# Structural Patterns

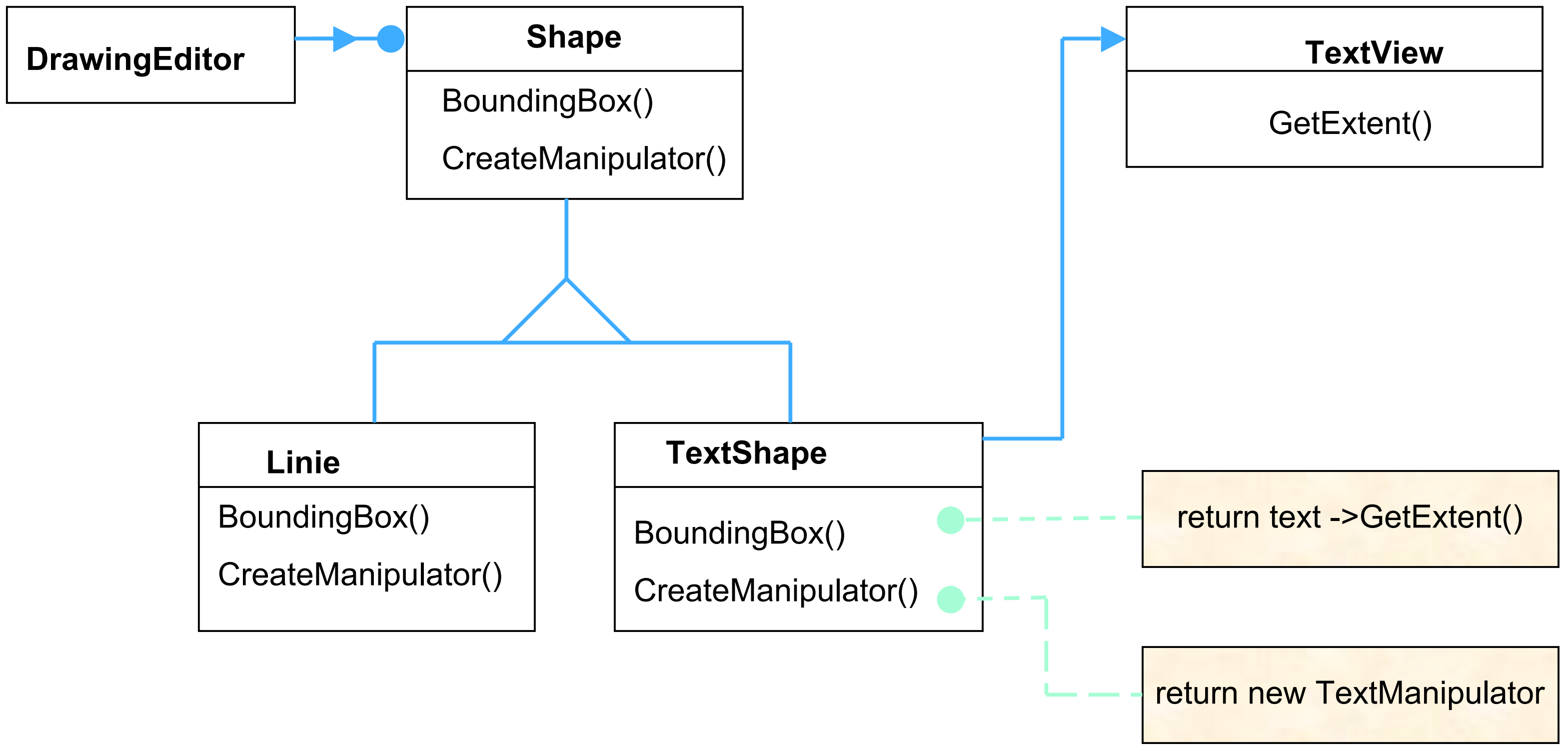
- Describe how classes and objects are composed to form larger structures

- Structural **class** patterns use inheritance to compose interfaces or implementations
- Structural **object** patterns describe ways to compose objects to realize new functionality

# 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 because of incompatible interfaces.
- Motivation:
  - Sometimes a toolkit class that‘s designed for reuse is not reusable because ist interface does not match the domain-specific interface an application requires

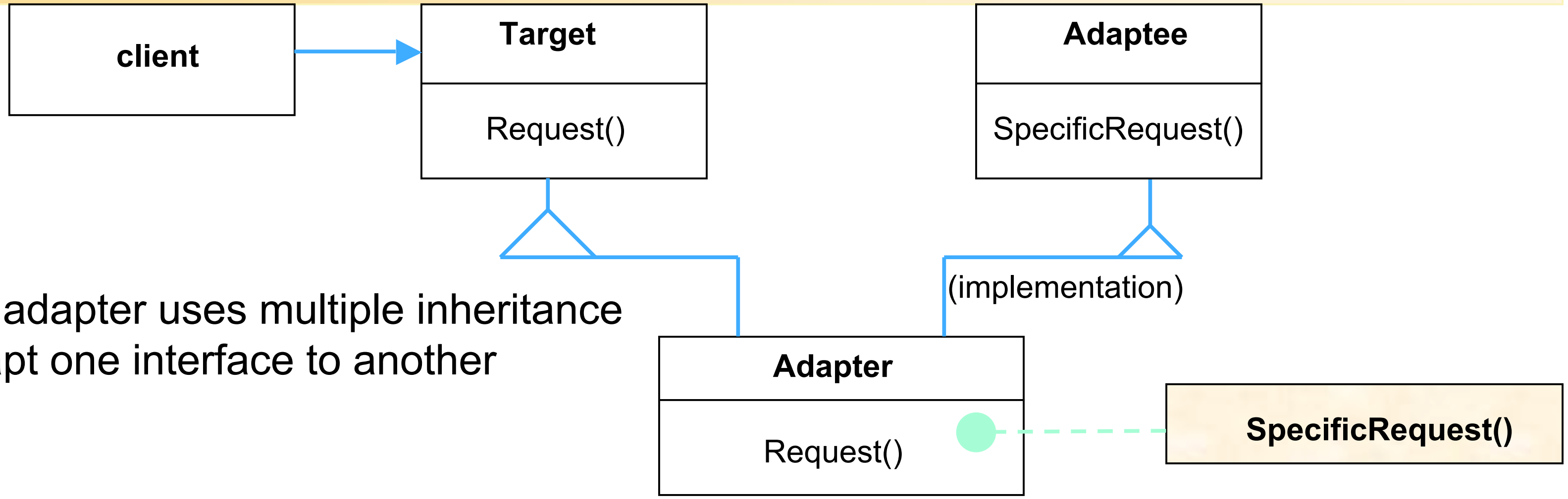
# ADAPTER - Motivation



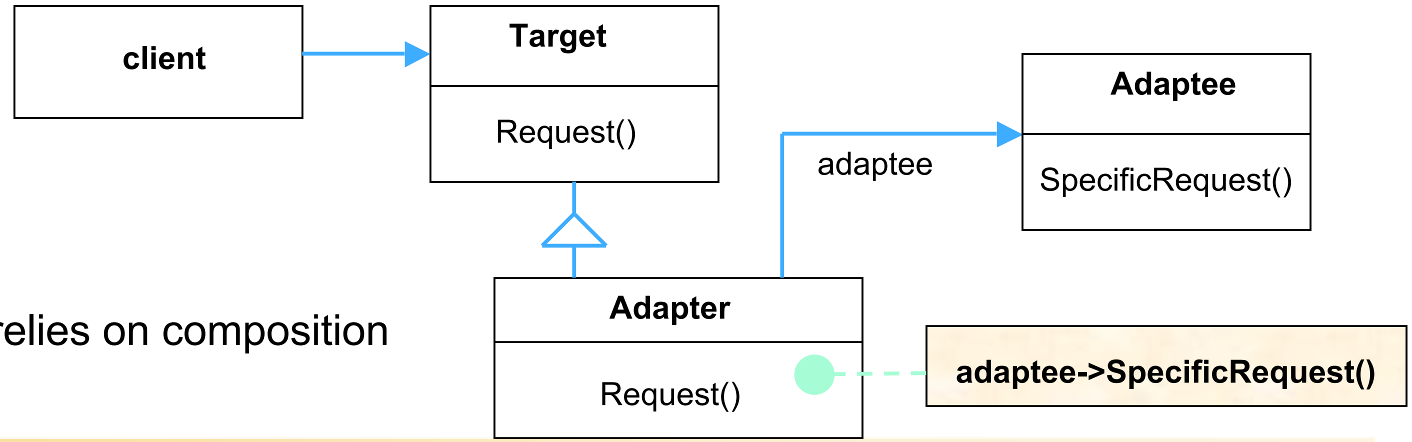
# Applicability

- Use the Adapter pattern when
  - you want to use an existing class, and its interface does not match the one you need.
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don´t necessarily have compatible interfaces.
- (object adapter only)
  - you need to use several existing subclasses, but it´s impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

