LIYANA ARACHCHIGE RANIL

SHORT NOTES / DESIGN PATTERNS

1. There are two major types of design patterns
    a. CORE J2EE PATTERN
    b. GANG OF FOUR PATTERN (FOG Pattern)

CORE J2EE PATTERNS

1. A Pattern is an idea that has been useful in one PRACTICLE CONTEXT and will probably be useful in others
2. PRESENTAION TIER DESIGN CONSIDERATION
    a. SESSION MANAGEMENT
        i. Session state can be saved on the client / SESSION STATE ON THE CLIENT
        Saving session state on the client involves SERIALIZING and EMBEDDING the session state within the VIEW MARKUP HTML page that is returned to the client
            1. Relatively easy to implement
            2. works well when saving minimal data
            3. eliminate the problem of replicating state across severs where load balancing is done
            4. There are two strategies that can be used to save state on the client , HTML HIDDEN fields and HTTP COOKIES
            5. HTML HIDDEN FIELDS are only good for small amount of data , if the data is larger then performance is severely effected

6. HTML HIDDED FIELDS can only save STRING value , hence any object referenced in the state would be STRINGIFIED
7. HTTP COOKIES are also only good for small data sizes
8. For HTTP COOKIES there are SIZE LIMITATIONS
9. Another strategy is EMBEDDING the state to the URL
10. When saving session state on the CLIENT, SECURITY issues are introduced. Hence it is required to ENCRYPT the state

ii. SESSION STATE ON THE PRESENTATION LAYER
1. When session state is maintained on the SERVER , it is retrieved using a SESSION ID
2. It is clearly preferable to save the SESSION state of an application which contains lot of SESSION state related data on the SERVER

iii. SESSION STATE ON THE BUSINESS TIRE / RESOURCE TIER
1. EJB can be used to HOLD session state in the BUSINESS TIER
2. RELATIONAL DB can be used at the RESOURCE TIER

b. CONTROLLING CLIENT ACCESS

i. Guard a view or portion of view from direct access by client
1. For this you can EMBED a GUARD within the view itself. It can be done in such a way that All-Or-Nothing , or only portion of the view

            2. GUARD by CONFIGURATION. For this you can leverage the security built in to web container
        ii. Controlling the flow of the USER through the application (Stopping DUPLICATE form submissions)
            1. Synchronizer Token (Déjà vu)
        iii. VALIDATION
            1. Validation on the CLIENT
                a. Client side validation is always a COMPLIMENT to server side validation
            2. Validation on the SERVER
        iv.
    c.

3. PRESENTATION TIER BAD PRACTICES
    a. Control Code in Multiple VIEWS
        i. Consolidate the control code INTRODUCING CONTROLLERS and associated HELPERS – Front Controller / View Helper
    b. Exposing presentation tier data structures (HTTP SERVLET REQUEST) to Business Tier
        i. Copy Presentation tier data structure's data to another object such as CONTEXT OBJECT and use this to pass value between layers or pass those values as parameters
    c. Exposing Presentation tier data structures to Domain Objects
        i. Domain Object are reusable objects and Presentation tier data structures MUST not be used in side them which makes tight coupling between those two
    d. Allowing Duplication Form Submissions
        i. Introduce Synchronization Token

e. Exposing Sensitive Resources to Direct Client Access
f. Creating FAT controllers
    i. If TOO MUCH code is added to the controller it becomes cumbersome to test , debug and controller is heavy weight
    ii. Controller is the INITIAL CONTACT point as well as it should be a DELEGATE POINT
    iii. Use COMMAND OBJECT to encapsulate different control logic, so that CONTRLLER will be light weight and individual COMMAND objects can be TESTED relatively in ISOLATION
g. Using HELPERS as SCRIPLETS
    i. HELPERS must only expose HIGHER LEVEL ABSTRACTIONS

4. BUSINESS TIER DESIGN CONSIDERATION
    a. USING SESSION BEANS
        i. Stateless versus Statefull
            1. In case of stateless , the state has to be passed from the client or retrieved from the persistence store to stateless session bean
        ii. Storing State on the Business Tier
            1. If the architecture is solely based on Web-based application then maintaining state on the WEB TIER makes sense
            2. If the APPLICATION supports different client including web clients , java applications , other enterprise beans etc then conversational state can be maintained in the EJB layer using STATEFULL beans

- b. USING ENTITY BEANS
  - i. Entity beans are best suited as CORSE GRAINED business objects
  - ii. Entity bean PRIMARY KEYS (simple key or Composite key)
  - iii. Business Logic in Entity Beans
    1. Entity bean should contain business logic that is self contained to manage ITS OWN DATA and DEPENDENT OBJECTS data. It is required to take out the business logic that requires Entity Bean to Entity Bean interaction and put that in a SESSION BEAN. Composite Entity and Session Façade come in to play in this case
  - iv. Caching Enterprise Bean remote reference and Handles

5. BUSINESS AND INTEGRATION TIER BAD PRACTICES
   - a. Mapping the Object model directly to the Entity Bean Model
     - i. Results in LARGE number of FINE-GRAINED entity beans
     - ii. The container and NETWORK OVERHEAD is increased
     - iii. Identify the PARENT DEPENDENT OBJECT RELATIONSHIPS in the OBJECT MODEL and DESIGN them as COARSE-GRAINED ENTITY BEANS
     - iv. This results in FEWER ENTITY BEANS
   - b. Mapping the relational model directly to the Entity Bean Model
     - i. It is a BAD practice to design each row in a table as an ENTITY BEAN
   - c. Mapping each use case to a SESSION BEAN

    i. This creates FINE GRAINED controller responsible to service a ONE TYPE OF INTERRACTION
    ii. This would SIGNIFICANTLY increase the COMPLEXITY of the application
    iii. Apply SESSION FAÇADE pattern to AGGREGATE a GROUP of the related interactions in to a SINGLE SESSION BEAN
    iv. This results in a FEWER session beans and leverage the advantage of applying SESSION FAÇADE

d. Exposing all enterprise bean Attributes via GETTER/SETTER methods
    i. This forces CLIENT to invoke NUMEROUS FINE-GRAINED REMOTE invocations and creates the potential of introducing a significant NETWORK OVERHEAD or CHATTINESS
    ii. Use a VALUE OBJECT/TRANSFER OBJECT to transfer AGGREGATE DATA to and from the client instead of EXPOSING GETTERS and SETTERS

e. Embedding SERVICE LOOKUPS in CLIENT CODE
    i. Any changes to the LOOKUP code effects all the clients , hence very bad
    ii. Clients are exposed to the complexity of the underline implementations and introduce dependency on the look up code
    iii. Encapsulate implementation details of the look up mechanism using a SERVICE LOCATOR. Encapsulate the implementation details of business tier components such as Session, Entity Beans in a BUSINESS DELEGATE.

   iv. This simplifies the client code since there is no longer dependency on Enterprise Beans and Services
   v. Business Delegate CAN intern use the SERVICE LOCATOR

f. Using Entity Beans as READ ONLY objects
   i. Using and Entity Bean as a READ ONLY object simply WASTES EXPENSIVE resources and results in UNNECESSARY UPDATE transactions to the PERSISTANCE store
   ii. This is due to the invocation of ejbStore() methods by the container during entity beans life cycle. Container does not have a way to know whether the data was changed during the invocation of the method , hence it assumes data is changed and invokes ejbStore() on the bean
   iii. Encapsulate all access to Data Source using DAO pattern. This provides a CENTRALISED layer of data access code and also SIMPLIFIES ENTITY BEAN CODE
   iv. Implements access to READ ONLY functionality using a SESSION BEAN typically as a SESSION FAÇADE which uses DAO
   v. Use VALUE LIST HANDLER to obtain LIST OF TRANSFER OBJECTS
   vi. Use TRANSFER OBJECT ASSEMBLER to obtain COMPLEX DATA MODEL from business tier

g. Using ENITY BEANS as FINE GRAINED OBJECTS
   i. Use composite entity to come up with a COARSE GRAINED object

h. Storing ENTIRE ENTITY BEAN dependent object graph

   i. When complex tree structure of dependent objects are used in an entity bean , performance can degrade rapidly when loading an storing an entire tree of dependent objects
   ii. Identify the dependent objects that have changed and store only those (Composite Entity and Store Optimization Strategy[Dirty marker])
   iii. Implement LAZY LOADING strategy

i. Exposing EJB related exceptions to NON-EJB clients
   i. Decouple the CLIENT from the Business tier and hide the business tier implementation details from the client using BUSINESS DELEGATE
   ii.

j. Using Entity Bean Finder methods to return a LARGE RESULT SET
   i. Using an EJB FINDER method would result in a LARGE collection of REMOTE REFERENCES
   ii. Consequently the client would have to invoke methods on these remote references to get data. This will become very expensive network call
   iii. Implements Queries using SESSION BEANS and DAOs to obtain a list of TRANSFER OBJECTS. Use DAO patterns to SEARCH instead of EJB finder methods. User VALUE LIST HANDLER to have list of TRANSFER OBJECTS

k. Clients aggregate DATA from business components
   i. The application clients typically need the DATA model for the application from the BUSINESS TIER. Since the model is implemented as an Entity , Session , and arbitrary Objects in the business tier

the client MUST locate , interact with and extract the necessary data from various business components to construct the data model

  ii. These CLIENT ACTIONS introduce NETWORK OVERHEAD due to multiple invocations
  iii. Client would become TIGHTLY coupled with the business services as well
  iv. Decouple the CLIENT from model CONSTRUCTION and introduce TRANSFER OBJECT ASSEMBLER

l. Using EJB for LONG LIVED transactions
  i. Enterprise beans are suitable for SYNCRHONOUS processing , EJB do well if the method implemented in a bean produces an OUTCOME within a predictable and ACCEPTABLE time period
  ii. If EJB method takes significant amount of time to process a client requests or it if BLOCKS while processing , this also BLOCKS CONTAINER RESOURCES
  iii. Implement ASYNCHRONOUS processing service using a MESSAGE-ORIENTED middleware with a JMS API to facilitate long lived transactions
  iv. Use SERVICE ACTIVATOR hence

m. Stateless Session Bean RECONSTRUCT conversational state for each invocation
  i. Analyze the INTERACTION model before choosing between STATELESS or STATEFUL

n.

PRESENTATION TIRE REFACTORING

1. **INTRODUCE A CONTROLLER –** Control LOGIC is scattered throughout the application , typically DUPLICATED in MULTIPLE JSP views
    a. **Extract control logic to ONE or MORE controller classes that serve as the INITIAL contact point for handling client requests**
2. **INTRODUCE SYNCHRONIZER TOKEN –** Clients make DUPLICATE resource requests.
3. **LOCALIZE DISPARATE LOGIC –** Business logic and presentation formatting is intermingled within JSP.
    a. **Extract business logic to HELPER classes so that JSP and CONTROLLERS can reuse** (Business Logic is the one which is sent to HELPER CLASSES)
4. **HIDE PRESENTATION TIER SPECIFIC FROM BUSINESS TIER –** Request handling or protocol related data structures are exposed from the presentation tier to business tier
    a. **Pass values between layers using more GENERIC data structures**
5. **REMOVE CONVERSIONS FROM VIEW –** Portions of MODEL are converted to DISPLAY within a VIEW component
    a. **Extract all conversion code from view and encapsulate it in one or more VIEW HELPERS**
6. **HIDE RESOURCES FROM A CLIENT –** Certain resources such as JSP views are directly accessible by clients though access should be restricted
    a. **Hide certain resources via CONTAINER CONFIGURATION or using a controller component**

BUSINESS AND INTEGRATION TIER REFACTORING

1. **WRAP ENTITIES WITH SESSION –** Entity beans in BUSINESS tier are EXPOSED To clients in another tier
    a. **Use a SESSION FAÇADE to encapsulate ENTITY BEAN**
2. **INTRODUCE BUSINESS DELEGATE –** Session BEANS in the BUSINESS tier are exposed to clients in another tier
    a. **Use BUSINESS DELEGATE to decouple the TIERS and HIDE the implementation details**
3. **MERGE SESSION BEANS –** There is one to one mapping between SESSION beans and ENITITY beans
    a. **Maps business services to SESSION BEANS , eliminate or combine SESSION BEANS that act solely as ENTITY BEAN PROXIES into SESSION BEANS that represents COARSE GRAINED business services**
4. **REDUCE INTER ENTITY BEAN COMMUNICATION –** Inter entity bean relationship introduces OVERHEAD in the MODEL
    a. **Reduce the INTER ENTITY BEAN communication using COARSE GRAINED entity beans (COMPOSITE ENTITY)**
5. **MOVE BUSINESS LOGIC TO SESSION –** Inter ENTITY bean relationships introduce OVERHEAD in the MODEL
    a. **Encapsulate WORKFLOW related to INTER ENTITY BEAN relationships in a SESSION BEAN (SESSION FAÇADE)**

GENERAL REFACTORING

1. **SEPARATE DATA ACCESS CODE –** Data Access Code is embedded within a class that has other RESPONSIBILITIES

   a. **Extracts the DATA Access code to a new class and move the new class LOGICALLY and / or PHYSICALLY closer to the DATA SOURCE**

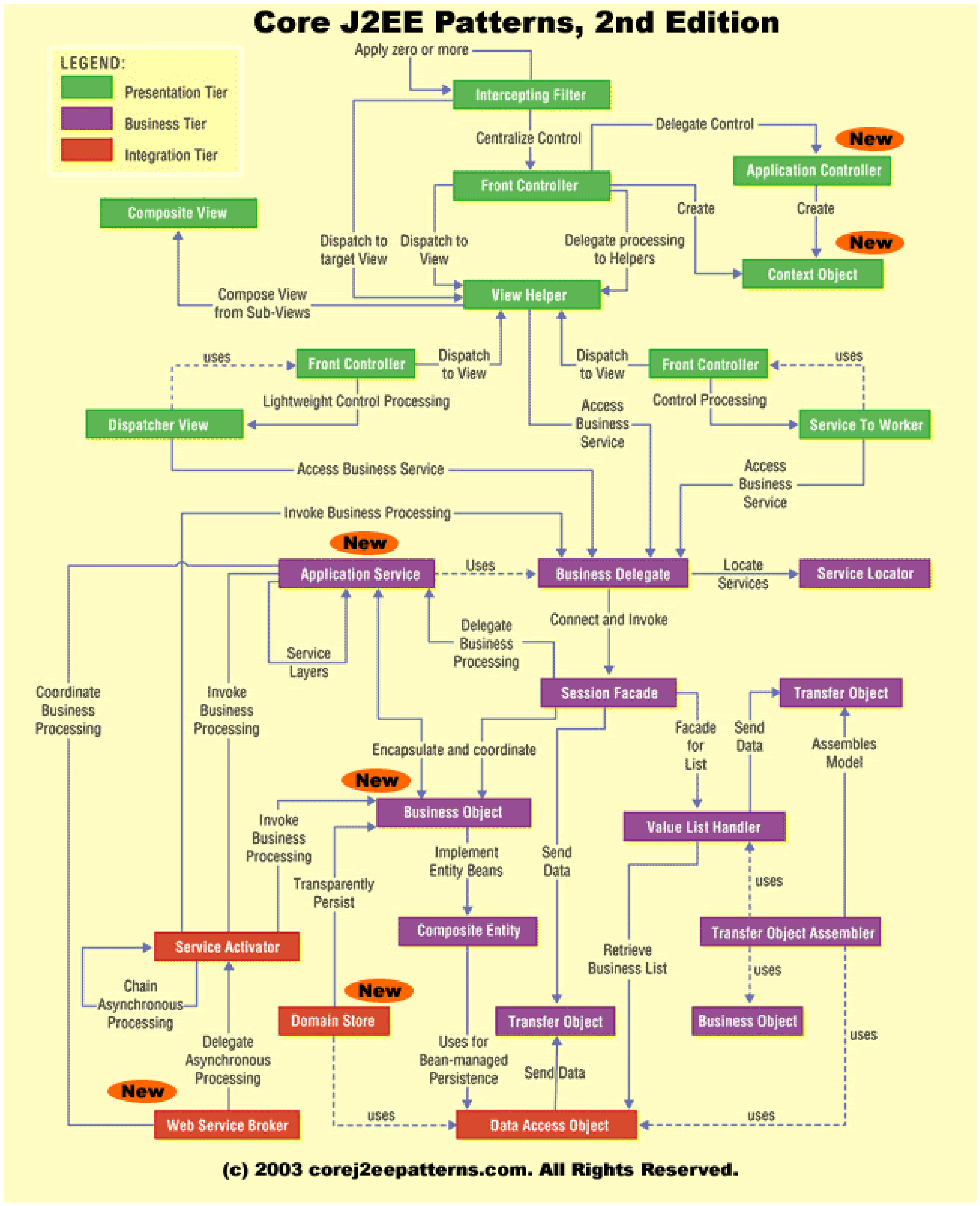
2. **USE A CONNECTION POOL –** data base connections are not shared , instead clients manage their own connections for making data base invocations
   a. **Use a Connection Pool to pre initialize multiple connections improving SCALABILITY and PERFORMANCE**

s/no	**If you are looking for this**	**Find it Here**
PRESENTATION TIER PATTERNS		
1	Preprocessing or Post Processing of REQUESTS	Intercepting Filter
2	Adding Logging, Debugging or some other behavior to be completed FOR EACH REQUEST	Front Controller , Intercepting Filter
3	Centralizing CONTROL for REQUEST HANDLING	Front Controller ,Intercepting Filter ,
4	Creating a generic COMMAND interface or CONTEXT object for reducing coupling between control component and helper components	Front Controller , Application Controller, Context Object
5	Whether to implement a Controller as a Servlet or JSP	Front Controller
6	Creating a VIEW from NUMEROUS SUB views	Composite View
7	Whether to implement the VIEW as a SERVLET of JSP	View Helper
8	How to PARTITION the view and Model	View Helper
9	Where to encapsulate PRESENTATION related data formatting LOGIC	View Helper

10	Combining Multiple Presentation patterns	Intercepting Filter , Dispatcher View
11	Where to encapsulate VIEW MANAGEMENT and NAVIGATION LOGIC which involves CHOOSING and VIEW and DISPATCHING it	Service to Worker, Dispatcher View
12	Where to store SESSION state	Session state in Web Tier , Session State on Business Tier , Session state on Client Tier
13	Controlling client access to a certain view or a sub-view	Controlling client access , hide resource from client
14	Controlling the FLOW of request in to the application	Synchronizer Token , duplicate form submission
15	Controlling duplicate form submission	Introduce Synchronizer token , Duplicate form submission
16	Reducing coupling between PRESENTATION and BUSINESS Tier	Business delegate
17	Partition data access code	DAO
BUSINESS TIER PATTERNS		
1	Minimize coupling between presentation and business tier	Business Delegate
2	Cache business service for clients	Business Delegate
3	Hide implementation details of Business Service lookup , creation and access	Business Delegate , Service Locator
4	Isolate VENDOR and TECHNOLOGY dependencies for service lookup	Service Locator
5	Provide uniform method for business service lookup and creation	Service Locator
6	Hide the complexity and	Service Locator

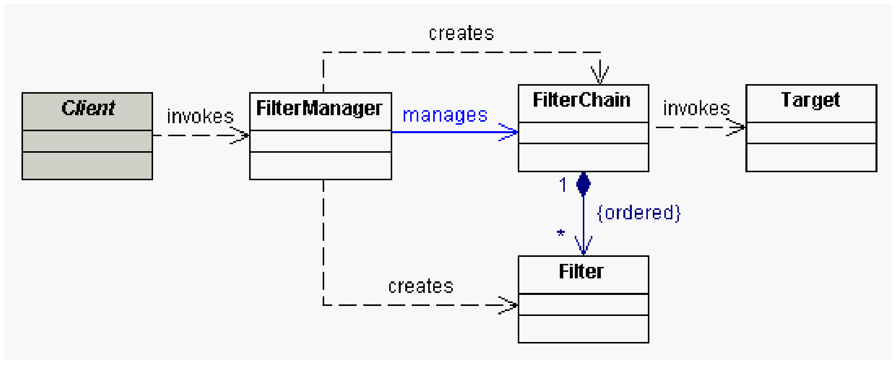
	dependencies of EJB beans and JMS component look up	
7	Transfer Data between business object and client access tier	Transfer Object
8	Provide SIMPLER uniform INTERFACE to remote CLIENTS	Business Delegate , Session Façade , Application Service
9	Reduce Remote method invocations by providing COARSE-GRAINED method access to business tier components	Session Façade
10	Manage the relationships between Enterprise bean components and hide the COMPEXITY of interactions	Session Façade
11	Protect the Business Tier components from DIRECTLY EXPOSING to CLIENTS	Session Façade , Application Service
12	Provide Uniform boundary access to business tier components	Session Façade , Application Service
13	Implement complex conceptual domain model using objects	Business Objects
14	Identify COARSE-GRAINED objects and DEPENDENT objects for Business Objects and Entity Bean design	Business Object , Composite Entity
15	Design for COARSE-GRAINED entity beans	Composite entity
16	Reduce or eliminate the entity bean clients’ dependency on the DATABASE schema	Composite Entity
17	Reduce or eliminate entity bean to entity bean remote relationships	Composite Entity
18	Reduce number of Entity beans and improve manageability	Composite Entity
19	Obtain the application data model for the application from various Business	Transfer Object Assembler

	Components	
20	On the fly construction of the application data model	Transfer Object Assembler
21	Hide the complexity of the data model CONSTRUCTION from the client	Transfer Object Assembler
22	Provide business tier query and results list processing facility	Value List Handler
23	Minimize the overhead of using EJB finder methods	Value List Handler
24	Provide QUERY RESULTS CACHING for the CLIENT on the SERVER side with FORWARD and BACKWARD navigability	Value List Handler
INTEGRATION TIER		
1	Minimize COUPLING between Business and resource tier	Data Access Object
2	Centralize access to Resource Tier	Data Access Object
3	Minimize Complexity of Resource access from Business tier components	Data Access Object
4	Provide Asynchronous Processing for enterprise applications	Service Activator
5	Send and Asynchronous request to a Business Service	Service Activator
6	Asynchronously process a request as a set of parallel tasks	Service Activator
7	Transparently Persist and Object Model	Domain Store
8	Implement a CUSTOM persistence FRAMEWORK	Domain Store
9	Expose a WEB SERVICE using XML and standard Internet Protocol	Web Service Broker
10	Aggregate and Broker existing services as WEB SERVICES	Web Service Broker



1. PRESENTATION TIER PATTERN
    a. Intercepting Filter

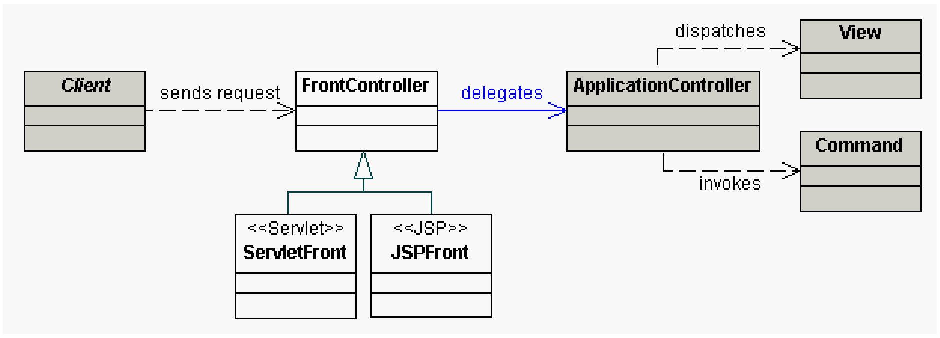
i. You want to INTERCEPT and MANIPULATE request and a response BEFORE and AFTER the request is PROCESSED
ii. Checking the **session validity** , checking **request path violations** , checking the **BROWSER type** , checking the **ENCODING** that the client is using , checking if the **request is ENCRYPTED** or COMPRESSED etc could be done using an **INTERCEPTING FILTER**
iii. Sharing INFORMATION between filters CAN BE INEFFICIENT
iv.



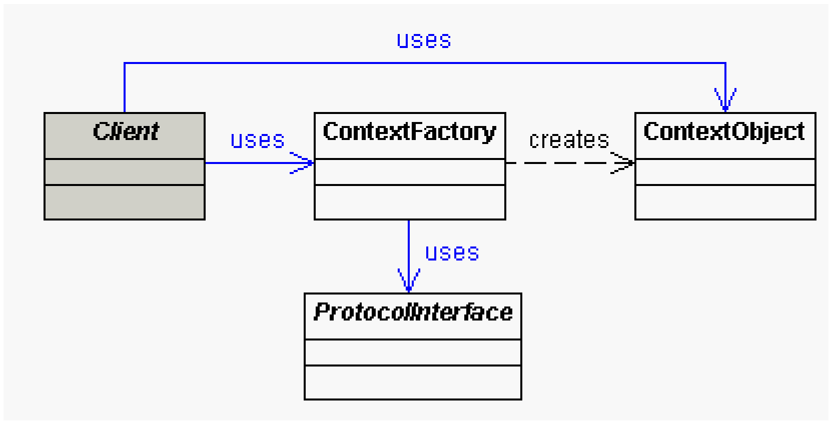
b. Front Controller
i. Centralized access point for PRESENTATION tier REQUEST handling
ii. Without centralized place the CONTROL code that is common across MULTIPLE request would have to be DUPLICATED
iii. Avoid duplicate control logic , common logic for multiple request , separate system processing logic

from the View , centralize controlled access points to the system

iv. Use FRONT CONTROLLER as the INITIAL point of contact
v. A controller TYPICALLY users a APPLICATION CONTROLLER for ACTION and VIEW management
vi. Contains PROTOCOLE SPECIFIC code
vii.

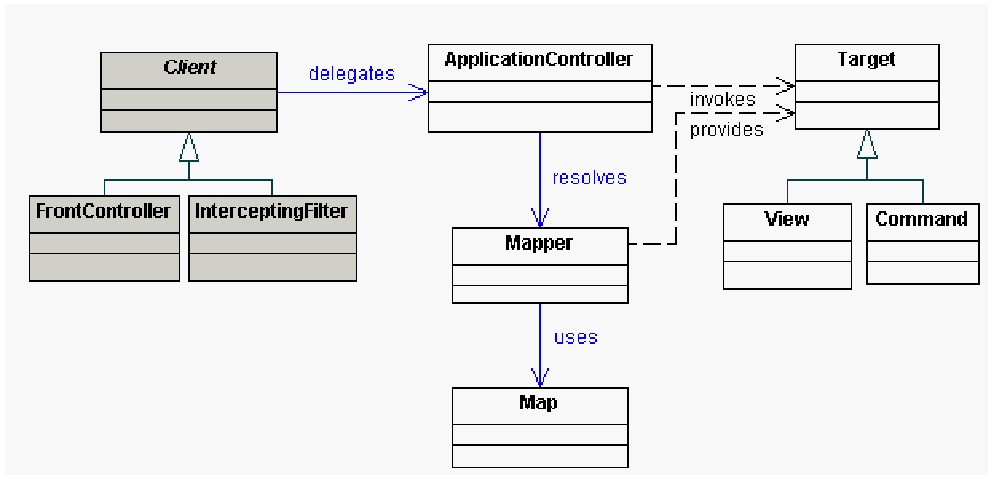


c. Context Object
   i. Avoid PROTOCOL specific SYSTEM information OUTSIDE of its RELEVANT context
   ii. Ex; HTTPREQUEST must not be passed between layers , instead HTTPREQUEST values must be transferred to a CONTEXT object and that object must be used
   iii. Application would not be coupled to a SPECIFIC PROTOCOL
   iv. Context Objects MAIN goal is to SHARE SYSTEMS INFORMATION IN A PROTOCOL INDEPENDENT WAY , Transfer Objects MAIN goal is to REDUCE NETWORK COMMUNICATION , IMPROVING PERFORMACE

v. 

d. Application Controller
   i. Centralize and Modularize ACTION and VIEW management
   ii. Resolving the incoming request to an ACTION is ACTION MANAGEMENT
   iii. Locating an APPROPRIATE VIEW is VIEW MANAGEMENT
   iv. Reuse action an view management , improve request handling EXTENSIBILITY , improve code MODULARITY and MAINTAINABILITY

v.

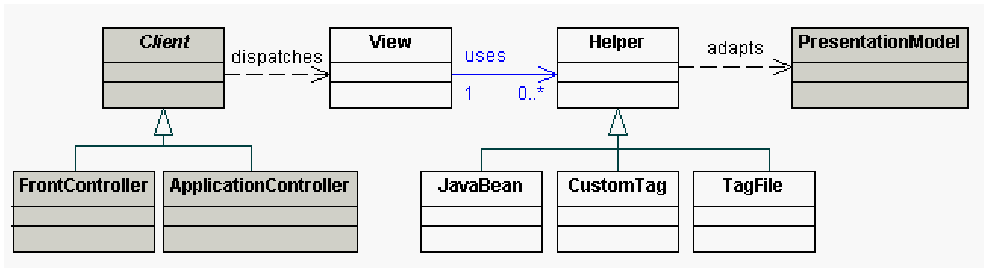


e. View Helper

i. Separate a view from its PROCESSING LOGIC

ii. Want to use TEMPLATE based views such as JSP , want to AVOID embedding program logic in the VIEW , Want to SEPARATE programming logic from the view to facilitate DIVISION of LABOR between SOFTWARE DEVELOPERS and WEB PAGE DESIGNERS

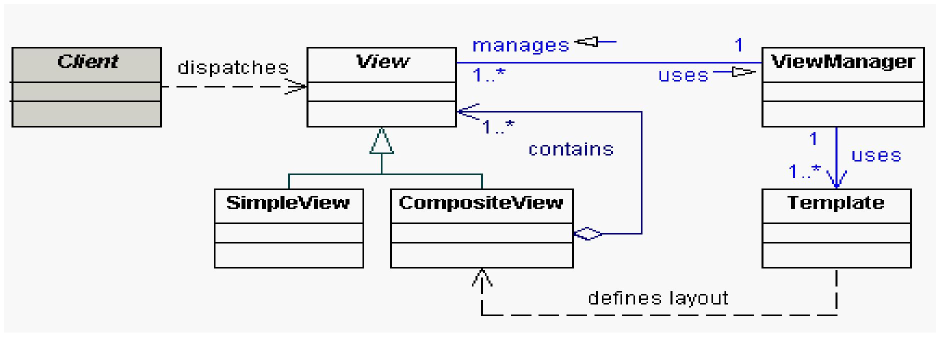
iii.



f. Composite View

i. Build a view from MODULAR , ATOMIC component parts that are COMBINED to create a COMPOSITE WHOLE, while managing the content and the LAYOUT independently

ii.



g. Service to Worker

i. Perform CORE request HANDLING and INVOKE BUSINESS LOGIC before control is passes to the VIEW

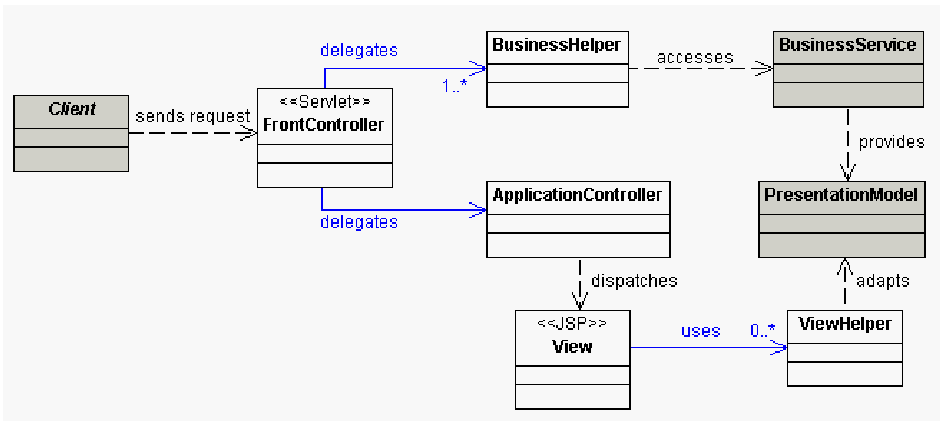
ii. Business and DATA service is invoked BEFORE the view is rendered

iii. How sophisticated is the Control Logic , How dynamic is the RESPONSE content , How sophisticated is the BUSINESS LOGIC and MODEL

iv. Want specific BUSINESS logic executed to SERVICE a request in order to RETRIEVE content that will be used to GENERATE a DYNAMIC response

v. Have View selections which may depend on the response from BUSINESS SERVICE

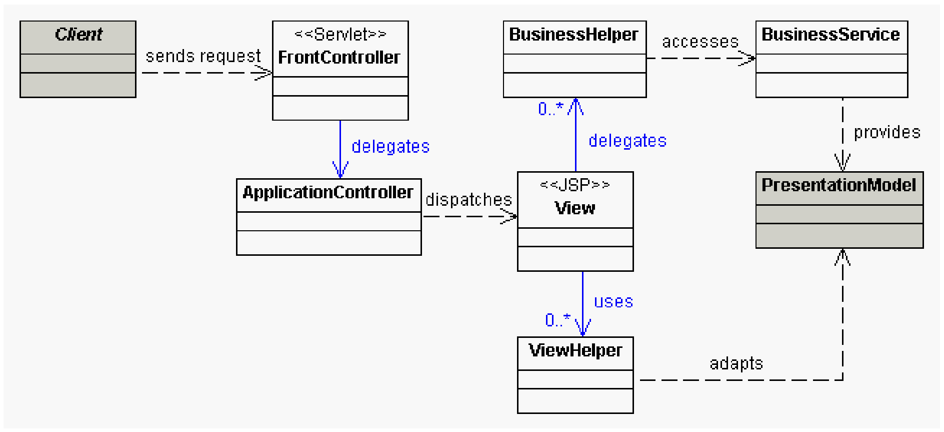
vi.



h. Dispatcher View

  i. Want a VIEW to handle a REQUEST and GENERATE a RESPONSE , while managing LIMITED amount of BUSINESS PROCESSING
  ii. Have static views , have views GENERATED from an EXISTING MODEL , have views which are INDEPENDENT of any BUSIENSS SERVICE RESPONSE , have LIMITED business processing

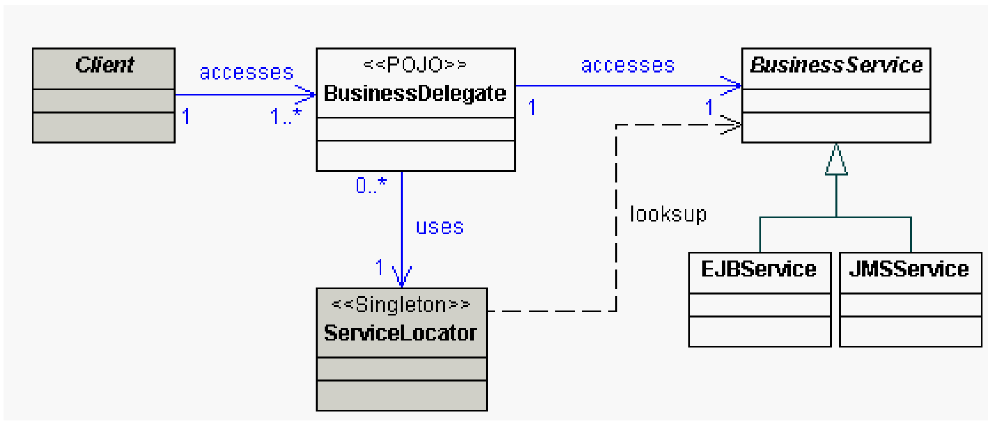
iii.



