Design Patterns

- ***Overview of Design Patterns***
  - Gang of Four
  - Composites, MVC/Model 2.
- ***Creational Patterns***
  - Factory/Abstract, Singleton, Prototype, Builder.
- ***Structural Patterns***
  - Adapter, Composite, Decorator, Bridge, Façade, Flyweight, Proxy
- ***Behavioural Patterns***
  - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor

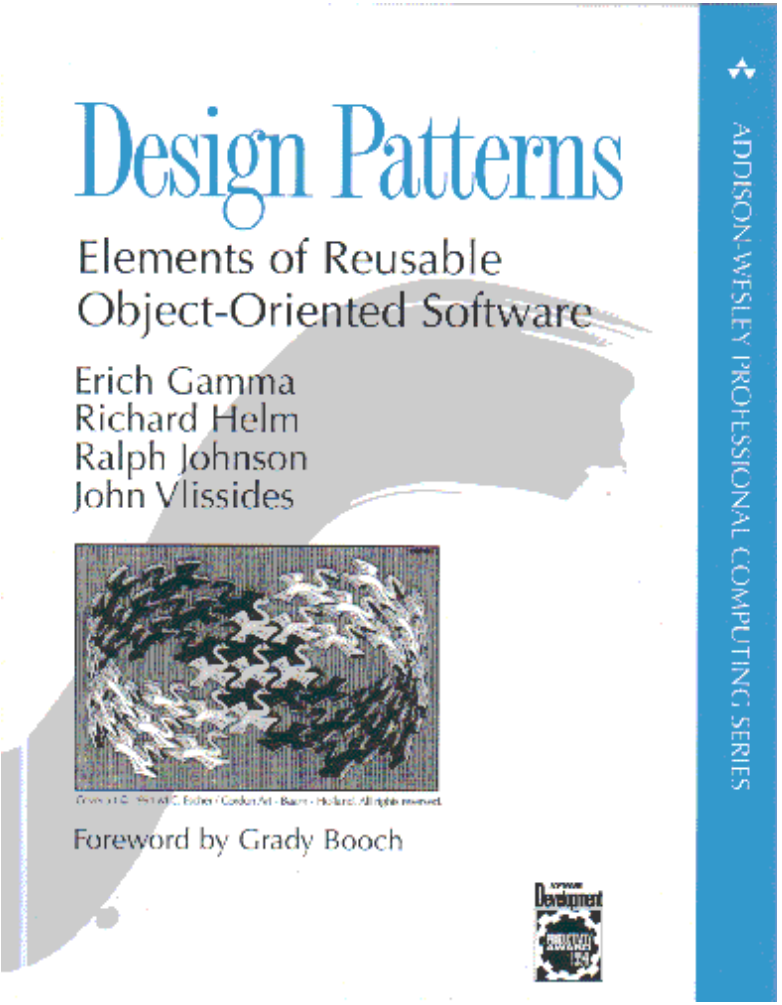
Design Patterns

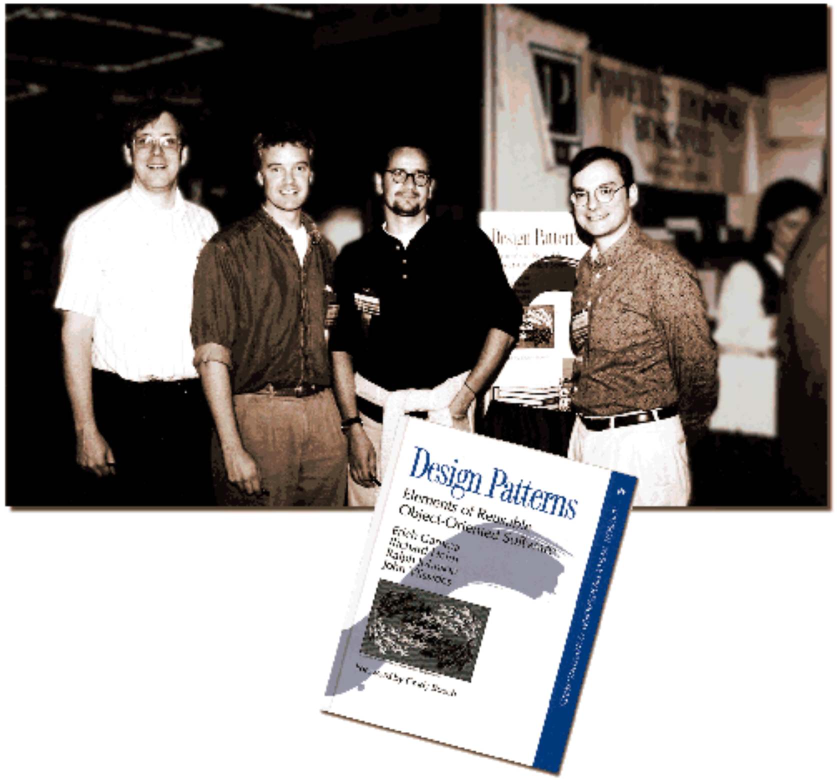
- *Convenient ways of reusing object oriented code* between programmes and programmers.
  - Catalog ***common***, ***proven***, ***useful*** interactions between objects.
- Dates back to 1980s (Smalltalk & C++ later).
  - Model View Controller (Krasner & Pope 1988).
  - 1995, ***Design Patterns, Elements of Reusuable Software*** (Gamma, Helm, Johnson, Vlissides - GoF) seminal work on patterns.
- *Describe how objects can communicate without becoming entangled in each other's data models and methods.*
  - It is the design of simple, but elegant, ***methods of communication*** that make design patterns important.
- *Patterns identify the and specify abstractions that are above the level of single classes and instance, or of components.* (Gamma, 1996)
- Patterns are *discovered rather than written*.
  - ***Pattern Mining:*** process of looking for these patterns.

What is a Design Pattern?

- A ***solution*** to a general software ***problem*** within a particular ***context***.
  - ***Context:*** A recurring set of situations where the pattern applies.
  - ***Problem:*** A system of forces (goals and constraints) that occur repeatedly in this context.
  - **Solution:** A description of communicating objects and classes (collaboration) that can be applied to resolve those forces.
- *Capture solutions that have evolved over time as developers strive for greater flexibility in their software.*
  - ***Class libraries*** - reusable source code
  - ***Components*** - reusable packaged objects
  - ***Patterns*** - generic, reusable design descriptions that are customized to solve a specific problem.
- Study of design patterns provides a common vocabulary for communication and documentation.
  - Provides a framework for evolution and improvement of existing patterns.

The Gang of Four (GoF)





Design Patterns

- *Fundamental reason for using patterns is to keep classes separated and prevent them from having to know too much about one another.*
  - ***Creational Patterns:***
    - Create object for you rather than having you instantiate the objects directly. Gives flexibility in deciding which objects need to be created for a given case.
  - ***Structural Patterns:***
    - Describe how classes and objects can be combined to form larger structures.
  - ***Behavioural Patterns:***
    - Help define the communication between objects and how the flow is controlled in a complex program.
  - ***Antipatterns***
    - Dark patterns, oxymoronic neologisms that describe bad practice and solutions to them. Examples include, God Object, Object Cesspool, Poltergeists and YAFL.

Design Patterns

- All of the design patters are based on the following high-level principles of object-oriented design:

Basic Principles of Good OO Design:

- Encapsulate what varies.
- Always program to an interface, not to an implementation.
- Favour object composition over inheritance.
- Strive for loosely-coupled designs between objects that interact.
- Classes should be open for extension, but closed for modification (Open-Closed Principle).
- Depend on abstractions, do not depend on concrete classes.

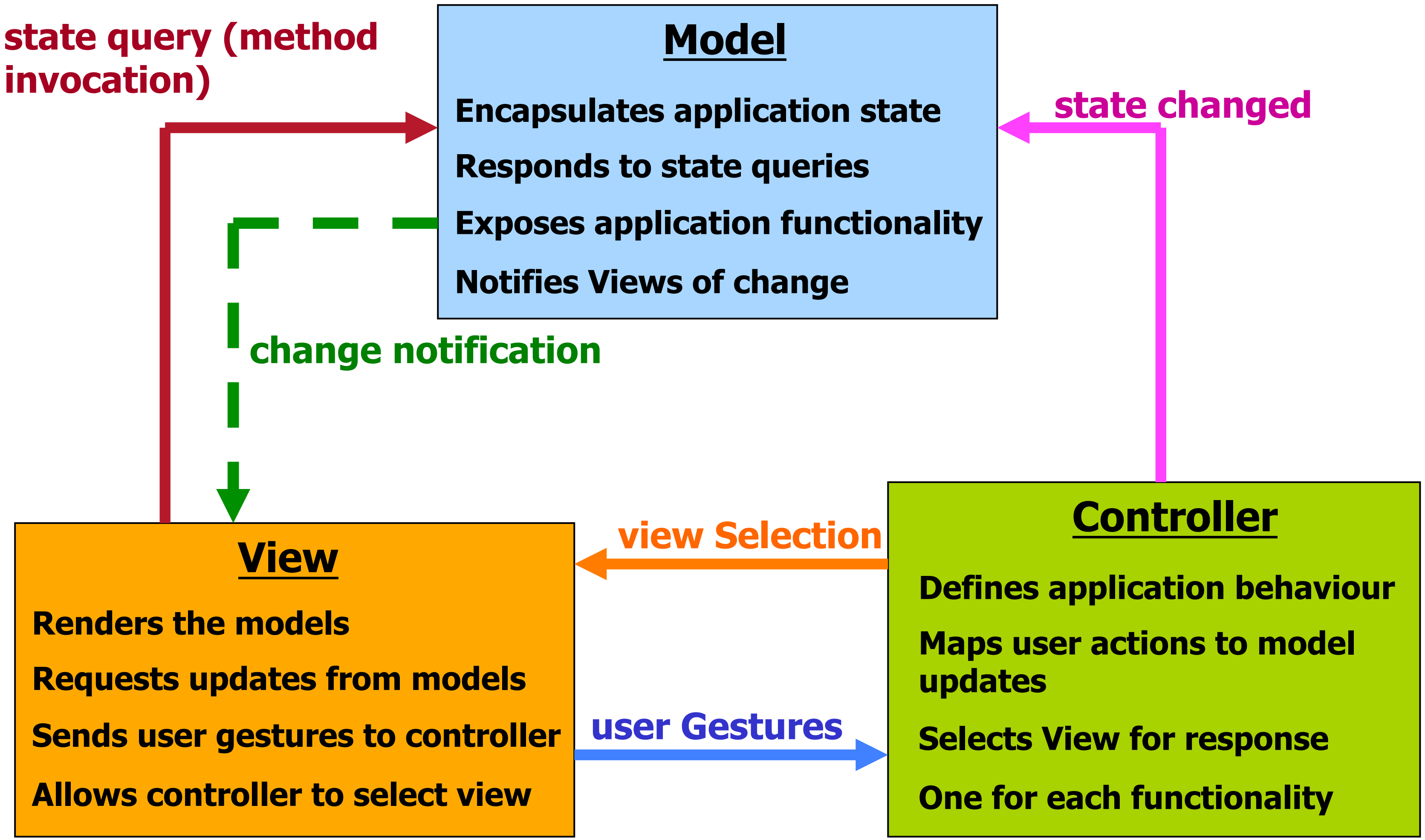
Composite Patterns

- Knowledge of patterns is not enough.
  - It is vital that you know when and how to use them.
- In reality, patterns are *often used together* and combined within the same design solution.
  - ***A compound pattern combines two or more patterns into a solution that solves a general or recurring problem.***
- The most widely used compound pattern is *Model-View-Controller* and its variant, Model 2 (web). More later...
  - ***MCV*** composed of strategy, composite and observer patterns.
  - ***JUnit*** (pre 4.0) framework based on command, composite and adapter patterns.
- *MVC is a paradigm for factoring code into functional segments to achieve reusuability and keep boundaries clear and clean.*
  - Model represents applications *raison d' être*. Controller mediates (push/pull) between view and model. Keeps coupling loose.

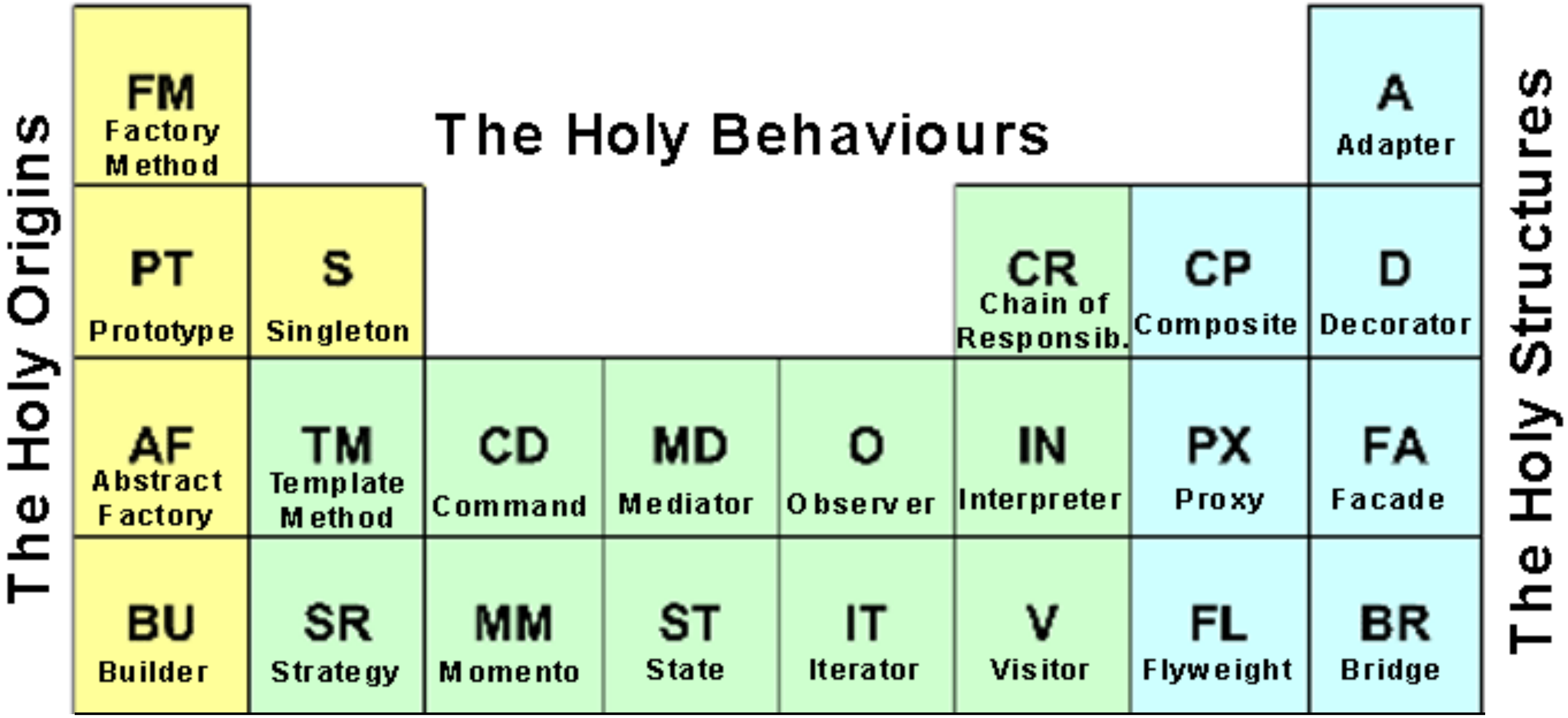
Model-View-Controller (MVC)

- *Increases reusuability by partially decoupling data presentation, data representation and application operations. Enables multiple simultaneous data views.*
  - Facilitates maintenance and extensibility by decoupling software layers from one another.
- Want to avoid applications with intertwined persistent data manipulation, application functionality and display code.
  - MVC separates application data from the way data can be viewed/ accessed and from the mapping between system events and application behaviour.
- ***Consists of Three Layers (Components):***
  - ***Model Component:*** represents the application data and methods that operate on that data (with no UI).
  - ***View Component:*** presents the data to the user.
  - ***Controller Comnponent:*** translates user actions into operations on the model.
    - Model in turn updates the View to reflect changes to the data.

Model-View-Controller (MVC)



Sacred Elements of the Faith…



Creational Patterns

- ***Deal with the best way to create instances of objects.***
  - Important because a program ***should not depend on how objects are created and arranged***.
- In Java, the simplest way to create an instance of an object is to use the ***new*** operator:

```
Student s = new Student();
```

  - This is essentially ***hard coding***.
- There are cases where the exact nature of the object that is instantiated *can vary with the need of the program* – e.g. An XML Parser.
  - Abstracting the creation process into a special "creator" class can ***make a program more flexible and general***.
  - ***Programming to an abstraction*** makes our code more flexible, robust and reusuable.

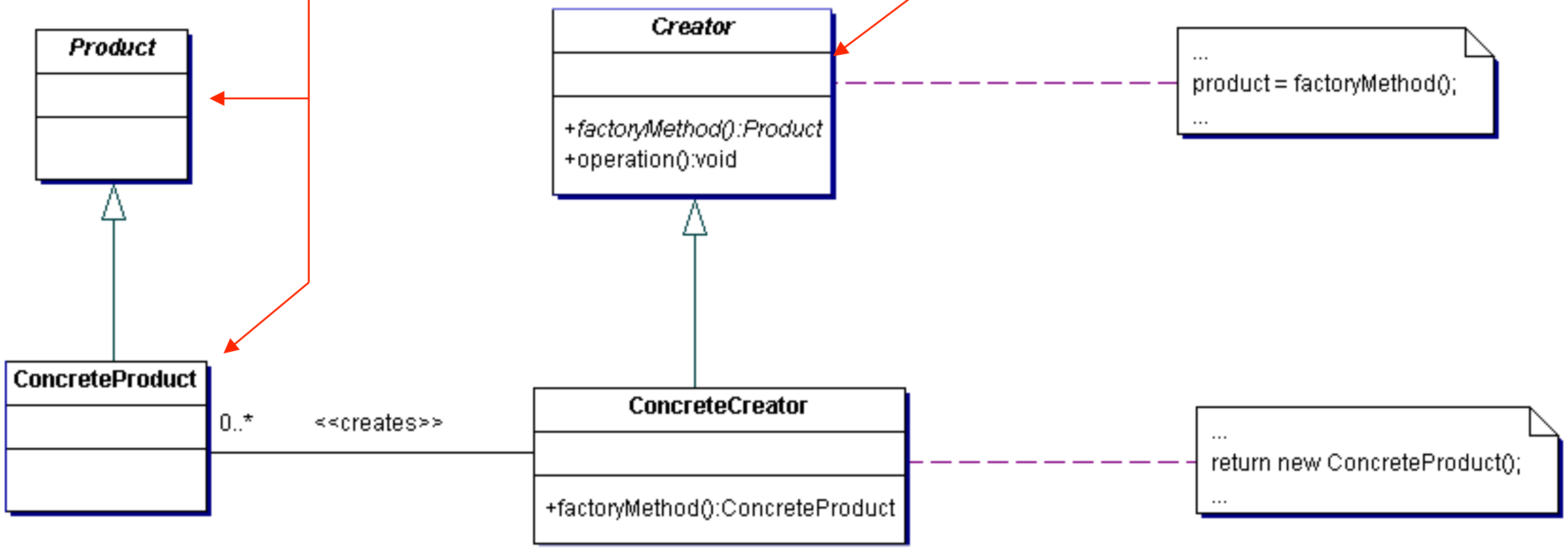
Factory Pattern

- *Defines an interface for creating an object, but lets subclasses which class to instantiate. Factory method lets a class defer instantiation to subclasses.*
  - Allows us to ***encapsulate the instantiation*** of concrete types.
  - Returns an instance of one of several possible classes, depending on the data provided to it.
- Factories *decouple* the implementation of a class from its use.
  - Allows us to vary the implementation without affecting dependencies from other classes.
  - ***Recall our basic principle:*** encapsulate what varies!
- Usually all classes it returns have a *common parent class* and common methods.
  - The factory method and the creator do not ***have to be*** abstract.
  - Each subclass performs a task differently and is optimised for different kinds of data.
- ***Factory pattern is actually an implementation of the DIP.***

Factory Pattern

All products must implement the same interface to that classes that use the products refer to the interface, not the implementation.

Creator contains implementations for all of the methods to manipulate products, except for the factory method.



ConcreteCreator responsible for creating one or more concrete products. Is the only class that has _knowledge of how to create_ products.

ConcreteCreator implements the factoryMethod() which actually produces products.

