

Lecture 17: Patterns Potpourri

GOF Patterns

- GOF: Gamma, Helm, Johnson, Vlissides
 - Design Patterns, Addison Wesley, 1995
- Patterns we have seen so far
 - Creational Patterns
 - e.g. Factory, Singleton
 - Structural Patterns
 - e.g. Wrappers (Decorator, Adapter)
 - Behavioral Patterns
 - e.g. Observer/Observable, State

Some Additional GOF Patterns

- Structural
 - Bridge, Composite (Compound), Flyweight, Facade, Proxy
- Behavioral
 - Command, Iterator, Mediator, Chain of Responsibility, Visitor, Memento, Strategy

Other Patterns

- Misc: Indirection, Pure Fabrication,
- System architecture
 - layers, filters

Bridge Pattern

- Purpose: Decouple an abstraction from its implementation so that the implementation can be easily changed
- Strategy: make the implementation details an object that is a component in the abstraction
- E.g. Graphics objects such as Window have a WindowPeer component that is constructed to work for a particular graphics platform

Composite Pattern

- Purpose: build arbitrarily large, complex objects from simple parts.
- Strategy:
 - Use recursion

Recursion

- Languages, e.g.

char $\rightarrow$ A | B | ... | Z

alphaString $\rightarrow$ char | string + char

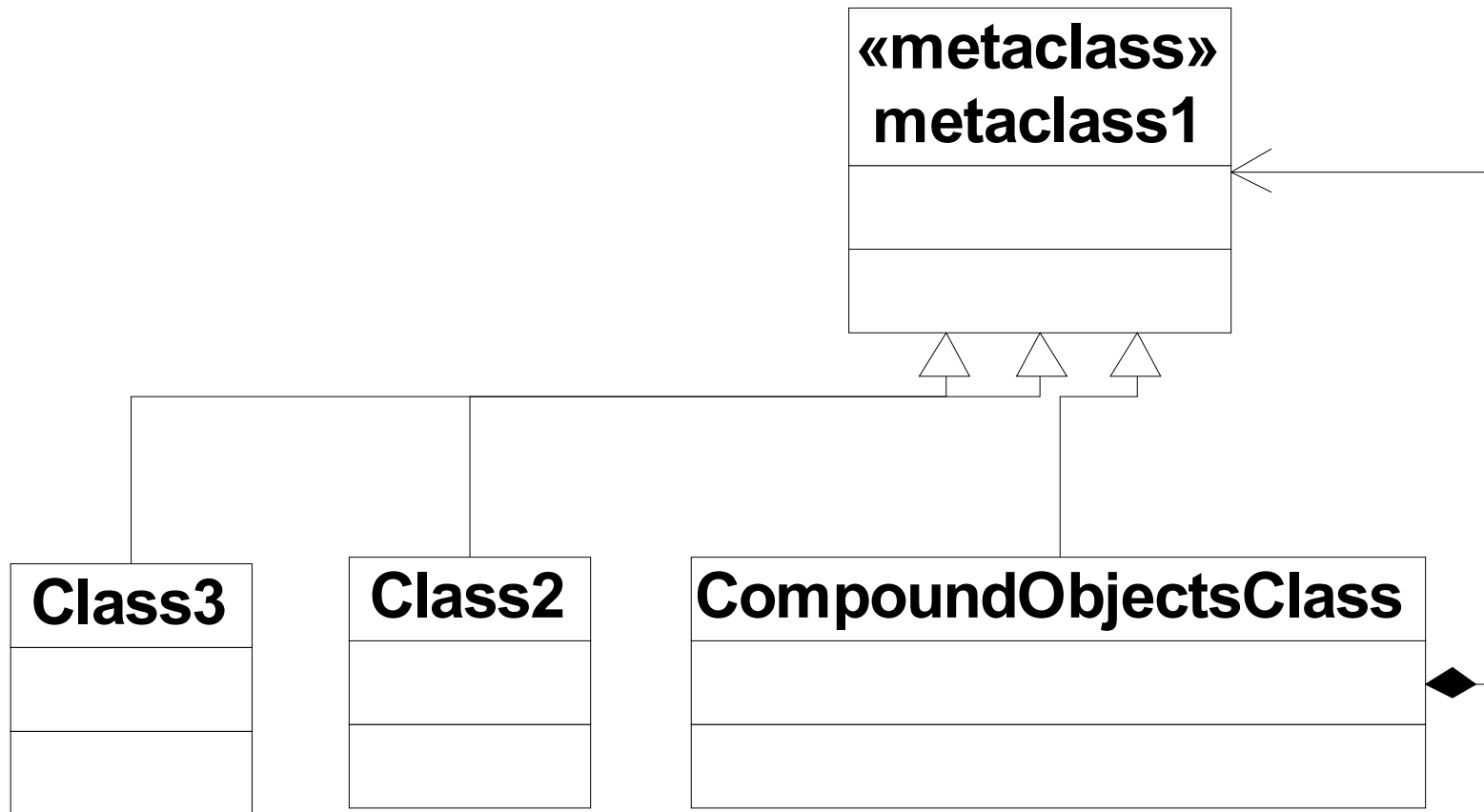
- Functions, e.g.

