Patterns

1. Introduction

Motivating Example

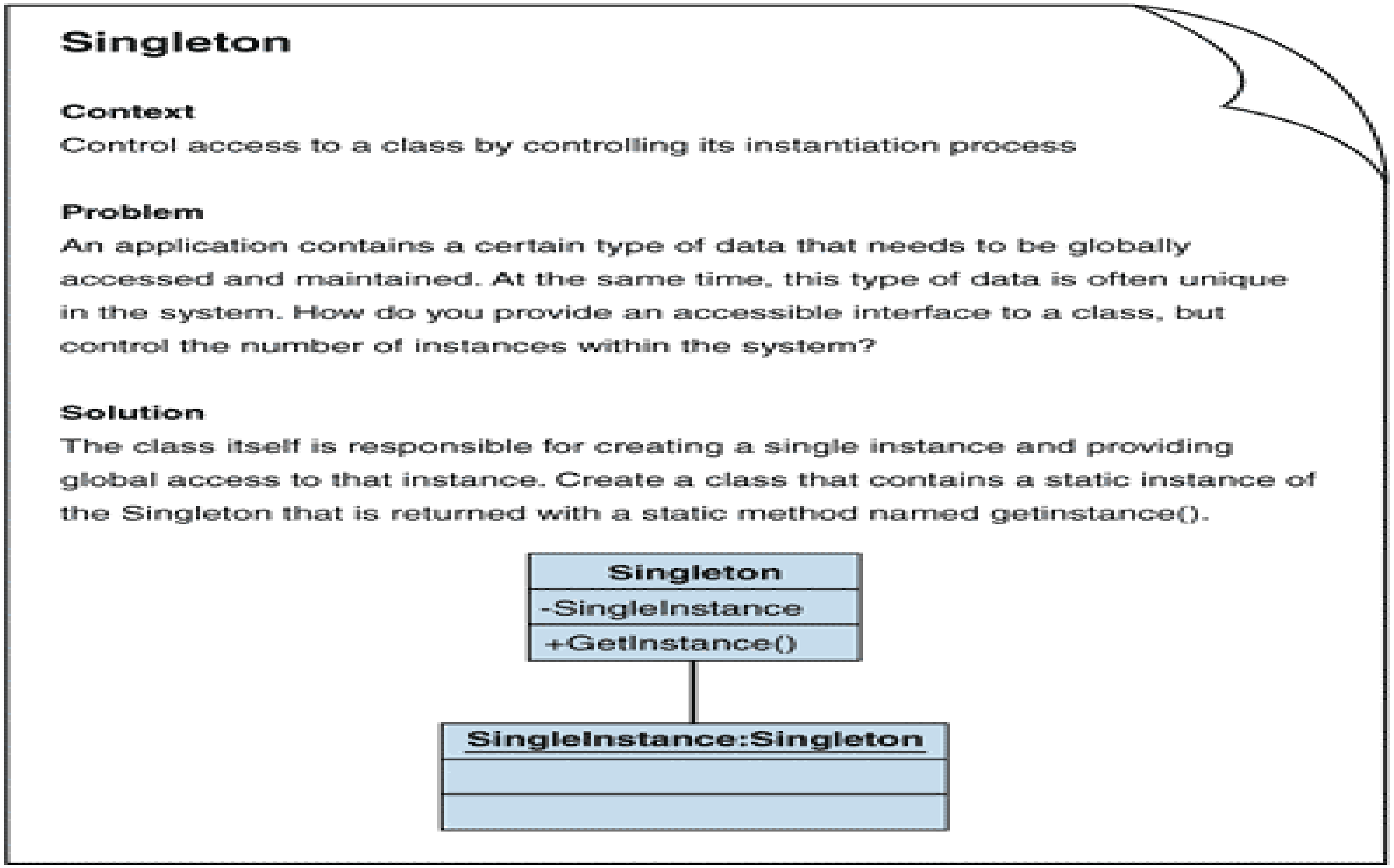
Problem: *You are building a quote application, which contains a class that is responsible for managing all of the quotes in the system. It is important that all quotes interact with one and only one instance of this class. How do you structure your design so that only one instance of this class is accessible within the application?*

Simple solution:

- Create a **QuoteManager** class with a private constructor so that no other class can instantiate it.
- This class contains a static instance of **QuoteManager** that is returned with a static method named **GetInstance()**.

```
public class QuoteManager { //NOTE: For single threaded applications only
        private static QuoteManager _Instance = null;
        private QuoteManager() {}
        public static QuoteManager GetInstance() {
                if (_Instance==null) {
                        _Instance = new QuoteManager ();
                }
                return _Instance;
        }
//... functions provided by QuoteManager
}
```

- Recurring problems like the previous kind have been observed by experienced developers and a common solution has been distilled, as the ***Singleton*** pattern.



Singleton

Context
Control access to a class by controlling its instantiation process

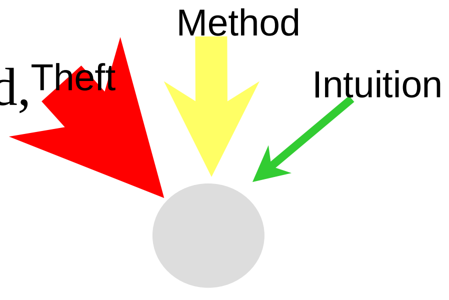
Problem
An application contains a certain type of data that needs to be globally accessed and maintained. At the same time, this type of data is often unique in the system. How do you provide an accessible interface to a class, but control the number of instances within the system?

Solution
The class itself is responsible for creating a single instance and providing global access to that instance. Create a class that contains a static instance of the Singleton that is returned with a static method named getinstance().

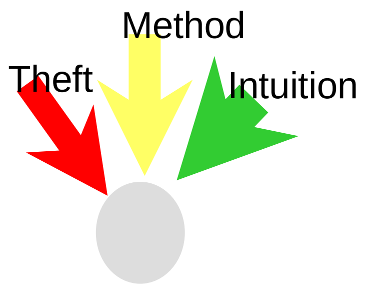
- Notice that unlike the previous source code, the *Singleton* pattern description does not mention a **Quote** or a **QuoteManager** class.
- Instead, the pattern description illustrates a generalized problem-solution pair, while the application of the pattern yields a very specific solution to a very specific problem.

Pattern Overview

- A classical engineering rule of thumb consists of reusing extensively available knowledge base for any kind of problem solving that occurs during system design.
  - When the knowledge doesn't exist for the problem at hand, then intuition takes the lead.
  - Architectural styles or patterns define experience-based classes of designs along with their known characteristics.



Classical system



Unprecedented system

- A pattern is a ***solution to a problem in a context***
  - A pattern codifies specific knowledge collected from experience in a domain
  - There are three kinds of patterns
    - –***Idioms:*** provide a solution for specific programming issues in given language.
    - –***Design patterns:*** provide solutions to commonly recurring design issues.
    - –***Architectural patterns:*** provide templates for designing entire architectures.

2. Pattern Description and Application

Pattern Description

-A pattern can be described by using a three-part schema:

- ***Context***: description of situations in which the problem arise.
- ***Problem***: general description of the design issues that need to be solved by the pattern, and the driving forces or (risk, quality) factors behind these issues.
- ***Solution***: the solution consists of two aspects: a static aspect that describes the structural components involved, a dynamic aspect that describes their run-time behaviour.

Pattern Name
Context - Quality/Risk factors giving rise to design issue (s)
Problem - Design issues
Solution -Components/connectors -Run-time behaviour

-The solution can be further described by providing UML models.

Example of Pattern Application: the Command Pattern

-Suppose that we build a simple program allowing users to edit files:

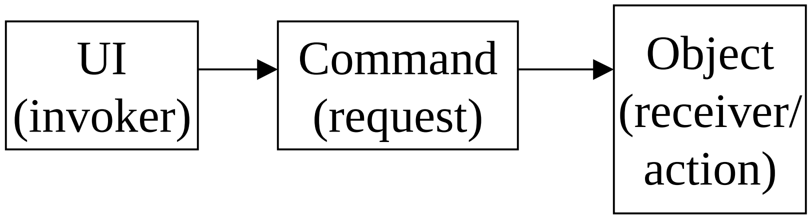
```
public class UI {
    public void actionPerformed(ActionEvent e) {
        Object obj = e.getSource();
        if (obj==mnuOpen) fileOpen(); //open file
        if (obj==mnuExit) exitClicked(); //exit from program
        if (obj==mnuClose) fileClose(); //close file
                        …
    }
    private void exitClicked() {System.exit(0); }
    private void fileOpen() {…}
}
```

- The *actionPerformed* code can get pretty unwieldy as the number of menu items and buttons increases.
- To avoid this, possible solution consists of using the ***Command*** pattern.

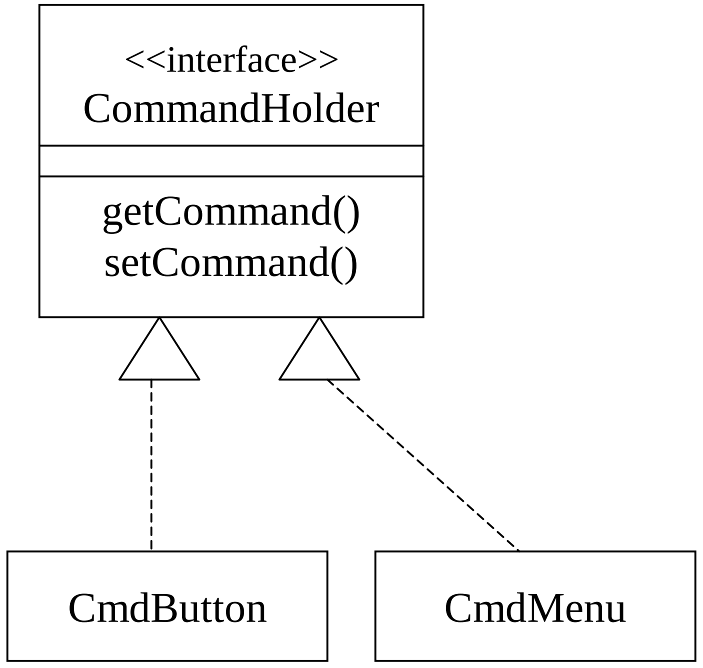
-The command pattern forwards client requests to corresponding objects.

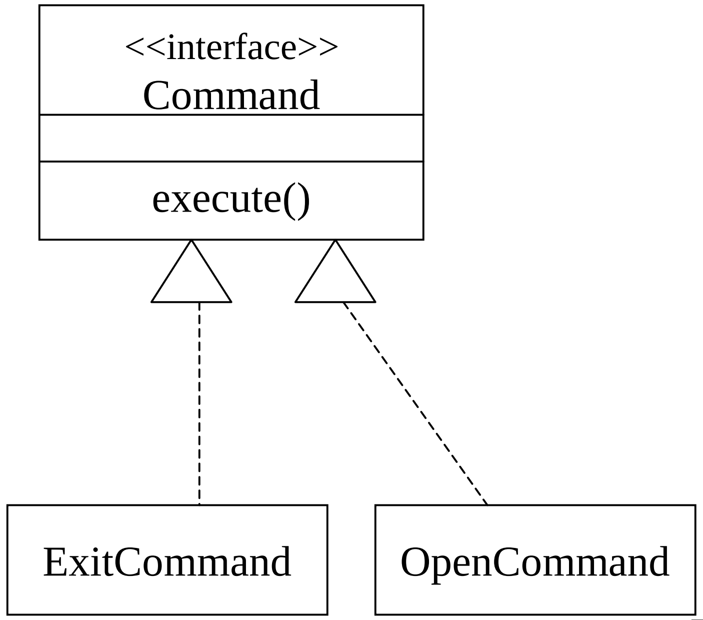
- Its main purpose is to keep the program and user interface objects completely separate from the actions they initiate.
- This decouples the UI class from the execution of specific commands, thereby making it possible to modify the action code independently.

-The ***Command pattern*** provides an alternative by creating individual *Command* objects and ensuring that every object receives its own command directly.



- *Command* objects implement the *Command* interface:
- UI elements (menu, button etc.) implement a *CommandHolder* interface, which provides a place holder for Command objects.





-Example of command implementation:

- *Command objects:*

```
/*Command objects implement the Command interface*/

public interface Command {
        public void execute();
  }

/*Example of Command  object implementation*/

public class OpenCommand implements Command {
      JFrame frame;
      public OpenCommand(JFrame fr) {frame=fr;}
      public void execute() {
          FileDialog fdlg = new FileDialog(frame, “Open file”);
          fdlg.show();          //show file dialog
       }
}
```

•*CommandHolder objects:*

/*UI elements (menu, button etc.) implement a CommandHolder interface*/

```
   public interface CommandHolder{
        public void setCommand(Command cmd);
        public Command getCommand();
   }
```

/*Example of CommandHolder Implementation*/

```
public class CmdMenu extends JMenuItem implements CommandHolder {
        protected Command menuCommand; //internal copies
        protected JFrame frame;

        public CmdMenu(String name,JFrame frm) {
              super(name);                //menu string
              frame=frm;                  //containing string
         }
        public void setCommand(Command cmd) {
              menuCommand=cmd;       //save the command
         }
        public Command getCommand() {
               return menuCommand; //return the command
         }
}
```

- *UI objects implementation:*

```
public class UI {
    public void actionPerformed(ActionEvent e) {
          CommandHolder obj = (CommandHolder) e.getSource();
          Command cmd = obj.getCommand();
          cmd.execute();
     }


  public UI () {
                …
 /*Create instances of the menu and pass them different command objects*/
   mnuOpen = new CmdMenu("Open…",this);
   mnuOpen.setCommand(new OpenCommand(this));
   mnuFile.add(mnuOpen);


                …
  }
                …
 }
```

Traditional Design without the Command Pattern

```
public class UI {
     public void actionPerformed(ActionEvent e) {
          Object obj = e.getSource();
          if (obj==mnuOpen) fileOpen(); //open file
          if (obj==mnuExit) exitClicked(); //exit from program
          if (obj==mnuClose) fileClose(); //close file
                         …
     }
     private void exitClicked() {System.exit(0); }
     private void fileOpen() {…}
}
```

Modeling a Design Pattern Using the UML

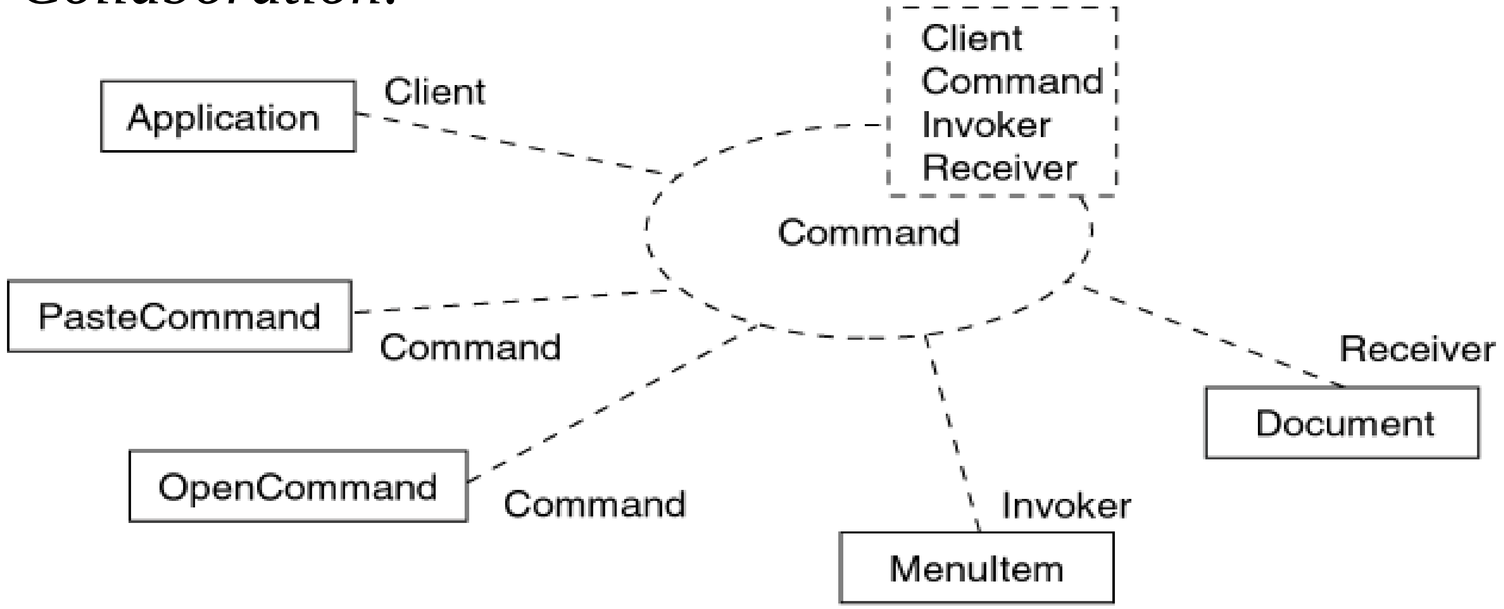
•***Design patterns*** are modeled by providing both external and internal views:

-From outside: as a parameterized collaboration.

