SOFTWARE ENGINEERING: DESIGN PATTERNS

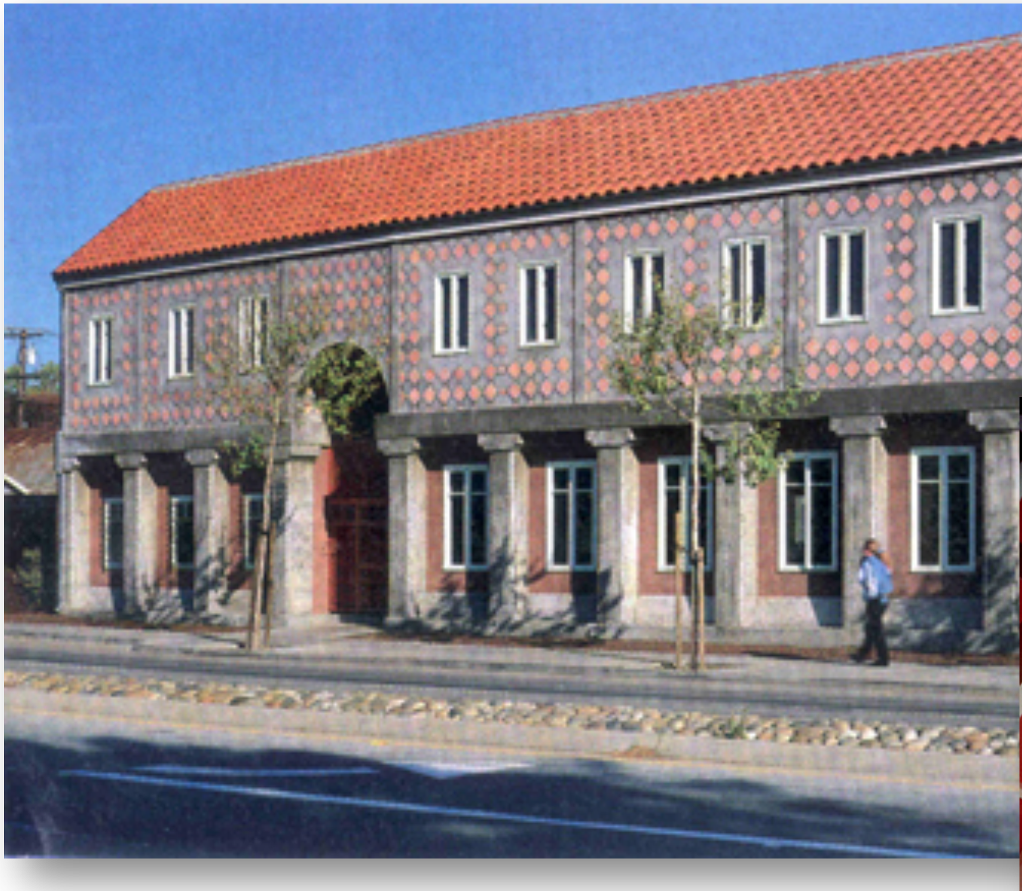
Professors M. Brian Blake and Iman Saleh

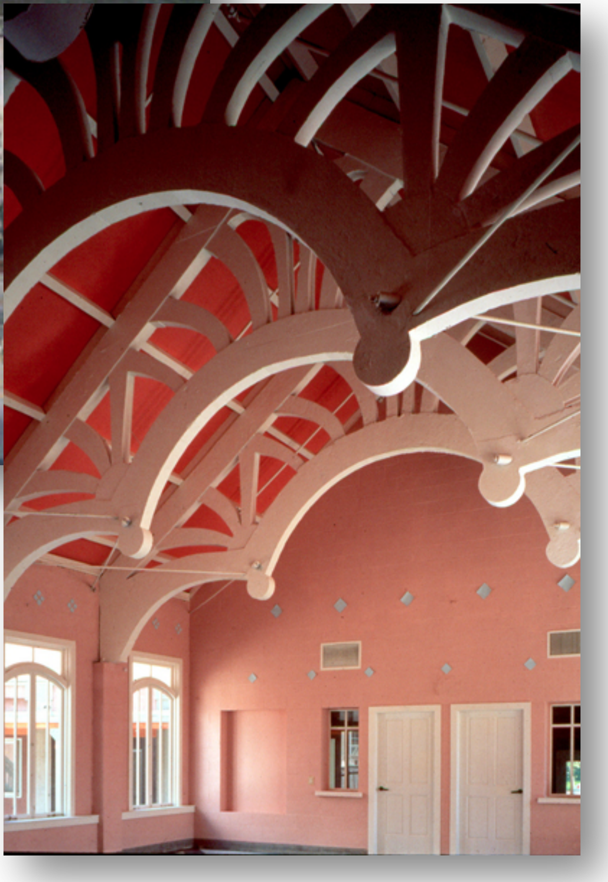
Object Design

- Object design is the process of adding details to the requirements analysis and making implementation decisions
- The object designer must choose among different ways to implement the analysis model with the goal to minimize execution time, memory and other measures of cost.
- Object Design: Iterates on the models, in particular the object model and refine the models
- Object Design serves as the basis of implementation

Object Design Activities

- Identification of existing components
- Full definition of associations
- Full definition of classes
- Specifying the contract for each component
- Choosing algorithms and data structures
- Identifying possibilities of reuse
- Optimization
- Increase of inheritance
- Decision on control
- Packaging





Julian Street Inn - San Jose, CA

Architect: Christopher Alexander

Design Patterns: History and Observations

- 1977/79 – Patterns originated as an architectural concept by Christopher Alexander

