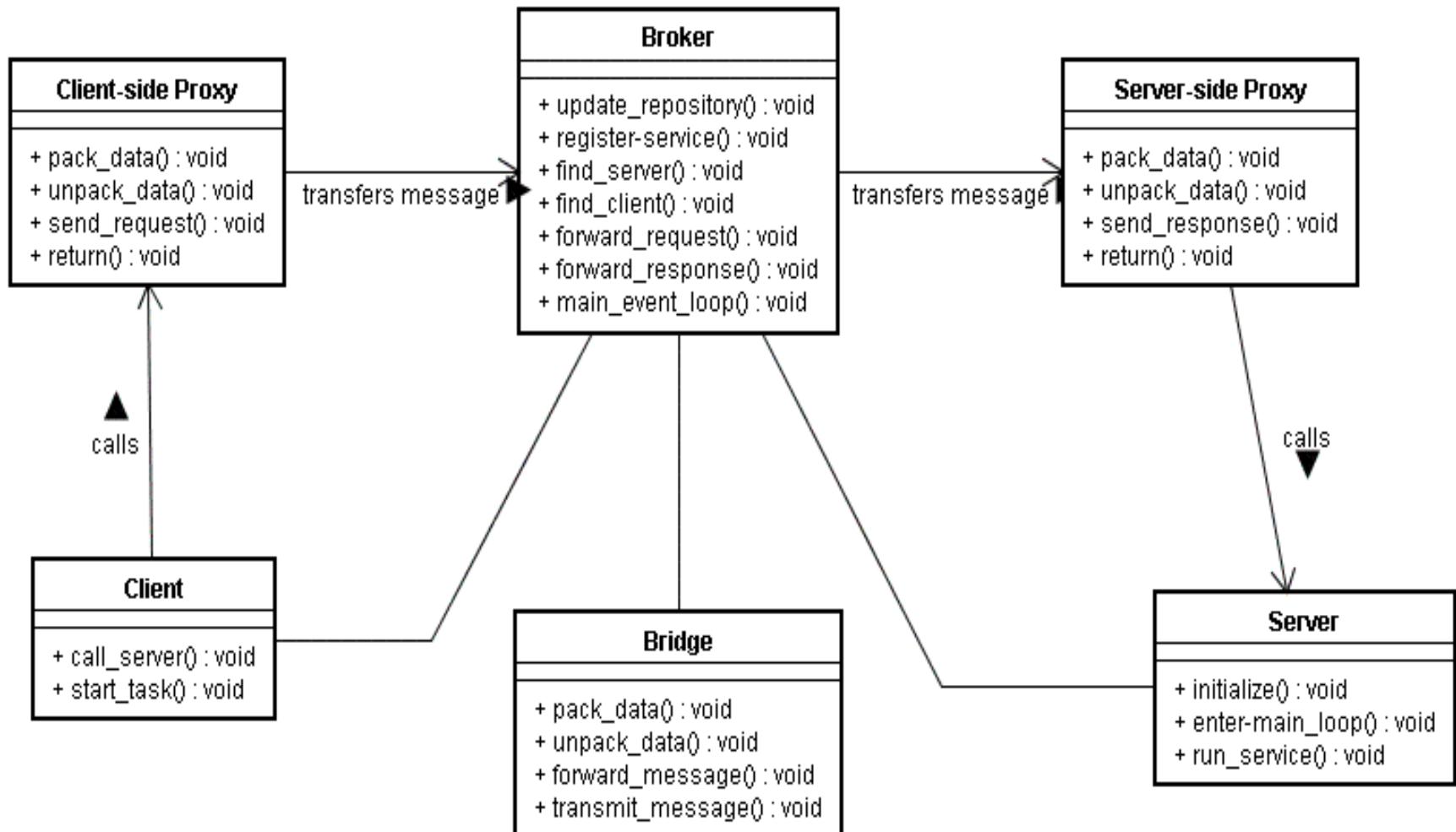
Distributed System [MSDN]



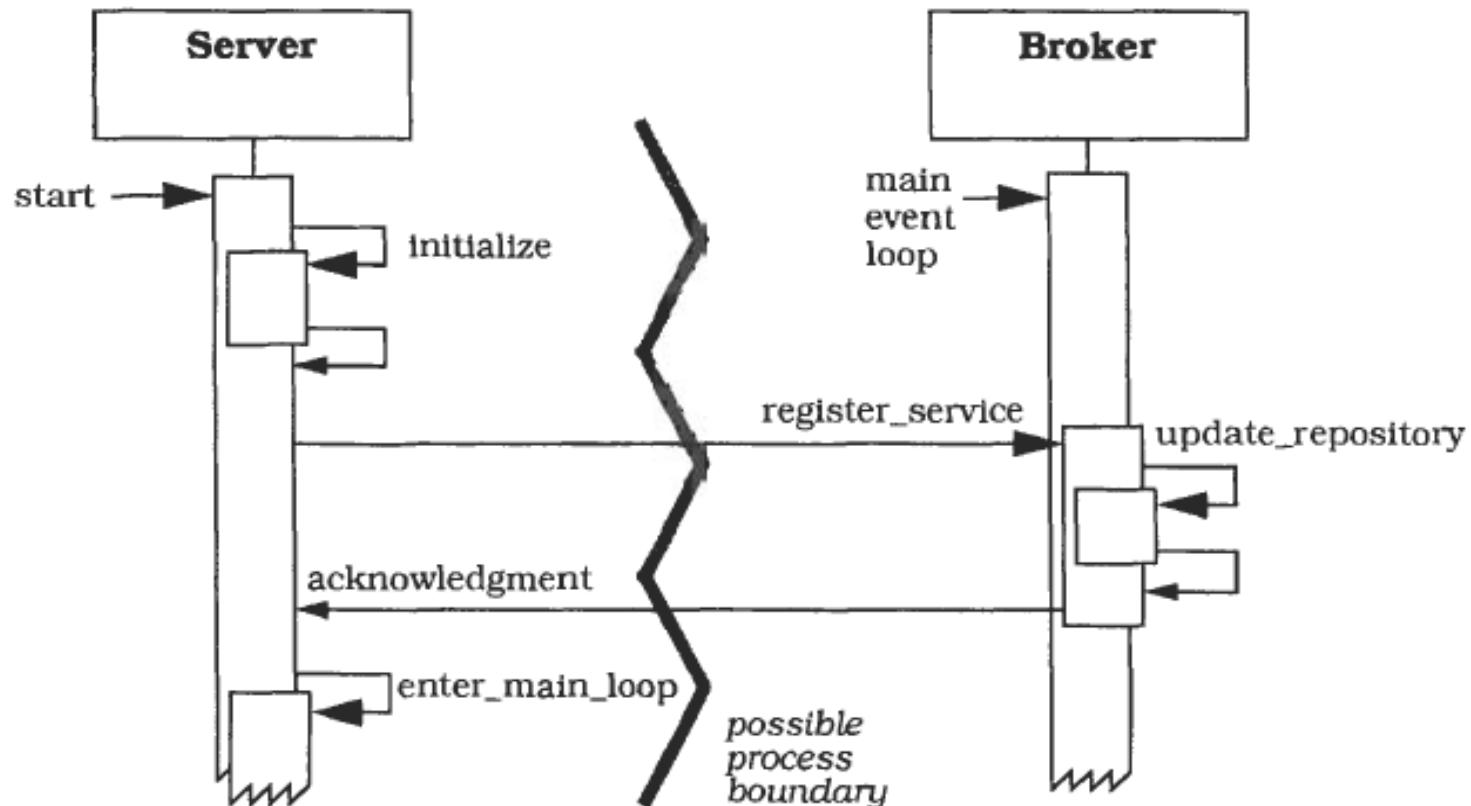
Distributed System – Problems?