factorial(n) = {if n=0 then 1 else n*factorial(n-1)}

- Classes ?

Compound objects are constructed from parts that are instances of subclasses of an abstract class, of which the compound object is itself a subclass

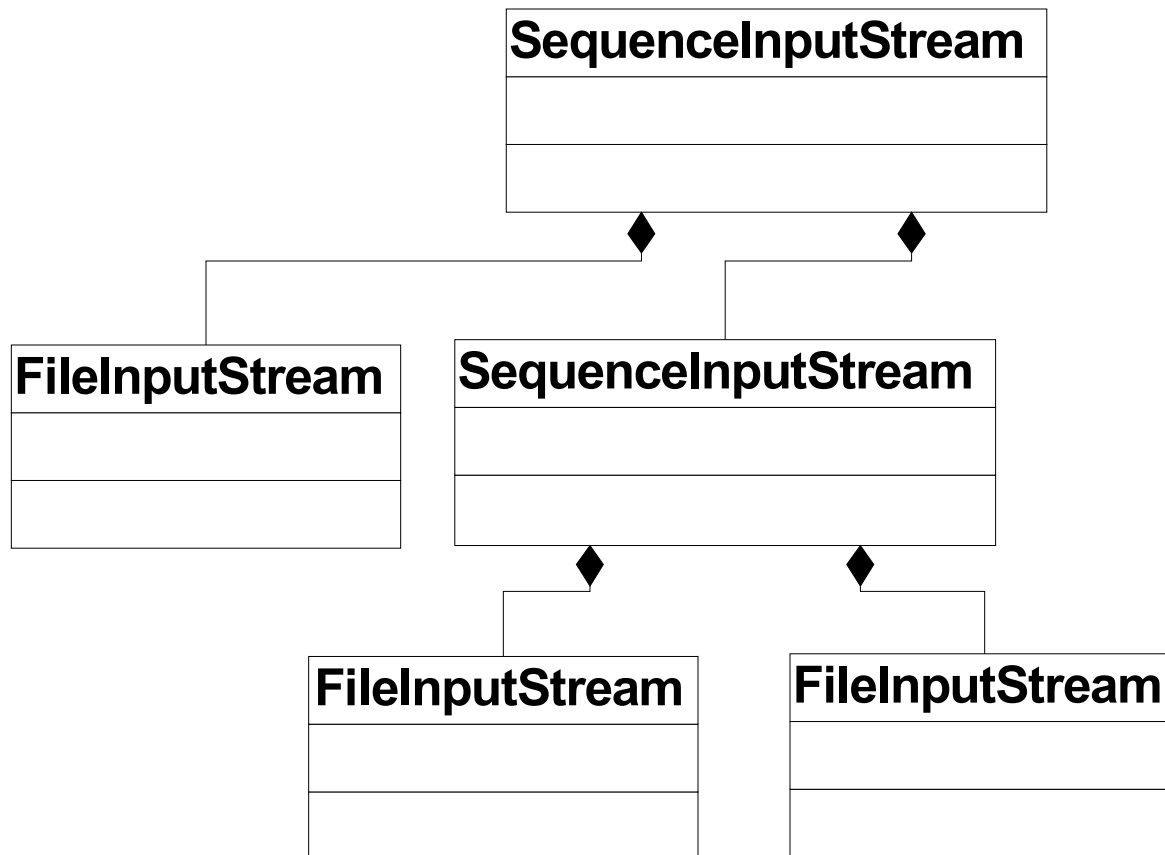
Composite Structure



Composite Example 1

- SequenceInputStream constructor can take 2 or more concrete instances of InputStream and concatenate them to produce a new concrete instance of Inputstream
- Abstract class: InputStream
- Compound Object: SequenceInputStream

