Design Patterns

What is a Design Pattern

„Each pattern describes a problem which occurs
over and over again in our environment,

and then describes the core of the 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“

(Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel, “A Pattern Language: Towns/Buildings/ Construction”, Oxford University Press, New York, 1977)

What is a Design Pattern (II)

- Description of communicating objects and classes that are customized to solve a general design problem in a particular context.

  (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994 (22nd printing July 2001))
- Each pattern focuses in a particular object-oriented design problem or issue

Designing for Change – Causes for Redesign (I)

- Creating an object by specifying a class explicitly
  - Commits to a particular implementation instead of an interface
  - Can complicate future changes
  - Create objects indirectly
  - Patterns: Abstract Factory, Factory Method, Prototype
- Dependence on specific operations
  - Commits to one way of satisfying a request
  - Compile-time and runtime modifications to request handling can be simplified by avoiding hard-coded requests
  - Patterns: Chain of Responsibility, Command

Causes for Redesign (II)

- Dependence on hardware and software platform
  - External OS-APIs vary
  - Design system to limit platform dependencies
  - Patterns: Abstract Factory, Bridge
- Dependence on object representations or implementations
  - Clients that know how an object is represented, stored, located, or implemented might need to be changed when object changes
  - Hide information from clients to avoid cascading changes
  - Patterns: Abstract factory, Bridge, Memento, Proxy

Causes for Redesign (III)

- Algorithmic dependencies
  - Algorithms are often extended, optimized, and replaced during development and reuses
  - Algorithms that are likely to change should be isolated
  - Patterns: Builder, Iterator, Strategy, Template Method, Visitor
- Tight coupling
  - Leads to monolithic systems
  - Tightly coupled classes are hard to reuse in isolation
  - Patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

Causes for Redesign (IV)

- Extending functionality by subclassing
  - Requires in-depth understanding of the parent class
  - Overriding one operation might require overriding another
  - Can lead to an explosion of classes (for simple extensions)
  - Patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Inability to alter classes conveniently
  - Sources not available
  - Change might require modifying lots of existing classes
  - Patterns: Adapter, Decorator, Visitor

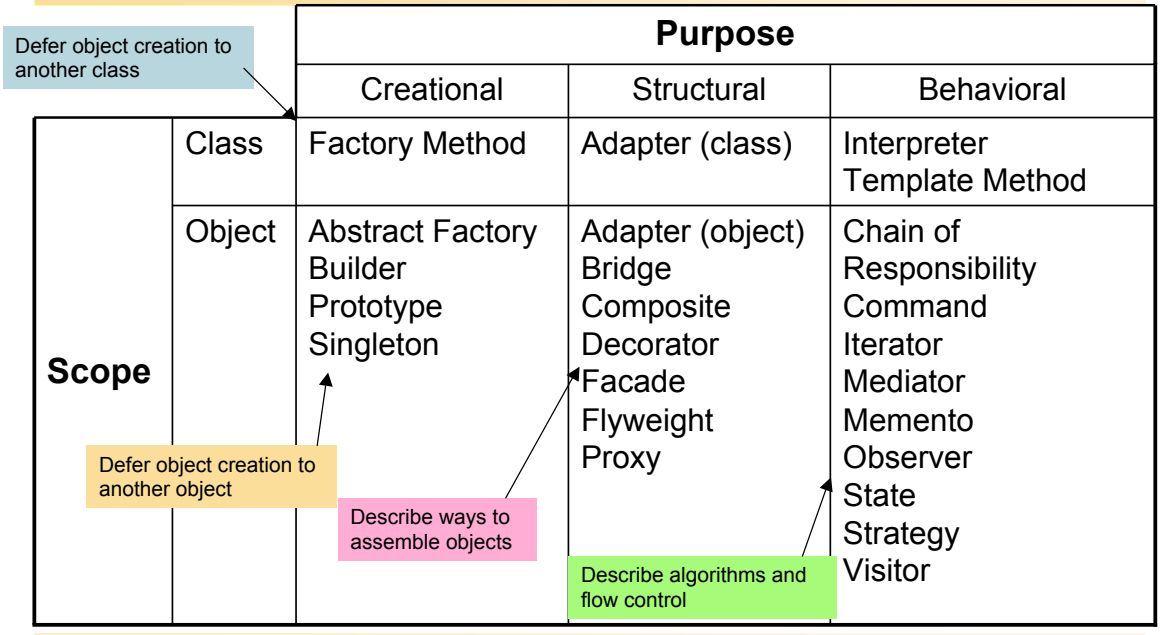
How Design Patterns Solve Design Problems

- Finding Appropriate Objects
  - *Decomposing* a system into objects is the hard part
  - OO-designs often end up with classes with no counterparts in real world (low-level classes like arrays)
  - Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrows
  - Design patterns identify *less-obvious abstractions*
- Determining Object Granularity
  - Objects can vary tremendously in size and number
  - **Facade pattern** describes how to represent subsystems as objects
  - **Flyweight pattern** describes how to support huge numbers of objects

Elements of Design Patterns

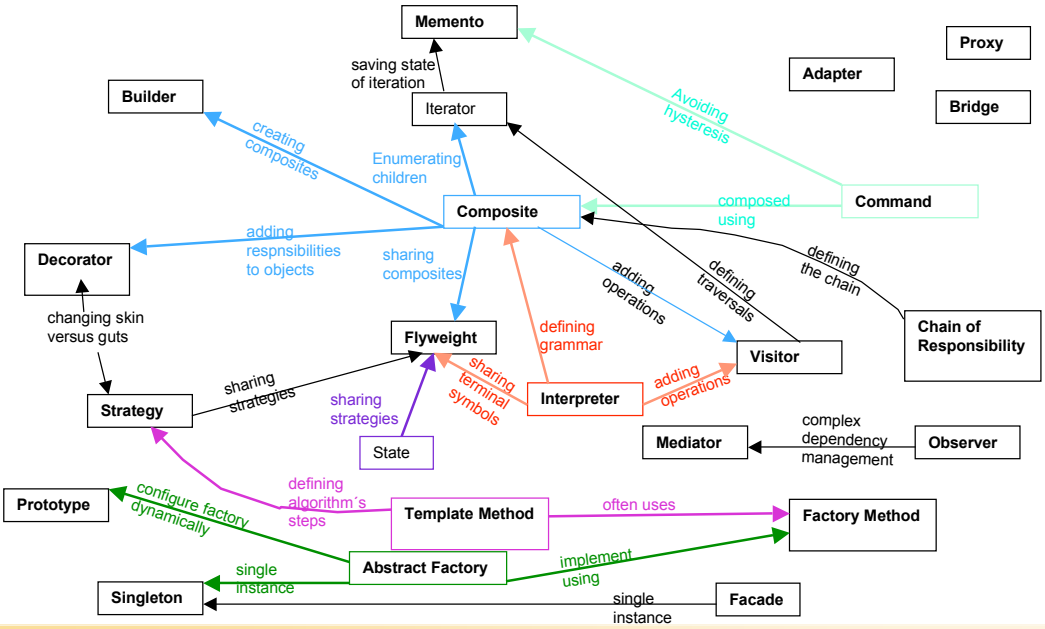
- Pattern Name
  - Increases design vocabulary, higher level of abstraction
- Problem
  - When to apply the pattern
  - Problem and context, conditions for applicability of pattern
- Solution
  - Design elements with their relationships, responsibilities, and collaborations
  - Not any concrete design or implementation, rather a template
- Consequences
  - Results and trade-offs of applying the pattern
  - Space and time trade-offs, reusability, extensibility, portability

Design Pattern Space



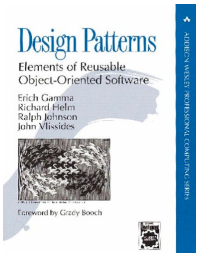
		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer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

Relations among Design Patterns



How to Select a Design Pattern

- Consider how design patterns solve design problems
  - Find appropriate objects
  - Determine object granularity
  - Specify object interfaces
- Scan intent sections
- Study how patterns interrelate
- Study patterns of like purpose
  - Creational, structural, behavioral patterns
- Examine cause of redesign
- Consider what should be variable in your design



How to Apply a Design Pattern

- Study applicability and consequences
- Study structure, participants, collaborations
- Choose names for pattern participants that are meaningful in the application context
- Define the classes
  - Declare interfaces, inheritance relationships; define instance variables
  - Identify existing classes in your app that the pattern will affect
- Define application-specific names for ops in the pattern
- Implement operations to carry out responsibilities and collaborations in the pattern

Design aspects that design patterns let you vary

Purpose	Design Pattern	Aspect(s) that can vary
Creational	Factory Method	The concrete class of the product object
	Builder	Representation of created complex object
	Abstract Factory	Families of product objects
	Prototype	Class of object that is instantiated
	Singleton	The sole instance of a class
Structural	Adapter	Interface to an object
	Bridge	Implementation of an object
	Composite	Structure and composition of an object
	Decorator	Responsibilities of an object without subclassing
	Facade	Interface to a subsystem
	Flyweight	Storage cost of objects
	Proxy	How an object is accessed, its location

Design aspects that design patterns let you vary (contd.)

Purpose	Design Pattern	Aspect(s) that can vary
Behavioral	Chain of Resp.	Object that can fulfill a request
	Command	When and how a request is fulfilled
	Interpreter	Grammar and interpretation of a language
	Iterator	How an aggregate's elements are traversed
	Mediator	How and which objects interact with each other
	Memento	What private information is stored outside an object, and when
	Observer	Number of objects that depend on another object; how the dependent objects stay up to date
	State	States of an object
	Strategy	An algorithm
	Template Method	Steps of an algorithm
	Visitor	Operations that can be applied to object(s) without changing their class(es)

Notation

- Interface:
  - Set of all signatures defined by an object's operations
  - Any request matching a signature in the objects interface may be sent to the object
  - Interfaces may contain other interfaces as subsets
- Type:
  - Denotes a particular interfaces
  - An object may have many types
  - Widely different object may share a type
  - Objects of the same type need only share parts of their interfaces
  - A **subtype** contains the interface of its **supertype**
- Dynamic binding, polymorphism

Remember: Program to an interface, not an implementation

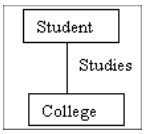
- Manipulate objects solely in terms of interfaces defined by abstract classes!

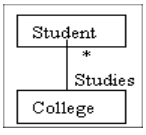
- Benefits:
  1. Clients remain unaware of the specific types of objects they use.
  2. Clients remain unaware of the classes that implement the objects. Clients only know about abstract class(es) defining the interfaces

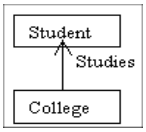
  - Do not declare variables to be instances of particular concrete classes
  - Use creational patterns to create actual objects.