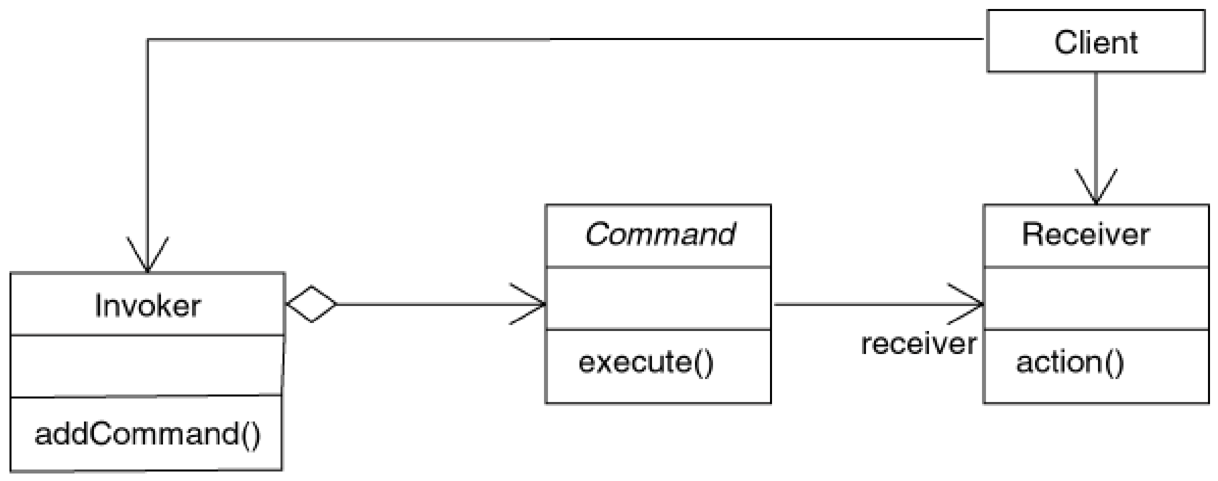
-From inside: as a collaboration with its structural (class diagrams) and behavioral parts (sequence diagrams).

Example: The Command Pattern

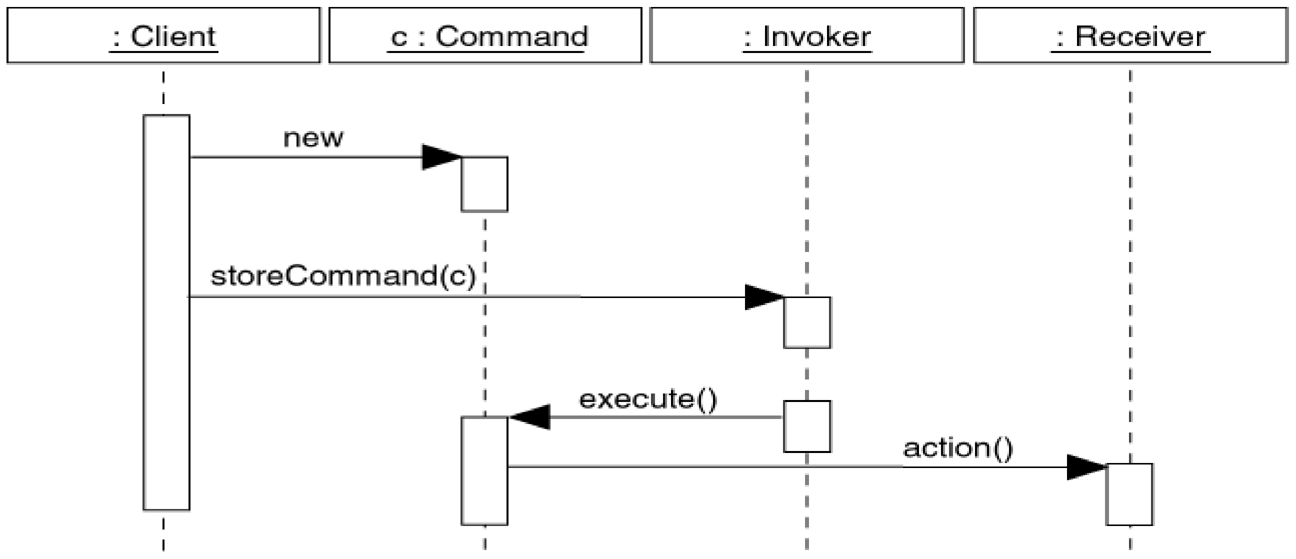
☞ *Collaboration:*



☞ *Class Diagram:*



☞ *Interactions:*



3. Design Patterns

-Design patterns: provide solutions to commonly recurring design issues.

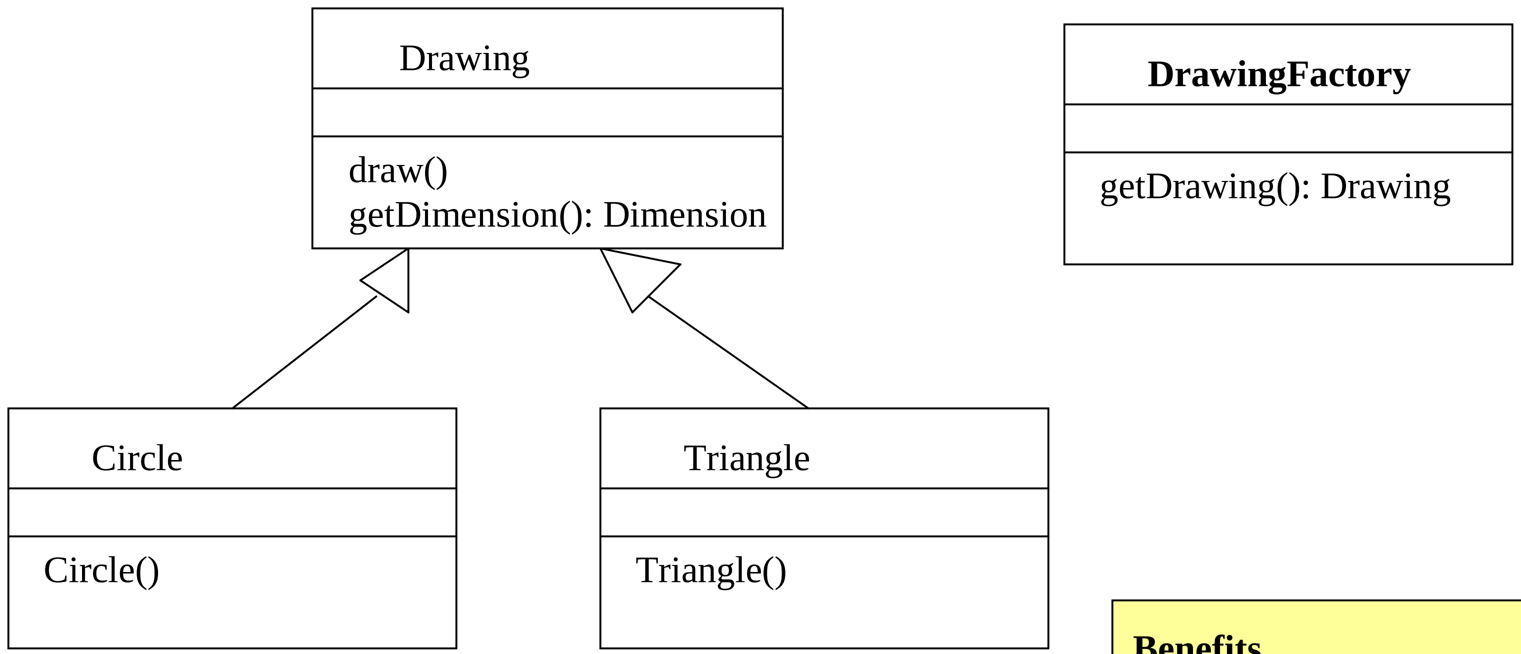
Categories

-Three categories of design patterns: creational, structural, and behavioral.

- ***Creational patterns***: create instances of objects for your application
- ***Structural patterns***: compose groups of objects into larger structures
- ***Behavioral patterns***: define the communication and flow of control in a complex program.

Example of Creational Pattern

- ***Factory pattern:*** provides a decision-making class that returns selectively one of several possible subclasses of an abstract base class.



```
public class DrawingFactory {
  public Drawing getDrawing(int criteria) {
     if (criteria = 0) return new Triangle();
     else return new Circle();
  }
}
```

Benefits

-Code is made more flexible and reusable by the elimination of instantiation of application-specific classes

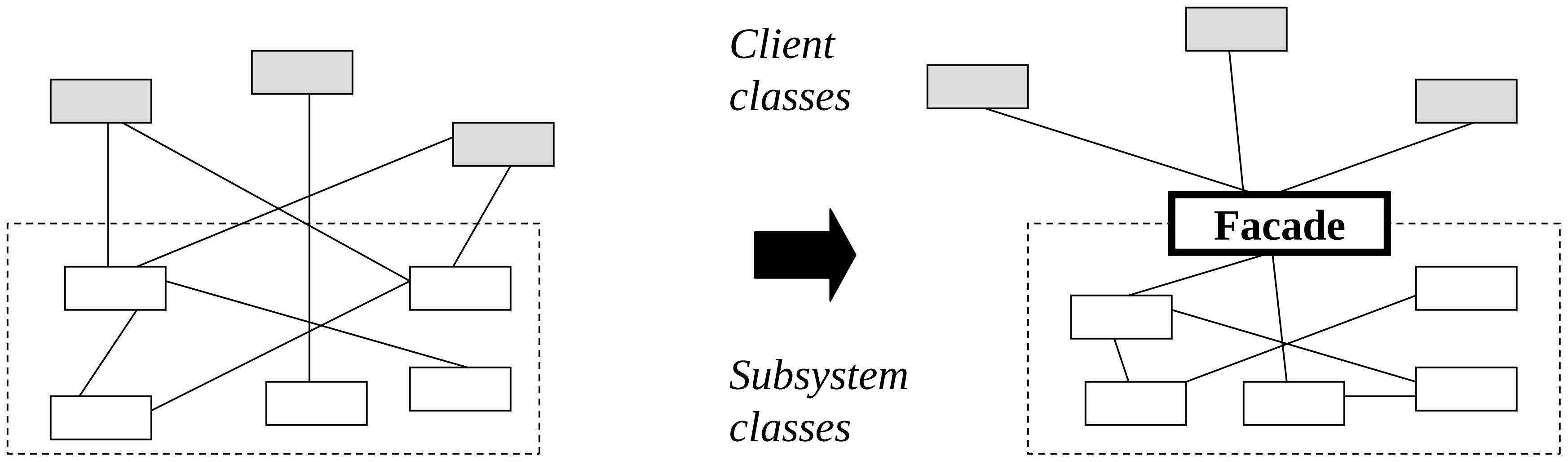
-Code deals only with the interface of the Product class and can work with any ConcreteProduct class that supports this interface

Example of Structural Pattern

Façade Pattern:

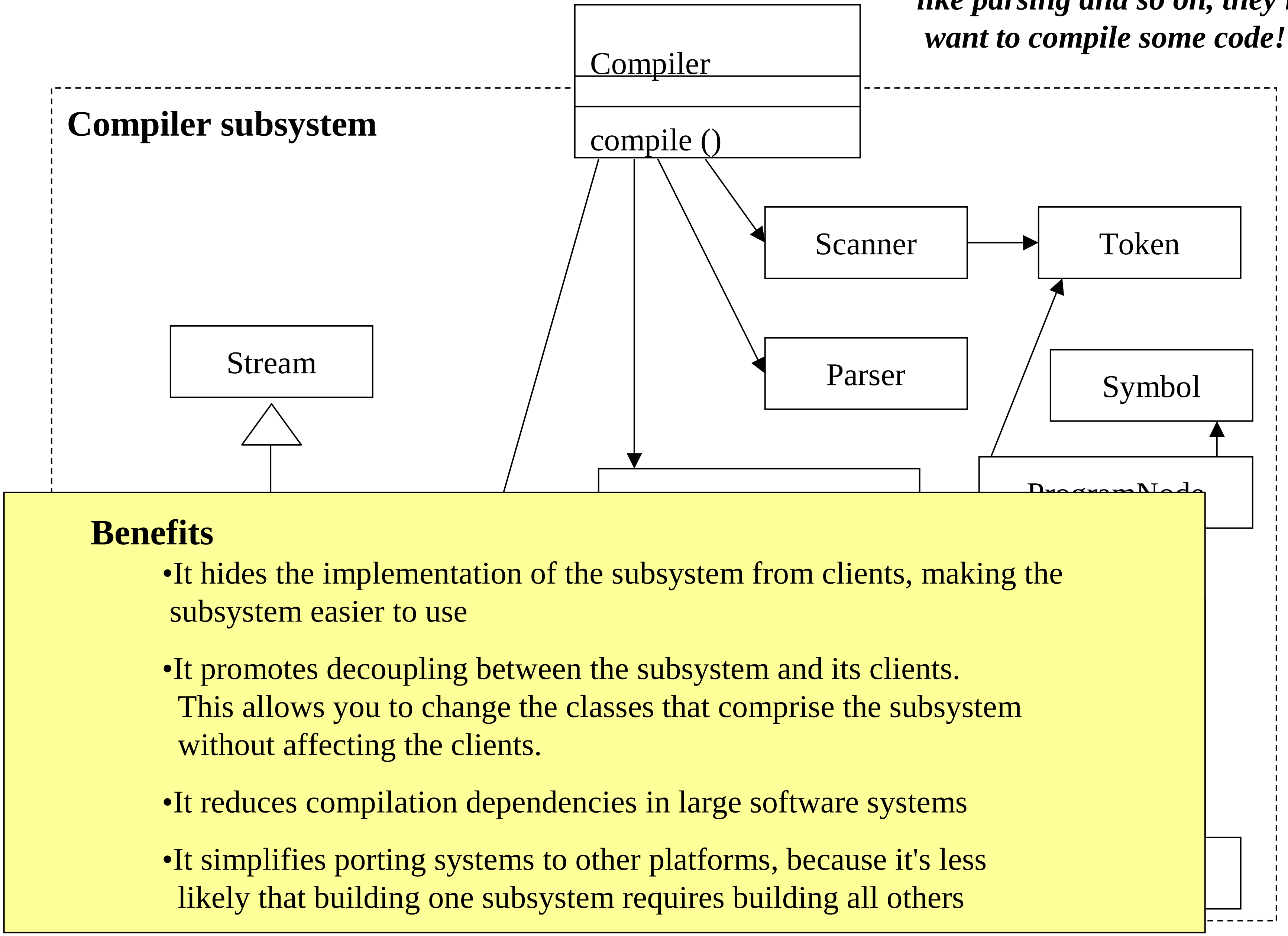
-Is used to wrap a set of complex classes into a simpler enclosing interface.

-Provides a unified and higher-level interface that makes a subsystem easier to use.



Example: a compiler subsystem

Most clients don't care about details like parsing and so on, they merely want to compile some code!!!!!!