2. BUSINESS TIER PATTERN
   a. Business Delegate
      i. Want to hide clients from the COMPLEXITY of remote communication with business service components
      ii. Want to access business tier components from your presentation tier components clients , such as DEVICES , WEB SERVICES and RICH CLIENTS
      iii. Want to AVOID un necessary invocation of REMOTE SERVICES
      iv. Want to translate network exceptions into APPLICATION or USER EXCEPTIONS
      v. Want to hide the details of SERVICE CREATION , RECONFIGURATION and INVOCATION RETRIES from the client
      vi. Reduced coupling , improved maintainability , translation of business exceptions , improves availability , expose a simpler uniform interface to

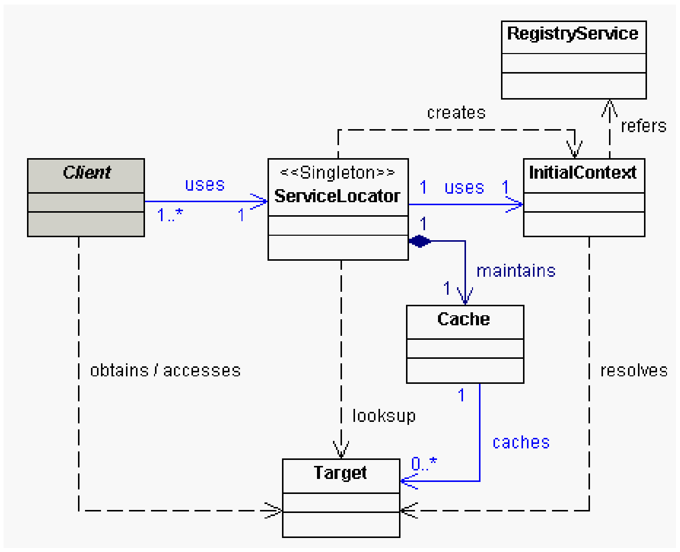
the business tier , improved performance , introduced additional layer , hides remoteness

vii.

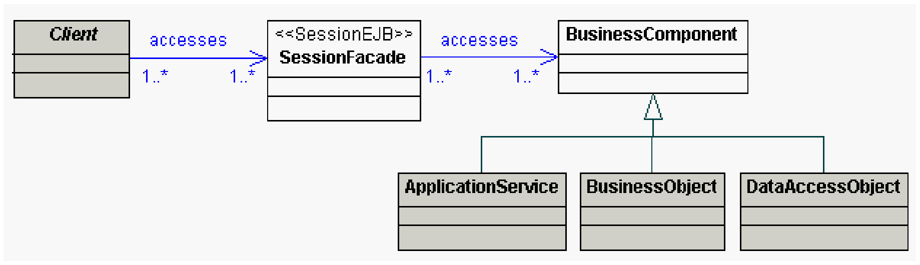


b. Service Locator

i. Want to transparently locate business components and services in a uniform manner
ii. Want to use JNDI lookup and business components , JMS components , data sources
iii. Centralize and reuse the implementation of lookup in J2EE clients
iv. Encapsulate vendor dependencies for registry implementation
v. Avoid performance overhead related to INITIAL context and Service Lookups
vi. Re-establish a CONNECTION to previously accessed enterprise beans instances using HANDLE
vii. If you need to locate a WEB SERVICE published in UDDI , Web Service Locator can be used

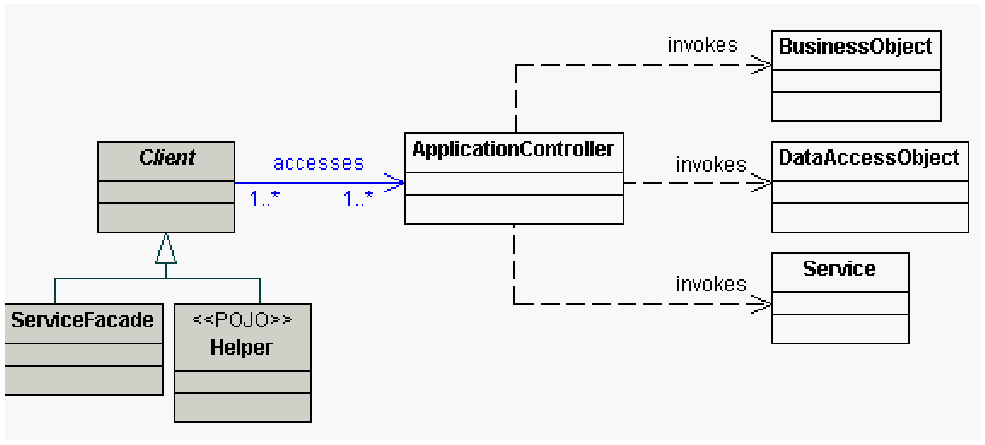
viii. 

c. Session façade
    i. To expose BUSINESS COMPONENTS and SERVICES to REMOTE CLIENTS
    ii. Address two issues, CONTROLLING CLIENT ACCESS to Business Objects , and LIMITING NETWORK TRAFFIC between remote clients and FINE-GRAINED business components and services
    iii.



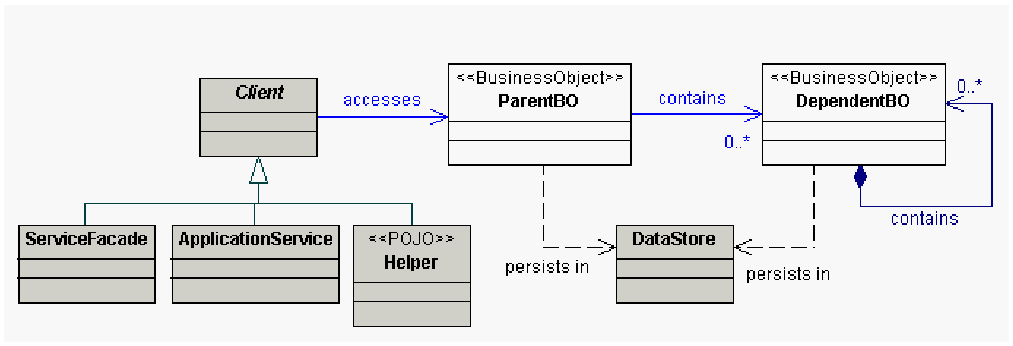
d. Application Service
    i. Want to CENTRALIZE business logic ACROSS several business tier components and services

ii. Want to MINIMISE business logic in SESSION FAÇADE(S)
iii. If business logic is embedded in Session Facades , the reusability of that logic is effected , hence use APPLICATION SERVICE
iv. When you see BUSINESS LOGIC is becoming duplicated in SESSION FAÇADE , it is good to introduce APPLICATION SERVICE
v.



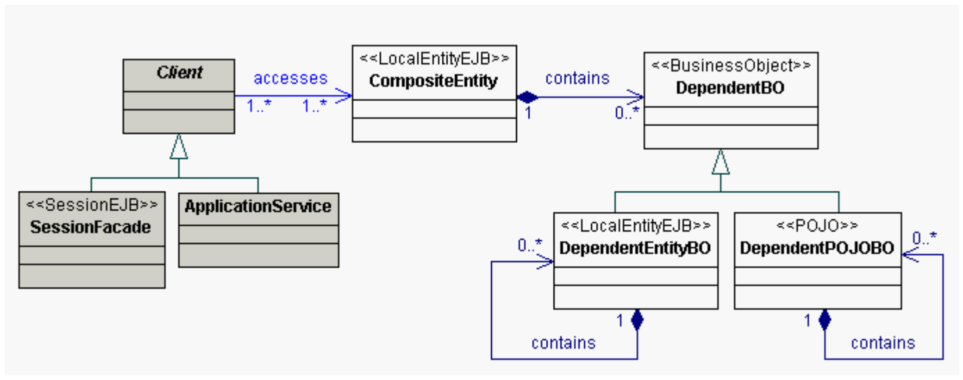
e. Business Object
   i. Have a CONCEPTUAL domain model with BUSINESS LOGIC and RELATIONSHIP
   ii. Have a conceptual model with sophisticated business logic , validation and business rules

iii.



f. Composite Entity
   i. Want to use ENTITY beans to implement the CONCEPTUAL domain model
   ii. Want to AVOID drawbacks of remote entity beans , such as network overhead , inter-entity bean relationships
   iii. Want to implement PARENT-CHILD relationship efficiently when implementing BUSINESS OBJECTS as ENTITY BEANS
   iv. Want to encapsulate PHYSICAL DB design from the clients

v.

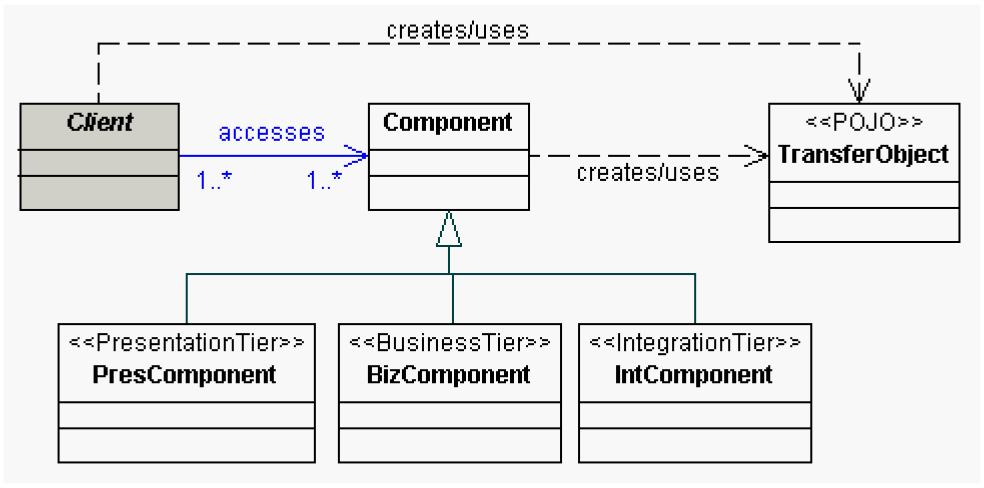


g. Transfer Object

   i. Want to TRANSFER multiple data elements over a TIER

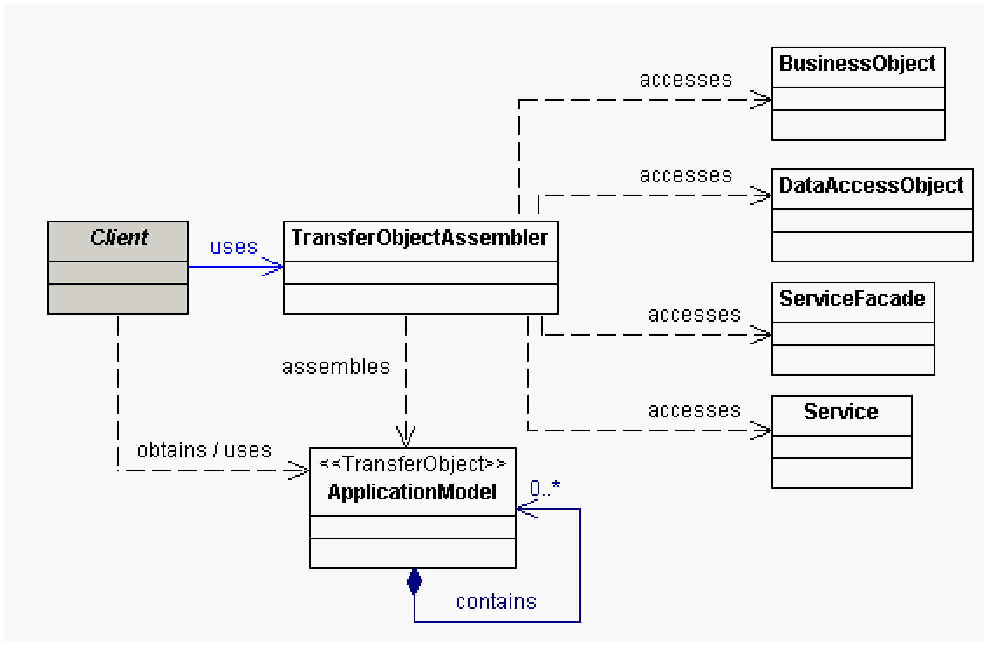
   ii. Reduces Network traffic , Simplifies remote object and interface , Transfer more data in fewer calls ,reduce code duplication, introduces stale transfer object

iii.



h. Transfer Object Assembler
   i. Want to obtain an application model that aggregates transfer objects from several business components
   ii. Transfer object Assembler helps BUILD an application model as a COMPOSITE TRANSFER OBJECT
   iii. Transfer Object Assembler AGGREGATES multiple TRANSFER OBJECTS from various business components and services and return to the client

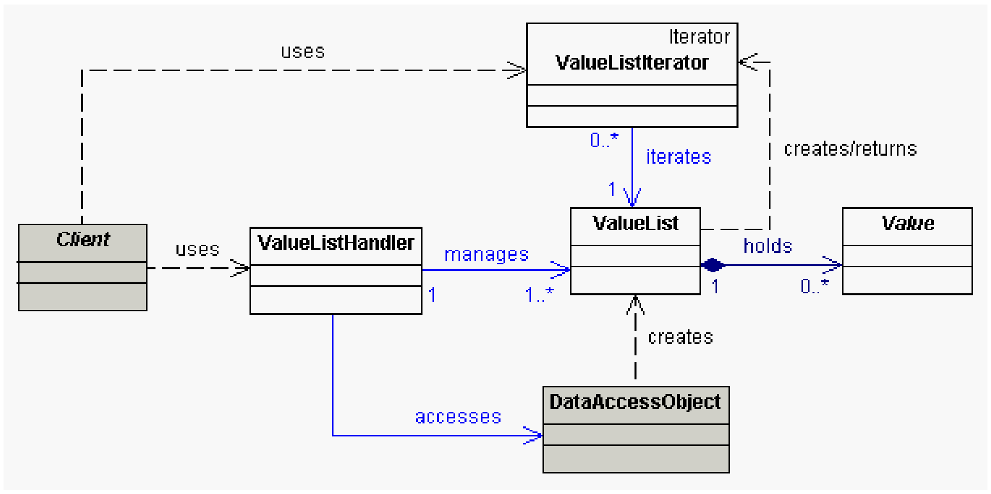
iv.



i. Value List Handler
   i. Have a REMOTE clients that wants to ITERATE over a LARGE RESULT SET
   ii. Want to AVOID overhead of using EJB FINDER methods , want to implement READ-ONLY use case that does not need transactions , Wants to provide CLIENTS with an EFFICIENT search and iterate mechanism over a large result set , wants to maintain the search results on the server side

iii. Use VALUE LIST HANDLER to SEARCH , CACHE results and allow the clients to TRAVERS and select ITEMS from the results

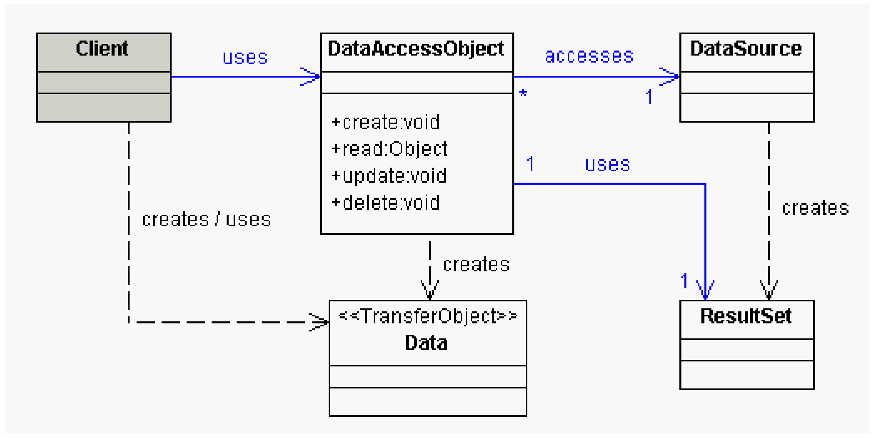
iv.



3. INTEGRATION TIER PATTERN
   a. Data Access Object
      i. Want to ENCAPSULATE data access and manipulation in a separate layer
      ii. Want to decouple the PERSISTENT STORAGE implementation from the rest of the application
      iii. Want to provide UNIFORM data access API for a persistence mechanism to various types of data sources , such as RDBMS , LDAP , OODB , XML repositories , FLAT files etc

iv. Want to organize data access logic and encapsulate proprietary features to facilitate maintainability and portability

v.



b. Service Activator

i. Want to INVOKE services ASYNCHRONOUSLY

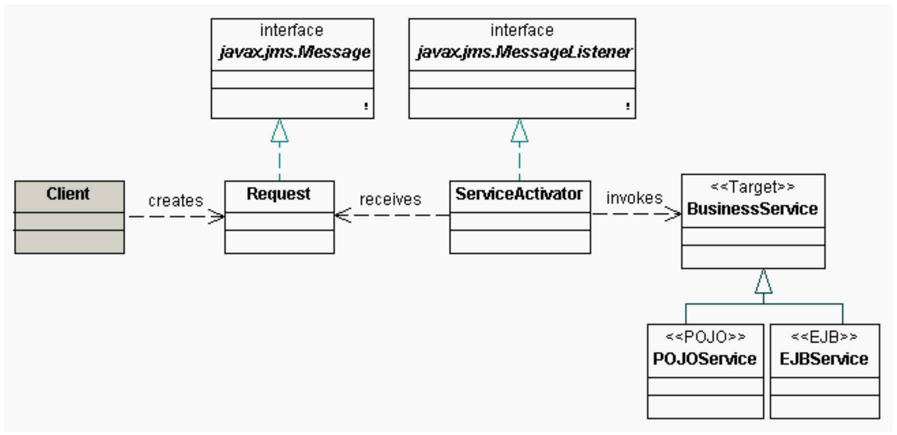
ii. Want to integrate PUBLISH/SUBSCRIBER and POINT to POINT messaging to enable ASYNCRONOUS processing services

iii. ServiceActivator can be implemented as a POJO service activator or a Message Driven Bean

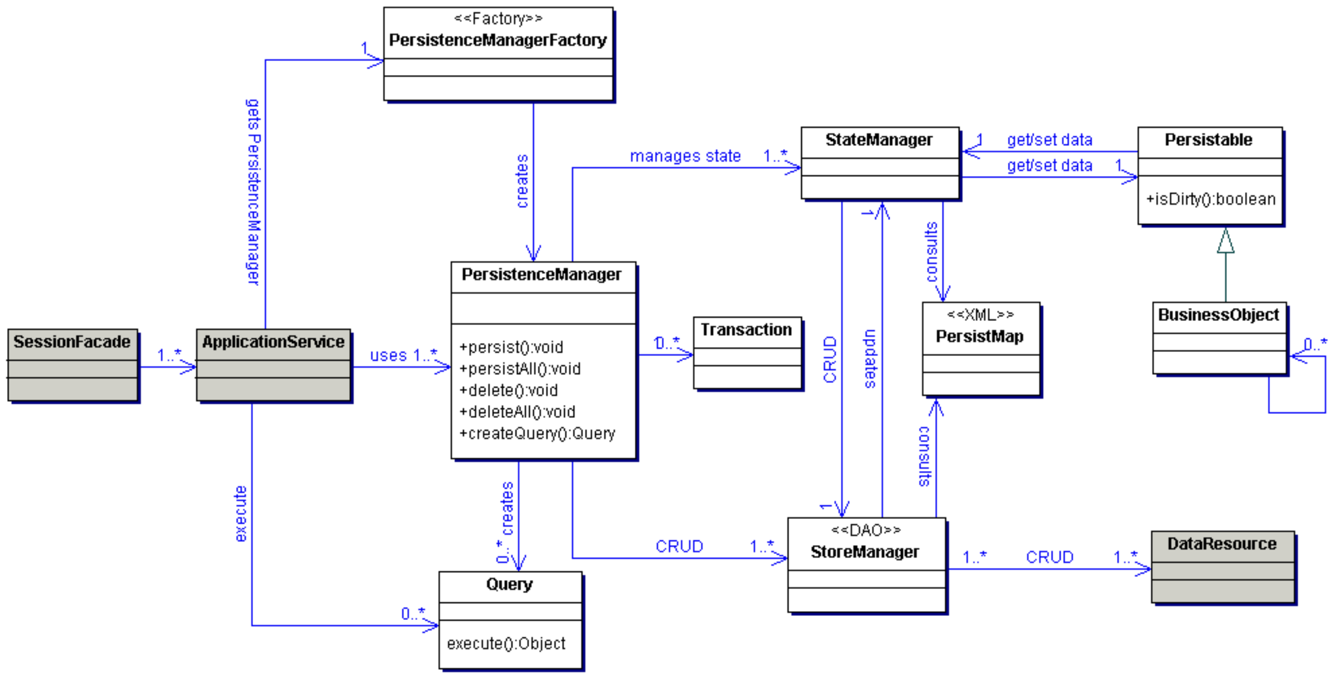
iv. In case of POJO service activator , many methods need to be written while MBD , only onMessage method needs to be written

v. Want to perform BUSINESS TASKS that is logically COMPOSED of SEVERAL BUSINESS TASKS

vi.



c. Domain Store
   i. Want to SEPARATE persistence FROM the OBJECT MODEL
   ii. Want to avoid putting persistence details in to the Business Objects
   iii. Do not want to use Entity Beans
   iv. The application might be running in a WEB SERVER
   v. Object model users INHERITANCE and COMPLEX RELATIONSHIPS



d. Web Service Broker

i. Want to provide access to ONE or MODE services using XML and WEB PROTOCOLS

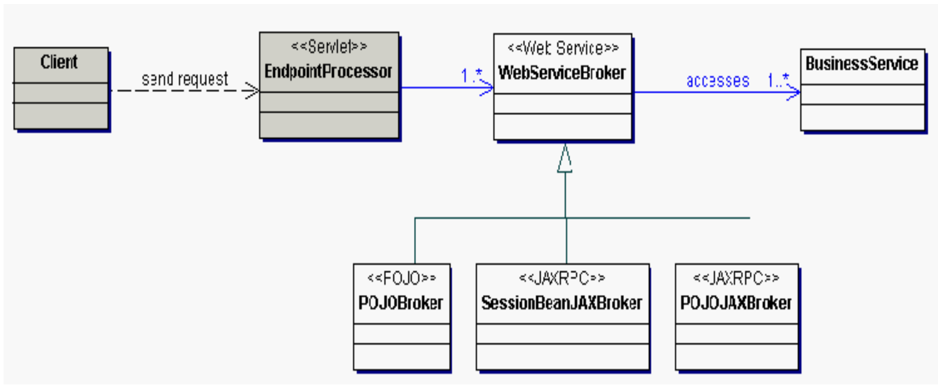
ii. Want to REUSE and EXPOSE existing services to CLIENTS

iii. Want to MONITOR an potentially LIMIT the usage of EXPOSED services

iv. Web service broker is a WEB SERVICE that serves as a BROKER to ONE or MODE services. Those services can be J2EE services , such as SESSION BEANS , APPLICATION SERVICES or LEGACY EIS SYSTEMS

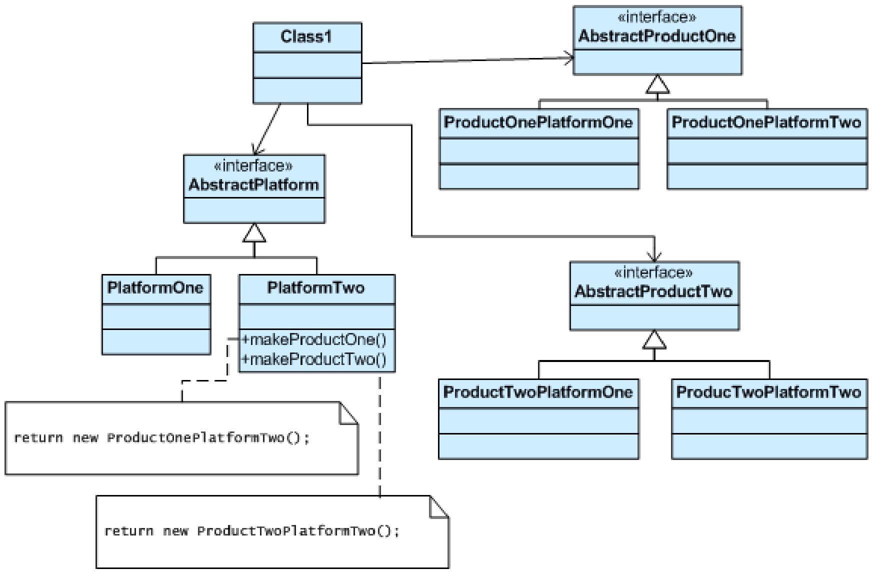
v. SessionBeanJAXBroker , POJOJAXBroker , POJOBroker are few realizations of web service brokers

vi.

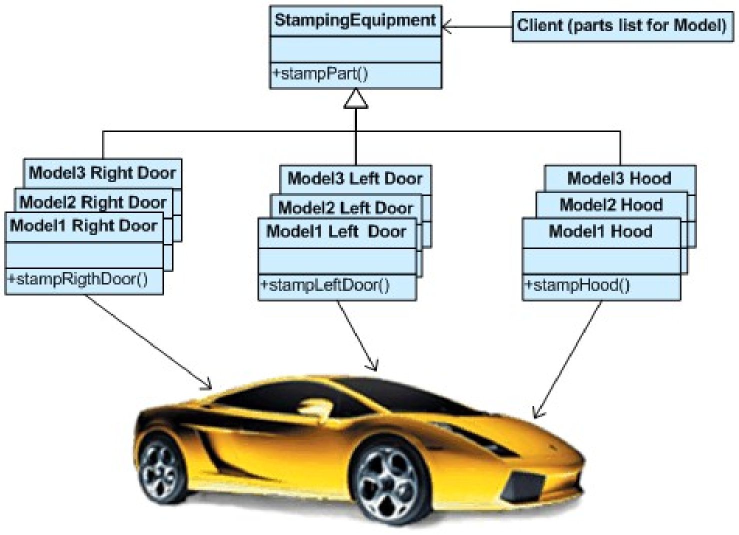


GANG OF FOUR (GOF) PATTERNS

1. CREATIONAL PATTERNS
   a. Abstract Factory
      i. Provides and INTERFACE for crating FAMILIES of RELATED or DEPENDENT objects without specifying their CONCREATE CLASSES
      ii. A hierarchy that encapsulate : many possible platforms and construction of suite of PRODUCTS
      iii. The new operator considered harmful


Class1
«interface» AbstractProductOne
ProductOnePlatformOne
ProductOnePlatformTwo
«interface» AbstractPlatform
PlatformOne
PlatformTwo
+makeProductOne()
+makeProductTwo()
«interface» AbstractProductTwo
ProductTwoPlatformOne
ProducTwoPlatformTwo
return new ProductOnePlatformTwo();
return new ProductTwoPlatformTwo();

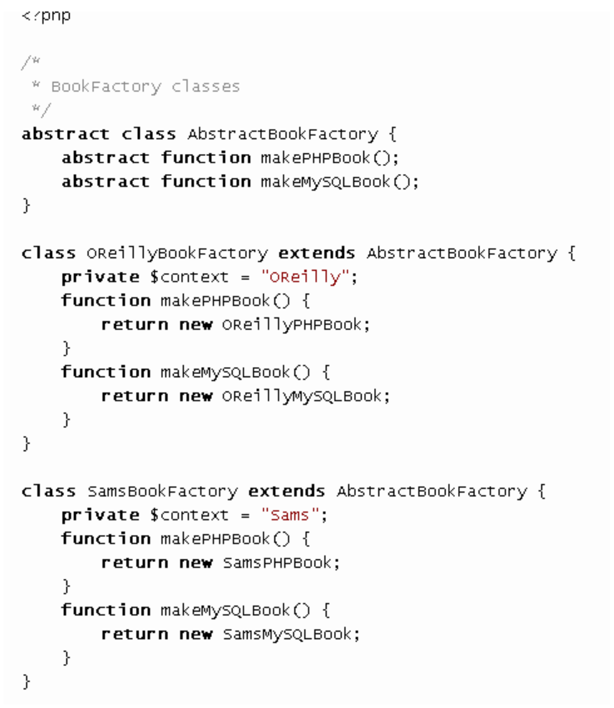
iv.



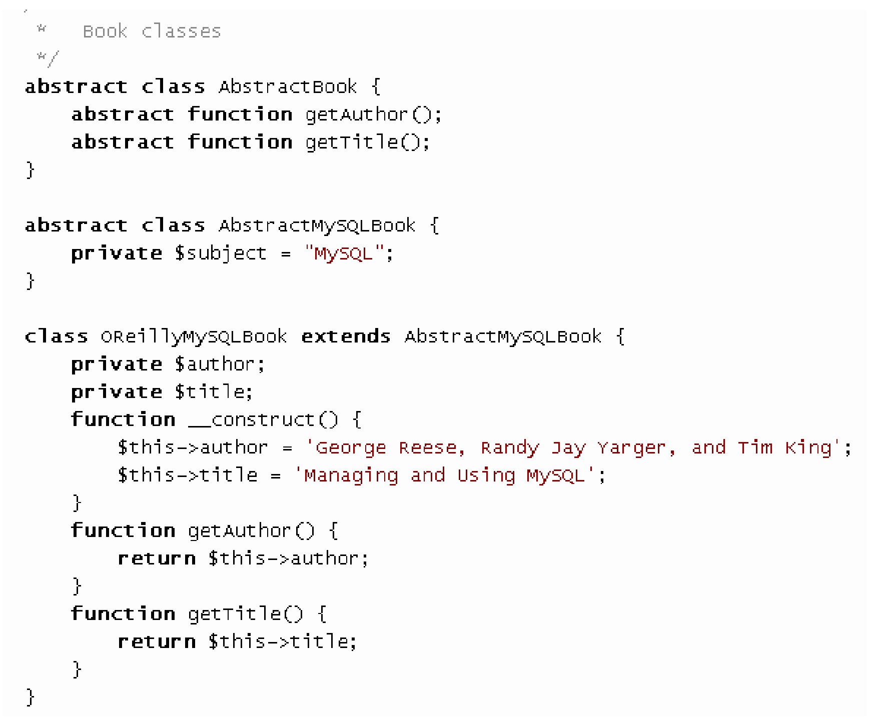
v. Sometimes Creational patterns are competitive, there are cases wither PROTOTYPE or ABSTRACT FACTORY could be used profitable. At other times they are complementary. Abstract factory might store a set of PROTOTYPES from which components get built

vi. Abstract Factory , Builder and Prototype can USE Singleton in their implementation
vii. Abstract Factory , Builder and Prototype defines a FACTORY object that is responsible for knowing and creating the class of product objects and make it parameter of the system
viii. Abstract Factory has the Factory object producing objects of several classes
ix. Builder has a Factory object building a COMPLEX product incrementally using a corresponding complex PROTOCOL
x. Prototype has the FACTORY object building a product by copying a PROTOTYPE object
xi. Abstract Factory can be used as an ALTERNATIVE to FAÇADE to hide platform specific classes
xii. Abstract Factory EMPHASIS on FAMILY of RELATED PRODUCTS and this is the most important one
xiii. Often design starts out using a FACTORY METHOD (less complicated) and EVOLVE towards ABSTRACT FACTORY , PROTOTYPE or BUILDER

LIYANA ARACHCHIGE RANIL

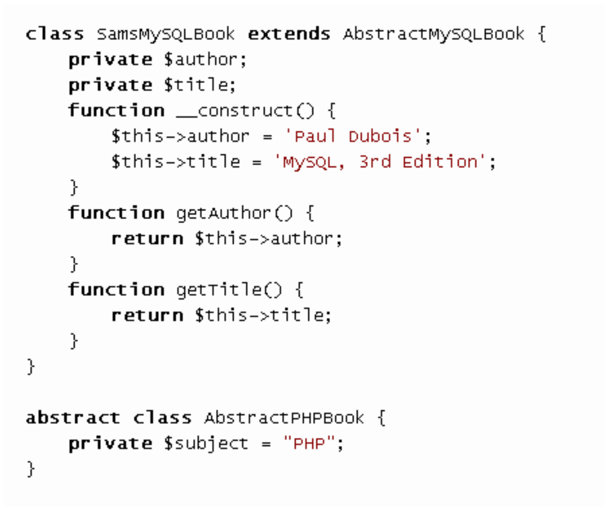


```
<?php

/*
 * BookFactory classes
 */
abstract class AbstractBookFactory {
    abstract function makePHPBook();
    abstract function makeMySQLBook();
}

class OReillyBookFactory extends AbstractBookFactory {
    private $context = "OReilly";
    function makePHPBook() {
        return new OReillyPHPBook;
    }
    function makeMySQLBook() {
        return new OReillyMySQLBook;
    }
}

class SamsBookFactory extends AbstractBookFactory {
    private $context = "Sams";
    function makePHPBook() {
        return new SamsPHPBook;
    }
    function makeMySQLBook() {
        return new SamsMySQLBook;
    }
}
```

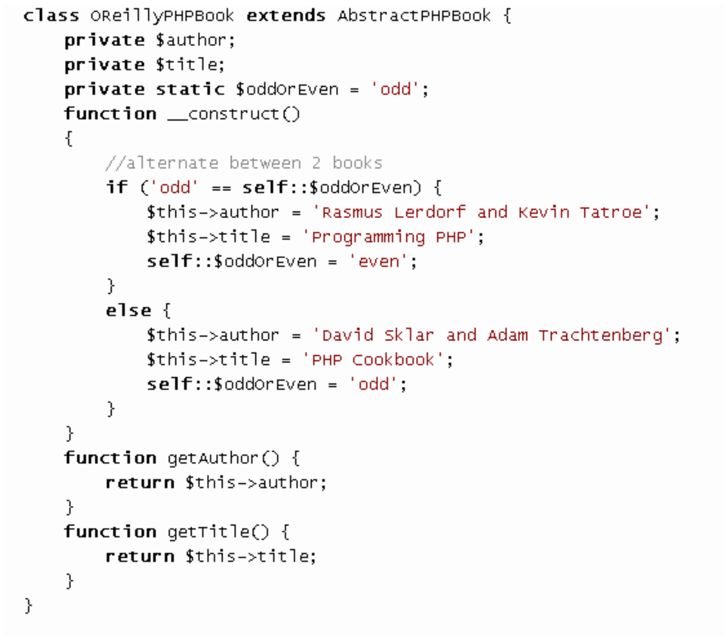


```
 *   Book classes
 */
abstract class AbstractBook {
    abstract function getAuthor();
    abstract function getTitle();
}

abstract class AbstractMySQLBook {
    private $subject = "MySQL";
}

class OReillyMySQLBook extends AbstractMySQLBook {
    private $author;
    private $title;
    function __construct() {
        $this->author = 'George Reese, Randy Jay Yarger, and Tim King';
        $this->title = 'Managing and Using MySQL';
    }
    function getAuthor() {
        return $this->author;
    }
    function getTitle() {
        return $this->title;
    }
}
```

