Software Component Design

...

Department of Software Engineering

...

Bekretsyon B.(Msc)

March 2, 2025

Basic Concepts in Software Component Design

Contents

- Introduction to SE
- Software Design
- Basic Concepts and Terminology
- Benefits of Component Based Software Engineering
- Processes involved during component design
- Application area
- Difference with other programming

Introduction

What is SE?

- An engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures.
- Without using software engineering principles, it would be difficult to develop large programs
- Software engineering helps to reduce this programming complexity by using two important techniques: abstraction and decomposition
  - Abstraction is simplifying the problem by omitting irrelevant details
  - Decomposition is dividing the problem into small and manageable sub problems
- The outcome of SE is to have a good software product

Cont'd

Types of Software

- Purpose: System Software or Application Software
- Platform: Native software (designed for a specific operating system) or Cross-platform software (designed to run on multiple operating systems)
- Deployment: Installed software (installed on the user's device) or Cloud-based software (hosted on remote servers and accessed via the internet)
- License: Proprietary software (owned by a single entity) or Open-source software
- Size: Small-scale software or enterprise software (available for free with the source code accessible to the public)
- User Interface: Graphical User Interface (GUI) or Command-Line Interface (CLI)

Cont'd

What is Design?

- A process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation
- The output of this process can directly be used into implementation in programming languages
- Software design moves the concentration from problem domain to solution domain
  - So good to stress on this phase for a better software development , next to the requirement

So, What to design?

- Data can be designed:database design
- User interfaces can be designed:front ends design
- Architecture of software can be designed:A blue print, how does it work
- Components/ sub divisions can be designed individually: functional modules with own tasks

Component Based Design

- It focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties.
  - The current mostly used types of software design
- It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.

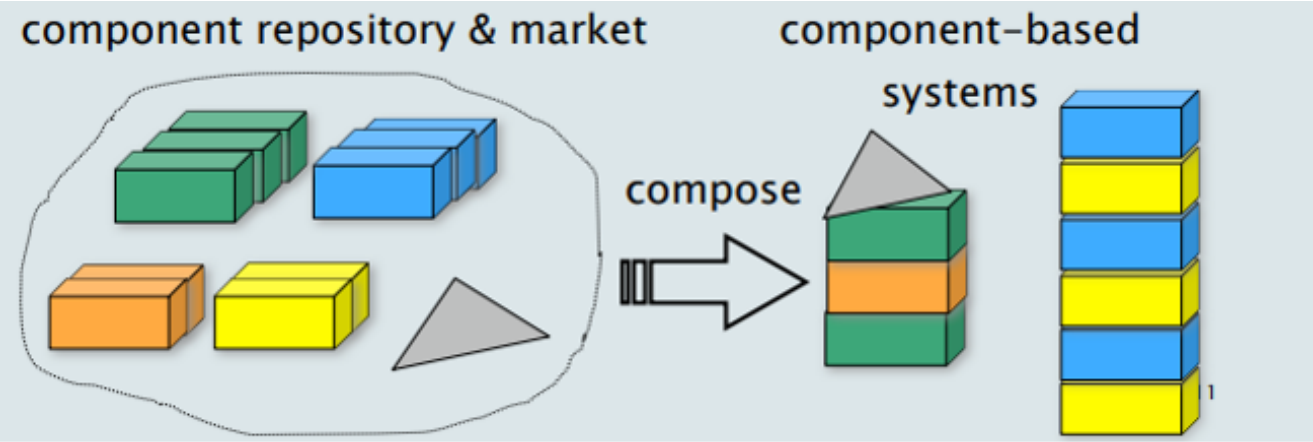
Why is Component Based Software Engineering?

- To reduces software crisis such as complexity, difficulty, quality...
- Component based engineering works well in other engineering disciplines. Example Car manufacturers do the design, buy components that they need from different component manufacturers, and assemble them into products.
- Different standalone components are being engineered to be used for own purpose
- Software developers are being tired of starting projects from the scratch since available reusable components are out there
- Software developer are more focusing on a "time to the market"

Cont'd

What is Component?

- A software component is a unit of composition with contractually specified interfaces, independently deployable/deliverable/configurable and subject to composition by third parties. "Clemens Szypersk"
- According to Johannes Sametinge, Components are self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status.
- Example: A payment gateway component in an e-commerce application that handles payment processing.



Cont'd

- The previous strategies to develop software system is "buy or build".
  - Buy components or build the system
- However with CBSE we can even use the strategies "Buy and build" or "follow design pattern" to develop software
  - Buy components and build system or follow their design pattern and build the system
- Here, the CBSE gives a chance to assemble the system from existing software components or follow the design pattern to build our own system

Benefits of CBSE

- The reuse of available/existing solution components than inventing new
- Reduces development time and cost to be lost while developing software from the scratch
- It allows the designers and developers to focus on important issues
  - Freeing them from having to worry about details of implementing each component
- Easy to replace or maintain, Reduce time to market
  - If the reuse of a component requires less time than the development of a component, systems can be built faster.

Characteristics of CBSE

Characteristics of CBSE

- Reusable
- Replaceable
- Composable
- Independent/self-contained/autonomous
- Deployable/configurable/deliverable
- Logically cohesive
- Loosely coupled
- Scalable/extensibility
- Maintainable

Cont'd

Independence

- Components should be independent of each other, meaning that changes in one component should not affect others.

Modularity

- Divide your software into smaller, manageable modules or components, each responsible for a specific task or feature.

High Cohesion

- Components should contain related and closely related functionalities.
- High cohesion reduces complexity and enhances maintainability.

Loose Coupling:

- Components should interact with each other through well-defined interfaces, minimizing dependencies between them.
- This makes it easier to replace or upgrade components.

Types of CBSE

- Class-Based Components: These are often implemented as classes in object-oriented programming and encapsulate data and behavior.
- Service-Based Components: These provide specific services or functionalities and may be implemented as standalone modules or microservices.
- Web Components: These are components designed to be used in web applications and are encapsulated HTML, CSS, and JavaScript.

Processes in CBSE

- Component design is a critical phase in software development, where the detailed design and specification of individual software components occur.
- It involves several processes and steps to ensure that the components are well-structured, maintainable, and meet their intended functionality.

Processes in CBSE

- Requirements Analysis:
  - Understand the functional and non-functional requirements for the component, including its inputs, outputs, and expected behavior.
  - Define component's interfaces and interactions with other components or systems.
- Component Identification:
  - Identify the specific functionality or feature that the component will provide within the overall system.
  - Consider reusability by evaluating if the component could be used in other projects or parts of the system.

Cont'd

Processes in CBSE

- Specification:
  - Create a detailed specification for the component, outlining its purpose, inputs, outputs, and behavior.
  - Define the component's interface, including methods, parameters, and data structures.
- Design Decomposition:
  - Break down the component into smaller, manageable sub-components or modules, if necessary.
  - Determine the structure and relationships between these sub-components.
- Data Design:
  - Define the data structures and data flows within the component, including how data is processed and stored.
  - Address issues related to data validation, storage, and access.
- Algorithm Design:
  - Specify the algorithms and logic that the component will use to perform its functions.
  - Consider performance optimization and algorithmic efficiency.

Benefits of CBSE over Traditional Software Dev't

- Reusability: Using pre-built components, dev'rs can save time and effort.
  - Example: Instead of writing authentication logic from scratch, developers can use a reusable authentication component across different systems.
- Maintainability: Since components are modular, updates or bug fixes can be applied to individual components without affecting the entire system.
  - Example: Updating a billing component in a shopping app won't require changes in other modules like product display or customer reviews.
- Scalability: Components can be added or replaced to accommodate growing business needs or new features.
  - Example: If an e-commerce platform grows to support international markets, a new currency conversion component can be added without altering the existing checkout process.
- Quality and Reliability: Reusable components are often tested thoroughly and used across multiple systems, which helps in ensuring stability and reliability.
  - Example: A component for secure user login used across several banking applications is likely to have fewer bugs.

Application Area

- Enterprise Applications: Large organizations use modular components for HR, payroll, and customer management systems.
  - Example: SAP and Oracle ERP systems use modular components like accounting, procurement, and inventory.
- Web Applications: Modern web applications rely on reusable front-end and back-end components.
  - Example: React.js or Angular front-end components for building user interfaces can be reused across multiple websites.
- Cloud-Based Systems: Cloud services often use microservices, which are individual components that can scale independently.
  - Example: Amazon Web Services (AWS) provides components for cloud storage (S3) and computing (EC2), which can be integrated into various applications.
- Embedded Systems: Devices such as smart appliances use modular components to manage specific tasks like sensor monitoring or user input.
  - Example: A smart thermostat may have a component for temperature control and another for Wi-Fi communication.

Software Component Design

...

Inter-Thread and Inter-Process Communication

...

Bekretsyon B.(Msc)

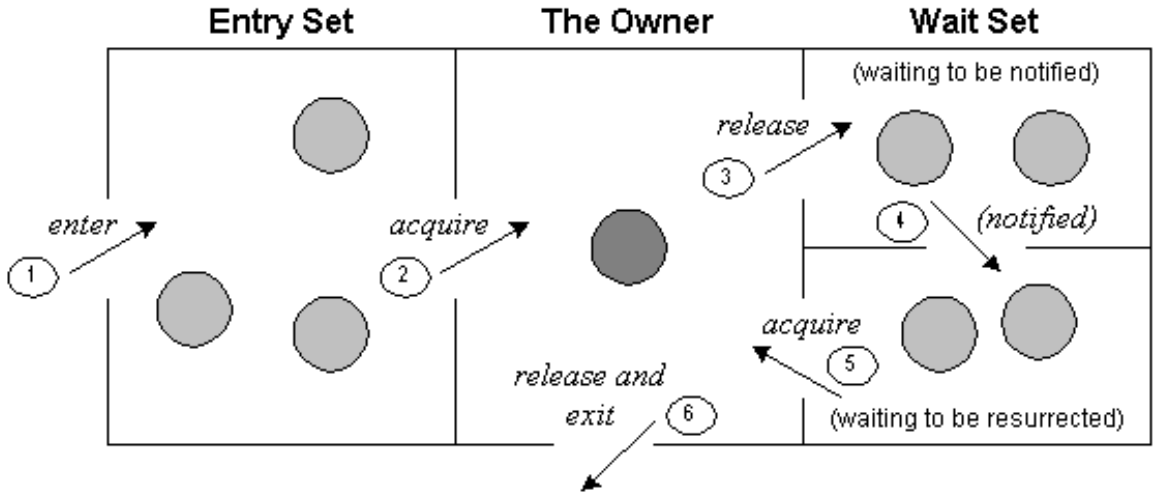
March 2, 2025

Inter-Thread Communication

- Allowing synchronized threads to communicate with each other.
- Is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- Can be implemented through three approaches:
  - wait(): Causes current thread to release the lock and wait until either another thread invokes the notify method.
  - notify(): Wakes up a single thread that is waiting on this object's monitor.
  - notify():All Wakes up all threads that are waiting on this object's monitor.

Cont'd

Example: Inter-Thread communication



How it Works

- Threads enter to acquire lock.
- Lock is acquired by on thread.
- Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
- If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
- Now thread is available to acquire lock.
- After completion of the task, thread releases the lock and exits the monitor state of the object

Inter Process Communications

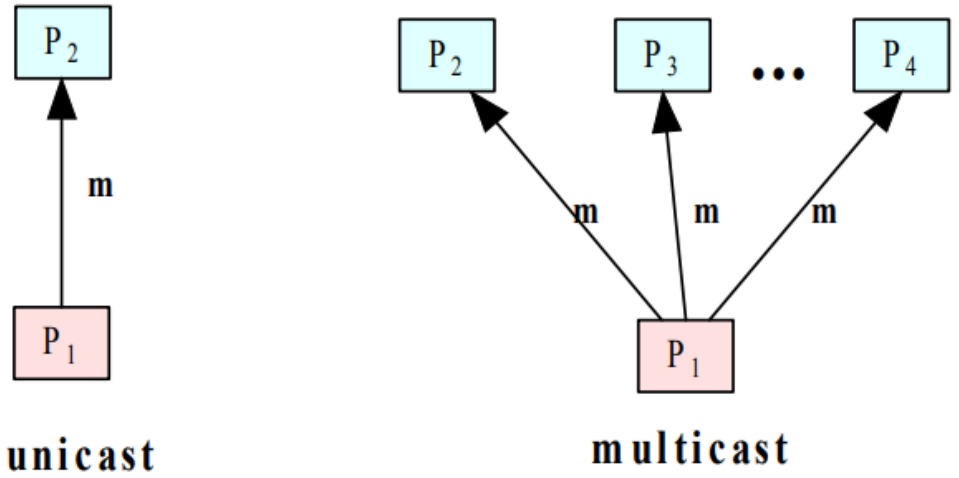
IPC

- Exchange of data between two or more separate, independent processes/threads.
- Operating systems provide facilities/resources for interprocess communications (IPC), such as message queues, semaphores, and shared memory.
- Distributed computing systems make use of these facilities/ resources to provide application programming interface (API) which allows IPC to be programmed at a higher level of abstraction. (e.g., send and receive)
- Distributed computing requires information to be exchanged among independent processes.

