Software Design

- 'Requirements' defines
  - The goals the system needs to satisfy.
- 'Specification' defines
  - The externally-observable behaviour of the system.
- 'Architecture' defines
  - The major system-level components
  - Their methods of interaction
  - Technology used
- 'Design' defines
  - how the job will get done
  - The code that needs to be written.
  - We will focus exclusively on OO design.

Software Design Is:

- The Process of figuring out:
  - How it will get done
  - How the classes described in the OOA will work together in software
  - How the links and associations should be implemented.
  - How purchased or otherwise acquired components can help
  - Improving our estimates of cost and time to market
  - Assessing the prerequisites in terms of labor and infrastructure

A Good Design Process:

- Breaks into phases so progress is measurable.
- Is traceable, namely the decisions made during the design can be reasonably directly attributed to requirements.
- Cheaper than “just doing it”.
- Reduces risk over “just doing it”.
- Accommodates experimentation to explore truly unknown issues.
- Helps avoid “death marches” and wild cost overruns by supporting the setting of reasonable costs and project schedules.

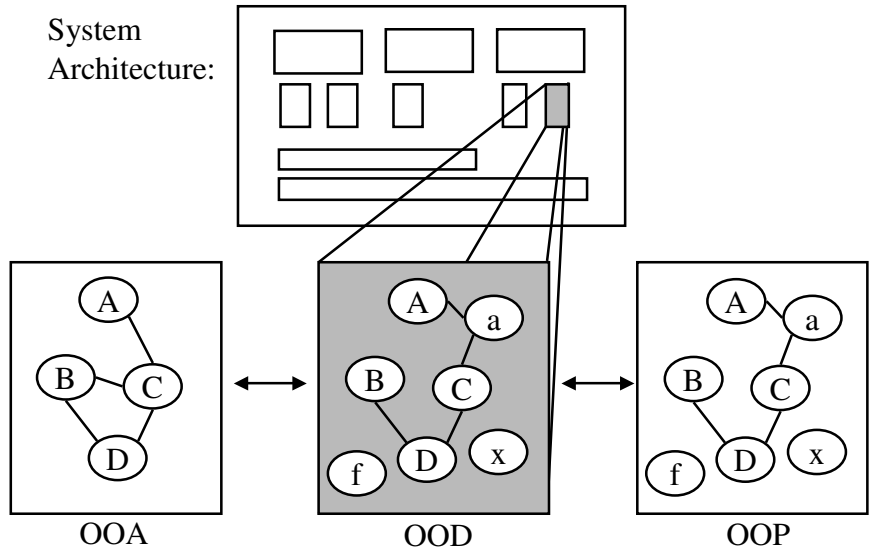
Object Oriented Design

- The process of further describing the classes we will build our system out of in terms of their operations and attributes.
- Adding classes that aren’t obviously part of the domain, like abstract classes and interfaces.
- Describing how classes make up components.

Where OOD Fits

- OOA
  - Understand the problem domain: Requirements
    - independent of any solution systems
  - Provide a basis for design
  - Use cases describe tasks the users require the system to support
- Architecture
  - Decide on technology choices
  - Decide on major sub-system breakdowns
- OOD
  - Transform OOA according to the architecture into a class-level design.
- OOP
  - Program (according to OO precepts) based on the OOD.

Where OOD Fits



Output of Design

- A document:
  - Prose description
  - UML
    - Classes
      - Associations
      - Methods
      - Attributes
    - Object diagrams
    - Sequence and collaboration diagrams
    - Statechart and activity diagrams
  - Formulae & algorithms
- Must relate to the architecture
- Can reference the requirements and specification
- Must be sufficient to allow coding to commence

Necessary Background

- Experience in OO programming
- Experience in OO analysis
- An understanding of OO concepts
  - Encapsulation (data hiding)
  - Objects, classes, meta-classes
  - Classes v.s. interfaces (types)
  - Inheritance, multiple inheritance
  - Polymorphism (run-time typing)
  - Implementation inheritance v.s. interface inheritance