Factory Pattern

- Used in ***java.util.Calendar*** (abstract base class).

```
Calendar cal = Calendar.getInstance();
System.out.println("Time: " + cal.getTime());
```

- The *getInstance()* method of *Calendar* looks as follows:

```
public static Calendar getInstance(){
    Calendar cal = createCalendar(TimeZone.getDefaultRef(),
                                  Locale.getDefault());
    cal.sharedZone = true;
    return cal;
}
```

Factory Pattern

- ***Should consider using a factory pattern when:***
  - A class ***cannot anticipate*** which kinds of objects it must create.
  - A class uses its ***subclasses to specify*** which object it creates.
  - You want to ***localise the knowledge*** of which class gets created.
- *There are several similar variations on the factory pattern to recognise:*
  1. The ***base class is abstract*** and the pattern must return a complete working class.
  2. The ***base class contains default methods*** and is only subclassed for cases where the default methods are insufficient.
  3. ***Parameters are passed to the factory*** telling it which of several class types to return. In this case the may share the same method names but do something quite different.

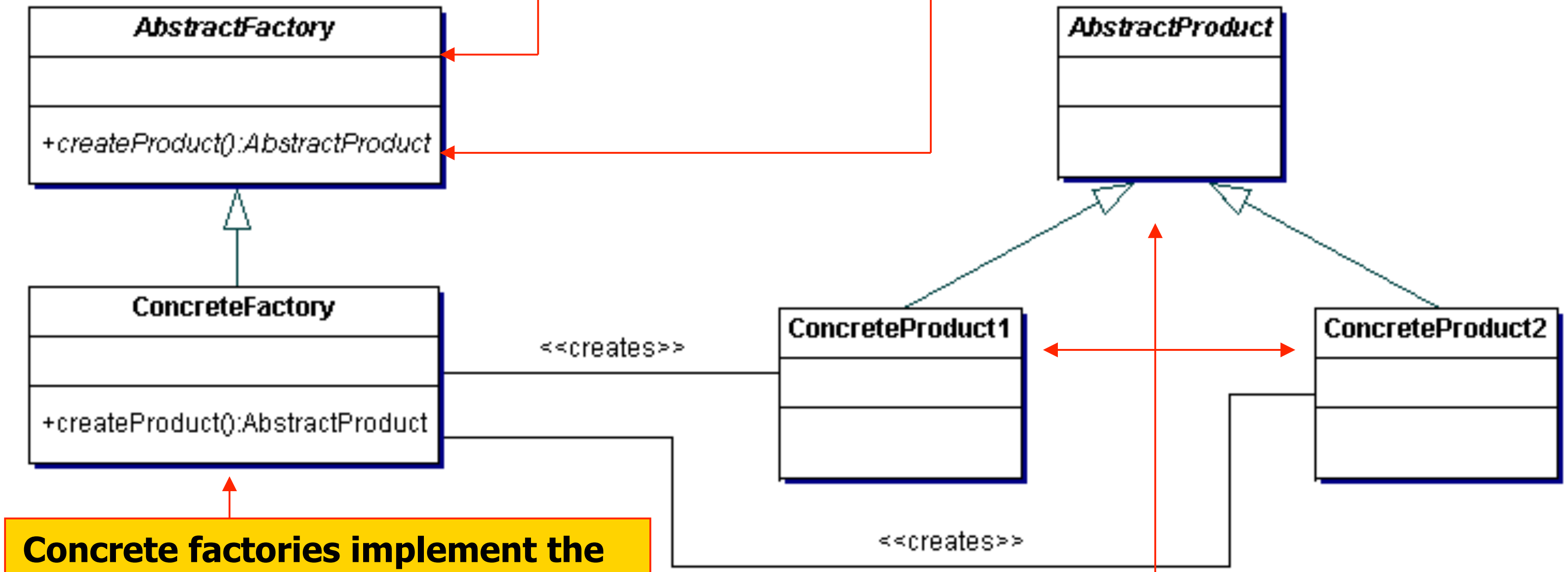
Abstract Factory Pattern

- *Provides an interface for creating families of related or dependent objects without specifying their concrete classes.*
  - Allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced.
- The *client is decoupled from any of the specifics* of the concrete products.
  - A ***level of abstraction higher*** than Factory.
  - ***An abstract factory returns one of several factories***.
  - A factory of factories! The methods of an abstract factory are often implemented as factory methods.
- A factory works through inheritance, an abstract factory through composition.
  - However, if we need to extend the set of related products, we will have to change the abstract interface.

Abstract Factory Pattern

Abstract factory defines an interface that each concrete factory implements, which is a set of methods for producing products.

Client is written against the abstract factory and then composed at runtime with an actual factory.



Concrete factories implement the different product families. To create a product, a client uses one of these factories so it never has to instantiate a product object directly.

Each concrete factory can produce an entire set of products.

Abstract Factory Pattern

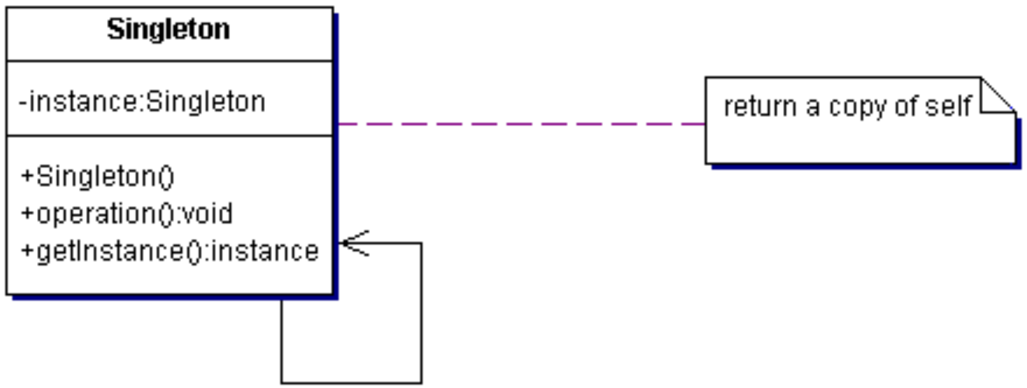
- Commonly used pattern in Java APIs, e.g. java.sql.

  Connection conn = DriverManager.getConnection(name, pwd);

  – Will load a Connection object for a specified DB type.
- ***DocumentBuilderFactory*** uses static ***newInstance()*** to return a ***DocumentBuilder*** factory.
- *Abstract Factory isolates the concrete classes that are generated*.
  – Class names are ***hidden in factory and client not required to know*** about them.
  – Allows us to ***change/interchange class families freely***.
  – Since factory only returns one kind of concrete class, do not have problem of inadvertently using classes from different factory families.
- Derived classes *can still have additional methods* to those of base class.
  – Means we have to ***test to check the type*** of class instance we have.

Singleton Pattern

- *The singleton pattern ensures that a class has only one instance, and provides a global point of access to it.*
  - Singleton manages instance of itself. To get an instance of the class, you must do so through the class itself.
- Provides a means of ensuring that there is one and only one instance of a class.



- Can implement using a ***static boolean*** variable.
- Better to throw an Exception if attempt to instantiate class twice.

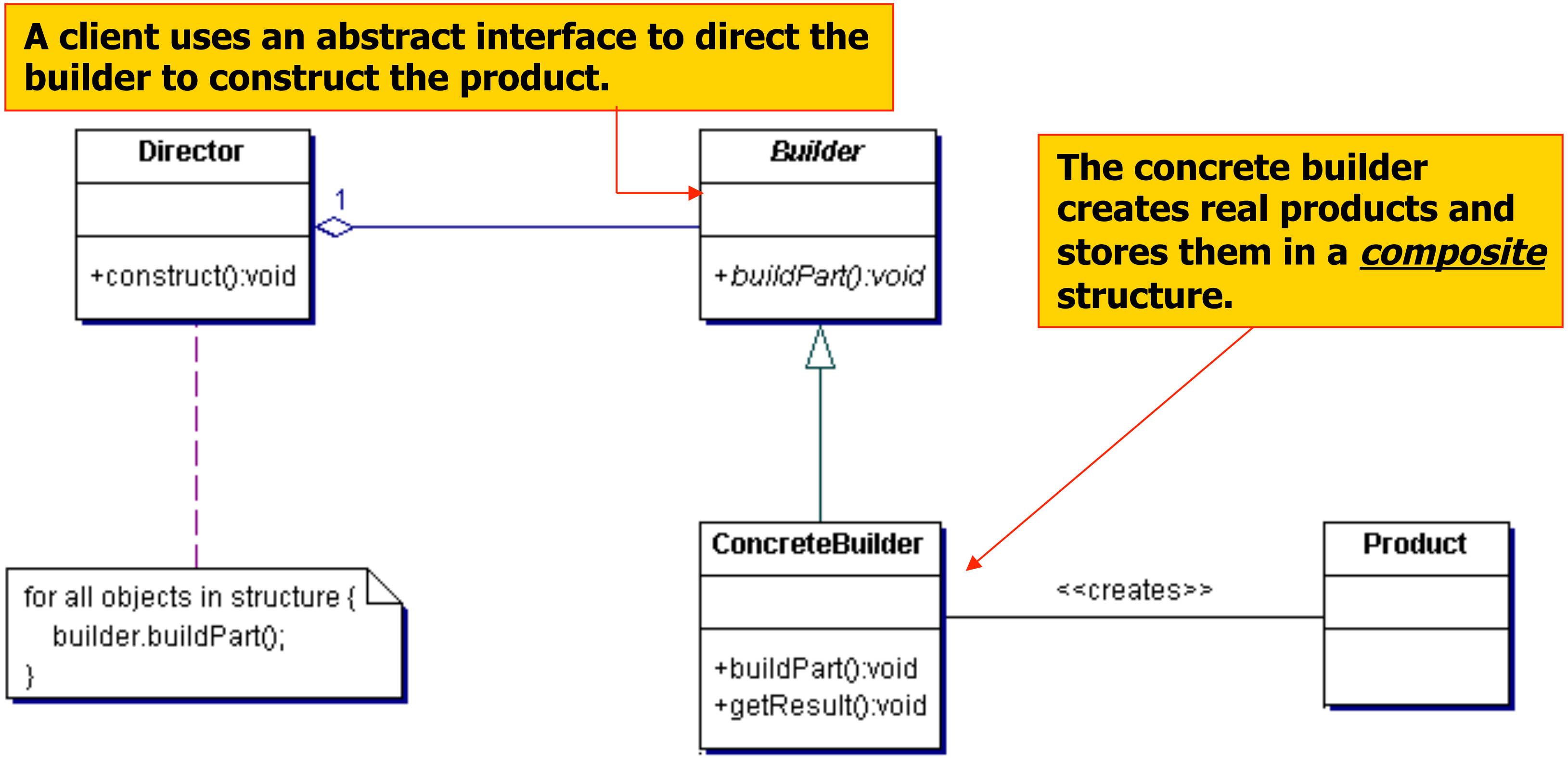
Singleton Pattern

- Another approach is to declare a *private constructor* and use a *static public method* to create a single instance:
- ***java.lang.System*** and ***java.sql.DriverManager*** use Singleton pattern.
  - Not implemented using approach recommended by GoF but with static methods. Can make Singletons *final* to prevent subclassing.
- Singleton can cause problems in multithreaded applications.
  - Can synchronize *getInstance()* to force threads to wait for access to methods. ***But*** – this can cause a ***major overhead***. Fine if performance of *getInstance()* not critical to application.
- Can also solve problem by creating an *eager* (as opposed to a *lazy*) singleton that initialises static field using the *new* operator.
  - The JVM will always create the unique instance of the singleton when the class is loaded. Can also double-checked lock by applying the ***volatile*** keyword.

Builder Pattern

- *Separate the construction of a complex object from its representation so that the same construction process can create different representations.*
  - Builder Pattern gives user the choice to create the type of object they want - ***but the construction process is the same***.
- Builder pattern involves *Builder* and *Director* objects.
- ***Builder:*** *abstract interface* for creating parts of some Product object to be constructed.
  - ***Concrete subclasses*** of builder know how to put together different kinds of parts for different kinds of products.
  - ***Don't know what it is they are building.***
- ***Director:*** knows what needs to be built. Knows recipe for how to build it.
  - ***Doesn't know how to put individual parts together.***
  - Director uses Builder interface to direct builders to put the various parts together.

Builder Pattern



- Builder focuses on constructing a <u>*complex object step by step*</u>.
  - Encapsulates way a complex (***<u>composite</u>***) object is constructed.
  - Constructs objects in a ***<u>multistep varying process</u>***, as opposed to 1-step factories. Hides internal representation of product from client.

Builder Pattern

- *Subtle difference* between Builder and Factory Patterns.
  - Client uses ***abstract factory*** class methods to create its own object from a family of products.
  - ***Client instructs the builder class*** on how to create the object and then asks it for the result. ***How the class is put together is up to the Builder class***.
  - Requires more ***domain knowledge*** than when using a factory.
- Builder lets us *vary the internal representation* of product it builds.
  - Also ***hides details*** of how product is assembled.
- Specific builder is independent of other builders & rest of program.
  - ***Improves modularity*** and makes addition of other builders simple.
- Step by step nature gives *finer control* over product builder creates.

Prototype Pattern

- *Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.*
  - Used when creating class instance is either expensive or complicated.
- Basic idea is than when passed an object, we can *use that object as a template* for creating a new object.
  - Might not know the ***implementation details*** of the object as some data may not me available via accessor methods.
  - Might not even know ***which specific class*** is being instantiated.
- Simple to do in Java by implementing *Cloneable*.
  - ***clone()*** – requires an object to create an object.

    Student s1 = (Student) s0.clone(); //requires a cast!
  - ***clone()*** is protected. Restricted to package or same class.
- Can only clone objects that implement ***Cloneable***. Object that cannot be cloned throw a ***CloneNotSupported*** exception.
  - Note that *Cloneable* returns a shallow copy of a class.

Prototype Pattern

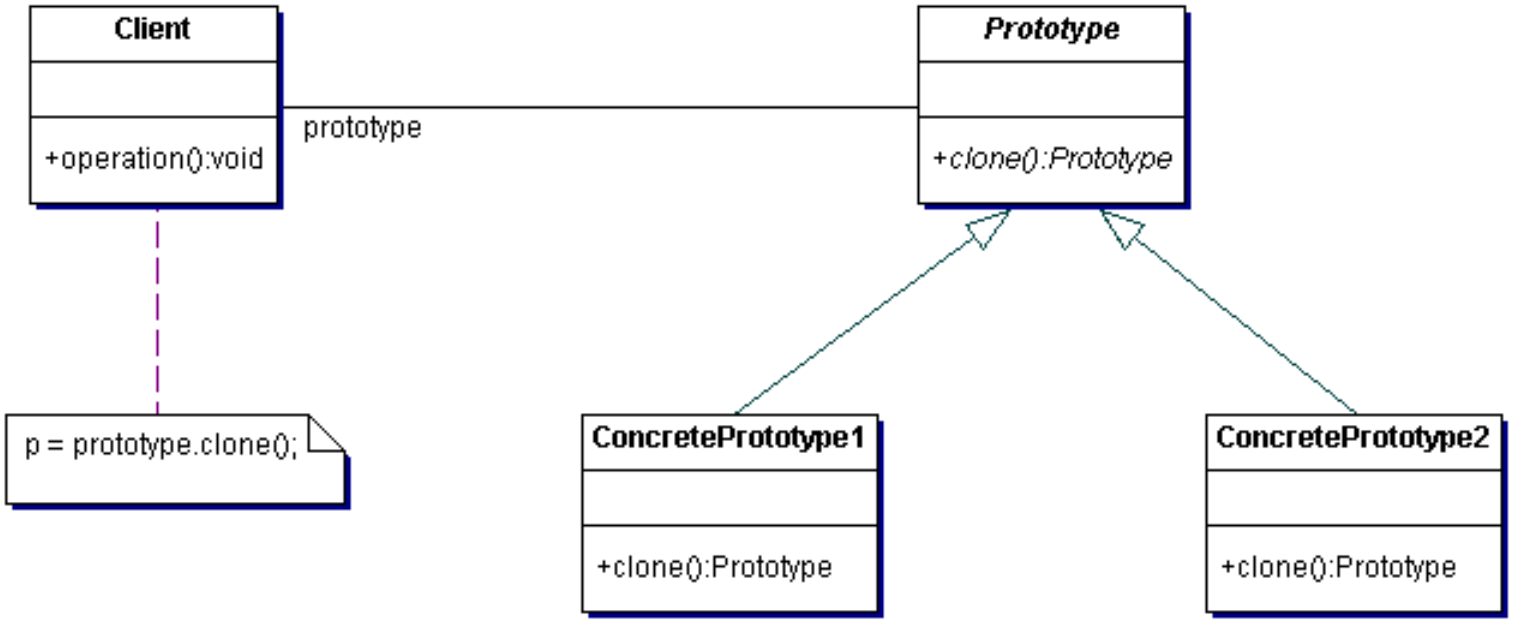
- Can solve restricted protected (package) access to clone by packaging clone method inside a class where it can access the real *clone()*.

```
public Student make(){
   try{
         return super.clone();
   }catch(Exception e){...}
}
```

  - Newly created object can be manipulated using instance variables.
- Real power of prototype comes when you *don't know what you're actually cloning.*
  - If each prototypical object implements a common interface.
  - For example, could have FileCustomer, RDBCustomer and EJBCustomer that implement *Customer*.
  - Pattern will work for *any* type of Customer, no matter how the data is stored: *manager.storeCustomer(Customer c)*

Prototype Pattern

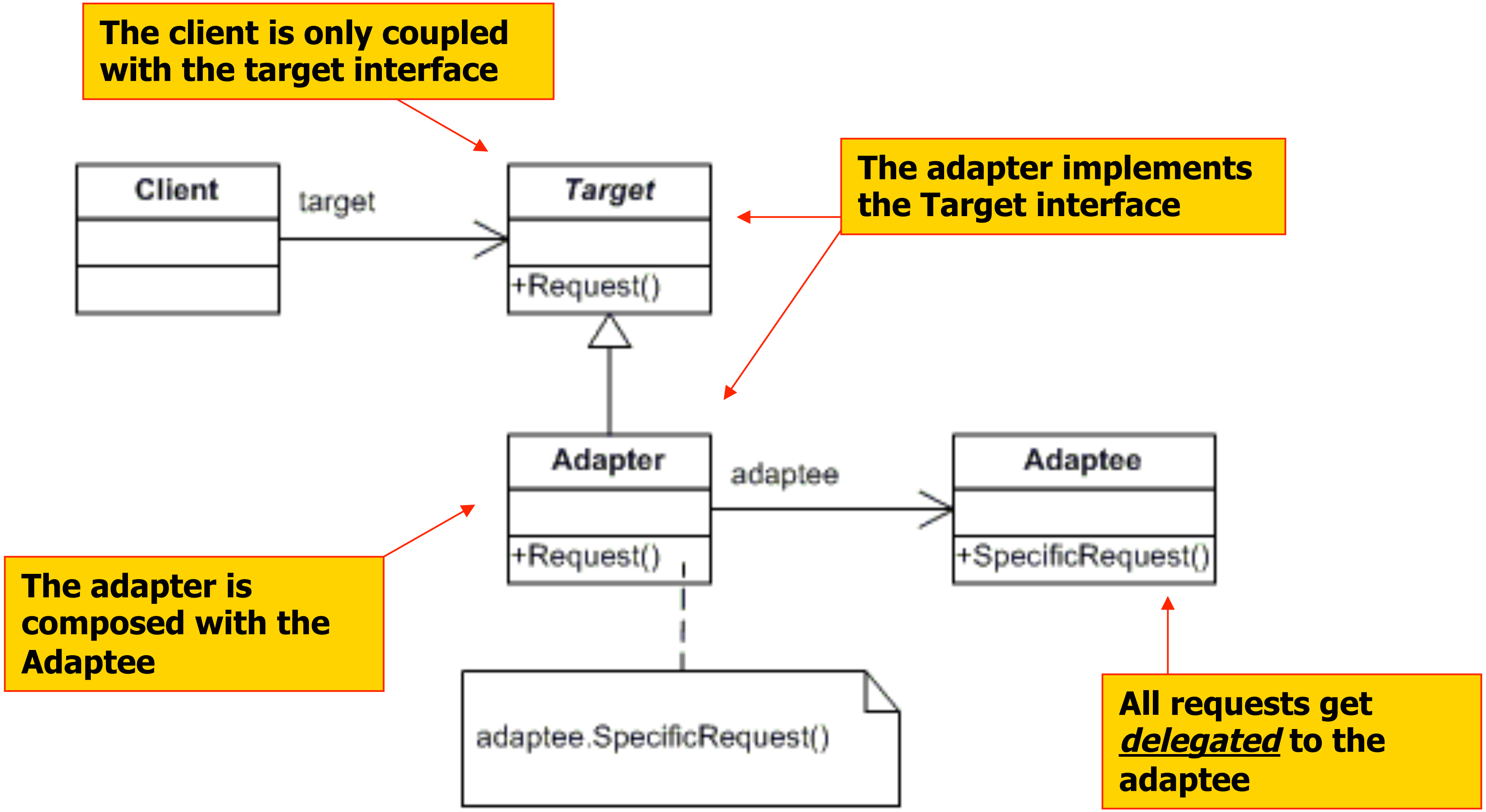
- Prototype pattern allows us to add/remove classes at runtime.
- ***Problems implementing deep-clones*** if all objects in clone tree not serialisable.
  - Classes with ***circular references cannot be cloned***.