IPC – Unicast and Multicast

Unicast and Multicast

- In distributed computing, two or more processes engage in IPC using a protocol agreed upon by the processes. A process may be a sender at some points during a protocol, a receiver at other points.
- When communication is from one process to a single other process, the IPC is said to be a unicast, e.g., Socket communication.
- When communication is from one process to a group of processes, the IPC is said to be a multicast, e.g., Publish/Subscribe Message model



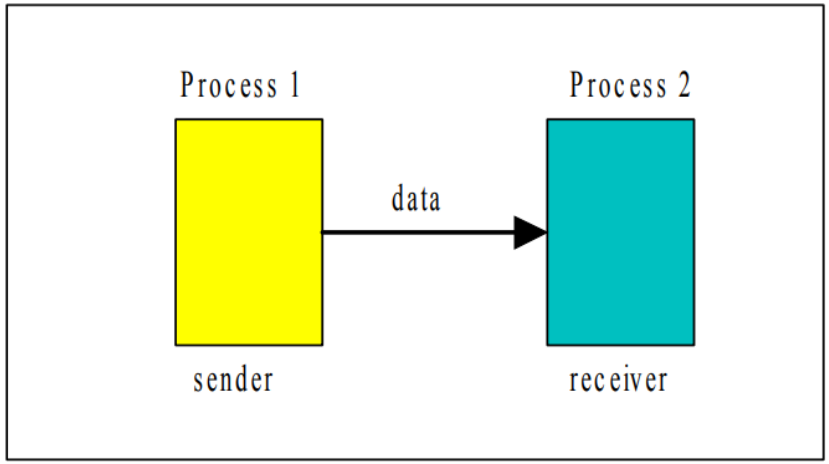
Publish/Subscribe Message model

The publish/subscribe communication paradigm:

- **Publishers:** produce data in the form of events.
- **Subscribers:** declare interests on published data with subscriptions.
- Each**subscription** is a filter on the set of published events.
- An **Event Notification Service:** (ENS) notifies to each subscriber every published event that matches at least one of its subscriptions.

IPCs in Distributed Computing

Distributed Computing



Operations provided in an archetypal IPCs API

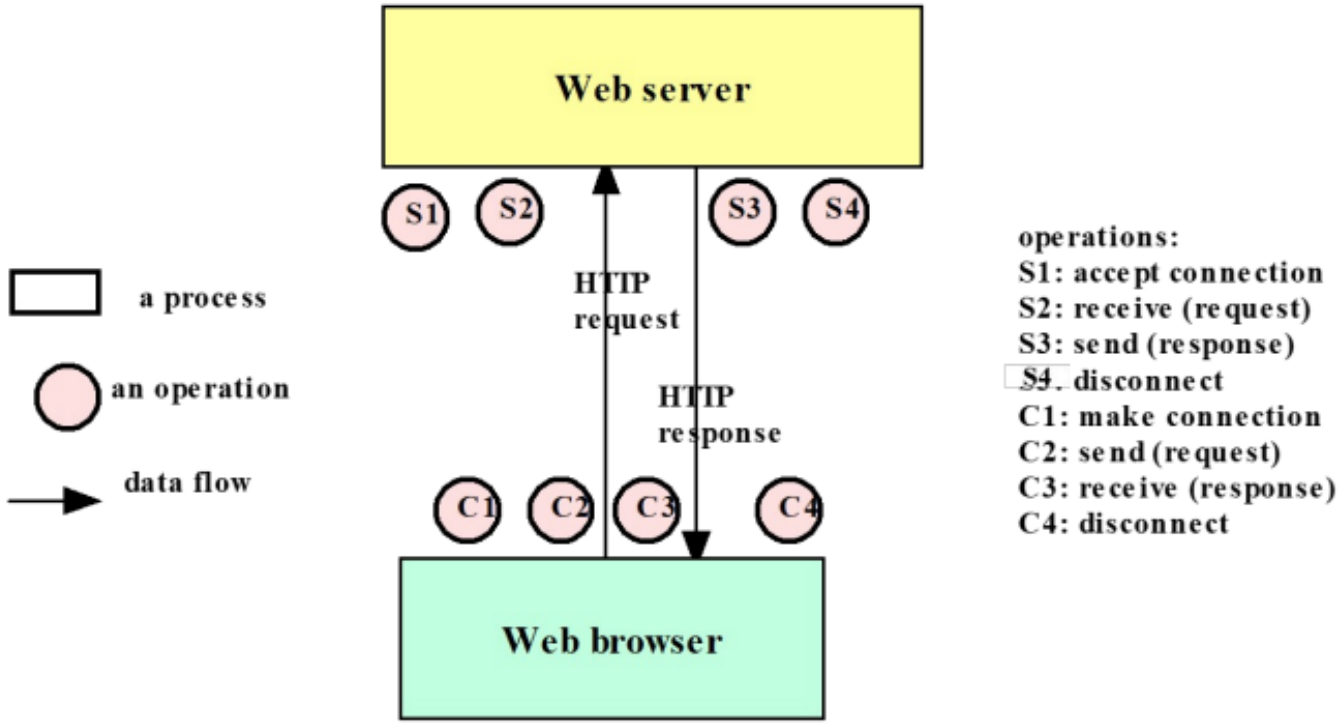
- Consider a basic API that provides the minimum level of abstraction to facilitate IPC. Four primitive operations are needed. They are:
  1. 1. Send ([receiver], message)
  2. 2. Receive ([sender], message storage object)
  3. 3. Connect (sender address, receiver address), for connection oriented communication.
  4. 4. Disconnect (connection identifier), for connection-oriented communication.

Inter process Communication in Basic HTTP

IPC in Basic HTTP

- Network service protocols can be implemented using primitive IPC operations.
- For example, in basic HTTP (used extensively on the WWW), one process and a Web browser issues a connect operation to establish a logical connection to another process.
- A Web server, followed by a send operation to the Web server, transmits data representing a request.
- The Web server process in turn issues a send operation that transmits data requested by the Web browser process.
- At the end of the communication, each process issues a disconnect operation to terminate the connection.

IPC in Basic HTTP



Processing order: C1, S1, C2, S2, S3, C3, C4, S4

Event Synchronization

- A main difficulty with IPC is to execute multiple processes involved in a system independently, with neither process knowing what takes place in the process at the other end.
- Inter-process communication may require that the two processes synchronize their operations: one side sends, then the other receives until all data has been sent and received.
- The send operation starts before the receive operation commences (start).
- In practice, the synchronization requires system support.

Synchronous vs. Asynchronous Communication

- IPC operations may provide synchronization necessary using blocking.
- A blocking operation issued by a process will block further processing of the process until the operation is fulfilled.
- Alternatively, IPC operations may be asynchronous or nonblocking.
- An asynchronous operation issued by a process will not block further processing of the process. Instead, the process is free to proceed with its processing, and may optionally be notified by the system when the operation is fulfilled.

Types of IPC Operations

- Synchronous Send and Synchronous Receive
- Asynchronous Send and Synchronous Receive
- Synchronous Send and Asynchronous Receive
- Asynchronous Send and Asynchronous Receive

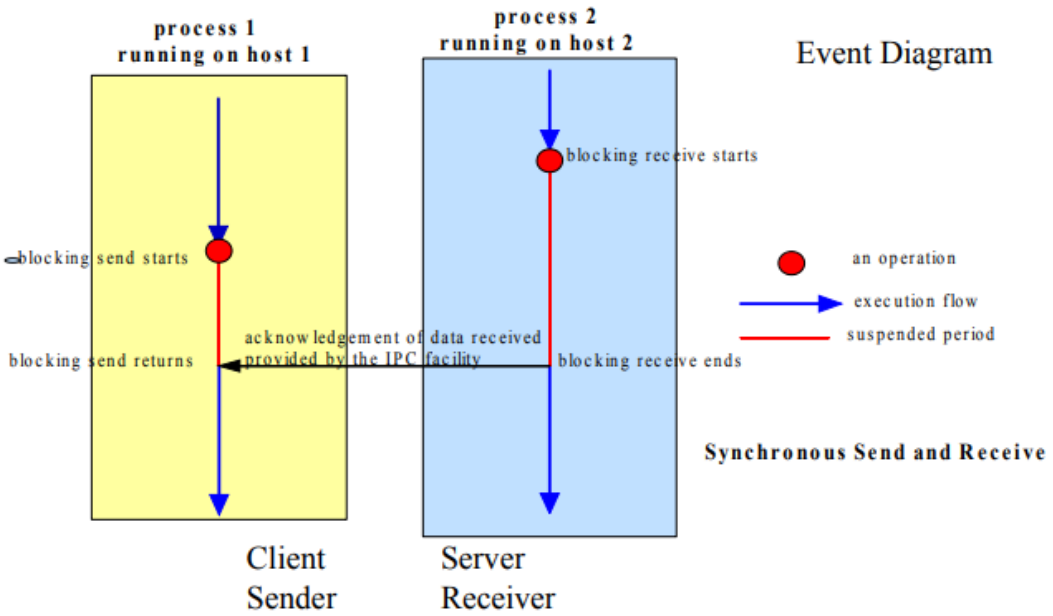
Cont'd

Synchronous Send and Receive

- Both the sending and receiving processes are blocked until the message is successfully delivered and, optionally, a response is received.
- A receive operation issued causes the suspension of the issuing process (process 2) until data is received to fulfill the operation.
- A send operation issued causes the sending process (process 1) to suspend.
- When the data sent has been received by process2, the IPC facility on host 2 sends an acknowledgment to the IPC facility on host I, and process 1 may subsequently be unblocked.
- Depending on the implementation of the IPC facility, the synchronous receive operation may not be fulfilled until the amount of data the receiver expects to receive has arrived.
- The use of synchronous send and synchronous receive is warranted if the application logic of both processes requires that the data sent must be received before further processing can proceed.

Cont'd

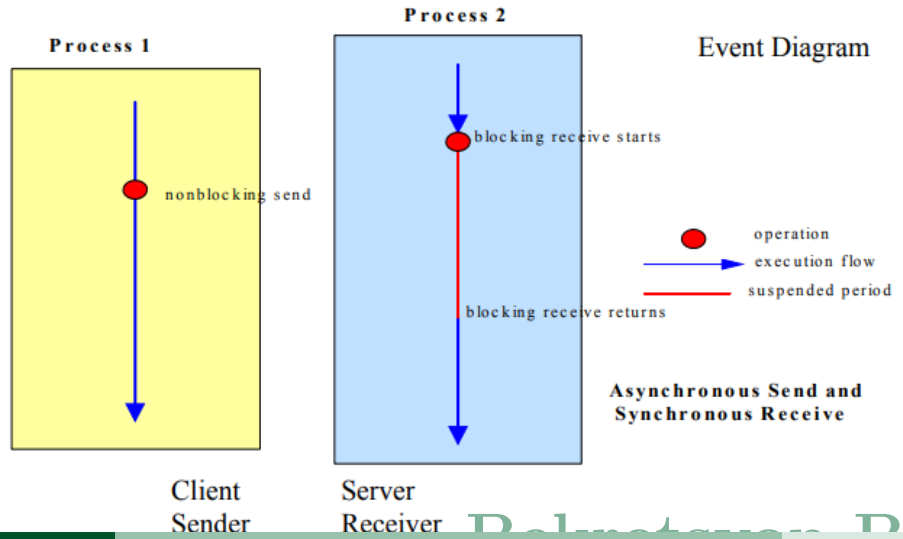
Synchronous Send and Receive



Cont'd

Asynchronous Send and Synchronous Receive

- A receive operation issued will cause the suspension of the issuing process until data is received to fulfill the operation.
- A send operation issued will not cause sending process to suspend. In this case sending process is never blocked, so no acknowledgment is necessary from IPC facility on the host of process 2, if sender's application logic does not depend on receiving of data at the other end.
- Depending on the implementation of the IPC facility, there is no guarantee that the data sent will actually be delivered to the receiver.



Cont'd

Synchronous Send and Asynchronous Receive

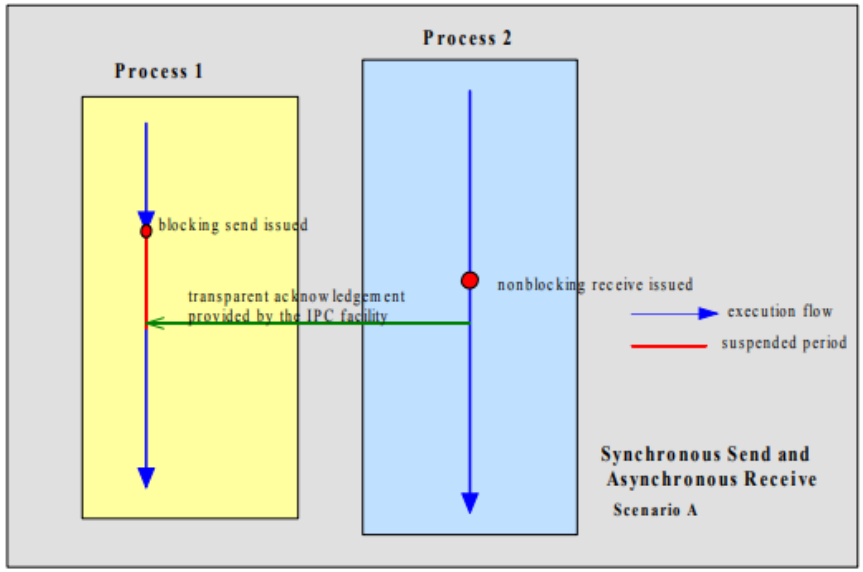
- An asynchronous receive operation causes no blocking of the process that issues the operation, and the outcome will depend on the implementation of the IPC facility.
- The receive operation will, in all cases, return immediately.
- There are three scenarios for what happens subsequently:

Synchronous send and Asynchronous Receive - 1

- The data requested by the receive operation has already arrived at the time when the receive operation is issued.
- The data is delivered to process 2 immediately and an acknowledgment from host 2's IPC facility will unblock process 1.