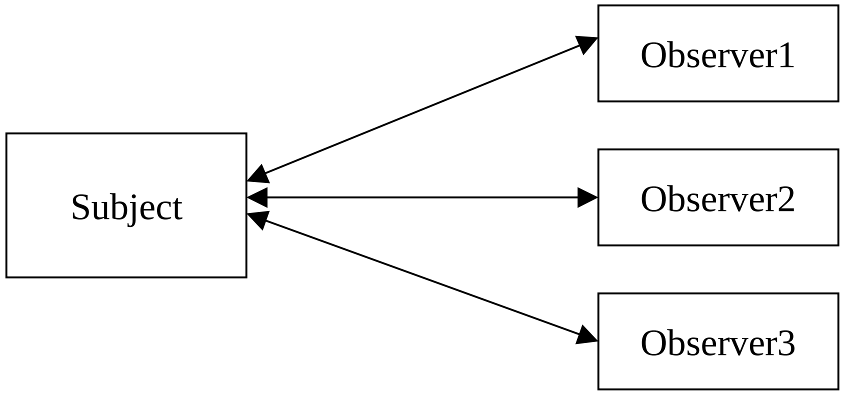


Benefits

- It hides the implementation of the subsystem from clients, making the subsystem easier to use
- It promotes decoupling between the subsystem and its clients. This allows you to change the classes that comprise the subsystem without affecting the clients.
- It reduces compilation dependencies in large software systems
- It simplifies porting systems to other platforms, because it's less likely that building one subsystem requires building all others

Example of Behavioral Pattern
- Observer Pattern

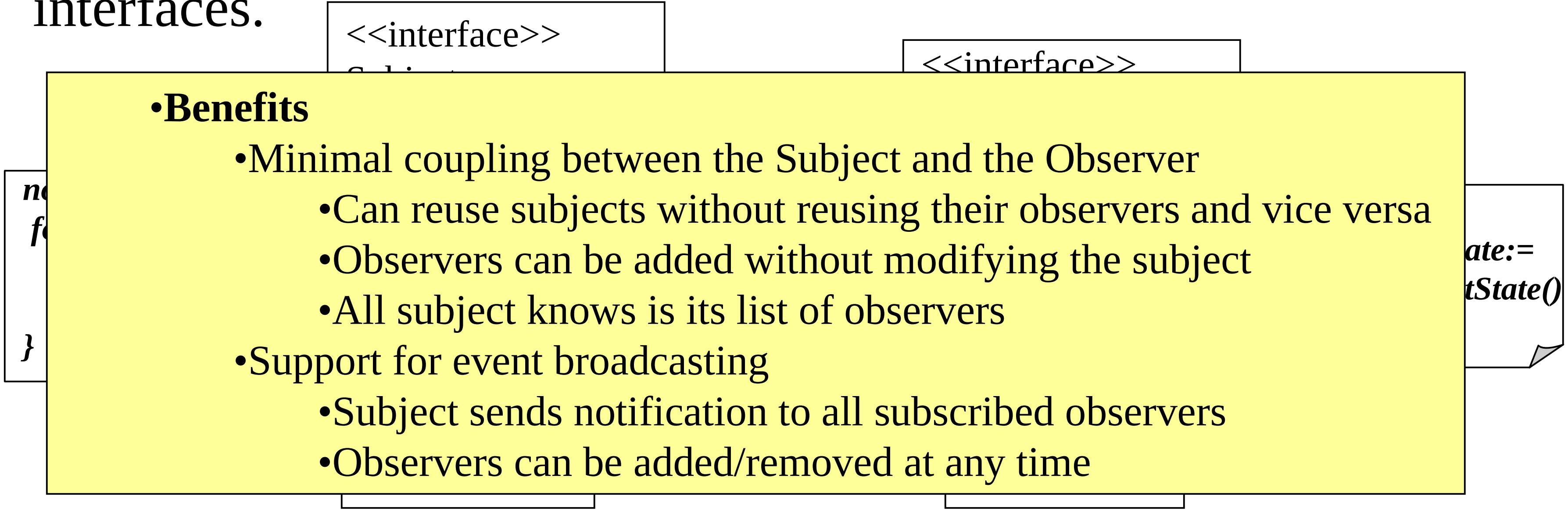
- Defines how several objects can be notified of a change.
- Maintains dependency between objects so that when the state of one object changes, the other objects are automatically notified and updated.



- Two kinds of components:

1. ***Observer***: component that might be interested in state changes of the subject. The observer registers interest in specific events, and defines an updating interface through which state change notifications are made
2. ***Subject*:** knows its observers and is expected to provide an interface for subscribing and unsubscribing observers.

- Observers and subjects objects may implement the following interfaces.



- **Benefits**
  - Minimal coupling between the Subject and the Observer
    - Can reuse subjects without reusing their observers and vice versa
    - Observers can be added without modifying the subject
    - All subject knows is its list of observers
  - Support for event broadcasting
    - Subject sends notification to all subscribed observers
    - Observers can be added/removed at any time

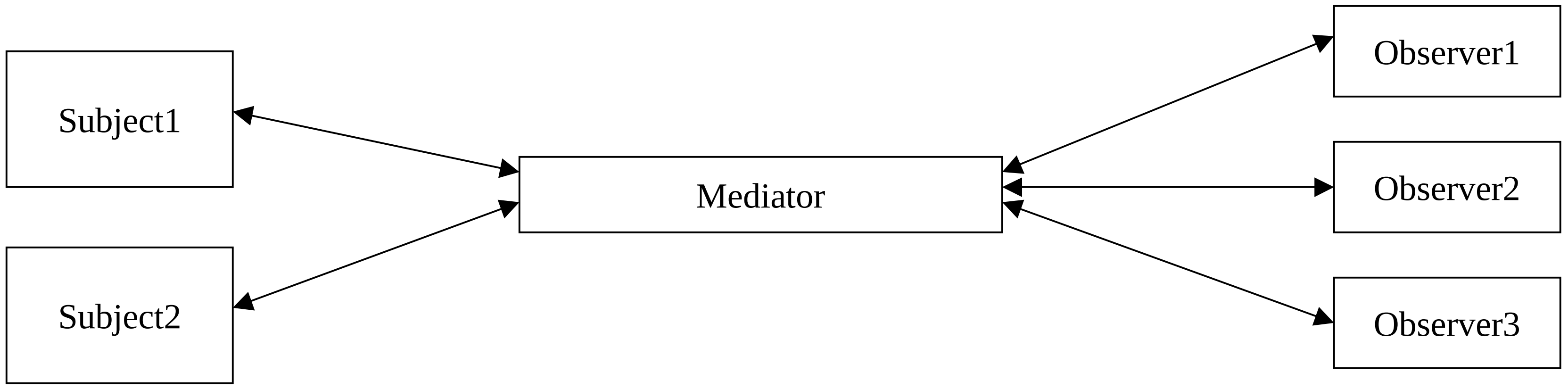
Examples
- Bound properties
  - Veto case
- ***Event handling***
  - JDK 1.2+

A.K.A:
- Publisher Subscriber

```
public interface Observer {
  //notify an individual observer that a change has taken place
   public void update();
 }
public interface Subject {
  //tell the subject that an object is interested in changes
  public void register (Observer obs);
  //notify observers that a change has taken place
  public void notify();
}
```

-*Mediator pattern*

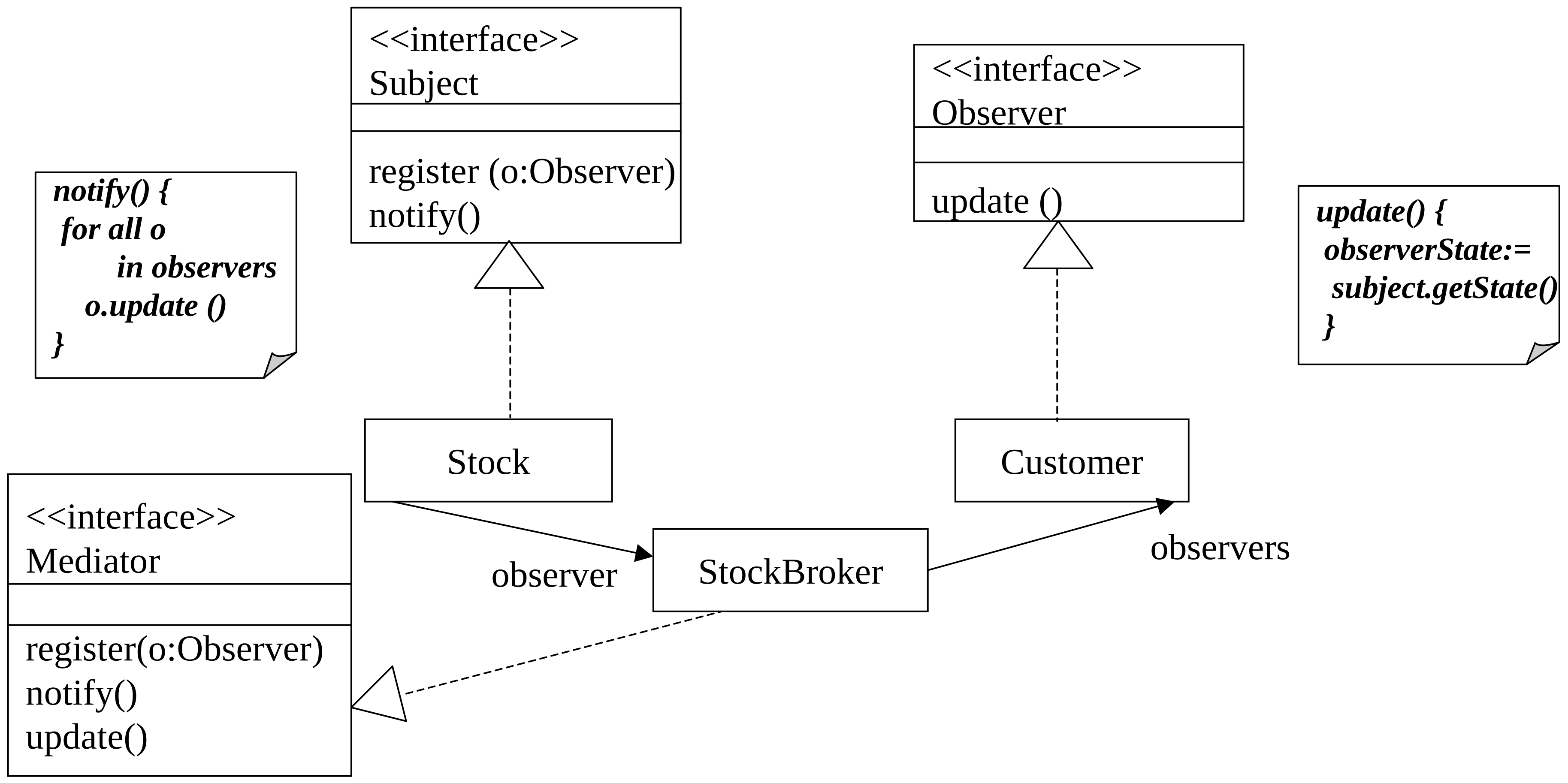
•In case where observers interact with several subjects and/or vice-versa it is more appropriate to extend the ***observer pattern*** with the ***mediator***.



•The pattern:

- Defines an object that encapsulates how a set of objects interact, named ***Mediator.***
- Promotes loose coupling by keeping objects from referring to each other explicitly and lets you vary their interaction independently.

• *Example:*



<<interface>>
Subject
register (o:Observer)
notify()
<<interface>>
Observer
update ()
notify() {
for all o
in observers
o.update ()
}
update() {
observerState:=
subject.getState()
}
Stock
Customer
<<interface>>
Mediator
register(o:Observer)
notify()
update()
observer
StockBroker
observers

4. Architectural Patterns

Overview

Definition:
An architectural style or pattern is a description of the component and connector types involved in the style, the collection of rules that constrain and relate them, and the advantages and disadvantages of using the style.

Examples of Architectural Patterns

- Distributed
- Event-driven
- Frame-based
- Batch
- **Pipes and filters**
- Repository-centric
- Blackboard
- Interpreter
- Rule-based
- **Layered**
- **MVC (Model-View-Controller)**

Pipes-and-Filters Pattern

Pipes and Filters

Context

Provides a structure for systems that process streams of data.

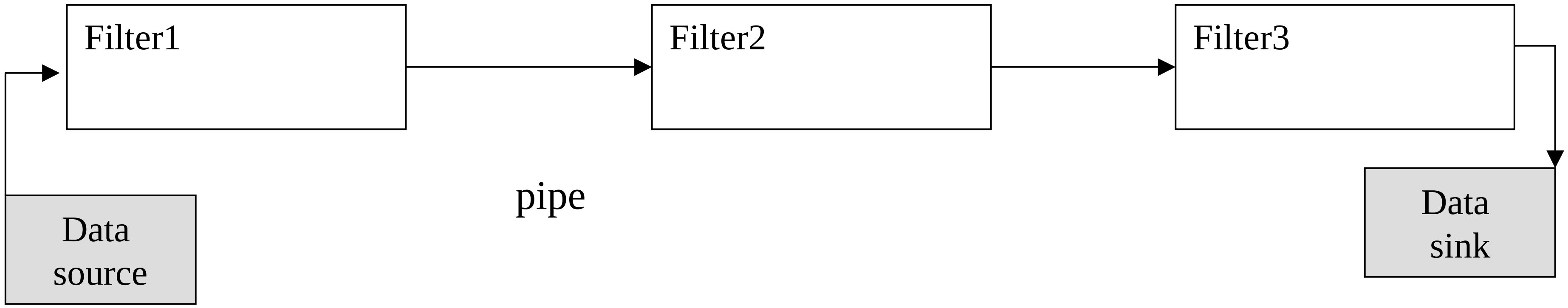
Problem

-Developing a system that processes a stream of data requires the design of several components that correspond to the different processing stages.
-These components may be developed by different developers, and may be available at different period in time.
-Requirements are subject to change; flexibility, changeability, and reusability are key.

Solution

-Tasks are organized into several sequential processing stages:
•***Filters*** implement processing stages.
•***Pipes*** are communication data flow between filters.
•***Data source:*** input to the system (e.g. file, sensor etc.).
•***Data sink:*** output from the system (e.g. file, terminal etc.).

-The combination of filters and pipes forms what is called a processing pipeline.



-Filters are categorized in 2 groups according to their triggering events:

- ***Passive filters:*** output data is ***pulled*** from the filter by the subsequent pipeline or input data is ***pushed*** in the filter by the previous pipeline.
- ***Active filters:*** function as separate threads or processes, and as such ***pull*** their input and ***push*** their output down the pipeline.

- When two active filters are connected either the pipe synchronizes them, or one of the filters control the communication, in which case the pipe can be implemented as a direct call.

- Data sink and source can also be modeled as either passive or active.

Guidelines for Developing Pipes and Filters Architectures

1. Decompose the system functionality into a sequence of processing stages.

2. Define the format of the data flow among the processing units.

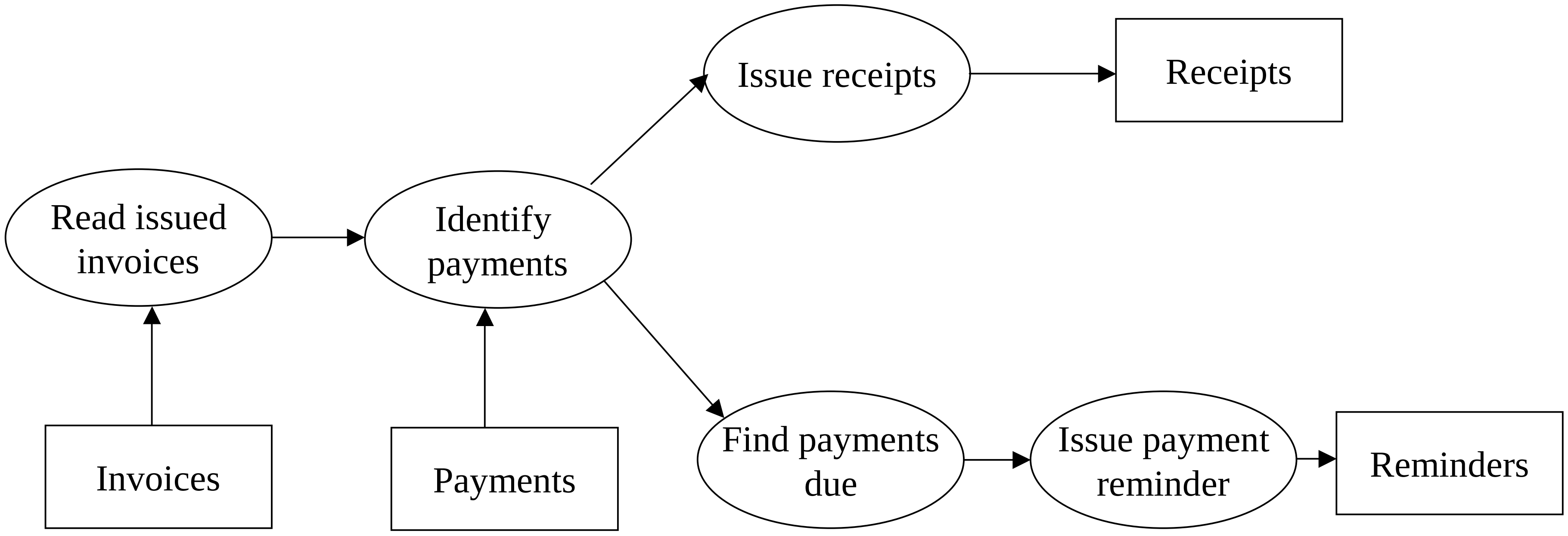
3. Decide about the model of pipes used for interconnection, either as direct call or synchronization pipeline.