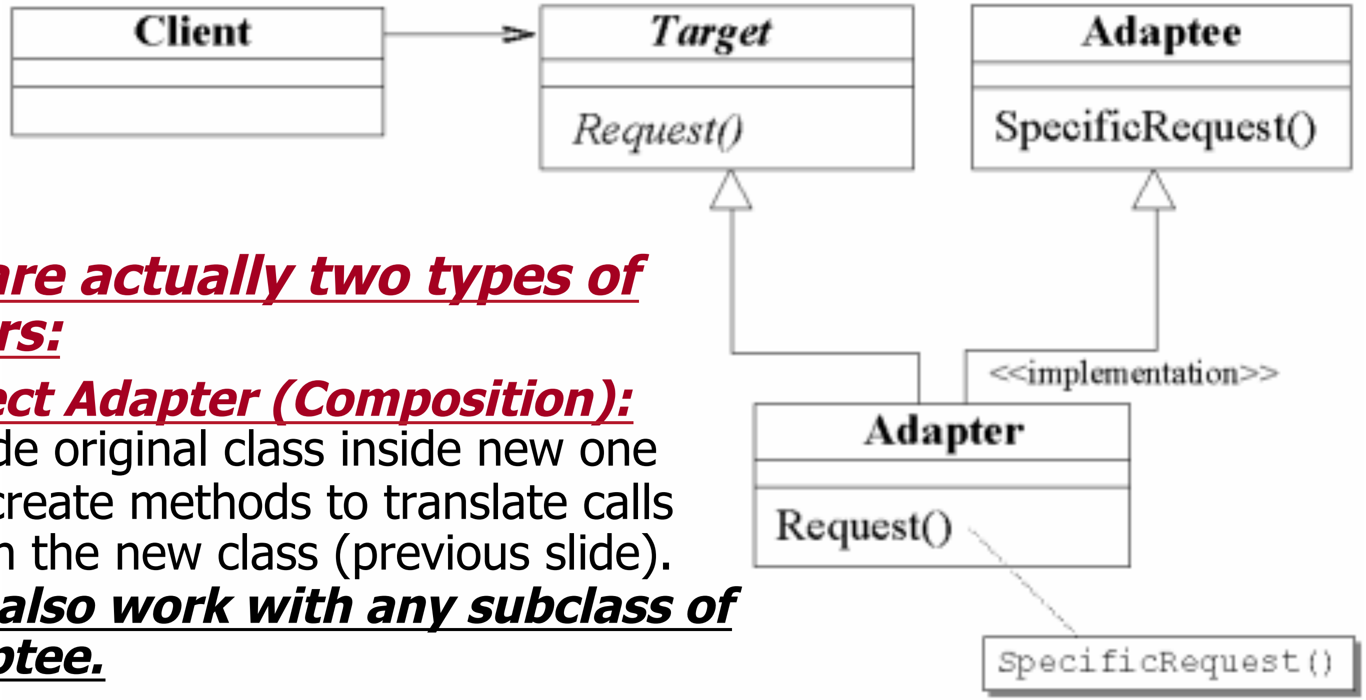
Adapter Pattern

- *Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.*
  - Basically a glue class that lets us put a square peg in a round hole.
- Use adapters to *get unrelated classes to work together* in a single programme.
  - Write a class that has a desired interface and make it communicate with the class that has a different interfaces.
- The adapter *decouples the client* from the implemented interface.
  - If we expect the interface to change over time, the ***change is encapsulated in the adapter***.
  - Means that a ***client doesn't have to be modified*** each time it has to operate against a different interface.
- Adapter binds a client to *an interface, not an implementation*.
  - Can add new implementations, as long as they adhere to the interface.

Adapter Pattern



Adapter Pattern



- ***There are actually two types of adapters:***
  - ***Object Adapter (Composition):*** include original class inside new one and create methods to translate calls within the new class (previous slide). ***Can also work with any subclass of Adaptee.***
  - ***Multiple-Inheritance):*** Instead of using composition to adapt the Adaptee, the Adapter subclasses ***both the Adaptee and the Target classes***.
    - Class adapter will not work with subclasses of Adaptee.

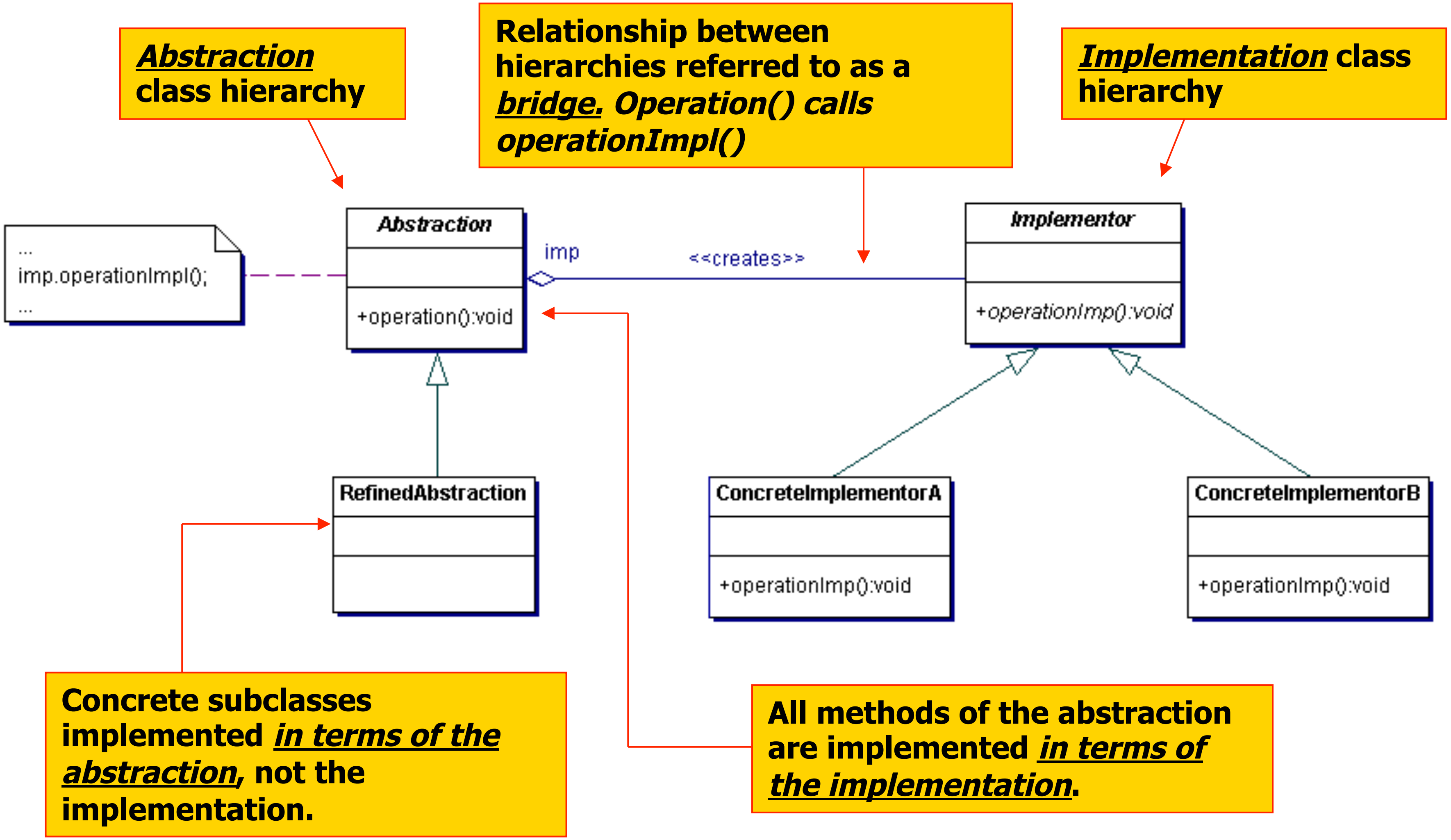
Adapter Pattern

- ***<u>Differences between class and object adapters:</u>***
  - ***<u>Class Adapters:</u>***
    - Won't work when you want to include a class and all of its subclasses (specify base class when we create adapter).
    - Lets the adapter change some of adapted class methods, but still allows others to be used.
  - ***<u>Object Adapters:</u>***
    - Allows subclasses to be adapted by simply passing them in as constructor arguments.
- *WindowAdapter* in AWT implements *WindowListener*.
  - Allows us a close a window without have to implement all the methods defined by WindowListener.
- Adapters like this are common in Java when a simple class can be used to encapsulate a number of events.
  - *MouseMotionAdapter*, *ComponentAdapter*, *FocusAdapter*.

Bridge Pattern

- *Decouple an abstraction from its implementation so that the two can vary independently.*
  - Allows us to vary abstractions, as well as implementations.
  - Done by placing abstractions and implementations in separate class hierarchies.
- Different intent to Adapter which exists only to adapt the interface of one class to another. ***A Bridge is by design. An Adaptor is a patch.***
  - Decompose component's interface and implementation into orthogonal class hierarchies.
- Use Bridge to ***separate interface from implementation***.
  - Moves abstract operations that an abstraction relies on into a separate interface.
  - Separate system-dependent from system-independent code.

Bridge Pattern



Bridge Pattern

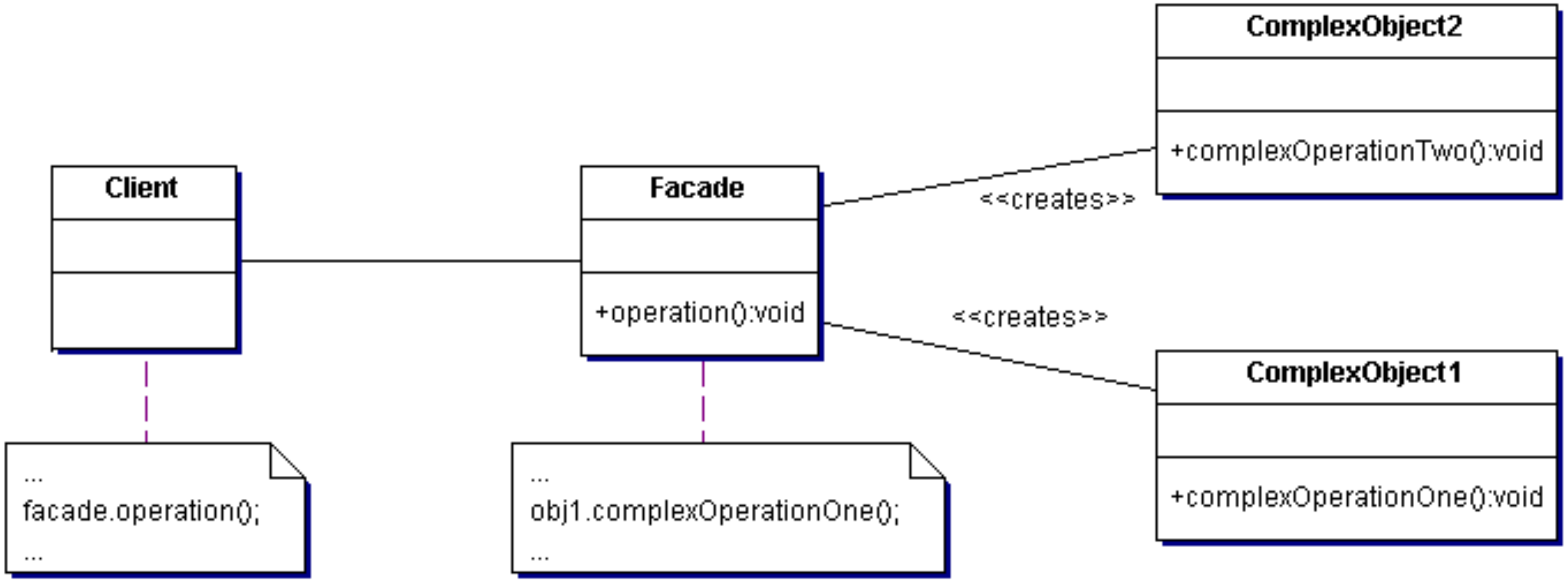
- Bridge saves us having to deal with a *Cartesian Product* of code:
  - If half the code is OS-dependent and half the code is CPU-dependent, then we have (#OSes)*(#CPUs) implementations to write.
  - If we use a bridge, we have (#OSes)+(#CPUs) to write.
- *Bridge decouples an implementation so that it is not bound permanently to an interface.*
  - Abstraction and implementation ***can be extended independently***. Can ***vary either side*** of the two hierarchies independently.
  - Changes to the concrete abstraction ***do not affect the client***.
- Used in graphic and windowing systems that need to run over multiple platforms.
  - Useful when we want to vary an interface and an implementation in different ways.
  - BUT, ***increases complexity***. More class hierarchies.

Façade Pattern

- *Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher level interface that makes the subsystem easier to use.*
- Used to create higher-level subsystems from lower-level building blocks. *Provides a simplified interface to subsystem*.
  - ***A semantic wrapper for existing objects.***
  - ***Hide complex interface*** of interactions between building blocks.
- *Façade object placed between client and group of objects.*
  - ***Shields clients*** from subsystem components, reducing the number of objects that clients deal with. Makes subsystem easier to use.
  - ***Promotes weak coupling*** between subsystem and clients.
- Does not prevent applications from using subsystem classes if they need to.
  - ***Can choose between ease of use and generality***.

Façade Pattern

- Swing's *JOptionPane* is a façade that creates different types of dialogs and displays them.
  - Provides low-level functionality for creating, populating, and showing windows.
  - High-powered features such as glass panes and choosers for selecting files and colours.



Façade Pattern

- Façade pattern based on the *Principle of Least Knowledge*:
  - Aka the ***Law of Demeter (LoD)***: Greek: distribution-mother and goddess of agriculture.

Talk only to your immediate friends

- Principle guides us to ***reduce interactions between objects*** to a few close "friends".
  - Means that we design systems where the number of interactions and coupling between objects is limited.
- Principle tells us that *a method should only invoke methods that belong to*:
  - The same object.
  - Objects passed in as a parameter to the method.
  - Any object that the method creates or instantiates.
  - Any components of the object (any instance variables).

Façade Pattern

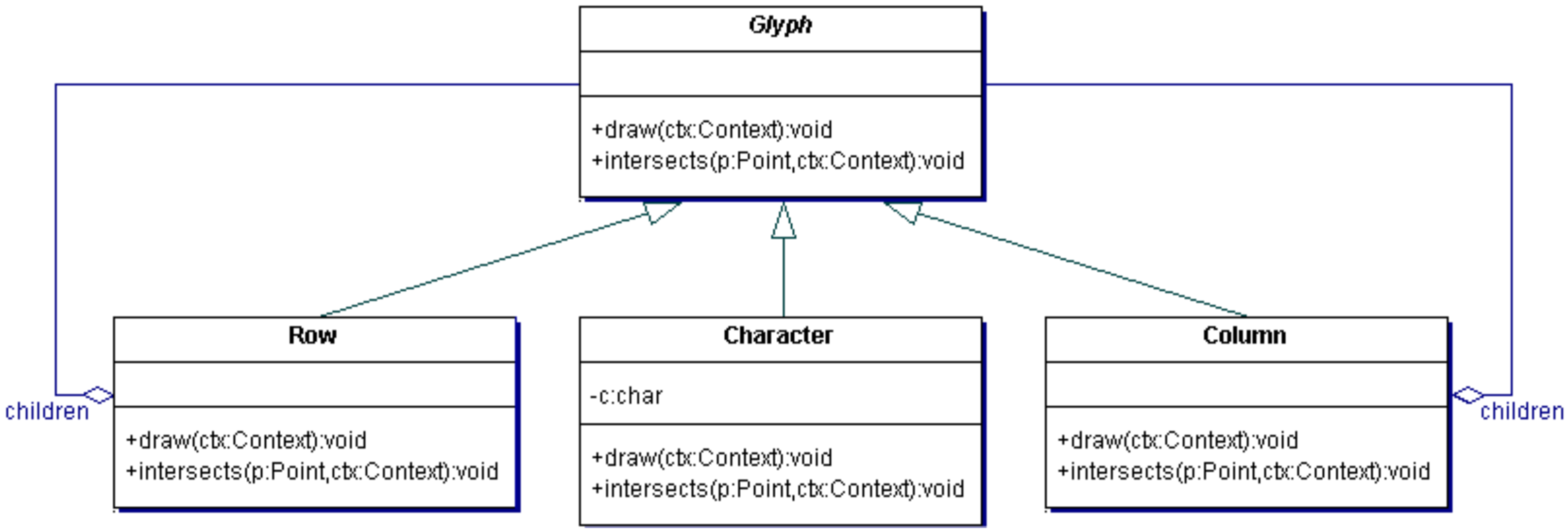
- ***Note:*** *principle tells us not to call methods on objects that were returned from calling other methods.*
  - Principle ***reduces dependencies between objects***. This reduces software maintenance.
  - But, results in ***wrapper classes*** that handle method calls to other components (possible Poltergeists). Can cause increased complexity, development time and reduced runtime performance.
- Principle of Least Knowledge a *better term to use* than LoD.
  - A law is something that is ***always*** true, and therefore should always be obeyed/applied.
  - Principles however are not laws and should only be applied where they are useful. All factors should be considered before applying them. ***Principles provide guidance. Laws are dictats.***

Flyweight Pattern

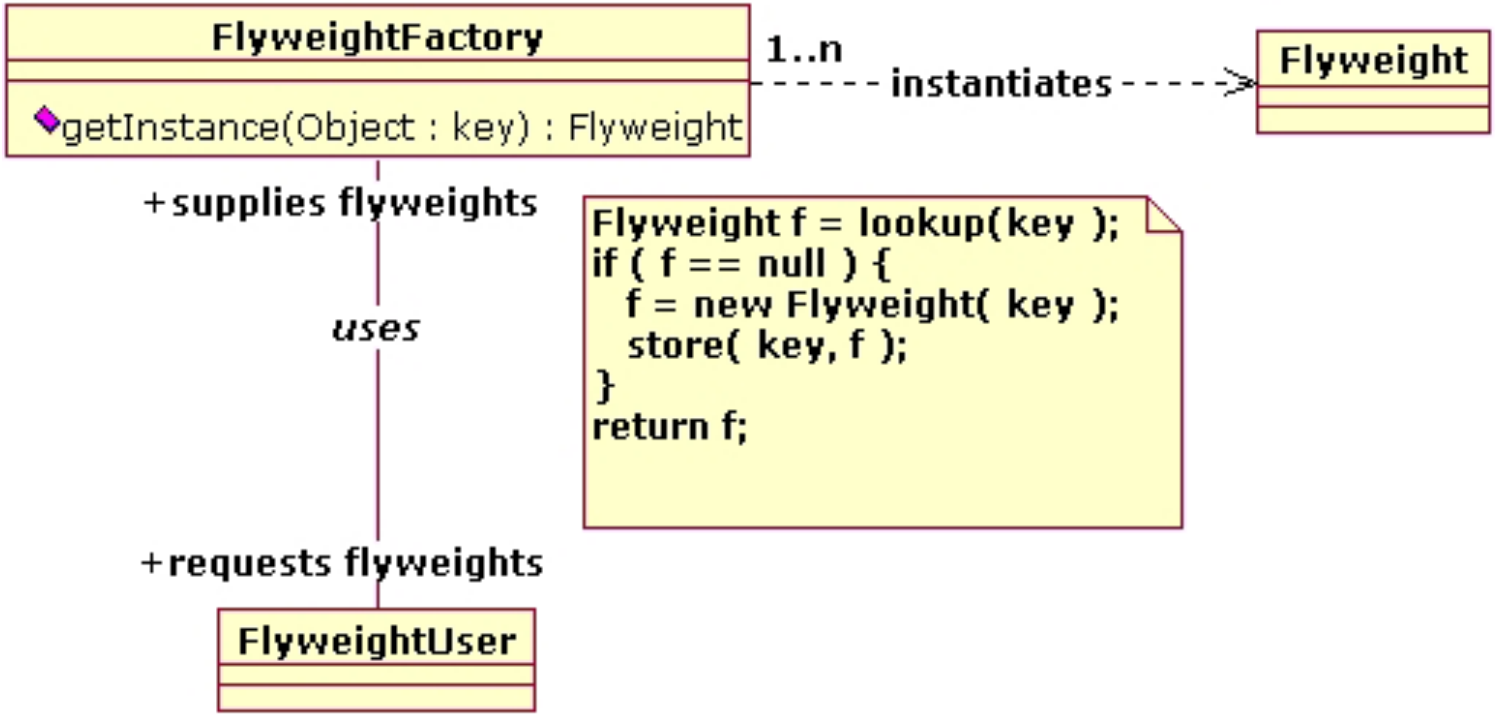
- *Use sharing to support large numbers of fine-grained objects efficiently.*
- May have a *large number of small class instances* to represent data. Instances may only differ by a few parameters.
  - *Move variables outside class* and pass them in as part of a method call when required. ***Number of separate instances greatly reduced.***
  - ***Instrisic Data:*** makes instance unique.
  - ***Extrinsic Data:*** can be passed in as arguments.
- Flyweights are *sharable instances of a class*.
  - Number of instances (ATGC) are decided as class instances are required. Usually accomplished with a ***FlyweightFactory*** class.
  - ***Factory class usually is a Singleton*** – needs to keep track if particular instance has yet been created.
- Two instances of a String constant with same characters could refer to the same storage location.
  - Can use == to determine object reference: ***if(string1==string2)***

Flyweight Pattern

- Flyweight is *<u>ideal for small fine-grained classes</u>* (individual craracters, screen icons).
  - May introduce run-time costs associated with transferring, finding, and/or computing extrinsic state. Off-set by space savings.
- Storage savings depend on reduction in total number of instances by sharing, amount of intrinsic state per object and whether extrinsic state is computed or stored.