Sample SequenceInputStream Structure



Composite Example 2

- Graphics structures: Picture is composed of instances of subclasses of Graphic abstract class
- Subclasses of Graphic:
 - primitive objects: Line, Rectangle, etc.
 - compound object: Picture

Flyweight Pattern

- Purpose: large numbers of objects that simultaneously share a significant amount of information/internal state. Excessive storage needed for redundant information
- Strategy: create a new class/object which the similar objects can share.
 - GOF: flyweight is the shared object(s)
 - Others: flyweights are the objects that share the information

Flyweight Example 1

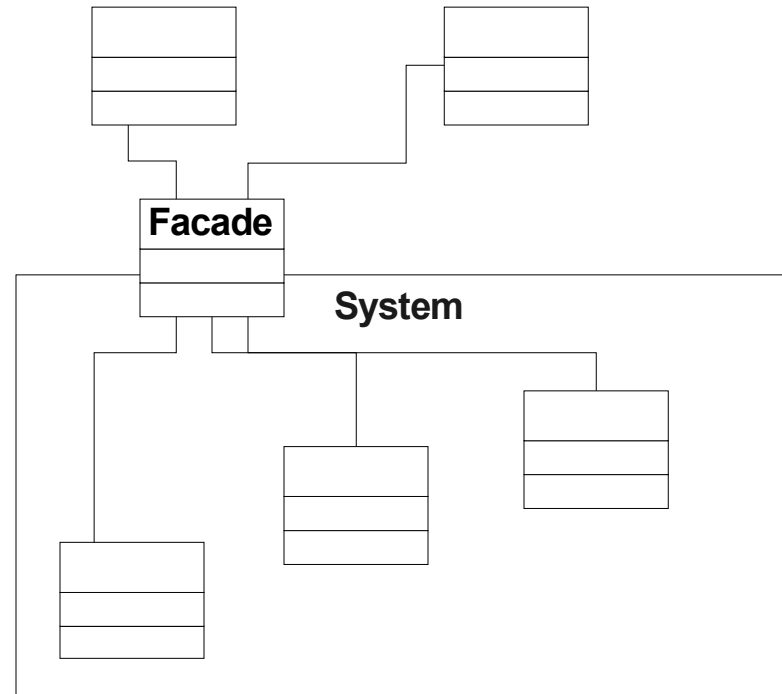
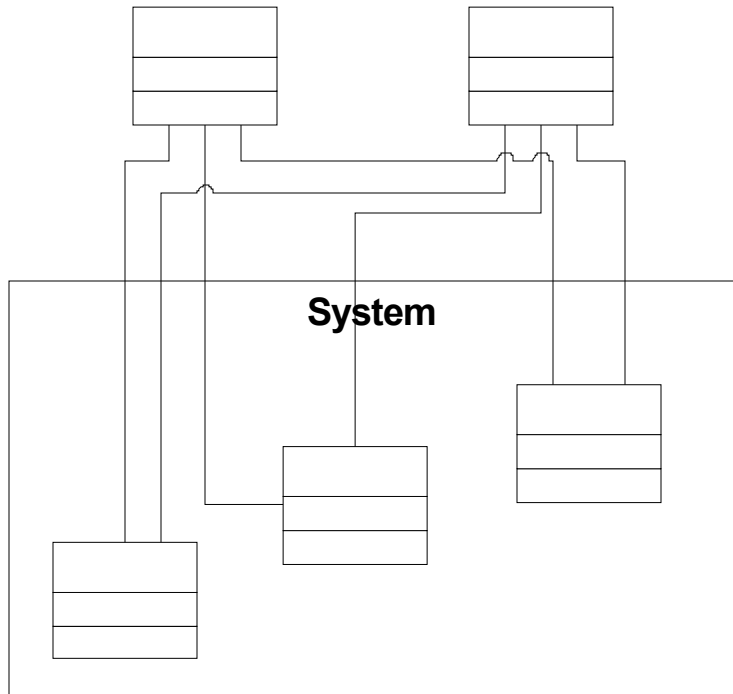
- Every instance of a class C has the following shared information
 - name of class
 - interface specifications
- Common information for instances of a class is stored in an instance of the Class class
- Each object “points” to the common Class instance for its class
- all objects are flyweights with respect to their Class information (definition 1 of “flyweight”)