4. Design and implement the filters: whether the filters are passive or active will depend on the interconnection mechanism selected previously.

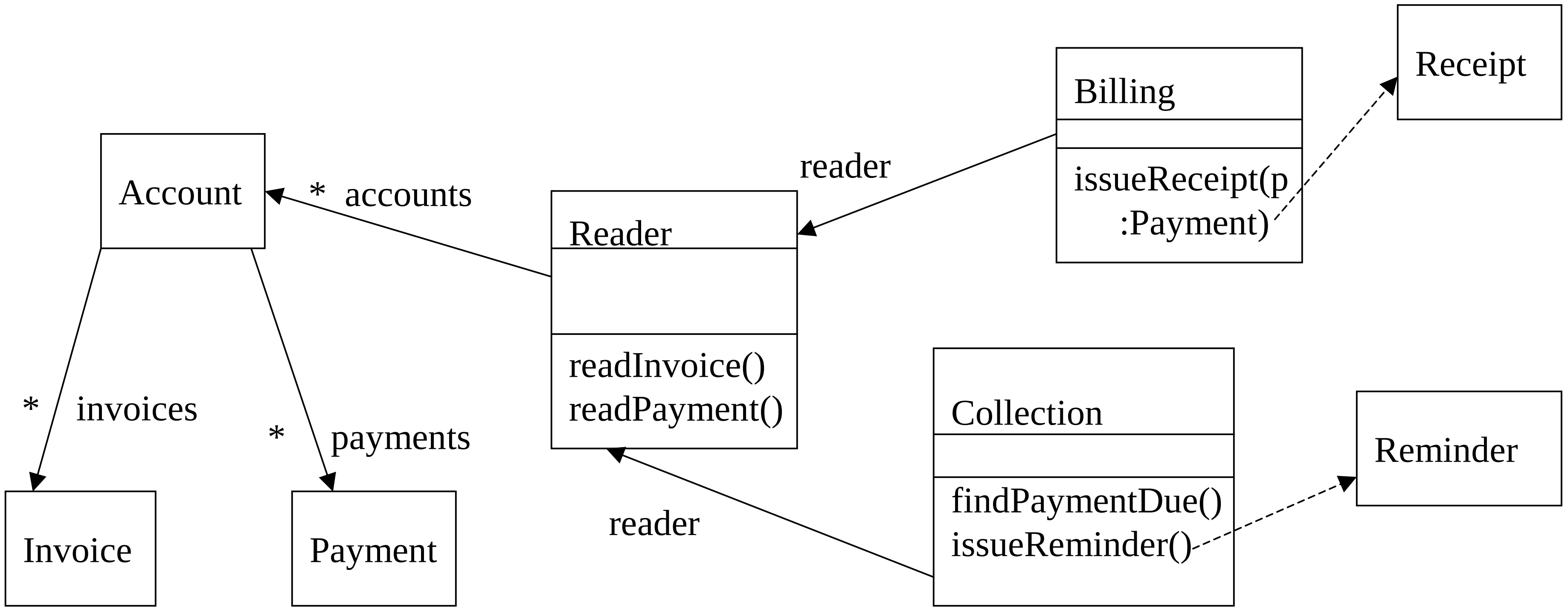
Example: Pipes-and-filters model of an invoice processing system

An invoice processing system checks issued invoices against payments received, and accordingly, issues either receipts or payment reminders for customers.

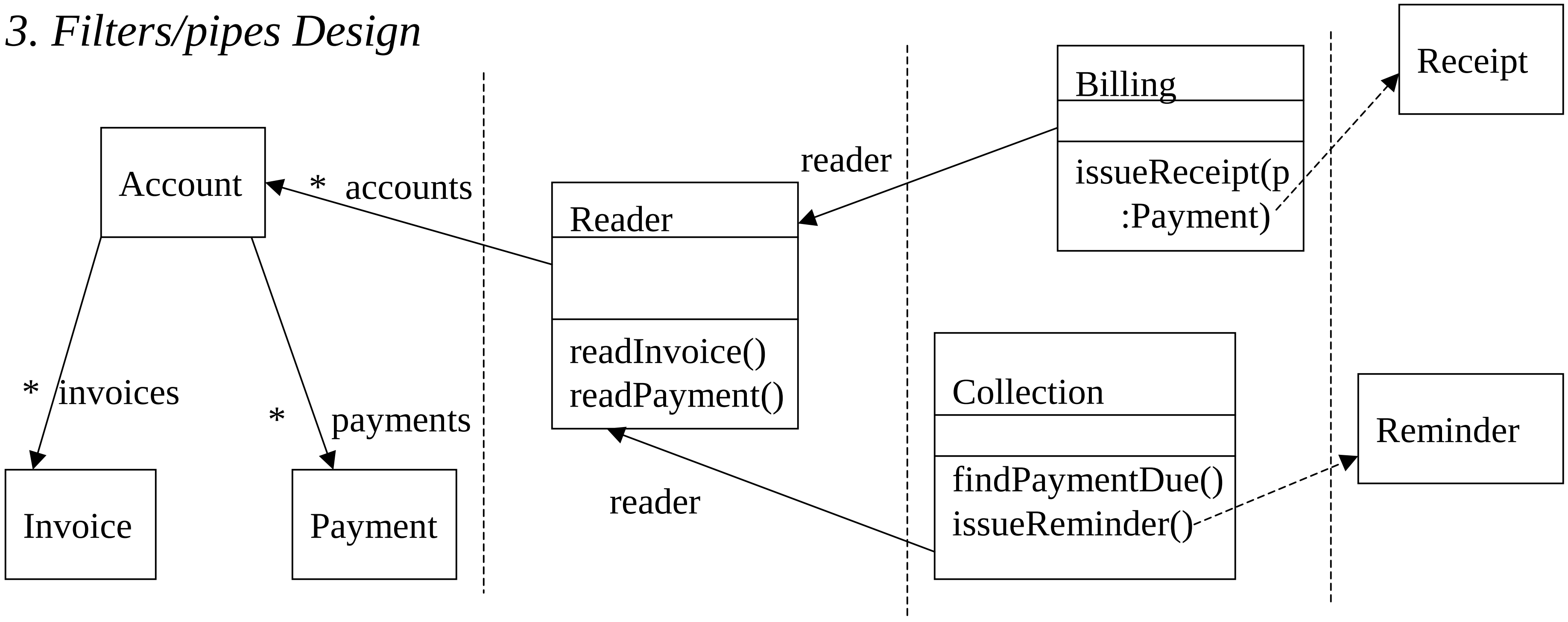
1. Processing Stages/Data Flow

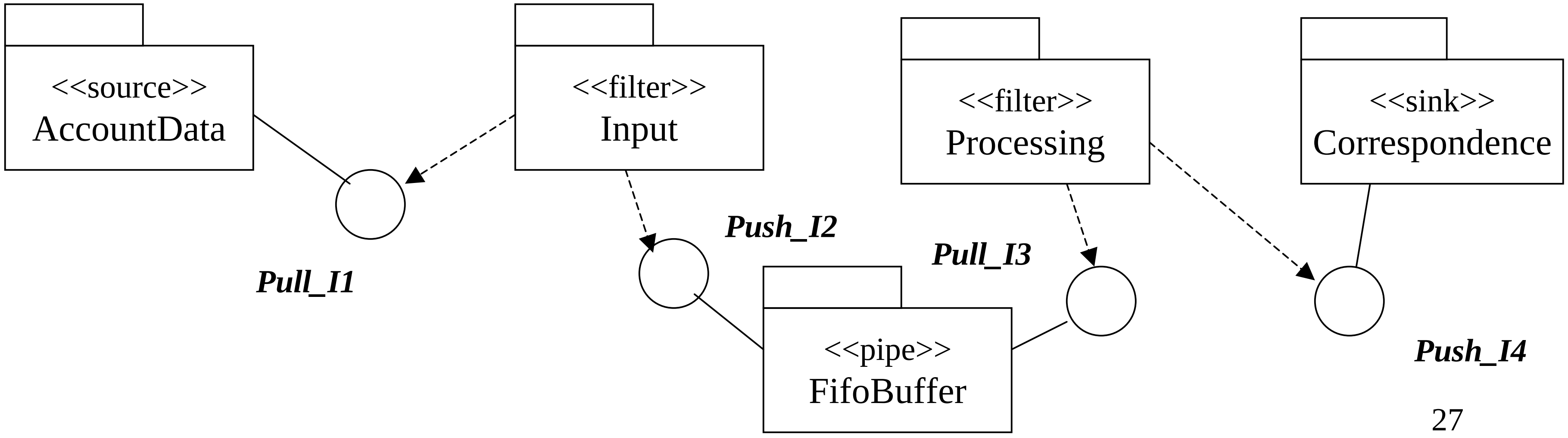


2. Data/Processing Model



3. Filters/pipes Design





Benefits and Drawbacks of Pipes and Filters

Benefits:

- Friendliness
- Reusability
- Evolvability, flexibility, and maintainability
- Concurrency

Drawbacks:

- Poor level of interactivity
- Difficulty to synchronize two related but separate streams of data
- Difficulty of sharing state information and high cost of data transfer

Model-View-Controller Pattern

Model-View-Controller

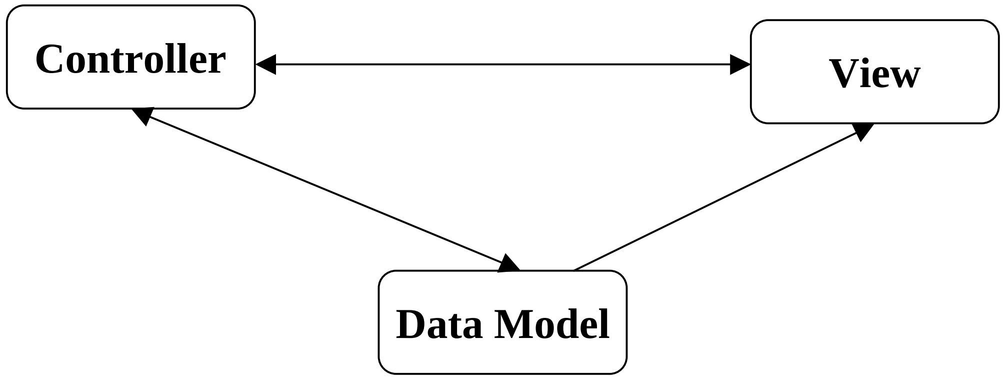
Context
Provides a flexible structure for developing interactive applications.

Problem
-User interfaces are subject to changes. As new features are added to the system, the UI must provide appropriate command and menus to handle them.
-Different platforms support different ‘look and feel’ standards; the UI must be portable.
-Different kind of users may expect different data format from the UI (bar char, spreadsheet etc.).

Solution
-Divide the system into three parts: processing, output, and input:
•***Model:*** contains the processing and the data involved.
•***View:*** presents the output; each view provides specific presentation of the same model.
•***Controller:*** captures user input (events-> mouse clicks, keyboard input etc.). Each view is associated to a controller, which captures user input.

-**Main goal**: facilitate and optimize the implementation of interactive systems, particularly those that use multiple synchronized presentations of shared information.

-**Key idea:** separation between the data and its presentation, which is carried by different objects.



-Controllers typically implement event-handling mechanisms that are executed when corresponding events occur.

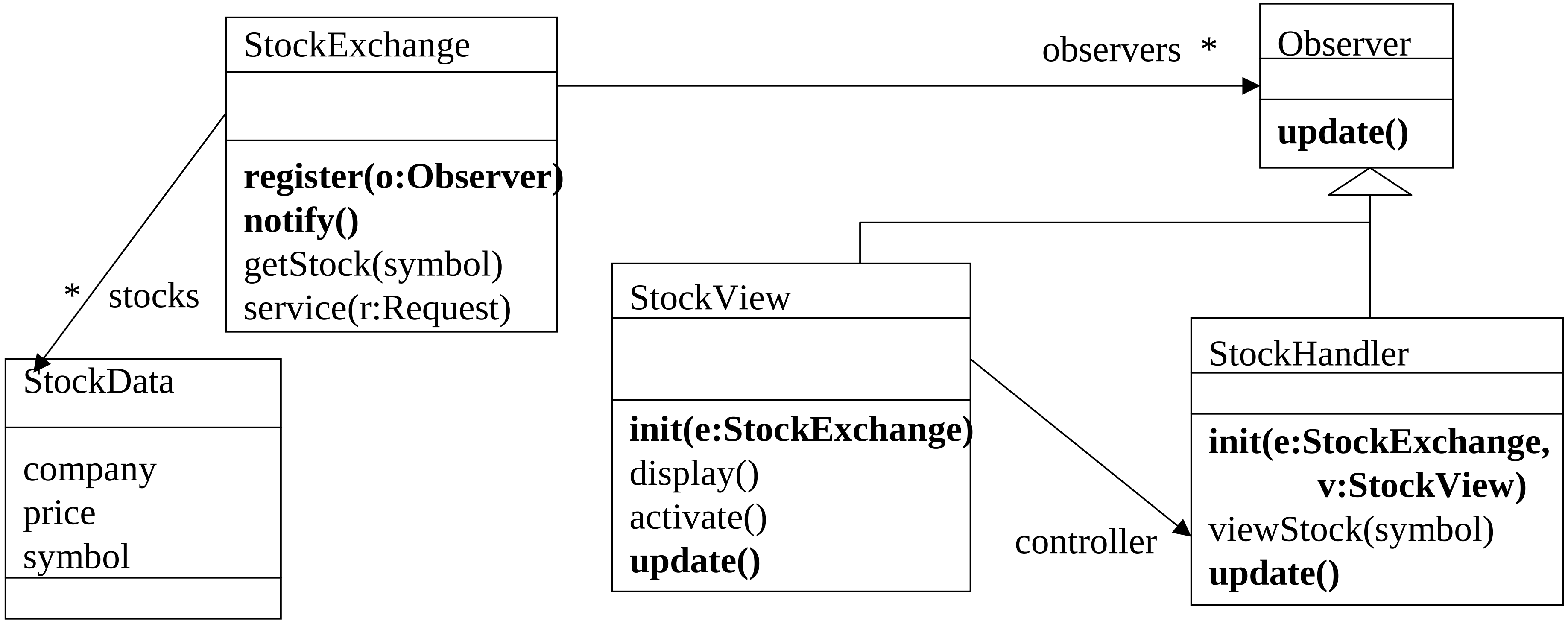
-Changes made to the model by the user via controllers are directly propagated to corresponding views. The change propagation mechanism can be implemented using the ***observer (design) pattern*.**

