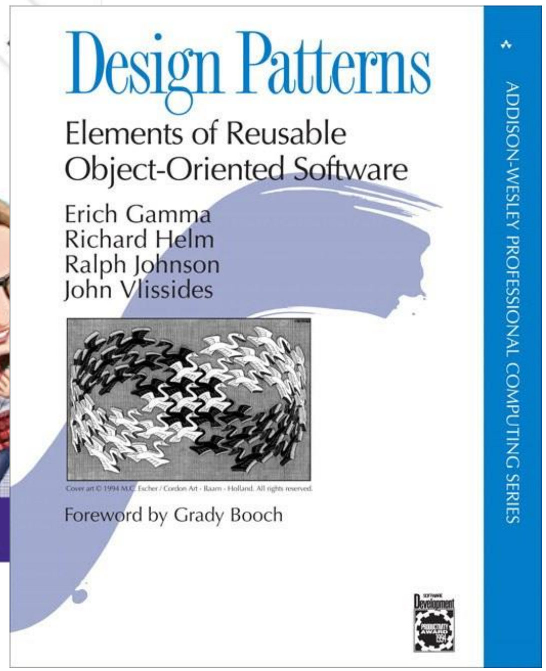
Design Pattern

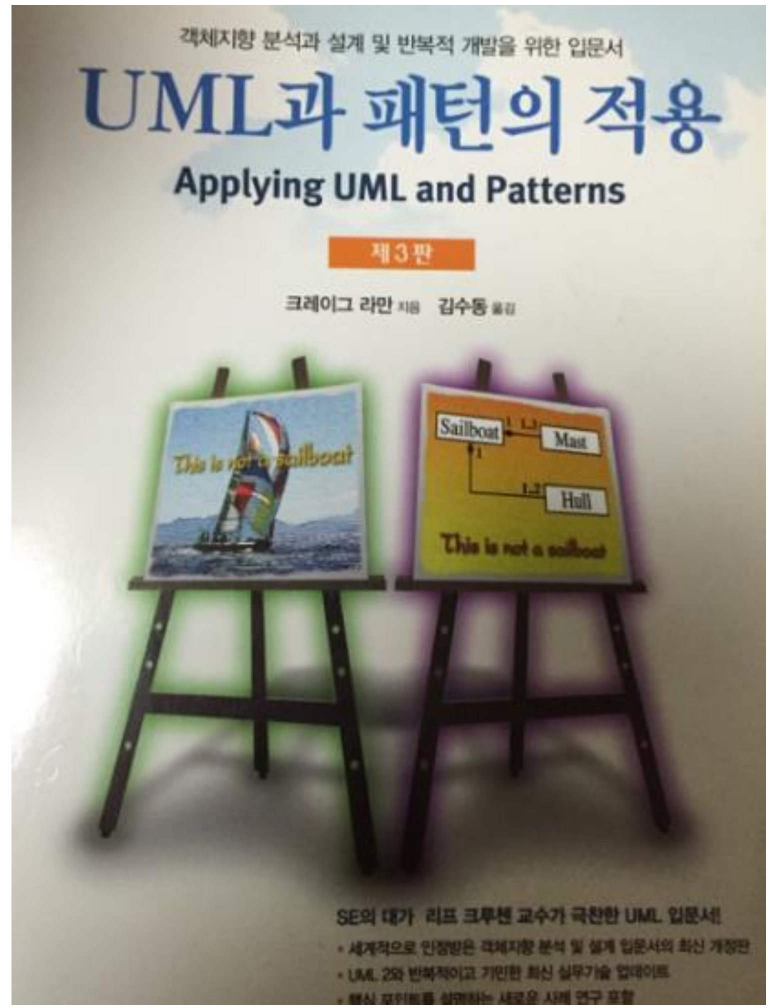
Woo Young Moon, Ph.D.
Professional Engineer Information Management
PMP, CFPS

Mobile 010-8709-1714
Wooyoungmoon@soongsil.ac.kr

References


O'REILLY
Head First
Design Patterns
Building Extensible & Maintainable Object-Oriented Software
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates
A Brain-Friendly Guide