Flyweight Pattern



- Flyweight used extensively in Swing API.
  - Cell renderers for JLists and JTables use flyweight components.
- Prevents logical instances of a class from being able to behave independently of other flyweight instances.

Flyweight Pattern

- If we wanted to add many Tree objects to a landscape design.
  - Might be tempted to create many tree objects that maintain their own state. May cause memory problems!
- Create a single, flyweight, *state-free* Tree object.
  - Store all the state for virtual tree objects in a 2D array managed by a TreeManager object.

Tree
xCoord   ***yCoord***   ***age***

display(x, y, age){
 //Use X-Y Coords
 //& complex age
 //related calculations
 }

TreeManager
treeArray

displayTrees(){
 //For all trees{
 //Get array row
 display(x, y, age);
 }
 }

Tree

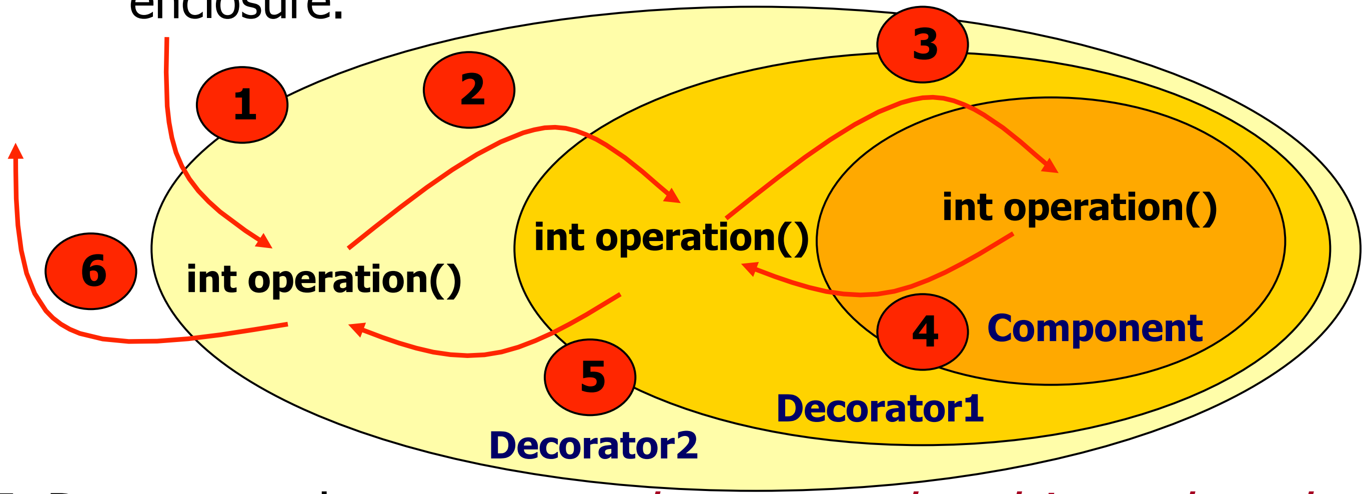
display(x, y, age){
 //Use X-Y Coords
 //& complex age
 //related calculations
 }

Decorator Pattern

- *Attach flexible additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.*
- Provides a *way of using composition instead of inheritance* to extend the behaviour of an object at run-time.
  - Can modify the behaviour of individual objects without having to create a new derived class.
- *Derive any number of specific Decorators from main Decorator.*
  - Each specific class provides some sort of decoration.
  - ***Decorator classes should be abstract. All working decorators (concrete implementations) derived from this***.
- More flexible than *static (compile-time) inheritance* where all subclasses must inherit the same behaviour.
  - Responsibilities added/removed at ***run-time through composition***. Can add new functionality without altering existing code (OCP).
  - Avoids feature-laden classes high up class hierarchy.

Decorator Pattern

- A decorator and it's component are ***not identical.***
  - *Tests for object type will fail*. *Decorator* acts as a transparent enclosure.



- Decorators have *same subtypes as the objects they decorate*.
  - Can use one or more decorators to wrap an object. Can pass around decorated object in place of the (wrapped) original object.

Decorator adds its own behaviour either before and/or after delegating to the object it decorates.

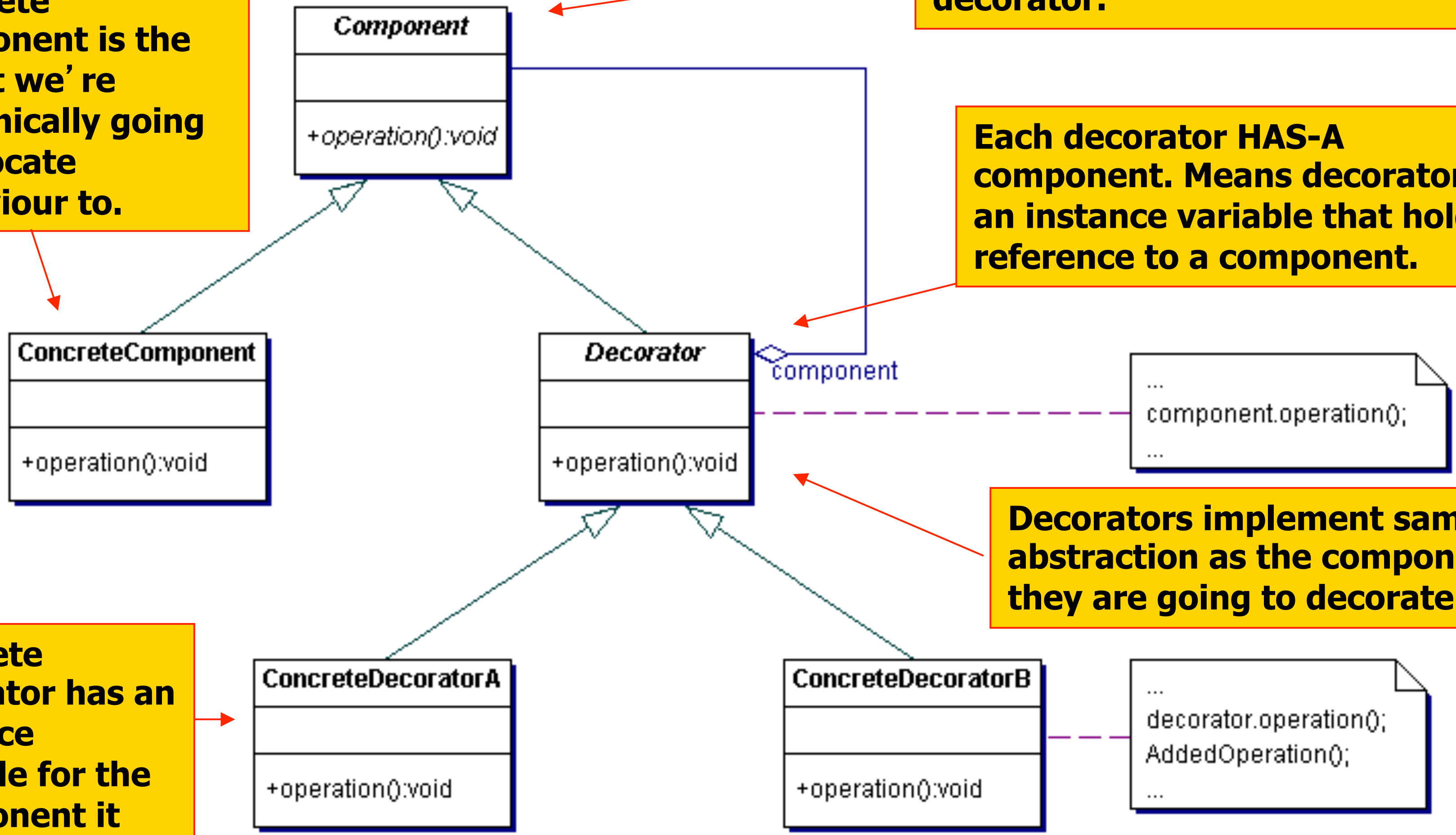
- Can *decorate objects dynamically at run-time* with many decorators.

Decorator Pattern

Each component can be used on its own or wrapped by a decorator.

Concrete component is the object we're dynamically going to allocate behaviour to.

Each decorator HAS-A component. Means decorator has an instance variable that holds a reference to a component.



Decorators implement same abstraction as the component they are going to decorate.

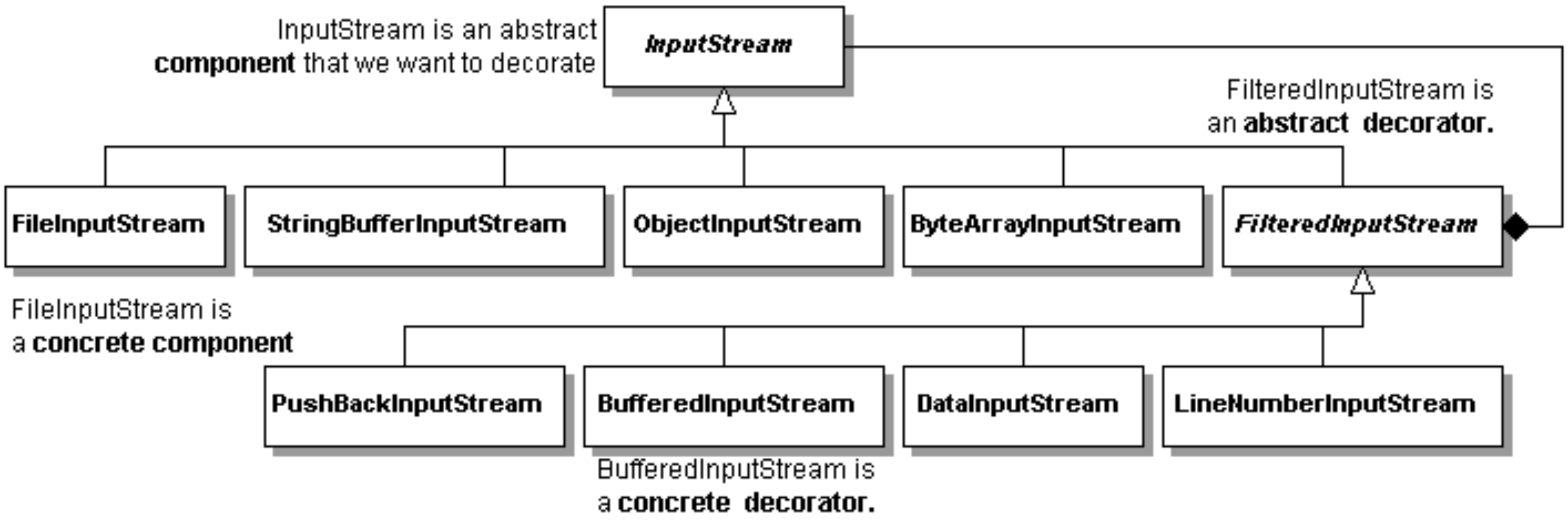
Concrete decorator has an instance variable for the Component it decorates

Decorators can extend the state of a component. Can also add new methods. These usually called after invoking existing method in component.

Decorator Pattern & java.io

- *FilterInputStream* overrides methods of *InputStream* with versions that pass requests to underlying input stream (see handout).
  - Subclasses of *FilterInputStream* ***may further override or provide additional methods***.
- *FilterInputStream is a Decorator that can be wrapped around any InputStream class.*
  - ***Abstract class. Does not do any processing.*** Provides a layer where the relevant methods have been duplicated.
  - ***Forwards calls*** to the enclosed parent stream class.
- Subclasses of FilterInputStream include:
  - ***BufferedInputStream:*** Adds I/O buffering to streams.
  - ***CheckedInputStream:*** Keeps a checksum of bytes as they are read.
  - ***DataInputStream:*** Reads primitive types from input stream.
  - ***DigestInputStream:*** Computes MessageDigest of input stream.
  - ***InflaterInputStream:*** Methods for uncompressing data.
  - ***PushbackInputStream:*** Buffer to “un-read” data if error detected.

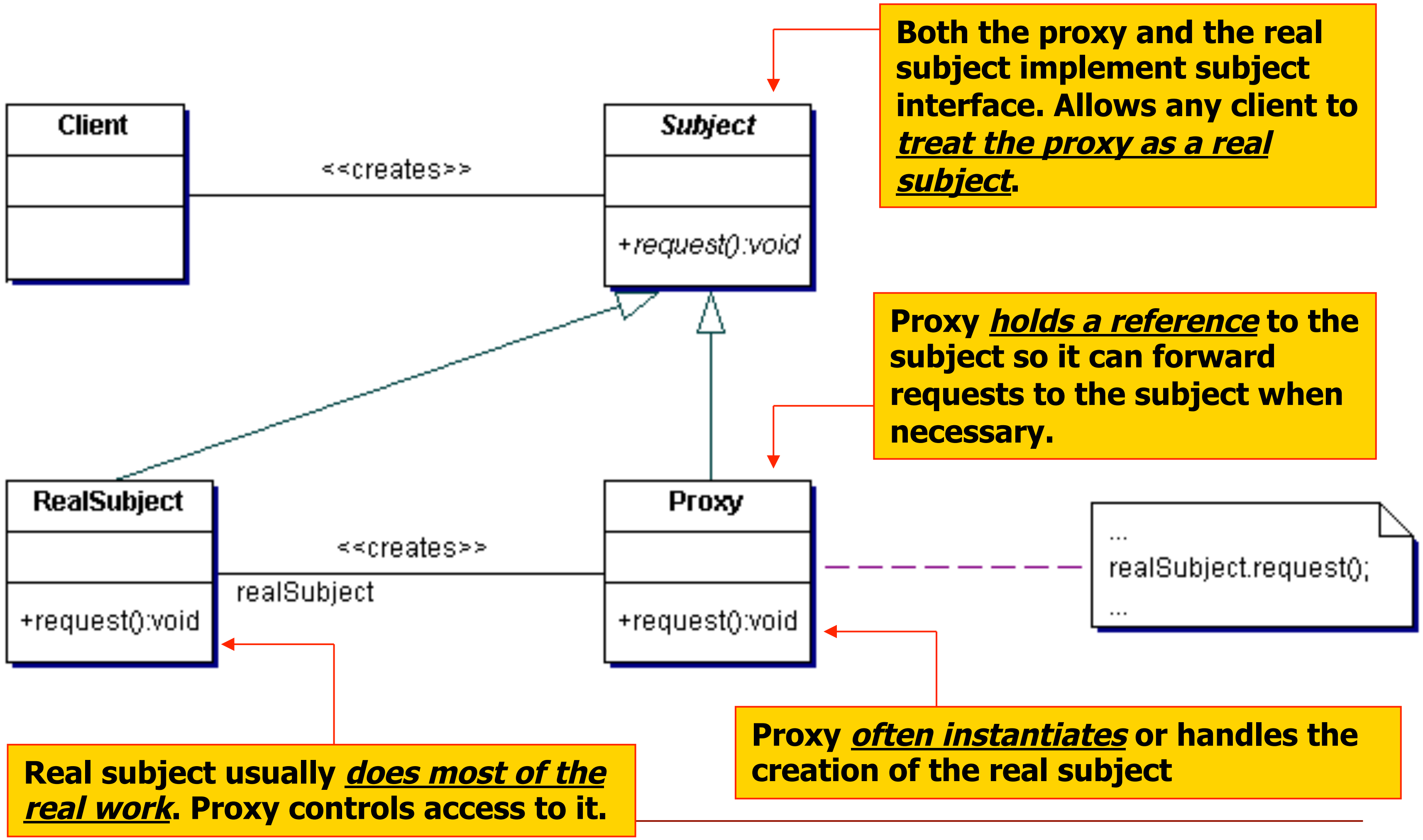
Decorator Pattern & java.io



Proxy Pattern

- *Provide a surrogate or placeholder for another object to control access to it.*
- A *representative object that controls* access to another object.
  - Real object may be remote, expensive to create or in need of securing.
- If creation of an object is expensive in time or resources, Proxy allows us to postpone creation until we actually need the object.
  - Usually has the ***same methods*** as the objects it represents.
  - One the object is loaded, the proxy ***passes on methods calls***.
  - *Can be used to distinguish between requesting an instance of an object and the actual need to access/create it.*
- Can also use proxies to ***keep copies of large objects*** that may or may not change.
  - If we create a second instance of an expensive object, proxy can decide that there is no reason to make a copy yet. *Uses original copy*.
  - If change made to new copy, proxy ***copies original object*** and makes change to the new instance (***Copy-on-write***).

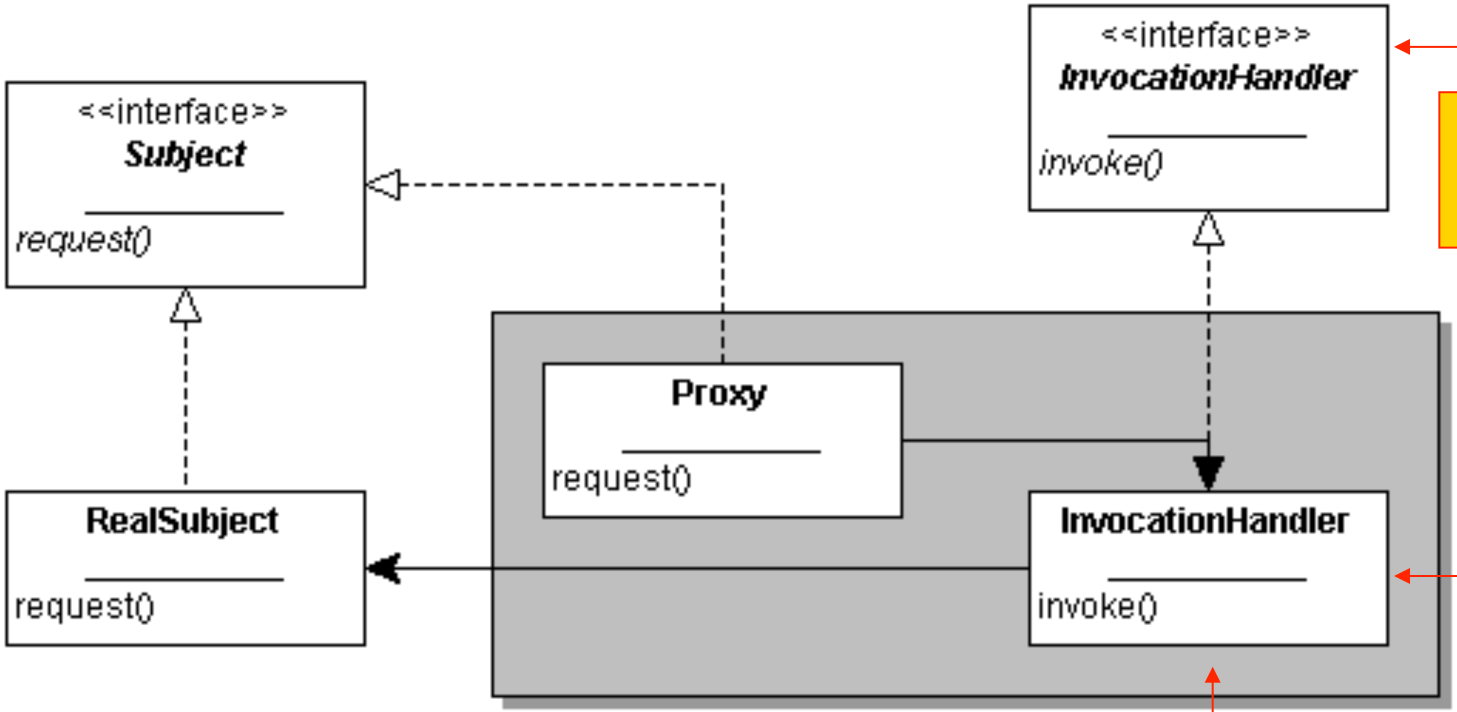
Proxy Pattern



Proxy Pattern

- Introduces a *level of indirection* when accessing an object.
  - ***Remote Proxy:*** acts as a local representative for an object that lives in a different address space.
  - ***Virtual Proxy:*** a representative for an object that may be expensive to create. Defers creation until object actually needed.
  - ***Caching Proxy:*** provides temporary storage for results of expensive operations. Allows multiple clients to share results.
  - ***Synchronisation Proxy:*** provides thread safe access to a subject.
  - ***Smart Reference Proxy:*** provides additional actions whenever a subject is referenced, e.g. reference counting.
  - ***Complexity Hiding Proxy:*** hides complexity and controls access to a complex set of classes. Aka façade proxy but not just a façade.
  - ***Copy-on-Write Proxy:*** Defers copying of an object until required by a client. A variant of virtual proxy.
  - ***Security/Protection Proxy:*** controls access to an object based on access rights.