Goal

- To teach you to create good OO designs.
- What you need:
  - Tools (UML notation)
  - Methods so you know what steps to go through
  - Experience
- How we will teach you:
  - UML
  - Show you what to do
    - but not how!
  - Next best thing to experience:
    - Other people's experience
    - Design Patterns

Finding Appropriate Objects

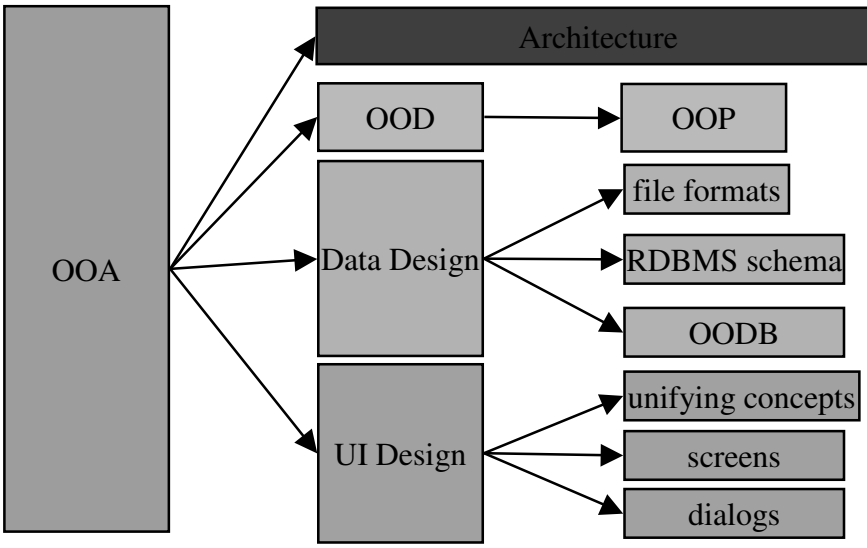
- Hard part about OOD is decomposing a system into objects.
- Many objects come directly from the
  - analysis model (you know what that is)

  or from
  - the implementation space (databases, files, UIs, IPC, …)
- As well, there are other classes that have no such counterparts.
  - used to generalize what would otherwise be an overly-specific design
  - e.g., use of Strategy
    - if you think an algorithm is likely to change
    - add classes to implement a "strategy" pattern

Why OOA first?

- OOA class diagrams
  - divide the problem space into separate and (by definition) highly cohesive pieces (classes)
  - specify the associations between the pieces
- Design
  - Each OOA class is transformed as directly as possible into a design class
  - These classes form the central component and the organizing principle for the design
  - The central classes are highly cohesive leading to good maintainability
    - ease of understanding
    - isolation of changes

OOA is a Prerequisite to many design tasks



OOD: The Degenerate Case

- In our assignment, we apply a trivial case of design
  - Unrealistic: there is 0 architecture required:
    - Program is 1 monolithic program
    - Invocation is via a simple command line
    - Output is simple sequential ASCII
    - Input is excluded from the design
    - There is only one operation to perform (plan a release)
  - Degenerate, but still not a walk in the park!
  - OOD consists of:
    - deciding how to implement associations
    - deciding how to implement attributes
    - deciding which classes should have which methods
    - adding additional classes for solution-space concepts
      - command line invocation (some kind of Main class)
      - file input interface (some kind of DataInput class)
      - output interfaces and implementation (DataOutput class)

Completing the OOA

- So far, we have taught you how to do an OOA Class diagram.
- A second important part of OOA is enumerating and elaborating the use cases.
- Moving towards OOD, but still with a foot on the OOA side, comes:

  sequence diagrams for how to implement use cases

  assigning operations to OOA classes

Pick Up From Previous Example

- We are asked to build a system for keeping track of the time our workers spend working on customer projects.
- We divide projects into activities, and the activities into tasks. A task is assigned to a worker, who could be a salaried worker or an hourly worker.
- Each task requires a certain skill, and resources have various skills at various level of expertise.