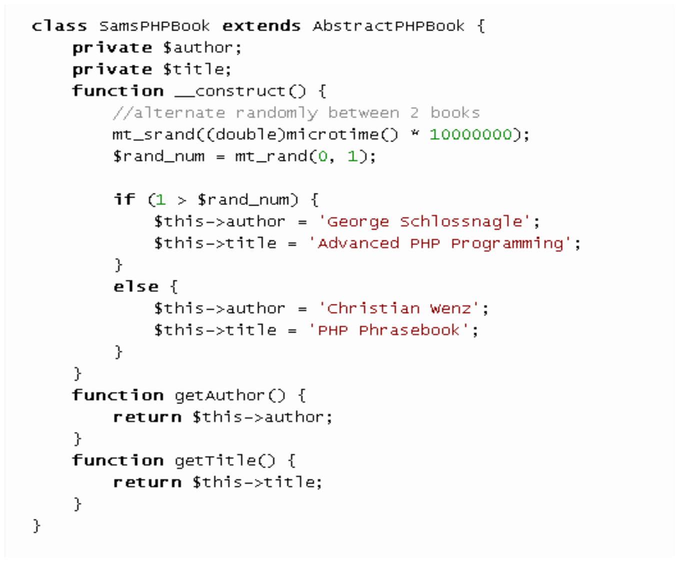
LIYANA ARACHCHIGE RANIL



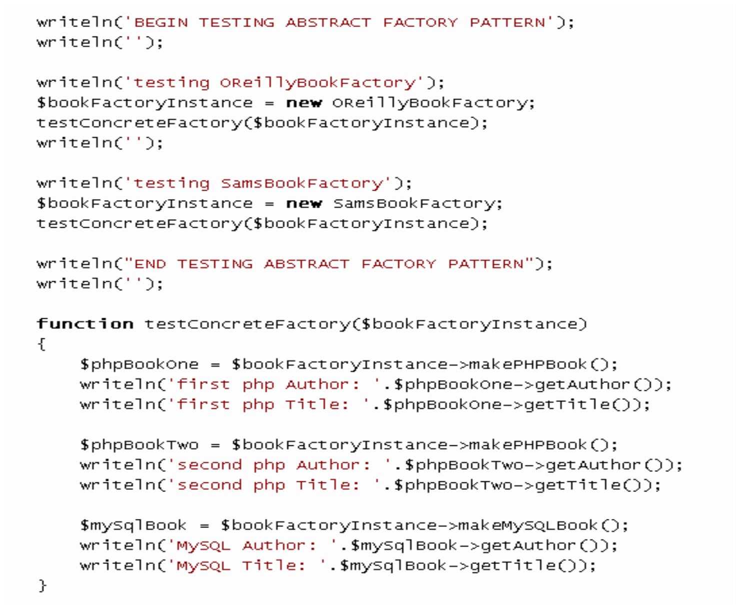
```
class SamsMySQLBook extends AbstractMySQLBook {
    private $author;
    private $title;
    function __construct() {
        $this->author = 'Paul Dubois';
        $this->title = 'MySQL, 3rd Edition';
    }
    function getAuthor() {
        return $this->author;
    }
    function getTitle() {
        return $this->title;
    }
}

abstract class AbstractPHPBook {
    private $subject = "PHP";
}
```



```
class OReillyPHPBook extends AbstractPHPBook {
    private $author;
    private $title;
    private static $oddOrEven = 'odd';
    function __construct()
    {
        //alternate between 2 books
        if ('odd' == self::$oddOrEven) {
            $this->author = 'Rasmus Lerdorf and Kevin Tatroe';
            $this->title = 'Programming PHP';
            self::$oddOrEven = 'even';
        }
        else {
            $this->author = 'David Sklar and Adam Trachtenberg';
            $this->title = 'PHP Cookbook';
            self::$oddOrEven = 'odd';
        }
    }
    function getAuthor() {
        return $this->author;
    }
    function getTitle() {
        return $this->title;
    }
}
```



```
class SamsPHPBook extends AbstractPHPBook {
    private $author;
    private $title;
    function __construct() {
        //alternate randomly between 2 books
        mt_srand((double)microtime() * 10000000);
        $rand_num = mt_rand(0, 1);

        if (1 > $rand_num) {
            $this->author = 'George Schlossnagle';
            $this->title = 'Advanced PHP Programming';
        }
        else {
            $this->author = 'Christian Wenz';
            $this->title = 'PHP Phrasebook';
        }
    }
    function getAuthor() {
        return $this->author;
    }
    function getTitle() {
        return $this->title;
    }
}
```



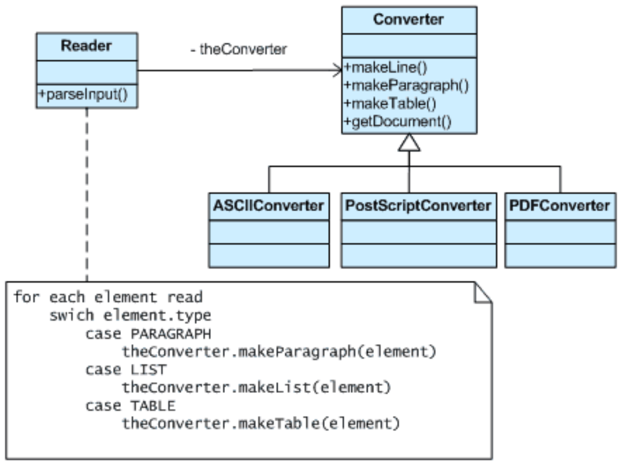
```
writeln('BEGIN TESTING ABSTRACT FACTORY PATTERN');
writeln('');

writeln('testing OReillyBookFactory');
$bookFactoryInstance = new OReillyBookFactory;
testConcreteFactory($bookFactoryInstance);
writeln('');

writeln('testing SamsBookFactory');
$bookFactoryInstance = new SamsBookFactory;
testConcreteFactory($bookFactoryInstance);

writeln("END TESTING ABSTRACT FACTORY PATTERN");
writeln('');

function testConcreteFactory($bookFactoryInstance)
{
    $phpBookOne = $bookFactoryInstance->makePHPBook();
    writeln('first php Author: '.$phpBookOne->getAuthor());
    writeln('first php Title: '.$phpBookOne->getTitle());

    $phpBookTwo = $bookFactoryInstance->makePHPBook();
    writeln('second php Author: '.$phpBookTwo->getAuthor());
    writeln('second php Title: '.$phpBookTwo->getTitle());

    $mySqlBook = $bookFactoryInstance->makeMySQLBook();
    writeln('MySQL Author: '.$mySqlBook->getAuthor());
    writeln('MySQL Title: '.$mySqlBook->getTitle());
}
```

b. Builder

   i. Separate the construction of a COMPLEX object from its REPRESENTATION , so that the SAME CONSTRUCTION process can create DIFFERENT REPRESENTATION

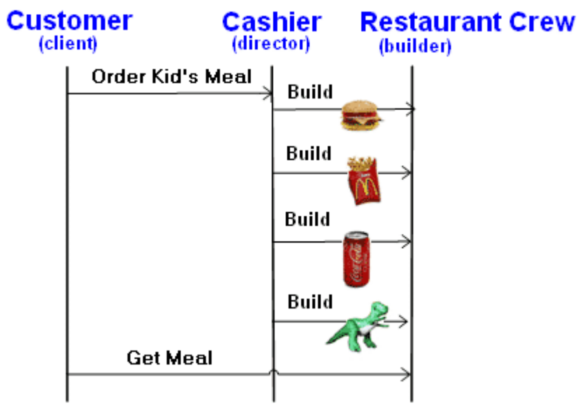
   ii. An application needs to create ELEMENTS of a COMPLEX AGGREGATE

iii. The DIRECTOR invokes BUILDER services as it interprets the external format
iv. The BUILDER creates parts of the complex object each time it is called and maintains all the INTERMEDIATE STATE
v. Afford FINER control over CONSTRUCT PROCESS
vi. In FAST FOOD stores this pattern is in use (Construction of a Childs meal)
vii. Check list
    1. Common Input and many possible representations
    2. Encapsulate parsing of the common input in a READER class
    3. Design a standard PROTOCOLE for creating all possible output REPRESENTATIONS. Capture the steps of this protocol in a BUILDER INTERFACE
    4. Define a builder derived class for each target REPRESENTATION
    5. Client creates READER object and a BUILDER object and register the latter with the former
    6. Client asks the READER to construct
    7. Client ASK the BUILDER to return the results



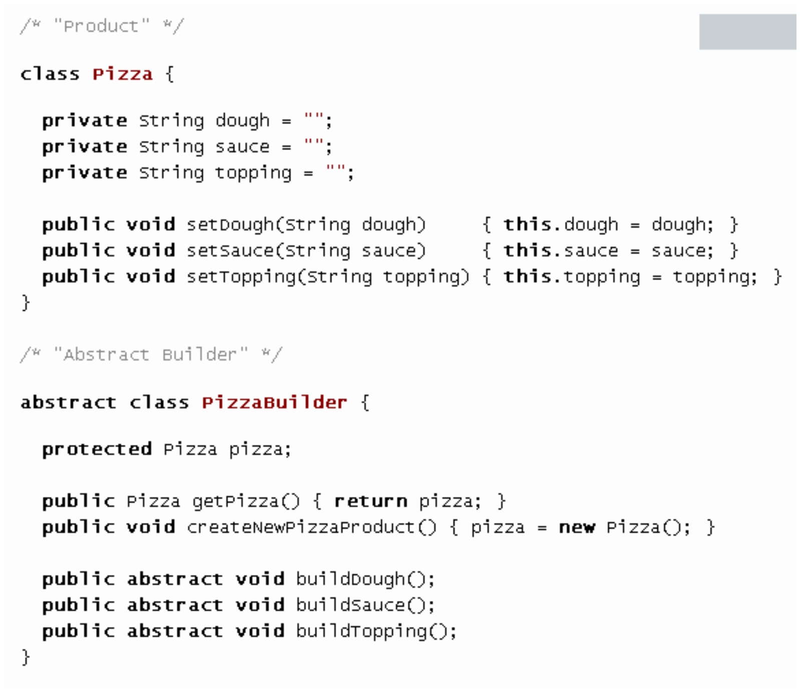
```
for each element read
    swich element.type
        case PARAGRAPH
            theConverter.makeParagraph(element)
        case LIST
            theConverter.makeList(element)
        case TABLE
            theConverter.makeTable(element)
```

viii.

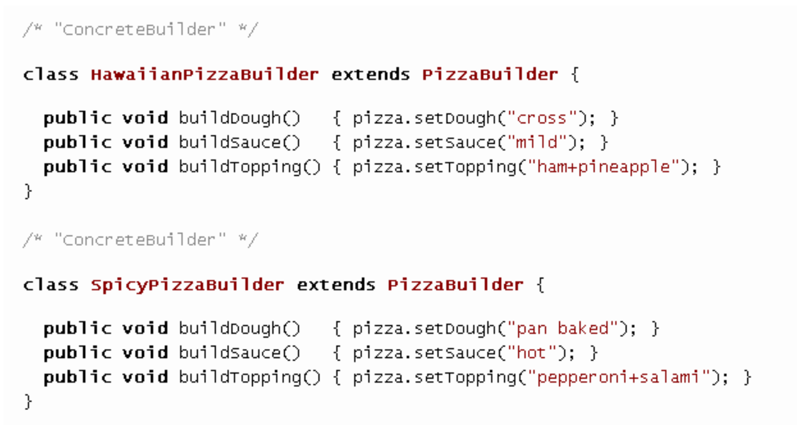


ix.

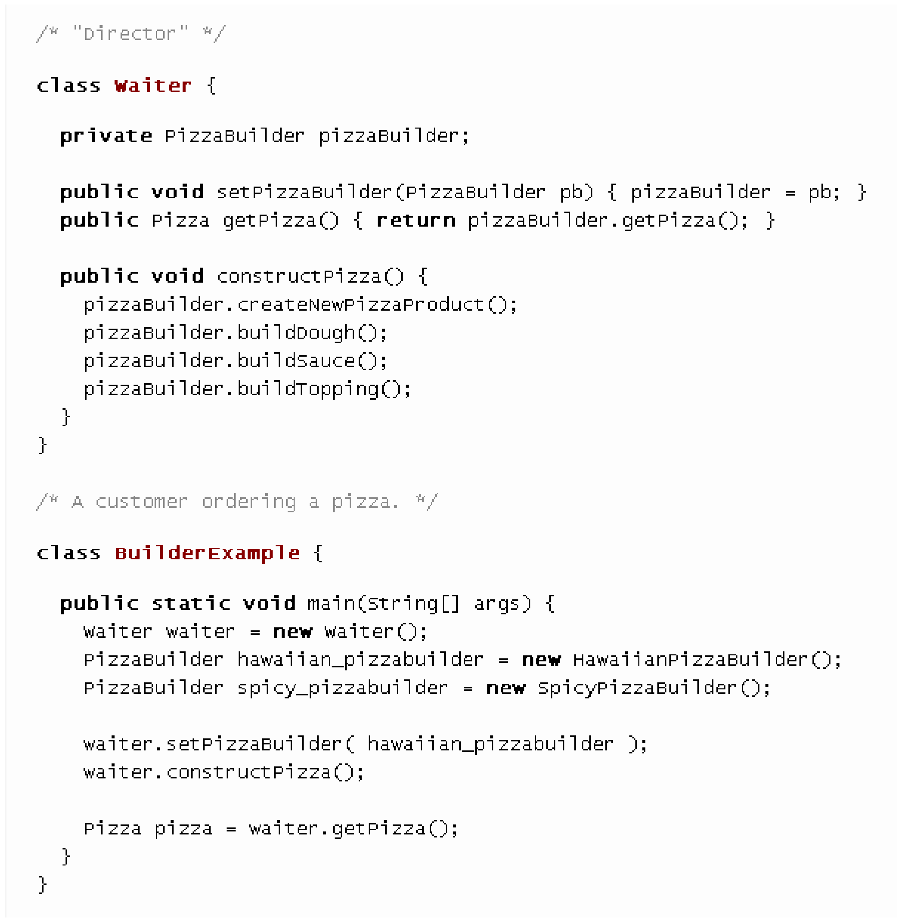
LIYANA ARACHCHIGE RANIL



```
/* "Product" */

class Pizza {

  private String dough = "";
  private String sauce = "";
  private String topping = "";

  public void setDough(String dough)     { this.dough = dough; }
  public void setSauce(String sauce)     { this.sauce = sauce; }
  public void setTopping(String topping) { this.topping = topping; }
}

/* "Abstract Builder" */

abstract class PizzaBuilder {

  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
  public void createNewPizzaProduct() { pizza = new Pizza(); }

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}
```



```
/* "ConcreteBuilder" */

class HawaiianPizzaBuilder extends PizzaBuilder {

  public void buildDough()   { pizza.setDough("cross"); }
  public void buildSauce()   { pizza.setSauce("mild"); }
  public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/* "ConcreteBuilder" */

class SpicyPizzaBuilder extends PizzaBuilder {

  public void buildDough()   { pizza.setDough("pan baked"); }
  public void buildSauce()   { pizza.setSauce("hot"); }
  public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}
```



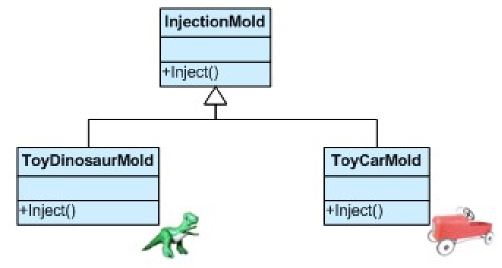
```
/* "Director" */

class Waiter {

  private PizzaBuilder pizzaBuilder;

  public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
  public Pizza getPizza() { return pizzaBuilder.getPizza(); }

  public void constructPizza() {
    pizzaBuilder.createNewPizzaProduct();
    pizzaBuilder.buildDough();
    pizzaBuilder.buildSauce();
    pizzaBuilder.buildTopping();
  }
}

/* A customer ordering a pizza. */

class BuilderExample {

  public static void main(String[] args) {
    Waiter waiter = new Waiter();
    PizzaBuilder hawaiian_pizzabuilder = new HawaiianPizzaBuilder();
    PizzaBuilder spicy_pizzabuilder = new SpicyPizzaBuilder();

    waiter.setPizzaBuilder( hawaiian_pizzabuilder );
    waiter.constructPizza();

    Pizza pizza = waiter.getPizza();
  }
}
```

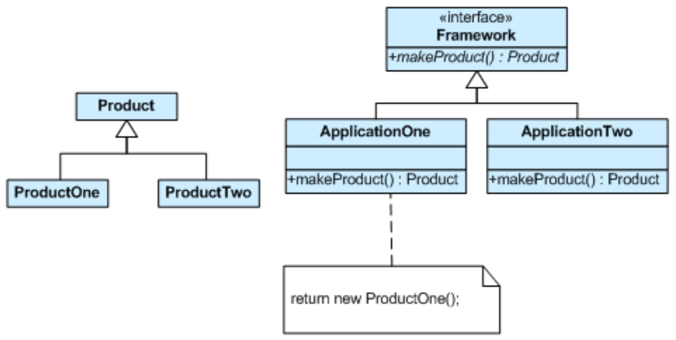
c. Factory Method
   i. Define INTERFACE for creating an object, BUT let SUBCLASS decide which class to INSTANTIATE. Factory method lets a class DEFER instantiation to SUB CLASSES
   ii. The new operator considered harmful
   iii. A FRAMEWORK needs to standardize the architectural model for a range of applications , but allow for individual application to define their own domain objects and provide for their instantiations
   iv. Factory method is to creating objects as Template methods to implement and Algorithm

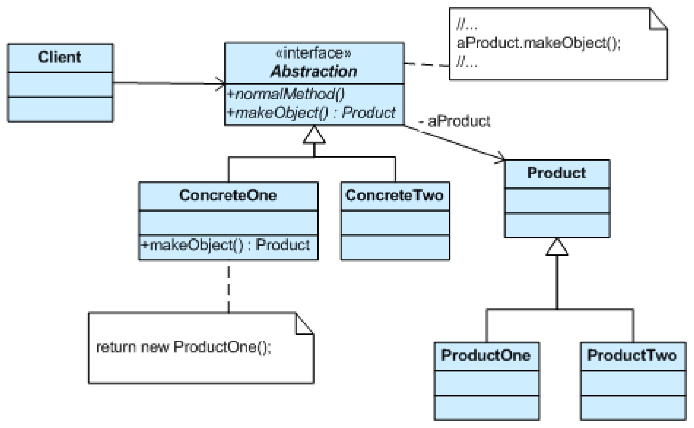
v. Factory method is SIMILAR to ABSTRACT FACTORY , but without EMPHASIS on FAMILIES

vi. Client is totally decoupled from the implementation details of the derived class , Polymorphic creation is now possible

vii. Injection molding process resembles Factory Method

viii. 

ix. 

x. 

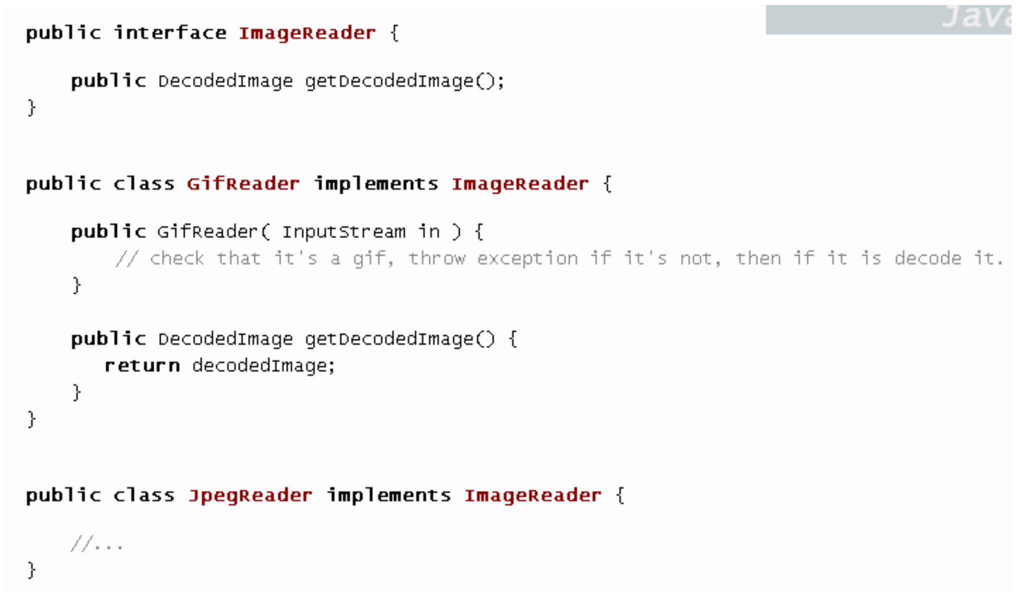
xi. If there is an INHERITANCE HIERACHY that exercise polymorphism , consider adding a polymorphic creation capability by defining a static factory method in the base class

xii. Abstract Factory classes are often implemented with Factory methods , but they can also be implemented using Prototype as well

xiii. Factory Method : creation through inheritance , Prototype : creation through delegation

xiv. Prototype DOES NOT require SUB CLASSING , but it requires INITIALIZING , Factory Method REQUIRES sub classing , but DOES NOT require INITIALIZING

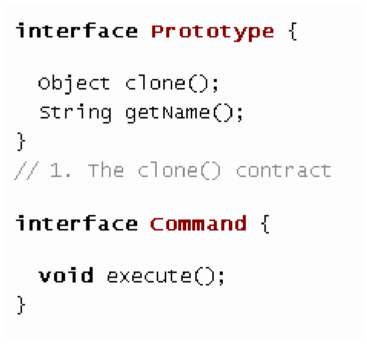
xv.



```
public interface ImageReader {

    public DecodedImage getDecodedImage();
}


public class GifReader implements ImageReader {

    public GifReader( InputStream in ) {
        // check that it's a gif, throw exception if it's not, then if it is decode it.
    }

    public DecodedImage getDecodedImage() {
       return decodedImage;
    }
}


public class JpegReader implements ImageReader {

    //...
}
```

d. Prototype

i. Specify the kinds of objects to create using a prototypical instance, and create new object by copying this prototype
ii. Co-Opt one instance of a class as a breeder for all future instances
iii. The new operator considered harmful
iv. The Factory knows how to find the correct PROTOTYPE , and each RPODUCT knows how to SPAWN new instances of itself
v. Division of a CELL is an example of prototype
vi. Check List
    1. Add a clone method to the existing product hierarchy
    2. design a REGISTRY that maintains a cache of prototypical objects
    3. Design a Factory method that may accept an argument , find the correct prototype and call clone on that object
    4. Prototypes are useful when OBJECT INITIALIZATION are expensive
    5. Prototype does not require a class , it only requires an OBJECT

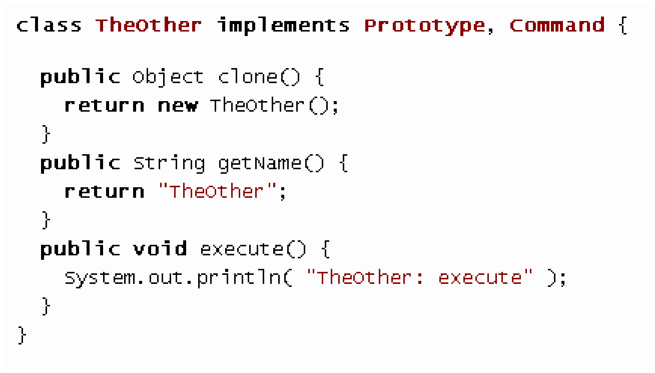


```
interface Prototype {

  Object clone();
  String getName();
}
// 1. The clone() contract

interface Command {

  void execute();
}
```

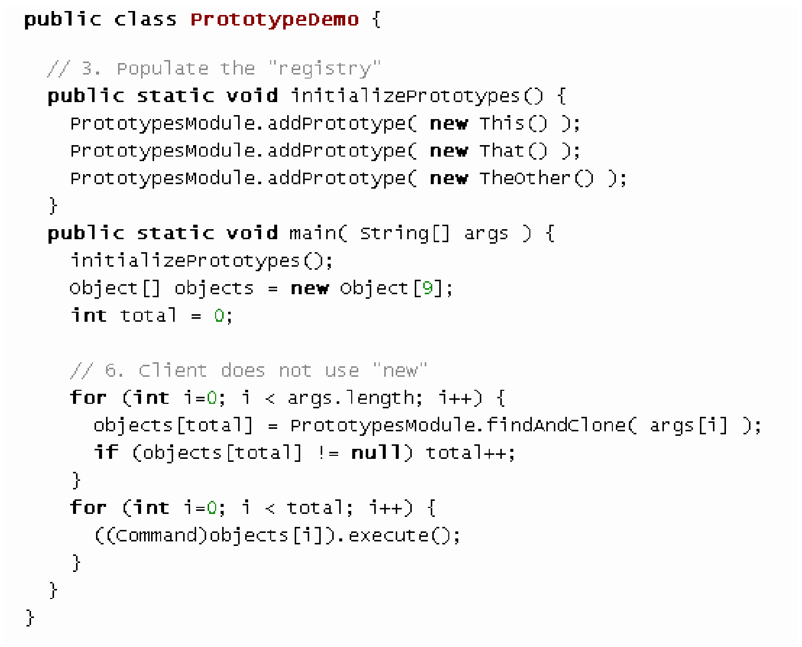
LIYANA ARACHCHIGE RANIL

```
class PrototypesModule {

  // 2. "registry" of prototypical objs
  private static Prototype[] prototypes = new Prototype[9];
  private static int total = 0;

  // Adds a feature to the Prototype attribute of the PrototypesModule class
  // obj  The feature to be added to the Prototype attribute
  public static void addPrototype( Prototype obj ) {
    prototypes[total++] = obj;
  }

  public static Object findAndClone( String name ) {
    // 4. The "virtual ctor"
    for ( int i = 0; i < total; i++ ) {
      if ( prototypes[i].getName().equals( name ) ) {
        return prototypes[i].clone();
      }
    }
    System.out.println( name + " not found" );
    return null;
  }
}
```

```
// 5. Sign-up for the clone() contract.
// Each class calls "new" on itself FOR the client.

class This implements Prototype, Command {

  public Object clone() {
    return new This();
  }
  public String getName() {
    return "This";
  }
  public void execute() {
    System.out.println( "This: execute" );
  }
}


class That implements Prototype, Command {

  public Object clone() {
    return new That();
  }
  public String getName() {
    return "That";
  }
  public void execute() {
    System.out.println( "That: execute" );
  }
}
```



```
class TheOther implements Prototype, Command {

  public Object clone() {
    return new TheOther();
  }
  public String getName() {
    return "TheOther";
  }
  public void execute() {
    System.out.println( "TheOther: execute" );
  }
}
```



```
public class PrototypeDemo {

  // 3. Populate the "registry"
  public static void initializePrototypes() {
    PrototypesModule.addPrototype( new This() );
    PrototypesModule.addPrototype( new That() );
    PrototypesModule.addPrototype( new TheOther() );
  }
  public static void main( String[] args ) {
    initializePrototypes();
    Object[] objects = new Object[9];
    int total = 0;

    // 6. Client does not use "new"
    for (int i=0; i < args.length; i++) {
      objects[total] = PrototypesModule.findAndClone( args[i] );
      if (objects[total] != null) total++;
    }
    for (int i=0; i < total; i++) {
      ((Command)objects[i]).execute();
    }
  }
}
```

e. Singleton

i. Ensure the class has only one INSTANCE , and provide a global point of access to it

ii. Encapsulate the “Just-In-Time initialization” or “Initialization on fist use”

iii. Applications need one and only one instance of an Object. Additionally LAZY initialization and GLOBLE access are necessary

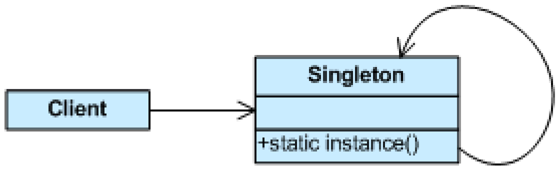
iv. Singleton should be considered only if

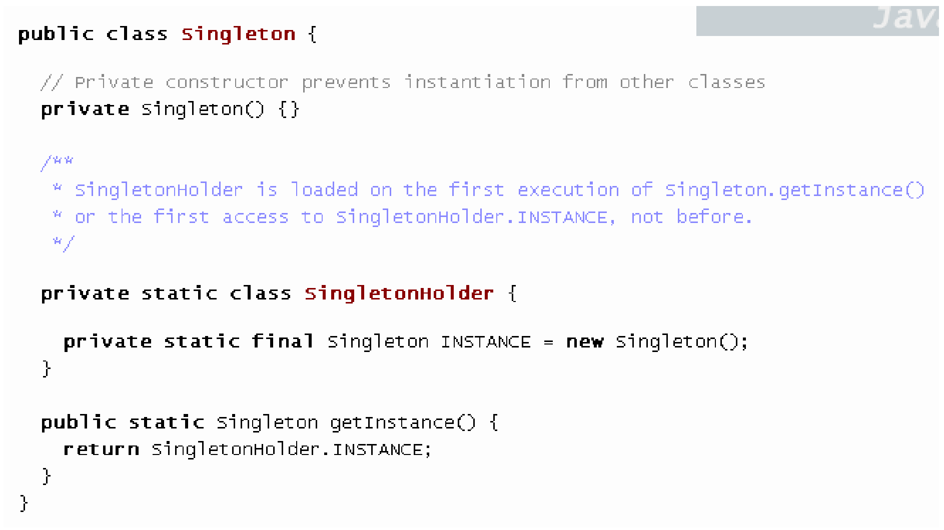
    1. Ownership of the single instance can not be reasonably assigned
    2. Lazy initialization is desirable
    3. Global access is not otherwise provided for

v. Singleton pattern can be extended to support access to an application specific number of instances

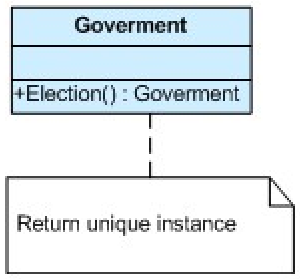
vi. Abstract Factory , Builder , and Prototype can use Singleton in their implementation

vii. Façade objects are often Singleton

viii. 



```
public class Singleton {

  // Private constructor prevents instantiation from other classes
  private Singleton() {}

  /**
   * SingletonHolder is loaded on the first execution of Singleton.getInstance()
   * or the first access to SingletonHolder.INSTANCE, not before.
   */

  private static class SingletonHolder {

    private static final Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```



2. STRUCTURAL PATTERNS

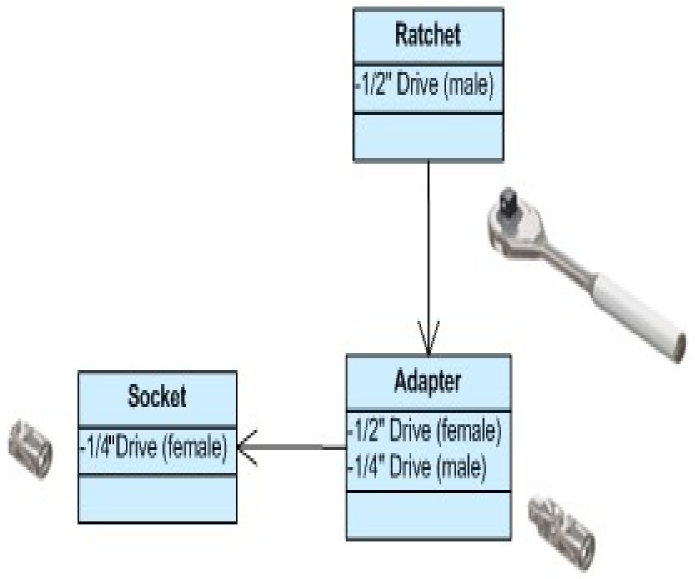
ADAPTOR is walking on the **BRIDGE**; this bridge is made out of **COMPOSITE** material. It is also **DECORATED** with a **FAÇADE**. The façade has **FLYWEIGHTS** attached to it. On the other side of the bridge there is a **PROXY** waiting to meet adaptor

a. Adapter
    i. Convert the interface of a class into another interface that the CLIENT expects, ADAPTOR lets classes work together that could not otherwise because of IMCOMPATIBLE interfaces
    ii. Wrap an EXISTING class with a NEW INTERFACE
    iii. IMPEDENCE mismatch an OLD component to a new system
    iv. An “Off the shelf” component offers compelling functionality that you would like to reuse , but its view of the world is not COMPATIBLE with the PHILOSOPHY and the ARCHITECTURE of the system currently being DEVELOPED
    v. This can be implemented either with the INHERITANCE or with AGGREGATION
    vi. Adaptor makes things work after they are DESIGNED, bridge makes them work BEFORE THEY ARE
    vii. BRIDGE designed upfront to let the ABSTRACTION and IMPLEMENTATION vary independently , ADAPTOR retrofitted to make unrelated classes work together
    viii. ADAPTOR provides a DIFFERENT interface to its client , PROXY provides the SAME , DECORATOR provides and ENHANCED one

ix. ADAPTOR is meant to change the interface of an EXISTING object , Decorator ENHANCES object without changing the interface

x. FAÇADE defines a NEW interface whereas ADAPTOR REUSES an OLD interface

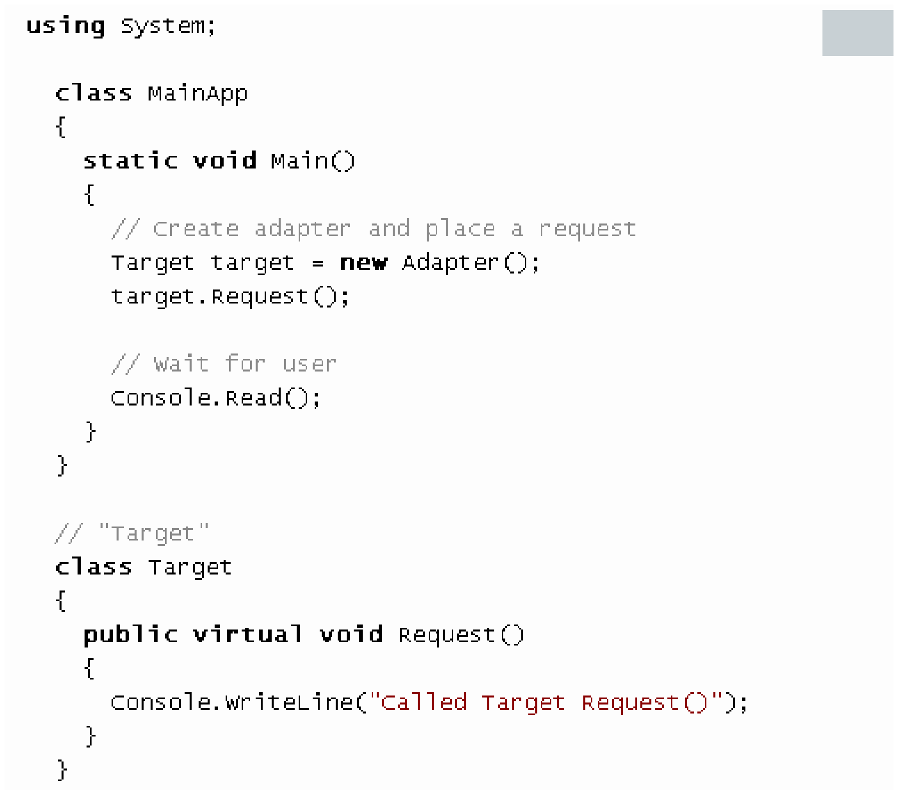
xi. ADAPTOR makes TWO EXISTING interfaces work TOGETHER as opposed to DEFINING NEW one



```
// "Adapter"
class Adapter : Target
{
  private Adaptee adaptee = new Adaptee();

  public override void Request()
  {
    // Possibly do some other work
    // and then call SpecificRequest
    adaptee.SpecificRequest();
  }
}

// "Adaptee"
class Adaptee
{
  public void SpecificRequest()
  {
    Console.WriteLine("Called SpecificRequest()");
  }
}
```

xii.

