EGR326 Software Design and Architecture

Lecture 10. Middleware Architectures

Spring 2020

Kim Peters, Ph.D.

Gordon and Jill Bourns College of Engineering
California Baptist University

Week 5-2 Objectives

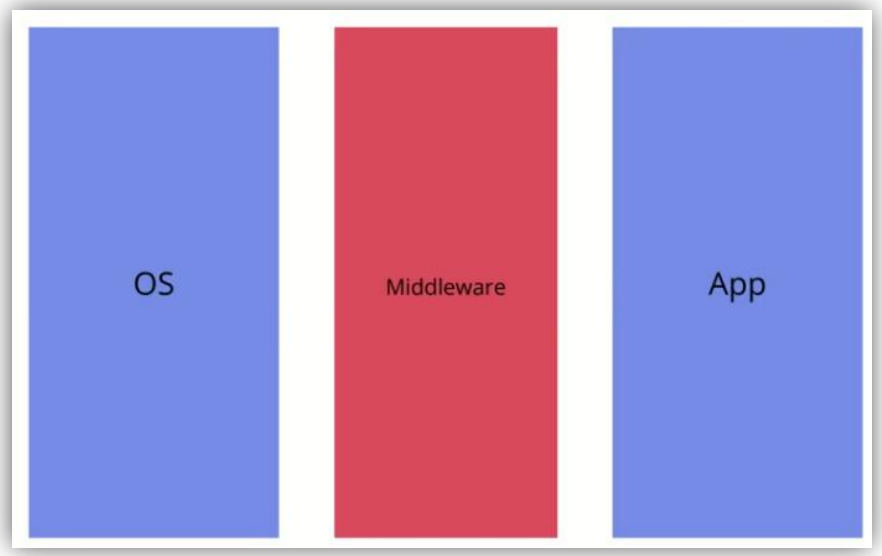
- Middleware
  - Application Architecture
- Distributed Objects
  - CORBA
  - IDL
- Message-Oriented Middleware
- Application Servers
- Message Brokers
- JEE - EJB

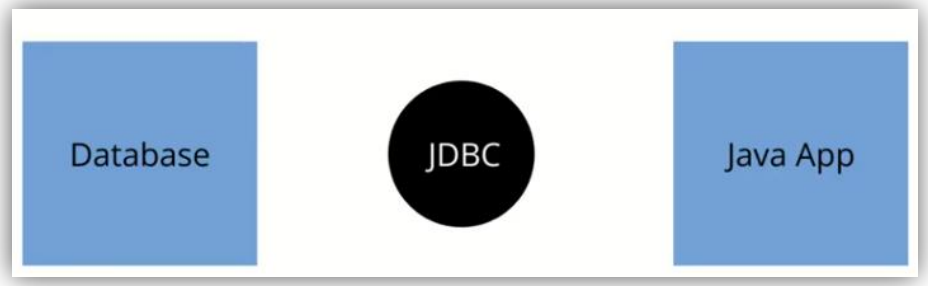
What is Middleware?

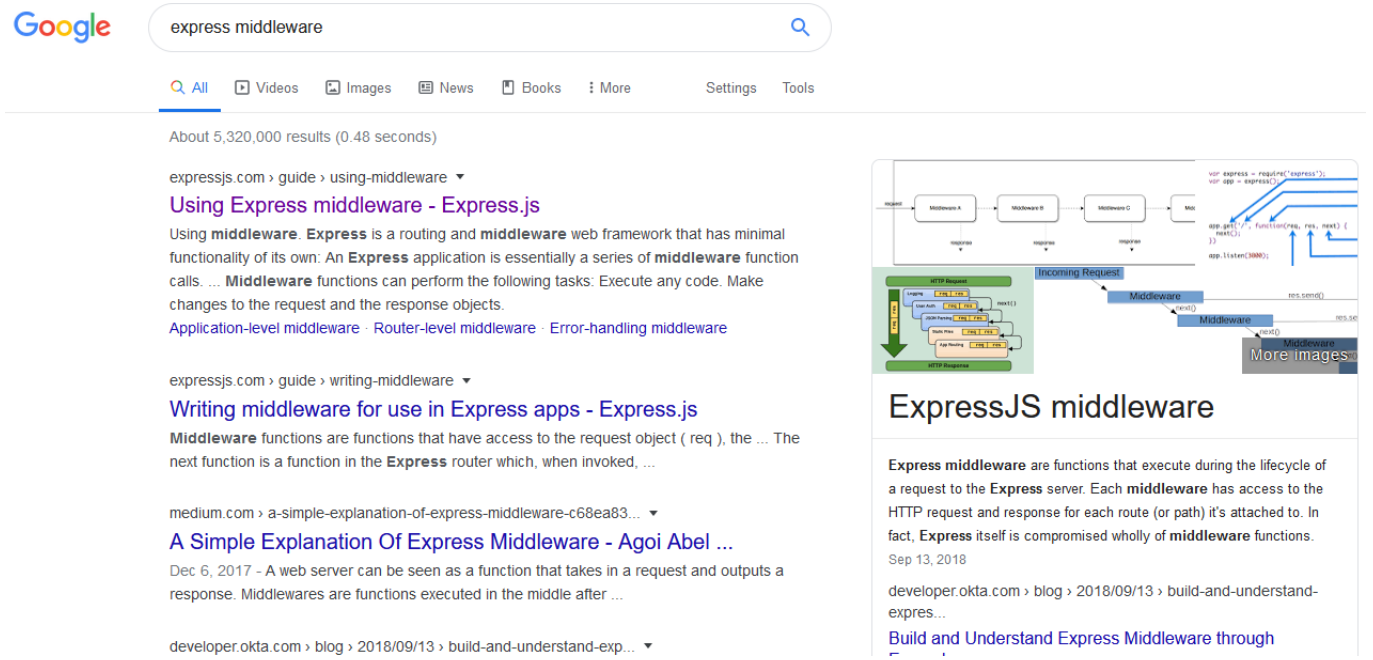
✓ **Middleware is software that connects components – *back end***

- The connections between a database and an application front-end
- Software that connects and coordinates pre-existing subsystems
- Software that functions as a conversion or translation layer
- The (hidden) infrastructure that makes an application work

✓ Middleware is software that connects components – back end

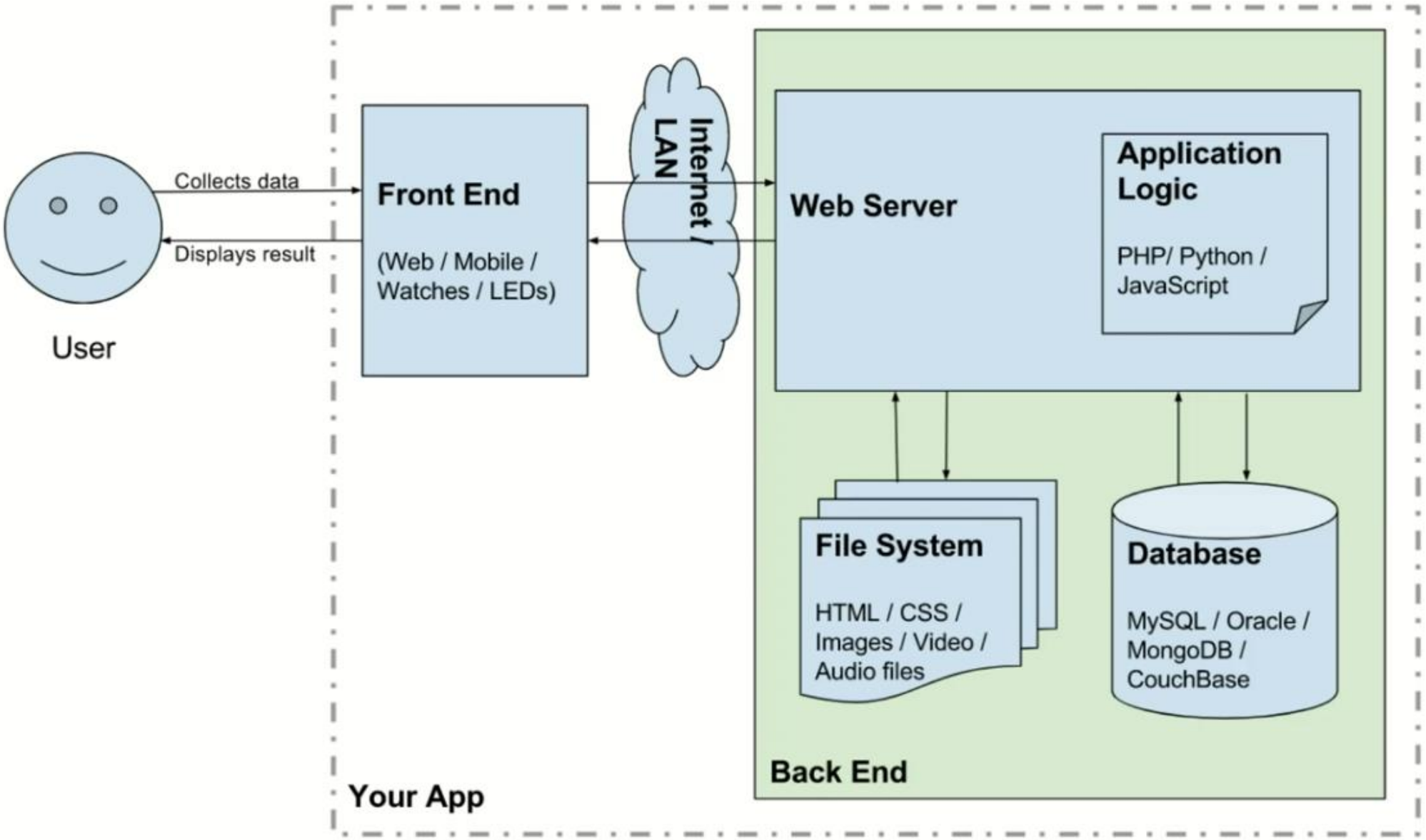






https://expressjs.com/en/guide/using-middleware.html

Application Architecture – Frontend, Backend and Web Servers



✓ Common middleware technologies fall into several levels of the software hierarchy

- BPOs allow long and complex workflows to be implemented
- Message brokers process messages and support multithreading
- Application servers also handle transactions and security
- Transport layer sends requests and moves data

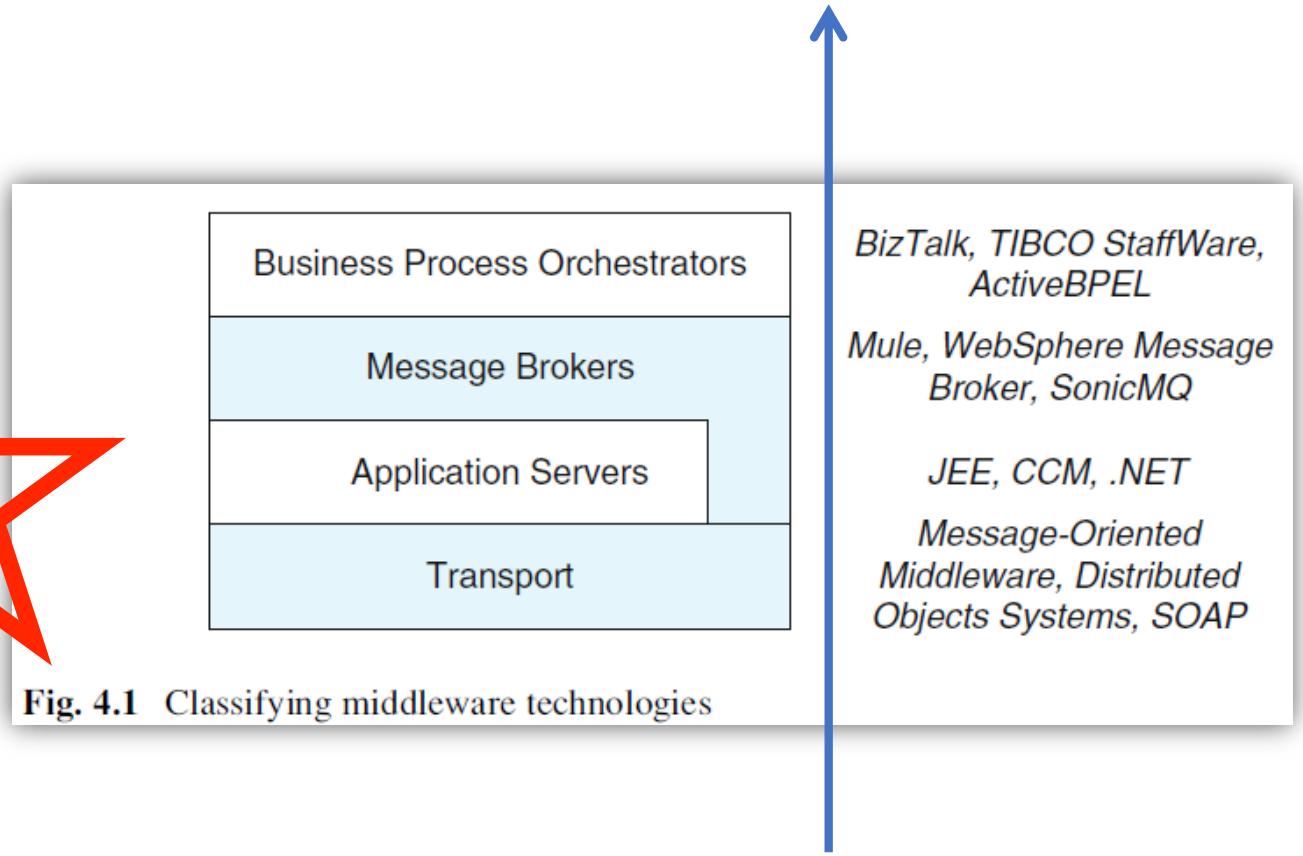
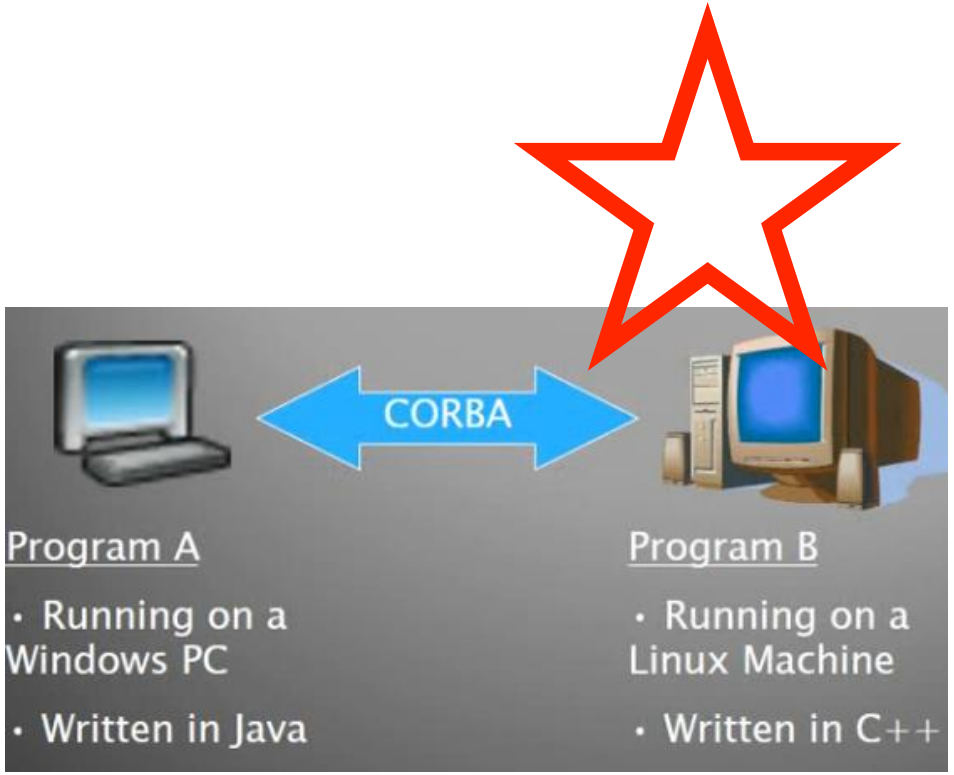


Fig. 4.1 Classifying middleware technologies

- ✓ **Distributed Objects allows the objects to be resident on different machines and to communicate through an Object Request Broker (ORB)**
  - **Pros:** Distributed objects allow you to break up an application across a network
  - **Cons:** As a distributed objects application gets larger, you need help from middleware services for transactions, security etc.
  - **Objects in CORBA** obey all of the pleasant characteristics that OOP brings to us
    - Separation of interface from implementation
    - Cohesion and encapsulation
  - **In CORBA,** objects communicate by sending requests to, and responding to requests from, the ORB

✓ CORBA supports **remote invocations** – where the requesting and serving objects are on *different machines*

- **CORBA (Common Object Request Broker Architecture)**
  - Introduced in 1991, defined the interface design language (IDL) and application programming interface (API)
  - Is a standard that enables an object written in one programming language, running on one platform to interact with objects across the network that are written in other programming languages and running on other platforms.
  - Ex) a client object written in C++ and running under Windows can communicate with an object on a remote machine written in Java running under UNIX.



CORBA (Common Object Request Broker Architecture)

✓ **CORBA supports remote invocations – where the requesting and serving objects are on *different machines***

- A complete distributed object platform
- CORBA is not a programming language
- Allows object to transparently make request and receive responses
- The world's leading middleware solution enabling the exchange of information, independent of hardware platforms, programming languages, and operating systems

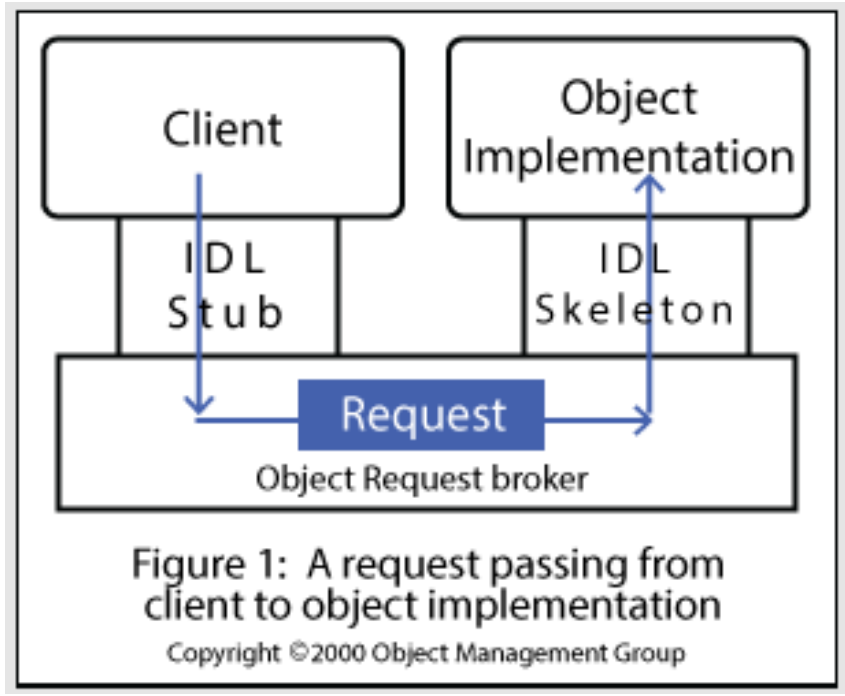
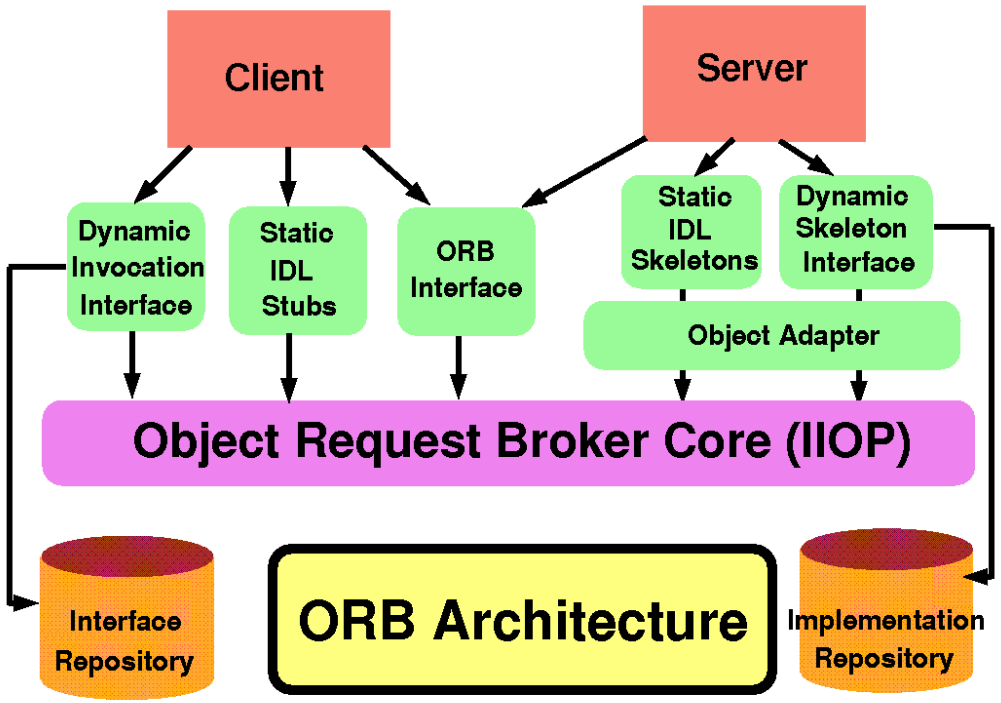


Figure 1: A request passing from client to object implementation

✓ CORBA architecture is based on the object model

- A collection of *object* that isolates requestors of service (clients) from the provider of services (servers) by a well defined encapsulating interface

- **Composed of Five major components:**
  1) ORB: object request brokers
  2) IDL: interface definition language
  3) Dynamic Invocation Interface
  4) Interface Repositories (IT)
  5) Object Adapters (OA)



http://www.omg.org/gettingstarted/corbafaq.htm

✓ CORBA architecture is based on the object model

A collection of *object* that isolates requestors of service (clients) from the provider of services (servers) by a well defined encapsulating interface

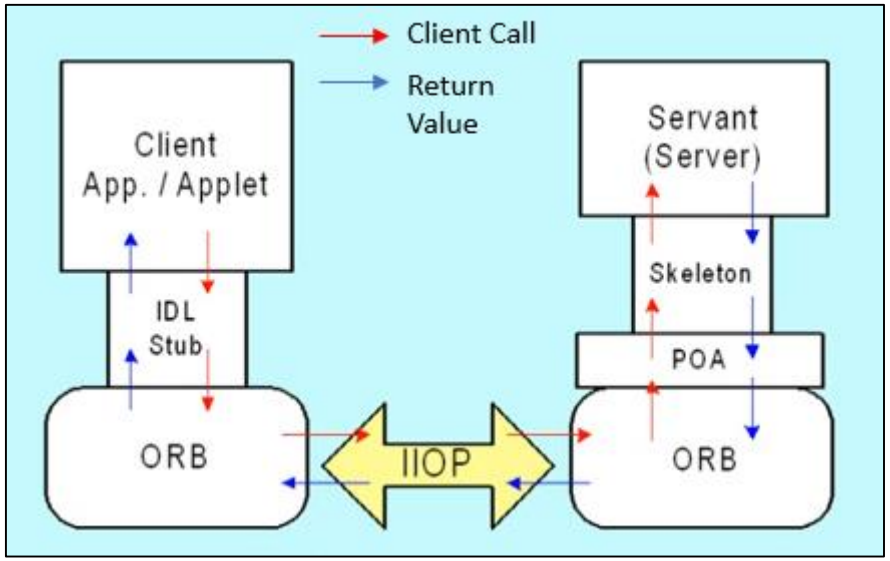
- **Objects Model Provides:**
  1) Abstraction: group objects and focus on their similarities
  2) Encapsulation: hide implementation details from the service provided
  3) Inheritance: ability to pass along (object-to-object) capabilities and behaviors
  4) Polymorphism: ability to substitute objects with matching interfaces run time

✓ CORBA architecture is based on the object model

- A collection of *object* that isolates requestors of service (clients) from the provider of services (servers) by a well defined encapsulating interface

- **Primary Elements**
  - **IDL**: interface definition language
  - **Client/Server CORBA Objects**: abstract objects based upon a concrete implementation
  - **ORB**: object request brokers
  - **GIOP/IIOP**: general and internet inter-object protocols

Standard Call and Return

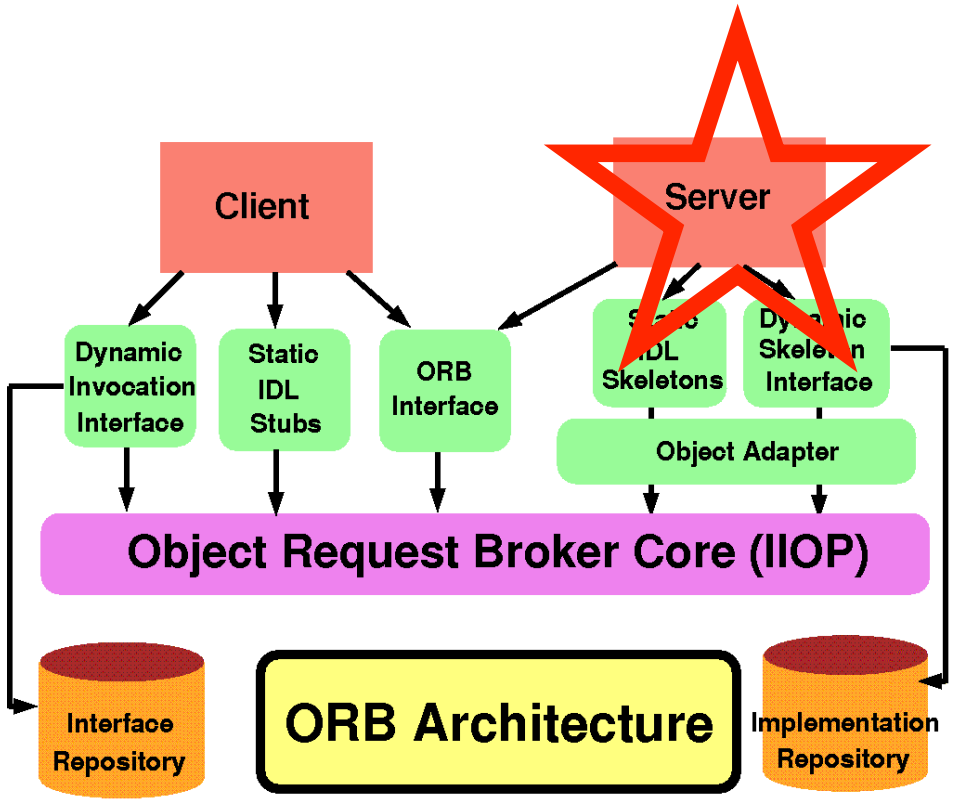


✓ CORBA architecture is based on the object model

- A collection of *object* that isolates requestors of service (clients) from the provider of services (servers) by a well defined encapsulating interface

- **Primary Elements**

  - **IDL**: interface definition language
    - Defines public interface for any CORBA server
    - C++ like syntax
    - Client and Server implemented based on compilation of the same IDL, most often
    - OMG has defined mappings for: C, C++, Java, COBOL, Smallltalk, ADA, Lisp, Python and IDL script



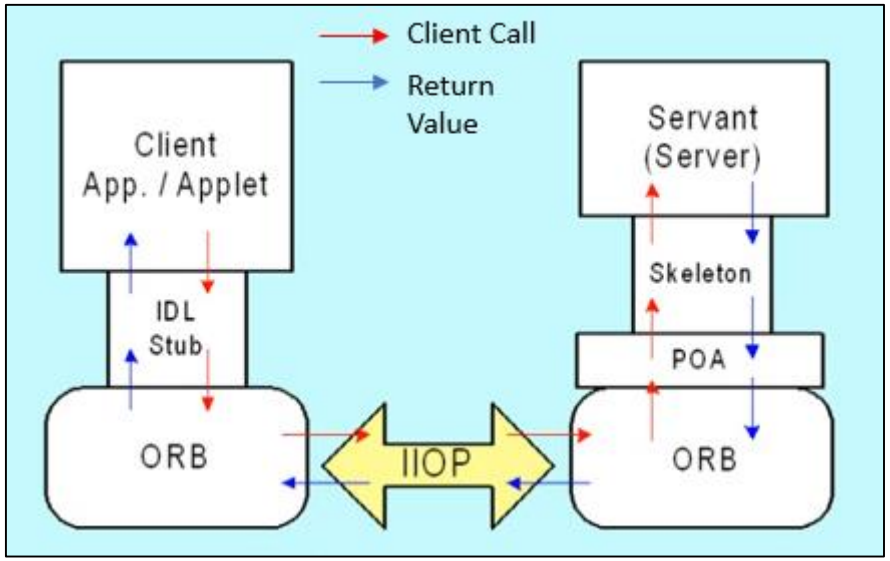
http://www.omg.org/gettingstarted/corbafaq.htm

✓ CORBA architecture is based on the object model

- A collection of *object* that isolates requestors of service (clients) from the provider of services (servers) by a well defined encapsulating interface

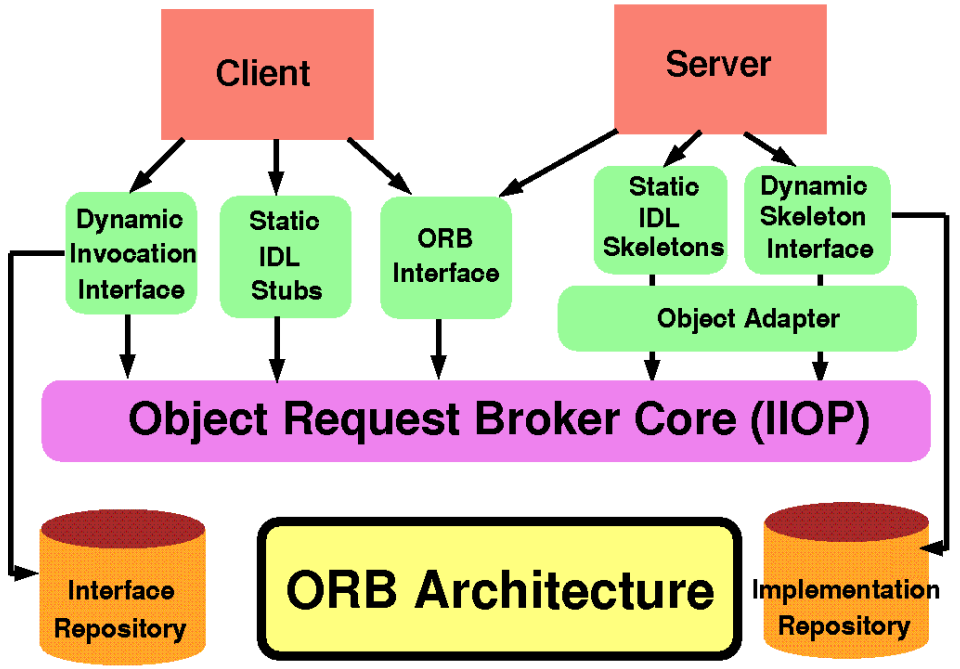
- **Highlighted IDL Features**
  - Pass by refence and by value
  - In, out and input parameters
  - Inheritance
  - Throwing of exceptions
  - Callbacks: enables peer-to-peer object communication
  - Also supports: structs, unions, enumerations, all C++ scalars, arrays, sequences, strings, constraints and typedefs

Standard Call and Return



✓ Object Request Brokers (ORBs)

- **Responsible for all communication**
  - Locating object
    - Implementation specific
    - Known IOR (Inter-Object Reference)
    - Naming and Trading Services (DSN like)
  - Transferring invocations and return values
  - Notifying other ORBs of hosted Objects
- Must be able to communicate IDL invocations via IIOP
- If an ORB is OMG compliant, then it is interoperable with all other OMG compliant ORBs.



http://www.omg.org/gettingstarted/corbafaq.htm

✓ Object Adapter Structure

- **Object Adapter (server)**
  - Bridges the gap between CORBA objects and the programming language interfaces of the servant classes
  - Creates remoter object references for the CORBA objects
  - Dispatches each RMI to the appropriate servant class via a skeleton and activate objects
  - Assigns a unique name to itself and each object
  - Called the Portable Object Adapter in CORBA 2.0
    - Processes can run on ORB's produced by different developers

✓ Skeletons and Proxies Structure

- **Skeletons (server)**
  - An IDL compiler generates skeleton classes in the server's language
  - Dispatch RMI's to the appropriate servant class

- **Client Proxies / Stubs**
  - Generated by an IDL compiler in the client language
  - A proxy class is created for object oriented languages
  - Stub procedures are created for procedural languages

✓ Both are responsible for marshalling and unmarshalling arguments, results and exceptions

✓ Repositories Structure

- **Implementation Repository**
  - Activates registered servers on demand and locates servers that are currently running

- **Interface Repository**
  - Provides information about registered IDL interfaces to the clients and servers that require it.
  - Optional for static invocation; required for dynamic invocation

✓ GIOP and IIOP Features

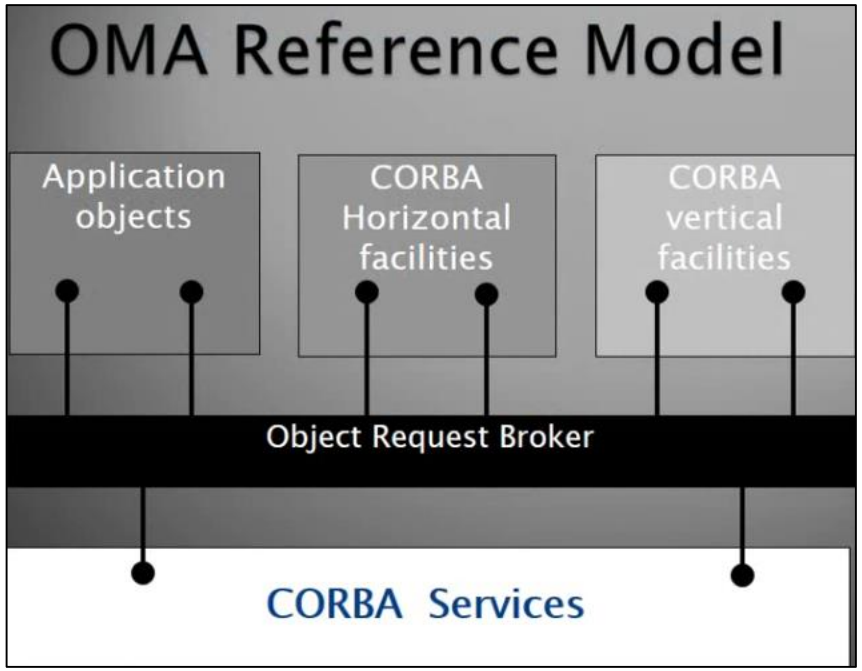
- **GIOP:**
  - General Inter-ORB Protocols are the standards (included in CORBA 2.0), which enable implementations to communicate with each other regardless of who developed it

- **IIOP:**
  - Internet Inter-ORB Protocol is an implementation GIOP that uses the TCP / IP protocol for the Internet

✓ Center of all the activity undertaken by OMG

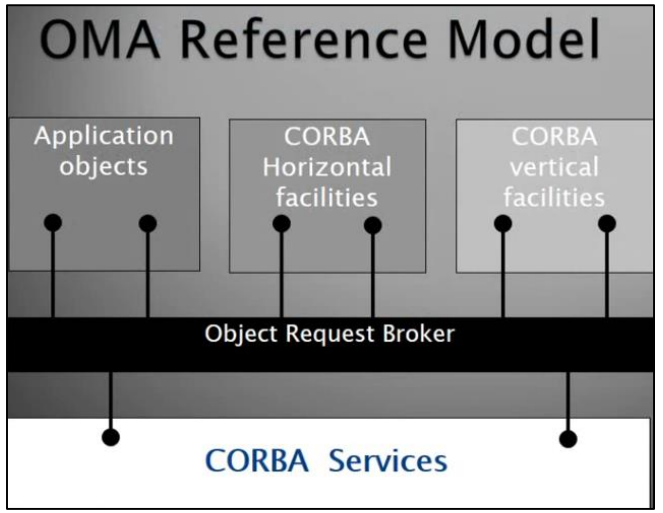
- OMA specifies a range of architectural entities surrounding the core ORB, which is CORBA proper

- Detailed specifications for each component and interface category is populated in OMA Reference Model



✓ OMA Reference Model

- **CORBA Services**
  - Provides basic functionality, similar to the services that system library call do in UNIX.
  - Functions includes creating objects, controlling access to objects, keeping track of relocated objects and to consistently maintain relationship between objects



- **Horizontal CORBA Facilities**
  - Sits between the CORBA services and application objects
  - Provide support across an enterprise and across business
  - Four facilities
    1) Printing facilities
    2) Secure time facilities
    3) Internationalization facilities
    4) Mobile agent facilities

- **Domain (Vertical) Facilities**
  - The most exciting work at OMG
  - Define a standard interfaces for standard objects shared by companies within a specific vertical market (eg. Manufacturing, finance, healthcare)
  - Specific task force for industries

How do we use CORBA?