Dynamic Proxy



Proxy now consists of two classes

Proxy generated by Java and implements the entire subject interface.

We supply the InvocationHandler, which gets passed all method calls that are invoked on the proxy. *__InvocationHandler controls access__* to the methods of the real subject.

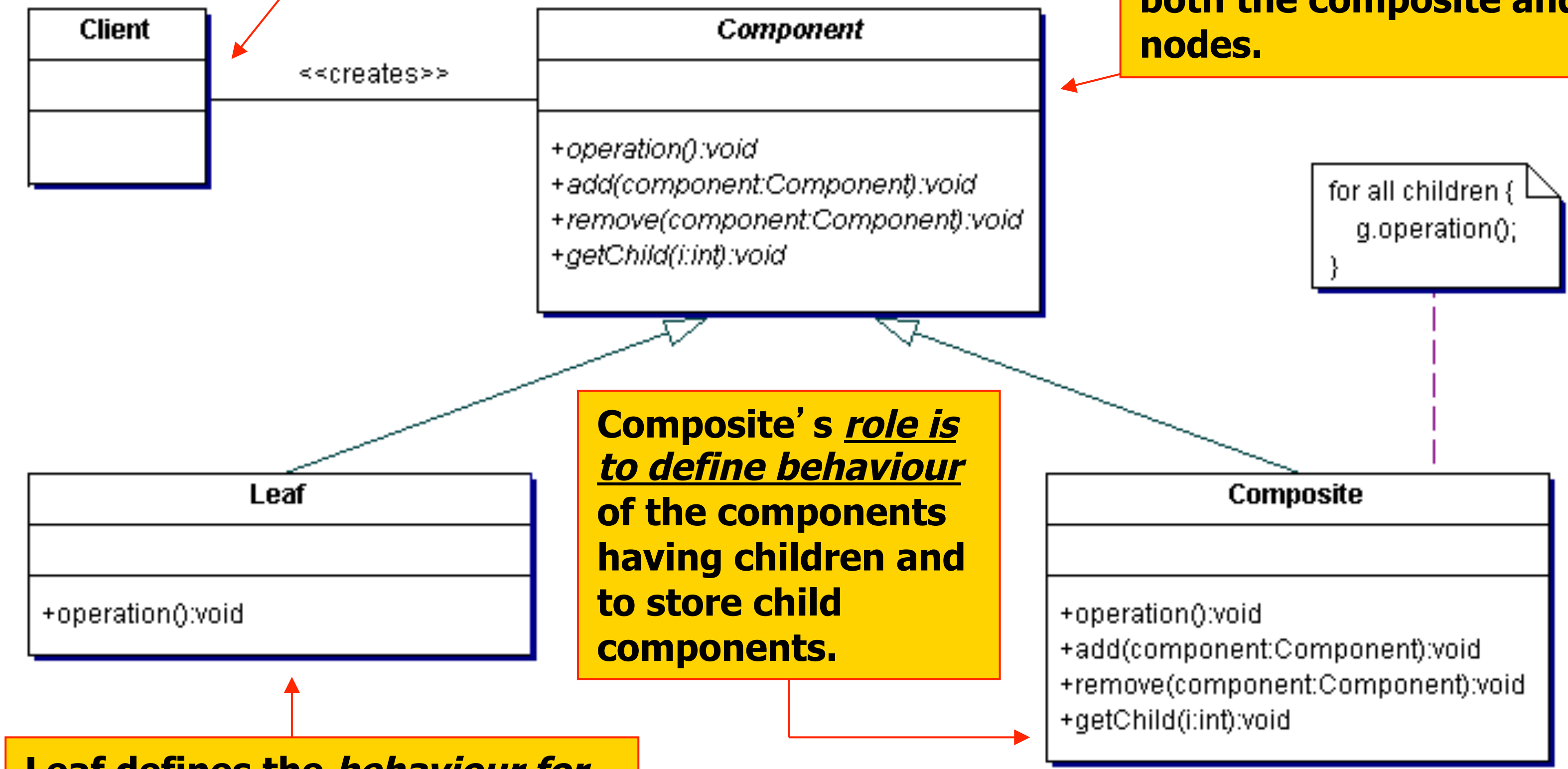
Composite Pattern

- *Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.*
  - Allows us to build structures of objects in the form of ***trees*** that contain both compositions of objects and individual objects as nodes.
  - Can apply the ***same operations over composites and individual objects***. Can ignore the differences between compositions and nodes.
- Use to create whole-part hierarchies or tree representations.
  - Each element (tree node or primitive) should have the ***same interface***.
    - How do we distinguish a leaf from a node with children?
    - Use an ***Enumerator/Iterator*** to accomplish this. ***hasMoreElement()/next()*** can tell us when we are dealing with a leaf.
- When an operation is carried out on a composite, the *composite responsible for propagating method call to constituents*.
  - Does this by iterating through hierarchy and invoking same method.

Composite Pattern

Client ***uses Component interface*** to manipulate objects in the composition.

Component defines an *interface for all objects* in the composition: both the composite and leaf nodes.



Composite's *role is to define behaviour* of the components having children and to store child components.

Leaf defines the *behaviour for elements in the composition*. Does this by implementing methods that the Composite supports.

Composite *also implements Leaf-related operations*.

Composite Pattern

- A composite holds a set of children. Those children *<u>may be other composites or leaf elements</u>*.
  - Organising data in this way ***<u>creates a tree</u>***, with a composite at the root and branches of composites growing up to tree nodes.
  - Usually use an iterator to transverse the tree.
- Clients can treat composite structures and individual objects uniformly. Easier to add new kinds of components.
  - Clients don't have to be changed for new Component classes.
- *<u>Can make design overly general.</u>* Harder to restrict the components of a composite.

Iterator Pattern

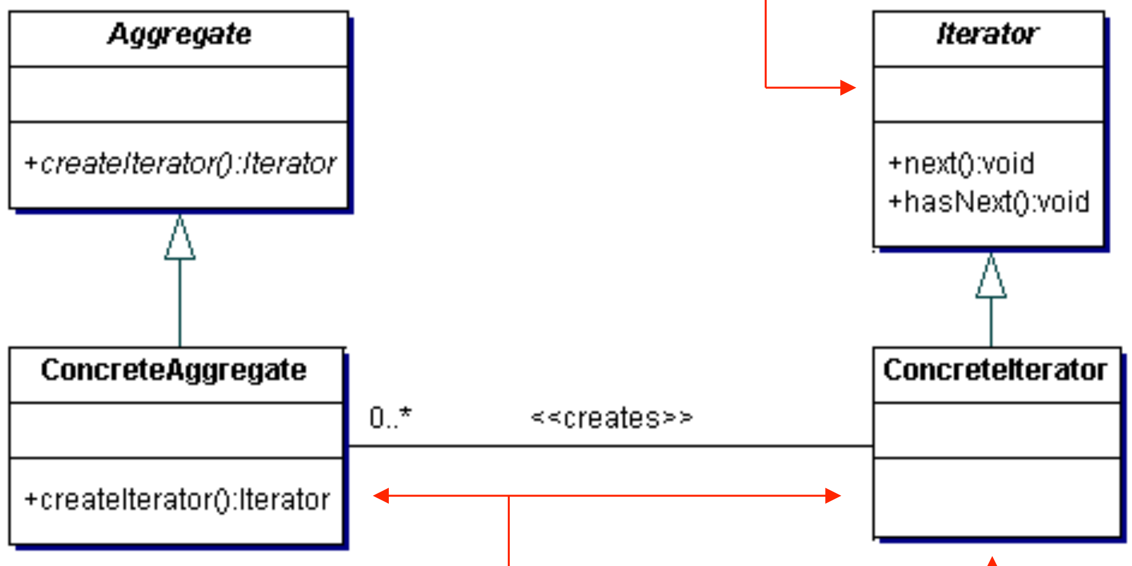
- *Provide a way to access the elements of an aggregate object sequentially without exposing its underlying implementation.*
  - Places the task of transversal on the iterator, not on the aggregate.
  - Simplifies the aggregate interface and places responsibility in the correct place (SRP).
- Can move through a collection of data using standard interface *without knowing details of internal representations* of that data.
  - Can also define special iterators that perform special processing (filtered iterators) and return only specified elements of the data.
  - Supported through ***java.util.Enumeration*** and ***java.util.Iterator/ListIterator***.
  - Can also use for/in loops in Java 5.

    for (Object o: collection) {...}
- ***Data modification*** can cause problems - need to synchronise thread access to loop.
  - ***Privileged access***. Inner class implementations manipluate containing class.

Iterator Pattern

Iterator interface defines methods for transversing over elements of a collection.

Having a common interface for aggregates decouples a client from the implementation of a collection of objects.



ConcreteAggregate has a collection of objects and implements the method that returns an iterator for its collection

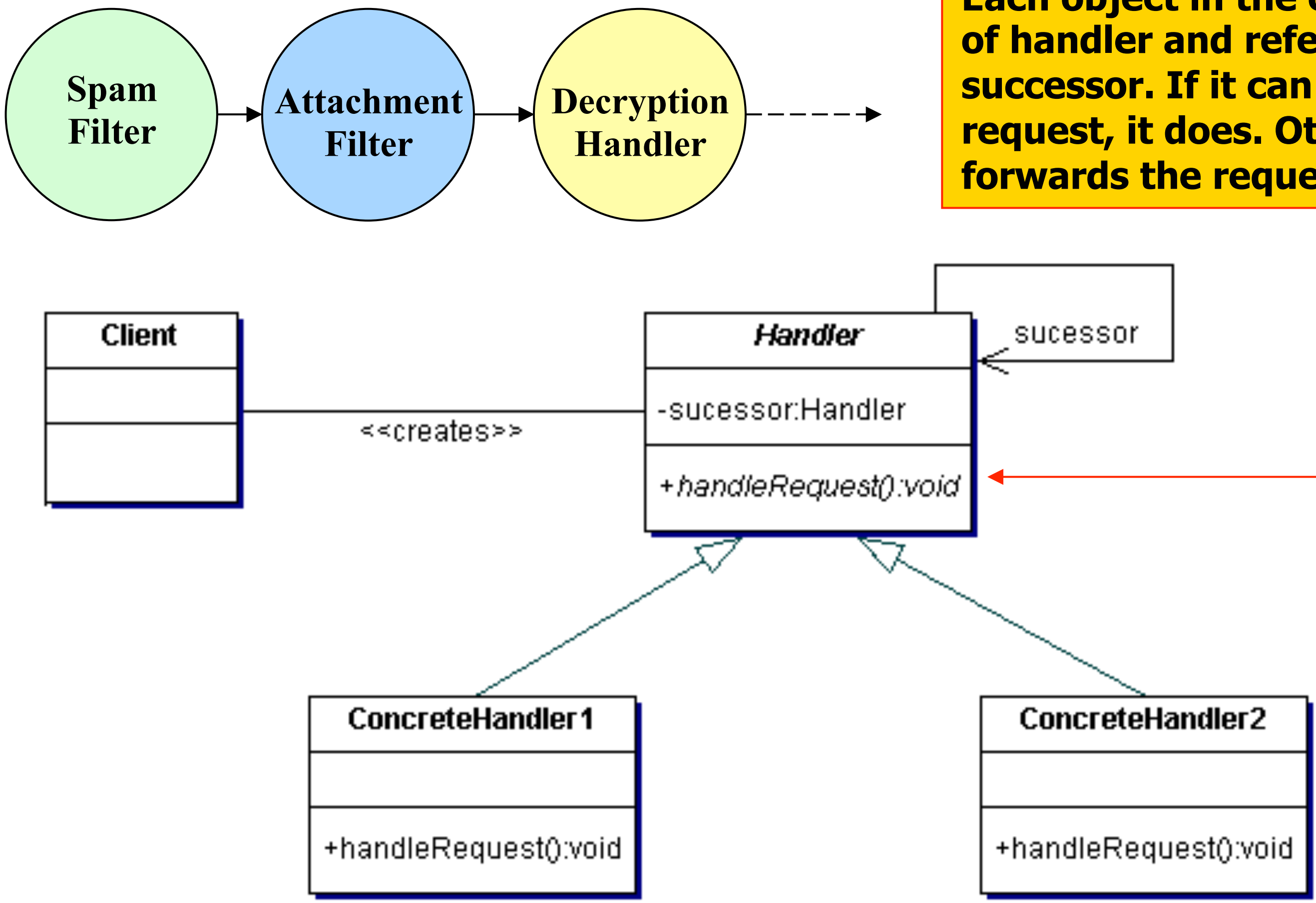
Each ConcreteAggregate is responsible for instantiating a ConcreteIterator that can iterate over its collection of objects.

ConcreteIterator responsible for managing current position of the iteration.

Chain of Responsibility

- *Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass request along the chain until an object handles it.*
- Allows a number of classes to attempt to handle a request, *without any of them knowing the capabilities* of the other classes.
  - ***Loose coupling*** between classes. Only common link is the request that is passed to them. Request passed along until a class can handle it.
  - Each object in the chain acts as a handler and references a successor object.
- Used when *more than one handler available* to handle a request and no way to know which handler to use.
  - Also may want to issue a request to one of several objects ***without explicitly specifying which one to use***.
  - May also want to ***modify the set of objects dynamically*** that can handle requests.
- In Java, basic chain class ***must be an interface***, to allow individual objects inherit from another useful hierarchy.

Chain of Responsibility



Each object in the chain is a type of handler and references a successor. If it can handle the request, it does. Otherwise it forwards the request.

Chain of Responsibility

- CoR *distributes responsibilities* between objects.
  - Any object can satisfy some or all requests. Can change the chain and responsibilities at runtime. Can discard request if no object to handle it.
- CoR *decouples sender of request and its receivers*.
  - ***Simplifies an object*** because it does not need to know the chain's structure and keep direct references to its members.
  - Can ***add/remove responsibilities dynamically*** by changing members or the order of the chain.
- ***Note:*** execution of request **NOT** guaranteed!
  - Request may ***fall off the end of the chain*** if no object handles it. This may be an advantage or a disadvantage.
  - Can be hard to observe runtime characteristics and debug.
- Inheritance is basically a chain of responsibility.
  - Method invocations on a deeply derived class cause invocations up the inheritance chain until first parent class containing method is found.

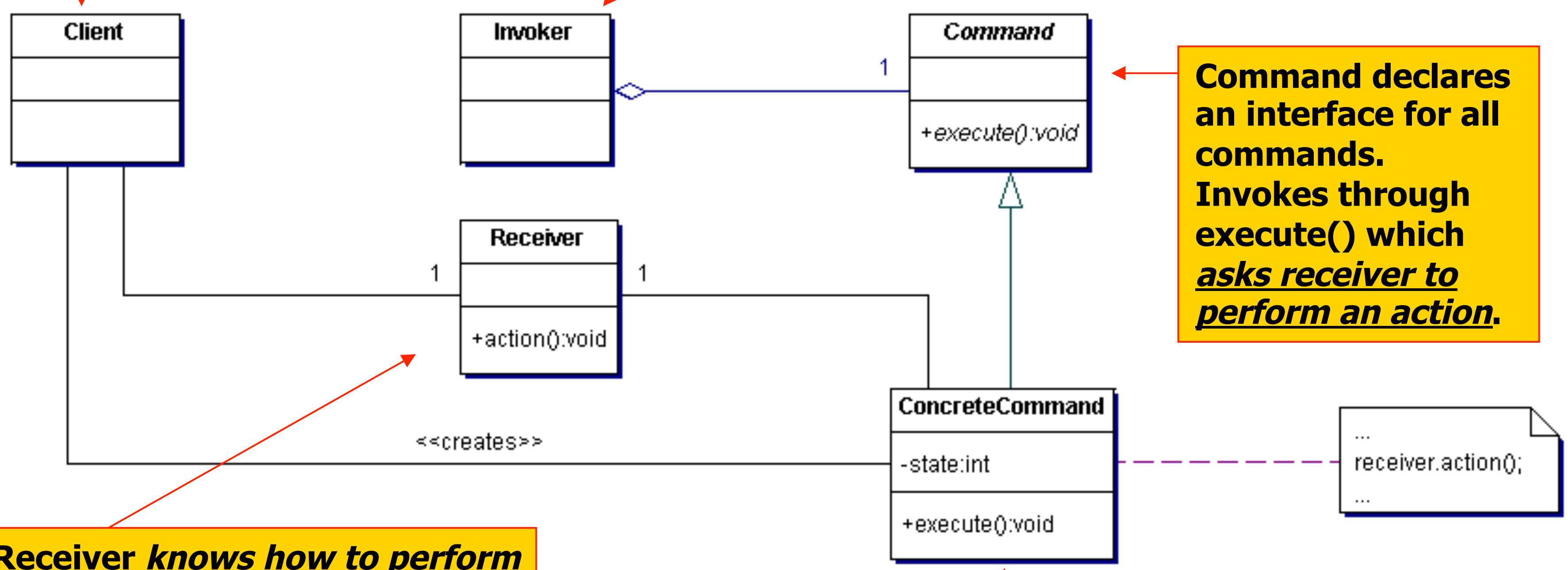
Command Pattern

- *Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undo operations.*
- Command pattern *decouples an object making a request* from an object that knows how to perform it.
  - A ***command object*** is at the centre of this decoupling, by ***encapsulating a receiver with an action or a set of actions***.
  - Gives a ***client*** the ability to make request ***without knowing the actual action*** that will be performed.
  - Because the ***invocation is encapsulated***, we can change the action without affecting the client in any way.
- An *invoker makes the request* of a command object by calling an execute method. This method *invokes the action of the receiver*.
  - Invokers can be parameterised with commands, even dynamically.
  - ***Undo operations*** supported by command object maintaining the state of the previous command operation.

Command Pattern

Client responsible for creating a ConcreteCommand and *setting its receiver*.

Invoke *holds a command* and at some point *asks the command to execute* a request

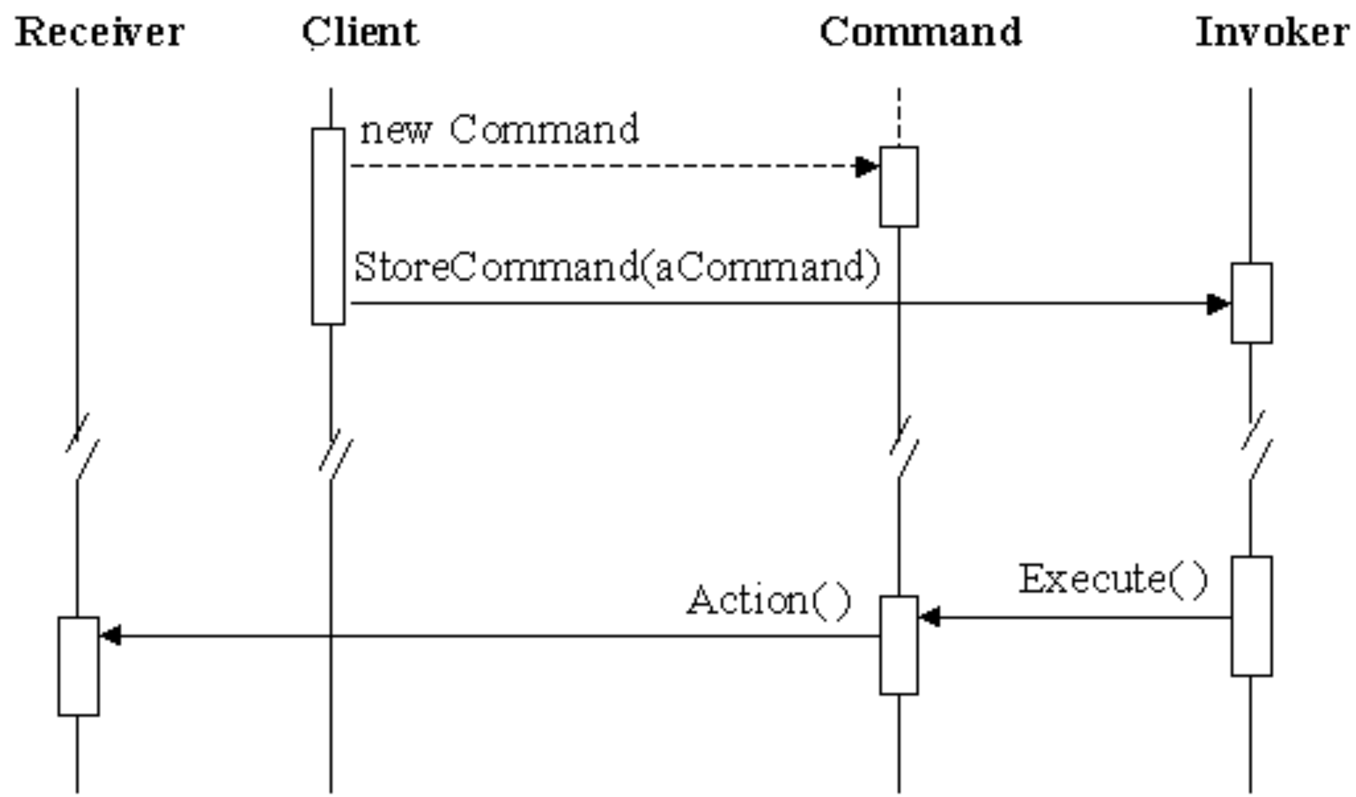


Command declares an interface for all commands. Invokes through execute() which *asks receiver to perform an action*.