```
using System;

  class MainApp
  {
    static void Main()
    {
      // Create adapter and place a request
      Target target = new Adapter();
      target.Request();

      // Wait for user
      Console.Read();
    }
  }

  // "Target"
  class Target
  {
    public virtual void Request()
    {
      Console.WriteLine("Called Target Request()");
```



```
using System;

  class MainApp
  {
    static void Main()
    {
      // Create adapter and place a request
      Target target = new Adapter();
      target.Request();

      // Wait for user
      Console.Read();
    }
  }

  // "Target"
  class Target
  {
    public virtual void Request()
    {
      Console.WriteLine("Called Target Request()");
    }
  }
```

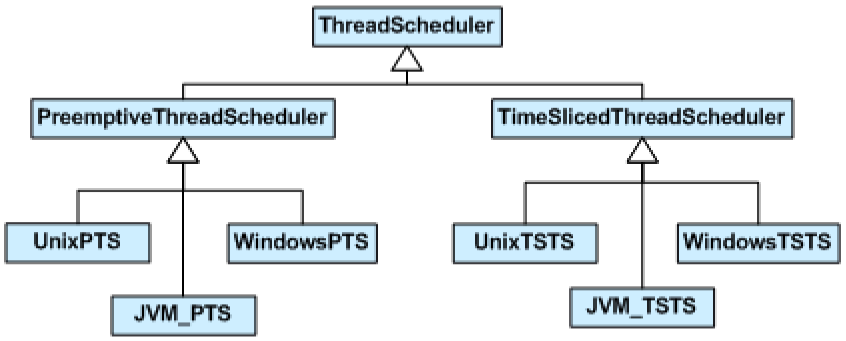
b. Bridge

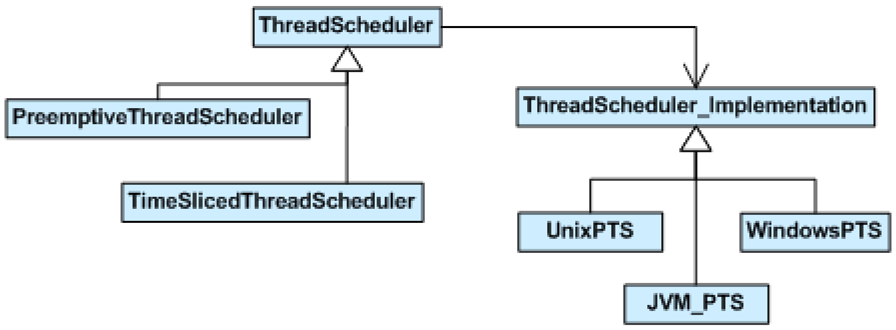
   i. Decouple an ABSTRACTION from its REPRESENTATION so that the two can vary INDEPENDENTLY

   ii. PUBLISH interfaces in an INHERITANCE hierarchy , and BURY IMPLEMENTATION in its own INHERITANCE HIERACHY

   iii. PROBLEM: Hardening of the software arteries has occurred by using sub classing of an ABSTRACT class to provide ALTERNATIVE implementations. This locks in COMPILE TIME binding between INTERFACES and IMPLEMENTATIONS

iv.




ThreadScheduler
PreemptiveThreadScheduler
TimeSlicedThreadScheduler
ThreadScheduler_Implementation
UnixPTS
WindowsPTS
JVM_PTS

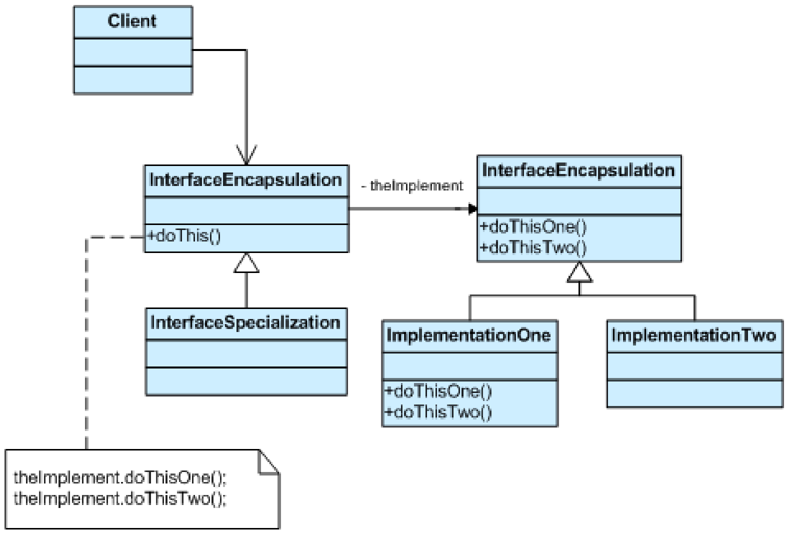
v. The INTERFACE object is the “HANDLE” known and used by the CLIENT , while the IMPLEMENTATION object or “BODY” is safely encapsulated to ensure that it may CONTINUE to EVOLVE or be ENTIRELY REPLACED

vi. USE BRIDGE pattern when

1. Want RUN-TIME binding of the IMPLEMENTATION

2. Have proliferation of CLASSES resulting from a COUPLED interfaces and numerous implementations

vii. Improved EXTENSIBILITY

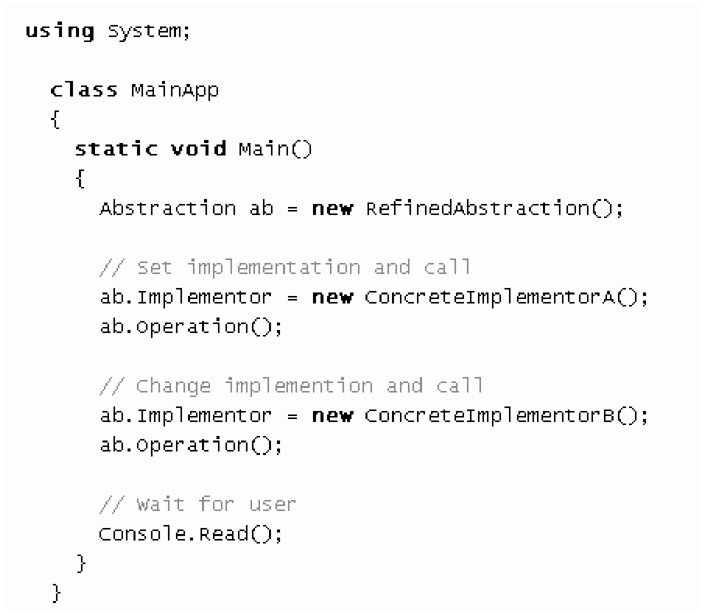


viii.

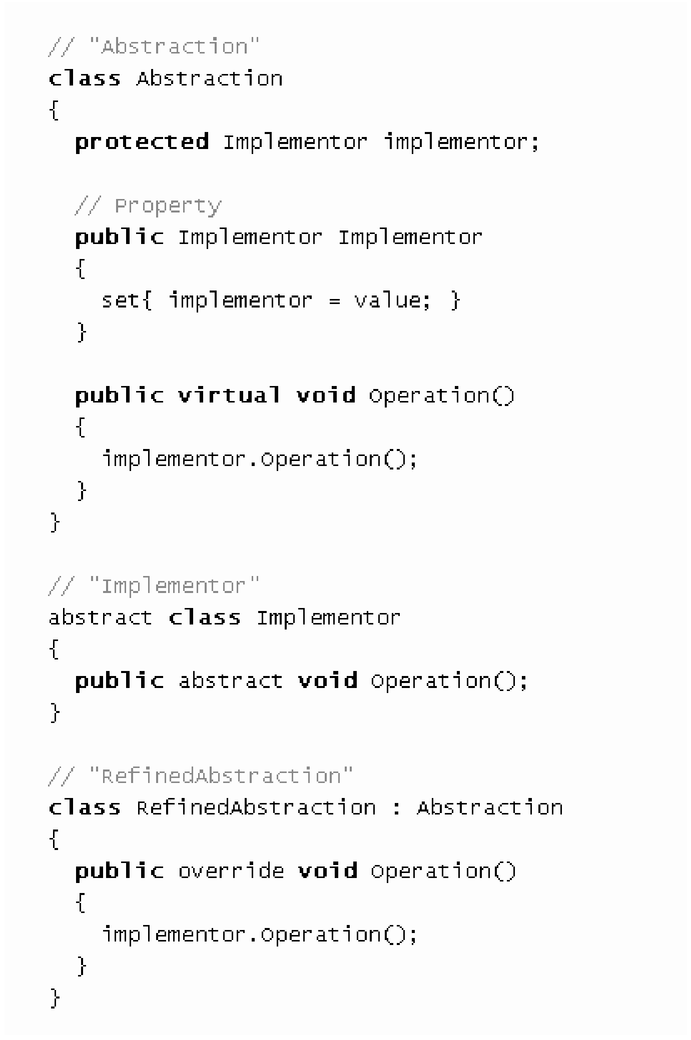
ix. State, Strategy, BRIDGE have similar solution structures, they all share elements of BODY / HANDLE idiom. They DIFFER in INTENT, they resolve DIFFERENT PROBLEMS

x. The STRUCTURE of STATE and BRIDGE are identical. The two patterns use the SAME structure to SOLVE different problems

xi.

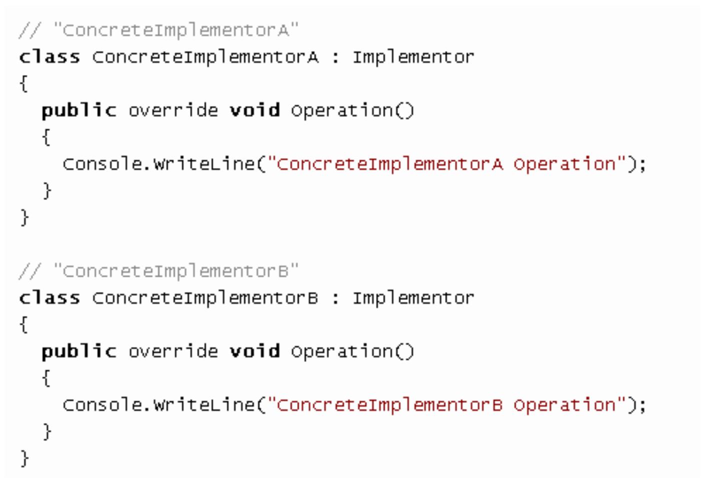


```
using System;

  class MainApp
  {
    static void Main()
    {
      Abstraction ab = new RefinedAbstraction();

      // Set implementation and call
      ab.Implementor = new ConcreteImplementorA();
      ab.Operation();

      // Change implemention and call
      ab.Implementor = new ConcreteImplementorB();
      ab.Operation();

      // Wait for user
      Console.Read();
    }
  }
```



```
// "Abstraction"
class Abstraction
{
  protected Implementor implementor;

  // Property
  public Implementor Implementor
  {
    set{ implementor = value; }
  }

  public virtual void Operation()
  {
    implementor.Operation();
  }
}

// "Implementor"
abstract class Implementor
{
  public abstract void Operation();
}

// "RefinedAbstraction"
class RefinedAbstraction : Abstraction
{
  public override void Operation()
  {
    implementor.Operation();
  }
}
```

xii.

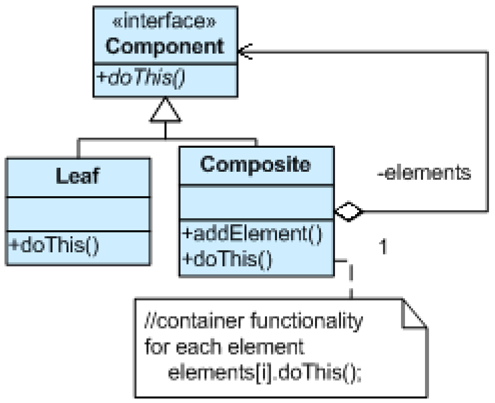


```
// "ConcreteImplementorA"
class ConcreteImplementorA : Implementor
{
  public override void Operation()
  {
    Console.WriteLine("ConcreteImplementorA Operation");
  }
}

// "ConcreteImplementorB"
class ConcreteImplementorB : Implementor
{
  public override void Operation()
  {
    Console.WriteLine("ConcreteImplementorB Operation");
  }
}
```

xiii.

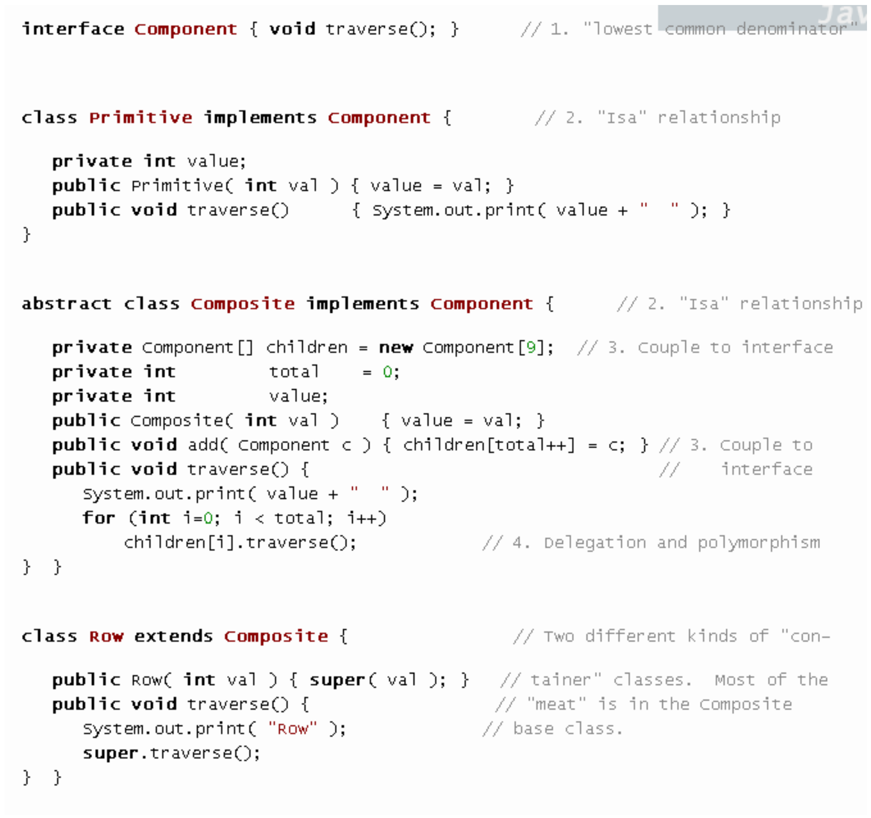
c. Composite

i. Compose objects in to TREE structure to REPRESENT WHOLE/PART hierarchies. COMPOSITE lets clients treat individual objects and COMPOSITIONS of objects UNIFORMLY
ii. Recursive Composition
iii. PROBLEM: Application NEEDS to manipulate a HIERACHICAL collection of “PRIMITIVE” and “COMPOSITE” objects. Processing of PRIMITIVE object is handled one way and processing of COMPOSITE object is handled DIFFERENTLY. Having to QUERY the type of each object before attempting to process is NOT DESIRABLE
iv.



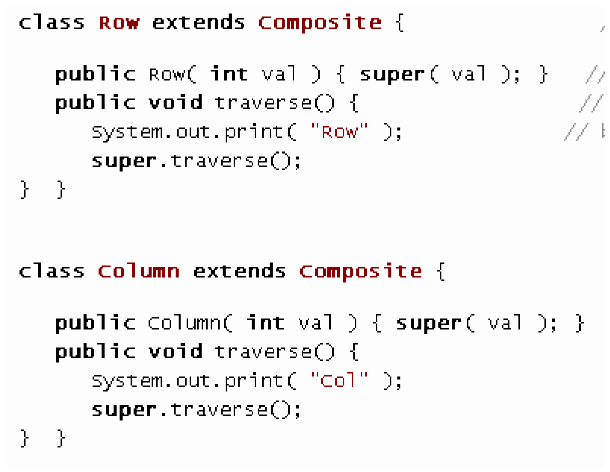
v. Check List
   1. Ensure that the problem is WHOLE/PART hierarchical relationships
   2. Creates a LOWEST COMMON DENOMINATOR interface that makes your containers and containees interchangeable
vi. Composite and Decorator both have SIMILAR structure diagrams

vii. Composite can be traversed with an ITERATOR , VISITOR can apply an operation over a COMPOSITE
viii. DECORATOR is designed to let you add RESPONSIBILITIES to OBJECTS without SUBCLASSING
ix. COMPOSITES focus is NOT on EMBELLISHMENT but on REPRESENTATION
x. The whole point of the COMPOSITE pattern is that the COMPOSITE can be treated ATOMICALLY



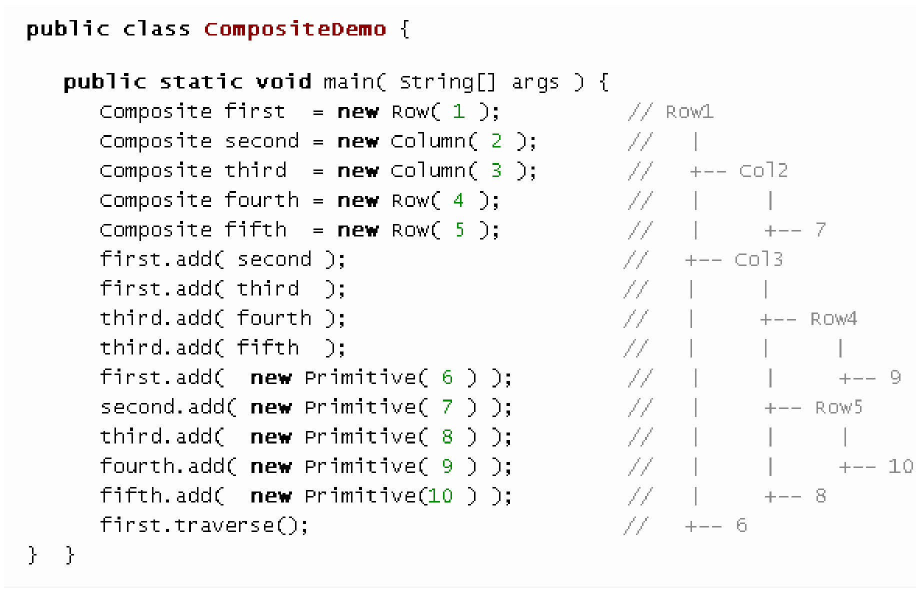
```
interface Component { void traverse(); }      // 1. "lowest common denominator"


class Primitive implements Component {        // 2. "Isa" relationship

   private int value;
   public Primitive( int val ) { value = val; }
   public void traverse()      { System.out.print( value + "  " ); }
}


abstract class Composite implements Component {      // 2. "Isa" relationship

   private Component[] children = new Component[9];  // 3. Couple to interface
   private int         total    = 0;
   private int         value;
   public Composite( int val )    { value = val; }
   public void add( Component c ) { children[total++] = c; } // 3. Couple to
   public void traverse() {                                  //    interface
      System.out.print( value + "  " );
      for (int i=0; i < total; i++)
          children[i].traverse();           // 4. Delegation and polymorphism
}  }


class Row extends Composite {                // Two different kinds of "con-

   public Row( int val ) { super( val ); }   // tainer" classes.  Most of the
   public void traverse() {                  // "meat" is in the Composite
      System.out.print( "Row" );            // base class.
      super.traverse();
}  }
```

xi.



```
class Row extends Composite {                              /

   public Row( int val ) { super( val ); }   //
   public void traverse() {                  //
      System.out.print( "Row" );             // l
      super.traverse();
}  }


class Column extends Composite {

   public Column( int val ) { super( val ); }
   public void traverse() {
      System.out.print( "Col" );
      super.traverse();
}  }
```

xii.



```
public class CompositeDemo {

   public static void main( String[] args ) {
      Composite first  = new Row( 1 );          // Row1
      Composite second = new Column( 2 );       //   |
      Composite third  = new Column( 3 );       //   +-- Col2
      Composite fourth = new Row( 4 );          //   |     |
      Composite fifth  = new Row( 5 );          //   |     +-- 7
      first.add( second );                      //   +-- Col3
      first.add( third  );                      //   |     |
      third.add( fourth );                      //   |     +-- Row4
      third.add( fifth  );                      //   |     |     |
      first.add(  new Primitive( 6 ) );         //   |     |     +-- 9
      second.add( new Primitive( 7 ) );         //   |     +-- Row5
      third.add(  new Primitive( 8 ) );         //   |     |     |
      fourth.add( new Primitive( 9 ) );         //   |     |     +-- 10
      fifth.add(  new Primitive(10 ) );         //   |     +-- 8
      first.traverse();                         //   +-- 6
}  }
```

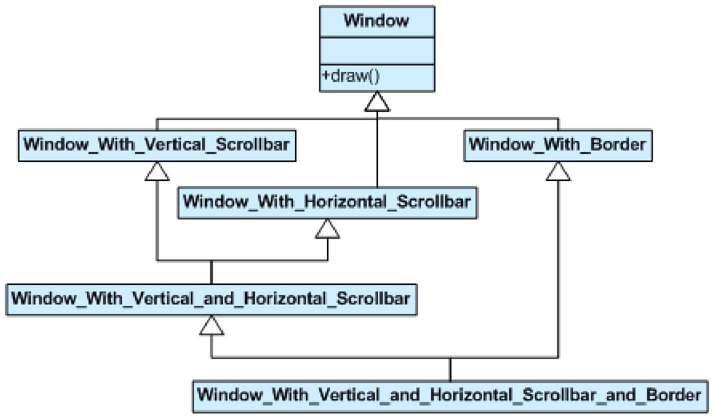
d. Decorator

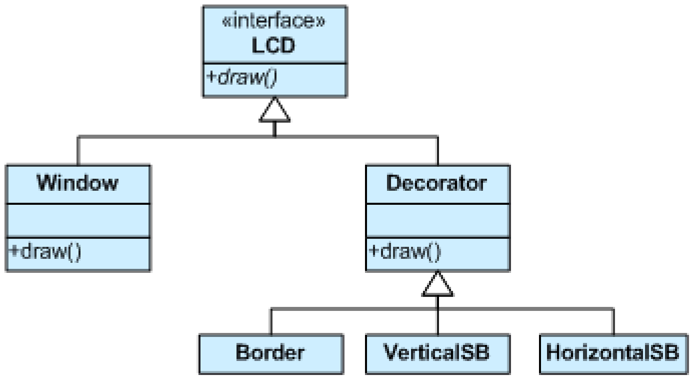
   i. Attach ADDITIONAL RESPONSIBILITIES to an object DYNAMICALLY. DECORATOR provides a FLEXIBLE ALTERNATIVE to SUBCLASSING for extending functionality

   ii. Client specified EMBELISHMENT of a CORE object by recursively wrapping it

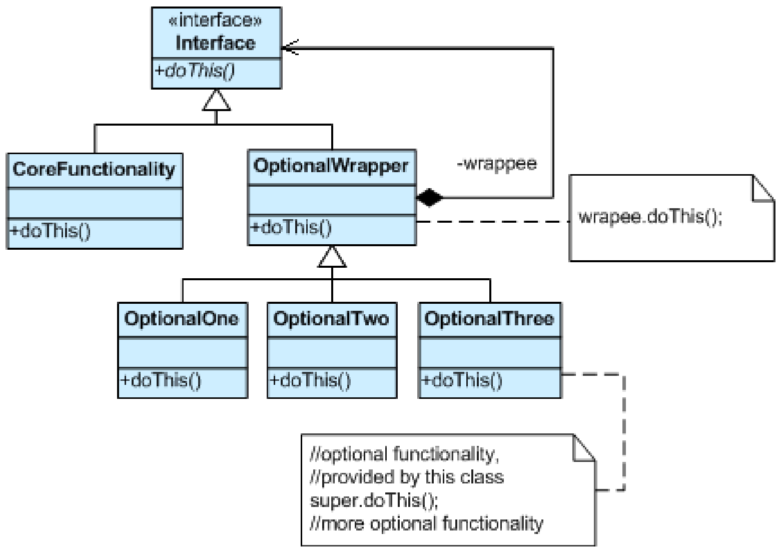
   iii. Wrapping a GIFT , putting it in a BOX , Wrapping the BOX

iv. You want to ADD BEHAVIOURS or STATE to INDIVIDUAL objects at RUN TIME. Inheritance is not FEASEABLE because it is STATIC and APPLIES to an ENTIRE class

v. 
Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Border
Window_With_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border

vi. 

vii. This pattern allows responsibilities to be added to an Object, not METHODS to an Objects interface. The interface presented to the client MUST remain CONSTANT



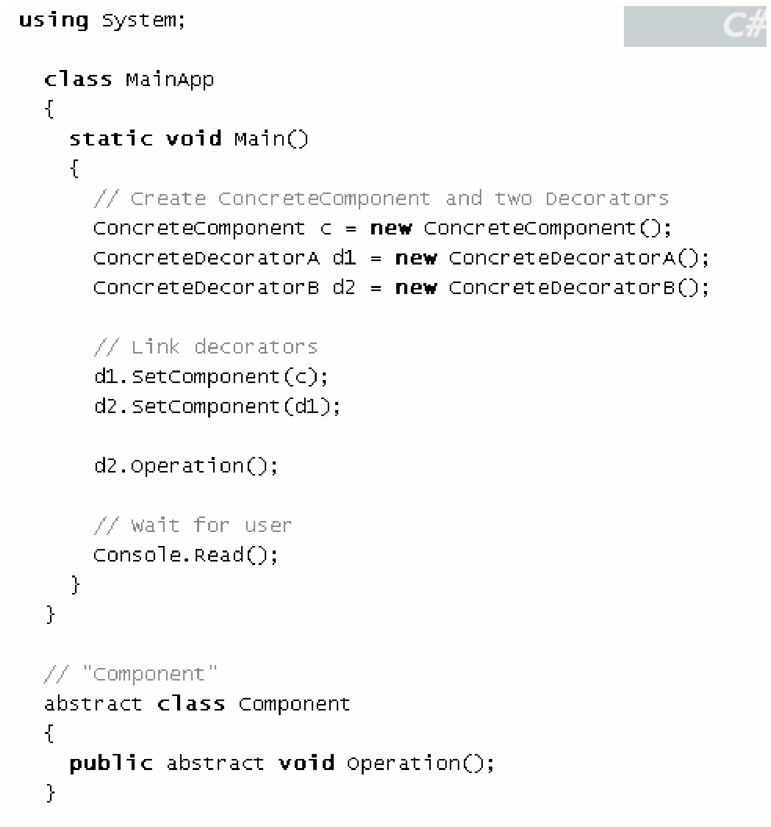
viii.

ix. Decorator provides and ENHANCED interface , Adapter provides a DIFFERENT interface to its subject, proxy provides the SAME interface

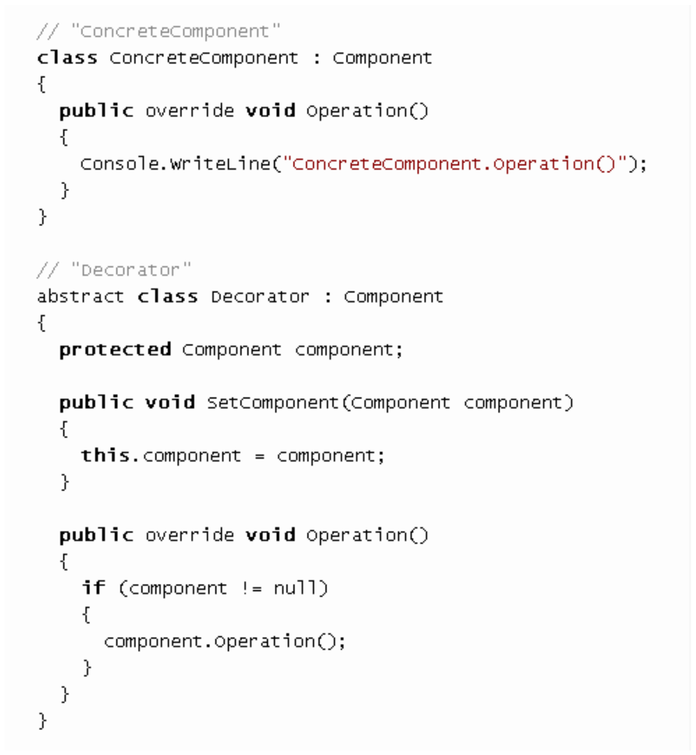
x. Decorator is designed to let you add RESPONSIBILITIES to objects without sub classing

xi. Decorator and Proxy have different purpose , but similar structures. Both describes how to provide level of INDIRECTION to another object

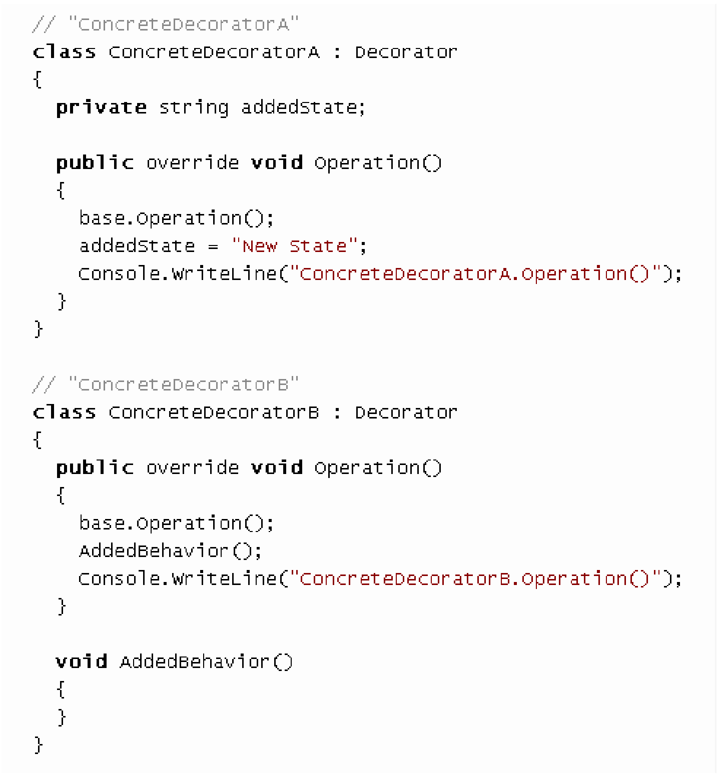
xii. DECORATOR lets you change the SKIN of an object, STRATEGY lets you change the GUTS

xiii. 

```
using System;

 class MainApp
 {
   static void Main()
   {
     // Create ConcreteComponent and two Decorators
     ConcreteComponent c = new ConcreteComponent();
     ConcreteDecoratorA d1 = new ConcreteDecoratorA();
     ConcreteDecoratorB d2 = new ConcreteDecoratorB();

     // Link decorators
     d1.SetComponent(c);
     d2.SetComponent(d1);

     d2.Operation();

     // Wait for user
     Console.Read();
   }
 }

 // "Component"
 abstract class Component
 {
   public abstract void Operation();
 }
```

xiv. 