✓ **CORBA is a standard – to use it, we must obtain an implementation; several open-source versions are available**

- Learn by example http://www.omg.org/gettingstarted/corbafaq.htm

- Much of the application software has to do with creating and registering components with the ORB

- In early days there were problems with incompatibility; these have been resolved

- **Netscape browsers**
- **Netscape ONE (Open Network Environment) –** Netscape's application environment based on open internet standards
- **Enterprise Edition of IBM's Web Sphere**

- **The steps involved are:**
  1) Define an interface (IDL file)
  2) Map IDL to Java (idlj compiler)
  3) Implement the Interface
  4) Write s Server
  5) Write a Client
  6) Run the Application

Step 1 – Define the Interface

- Hello.idl

```
module HelloApp {
   interface Hello {
      string sayHello();
   };
};
```

Step 2 – Define the Interface

- Use the idlj compiler
  - Idlj -fall Hello.idl (Inheritance model)
  - Idlj -fallTie -oldImplBase Hello.idl(Tie model)
- This will Generate :
  - _HellImplBase.java (Server Skeleton)
  - HelloStub.java (Client Stub)
  - Hello.java
  - HelloHelper.java
  - HelloHolder.java
  - HelloOperations.java

Step 3 – Implement the Interface

- Implement the Server

```
public class HelloImpl extends _HelloImplBase {

   public String sayHello() {
      return "\nHello world !!\n";
   }
}
```

Step 4 – Implement the Server

- Create and initializes an ORB instance

  ORB orb =ORB.init(args,null);
- Creates a servant instance(the Implementation of one CORBA Hello object)

  Helloimp helloimpl = new Helloimpl();
- Obtain a Reference for desired object and Register it with ORB
  - By creating a Tie with Servant (Tie model)
  - By naming context (Inheritance model)
- "Stringify" the Reference
  - String ior =orb.object_to_String (Hello);
- Waits for invocations of new object from the Client
  - orb.run();

    Thread.current Thread().join();

Step 5 – Write a Client

- Creates and initializes an ORB instance

  ORB orb =ORB.init(args,null);

- Obtain a reference for the desired object from IOR

  - read stringified object from IOR

  - resolve the Object Reference(by HelloHelper.narrow() method)

- Invokes the object's say Hello operations and print the result

Step 6 – Run the Application

- Compile the .java files including the stubs and skeletons

  Javac HelloApp.java

- Start orbd

  –orbd ORBInitialPort1050 –ORBInitialHost localHost&

- Start the Hello Server

  java HelloServer -ORBInitialPort 1050 –ORBInitialHost localHost&

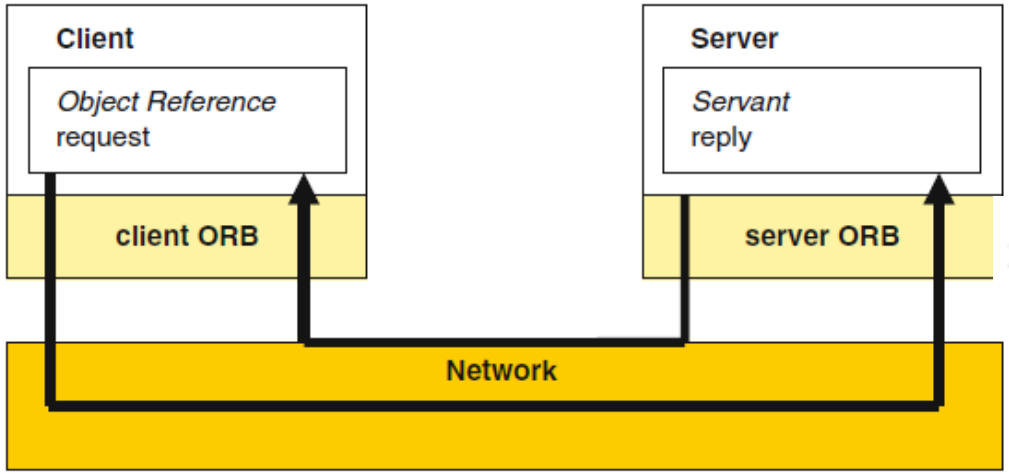
- Run The Client Application

  java HelloClient -ORBInitialPort 1050 –ORBInitialHost localHost&

- ✓ CORBA supports **remote invocations** – where the requesting and serving objects are on *different machines*

```
ORB orb = ORB.init(args, null);
// Lookup is a wrapper that actually access the CORBA Naming //
Service directory - details omitted for simplicity
MyServant servantRef = lookup("Myservant")
String reply = servantRef.isAlive();
```

```
class MyServant extends _MyObjectImplBase {
  public String isAlive()    {
    return "\nLooks like it…\n";
    }
}
```




```
ORB orb = ORB.init(args, null);
MyServant objRef = new MyServant();
orb.connect(objRef);
```

Fig. 4.2 Distributed objects using CORBA

- Internet InterORB Protocol (IIOP) maps CORBA messages through the Internet protocol

- ✓ CORBA supports **remote invocations** – where the requesting and serving objects are on *different machines*

Advantages:

- Object-Oriented distributed communication
- Easy to define messages and data contents using IDL
- Once setup/configuration of ORB's, POA's, Servants, IOR's complete, communication is fairly straightforward
- Relieves programmers from having to write Socket software

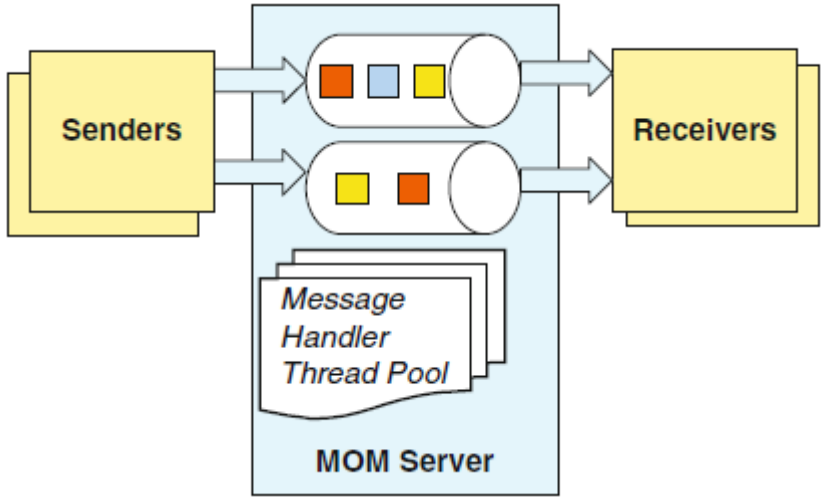
Disadvantages:

- Can be difficult to learn
  - Even harder to master
- Scalability issues could cause a mind-boggling amount of ORB's, POA, threads, etc. to be used
- Acquiring IOR's can be difficult

✓ Message-Oriented Middleware provides a standard queue between message creators and receivers (clients and servers)

- **CORBA is an example of a synchronous architecture**
  - Servers respond to client requests "promptly“

- **MOM supports multiple queues**
  - One or more processes can send messages to a given queue
  - Any process can receive from one or more queues
  - FIFO message receipt
  - Queues are referred to by name
  - Sending is a non-blocking operation (in general)
  - TCP/IP sockets are commonly used



✓ As an aside: TCP/IP Sockets

- A network socket is just an endpoint for communication
- TCP/IP is the reliable protocol for sending data over an IP network
- Programming languages have sockets APIs
  - Functions to create a socket (assign to a given IP address), read from and write to, and close a socket

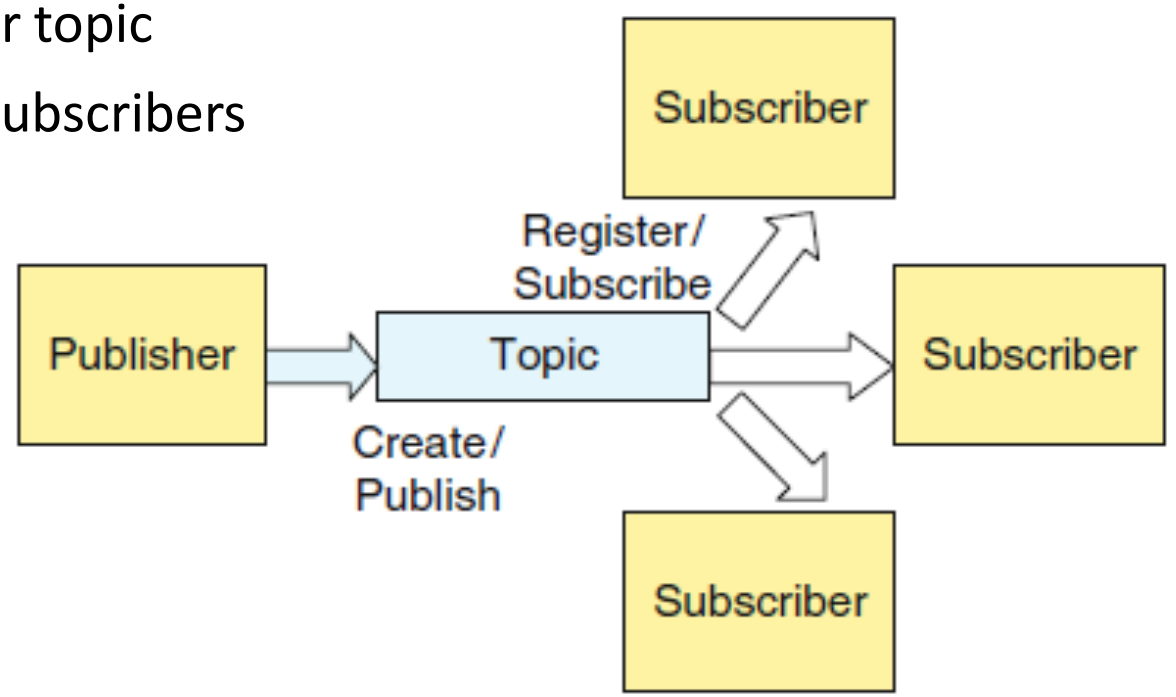
```
struct sockaddr_in serv_addr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    server = gethostbyname(argv[1]);

    serv_addr.sin_family = AF_INET;
        serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    …
    n = write(sockfd,buffer,strlen(buffer));
    …
    n = read(sockfd,buffer,255);
    …
    close(sockfd);
    return 0;
```

✓ Message-Oriented Middleware is based on a 1-1 messaging style, but we can also build "publish-subscribe" systems

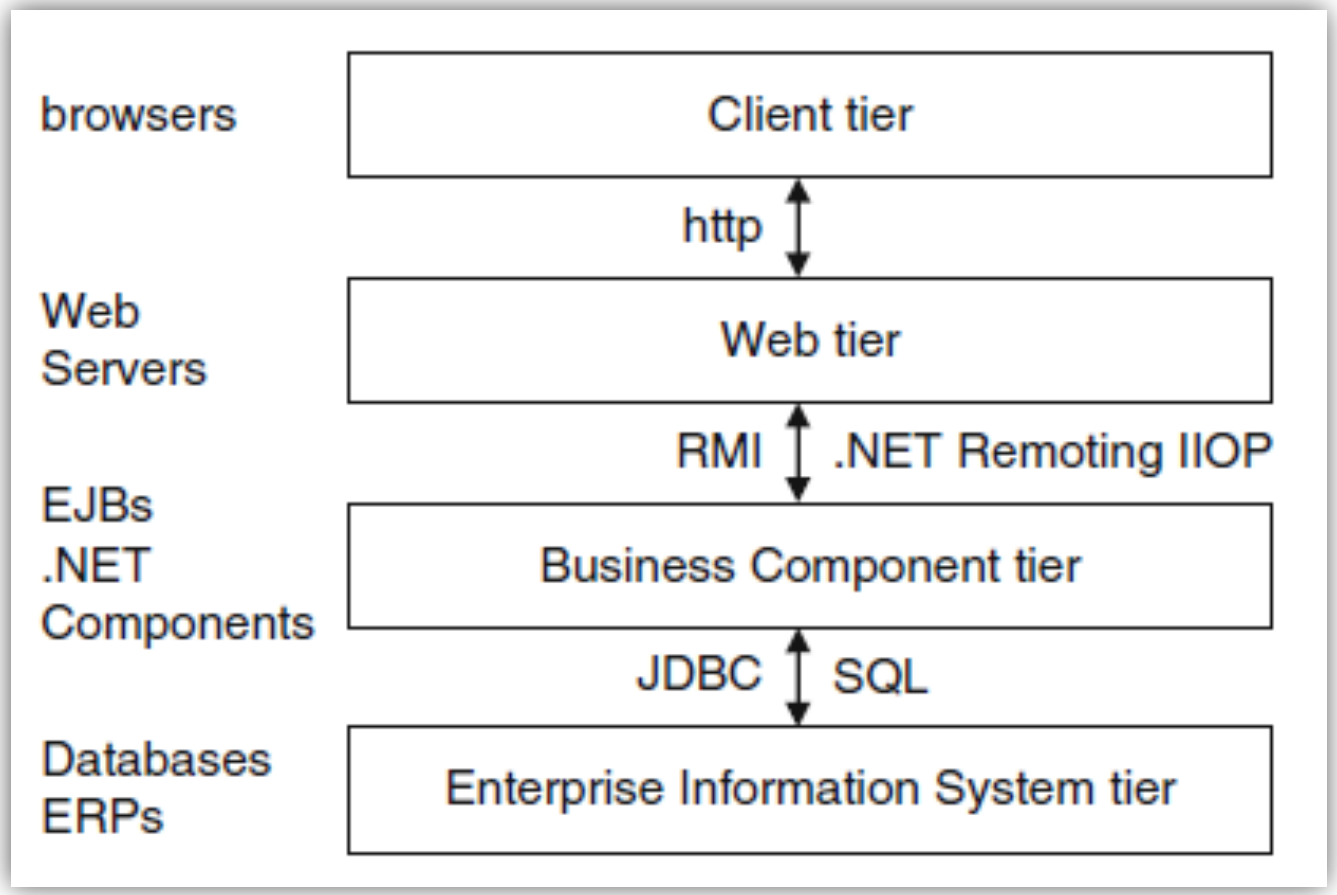
- Many applications require 1-many, many-1 or many-many communication between components
- Sources of messages are called *Publishers*
- Publishers send a message to a named *topic* or *subject*
- Subscribers register to *listen* to a particular topic
- All messages on that topic are sent to its subscribers



✓ Application Servers are middleware that provide: distributed communications, security, transactions and persistence

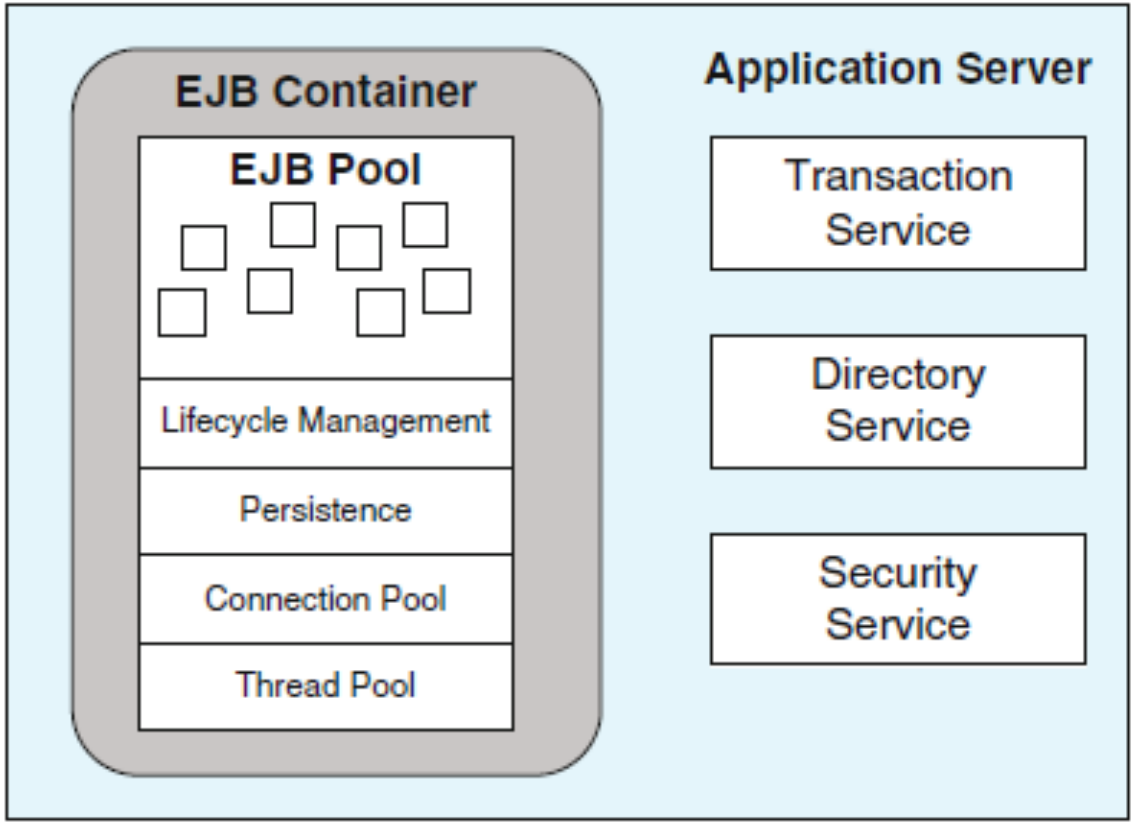
- A familiar example is **JEE, Java Enterprise Edition** (still often called J2EE)

- An application server will implement the two middle tiers of the N-tier model shown here



✓ A big part of JEE is Enterprise JavaBeans (EJB) – but what is it?

- A **JEE - compliant server** features a Java virtual machine that hosts EJB components
- An **EJB client** invokes a component
- The container allocated a thread and invokes an instance of the desired component
- All interactions with the EJB component must go through the container
  - Security

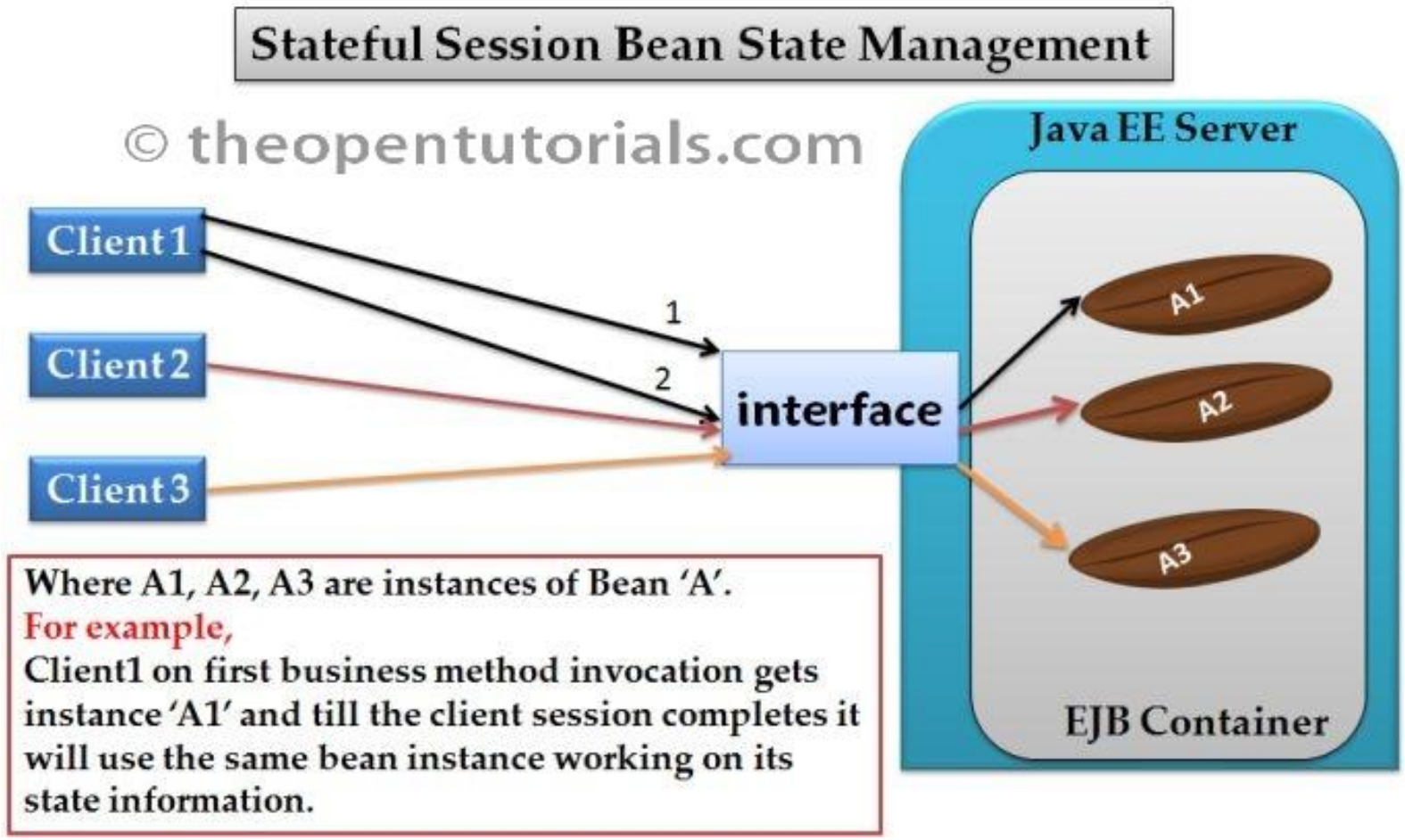


✓ EJB identifies two types of beans – Session beans and Message-driven beans

- Message-driven beans process messages asynchronously
- When a message arrives, it executes some business logic, then waits for another message

- Session beans aren't necessarily driven by external messages
- Session beans can be *stateless* or *stateful*
  - **with stateless beans**, external clients need to keep track of what they have asked for and of progress towards goals
  - **stateful beans** is considered to be "conversational"

- ✓ A conversation with a stateful bean must continue with that same bean…



EJB Session Beans - stateless

```
import javax.ejb.Remote;
@Remote
public interface Broker {
    public int newAccount(String name, String address,
        int credit)
        throws EJBException, SQLException;

    public void buyStock(int accno, int stock_id, int amount)
        throws EJBException, SQLException;

    public void updateAccount(int accno, int credit)
        throws EJBException, SQLException;
}
```

```
import javax.ejb.Stateless;
@Stateless
public class BrokerBean implements Broker {
       // methods defined here …(not shown)
}
```

```
@EJB BrokerBean broker;
Broker.updateAccount ( 99, 10000);
```

For more information on EJB

- ✓ Setting up a simple EJB project in Eclipse…

http://theopentutorials.com/examples/java-ee/ejb3/how-to-create-a-simple-ejb3-project-in-eclipse-jboss-7-1/

✓ Common middleware technologies fall into several levels of the software hierarchy

- BPOs allow long and complex workflows to be implemented
- Message brokers process messages and support multithreading
- Application servers also handle transactions and security
- Transport layer sends requests and moves data

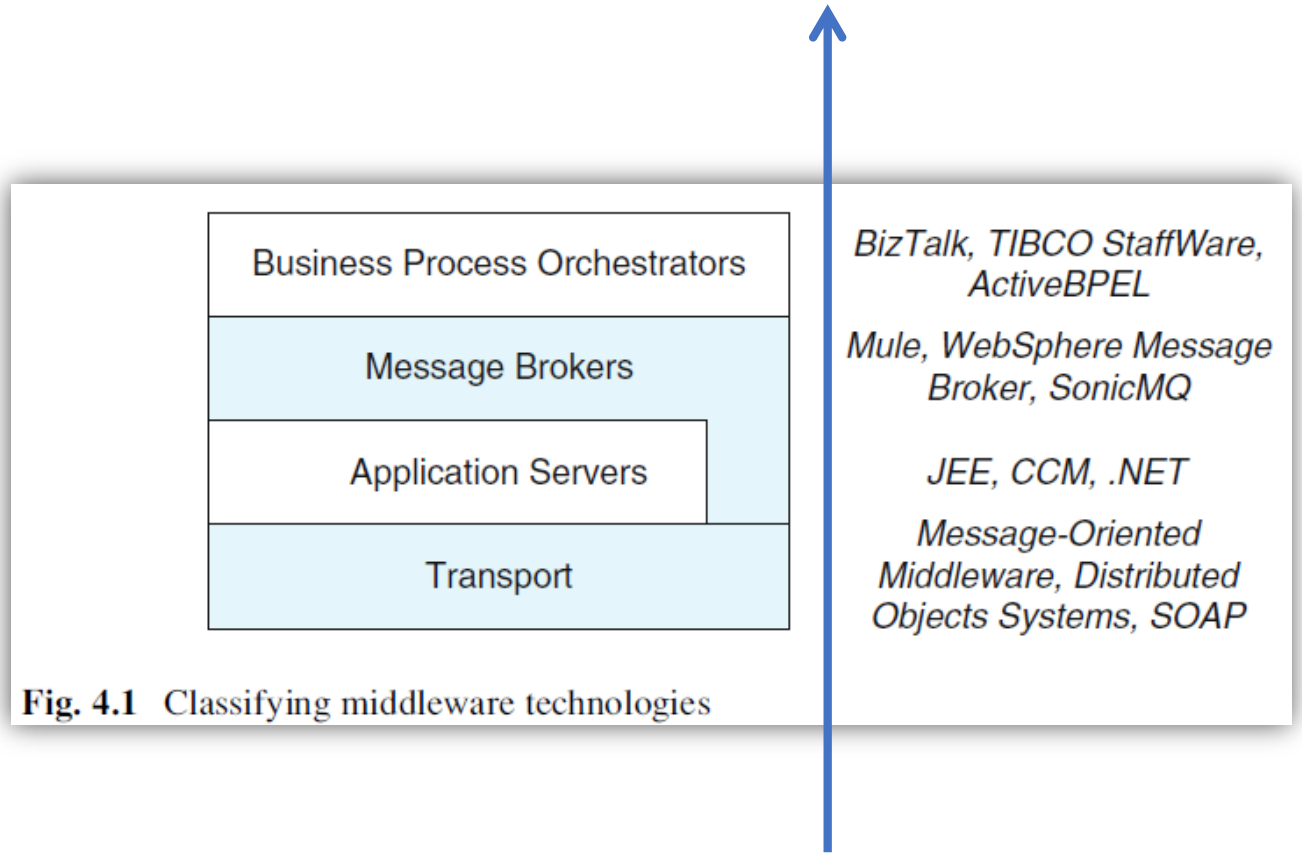


Fig. 4.1 Classifying middleware technologies

Week 5-2 Objectives

- Middleware
- Distributed Objects
- Message-Oriented Middleware
- Application Servers
- Message Brokers
- JEE - EJB

- **Assignment 6: Component Class Design**
  - We are looking at the use of Design Patterns in design; most specifically, in the Component design portion of a SW project. This is the phase in which we decide what classes and arrangement of classes will fill a particular need, once the large functional blocks have been determined.
  - Submit a diagram showing your class design. Explain the use case scenario at each level
  - Upload your work in word document via blackboard >> EGR326 >> Assignment >> **Asn 6:** Component Design
  - **Due date**: Friday, Feb 14 by 10:00pm

- **Software:**
  - Lucichart: webapp
  - Draw.io: webapp

EGR326 Software Design and Architecture

Lecture 9. SW Quality Attributes

Spring 2020

Kim Peters, Ph.D.

Gordon and Jill Bourns College of Engineering
California Baptist University

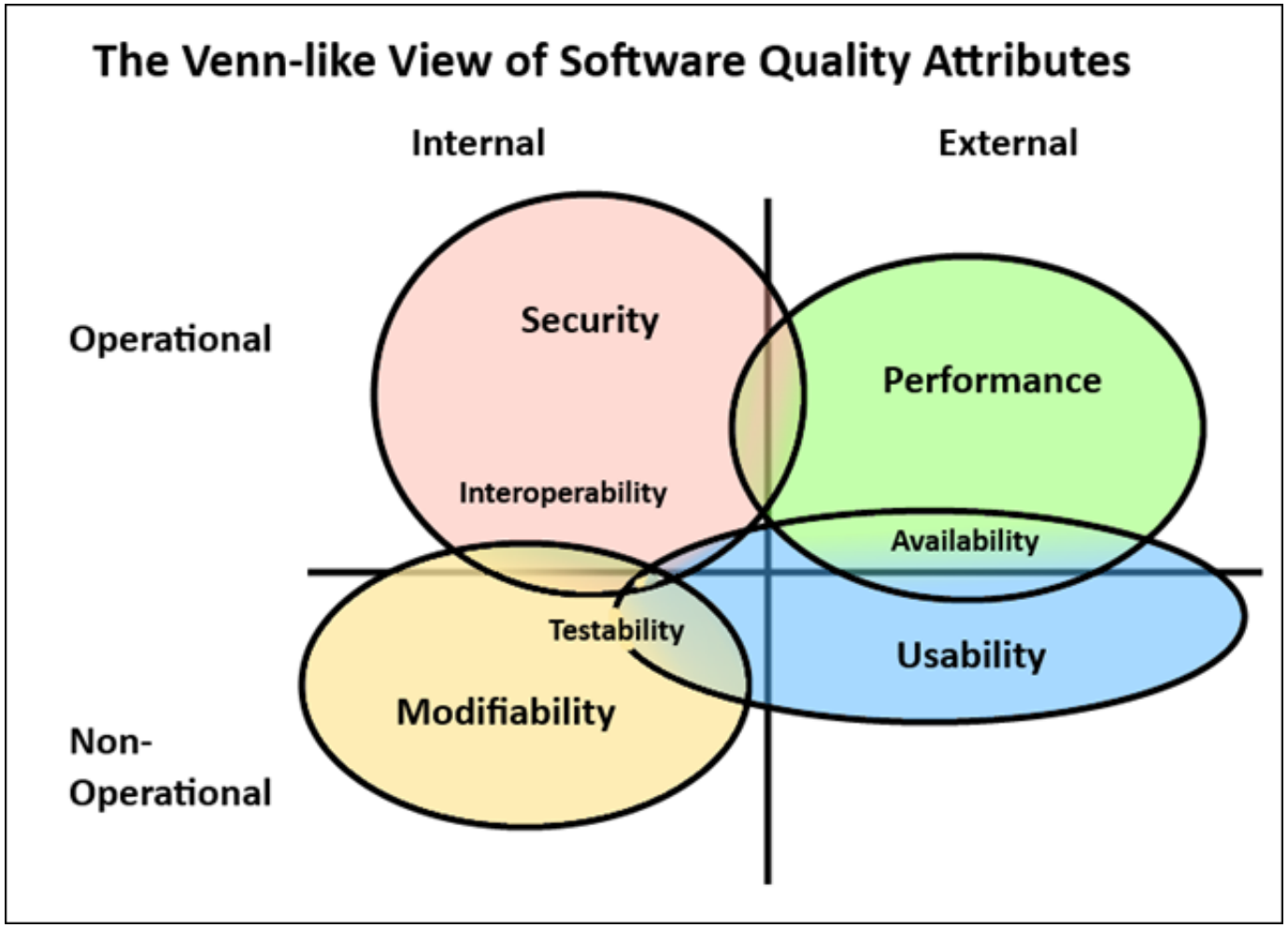
Week 5 Objectives

- **SW Quality Attributes**
  - What is SW Quality Attributes
  - ISO 9126-1 SW Architecture Standards
- **Further SW Quality Attributes**
  - Modifiability
  - Security
  - Availability
  - Integration
  - Other Qualities
    - Portability
    - Testability
    - Supportability
- **Design Trade-Offs**

- **Assignment 6: Component Class Design**
  - We are looking at the use of Design Patterns in design; most specifically, in the Component design portion of a SW project. This is the phase in which we decide what classes and arrangement of classes will fill a particular need, once the large functional blocks have been determined.
  - Submit a diagram showing your class design. Explain the use case scenario at each level
  - Upload your work in word document via blackboard >> EGR326 >> Assignment >> **Asn 6:** Component Design
  - **Due date**: Friday, Feb 14 by 10:00pm

- **Software:**
  - Lucichart: webapp
  - Draw.io: webapp

- ✓ Software Quality Attributes can be looked at in terms of Operational/Non-operational and Internal/External

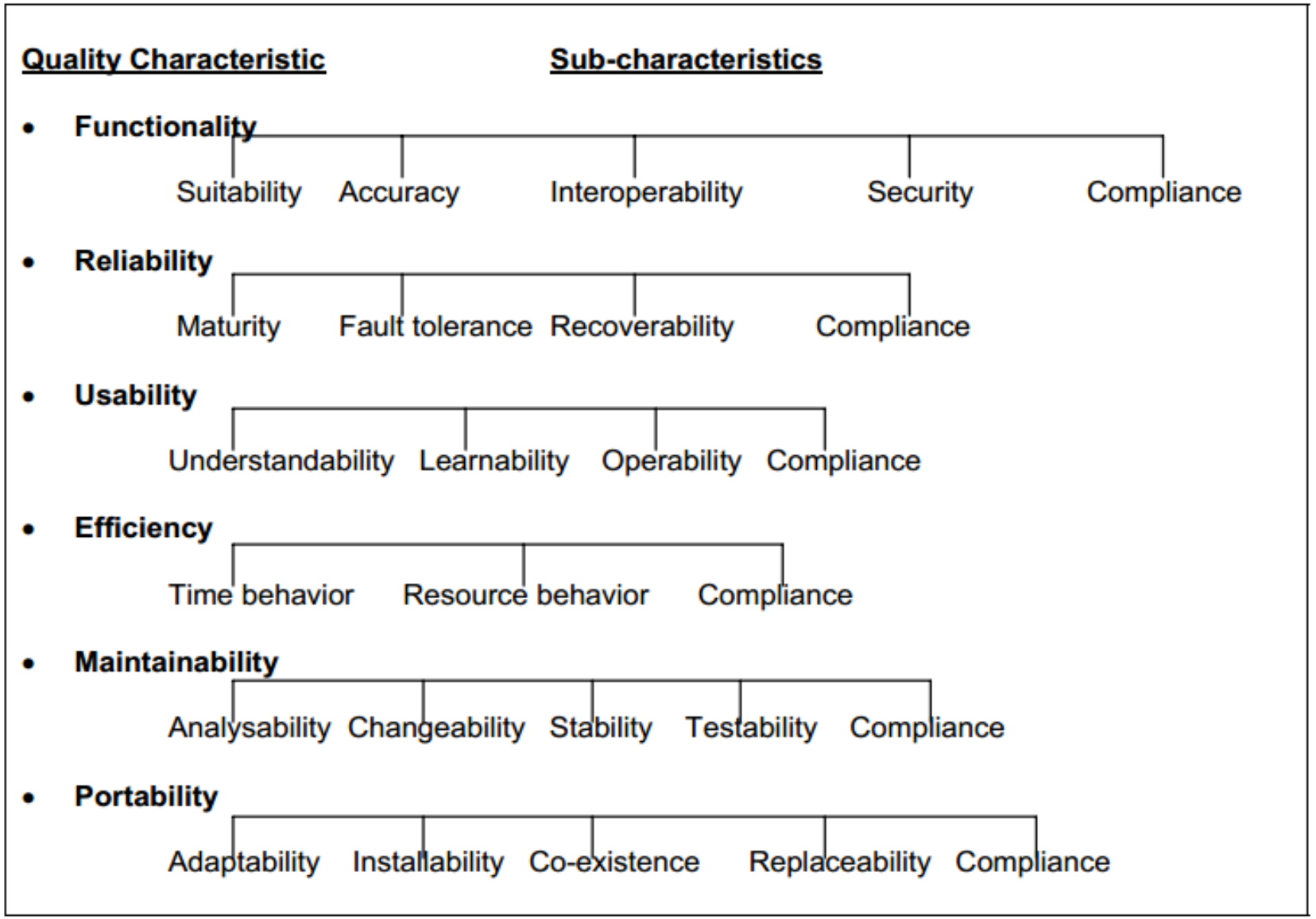


✓ The ISO 9126-1 standard lists six characteristics to a quality SW architecture

Characteristics	Description
Functionality	The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions (what the software does to fulfil needs)
Reliability	The capability of the software product to maintain its level of performance under stated conditions for a stated period of time
Usability	The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions (the effort needed for use)
Efficiency	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in the environment and in the requirements and functional specifications (the effort needed to be modified)
Portability	The capability of the software product to be transferred from one environment to another. The environment may include organizational, hardware or software environment

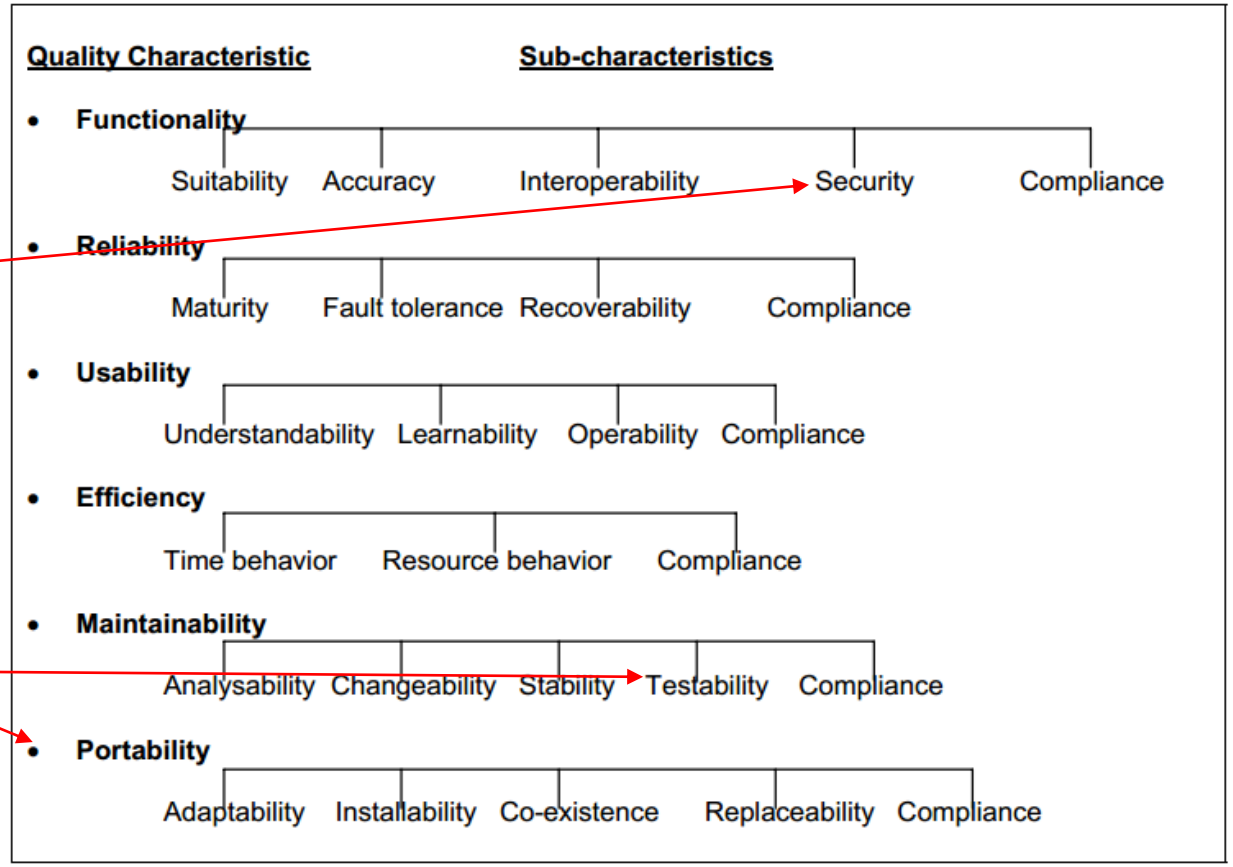
✓ The ISO 9126-1 standard lists six characteristics to a quality SW architecture

✓ In the ISO 9126-1 model, each characteristic has a number of sub-characteristics



✓ Gorton uses a slightly different list that embodies the same overall considerations

- Performance
- Scalability
- Modifiability
- Security
- Availability
- Integration
- Portability
- Testability
- Supportability



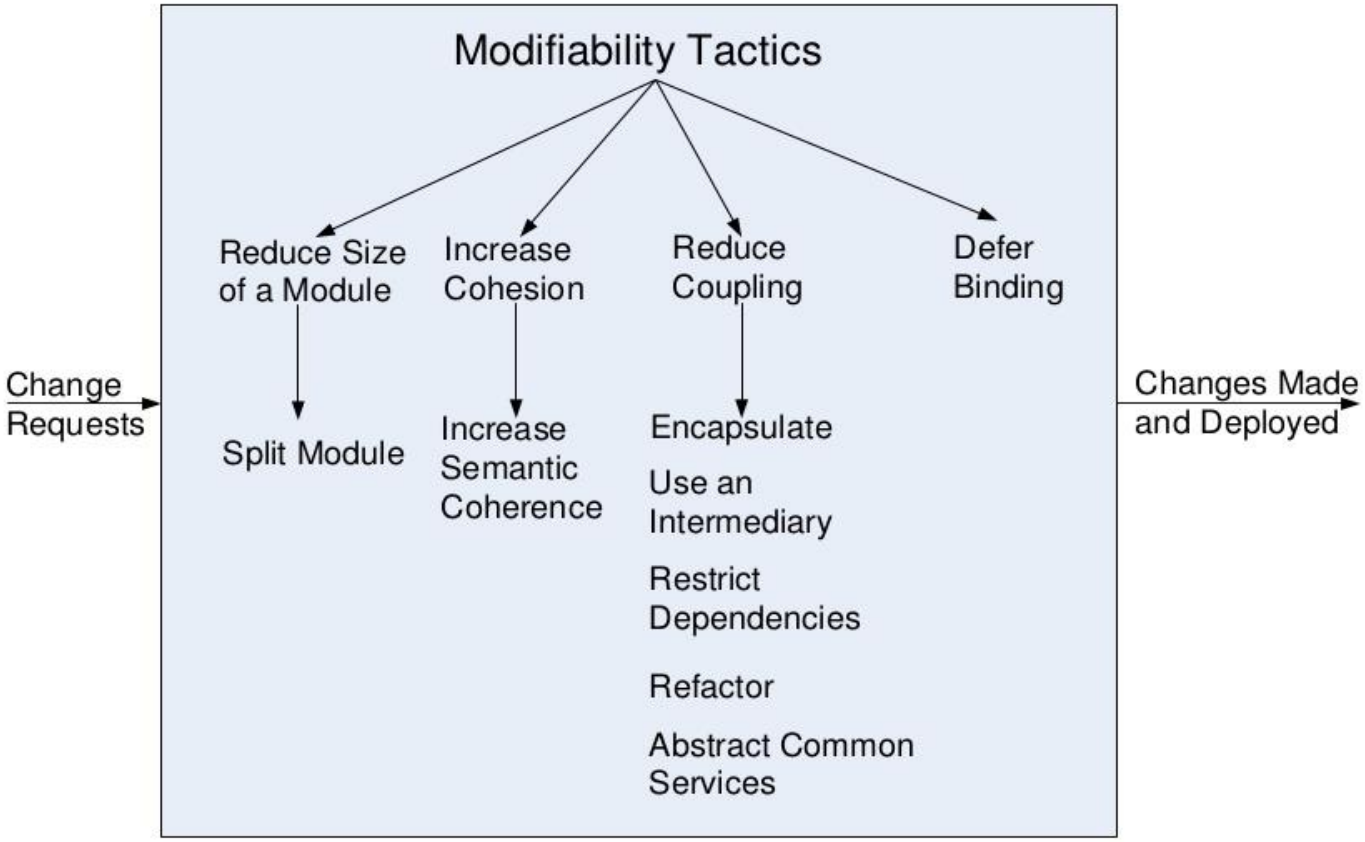
ISO 9126-1 SW Architecture Standards

✓ How "easy" will it be to change the final, working product to meet new functional or non-functional requirements

- Of course, we can't know future needs, so this is a guess at best

- But there are areas of a SW design that are most likely to need modification:
  - Interfaces
  - Hardware platform
  - New workflows
  - UI

Modifiability Tactics



✓ Tough to predict modification requirements for the Information Capture and Dissemination Environment (ICDE)

- Perhaps additional events will need to be captured and stored (beyond queries and responses)
  - would impact both client and server
- Perhaps new third-party tools will require additional messages to function
  - server messaging protocol
- The general strategies should meet these needs for ICDE
  - Smaller modules
  - High cohesion
  - Low coupling
  - Deferred binding

- ✓ **Sometimes, existing applications need to work with added levels of, and mechanisms for, security**

  - SSL (Secure Sockets Layer)
  - PKI (Public Key Infrastructure)
  - JAAS (Java Authentication and Authorization Service)

  - Use existing standard methods - ICDE

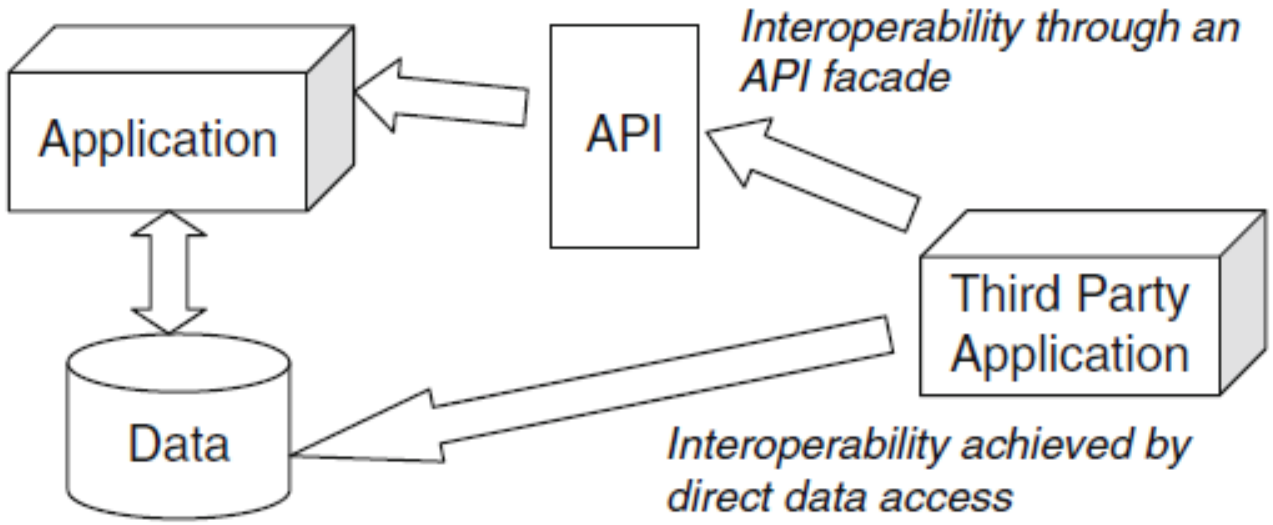
  - **In the ICDE system, use standard packages for**
    - User authentication
    - Third-party tool authentication
    - Data encryption

✓ Availability is a measure of system reliability (uptime)

- The availability of the system itself is largely dependent on the platform
  - Hardware
  - OS and components
- The system SW should enable maintenance operations to run in compliance with availability requirements
  - Support hot-swap of HW components if need be
  - Maintenance operations while online
- The ICDE system is used during business hours, allowing maintenance to be done after-hours

✓ Integration refers to the system's ability to be used as part of a larger process

- Specifications related to integration often include –
  - consistency of the UI and workflow
  - exposing a complete API
  - data formats (XML?)