Flyweight Example 2

- Document consists of rows of characters
- Objects: document, row, character
 - Character object attributes: character code, location, font
 - Character object methods: draw() (can draw itself)
- Remove common, character code information and store it in shared objects
- New draw method: draw(font, location)

Facade Pattern

- Purpose: Provide a unified interface to a set of interfaces in a subsystem.
- Rationale: Makes system easier to use, promotes low coupling, supports layering, re-use
- DS Example: the Business/Domain Logic and the DataBase subsystems have facade classes that serve as subsystem controllers and interfaces



Proxy Pattern

- Purpose: Provide a way to delay or ignore implementation considerations
- Strategy: use a surrogate or placeholder for another object to control access to it
- Kinds of
 - remote: local representative for object in a different address space
 - virtual: create expensive objects on demand
 - protection: controls access
 - smart reference: e.g. keep track of pointers to an object

Proxy Pattern (cont')

- Rationale:
 - defer cost of creation and initialization until needed
 - has an interface that looks like the real thing
- DS examples
 - DB virtual proxy (deferred construction).
During the first phase of the DS we might build a virtual proxy for testing purposes, and substitute a real one later

Proxy versus Facade

- Facade relates to implemented/existing classes/services
- Proxy contains the idea that the real objects may not yet exist
- Proxy and Facade interfaces may be the same
- May replace a proxy with a facade

Pure Fabrication Pattern

- Purpose: Collection of cohesive set of responsibilities with no domain concept
- Strategy: Create an “artificial” object/class to take care of a responsibilities. Improves cohesion
- DS example: DomainLogic is the controller/interface class for the DomainLogic subsystem. No such concept in the problem domain

Pure Fabrication and Other Patterns

- Pure Fabrication versus Facade and Proxy
 - Pure Fab involves idea that resulting object is not a part of the problem concept domain
 - could be a proxy
 - could be a facade
 - could be a “non-interface” object
 - proxy and facade may not be pure fabrication, i.e. they could correspond to domain concepts

Command Pattern

- Purposes:
 - Decouple knowledge of how to execute command from invoker of command
 - Allow a command to be stored, transmitted, without knowledge of command's properties
- Strategy: encapsulate command as an object
 - has its own execute command
 - may have receiver attribute that is set by command creator
 - set of related commands may be instances of classes that are subtypes of an abstract command class

Command Example 1

- Suppose we have a distributed DS, and the client needs to send a DB command to the server.
- Client creates a command, setting attributes, etc
 - command has an execute method that knows how to execute a command associated with object
- Transmit command to server
- Possible pros
 - easy queuing of and execution by server
 - may use composite pattern for multi-commands
 - easy to add and delete different kinds of commands

Command Example 2

- Application: Java 2 Event Delegation Model
- When a button is pushed a “command” must be executed
- Commands are encapsulated as ActionListener objects and attached to the control
- Their actionPerformed() method is called (not quite an “execute” method)
- Command objects can be attached to multiple controls

Iterator Pattern

- Purpose: Iterate over a collection of objects without exposing its implementation structure
- Strategy: An intermediate object is used that knows the implementation structure but presents an implementation independent iteration interface to clients

Iterator Example

- Java Collection Classes (List, Set, Sorted) are subclasses of the Collection class, and inherit an iterator() method that returns an Iterator object.
- Iterator object has standard methods such as hasNext(), next()

Iterator Sample Code

- Suppose that collection is a HashSet() object.
 - This is a Set entity implemented using a HashTable.
 - Will not add duplicates because it is a set. Iterator
iterator = collection.iterator();
while (iterator.hasNext())
{ System.out.println(iterator.next())
}

Mediator Pattern

- Purpose: Chaotic set of interactions between a set of objects/classes.
- Strategy: Delink interactions between objects and make them pass through a single mediator object
- Similar to Facade Pattern
 - Facade is unidirectional, providing an interface between a collection of classes in a subsystem from users outside the subsystem

Chain of Responsibility

- Purpose: Allow different possible objects to take care of a request. Flexible changes to system later on.
- Strategy
 - i) decouple initiator of request from responder
i.e. initiator will not know the exact responder
 - iii) give multiple objects a chance to handle the request
by passing down a chain of potential responders