Cont'd

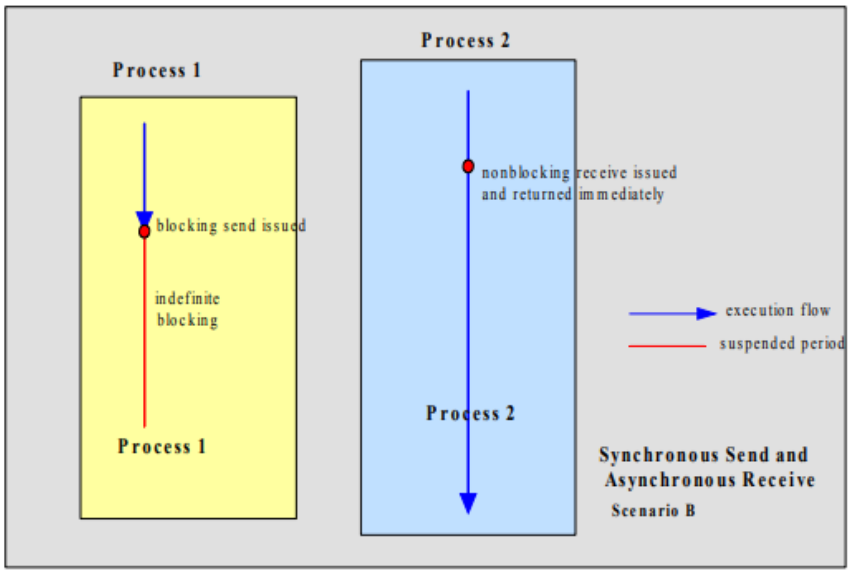
Synchronous send and Asynchronous Receive



Data from P1 was received by P2
before issuing a non-blocking receive op in P2

Synchronous Send and Asynchronous Receive - 2

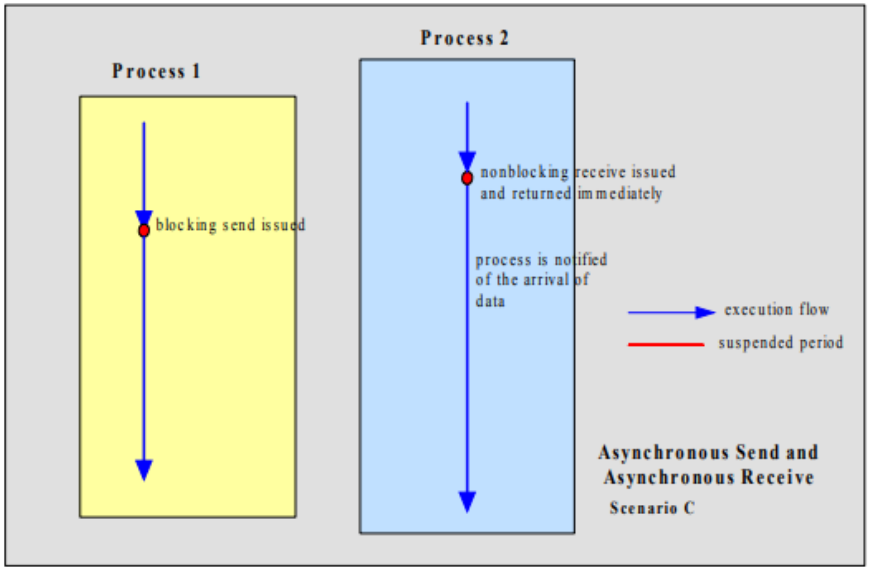
- The data requested by the receive operation has not yet arrived and thus no data is delivered to the process.
- It is the receiving process's responsibility to confirm that it has indeed received data and repeat receive operation until data has arrived.
- P1 is blocked indefinitely until P2 reissues a receive request and an acknowledgment eventually arrives from host 2's IPC facility



Data from P1 arrived to P2 **after** P2 issued a non-blocking receive op

Asynchronous Send and Asynchronous Receive

- Without blocking on either side, the only way that the data can be delivered to the receiver is if the IPC facility retains the data received.
- The receiving process can then be notified of the data's arrival.
- Alternatively, the receiving process may poll for the arrival of the data and process it when the awaited data has arrived.



Does P1 need an acknowledgement from P2?

IPC Approaches

Approaches

Method	Short Description	Provided by
File	A record stored on disk, or a record synthesized on demand by a file server, which can be accessed by multiple processes.	Most operating systems
Signal	A system message sent from one process to another, not usually used to transfer data but instead used to remotely command the partnered process.	Most operating systems
Socket	A data stream sent over a network interface, either to a different process on the same computer or to another computer on the network. Typically byte-oriented, sockets rarely preserve message boundaries. Data written through a socket requires formatting to preserve message boundaries.	Most operating systems
Shared memory	Multiple processes are given access to the same block of memory which creates a shared buffer for the processes to communicate with each other.	All POSIX systems, Windows

What is a Socket?

Socket

- The combination of an IP address and a port number. (RFC 793 ,original TCP specification)
- The name of the Berkeley-derived application programming interfaces (APIs) for applications using TCP/IP protocols.
- Two types
  - Stream socket : reliable two-way connected communication streams
  - Datagram socket

Socket Pair

- Specified the two end points that uniquely identifies each TCP connection in an internet.
- 4-tuple: (client IP address, client port number, server IP address, server port number)

Socket Programming with TCP

process communication Through TCP

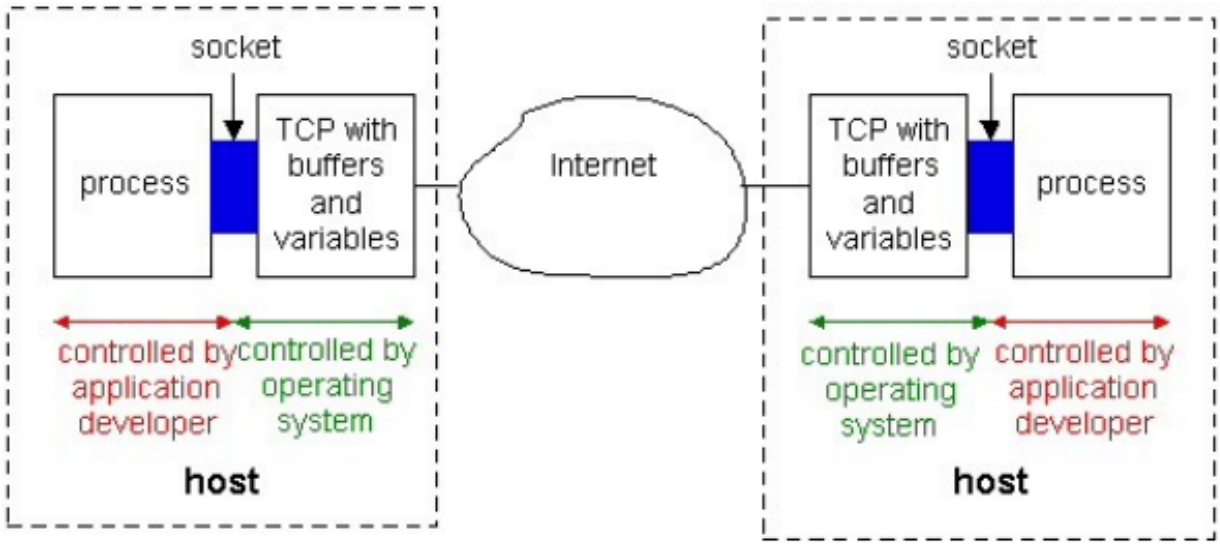


Figure : Processes communicating through TCP sockets

- The application developer has the ability to fix a few TCP parameters, such as maximum buffer and maximum segment sizes.

Sockets for Server and Client

Server

- Welcoming socket: Welcomes some initial contact from a client.
- Connection socket
  - Is created at initial contact of client.
  - New socket that is dedicated to the particular client

Client

- Client socket
  - Initiate a TCP connection to the server by creating a socket object. (Three-way handshake)
  - Specify the address of the server process, namely, the IP address of the server and the port number of the process.

Socket Functional Calls

- socket (): Create a socket
- bind(): bind a socket to a local IP address and port
- listen(): passively waiting for connections
- connect(): initiating connection to another socket
- accept(): accept a new connection
- Write(): write data to a socket
- Read(): read data from a socket
- sendto(): send a datagram to another UDP socket
- recvfrom(): read a datagram from a UDP socket
- close(): close a socket (tear down the connection)

Sockets

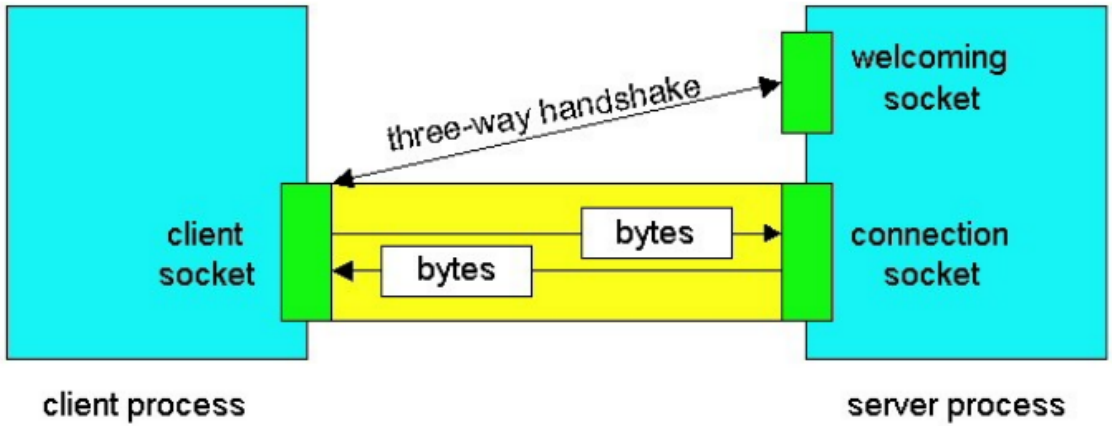
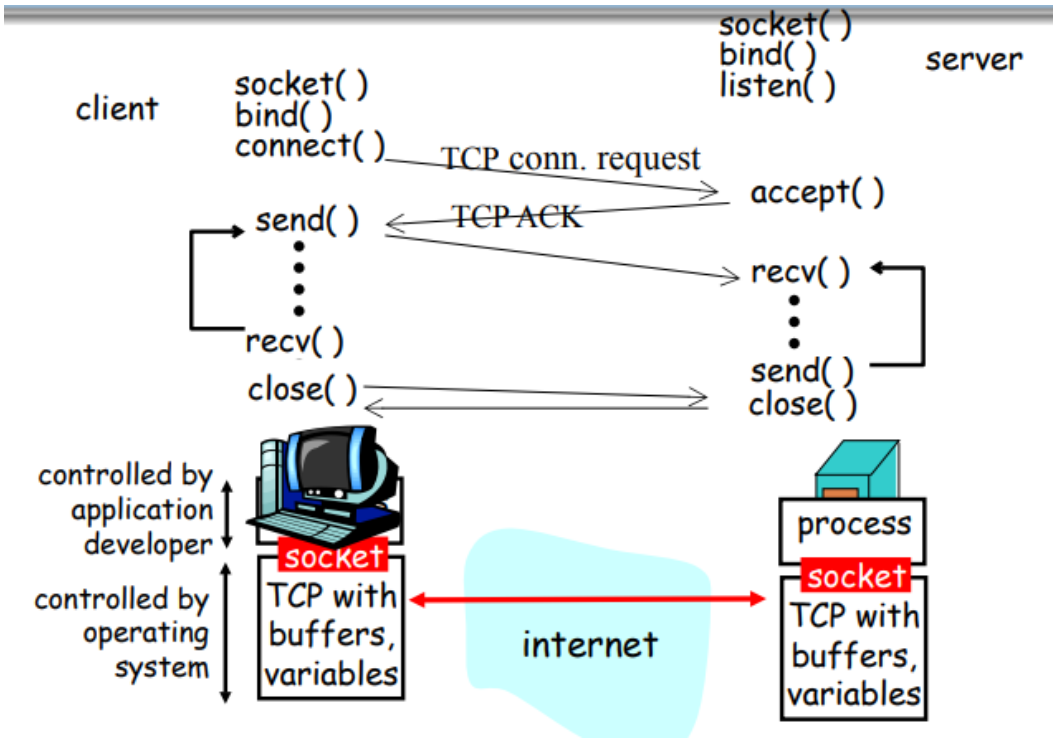


Figure 2.6-2: Client socket, welcoming socket and connection socket

Socket-programming using TCP

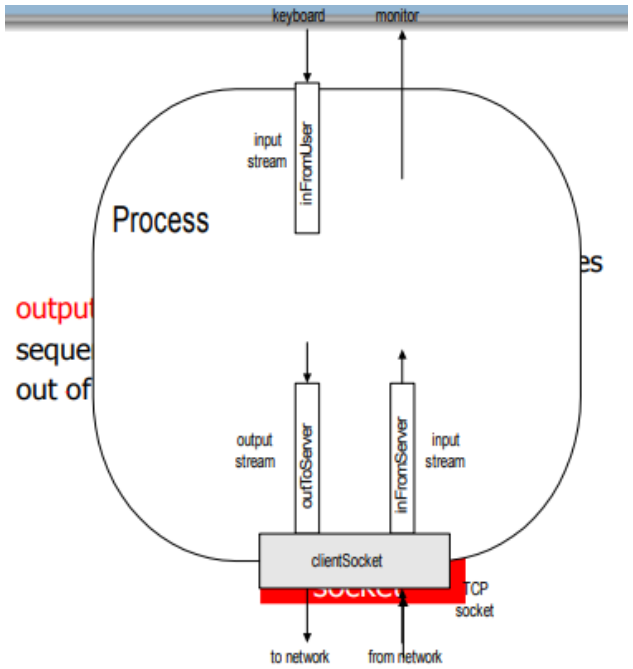
- TCP service: reliable byte stream transfer



Socket programming with TCP

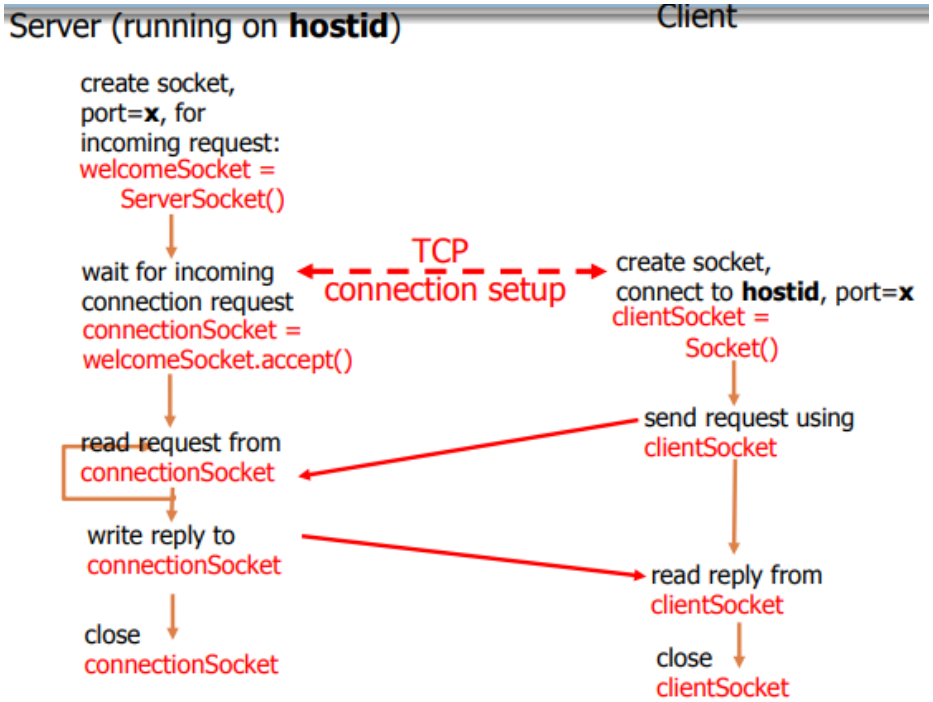
Example client-server app:

- client reads line from standard input(inFromUser stream) , sends to server via socket (outToServer stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (inFromServer stream)



Client/Server Socket Interaction: TCP

Socket Interaction



JAVA TCP Sockets

In Package java.net

- java.net.Socket
  - Implements client sockets (also called just "sockets").
  - An endpoint for communication between two machines.
  - Constructor and Methods
    - Socket(String host, int port): Creates a stream socket and connects it to the specified port number on the named host.
    - InputStream getInputStream()
    - OutputStream getOutputStream()
    - close()
- java.net.ServerSocket
  - Implements server sockets.
  - Waits for requests to come in over the network.
  - Performs some operation based on the request.
  - Constructor and Methods
    - ServerSocket(int port)
    - Socket Accept(): Listens for a connection to be made to this socket and accepts it. This method blocks until a connection is made.