✓ Extensible Markup Language (XML) is a way to define tags for value fields in a text document

- **Defined by a specification** (http://www.w3.org/TR/REC-xml/)
- **APIs to support it (read/write) exist** - http://docs.oracle.com/javase/tutorial/jaxp/sax/

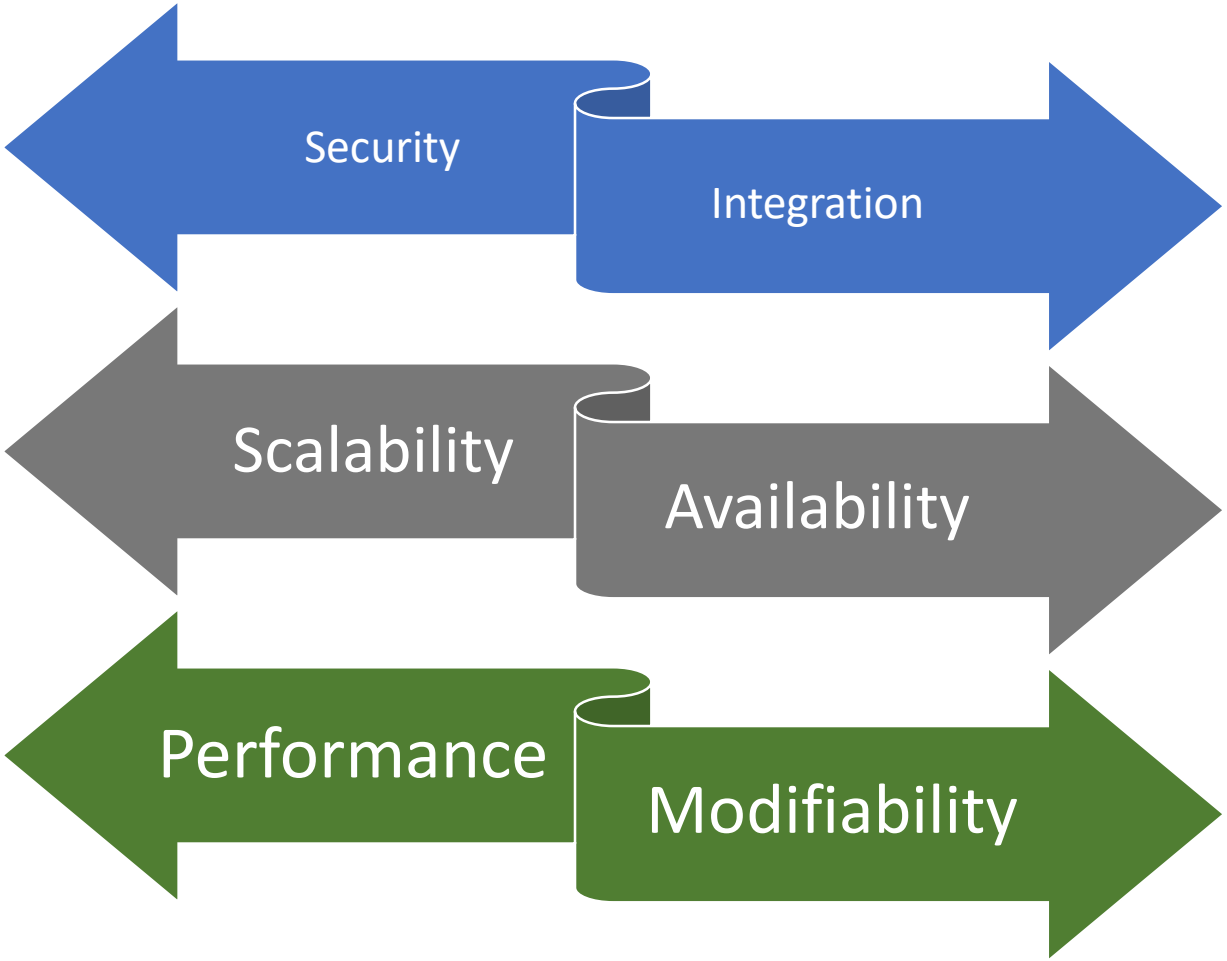
```
<CATALOG>
<CD>
<TITLE>Empire Burlesque</TITLE>
<ARTIST>Bob Dylan</ARTIST>
<COUNTRY>USA</COUNTRY>
<COMPANY>Columbia</COMPANY>
<PRICE>10.90</PRICE>
<YEAR>1985</YEAR>
</CD>
<CD>
<TITLE>Hide your heart</TITLE>
<ARTIST>Bonnie Tylor</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>CBS Records</COMPANY>
<PRICE>9.90</PRICE>
<YEAR>1988</YEAR>
</CD>
```

✓ Use with third-party tools is the major requirement for integration of the ICDE system

- Assuming that these tools will access the data store:
  - What data formats will be exposed?
  - What are the protocols for access (data protection)?
  - Will direct access to storage be provided? (Nah, probably not)
  - Encryption?

- **Portability**
  - How difficult would it be to change to another platform (HW and/or SW)?
  - Isolate dependencies on platform-specific services
- **Testability**
  - Code reuse helps this
- **Supportability**
  - Built-in diagnostics

- ✓ There are typical engineering Trade-Offs involved in many of these quality metrics



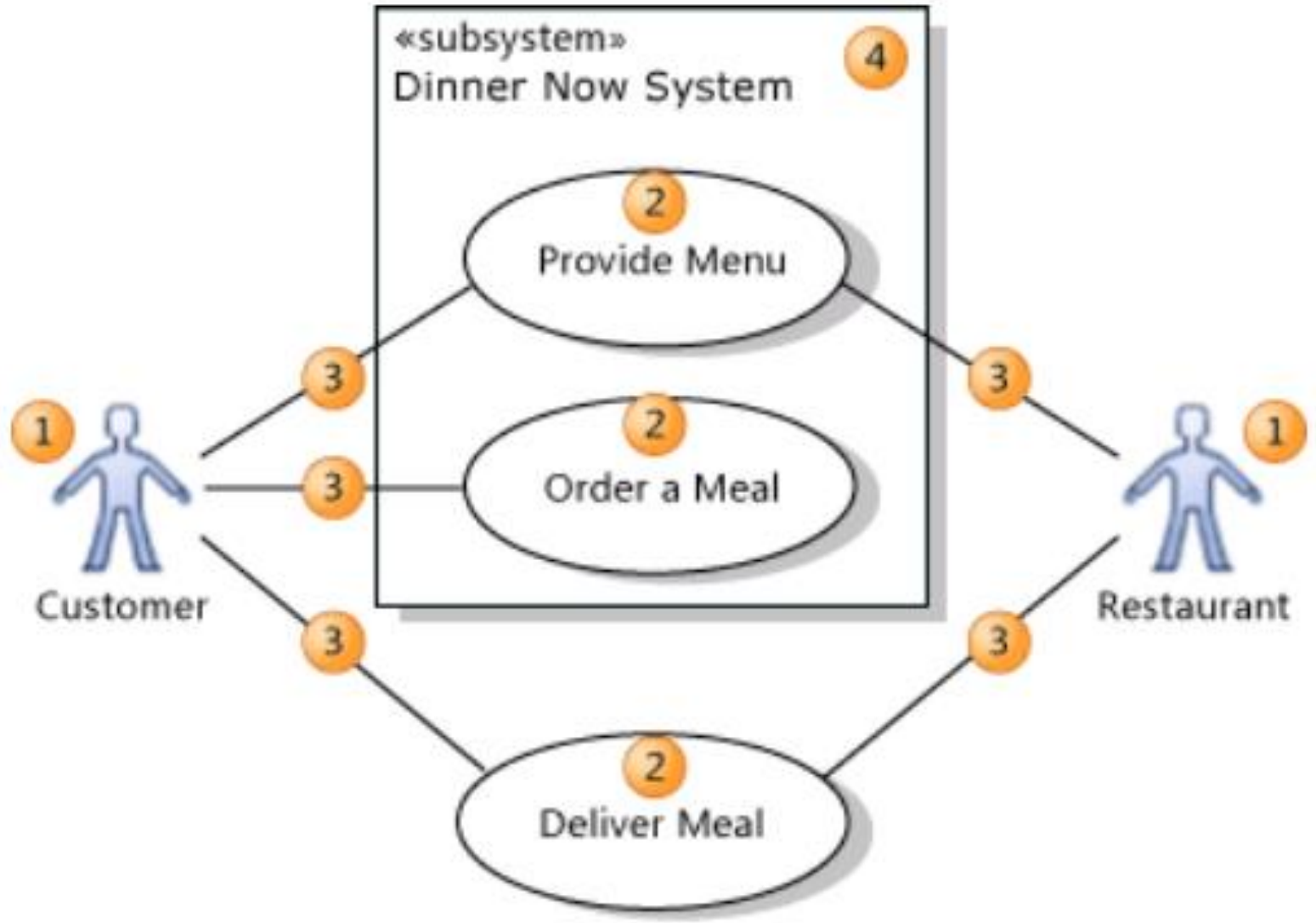
Review: Week 5 Objectives

- **SW Quality Attributes**
  - What is SW Quality Attributes
  - ISO 9126-1 SW Architecture Standards
- **Further SW Quality Attributes**
  - Modifiability
  - Security
  - Availability
  - Integration
  - Other Qualities
    - Portability
    - Testability
    - Supportability
- **Design Trade-Offs**

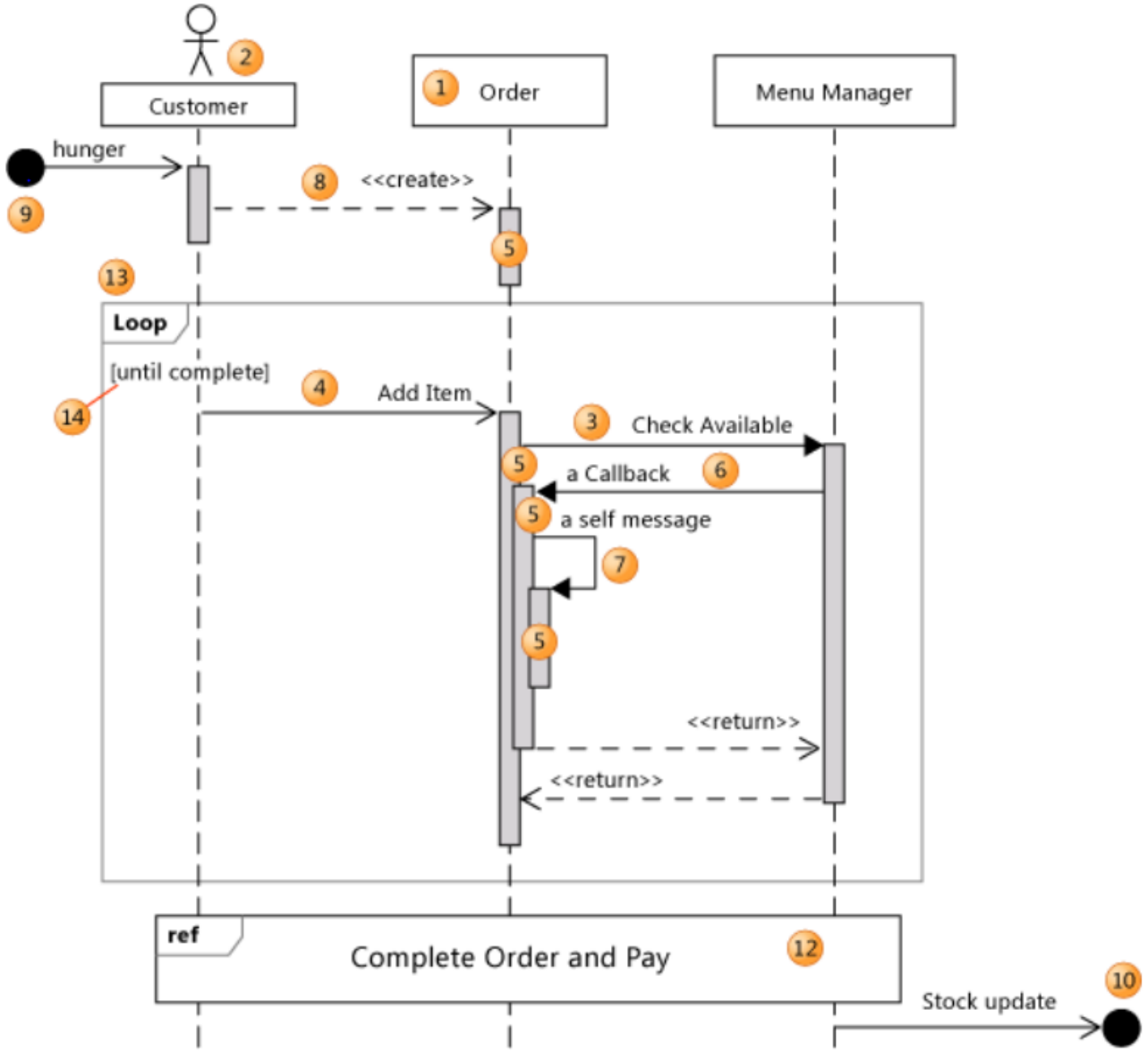
- **Assignment 6: Component Class Design**
  - We are looking at the use of Design Patterns in design; most specifically, in the Component design portion of a SW project. This is the phase in which we decide what classes and arrangement of classes will fill a particular need, once the large functional blocks have been determined.
  - Submit a diagram showing your class design. Explain the use case scenario at each level
  - Upload your work in word document via blackboard >> EGR326 >> Assignment >> **Asn 6:** Component Design
  - **Due date**: Friday, Feb 14 by 10:00pm

- **Software:**
  - Lucichart: webapp
  - Draw.io: webapp

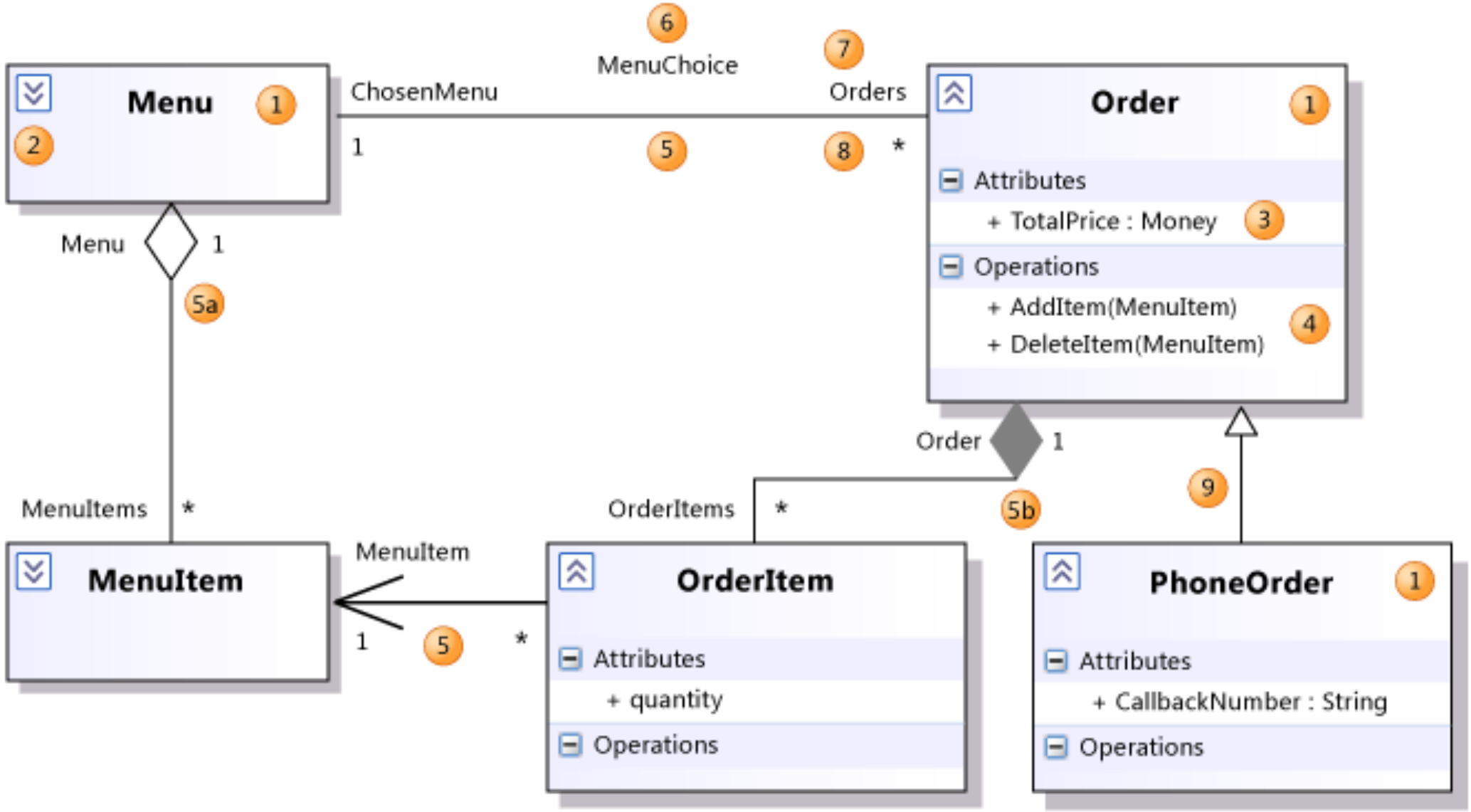
Use Case Diagram



Sequence Diagram

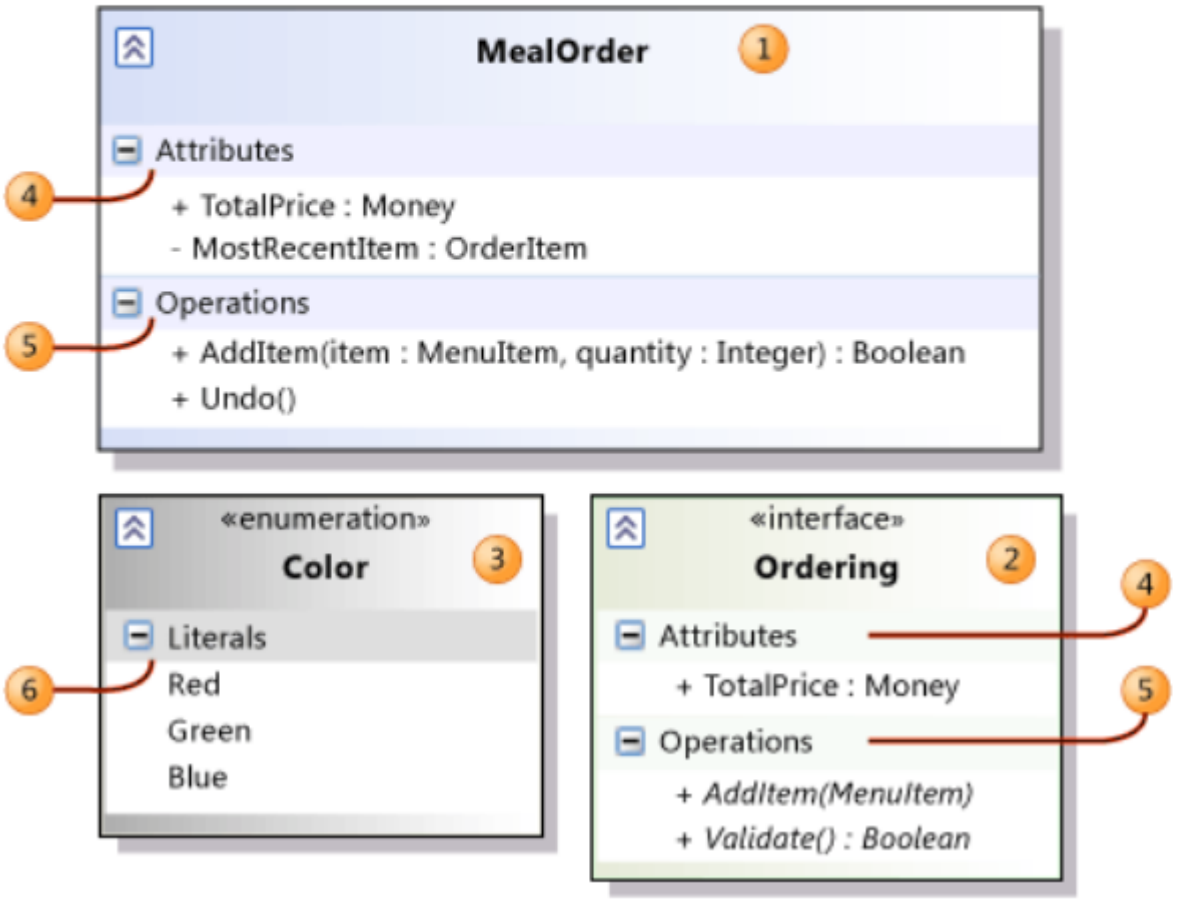


Class Diagram



Using Classes, Interfaces, and Enumerations

There are three standard kinds of classifier available on the toolbox. These are referred to as types throughout this document.



EGR326 Software Design and Architecture

Lecture 7. Introduction to Architecture

Spring 2020

Kim Peters, Ph.D.

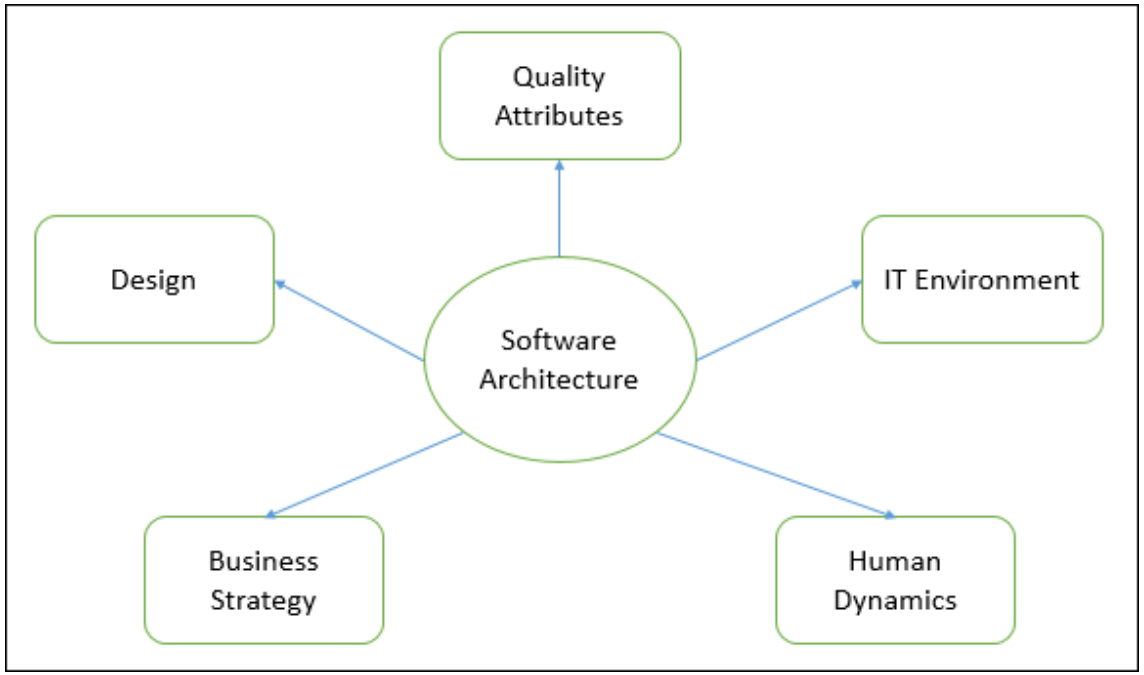
Gordon and Jill Bourns College of Engineering
California Baptist University

Week 4 Objectives

- **Software Architecture**
  - What is Software Architecture
  - Why Software Architecture?
  - What Software Architecture Do?
- Relationship to Non-functional requirements
- Views of SW Architecture
  - SW Architecture Views
  - 4+1 Model
  - Views and Beyond
- Challenges
- Software Architect Roles

- **Assignment 5: Software Architecture**
  - Develop the high-mid level of your project architecture
  - Develop your software components accordingly
  - Examine your system requirements and explain your use case scenario at each level
  - Upload your work in word document via blackboard >> EGR326 >> Assignment >> **Asn 5: Software Architecture**
  - **Due date**: Friday, Feb 7 by 10:00pm

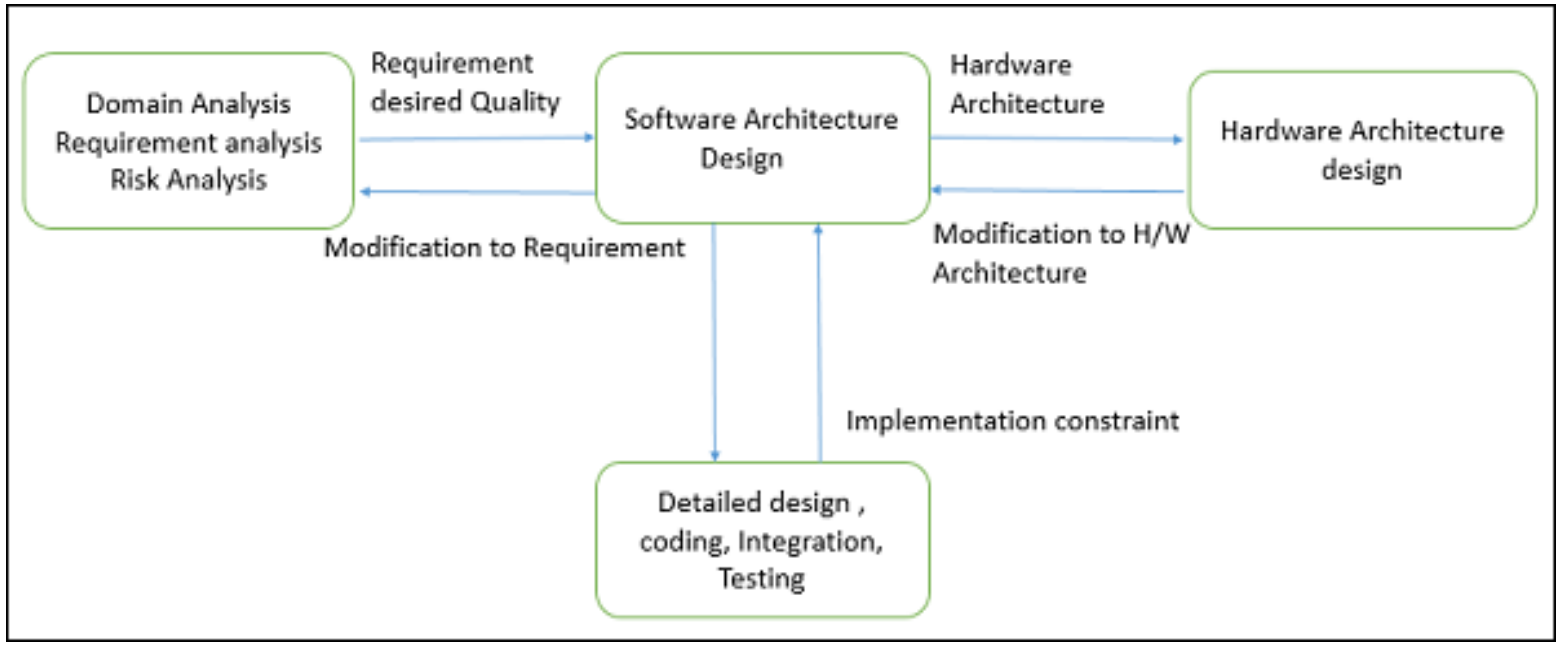
✓ Software **architecture** and **design** includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



- We can segregate Software Architecture and Design into two distinct phases**: Software Architecture and Software Design.**
  - In **Architecture**, non-functional decisions are cast and separated by the functional requirements.
  - In **Design**, functional requirements are accomplished.

✓ Provides a design plan that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system.

- Guide the implementation tasks, including detailed design, coding, integration, and testing.
- It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.



https://www.tutorialspoint.com/software_architecture_design/introduction.htm

✓ Software Architecture serves as a **blueprint** for a system

- It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

- **Goal**:
  - To identify requirements that affect the structure of the application.

✓ A well-laid architecture
  - reduces the business risks associated with building a technical solution and
  - builds a bridge between business and technical requirements.

https://www.tutorialspoint.com/software_architecture_design/introduction.htm

✓ Software Architecture has to do with the entire system, thinking in large pieces

- **Architecture** is defined by the recommended practice as the fundamental organization of a system,
  - embodied in its components,
  - their relationships to each other and the environment, and
  - the principles governing its design and evolution.

- IEEE

- The **software architecture** of a program or computing system is the structure or structures of the system, which comprise
  - software elements,
  - the externally visible properties of those elements, and the
  - relationships among them.

- Bass and Clements, 2003

✓ Architecture Defines Structure

- **Architecture** is defined by the recommended practice as the fundamental organization of a system,
  - embodied in its components,
  - their relationships to each other and the environment, and
  - the principles governing its design and evolution.

- IEEE

- The **software architecture** of a program or computing system is the structure or structures of the system, which comprise
  - software elements,
  - the externally visible properties of those elements, and the
  - relationships among them.

- Bass and Clements, 2003

✓ Components are distinct software subsystems (one or more classes, libraries, etc…)

- **Architecture** is defined by the recommended practice as the fundamental organization of a system,
  - embodied in its components,
  - their relationships to each other and the environment, and
  - the principles governing its design and evolution.

- IEEE

- The **software architecture** of a program or computing system is the structure or structures of the system, which comprise
  - software elements,
  - the externally visible properties of those elements, and the
  - relationships among them.

- Bass and Clements, 2003

✓ **Key point**: Choosing an Architecture for a problem determines the relationships (collaborations) among the components

- **Architecture** is defined by the recommended practice as the fundamental organization of a system,
  - embodied in its components,
  - their relationships to each other and the environment, and
  - the principles governing its design and evolution.

- IEEE

- The **software architecture** of a program or computing system is the structure or structures of the system, which comprise
  - software elements,
  - the externally visible properties of those elements, and
  - the relationships among them.

- Bass and Clements, 2003

✓ The Architecture determines how the system talks to its surroundings (user, network, etc.)

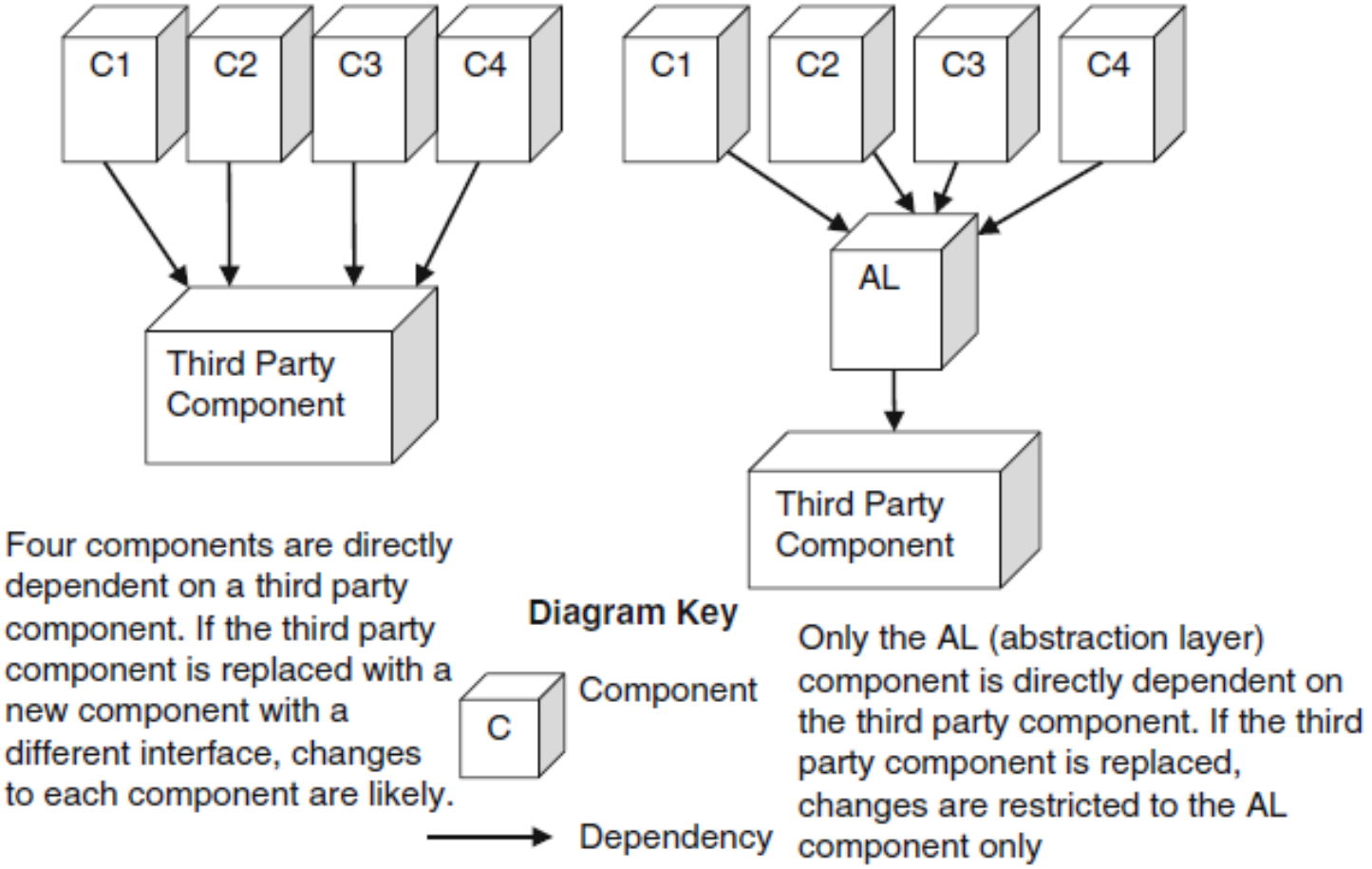
- **Architecture** is defined by the recommended practice as the fundamental organization of a system,
  - embodied in its components,
  - their relationships to each other and the environment, and
  - the principles governing its design and evolution.

- IEEE

- The **software architecture** of a program or computing system is the structure or structures of the system, which comprise
  - software elements,
  - the externally visible properties of those elements, and
  - the relationships among them.