Chain of Responsibility Example

- DS: GUI gets a getDate request from user and constructs a request object R
- GUI passes R to DomainLogic interface
 - checks to see if it is Pamela, if so return Dave
 - if not send request to DateMatcher
 - builds appropriate request object and tells it to execute itself

Visitor Pattern

- Purpose: We have a collection of one or more operations to be performed on a collection of possibly diverse objects. We want to be easily able to add new operations without changing the classes of the elements it operates on
- Strategy: remove the operations from the classes/objects and put them in a “visitor” that visits them, i.e. pass the function around to the data and have the data execute the appropriate version of the function

Visitor DS Example

- Restaurant reports
 - sent in by restaurant
- Data
 - start time, finish time, type of meal, alcohol consumed, type of desert, etc.
- Operations on data
 - per report
 - validate (e.g. time for dinner = 27 hrs?)
 - process (e.g. cheapness = entre cost/average entre)
 - per report collection
 - process (e.g. average time at dinner)

Visitor DS Example (Cont.)

- Assumptions
 - different kinds of restaurant reports used
 - basic data in reports does not change
 - may wish to change operations, or add new operations
- Possible approaches
 - functional, expert pattern, visitor pattern

Functional Approach

- Data is held in records
- Processing function or procedure contains all the functionality
 - reads in data records one at a time and processes their data
- has a big switch statement with an entry for each of the different versions of reports
- Run different programs for different functions

Expert Pattern Approach

- Each report is an instance of a subclass of an abstract report class (or interface)
- The abstract class contains operations for each of the operations to be performed on an object
- Some kind of iterator passes over the object collection, and tells each object to perform related operations, such as `result = x.validate`, or `result = x.romanceIndex`

