OOA/OOD Example wrapup
(hour1)
Pattern Intro
(hour2)

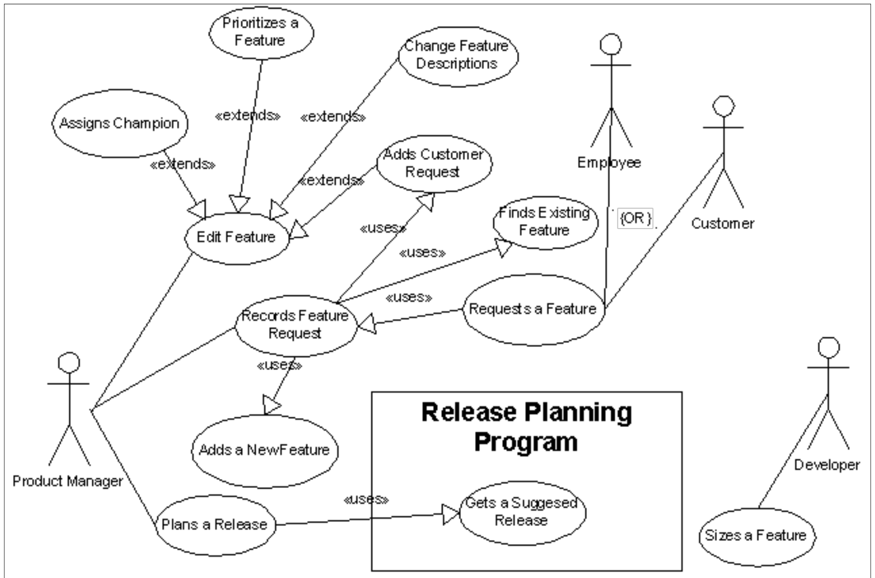
See `http://www.cs.toronto.edu/~matz/instruct/csc407/eg`

Admin/Housekeeping

- 30 min office hour cum "Q + A" following lecture in BA1170 so long as no special event needs the room
- Who conveniently reads the newsgroup from home?
  - It looks to me like news.cdf.toronto.edu is open for NNTP.
- A2(a) is in final draft.
  - part a OOA
  - part b OOD
- Tutorials
  - Do read the slides over
  - This week will feature repeat performance with more detailed script.

Which use cases

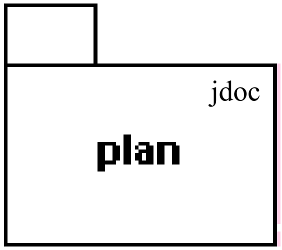
- Oftentimes only a small fraction of a system is carried out in software.
- It's nice to see how the software fits into the rest of the workplace.
- Use cases are good for this.
- Oftentimes projects start out ambitious and contract.
  - Prioritizing use cases help trim the fat.



Divide and Conquer

- Like almost any complicated effort we need a way of attacking our design in pieces.
- Packages, over and above any Java implementation issues, are a way of focusing our design activities.
- Packages are a good way of separating our documentation into sections.

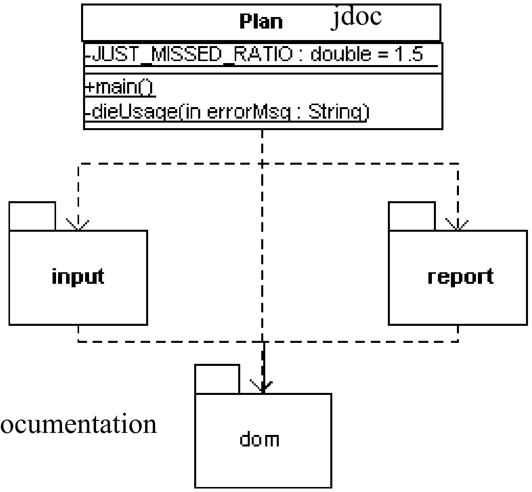
Top Package



jdoc

plan

Package Plan



Plan

jdoc

-JUST_MISSED_RATIO : double = 1.5

+main()