# ADAPTER - Structure




Class adapter uses multiple inheritance to adapt one interface to another




Object adapter relies on composition

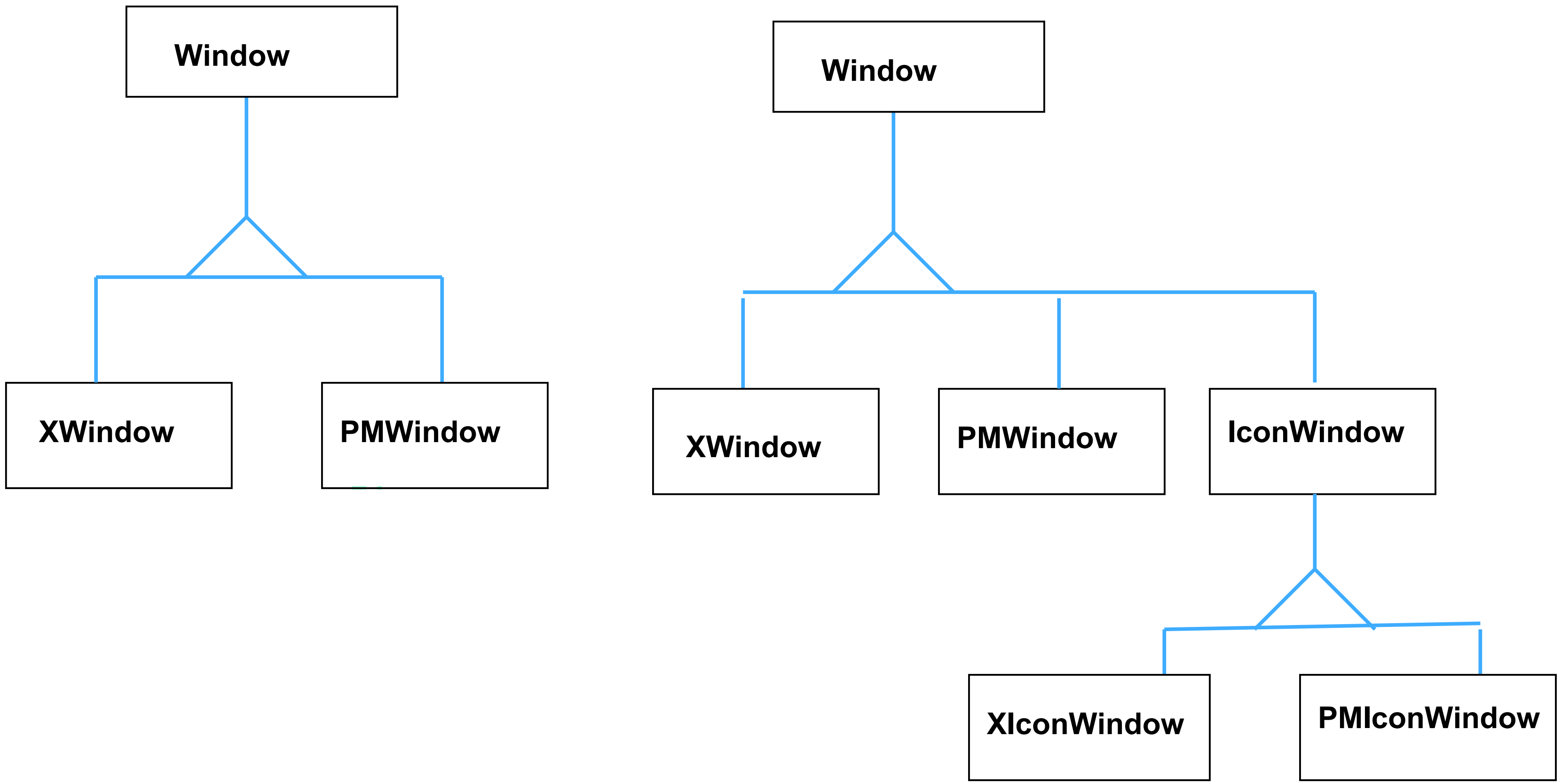
# Participants and Collaborations

- 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
- Collaborations:
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adapter operations that carry out the request.

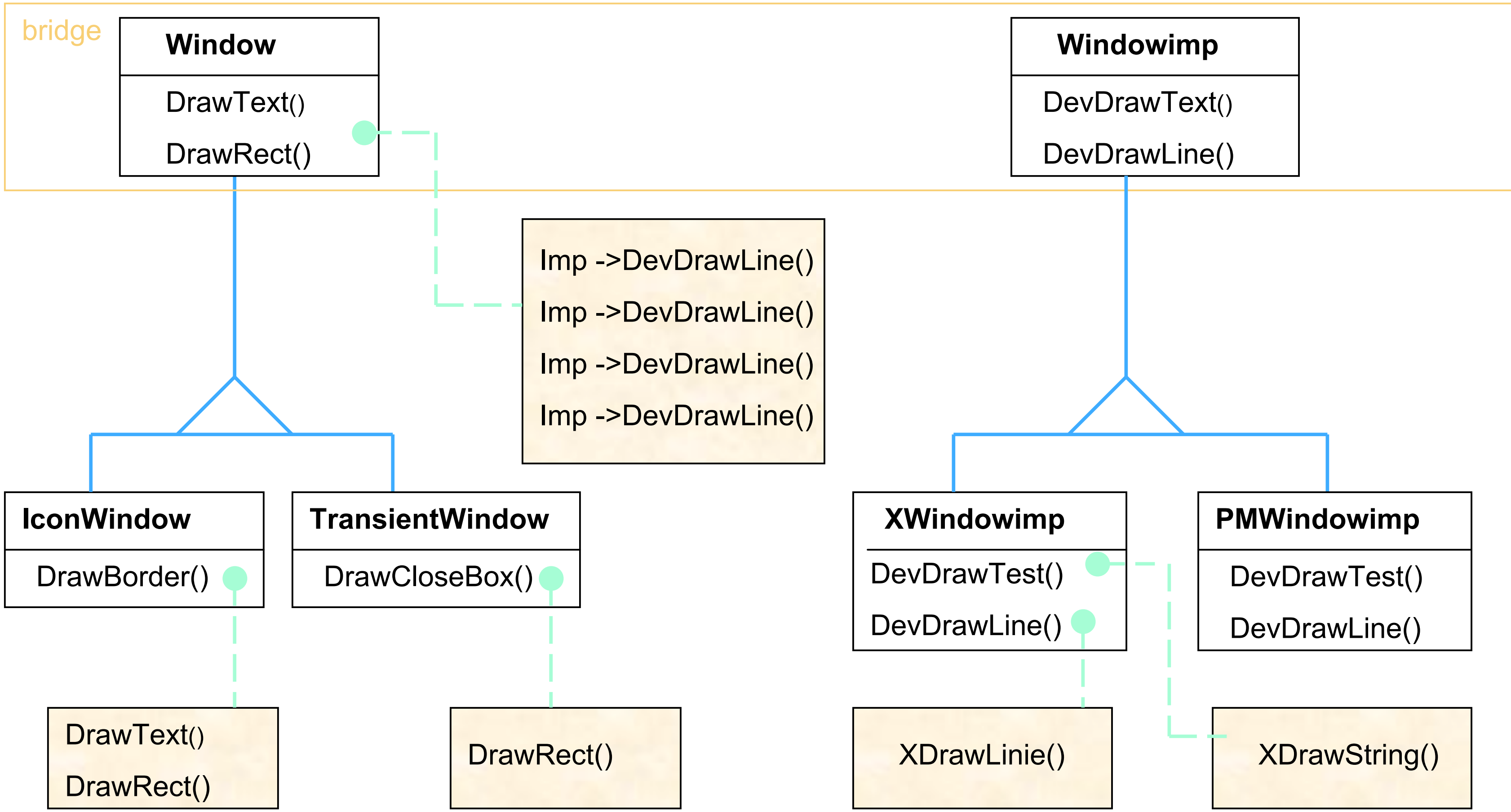
# BRIDGE
# (Object Structural)

- Intent:
  - Decouple an abstraction from its implementation so that the two can vary independently
- Motivation:
  - Inheritance helps when an abstraction can have multiple possible implementations but is sometimes not flexible enough
  - The bridge patterns puts an abstraction and its implementation in separate class hierarchies
  - Example: There is one class hierarchy for Window interfaces (Window, IconWindow, TransientWindow) and a separate hierarchy for platform-specific windows implementations (with WindowImp as root)

# BRIDGE - Motivation



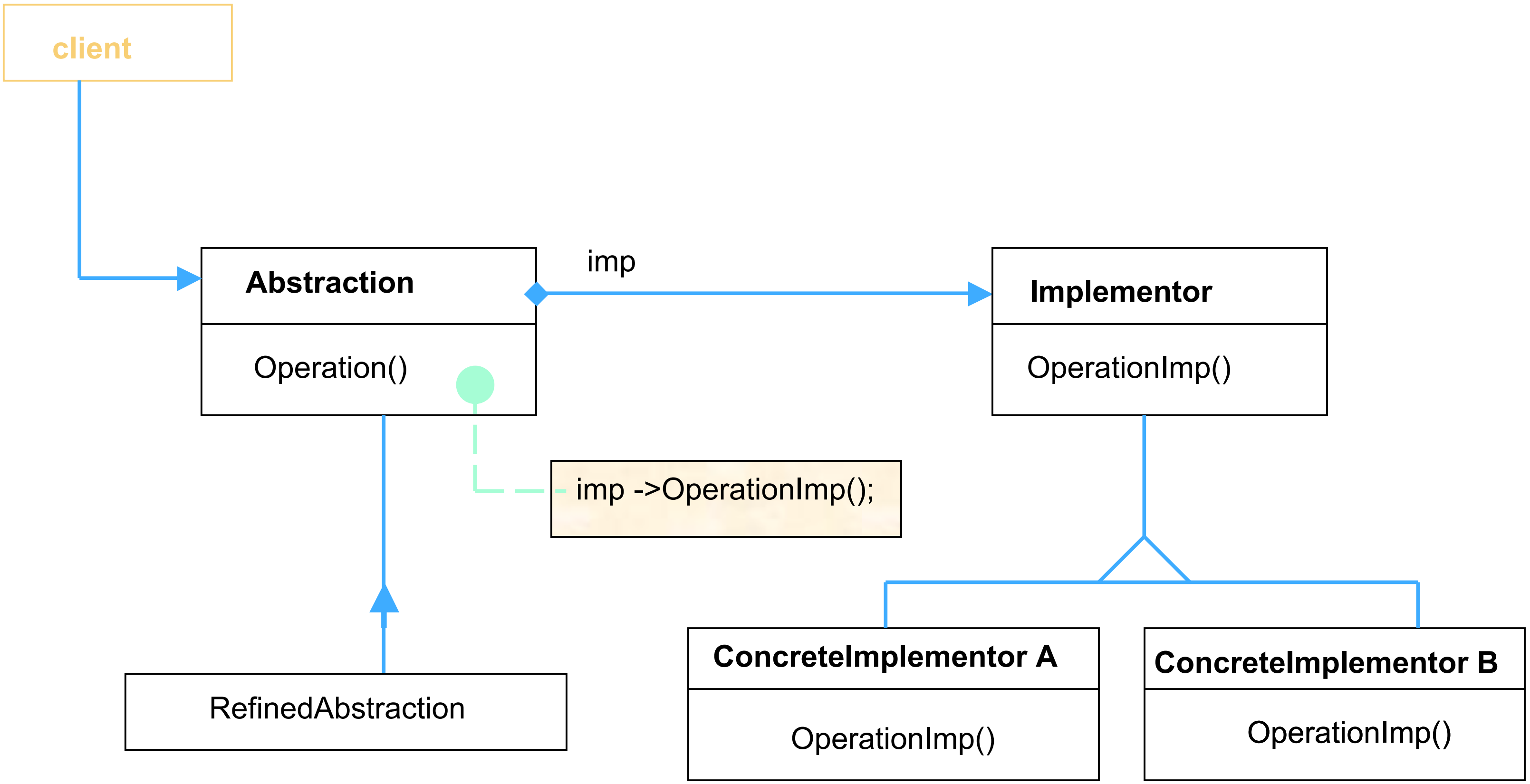
# BRIDGE - Motivation



# Applicability

- Use the Bridge pattern when:
  - you want to avoid a permanent binding between an abstraction and its implementation. (when the implementation must be selected or switched at run-time)
  - both the abstractions and their implementations should be extensible by subclassing.
    Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
  - (C++) you want to hide the implementation of an abstraction completely from clients. In C++ the representation of a class is visible in the class interface.
  - You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should bi hidden from the client.