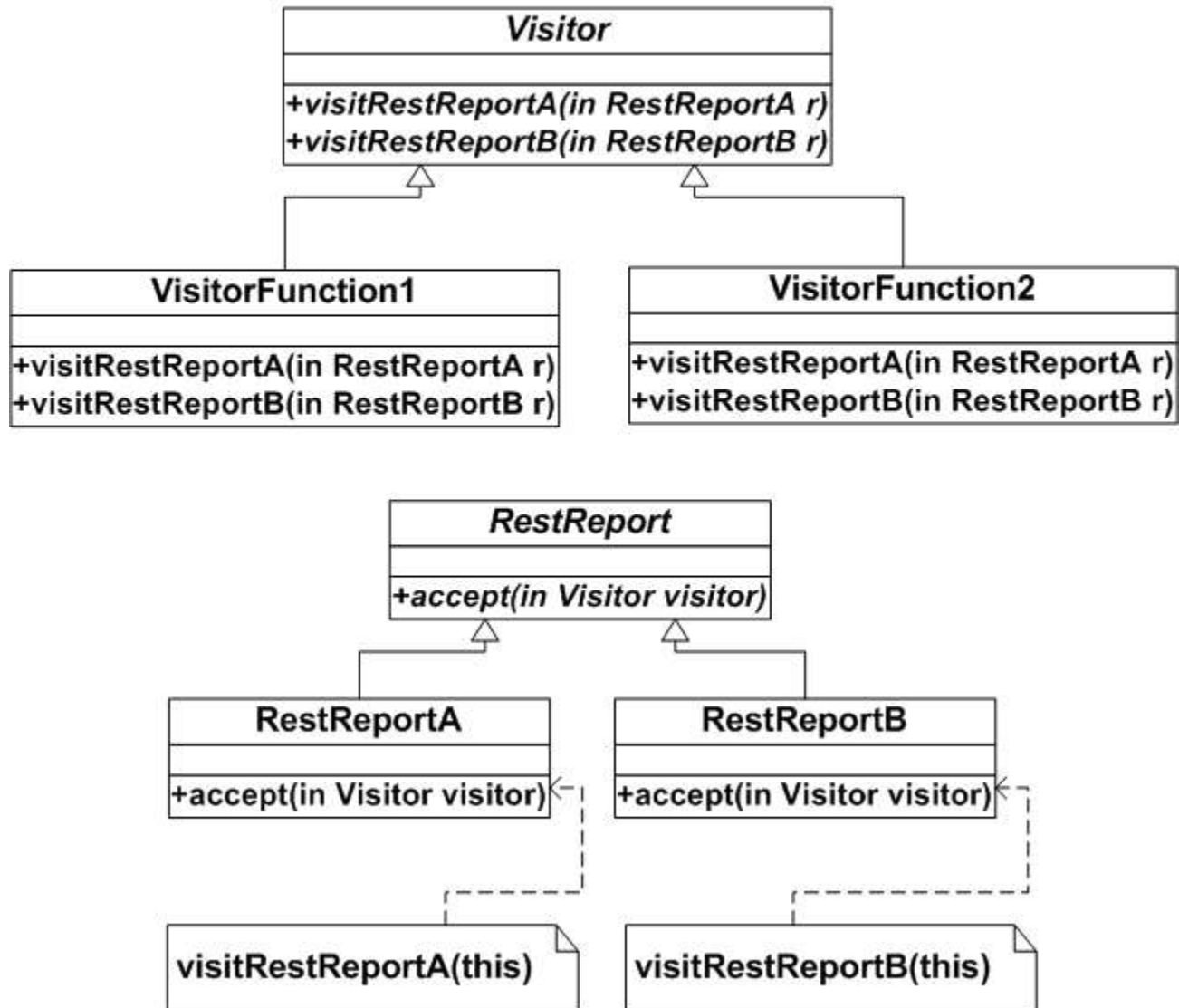
Visitor Approach

- Encapsulate methods to be performed in functional objects
 - different versions of methods for different versions of reports
- Pass the functionality object to a data object which then performs the appropriate version of a method in the functionality object

DS Visitor Pattern Classes

- Visitor abstract class with a method for each version of restaurant report
- Visitor subclass for each kind of functionality to be performed
- Report abstract class that includes an accept (Visitor v) method
- Report subclasses for each version of report