- *“Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.”*

- 1987 – A small pattern language for Smalltalk is developed based on Alexander’s idea
- 1995 – The Gang of Four (GoF) published the *Design Patterns* book [Gamma et al]

- *“Strict modeling of the real world leads to a system that reflects today’s realities but not necessarily tomorrow’s.”*

- *"Designing object-oriented software is hard and designing reusable object-oriented software is even harder.”*

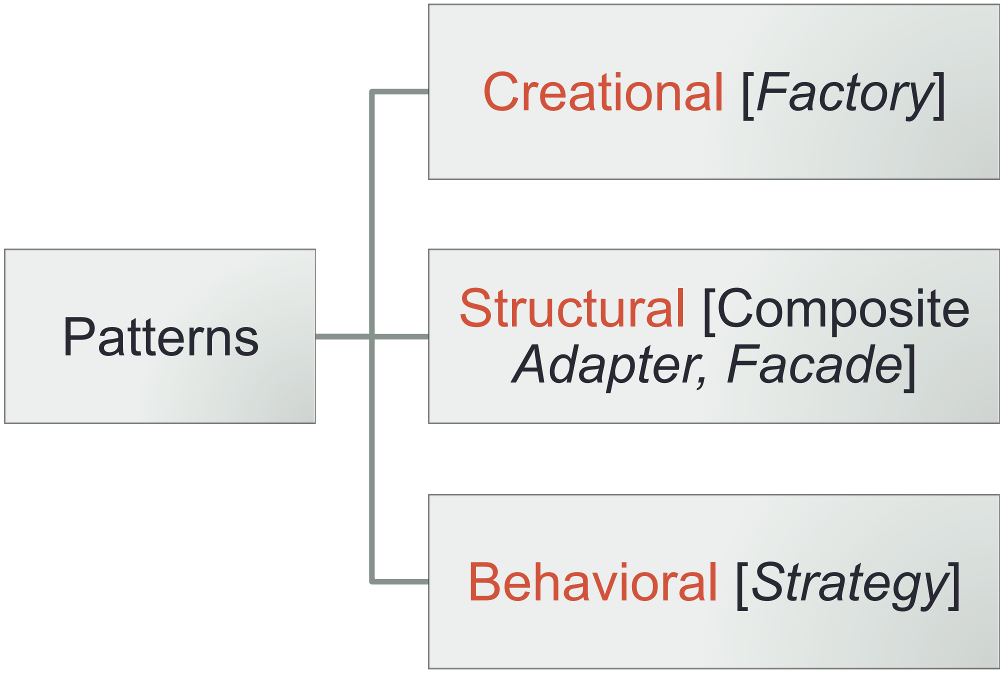
Why Patterns?

- A design pattern describes a problem which occurs over and over again in our environment. Then it describes the core of the solution to that problem, in such a way that you can use the this solution a million times over, without ever doing it the same way twice.

- A solution that worked in the past can be used by designers to save time, increase reusability and flexibility of their software, and avoid common bad practices.

GoF Classification of Design Patterns

By Purpose: What a pattern does…



Patterns

Creational [*Factory*]

Concern the process of object creation

Structural [Composite *Adapter, Facade*]

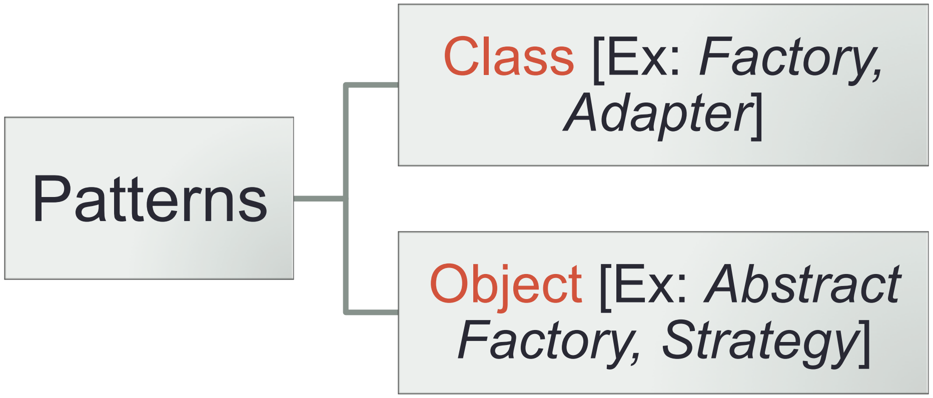
Deal with the composition of classes or objects

Behavioral [*Strategy*]

Characterize the ways in which classes or objects interact and distribute responsibility

GoF Classification of Design Patterns

By Scope: What the pattern applies to…



Patterns

Class [Ex: *Factory, Adapter*]

- Focus on the relationships between classes and their subclasses
- Static, compile-time relationships
- Involve inheritance reuse

Object [Ex: *Abstract Factory, Strategy*]

- Deal with object relationships
- Dynamic, can be changed at runtime
- Involve composition reuse