Receiver *knows how to perform work needed* to carry out a request. Any class can act as a receiver.

ConcreteCommand defines a *binding between an action and a receiver*. Invoker makes a request by calling execute(). ConcreteCommand carries out action by calling one or more methods on receiver.

Command Pattern



- Command pattern *widely used in client/server implementations*.
  - Decouples client from server. Provides flexibility to change server-side classes. Reduces number of network calls (EJBCommand).

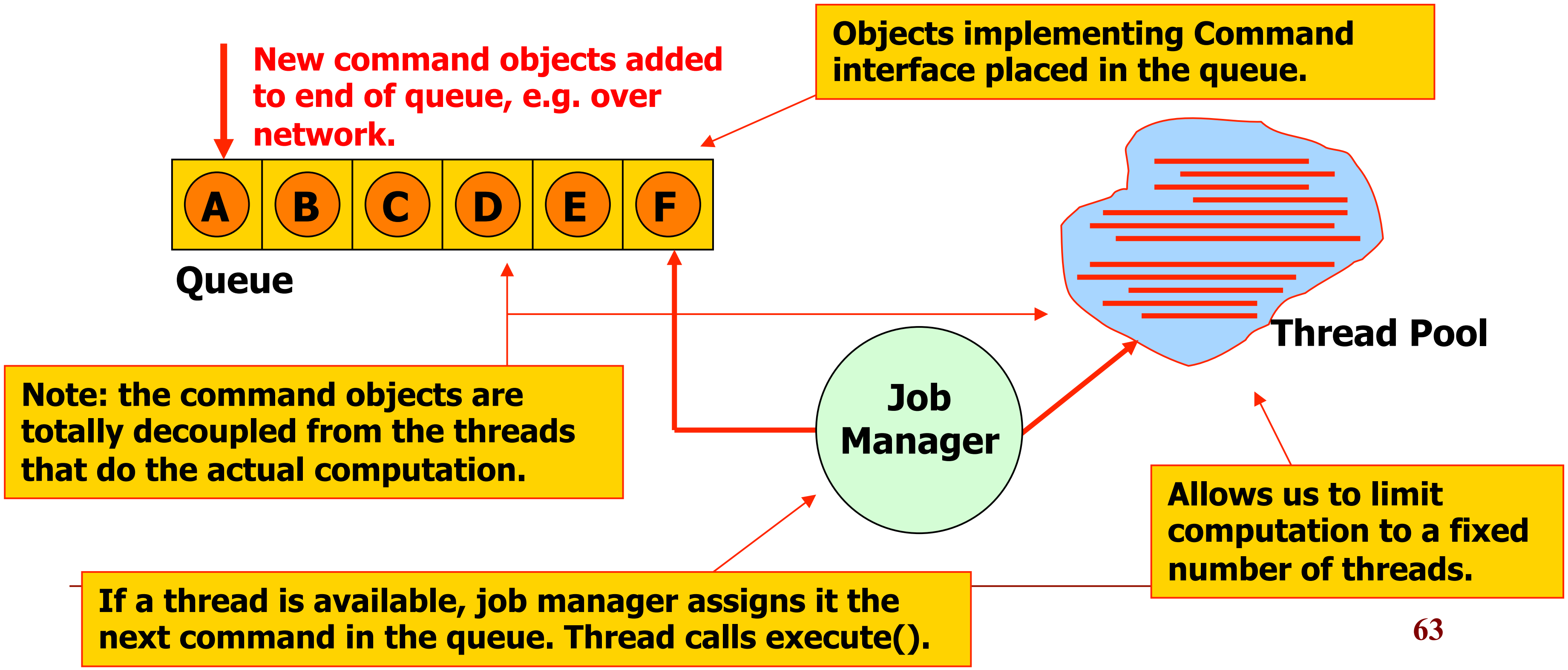
Command Pattern

- If overused, command can cause *proliferation of little classes*.
  - Clutters up program namespace. If inner-classes used, containing class can also get cluttered.
- ***Meta Command:*** create macros of commands that enable us to execute multiple commands at once:

```
class MacroCommand implements Command{
  Command[] commands;
  public MacroCommand(Command[] commands){
          this.commands = commands;
  }
  public void execute(){
    for (int i = 0; i < commands.length; i+){
      commands[i].execute();
    }
  }
}
```

Command Pattern

- Command provides us with means to package a piece of computation (receiver and set of actions) and pass it around as a first class object (used without restrictions).
  - Can be invoked at a later time or by a different thread.
  - Used to implement thread/connection pools, job queues etc.



Interpreter Pattern

- *Given a language, define a representation for its grammar, along with an interpreter that uses the representation to interpret sentences in the language.*
  - An Interpreter can be used to evaluate the results of an expression given as a sequence of operations, operators, and operands.
- Each ***grammar rule*** is represented by a class.
  - Allows us to easily change or extend a grammar (BNF).

A programme is an expression consisting of commands and repetitions.

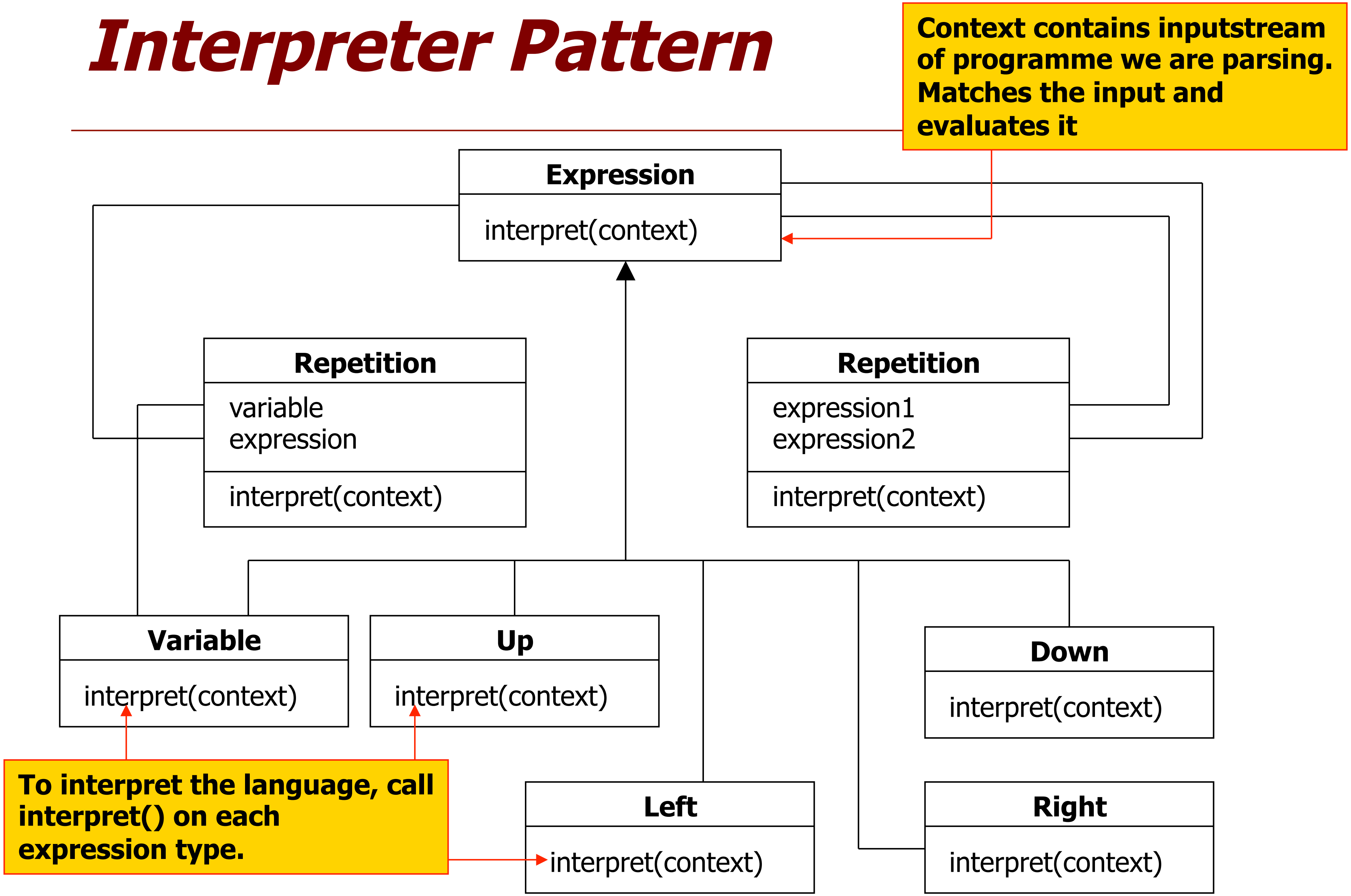
```
expression ::= <command> | <sequence> | <repetition>
sequence ::= <expression> ';' <expression>
command ::= up | down | left | right
repetition ::= while '(' <variable> ')' <expression>
variable ::= [A-Z, a-z]+
```

A sequence is a set of expressions separated by semi-colons.

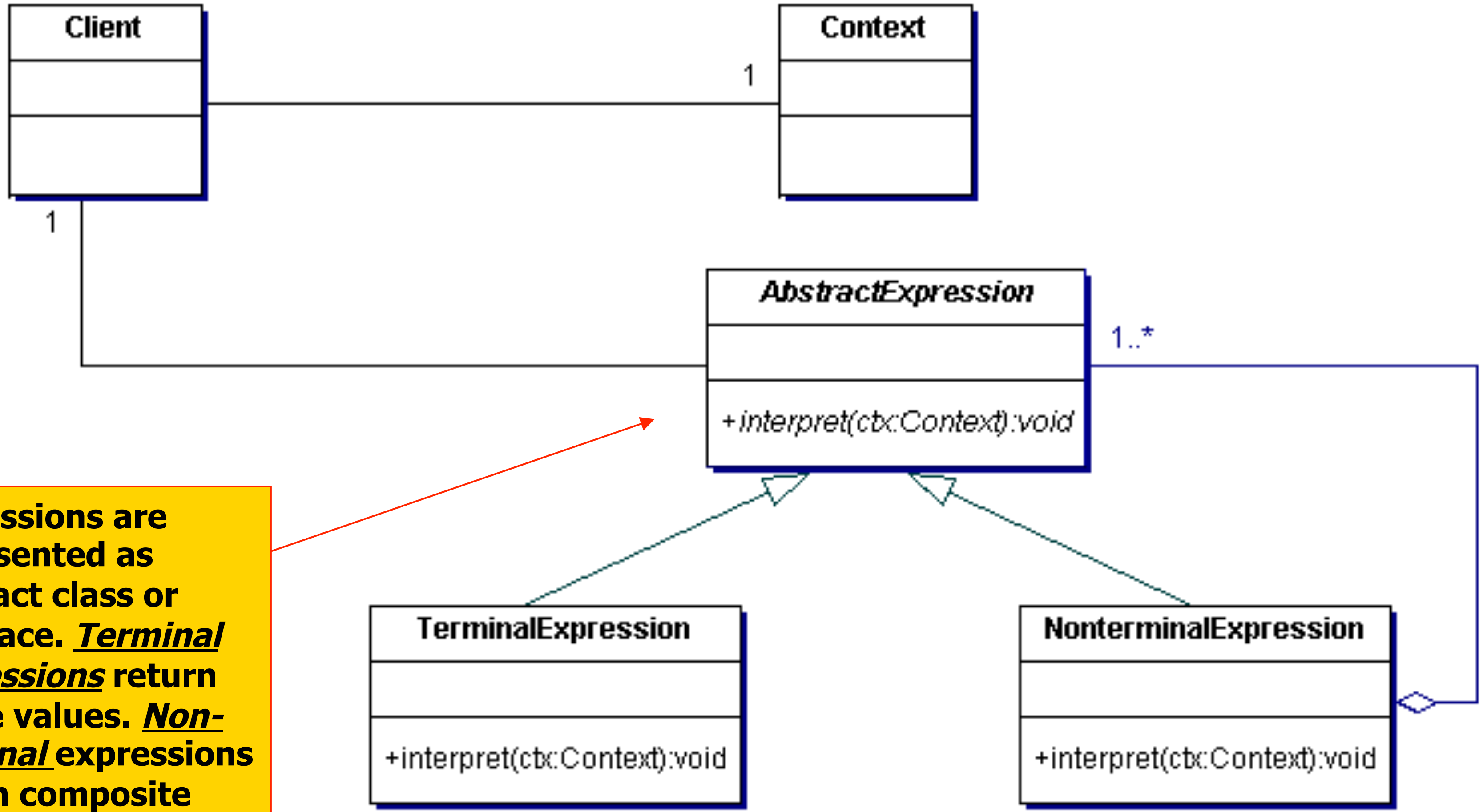
A while statement is just a conditional variable and an expression

We have four commands.

Interpreter Pattern



Interpreter Pattern



Expressions are represented as abstract class or interface. *Terminal expressions* return single values. *Non-terminal* expressions return composite objects.

Interpreter Pattern

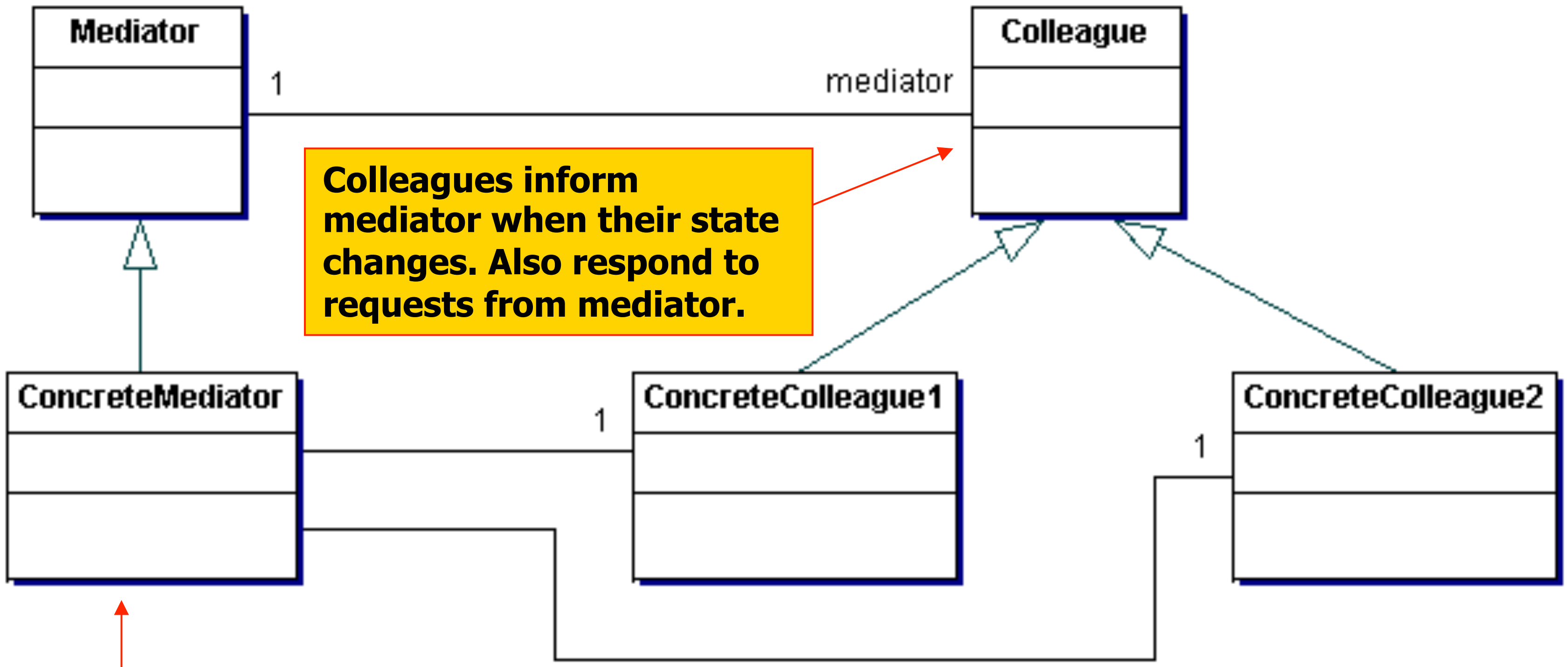
- Expression consists of one or more ***terms***.
  - Term can be a ***value***, or it might be an ***operation*** which requires a single operator and one or more ***operands***.
    - Each operand ***may be another term***: either a single value or yet another operation.
  - Can represent each operation and operand as an object with ***evaluate()*** or ***interpret()*** that returns operation result an operand value.
- A *Context* object that stores information about the circumstances in which the expression is being evaluated.
  - Operator precedence, global information about variables/symbols.
- Enables us to build up an expression as a tree of *Expressions*.
  - Result interpreted by invoking ***interpret()*** of root of tree.
  - Root responsible for recursive invocation of *interpret()*.
- Fine for simple grammars. More complex grammar rules require a parser/compiler generator.

Mediator Pattern

- *Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.*
- Mediator class is the *only class with a detailed knowledge* of methods of other classes. ***Other classes to no know about each other.***
  - Classes inform mediator when changes occur. Mediator passes messages on to any other classes that need to be informed.
- Mediator ***localises behaviour*** that otherwise would have to be distributed among several objects.
  - Can change behaviour of the program by changing or subclassing the mediator.
- Makes it possible to add new*Colleagues* to a system without having to change any other part of the program.
  - Solves problem of a *Command* object having to know too much about the objects and methods in the rest of a UI.

Mediator Pattern

- Mediator ***can become monolithic in complexity***, making it difficult to change and maintain.
- Mediator pattern normally used in visual apps – Swing.



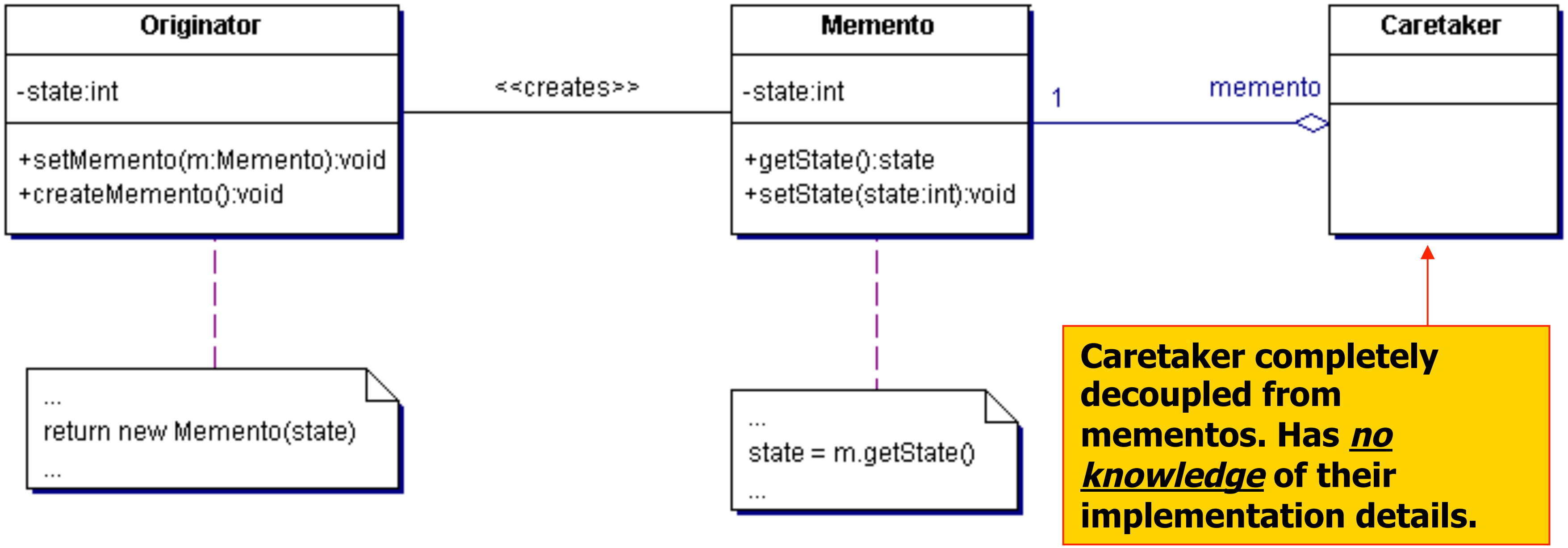
Instead of tightly coupled objects (PLK↓), mediator *contains all control logic* for object interactions.