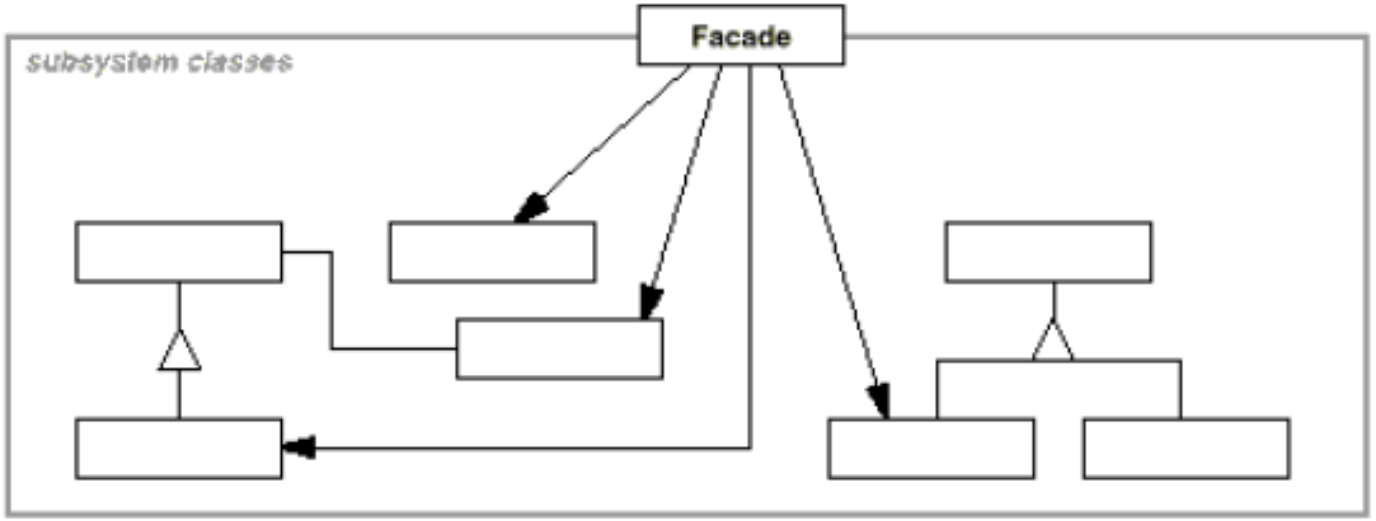
- Bass and Clements, 2003

- ✓ Architectural choices can determine adherence to good SW practices like, in this case, loose coupling (low dependency)



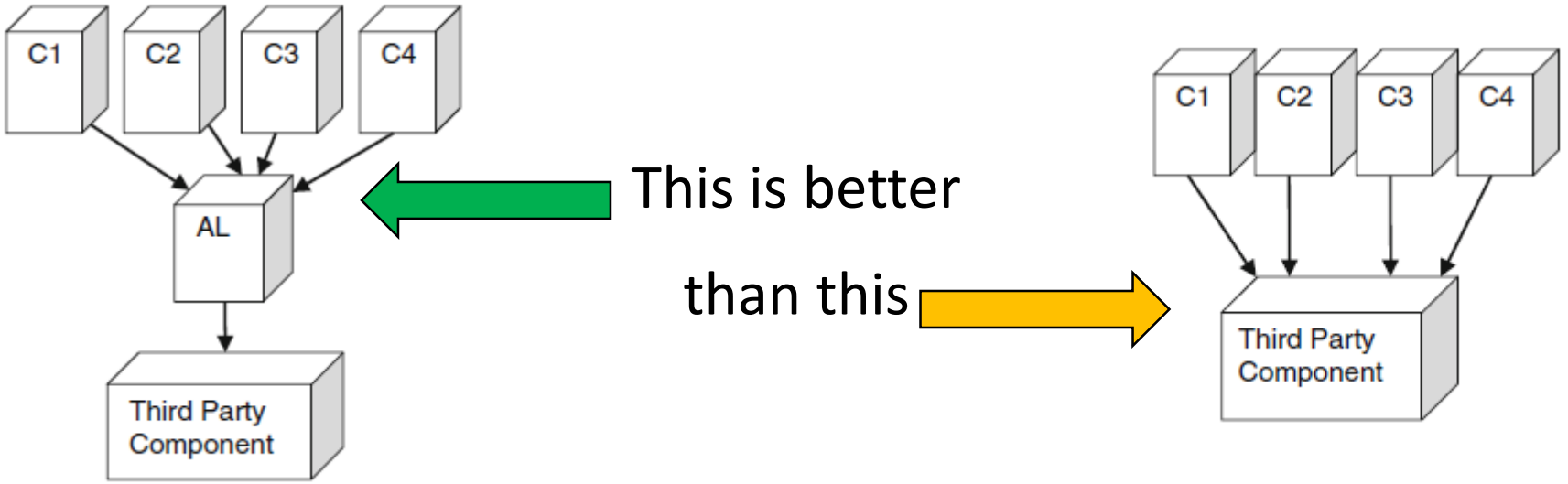
✓ This can be accomplished by use of the Facade pattern

- Facade implements an abstraction layer



		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

- ✓ **Another key point: Architectural decisions are not just personal taste; some are better than others**



- It depends on the context (the requirements, existing code, system considerations, etc)
- Some architectures are better at times, worse at other times
- Some are always terrible

- ✓ **SW Non-functional requirements are those that don't appear in test cases; not *what* it does, but conditions on *how* it does it**
  - **Three major kinds of non-functional requirements:**
    - **Technical constraints:**
      - "We will be developing in Java“
      - "The system has 8GB of RAM“
      - these are usually non-negotiable
    - **Business constraints**
      - "We must interface with MATLAB“
      - "We must use open-source code for the backend"
      - also usually non-negotiable
    - **Quality attributes**
      - Portability
      - Scalability
      - Performance
      - …

Relationship to Non-functional Requirements

✓ The Architecture largely determines our compliance with Non-functional requirements

- **Three Major Kinds of Non-functional Requirements:**
  - **Technical Constraints:**
    - "We will be developing in Java“
    - "The system has 8GB of RAM“
    - these are usually non-negotiable

    Component Selection
  - **Business Constraints**
    - "We must interface with MATLAB“ **External Interface**
    - "We must use open-source code for the backend"
    - also usually non-negotiable
  - **Quality Attributes**
    - Portability
    - Scalability
    - Performance
    - …

    Component Selection

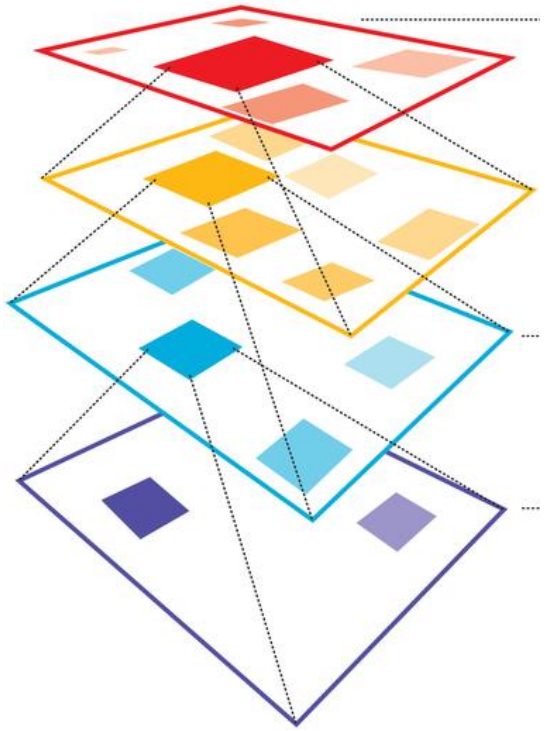
    Interrelationships between Components

Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

- IEEE

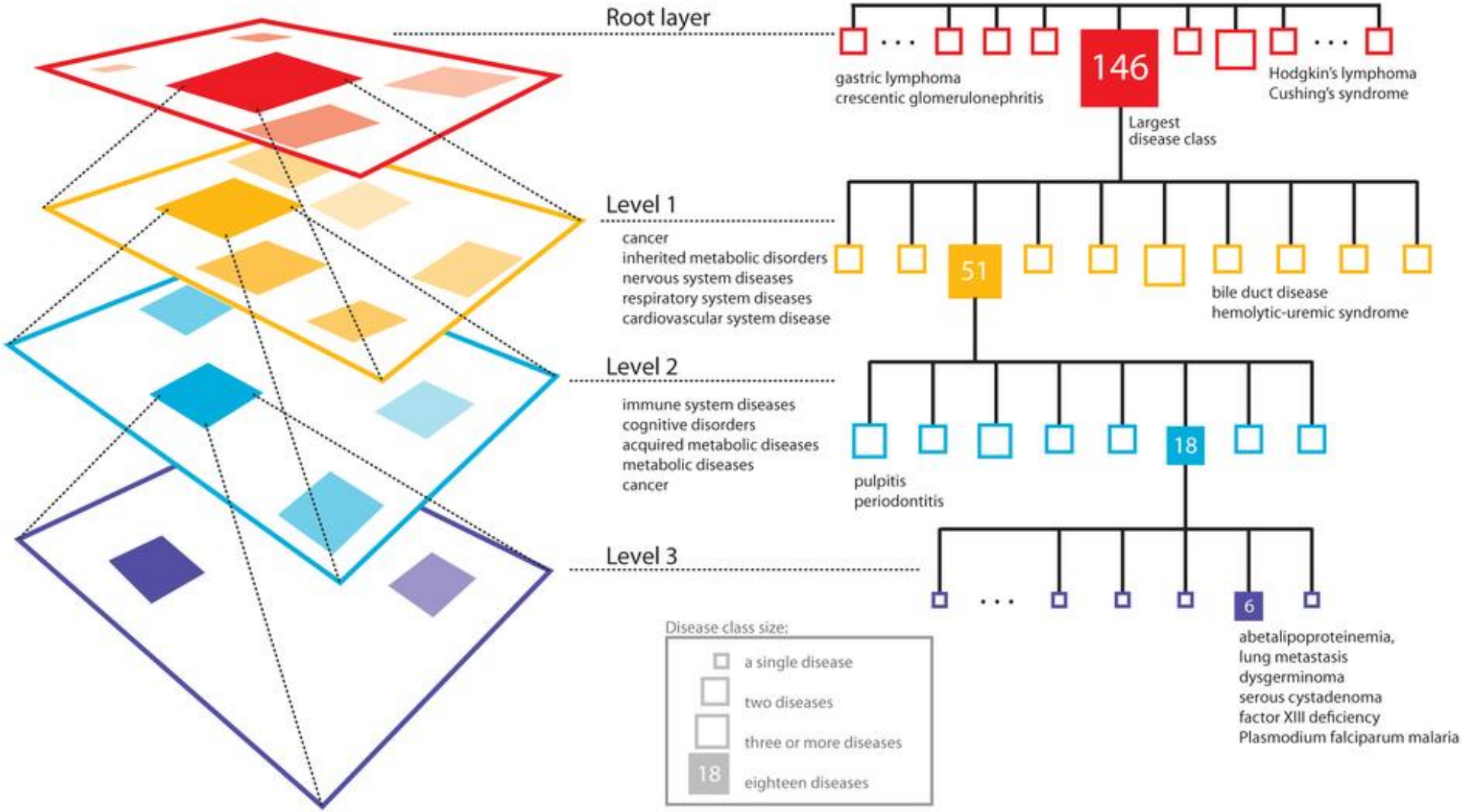
✓ SW Architecture is an *Abstraction*

- There is a highest-level description in terms of large pieces
- Each piece can be described in terms of smaller pieces
- This is called Hierarchical Decomposition



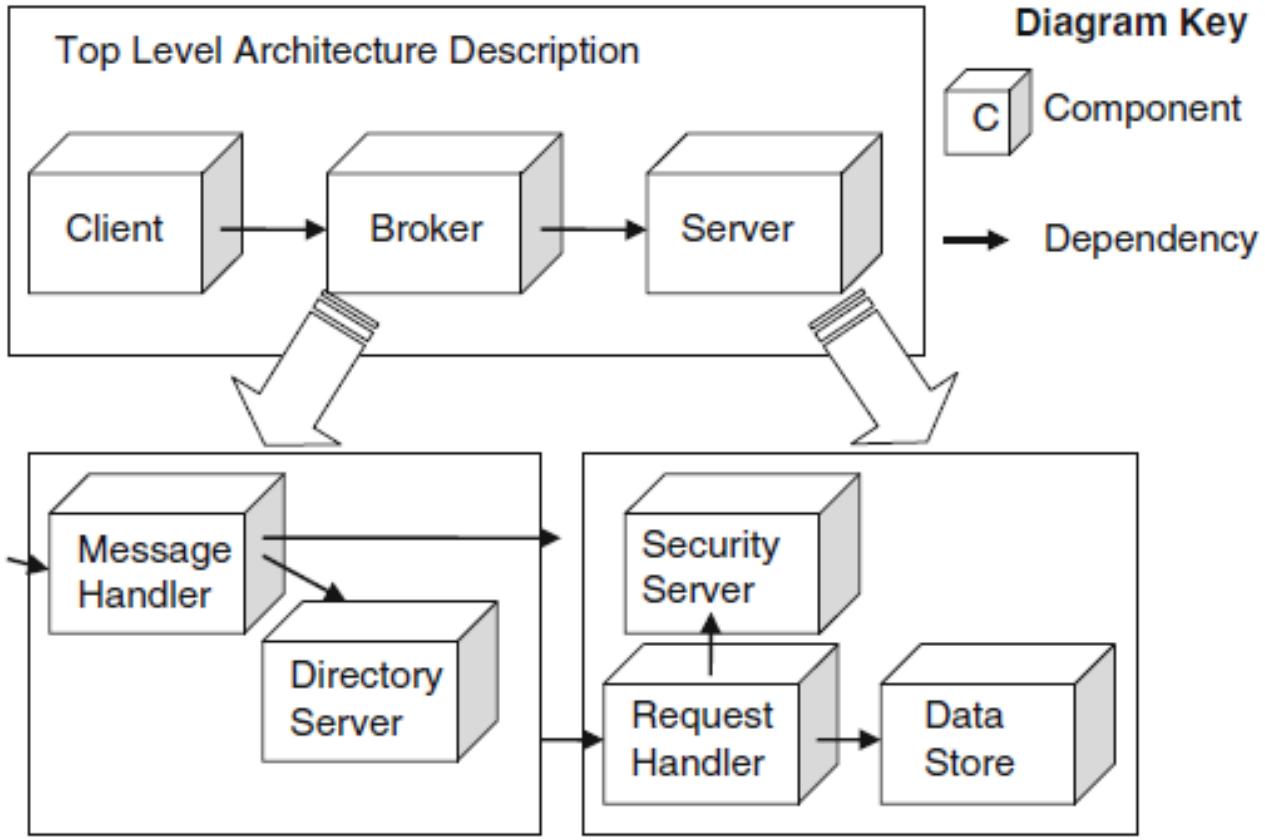
✓ SW Architecture is an Abstraction

- Hierarchical Decomposition



✓ An example of a two-layer hierarchical decomposition of a SW architecture

- Usually we design the top level first
- Then each block becomes a design at the next level



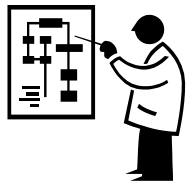
- ✓ **There are four common "views" in which to describe an architecture: Logical, Process, Physical and Development**

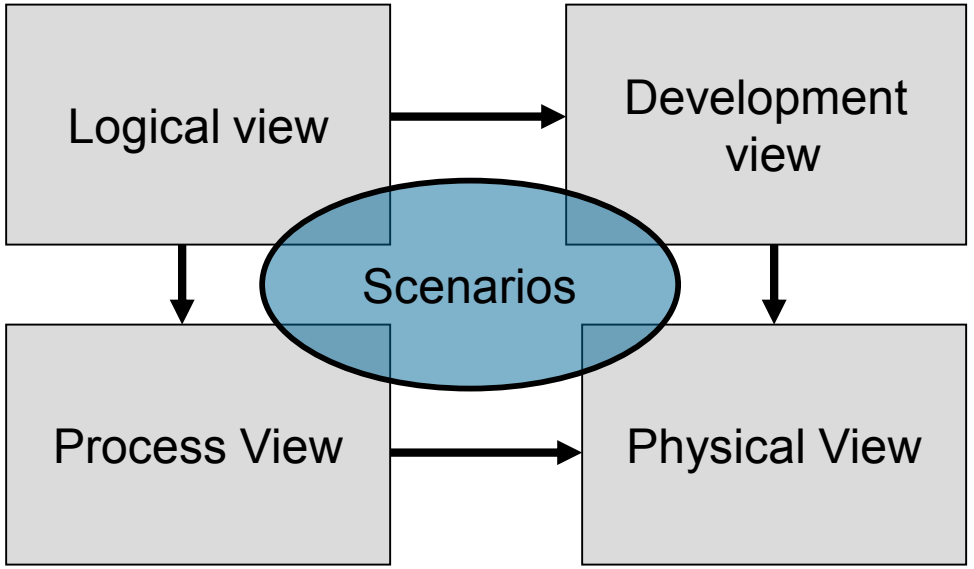
1) **Logical View**
   - A block diagram of the components
2) **Process View**
   - Shows the order of processing
3) **Physical View**
   - Concerned with how the components execute on the hardware
4) **Development View**
   - The description of project and solutions, or packages, that the system SW is developed and maintained in

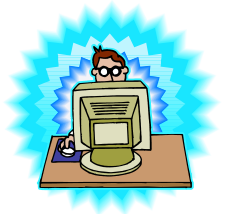
✓**This is called the 4+1 Architectural View Model**

✓ The "4+1" model portrays how different roles can see the same SW in very different ways

End user







Programmers & software managers

System Engineer

✓ Another useful set of ways to look at architecture is called the "Views and Beyond" approach

- **Three Views:**

a) Module

- The structural layout of the components – but with a little more detail (classes, inheritance, etc)

b) Component and Connector

- Identifies components that need to talk and the mechanisms for doing so (CORBA, sockets, etc.)

c) Allocation

- Communications, hardware and development responsibilities

✓ Another useful set of ways to look at architecture is called the "Views and Beyond" approach

- **Three Views:**
  - **a) Module**
    - The structural layout of the components – but with a little more detail (classes, inheritance, etc)

    We will call this the "Structural View"

  - **b) Component and Connector**
    - Identifies components that need to talk and the mechanisms for doing so (CORBA, sockets, etc.)

    We will call this the "Behavioral View"

  - **c) Allocation**
    - Communications, hardware and development responsibilities

✓ Another useful set of ways to look at architecture is called the "Views and Beyond" approach

- **Three Views:**

a) Module

- The structural layout of the components – but with a little more detail (classes, inheritance, etc)

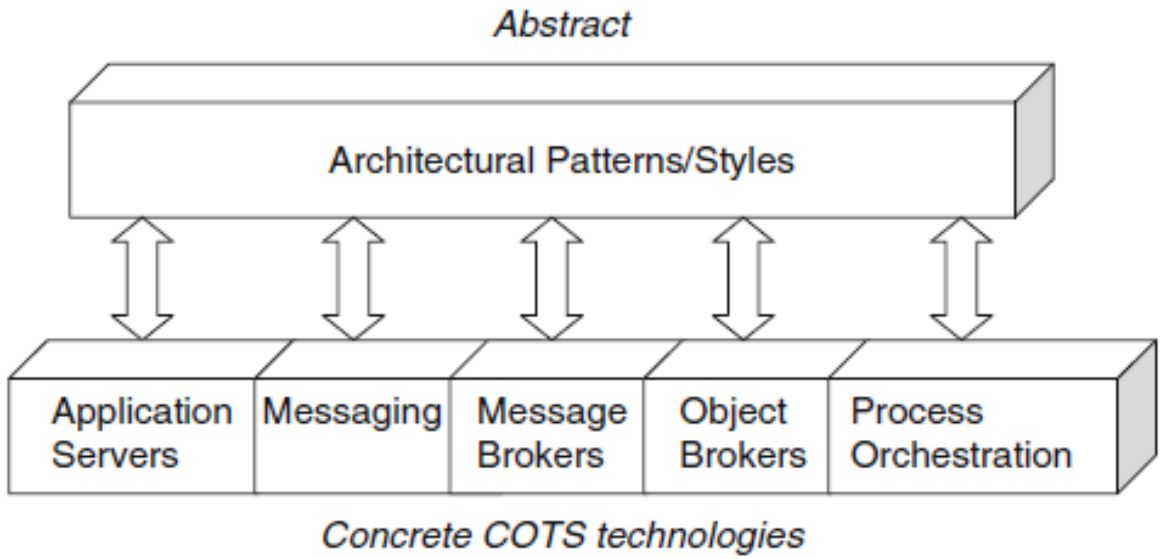
b) Component and Connector

- Identifies components that need to talk and the mechanisms for doing so (CORBA, sockets, etc.)

c) Allocation

- Communications, hardware and development responsibilities

- ✓ **There are many tested SW architectures, design patterns and SW packages that implement or facilitate architectures, available to us**
  - Many are Commercial Off-the-Shelf (COTS) components
    - These days, this term is also used to include open-source



  - Our challenge is to understand these and make key decisions early on …

- ✓ **In some organizations, Architects are specific people – in others, developers do a little or a lot of the architecture work**
  - **Chief Architect:**
    - Typically a senior position who manages a team of architects within an organization.
    - Operates at a broad, often organizational level, and coordinates efforts across system, applications, and product lines.
    - Very experienced, with a rare combination of deep technical and business knowledge.
  - **Product/Technical/Solution Architect:**
    - Typically someone who has progressed through the technical ranks and oversees the architectural design for a specific system or application.
    - They have a deep knowledge of how some important piece of software really works.
  - **Enterprise Architect:**
    - Typically a much less technical, more business-focus role.
    - Enterprise architects use various business methods and tools to understand, document, and plan the structure of the major systems in an enterprise.

- ✓ **In some organizations, Architects are specific people – in others, developers do a little or a lot of the architecture work**
  - **Chief Architect:**
    - Typically a senior position who manages a team of architects within an organization.
    - Operates at a broad, often organizational level, and coordinates efforts across system, applications, and product lines.
    - Very experienced, with a rare combination of deep technical and business knowledge.
  - **Product/Technical/Solution Architect:**
    - Typically someone who has progressed through the technical ranks and oversees the architectural design for a specific system or application.
    - They have a deep knowledge of how some important piece of software really works.
  - **Enterprise Architect:**
    - Typically a much less technical, more business-focus role.
    - Enterprise architects use various business methods and tools to understand, document, and plan the structure of the major systems in an enterprise.

Review: Week 4 Objectives

- **Software Architecture**
  - What is Software Architecture
  - Why Software Architecture?
  - What Software Architecture Do?
- Relationship to Non-functional requirements
- Views of SW Architecture
  - SW Architecture Views
  - 4+1 Model
  - Views and Beyond
- Challenges
- Software Architect Roles

Reminder: To-Do This Week

- **Assignment 5: Software Architecture**
  - Develop the high-mid level of your project architecture
  - Develop your software components accordingly
  - Examine your system requirements and explain your use case scenario at each level
  - Upload your work in word document via blackboard >> EGR326 >> Assignment >> **Asn 5: Software Architecture**
  - **Due date**: Friday, Feb 7 by 10:00pm

EGR326 Software Design and Architecture

Lecture 8. Performance and Scalability

Spring 2020

Kim Peters, Ph.D.

Gordon and Jill Bourns College of Engineering
California Baptist University

Week 4 Objectives

- **Software Architecture**
  - What is Software Architecture
  - Why Software Architecture?
  - What Software Architecture Do?
- Relationship to Non-functional requirements
- Views of SW Architecture
  - SW Architecture Views
  - 4+1 Model
  - Views and Beyond
- Challenges
- Software Architect Roles

- **Performance and Scalability**
- **The Textbook Case Study**
  - The ICDE system
  - Project Context
  - Business Goals
  - Constraints
- **Software Quality Attributes**
  - Performance
  - Scalability

- **Assignment 5: Software Architecture**
  - Develop a high-mid level of your project architecture
  - Develop your software components accordingly
  - Examine your system requirements and explain the use case scenario at each level
  - Upload your work in word document via blackboard >> EGR326 >> Assignment >> **Asn 5: Software Architecture**
  - **Due date**: Friday, Feb 7 by 10:00pm

✓ **The author's case study example, called the ICDE, is centered on a database that stores web queries and their results.**

- Multiple users can share data
- The target user group are analysts –
  - Financial analysts
  - Scientific researchers
  - Intelligence analysts

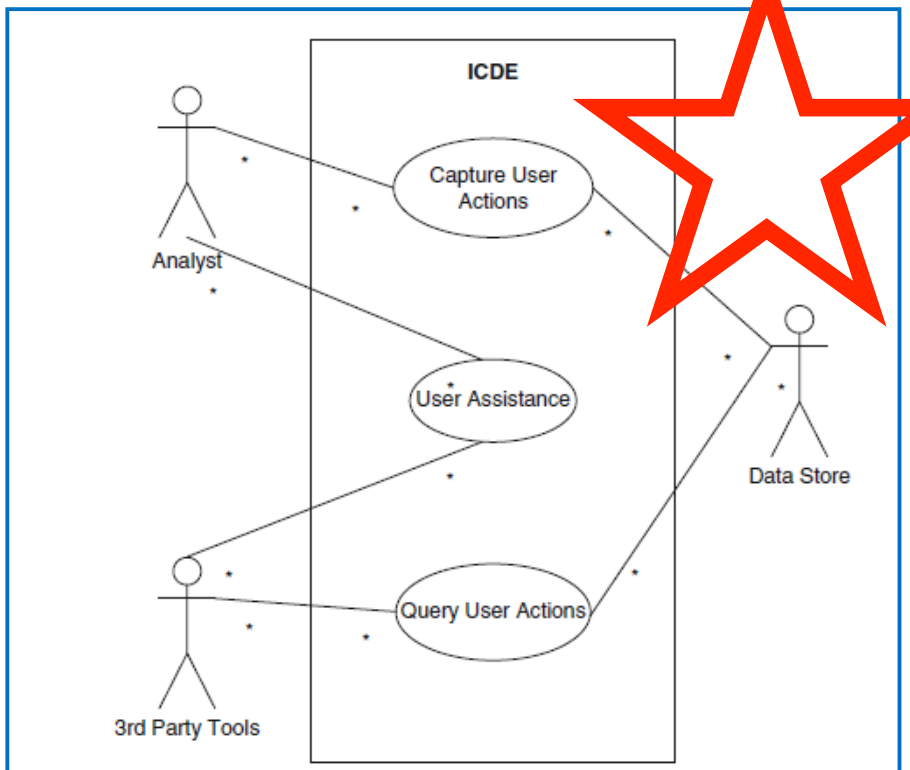
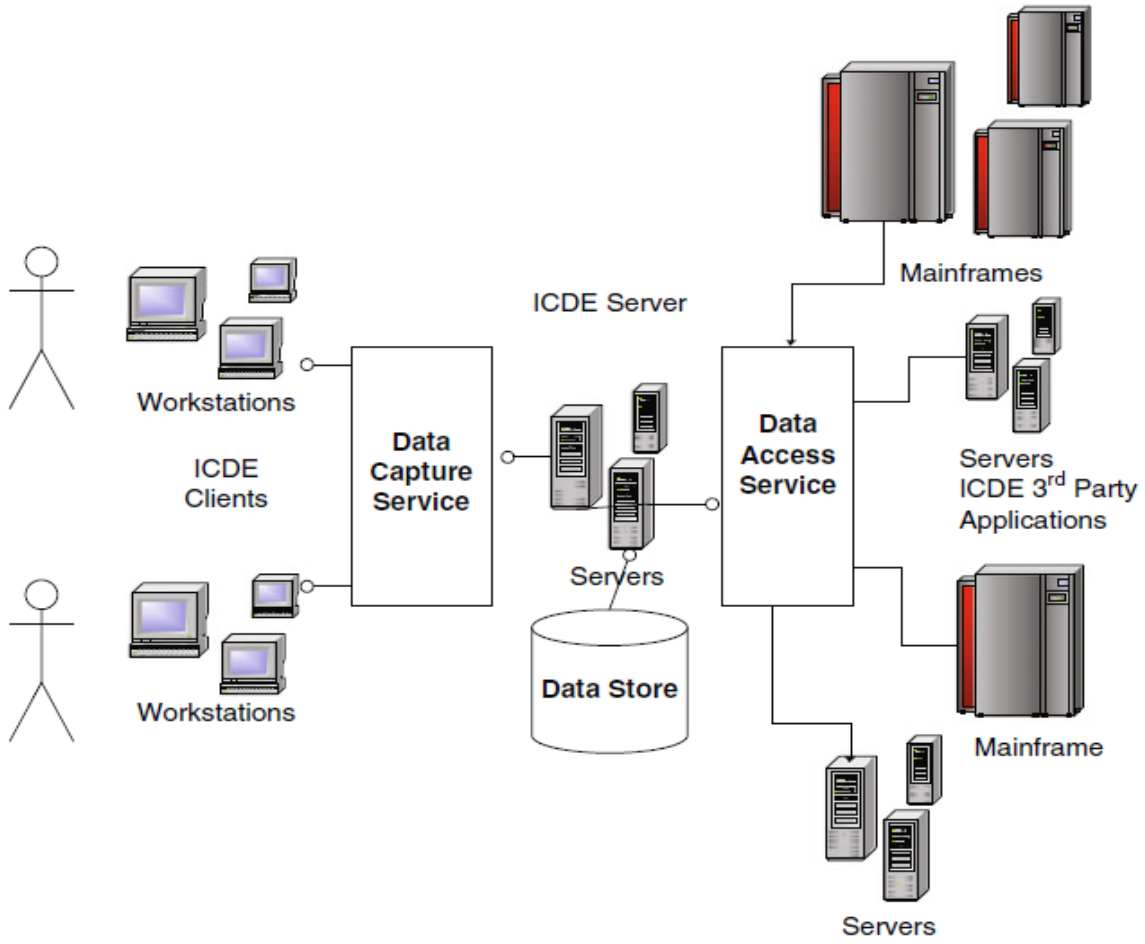


Fig. 2.2 ICDE system use cases

Information Capture and Dissemination Environment (ICDE)

- ✓ Queries and resulting web pages are stored; third-party tools can access this data to provide help (suggestions, additional queries, etc.)



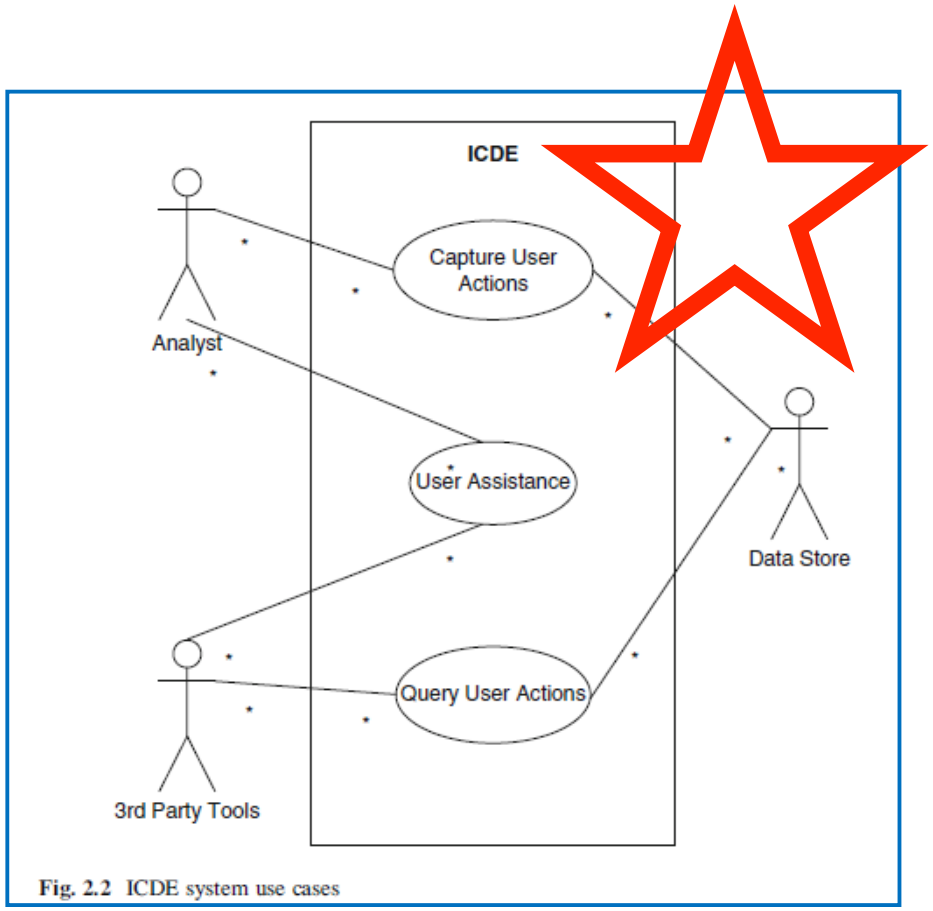
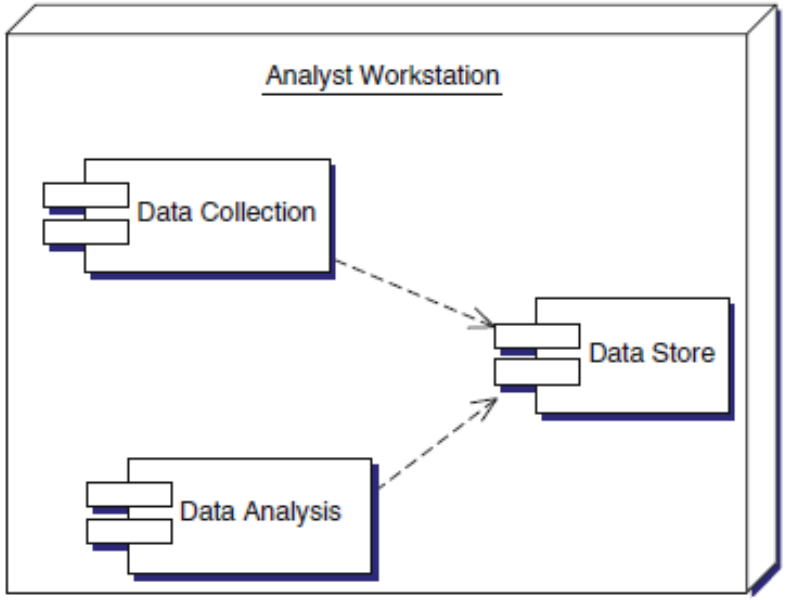


Fig. 2.2 ICDE system use cases

✓ Project Context – an initial production version is in place

- **ICDE 1.0 captures user actions**
  - versions of workstation client and backend store are in place
  - client written in Java; accesses data store using JDBC2
  - http://www.oracle.com/technetwork/java/javase/jdbc/index.html

- **Data Collection**: multi-process, tracks user actions and stores them as events
- **Data Store**: COTS relational database, stores events with timestamps
- **Data Analysis**: A GUI that allows DB queries to be created and run



✓ ICDE 2.0 was requested to serve some important business goals

Business Goal	Supporting Technical Objective
Encourage third-party developers	• Simple and reliable programmatic access to data store for third-party tools • Heterogeneous (i.e., non-Windows) platform support for running third-party tools • Allow third-party tools to communicate with ICDE users from a remote machine
Promote the ICDE concept to users	• Scale the data collection and data store components to support up to 150 users at a single site • Low-cost deployment for each ICDE user workstation

✓ Each business goal has an impact on the design

✓ The client will not be changed, for simplicity and redeployment reasons

- An interim release to tool developers after 6 months, exposing the API
- Final release after 12 months
- Budget constraints

✓ **The ICDE architecture example will be referred to throughout the textbook**

- Especially Chapter 9 and later
- We will see how this situation can be handled through architectural decisions
- This is a realistic scenario
  - Aren't starting from scratch
  - Certain pieces must be left in place
  - Wide range of applications
  - Constraints

- ✓ **The software quality attributes is usually guaranteed, or at least enabled, through architectural decisions**

- Quality attributes include things like:
  - scalability
  - security
  - performance
  - reliability
- Mushy general statements are not useful
- "The system must be scalable"

- ✓ **The software quality attributes is usually guaranteed, or at least enabled, through architectural decisions**
  - Quality attributes include things like:
    - scalability
    - security
    - performance
    - reliability
  - Mushy general statements are not useful
  - ~~"The system must be scalable"~~
  - "It must be possible to scale the deployment from an initial 100 geographically dispersed user desktops to 10,000 without an increase in effort/cost for installation and configuration."

- ✓ **Performance is often not just a latency requirement (must finish in 10 ms) but more often an average across a certain volume of traffic**

- **Throughput**
  - "1000 transactions per second"
    - Is this average? Worst case?
    - Often see things like "500 tps continuous, with up to ten-second bursts of 1000 tps"
- **Latency**
  - "95% of all requests must be processed in less than 4 s, and no requests must take more than 15 s."
- **Deadlines**
  - How long does an entire batch or job take to finish

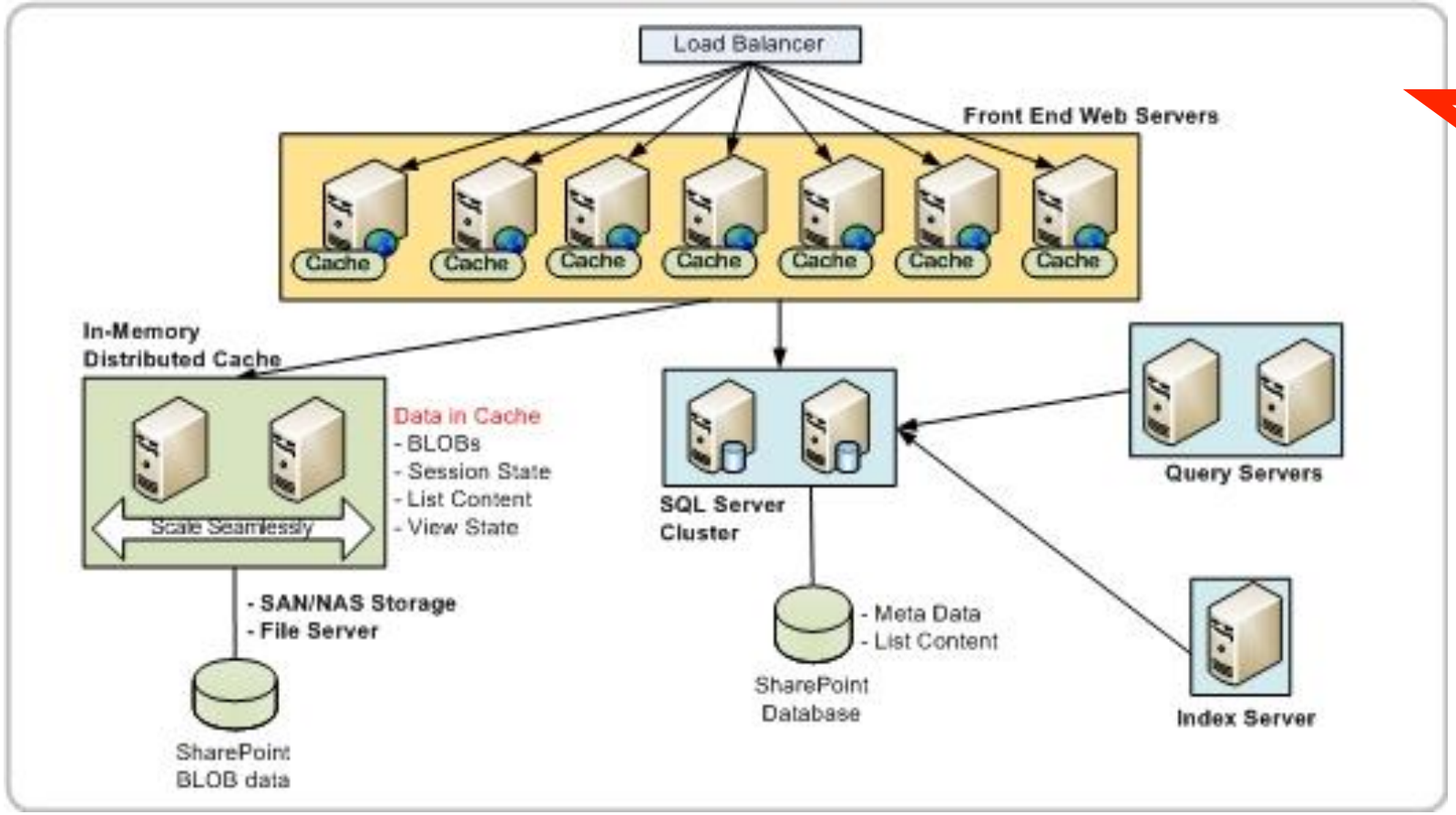
✓ The ICDE system is interactive; users must have their requests fielded rapidly enough to facilitate their work

- The ICDE system is interactive; users must have their requests fielded rapidly enough to facilitate their work

- Currently the client uses one process to capture user interactions and place them in a queue, while another process removes them from the queue and calls the server for storage
  - This is all observed to run very quickly

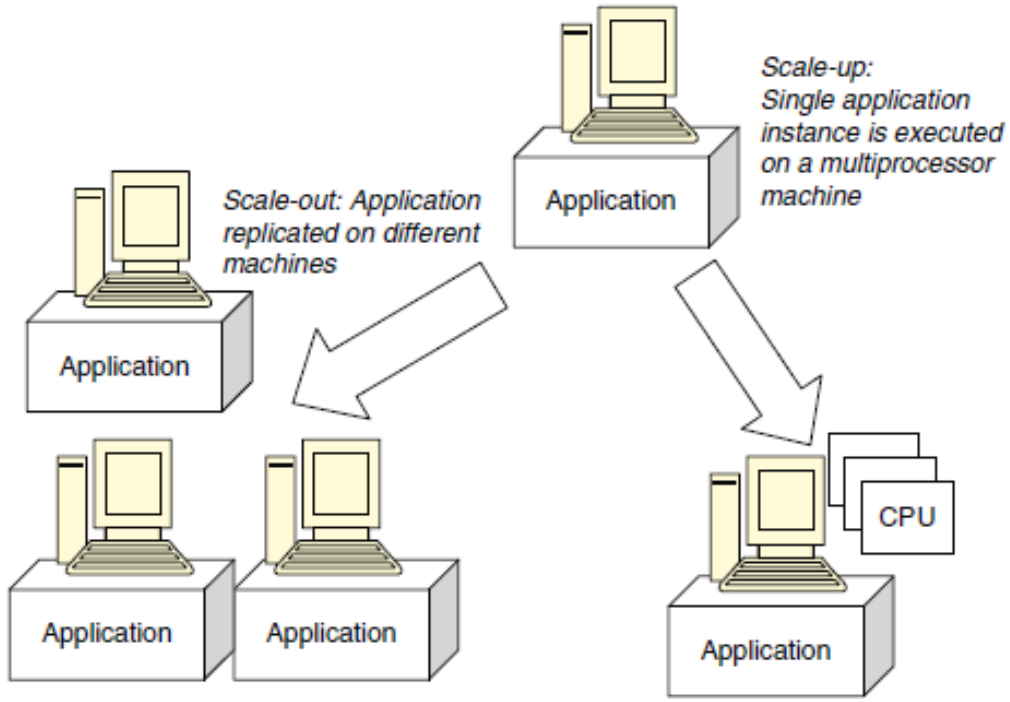
- To meet the interactivity requirement, the server must respond to ICDE client requests in less than one second, on average
  - An occasional longer delay can be tolerated

- ✓ Scalability is ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.



✓ Scalability is important – but in what attribute are we looking to scale? What is going to get bigger?

- **Request load** – what if we go from 100 transactions per second to 1000?
- We can "**scale out**" (more machines) or "**scale up**" (add CPUs and beef up our single server machine)



✓ Scale out vs. Scale up – architecture determines which is more straightforward

- If the requests can easily be sent to multiple machines, then scale out might be relatively easy
- If the app is multithreaded, then using multiple cores is not difficult

- **NOTE:**
  - We are designing to support future scaling, not to support the higher load at this time
  - When we scale, we don't want to change the architecture and the design

- ✓ **We might also need to scale in terms of the number of simultaneous connections, or in terms of the data size**

  - What if the analyst's work becomes more automated, and more are needed to do 1000 tps?
  - We don't need to process more transactions, but more users need to connect

  - What if the results of queries are expected to grow significantly (images instead of text)?
  - How does the storage change?