Factory

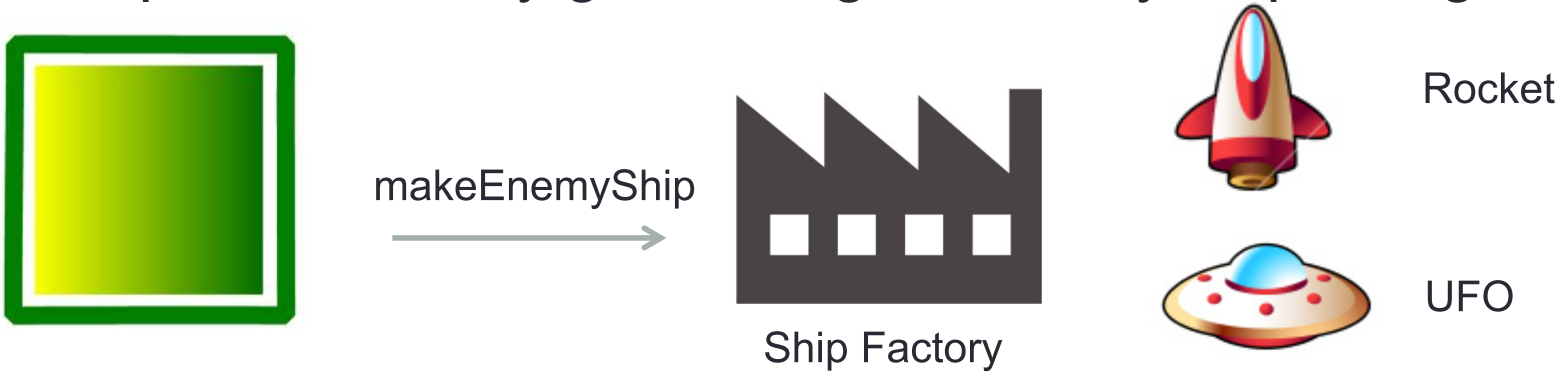
- Define an interface for creating an object, but let subclasses decide which class to instantiate.

- The class to be instantiated is decided at runtime.

- Used when you don’t know ahead of time what class object you need
- All potential classes must inherit from the same parent class
- Encapsulate object creation