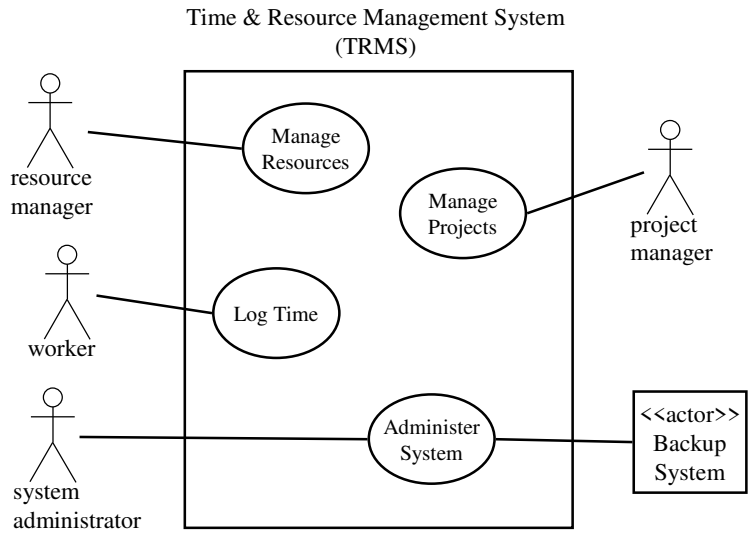
Steps

- Analyze the written requirements
  - Extract nouns: make them classes
  - Extract verbs: make them associations
  - Draw the OOA UML class diagrams
  - Draw object diagrams to clarify class diagrams
  - Determine attributes
- Determine the system's use cases
  - Identify Actors
  - Identify use case
  - Relate use cases
- Draw sequence diagrams
  - One per use case
  - Use to assign responsibilities to classes
- Add methods to OOA classes

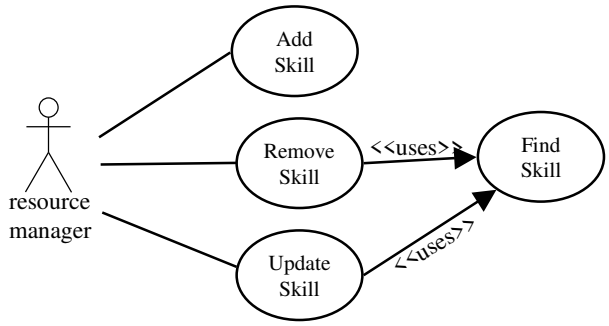
Use Cases

- Actors:
  - Represent users of a system
    - human users
    - other systems
- Use cases
  - Represent functionality or services required by users
  - Some uses cases will be assisted by the system we build.
    - Identifying system boundaries.