-dieUsage(in errorMsg : String)

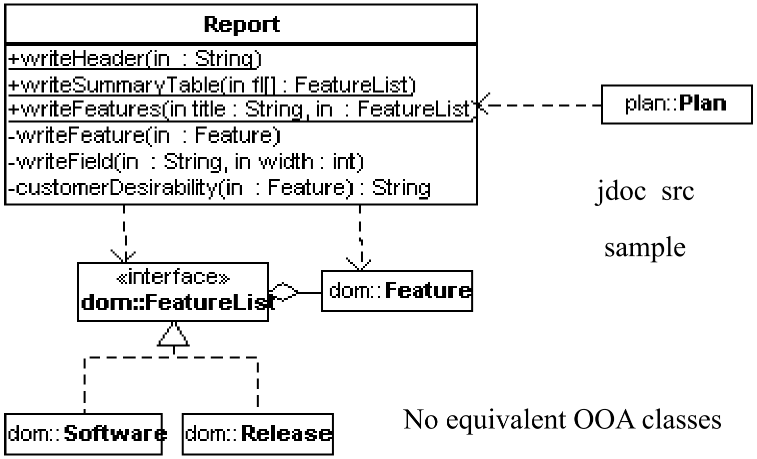
input

report

dom

package documentation

Package Report



Report

+writeHeader(in : String)

+writeSummaryTable(in fl[] : FeatureList)

+writeFeatures(in title : String, in : FeatureList)

-writeFeature(in : Feature)

-writeField(in : String, in width : int)

-customerDesirability(in : Feature) : String

plan::**Plan**

«interface»

dom::FeatureList

dom::**Feature**

dom::**Software**

dom::**Release**

jdoc src

sample

No equivalent OOA classes

An OOA can be overly general

- In the early stages of an OOA it is usual to create domain models that are more general than than the design models that are eventually created.
- To emphasize this point we will consider a few associations from the point of view of navigability.
- We will see that a design can be simplified considerably if only the required navigability is built.
- On the other hand the extensibility of a system can be reduced if this is carried too far.
- Consider “develops” association between Company and Software.
  - For in house application there is only one company..
  - A merger or two and.. oh oh.

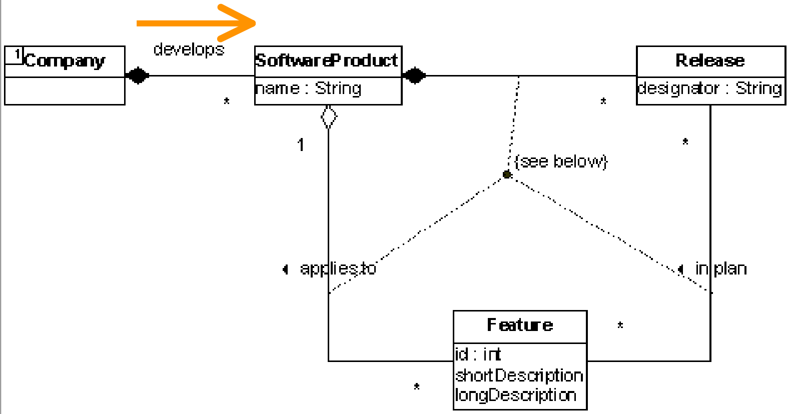
Navigation

- It is often not necessary to implement associations between classes as generally as the OOA might imply.
- When software actually runs we need to get from one object to another.
- One of the decisions that can be made at design time is that a given program only navigates an association in one direction.
  - Whereas the process by which OOAs are done makes it unlikely to have been noticed. (Remember Point example?)
  - Significant opportunity for simplifying design.