Memento Pattern

- *Without violating encapsulation, capture and internalize an object's internal state so that objects can be restored to this state later.*
  - Create a snapshot of an object so that object can later be restored from snapshot – Undo operation.
- An *originating object creates a memento object of its state*.
  - Keeping the saved state external from the originator object helps maintain cohesion. Would breach SRP if originator stored previous state.
  - Keeps the originator’s data encapsulated.
- State ***stored in a caretaker*** for such mementos.
  - If originator needs to recall or reset previous state, it asks the caretaker to return the previously stored memento.
- *Caretaker object requires no knowledge of any of the implementation details about the mementos it holds.*
  - Can use *Serializable* and *Externalizable* APIs to achieve this.

Memento Pattern

- Memento *<u>need not always capture the entire object</u>*.
  - ***Preserves encapsulation*** boundaries and ***simplifies*** originator.
  - Pattern appropriate when ***<u>memento size not large</u>***.



Observer Pattern

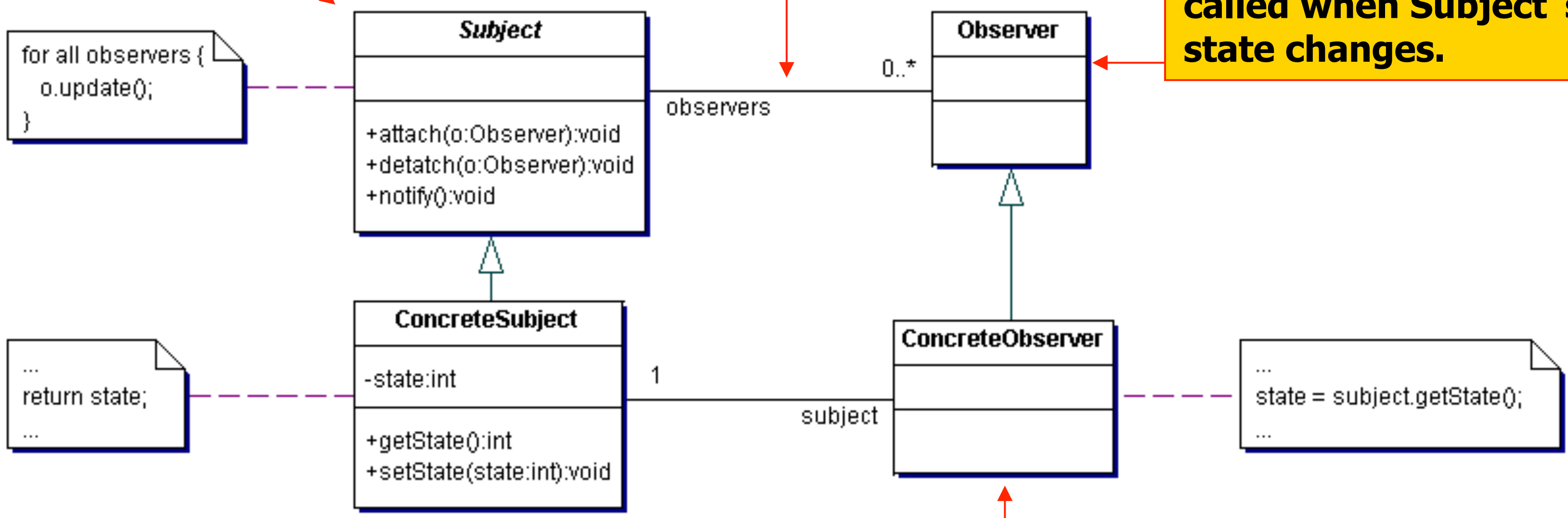
- *Define a one-to-many dependency so that when one object changes state, all its dependents are notified and updated automatically.*
- *Publishers + Subscribers = Observer Pattern*
  - 1:M relationship w.r.t. publisher (Subject) and subscriber (Observer).
- *Observer pattern assumes that object containing the data (subject) is separate from objects that use the data (observers).*
  - ***Two basic styles of notification:*** *PushModel* and *PullModel*. In former the subject notifies observers. Can cause problems with cascading updates. Latter method involves clients polling subject for updates.
  - Subject ***does not know anything about observers (loosely coupled)***. It publishes a change and observers get notified of the change.
- ***Note:*** *should not depend on a specific order of notification.*

Observer Pattern

Objects use subject interface to register or remove themselves as observers

Each Subject can have many observers.

All potential observers implement Observer. Its update method called when Subject’s state changes.



Notify() used to update all current observers when the state of ConcreteSubject changes.

A ConcreteObserver is any class that implements Observer. *<u>Each observer registers with a ConcreteSubject</u>* to receive updates.

Observer Pattern

- Observers ***promote abstract coupling to subjects***. A subject does not know the details of its observers.
  - Can ***reuse*** subject and observers independently of each other. Changes to either subject or observer will ***not affect each other.***
  - ***Any object*** that implements Observer interface can register to be notified by Subject.
- Implemented using an *abstraction* for a subject.
  - Implementators responsible for registering and notifying observers.
  - Can ***dynamically establish relationships between objects*** – can connect an observer to an observable object when program is running.
- Used in MVC. Basis for *publish-subscribe* messaging architectures.
- Used in Java event model and by *java.util.Observable/Observer*.
  - Some problems with ***java.util*** version. Better to write your own!

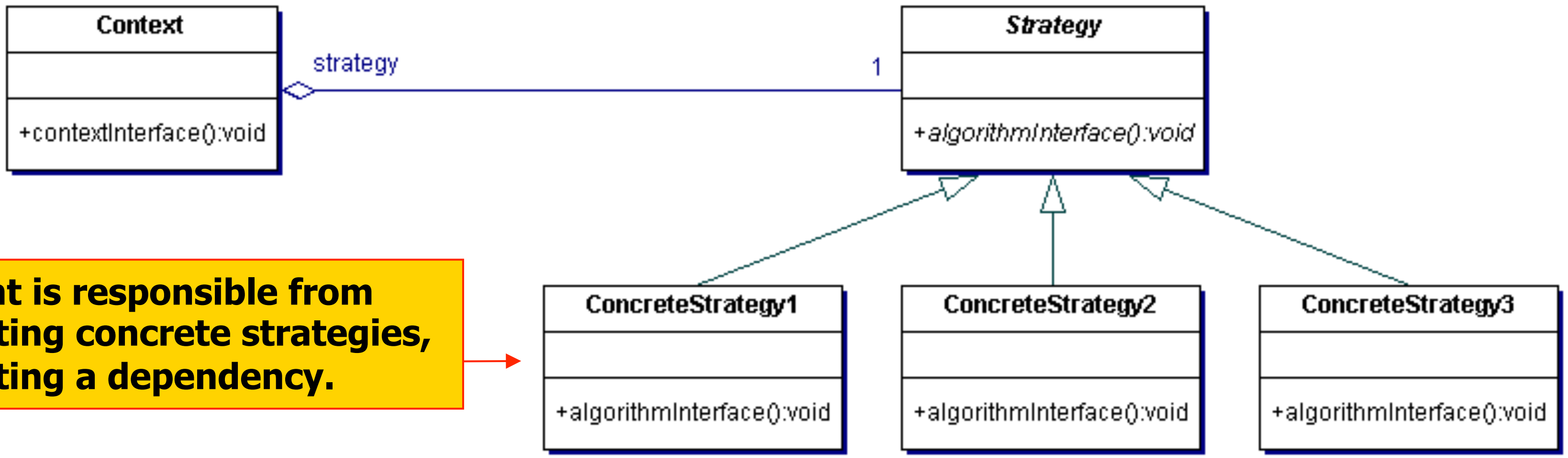
Strategy Pattern

- *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.*
- Defines an *abstract policy* *for performing an algorithm*.
  - Provides multiple mechanisms that can comply with that policy.
  - ***Allows mechanism to vary independently*** from the policy required.
- Consists of a number of related algorithms encapsulated in a driver class called a *Context*. *Context decides the strategy to use*.
  - **Intent is similar to state pattern:** switch easily between algorithms without any monolithic conditional statements. ***Don't switch often.***
- Strategy encapsulates several algorithms that do, more or less, the same thing.
  - E.g. a *ListModifierStrategy* policy may have FIFO/LIFO implementations of *addElement(), removeElement(), nextElement().*
- Java *LayoutManager* specifies a layout policy for GUIs.
  - Concrete layout manager classes for *GridLayout*, *FlowLayout* etc..

Strategy Pattern

- Allows us to dynamically select one of several algorithms.
  - Algorithms ***can be related*** (through inheritance) ***or unrelated*** (if they implement the same interface). Context interface and strategy methods must be broad to accommodate disparity of implementations.
- Using a context to switch between requests provides flexibility.
  - Avoids use of conditional statements (difficult to maintain).
- *Decision on algorithm to use rests with client.*
  - ***Client must know*** that there are different strategies available.

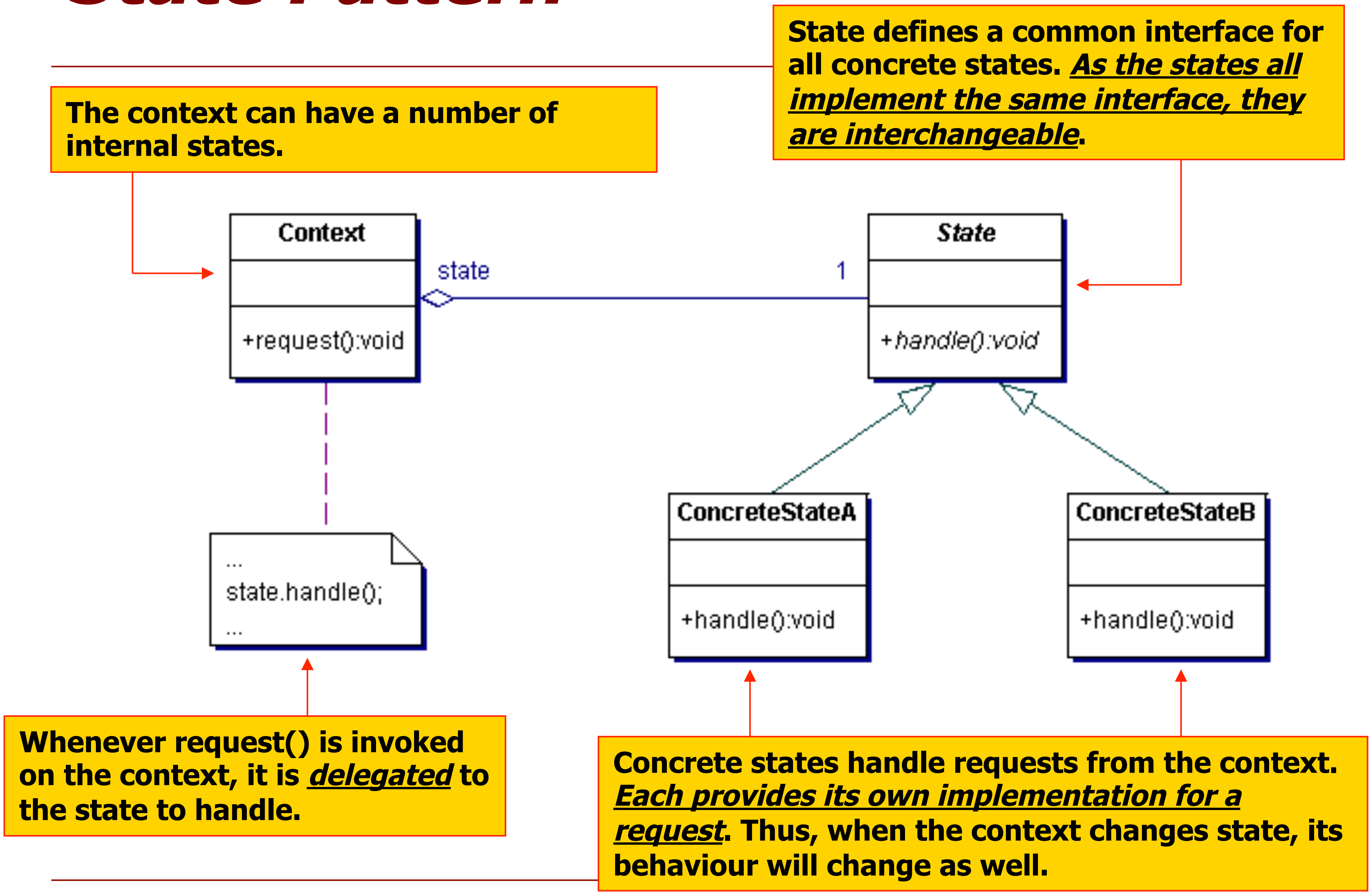
Client is responsible from creating concrete strategies, creating a dependency.



State Pattern

- *Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.*
- State of an object refers to the set of *values of all its member variables*.
- Permits object to change behavior dynamically by changing internal state.
  - Creates the illusion that the object has changed.
  - ***State pattern uses delegation and polymorphism to define different behaviours for different states of an object***.
- *Encapsulates state into separate classes* and ***delegates*** to the object representing the current state.
  - Uses composition to give appearance of a class change by referencing different state objects.
  - A context has a ***composition relationship*** with a set of objects defined by an abstraction.
  - Concrete implementations of abstraction define different behaviours.

State Pattern



State Pattern

- ***Note:*** *state and strategy patterns may look the same (UML), but they differ in intent.*
  - Strategy provides a ***flexible alternative to subclassing*** by configuring context classes with a behaviour or algorithm.
  - State is an alternative to inserting lots of conditionals in a context.
- By ***encapsulating each state into a class***, we localise any changes that will need to be made.
  - All behaviour associated with a particular state placed in one object.
  - Allows new states and transitions to be easily added.
- *Makes state transitions explicit* - no requirement for client to track state.
  - Transitions can be controlled by the state classes or context classes.
- If state objects have no instance variables, they ***can be shared*** among context instances.
  - Application of State pattern typically results in a ***greater number of classes*** in a design.

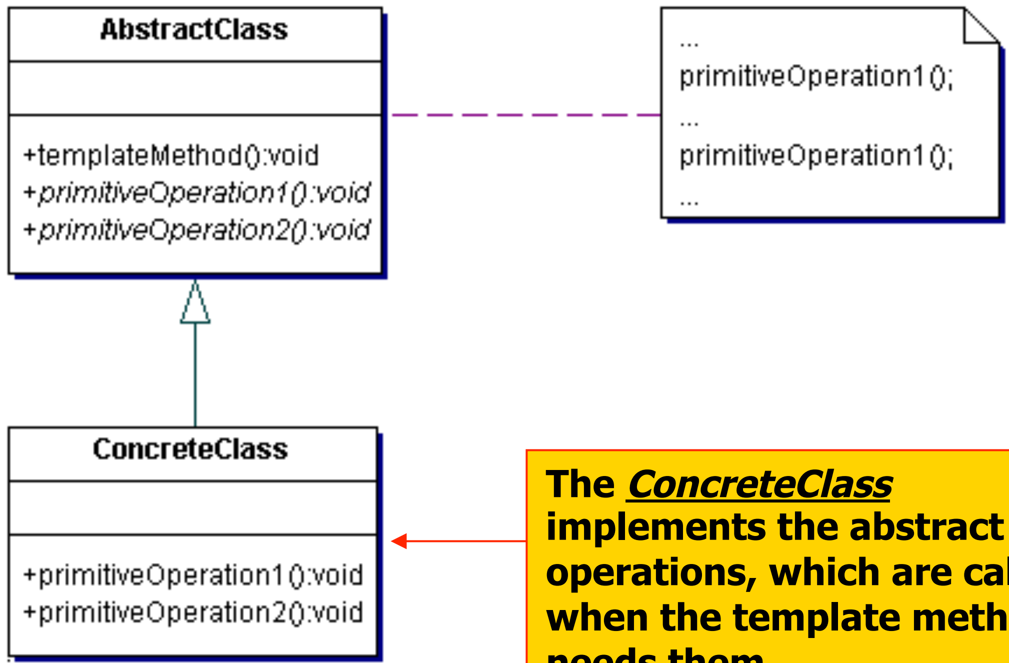
Template Pattern

- *Define the skeleton of an algorithm in an operation, deferring some steps to its subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.*
- Creates a *template for an algorithm*. A method that defines an algorithm as a series of steps.
  - One or more of the steps is abstract and implemented by subclasses.
  - Ensures that the ***algorithm structure stays unchanged***, while subclasses can provide some part of the implementation.
- Base class defines only some of the methods it will be using.
  - Other methods are implemented in derived classes. Methods in base class may invoke methods that are not yet implemented.
  - The ***details of algorithm not completely determined*** in base class.
- Used in Swing - *paint() and in* Applets *init(), paint(), start() and stop()* . *Deferred for us to implement.*
  - ***But we cannot control when they are called.***

Template Pattern

The abstract class contains the template method and abstract versions of the operations used in the template method.

Template method makes use of the primitiveOperations to implement an algorithm. *<u>It is decoupled from the actual implementation of these operations</u>*.



Can be many ConcreteClasses, each implementing the full set of operations required by the template method.

The *<u>ConcreteClass</u>* implements the abstract operations, which are called when the template methods needs them.

Template Pattern

- *Template* class has *four types of methods* derived classes can use:
  - ***Concrete Methods:*** complete methods that carry out functions that all subclasses will want to use.
  - ***Abstract Methods:*** deferred to derived classes. This approach is used when a subclass must provide an implementation of the method or step in the algorithm.
  - ***Hook Methods:*** declared in the abstract class, but only given an empty or default implementation. Gives subclasses the ability to "hook into" algorithm at various points. Hook methods, unlike abstract methods, ***are optional*** for subclasses. Hooks also give subclass the ***chance to react to a step in the template method*** that just happened or is about to happen.
  - ***Template Methods:*** calls any combination of concrete, abstract or hook methods. Not intended to be over-ridden. Describes an algorithm *without actually implementing all of its details*.

Template Pattern

- Templates exemplify the *Hollywood Principle:*

Don't call us, we'll call you!

- Hollywood Principle provides us with a way of preventing ***dependency rot***.
  - High and low-level components ***interdependent on each other***.
- Principle allows low-level components to hook themselves into a system, but high-level components determine when they are needed and how.
  - Same goal as DIP (decoupling) but DIP makes a stronger statement about how to avoid dependencies in design (by abstraction).