✓ **We know that the client will not change, so we don't need to worry about those machines**

- But, if the whole user system scales up, will we need to physically visit remote systems to install the client?
  - Probably costly
  - Maybe impossible (think about a classified environment)

- Remote deployment of the client is important for scaling
- Also, remote distribution of any new system parameters

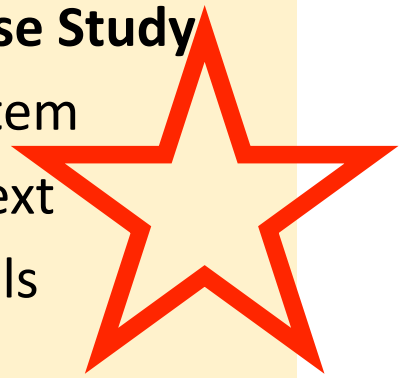
Review: Week 4 Objectives

- **Software Architecture**
  - What is Software Architecture
  - Why Software Architecture?
  - What Software Architecture Do?
- Relationship to Non-functional requirements
- Views of SW Architecture
  - SW Architecture Views
  - 4+1 Model
  - Views and Beyond
- Challenges
- Software Architect Roles

Week 4 Objectives

- **Software Architecture**
  - What is Software Architecture
  - Why Software Architecture?
  - What Software Architecture Do?
- Relationship to Non-functional requirements
- Views of SW Architecture
  - SW Architecture Views
  - 4+1 Model
  - Views and Beyond
- Challenges
- Software Architect Roles

- **Performance and Scalability**
- **The Textbook Case Study**
  - The ICDE system
  - Project Context
  - Business Goals
  - Constraints
- **Software Quality Attributes**
  - Performance
  - Scalability



Reminder: To-Do This Week

- **Assignment 5: Software Architecture**
  - Develop a high-mid level of your project architecture
  - Develop your software components accordingly
  - Examine your system requirements and explain the use case scenario at each level
  - Upload your work in word document via blackboard >> EGR326 >> Assignment >> **Asn 5: Software Architecture**
  - **Due date**: Friday, Feb 7 by 10:00pm

EGR326 Software Design and Architecture

Lecture 5. Structural Patterns

Spring 2020

Kim Peters, Ph.D.

Gordon and Jill Bourns College of Engineering
California Baptist University

- **Structural Patterns**
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

To-Do Week 3 (Asn4: Design Pattern)

- **Discussion: Design Pattern**
  - Prepare a three-slide presentation on a specific design pattern that we will be discussing later (read ahead on your particular pattern).
  - Your presentation should follow this general form:
    - Slide 1 – Title slide
    - Slide 2 – Name of the pattern, intent and typical use
    - Slide 3 – The pattern's structure (participants and collaboration)
    - Slide 4 – When and how it would be used – an example if there is room and it makes sense
  - Here are the patterns that you will be addressing.
    - Adapter
    - Bridge
    - Decorator
    - Interpreter
    - Iterator
    - Observer

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Recall: Why Creational Patterns?

✓ **Creational Patterns are used to implement the instantiation process** – *generally providing greater abstraction*

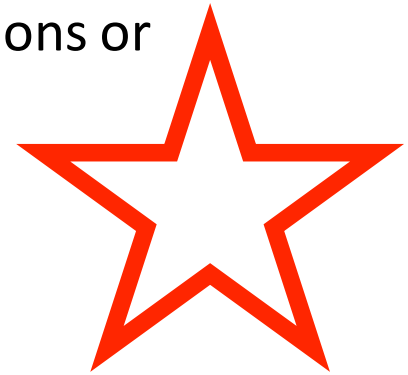
- Keep implementation details as contained as possible
- Make it easy to change platform/components/design

- **Two common themes**:
  - **Encapsulate** knowledge about which concrete classes are being used.
  - **Hide** the way in which instances of these classes are created and put together.

- Interfaces are specified in one or more abstract classes, which derived classes must implement, but clients can then use.

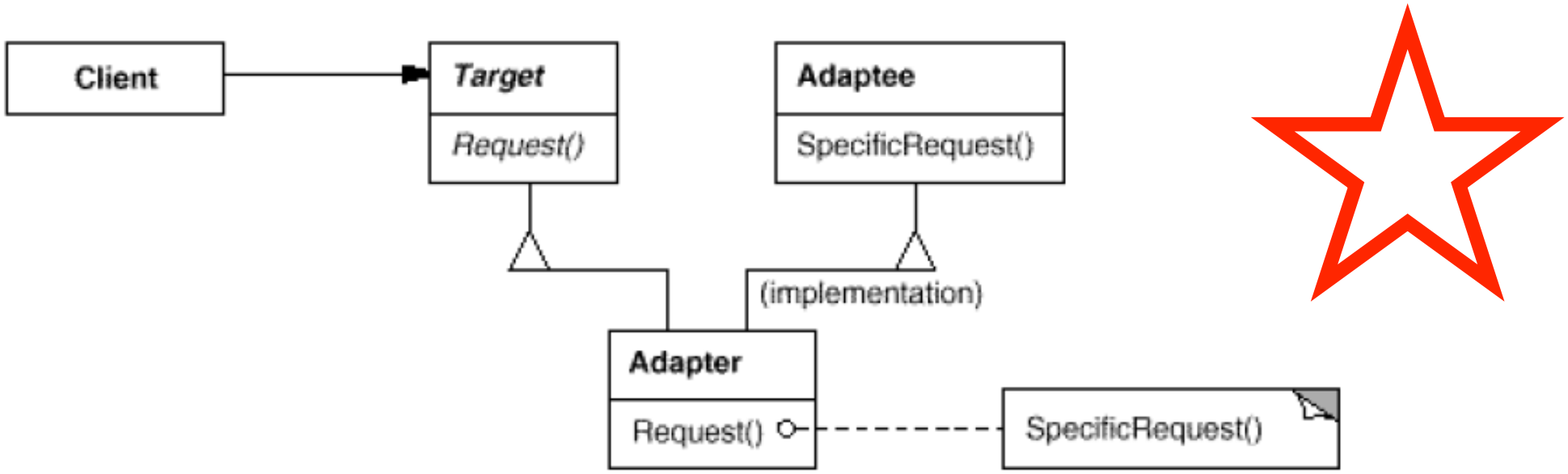
✓ The Adapter Pattern allows a class with one interface to be used by clients that expect another

- **Why Adapter? Often When Reusing Code:**
  - the existing interface needs to be modified to fit with a newer design
  - interfaces may need to be adapted when supporting multiple implementations or algorithms
  - editing a working class directly, to modify the interface, is a bad idea!



- **What Adapter Do?**
  - **The Adapter Provides** an in-between layer to transform one interface into another
  - This adaptation can happen at the Class or Object level…

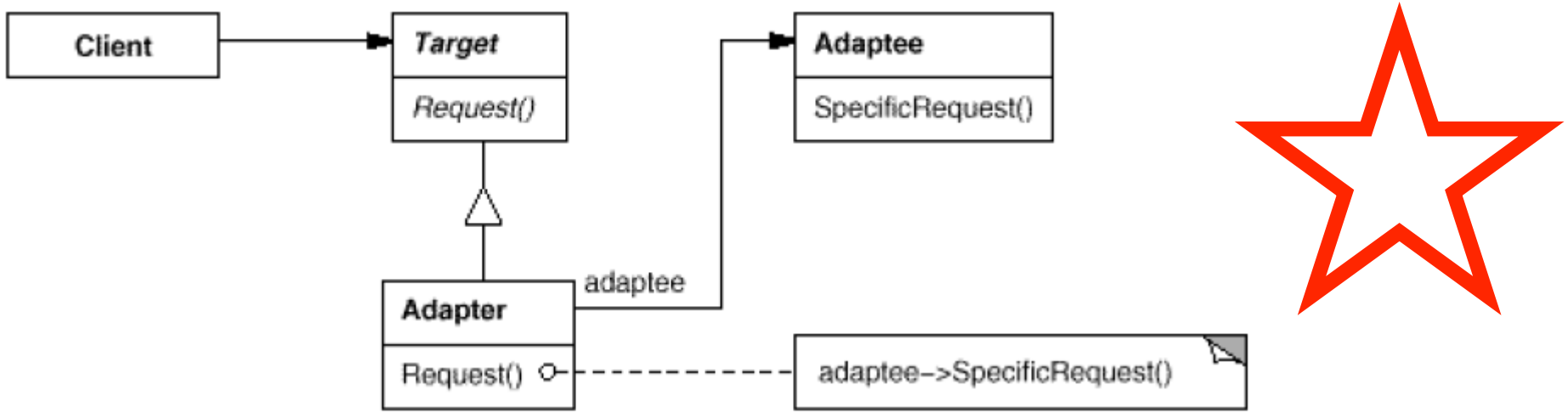
- ✓ The Adapter Pattern allows a class with one interface to be used by clients that expect another



- **The Adapter inherits from both the Target and Adaptee classes**
  - The Request() interface is what the client sees
  - The SpecificRequest() interface is what we are adapting (what exists in the class to be adapted)

Adapter Pattern: The Object Adapter Pattern

✓ The Adapter Pattern allows a class with one interface to be used by clients that expect another



- **The Adapter inherits from Target but references an Adaptee object**
  - Calls to Request() generate calls to one or more SpecificRequest() functions, and probably some additional code is executed (in Adapter)

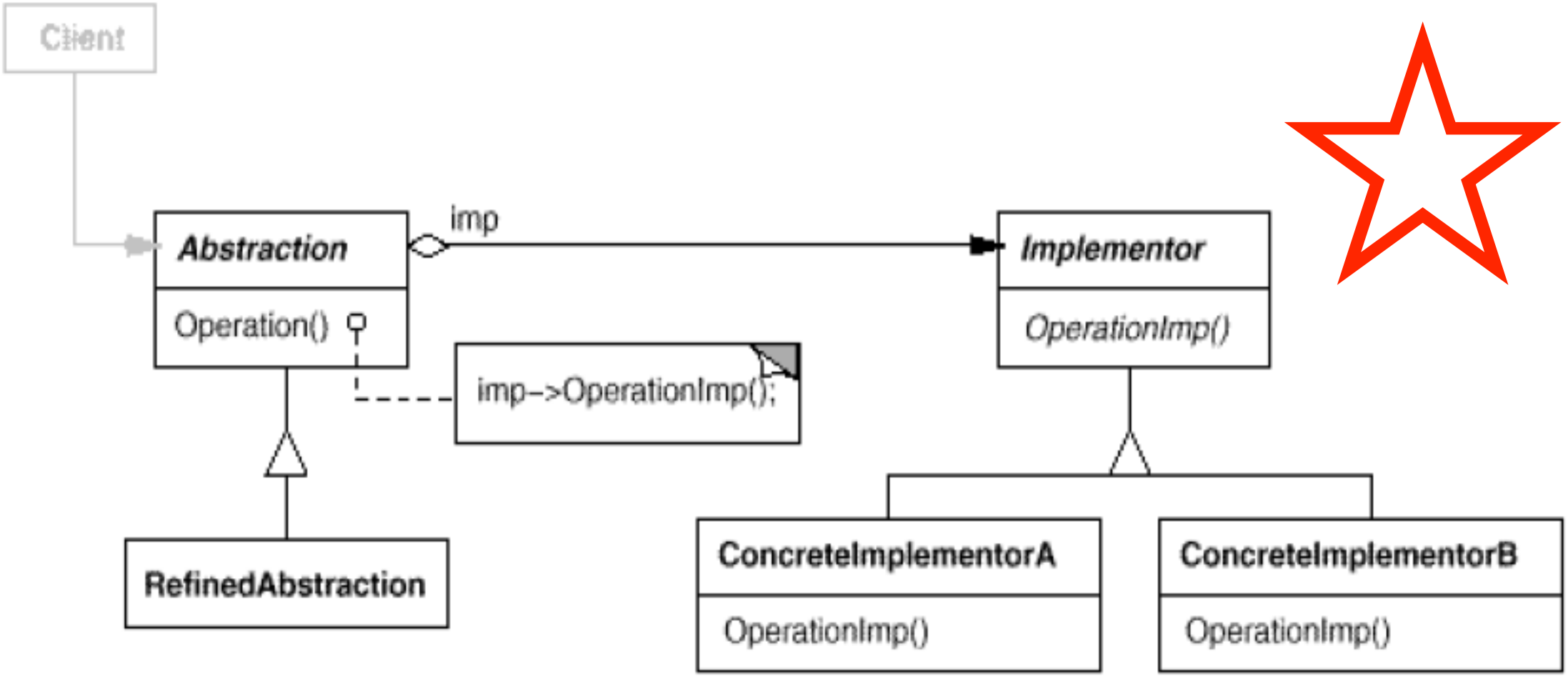
- **Target**
  - defines the domain-specific interface that Client uses.
- **Client**
  - collaborates with objects conforming to the Target interface.
- **Adaptee**
  - defines an existing interface that needs adapting.
- **Adapter**
  - adapts the interface of Adaptee to the Target interface.

- **Collaborations**
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

- ✓ **The Bridge pattern separates a Class implementation from its interface; this is useful for several reasons**

  - If one class may have several implementations
  - If we want to hide even the interface details from the client
  - If several classes share the same (or parts of the same) implementation
    - Think of the STL container classes

- ✓ The Bridge pattern hides the implementation(s) of a particular class



- **Abstraction**
  - defines the abstraction's interface and maintains a reference to an object of type Implementor.
- **RefinedAbstraction**
  - Extends the interface defined by Abstraction.
- **Implementor**
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor**
  - implements the Implementor interface and defines its concrete implementation.

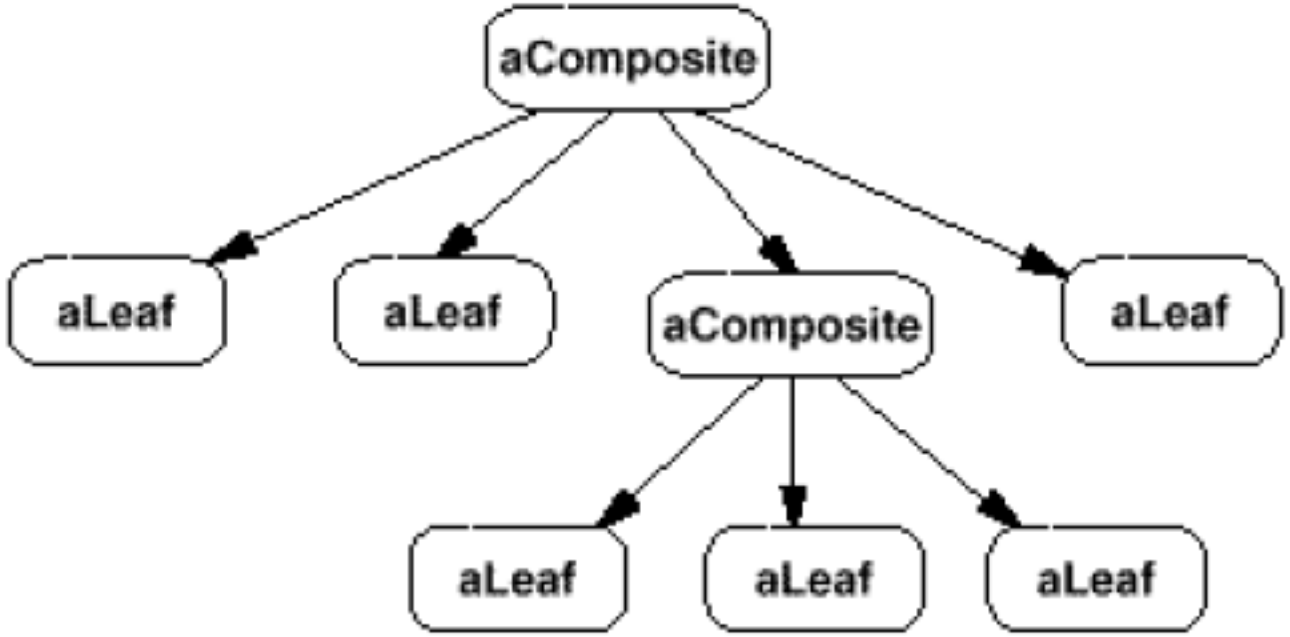
- **Collaborations**
  - Abstraction forwards client requests to its Implementor object.

- **Abstraction**
  - defines the abstraction's interface and maintains a reference to an object of type Implementor.
- **RefinedAbstraction**
  - Extends the interface defined by Abstraction.
- **Implementor**
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor**
  - implements the Implementor interface and defines its concrete implementation.

- **Collaborations**
  - Abstraction forwards client requests to its Implementor object.

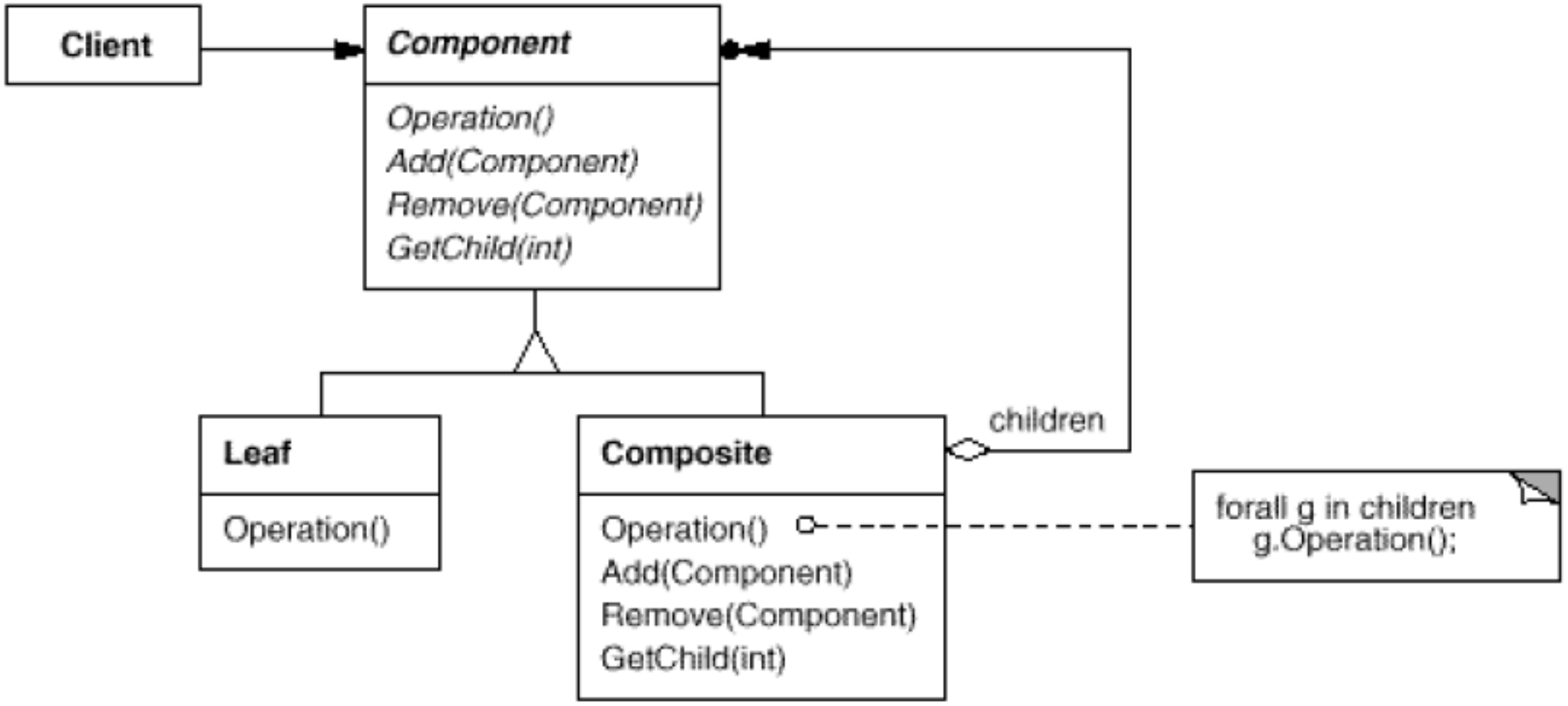
✓ The Composite Pattern allows related objects to be treated as a whole, in a defined hierarchy

- When objects form a tree, it's nice to have a consistent way to operate on the whole tree or on a subtree…



✓The Composite pattern describes an inheritance strategy to facilitate this

- ✓ In the Composite pattern, both the Composite itself and its Leaf members inherit from the ***Component*** class



Composite Pattern: Participants

- **Component**
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Leaf**
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- **Composite**
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface
- **Client** - manipulates objects in the composition through the Component interface.
- **Collaborations**
  - Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

✓ The Decorator pattern is used to add functionality (responsibilities) to an object at run-time.

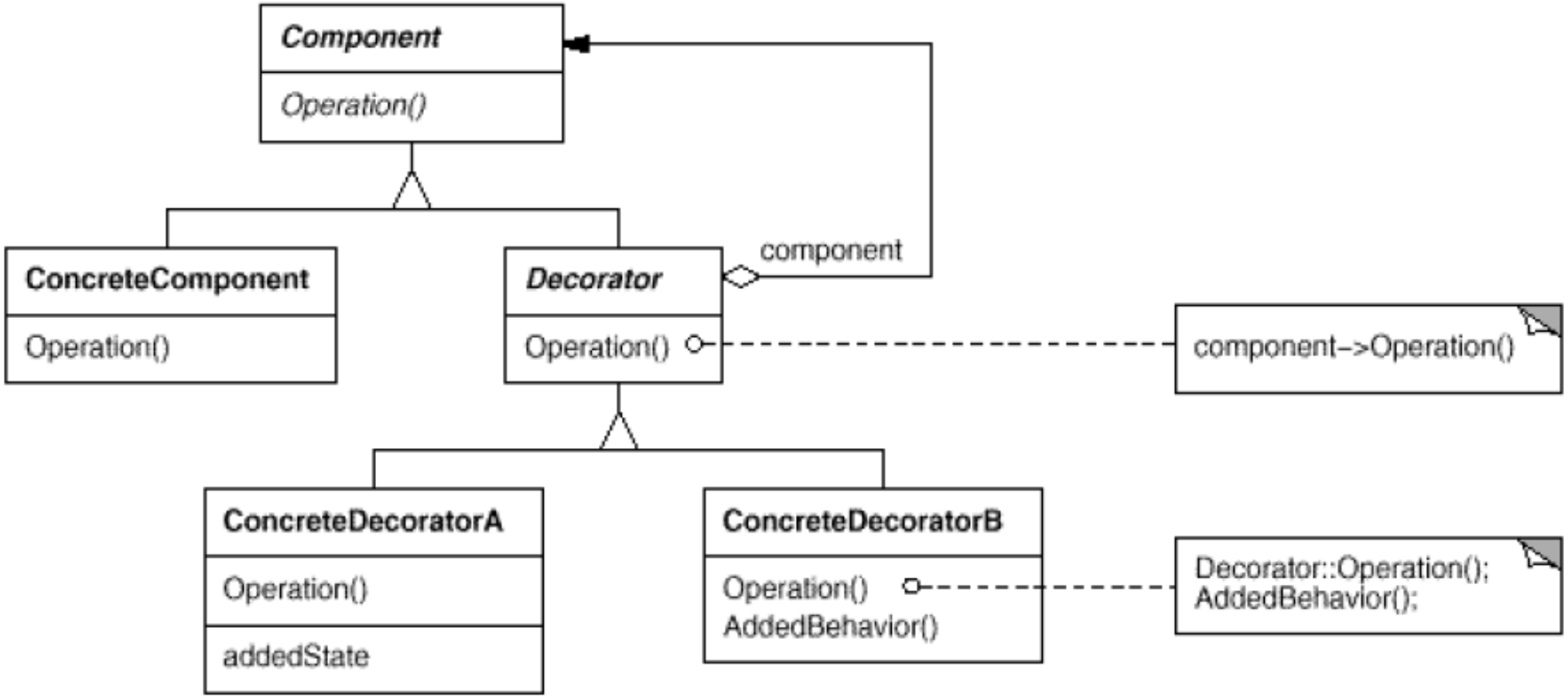
- It's also called a Wrapper

- The essence is that the base class (the Component) defines the interface, but multiple derived classes with added "stuff" may be instantiated and addressed thereby
  - Of course, addressing through base class references won't expose any additional functions
  - That would require addressing in terms of the derived class

- Can be more flexible than simple inheritance

Decorator Pattern (Wrapper)

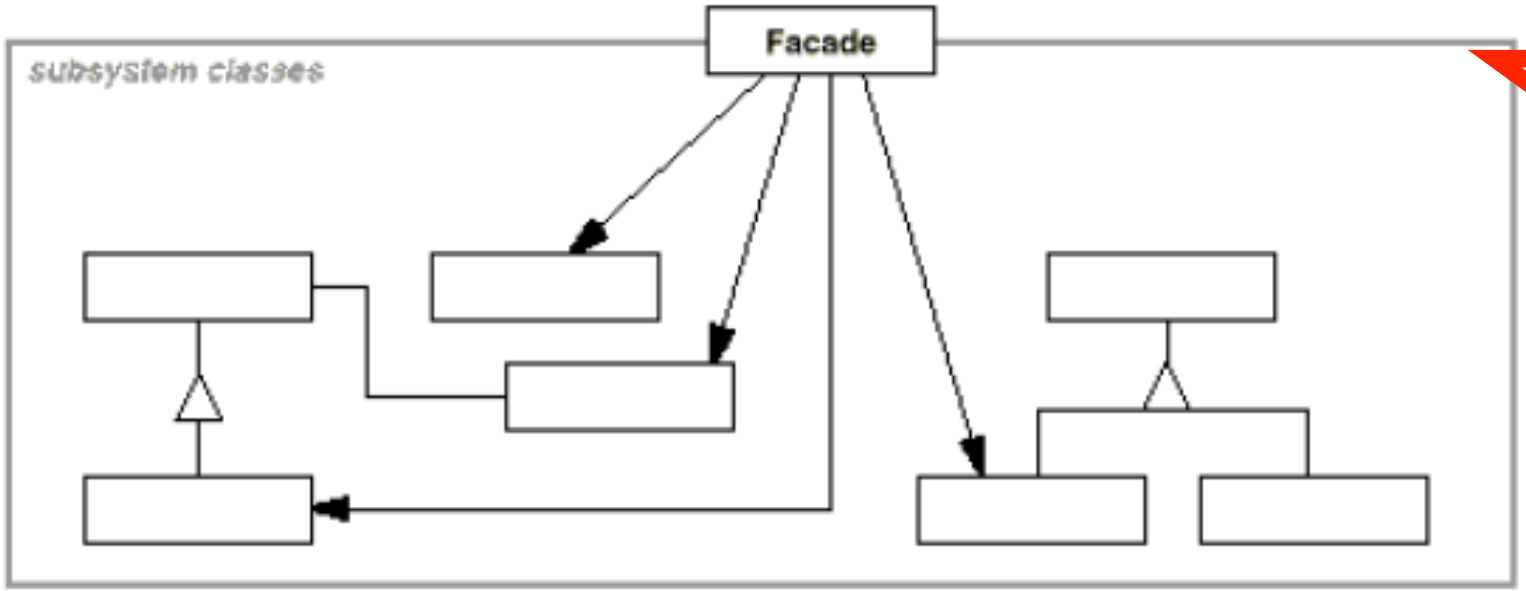
- ✓ **The Decorator pattern is used to add functionality (responsibilities) to an object at run-time.**
  - Can be more flexible than simple inheritance



- **Component**
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent**
  - defines an object to which additional responsibilities can be attached.
- **Decorator**
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**
  - adds responsibilities to the component.

- **Collaborations**
  - Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

- **This pattern is widely used** – often unknowingly
  - It's just a good design idea



Facade Pattern: Participants

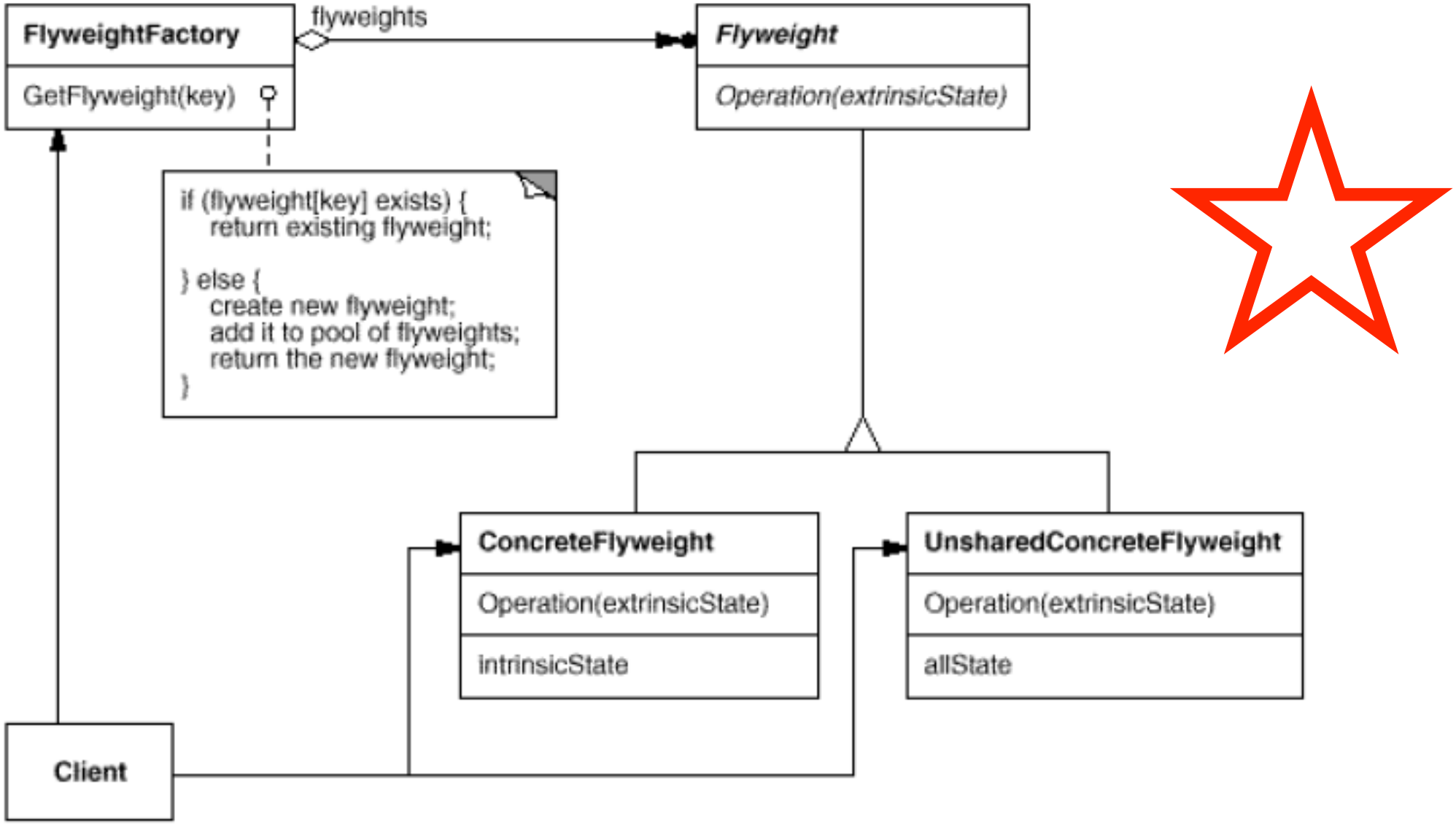
- **Facade**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **subsystem classes**
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.

- **Collaborations**
  - Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
  - Clients that use the facade don't have to access its subsystem objects directly.

- Requests for a new (flyweight) object may return a reference to an existing object
- Intrinsic state information is stored in the object
- Extrinsic (how it's used) information is passed in and out
- The book's example – characters in a document
  - Intrinsic is the character code, font, color, etc.
  - Extrinsic includes position and perhaps other attributes

- **Why would we do this?**
  - Saves space
  - Reduces complexity

Flyweight


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Flyweight: Participants

- **Flyweight**
  - declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight**
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight**
  - not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
- **Client**
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).

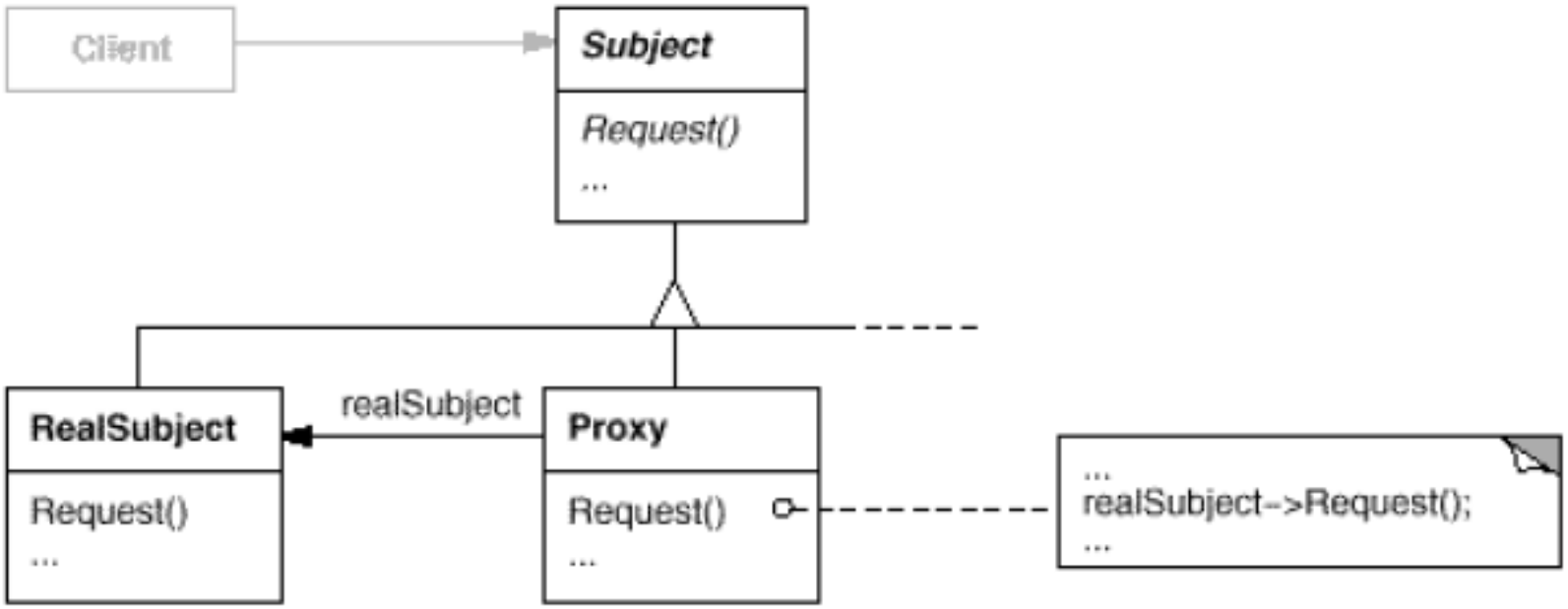
- ✓ The Proxy pattern provides a surrogate or stand-in for another object; generally for access control or data validation

  - **For example, clients only address an Image through the ImageProxy**

  - **ImageProxy can do several important things:**
    - Validate data before the image is modifies
    - Check access rights
    - Wait for a particular time before updating Image

  - **The name "Surrogate" is also used**

- ✓ The Proxy pattern provides a surrogate or stand-in for another object; generally for access control or data validation



- **A Subject base class enforces the common interface**
  - Reducing need for transformation of interface

Proxy Pattern (Surrogate): Participants

- **Proxy**
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
  - virtual proxies may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
  - protection proxies check that the caller has the access permissions required to perform a request.
- **Subject**
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
- **RealSubject** - defines the real object that the proxy represents.
- **Collaborations**
  - Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

- **Structural Patterns**
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

To-Do This Week (Week 3, Asn4: Design Pattern)

- **Discussion: Design Pattern**
  - Prepare a three-slide presentation on a specific design pattern that we will be discussing later (read ahead on your particular pattern).
  - Your presentation should follow this general form:
    - Slide 1 – Title slide
    - Slide 2 – Name of the pattern, intent and typical use
    - Slide 3 – The pattern's structure (participants and collaboration)
    - Slide 4 – When and how it would be used – an example if there is room and it makes sense
  - Here are the patterns that you will be addressing.
    - Adapter
    - Bridge
    - Decorator
    - Interpreter
    - Iterator
    - Observer

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

EGR326 Software Design and Architecture

Lecture 6. Behavioral Patterns

Spring 2020

Kim Peters, Ph.D.

Gordon and Jill Bourns College of Engineering
California Baptist University

Feedback: To-Do Last Week (Asn3: Project Selection)

- **Discussion:** Tell us about your group project!
- **Background Research: SWOT** (Strength, Weakness, Opportunity, Threats) Analysis
- **SOW:** Summarize the goal of your project in 15 words or less
- **Requirements**

- **Assignment (Asn 3):**
  1. Background Research – limit to one page
  2. SOW (Statement of Work) – limit to one sentence
  3. **Requirements** - Focus on capturing functional requirements like writing user stories

  - Upload your work, one document per group, via blackboard >> EGR326 >> Assignment >> Asn 3
  - **Due date**: Friday, Jan 24 by 10:00pm

- **Structural Patterns**
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

- **Behavioral Patterns**
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer (Publish/Subscribe)
  - State
  - Strategy
  - Template Method
  - Visitor

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

To-Do Week 3 (Asn4: Design Pattern)

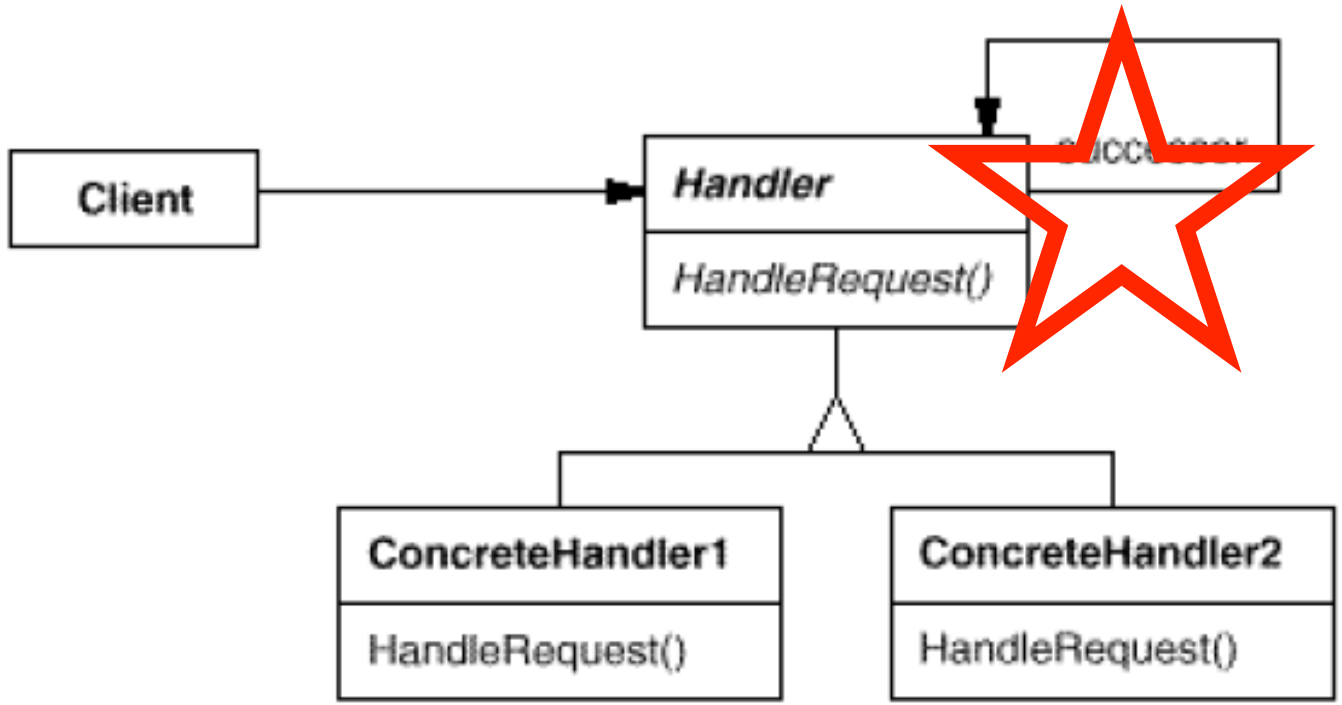
- **Discussion: Design Pattern**
  - Prepare a three-slide presentation on a specific design pattern that we will be discussing later (read ahead on your particular pattern).
  - Your presentation should follow this general form:
    - Slide 1 – Title slide
    - Slide 2 – Name of the pattern, intent and typical use
    - Slide 3 – The pattern's structure (participants and collaboration)
    - Slide 4 – When and how it would be used – an example if there is room and it makes sense

  - Here are the patterns that you will be addressing.
    - Adapter
    - Bridge
    - Decorator
    - Interpreter
    - Iterator
    - Observer

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

✓ It allows a request to be forwarded from object to object until one handles it

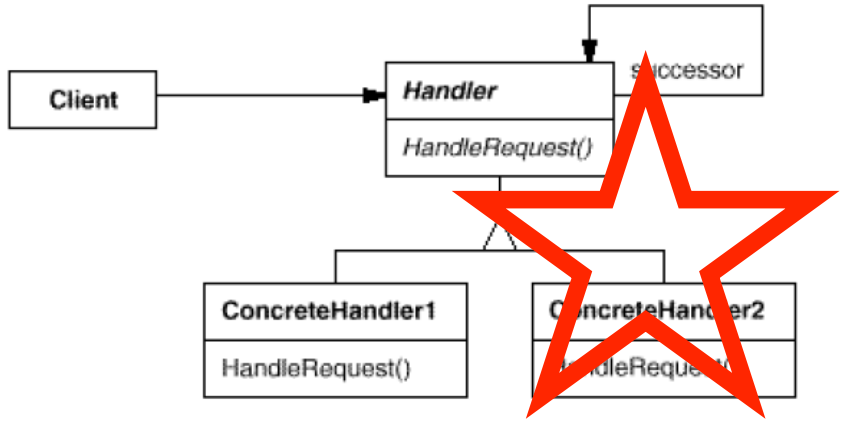
- **Objects** that inherit from Handler have a reference to a successor;
- **Calls** to HandleRequest() may or may not initiate to the successor's HandleRequest() method



Chain of Responsibility Pattern (CRP) Participants

✓ CRP allows a request to be forwarded from object to object until one handles it

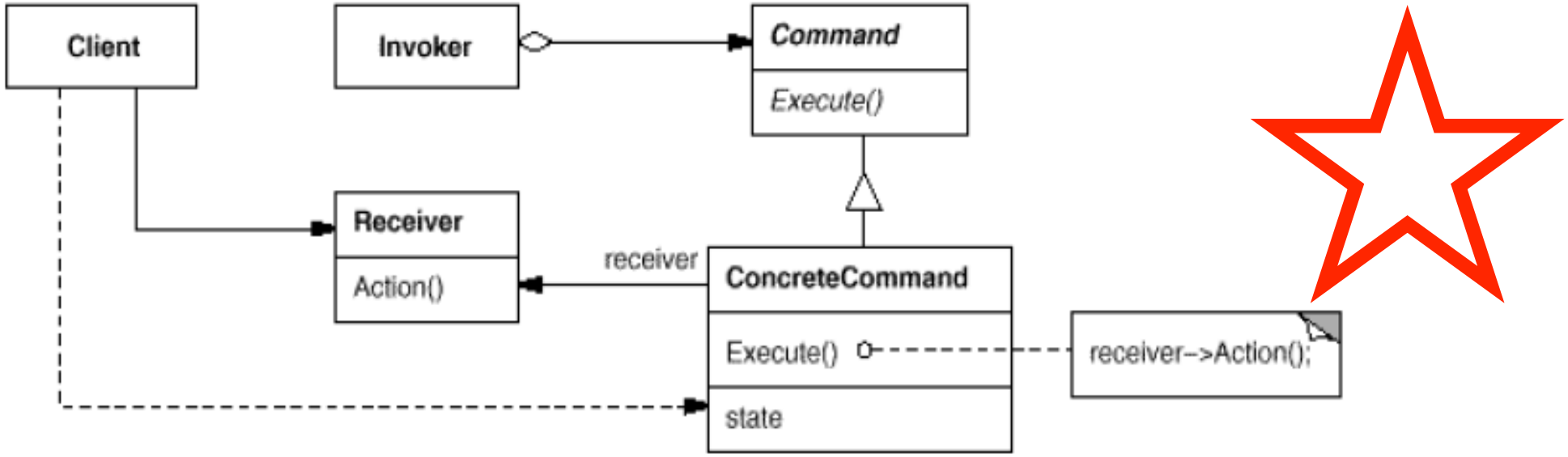
- **Handler**
  - defines an interface for handling requests.
  - (optional) implements the successor link.
- **ConcreteHandler**
  - handles requests it is responsible for.
  - can access its successor.
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
- **Client**
  - initiates the request to a ConcreteHandler object on the chain.