# BRIDGE - Structure



# Participants and Collaborations

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 defines higher-level ops.
- ConcreteImplementor (XWindowImp, PMWindowImp)
  - Implements the Implementor interface, defines concrete implementation

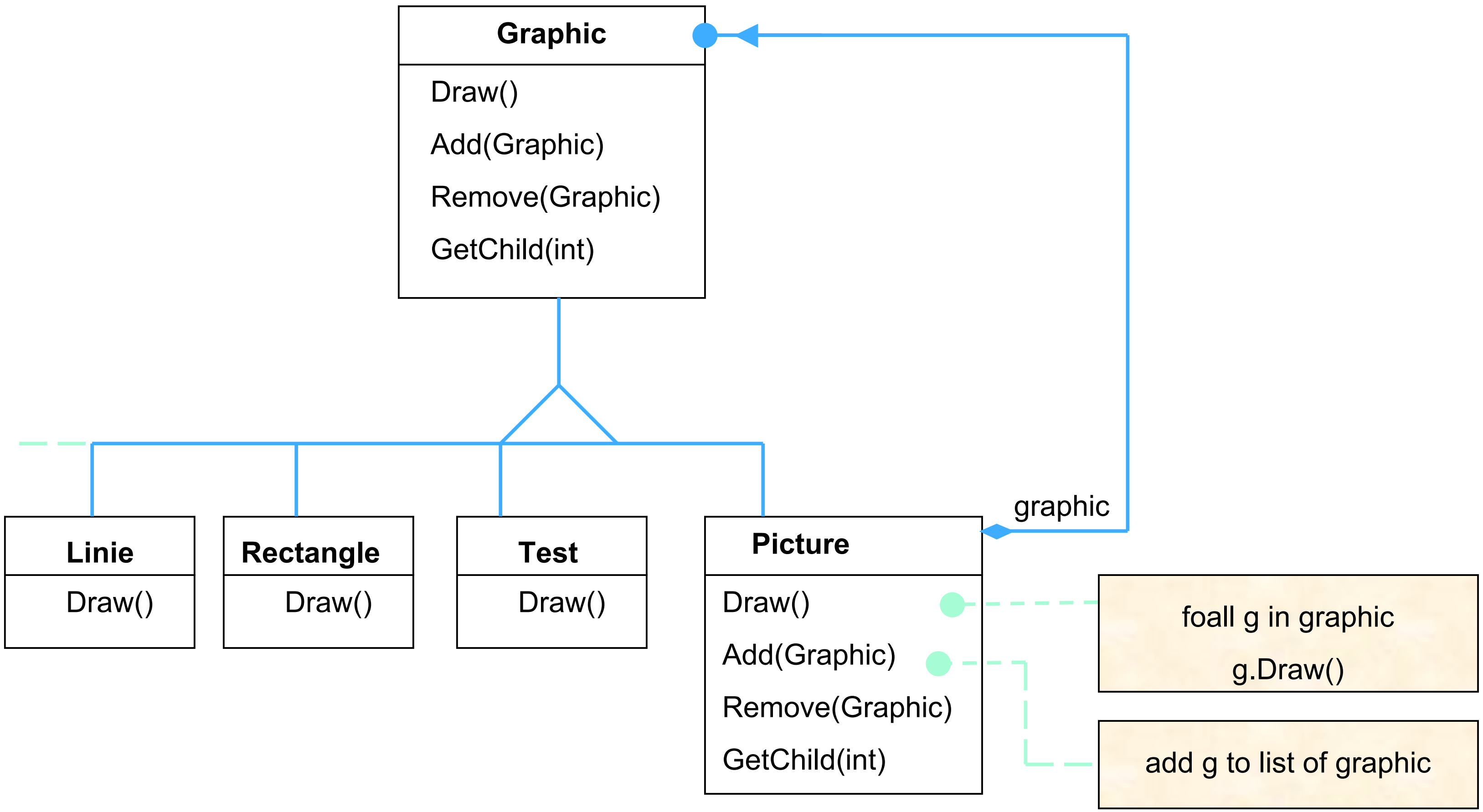
Collaborations:

- Abstraction forwards client requests to its Implementor object.

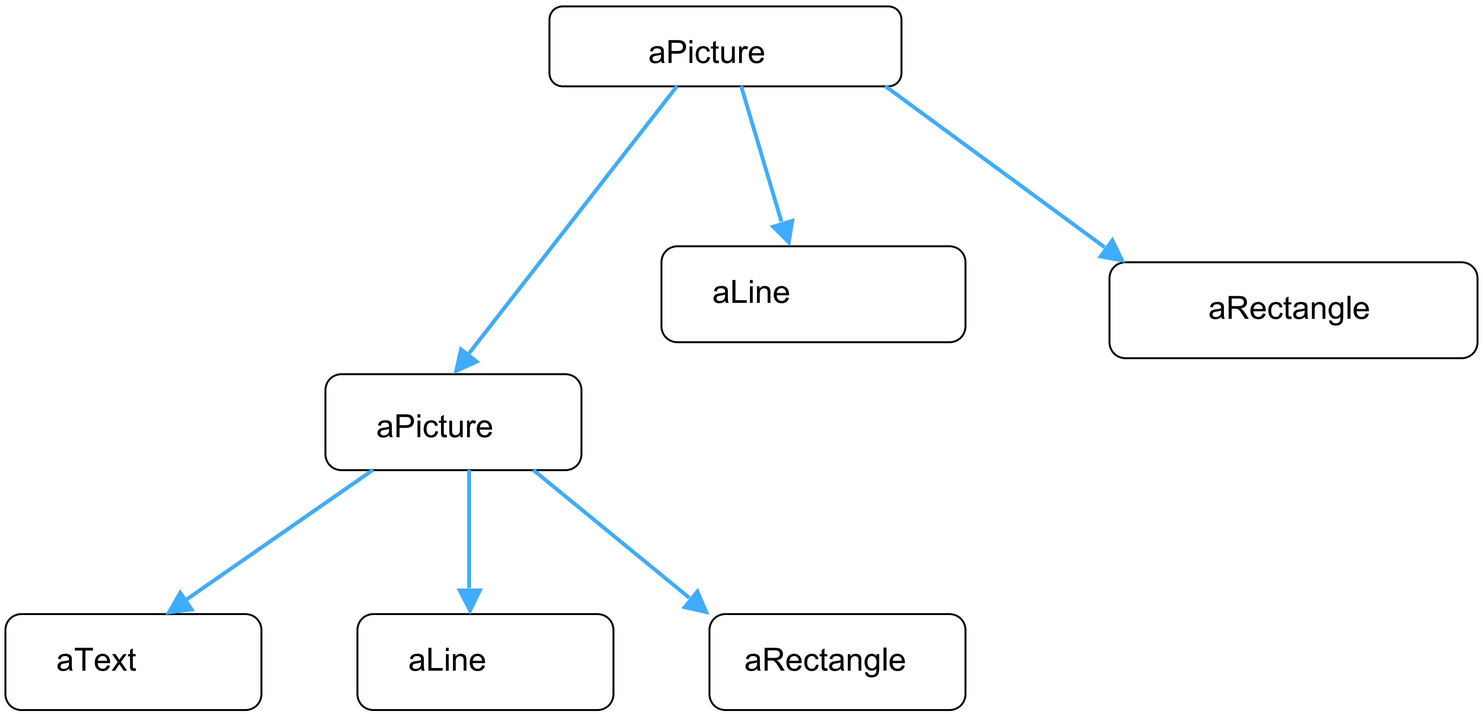
# COMPOSITE
# (Object Structural)

- Intent:
  - Compose objects into tree structures to represent part-whole hierarchies.
  - Composite lets clients treat individual objects and compositions of objects uniformly.
- Motivation:
  - Apps often allow grouping of objects into more complex structures
  - Single implementation could define classes for graphical primitives (Text, Lines) plus other classes that act as containers for primitives
  - But: code that uses these classes must treat primitive objects and containers differently (even if user treats them identically)

# COMPOSITE - Motivation



# COMPOSITE - Motivation

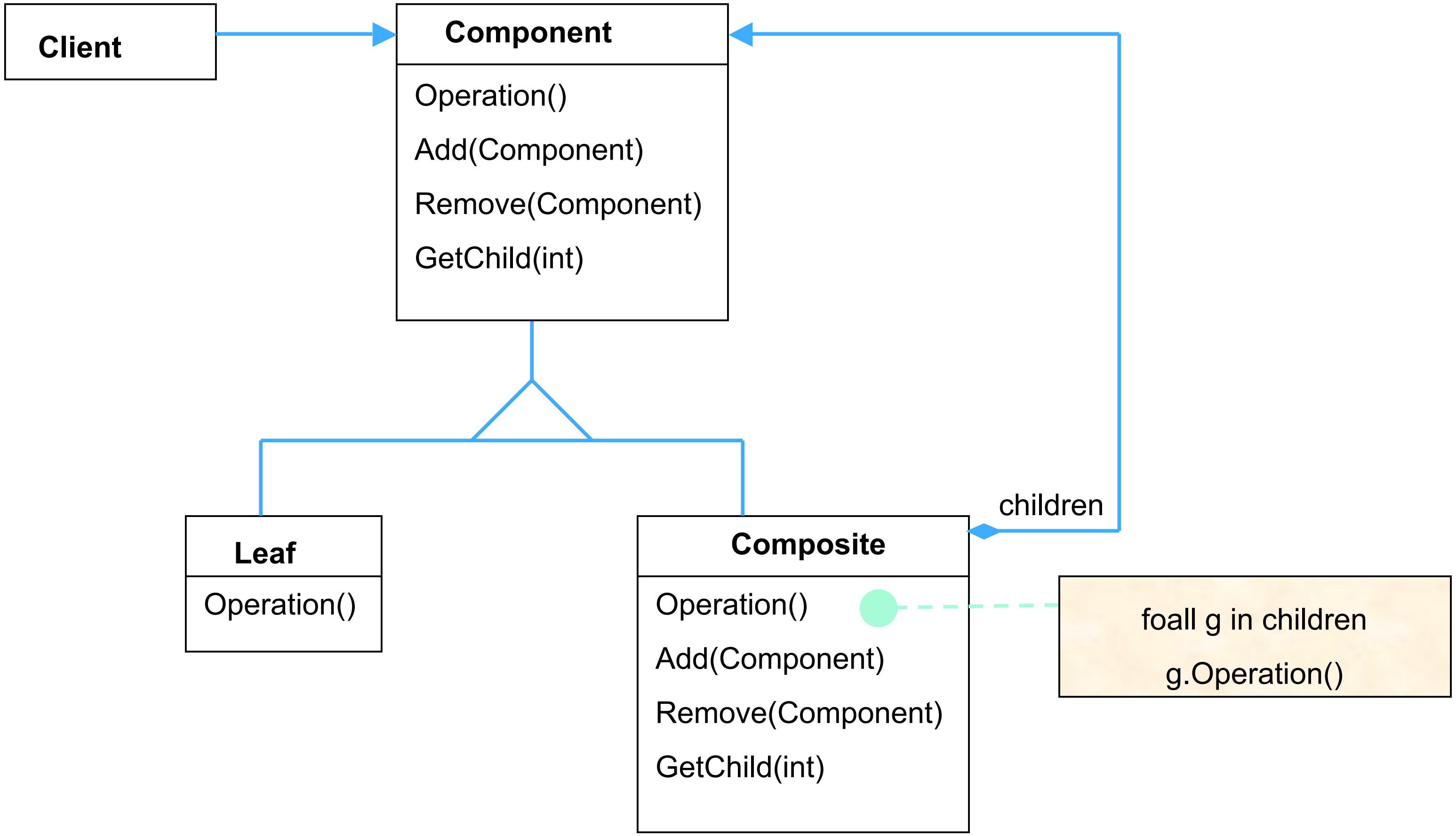



A typical composite object structure of recursively composed Graphic objects.