Example: A Stock Trading Application

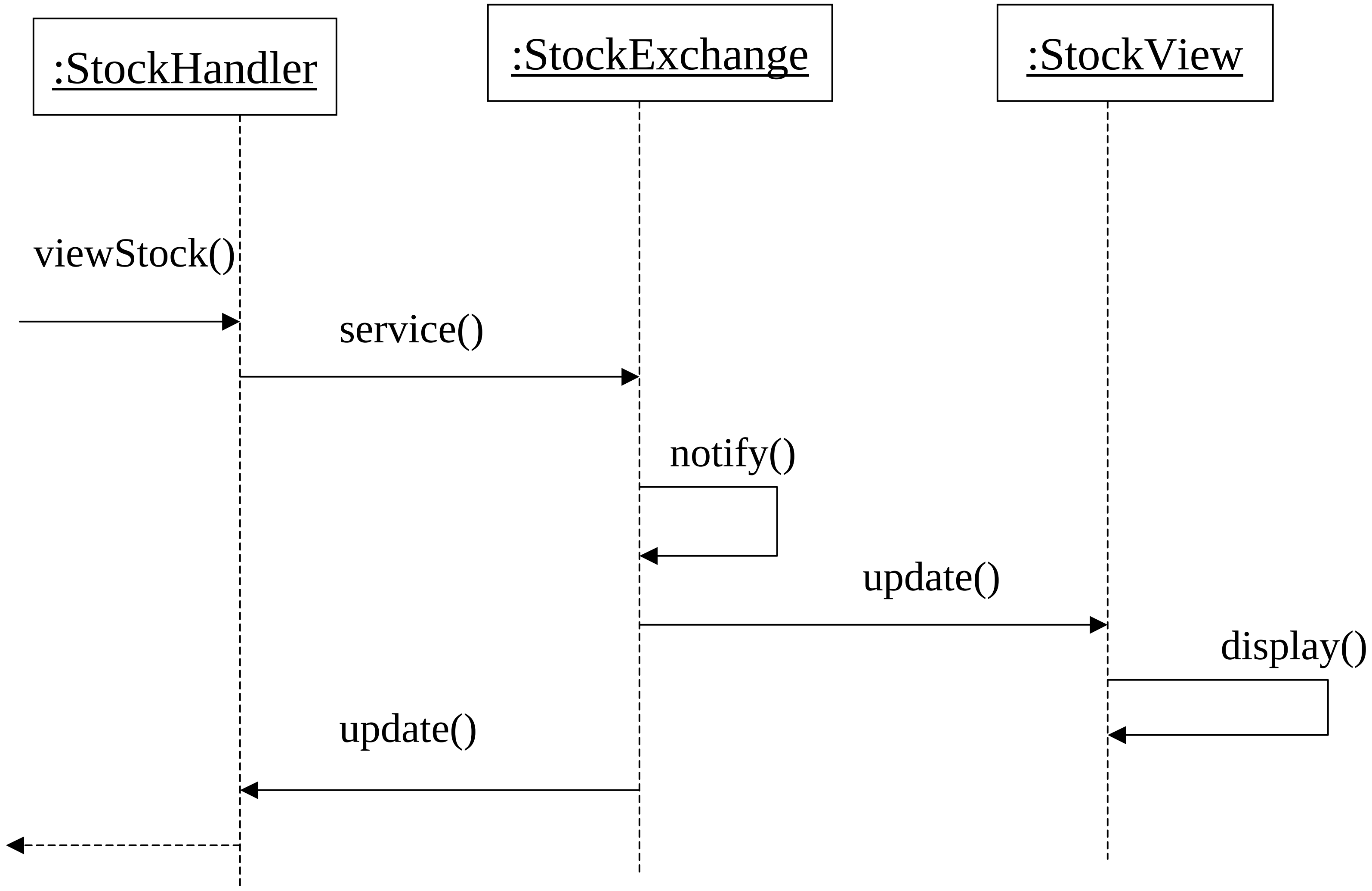
The stock trading application must allow users to buy and sell stocks, watch the activity of stocks. The user is presented with a list of available stocks identified by their stock symbols.

*The user can select a stock and then press a View stock button. That results in a report about the stock: **a spreadsheet or a graph** that shows the stock price over some interval. The report is **automatically updated** as new stock data becomes available.*

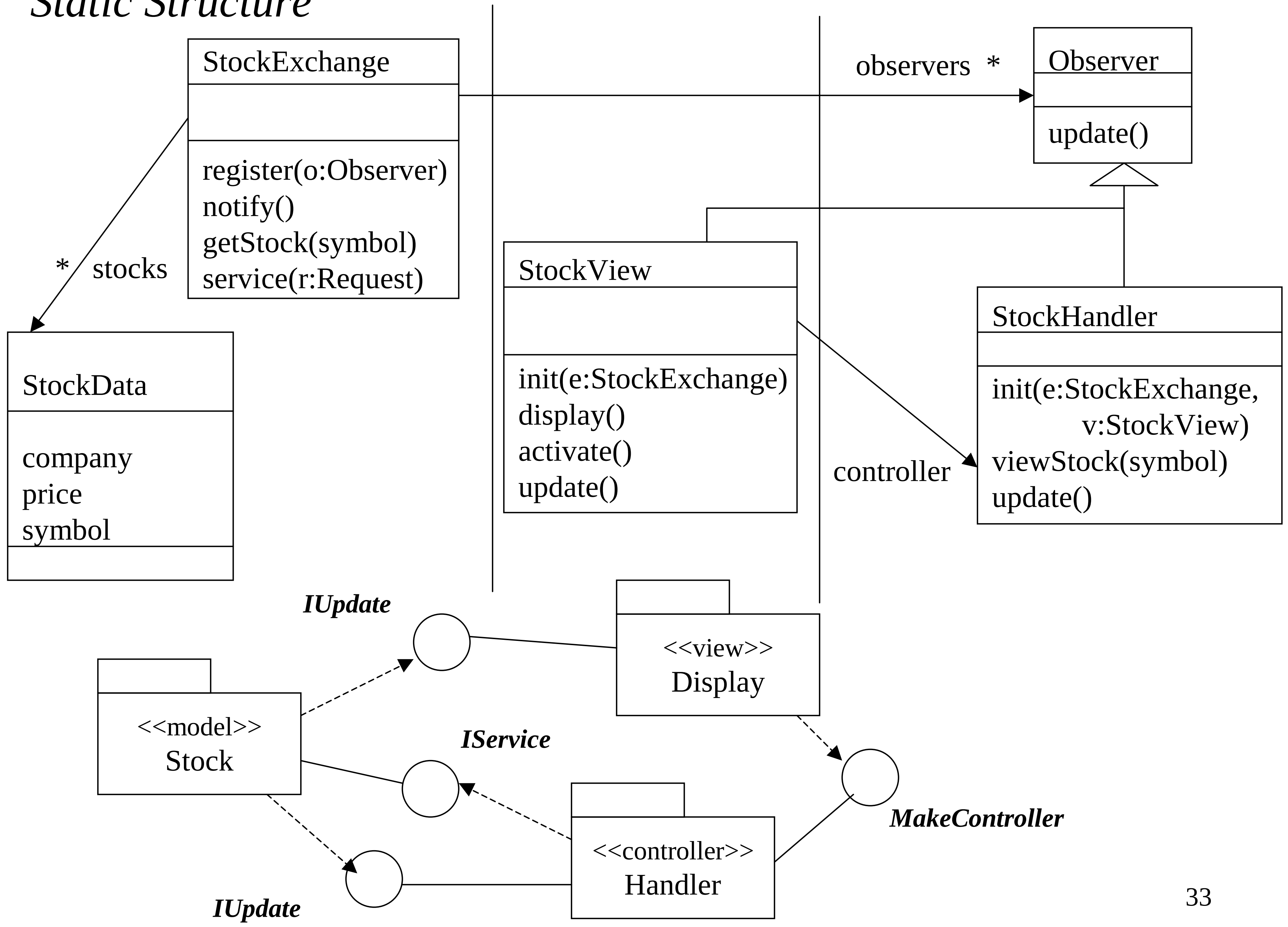
Class Diagram



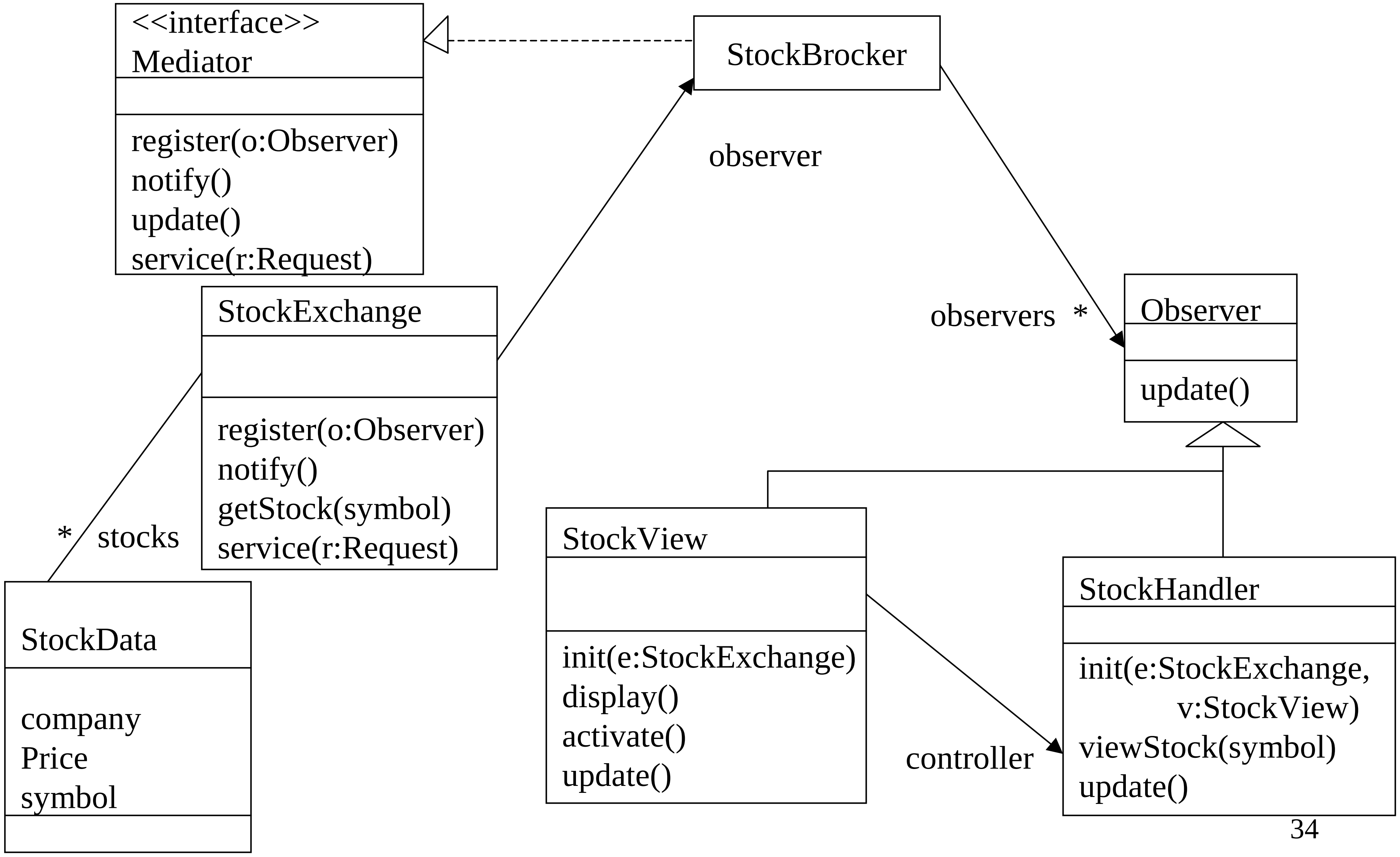
Interaction Diagram

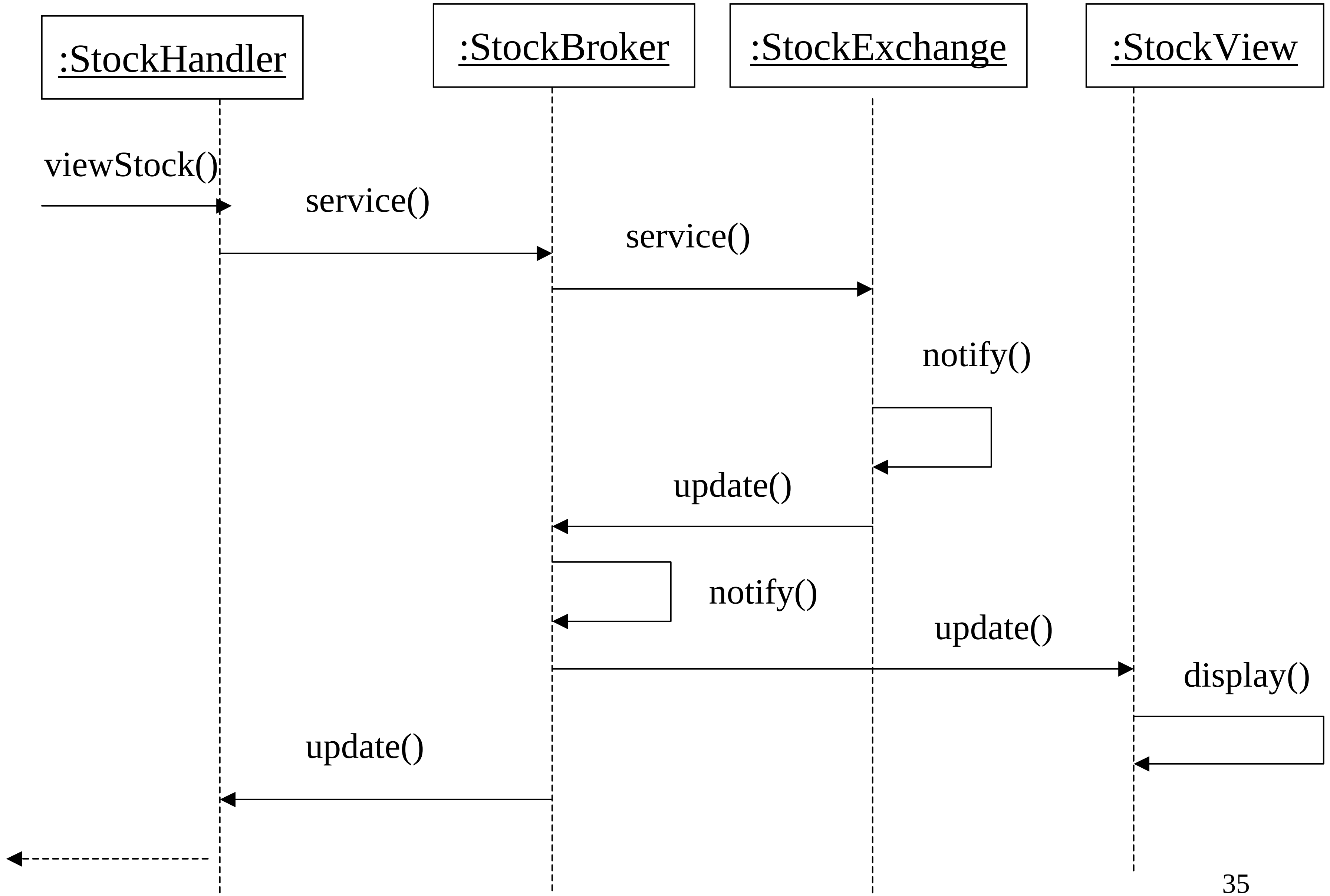


Static Structure



-Assuming that customers may trade through a stock broker, which may trade stocks at several stock exchanges, it is more practical to extend the observer with the mediator pattern.