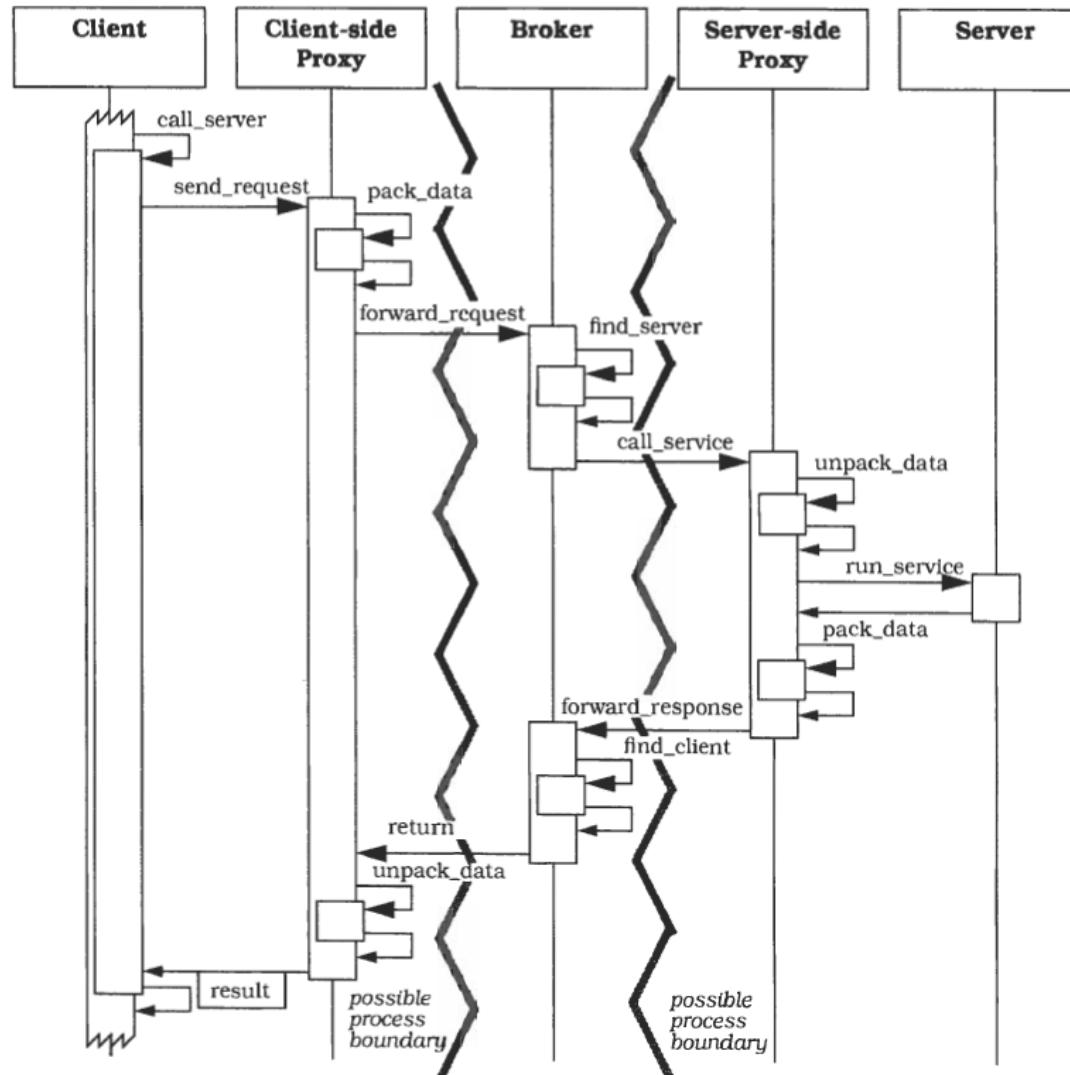
Solution



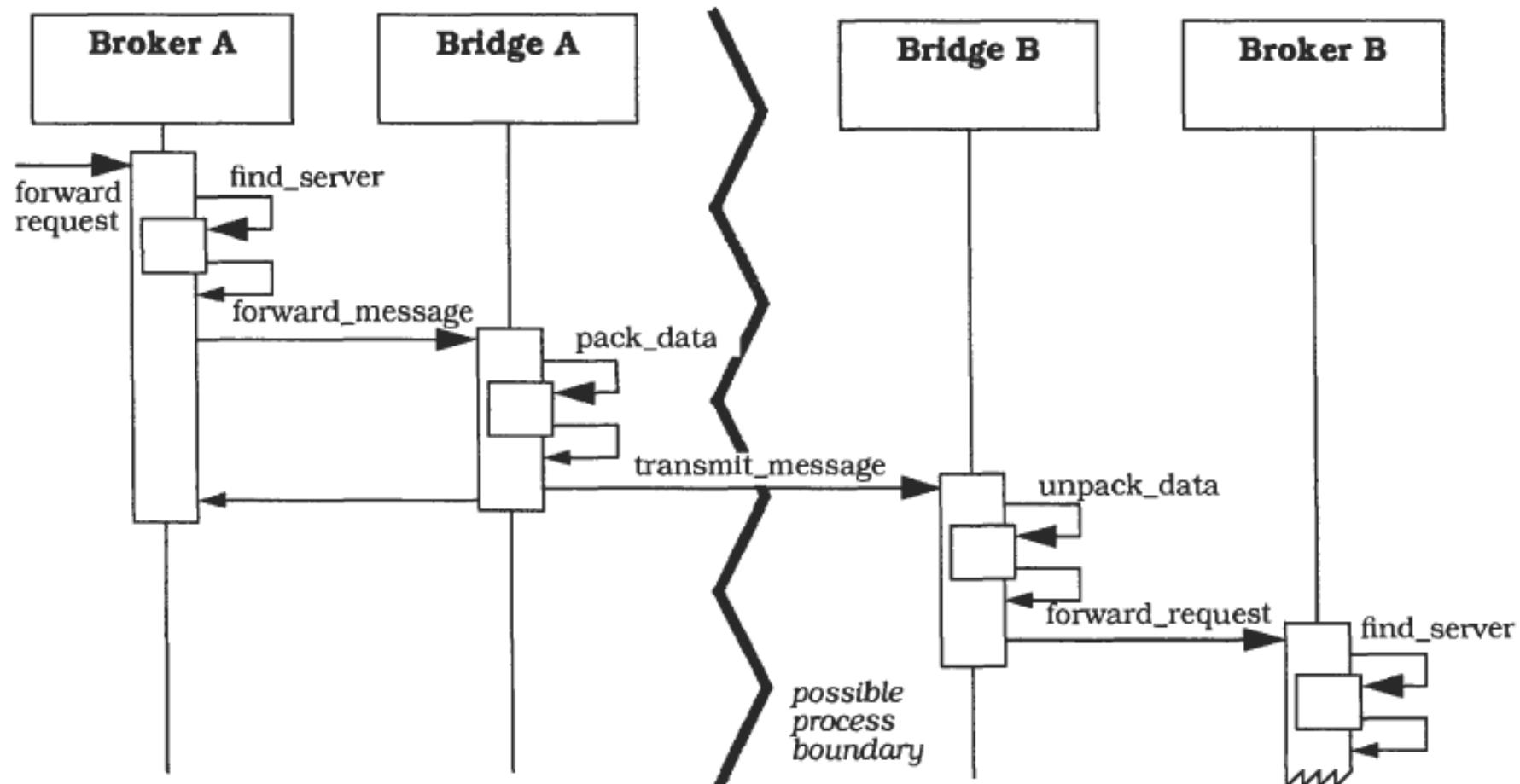
Scenario I



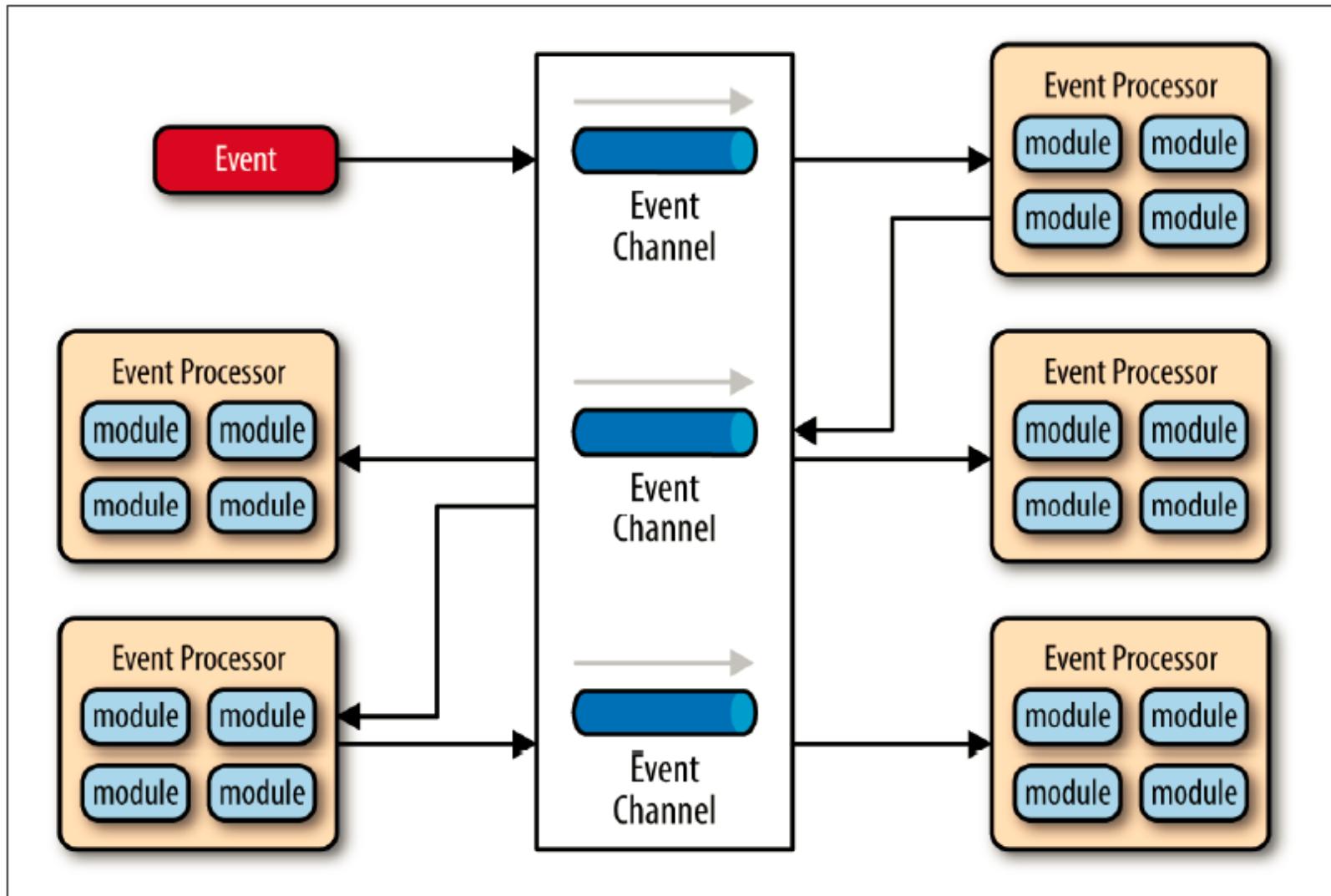
Scenario II



Scenario III



Event-driven perspective



Consequences

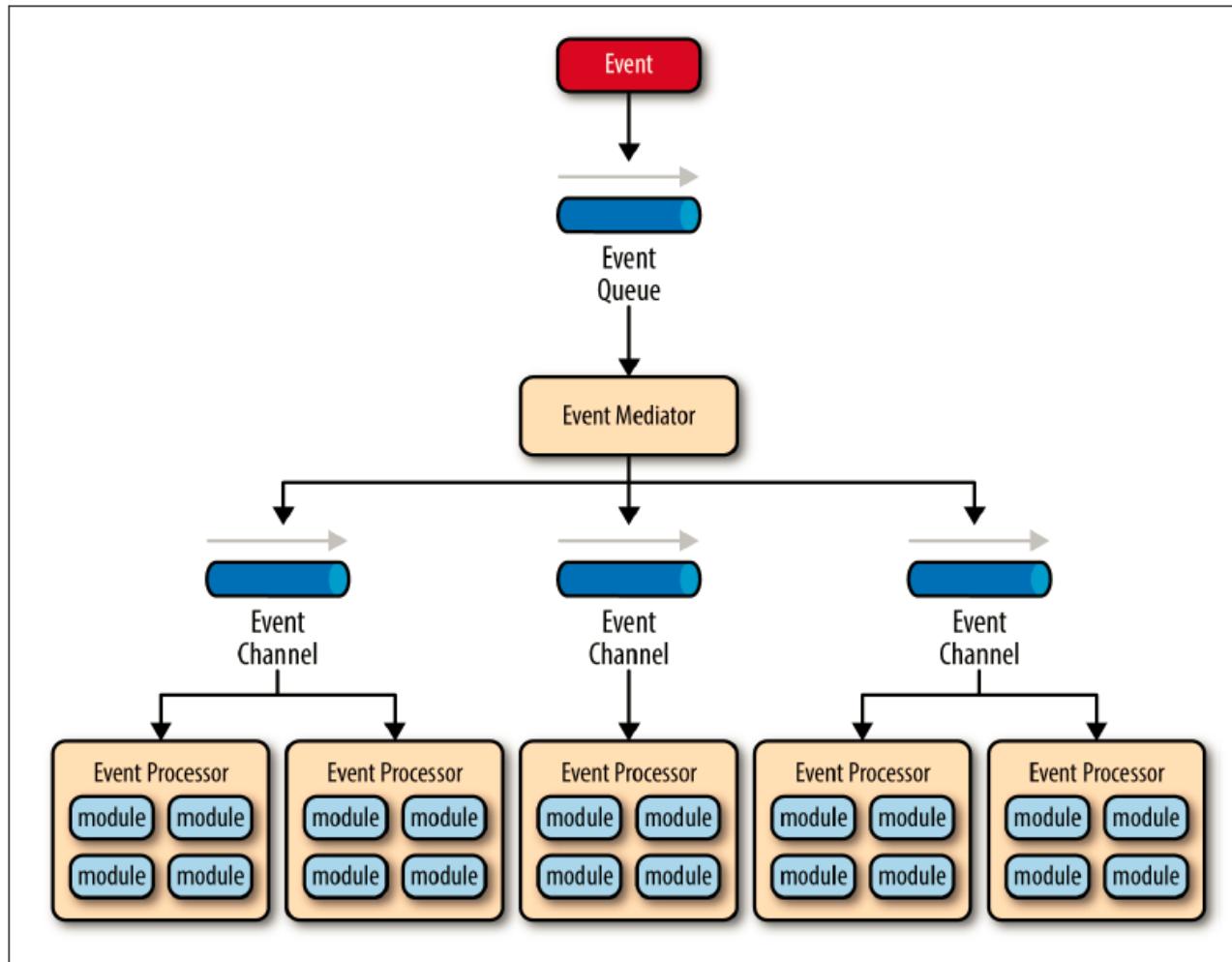
- Benefits
 - Location transparency
 - Changeability and extensibility of components
 - Portability of a Broker System
 - Interoperability between Broker Systems
 - Reusability
- Liabilities
 - Reliability – remote process availability, lack of responsiveness, broker reconnection
 - Lack of atomic transactions for a single business process

Considerations

Quality Attribute	Issues
Availability	To build high availability architectures, brokers must be replicated. This is typically supported using similar mechanisms to messaging and publish-subscribe server clustering.
Failure handling	As brokers have typed input ports, they validate and discard any messages that are sent in the wrong format. With replicated brokers, senders can fail over to a live broker should one of the replicas fail.
Modifiability	Brokers separate the transformation and message routing logic from the senders and receivers. This enhances modifiability, as changes to transformation and routing logic can be made without affecting senders or receivers.
Performance	Brokers can potentially become a bottleneck, especially if they must service high message volumes and execute complex transformation logic. Their throughput is typically lower than simple messaging with reliable delivery.
Scalability	Clustering broker instances makes it possible to construct systems scale to handle high request loads.

Mediator

- for events that have multiple steps and require some level of orchestration to process the event