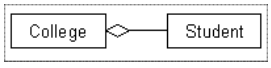
Remember: UML Class Diagram

- Association
  - Bi-directional class connection
- Multiplicity
  - "one to many", "many to many"
- Direct Association
  - Container-contained relationship
  - Special case: Reflexive Association
    - Relation to same class
- Aggregation
  - Class as collection of other classes
  - "has a" relationship



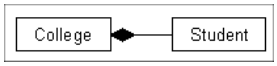




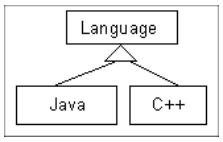


Remember: UML Class Diagram (II)

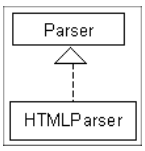
- Composition
  - Aggregation with strong lifecycle



- Inheritance / Generalization
  - “is a” relationship
  - Child class is a type of the parent class
  - Child class inherits functionality

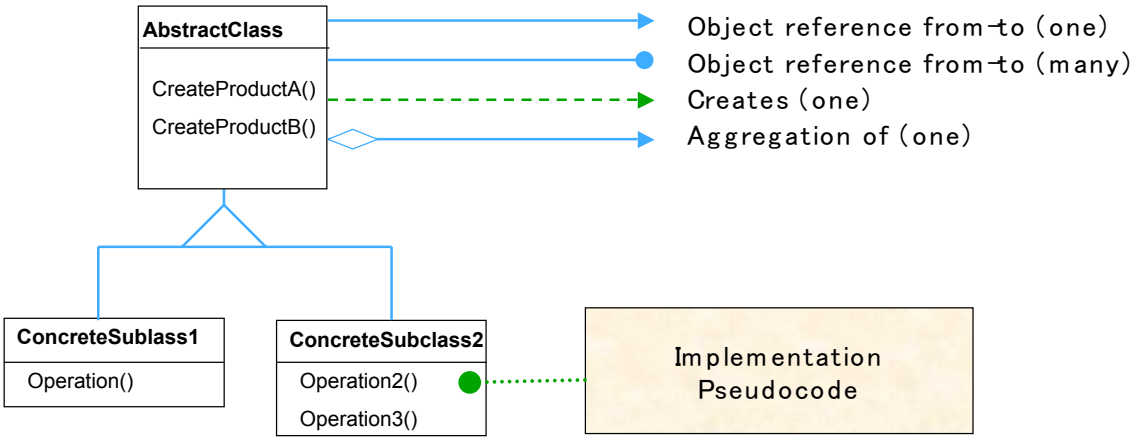


- Realization
  - One entity defines contract for functionalities
  - Other entity realizes the contract (implements to functionality)
  - Example: Interface vs. class



Object Modeling Technique

- Class diagram
  - Classes, their structure, static relationship between them



Design Pattern Space

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

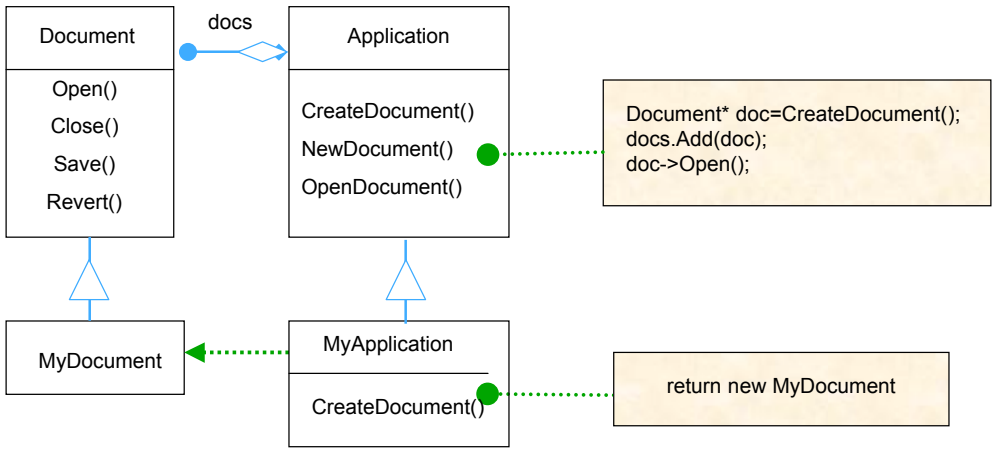
Creational Patterns

- Abstract the instantiation process
  - Make a system independent of how objects are created, composed, and represented
- Use case: Important if systems evolve to depend more on object composition than on class inheritance
  - Emphasis shifts from hard-coding fixed sets of behaviors towards a smaller set of composable fundamental behaviors
- Encapsulate knowledge about concrete classes the system uses and instantiates

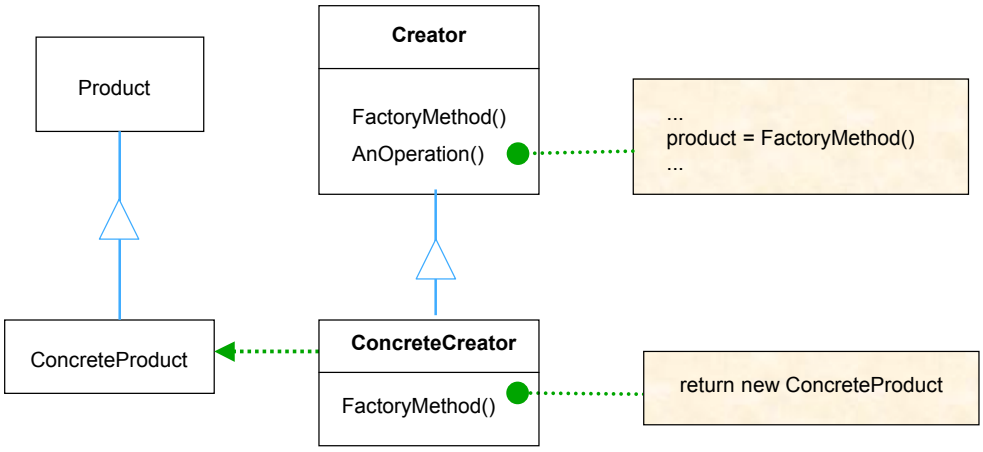
FACTORY METHOD (Class Creational)

- Intent:
  - Define an interface for creating an object, but let subclasses decide which class to instantiate
  - Factory Method lets a class defer instantiation to subclasses.
  - Also known as: Virtual Constructor
- Motivation:
  - Framework use abstract classes to define and maintain relationships between objects
  - Framework has to create objects as well - must instantiate classes but only knows about abstract classes - which it cannot instantiate
  - Factory method encapsulates knowledge of which subclass to create - moves this knowledge out of the framework

FACTORY METHOD Motivation



FACTORY METHOD
Structure



FACTORY METHOD
Participants

- Product
  - Defines the interface of objects the factory method creates
- ConcreteProduct
  - Implements the product interface
- Creator
  - Declares the factory method which returns object of type “Product”
  - May contain a default implementation of the factory method
  - Creator relies on its subclasses to define the factory method
  - Subclass returns an instance of the appropriate “ConcreteProduct”
- ConcreteCreator
  - Overrides factory method to return instance of ConcreteProduct

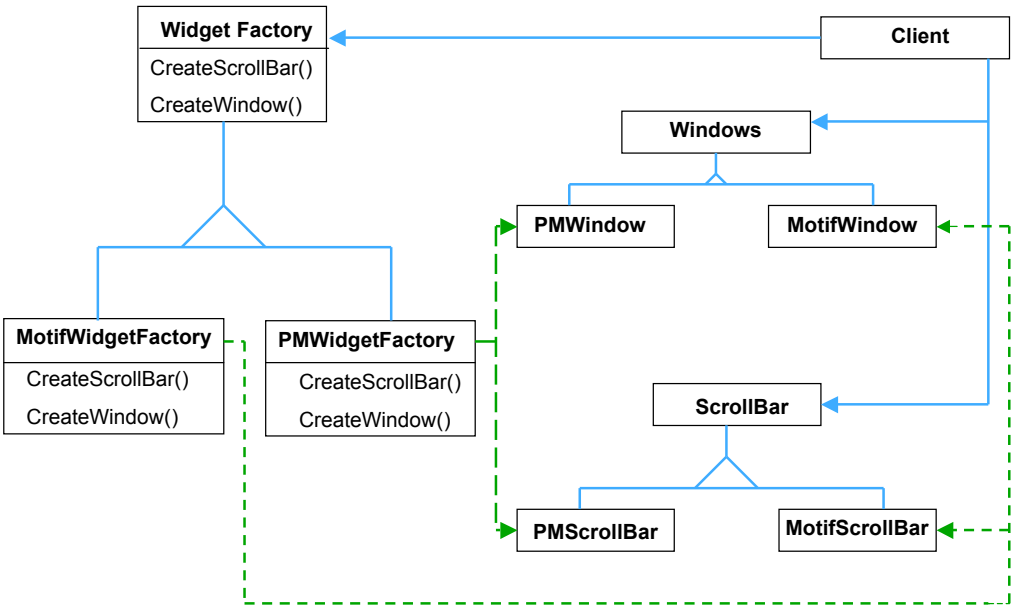
FACTORY METHOD
Applicability / Benefits

- Use the Factory Method pattern when
  - a class can't anticipate the class of objects it must create
  - a class wants its *subclasses to specify the objects it creates*
- Benefits
  - Eliminate need to bind application-specific classes to your code
  - Factory method gives according subclass a hook
  - Concrete factory method might be used directly by the client

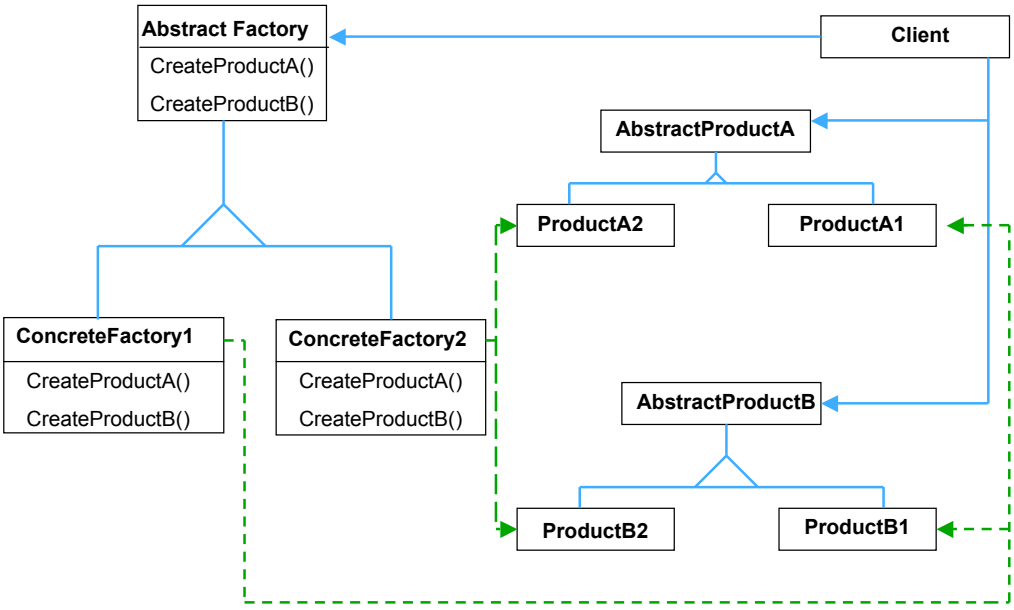
ABSTRACT FACTORY
(Object Creational)