:StockHandler
:StockBroker
:StockExchange
:StockView
viewStock()
service()
service()
notify()
update()
notify()
update()
display()
update()

Benefits and Drawbacks of the MVC Pattern

Benefits:

-Decouple the underlying computation from the information presentation

-Possibility to associate multiple views to a single data model

-Increased flexibility, and reusability

Drawback:

-Increased complexity

-Intimate connection between view and controller

-Difficulty of using MVC with modern user-interface tools

Layered Pattern

Layers

Context

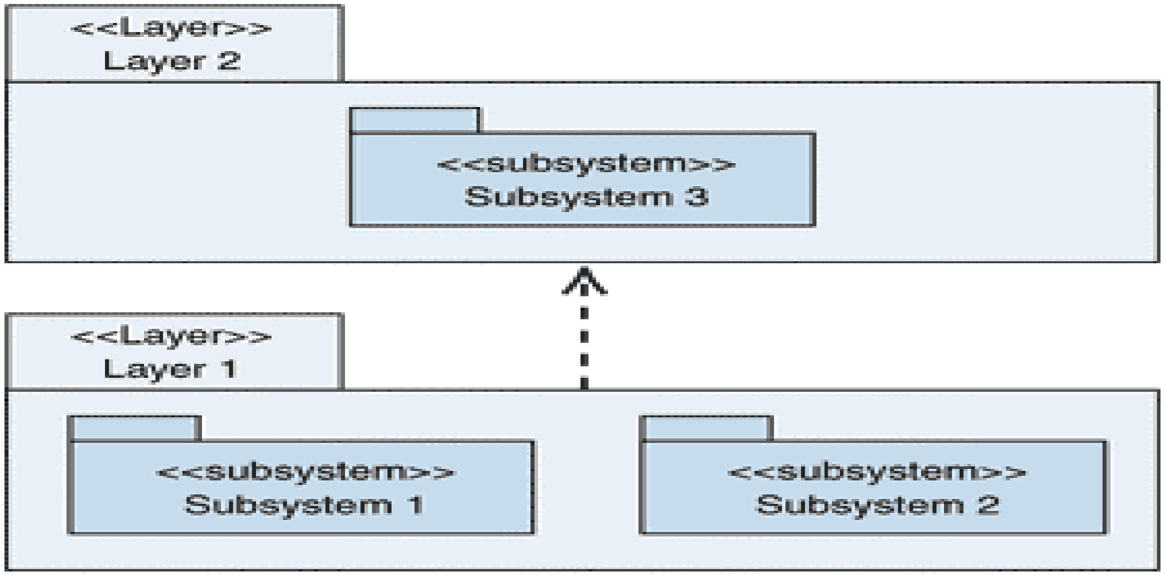
You are working with a large, complex system and you want to manage complexity by decomposition

Problem

How do you structure an application to support such operational requirements as maintainability, reusability, extensibility, scalability, robustness, and security?

Solution

Compose the solution into a set of layers. Each layer should be cohesive and at roughly the same level of abstraction. Each layer should be loosely coupled to the layers underneath ....



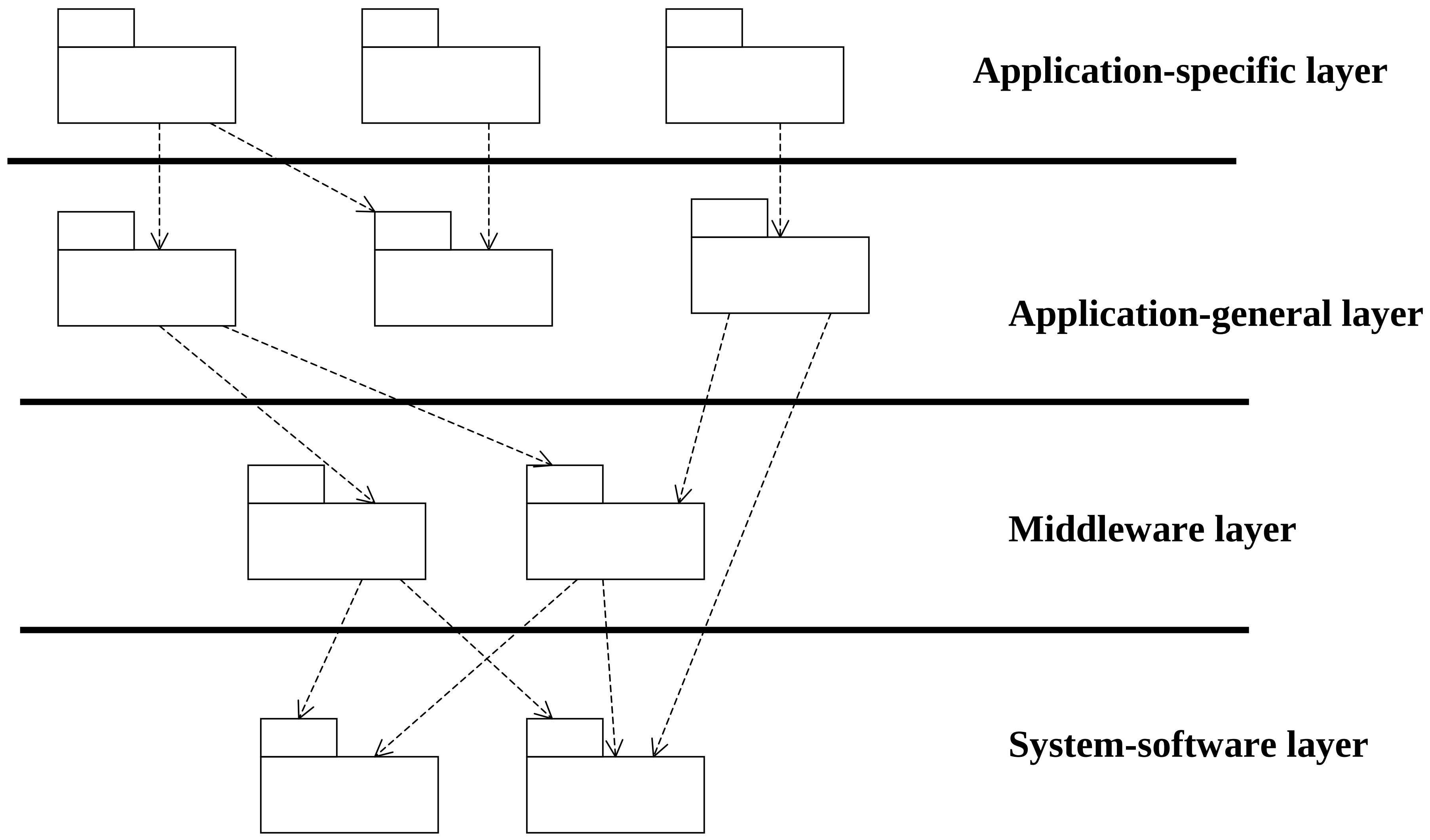
-Layering consists of a ***hierarchy of layers***, each ***providing service to the layer above*** it and ***serving as client to the layer below***.

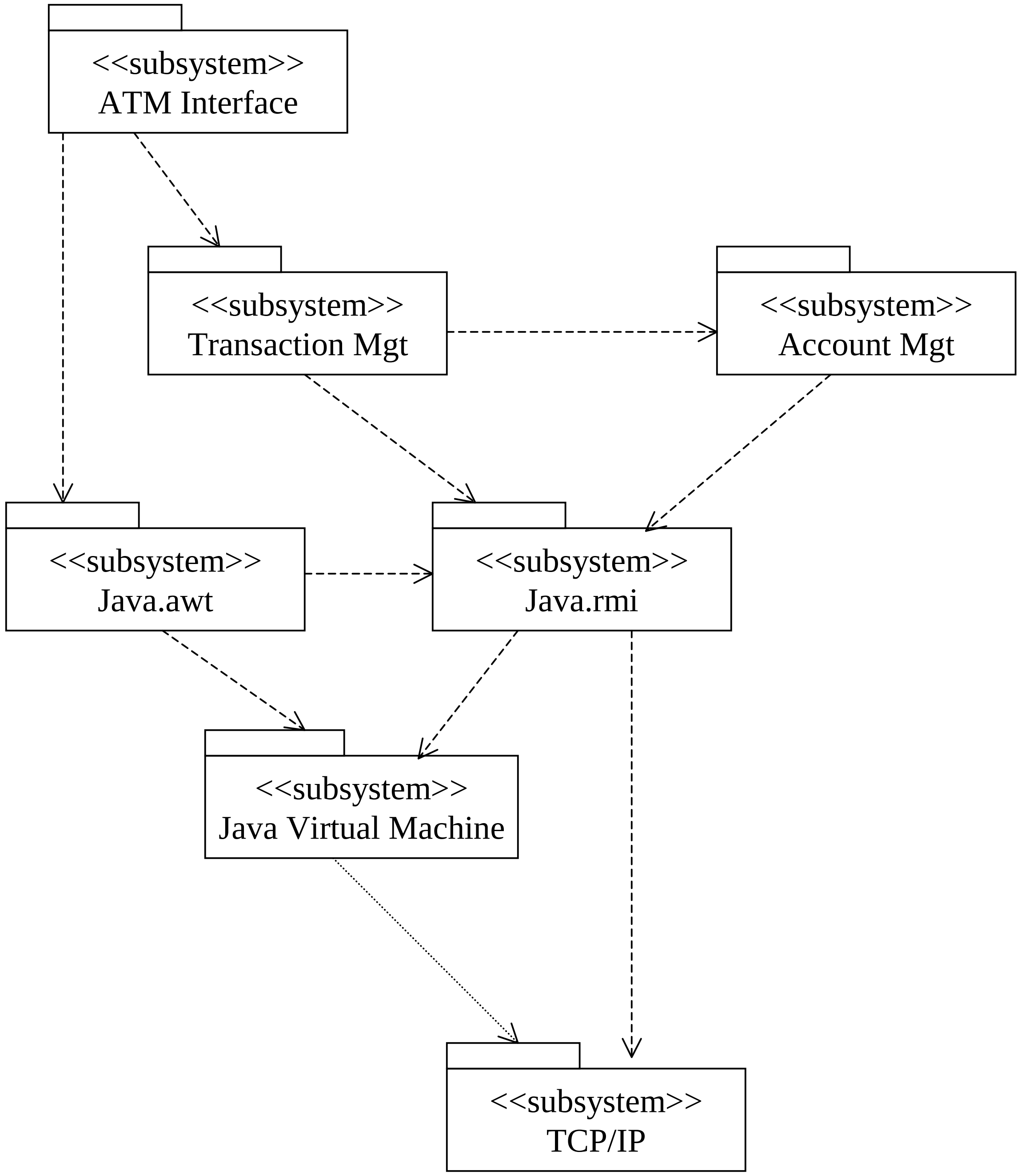
- Interactions among layers are defined by suitable communication protocols. Interactions among non-adjacent layers must be kept to the minimum possible.

-Layering is different from composition

- higher-layers do not encapsulate lower layers
- lower layers do not encapsulate higher layers (even though there is an existence dependency)

Example of Layered Architecture





Application-specific layer

Application-general layer

Middleware layer

System-software layer

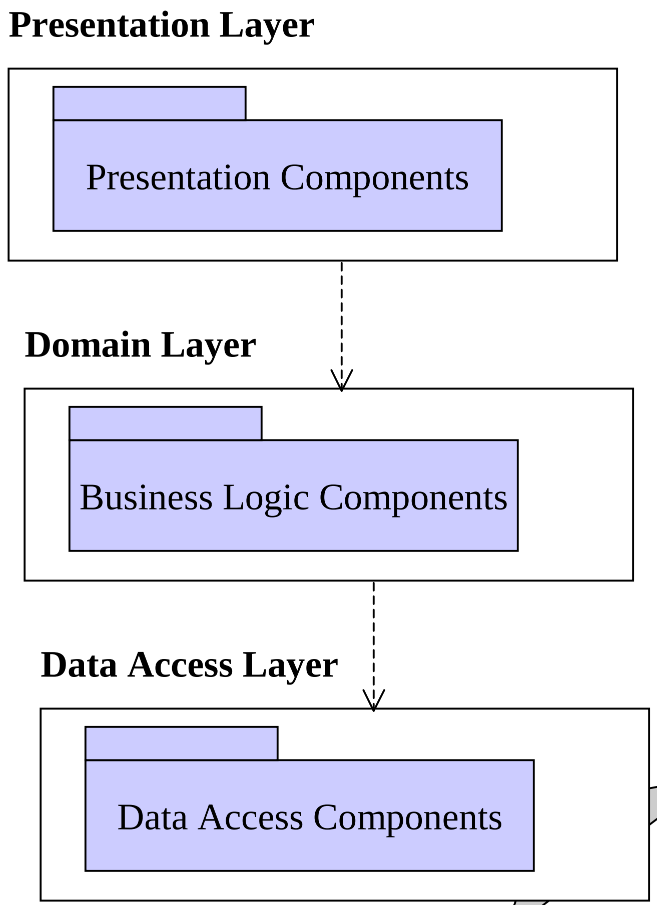
Three-Layered Pattern

Context
You are building a business solution using layers to organize your application.

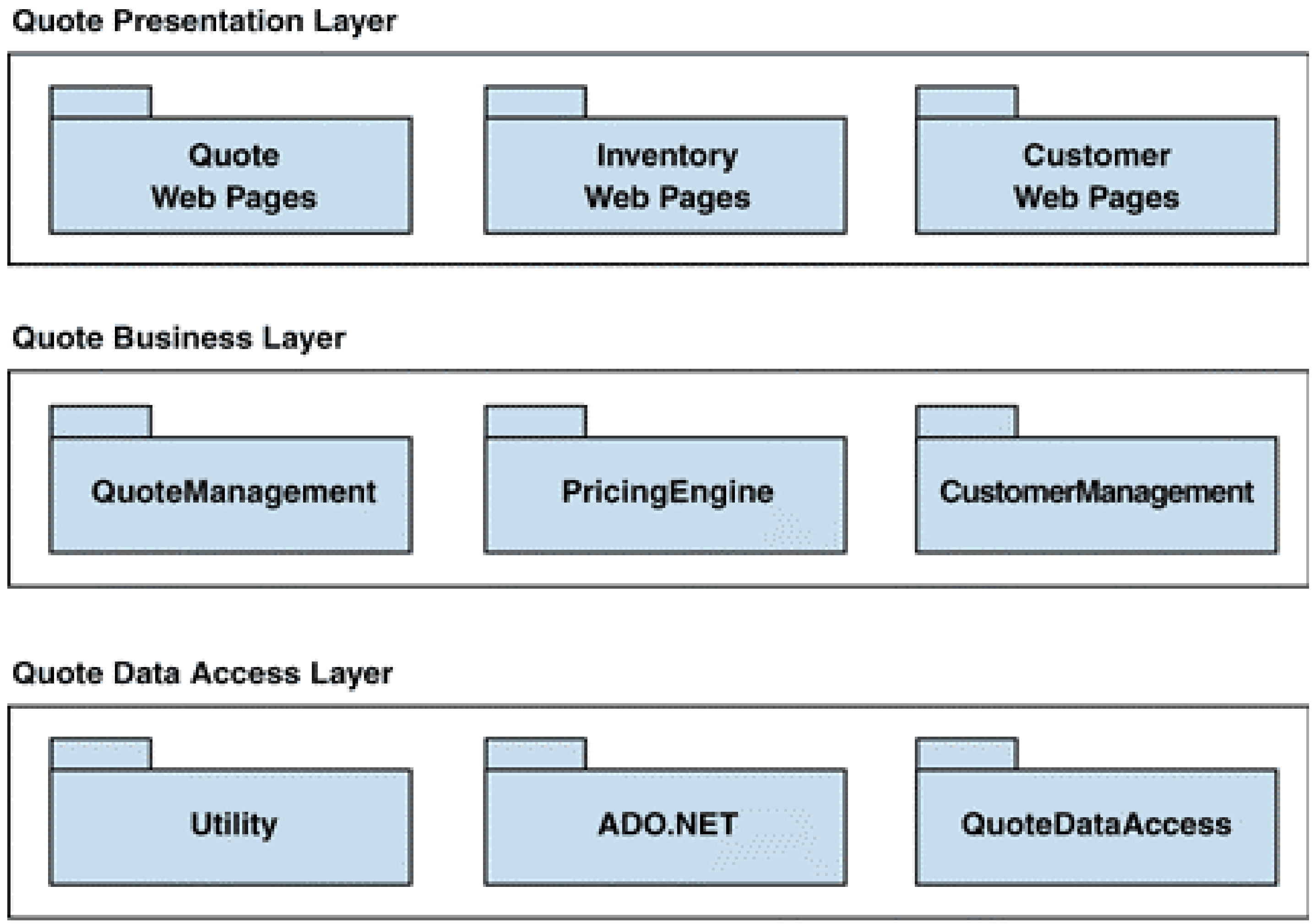
Problem
How do you organize your application to reuse business logic, provide deployment flexibility and conserve valuable resource connections?