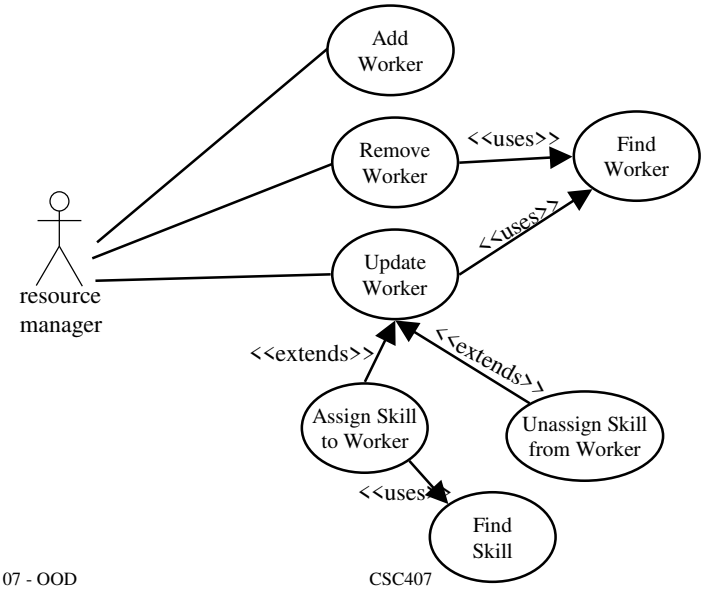
Use Case Diagrams



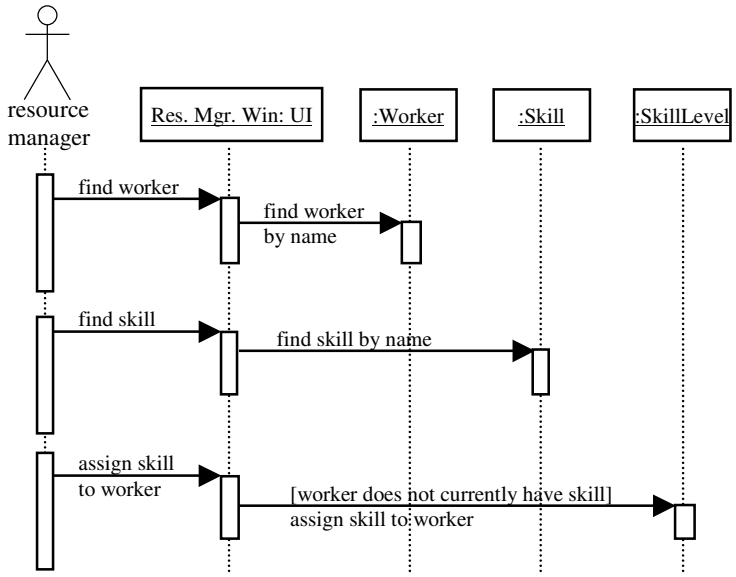
Resource Manager Use Cases



More Resource Manager Use Cases



Sequence Diagram – Assign Skill to Worker Use Case



Add Methods

- Read sequence diagrams to identify necessary methods

Worker
name: string
+ static Worker findWorker(String name); + static list of Workers getWorkers();

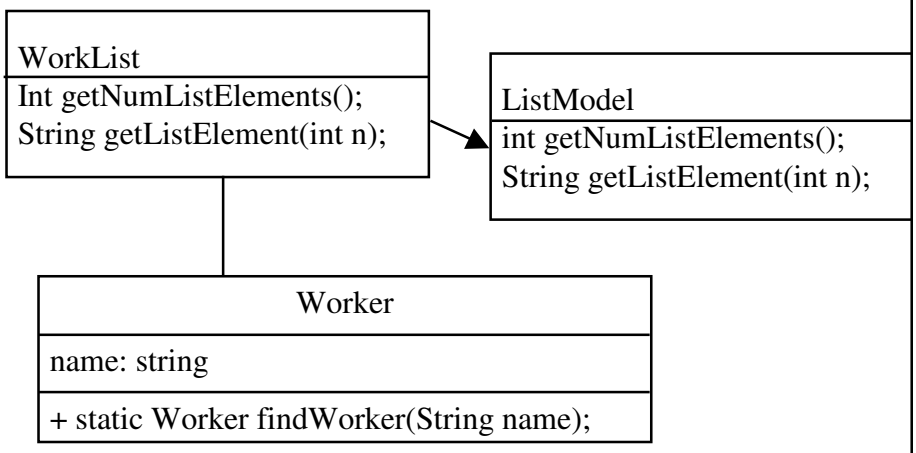
In Design

- Bring methods closer to implementation

Worker
name: string
+ static Worker findWorker(String name); + static int getNWorkers(); + static Worker getWorker(int);

In Design

- Bring methods closer to implementation



OOD: Assigning Methods

- For each system use case draw a sequence diagram
  - while doing this, one must decide what operations will be associated with which classes
- Decide how information is sent and returned
  - parameters? (yes, mostly)
  - global lookup?
  - helper classes?
- Points the way to which attributes and associations are required, and what the navigability of associations ought to be.
  - In OOA:
    - record attribute access (public/private/protected)
    - record navigability of associations