# Applicability

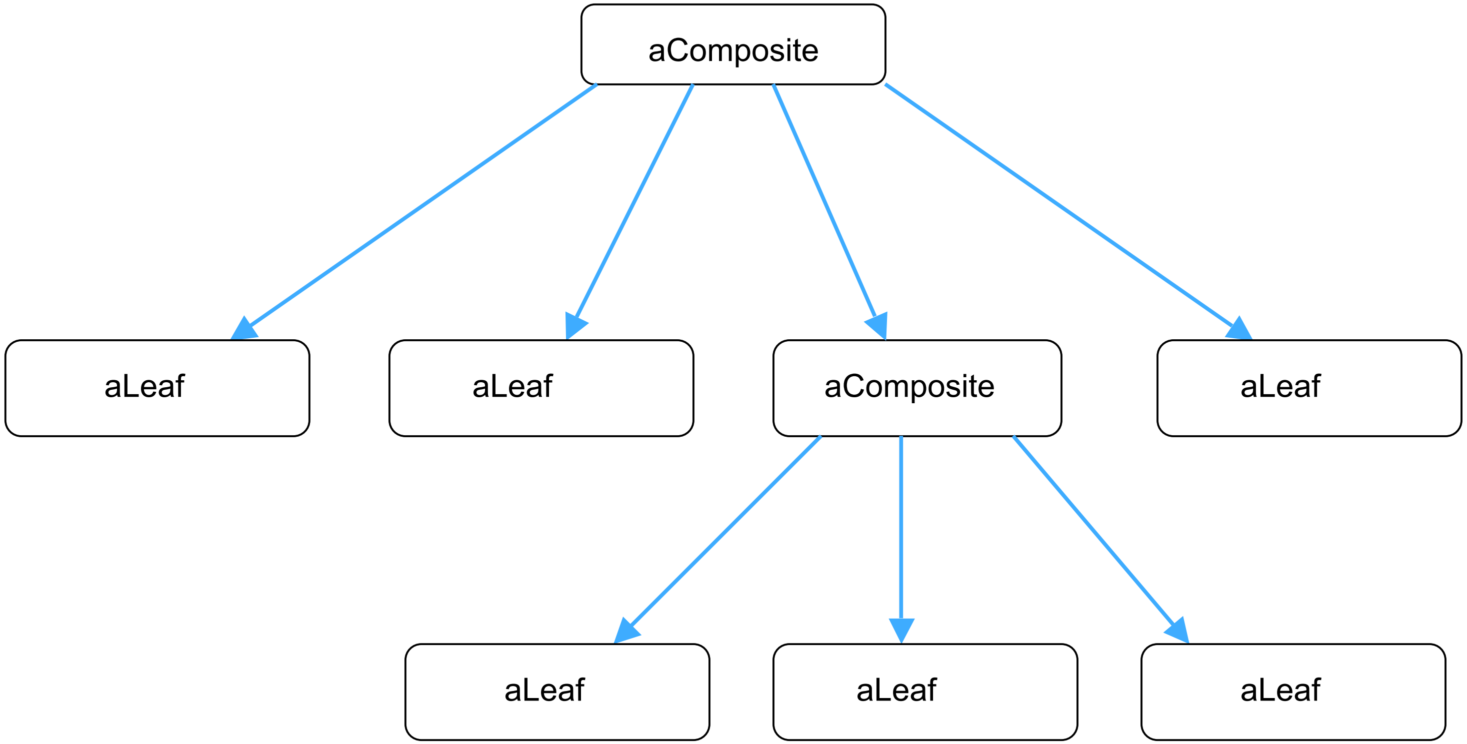
- Use the Composite pattern when
  - You want to represent part-whole hierarchies of objects.
  - You wants clients to be able to ignore the difference between compositions of objects and individual objects.
  - Clients will treat all objects in the composite structure uniformly.

# COMPOSITE - Structure



# COMPOSITE - Structure

A typical Composite object structure might look like this:



# Participants

- Component (Graphic)
  - Declares interface for objects in the composition
  - Implements default behavior for the interface common to all classes
  - Declares interface for accessing and managing child components
  - (optional) defines interface for accessing component’s parent
- Leaf (rectangle, Line, Text, etc.)
  - Represents leaf objects in the composition - has no children
  - Defines behavior for primitive objects in the composition
- Composite (Picture)
  - Defines behavior for components having children
  - Stores child components
  - Implements child-relate operations in the Component interface
- Client
  - Manipulates objects through Component interface

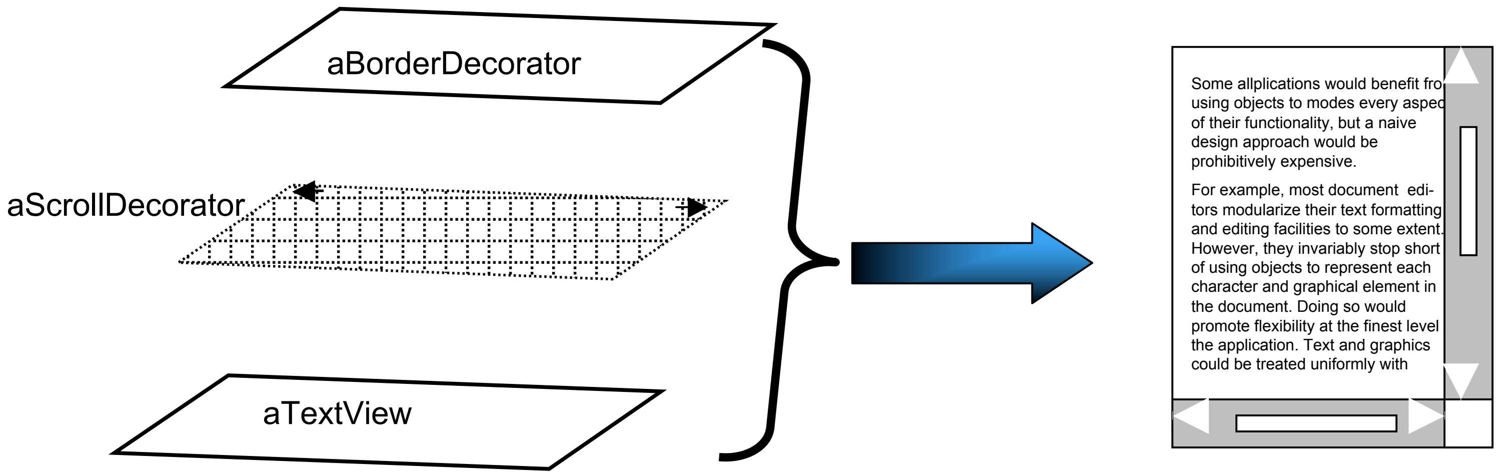
# Collaborations

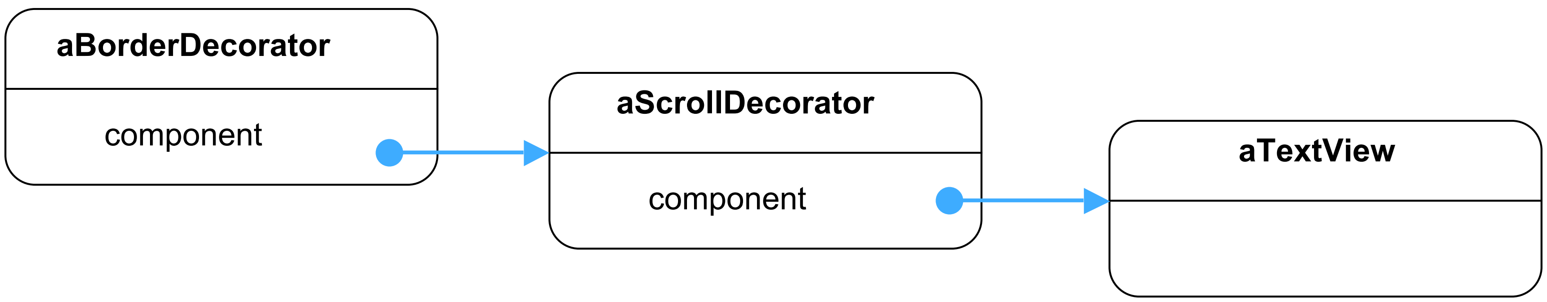
- Clients use the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

# DECORATOR
# (Object Structural)

- Intent:
  - Attach additional responsibilities to an object dynamically.
  - Decorators provide a flexible alternative to subclassing for extending functionality.
- Motivation:
  - Sometimes we want to add responsibilities to individual objects, not an entire class
  - Inheritance is an inflexible (static) solution to the problem. Clients cannot control the way how an object‘s functionality is extended
  - Enclosing the object into another object that adds the functionality is the more flexible approach - the **decorator**