- Intent:
  - Provide an interface for creating families of related or dependent objects
  - No need to specify their concrete classes
  - Also known as: Kit
- Motivation:
  - User interface toolkit supports multiple look-and-feel standards (Motif)
  - Different appearances and behaviors for UI widgets (no hard-coding)
- Solution:
  - Abstract classes for WidgetFactory and each kind of widget
  - WidgetFactory with interface to create each kind of widget
  - Concrete factory subclass for each look-and-feel standard
  - Concrete widget subclasses implement specific look-and-feel

ABSTRACT FACTORY
Motivation



ABSTRACT FACTORY
Structure



ABSTRACT FACTORY Participants

- AbstractFactory
  - Declares interface for operations that create abstract product objects
- ConcreteFactory
  - Implements operations to create concrete product objects
- AbstractProduct
  - Declares an interface for a type of product object
- ConcreteProduct
  - Defines a product object to be created by concrete factory
  - Implements the abstract product interface
- Client
  - Uses only interfaces declared by *AbstractFactory* and *AbstractProduct* classes

ABSTRACT FACTORY Applicability

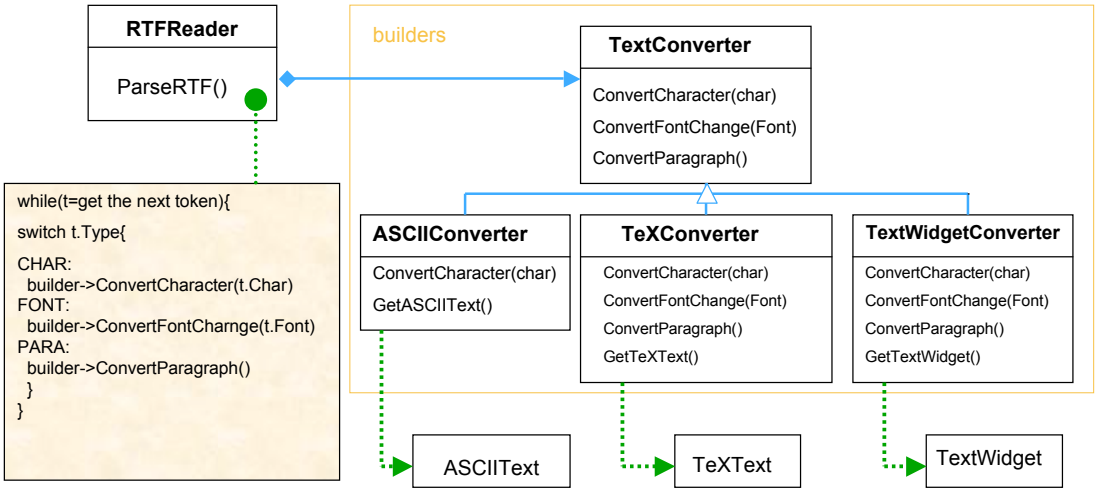
Use the Abstract Factory pattern when

- A system should be independent of how its products are created, composed, and represented
- A system should work with *one of multiple families* of produces
- A family of related product objects is designed to be used together, and you need to *enforce* this constraint
- You want to provide a class library of products, and you want to *reveal just their interfaces*, not their implementations

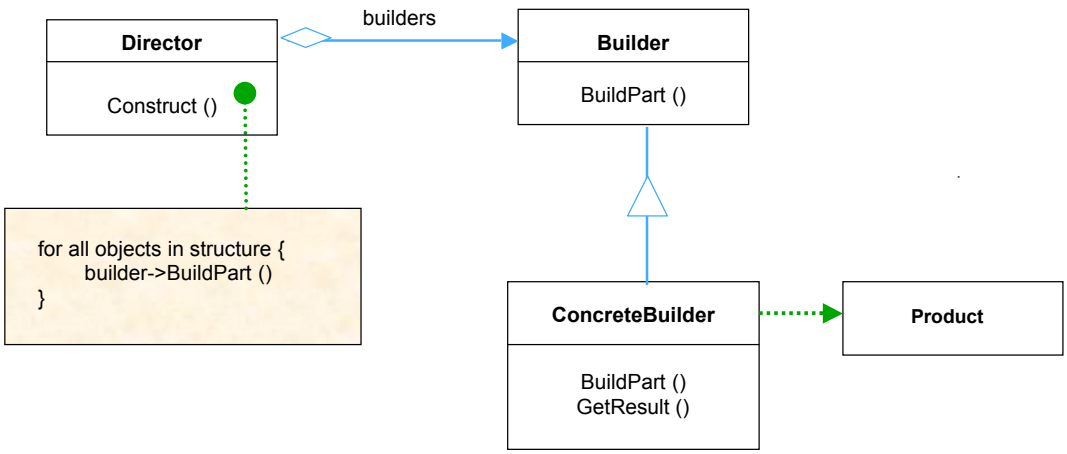
BUILDER
(Object Creational)

- Intent:
  - Separate the construction of a complex object from its representation
  - Same construction process shall create different representations
- Motivation:
  - RTF reader should be able to represent RTF in different formats
  - Adding new conversions without modifying the reader should be easy
- Solution:
  - Configure RTFReader class with a TextConverter object
  - Subclasses of TextConverter specialize in different conversions
  - Some subclasses might ignore partial conversion requests (e.g. formatting instructions)

BUILDER
Motivation



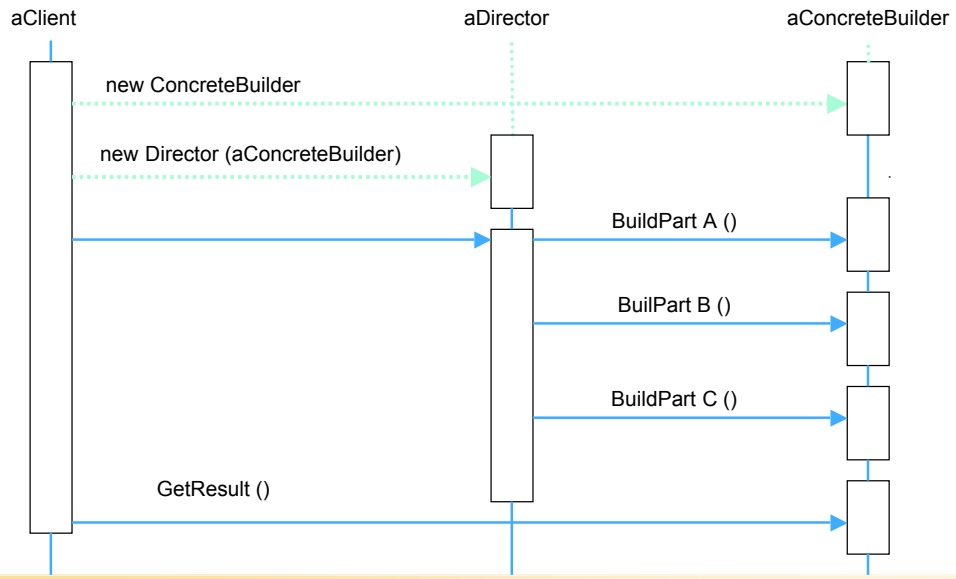
BUILDER
Structure



BUILDER
Collaborations

- *Client* creates *Director* object and configures it with the desired *Builder* object
- *Director* notifies *Builder* whenever a part of the product should be built
- *Builder* handles requests from the *Director* and adds parts to the product
- *Client* retrieves the product from the *Builder (!)*

BUILDER
Collaborations



BUILDER
Applicability / Benefits

Use the Builder pattern when

- The *algorithm for creating* a complex object should be independent of the parts that make up the object and how they are assembled
- The construction process must allow different representations for the object that is constructed

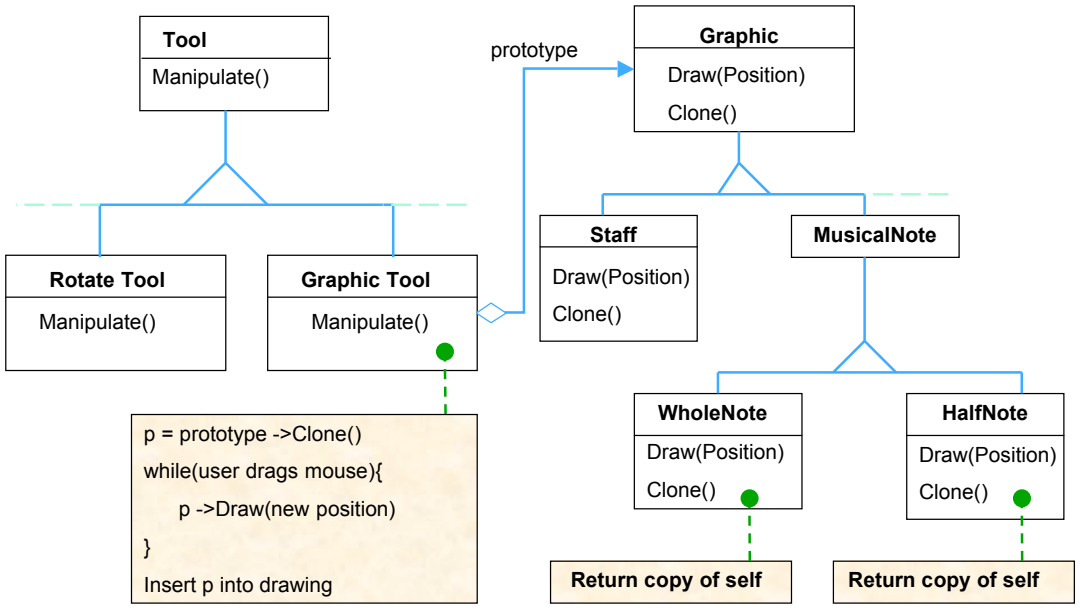
Benefits:

- Vary a product's internal representation
- Isolate code for construction from representation
- Fine-grained construction process leads to more control

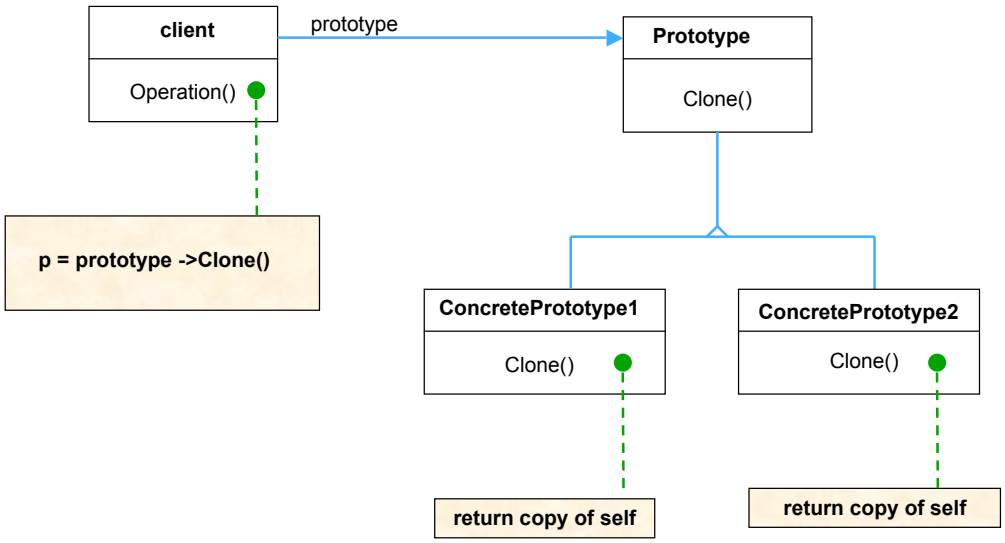
PROTOTYPE
(Object Creational)