Establishing a Simple Server Using Stream Sockets

Steps

- Step 1: Create a ServerSocket
  - ServerSocket server = new ServerSocket(portNumber, queueLength);
- Step 2: Wait for a Connection
  - Socket connection = server.accept();
- Step 3: Get the Socket's I/O Streams
  - ObjectInputStream input = new ObjectInputStream(connection.getInputStream());
  - ObjectOutputStream output = new ObjectOutputStream(connection.getOutputStream());
- Step 4: Perform the Processing
  - in which the server and the client communicate via the OutputStream and InputStream objects.
- Step 5: Close the Connection
  - when the transmission is complete, the server closes the connection by invoking the close method on the streams and on the Socket.

Establishing a Simple Client in Java

Steps

- Step 1: Create a Socket to Connect to the Server
  - Socket connection = new Socket(serverAddress, port);
- Step 2: Get the Socket's I/O Streams
  - the client uses Socket methods getInputStream and getOutputStream to obtain references to the Socket's InputStream and OutputStream.
- Step 3: Perform the Processing
  - is the processing phase in which the client and the server communicate via the InputStream and OutputStream objects.
- Step 4: Close the Connection
  - the client closes the connection when the transmission is complete by invoking the close method on the streams and on the Socket

TCPClient.java

```
import java.io.IOException;
import java.io.PrintStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;
public class cli {
   public static void main(String args[]) throws UnknownHostException, IOException{
            int number,temp;
            Scanner sc=new Scanner (System.in);
            Socket s=new Socket("127.0.0.1",1342);
            Scanner sc1=new Scanner (s.getInputStream());
            System.out.println("Enter any number");
            number=sc.nextInt();
            PrintStream p=new PrintStream(s.getOutputStream());
            p.println(number);
            temp=sc1.nextInt();
            System.out.println(temp);
        }}
```

TCPServer.java

```
import java.net.ServerSocket;
import java.io.PrintStream;
import java.net.Socket;
import java.util.Scanner;
public class ser {
   public static void main(String args[]) throws IOException{
      int number,temp;
      ServerSocket s1=new ServerSocket(1342);
      Socket ss=s1.accept();
      Scanner sc=new Scanner(ss.getInputStream());
      number=sc.nextInt();
      temp=number*2;
      PrintStream p=new PrintStream(ss.getOutputStream());
      p.println(temp);
         }}
```

Socket Programming with UDP

UDP

- Connectionless and unreliable service.
- There isn't an initial handshaking phase.
- Doesn't have a pipe.
- transmitted data may be received out of order, or lost

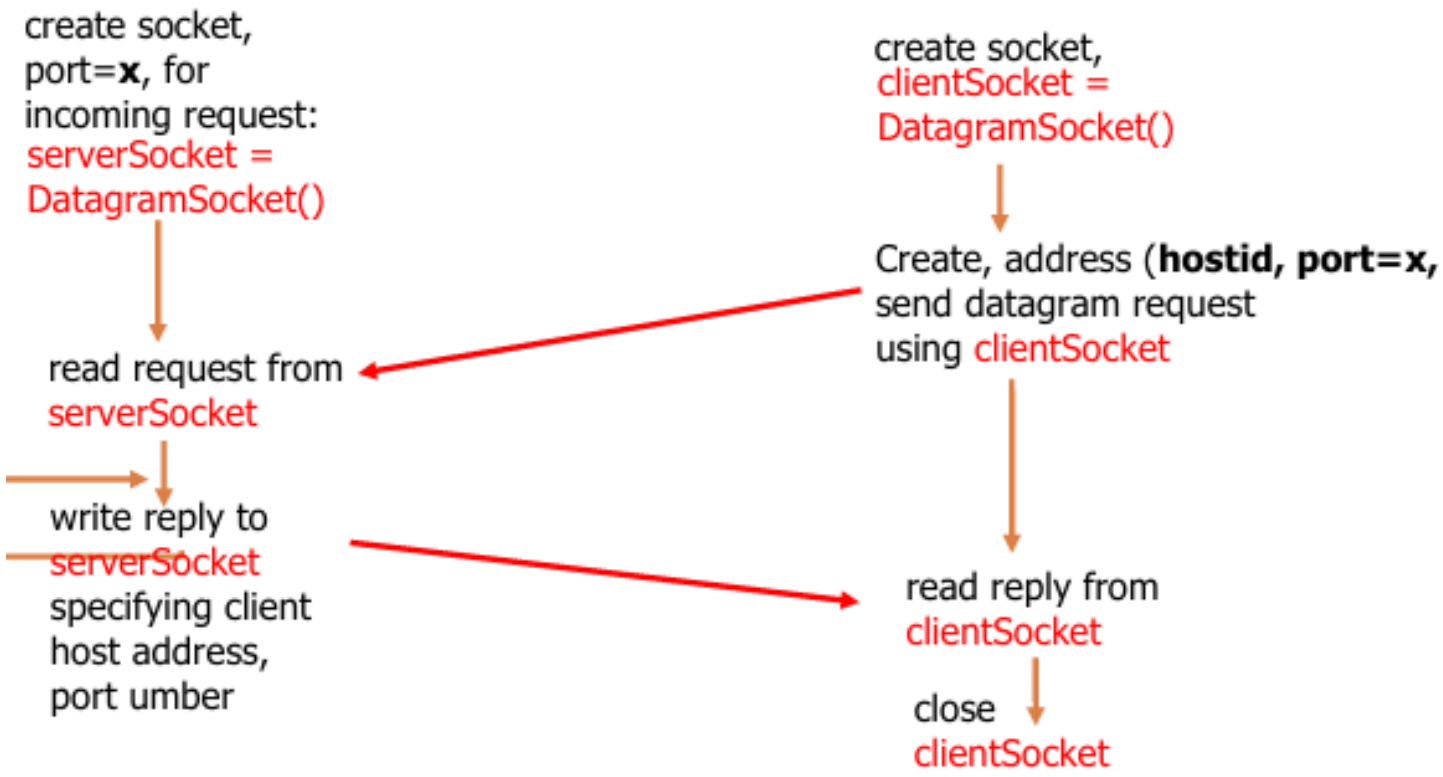
Socket Programming with UDP

- No need for a welcoming socket.
- No streams are attached to the sockets.
- the sending hosts creates "Packets" by attaching the IP destination address and port number to each batch of bytes.
- The receiving process must unravel to received packet to obtain the packet's information bytes.

Client/Server Socket Interaction: UDP

UDP

Server (running on hostid) Client



JAVA UDP Sockets

In Package java.net

- java.net.DatagramSocket
  - A socket for sending and receiving datagram packets.
  - Constructor and Methods
    - DatagramSocket(int port): Constructs a datagram socket and binds it to the specified port on the local host machine.
    - void receive(DatagramPacket p)
    - void send(DatagramPacket p)
    - void close()

UDPClient.java and UDPServer.java

UDPClient

```
import java.io.*;

public class udpclient {

   public static void main(String args[])throws Exception {

      DatagramSocket ds = new DatagramSocket();
      byte[] b = "this is udp".getBytes();
      InetAddress ip = InetAddress.getByName("localhost");
      int port = 2000;
      DatagramPacket dp = new DatagramPacket(b, b.length, ip, 2000);
      ds.send(dp);
   }
}
```

UDPServer

```
import java.net.*;
import java.io.*;
public class updserver {
   public static void main(String args[]) throws Exception {
      DatagramSocket ds=new DatagramSocket(2000);
      byte [] b=new byte[100];

      DatagramPacket dp=new DatagramPacket(b,b.length);
      ds.receive(dp);
      String msg=new String(b);
      System.out.println("Received"+msg);
   }

}
```

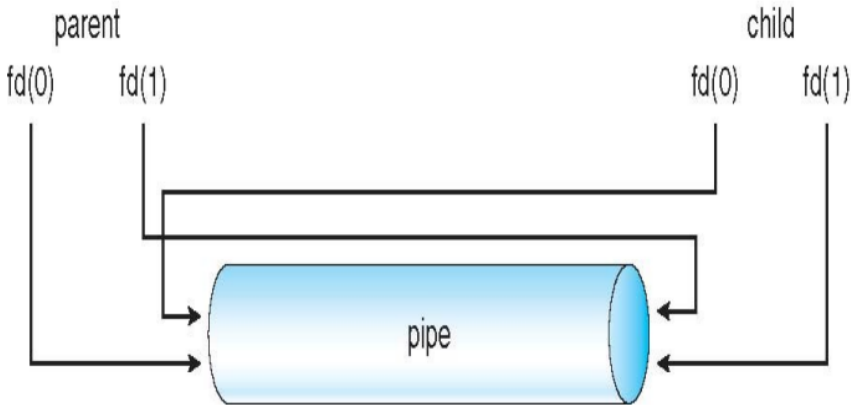
Pipes

- Acts as a conduit allowing two processes to communicate.
- Some issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- Allow communication in standard producer-consumer style.
- Producer writes to one end (the write-end of the pipe).
- Consumer reads from the other end (the read-end of the pipe).
- Ordinary pipes are therefore unidirectional.
- Require parent-child relationship between communicating processes.

Cont'D

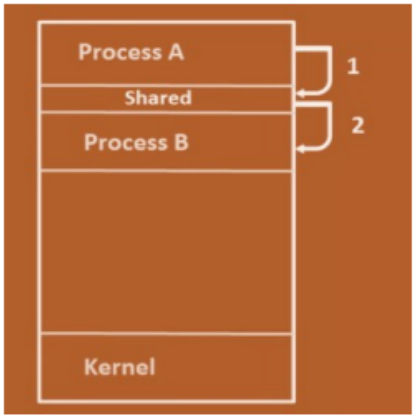


Named Pipes

- Named Pipes are more powerful than ordinary pipes.
- Communication is bidirectional.
- No parent-child relationship is necessary between the communicating processes.
- Several processes can use the named pipe for communication.
- Provided on both UNIX and Windows systems.

Shared Memory

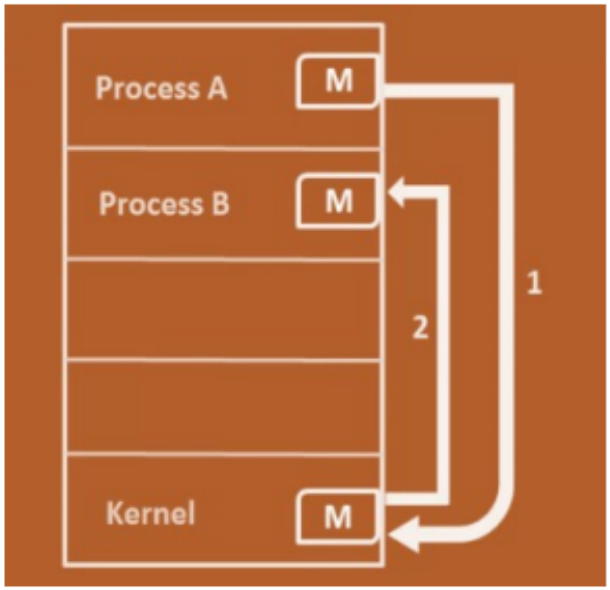
- Inthis model,region of memory that is shared by cooperating processes is established.
- Processes can then exchange information by reading and writing data to the shared region.



- Components:
  - Process A and B: Separate processes with their own address space.
  - Shared Memory: A region of memory that is accessible by both Process A and Process B. This is the key element for their communication.
  - Kernel: core of OS which manages the shared memory segment and facilitates mapping of this segment into address spaces of the participating processes.

Message Passing

- In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.



Software Component Design

...

Design Patterns:Definition and Usage

...