Solutions
-Create three layers: ***presentation, business logic*** and ***data access***.

- Locate all database-related code, including database clients access and utility components, in the data access layer.
- Require the data access layer to be responsible for ***connection pooling*** when accessing resources.
- Eliminate dependencies between business layer components and data access components.
- Either eliminate the dependencies between the business layer and the presentation layer or manage them using the ***Observer*** pattern.



Example


Quote Presentation Layer
Quote Web Pages
Inventory Web Pages
Customer Web Pages
Quote Business Layer
QuoteManagement
PricingEngine
CustomerManagement
Quote Data Access Layer
Utility
ADO.NET
QuoteDataAccess

Example: Integrating (Web) Services

Problem: *You built a quote application for a successful enterprise that is rapidly expanding. Now you want to extend the application by exposing your quote engine to business partners and integrating additional partner services (such as shipping) into the quote application. How do you structure your business application to provide and consume service?*

Solution:

•Extend *Three-Layered Application* by adding additional service-related responsibilities to each layer.

-The business layer adds the responsibility for providing a simplified set of operations to client applications through ***Service Interface*s**.

-The responsibilities of the data access layer broaden beyond database and host integration to include communication with other service providers through ***Service Gateway*** components, which are responsible for connecting to services and notifying ***business process*** components of significant service-related events.

Three-Layered Services Application

Context

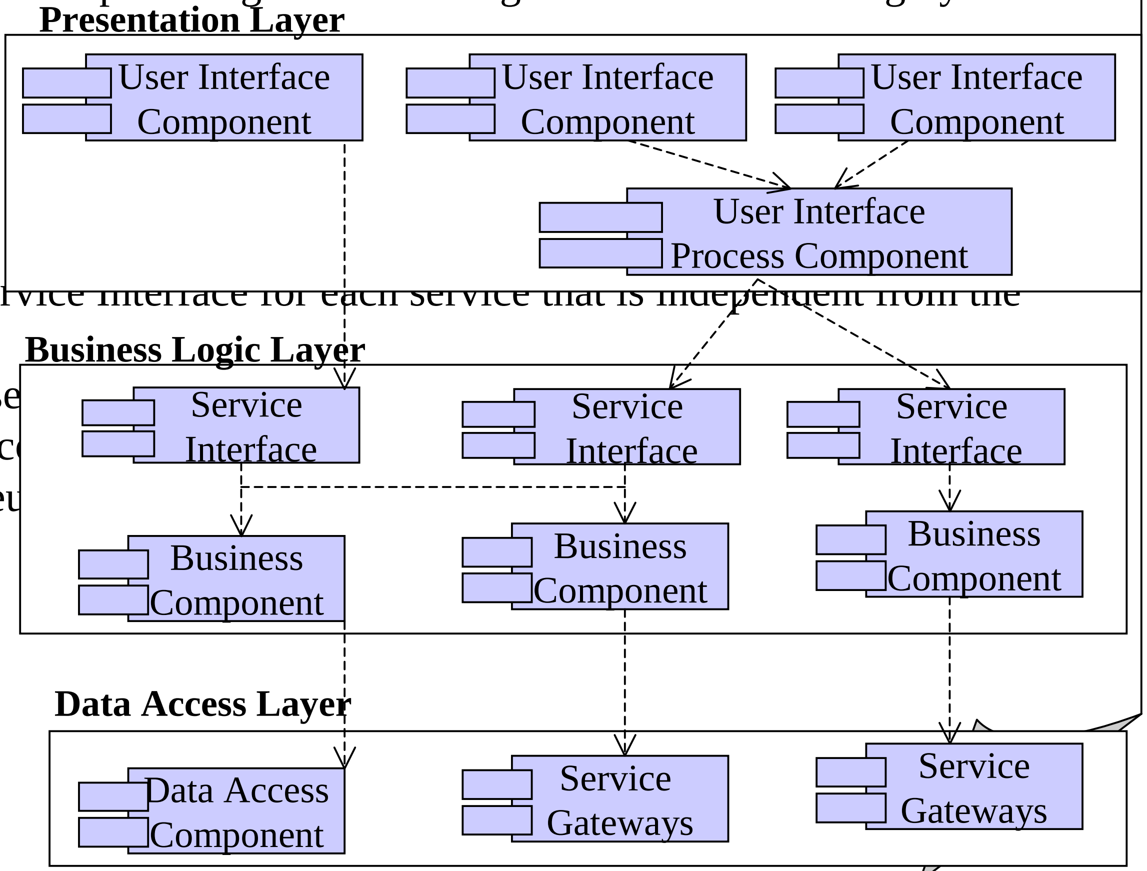
You are building a business solution that uses presentation, business, and data access layers to organize your application. You want to expose some of the core functionality as services that other applications can consume and enable your application to consume other services.

Problem

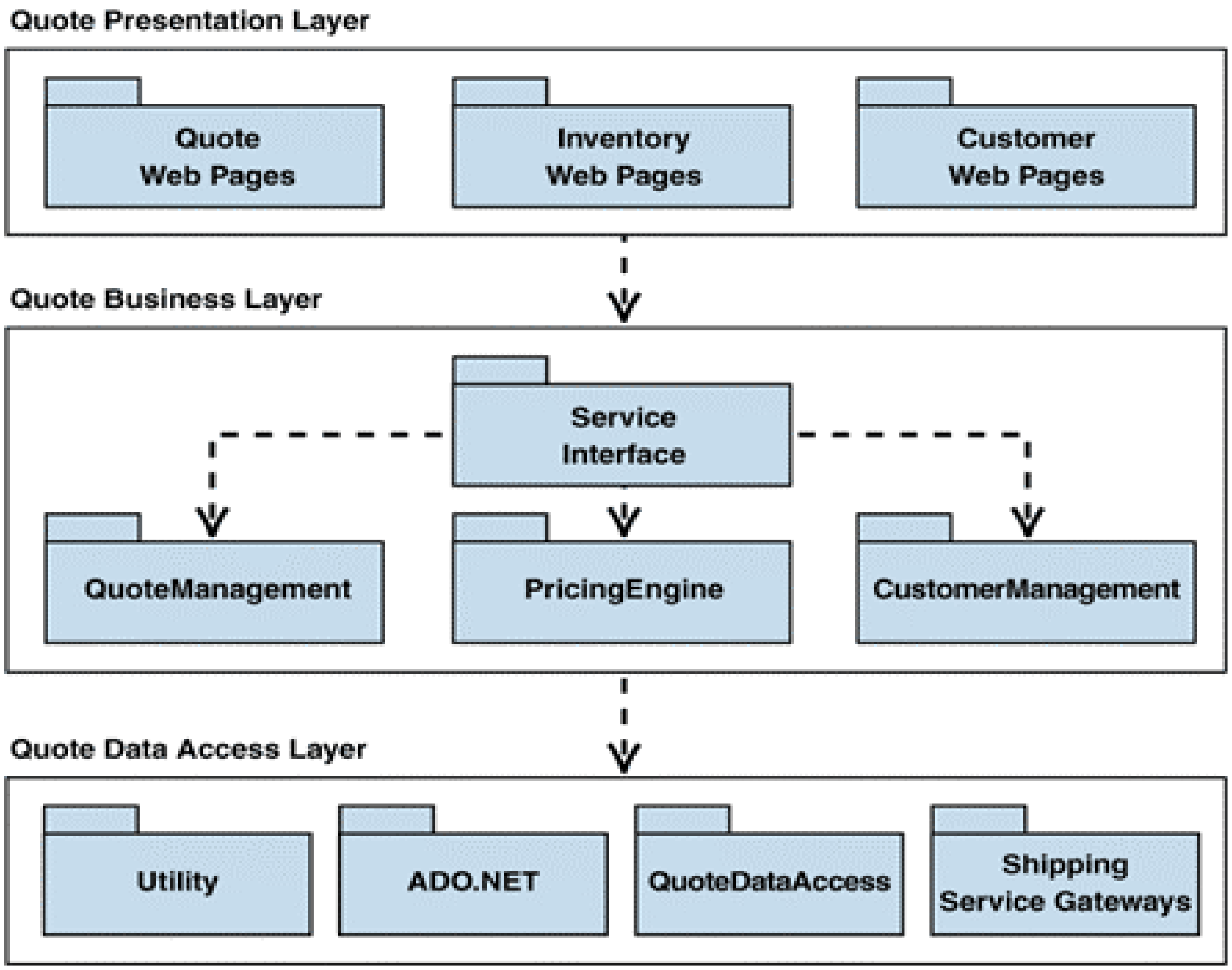
How do you organize your application to provide granular and logical elements from highly variable sources?

Solutions

1. Decompose your application into
 functionality.
2. Identify in the domain layer, a Service Interface for each service that is independent from the
 underlying implementation.
3. Extend the data access layer to use
4. If application navigation logic is c
 components to encapsulate and reu



Example


Quote Presentation Layer
Quote Web Pages
Inventory Web Pages
Customer Web Pages
Quote Business Layer
Service Interface
QuoteManagement
PricingEngine
CustomerManagement
Quote Data Access Layer
Utility
ADO.NET
QuoteDataAccess
Shipping Service Gateways

Benefits and Drawbacks of Layering

Benefits:

- Suitable for complex and evolutionary problems
- Communicability, increased reusability due to the decoupling and strong independence between the different functionality
- Extensibility and flexibility

Drawbacks:

- Difficulty of structuring some systems in a layered manner, especially when some functions require crossing several layers
- Performance problems when high-level functions require close coupling to low level functions

4. Appendix: Other Patterns

4A. Appendix: Examples of Design Patterns

☞**Creational patterns**

- ***Factory Method pattern*:** provides a decision-making class that returns selectively one of several possible subclasses of an abstract base class.

- ***Abstract factory pattern*:** provides an interface to create and return one of several families of related objects.

- ***Builder pattern*:** separates the construction of a complex object from its representation so that different representations can be created according to the needs of the program.

☞ **Structural Patterns**

-Describes how classes and objects can be combined to form larger structures.

- ***Composite patterns*:** creates a composition of objects
- ***Proxy pattern*:** creates a simple object that takes the place of a more complex object which may be invoked later
- ***Façade pattern*:** provides a uniform interface to a complex subsystem.

☞ **Behavioral Patterns**

-Are concerned specifically with communication between objects.

- ***Chain of responsibility pattern*:** allows decoupling between objects by passing a request from one object to the other until it is recognized
- ***Command pattern*:** use simple objects to represent the execution of software commands
- ***Observer pattern*:** defines how several objects can be notified of a change.

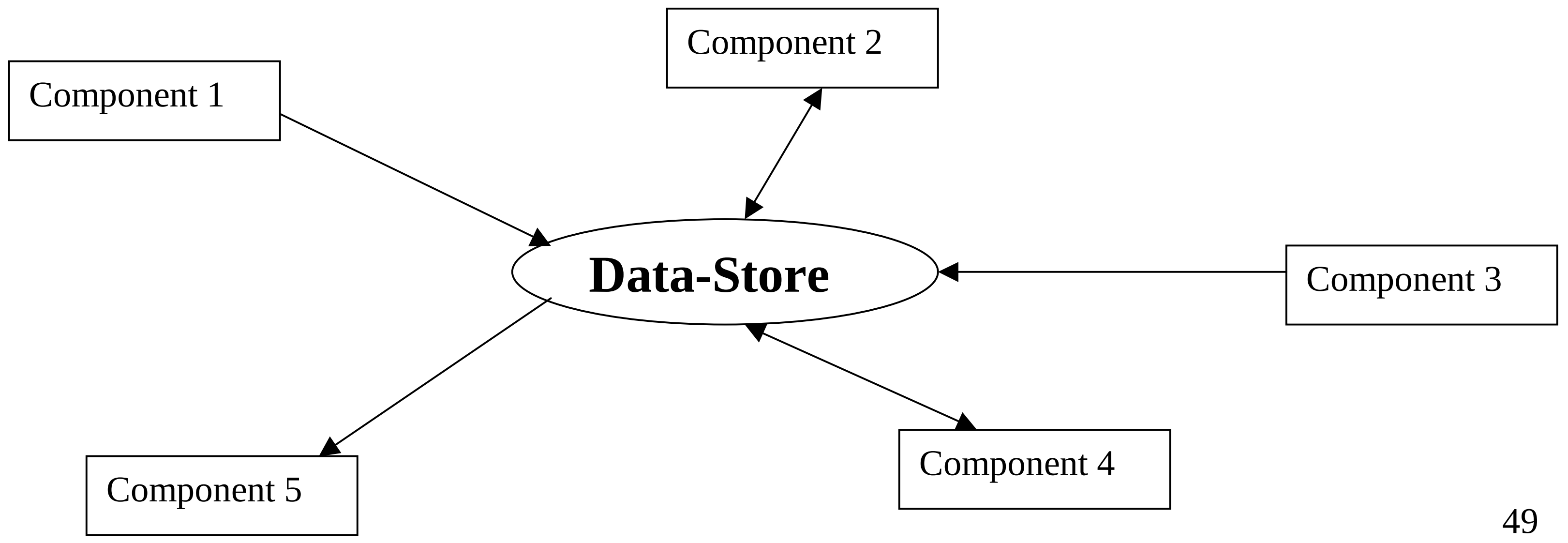
4B. Appendix: Other Architectural Patterns

Repository Pattern

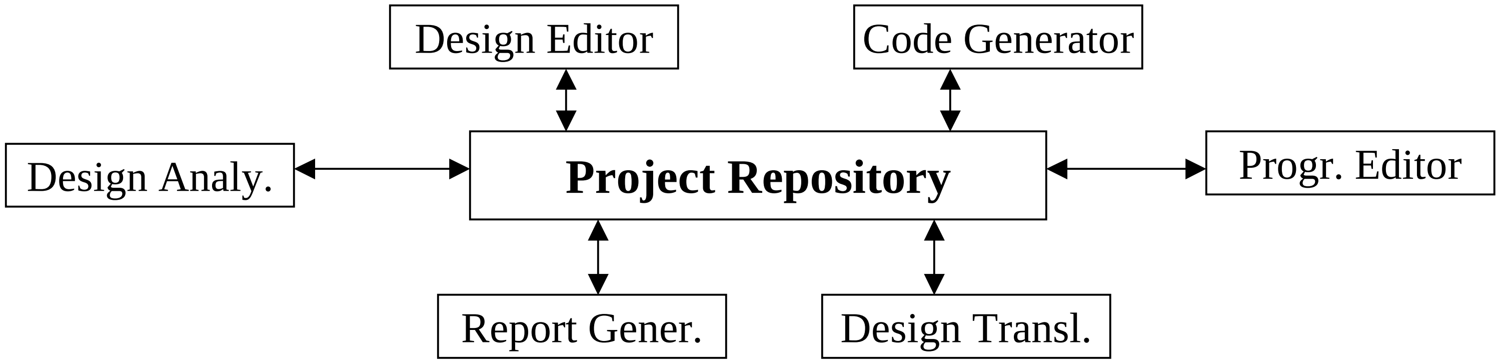
-Used for applications in which sub-systems exchange large amounts of data during their activities.

-Provides a shared repository that can be used by the sub-systems to exchange their data.