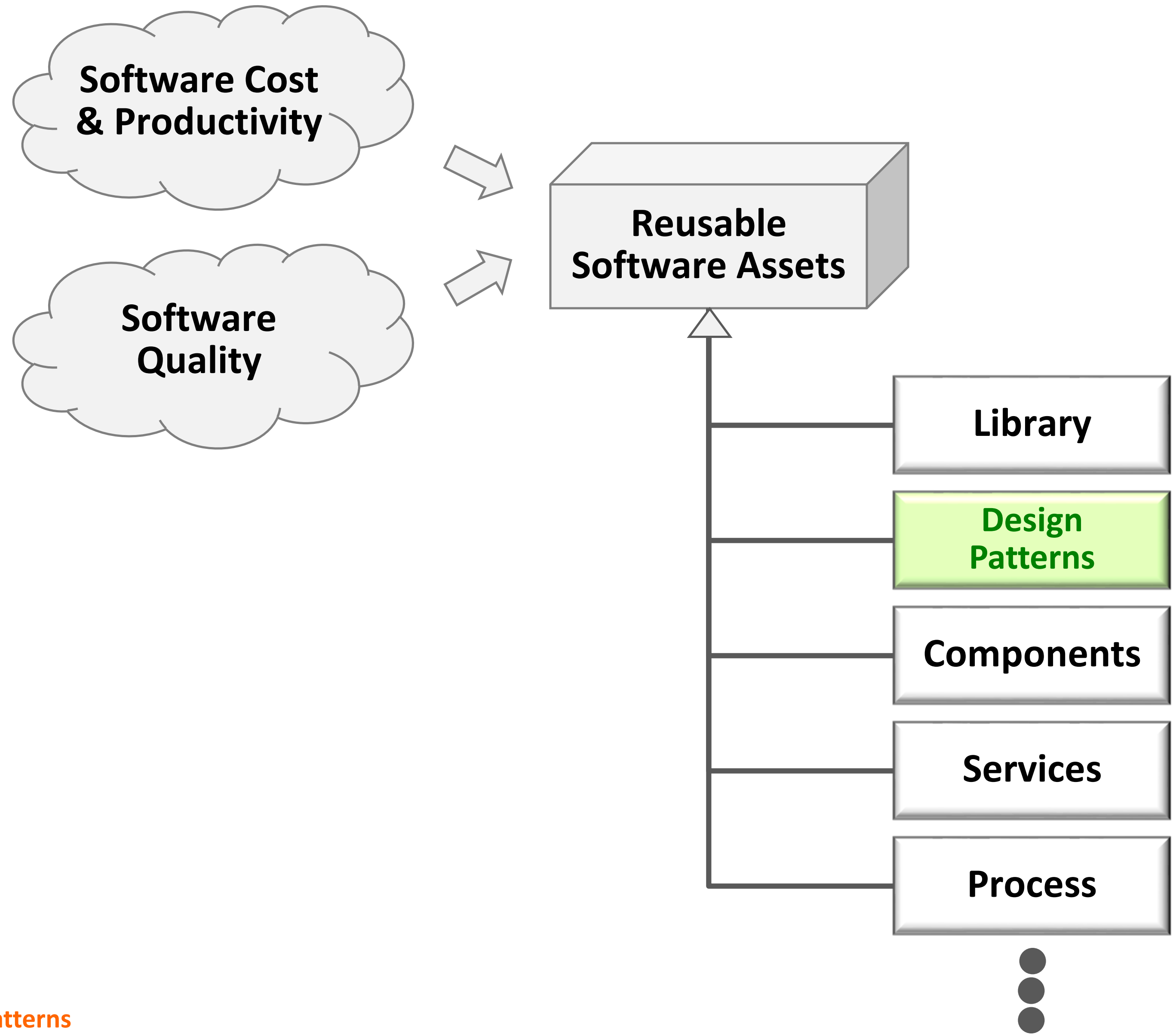

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


객체지향 분석과 설계 및 반복적 개발을 위한 입문서
UML과 패턴의 적용
Applying UML and Patterns
제3판
크레이그 라만 지음 김수동 옮김
This is not a sailboat
Sailboat
Mast
Hull
This is not a sailboat