✓**Collaborations**

- When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

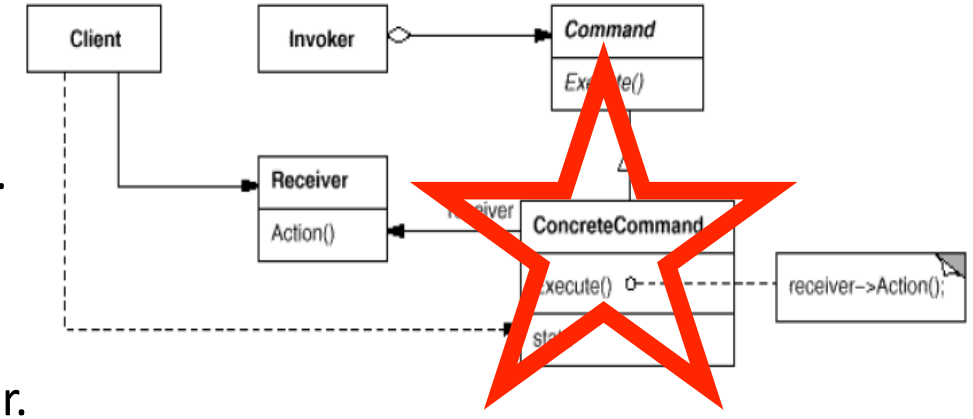
✓ Encapsulates requests inside objects, to allow flexible processing (delayed, prioritized, etc.)



- This pattern can be implemented so as to support Undo of a request, queueing and logging, and processing at different priority levels.
- The paradigm is commonly known as transaction processing.

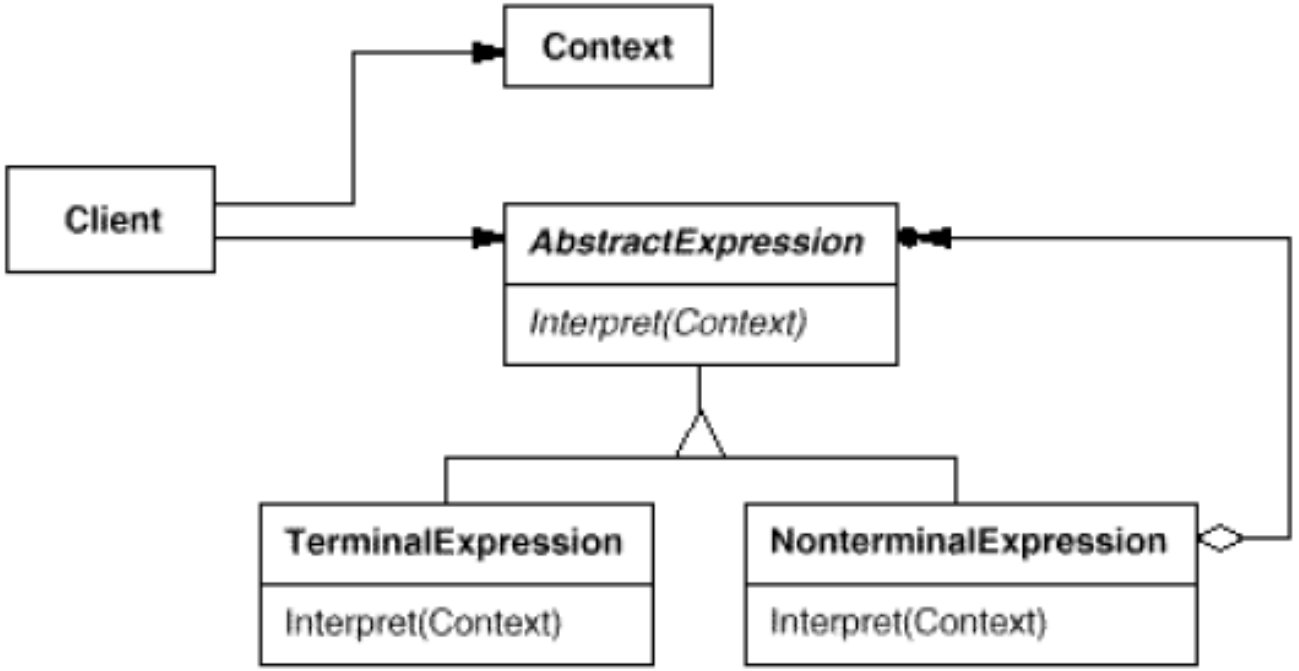
Command Participants

✓ Encapsulates requests inside objects, to allow flexible processing (delayed, prioritized, etc.)



- **Command** - declares an interface for executing an operation.
- **ConcreteCommand**
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.
- **Client** - creates a ConcreteCommand object and sets its receiver.
- **Invoker** - asks the command to carry out the request.
- **Receiver -** knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.
- **Collaborations**
  - The client creates a ConcreteCommand object and specifies its receiver.
  - An Invoker object stores the ConcreteCommand object.
  - The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
  - The ConcreteCommand object invokes operations on its receiver to carry out the request.

- ✓ **Provides interpretation for a hierarchical language with simple grammar –** to evaluate language grammar or expression. Used in SQL parsing, symbol processing engine etc.

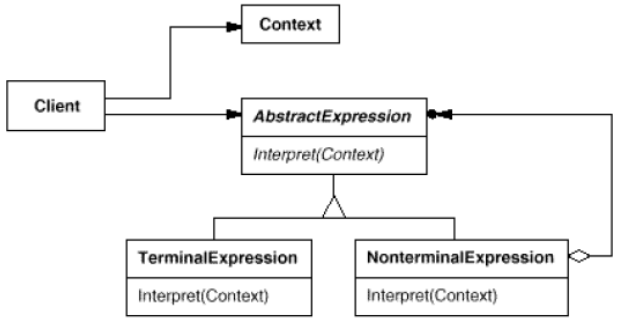


Don't think merely in terms of linguistic interpretation;

this can also mean parsing input statements or commands, or reshaping data structures such as trees

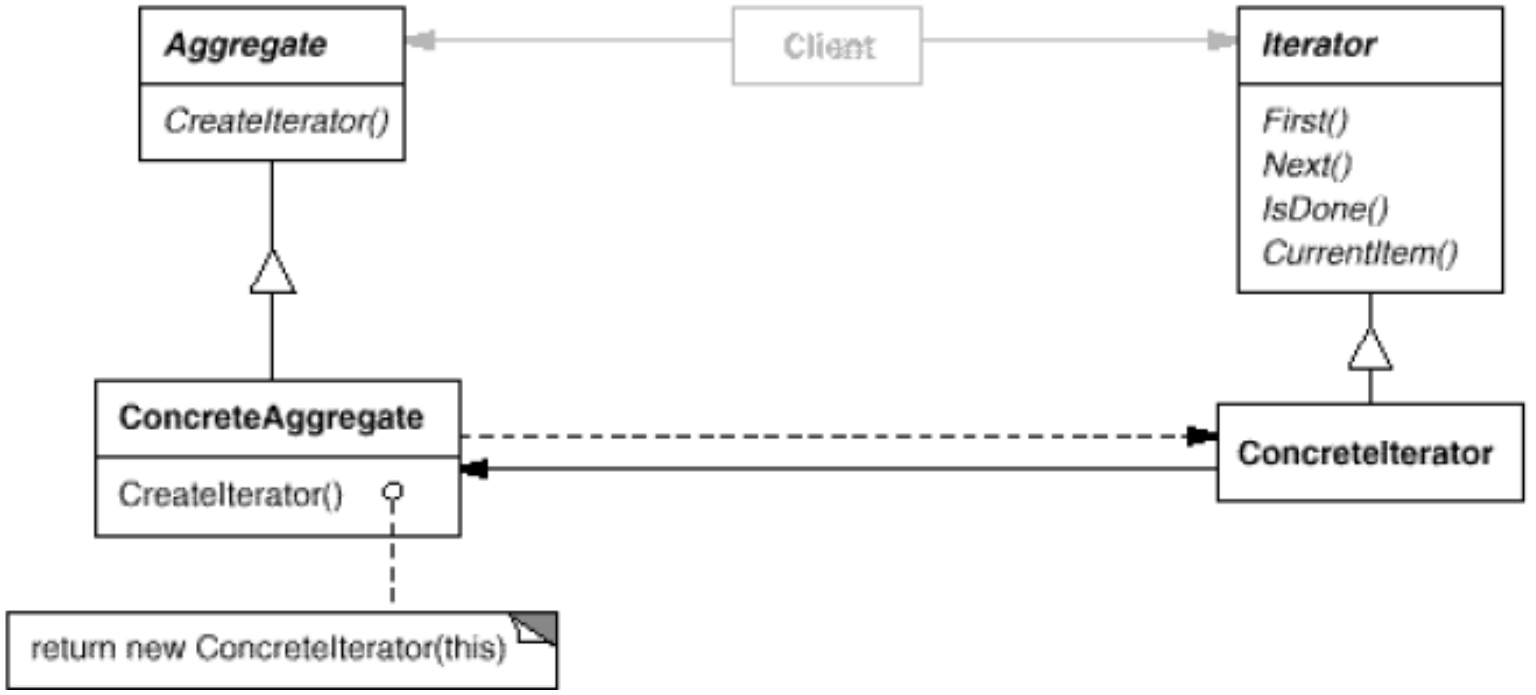
Interpreter Participants

- **AbstractExpression** - declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.
- **TerminalExpression**
  - implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in a sentence.



- **NonterminalExpression**
  - one such class is required for every rule R ::= R1 R2 ... Rn in the grammar.
  - maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
  - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.
- **Context** - contains information that's global to the interpreter.
- **Client -** builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances ofthe NonterminalExpression and TerminalExpression classes.
  - invokes the Interpret operation.
- **Collaborations**
  - The client builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances. Then the client initializes the context and invokes the Interpret operation.
  - Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression.
  - The Interpret operation of each TerminalExpression defines the base case in the recursion.
  - The Interpret operations at each node use the context to store and access the state of the interpreter.

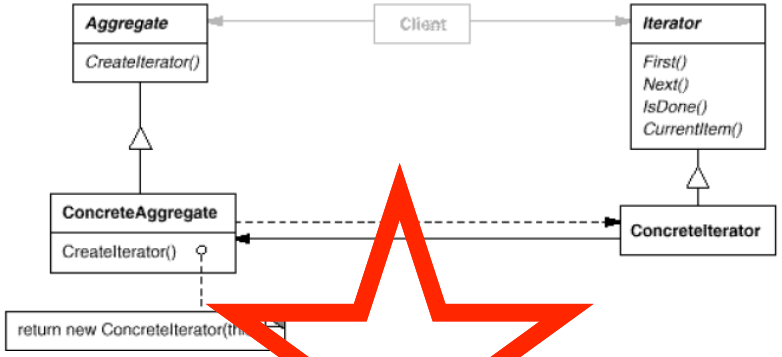
✓ Allows access to contained objects or elements, independent of the implementation



- This lays out a standard interface for the minimal set of methods required to walk through a collection of objects

Iterator Participants

- **Iterator**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate**
  - defines an interface for creating an Iterator object.
- **ConcreteAggregate**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

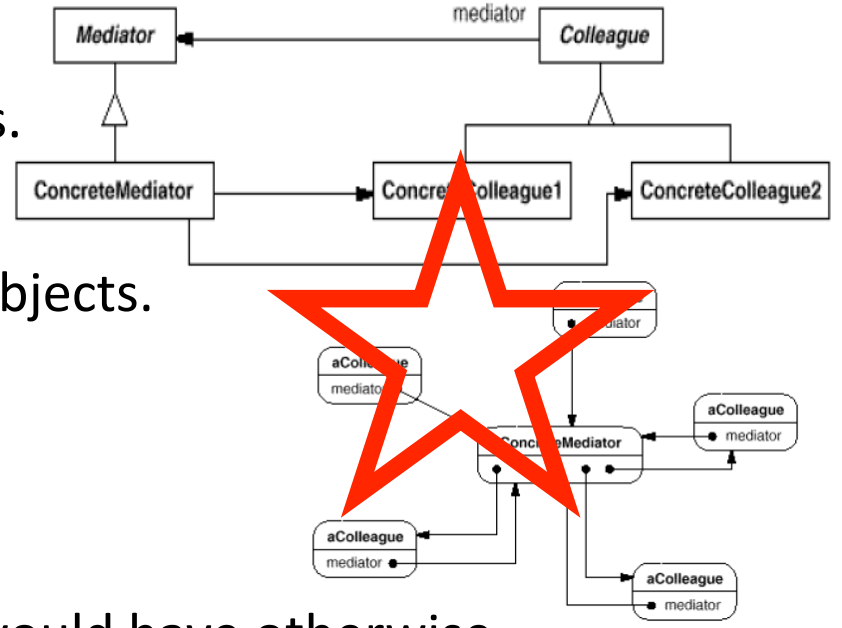


✓**Collaborations**

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

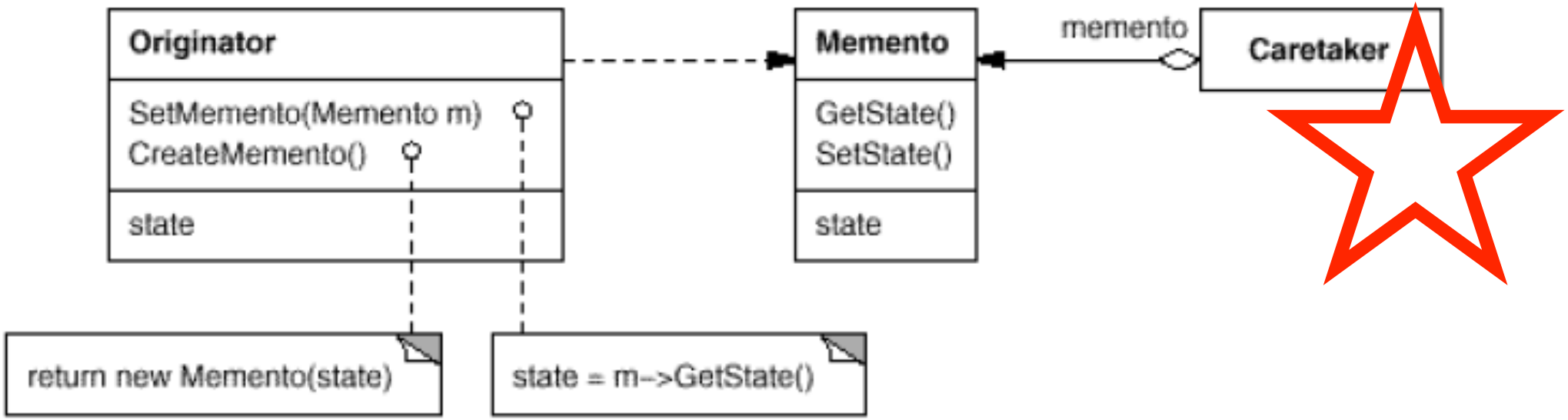
Mediator Participants

- **Mediator**
  - defines an interface for communicating with Colleague objects.
- **ConcreteMediator**
  - implements cooperative behavior by coordinating Colleague objects.
  - knows and maintains its colleagues.
- **Colleague classes**
  - each Colleague class knows its Mediator object.
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.



- **Collaborations**
  - Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).

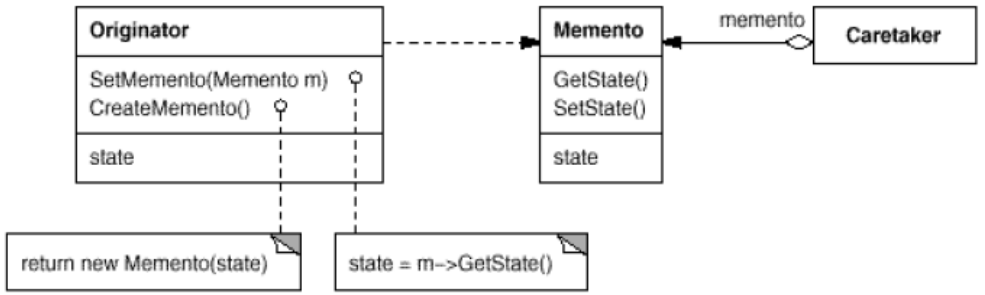
✓ Preserves the internal state of an object, for later restoration



- Our objective in using this pattern is to save a snapshot of an object, without having to expose all of that object's implementation details and internals.

Memento Participants

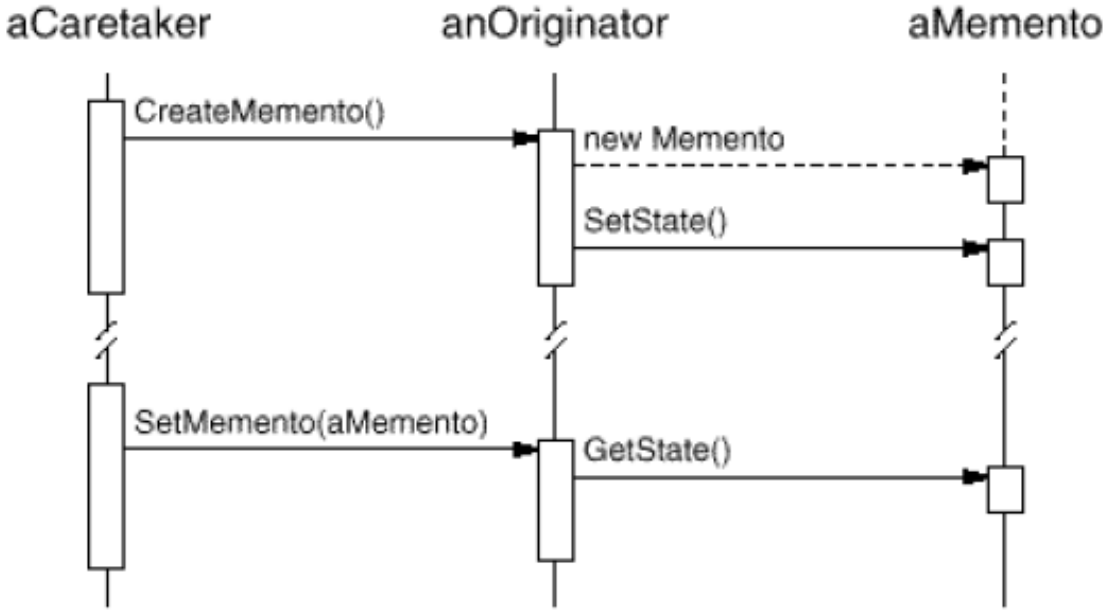
- **Memento**
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento — it can only pass the memento to other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.
- **Originator**
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state.
- **Caretaker**
  - is responsible for the memento's safekeeping.
  - never operates on or examines the contents of a memento.

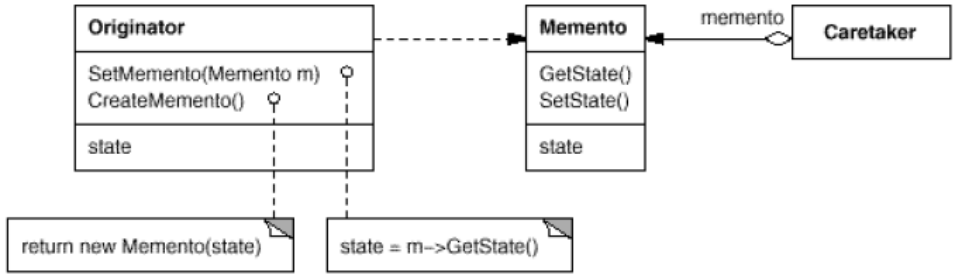


- **Collaborations**
  - A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator.

Memento Pattern Interaction Diagram

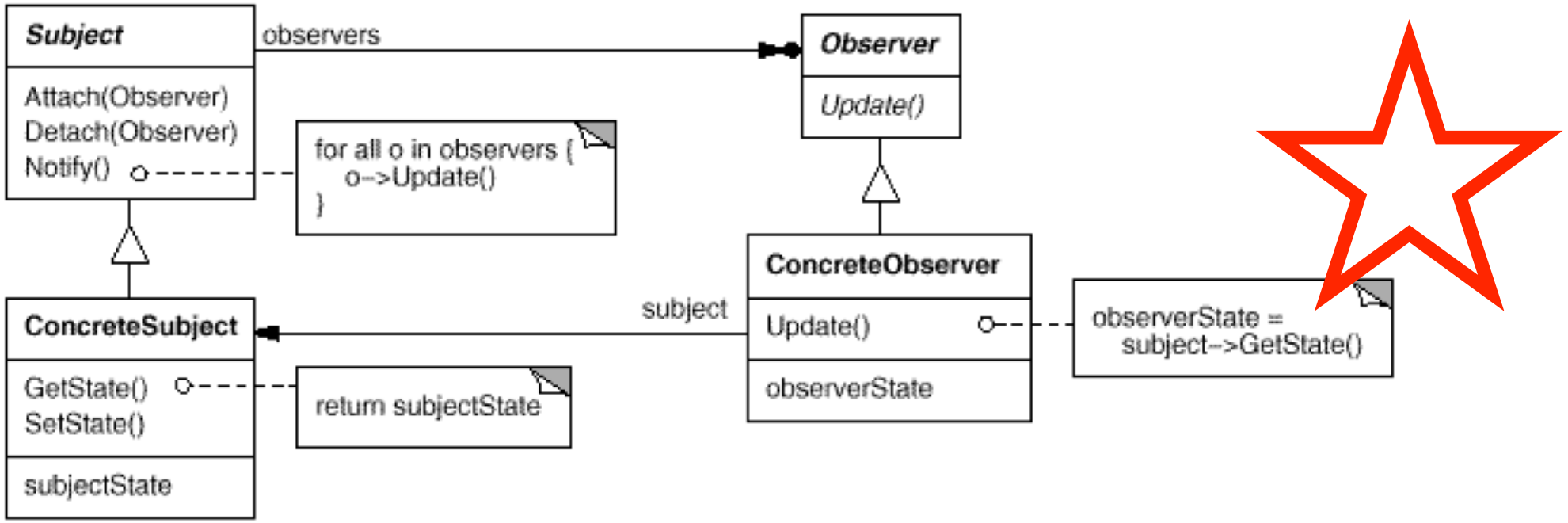
✓ Preserves the internal state of an object, for later restoration





- The first interactions have to do with saving the state
- The last interaction is an occurrence of restoring that state

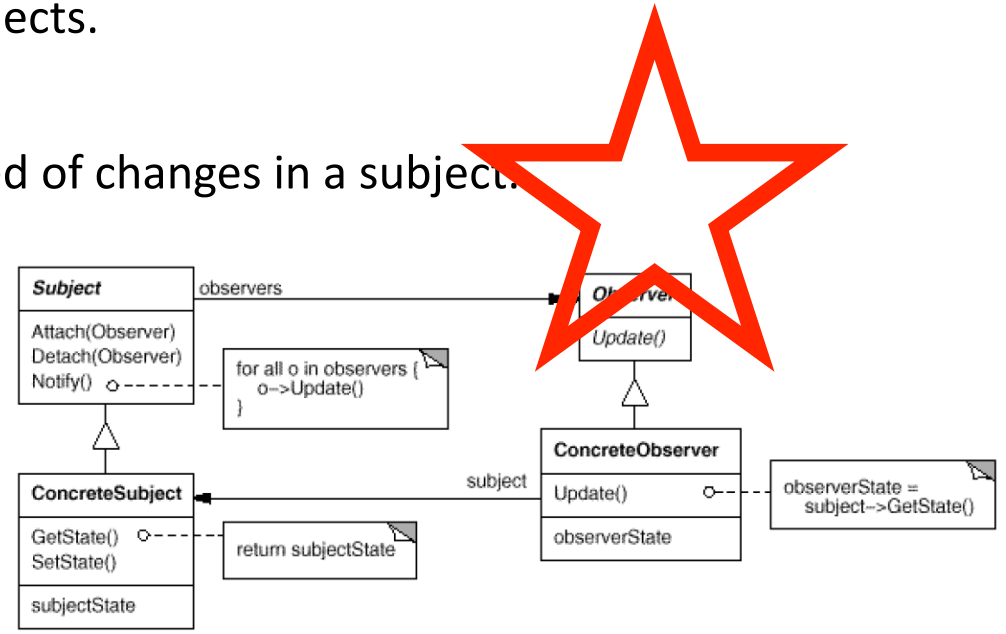
✓ Relates an object that may change to other objects that will be notified of the change



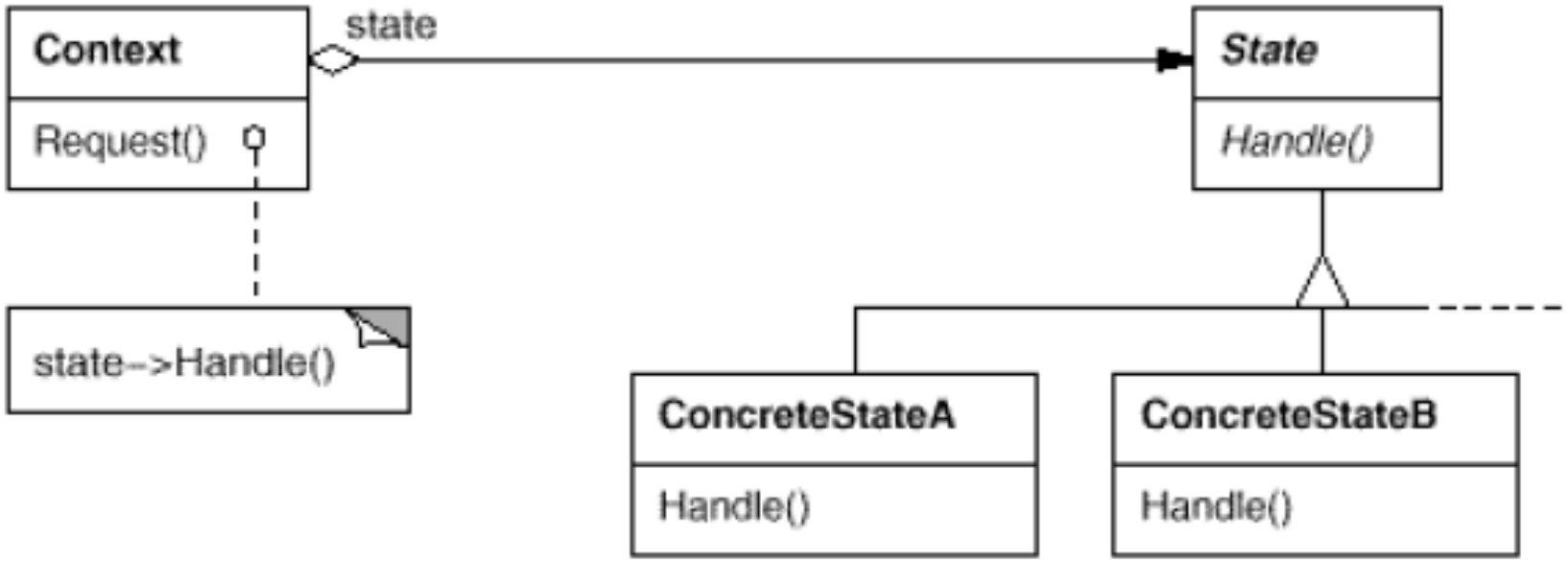
- Can also be referred to as a "Publish-Subscribe" arrangement

Observer Participants

- **Subject**
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- **ConcreteObserver**
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's
- **Collaborations**
  - ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
  - After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.



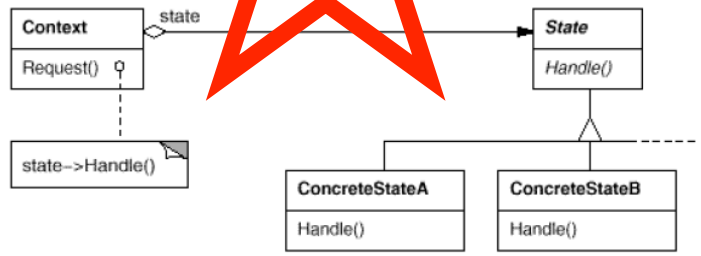
✓ Allows an object to change its response to messages, etc., when its internal state changes



- The different states are instantiated in different derived classes, each with different behavior – providing a common interface through the Handle() function

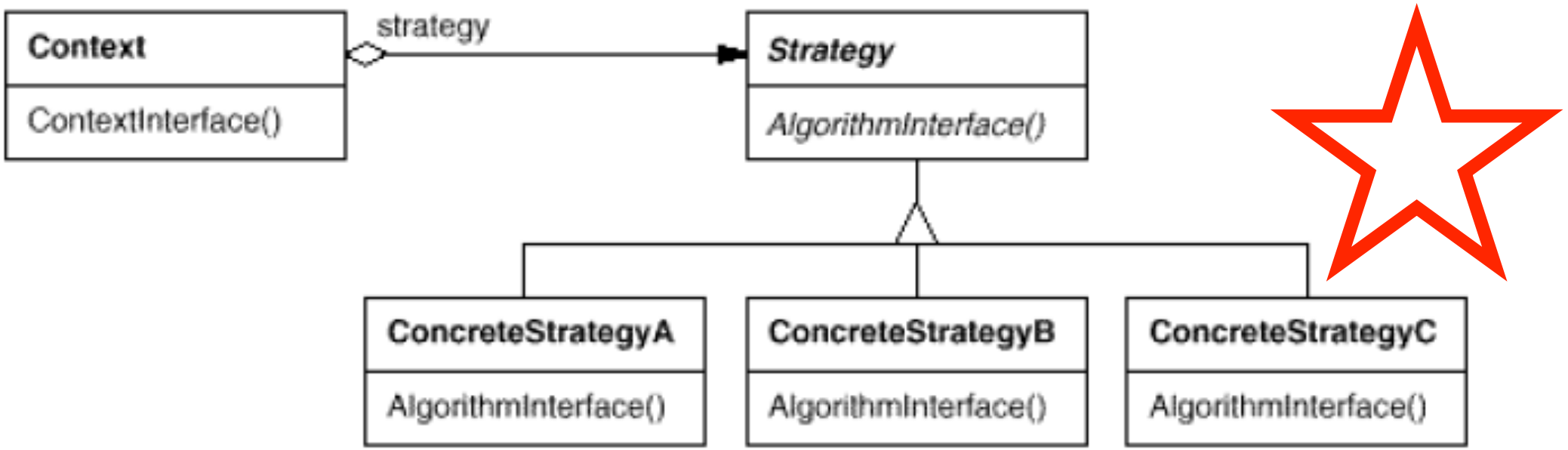
State Participants

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



- **Collaborations**
  - Context delegates state-specific requests to the current ConcreteState object.
  - A context may pass itself as an argument to the State object handling the request. This lets the
  - State object access the context if necessary.
  - Context is the primary interface for clients. Clients can configure a context with State objects.
  - Once a context is configured, its clients don't have to deal with the State objects directly.
  - Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

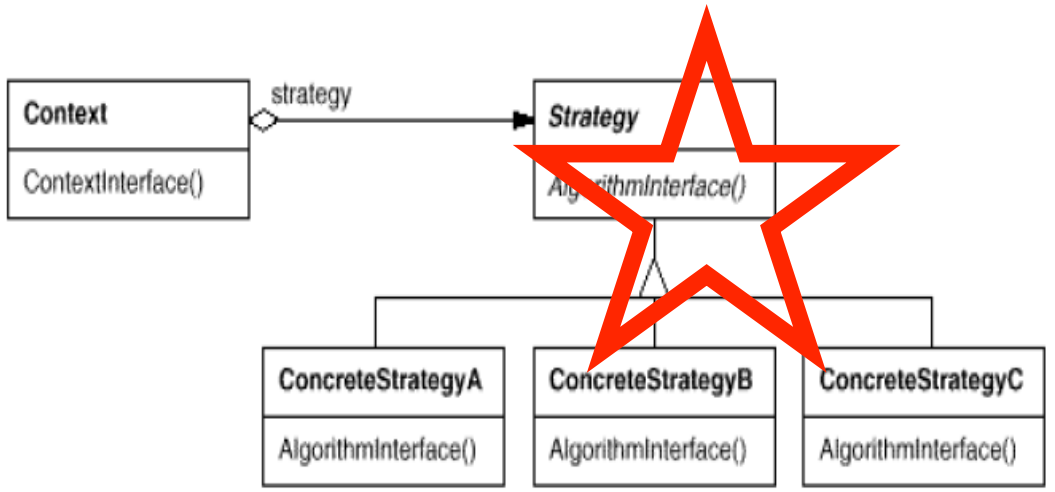
✓ Encapsulates a set of algorithms, which can vary independent of the client's interface



- The different states are instantiated in different derived classes, each with different behavior – providing a common interface through the AlgorithmInterface() function

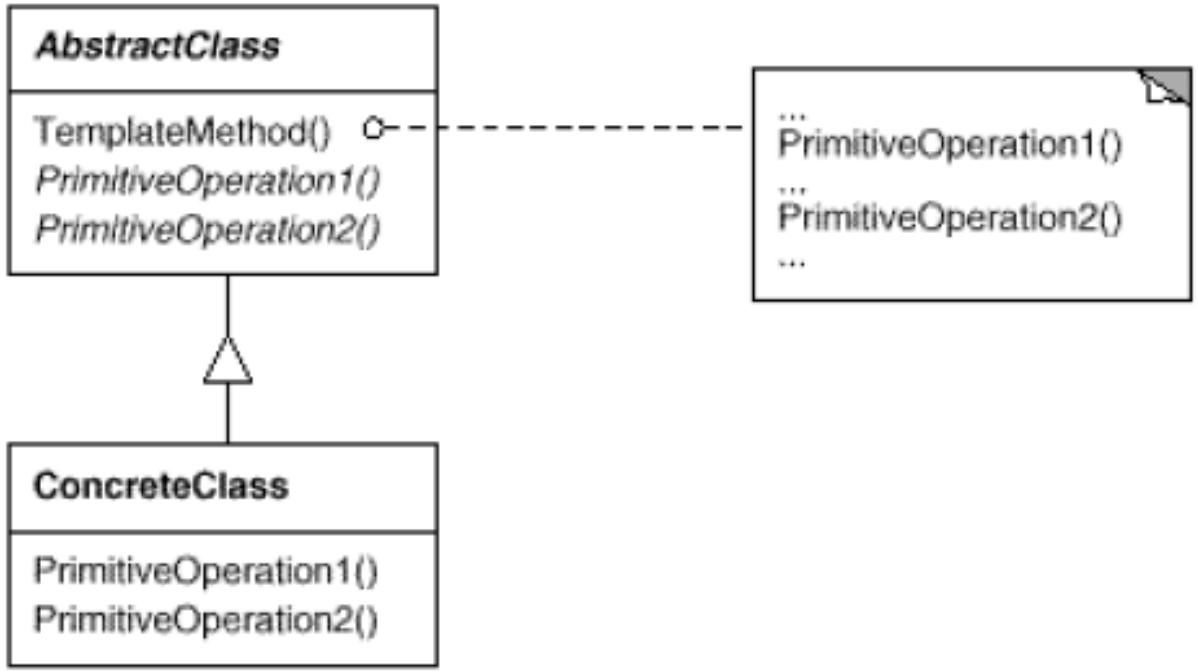
Strategy Participants

- **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**
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.



- **Collaborations**
  - Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
  - A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively.
  - There is often a family of ConcreteStrategy classes for a client to choose from.

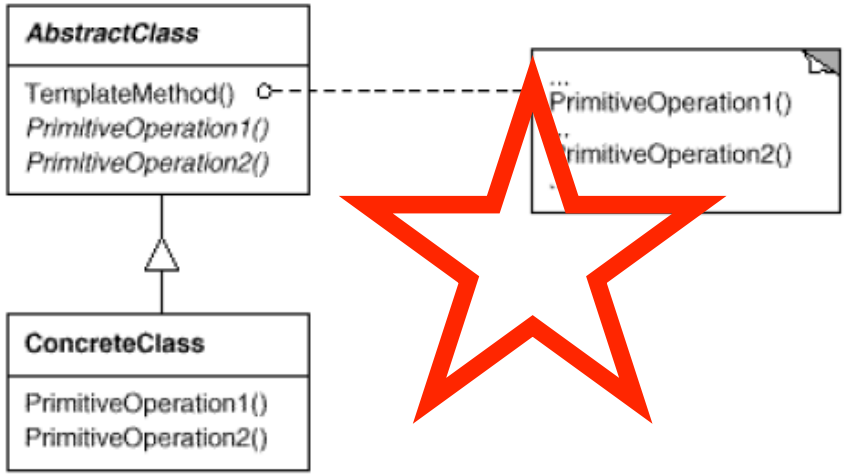
✓ Facilitates deferral of some implementation details until subclasses are implemented



- The Template Method pattern can be seen as pretty basic application of object-oriented design
- Polymorphism

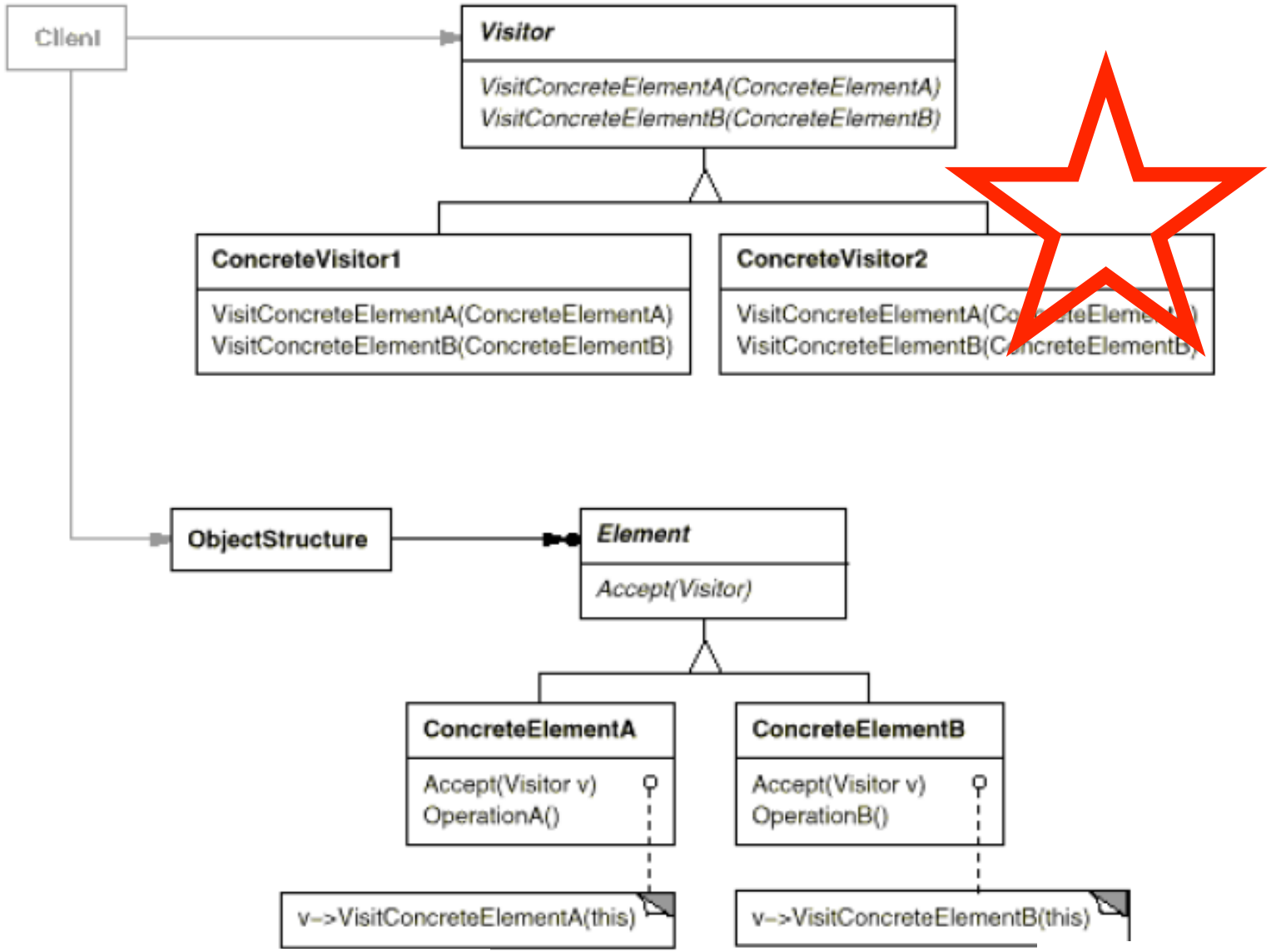
Template Method Participants

- **AbstractClass**
  - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass**
  - implements the primitive operations to carry out subclass-specific steps of the algorithm.