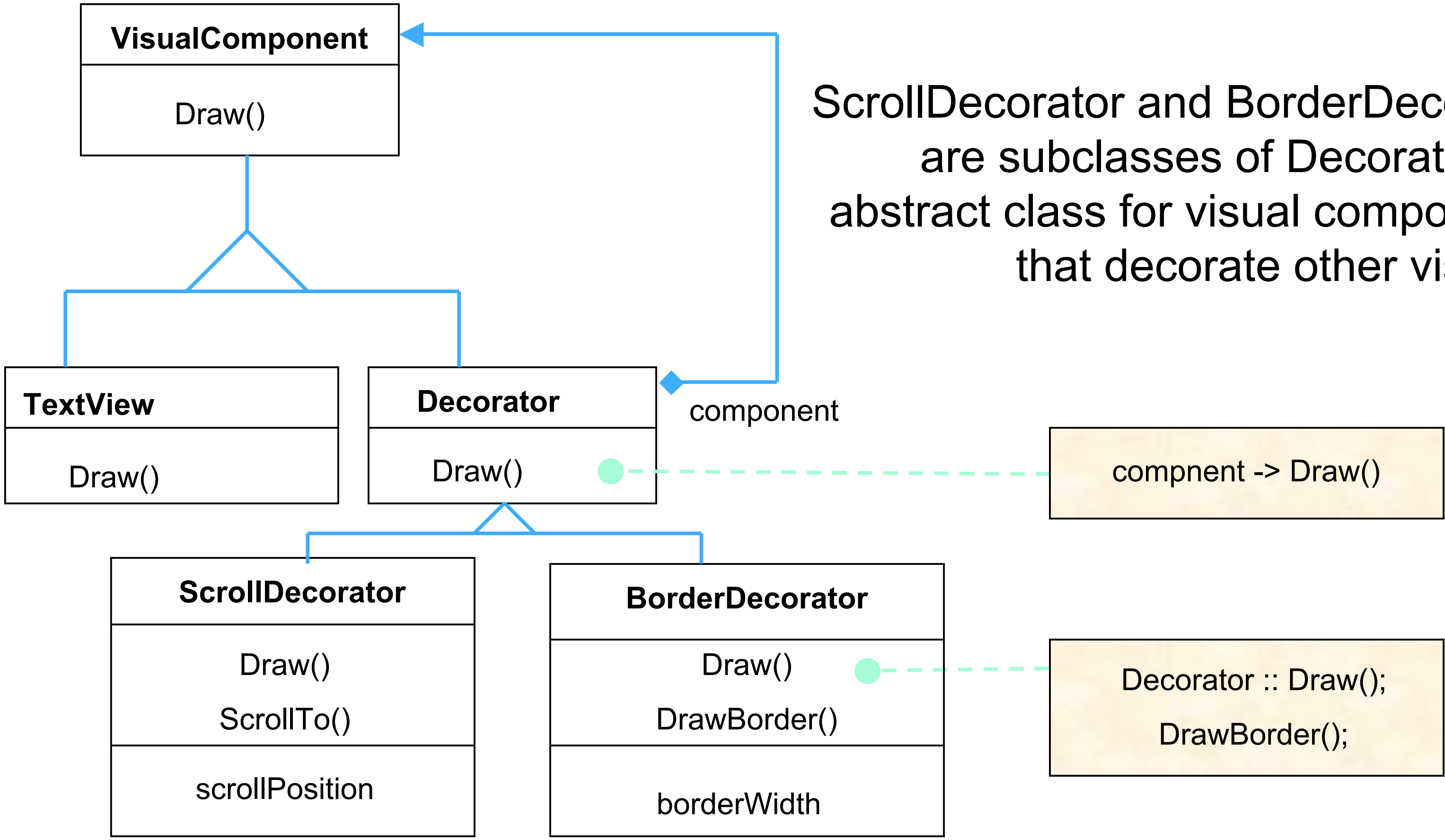
# DECORATOR - Motivation




Use composition to create a boredered, scrollable text view

# DECORATOR - Motivation



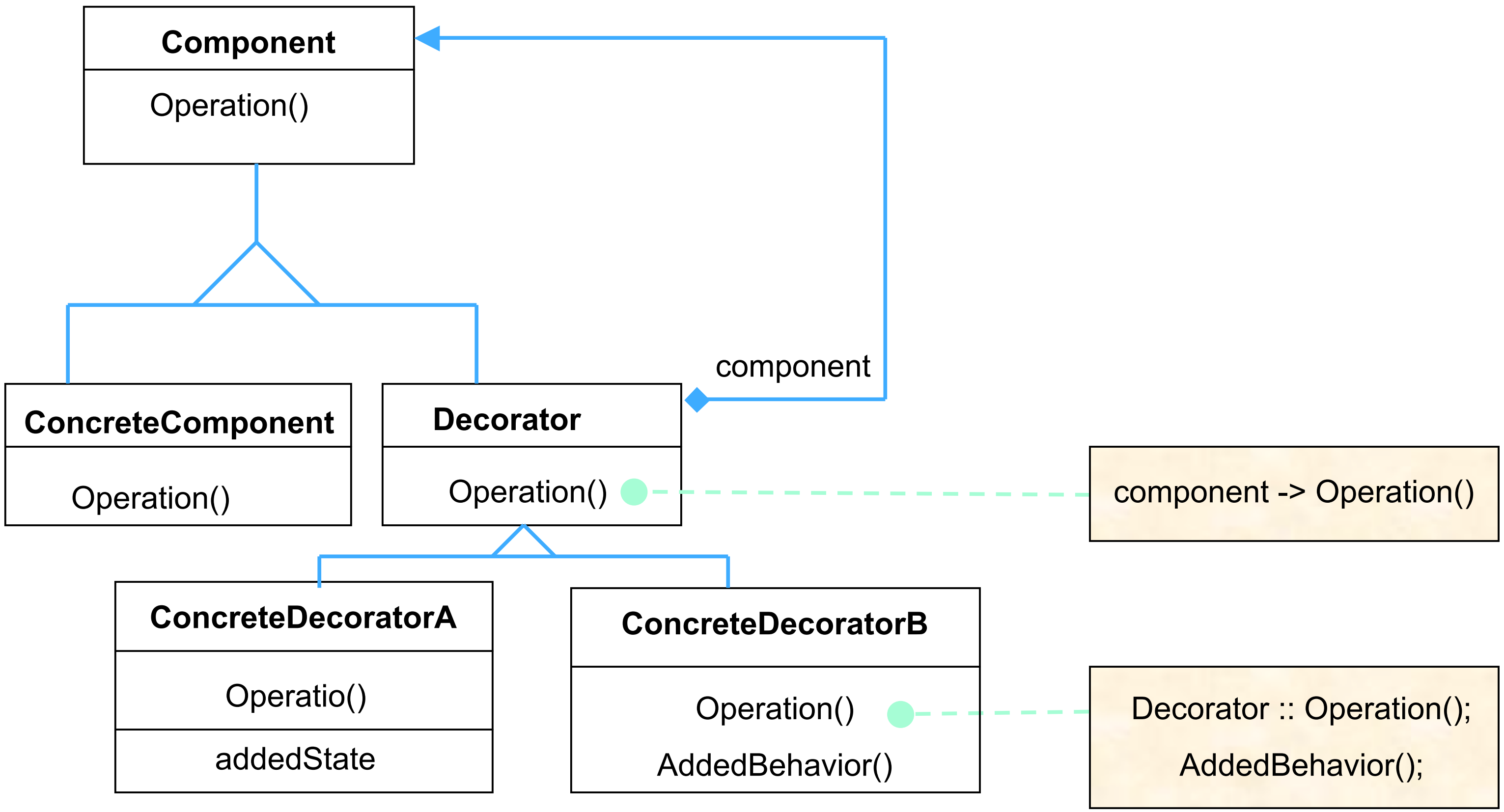

ScrollDecorator and BorderDecorator are subclasses of Decorator, an abstract class for visual components that decorate other visuals.

# Applicability

- Use Decorator
  - To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - For responsibilities that can be withdrawn.
  - When extension by subclassing is impractical.

    Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

# DECORATOR - Structure



# Participants and Collaborations

Participants:

- Component (VisualComponent)
  - Defines 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

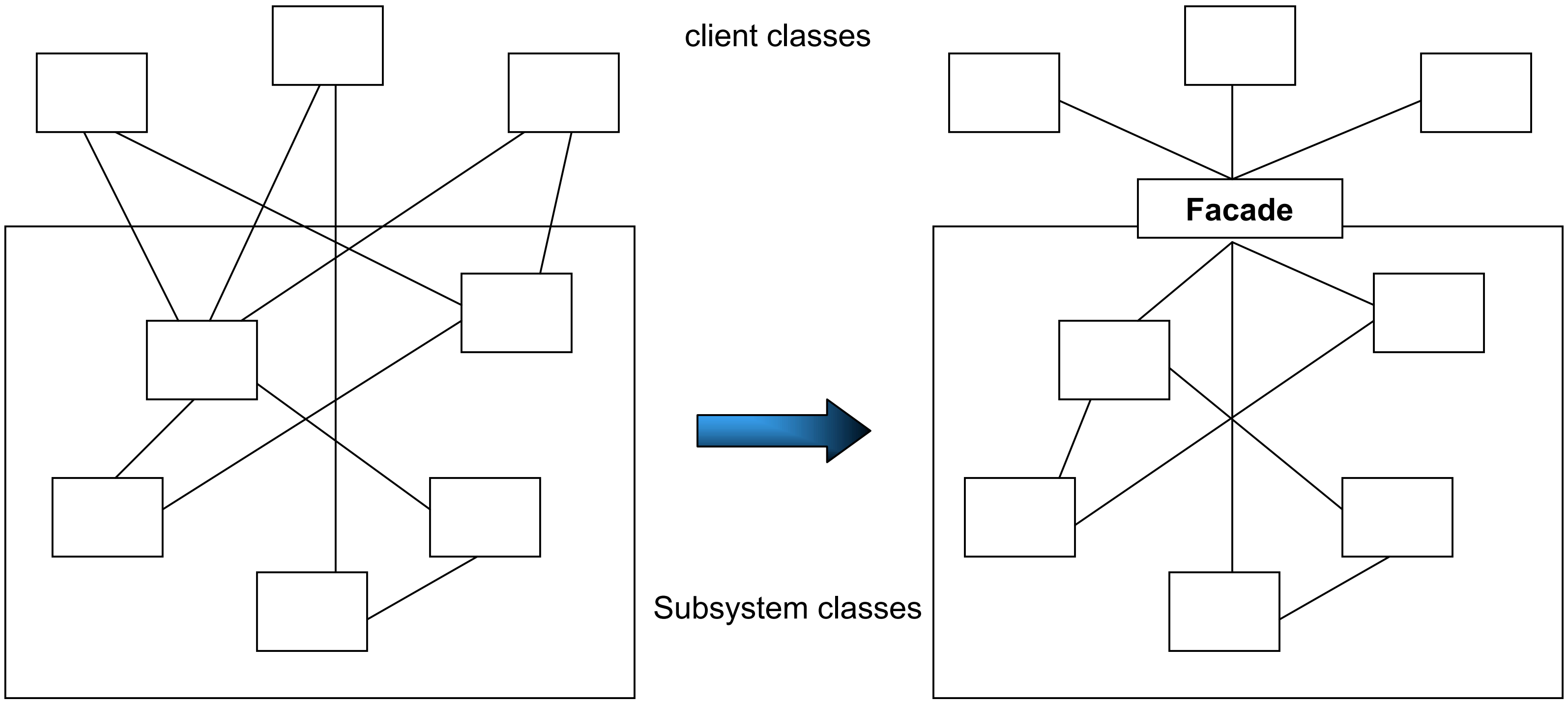
Collaborations:

- Decorator forwards requests to its Component object.
- It may optionally perform additional operations before and after forwarding the request.

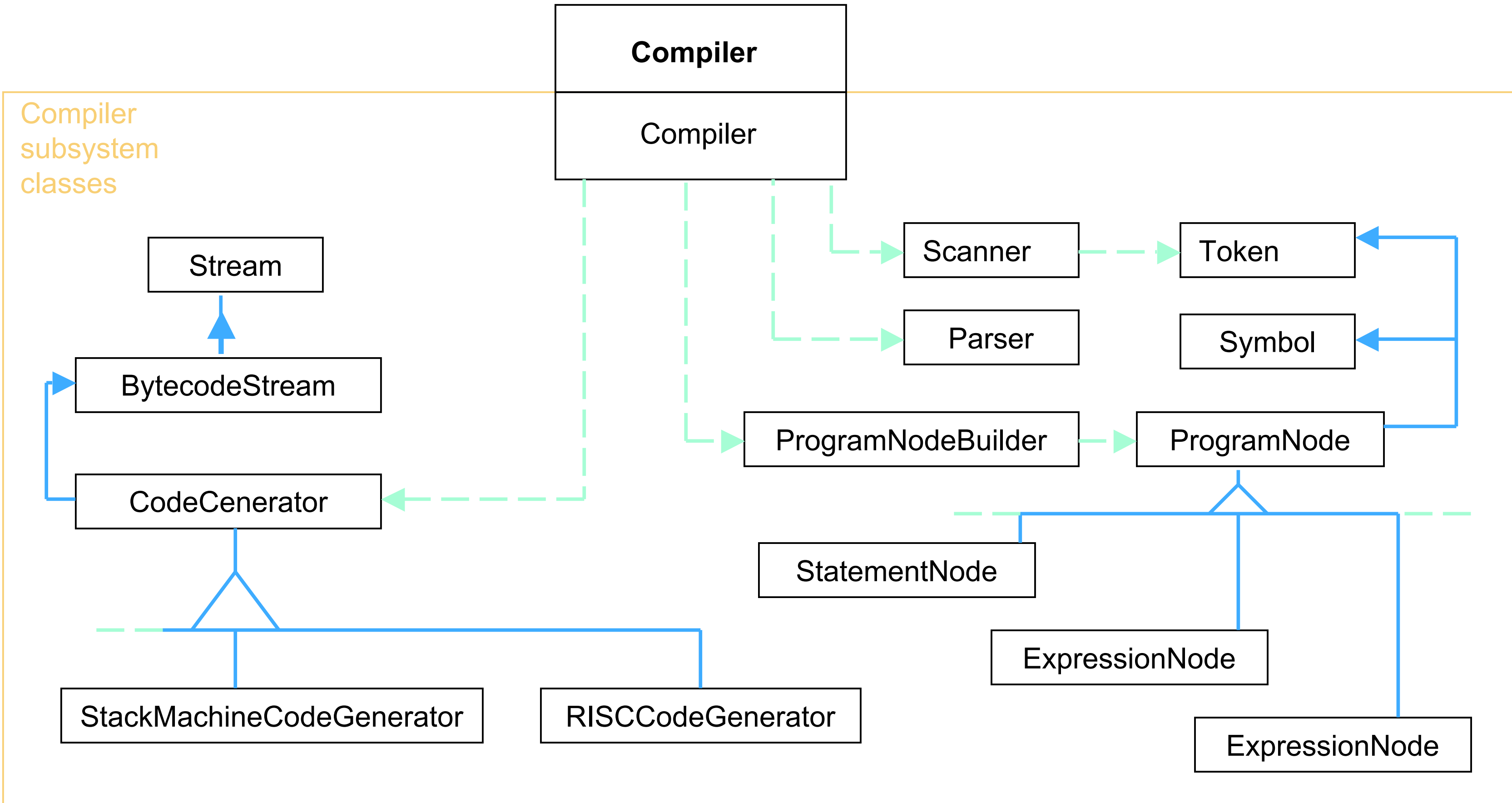
# FACADE
# (Object Structural)