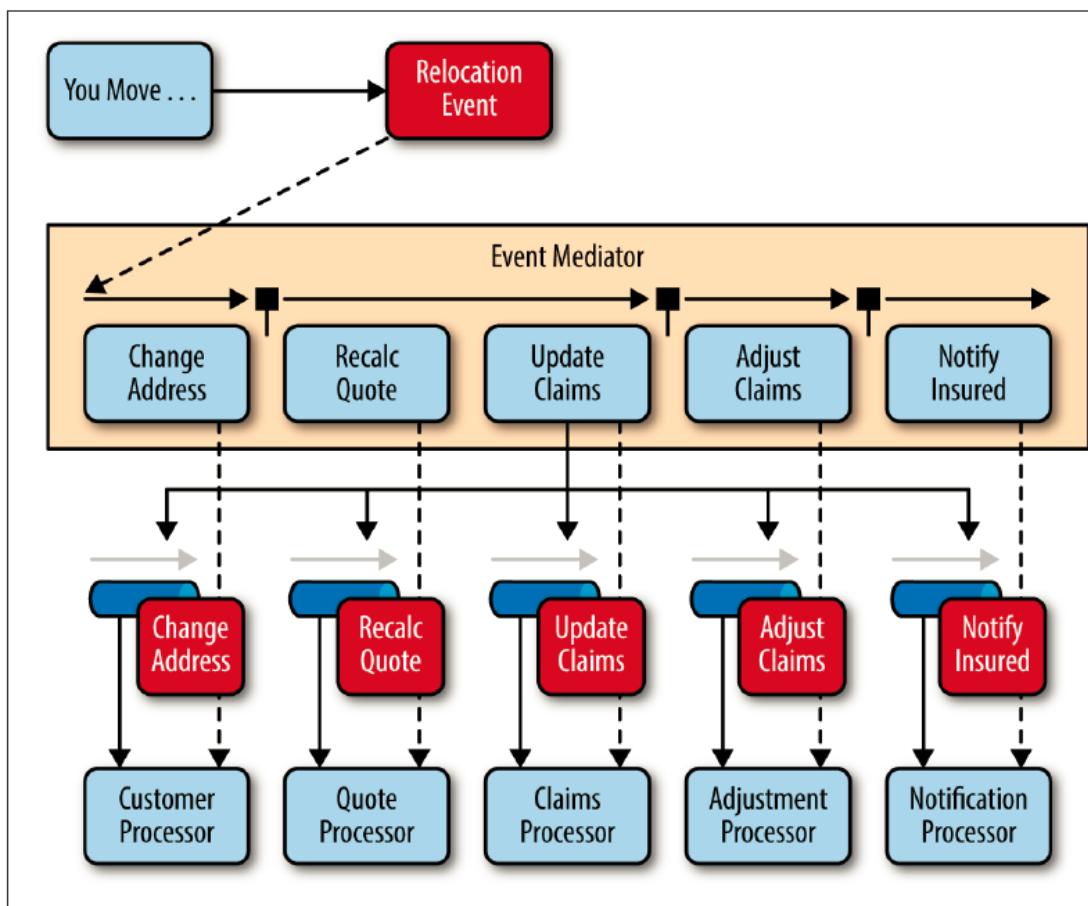


Components

- Event queue (hosts initial events)
 - Message queue
 - Web service endpoint
- Events
 - Initial
 - Processing
- Event channel (passes processing events)
 - Message queue
 - Message topic
- Event processor
 - self-contained, independent, highly decoupled business logic components

• Event mediator

- open source integration hubs such as Spring Integration, Apache Camel, or Mule ESB
- BPEL (business process execution language) coupled with a BPEL engine (ex. Apache ODE)
- business process manager (BPM) (ex. jBPM)



Interactive Systems

- Model-View-Controller
 - The Model-View-Controller architectural pattern (MVC) divides an interactive application into three components. The **Model** contains the core functionality and data. **Views** display information to the user. **Controllers** handle user input. Views and controllers together comprise the user interface.
- Many existing frameworks: JavaServer Faces (JSF), Struts, CakePHP, Django, Ruby on Rails, ...

MVC

- Example
 - Any Information System
- Context
 - Interactive applications with a flexible human-computer interface.
- Problem
 - User interfaces are especially prone to change requests.
 - Different users place conflicting requirements on the user interface.

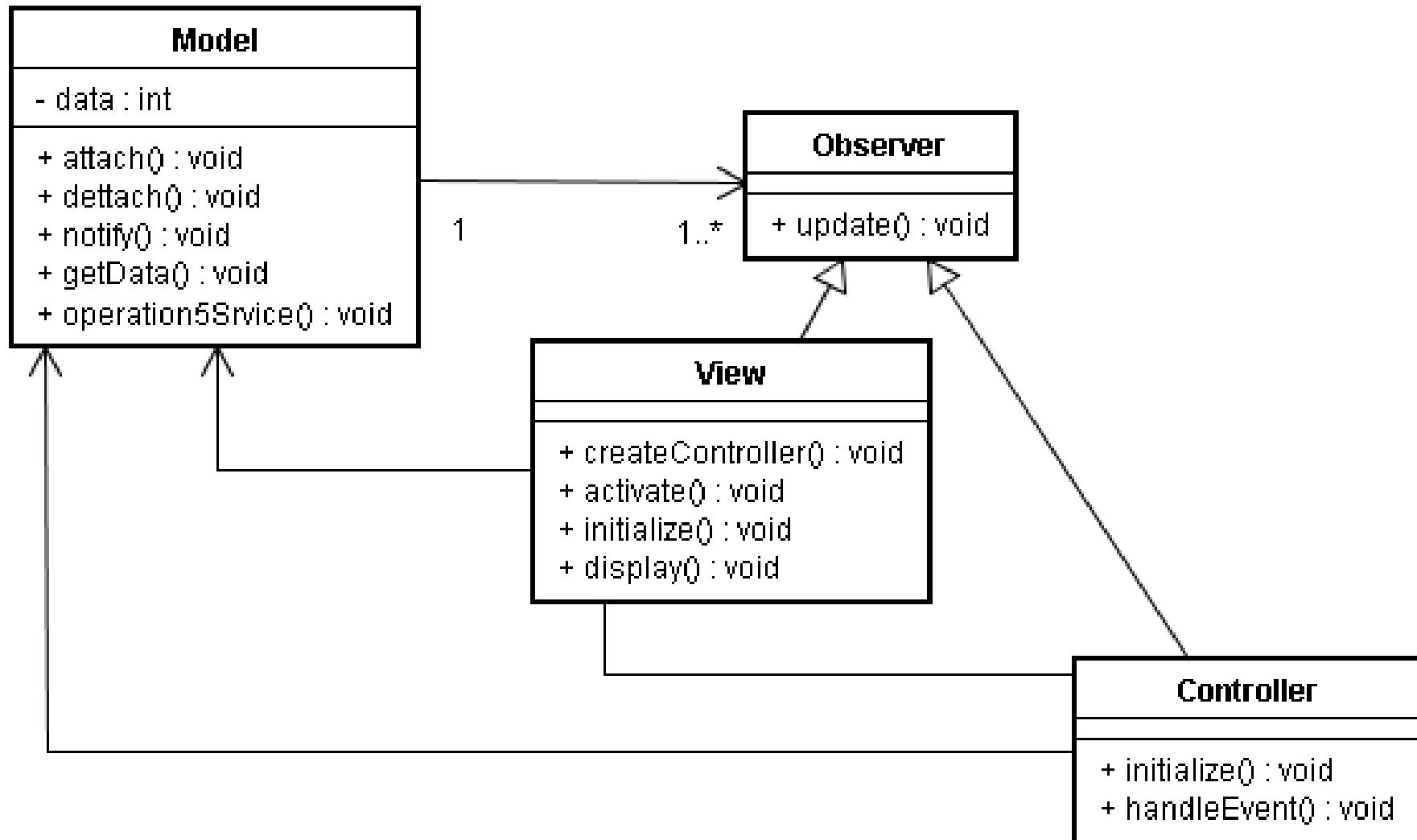
MVC

- Constraints
 - The same information is presented differently in different windows, for example, in a bar or pie chart.
 - The display and behavior of the application must reflect data manipulations immediately.
 - Changes to the user interface should be easy, and even possible at run-time.
 - Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