Implementing Associations

- Decide on interface for
  - Navigating the links
    - usually get method for 1 side, iterator for * side.
  - Adding new links
  - Deleting links (if necessary)
- Decide on implementation
  - Simple pointer to implement the [0..1] side
    - (if required by navigatability)
  - Array, Vector, Map, Linked List to do the [*] side
    - (if required by navigatability)
- Persistence Warning
  - Keeping associations up to date in database can be big source of complexity.

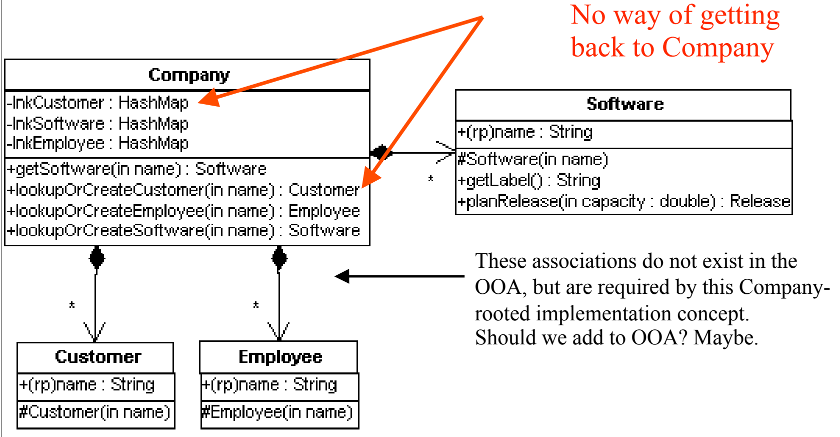
Features (from OOA)



ood

jdoc src

DOM Company



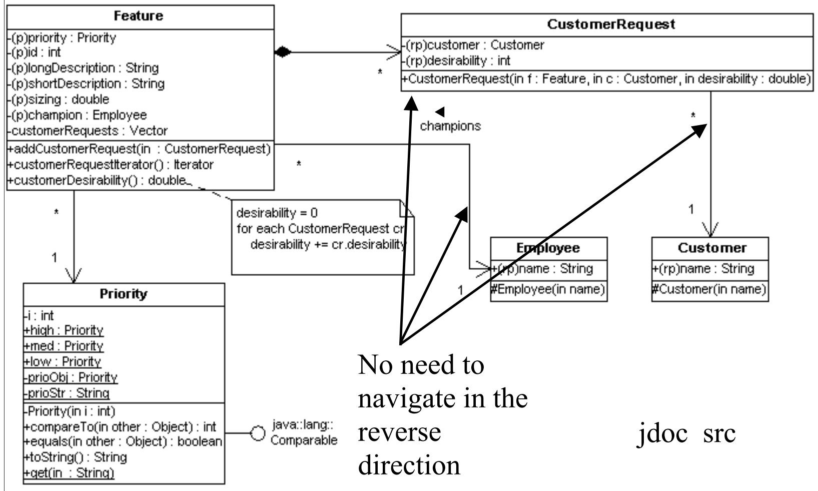
No way of getting back to Company

These associations do not exist in the OOA, but are required by this Company-rooted implementation concept. Should we add to OOA? Maybe.

Navigation of Features

- Example: To sort features we need to compare them.
- To compare features we need to compare priority and then desirability.
- Priority is easy
  - David goes to town with discrete value Priority class which implements Comparator
- Desirability is not so easy
  - We need to compare the total desirability of each Feature.
  - Thus we need to navigate from each Feature to its (multiple) CustomerRequests and add up the corresponding desirabilities.
- So, in fact, we need only to get from Features to CustomerRequests to do the sort.

ood



No need to navigate in the reverse direction

jdoc src

Prototyping

- When you start on a project there are often things that it is not clear how to accomplish.
- Probably pointless to design software until you know how to do it!
- Prototyping usually shows how to accomplish the task and also uncovers challenges to a clean structure for the software
- This knowledge should be integrated into your design.
- My position is that you ought to prototype to figure out how to do things, then you toss out most of the prototype and start working on OOD.
- This is different from RAD, I think.