Factory

- Example: Randomly generating an enemy ship in a game

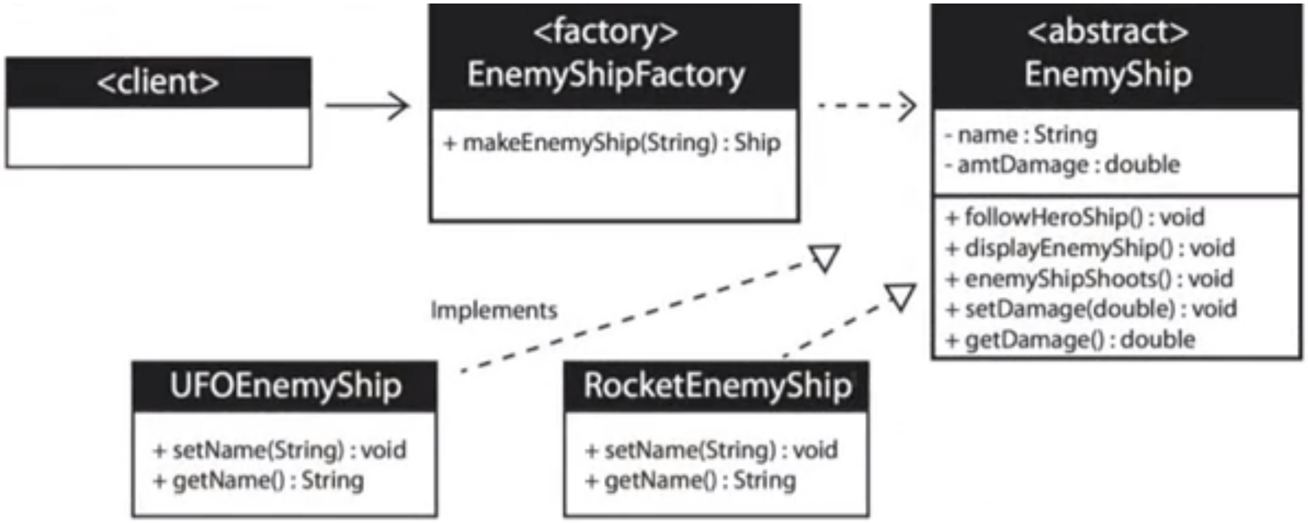


makeEnemyShip

Ship Factory

Rocket

UFO



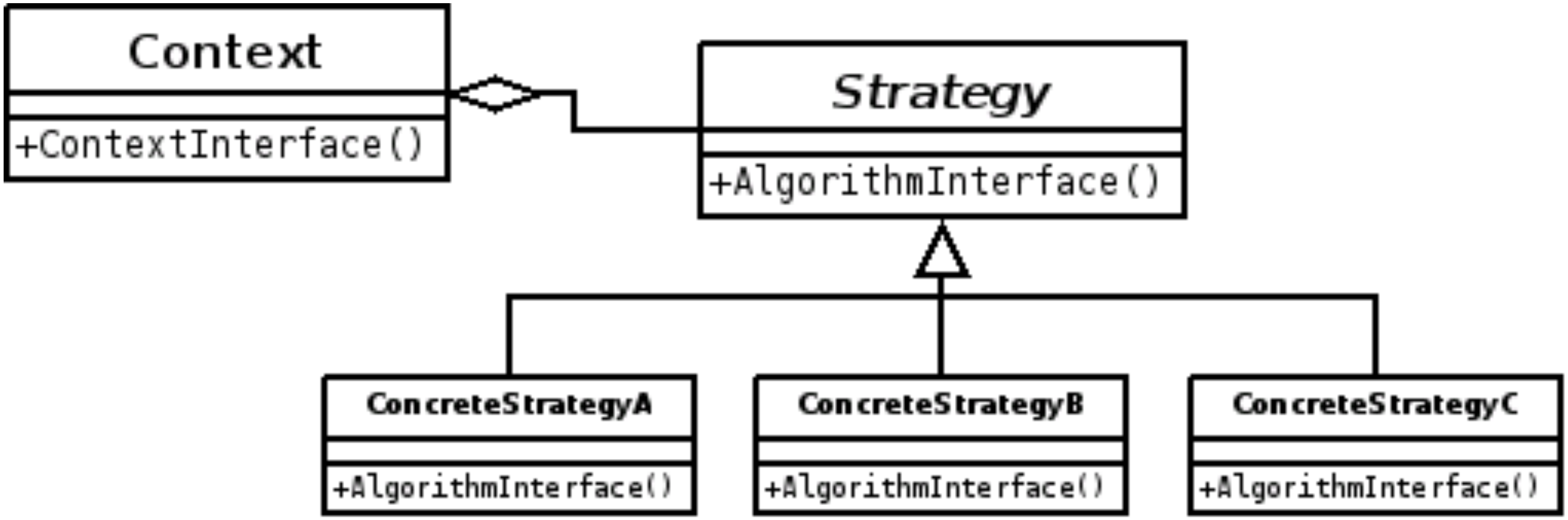
* Class diagram by Derek Banas

Strategy

- Define a family of algorithms, encapsulate each one, and make them interchangeable.

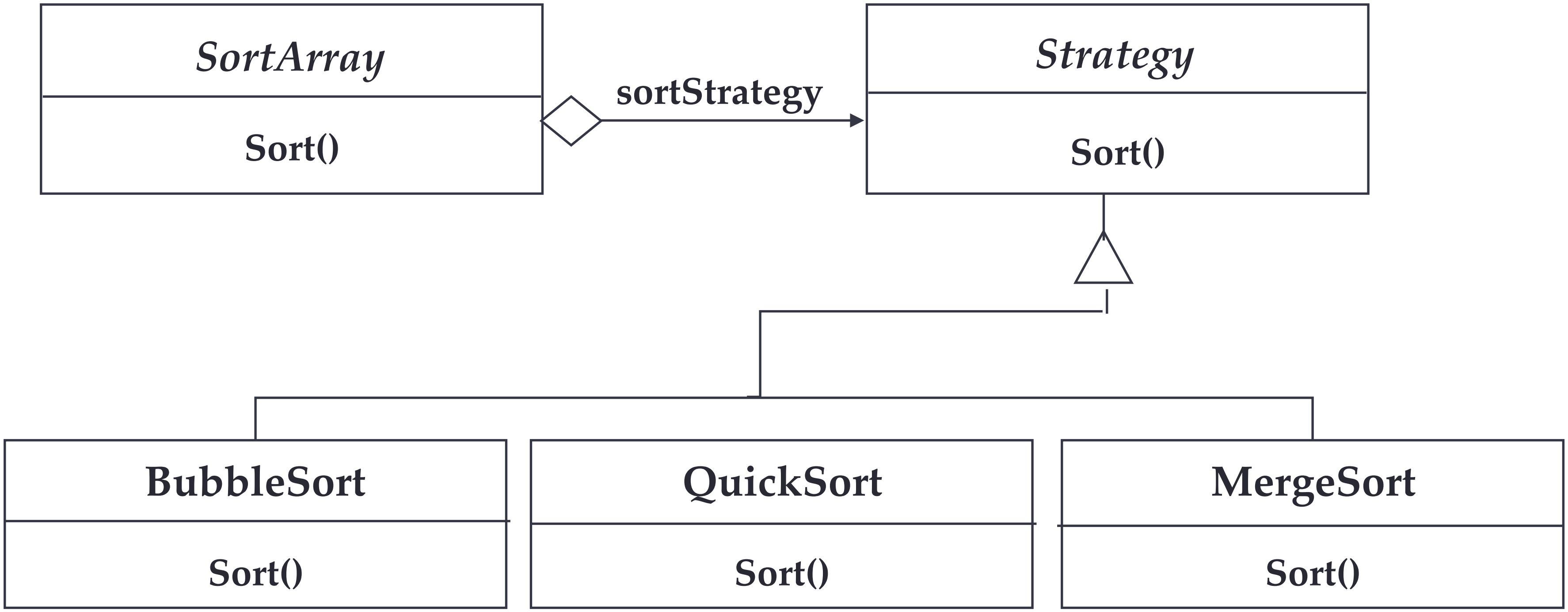
- We don’t want to support all the algorithms if we don’t need them.
- If we need a new algorithm, we want to add it easily without disturbing the application using the algorithm.

Strategy

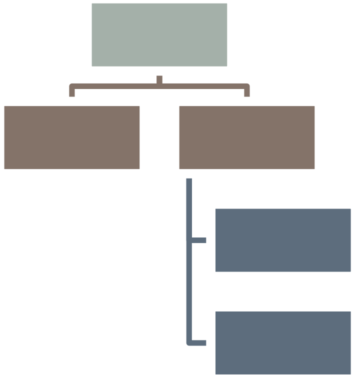

Context
+ContextInterface()
Strategy
+AlgorithmInterface()
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()

Strategy

- Example: A class wants to decide at run-time what algorithm it should use to sort an array. Many different sort algorithms are already available.