Bekretsyon B.(Msc)

March 2, 2025

Contents

- Introduction Design Pattern
- Essential Elements of Design Pattern
- GoF Design Patterns
- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Introduction

Design Pattern

- In software engineering, a design pattern is a general repeatable solution to a commonly occuring problem in software design
- It is not finished design that can be transformed directly into code, but a description or template for how to solve a problem that can be used In many different situations

Design Pattern

- Provide general solutions, documented in a format that doesn't require specifics tied to a particular problem
- Can speed up the development process by providing tested, proven development paradigms
- Help you Benefit from the experience of fellow developers
- Prevent subtle issues that can cause major problems
- Improve code readability for coders and architects familiar with them

Cont'd

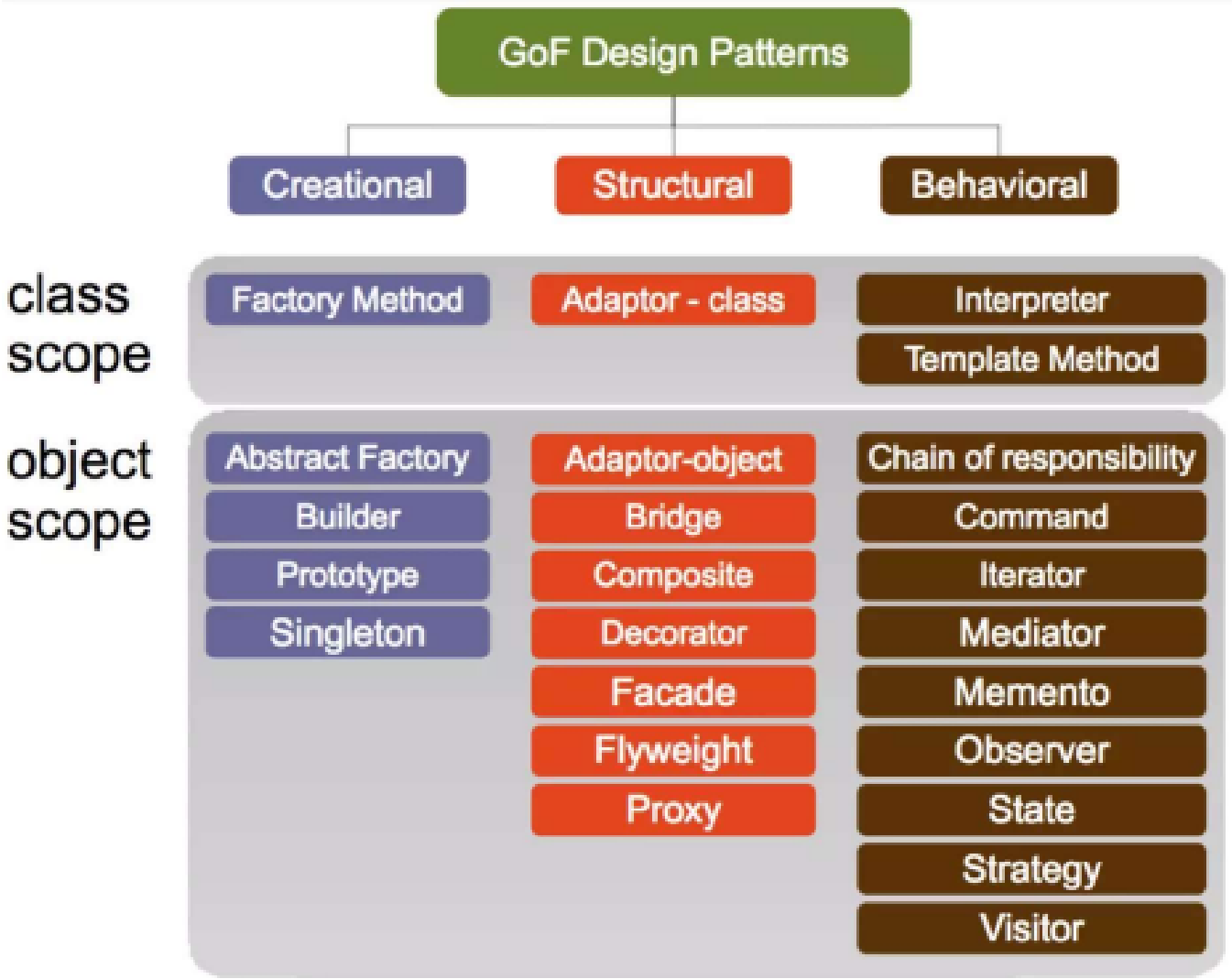
Design Pattern Essential Elements

- A pattern has four essential elements:
  - The pattern name that we use to describe a design problem
  - The problem that describes when to apply the pattern
  - The solution that describes the elements that make up the design
  - The consequences that are the results and trade-offs of applying the pattern

GoF Design Patterns

- The Gang of Four are the four authors of the book "Design Patterns: elements of Reusable Object-Oriented Software"
- Defined 23 design patterns for recurrent design issues, called Go design patterns
- Classified by purpose:
  - Structural: Concerns the composition of classes and objects
  - Behavioral : Characterizes the interaction and responsibility of objects and classes
  - Creational : Concerns the creation process of objects and classes
- by Scope:
  - Class Scope: Relationship between classes and subclasses, define statically
  - Object Scope: object relationships, dynamic

Cont'd


GoF Design Patterns
Creational
Structural
Behavioral
class scope
Factory Method
Adaptor - class
Interpreter
Template Method
object scope
Abstract Factory
Builder
Prototype
Singleton
Adaptor-object
Bridge
Composite
Decorator
Facade
Flyweight
Proxy
Chain of responsibility
Command
Iterator
Mediator
Memento
Observer
State
Strategy
Visitor

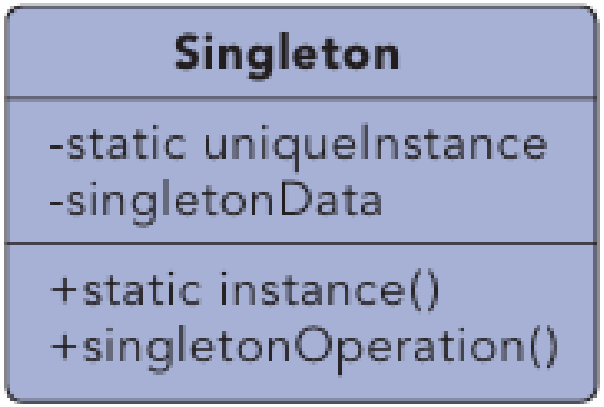
Creational Patterns: Definition

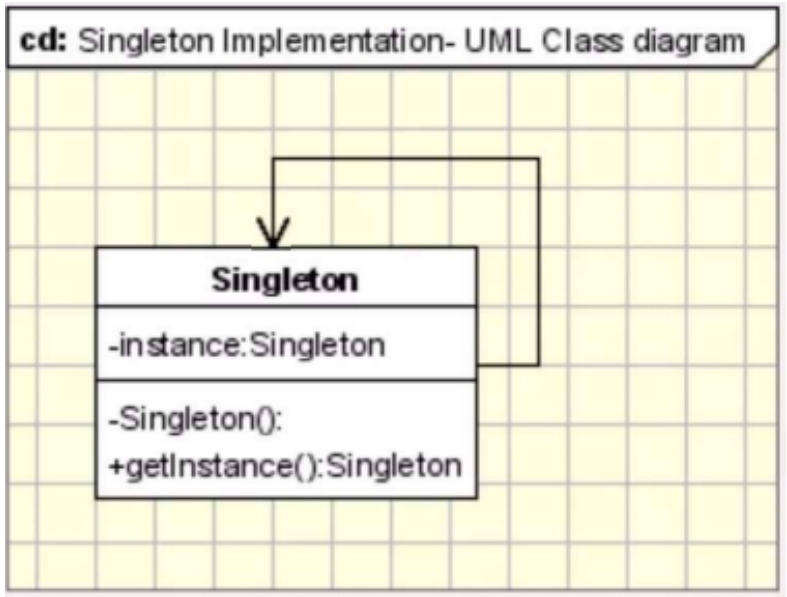
Creational Patterns

- Creational patterns abstract the instantiation process.
- They help to make a system independent of how its objects are created, composed, and represented
  - Creational patterns for classes use inheritance to vary the class that is instantiated.
  - Creational patterns for objects delegate instantiation to another object.
- Examples:
  - Singleton
  - Factory
  - Builder
  - Prototype

Singleton

- Ensure that only one instance of a class is created(allowed) within a system and provide a global access point to the object.





Cont'd

Singleton: Usage

- When to Use
  - When we must ensure that only one instance of a class is created
    - Exactly one instance of a class is required.
  - When the instance must be available through all the code
  - In multi-threading environments when multiple threads must access same resources through the same singlet object.

Example

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

Cont'd

Factory

- Creates objects without exposing the instantiation logic to the client
- Refers to the newly created object through a common interface.

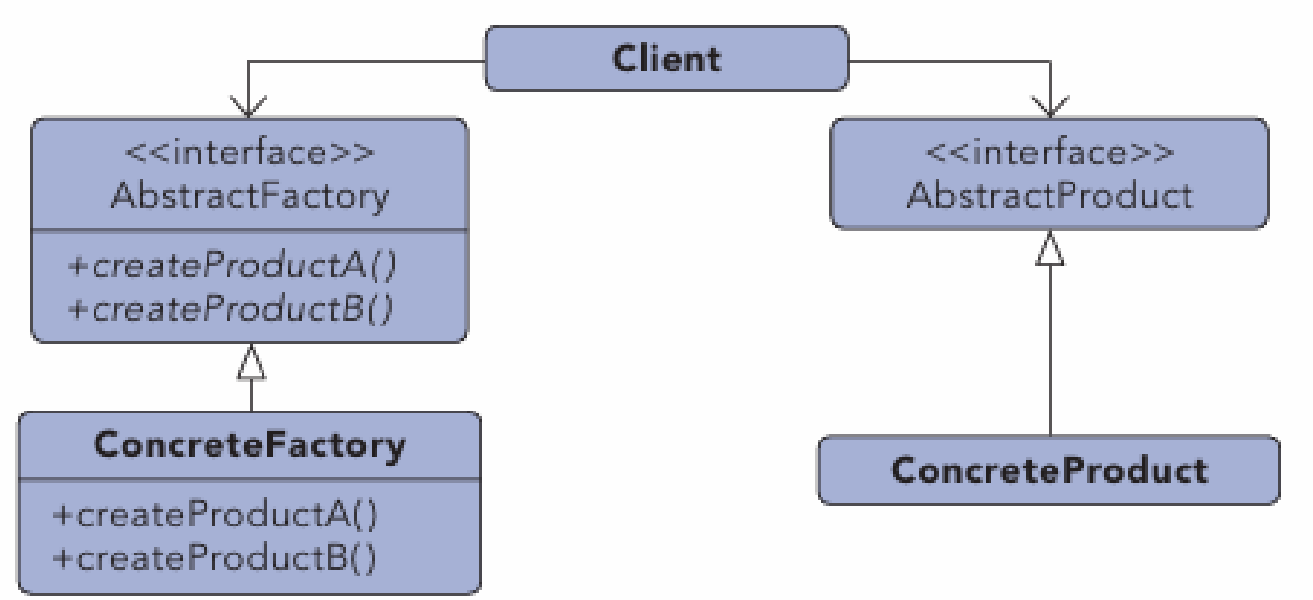
Factory: Usage

- When to use
  - When a framework delegates the creation of objects derived from a common superclass to the factory
  - When we need flexibility in adding new types of objects that must be created by the class
- Common Usage
  - factories providing an xml parser:
  - java.net.URLConnection

- Can be:
  - Abstract Factory
  - Factory Method

Abstract Factory

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



Cont'd

When to use

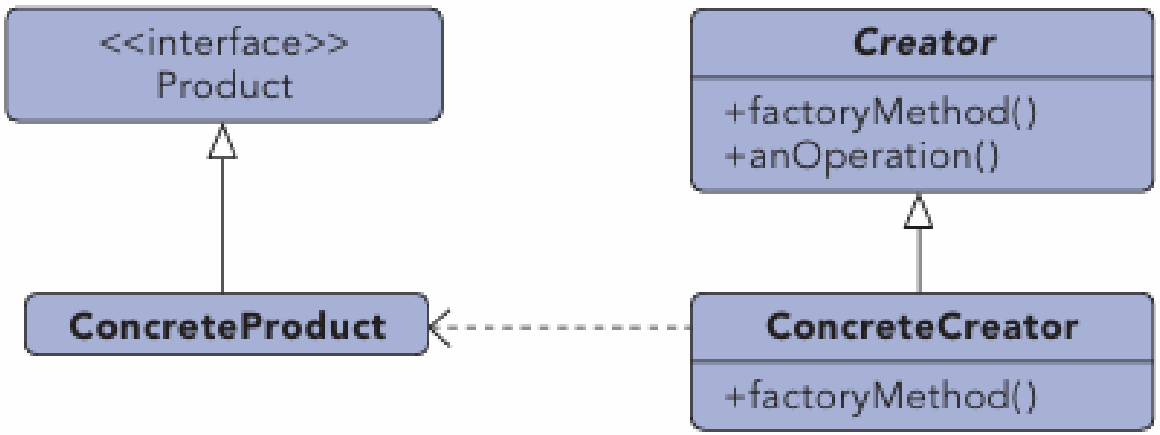
- Creation of objects should be independent of the system Using them.
- Systems should be capable of using multiple families of objects.
- Families of objects must be used together.
- Libraries must be published without exposing implementation details.
- Concrete classes should be decoupled from clients.