```
// "ConcreteComponent"
class ConcreteComponent : Component
{
  public override void Operation()
  {
    Console.WriteLine("ConcreteComponent.Operation()");
  }
}

// "Decorator"
abstract class Decorator : Component
{
  protected Component component;

  public void SetComponent(Component component)
  {
    this.component = component;
  }

  public override void Operation()
  {
    if (component != null)
    {
      component.Operation();
    }
  }
}
```

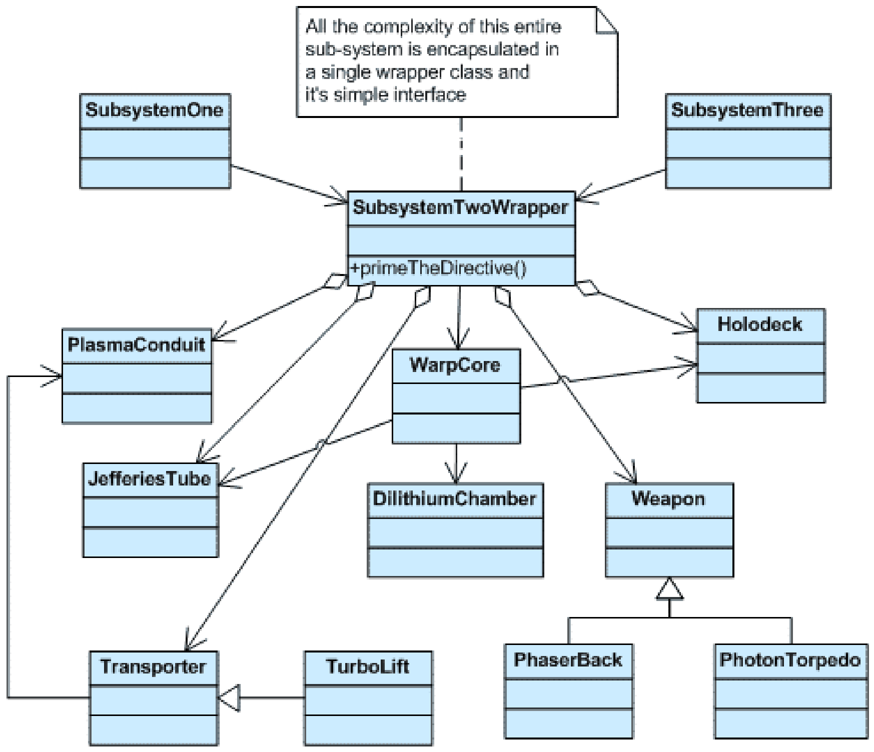
xv. 

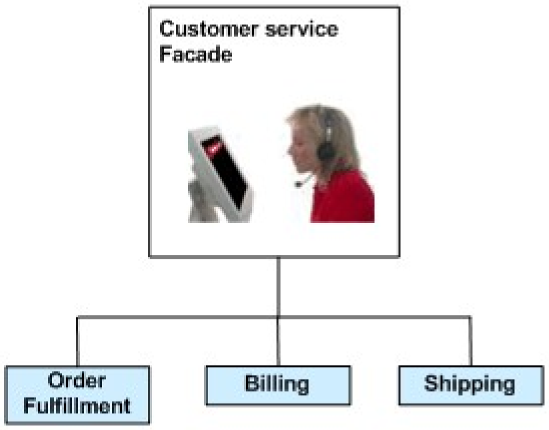
```
// "ConcreteDecoratorA"
class ConcreteDecoratorA : Decorator
{
  private string addedState;

  public override void Operation()
  {
    base.Operation();
    addedState = "New State";
    Console.WriteLine("ConcreteDecoratorA.Operation()");
  }
}

// "ConcreteDecoratorB"
class ConcreteDecoratorB : Decorator
{
  public override void Operation()
  {
    base.Operation();
    AddedBehavior();
    Console.WriteLine("ConcreteDecoratorB.Operation()");
  }

  void AddedBehavior()
  {
  }
}
```

e. Façade
   i. Provides a unified interface to a set of interfaces in a SUBSYSTEM
   ii. Façade defines a HIGHER level interface that makes the SBSYSTEM easier to use
   iii. Wrap a COMPLICATED sub system with a SIMPLER interface
   iv. PROBLEM : A segment of the client community needs a SIMPLIFIED interface to the overall FUNCTIONALITY of a COMPLEX sub system
   v. Façade discusses ENCAPSULATIN a complex subsystem within a SINGLE interface object. This promotes DECOUPLING the SUB SYSTEM from its potentially many clients

vi. Façade should be FAIRLY SIMPLE advocate or facilitator , it should not become an ALL-KNOWING ORACLE or GOD object

vii.

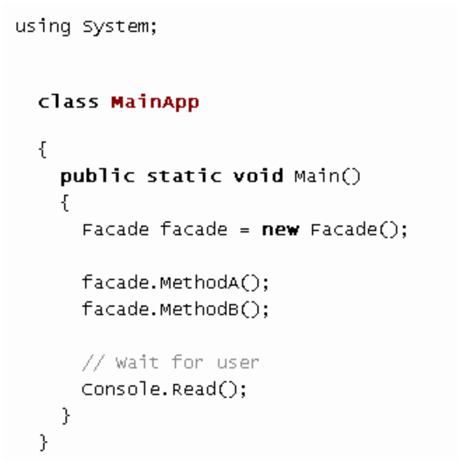




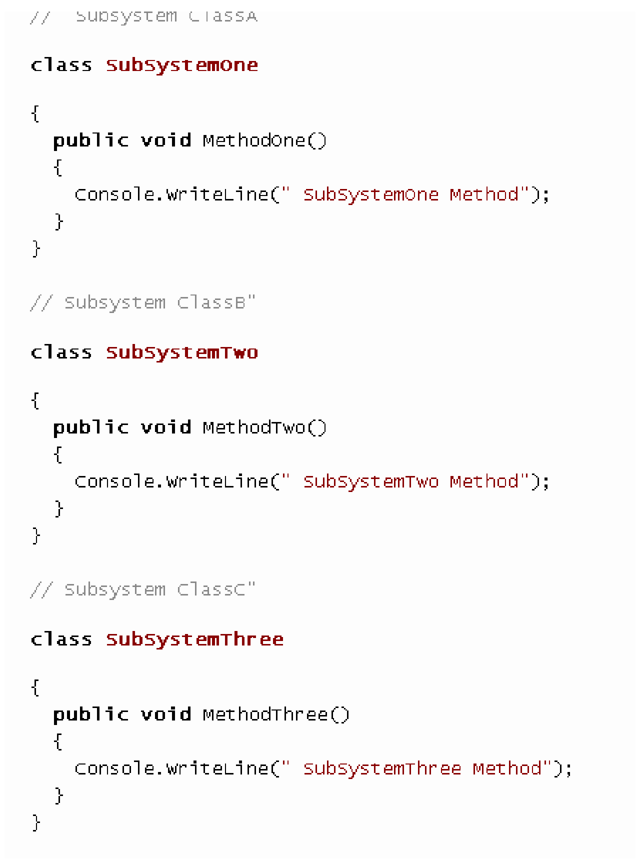
viii.

ix. Façade defines a NEW INTERFACE, whereas ADAPTER uses and OLD interface.

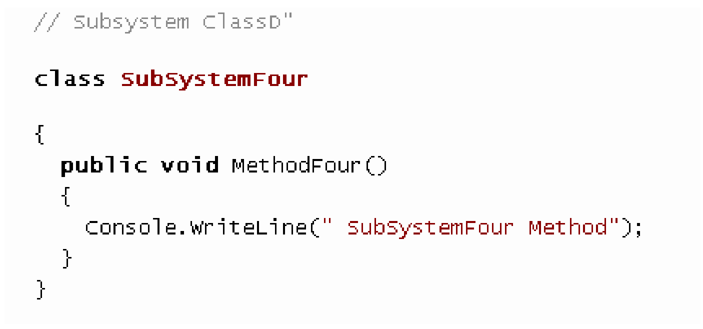
x. Façade objects are often SINGLETONS , since only one Object is needed

xi. 

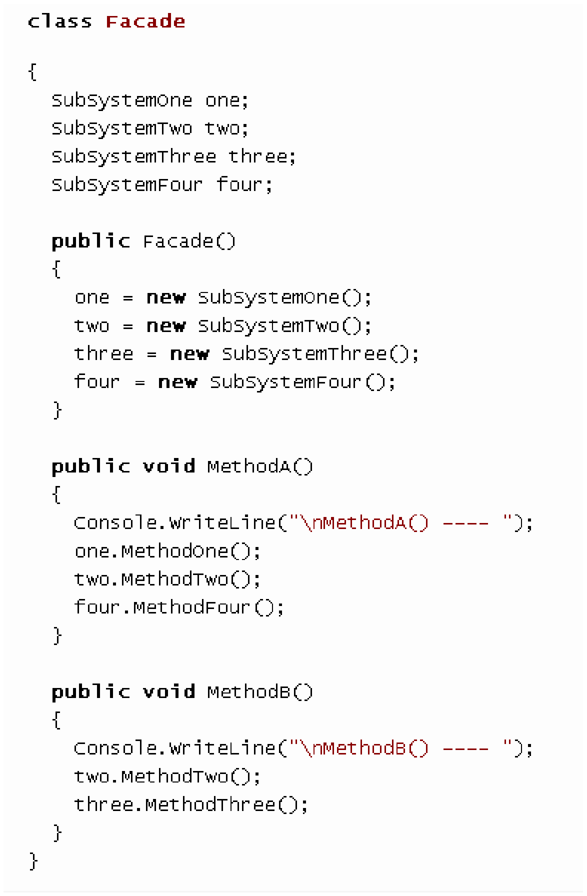
```
using System;

  class MainApp

  {
    public static void Main()
    {
      Facade facade = new Facade();

      facade.MethodA();
      facade.MethodB();

      // Wait for user
      Console.Read();
    }
  }
```

xii. 

```
//  Subsystem ClassA

class SubSystemOne

{
  public void MethodOne()
  {
    Console.WriteLine(" SubSystemOne Method");
  }
}

// Subsystem ClassB"

class SubSystemTwo

{
  public void MethodTwo()
  {
    Console.WriteLine(" SubSystemTwo Method");
  }
}

// Subsystem ClassC"

class SubSystemThree

{
  public void MethodThree()
  {
    Console.WriteLine(" SubSystemThree Method");
  }
}
```

xiii. 

```
// Subsystem ClassD"

class SubSystemFour

{
  public void MethodFour()
  {
    Console.WriteLine(" SubSystemFour Method");
  }
}
```

xiv. 

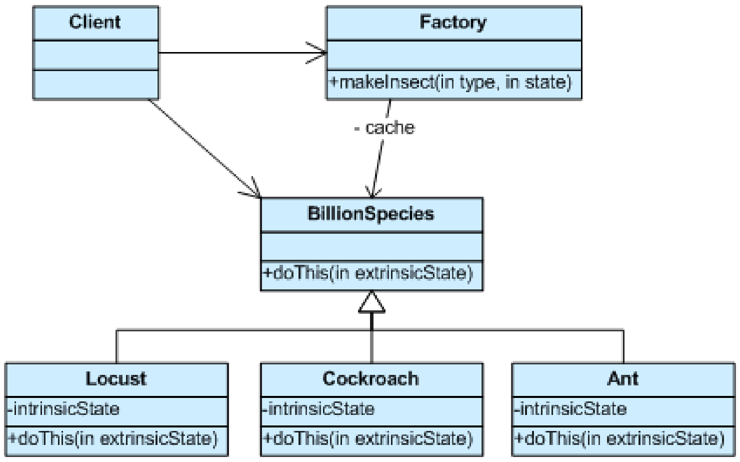
```
class Facade

{
  SubSystemOne one;
  SubSystemTwo two;
  SubSystemThree three;
  SubSystemFour four;

  public Facade()
  {
    one = new SubSystemOne();
    two = new SubSystemTwo();
    three = new SubSystemThree();
    four = new SubSystemFour();
  }

  public void MethodA()
  {
    Console.WriteLine("\nMethodA() ---- ");
    one.MethodOne();
    two.MethodTwo();
    four.MethodFour();
  }

  public void MethodB()
  {
    Console.WriteLine("\nMethodB() ---- ");
    two.MethodTwo();
    three.MethodThree();
  }
}
```

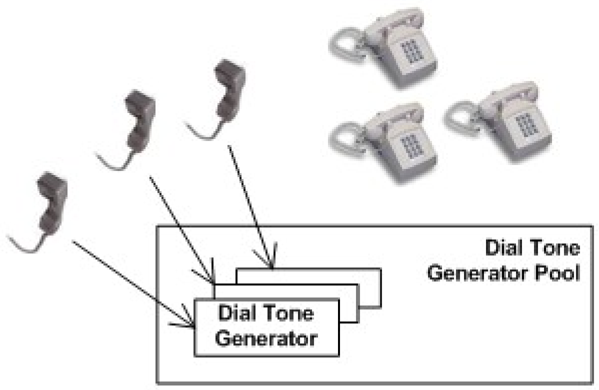
f. Flyweight
   i. Use sharing to support LARGE NUMBER of FINE-GRAINED objects efficiently
   ii. The MOTIF GUI strategy of REPLACING heavy-weight WIDGETS with light-weight GADGETS
   iii. PROBLEM : designing objects down to the lowest levels of system “GRANULARITY” provides optimal “FLEXIBILITY’, but can be unacceptably EXPENSIVE in terms of PERFORMANCE and MEMORY usage

iv. Each FLYWEIGHT object is divided into two pieces, STATE-DEPENDENT (EXTRINSIC) , STATE-INDEPENDENT (INSTRINSIC)

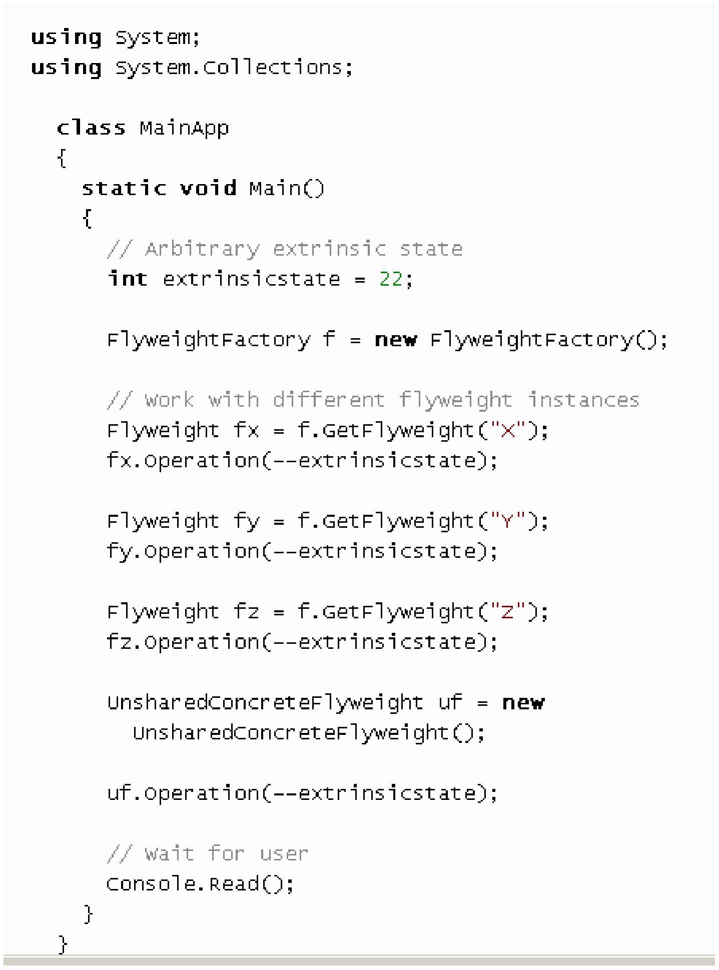
v. INTRINSIC state is stored in the FLYWEIGHT object , EXTRINSIC state is stored or computed by the CLIENT

vi. FLYWEIGHTS are stored in a FACTORY'S REPOSITORY, the client restrain herself creating Flyweight directly

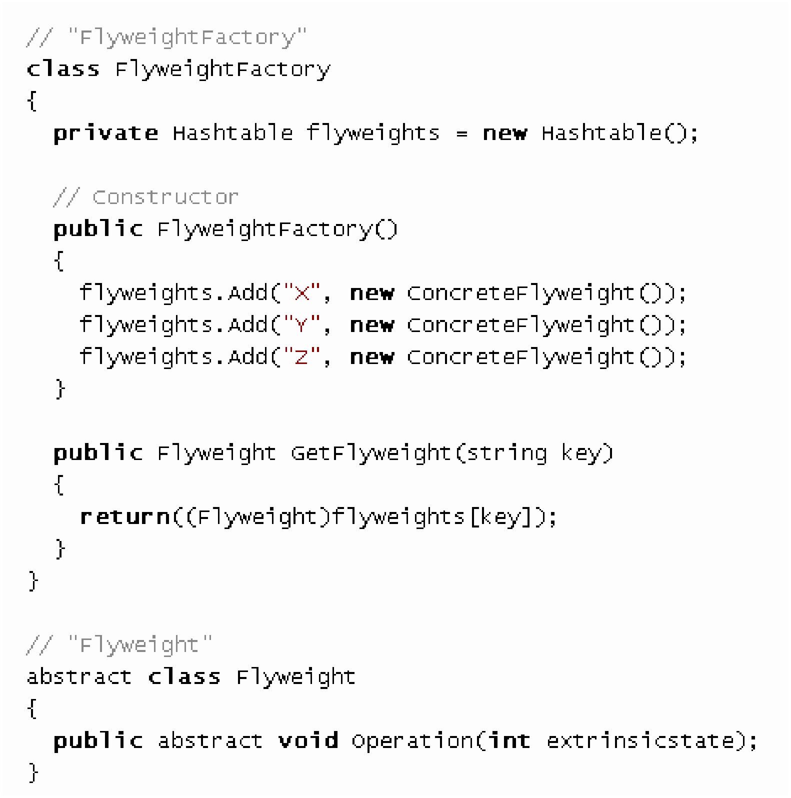
vii. 
Client
Factory
+makeInsect(in type, in state)
- cache
BillionSpecies
+doThis(in extrinsicState)
Locust
-intrinsicState
+doThis(in extrinsicState)
Cockroach
-intrinsicState
+doThis(in extrinsicState)
Ant
-intrinsicState
+doThis(in extrinsicState)



viii. Create a FACTORY that can cache and reuse existing class instances

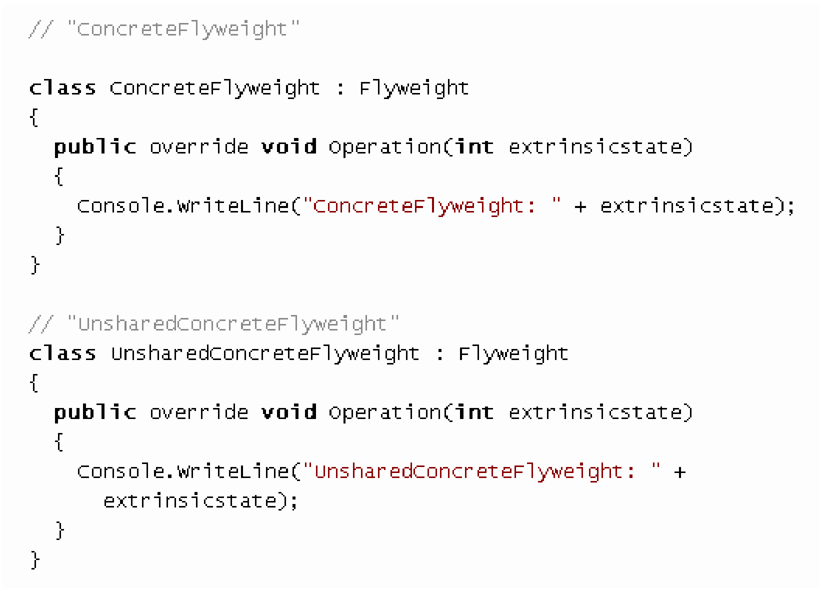
ix. 

```
using System;
using System.Collections;

  class MainApp
  {
    static void Main()
    {
      // Arbitrary extrinsic state
      int extrinsicstate = 22;

      FlyweightFactory f = new FlyweightFactory();

      // Work with different flyweight instances
      Flyweight fx = f.GetFlyweight("X");
      fx.Operation(--extrinsicstate);

      Flyweight fy = f.GetFlyweight("Y");
      fy.Operation(--extrinsicstate);

      Flyweight fz = f.GetFlyweight("Z");
      fz.Operation(--extrinsicstate);

      UnsharedConcreteFlyweight uf = new
        UnsharedConcreteFlyweight();

      uf.Operation(--extrinsicstate);

      // Wait for user
      Console.Read();
    }
  }
```



```
// "FlyweightFactory"
class FlyweightFactory
{
  private Hashtable flyweights = new Hashtable();

  // Constructor
  public FlyweightFactory()
  {
    flyweights.Add("X", new ConcreteFlyweight());
    flyweights.Add("Y", new ConcreteFlyweight());
    flyweights.Add("Z", new ConcreteFlyweight());
  }

  public Flyweight GetFlyweight(string key)
  {
    return((Flyweight)flyweights[key]);
  }
}

// "Flyweight"
abstract class Flyweight
{
  public abstract void Operation(int extrinsicstate);
}
```

x.

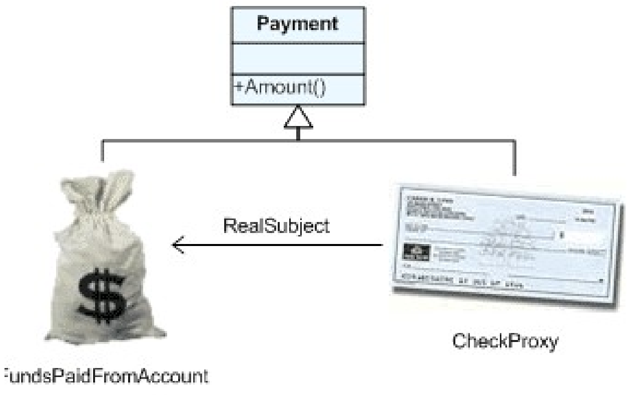


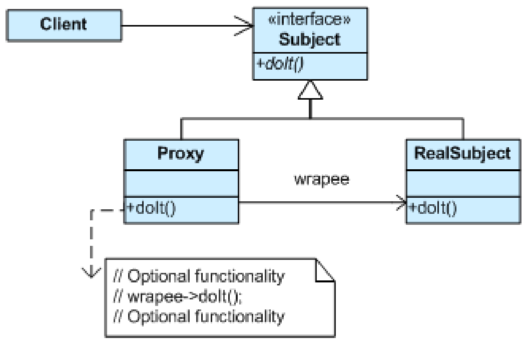
```
// "ConcreteFlyweight"

class ConcreteFlyweight : Flyweight
{
  public override void Operation(int extrinsicstate)
  {
    Console.WriteLine("ConcreteFlyweight: " + extrinsicstate);
  }
}

// "UnsharedConcreteFlyweight"
class UnsharedConcreteFlyweight : Flyweight
{
  public override void Operation(int extrinsicstate)
  {
    Console.WriteLine("UnsharedConcreteFlyweight: " +
      extrinsicstate);
  }
}
```

g. Proxy

i. Provides a SURROGATE or PLACEHOLDER for another object to CONTROL access to it

ii. Use an EXTRA LEVEL of INDIRECTION to support distributed , controlled or intelligent access
iii. Add a wrapper and delegation to protect real component from undue complexity
iv. PROBLEM : Need to support RESOURCE HUNGRY object and do not want to INSTANTIATE such objects unless and until they are actually REQUESTED by the CLIENT
v. There are FOUR common situations in which the PROXY pattern in applicable
    1. A VITUAL PROXY : a place holder for EXPENSIVE to CREATE objects
    2. A REMOTE PROXY : provides a LOCAL REPRESENTATIVE for an object that resides in a different ADDRESS SPACE, this is what the STUB code in RPC and CORBA provides
    3. A PROTECTIVE PROXY: controls the access to a SENSITIVE MATTER object. The surrogate object checks that the caller has the access permissions required prior to forwarding the request
    4. A SMART PROXY: interposes ADDITIONAL actions when an object is accessed. Typically CONTROLLING the NUMBER OF REFERENCE to the real object so that when there are no references it can be FREED, LOADING a PERSISTANCE object into MEMORY when its first accessed , checking that the real object is LOCKED before it is accessed to ensure that no other object can CHANGE it

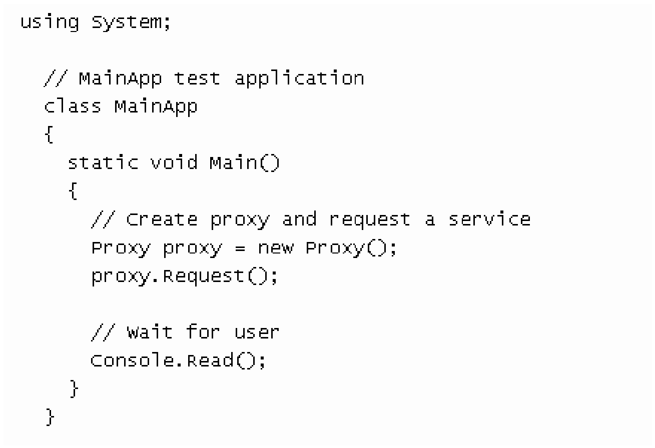
vi. 



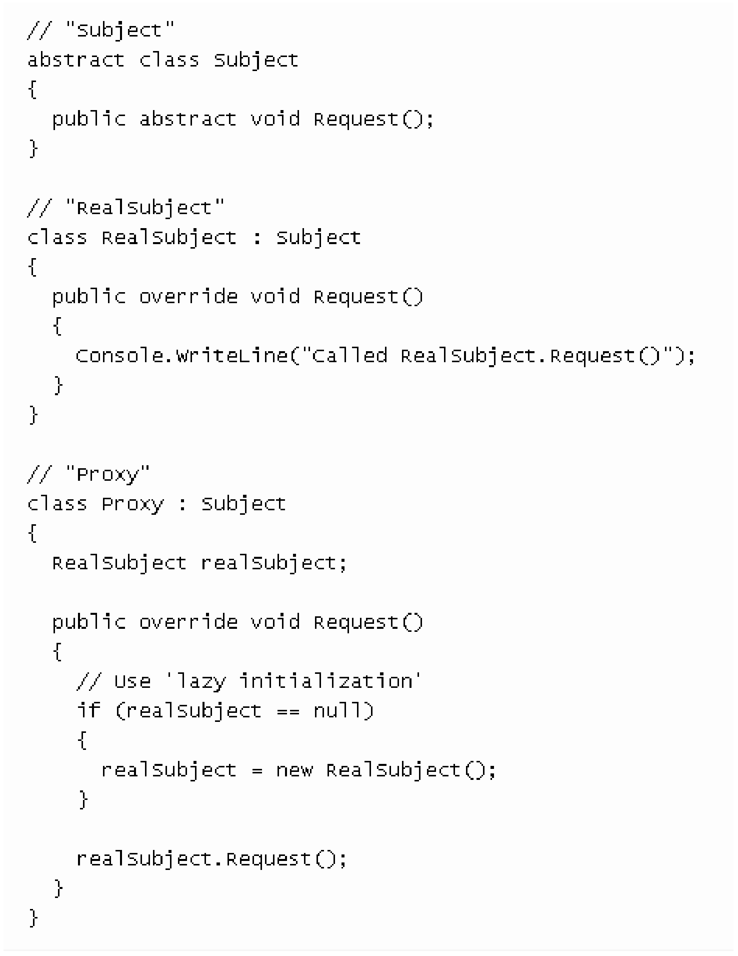
vii. CHECK LIST

1. Define an INTERFAC that will make the PROXY and the ORIGINAL component interchangeable
2. Consider defining a FACTORY that can encapsulate the decision of whether a PROXY or ORIGINAL object is desirable
3. The wrapper class holds a POINTER to the REAL class and implements the interface
4. The pointer may be initialized at the CONSTRUCTION ot on FIRST USE

viii. Decorator and Proxy have different purposes, but similar structures. Both describes how to provide a level of indirection to another object

ix. 

```
using System;

  // MainApp test application
  class MainApp
  {
    static void Main()
    {
      // Create proxy and request a service
      Proxy proxy = new Proxy();
      proxy.Request();

      // Wait for user
      Console.Read();
    }
  }
```

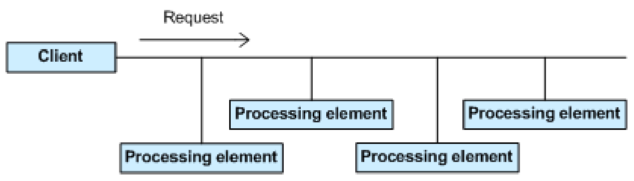
x. 

```
// "Subject"
abstract class Subject
{
  public abstract void Request();
}

// "RealSubject"
class RealSubject : Subject
{
  public override void Request()
  {
    Console.WriteLine("Called RealSubject.Request()");
  }
}

// "Proxy"
class Proxy : Subject
{
  RealSubject realSubject;

  public override void Request()
  {
    // Use 'lazy initialization'
    if (realSubject == null)
    {
      realSubject = new RealSubject();
    }

    realSubject.Request();
  }
}
```

3. BEHAVIORAL PATTERNS

As a boss I have a **RESPONSIBILITY** to **COMMAND** my **INTERPRETER** to **MEADIATE** my interviews. He should **OBSERVE** all the candidates and keep their **STATE** in his **MEMORY**. Later he should **ITERATE** over all the candidate results using an appropriate **STRATEGY** and prepare a **TEMPLATE** along with a **VISITOR**

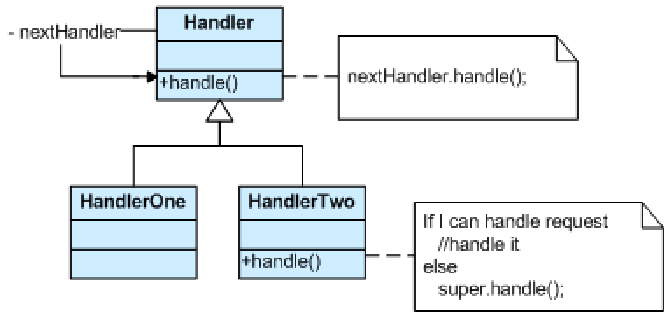
a. Chain of Responsibility
   i. Avoid coupling the sender of a request to its receiver by giving MORE THAN ONE OBJECT to HANDLE the request.
   ii. CHAIN the RECEIVING objects and PASS the REQUEST along the CHAIN until and object HADLES it
   iii. Single processing PIPELINE with many possible HANDLERS
   iv. PROBLEM : There is a potentially variable number of “HANDLERS ” or PROCESSING ELEMENTS or NODE objects and a STREAM of REQUESTS that must be handled
   v. Need to efficiently process the requests without HARD WIRING handler relationships and precedence



vi.

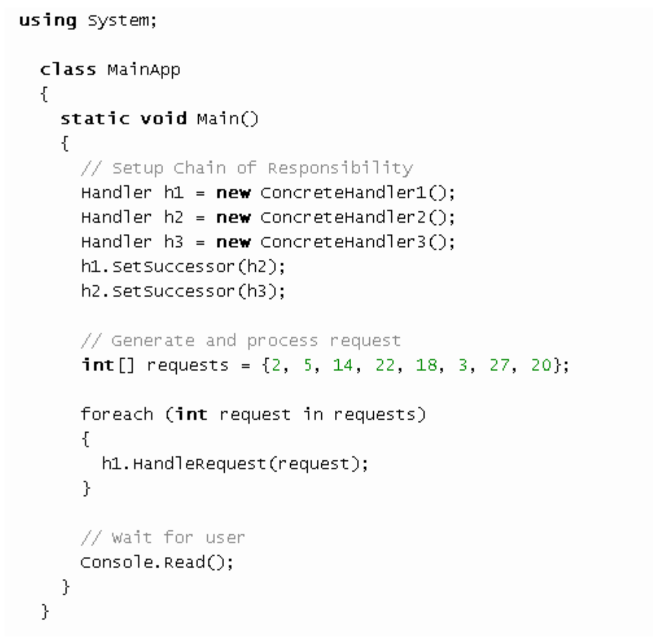
vii. Chain of Responsibility SIMPLIFIED OBJECT INTERCONNECTION

viii. Do not USE Chain of responsibility when each request is ONLY handled by ONE HANDLER or when the CLIENTS knows which SERVICE OBJECT should handle the request

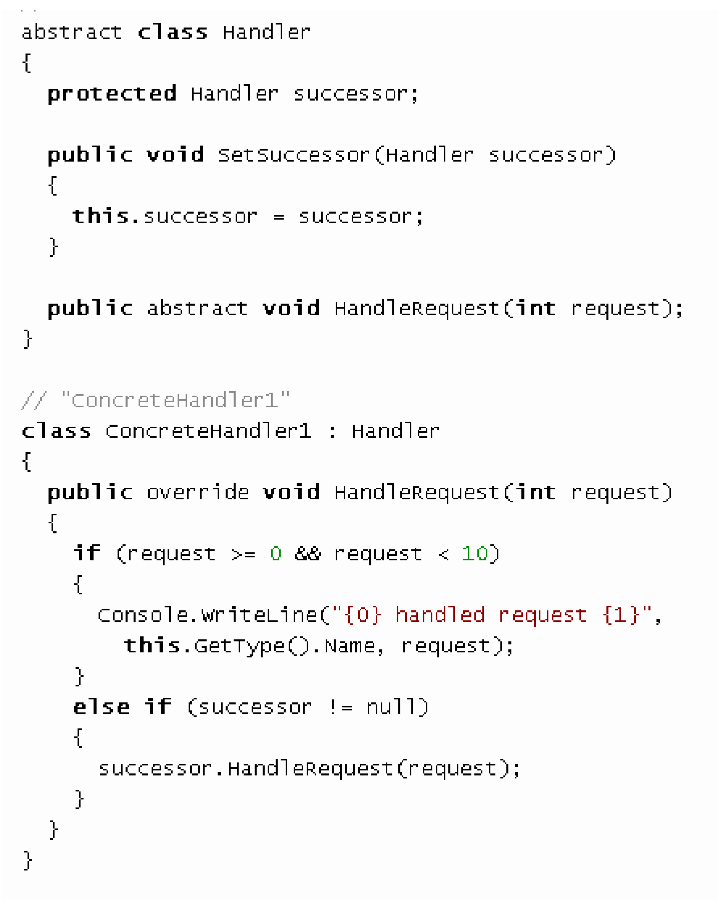


ix.

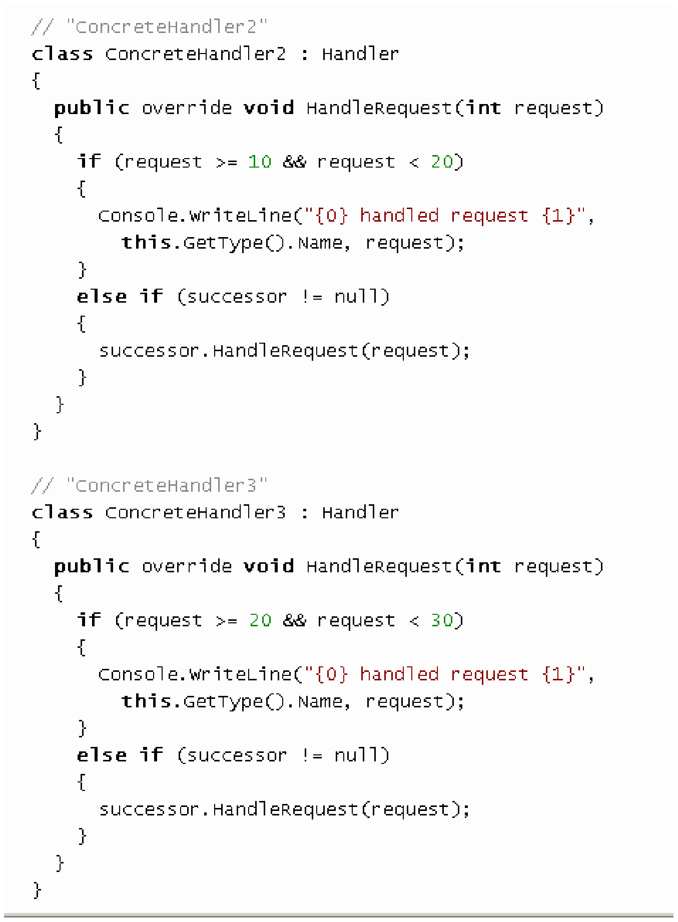
x. Chain of Responsibility, Command , Mediator and Observer address how you can DECOUPLE SENDERS and RECEIVERS

xi. 

```
using System;

  class MainApp
  {
    static void Main()
    {
      // Setup Chain of Responsibility
      Handler h1 = new ConcreteHandler1();
      Handler h2 = new ConcreteHandler2();
      Handler h3 = new ConcreteHandler3();
      h1.SetSuccessor(h2);
      h2.SetSuccessor(h3);

      // Generate and process request
      int[] requests = {2, 5, 14, 22, 18, 3, 27, 20};

      foreach (int request in requests)
      {
        h1.HandleRequest(request);
      }

      // Wait for user
      Console.Read();
    }
  }
```

xii. 

```
abstract class Handler
{
  protected Handler successor;

  public void SetSuccessor(Handler successor)
  {
    this.successor = successor;
  }

  public abstract void HandleRequest(int request);
}

// "ConcreteHandler1"
class ConcreteHandler1 : Handler
{
  public override void HandleRequest(int request)
  {
    if (request >= 0 && request < 10)
    {
      Console.WriteLine("{0} handled request {1}",
        this.GetType().Name, request);
    }
    else if (successor != null)
    {
      successor.HandleRequest(request);
    }
  }
}
```

xiii. 