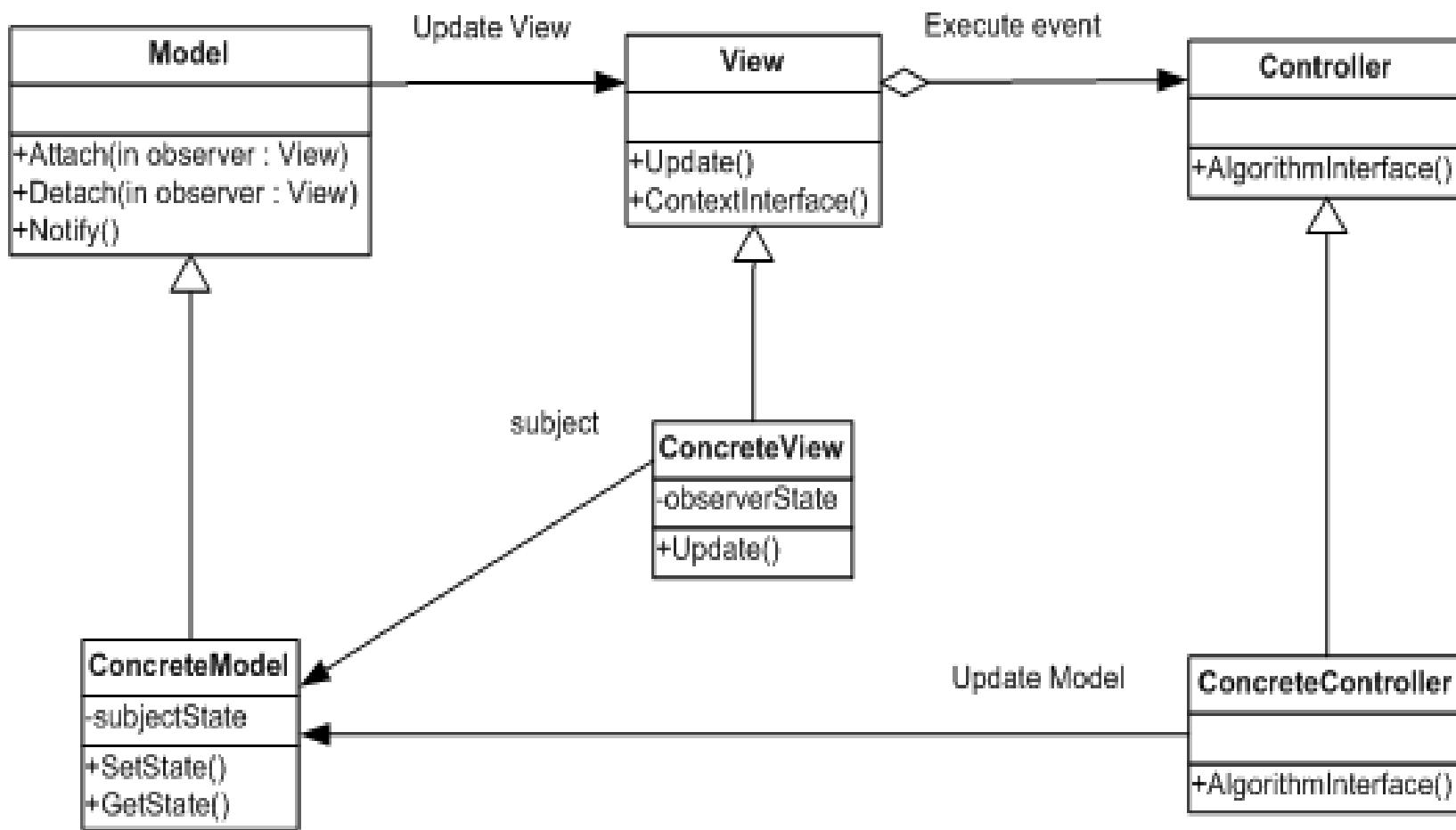
MVC

- Solution
 - 3 areas: handle input, processing, output
 - The **Model** component encapsulates core data and functionality (processing).
 - **View** components display information to the user. A view obtains the data from the model (output).
 - Each view has an associated **controller** component. Controllers handle input.

MVC Structure



MVC more detailed



The Model

- The model encapsulates and manipulates the domain data to be rendered
- The model has no idea how to display the information it has nor does it interact with the user or receive any user input
- The model encapsulates the functionality necessary to manipulate, obtain, and deliver that data to others, *independent* of any user interface or any user input device

The View

- A View is a specific visual rendering of the information contained in the model.
- A view may be graphical or text based
- Multiple views may present multiple renditions of the data in the model
- Each view is a *dependent* of a model
- When the model changes, all dependent views are updated

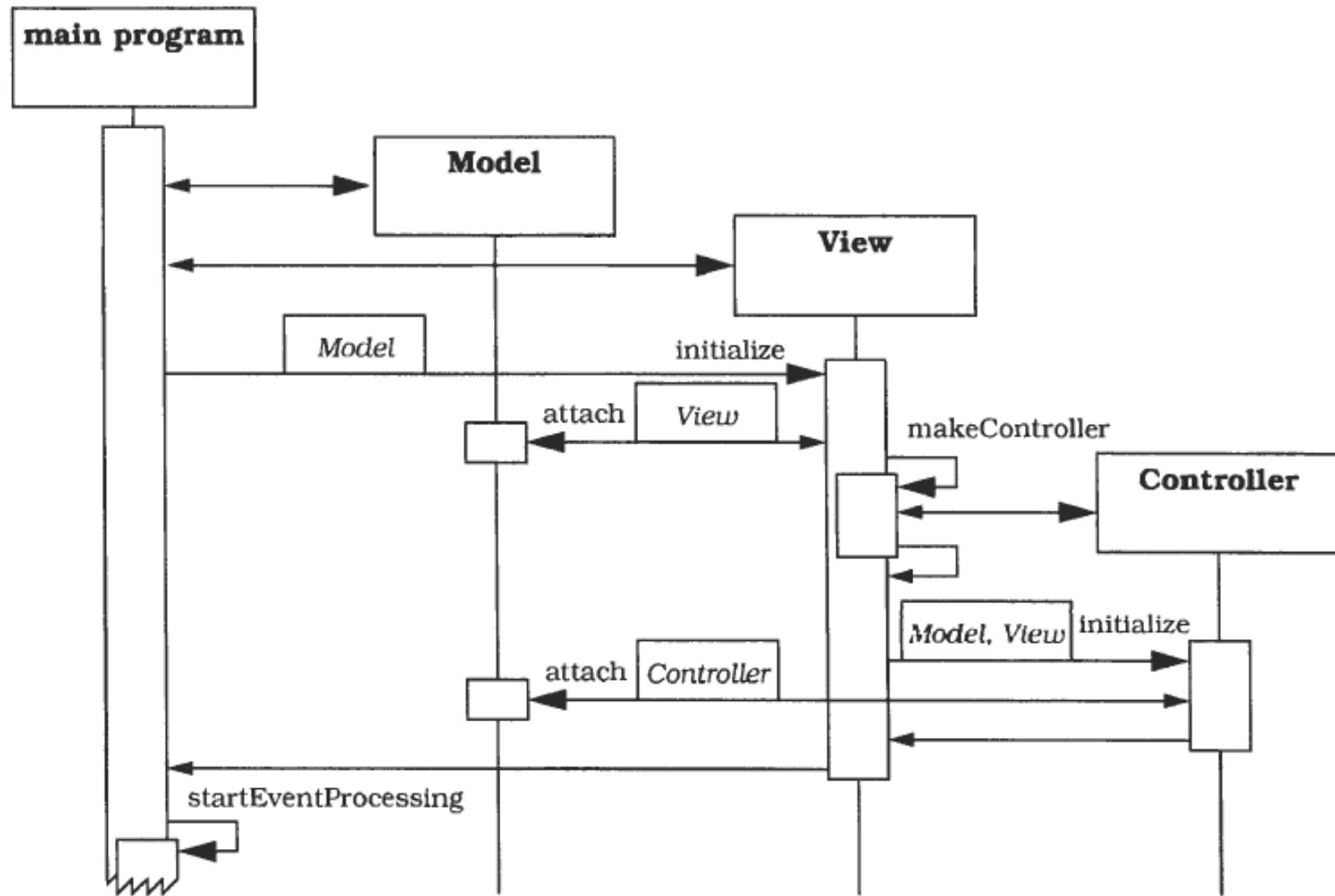
The Controller

- Controllers handle user input. They “listen” for user direction, and *handle* requests using the model and views
- Controllers often watch mouse events and keyboard events
- The controller allows the decoupling of the model and its views, allowing views to simply render data and models to simply encapsulate data
- Controllers are “paired up” with collections of view types, so that a “pie graph” view would be associated with its own “pie graph” controller, etc.
- **The *behavior* of the controller is dependent upon the *state* of the model**