- Intent:
  - Specify the kinds of objects to create using a prototypical instance
  - Create new objects by copying this prototype
- Motivation:
  - Framework provides abstract *Graphic* class for graphical components
  - Concrete *GraphicTool* class for manipulating/creating instances of these *Graphic* components
  - Actual graphical components are application-specific
  - How to parameterize instances of *GraphicTool* class with type (== class) of objects to create ?
  - Solution: create new objects in *GraphicTool* by cloning a **prototype** object instance

PROTOTYPE
Motivation



PROTOTYPE
Structure



PROTOTYPE
Participants

- Prototype (Graphic)
  - Declares an interface for cloning itself
- ConcretePrototype (Staff, WholeNote, HalfNote)
  - Implements an interface for cloning itself
- Client (GraphicTool)
  - Creates a new object by asking a *Prototype* to clone itself

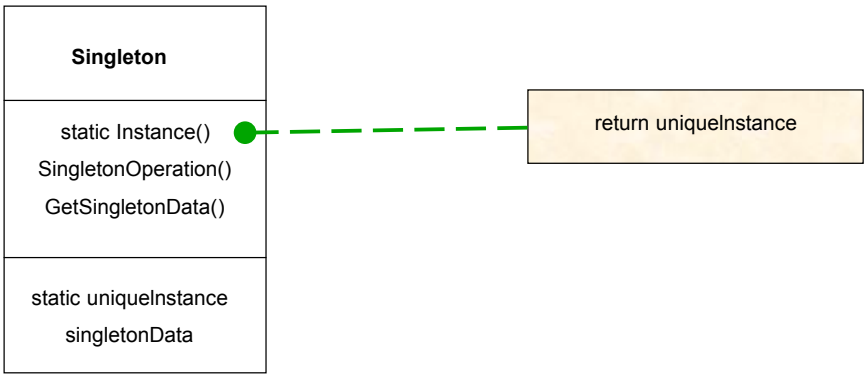
PROTOTYPE
Applicability

- Use the Prototype pattern when
  - a system should be independent of how its *products are created, composed, and represented*
  - when the *classes to instantiate are specified at run-time*, for example, by dynamic loading; **or**
  - to *avoid building a class hierarchy of factories* that parallels the class hierarchy of products; **or**
  - when instances of a class can have one of *only a few different combinations of state*
    - Install a corresponding number of prototypes
    - Clone them
    - Better than instantiating the class manually, each time with the appropriate state.

SINGLETON
(Object Creational)

- Intent:
  - Ensure a class only has one instance
  - Provide a global point of access to it
- Motivation:
  - Some classes should have exactly one instance (one print spooler, one file system, one window manager)
  - Global variable makes an object accessible but doesn't prohibit instantiation of multiple objects
  - Class should be responsible for keeping track of its sole interface
    - Intercept requests to create new objects
    - Provide way to access the interface

SINGLETON
Structure



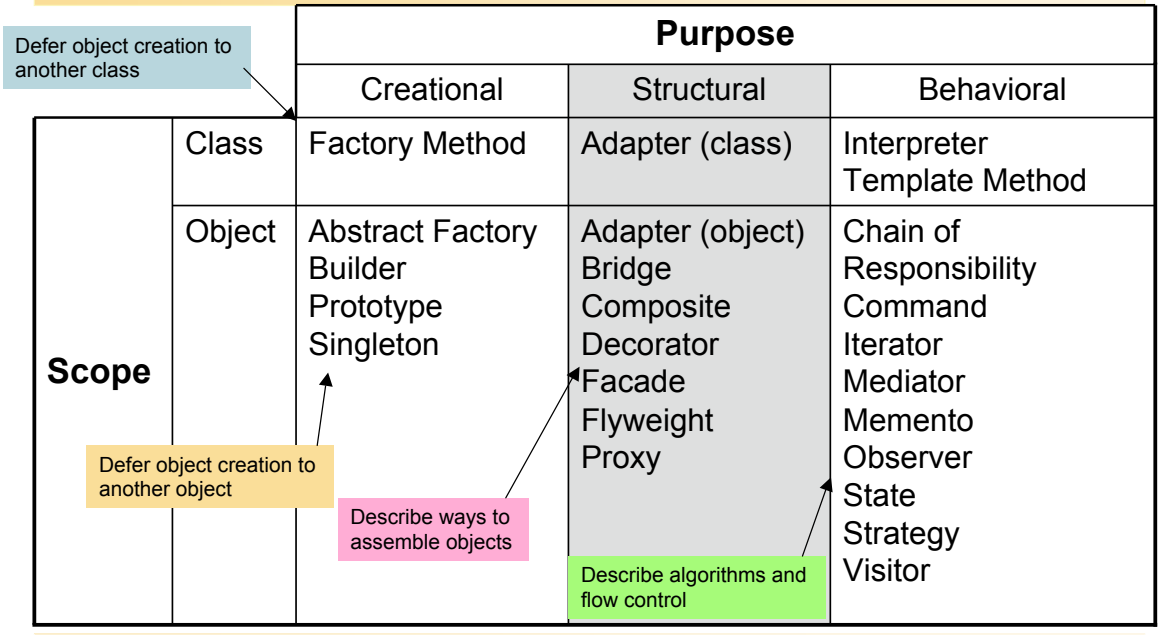
SINGLETON
Example (C++)

```
Class Singleton {
  public:
    static Singleton* Instance();
  protected:
    Singleton();
  private:
    static Singleton* _instance;
}
```

SINGLETON
Applicability / Benefits

- Use the Singleton pattern when
  - there must be exactly one instance of a class
  - the instance must be accessible from a well-known access point
  - sole instance should be extensible by subclassing
  - clients should be able to use an extended instance without modifying their code
- Benefits
  - Controlled access to sole interface
  - Reduced global variable name space
  - Refinement of operations and representations (run-time choice)
  - Variable number of instances
  - Better than class (C++ static member) operations
    - Hard to have more than one instance, no polymorphism

Design Pattern Space



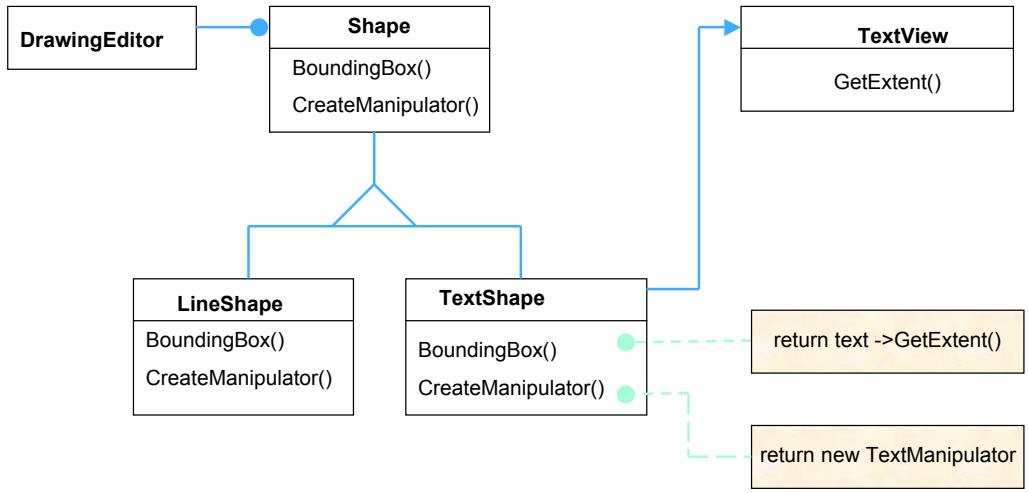
Structural Patterns

- Describe how classes and objects are composed to form larger structures
- Structural **class** patterns
  - Use inheritance to compose interfaces or implementations
- Structural **object** patterns
  - Describe ways to compose objects to realize new functionality
  - Added flexibility through ability to change composition at runtime
- Most structural patterns are related to some degree

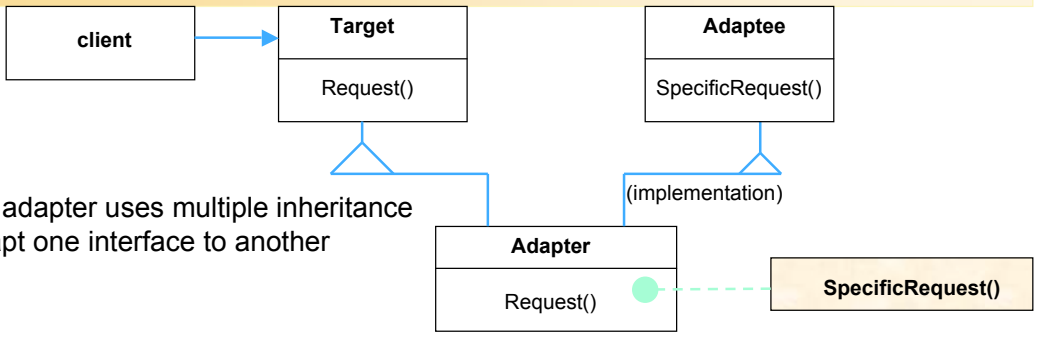
ADAPTER
(Class, Object Structural)

- Intent:
  - Convert the interface of a class into another interface clients expect
  - Adapter lets classes work together that could not otherwise
    - Incompatible interfaces
- Motivation:
  - Toolkit class that‘s designed for reuse is not reusable
    - Interface does not match the domain-specific interface
  - Example: Drawing editor
    - Shape as abstraction for graphical objects
    - Combine existing shape classes (LineShape) with unrelated new class (TextView) from another source
    - Modification is no option

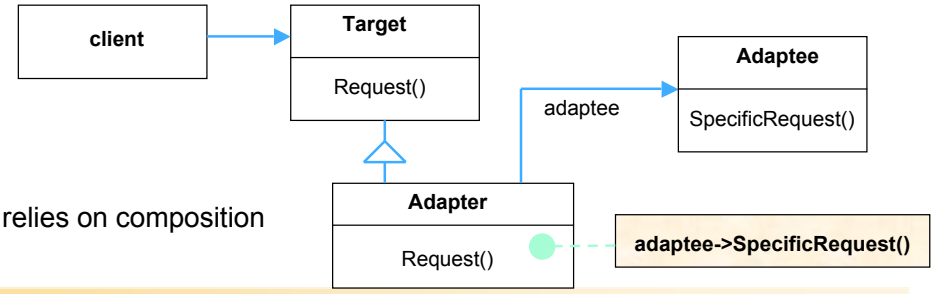
ADAPTER
Motivation



ADAPTER
Structure



Class adapter uses multiple inheritance to adapt one interface to another



Object adapter relies on composition

ADAPTER
Participants

- Target (Shape)
  - Defines the domain-specific interface that client uses
- Client (DrawingEditor)
  - Collaborates with objects conforming to the Target interface
- Adaptee (TextView)
  - Defines existing interface that needs adapting
- Adapter (TextShape)
  - Adapts the interface of Adaptee to the Target interface