Experiments show..

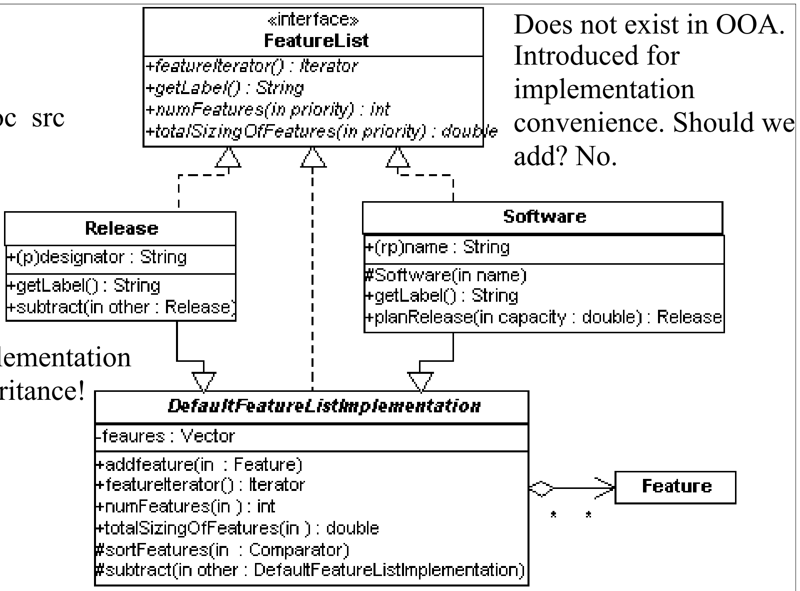
```
/** Suggests a release of this software product.
 *  @param capacity number of person-days of  effort available to work release
 *  @return a Release containing a suggested list of features
 */

public Release planRelease(double capacity) {
      double inplan = 0.0;
      //Sort in order of desirability somehow
      sortFeatures(ReverseFeaturePlanningOrder.get());
      Release r = new Release();
      for (Iterator i = featureIterator(); i.hasNext(); ) {
          Feature f = (Feature)i.next();
          if (inplan + f.getSizing() <= capacity) {
              r.addFeature(f);
              inplan += f.getSizing();
          }
      }
      return r;
  }
```

Design and Code factoring

- It's not just that we hate typing..
- It's not just that we hate fixing bugs twice..
- It's not just that we particularly hate looking for cloned code that has to be kept in sync..
- In fact the techniques we use to factor code has little to do with the structure of our classes so far.
- Inheritance can be used to explicitly factor out common behavior
  - This is NOT the "is-a" relationships we detected during OOA.
  - On the next slide a we don't mean to say that Releases and Software (products) are specializations of the same concept
  - Rather we are just packaging code in a way that makes sharing of methods to deal with lists of features explicit.

ood

jdoc src


«interface»
FeatureList
+featureIterator() : Iterator
+getLabel() : String
+numFeatures(in priority) : int
+totalSizingOfFeatures(in priority) : double
Release
+(p)designator : String
+getLabel() : String
+subtract(in other : Release)
Software
+(rp)name : String
#Software(in name)
+getLabel() : String
+planRelease(in capacity : double) : Release
DefaultFeatureListImplementation
-feaures : Vector
+addfeature(in : Feature)
+featureIterator() : Iterator
+numFeatures(in) : int
+totalSizingOfFeatures(in) : double
#sortFeatures(in : Comparator)
#subtract(in other : DefaultFeatureListImplementation)
Feature
*
*