- **Collaborations**
  - ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

- ✓ Allows definition of some common operation on a whole collection of objects, using polymorphic referencing



Visitor Participants

- **Visitor**
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor.
  - That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.
- **ConcreteVisitor**
  - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- **Element**
  - defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement**
  - implements an Accept operation that takes a visitor as an argument.
- **ObjectStructure**
  - can enumerate its elements.
  - may provide a high-level interface to allow the visitor to visit its elements.
  - may either be a composite (see Composite) or a collection such as a list or a set

The Behavioral Patterns – a few comments

- Many of these patterns are about hiding elements that will change (or at least encapsulating changeable items)

- **Mediator** and **Observer** are opposites;
  - Mediator encapsulates communication, while
  - Observer distributes it

- Often a mediating object exists between a Sender and a Receiver

To-Do This Week (Week 3, Asn4: Design Pattern)

- **Discussion: Design Pattern**
  - Prepare a 7-10 slide presentation on a specific design patterns
  - Your presentation should follow this general form:
    - Slide 1 – Title slide
    - Slide 2 – Name of the pattern, intent and typical use
    - Slide 3 – The pattern's structure (participants and collaboration)
    - Slide 4 – When and how it would be used – an example if there is room and it makes sense
  - Here are the patterns that you will be addressing.
    - Adapter
    - Bridge
    - Decorator
    - Interpreter
    - Iterator
    - Observer

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

- **Behavioral Patterns**
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

EGR326 Software Design and Architecture

Lecture 4. Creational Patterns

Spring 2020

Kim Peters, Ph.D.

Gordon and Jill Bourns College of Engineering
California Baptist University

Week 2 Objectives

- Creational Patterns
  - Abstract Factory
  - Factory Method
  - Builder
  - Prototype
  - Singleton

To-Do Week 2 (Asn3: Project Selection)

- **Discussion:** Tell us about your group project!
- **Background Research** - SWOT Analysis
- **SOW** – Summarize the goal of your project in 15 words or less
- **Requirements**

- **Assignment (Asn 3):**
  - Background Research – limit to one page
  - SOW (Statement of Work) – limit to one sentence
  - Requirements
    - Focus on capturing functional requirements like writing user stories (Ex. “As a user, I can send a private message”)
  - Upload your work via blackboard >> EGR326 >> Assignment >> Asn 3
  - Due date: Friday, Jan 24 by 10:00pm

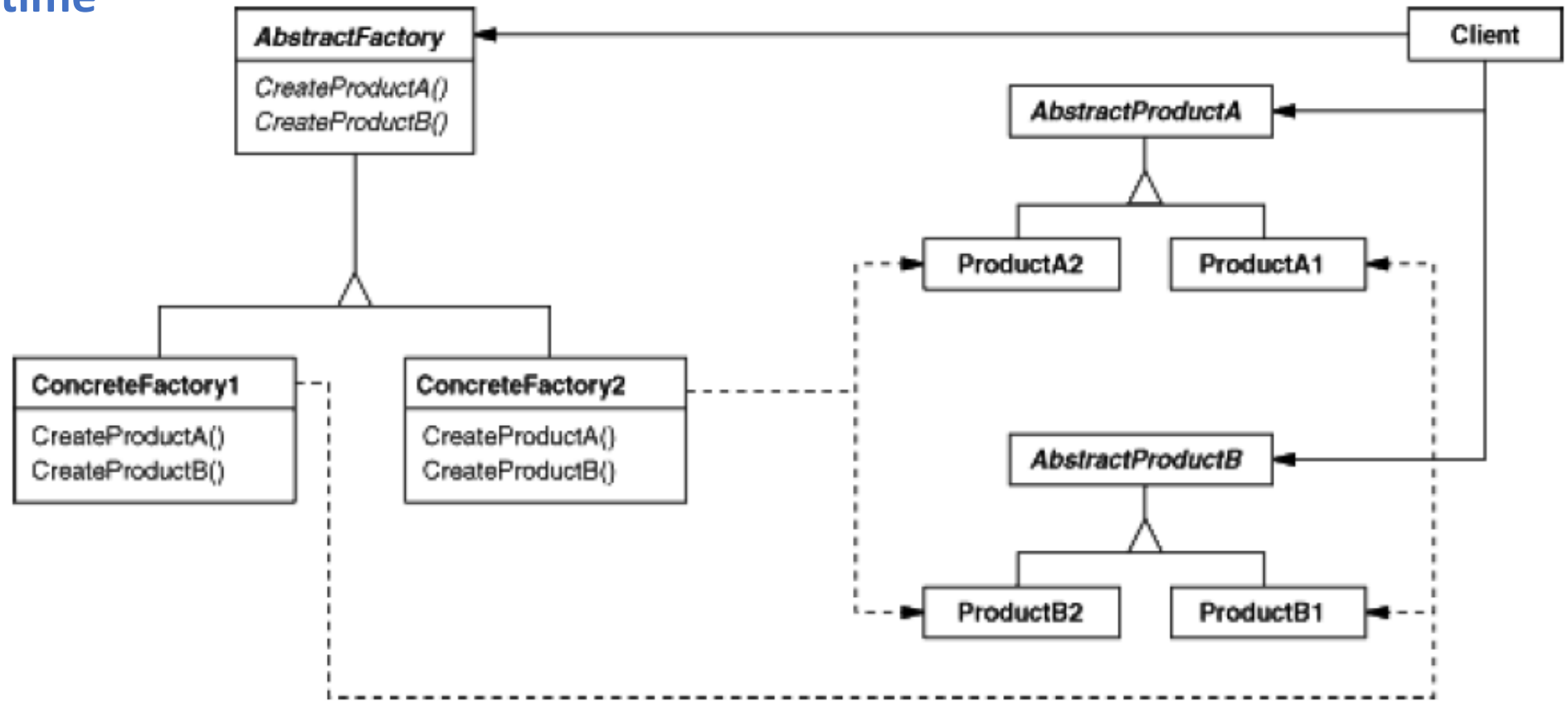
- ✓ **Creational Patterns are used to implement the instantiation process – generally providing greater abstraction**

- Keep implementation details as contained as possible
- Make it easy to change platform/components/design

- **Two common themes**:
  - **Encapsulate** knowledge about which concrete classes are being used.
  - **Hide** the way in which instances of these classes are created and put together.

- Interfaces are specified in one or more abstract classes, which derived classes must implement, but clients can then use.

- ✓ An AbstractFactory allows creation of different specializations of a base class, determined at run-time



- Clients only use the interfaces declared in the abstract classes
  - The concrete methods and products handle implementation-specific details

✓ The participants in AbstractFactory work together to create real objects of different types

- **AbstractFactory**
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory**
  - implements the operations to create concrete product objects.
- **AbstractProduct**
  - declares an interface for a type of product object.
- **ConcreteProduct**
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

- ✓ **A ConcreteFactory will use the inherited interface to return new objects of the concrete type, through a pointer (reference) to the abstract type**
  - The factory can create a variety of types

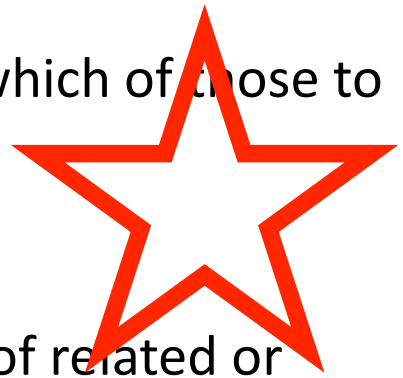
```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();
    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

✓ An AbstractFactory is like a Pasta machine (thanks to Brent Ramerth for this analogy)

- Your code is the pasta maker
- Different disks create different pasta shapes: these are the concrete factories
- All disks have certain properties in common, so that they will work with the pasta maker (these are specified by the abstract class)
- All pastas have certain characteristics in common that they inherit from the generic "Pasta" object

✓ What are the differences?

- **FactoryMethod**
  - defines an interface for creating an object, but lets subclasses decide which of those to instantiate.
  - lets classes defer instantiation to subclasses.
- By contrast, an **AbstractFactory** provides an interface for creating families of related or dependent objects without specifying their concrete classes."

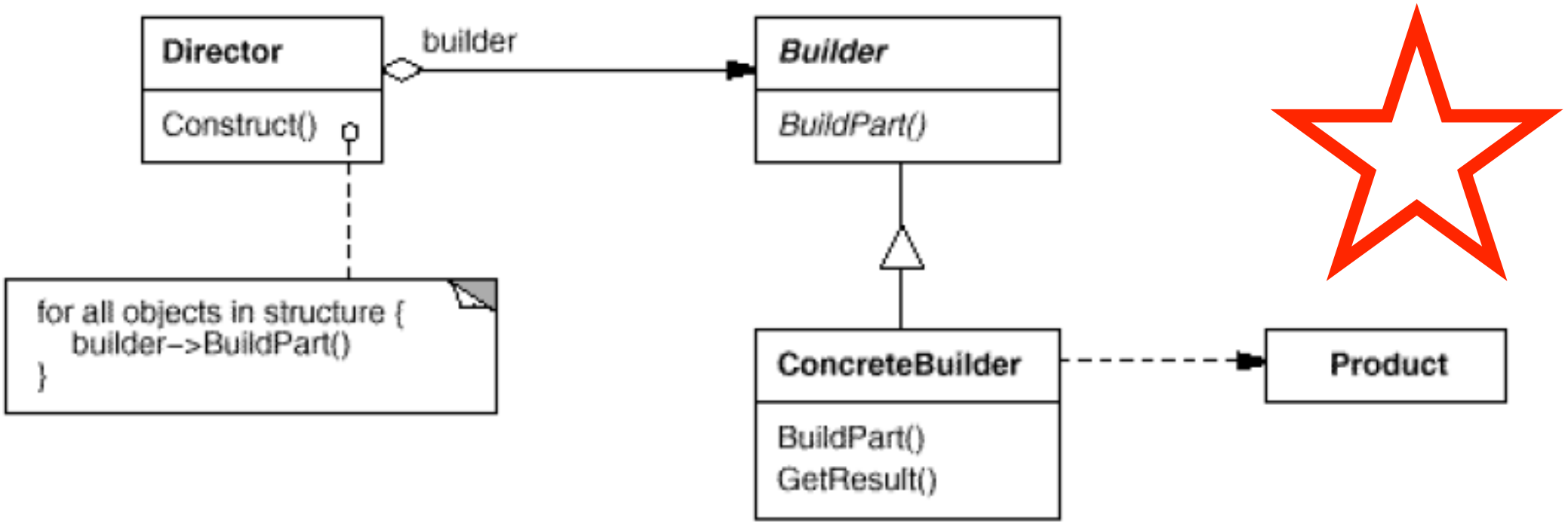


A. **FactoryMethod** instantiates one class, **AbstractFactory** provides creation for a whole related family

B. **FactoryMethod** instantiates what its derived class is designed for; **AbstractFactory** uses composition (what is called or passed in) to determine what to create.

✓**AbstractFactory** is a higher level of abstraction (hence the name…)

- ✓ The Builder separates the construction of an object from its representation details



✓ Builder's participants allow creation of concrete objects through the abstract interface

- **Builder**
  - specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder**
  - constructs and assembles parts of the product by implementing the Builder interface.
  - defines and keeps track of the representation it creates.
  - provides an interface for retrieving the product (e.g., GetASCIIText, GetTextWidget).
- **Director**
  - constructs an object using the Builder interface.
- **Product**
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Factory vs. Builder

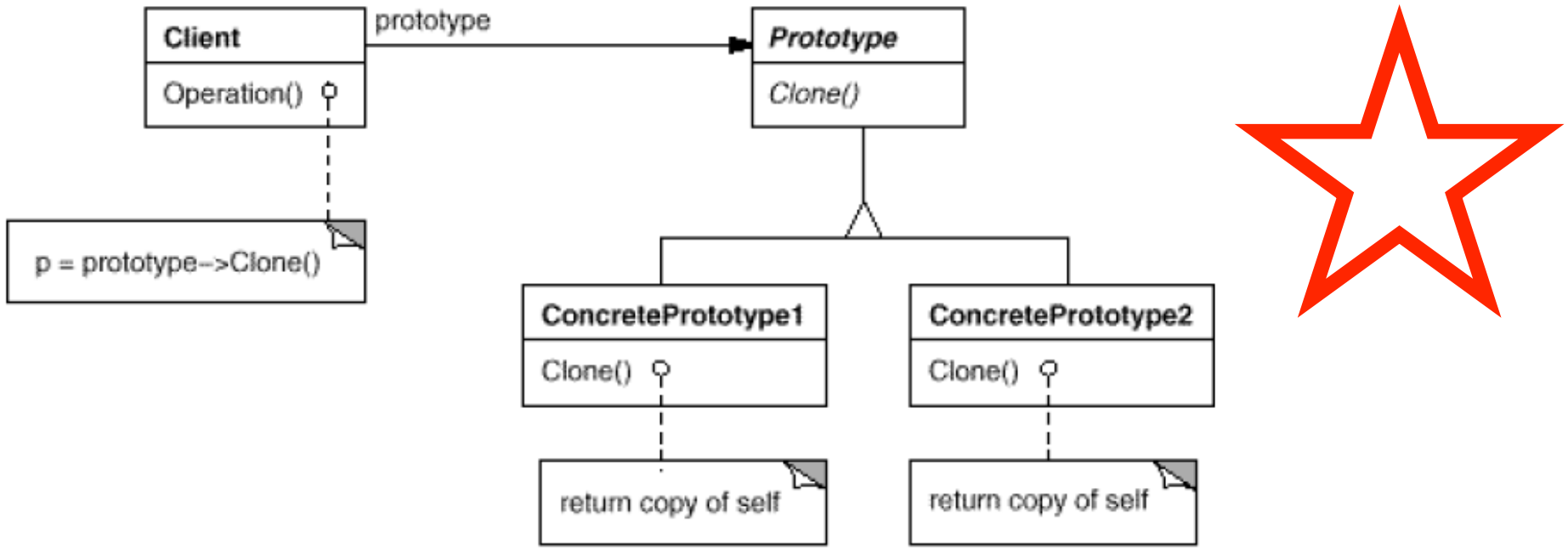
✓ Each has their uses

```
// Factory
static class FruitFactory {
    static Fruit create(name, color, firmness) {
        // Additional logic
        return new Fruit(name, color, firmness);
    }
}
// Usage
Fruit fruit = FruitFactory.create("apple", "red",
"crunchy");
```

```
// Builder
class FruitBuilder {
  String name, color, firmness;
  FruitBuilder setName(name) { this.name = name; return this; }
  FruitBuilder setColor(col) { this.color = col; return this; }
  FruitBuilder setFirmness(f) { this.firmness = f; return this;}
  Fruit build() {
    return new Fruit(this); // Pass in the builder
  }
}
// Usage
Fruit fruit = new FruitBuilder().setName("apple")
        .setColor("red")
        .setFirmness("crunchy")
        .build();
```

- **Factory** is usually used to create a simple object
- **Builder** is more related to building a complex object with many parts – an XML document, for example

✓ Prototype creates new objects by copying a sample instance or "prototype"



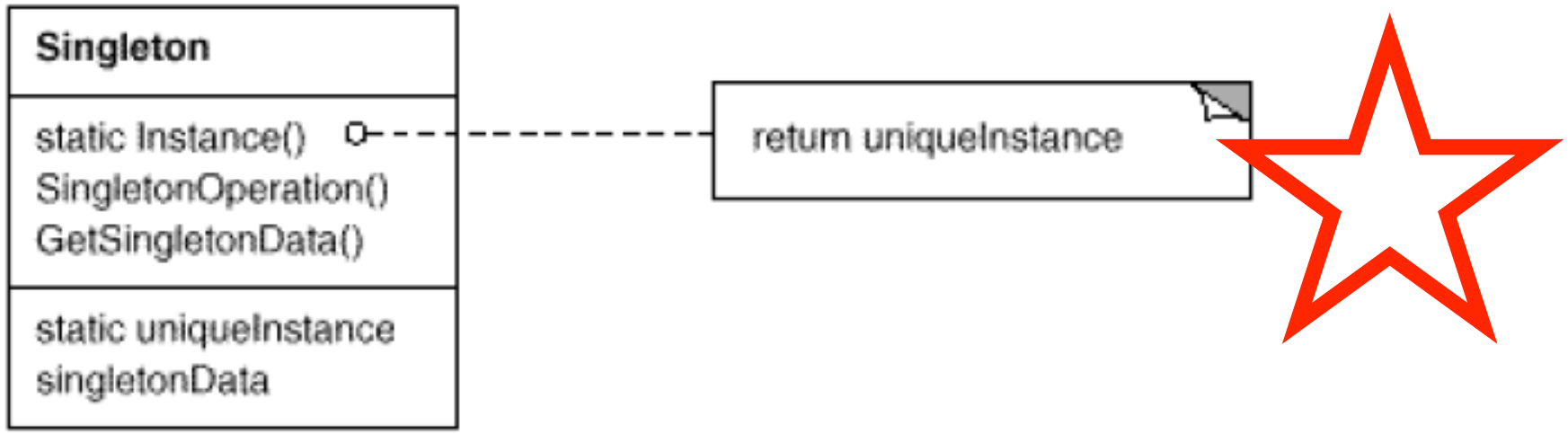
- Calls to the Prototype's interface for a certain concrete object will return a copy of object

- ✓ **In the Prototype pattern, the abstract class essentially specifies an interface for concrete classes to clone themselves**

- **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.

- **Collaborations**
  - A client asks a prototype to clone itself.

✓ The Singleton pattern shows how to create a class for which there can only be one instance



- **uniqueInstance** is the actual object – created by a call to *Instance()*
  - the constructor is protected
- A pointer or reference to the uniqueInstance is returned by *Instance()* and is used to access the object

- ✓ Sample code for Singleton – the Instance() method returns a pointer to the actual object, and creates it if needed