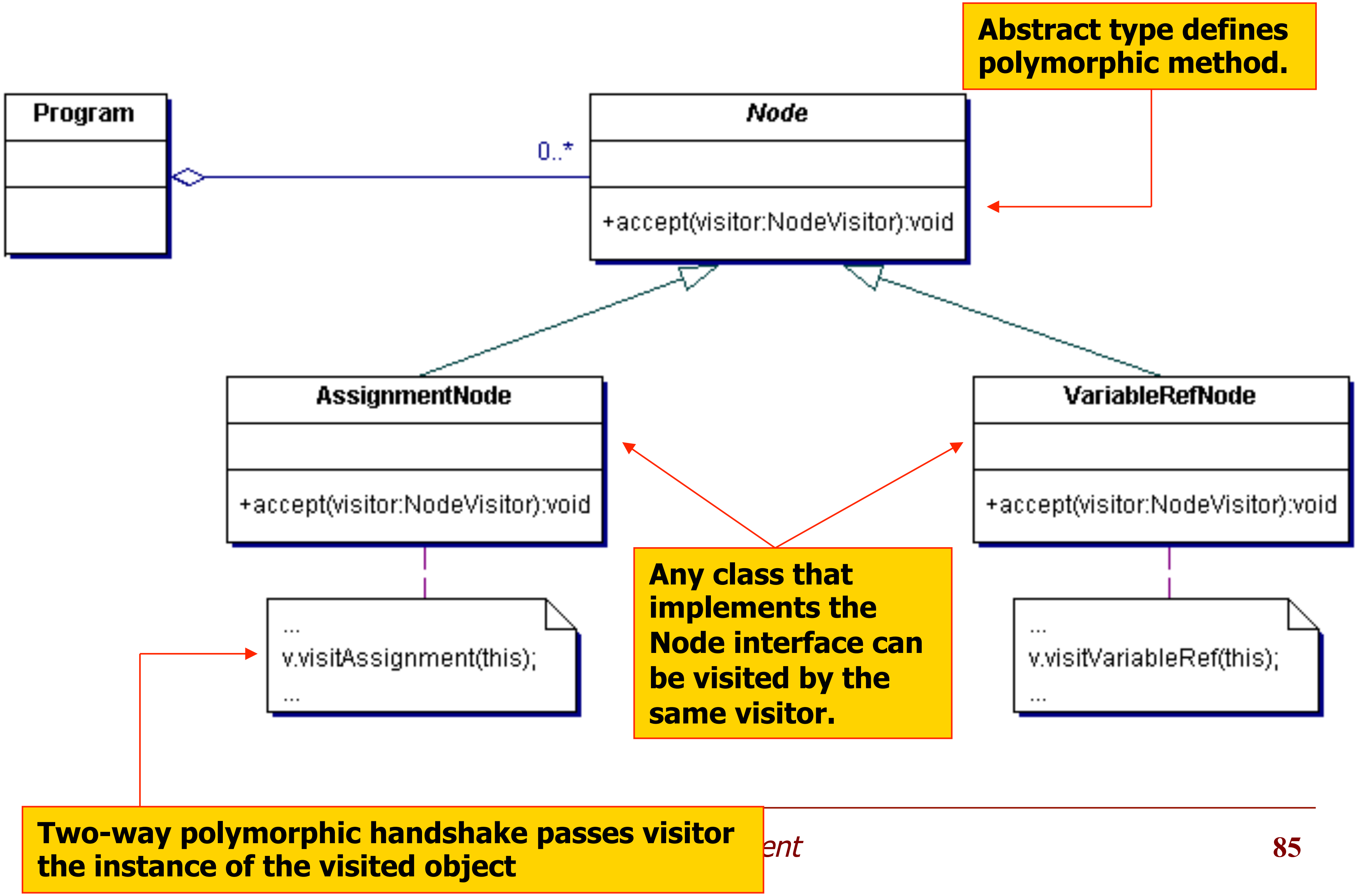
Visitor Pattern

- *Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.*
- *Creates an external class to act on data in other classes.*
  - Useful if there is a large instance of a small number of objects and we want to perform an operation that involves all or most of them.
- Goes against the spirit of the principles already discussed!
  - A set of closely related functions may be scattered throughout a number of different classes. *draw()* or circle, rectangle, triangle.
- Visitor class contains all related methods.
  - Visits each object in succession. A visit is implemented by invoking a ***public accept(Visitor v)*** method on target class. Target class then invokes ***visit()*** method of visitor passing ***this*** as an argument, eg:

```
public void accept(Visitor v){
    v.visit(this);
}
```

Visitor object receives a reference to each of the instances. Can now call their method and get state.

Visitor Pattern



Visitor Pattern

- The *accept(Visitor v)* signature is an example of ***dynamic-dispatch*** (also called single-dispatch).
  - ***Invocation of visit() is polymorphic***. Method implementation to dispatch is determined at run-time.
- *Visitor pattern solves problem of double-dispatch*.
  - Implementation depends on the class of ***two arguments***, not just one (*Visitor* and ***this***).
  - Methods dispatched based on multiple arguments are called ***multi-methods*** and require ***multiple-dispatch***.
- *Visitor simulates a double dispatch by using a **two-way polymorphic handshake**.*
- Visitor *visits each object in a structure* and asks it to accept this visitor by invoking the object's *accept/acceptVisitor()* method.
  - Object's *accept/acceptVisitor()* invokes *visit()* of visitor that is designed for operating on that type of object.

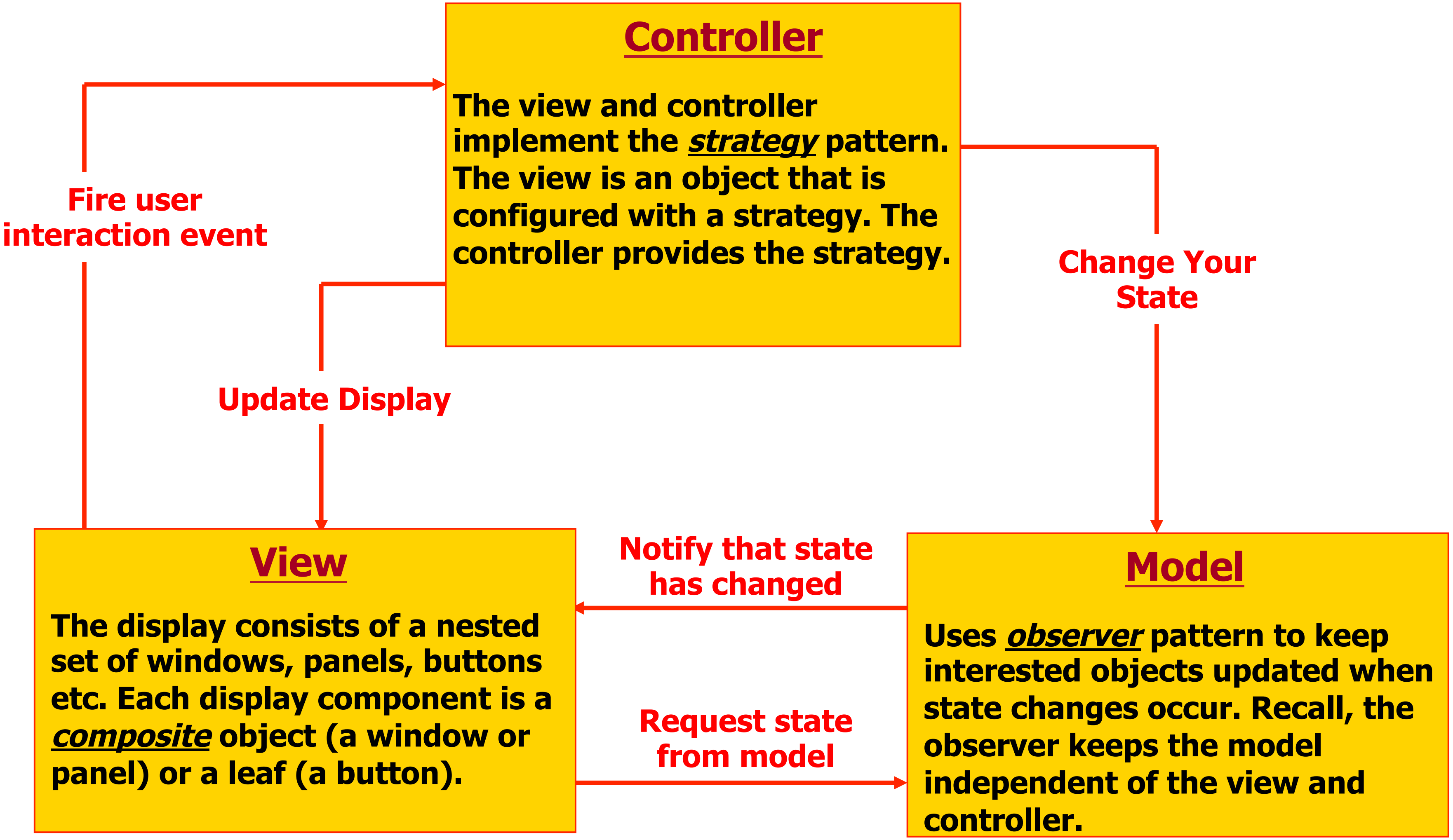
Visitor Pattern

- Can use visitor to perform an operation on the data contained by a number of objects with different interfaces.
  - Can also be used to perform a number of unrelated operations on classes.
- *Only works when we do not expect new classes to be added*.
  - Otherwise ***new abstract methods need to be added*** to visitor base class.
- ***Java reflection offers a way around this restriction:***

```
public interface ReflectiveVisitor {
        public void visit(Object o);
}
```

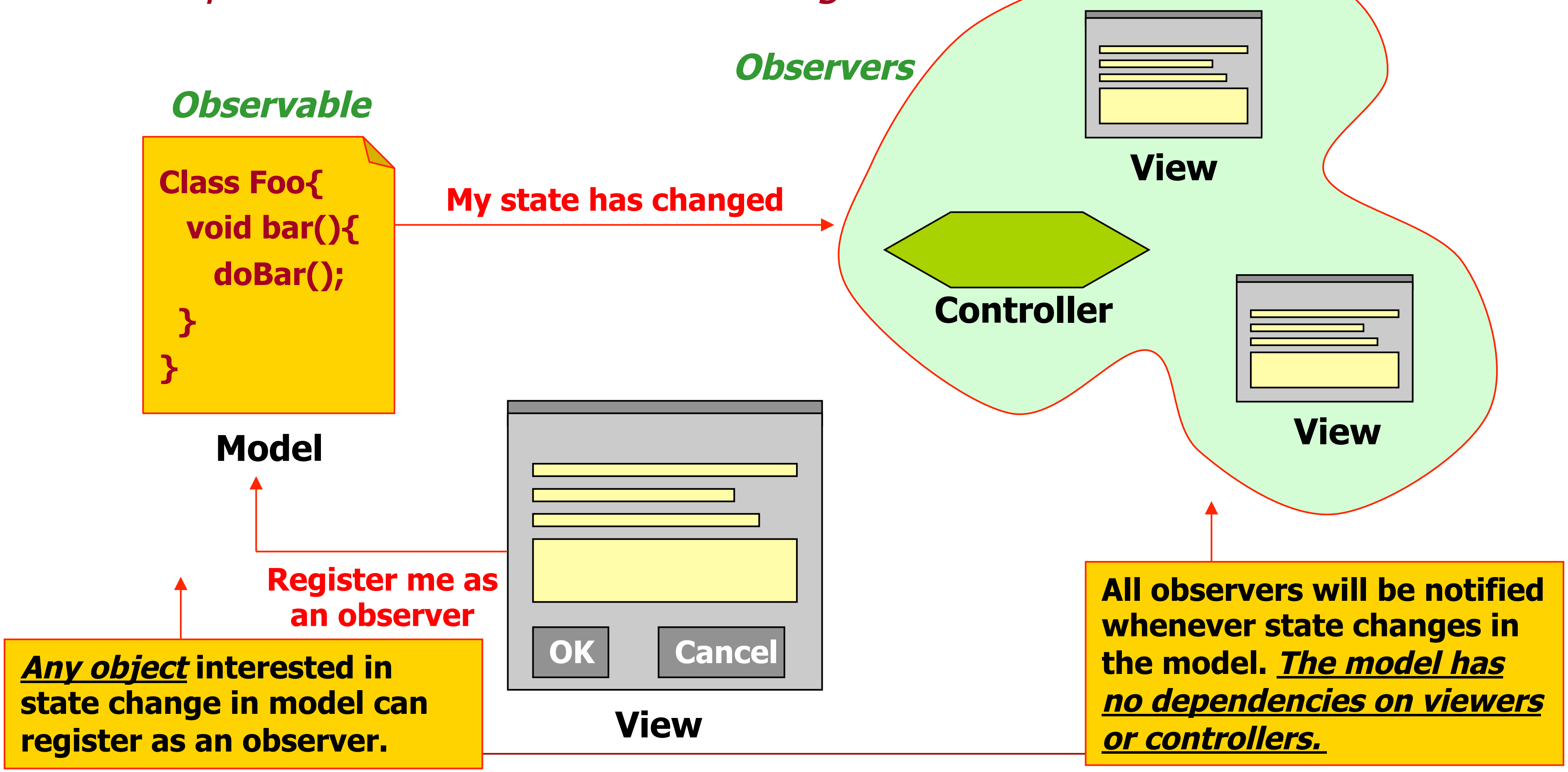
Concrete Visitor classes can use Class and Method classes to invoke correct method on visitee.

Model-View-Controller (MVC)



MVC – The Observer

- *The model uses an observer to keep the views and controllers updated on the latest state changes.*

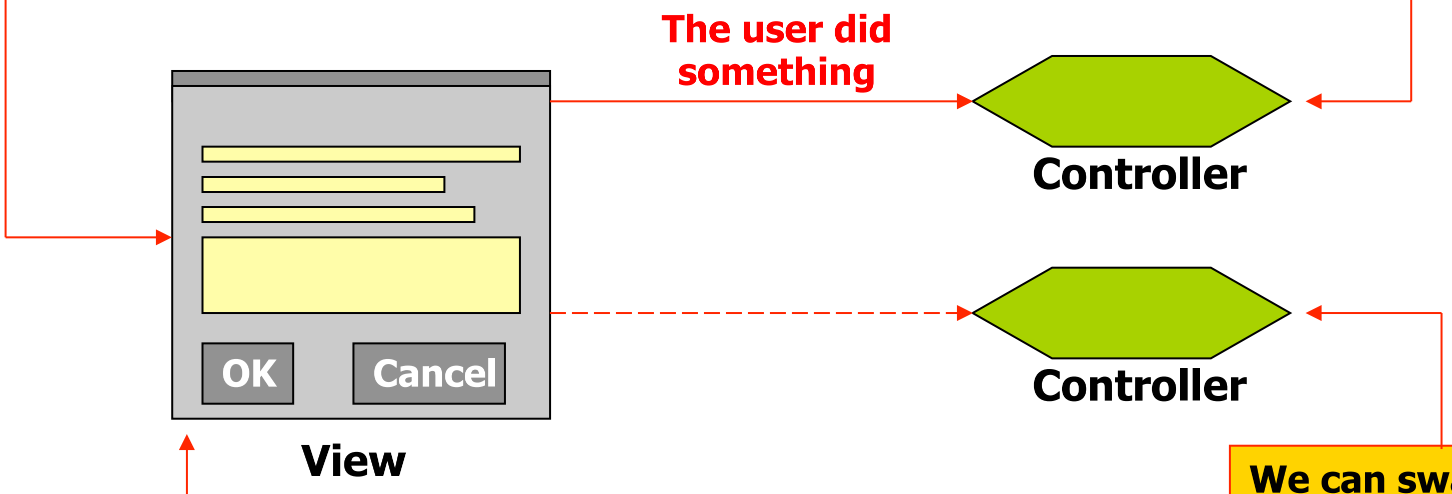


MVC – Strategy

- *The view and controller implement the strategy pattern.*
  – The ***controller is the behaviour of the view*** and can easily be swapped with a different controller for different behaviour.

The view delegates to the controller to handle the user actions.

The controller is the strategy for the view. It is responsible for handling the user actions.

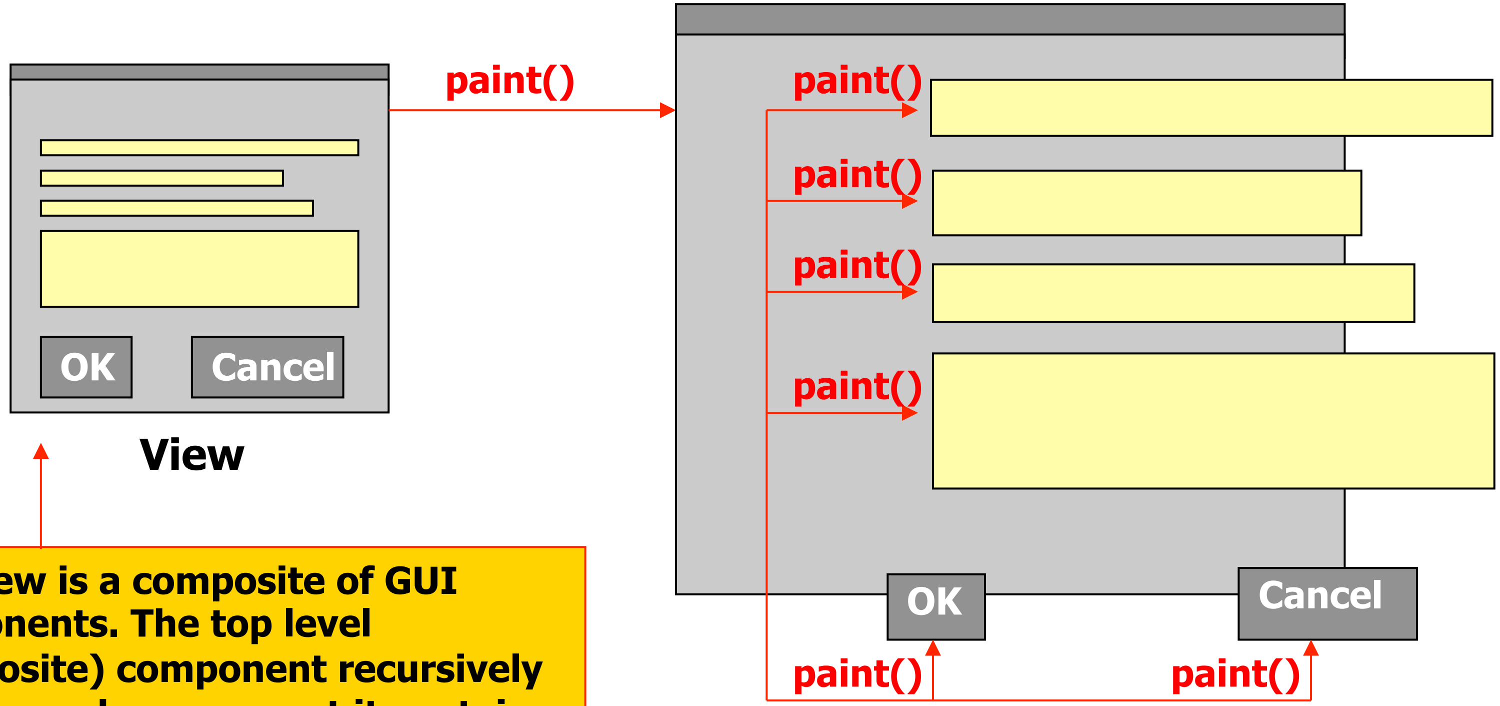


We can swap another behaviour for the view by changing the controller.

The view's single responsibility is the presentation. The controller is responsible for translating user input to actions on the model.

MVC – Composite

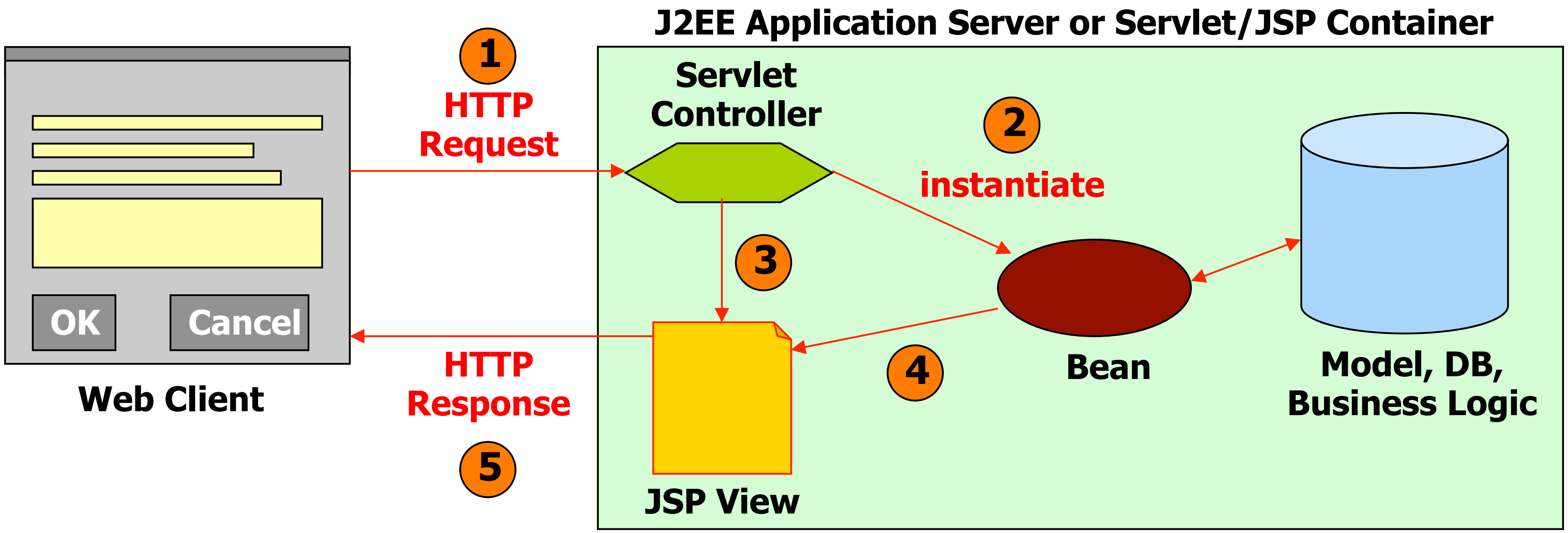
- *The view uses a composite pattern to manage the windows, buttons and other components of the display.*
  - In Swing, each component may use a ***template hook*** to override/ extend the look of the component by overriding ***paint()***.



The view is a composite of GUI components. The top level (composite) component recursively invokes each component it contains, until all composites and leaf nodes have been called.

Model 2

- MVC has also been applied to web applications.
  - Prevailing adaptation is known as ***Model 2***. Uses a combination of servlets and JSPs to achieve same separation of model, view and controller.



Servlet acts as controller and process request by making invocations on the model (a database). Result of processing usually bundled into a bean. JSP presents view (HTML and JavaScript) by accessing data encapsulated in the bean.

Design Patterns in JUnit 3