```
// "ConcreteHandler2"
class ConcreteHandler2 : Handler
{
  public override void HandleRequest(int request)
  {
    if (request >= 10 && request < 20)
    {
      Console.WriteLine("{0} handled request {1}",
        this.GetType().Name, request);
    }
    else if (successor != null)
    {
      successor.HandleRequest(request);
    }
  }
}

// "ConcreteHandler3"
class ConcreteHandler3 : Handler
{
  public override void HandleRequest(int request)
  {
    if (request >= 20 && request < 30)
    {
      Console.WriteLine("{0} handled request {1}",
        this.GetType().Name, request);
    }
    else if (successor != null)
    {
      successor.HandleRequest(request);
    }
  }
}
```

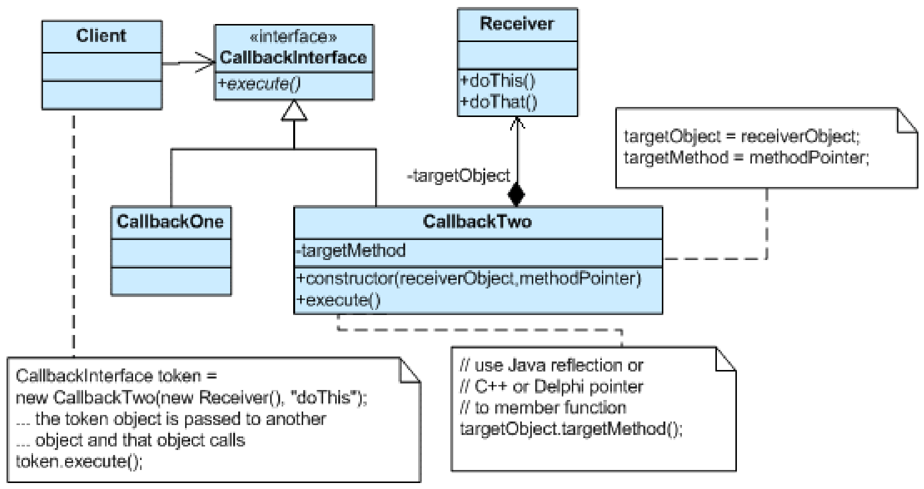
b. Command

i. Encapsulate a request as an Object ,thereby letting you PARAMETERIZE clients with DIFFERENT REQUESTS , QUEUE of LOG requests and SUPPORT undoable operations

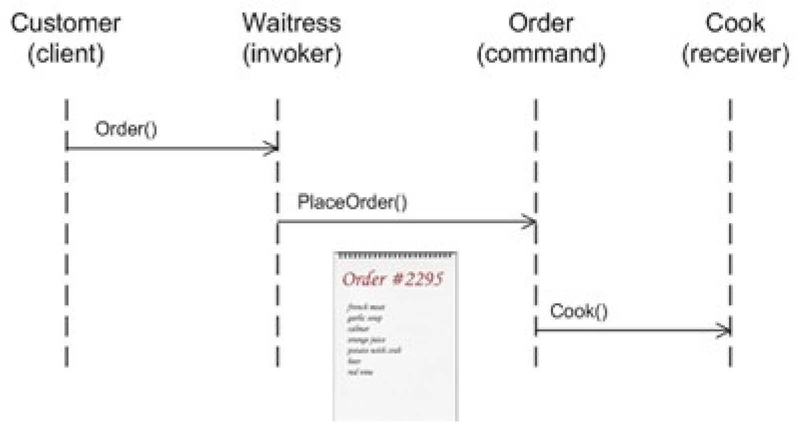
ii. Command DECOUPLES the object that INVOKES the OPERATION from the one that KNOWS how to PERFORM it

iii. PROBLEM : need to issue requests to objects WITHOUT knowing anything about the OPERATION being REQUESTED or the RECEIVER of the REQUEST

iv.


Client
«interface»
CallbackInterface
+execute()
Receiver
+doThis()
+doThat()
targetObject = receiverObject;
targetMethod = methodPointer;
-targetObject
CallbackOne
CallbackTwo
-targetMethod
+constructor(receiverObject,methodPointer)
+execute()
CallbackInterface token =
new CallbackTwo(new Receiver(), "doThis");
... the token object is passed to another
... object and that object calls
token.execute();
// use Java reflection or
// C++ or Delphi pointer
// to member function
targetObject.targetMethod();

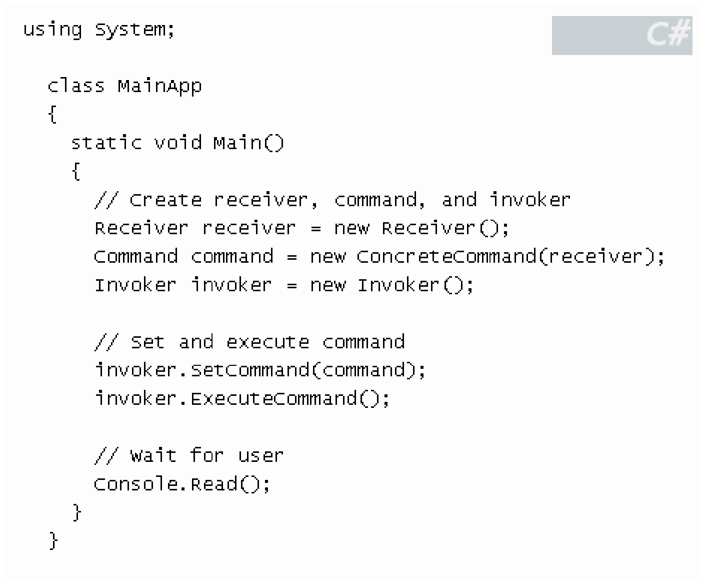
v. Command objects can be thought of as TOKENS that are CREATED by ONE CLIENT and PASSED to ANOTHER CLIENT that has resource for DOING IT



vi.

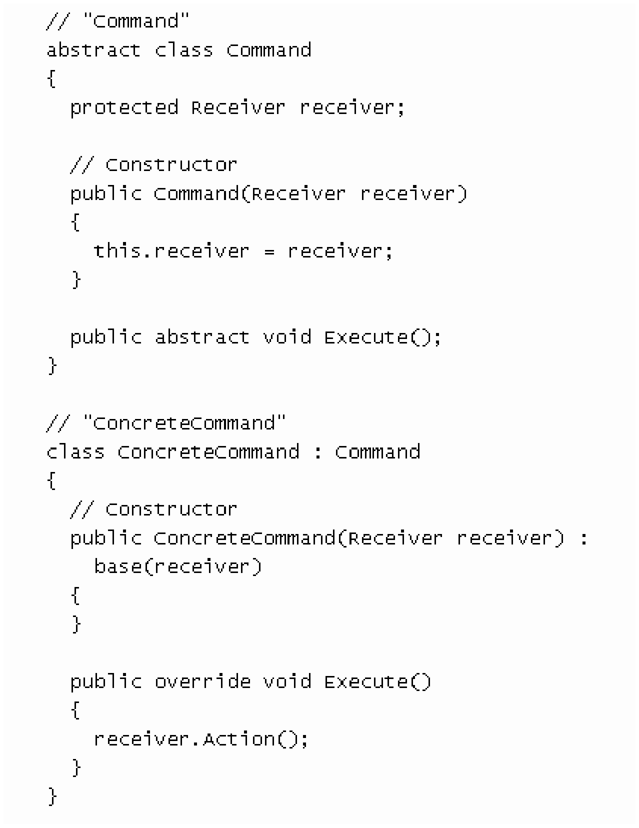
vii. CHAIN OF RESPONSIBILITY , COMMAND , MEADIATOR , OBSERVER address how to DECOUPLE SENDERS and RECEIVERS

viii. TWO important aspects of COMMAND pattern, INTERFACE SEPARATION (invoker is isolated from receiver) , TIME SEPARATION (stores a ready to go processing request that is to be started LATER)



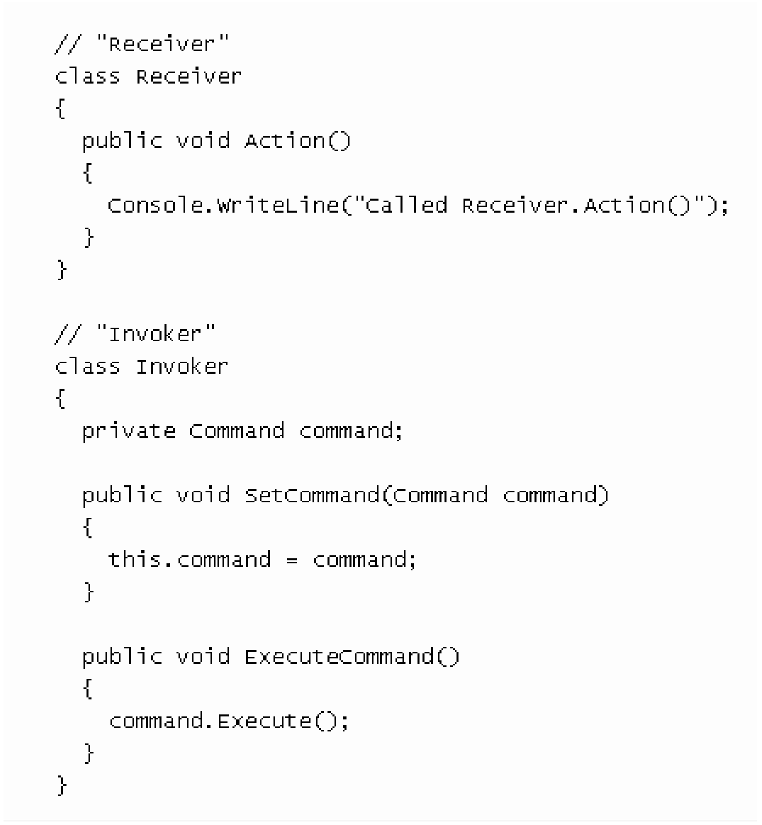
```
using System;

  class MainApp
  {
    static void Main()
    {
      // Create receiver, command, and invoker
      Receiver receiver = new Receiver();
      Command command = new ConcreteCommand(receiver);
      Invoker invoker = new Invoker();

      // Set and execute command
      invoker.SetCommand(command);
      invoker.ExecuteCommand();

      // Wait for user
      Console.Read();
    }
  }
```

ix.



```
  // "Command"
  abstract class Command
  {
    protected Receiver receiver;

    // Constructor
    public Command(Receiver receiver)
    {
      this.receiver = receiver;
    }

    public abstract void Execute();
  }

  // "ConcreteCommand"
  class ConcreteCommand : Command
  {
    // Constructor
    public ConcreteCommand(Receiver receiver) :
      base(receiver)
    {
    }

    public override void Execute()
    {
      receiver.Action();
    }
  }
```

x.



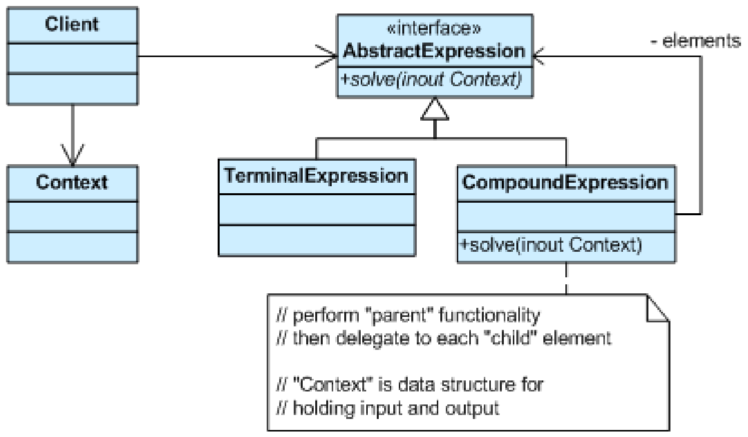
```
// "Receiver"
class Receiver
{
  public void Action()
  {
    Console.WriteLine("Called Receiver.Action()");
  }
}

// "Invoker"
class Invoker
{
  private Command command;

  public void SetCommand(Command command)
  {
    this.command = command;
  }

  public void ExecuteCommand()
  {
    command.Execute();
  }
}
```

xi.

c. Interpreter
   i. Given a LANGUAGE define a REPRESENTATION for its GRAMMER along with an INTERPRETER that uses the representation to INTERPRET sentences in the LANGUAGE
   ii. MAP a DOMAIN -> (to a) -> LANGUAGE , the LANGUAGE ->(to a) -> GRAMMER and GRAMMER -> (to a) -> HIERACHYCAL OBJECT ORIENTED DESING
   iii. A class of problems occurs REPEATEDLY in a WELL-DEFINED and WELL-UNDERSTOOD domain. If the DOMAIN were characterized with a LANGUAGE , then the PROBLEM could be easily SOLVED with an INTERPRETATION ENGINE

iv. Each RULE in the GRAMMER is either a COMPOSITE or a TERMIANL

v. INTERPRETER RELIES on the RECURSIVE traversal of the COMPOSITE PATTERN to INTERPRET the SENTENCES it is asked to PROCESS

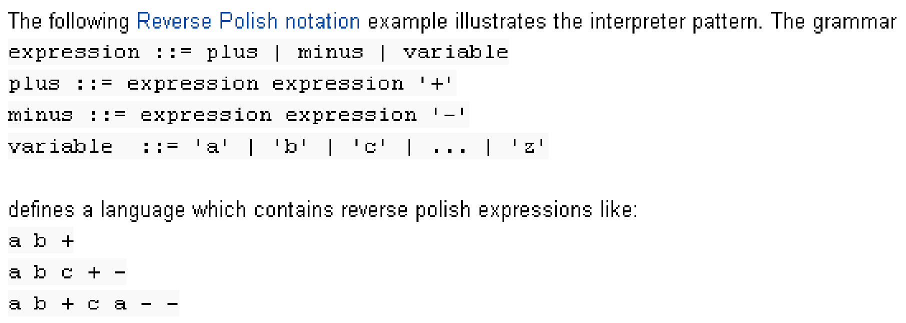

Client
«interface»
AbstractExpression
+solve(inout Context)
- elements
Context
TerminalExpression
CompoundExpression
+solve(inout Context)
// perform "parent" functionality
// then delegate to each "child" element
// "Context" is data structure for
// holding input and output

vi.

vii. Example situation is a MATEMATICAL EQUATION and the input values

viii. The INTEPRETER patterns DEFINES a GRAMMERTICAL representation for a LANGUAGE and an INTERPRETER to INTERPRET the GRAMMER

ix.


The following Reverse Polish notation example illustrates the interpreter pattern. The grammar

```
expression ::= plus | minus | variable
plus ::= expression expression '+'
minus ::= expression expression '-'
variable  ::= 'a' | 'b' | 'c' | ... | 'z'
```

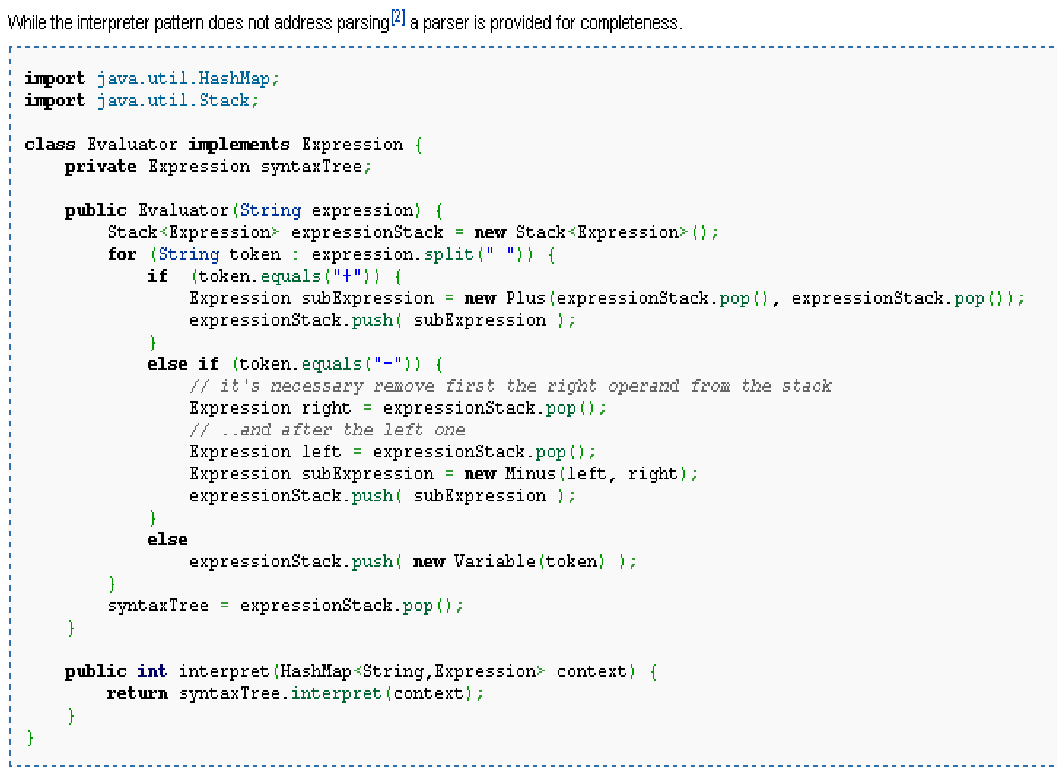
defines a language which contains reverse polish expressions like:

```
a b +
a b c + -
a b + c a - -
```

LIYANA ARACHCHIGE RANIL

x.

xi.



While the interpreter pattern does not address parsing[2] a parser is provided for completeness.

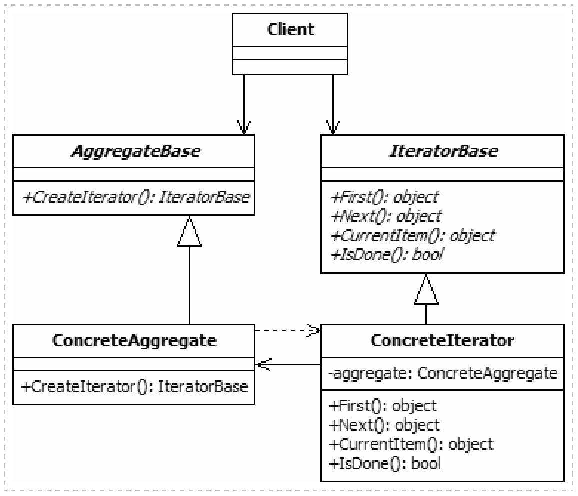
```
import java.util.HashMap;
import java.util.Stack;

class Evaluator implements Expression {
    private Expression syntaxTree;

    public Evaluator(String expression) {
        Stack<Expression> expressionStack = new Stack<Expression>();
        for (String token : expression.split(" ")) {
            if  (token.equals("+")) {
                Expression subExpression = new Plus(expressionStack.pop(), expressionStack.pop());
                expressionStack.push( subExpression );
            }
            else if (token.equals("-")) {
                // it's necessary remove first the right operand from the stack
                Expression right = expressionStack.pop();
                // ..and after the left one
                Expression left = expressionStack.pop();
                Expression subExpression = new Minus(left, right);
                expressionStack.push( subExpression );
            }
            else
                expressionStack.push( new Variable(token) );
        }
        syntaxTree = expressionStack.pop();
    }

    public int interpret(HashMap<String,Expression> context) {
        return syntaxTree.interpret(context);
    }
}
```

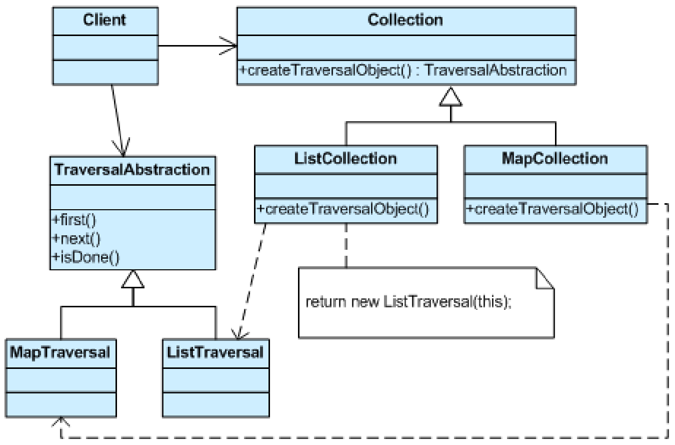
d. Iterator

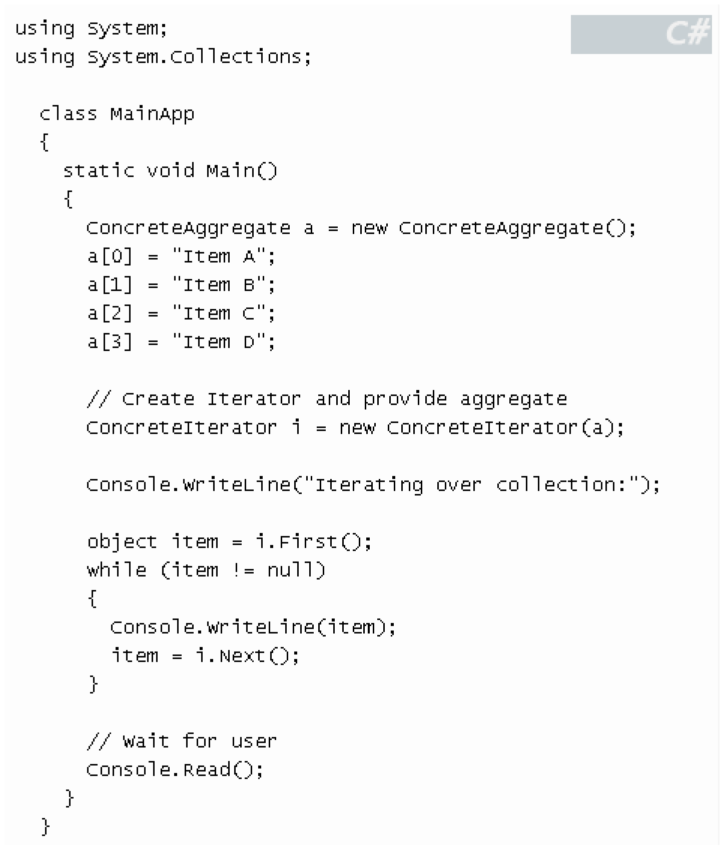
i. Provides a way to access the elements of an AGGREGATE object sequentially WITHOUT EXPOSING its underlying REPRESENTATION

ii. C++ and JAVA that makes it possible to DECOUPLE COLLECTIONS classes and ALGORITHMS

iii. Promote to FULL OBJECT STATUS the traversal of a COLLECTION
iv. Polymorphic traversal
v. Nee to ABSTRACT the TRAVERSAL of WILDLY different DATA STRUCTURES so that ALGORITHMS can be defined that are capable of INTERFACING with each transparently
vi. An AGGREGATE object such as a LIST should give a way to access its elements without EXPOSING its internal structure. Also it is needed to traverse the LIST in different ways. But you don’t BLOAT the list interface with these traversal related operations

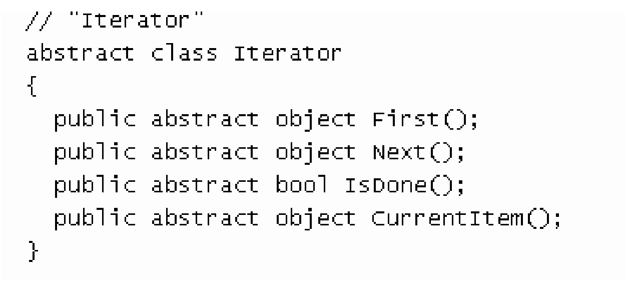

Client
AggregateBase
+CreateIterator(): IteratorBase
IteratorBase
+First(): object
+Next(): object
+CurrentItem(): object
+IsDone(): bool
ConcreteAggregate
+CreateIterator(): IteratorBase
ConcreteIterator
-aggregate: ConcreteAggregate
+First(): object
+Next(): object
+CurrentItem(): object
+IsDone(): bool


Client
Collection
+createTraversalObject() : TraversalAbstraction
TraversalAbstraction
+first()
+next()
+isDone()
ListCollection
+createTraversalObject()
MapCollection
+createTraversalObject()
return new ListTraversal(this);
MapTraversal
ListTraversal

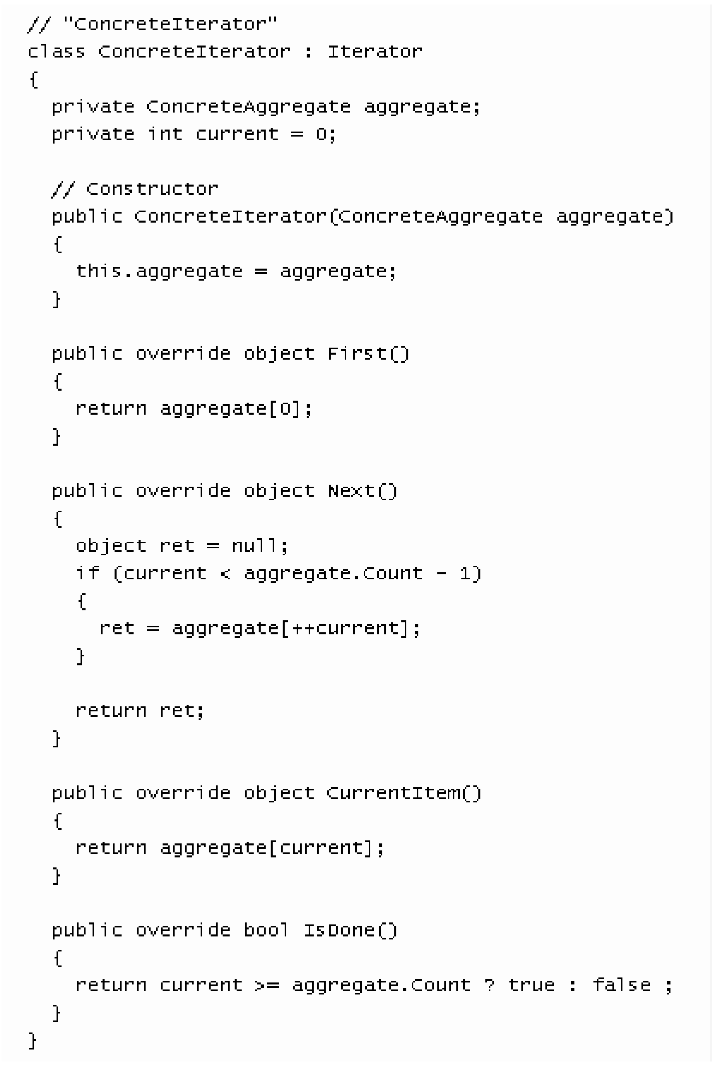


vii.

```
using System;
using System.Collections;

  class MainApp
  {
    static void Main()
    {
      ConcreteAggregate a = new ConcreteAggregate();
      a[0] = "Item A";
      a[1] = "Item B";
      a[2] = "Item C";
      a[3] = "Item D";

      // Create Iterator and provide aggregate
      ConcreteIterator i = new ConcreteIterator(a);

      Console.WriteLine("Iterating over collection:");

      object item = i.First();
      while (item != null)
      {
        Console.WriteLine(item);
        item = i.Next();
      }

      // Wait for user
      Console.Read();
    }
  }
```



```
// "Iterator"
abstract class Iterator
{
  public abstract object First();
  public abstract object Next();
  public abstract bool IsDone();
  public abstract object CurrentItem();
}
```

viii. 

```
// "ConcreteIterator"
class ConcreteIterator : Iterator
{
  private ConcreteAggregate aggregate;
  private int current = 0;

  // Constructor
  public ConcreteIterator(ConcreteAggregate aggregate)
  {
    this.aggregate = aggregate;
  }

  public override object First()
  {
    return aggregate[0];
  }

  public override object Next()
  {
    object ret = null;
    if (current < aggregate.Count - 1)
    {
      ret = aggregate[++current];
    }

    return ret;
  }

  public override object CurrentItem()
  {
    return aggregate[current];
  }

  public override bool IsDone()
  {
    return current >= aggregate.Count ? true : false ;
  }
}
```

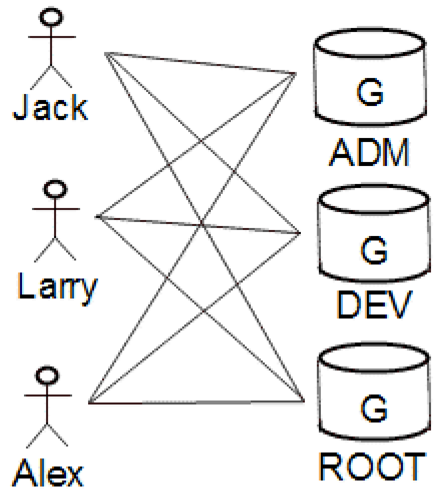
e. Mediator

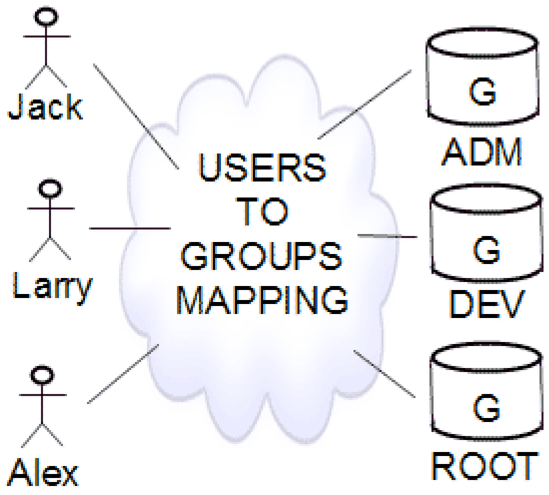
   i. Define an object that ENCAPSULATE how a set of objects INTERACT

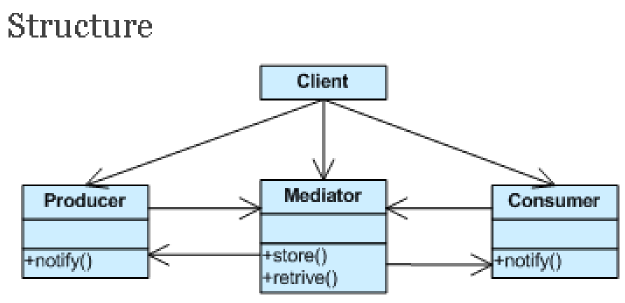
   ii. MEADIATOR promotes LOOSE COUPLING by keeping objects from REFERRING each other EXPLICITLY and it lets you vary their INTERACTION independently

   iii. Design an INTERMEADIATORY to DECOUPLE many PEERS

iv. PROBLEM : Want to design REUSABLE components , but DEPENDENCIES between the potentially reusable pieces demonstrate the SPAGHETTI CODE

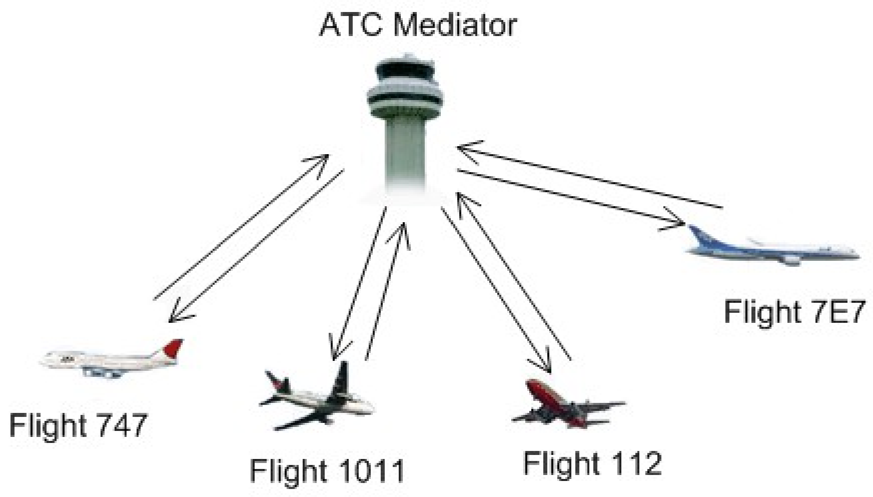
v. 

vi. 

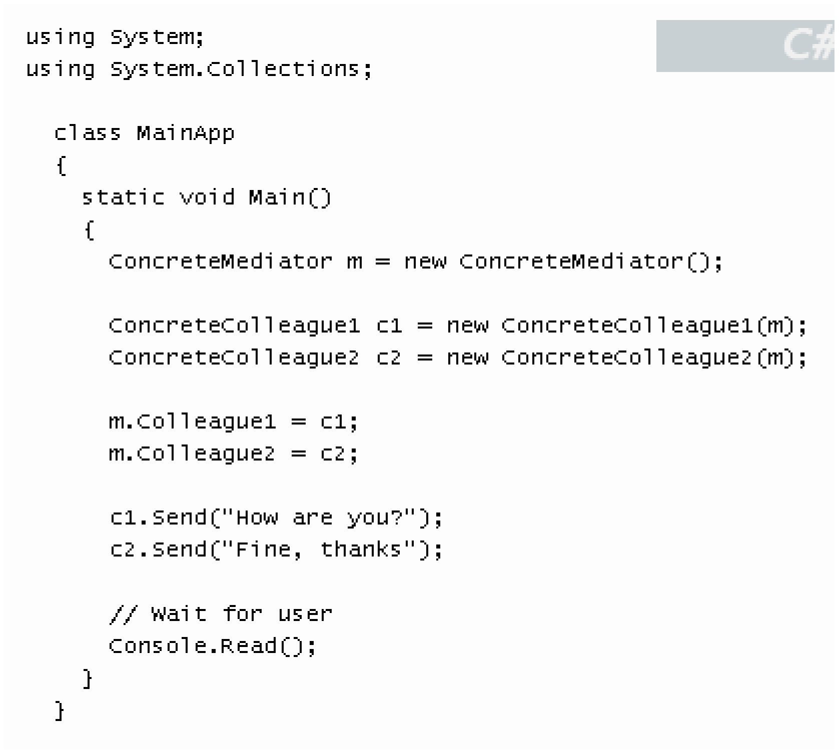
vii. 

viii. Colleagues or Peers are not coupled to each other , each TALK to MEADIATOR

ix.

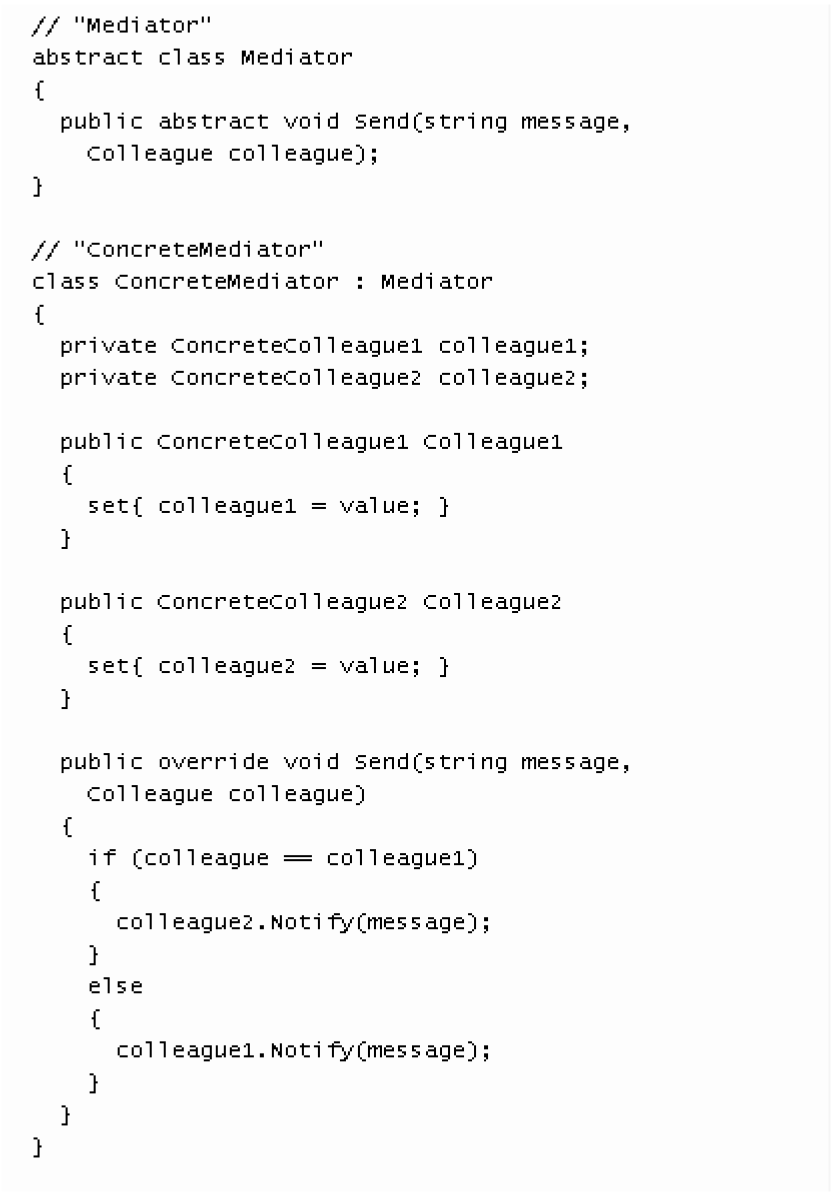


x. Be careful not to create a CONTROLLER or GOD object