- Intent:
  - Provide a unified interface to a set of interfaces in a subsystem.
  - Facade defines a 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.
  - Facade may provide a single, simplified interface to the more general facilities of a subsystem.

# FACADE - Motivation



# FACADE - Motivation

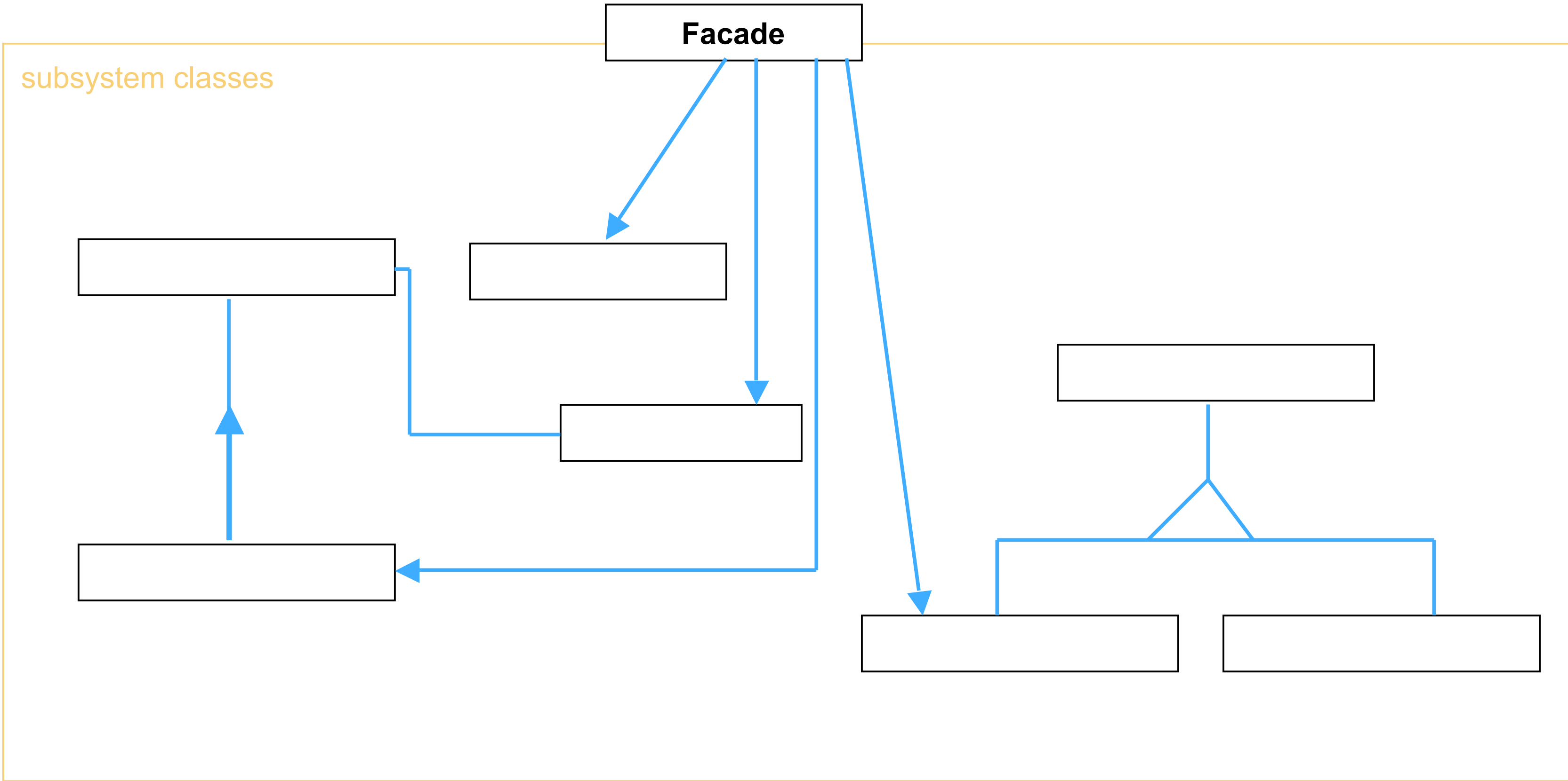


# Applicability

Use the Facade pattern:

- to provide a simple interface to a complex subsystem.
  - Subsystems often get more complex as they evolve.
- when there are many dependencies between clients and the implementation classes of an abstraction.
  - Introduce a facade to decouple the subsystems from clients and other subsystems, thereby promoting subsystem independence and portability.
- to layer subsystems.
  - Use facade to define an entry point to each subsystem level.
  - Minimize subsystem inter-dependencies

# FACADE - Structure



# 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 (i.e.; keep no references to it)

Collaborations:

– Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s).
– The facade may have to translate its interface to subsystem interfaces.
– Clients do not have to access subsystem objects directly.

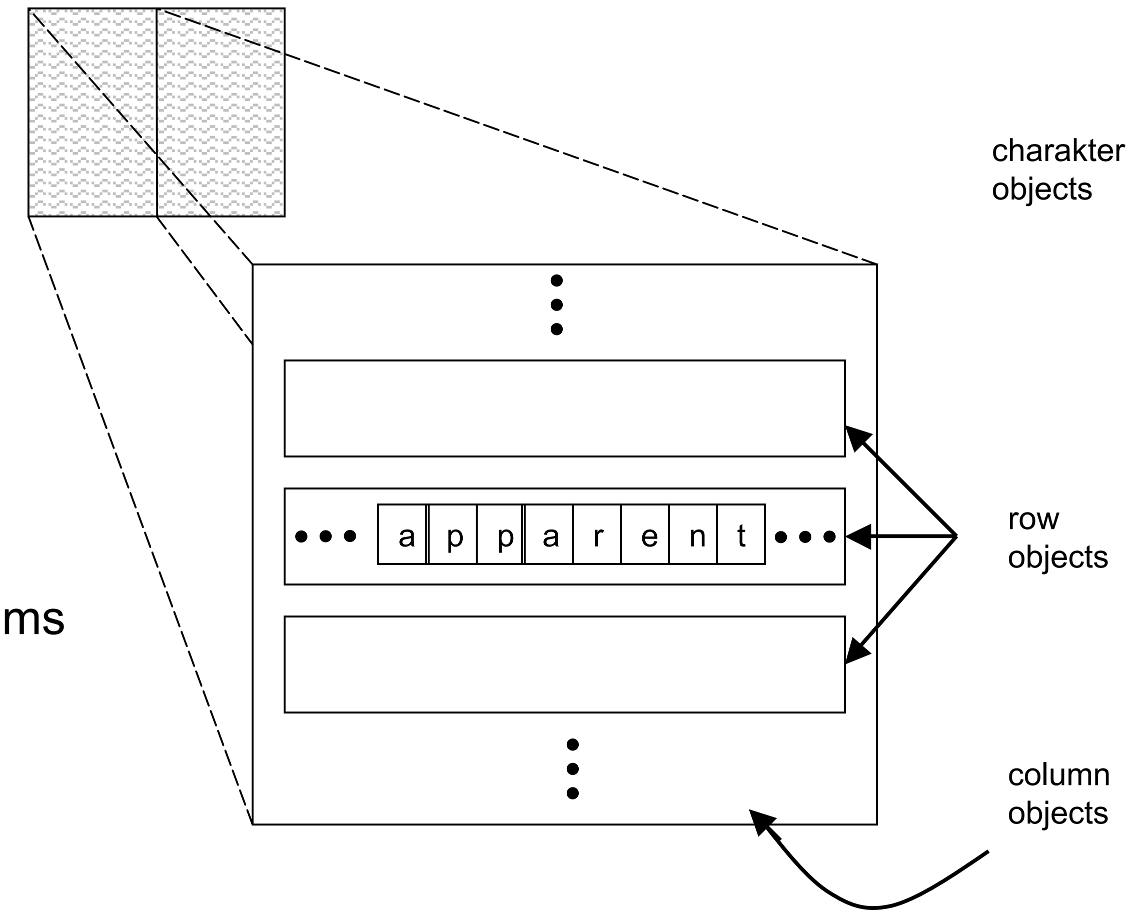
# FLYWEIGHT
# (Object Structural)

- Intent:
  - Use sharing to support large numbers of small objects efficiently.
- Motivation:
  - Some applications could benefit from using objects throughout their design, but a naïve implementation would be prohibitevly expensive