Example

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

Factory Methods

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



Cont'd

When to use

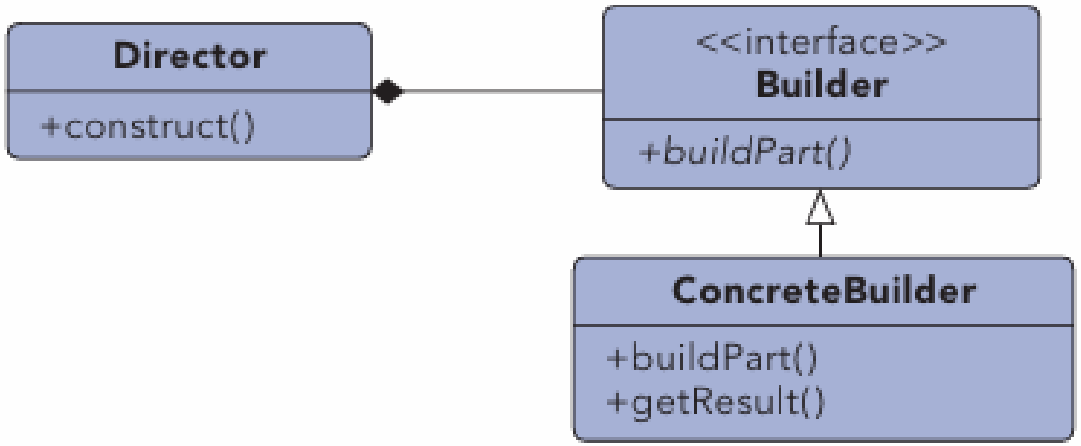
- A class will not know what classes it will be required to create.
- Subclasses may specify what objects should be created.
- Parent classes wish to defer creation to their subclasses.

Example

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

Builder

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



Cont'd

When to use

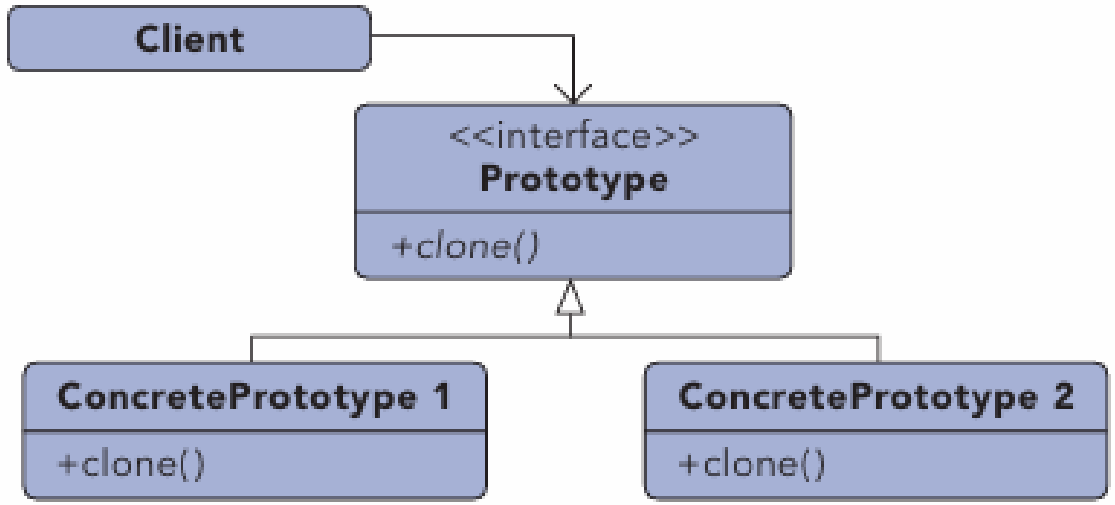
- Object creation algorithms should be decoupled from the system.
- Multiple representations of creation algorithms are required.
- The addition of new creation functionality without changing the core code is necessary.
- Runtime control over the creation process is required.

Example

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

Prototype

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



Cont'd

When to use

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

Example

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

Structural Patterns

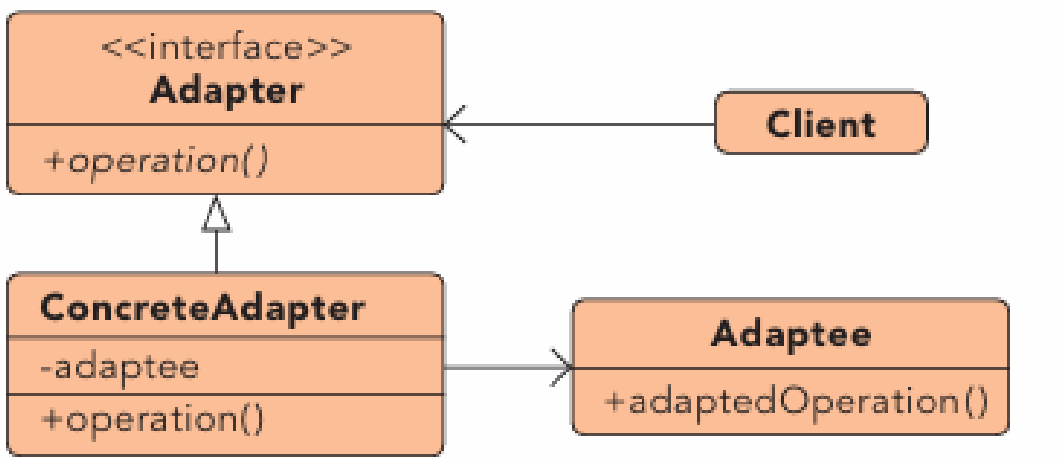
Definition

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
  - Structural class patterns use inheritance to compose interfaces or implementations.
  - Structural object patterns describe ways to compose objects to realize new functionality. the added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition.
- Examples:
  - Adapter
  - Proxy
  - Bridge
  - Composite

Cont'd

Adapter

- Converts the interface of a class into another interface the clients expect
- Permits classes with disparate interfaces to work together by creating a common object by which they may communicate and interact.
- Lets classes work together, that normally wouldn't, due to incompatible interfaces



Cont'd

When to use

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

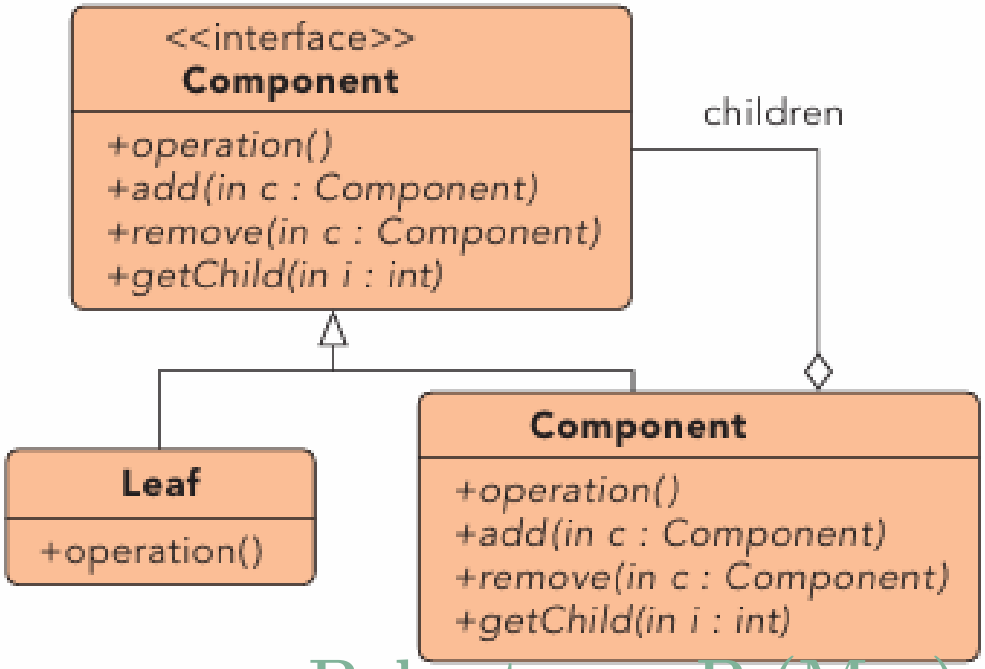
Example

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

Cont'd

Composite

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
- Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.