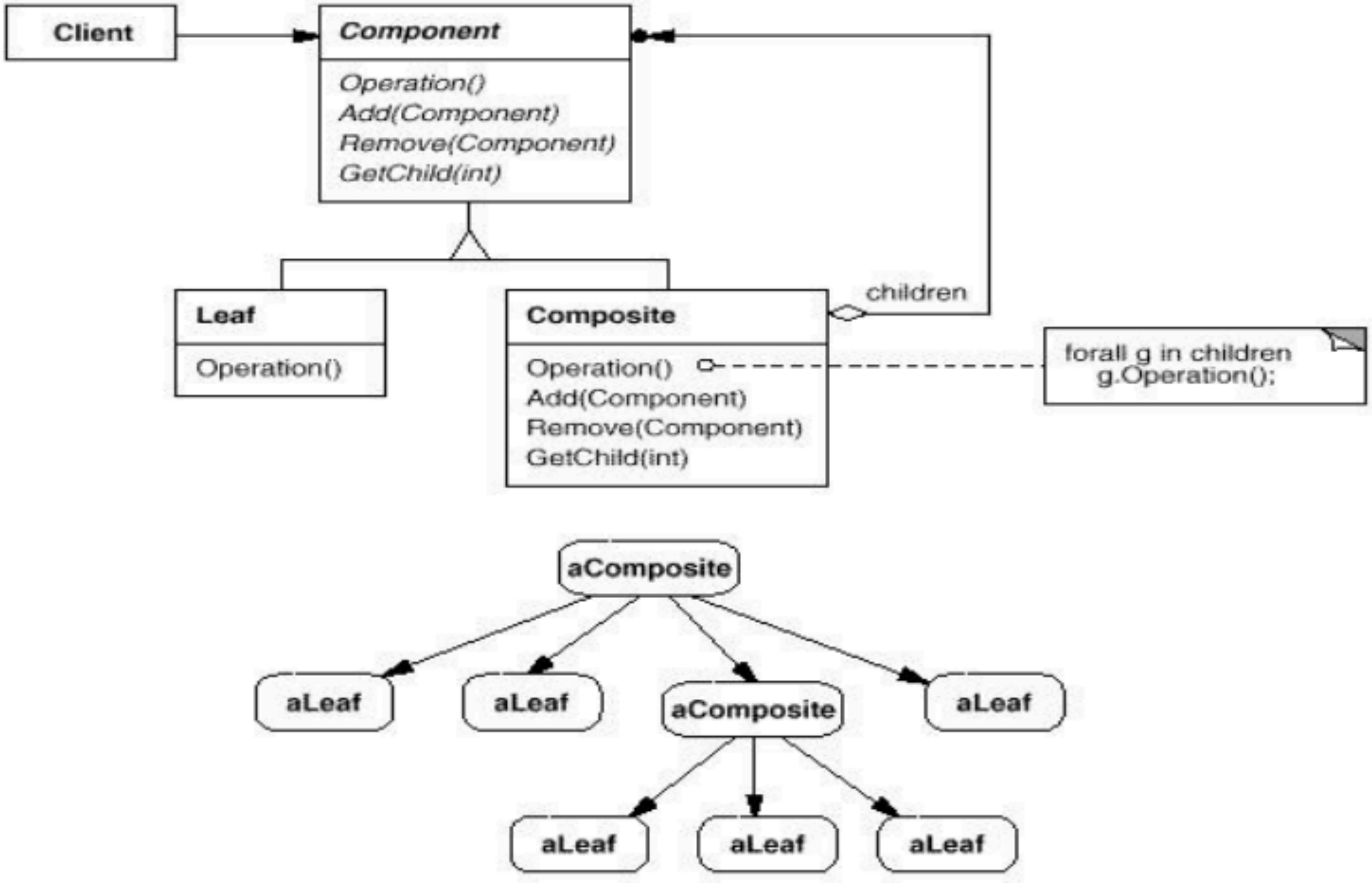


Composite



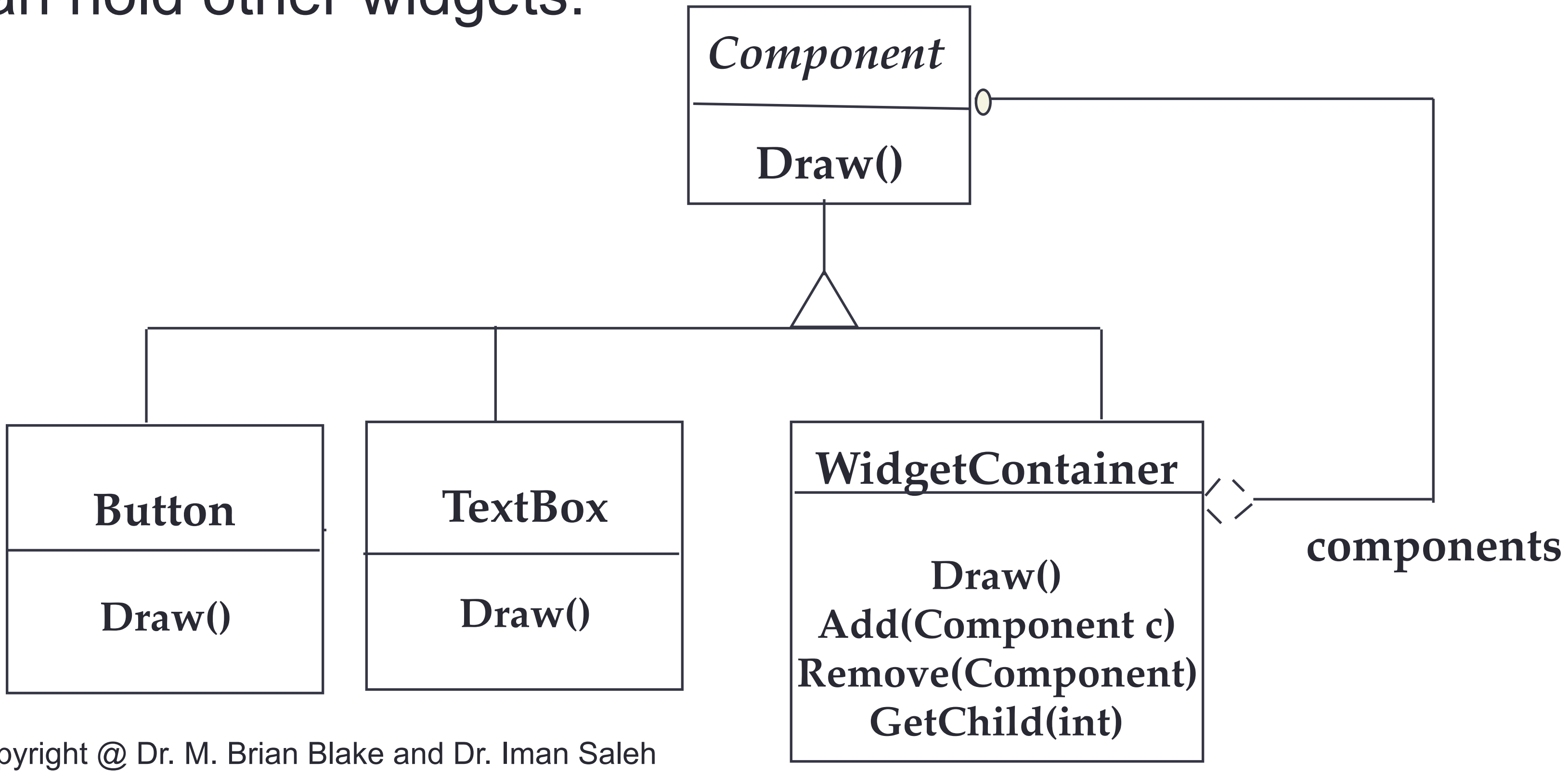
- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly. This is called *recursive composition*.
- It makes it easy to add new kinds of components
- It makes clients simpler, since they do not have to know if they are dealing with a leaf or a composite component

Composite