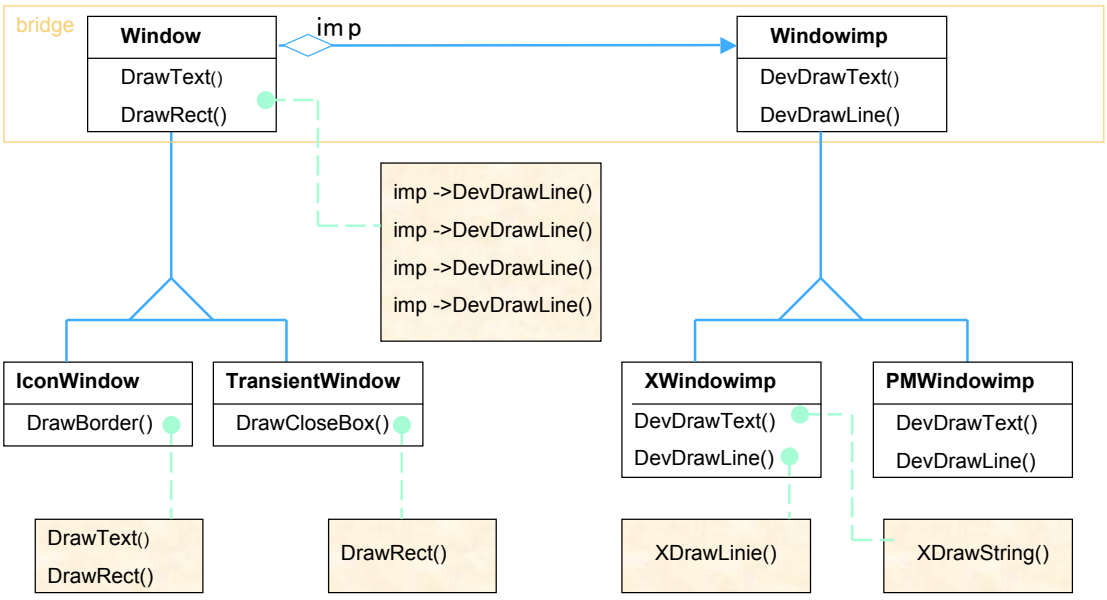
ADAPTER
Applicability / Benefits

- Use the Adapter pattern when
  - you want to use an existing class with non-matching interface
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes
  - you need to use several existing subclasses, but it's impractical to adapt their interfaces
    - object adapter can adapt the interface of the parent class
- Benefits
  - Class adapter
    - Introduces only one object (no additional indirection)
    - Lets Adapter override some of Adaptee's behavior
  - Object adapter
    - Lets a single Adapter work with many Adaptees

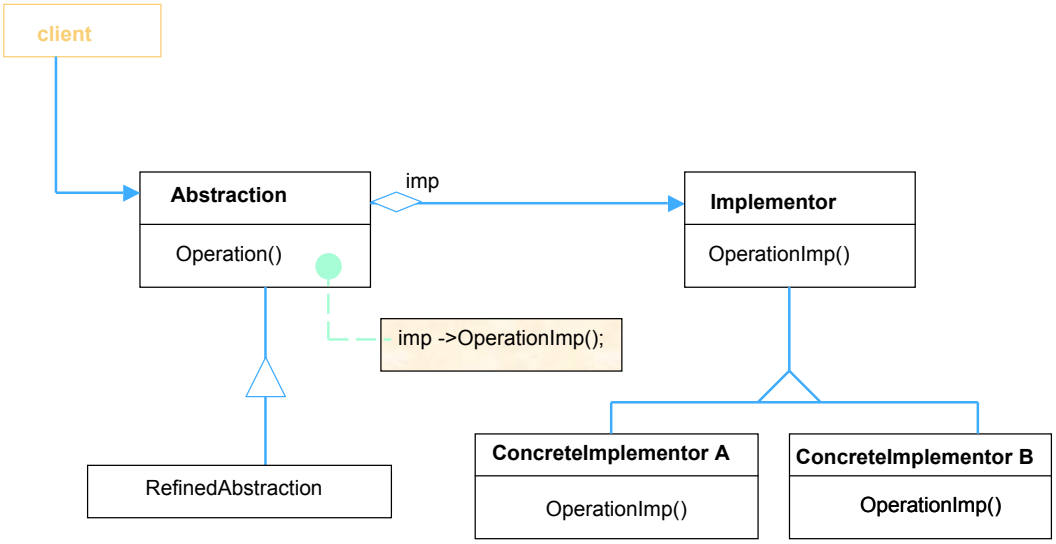
BRIDGE
(Object Structural)

- Intent:
  - Decouple an abstraction from its implementation
  - Allows that both can vary independently
- Motivation:
  - Inheritance helps when an abstraction can have multiple possible implementations - sometimes not flexible enough
  - Put abstraction and its implementation in separate class hierarchies
  - Example:
    - One class hierarchy for Window interfaces (Window, IconWindow, TransientWindow)
    - Separate hierarchy for platform-specific implementations
    - Decoupling through abstract implementation root class

BRIDGE
Motivation



BRIDGE
Structure



BRIDGE
Participants

- Abstraction (Window)
  - Defines the abstraction's interface
  - Maintains a reference to an object of type *Implementor*
- RefinedAbstraction (IconWindow)
  - Extends the interface defined by Abstraction
- Implementor (WindowImp)
  - Defines interface for implementation class
  - Not necessarily identical to *Abstraction's* interface
  - Typically provides primitive operations
    - *Abstraction* is responsible for higher-level operations
- ConcreteImplementor (XWindowImp, PMWindowImp)
  - Implements the *Implementor* interface, defines concrete implementation

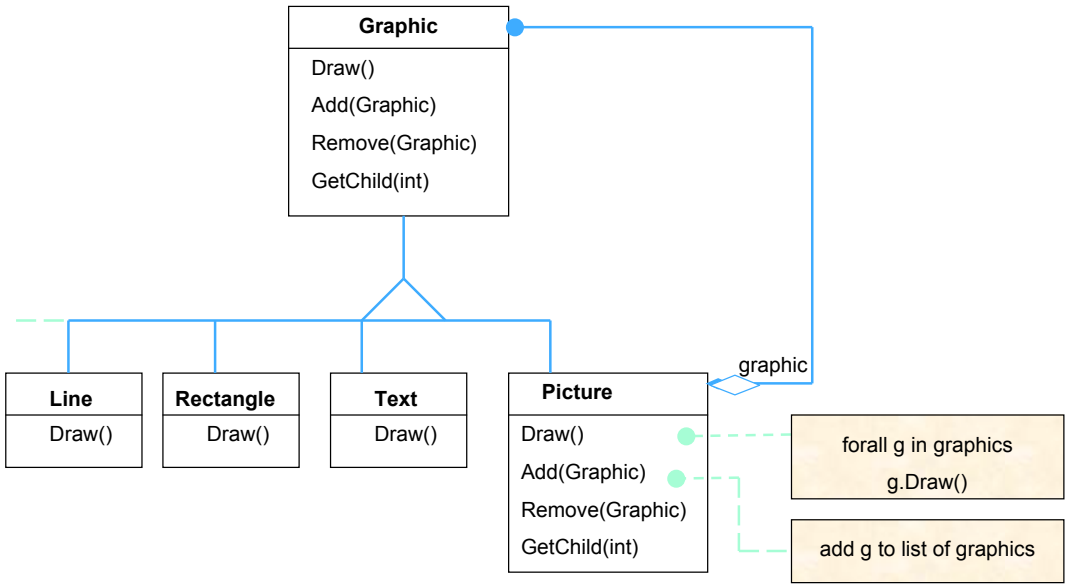
BRIDGE
Applicability

- Use the Bridge pattern when:
  - you want to avoid a permanent binding between an abstraction and its implementation
    - implementation might be selected or switched at run-time
  - both the abstractions and their implementations should be extensible by subclassing
  - you want to hide the implementation of an abstraction completely from clients (C++ represents class in the interface)
  - you want to share an implementation among multiple objects
    - Hidden from the client
    - Might use reference counting

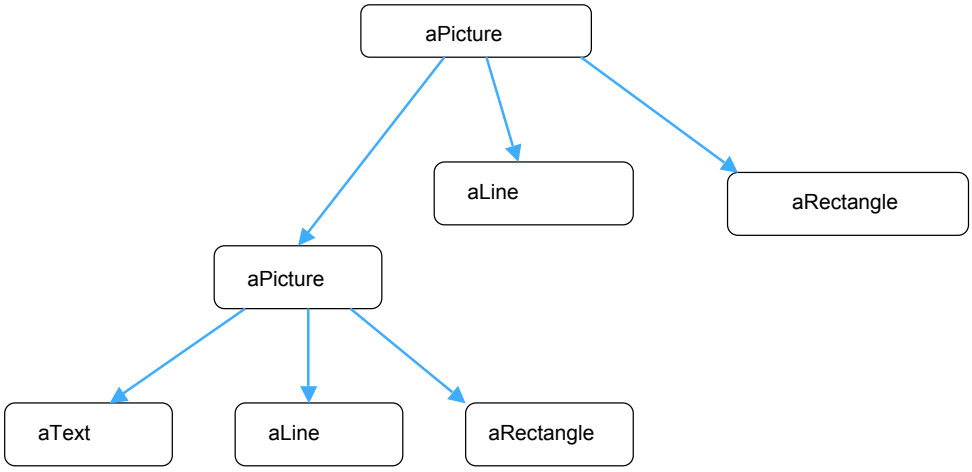
COMPOSITE
(Object Structural)

- Intent:
  - Compose objects into tree structures
  - Represent part-whole hierarchies
  - Let clients treat individual objects and compositions uniformly
- Motivation:
  - Example: Graphical application which allow grouping of objects into more complex structures
  - Simple implementation:
    - Classes for graphical primitives (Text, Lines)
    - Other classes that act as containers for primitives
  - Leads to different treatment of primitive objects and containers
  - More complexity, even though the user treats them identically