# FLYWEIGHT - Motivation

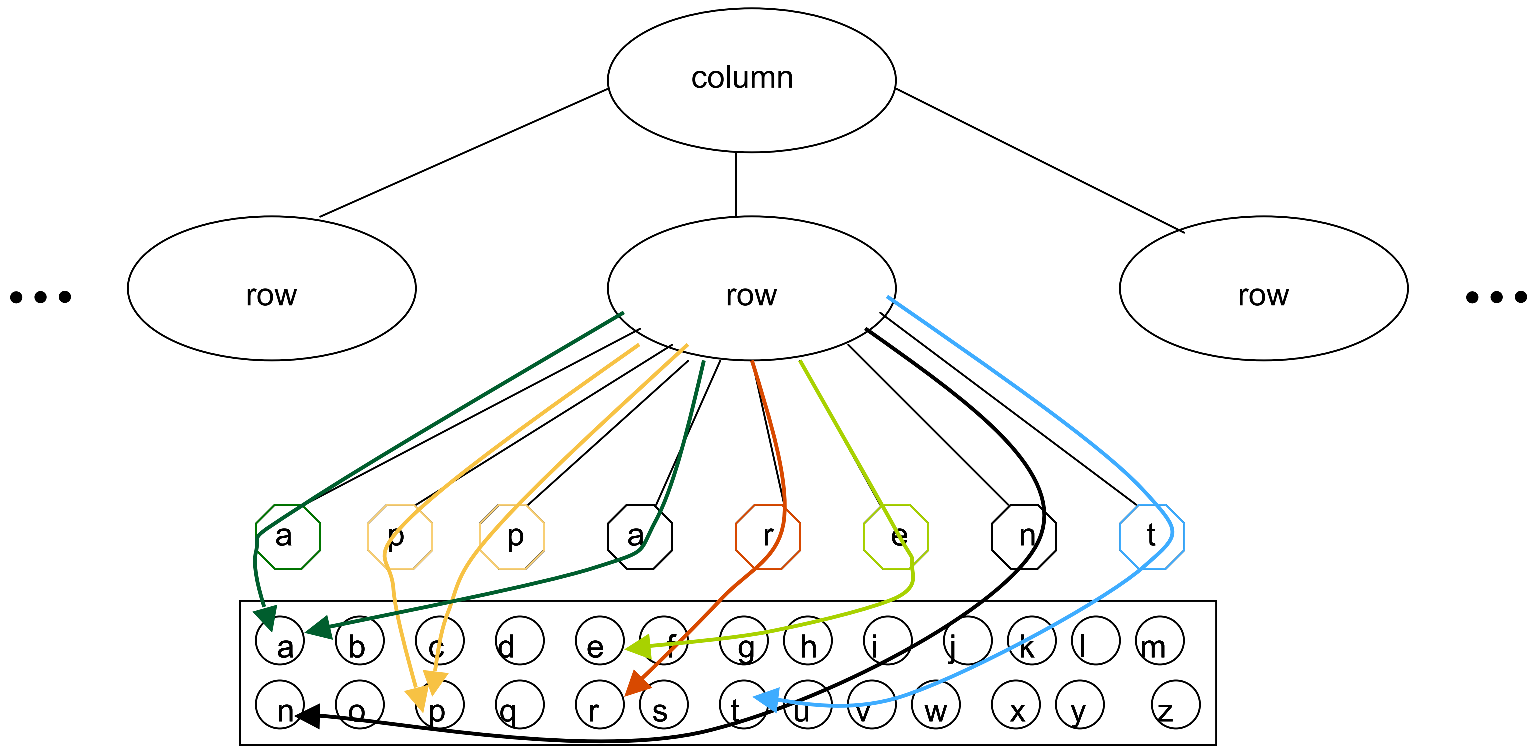
OO editors use objects to represent embedded elements like tables and figures

But treating characters uniquely (as objects) seems to be too expensive



# FLYWEIGHT - Motivation

Logically - one object per character in the document




Physically - one shared flyweight object per character

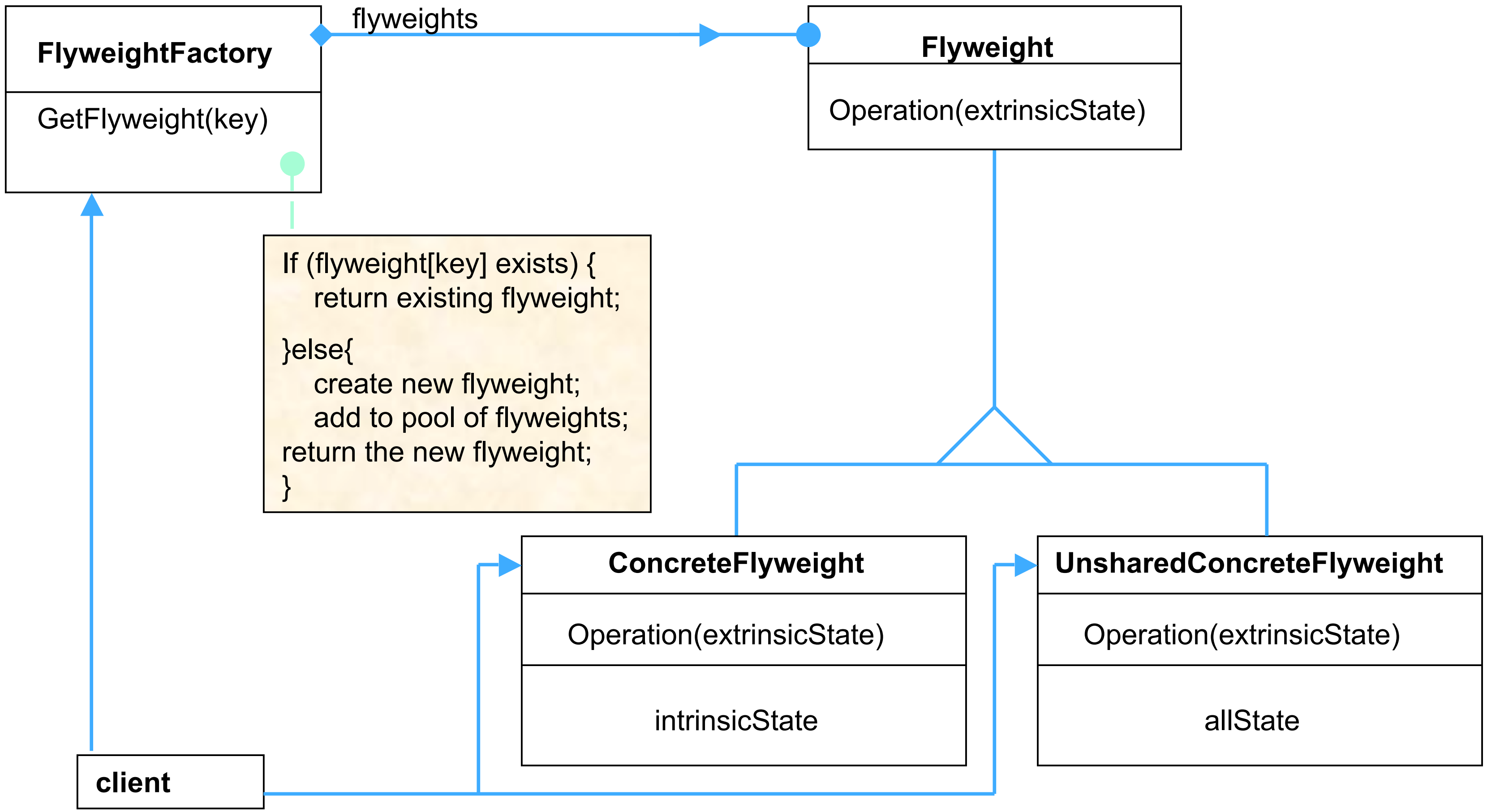
# Applicability

The Flyweight pattern‘s effectiveness depends heavily on how and where it‘s used.

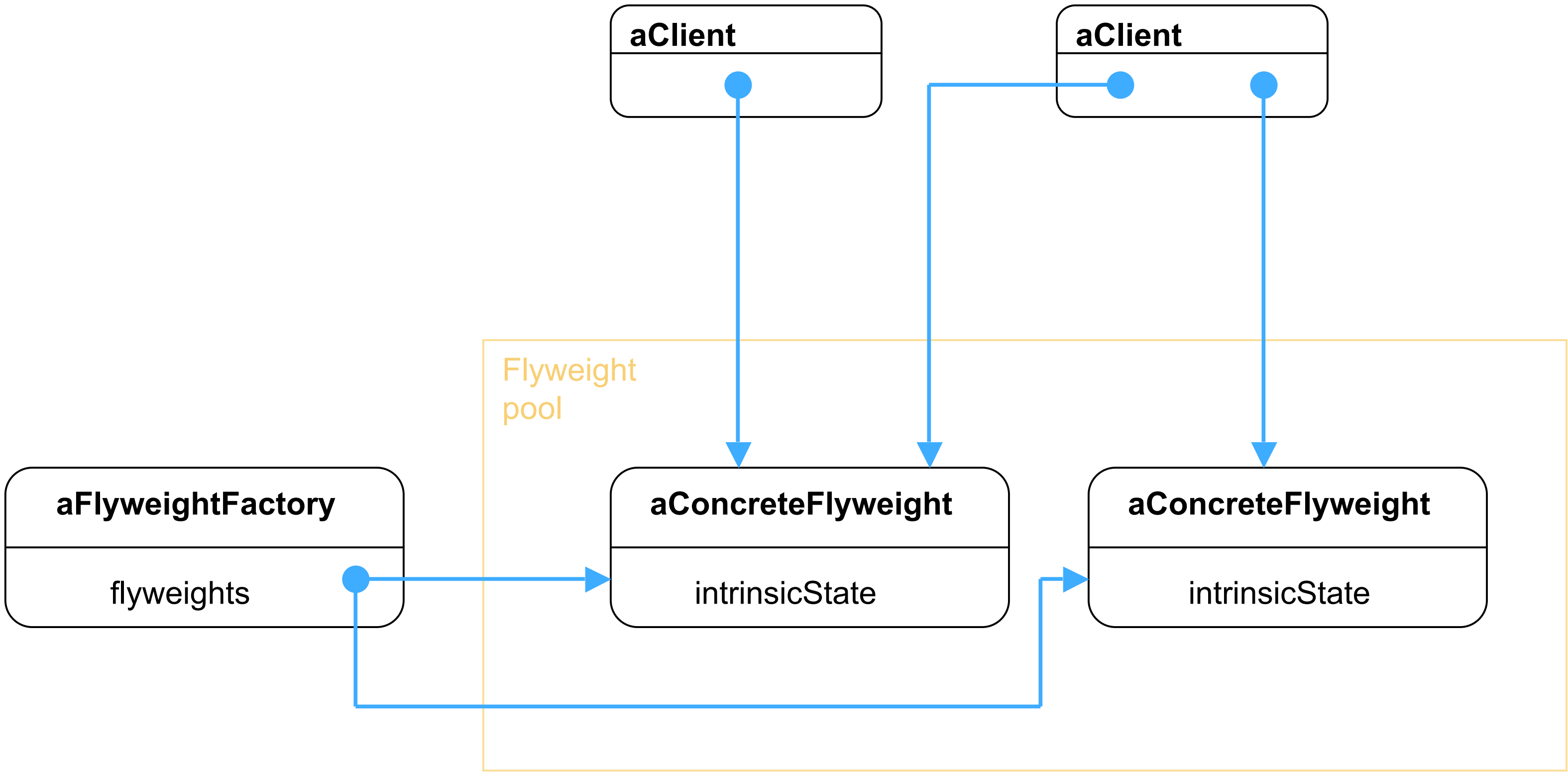
Apply the Flyweight pattern when all of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

# FLYWEIGHT - Structure



# FLYWEIGHT - Structure



# Participants

- Flyweight (Glyph)
  - Declares an interface through which flyweights can receive and act on extrinsic state
- ConcreteFlyweight (Character)
  - 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)