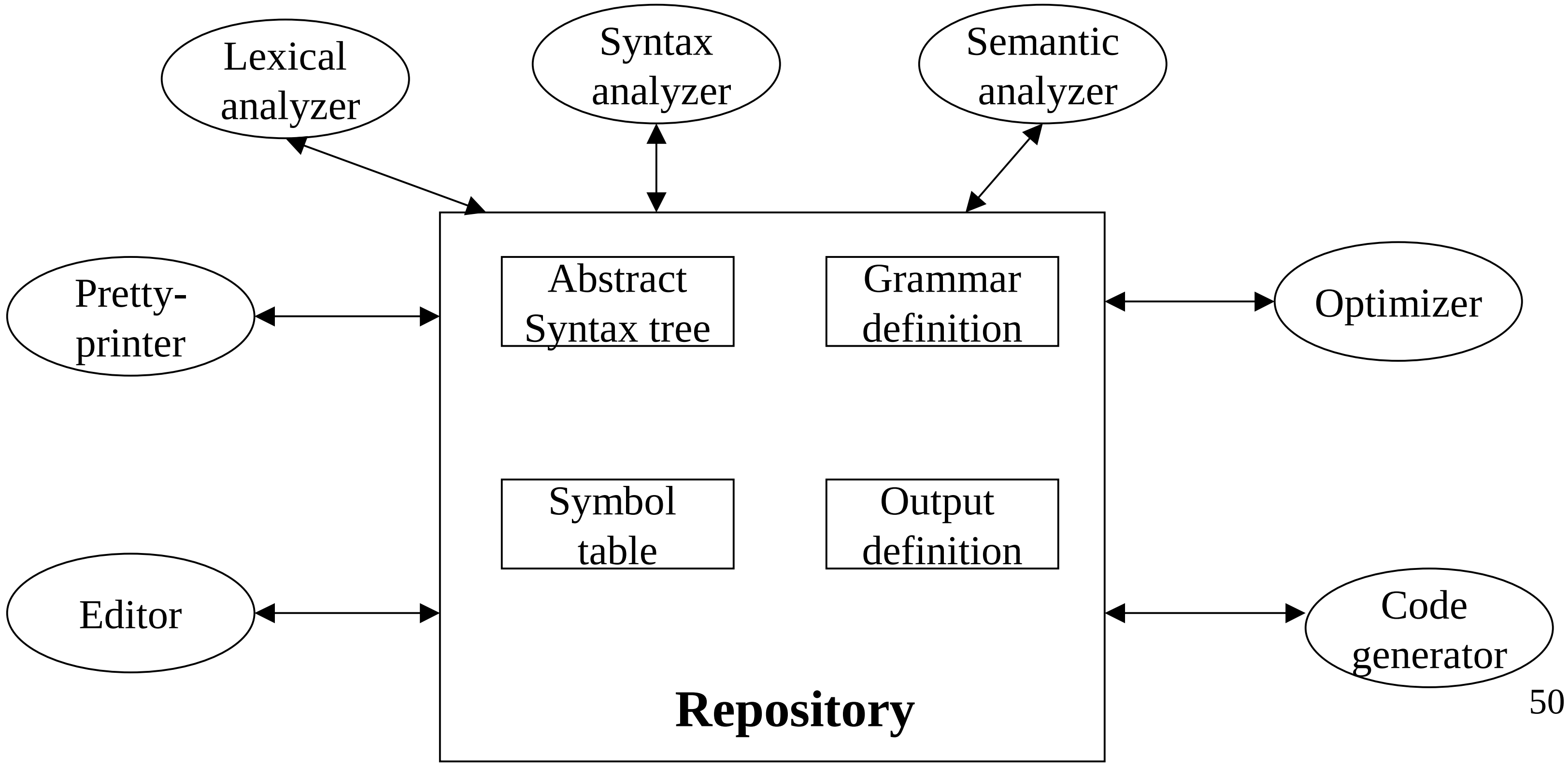
-Two kinds of components: a central data-store carrying the system current state, and a set of independent components that perform operations on the data carried by the central data-store.



Example 1: Repository Model of a CASE Tool



Example 2: Repository Model of a Language Processing System



Benefits and Drawbacks of the Repository Pattern

Benefits:

- Efficient for sharing large amount of data

Drawbacks:

- Performance may be seriously affected
- Distribution of the repository may be quite complex
- The data compatibility mechanisms selected may also impact the extensibility of the overall system

Example Uses:
-Large IDEs
-Language processors
-Operating systems

Blackboard Pattern

-Variant of the repository style used for systems involving complex problem solving (e.g. signal processing, programming environments)

-Provides a high-level organization and control of the knowledge needed for complex problem solving.

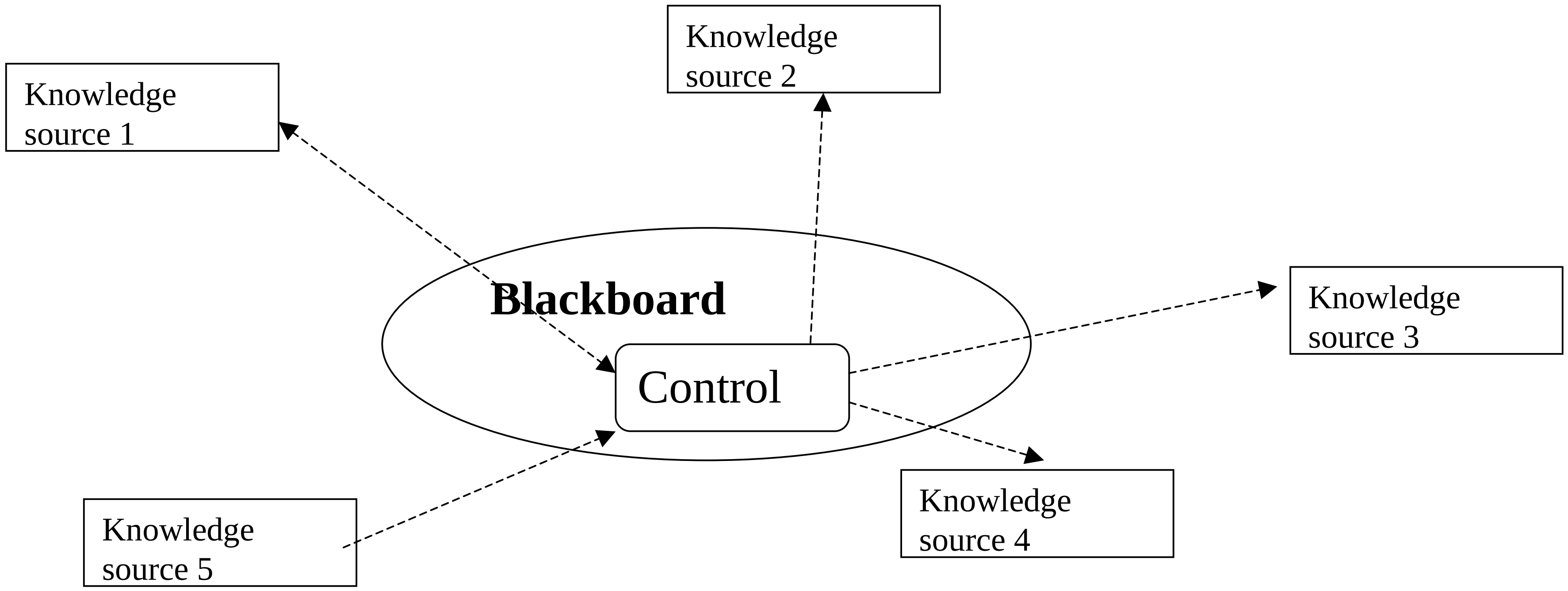
Context: An immature domain in which no closed approach to a solution is known or feasible.

Known Uses:
-Speech recognition
-Enemy detection
-Object monitoring
-Expert systems

A collection of independent programs that work cooperatively on a common data structure.
"Opportunistic problem solving"

-Three kinds of components involved:

- ***Knowledge sources*:** pieces of application specific knowledge (e.g. data, algorithms and procedures needed to solve the problem).
- ***The blackboard and its data structure*:** describe the state of the solution in a global data store.
- ***The control component*:** triggered by changes in the blackboard; decides and activates the most suitable knowledge sources.



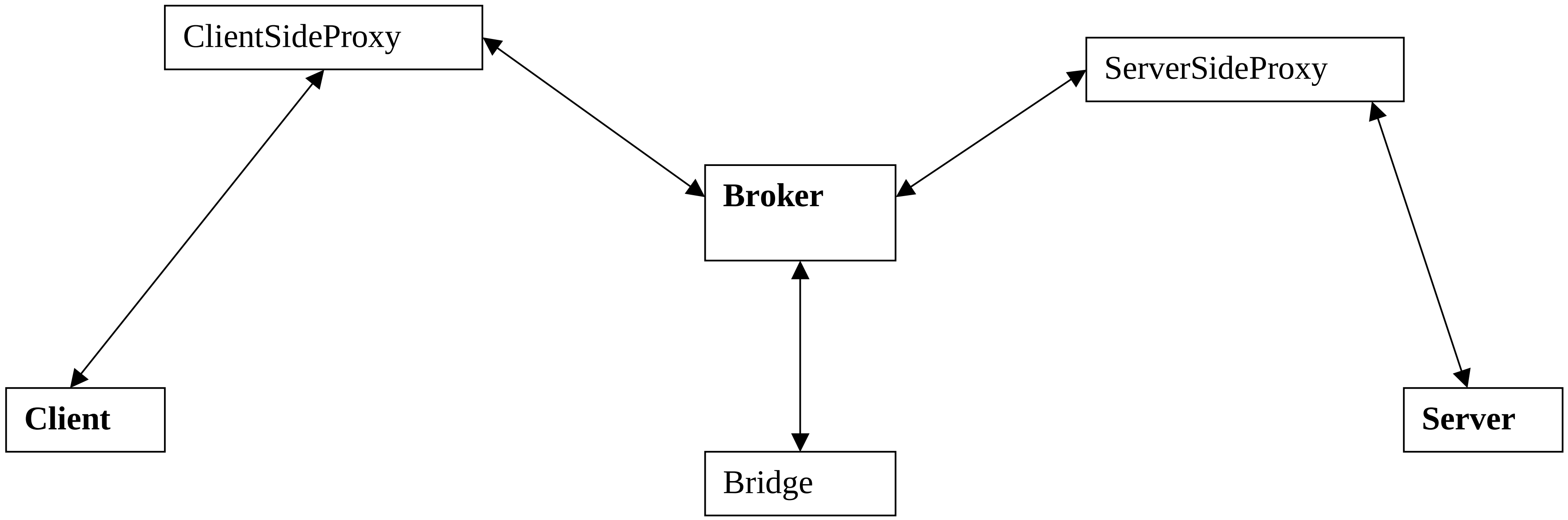
***Benefits*:** flexibility and extensibility, and its support for concurrency.

Broker Pattern

-Organizes and coordinates distributed and possibly heterogeneous software components interacting remotely.

-Three kinds of components:

- ***Broker:*** encapsulates interactions between clients and servers.
- ***Servers:*** provide services through the broker.
- ***Clients:*** access servers' services through the broker.



-Three kinds of intermediary components involved:

- ***Client-side proxy:*** mediates between the client and broker.
- ***Server-side proxy:*** mediates between the server and broker.
- ***Bridge:*** connects local broker with remote brokers via the network.

Benefits and Drawbacks of the Broker Pattern

Benefits:

- Transparencies (e.g. location, time, replication, distribution etc.),
- Low coupling among components, reusability, extensibility, portability, and interoperability.

Drawbacks:

- Lower level of fault tolerance and reduced performance.

Context: Your environment is a distributed and possibly heterogeneous system with independent cooperating components.

Known Uses:
-CORBA
-OLE
-WWW

Sponsor-Selector Pattern

-Intended for dynamic changing applications, in which decision about using specific set of resources is made transparently at run-time.

-Approach: provide a mechanism for resource selection when it is needed. A clear separation is made between the decision about the utility of the resource, the selection of the resource, and the use of the resource.

-Three kinds of components:

- ***Selector:*** decides which resource to prefer under particular conditions.
- ***Sponsor:*** associated to a particular resource and determines when its resource may be used.
- ***Resource:*** represent specific data or functionality provided in the context of the system.

Benefits: reusability, extensibility and flexibility.

Drawback: potential for tight coupling between the client and the selector in one hand, and the selector and the sponsors in the other hand in case the selection procedures involve large context-specific knowledge.

Generalized Framework for Access Control (GFAC)

-Dedicated for systems in which there are important access control requirements.

-Separates the access control procedure in two parts:

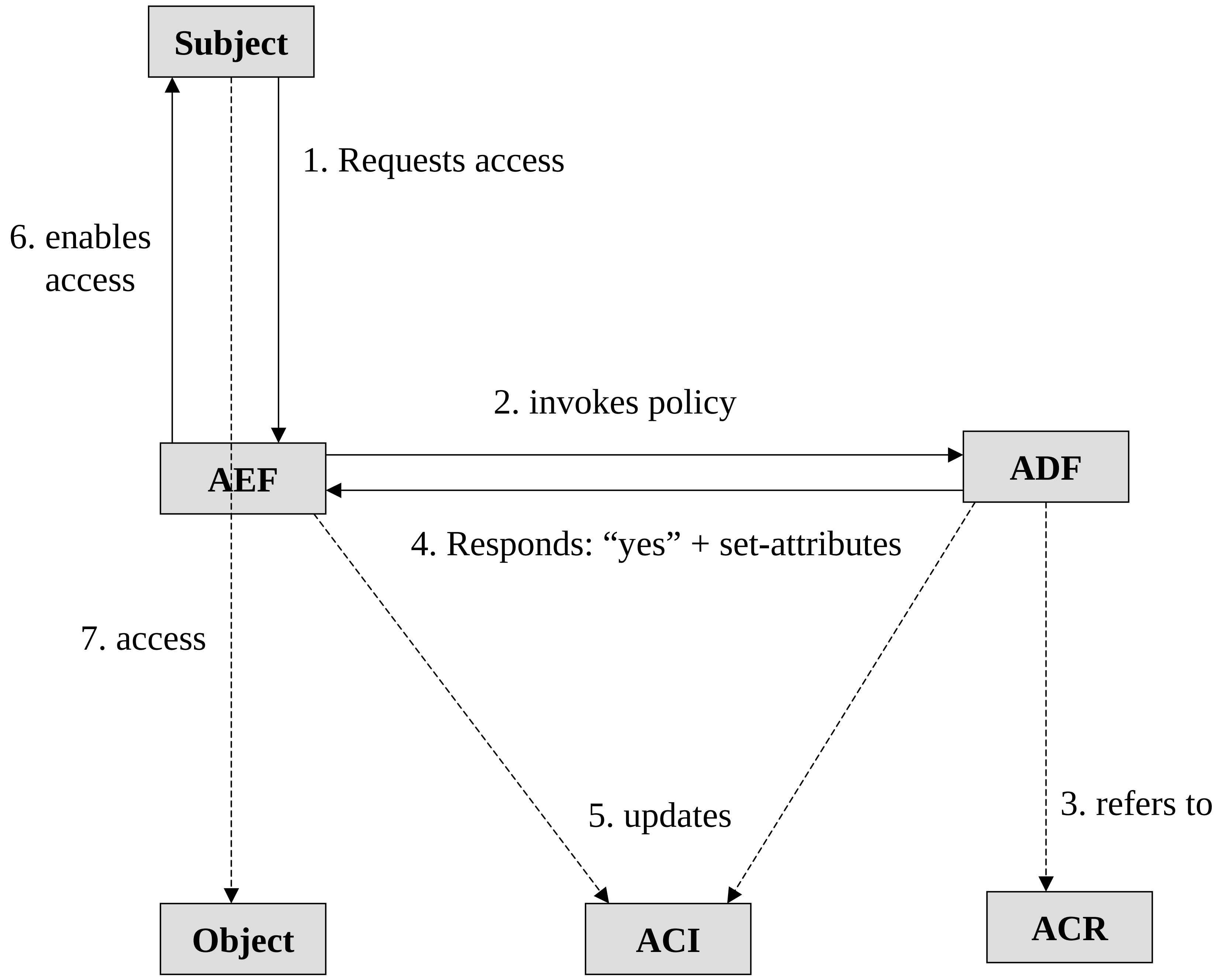
- *Adjudication*: performed by the *Access Decision facility (ADF)*
- *Enforcement:* made by the *Access Enforcement Facility (AEF)*

-Other components involved in the framework are the following:

- *Subject*
- *Object*
- *Access Control Information (ACI):* information used by the ADF to make access control decisions.
- *Access Control Rules (ACR):* embodies access control rules.

Modus Operandi

In practice when a subject wants to access an object, it sends a request to the *AEF*. The *AEF* then forwards the request to the *ADF* with additional information (related to the subject) encapsulated in the *ACI*. Basing itself on the rules defined in the *ACR*, the *ADF* makes decision about whether access shall be granted or not, and notifies the *AEF* about that decision. Then accordingly the *AEF* will or will not grant access to the resource.


Subject
1. Requests access
6. enables access
2. invokes policy
AEF
ADF
4. Responds: “yes” + set-attributes
7. access
5. updates
3. refers to
Object
ACI
ACR