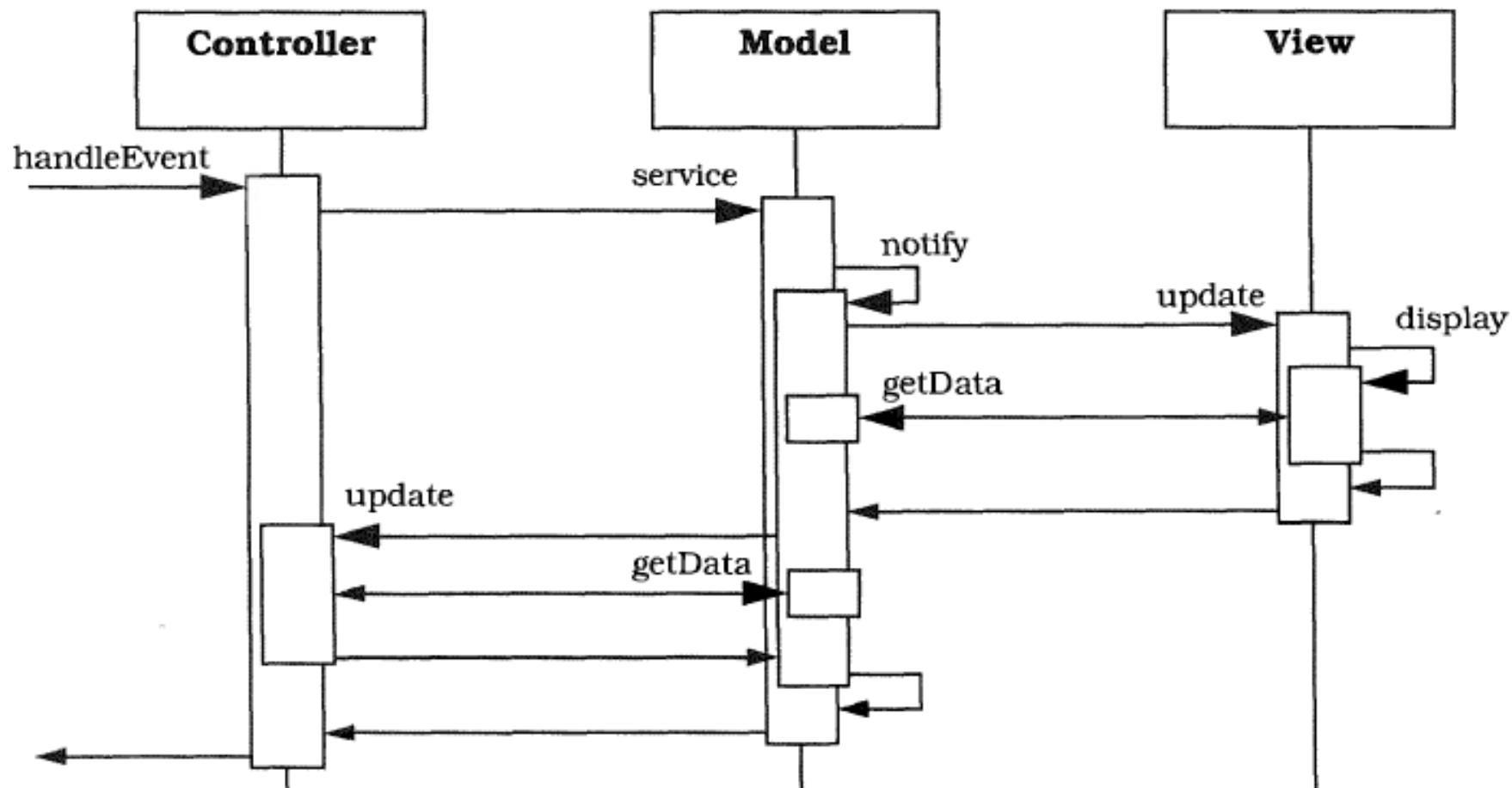
How does this work?

- The model has a list of views it supports
- Each view has a reference to its model, as well as its supporting controller
- Each controller has a reference to the view it controls, as well as to the model the view is based on. However, models know nothing about controllers.
- On user input, the controller notifies the model which in turn notifies its views of a change
- Changing a controller's view will give a different *look*.
- Changing a view's controller will give a different *feel*.

Scenario I



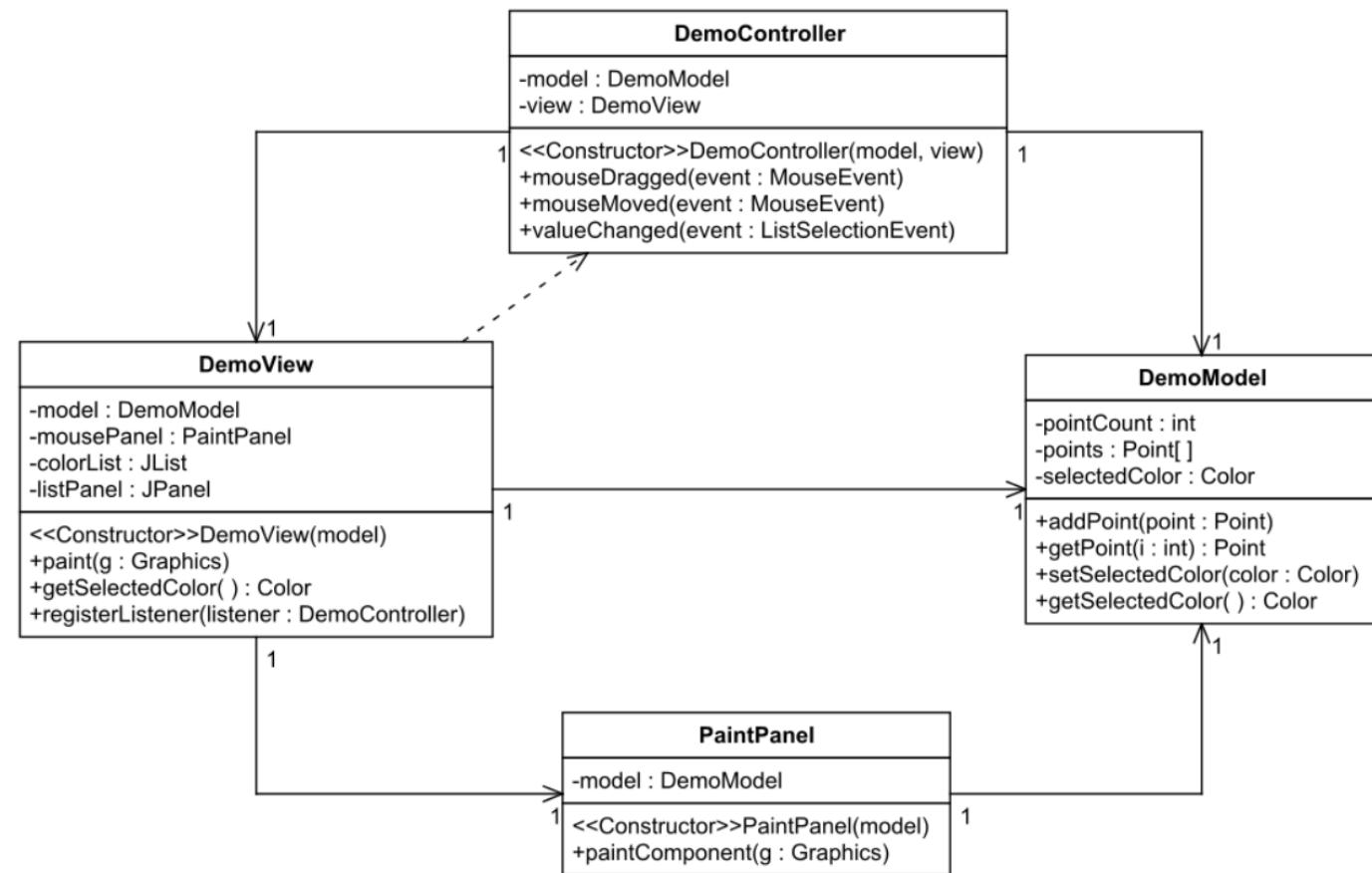
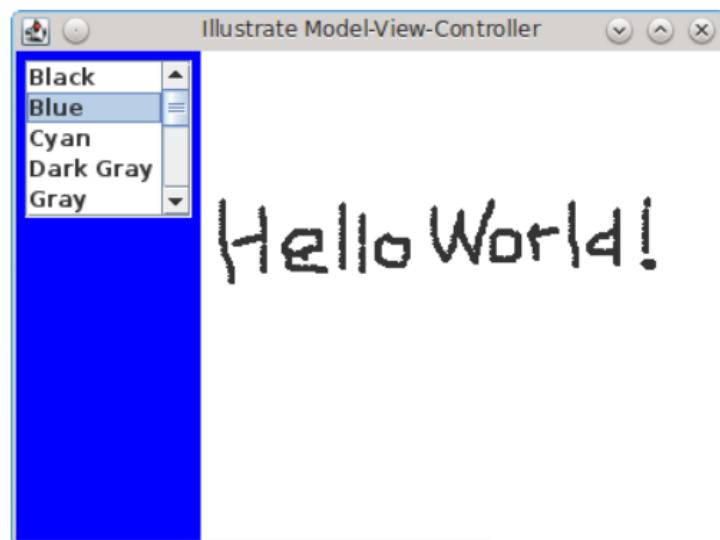
Scenario II



Consequences

- Benefits
 - Multiple views of the same model.
 - Synchronized views.
 - 'Pluggable' views and controllers
 - Exchangeability of 'look and feel'.
 - Framework potential.
- Liabilities
 - Increased complexity.
 - Potential for excessive number of updates.
 - Intimate connection between view and controller.
 - Close coupling of views and controllers to a model
 - Inefficiency of data access in view.
 - Inevitability of change to view and controller when porting.