Visitor Pattern Class Structure

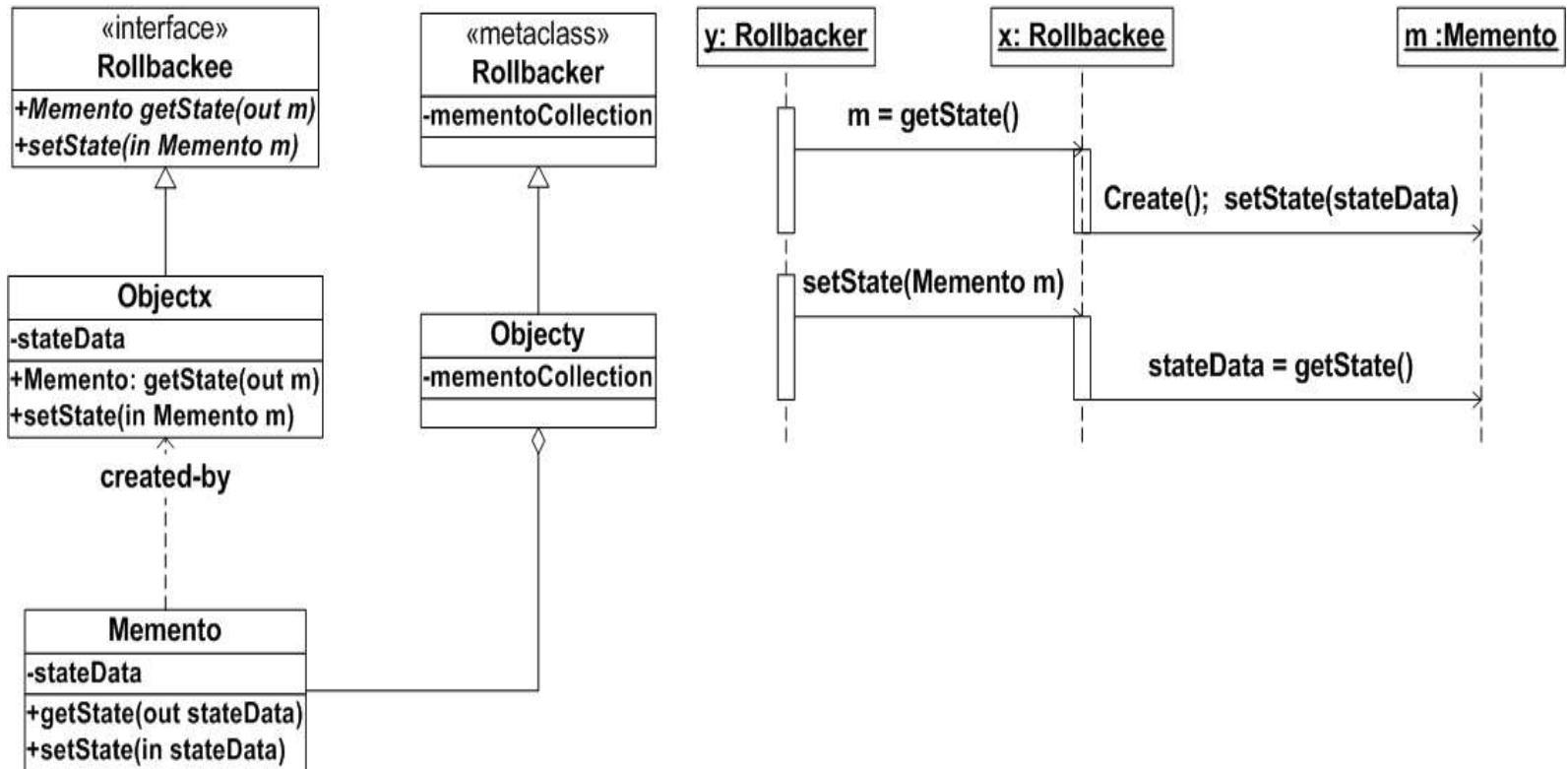


DS Visitor Strategy

- For each object x in the report object collection
 - execute its `visit(Visitor v)` method with a visitor v that is an instance of the subclass for the kind of function we are performing
 - the object x knows what version of report it is, i.e. it's subclass S , so it calls the corresponding visitor method `visitS(x)`, which performs the appropriate function

Memento Pattern

- Purpose: make it possible for an object to return to a previous state, by saving state in a memento object
- Strategy:
 - object that might need to be rolled back supports a rollbackee interface
 - object that recognizes that a rollbackee object may need to be rolled back is a rollbacker subclass
 - rollbacker sends a getState message to rollbackee which returns a Memento object that is saved by the rollbacker
 - rollbacker sends a setState message to rollbackee with a Memento from which state can be restored



Memento Notes

- May not actually have Rollbackee, Rollbacker, and Memento metaclasses
 - e.g. in Java do not have multiple subclasses so maynot be able to subclass Objecty from a Rollbacker supertype
- Need to define Memento objects in a way that only the Rollbackee can access its saved state data, i.e. it is hidden to Rollbacker object. Java? Friend Classes?

DS Memento Example 1

- User actions, involving getDate dialog
 - enter data (preferences) and click find
 - click undo to go back to previous choice
 - system restores old data entry and old answer
- getDate is both rollbackee and rollbacker
 - find method will create a memento which will be stored in getDate.
 - undo will get state back from memento and execute setState(stateData) in getDate

DS Memento Example 2

- DS properties
 - User can ask for getDate() or redo, (backs up to last answer.)
Can also ask for date repeatedly, changing data each time
 - Suppose :DateRequest has internal data that is determined by the sequence of data requests so far, and uses it to return advice along with the answer to the current request
 - System keeps the same DateRequest object for a user session
Before each request except the first, :DL asks the date request for a memento in case there is a redo
 - If there is a redo, system sends DateRequest a setDate (memento) message
 - this is so subsequent answers will not involve the part of the request sequence that has just been undone, i.e. cannot just return the previous answer

Example 2 Comments

- Why not have DateRequest remember its previous state information so it can back up if requested?
 - DL is the controller here, and it may be better design for it to manage the mementos, and isolate the changes to be made if we do things like allow multiple redos in a row.

Strategy Pattern

- Purpose: to allow a program that uses variations of an algorithm to use the one that is appropriate to the variation of the application. Focus on time-space tradeoff differences rather than different functionality
- Strategy: make the different algorithms instances of subclasses of the same interface class.