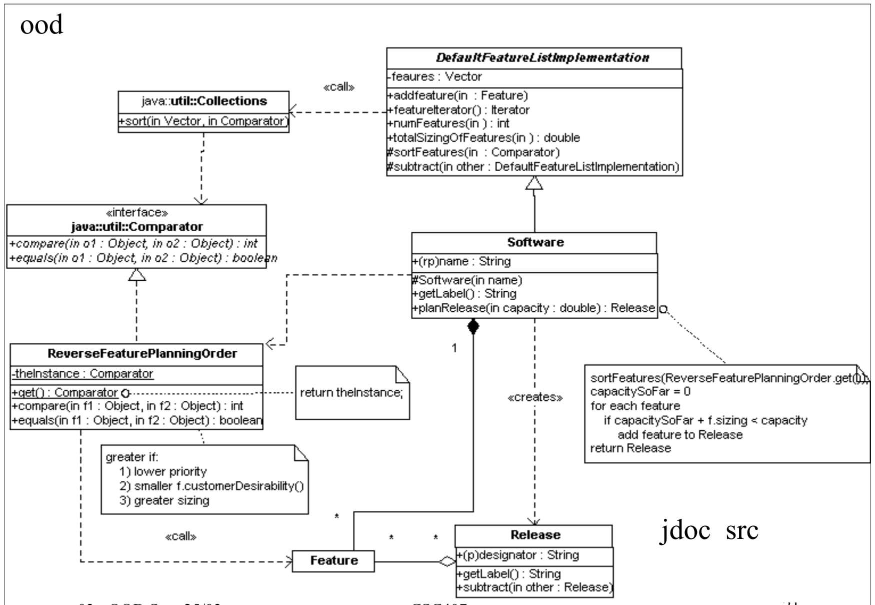
Does not exist in OOA. Introduced for implementation convenience. Should we add? No.

Implementation inheritance!

Taking the lead from existing designs

- Java already has a well thought out infrastructure for sorting Collections.
- `Collections.sort(List l, Comparator c)`
- Designing your own from scratch would be silly, right?
- This certainly involves a detailed design that is motivated by object oriented thinking -- but may have nothing to do with a particular OOA.
- So our Priority class implements the Java Foundation Classes `Comparator` interface.
- We introduce our a class ReverseFeaturePlanningOrder that implements `Comparator`.

ood


DefaultFeatureListImplementation
-feaures : Vector
+addfeature(in : Feature)
+featureIterator() : Iterator
+numFeatures(in) : int
+totalSizingOfFeatures(in) : double
#sortFeatures(in : Comparator)
#subtract(in other : DefaultFeatureListImplementation)
«call»
java::util::Collections
+sort(in Vector, in Comparator)
«interface»
java::util::Comparator
+compare(in o1 : Object, in o2 : Object) : int
+equals(in o1 : Object, in o2 : Object) : boolean
Software
+(rp)name : String
#Software(in name)
+getLabel() : String
+planRelease(in capacity : double) : Release
ReverseFeaturePlanningOrder
-theInstance : Comparator
+get() : Comparator
+compare(in f1 : Object, in f2 : Object) : int
+equals(in f1 : Object, in f2 : Object) : boolean
return theInstance;
1
«creates»
*
«call»
*
*
Feature
Release
+(p)designator : String
+getLabel() : String
+subtract(in other : Release)

```
greater if:
   1) lower priority
   2) smaller f.customerDesirability()
   3) greater sizing
```

```
sortFeatures(ReverseFeaturePlanningOrder.get());
capacitySoFar = 0
for each feature
   if capacitySoFar + f.sizing < capacity
      add feature to Release
return Release
```

jdoc src

Hey, is this interesting?

- We want our design to adapting to reuse existing object work?
- This is the motivation behind Software Patterns.
- More next lecture.
- Time for a Break.

Patterns

Genesis

- Christopher Alexander, *et. al.*
  - *A Pattern Language*
  - Oxford University Press, 1977
  -
  - *“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”*
  -
  - Talking about buildings, bridges and towns.
- (NB. His communities weren’t all smashing successes.)
- During the last decade, a “pattern community” has developed in the field of software design.