OOD: Implementing Attributes

- Decide which OOA attributes will stay in the OOD, and which are not required.
- Decide on public/private nature of attributes, and provide an interface for accessing the (conceptually) public attribute.
- Decide if attributes are stored as part of the class
  - May be more efficient to pack values into a big array somewhere and extract them using accessor methods (or leave in an input file, or OODB, or compute them on the fly in some way)
- Decide on a type for the attribute:
  - depends on programming language
  - may need to design new classes for a type
    - e.g., Date class, or TransformationMatrix class
- OOA attributes may have multiplicities
  - decide how to implement in the language
    - simple array
    - Vector type
    - other

OOD: Implementing Associations

- Decide which OOA associations will stay in the OOD, and which are not required.
- Decide on navigability (which is the more commonly accessed direction?)
- Decide on an interface for accessing associations
  - adding (?removing?) links, traversing.
  - consistency is good
  - iterators?, pass entire relationship as a class?, …
- Decide how to implement
  - Does association have an association class?
    - If so, how will data be stored?
  - pointers?, store all in some big central lookup dictionary?, …
  - 1-1 association: embedded data?
  - 1-*: array or Vector data type?
  - *-*: need to invent a new class

OOD: Implementing Operations

- Most operations will show up in an OOD as methods
  - In addition to methods required to modify/access attributes and associations.
- How will operations be implemented?
  - need for additional data members?
    - e.g., for cached values, to store the state of iterations, …
  - algorithms

Components

- The OOA is transformed into the
  Problem Domain Component
  of the solution program.
- There are many other components required as well
  - though not so many for assignment #1!
- OOA will also form the basis for the design of
  - input and output file formats
    - model classes straight into XML elements
  - persistence design
    - relational database tables
    - OODB
  - UI
    - e.g., web pages corresponding with objects

Components of the Solution

- The precise set of components is architecture dependent
  - Problem Domain Component
    - a.k.a. the Domain Object Model for the application
  - Data Management Component
    - how will data be input into the system?
    - how will modified data be saved back and under what conditions?
    - how will transactions (if required) be done?
    - does design need to be re-targetable to other data back-ends?
  - Reporting Component
    - how will report data be gathered and output?
  - Task Management Component
    - how will commands be invoked?
    - and possibly undone?
    - multi-threaded?
  - Human Interaction Component
    - how will the user interface interact with the rest of the program?
    - Re-targetable?
  - IPC (Inter-Process Communications) Component
    - how will this tier of the solution interact with other tiers?

Problem Domain Component

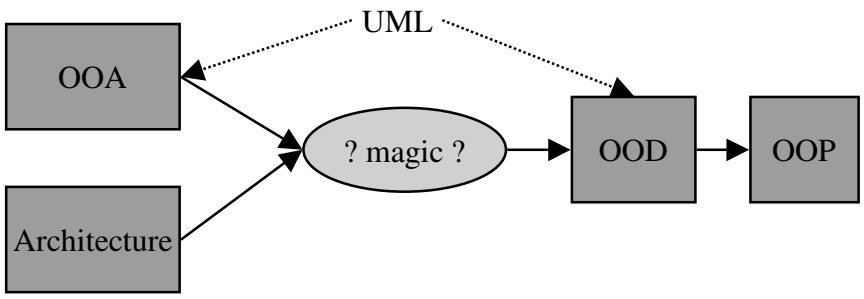
- Reuse design and programming classes.
- Group problem-domain-specific classes and establish a protocol by adding generalization classes.
- Accommodate inheritance limitations in implementation language.
- Add design classes
  - Associations
  - Run-time modifiability
  - …
- Improve performance
  - Speed, memory, perceived speed
- Support the data management component

How to do it?