Cont'd

When to use

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

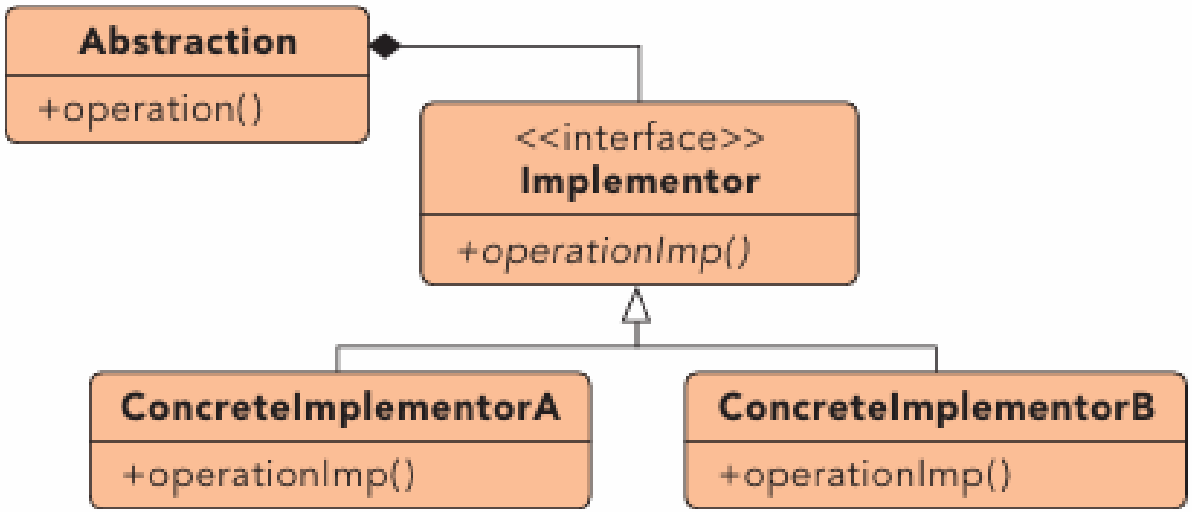
Example

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

Cont'd

Bridge

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



Cont'd

When to use

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

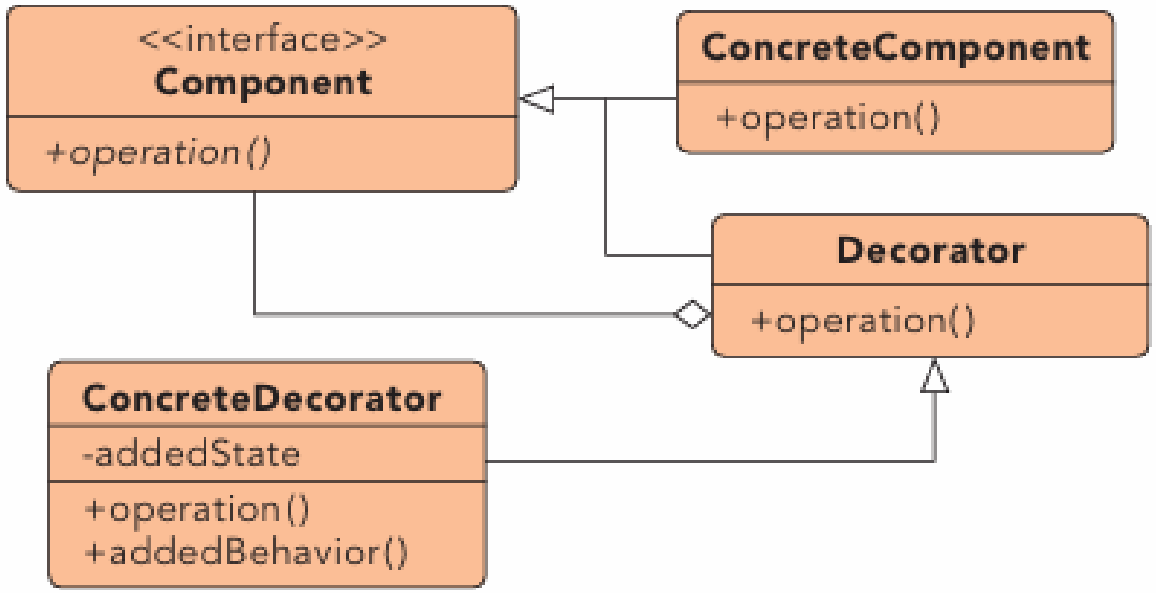
Example

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

Cont'd

Decorator

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



Cont'd

When to use

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

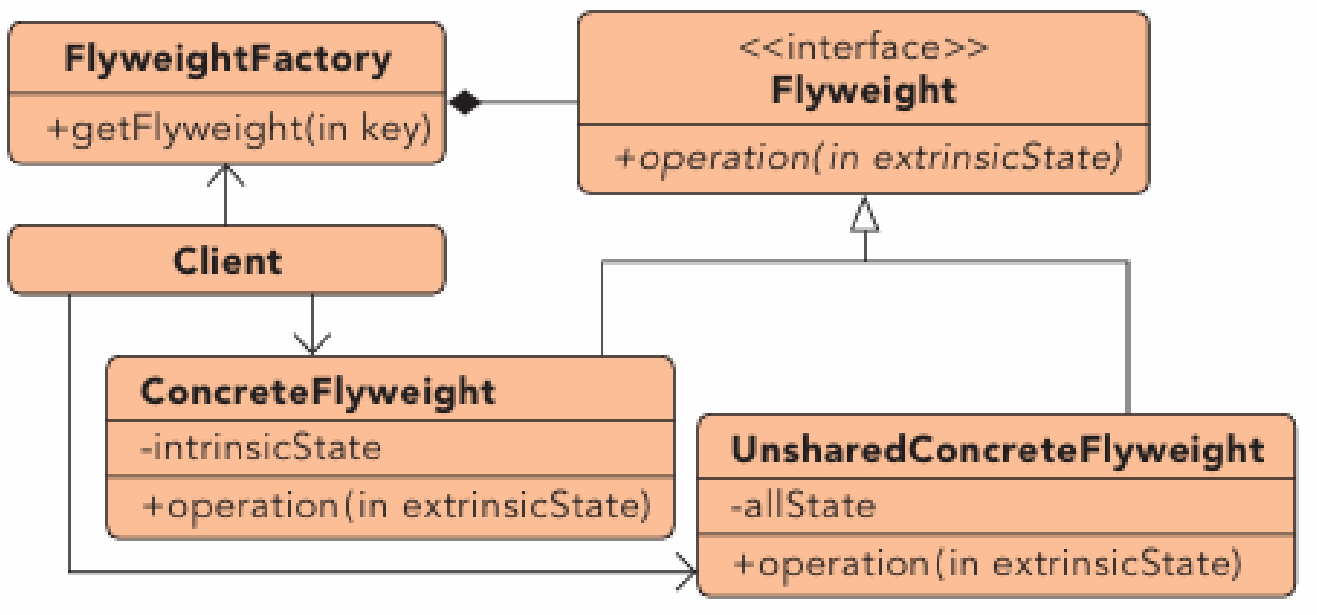
Example

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

Cont'd

FlyWeight

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



Cont'd

When to use

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

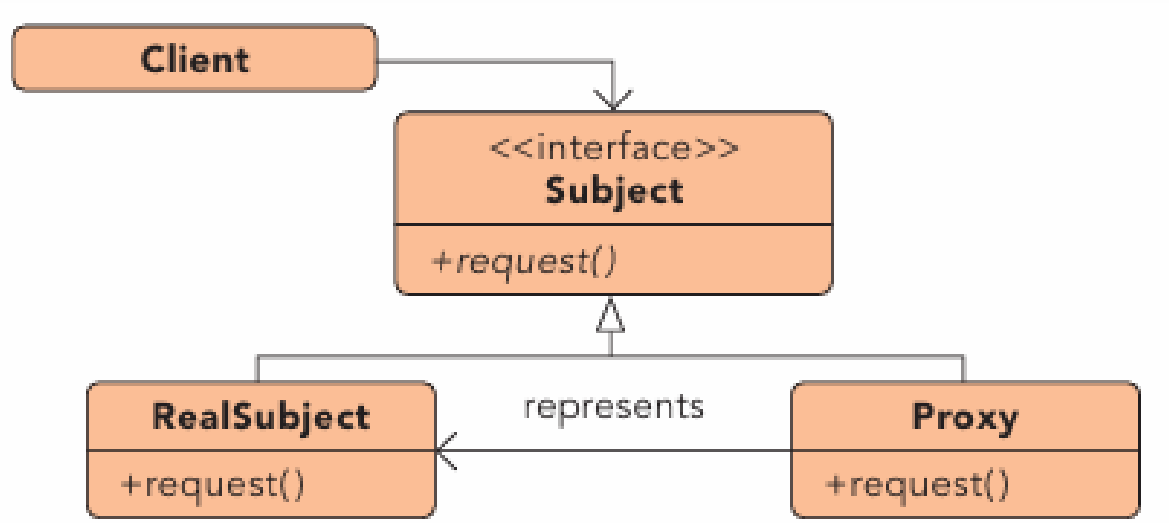
Example

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

Cont'd

Proxy

- Allows for object level access control by acting as a pass through entity or a placeholder object.
- Provide a "Placeholder" for an object to control references to it



Cont'd

When to use

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

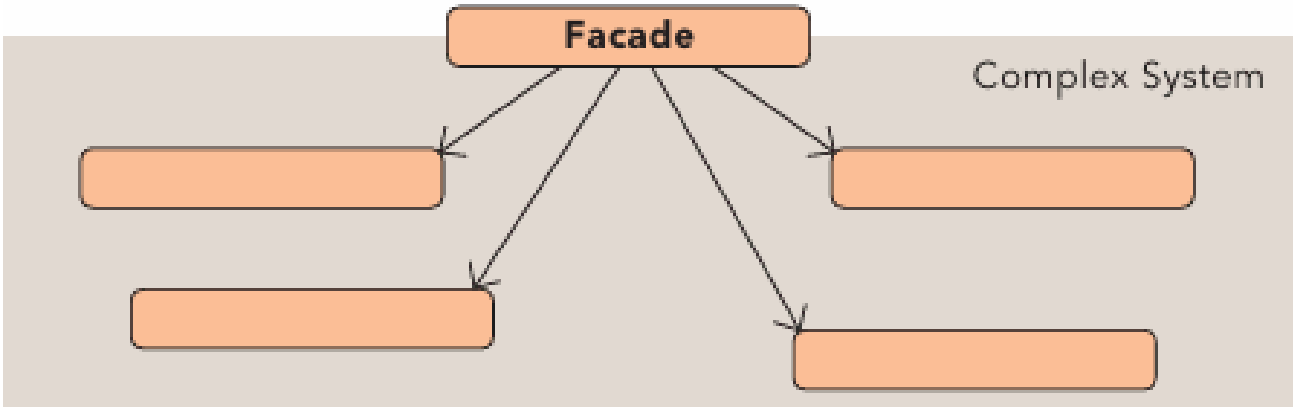
Example

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

Cont'd

Facade

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



Cont'd

When to use

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

Example

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

Behavioral Patterns

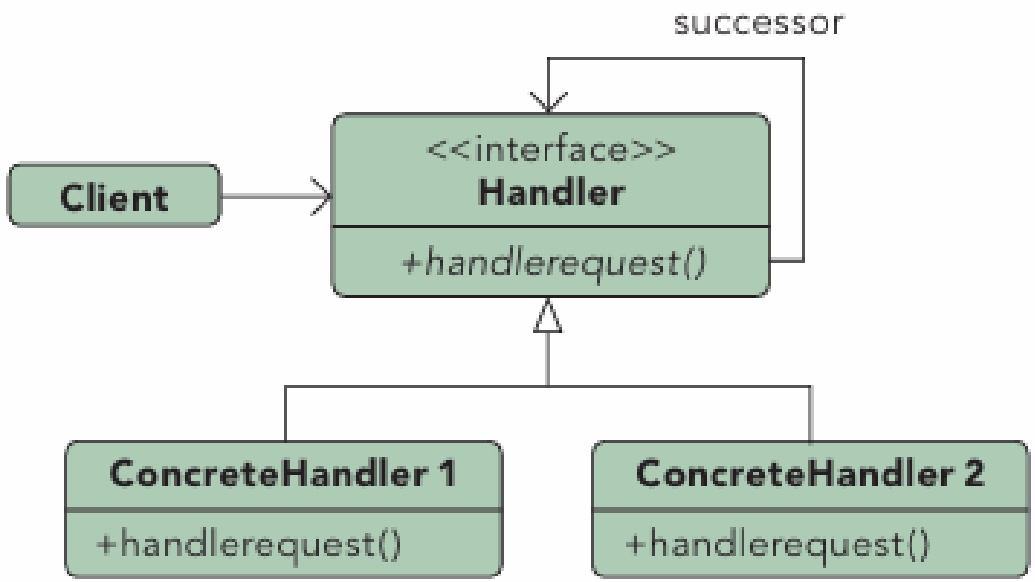
Definition

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects
  - Behavioral class patterns use inheritance to distribute behavior between Classes.
  - Behavioral object patterns use composition rather than inheritance. Some describe now a group of peer objects cooperate to perform a task that no Single object can carry out by Itself.
- Examples:
  - Command
  - Iterator
  - Observer
  - Strategy

Cont'd

Chain of Responsibility

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



Cont'd

When to use

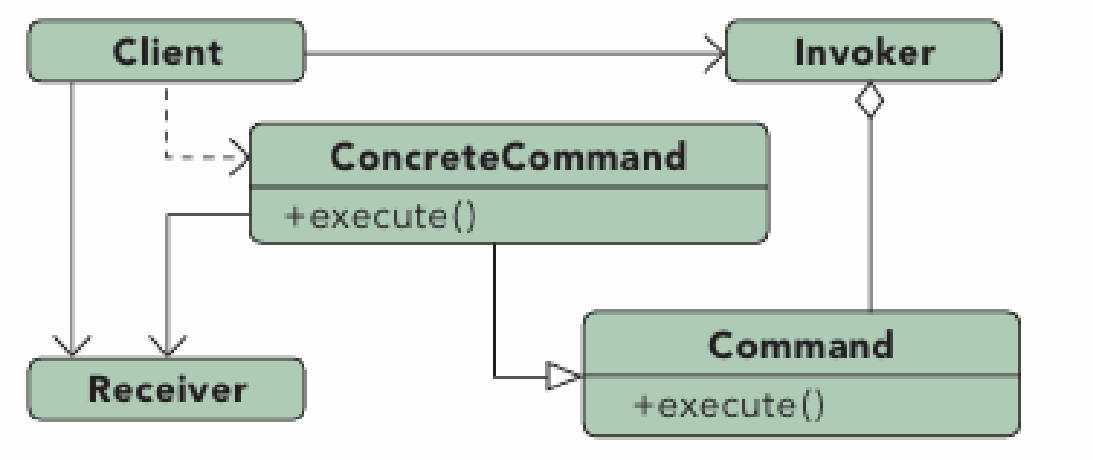
- Multiple objects may handle a request and the handler doesn't have to be a specific object.
- A set of objects should be able to handle a request with the handler determined at runtime.
- A request not being handled is an acceptable potential outcome

Example

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

Command

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



Cont'd

When to use

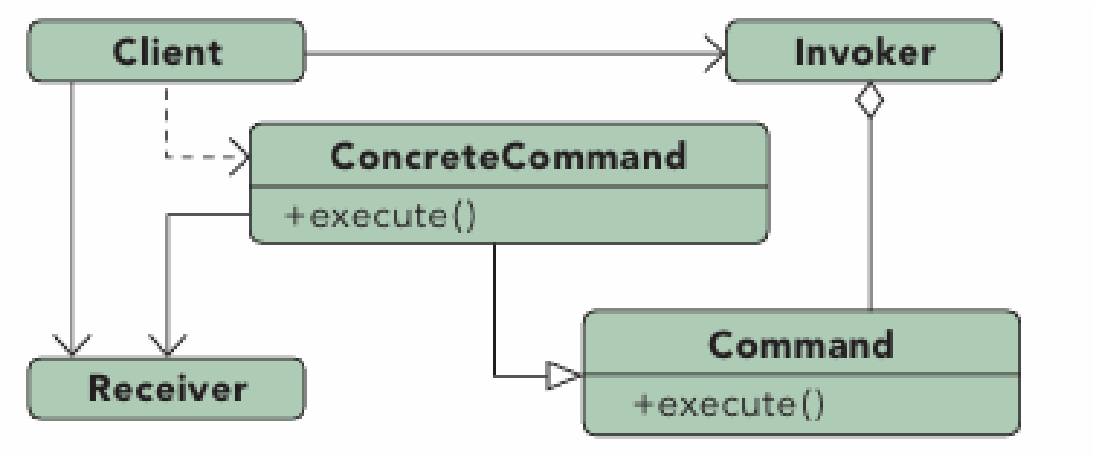
- You need callback functionality.
- Requests need to be handled at variant times or in variant orders.
- A history of requests is needed.
- The invoker should be decoupled from the object handling the invocation.

Example

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

Interpreter

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



Cont'd

When to use

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

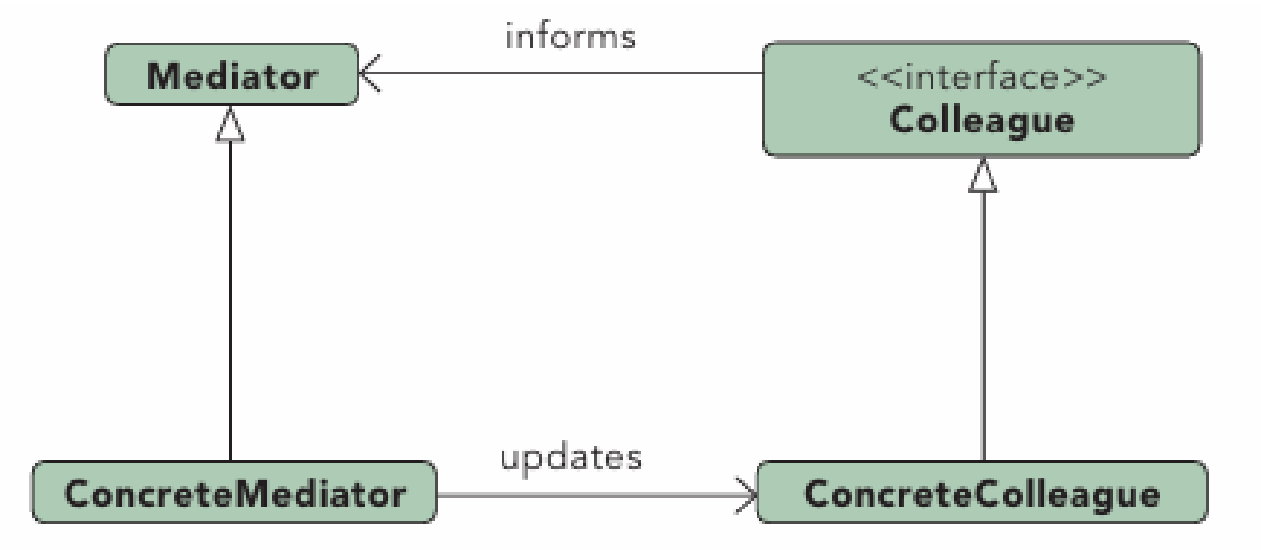
Example

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

Cont'd

Mediator

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



Cont'd

When to use

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

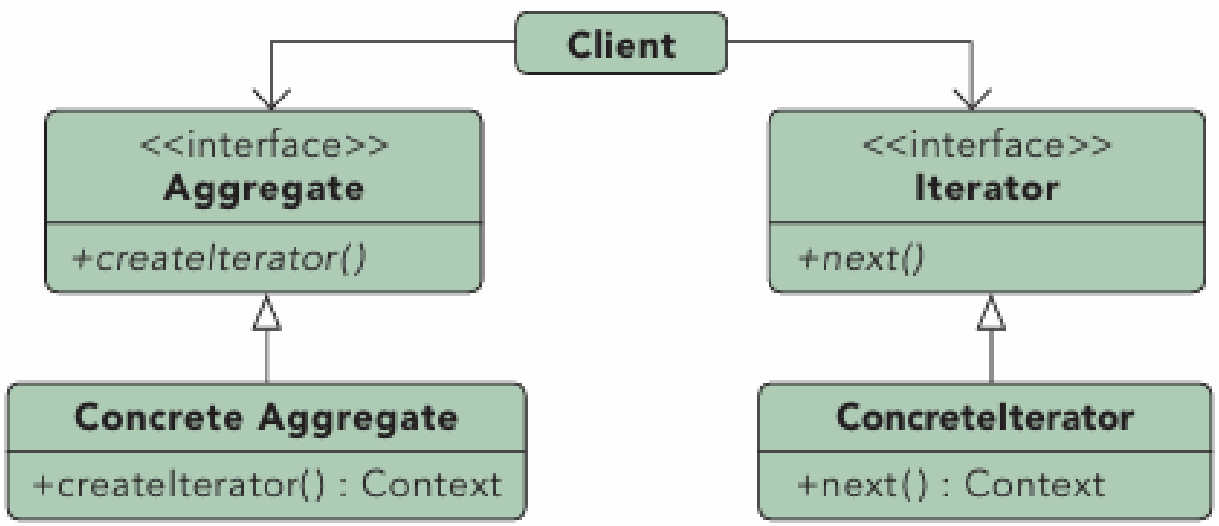
Example

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

Cont'd

Iterator

- Allows for access to the elements of an aggregate object without allowing access to its underlying representation.
- The iterator pattern allows us to:
  - Access contents of a collection without exposing its internal Structure.
  - support multiple simultaneous traversals of a collection.
  - Provide a uniform interface for traversing different collections