COMPOSITE
Motivation

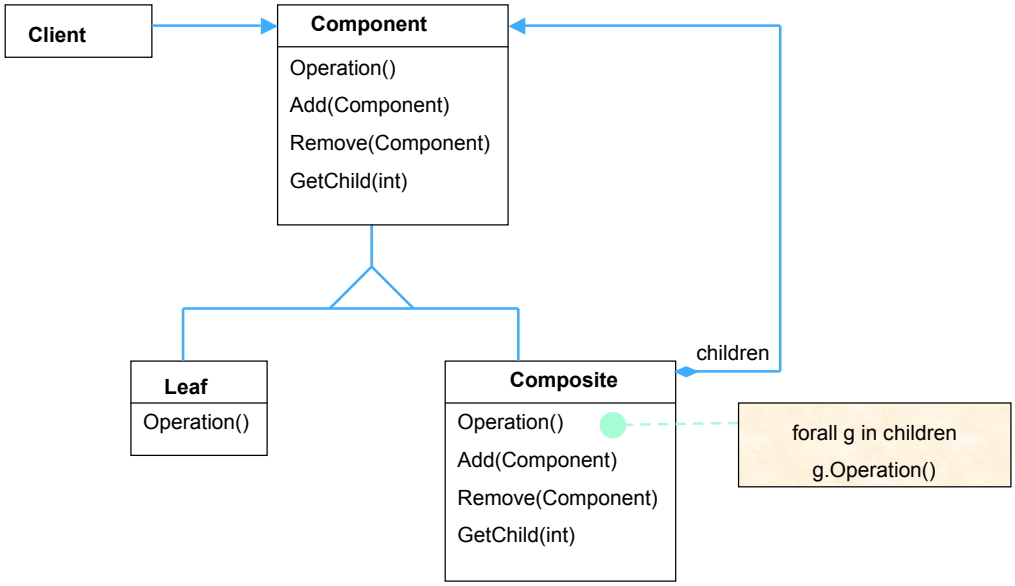


COMPOSITE
Motivation

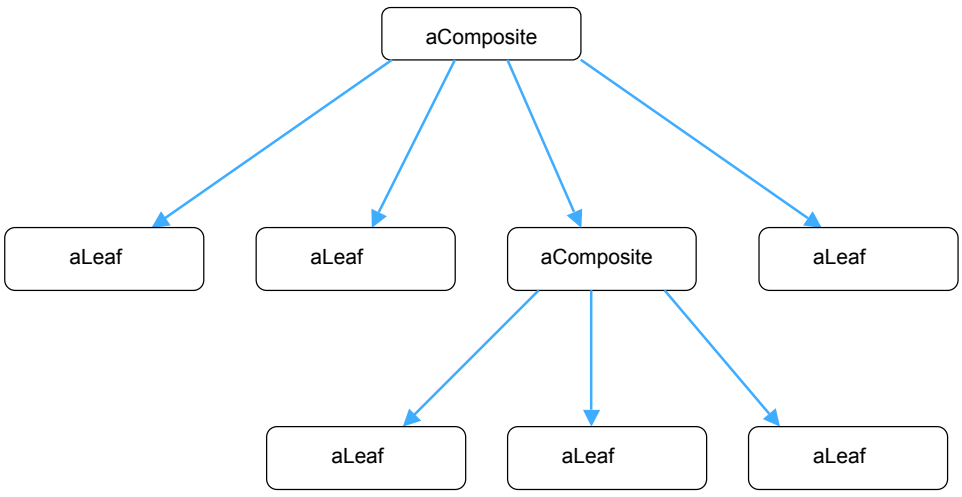


A typical composite object structure of recursively composed Graphic objects.

COMPOSITE
Structure



COMPOSITE
Typical Object Structure



COMPOSITE
Participants

- Component (Graphic)
  - Declares interface for objects in the composition
  - Implements default behavior for the interface
  - Declares interface for accessing and managing child components
- Leaf (Rectangle, Line, Text)
  - Leaf objects in the composition, without children
  - Implements behavior for primitive objects in the composition
- Composite (Picture)
  - Defines behavior for components having children
  - Stores child components
  - Implements child-related operations in the Component interface
- Client
  - Manipulates objects through Component interface

COMPOSITE
Collaborations

- Clients use the Component class interface
  - Interaction with objects in the composite structure
- Recipient is a Leaf
  - Request is handled directly
- Recipient is a Composite
  - Usually forwards requests to its child components
  - Possibly performing additional operations before / after forwarding

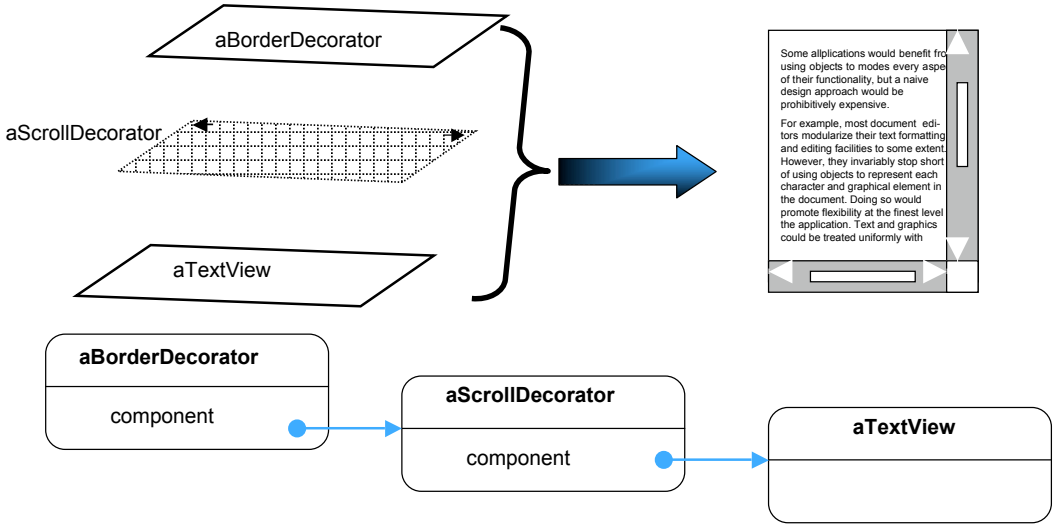
COMPOSITE
Applicability / Benefits

- Use the Composite pattern when
  - you want to represent part-whole hierarchies of objects
  - you want clients to be able to ignore the difference between compositions of objects and individual objects
  - You want clients to treat all objects in the structure uniformly
- Benefits
  - Whenever client code expects a primitive object, it can also take a composite object
  - Makes the client simple
  - Makes it easier to add new kinds of components

DECORATOR
(Object Structural)

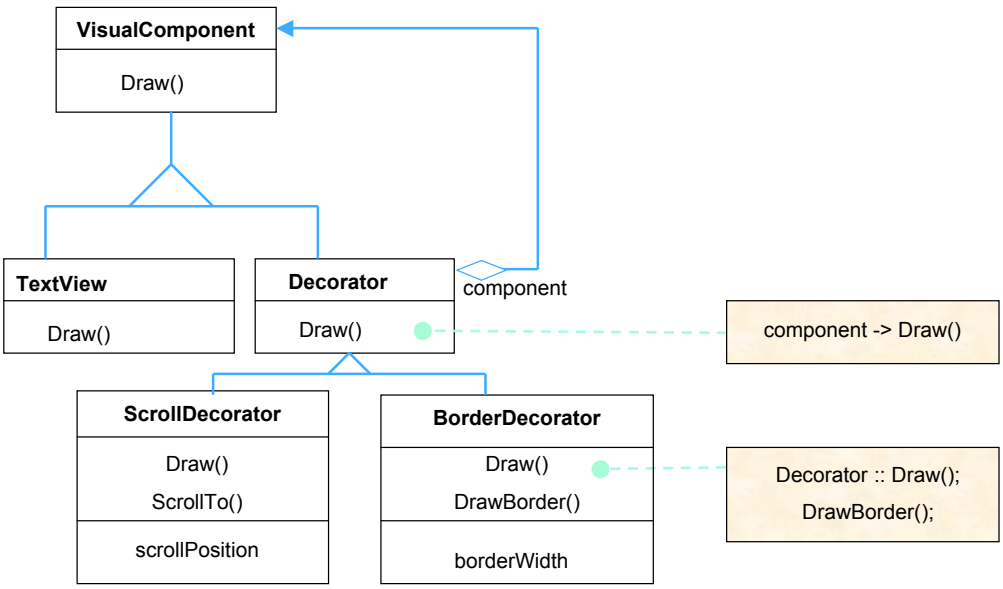
- Intent:
  - Attach additional functionality to an object dynamically
  - Flexible alternative to subclassing
  - Also known as *wrapper*
- Motivation:
  - Add responsibilities to individual objects, not an entire class
    - Inheritance as inflexible (static) solution
    - Clients cannot control the extension of object‘s functionality
  - Enclosing the object into another object
    - Decorator object adds the functionality
    - Conforms to the interface of the decorated component
    - Presence is transparent to clients
    - Recursive nesting possible

DECORATOR
Motivation

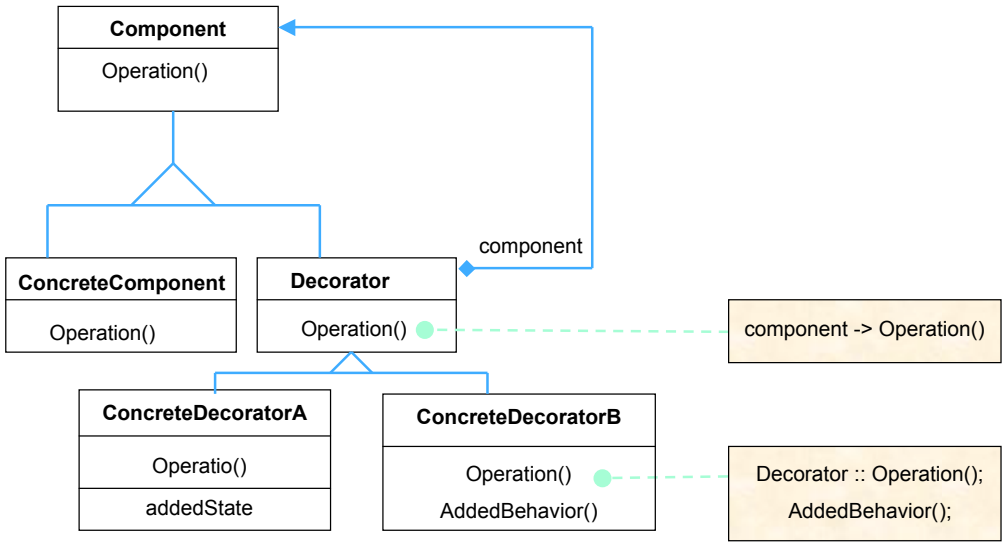


Example: Create a bordered, scrollable text view

DECORATOR
Motivation



DECORATOR
Structure



DECORATOR
Participants and Collaborations

Participants:

- Component (VisualComponent)
  - Interface for objects that can have responsibilities added to them dynamically
- ConcreteComponent (TextView)
  - Defines an object to which additional responsibilities can be attached
- Decorator
  - Maintains a reference to a Component object and defines interface that conforms to Component's interface
- ConcreteDecorator (BorderDecorator, ScrollDecorator)
  - Adds responsibilities to the component

Collaborations:

- Decorator forwards requests to its Component object
- May perform additional operations before and after forwarding