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();
aComposite
aLeaf
aLeaf
aComposite
aLeaf
aLeaf
aLeaf
aLeaf

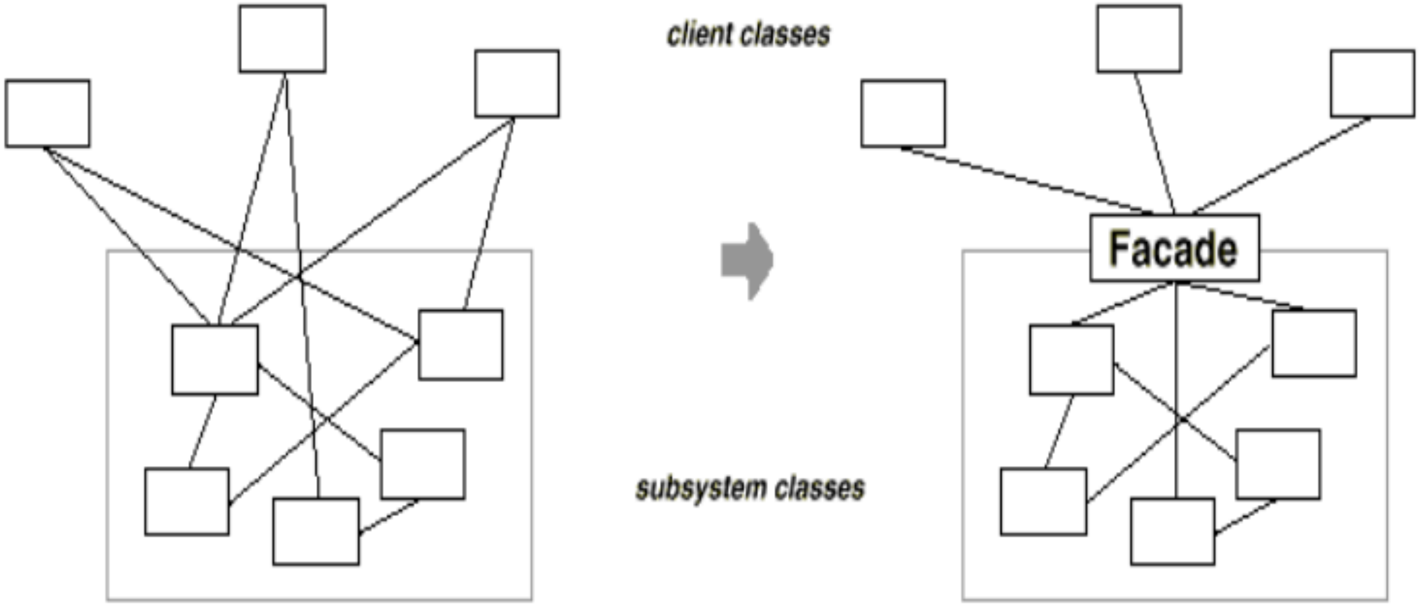
Composite

- Example: A GUI has window objects which can contain various GUI widgets such as, buttons, text boxes and menus. A window can also contain widget container objects which can hold other widgets.