Cont'd

When to use

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

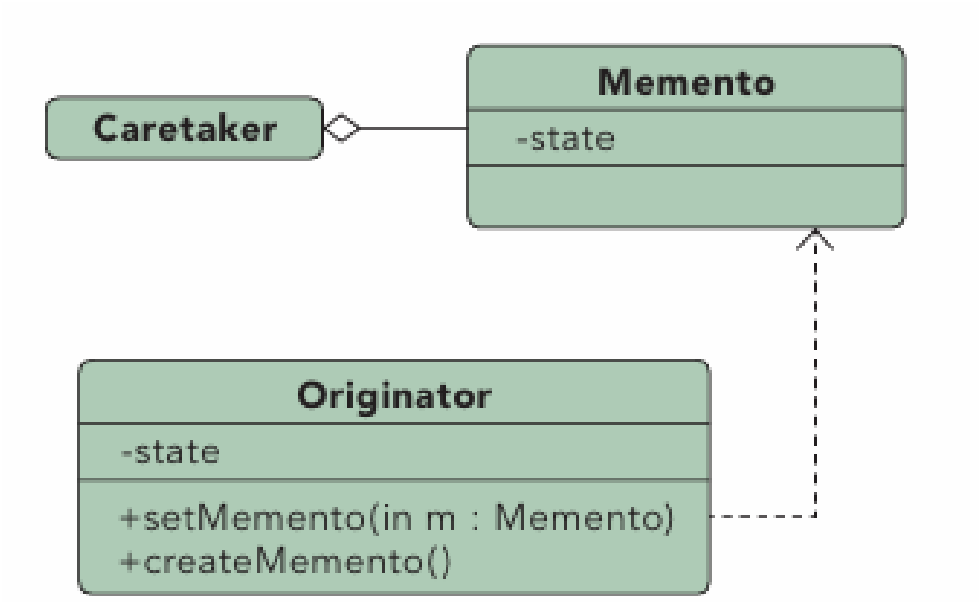
Example

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

Cont'd

Memento

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



Cont'd

When to use

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

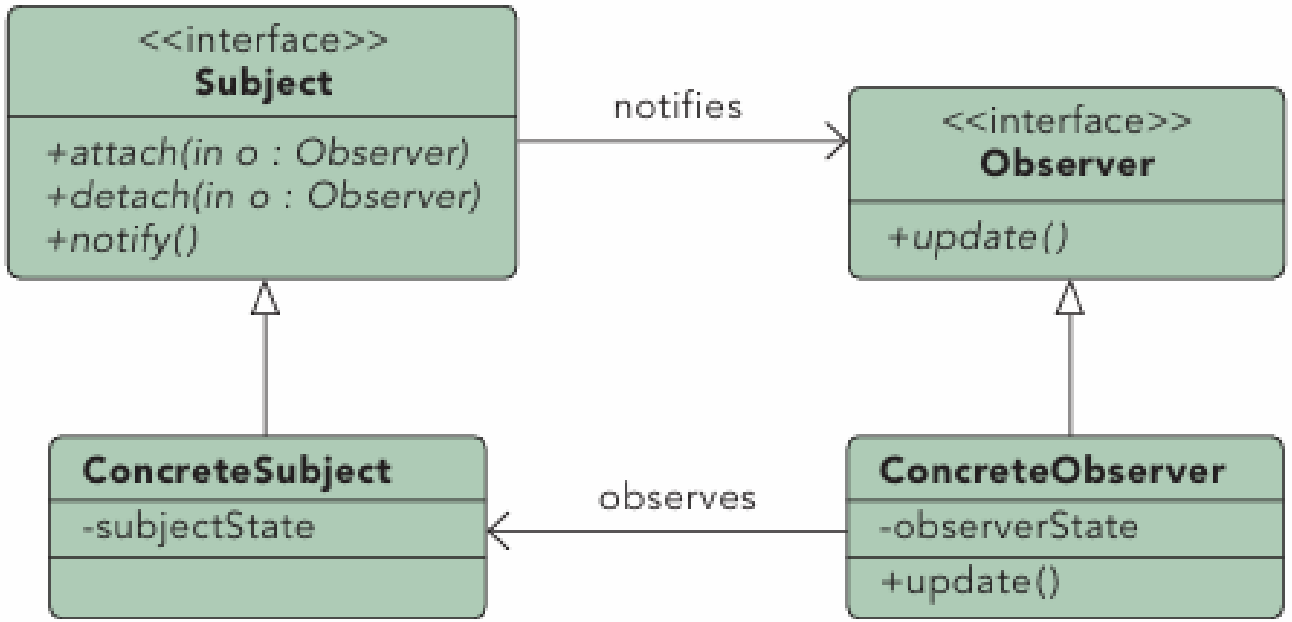
Example

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

Cont'd

Observer

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



Cont'd

When to use

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

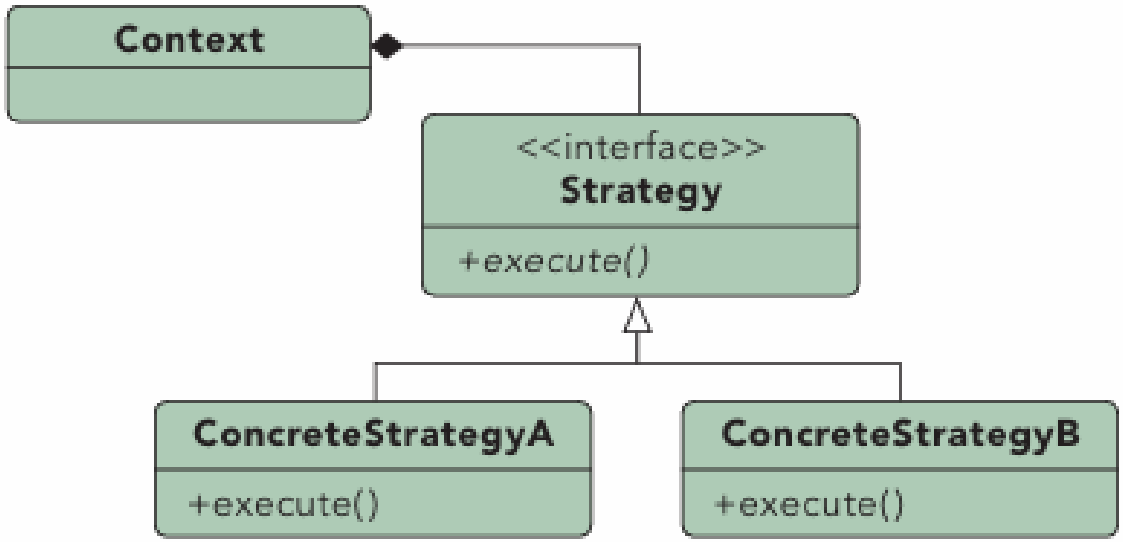
Example

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

Cont'd

Strategy

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



Cont'd

When to use

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

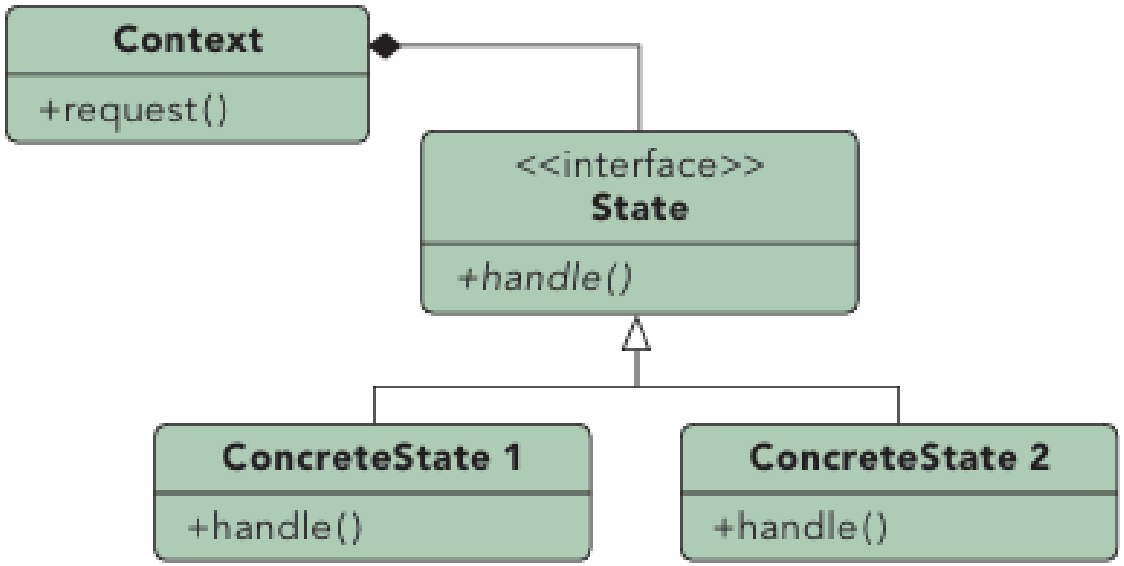
Example

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

Cont'd

State

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



Cont'd

When to use

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

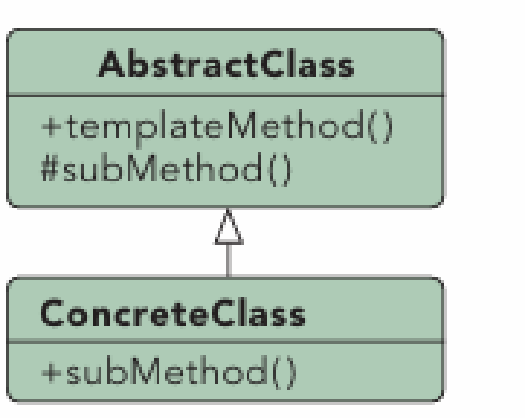
Example

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

Cont'd

Template Method

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



Cont'd

When to use

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

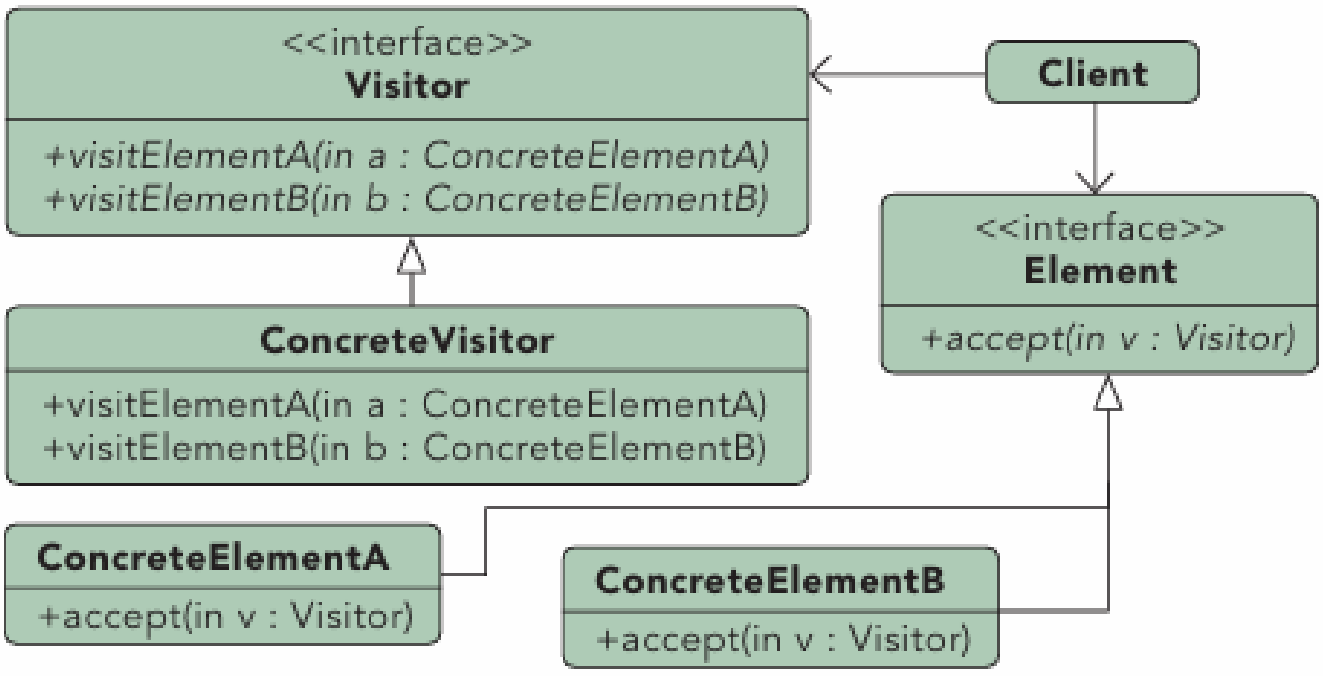
Example

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

Cont'd

Visitor

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



Cont'd

When to use

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

Example

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

Thank You...!!!!