Contents

Unit 1.
Principles of Design Patterns

Motivation


Software Cost
& Productivity
Software
Quality
Reusable
Software Assets
Library
Design
Patterns
Components
Services
Process

Design Patterns

- **To represent solutions to problems that arise when developing software within a particular context**
  - Pattern = Problem/Solution pair in a Context
- **To capture the static and dynamic structure and collaboration among key participants in software designs**
- **To facilitate reuse of successful software architectures and designs**

Classifications of Design Patterns

- **Creational Patterns**
  - Deal with initializing and configuring classes and objects
- **Structural Patterns**
  - Deal with decoupling interface and implementation of classes and Objects
- **Behavioral Patterns**
  - Deal with dynamic interactions among societies of classes and objects

5 Creational Patterns

- **Factory Method**
- **Abstract Factory**
- **Builder**
- **Prototype**
- **Singleton**

7 Structural Patterns

- **Adapter**
- **Bridge**
- **Composite**
- **Decorator**
- **Façade**
- **Flyweight**
- **Proxy**

11 Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Principles of Design Patterns

- **Design patterns are devised with 3 principles.**
- **Principle 1**
  - Separate interface from implementation
- **Principle 2**
  - Allow substitution of variable implementations via a common interface.
- **Principle 3**
  - Determine what is common and what is variable with an interface and an implementation
    - Common ⇔ Stable
    - Variable ⇔ Unstable, To be resolved
  - Open Closed Principle (OCP)

Open/Closed Principle

- **Determining Common vs. Variable Features**
  - Insufficient variation makes it hard for users to customize applications.
- **Components should be:**
  - The design of variable features should be <u>open</u> for customization and extension.
  - The design of common features should be <u>closed</u> for modification.
    - Cannot be modified.

Benefits of Design Patterns

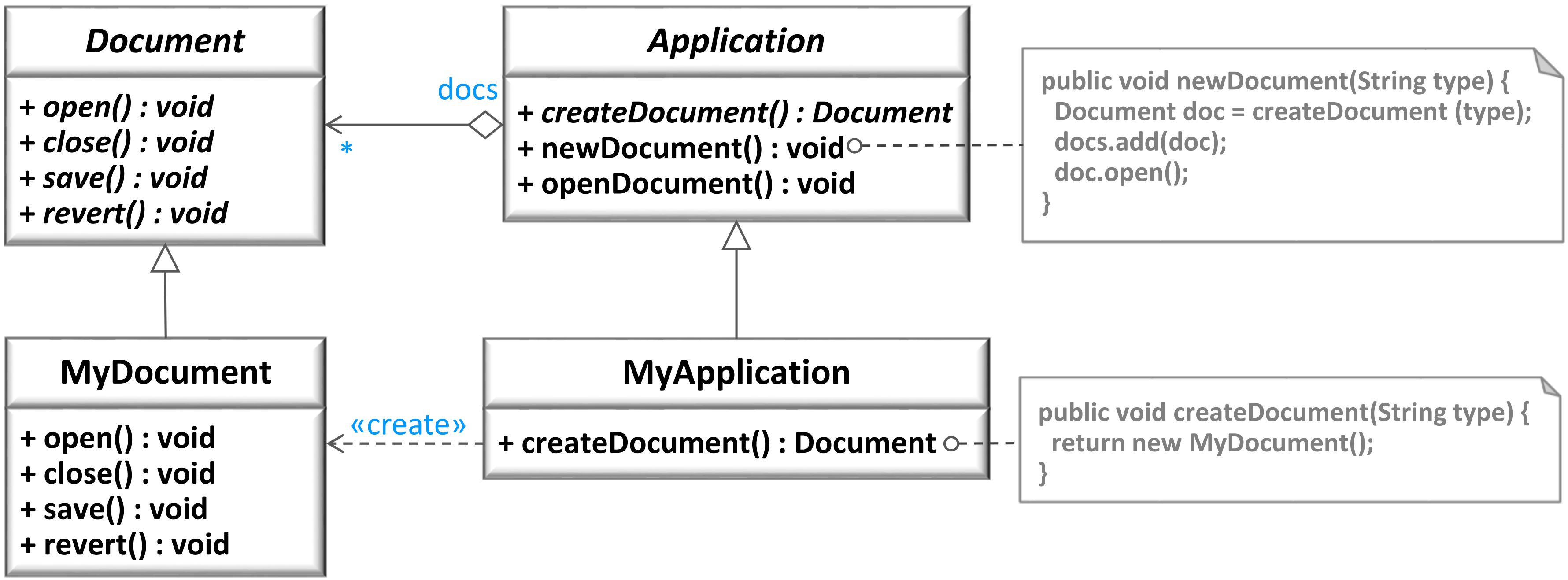
- **Utilizing *expert knowledge* embedded on design patterns**
- **Promoting *effective communication* among developers**
- **Assisting better quality object-oriented design with**
  - High Modularity
  - High Readability
  - High Modifiability
  - High Extendibility