Design Patterns

- Designing good and reusable OO software is hard.
  - Mix of specific + general
  - Impossible to get it right the first time
- Experienced designers will use solutions that have worked for them in the past.
- Design patterns
  - Systematically
    - names,
    - explains,
    - and evaluates
  - important, recurring designs in OO systems.

Finding Appropriate Objects

- Hard part about OOD is decomposing a system into objects.
- Many objects come directly from the analysis model or from the implementation space.
- OO designs often wind up with classes that have no such counterparts.
  - E.g., Composite, Strategy, Sate

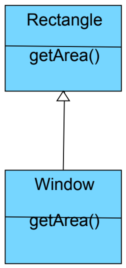
Determining Object Granularity

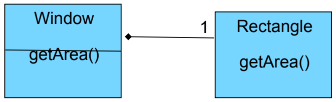
- Too large
  - Hard to change.
  - Procedural program inside an object.
  - Large, shared data structure.
  - Hard to understand
- Too small
  - Inefficiencies
    - Copied data
    - Method invocation overhead
  - Hard to understand
- Whatever the choice, negative consequences can be mitigated by judicious use of certain patterns:
  - Flyweight, Façade, Builder, Visitor, Command, …

Using Object Interfaces

- This is how Microsoft COM sees the world.
  - Can make the most sophisticated systems with no inheritance.
  - Can still use implementation inheritance under the covers.
- Never refer to a class by name. Always use interfaces.
  - Callers remain unaware of the specific types they use.
    - can extend the type structure
  - Callers remain unaware of the classes that implement the interfaces.
    - can dynamically load new implementations
- Sometimes difficult to put into practice.
  - Creational patterns help a great deal.

Inheritance v.s. Composition

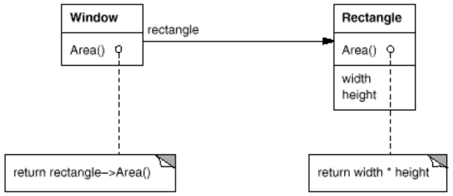




- Inheritance
  - Defined statically
  - Easier to modify sub-class (language supported)
  - Can affect behavior indirectly

- Composition
  - Can change implementations at run-time
  - Does not break encapsulation
  - Less “uselessly” general

Delegation



- Can implement inheritance using delegation.
- Makes it easier to compose behaviours at run-time (e.g., Window can become circular at run-time)
- Many design patterns rely on delegation.
-

Design Patterns in General

- Pattern name
  - A word or two that increases our design vocabulary
- Problem
  - Describes when to apply the pattern.
- Solution
  - Describes the elements that make up the design:
    - Responsibilities, relationships, collaborations
    - A general arrangement of classes
      - Must be adapted for each use
- Consequences
  - Results and trade-offs of applying the pattern
    - Space & time
    - Implementation issues
    - Impact on flexibility, extensibility, portability

Design Patterns Specifically

- **Pattern name and classification**
- **Intent**
  - What does it do? What’s its rationale
- **Also knows as**
- **Motivation**
  - A use scenario
- **Applicability**
  - In what situations can you apply it? How can you recognize these situations.
- **Structure**
  - UML
- **Participants**
- **Collaborations**
- **Consequences**
  - Trade-offs in applying this pattern
- **Implementation**
  - Any implementation tips when applying the pattern
- **Sample code**
- **Known uses**
- **Related patterns**

Design Pattern Coverage

- In this course, we will cover a limited number of very basic design patterns.
- This is only a fraction of what a real expert might know.

Design Pattern Space

		Purpose				
		Creational	Structural	Behavioral	Storage	Distributed
Scope	Class	Factory method	Adapter Template Base	Interpreter Template Method	Object File RDB Direct	
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor	OODB Proxy	Attribute Factory

Scope

- Class Patterns
  - Relationships between classes and their subclasses
  - No need to execute any code to set them up
  - Static, fixed at compile-time
- Object Patterns
  - Relies on object pointers.
  - Can be changed at run-time, are more dynamic.