```
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;        }
```

- ✓ There are three reasons to use a Singleton; access control, distributed use and only one instance
  - **A Singleton may be right if:**
    - there can be only one object;
    - the object controls concurrent access to a shared resource;
    - access to the resource will be requested from multiple, disparate parts of the system.

  - **Might use a Singleton for a system logging object.**
  - **This pattern is controversial** – some say that it's unnecessary in well-written code.

- Very often, a **ConcreteFactory** will be a **Singleton**
  - Makes little sense to have multiple factory objects for the same concrete class

- In the same project, some objects might use a **Builder** while others use an **AbstractFactory**

- **AbstractFactory** often uses **FactoryMethod**

Review Objectives

- Creational Patterns
  - Abstract Factory
  - Factory Method
  - Builder
  - Prototype
  - Singleton

To-Do Week 2 (Asn3: Project Selection)

- **Discussion:** Tell us about your group project!
- **Background Research** - SWOT Analysis
- **SOW** – Summarize the goal of your project in 15 words or less
- **Requirements**

- **Assignment (Asn 3):**
  - Background Research – limit to one page
  - SOW (Statement of Work) – limit to one sentence
  - Requirements
    - Focus on capturing functional requirements like writing user stories (Ex. “As a user, I can send a private message”)
  - Upload your work via blackboard >> EGR326 >> Assignment >> Asn 3
  - Due date: Friday, Jan 24 by 10:00pm

EGR326 Software Design and Architecture
Lecture 3. The Document Editor Example

Spring 2020

Kim Peters, Ph.D.

Gordon and Jill Bourns College of Engineering
California Baptist University

Week 2 Objectives

- The Lexi Problem
- Design problems we must tackle:
  - Document structure
  - Formatting
  - User interface additions
  - Multiple look-and-feel standards
  - Multiple window systems
  - Uniformity of user interactions
  - Spelling checking and auto-hyphenation.
- Application of Design Patterns

To-Do Week 2 (Asn3: Project Selection)

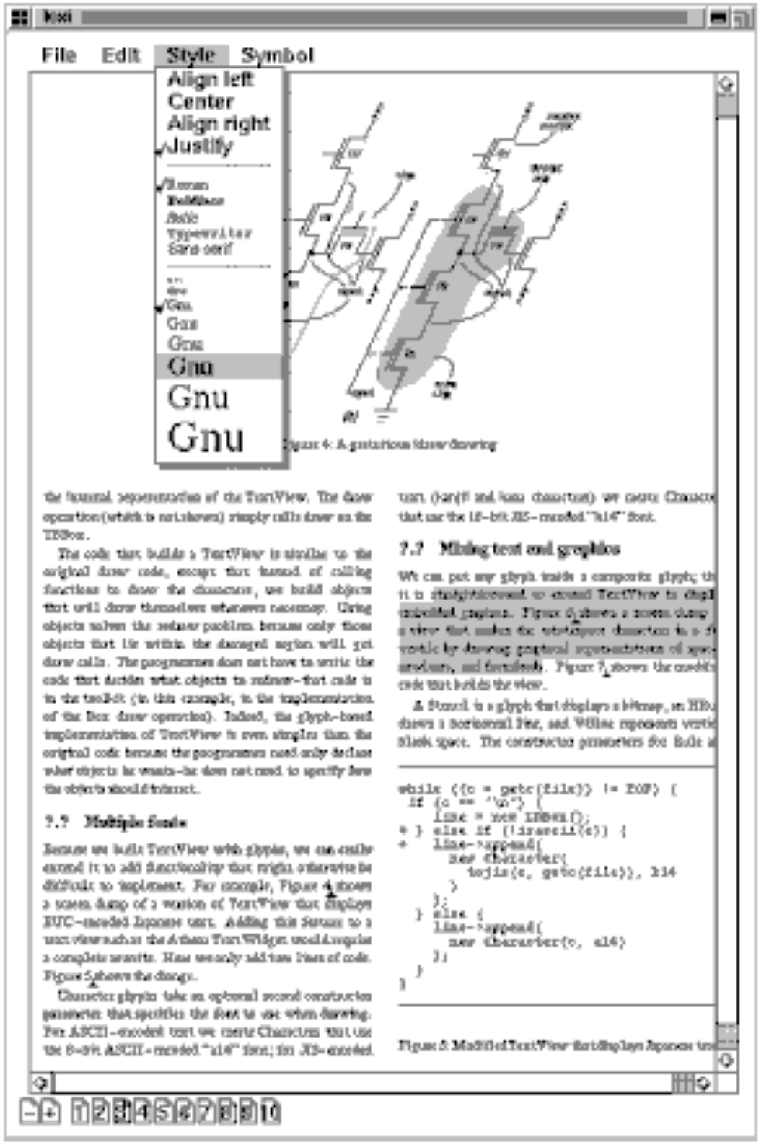
- **Discussion:** Tell us about your group project!
- **Background Research** - SWAT Analysis
- **SOW** – Summarize the goal of your project in 15 words or less
- **Requirements**

- **Assignment (Asn 3):**
  - Background Research – limit to one page
  - SOW (Statement of Work) – limit to one sentence
  - Requirements
    - Focus on capturing functional requirements like writing user stories (Ex. “As a user, I can send a private message”)
  - Upload your work via blackboard >> EGR326 >> Assignment >> Asn 3
  - Due date: Friday, Jan 24 by 10:00pm

The Lexi Problem

- ✓ The authors introduce the use of Design Patterns by walking through the design of a document editor they call "Lexi"

  - Like most editors these days, Lexi is to be "what you see is what you get" (WYSIWYG)
  - Pull-down menus will access most functions
  - Small page icons at the bottom will aid navigation

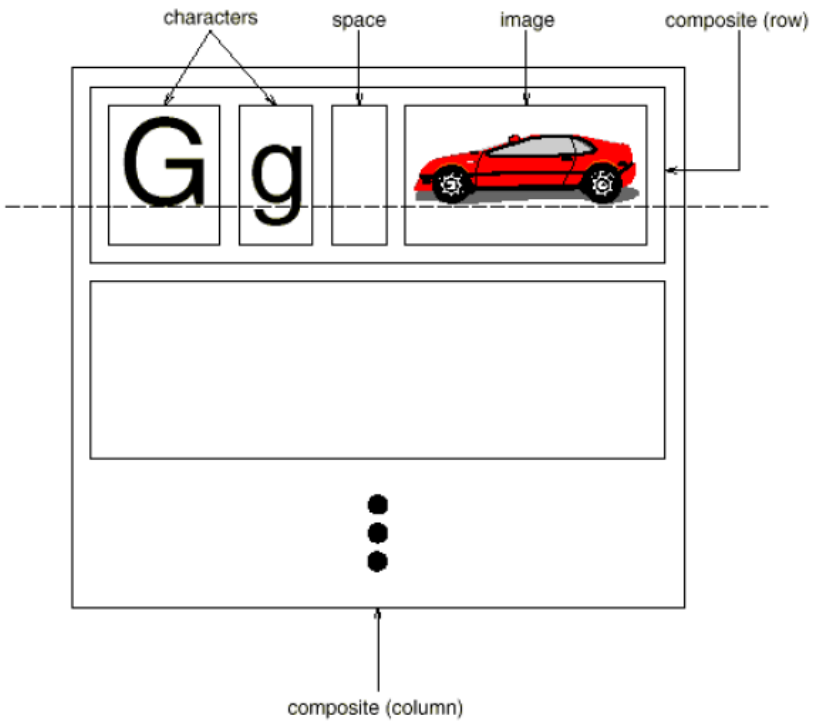


✓ There are seven design problems to be solved

1) **Document structure** – how is a document represented internally?
2) **Formatting** – how are screen layouts and fonts dictated; what objects handle this?
3) **User interface additions** – likely revisions include added functions and tools.
4) **Multiple look-and-feel standards** – we want to support different appearance and interaction models.
5) **Multiple window systems** – the interaction with X11 and so on should be flexible and contained in a few places.
6) **Uniformity of user interactions** – throughout the program.
7) **Spelling checking** and auto-hyphenation.

1) Document structure – how is a document represented internally?

- Documents are a mix of text and graphics
- We will use recursive composition; elements are composed of smaller elements, which are themselves composed of smaller elements…



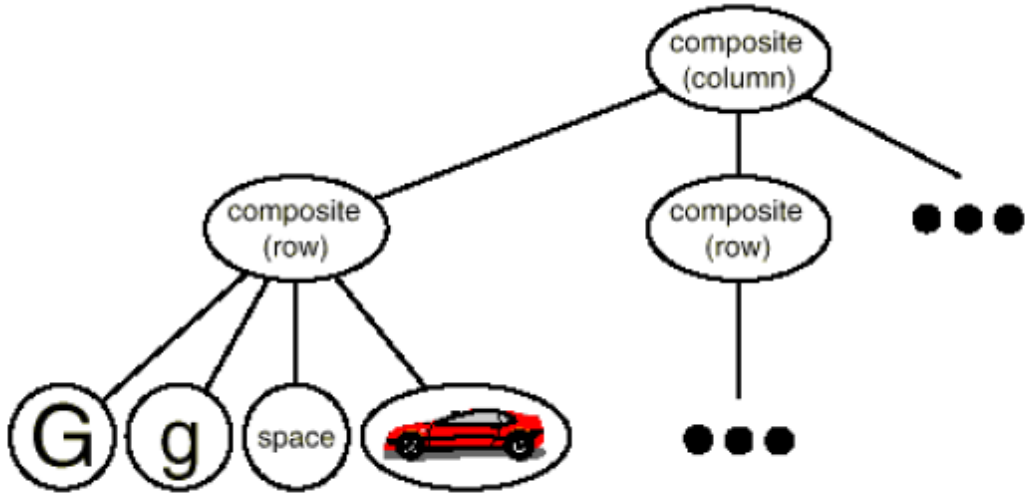


Figure 2.3: Object structure for recursive composition of text and graphics

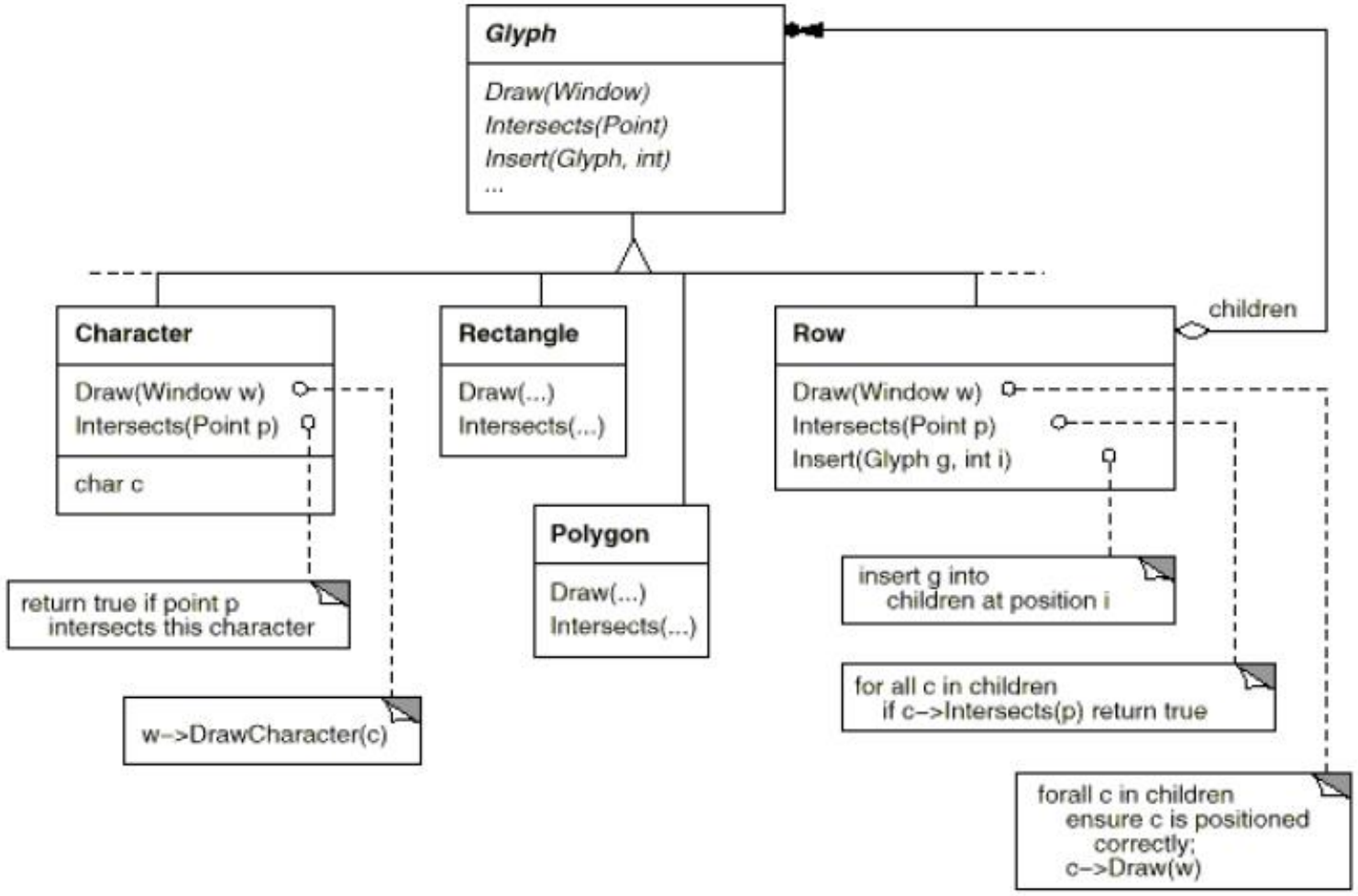
1) Document structure – how is a document represented internally?

- It turns out that it's easier to manage a hierarchy like this if all elements have a common interface of some kind; achieve this by inheriting from a common base class

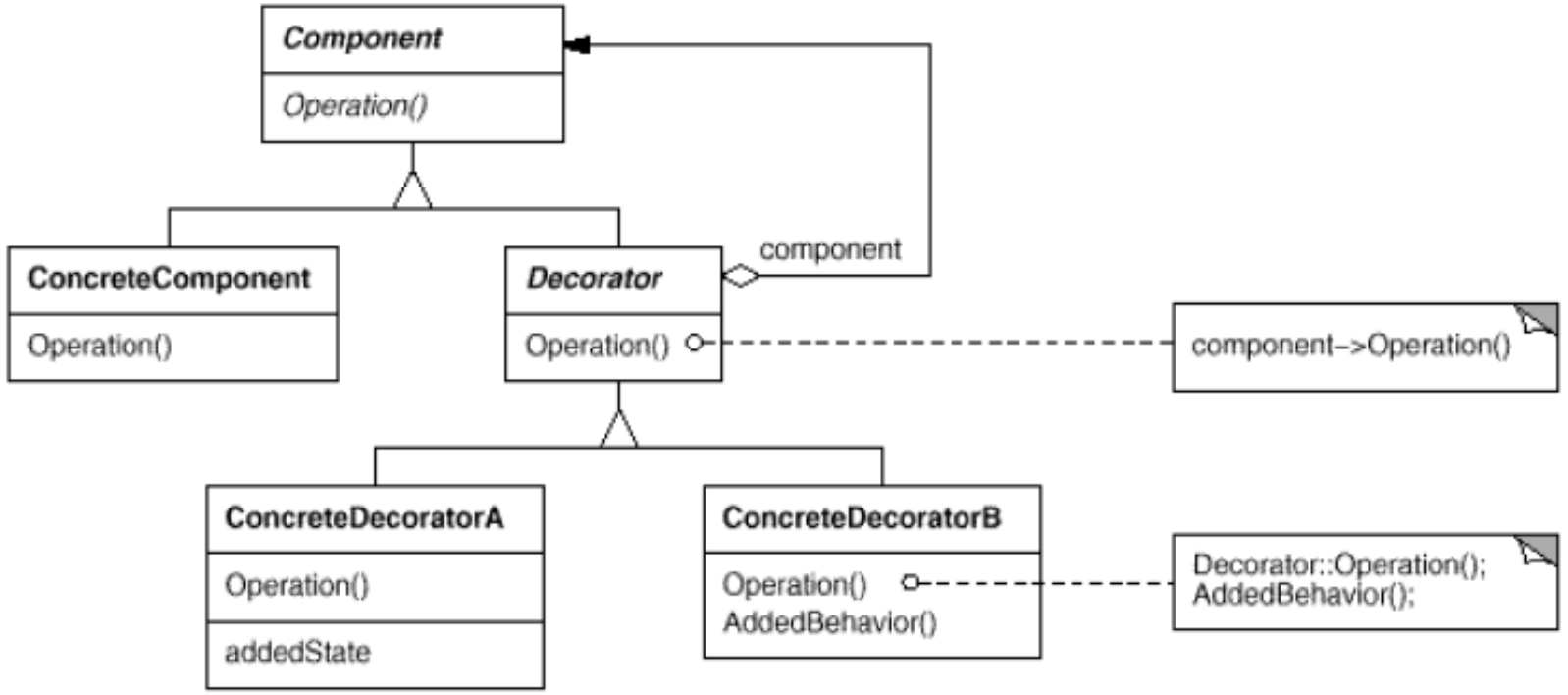
  - **Glyph** is an abstract class that serves as a base for any object that appears in a document structure.
    - Character will inherit from Glyph
    - Image will inherit from Glyph
    - etc.

  - Glyphs have three basic responsibilities.
    - They know:
      1 - how to draw themselves,
      2 - what space they occupy, and
      3 - their children and parent.

1) Document structure – how is a document represented internally?
   - A partial class hierarchy of the Glyph base class shows how this will work to allow all children of Glyph to behave similarly



1) Document structure – how is a document represented internally?
   - The ***Composite*** Pattern embodies the hierarchical relationship that we want



✓A quick look would show that the structure of the ***Composite*** pattern is what we are trying to accomplish.

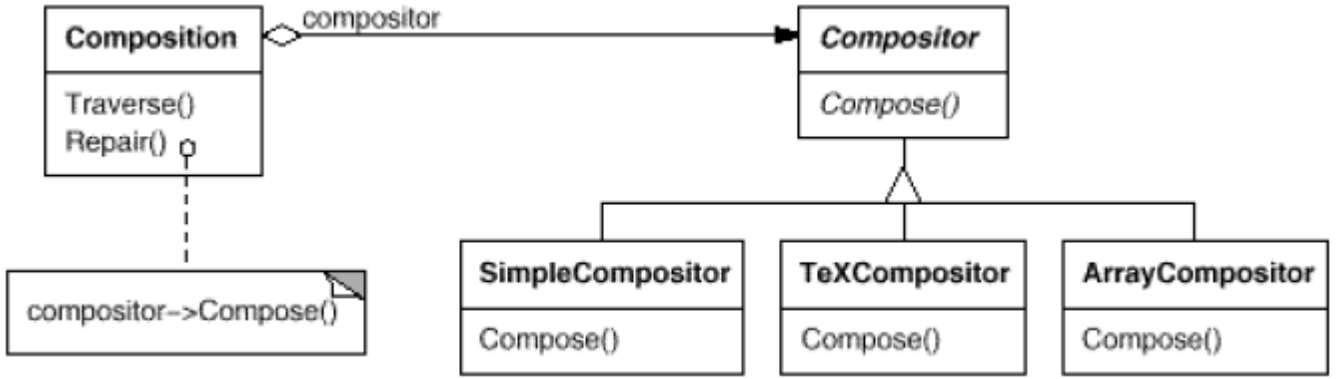
2) Formatting – how are screen layouts and fonts dictated; what objects handle this?

- Note: internal representation and document formatting are distinct.
- Formatting itself (determining word-wrap, line breaks and so on) is complex; there will probably be a sophisticated algorithm at work here.
- We want to define a way for an algorithm to access and format the document, or a part of it, while maintaining good encapsulation.

2) Formatting – how are screen layouts and fonts dictated; what objects handle this?

- The ***Strategy*** pattern is used to encapsulate an algorithm; in our application, it will provide an interface to support Composing (or formatting) a Composition (or part of a document)

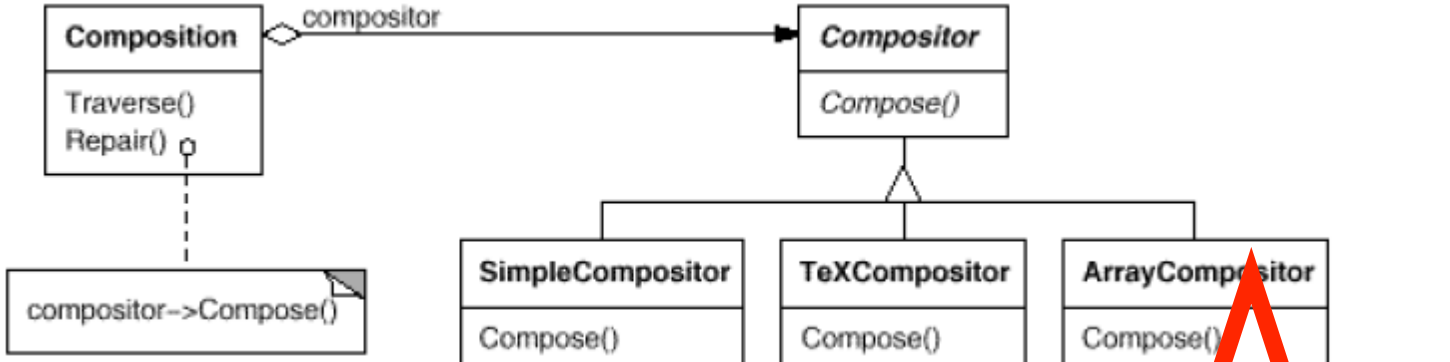
Specific



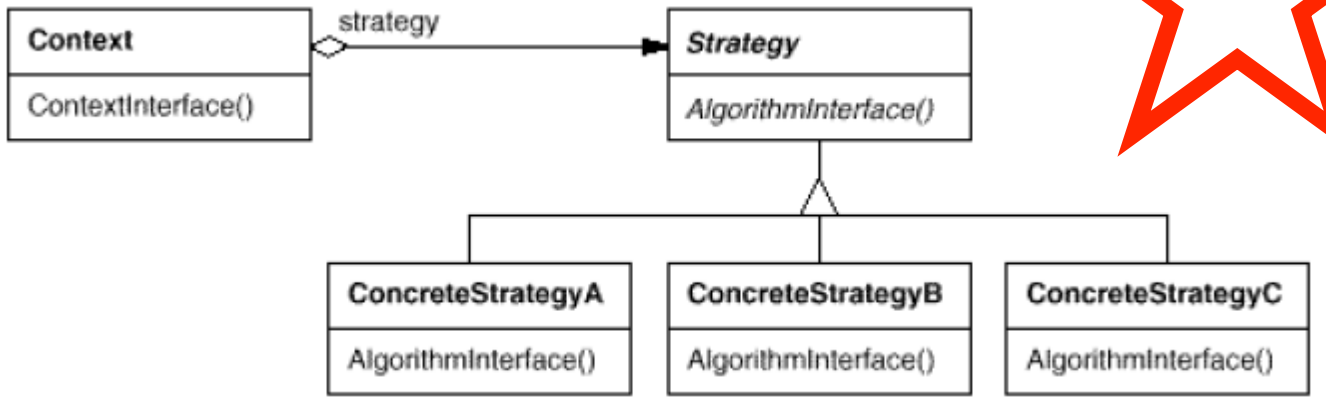
2) Formatting – how are screen layouts and fonts dictated; what objects handle this?

- The ***Strategy*** pattern is used to encapsulate an algorithm; in our application, it will provide an interface to support Composing (or formatting) a Composition (or part of a document)

Specific

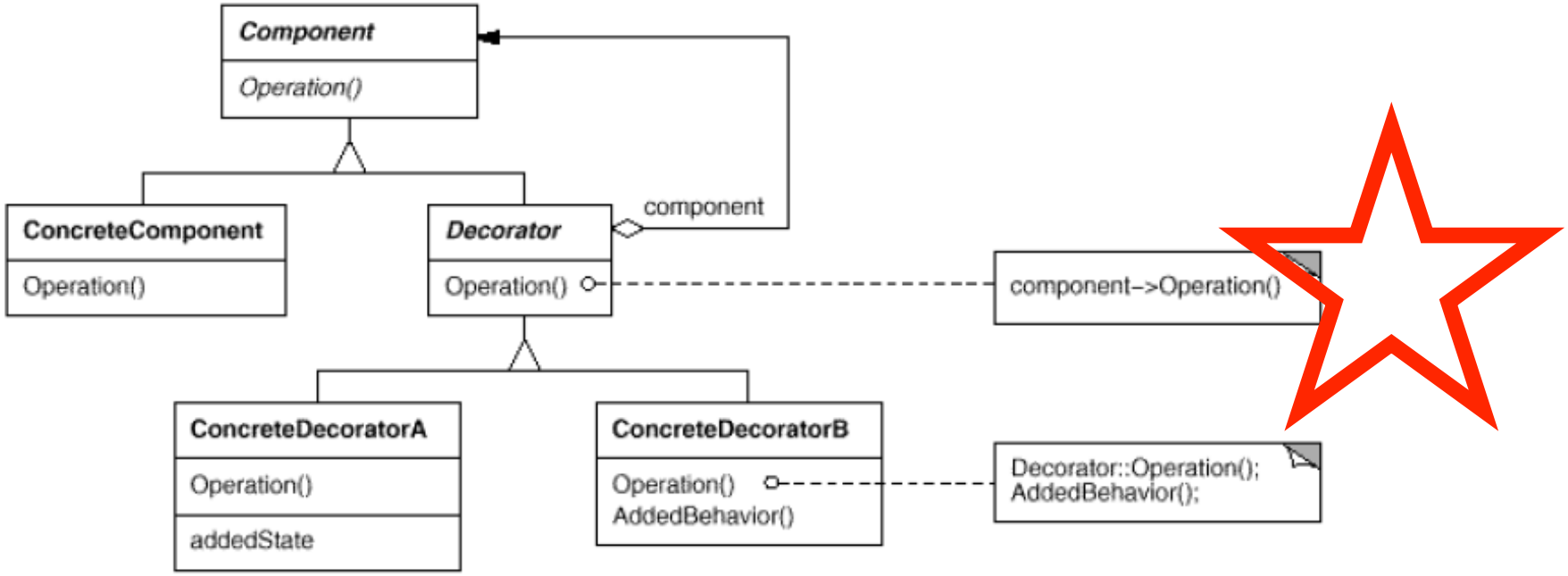


General



3) User interface additions – likely revisions include added functions and tools

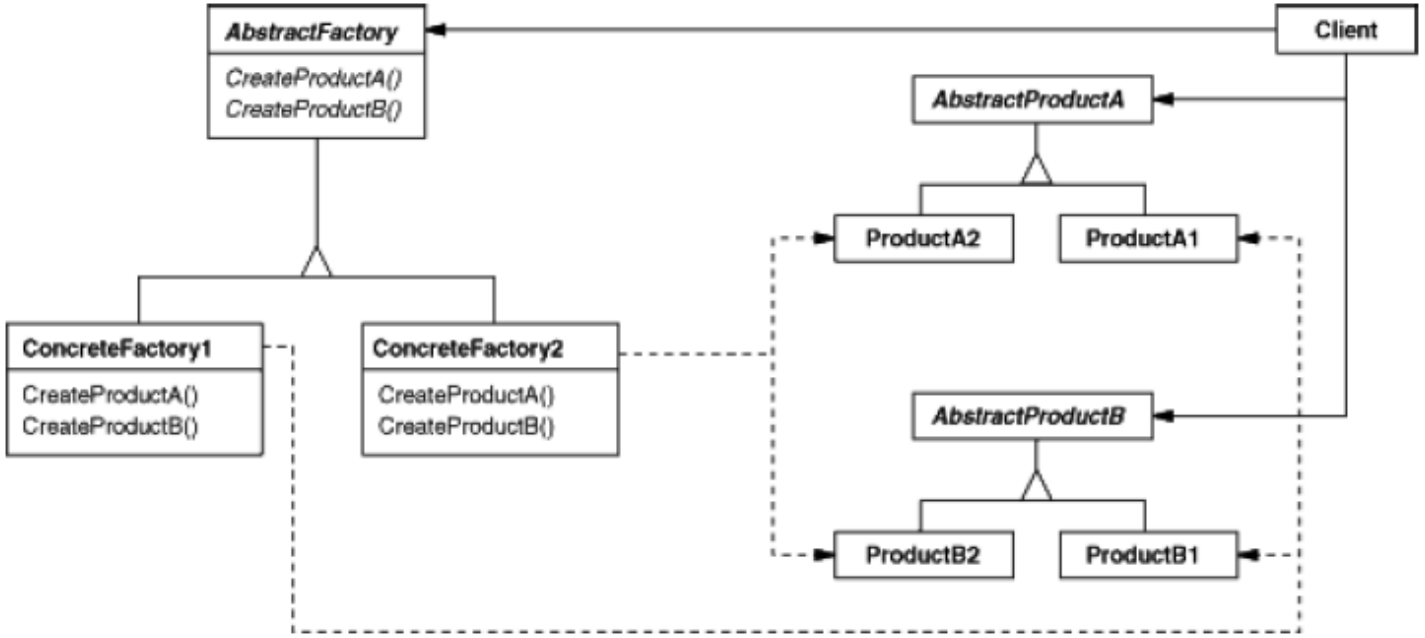
- We will want to add new pieces to the display and functionality
  - Not only to allow future expansion and maintenance, but –
  - because it supports agile development.
- The ***Decorator*** pattern allows us to add responsibilities to a core object easily…



4) Multiple look-and-feel standards

– We want to support different appearance and interaction models

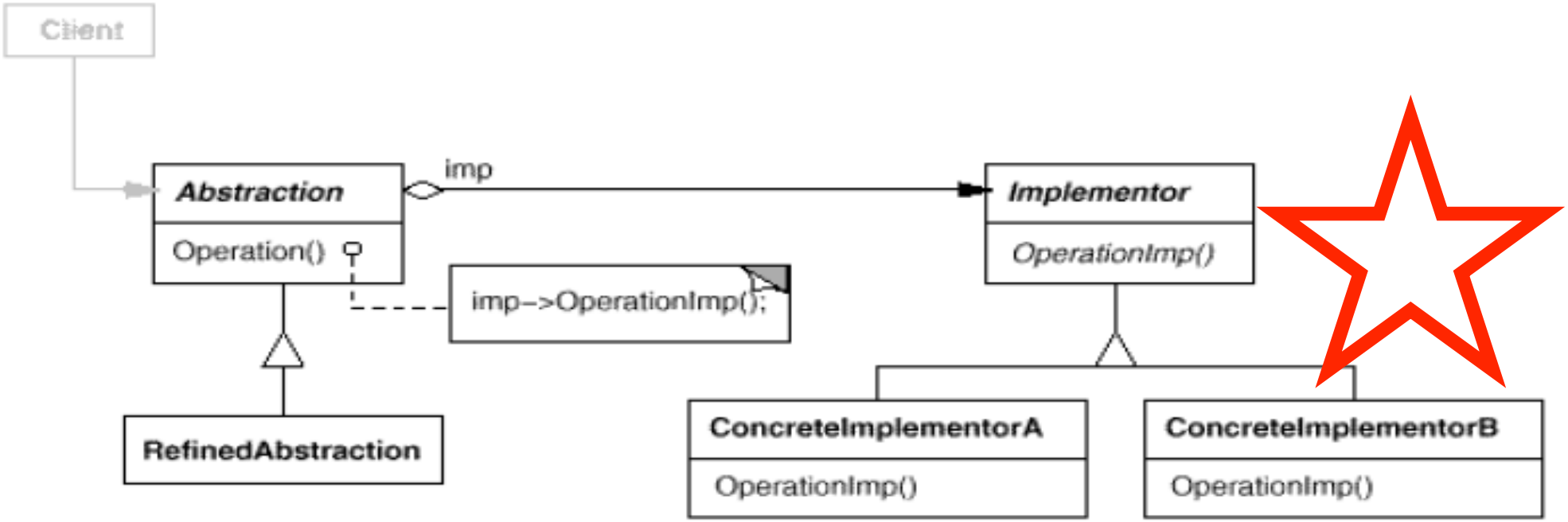
- Look-and-feel is determined by objects, so we want to allow creation of different types of objects at run time!
- This is only practical through polymorphism.
- This is supported by the ***Abstract Factory*** pattern:



5) Multiple window systems

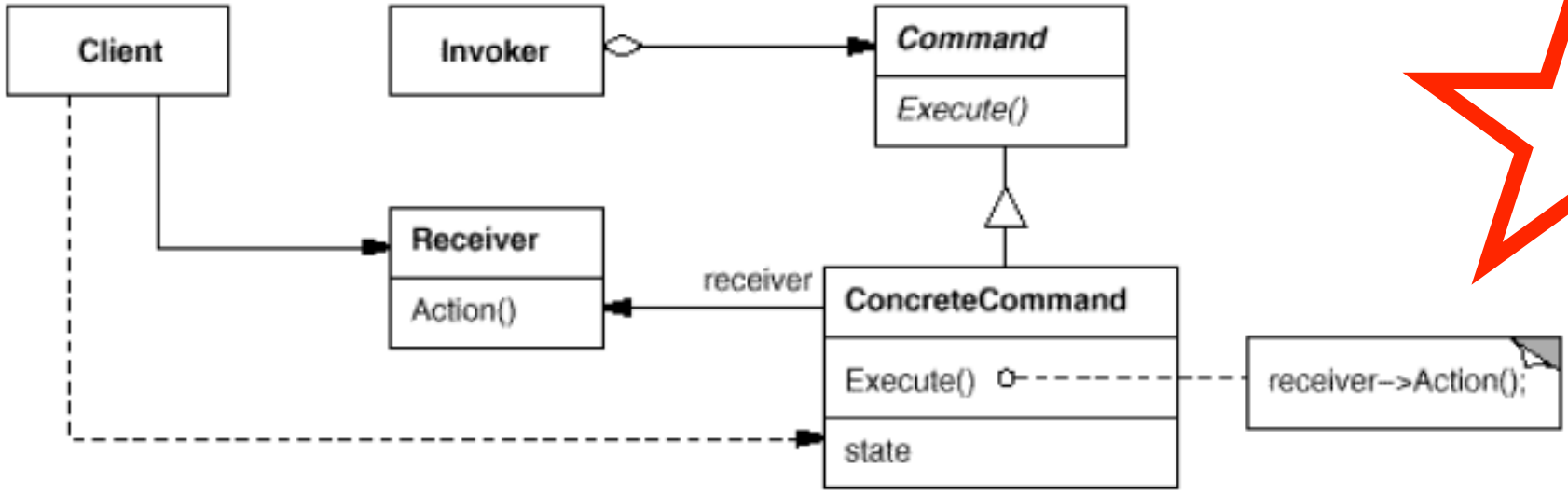
– the interaction with X11 and so on should be flexible and contained in a few places

- The key here is to manage the interaction between common routines (like Draw()) and platform-specific code
- This can be done using a ***Bridge*** pattern.



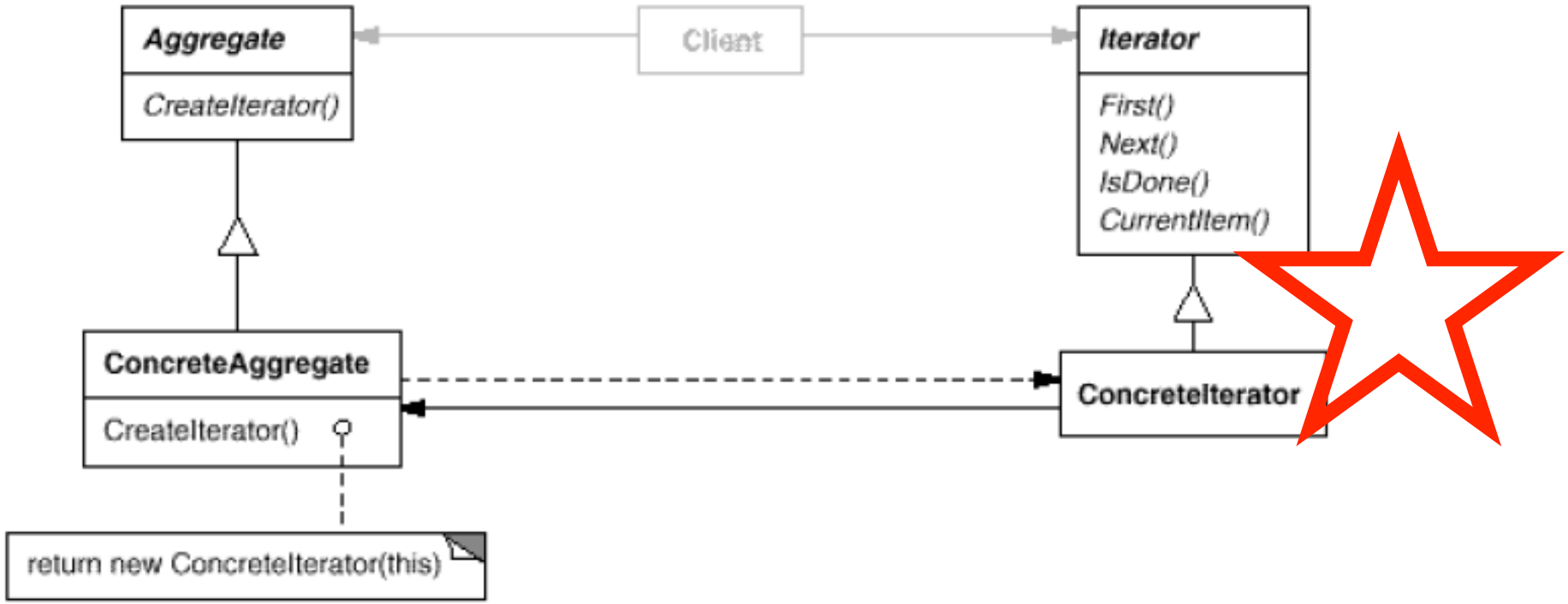
6) Uniformity of user interactions – *throughout the program*

- As in most editors, we want multiple ways to perform common operations
  - Ctrl-C, Edit…Copy, F12, etc…
- Undo requires a stack of past operations
- Some operations don't always make sense
- The ***Command*** pattern will encapsulate requests:



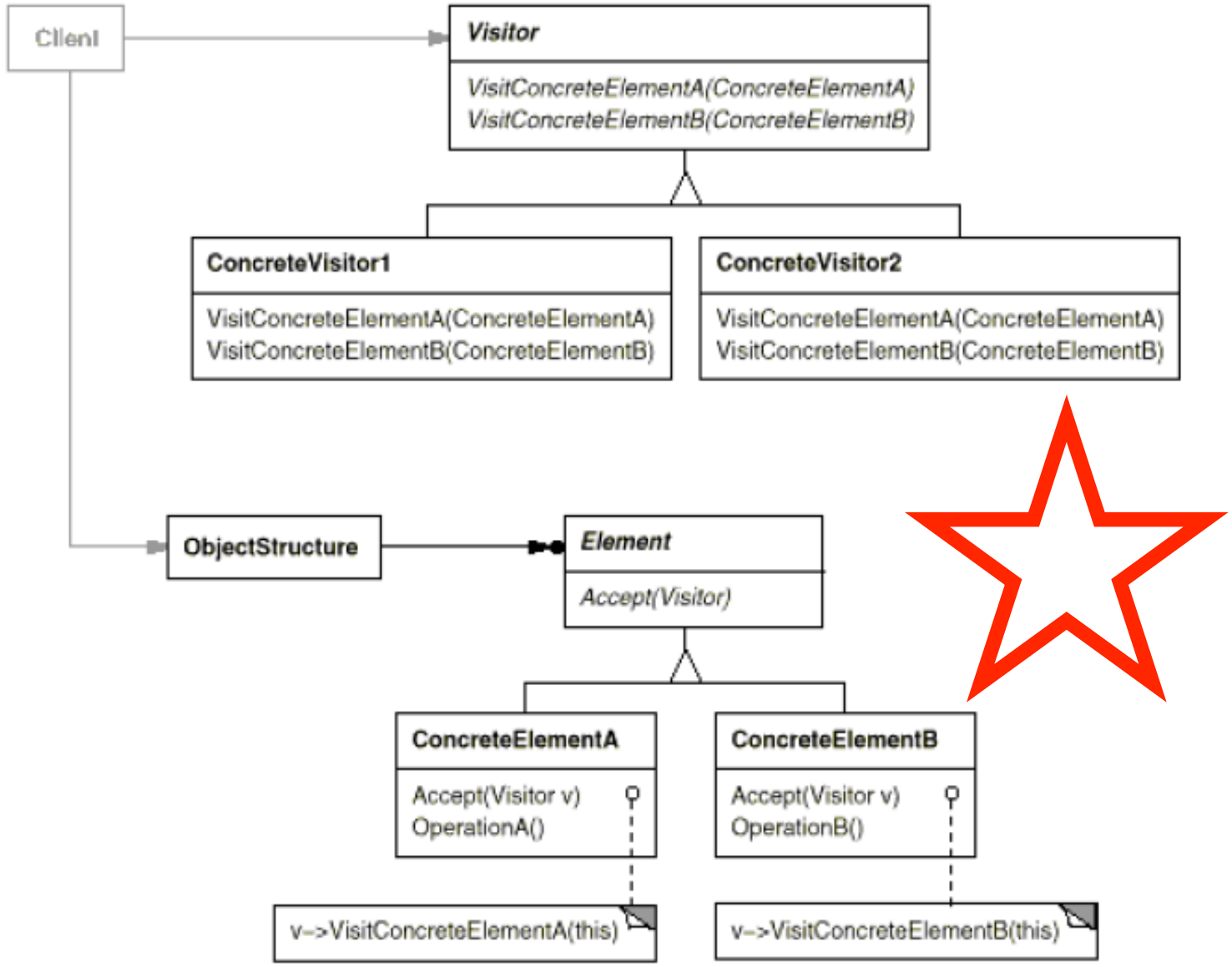
7) Spelling checking and auto-hyphenation

- Document operations like this will access data scattered throughout
  - as well as dictionaries
- The ***Iterator*** pattern supports traversal through object containers



7) Spelling checking and auto-hyphenation

- ✓ The ***Visitor*** pattern describes designs where one object wil interact with a collection of others without implementation knowledge



✓ Though we have identified eight patterns to use in our design, we are nowhere near "done" with the design

- Each pattern needs application-specific names
- Interactions between patterns?
- Not all functionality is in place yet, just the core of some of it
- Many patterns require app-specific class hierarchies
  - and those must be consistent

- Nevertheless, we have a lot of our design done, with proven working techniques, and sample code we can refer to!

- ✓ Though we have identified eight patterns to use in our design, we are nowhere near "done" with the design

  - The Lexi Problem
  - Design problems we must tackle:
    - Document structure
    - Formatting
    - User interface additions
    - Multiple look-and-feel standards
    - Multiple window systems
    - Uniformity of user interactions
    - Spelling checking and auto-hyphenation.
  - Application of Design Patterns

To-Do Week 2 (Asn3: Project Selection)

- **Discussion:** Tell us about your group project!
- **Background Research** - SWAT Analysis
- **SOW** – Summarize the goal of your project in 15 words or less
- **Requirements**

- **Assignment (Asn 3):**
  - Background Research – limit to one page
  - SOW (Statement of Work) – limit to one sentence
  - Requirements
    - Focus on capturing functional requirements like writing user stories (Ex. “As a user, I can send a private message”)
  - Upload your work via blackboard >> EGR326 >> Assignment >> Asn 3
  - Due date: Friday, Jan 24 by 10:00pm

- **All work shall have cover page with:**
  - Document title
  - Course description
  - Document date
  - Your full name
  - Your team member names with last names
    - in alphabetical order (group assignments)

✓ ***Upmost professionalism!***

- **File name must conform to the following:**
  - team# _assignment#.ext (doc, xls. mpp, ppt, etc.), Ex) group1_asn1.ppt
  - lastname_assignment#.ext (doc, xls. mpp, ppt, etc.) Ex) peters_asn1.ppt

EGR326 Software Design and Architecture

Lecture 2. Introduction to Design Patterns

Spring 2020

Kim Peters, Ph.D.

Gordon and Jill Bourns College of Engineering
California Baptist University

Week 2 Objectives

- What is a Design Pattern?
- Describing Design Patterns
- How Design Patterns Solve Problems
- Why Design Pattern?
- Selecting a Design Pattern
- Using a Design Pattern

To-Do Week 2 (Asn3: Project Selection)

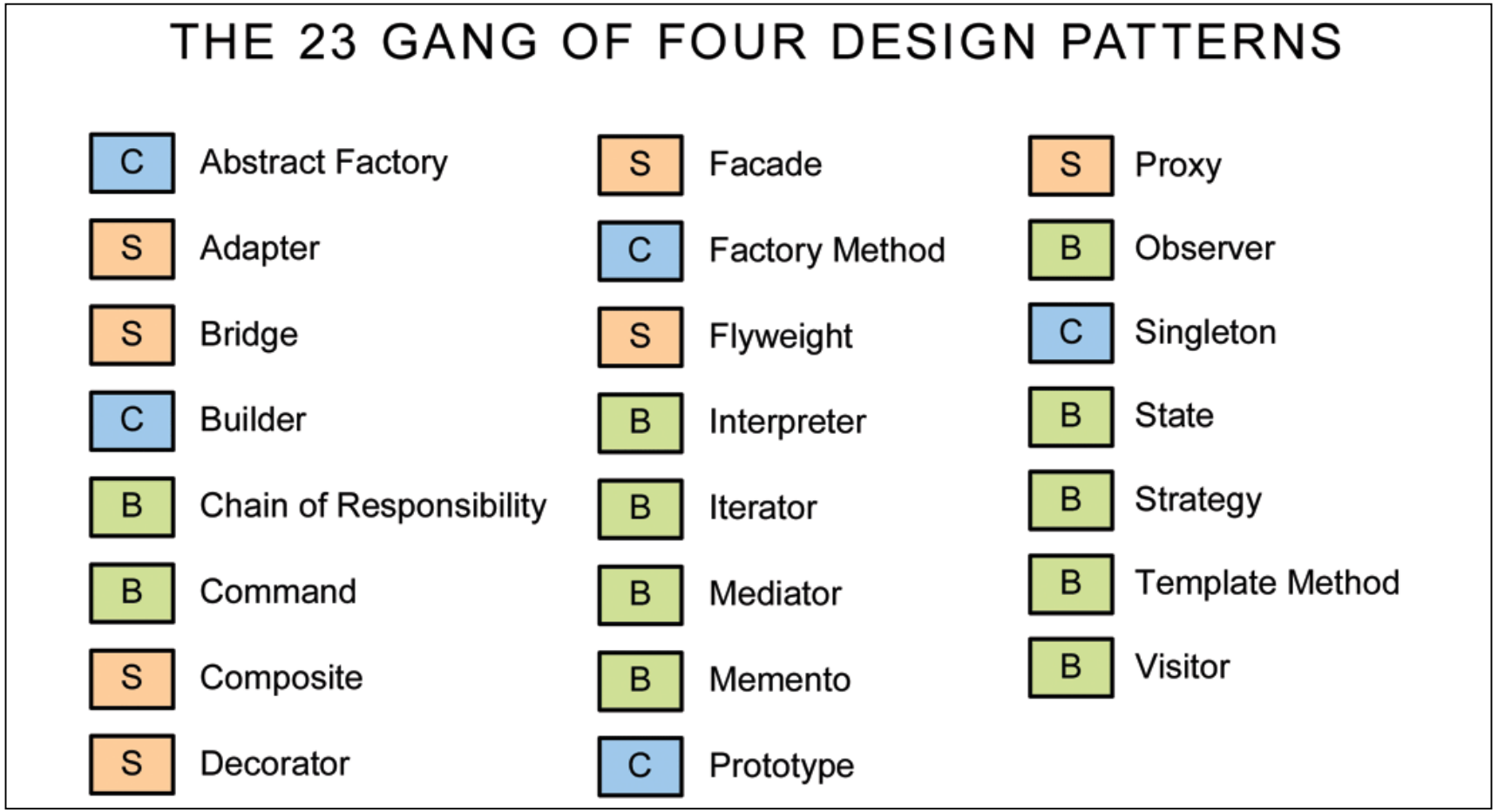
- **Discussion:** Tell us about your group project!
- **Background Research** - SWAT Analysis
- **SOW** – Summarize the goal of your project in 15 words or less
- **Requirements**

- **Assignment (Asn 3):**
  - Background Research – limit to one page
  - SOW (Statement of Work) – limit to one sentence
  - Requirements
    - Focus on capturing functional requirements like writing user stories (Ex. “As a user, I can send a private message”)
  - Upload your work via blackboard >> EGR326 >> Assignment >> Asn 3
  - Due date: Friday, Jan 24 by 10:00pm

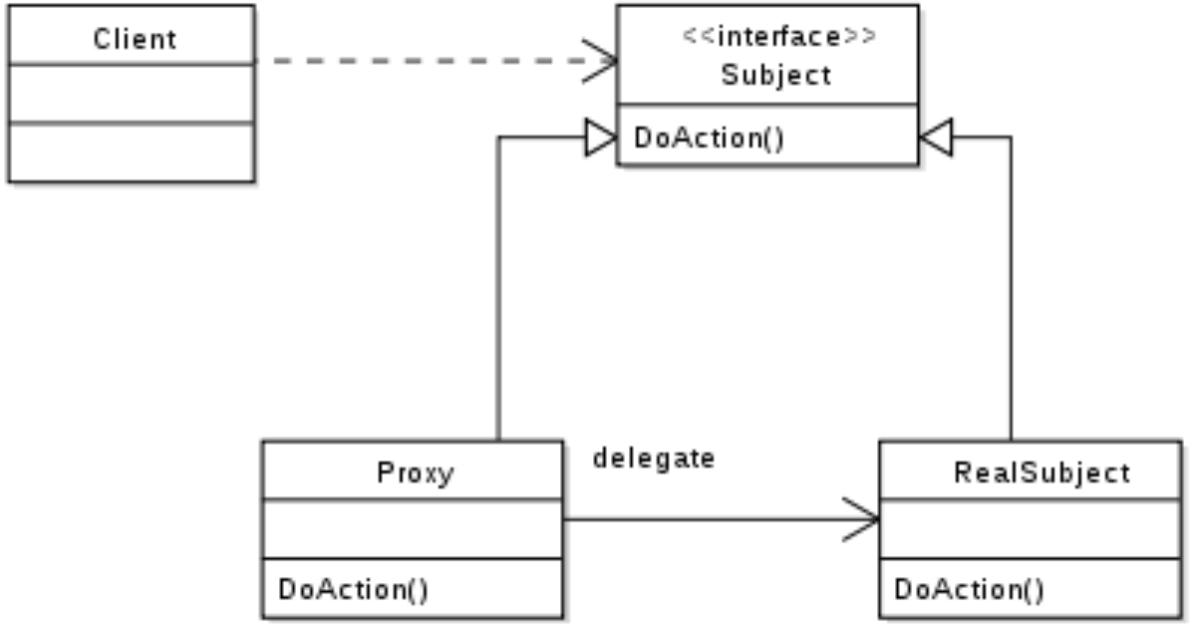
- ✓ A Design Pattern has four elements that define it – and determine the level of abstraction that we think and design at

1. Pattern Name – short and descriptive
2. Problem – when the pattern should be applied, and preconditions to its use
3. Solution – the elements of the design: objects, responsibilities, relationships and collaborations
4. Consequences – results and trade-offs

- ✓ There are 23 basic patterns covered in the "Design Patterns" book, of three types: Creational, Structural and Behavioral



- ✓ As an example, the Proxy pattern uses an intermediary to allow controlled access to a given object



- ***remote proxy:*** local representative of an object on another machine
- ***virtual proxy:*** defer object creation until needed
- ***protection proxy:*** checks access control rights, etc.

Ex) Proxy server

In computer networking, a proxy server is a server application or appliance that acts as an intermediary for requests from clients seeking resources from servers that provide those resources.

Proxy server Software

View 5+ more

Squid FreeProxy Nginx Tinyproxy Privoxy

Intent

Provide a surrogate or placeholder for another object to control access to it.

Also Known As

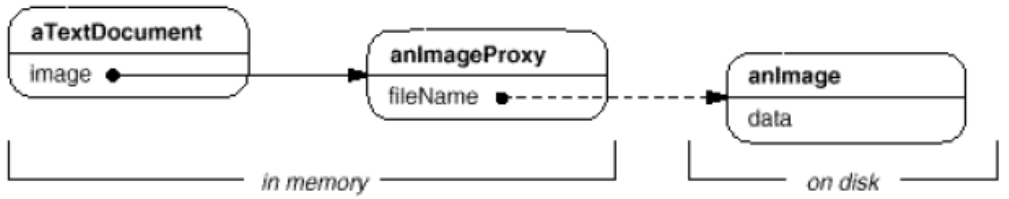
Surrogate

Motivation

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. This isn't necessary anyway, because not all of these objects will be visible in the document at the same time.

These constraints would suggest creating each expensive object *on demand*, which in this case occurs when an image becomes visible. But what do we put in the document in place of the image? And how can we hide the fact that the image is created on demand so that we don't complicate the editor's implementation? This optimization shouldn't impact the rendering and formatting code, for example.

The solution is to use another object, an image **proxy**, that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.



The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation. The proxy forwards subsequent requests directly to the image. It must therefore keep a reference to the image after creating it.

Let's assume that images are stored in separate files. In this case we can use the file name as the reference to the real object. The proxy also stores its **extent**, that is, its width and height. The extent lets the proxy respond to requests for its size from the formatter without actually instantiating the image.

Applicability

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP [Add94] uses the class NXProxy for this purpose. Coplien [Cop92] calls this kind of proxy an "Ambassador."

2. A **virtual proxy** creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.

3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system [CIRM93] provide protected access to operating system objects.

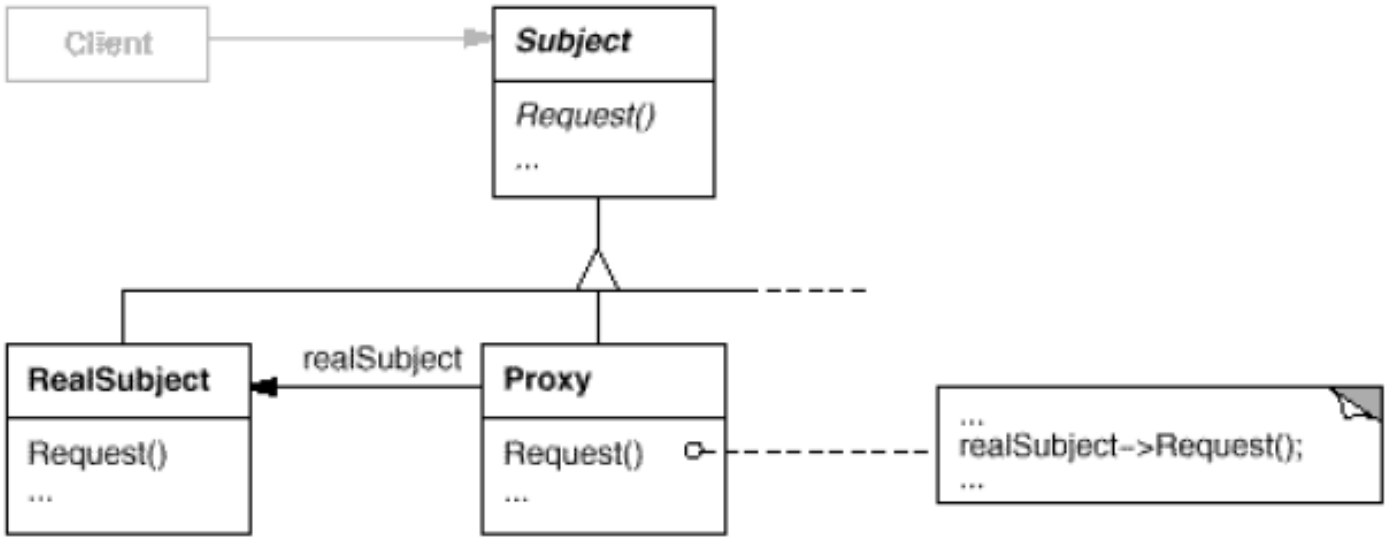
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include

    - counting the number of references to the real object so that it can be freed automatically when there are no more references (also called **smart pointers** [Ede92]).

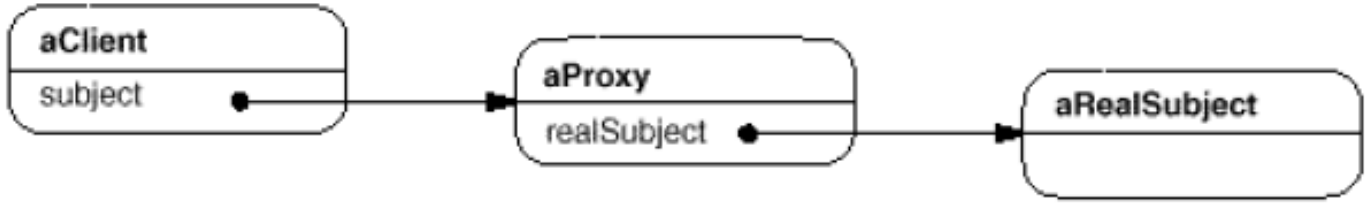
    - loading a persistent object into memory when it's first referenced.

    - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Structure



Here's a possible object diagram of a proxy structure at run-time:



Participants

- **Proxy** (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilities depend on the kind of proxy:
    - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
    - *protection proxies* check that the caller has the access permissions required to perform a request.
- **Subject** (Graphic)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject** (Image)
  - defines the real object that the proxy represents.

Known Uses

The virtual proxy example in the Motivation section is from the ET++ text building block classes.

NEXTSTEP [Add94] uses proxies (instances of class NXProxy) as local representatives for objects that may be distributed. A server creates proxies for remote objects when clients request them. On receiving a message, the proxy encodes it along with its arguments and then forwards the encoded message to the remote subject. Similarly, the subject encodes any return results and sends them back to the NXProxy object.

McCullough [McC87] discusses using proxies in Smalltalk to access remote objects. Pascoe [Pas86] describes how to provide side-effects on method calls and access control with "Encapsulators."

Related Patterns

Adapter (139): An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.

Decorator (175): Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

Proxies vary in the degree to which they are implemented like a decorator. A protection proxy might be implemented exactly like a decorator. On the other hand, a remote proxy will not contain a direct reference to its real subject but only an indirect reference, such as "host ID and local address on host." A virtual proxy will start off with an indirect reference such as a file name but will eventually obtain and use a direct reference.

- ✓ Design Patterns are ideal for solving design issues because good design practice and working techniques are already built-in

  - Many of the patterns exist to implement good design behavior (encapsulation, loose coupling, etc)

  - They are at a larger scale than classes, but smaller than components and frameworks

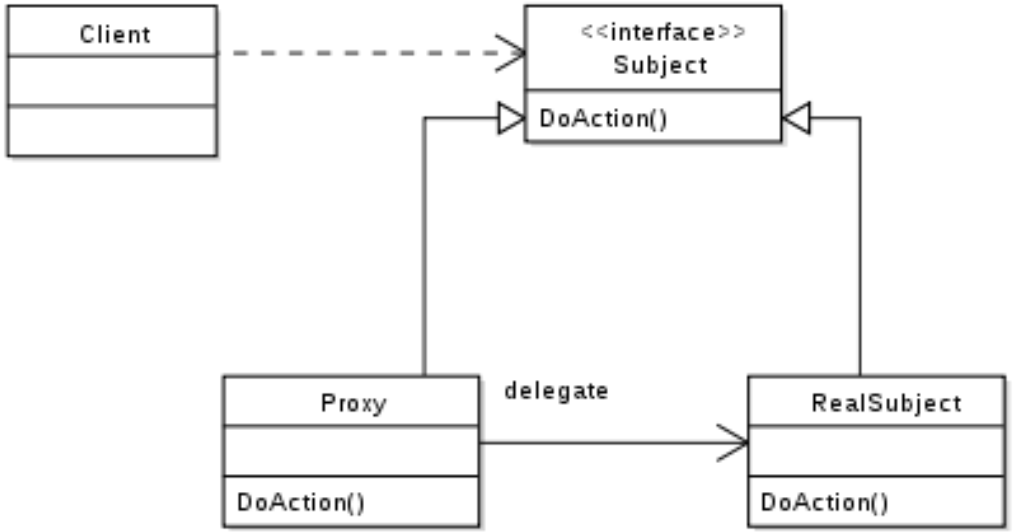
  - They provide a language for designs:
    "I used a factory in that component"

Selecting a Design Patterns?

- ✓ Selecting a Design Pattern can be challenging at first, but their usefulness grows with experience
  - Identify the problems your design may face
  - Become familiar with common design patterns
  - See how others use them
  - Study how patterns interrelate
  - If you ever need to redesign a project, look into why and what might have prevented the need

- ✓ Selecting a Design Pattern can be challenging at first, but their usefulness grows with experience

1. Read the pattern documentation
2. Study the Structure, Participants and Collaborations sections
3. Look at (and maybe copy) the sample code
4. Choose names for participants in your design
5. Define the classes
6. Define application-specific names for operations (methods)
7. Implement the operations (not part of the design phase)



- ✓ Selecting a Design Pattern can be challenging at first, but their usefulness grows with experience
  - What is a Design Pattern?
  - Describing Design Patterns
  - How Design Patterns Solve Problems
  - Why Design Pattern?
  - Selecting a Design Pattern
  - Using a Design Pattern

To-Do Week 2 (Asn3: Project Selection)

- **Discussion:** Tell us about your group project!
- **Background Research** - SWAT Analysis
- **SOW** – Summarize the goal of your project in 15 words or less
- **Requirements**

- **Assignment (Asn 3):**
  - Background Research – limit to one page
  - SOW (Statement of Work) – limit to one sentence
  - Requirements
    - Focus on capturing functional requirements like writing user stories (Ex. “As a user, I can send a private message”)
  - Upload your work via blackboard >> EGR326 >> Assignment >> Asn 3
  - Due date: Friday, Jan 24 by 10:00pm