- Now we know ***what*** to do in general terms:
  - Start from OOA
  - Come up with an architecture (trivial for assignment #1)
  - Elaborate use cases → sequence diagrams → add operations
  - Design problem domain component
  - Design other program components
  - Design UI, db schemas, file structures, output formats
- ***How*** do we do it?

The Bad News

- There is no step-by-step method to get from the OOA to an OOD.
  - At least the OOA gives you the problem domain component in a fairly direct manner.
  - For the rest, you need experience.



Experience

- Seasoned designers see the same old problems come up again and again:
  - how to design the classes for my 5th user interface
  - how to design the classes to support persistence to a database for the 3rd time
  - how to organize classes for reporting for the 5th time
  - …
- Each time a similar problem comes up, designers will typically start with something that has worked for them before
  - but then usually add a wrinkle inspired by something they could have done better the last time
- Technology keeps changing under our feet, and so our design experience is quickly made obsolete
  - (3-5 year half-life)

Design Patterns

- In an attempt to ensure that design experience is not
  - lost
  - obsoleted over quickly

  experienced designers have contributed *design patterns* to the world knowledge base.

- Design Patterns are the core of solutions to commonly arising problems.
- To help you to move forward on the "magic goes here" process of design, we will study a number of the basic design patterns.

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.

Using Design Patterns

- When faced with a design problem, a good designer will look for a published pattern that
  - solves that problem
  - or a closely related one
- Step 1: Understand the pattern
- Step 2: Use the pattern.
  - Either re-use it as is, adapting it to the specific situation
    - adaptation is ***always*** required
  - Use it as inspiration to come up with something that
    - either fits your problem more precisely
    - is a better solution than the published pattern
- Step 3: Write a book about it!

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.
- During the last decade, a "pattern community" has developed in the field of software design.

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.

- However,
  - you must know all these basic patterns
  - you must study easier patterns so that you understand how to read patterns, write patterns, and apply patterns

GofF Design Pattern Space

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

Scope

- Class
  - Relationships between classes and their subclasses
  - No need to execute any code to set them up
  - Static, fixed at compile-time
- Object
  - 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
  - Implement associations using template base classes

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)

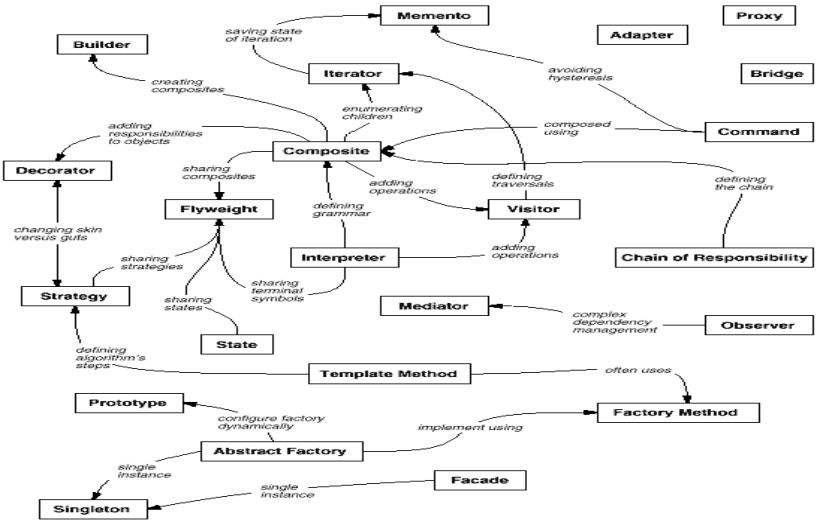
- Object (cont'd)
  - Decorator
    - Attach additional responsibilities to an object dynamically.
  - Façade
    - Provide a unified interface to a set of interfaces in a subsystem.
  - Flyweight
    - Use sharing to support large numbers of fine-grained objects efficiently.
  - Proxy
    - Provide 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.

Relationships Between Patterns