Purpose

- Creational
  - Concerns the process of object creation
- Structural
  - Concerns the relationships between classes and objects
- Behavioral
  - Concerns the ways objects and classes distribute responsibility for performing some task.
- Storage
  - Concerns the ways objects can be made persistent.
- Distributed
  - Concerns the ways server objects are represented on a client.

Creational Patterns

- Class
  - Factory Method
    - Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Object
  - Abstract Factory
    - Provide an interface for creating families of related objects without specifying their concrete classes.
  - Builder
    - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - Prototype
    - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - Singleton
    - Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

- Class
  - Adapter
    - Convert the interface of a class into another interface clients expect.
  - Template Base
    - Use templated base classes to specify associations.
- Object
  - Adapter
    - Convert the interface of a class into another interface clients expect.
  - Bridge
    - Decouple an abstraction from its implementation so that the two can vary independently (run-time inheritance)
  - Composite
    - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structural Patterns (cont’d)

- Object (cont’d)
  - Decorator
    - Attach additional responsibilities to an object dynamically.
  - Façade
    - Provide a unified interface to a set of interfaces in a subsystem.
  - Flyweight
    - Use sharing to support large numbers of fine-grained objects efficiently.
  - Proxy
    - Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

- Class
  - Interpreter
    - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
  - Template Method
    - Let subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Object
  - 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.
  - Command
    - Encapsulate a request as an object.
  - Iterator
    - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
  - Mediator
    - Define an object that encapsulates how a set of objects interact.

Behavioral Patterns (cont'd)

- Object (cont'd)
  - Memento
    - Capture and externalize an object's internal state so that the object can be restored to this state later.
  - Observer
    - When one object changes state, all its dependents are notified and updated automatically.
  - State
    - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
  - Strategy
    - Define a family of algorithms, encapsulate each one, and make them interchangeable.
  - Visitor
    - Represent an operation to be performed on the elements of an object structure.
-

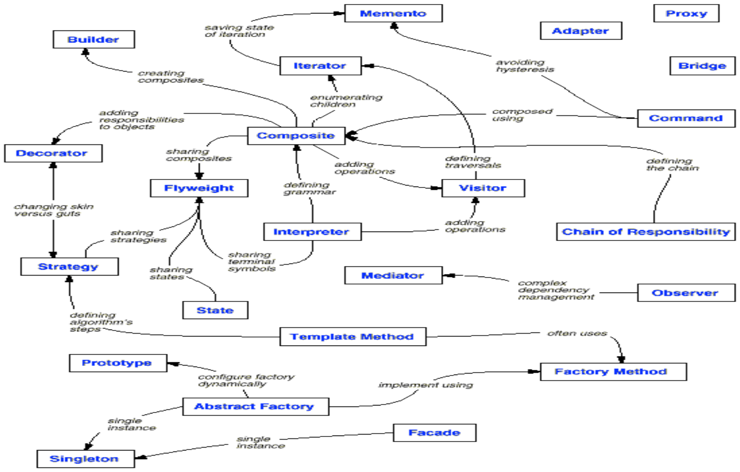
Storage Patterns

- Class
  - Object File
    - Store and retrieve a network of objects to a sequential file.
  - RDB Direct
    - Store and retrieve a network of objects to a relational database.
- Object
  - OODB Proxy
    - Store and retrieve objects from an object-oriented database.

Distributed Patterns

- Object
  - Attribute Factory
    - Generate a lightweight object graph on the client-side of a client-server system.

Relationships Between Patterns


Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
composed using
Command
adding responsibilities to objects
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
Interpreter
adding operations
Chain of Responsibility
sharing strategies
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
configure factory dynamically
implement using
Factory Method
Abstract Factory
single instance
Facade
single instance
Singleton