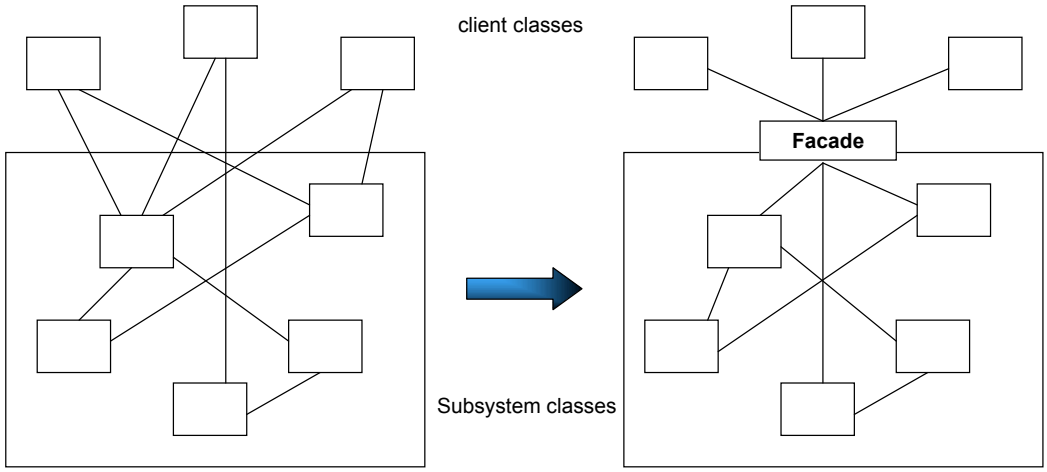
DECORATOR
Applicability / Benefits

- Use Decorator
  - to add responsibilities to individual objects dynamically /transparently
  - for responsibilities that can be withdrawn.
  - when extension by subclassing is impractical
    - large number of independent extensions might be possible
    - would produce an explosion of subclasses to support every combination
    - class definition may be hidden / unavailable for subclassing
- Benefits
  - More flexibility than static inheritance
  - Easy to add the same property more than once (double border)
  - Avoids feature-loaded classes at the top of the hierarchy
    - Add functionality incrementally when needed

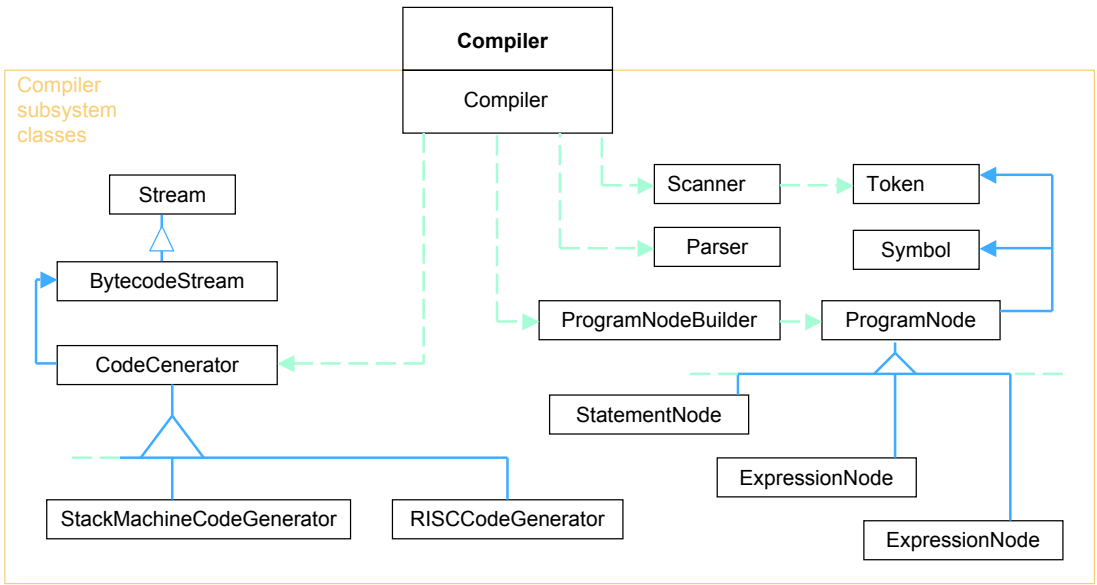
FACADE
(Object Structural)

- Intent:
  - Unified interface to a set of interfaces
  - Higher-level interface that makes the subsystem easier to use
- Motivation:
  - Structuring a system into subsystems
    - Helps reduce complexity
    - Minimize communication and dependencies between subsystems
  - Single, simplified interface to the more general facilities of subsystems
  - Example:
    - Compiler environment with subsystems (parser, scanner, …)
    - High-level compiler interface for most clients
    - Glue together specific functionalities, without hiding them completely

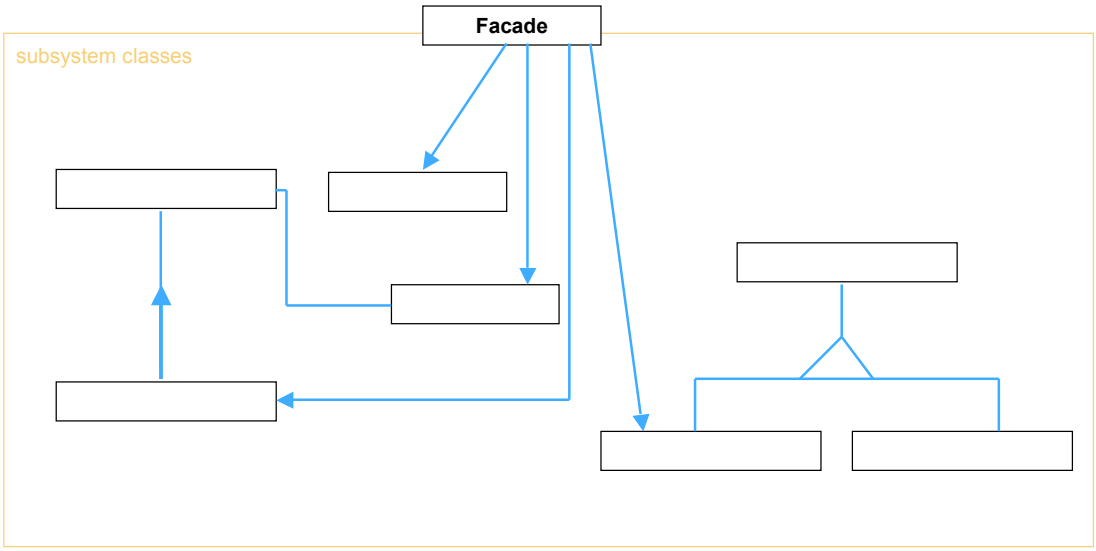
FACADE
Motivation



FACADE
Motivation



FACADE
Structure



FACADE
Participants and Collaborations

Participants:

- Facade (Compiler)
  - Knows which subsystem classes may handle a request
  - Delegates client requests to appropriate subsystem objects
- Subsystem classes (Scanner, Parser, ProgramNode)
  - Implement subsystem functionality
  - Have no knowledge of the facade (no references to it)

Collaborations:

- Clients sends requests to Façade
- Forwarded to the appropriate subsystem object(s)
- Facade may have to translate its interface to subsystem interfaces
- Clients do not have to access subsystem objects directly

FACADE
Applicability

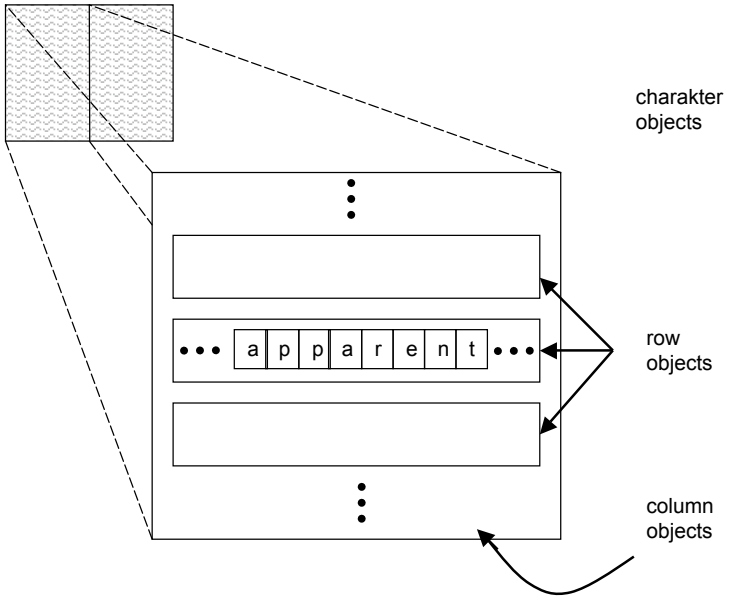
Use the Facade pattern:

- to provide a simple interface to a complex subsystem
  - Subsystems often get more complex as they evolve
  - Most patterns result in smaller classes – harder for clients
  - Simple default view, good enough for most clients
- in case of dependencies between client and abstraction classes
  - Facade decouples the subsystems from clients and other subsystems
  - Promoting subsystem independence and portability
  - Works also for subsystem interdependencies

FLYWEIGHT
(Object Structural)

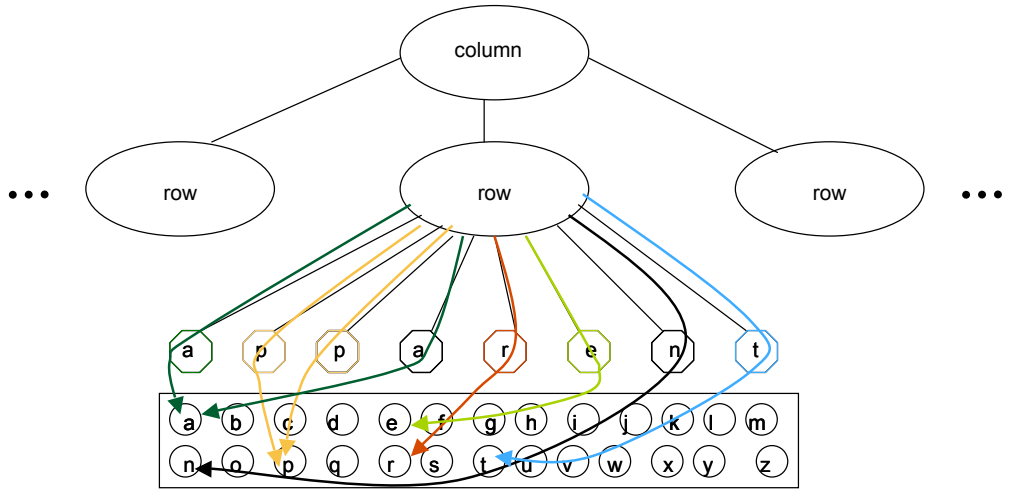
- Intent:
  - Use sharing to support large numbers of small objects efficiently
- Motivation:
  - Applications could benefit from using objects throughout their design
  - Naive implementation would be prohibitively expensive
  - Example: Document editor
    - Use objects to represent embedded elements (figures, tables)
    - Each character could be handled as own object
    - Memory / run-time overhead costs
  - Share objects to allow their use at fine granularities
    - Intrinsic state: stored in the flyweight
    - Extrinsic state: depends upon context, can't be shared; passed by client