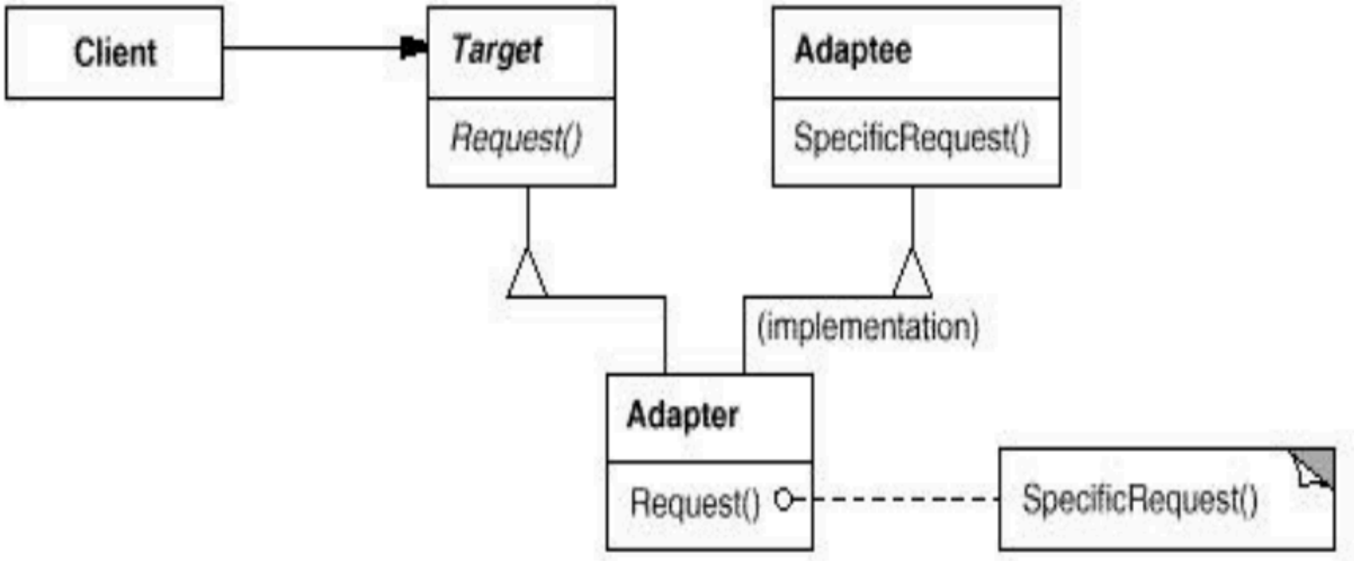
Façade

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)



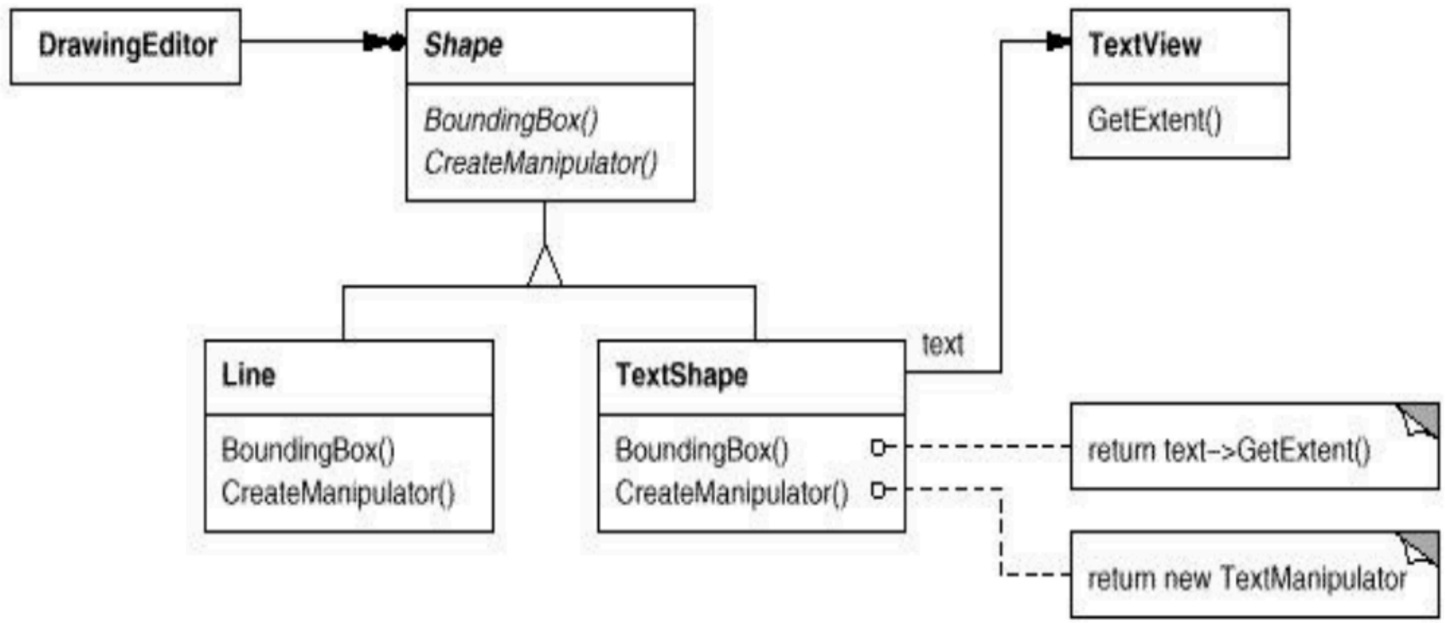
Adapter

- Allows 2 classes with incompatible interfaces to work together
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a wrapper