MVC in ASP.NET



MVC Code

```

public abstract class Model
{
    private readonly ICollection<View> Views = new Collection<View>();
    public void Attach(View view)
    {
        Views.Add(view);
    }
    public void Detach(View view)
    {
        Views.Remove(view);
    }
    public void Notify()
    {
        foreach (View o in Views)
        {
            o.Update();
        }
    }
}

public class ConcreteModel : Model
{
    public object ModelState { get; set; }
}

```

```

public abstract class Controller
{
    public abstract void AlgorithmInterface();
}

public class ConcreteController : Controller
{
    public override void AlgorithmInterface()
    {
        // code here
    }
}

```

```

public abstract class View
{
    public abstract void Update();
    private readonly Controller Controller;
    protected View()
    {
    }
    protected View(Controller controller)
    {
        Controller = controller;
    }
    public void ContextInterface()
    {
        Controller.AlgorithmInterface();
    }
}

public class ConcreteView : View
{
    private object ViewState;
    private ConcreteModel Model { get; set; }
    public ConcreteView(ConcreteModel model)
    {
        Model = model;
    }
    public override void Update()
    {
        ViewState = Model.ModelState;
    }
}

```

MVP

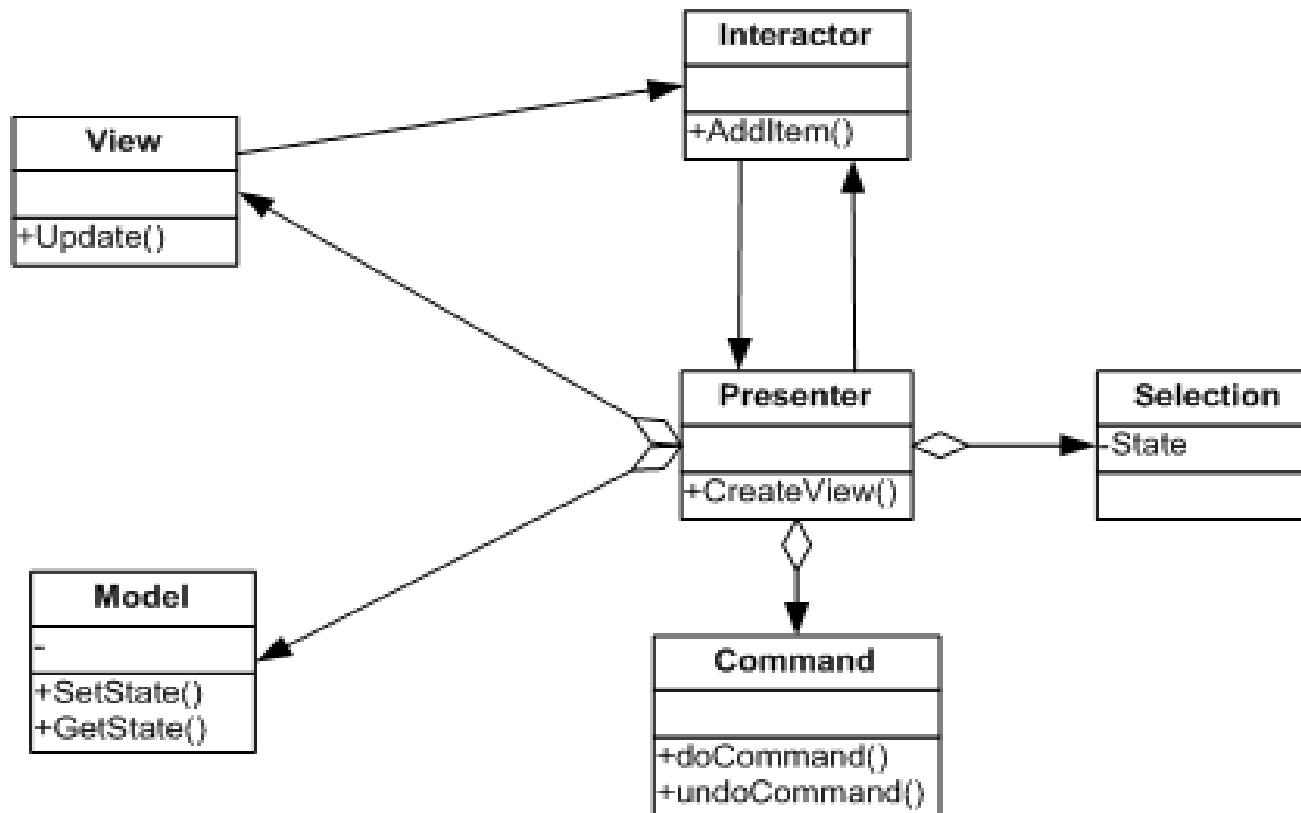


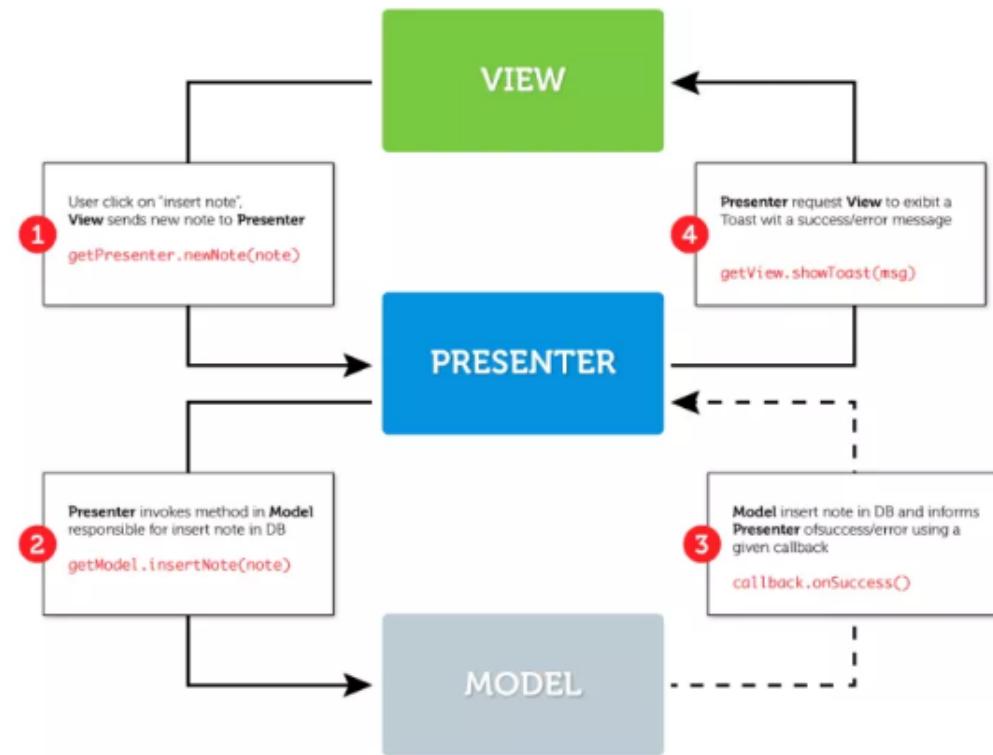
Figure 6: MVP

Usage

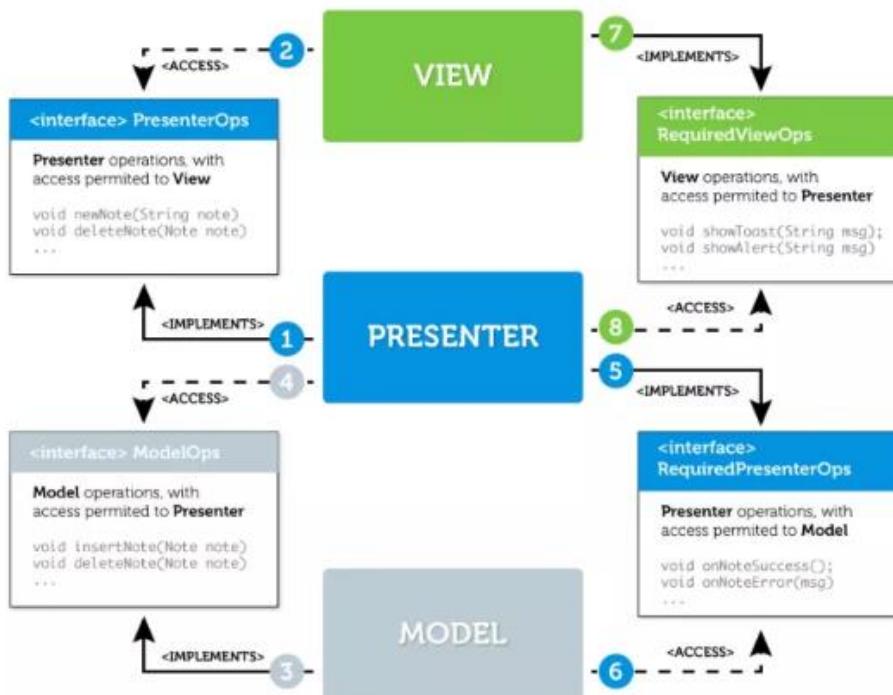
- MVP vs MVC

- delegates more work to the Presenter and removes the Controller.
- Presenter class encapsulates the View's state and commands. The Presenter is the mediator between Model and View!
- View is separated from Model => better separation of concerns