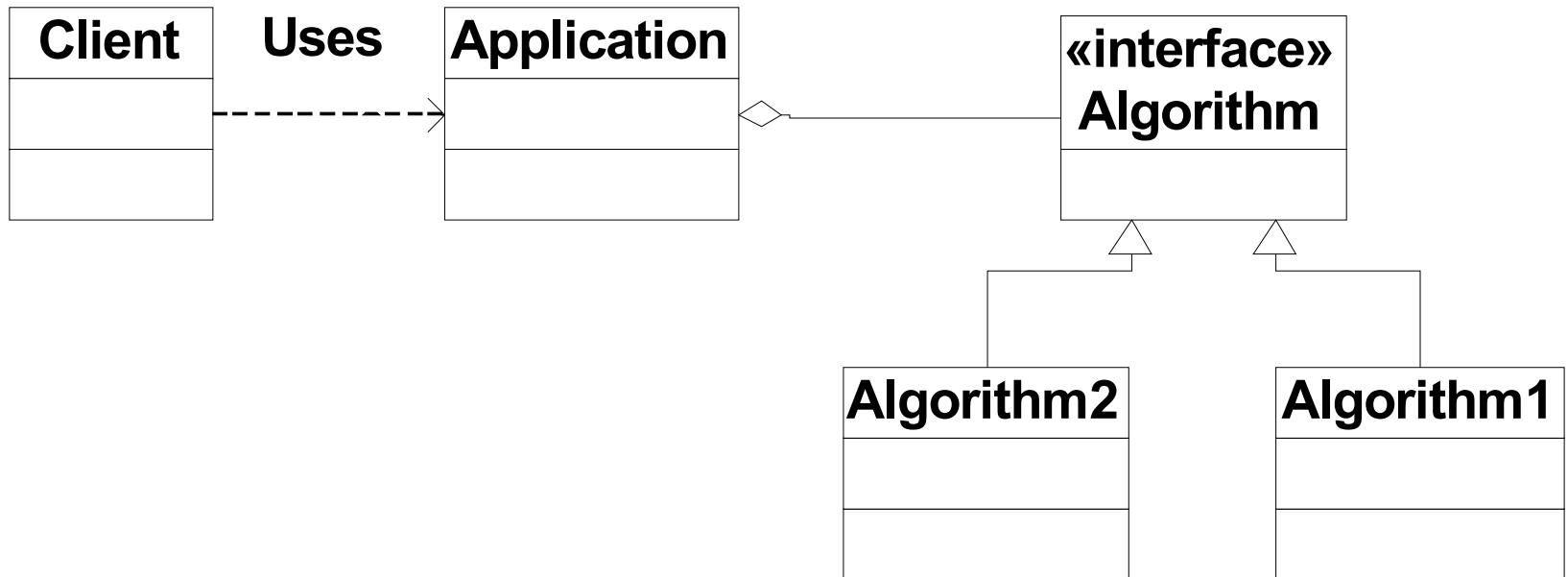
Possible Approaches

- Client uses Application which uses Algorithm
 - i) Functional: big switch that chooses an algorithm
 - ii) Inheritance: Application is a subclass of the chosen algorithm
 - iii) Factory: has an abstract Algorithm create() method. Subclass Application with a concrete method that returns a concrete instance of subclass of Algorithm
 - ✓ iv) Installation: Client calls method (e.g. constructor) in Application with instance of Algorithm subclass

Chosen Approach

- Client knows which algorithm to use. Client “installs” the appropriate algorithm in the application.
- ii) Client is a program that creates an instance of the correct subclass of Algorithm and passes it in the constructor for the Application, which has an abstract Algorithm parameter. Application accesses Algorithm instance via a class variable, or
- iii) Client sends an install message to Application object with an instance of the correct subclass of Algorithm

Class Relationships



DS – Strategy Example 1

- Programmer can choose between different layout managers for a frame
- E.g. for the Message Dialog object, we used
`buttonPanel.setLayout(new FlowLayout());`
- Other possible layout managers:
 - Border, Grid, Card, etc.

DS- Strategy Example 2

- 2 versions of the data base
 - i) “small” amounts of data - stored in memory
 - ii) data stored in secondary memory
- DataBase is an interface with subclasses for the different data base strategies
- Start up creates an instance of the appropriate DB subclass which is passed to DL when it is created
- DL has an attribute variable whose value is a DB

Comments on DS Example

- Combines the following patterns
 - Facade and Proxy for the DB class
 - Strategy for the idea of creating DL with different possible DB subtypes
 - Creator for who has the initializing information for DB

Strategy versus Bridge

- Both have separated functionality implementation choices that are subtypes
- Bridge: application user is unaware of separated implementation (choice)
 - e.g. choice of peer class for AWT components
- Strategy: application user
 - e.g. choice of layout manager when frame is created