```
using System;
using System.Collections;

  class MainApp
  {
    static void Main()
    {
      ConcreteMediator m = new ConcreteMediator();

      ConcreteColleague1 c1 = new ConcreteColleague1(m);
      ConcreteColleague2 c2 = new ConcreteColleague2(m);

      m.Colleague1 = c1;
      m.Colleague2 = c2;

      c1.Send("How are you?");
      c2.Send("Fine, thanks");

      // Wait for user
      Console.Read();
    }
  }
```

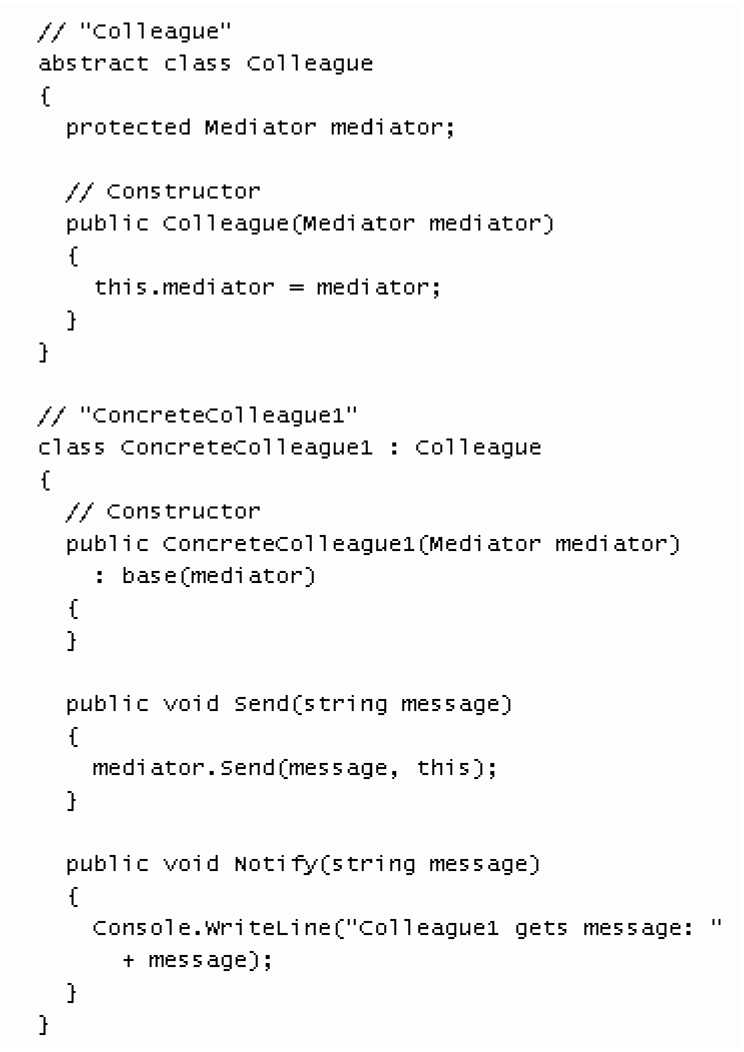
xi.



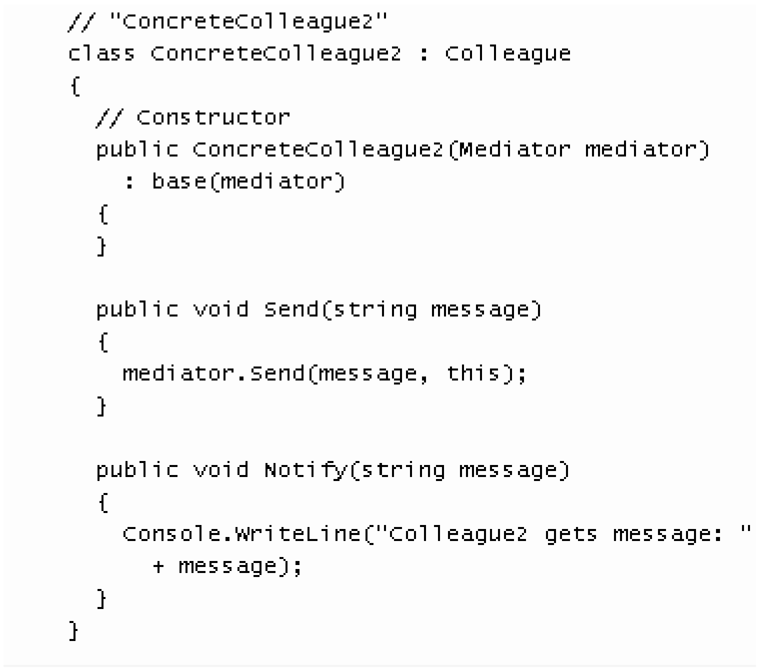
```
// "Mediator"
abstract class Mediator
{
  public abstract void Send(string message,
    Colleague colleague);
}

// "ConcreteMediator"
class ConcreteMediator : Mediator
{
  private ConcreteColleague1 colleague1;
  private ConcreteColleague2 colleague2;

  public ConcreteColleague1 Colleague1
  {
    set{ colleague1 = value; }
  }

  public ConcreteColleague2 Colleague2
  {
    set{ colleague2 = value; }
  }

  public override void Send(string message,
    Colleague colleague)
  {
    if (colleague == colleague1)
    {
      colleague2.Notify(message);
    }
    else
    {
      colleague1.Notify(message);
    }
  }
}
```

xii.

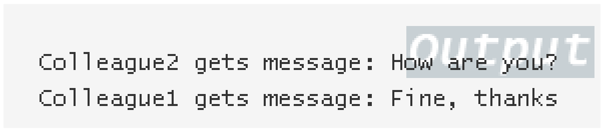
xiii.



```
// "Colleague"
abstract class Colleague
{
  protected Mediator mediator;

  // Constructor
  public Colleague(Mediator mediator)
  {
    this.mediator = mediator;
  }
}

// "ConcreteColleague1"
class ConcreteColleague1 : Colleague
{
  // Constructor
  public ConcreteColleague1(Mediator mediator)
    : base(mediator)
  {
  }

  public void Send(string message)
  {
    mediator.Send(message, this);
  }

  public void Notify(string message)
  {
    Console.WriteLine("Colleague1 gets message: "
      + message);
  }
}
```



```
// "ConcreteColleague2"
class ConcreteColleague2 : Colleague
{
  // Constructor
  public ConcreteColleague2(Mediator mediator)
    : base(mediator)
  {
  }

  public void Send(string message)
  {
    mediator.Send(message, this);
  }

  public void Notify(string message)
  {
    Console.WriteLine("Colleague2 gets message: "
      + message);
  }
}
```

xiv. 

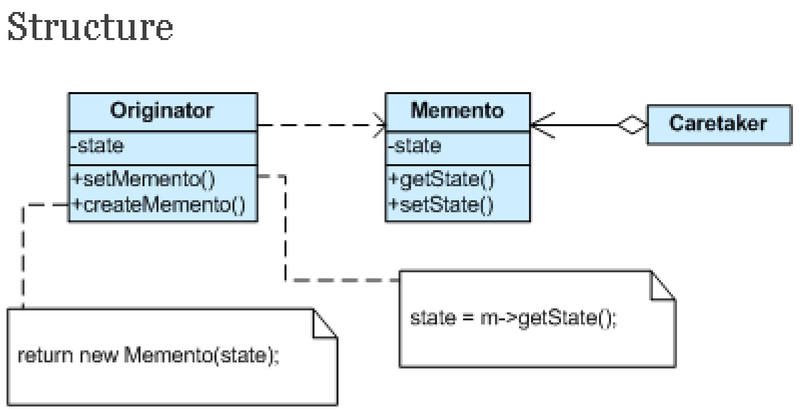
Output

```
Colleague2 gets message: How are you?
Colleague1 gets message: Fine, thanks
```

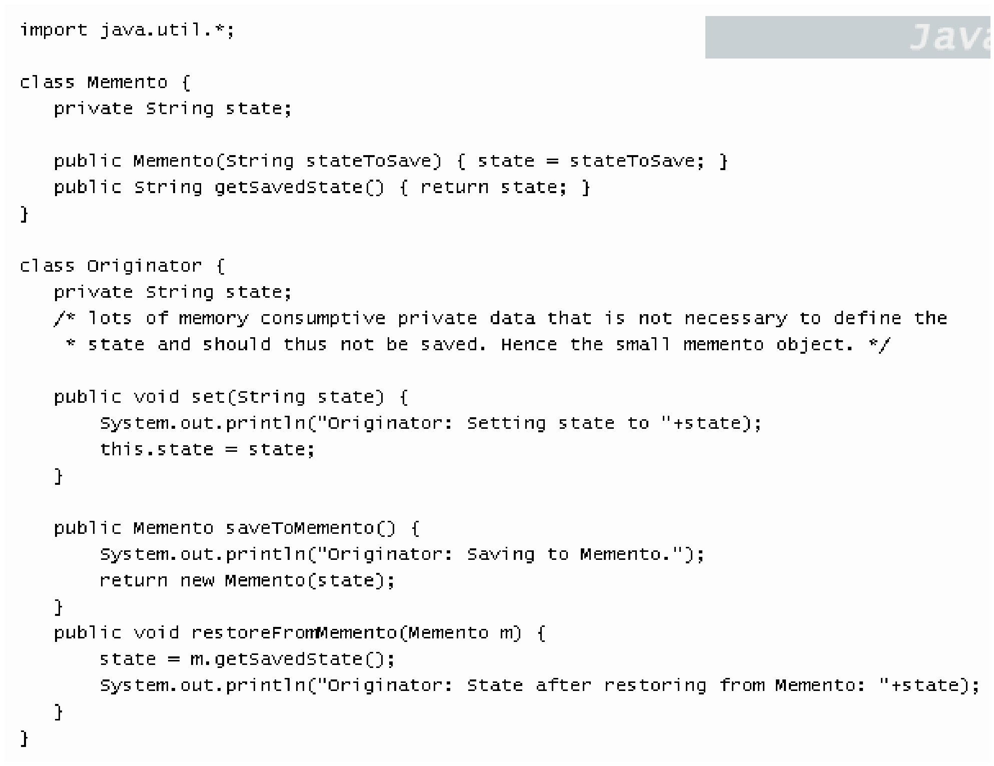
f. Memento
   i. Without VIOLATING encapsulation , CAPTURE and EXTERNALIZE an object INTERNAL state so that Objects internal state can be returned to this later
   ii. PROMOTE undo or ROLL back to FULL object status
   iii. PROBLEM : Need to restore an object back to its previous state
   iv. The client request MEMENTO object from the source object when it needs to CHECKPOINT the source objects state
   v. The source object initializes the MEMENTO with a characterization of its state
   vi. The client is the CARE TAKER of the MEMENTO

vii. But only the SOUCE object can store and retrieve information from the MEMENTO

viii. ORIGINATOR – the object that knows how to SAVE itself , CARETAKER – the object that knows why and when the originator needs to save and restore itself , MEMENTO – The lock box that is written and read by the ORIGINATOR and shepherded by the CARETAKER

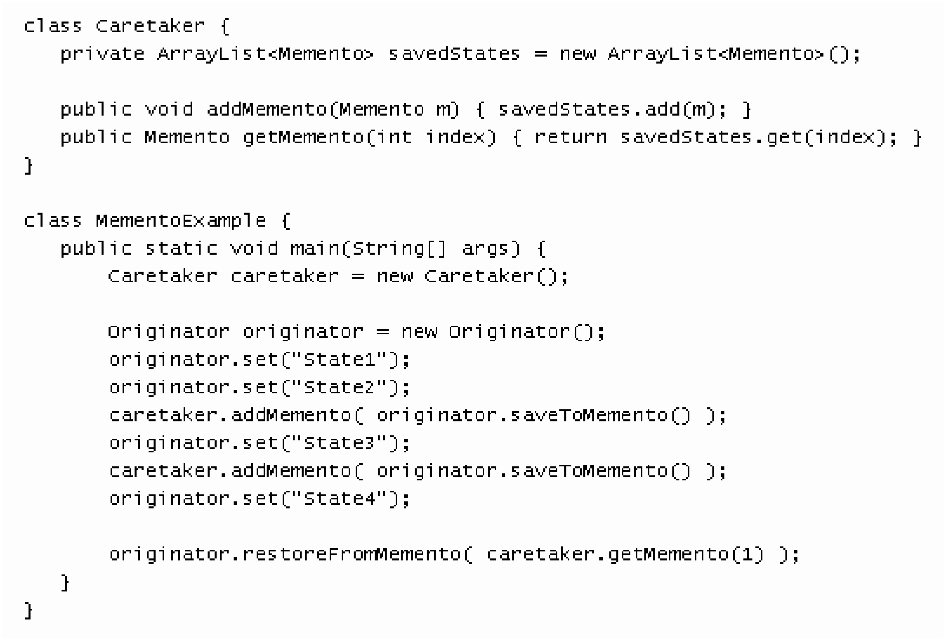
ix. 

x.



```
import java.util.*;

class Memento {
   private String state;

   public Memento(String stateToSave) { state = stateToSave; }
   public String getSavedState() { return state; }
}

class Originator {
   private String state;
   /* lots of memory consumptive private data that is not necessary to define the
    * state and should thus not be saved. Hence the small memento object. */

   public void set(String state) {
       System.out.println("Originator: Setting state to "+state);
       this.state = state;
   }

   public Memento saveToMemento() {
       System.out.println("Originator: Saving to Memento.");
       return new Memento(state);
   }
   public void restoreFromMemento(Memento m) {
       state = m.getSavedState();
       System.out.println("Originator: State after restoring from Memento: "+state);
   }
}
```

xi.

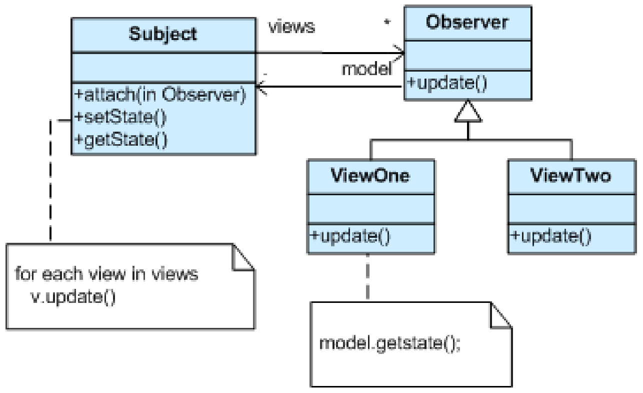


```
class Caretaker {
   private ArrayList<Memento> savedStates = new ArrayList<Memento>();

   public void addMemento(Memento m) { savedStates.add(m); }
   public Memento getMemento(int index) { return savedStates.get(index); }
}

class MementoExample {
   public static void main(String[] args) {
       Caretaker caretaker = new Caretaker();

       Originator originator = new Originator();
       originator.set("State1");
       originator.set("State2");
       caretaker.addMemento( originator.saveToMemento() );
       originator.set("State3");
       caretaker.addMemento( originator.saveToMemento() );
       originator.set("State4");

       originator.restoreFromMemento( caretaker.getMemento(1) );
   }
}
```

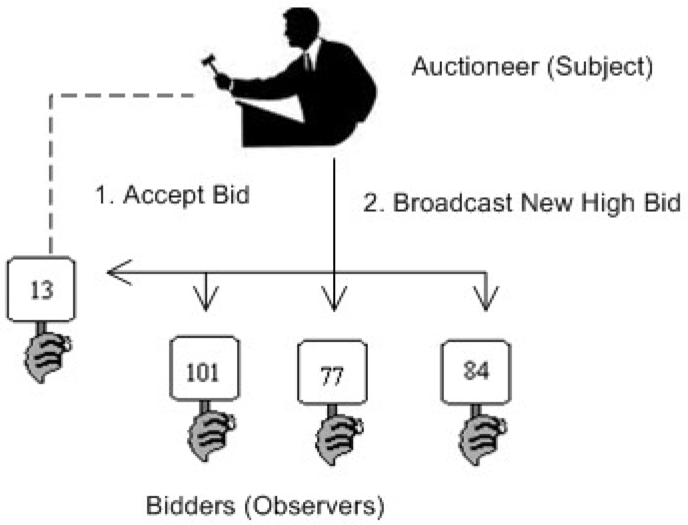
g. Observer
   i. Define a ONE-to-MANY dependency between objects so that when one object changes state , all its dependants are notified and updated automatically
   ii. The VIEW part of the MVC
   iii. PROBLEM : A large MONOLITIC design does not SCALE well as new GRAPHING or MONITORING requirements are levied
   iv. Observers REGISTER themselves with the SUBJECT
   v. The SUBJECT broadcast EVENTS to ALL REGISTRED Observers

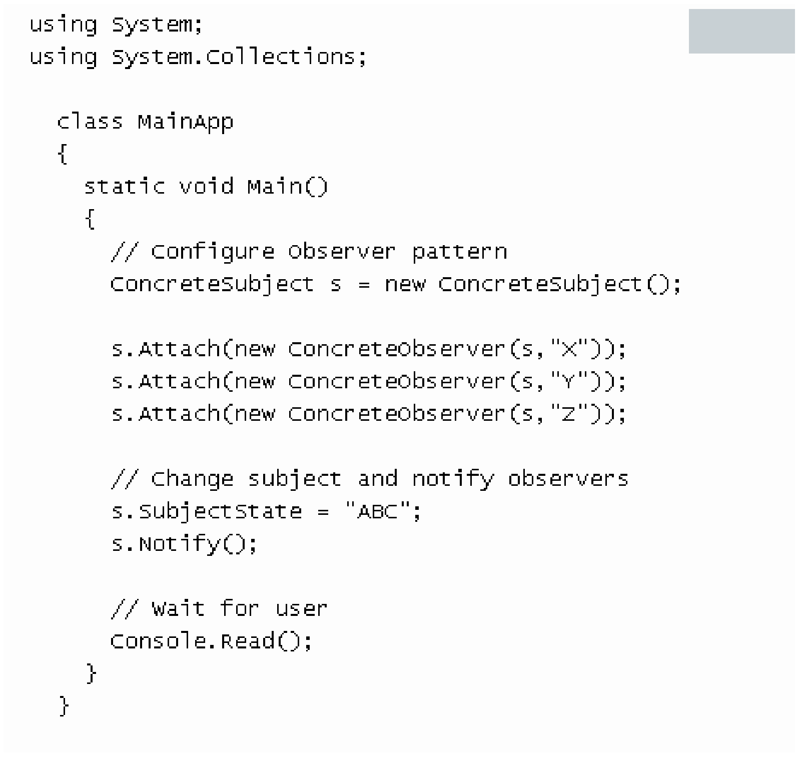
vi. The SUBJECT may PUSH information at the OBSERVER or , the OBSERVES may PULL the information they need from the SUBJECT

vii.



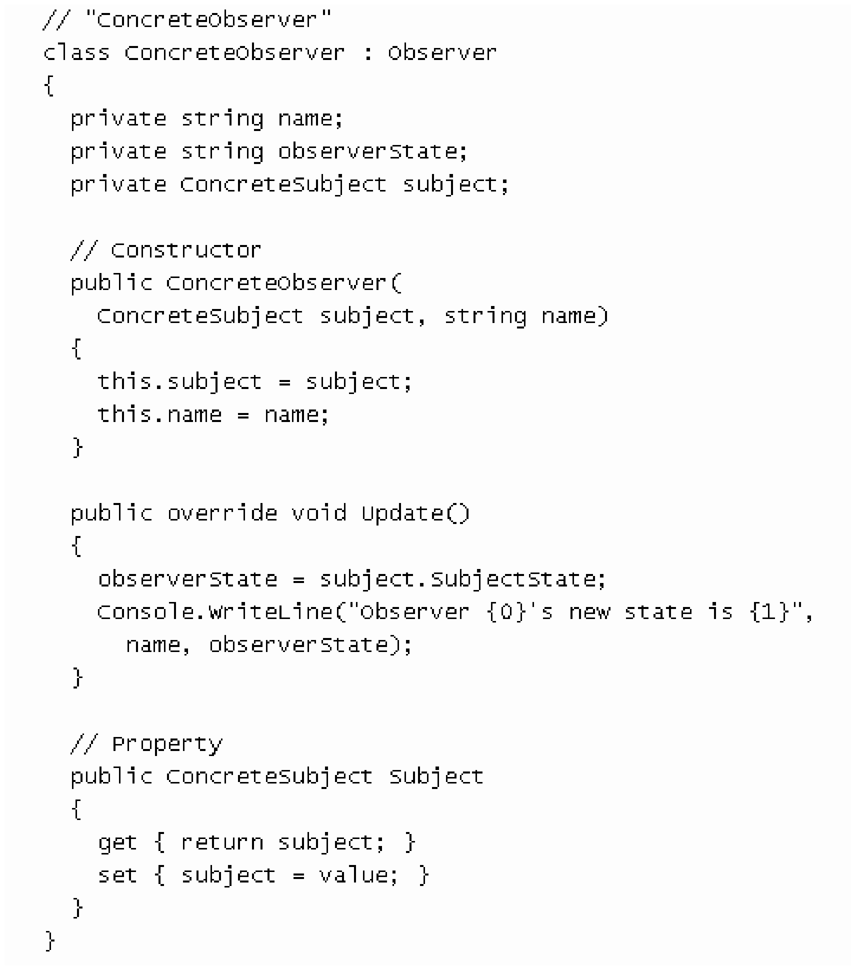
viii.





```
using System;
using System.Collections;

  class MainApp
  {
    static void Main()
    {
      // Configure Observer pattern
      ConcreteSubject s = new ConcreteSubject();

      s.Attach(new ConcreteObserver(s,"X"));
      s.Attach(new ConcreteObserver(s,"Y"));
      s.Attach(new ConcreteObserver(s,"Z"));

      // Change subject and notify observers
      s.SubjectState = "ABC";
      s.Notify();

      // Wait for user
      Console.Read();
    }
  }
```

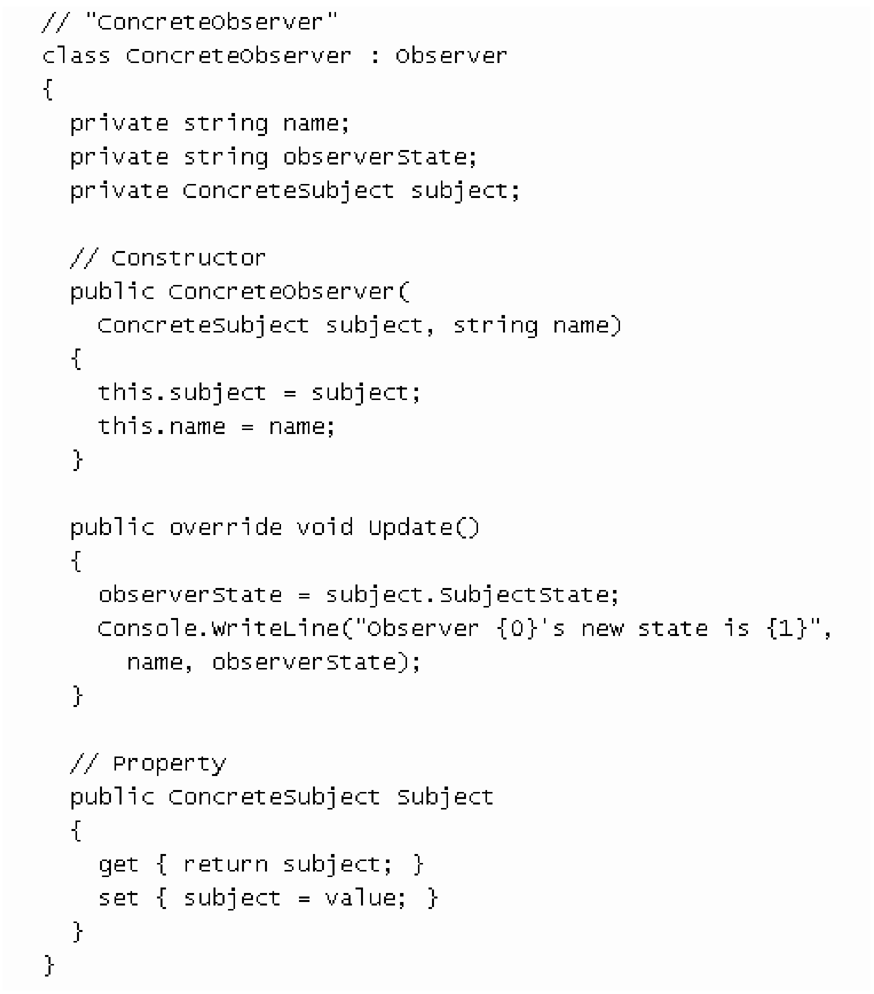
ix.



x.

```
// "ConcreteObserver"
class ConcreteObserver : Observer
{
  private string name;
  private string observerState;
  private ConcreteSubject subject;

  // Constructor
  public ConcreteObserver(
    ConcreteSubject subject, string name)
  {
    this.subject = subject;
    this.name = name;
  }

  public override void Update()
  {
    observerState = subject.SubjectState;
    Console.WriteLine("Observer {0}'s new state is {1}",
      name, observerState);
  }

  // Property
  public ConcreteSubject Subject
  {
    get { return subject; }
    set { subject = value; }
  }
}
```

xi.

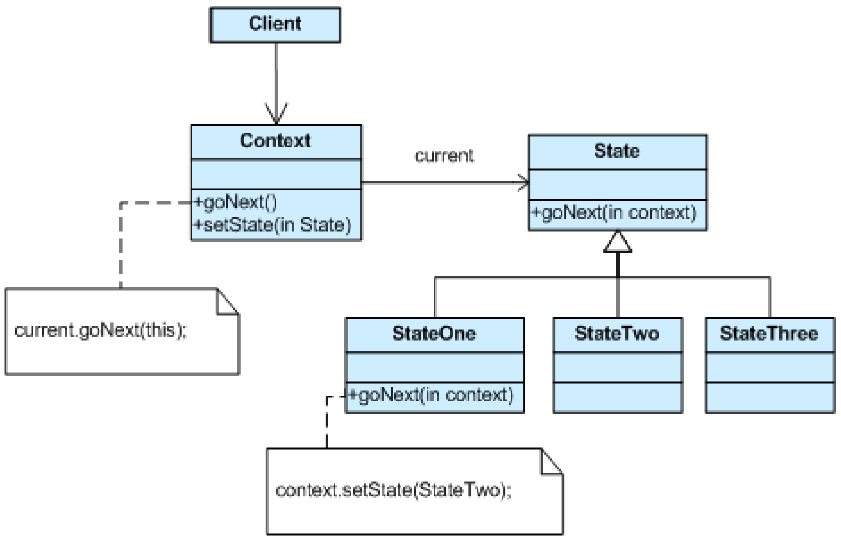


```
// "ConcreteObserver"
class ConcreteObserver : Observer
{
  private string name;
  private string observerState;
  private ConcreteSubject subject;

  // Constructor
  public ConcreteObserver(
    ConcreteSubject subject, string name)
  {
    this.subject = subject;
    this.name = name;
  }

  public override void Update()
  {
    observerState = subject.SubjectState;
    Console.WriteLine("Observer {0}'s new state is {1}",
      name, observerState);
  }

  // Property
  public ConcreteSubject Subject
  {
    get { return subject; }
    set { subject = value; }
  }
}
```

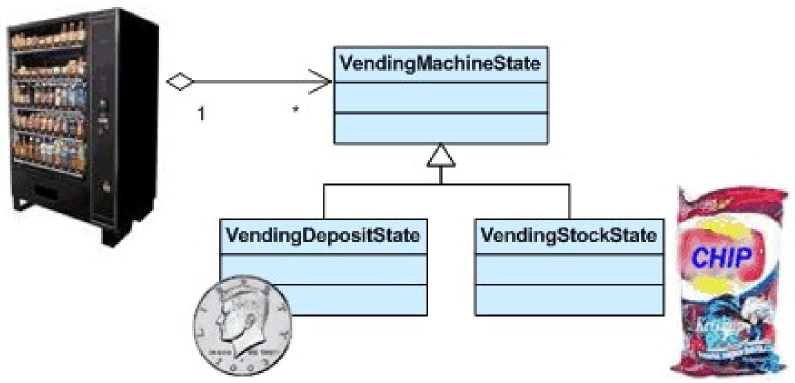
h. State

   i. Allow an Object to alter its BEHAVIOR when its INTERNAL state changes. The object will appear to change its class
   ii. A monolithic objects BEHAVIOUR is a FUNCTION of its state , and it must change its behavior at RUNTIME depending on the STATE
   iii. Define a CONTEXT class to present a SINGLE interface to the OUTSIDE world
   iv. Define a STATE abstract base class
   v. Represents the DIFFERENT state of the state machine as DERIVED classes of the State base class
   vi. Define state-specific behavior in the appropriate State derived class
   vii. Maintain pointer to the current state in the context class
   viii. To change the state of the MACHINE change the CURRENT STATE POINTER
   ix. The state pattern DOES not define WHERE the state TRANSITION will be DEFINED. There are two choices , The CONTEXT object or each individual STATE derived class

x.

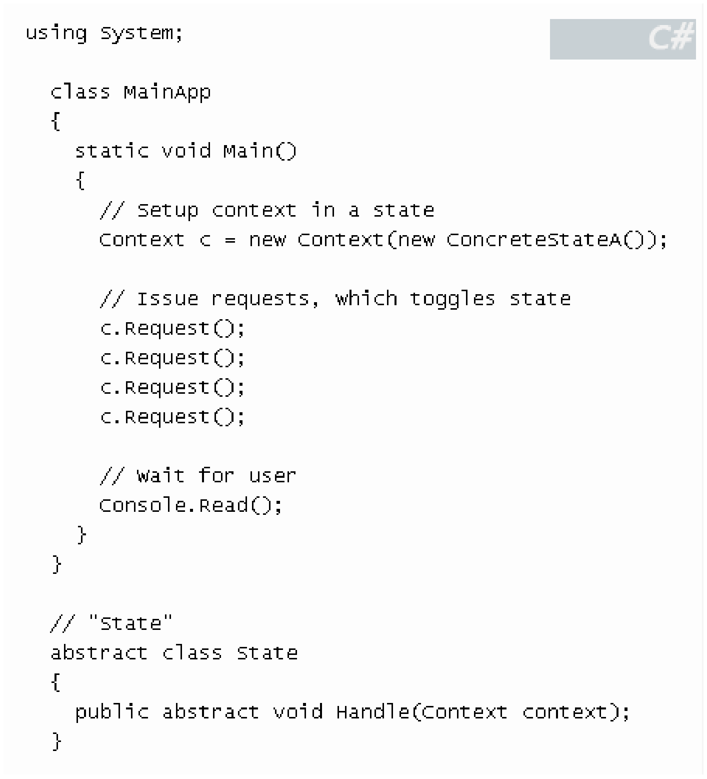


xi. Example of state is VENDING machine. Vending machine has states based on the INVENTORY , amount of CURRENCY deposited , etc



xii.

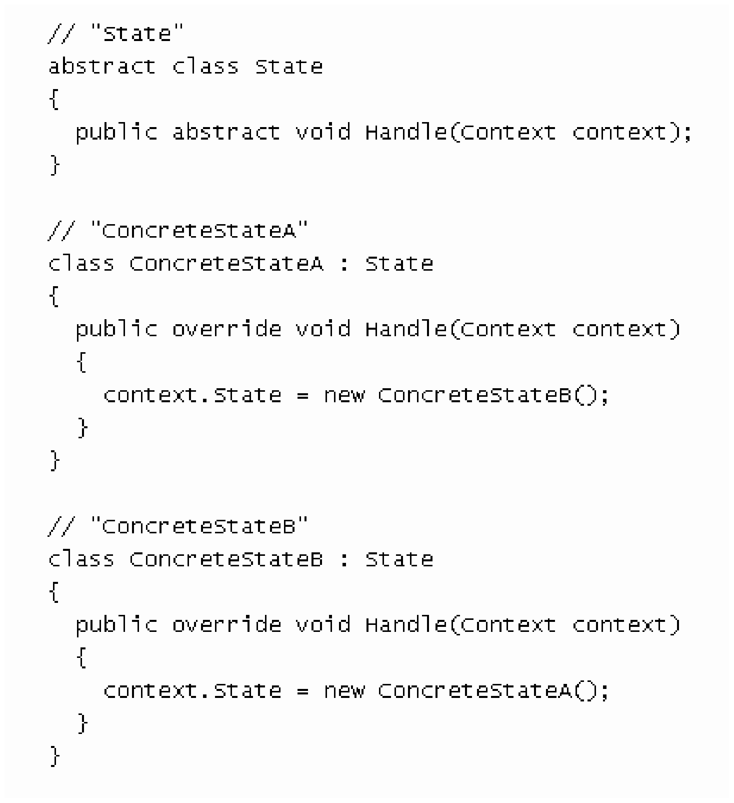
xiii. The STRUCTURE of STATE and BRIDGE are identical. The two patterns use the same structure to solve different problems



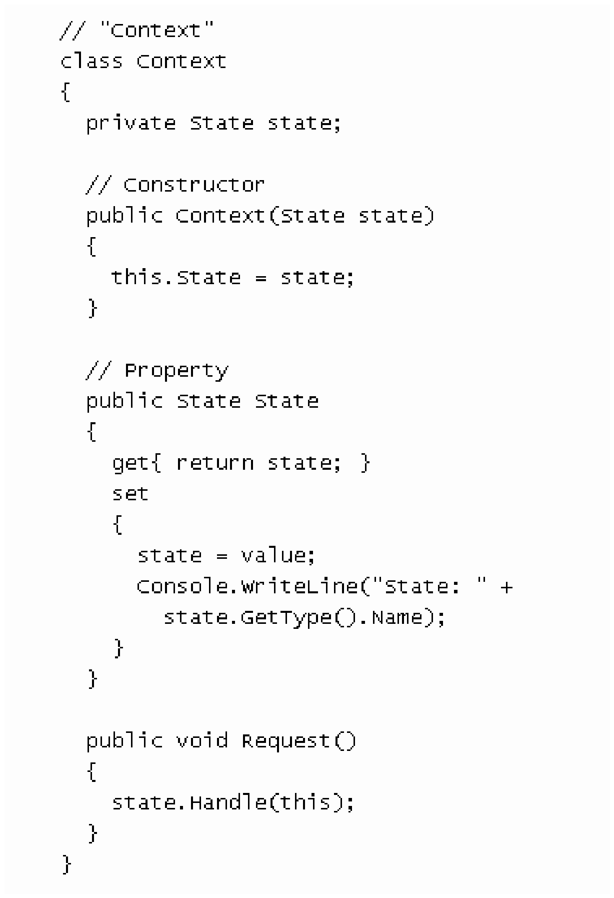
```
using System;

  class MainApp
  {
    static void Main()
    {
      // Setup context in a state
      Context c = new Context(new ConcreteStateA());

      // Issue requests, which toggles state
      c.Request();
      c.Request();
      c.Request();
      c.Request();

      // Wait for user
      Console.Read();
    }
  }

  // "State"
  abstract class State
  {
    public abstract void Handle(Context context);
  }
```

xiv.

XV.

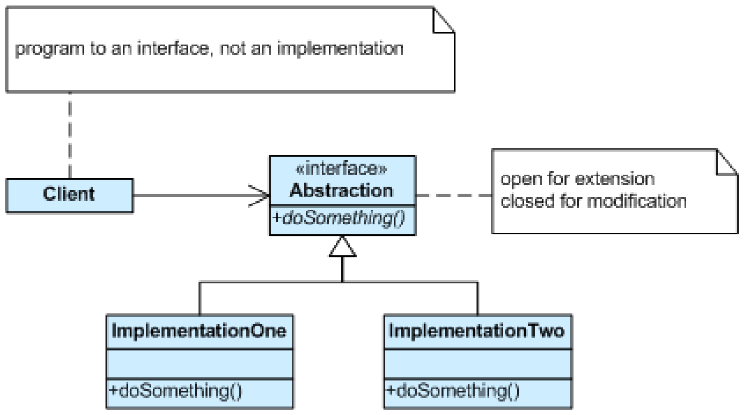


```
// "State"
abstract class State
{
  public abstract void Handle(Context context);
}

// "ConcreteStateA"
class ConcreteStateA : State
{
  public override void Handle(Context context)
  {
    context.State = new ConcreteStateB();
  }
}

// "ConcreteStateB"
class ConcreteStateB : State
{
  public override void Handle(Context context)
  {
    context.State = new ConcreteStateA();
  }
}
```

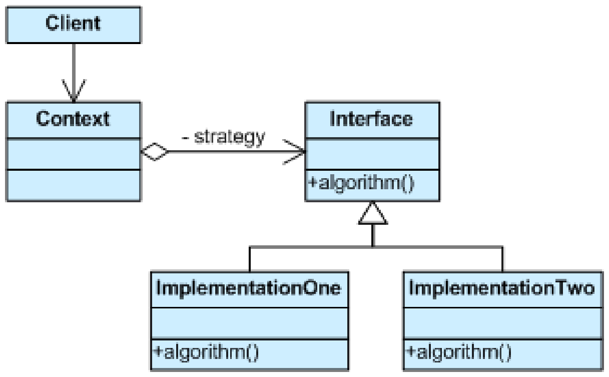
xvi. 