- MVP in Android example –
take notes on a journal



MVP Design



- **Presenter** implements **interface PresenterOps**
- **View** receives reference from **PresenterOps** to access **Presenter**
- **Model** implements **interface ModelOps**
- **Presenter** receives reference from **ModelOps** to access **Model**
- **Presenter** implements **RequiredPresenterOps**
- **Model** receives reference from **RequiredPresenterOps** to access **Presenter**
- **View** implements **RequiredViewOps**
- **Presenter** receives reference from **RequiredViewOps** to access **View**

MVP Code

[Android_MVP_code.txt](#)

MVVM

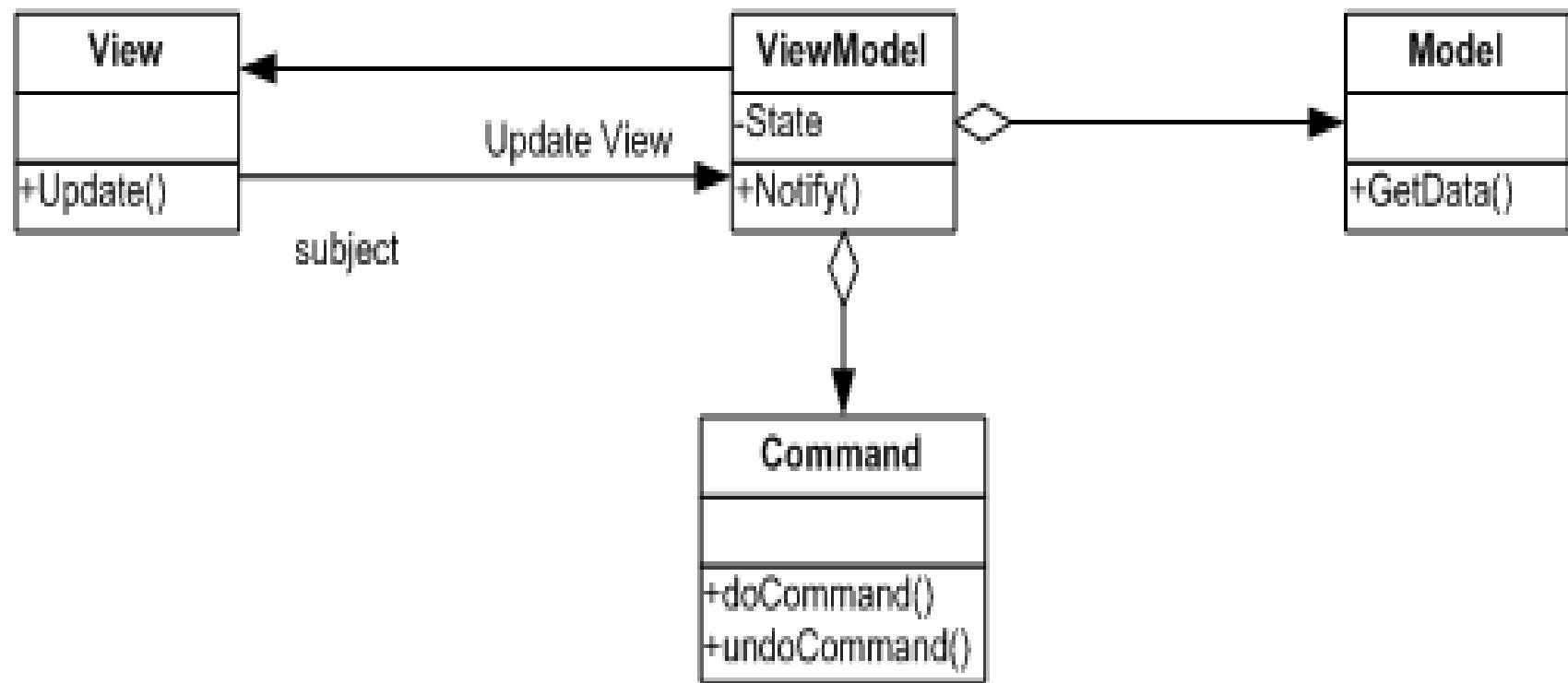


Figure 8: MVVM

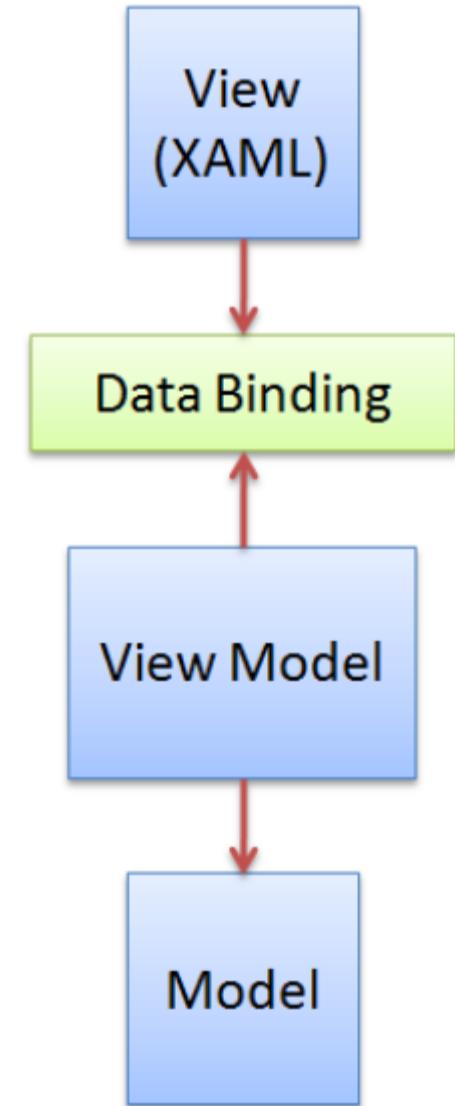
How it works

- View and ViewModel

- communicate via data-binding, method calls, properties, events, and messages
- The viewmodel exposes not only models, but other properties (such as state information, like the "is busy" indicator) and commands
- The view handles its own UI events, then maps them to the viewmodel via commands
- The models and properties on the viewmodel are updated from the view via two-way databinding

- ViewModel and Model

- The viewmodel may expose the model directly, or properties related to the model, for data-binding
- The viewmodel can contain interfaces to services, configuration data, etc., in order to fetch and manipulate the properties it exposes to the view



MVVM Code Example

```
namespace MVVMExample
{
    public class ContactModel : INotifyPropertyChanged
    {
        private string _firstName;

        public string FirstName
        {
            get { return _firstName; }
            set
            {
                _firstName = value;
                RaisePropertyChanged("FirstName");
                RaisePropertyChanged("FullName");
            }
        }

        private string _lastName;

        public string LastName
        {
            get { return _lastName; }
            set
            {
                _lastName = value;
                RaisePropertyChanged("LastName");
                RaisePropertyChanged("FullName");
            }
        }
    }
}
```

```
<UserControl x:Class="MVVMExample.DetailView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid x:Name="LayoutRoot" Background="White"
        DataContext="{Binding CurrentContact}">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <TextBlock Text="Name:" HorizontalAlignment="Right" Margin="5" Grid.Column="1"/>
        <TextBlock Text="{Binding FullName}" HorizontalAlignment="Left" Margin="5" Grid.Column="1"/>
        <TextBlock Text="Phone:" HorizontalAlignment="Right" Margin="5" Grid.Row="1"/>
        <TextBlock Text="{Binding PhoneNumber}" HorizontalAlignment="Left" Margin="5" Grid.Row="1" Grid.Column="1"/>
    </Grid>
</UserControl>
```

```
namespace MVVMExample
{
    public class ContactViewModel : BaseINPC
    {
        public ContactViewModel()
        {
            Contacts = new ObservableCollection<ContactModel>();
            Service = new Service();

            Service.GetContacts(_PopulateContacts);

            Delete = new DeleteCommand(
                Service,
                ()=>CanDelete,
                contact =>
                {
                    CurrentContact = null;
                    Service.GetContacts(_PopulateContacts);
                });
        }

        private void _PopulateContacts(IEnumerable<ContactModel> contacts)
        {
            Contacts.Clear();
            foreach(var contact in contacts)
            {
                Contacts.Add(contact);
            }
        }

        public IService Service { get; set; }

        public bool CanDelete
        {
            get { return _currentContact != null; }
        }
    }
}
```

Food for thought...😊

Why I No Longer Use MVC Frameworks,
posted by [Jean-Jacques Dubray](#) on Feb 03, 2016