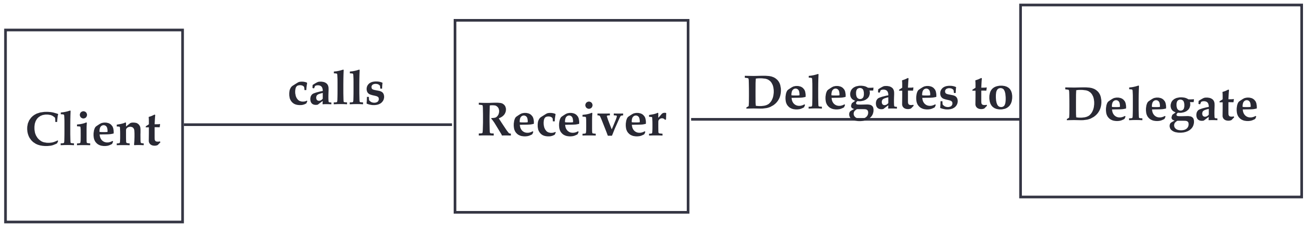
Adapter

- Delegation is used to bind an *Adapter* and an *Adaptee*
- Interface inheritance is use to specify the interface of the *Adapter* class.



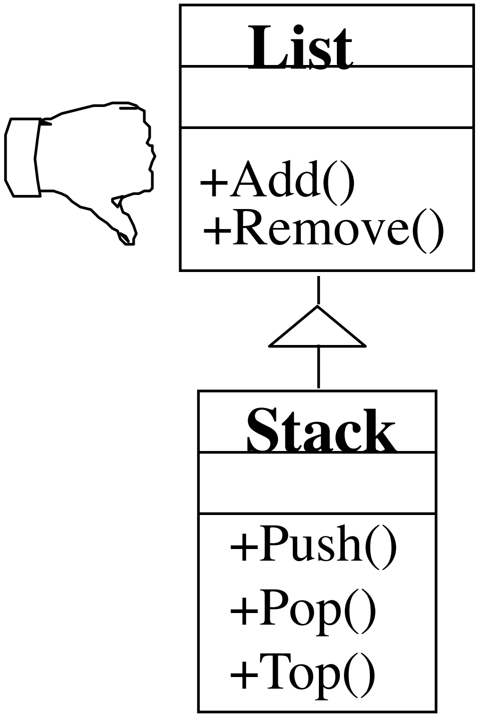
Delegation as alternative to Implementation Inheritance

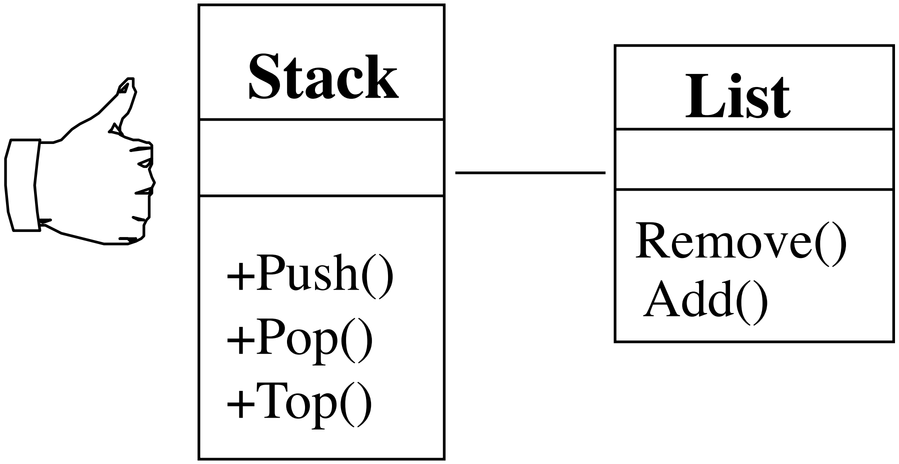
- Delegation is a way of making composition as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.
  - The developer can make sure that the receiving object does not allow the client to misuse the delegate object



Delegation instead of Implementation Inheritance

- **Inheritance**: Extending a Base class by a new operation or overwriting an operation.
- **Delegation**: Catching an operation and sending it to another object.
- Which of the following models is better for implementing a stack?





❖ Problem with implementation inheritance: Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?

Delegation vs. Implementation Inheritance

- Delegation
  - Pro:
    - Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)
  - Con:
    - Inefficiency: Objects are encapsulated.
- Inheritance
  - Pro:
    - Straightforward to use
    - Supported by many programming languages
    - Easy to implement new functionality
  - Con:
    - Inheritance exposes a subclass to the details of its parent class
    - Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)

Frameworks

- A framework is a reusable partial application that can be specialized to produce custom applications.
- Examples: Microsoft Foundation Classes (MFC), Ruby On Rails, Android Application Framework.
- Frameworks are targeted to particular technologies, such as data processing or cellular communications, or to application domains, such as user interfaces or Web applications.
- The key benefits of frameworks are reusability and extensibility.
- Software frameworks use the Hollywood Principle*: "Don't call us, we'll call you."* This means that the user-defined classes (for example, new subclasses), receive messages from the predefined framework classes.

Other Patterns

- Observer: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Command: Encapsulate a request to an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Bridge: Decouple am abstraction from its implementation so that the two can vary independently.
- Singleton: Ensure a class only has one instance, and provide a global point of access to it.
- And others…

Conclusion

- Design patterns
    - Provide solutions to common problems.
    - Lead to extensible models and code.
    - Can be used as is or as examples of interface inheritance and delegation.
    - Encourage reusable designs.
    - Apply the same principles to structure and to behavior.

Any Questions?

iman@miami.edu