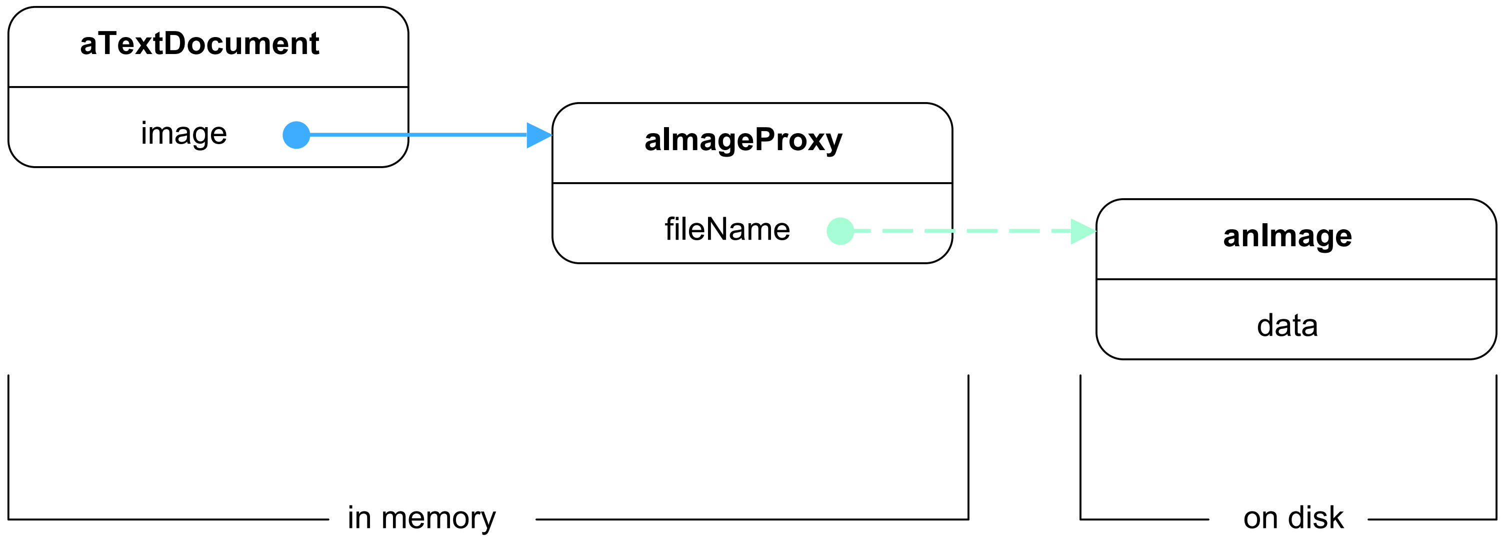
# Collaborations

- 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.
  - Clients pass this state to the flyweight when they invoke its operation.
- Clients should not instantiate ConcreteFlyweights directly.
  - Clients must obtain ConcreteFlyweights objects exclusively from the FlyweightFactory object to ensure they are shared properly.

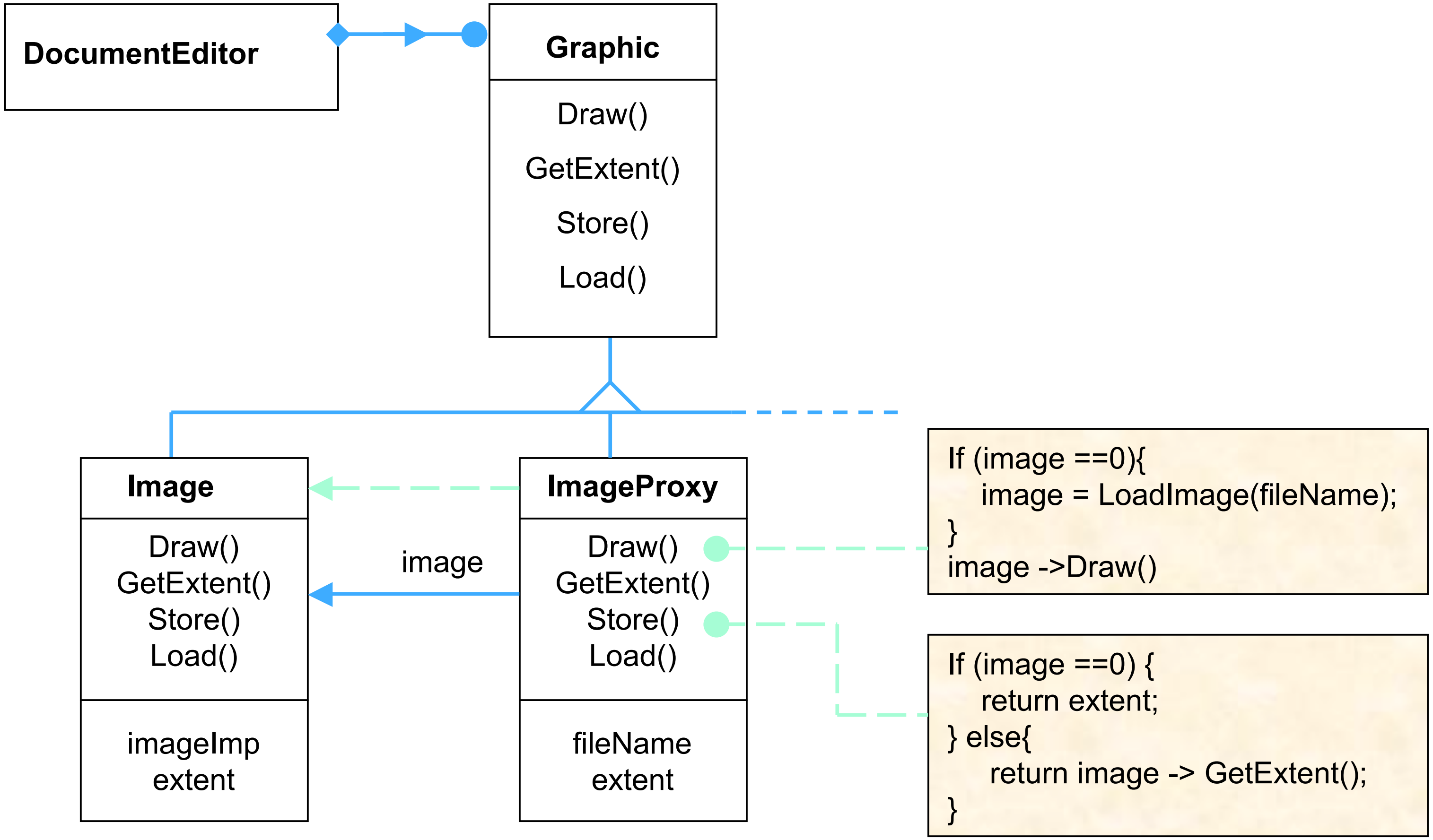
# PROXY
# (Object Structural)

- Intent:
  - Provide a surrogate or placeholder to control access another object.
- Motivation:
  - One reason for controlling access to an object is defer the full cost of its creation and initialization until we actually need to use it.
  - Consider a document editor that can embed graphical objects into an document - creation of those objects (raster images) can be expensive but opening the document should still be fast.
  - An image proxy might act as stand-in for the real image.

# PROXY - Motivation



# PROXY - Motivation



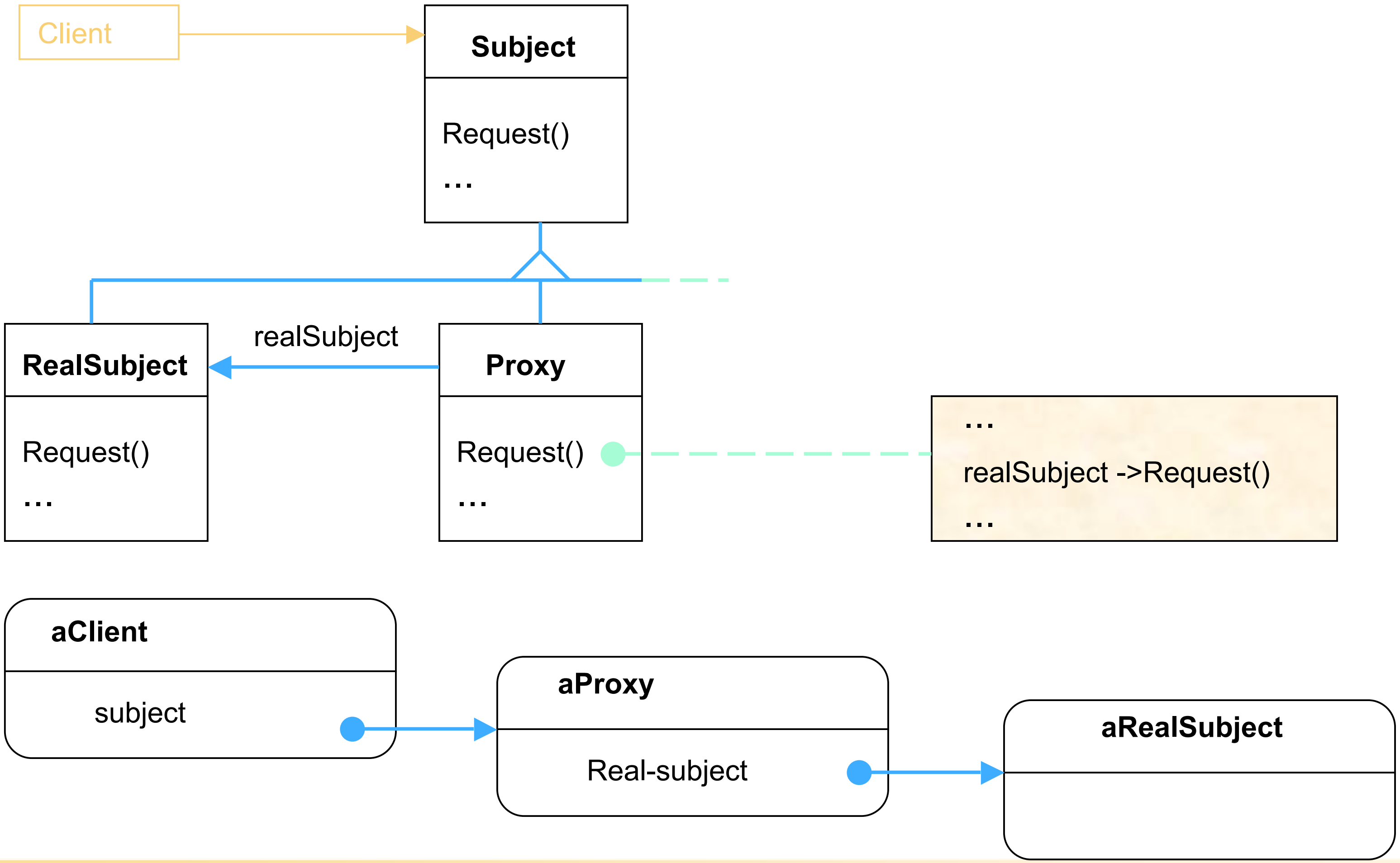
# Applicability

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.

Common situations in which the Proxy pattern is applicable:

1. A remote proxy provides a local representative for an object in a different address space. NeXTSTEP uses the class NXProxy for this purpose.
2. A virtual proxy creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.
3. A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.
   (KernelProxies in the Choices OS)
4. A smart reference is a replacement for a bare pointer that performs additional actions when an object is a accessed.

# PROXY - Structure



# Participants and Collaborations

Participants:

- Proxy (ImageProxy)
  - Maintains reference to the real subject
  - Provides interface identical to the real subject
  - Controls access to subject; 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, depending on the kind of proxy.