FLYWEIGHT
Motivation



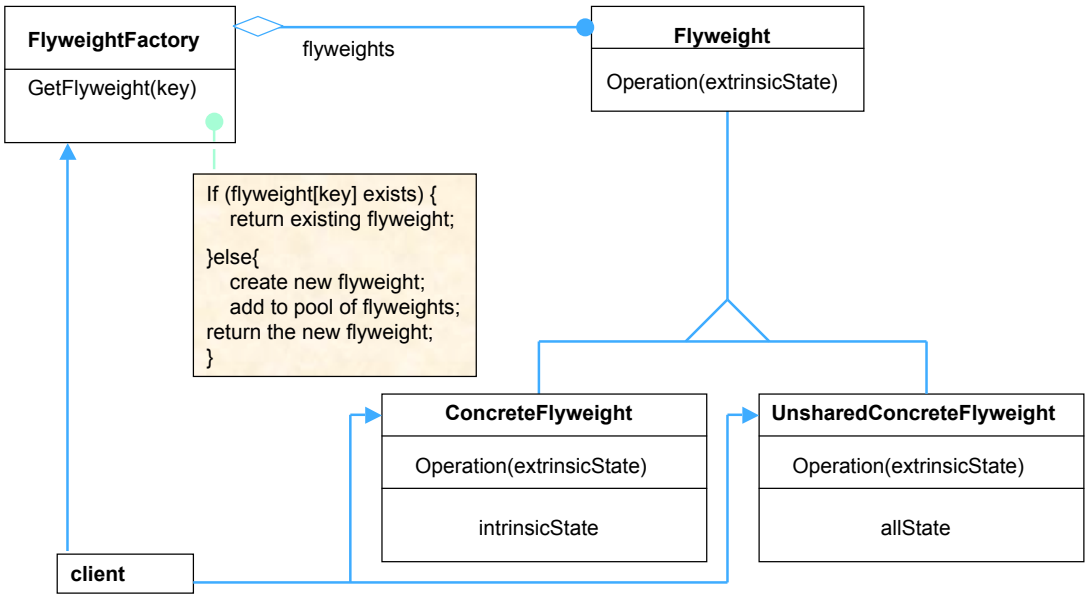
FLYWEIGHT
Motivation

Logically - one object per character in the document

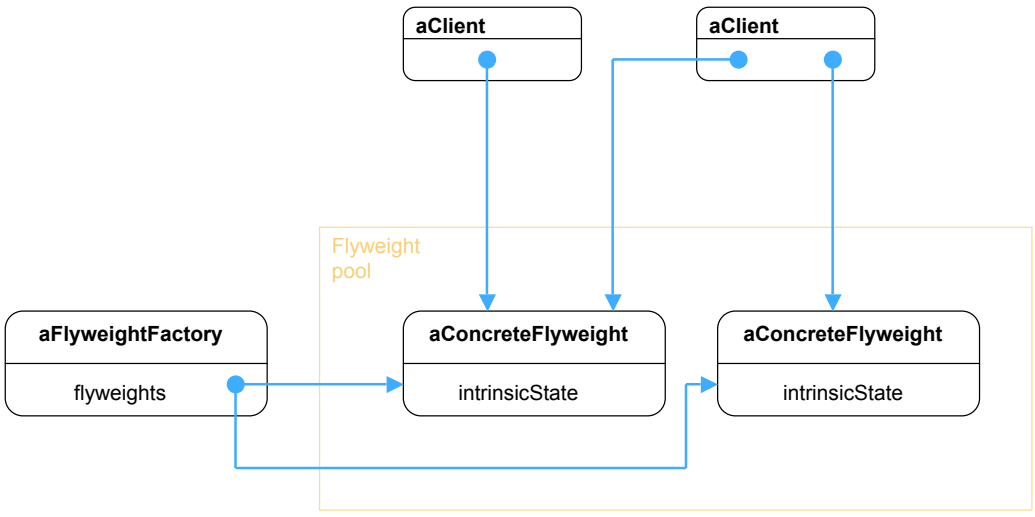


Physically - one shared flyweight object per character

FLYWEIGHT
Structure



FLYWEIGHT
Structure



FLYWEIGHT Participants

- Flyweight
  - Declares an interface through which flyweights can receive and act on extrinsic state
- ConcreteFlyweight
  - Implements Flyweight interface and adds storage for intrinsic state
  - Must be sharable
  - Any state it stores must be independent of concrete object‘s context
- FlyweightFactory
  - Creates and manages flyweight objects
  - Ensures that flyweights are shared properly
- Client
  - Maintains reference to flyweight(s)
  - Computes or stores the extrinsic state of flyweight(s)

FLYWEIGHT Collaborations

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic
  - *Intrinsic state* is stored in the ConcreteFlyweight object
  - *Extrinsic state* is stored or computed by Client objects
  - State is passed to flyweight with operation invocation
- Clients should not instantiate ConcreteFlyweights
  - Clients must obtain them exclusively from the FlyweightFactory
  - Ensures proper sharing of objects

FLYWEIGHT
Applicability / Benefits

Applicability:

- Effectiveness depends heavily on how and where the pattern is used
- Application when:
  - application uses a large number of objects, **and**
  - storage costs are high (sheer quantity of objects), **and**
  - most object state can be made extrinsic
- Groups of objects may be replaced by relatively few shared objects
  - extrinsic state must be removed

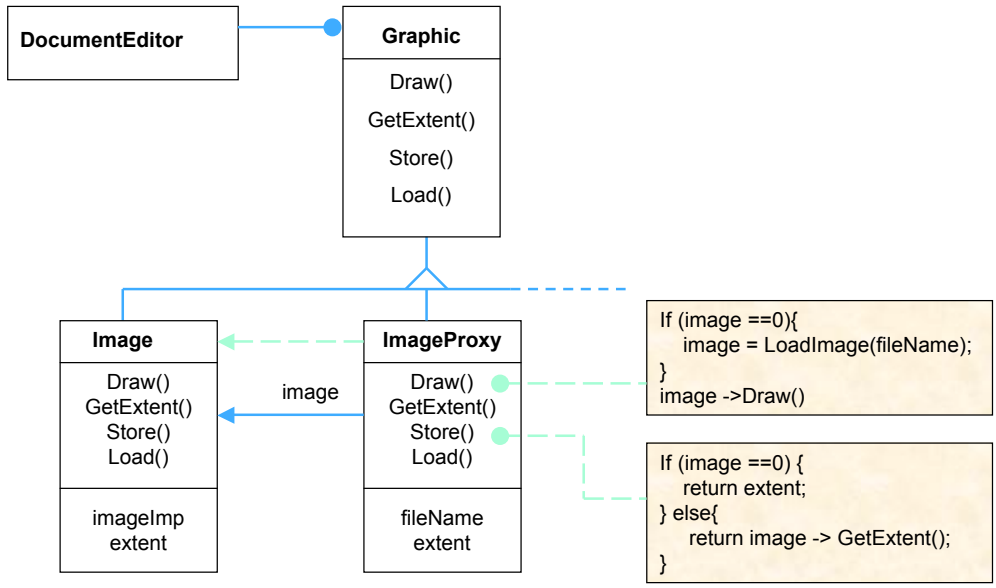
Benefits:

- Run-time costs (extrinsic state handling) vs. space saving (increases with amount of shared state)

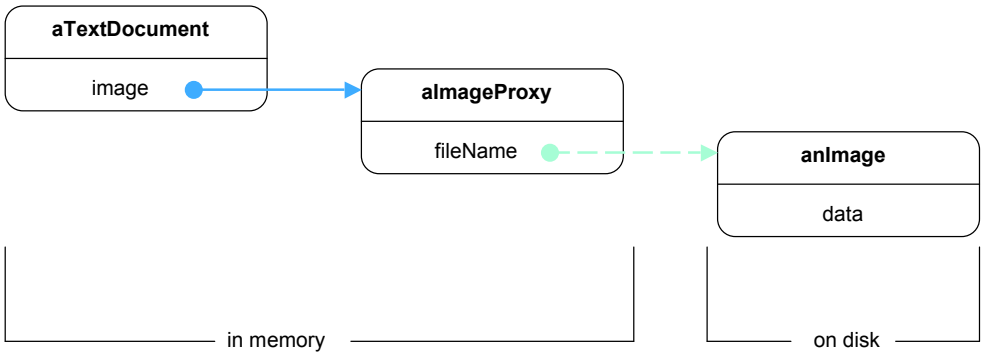
PROXY
(Object Structural)

- Intent:
  - Surrogate or placeholder to control access to another object
- Motivation:
  - Example: Defer the full cost of object creation and initialization until the real need for it
  - Document editor that can embed graphical objects into an document
  - Creation of image objects can be expensive
  - Opening the document should still be fast
  - Image proxy might act as stand-in for the real image

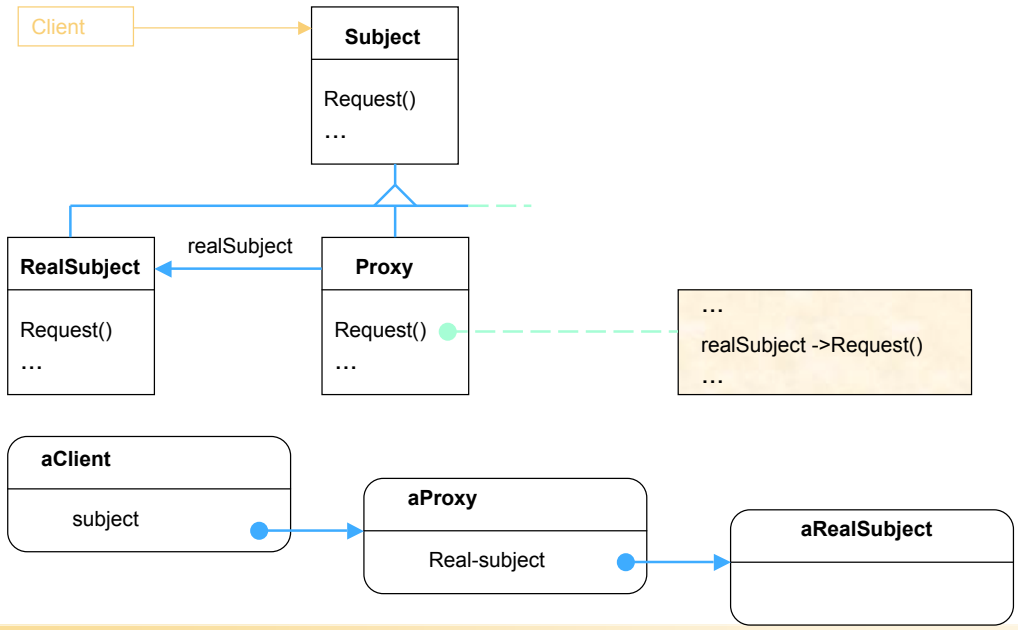
PROXY
Motivation



PROXY
Motivation



PROXY
Structure



PROXY
Participants and Collaborations

Participants:

- Proxy (ImageProxy)
  - Maintains reference to the *RealSubject*
  - Provides interface identical to the *RealSubject*
  - Controls access to *RealSubject*; manages creation and deletion
- Subject (Graphic)
  - Defines common interface for *RealSubject* and *Proxy*
- RealSubject (Image)
  - Defines the real object that the proxy represents

Collaborations:

- *Proxy* forwards requests to *RealSubject* when appropriate
- Depends on the kind of proxy

PROXY
Applicability

- Covers need for a more sophisticated reference to an object than a simple pointer
- Common situations:
  - *Remote proxy* provides a local representative for an object in a different address space (NeXTSTEP)
  - *Virtual proxy* creates expensive objects on demand (ImageProxy)
  - *Protection proxy* controls access to the original object (useful when objects should have different access rights)
  - *Smart reference* - replacement for a bare pointer that performs additional actions on access

Design Pattern Space

Defer object creation to another class

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

Defer object creation to another object

Describe ways to assemble objects

Describe algorithms and flow control