```
// "Context"
class Context
{
  private State state;

  // Constructor
  public Context(State state)
  {
    this.State = state;
  }

  // Property
  public State State
  {
    get{ return state; }
    set
    {
      state = value;
      Console.WriteLine("State: " +
        state.GetType().Name);
    }
  }

  public void Request()
  {
    state.Handle(this);
  }
}
```

i. Strategy
   i. Define a FAMILY of ALGORITHMS, encapsulate each one and make them INTERCHANGEABLE. Strategy lets the algorithm vary independently from the clients that use it
   ii. CAPTURE the ABSTRACTION in an INTERFACE , bury IMLEMENTATAION details in derived classes
   iii. OPEN for EXTENSION closed for MODIFICATION

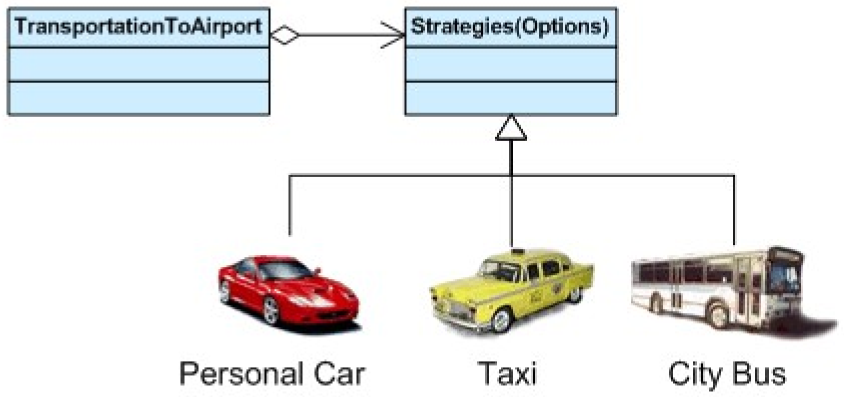

program to an interface, not an implementation
Client
«interface»
Abstraction
+doSomething()
open for extension
closed for modification
ImplementationOne
+doSomething()
ImplementationTwo
+doSomething()

iv.

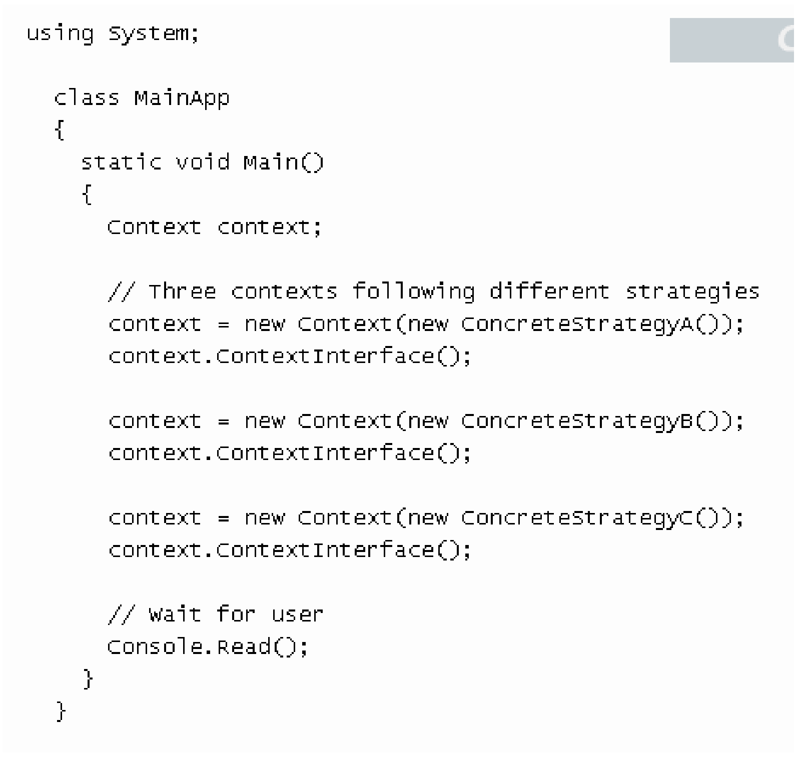

Client
Context
- strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()

v.

vi.

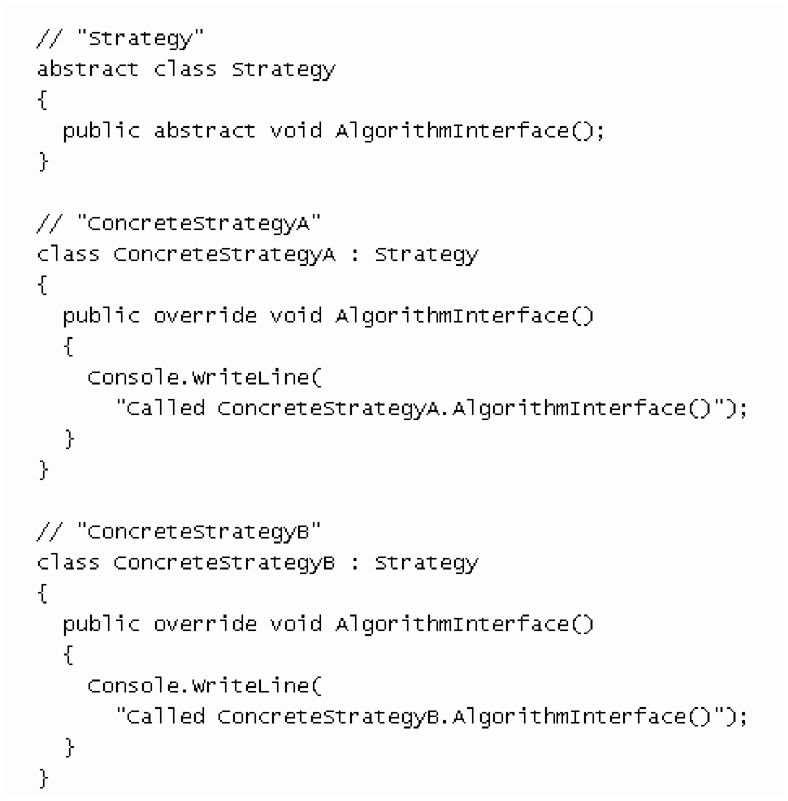


vii. Strategy lets you change the GUTS of an object while DECORATOR lets you change the SKIN viii.

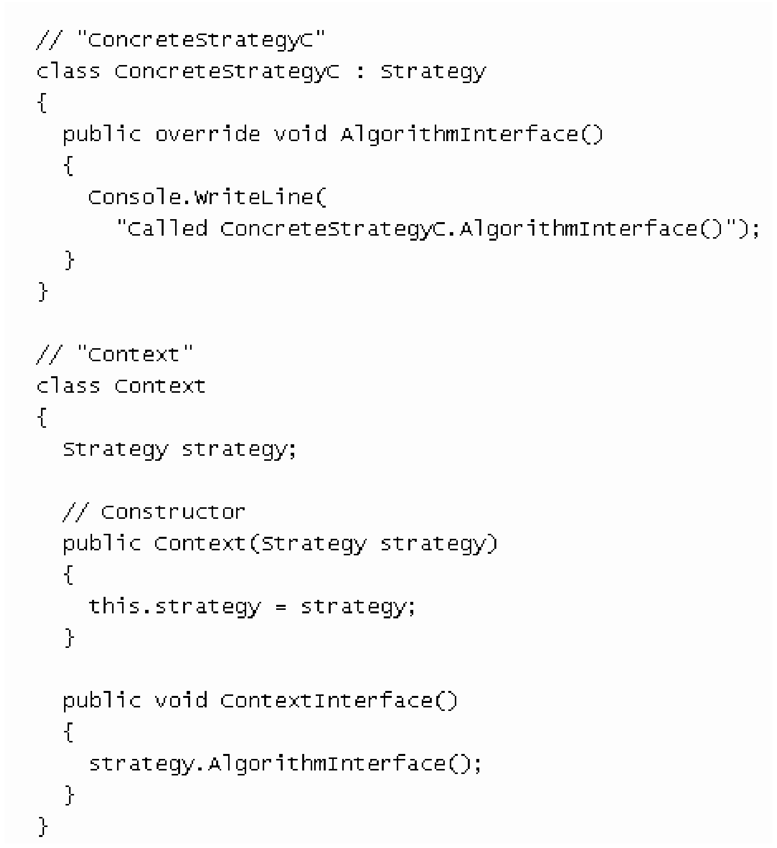


```
using System;

  class MainApp
  {
    static void Main()
    {
      Context context;

      // Three contexts following different strategies
      context = new Context(new ConcreteStrategyA());
      context.ContextInterface();

      context = new Context(new ConcreteStrategyB());
      context.ContextInterface();

      context = new Context(new ConcreteStrategyC());
      context.ContextInterface();

      // Wait for user
      Console.Read();
    }
  }
```

ix.

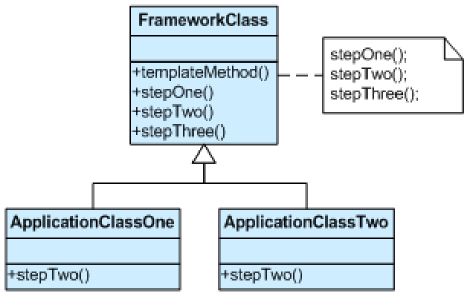


```
// "Strategy"
abstract class Strategy
{
  public abstract void AlgorithmInterface();
}

// "ConcreteStrategyA"
class ConcreteStrategyA : Strategy
{
  public override void AlgorithmInterface()
  {
    Console.WriteLine(
      "Called ConcreteStrategyA.AlgorithmInterface()");
  }
}

// "ConcreteStrategyB"
class ConcreteStrategyB : Strategy
{
  public override void AlgorithmInterface()
  {
    Console.WriteLine(
      "Called ConcreteStrategyB.AlgorithmInterface()");
  }
}
```

x. 

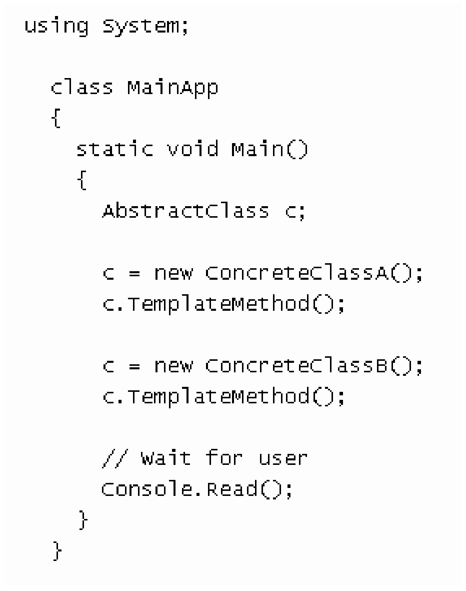
```
// "ConcreteStrategyC"
class ConcreteStrategyC : Strategy
{
  public override void AlgorithmInterface()
  {
    Console.WriteLine(
      "Called ConcreteStrategyC.AlgorithmInterface()");
  }
}

// "Context"
class Context
{
  Strategy strategy;

  // Constructor
  public Context(Strategy strategy)
  {
    this.strategy = strategy;
  }

  public void ContextInterface()
  {
    strategy.AlgorithmInterface();
  }
}
```

j. Template Method
   i. Define the SKELETON of an ALGORITHM in an OPERATION, deferring some steps to CLIENT subclasses. TEMPLATE METHOD lets SUBCLASSES REDEDFINE certain steps of an ALGORITHM without changing the ALGORITHMS structure
   ii. Base class DECLAIRS algorithm “PLACEHOLDERS” and derived classes IMPLEMENT the PLACEHOLDERS
   iii. TEMPLATE method is used PROMINANTLY in FRAMEWORKS

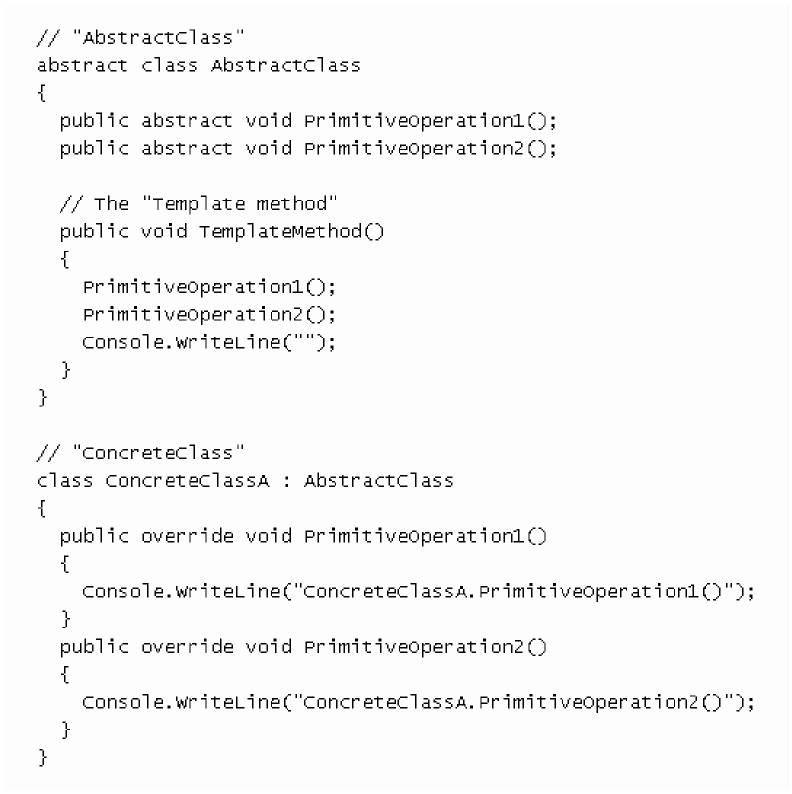
iv. 

v. TEMPLATE method uses INHERITANCE to vary PART of an ALGORITHM. STRATEGY uses DELEGATION to vary the ENTIRE ALGORITHM

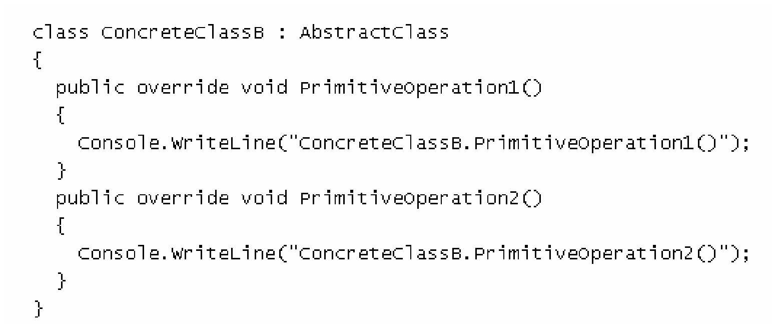
vi. FACTORY method is a SPECIALIZATION of TEMPLATE method

vii. 

```
using System;

  class MainApp
  {
    static void Main()
    {
      AbstractClass c;

      c = new ConcreteClassA();
      c.TemplateMethod();

      c = new ConcreteClassB();
      c.TemplateMethod();

      // Wait for user
      Console.Read();
    }
  }
```

viii. 

```
// "AbstractClass"
abstract class AbstractClass
{
  public abstract void PrimitiveOperation1();
  public abstract void PrimitiveOperation2();

  // The "Template method"
  public void TemplateMethod()
  {
    PrimitiveOperation1();
    PrimitiveOperation2();
    Console.WriteLine("");
  }
}

// "ConcreteClass"
class ConcreteClassA : AbstractClass
{
  public override void PrimitiveOperation1()
  {
    Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");
  }
  public override void PrimitiveOperation2()
  {
    Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");
  }
}
```

ix. 

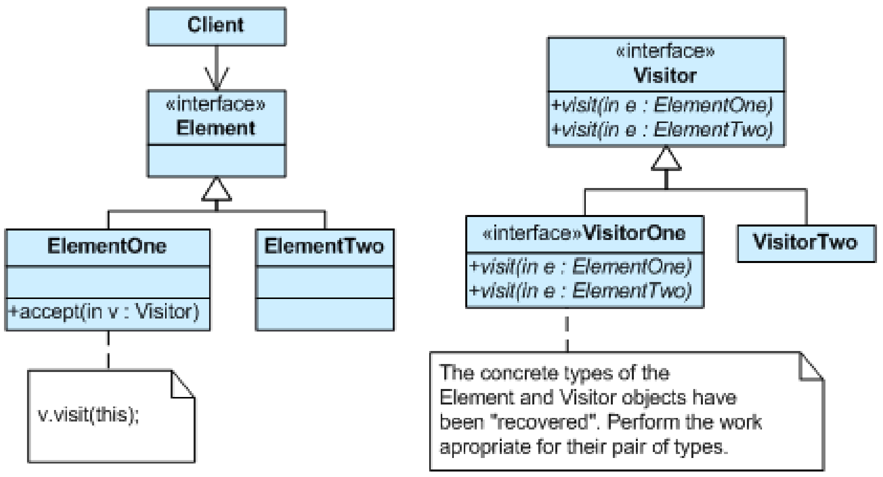
```
class ConcreteClassB : AbstractClass
{
  public override void PrimitiveOperation1()
  {
    Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");
  }
  public override void PrimitiveOperation2()
  {
    Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");
  }
}
```

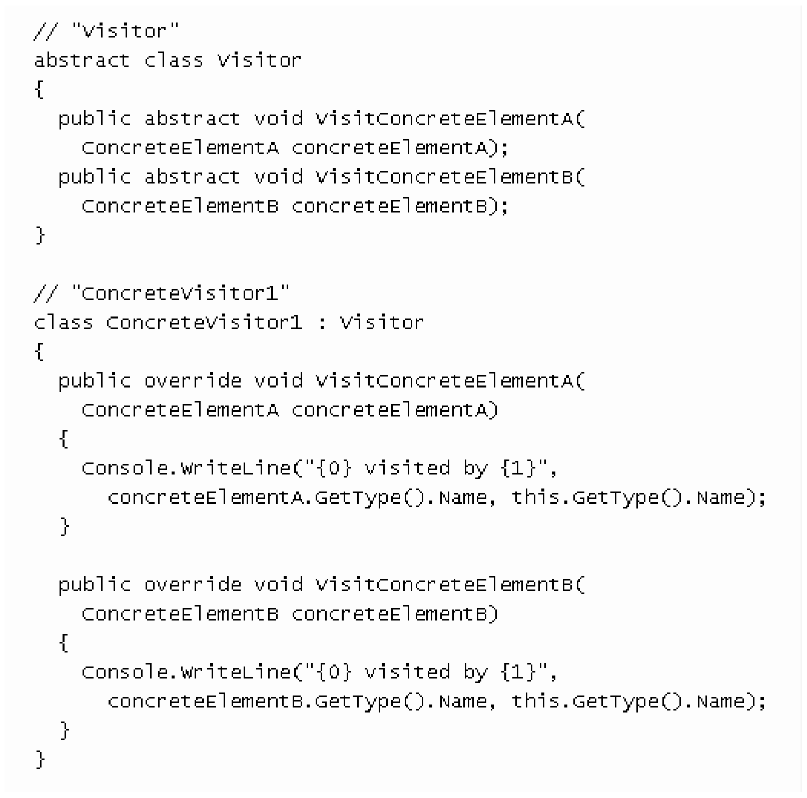
k. Visitor
   i. Represents an OPERATION to be performed on the elements of an object structure
   ii. The approach ENCOURAGES designing LIGHTWEIGHT element classes. Because processing functionality is removed from there list of RESPONSIBILITIES. New FUNCTIONALITY can easily be added to the ORIGINAL

inheritance HIERACHY by crating a new VISITOR subclass

iii. VISITOR lest you define a new operation without changing the classes of the elements on which it operates
iv. The classis technique of RECOVERING lost type information
v. PROBLEM : Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure
vi. You want to AVOID polluting the node classes with these operations.
vii. Visitor is not good for situations where VISITED classes are not stable. Every time a new COMPOSITE hierarchy derived class is added , every VISITOR derived class must be amended

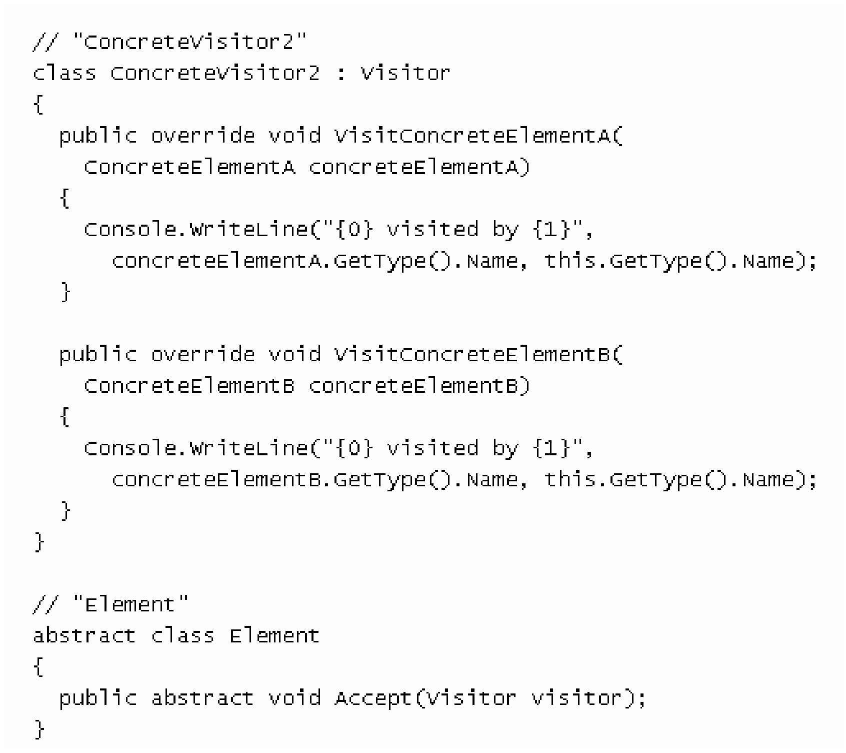
viii.


Client
«interface»
Element
ElementOne
+accept(in v : Visitor)
ElementTwo
v.visit(this);
«interface»
Visitor
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
«interface»VisitorOne
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
VisitorTwo
The concrete types of the Element and Visitor objects have been "recovered". Perform the work apropriate for their pair of types.



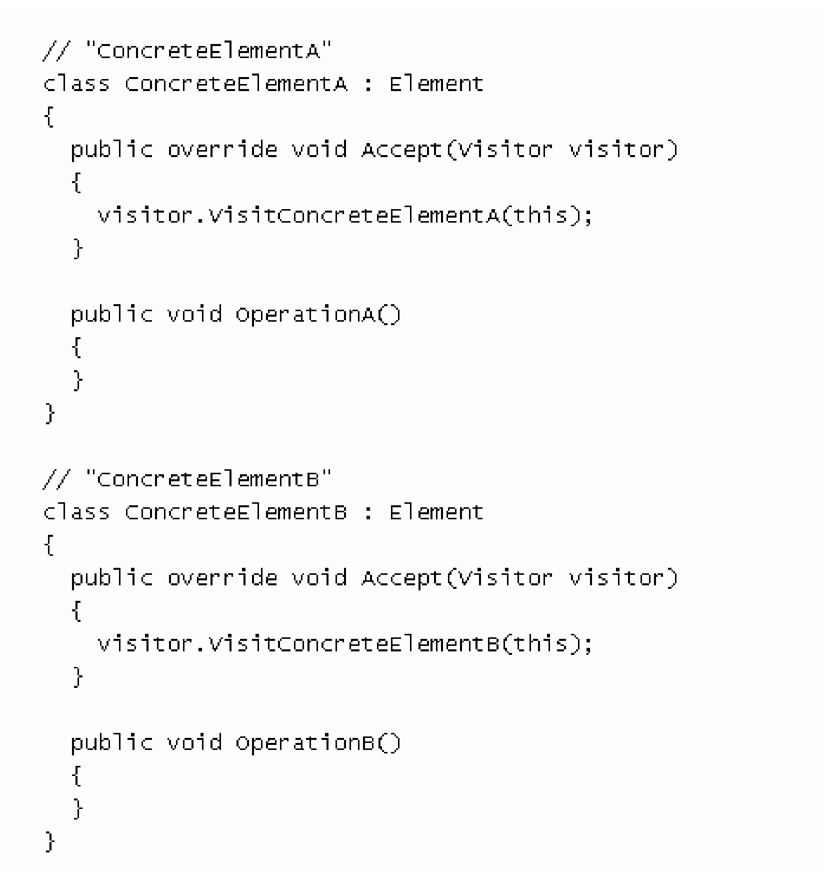
```
// "Visitor"
abstract class Visitor
{
  public abstract void VisitConcreteElementA(
    ConcreteElementA concreteElementA);
  public abstract void VisitConcreteElementB(
    ConcreteElementB concreteElementB);
}

// "ConcreteVisitor1"
class ConcreteVisitor1 : Visitor
{
  public override void VisitConcreteElementA(
    ConcreteElementA concreteElementA)
  {
    Console.WriteLine("{0} visited by {1}",
      concreteElementA.GetType().Name, this.GetType().Name);
  }

  public override void VisitConcreteElementB(
    ConcreteElementB concreteElementB)
  {
    Console.WriteLine("{0} visited by {1}",
      concreteElementB.GetType().Name, this.GetType().Name);
  }
}
```

ix.



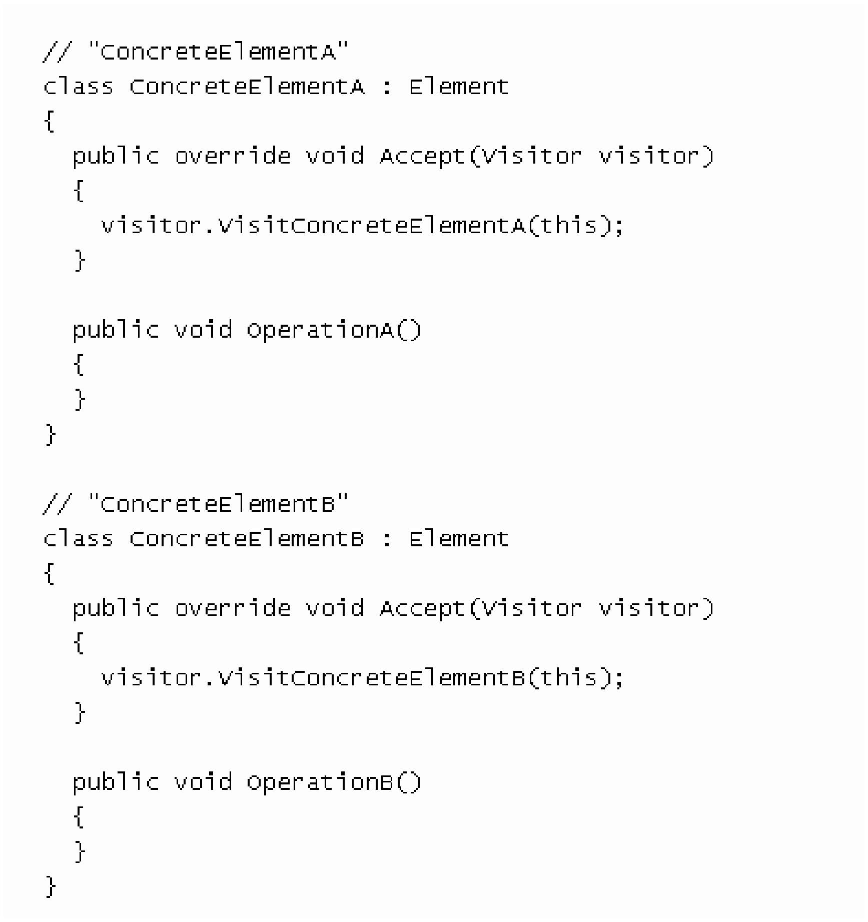
```
// "ConcreteVisitor2"
class ConcreteVisitor2 : Visitor
{
  public override void VisitConcreteElementA(
    ConcreteElementA concreteElementA)
  {
    Console.WriteLine("{0} visited by {1}",
      concreteElementA.GetType().Name, this.GetType().Name);
  }

  public override void VisitConcreteElementB(
    ConcreteElementB concreteElementB)
  {
    Console.WriteLine("{0} visited by {1}",
      concreteElementB.GetType().Name, this.GetType().Name);
  }
}

// "Element"
abstract class Element
{
  public abstract void Accept(Visitor visitor);
}
```

x.



```
// "ConcreteElementA"
class ConcreteElementA : Element
{
  public override void Accept(Visitor visitor)
  {
    visitor.VisitConcreteElementA(this);
  }

  public void OperationA()
  {
  }
}

// "ConcreteElementB"
class ConcreteElementB : Element
{
  public override void Accept(Visitor visitor)
  {
    visitor.VisitConcreteElementB(this);
  }

  public void OperationB()
  {
  }
}
```

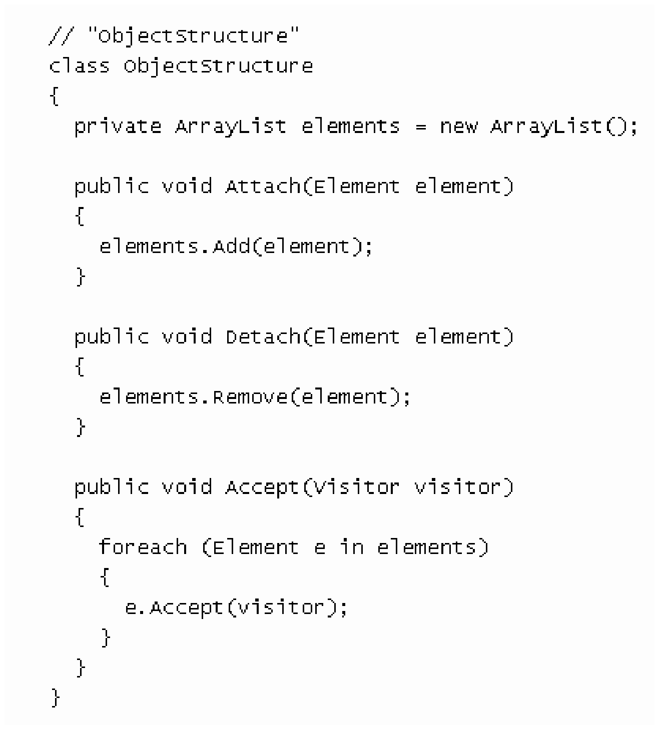
xi.



```
// "ConcreteElementA"
class ConcreteElementA : Element
{
  public override void Accept(Visitor visitor)
  {
    visitor.VisitConcreteElementA(this);
  }

  public void OperationA()
  {
  }
}

// "ConcreteElementB"
class ConcreteElementB : Element
{
  public override void Accept(Visitor visitor)
  {
    visitor.VisitConcreteElementB(this);
  }

  public void OperationB()
  {
  }
}
```

xii.

xiii.



```
// "ObjectStructure"
class ObjectStructure
{
  private ArrayList elements = new ArrayList();

  public void Attach(Element element)
  {
    elements.Add(element);
  }

  public void Detach(Element element)
  {
    elements.Remove(element);
  }

  public void Accept(Visitor visitor)
  {
    foreach (Element e in elements)
    {
      e.Accept(visitor);
    }
  }
}
```