Unit 2.
Factory Method

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.**

- **Consider the following framework:**

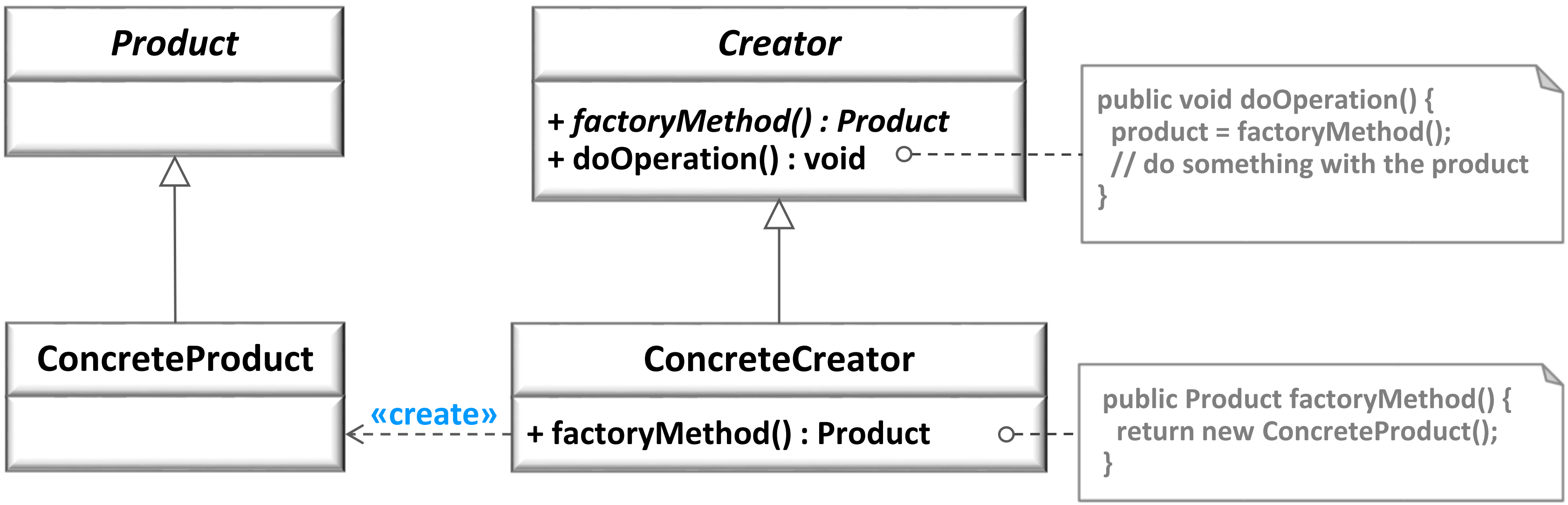


- **The createDocument( ) method is a factory method.**

Applicability

- **Use the Factory Method pattern in any of the following situations:**
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates

Structure



Product
Creator
+ factoryMethod() : Product
+ doOperation() : void
public void doOperation() {
product = factoryMethod();
// do something with the product
}
ConcreteProduct
«create»
ConcreteCreator
+ factoryMethod() : Product
public Product factoryMethod() {
return new ConcreteProduct();
}

Participants

- **Product**
  - To define the interface for the type of objects the factory method creates
- **ConcreteProduct**
  - To implement the Product interface
- **Creator**
  - To declare the factory method, which returns an object of type Product
- **ConcreteCreator**
  - To override the factory method to return an instance of a ConcreteProduct

Collaborations

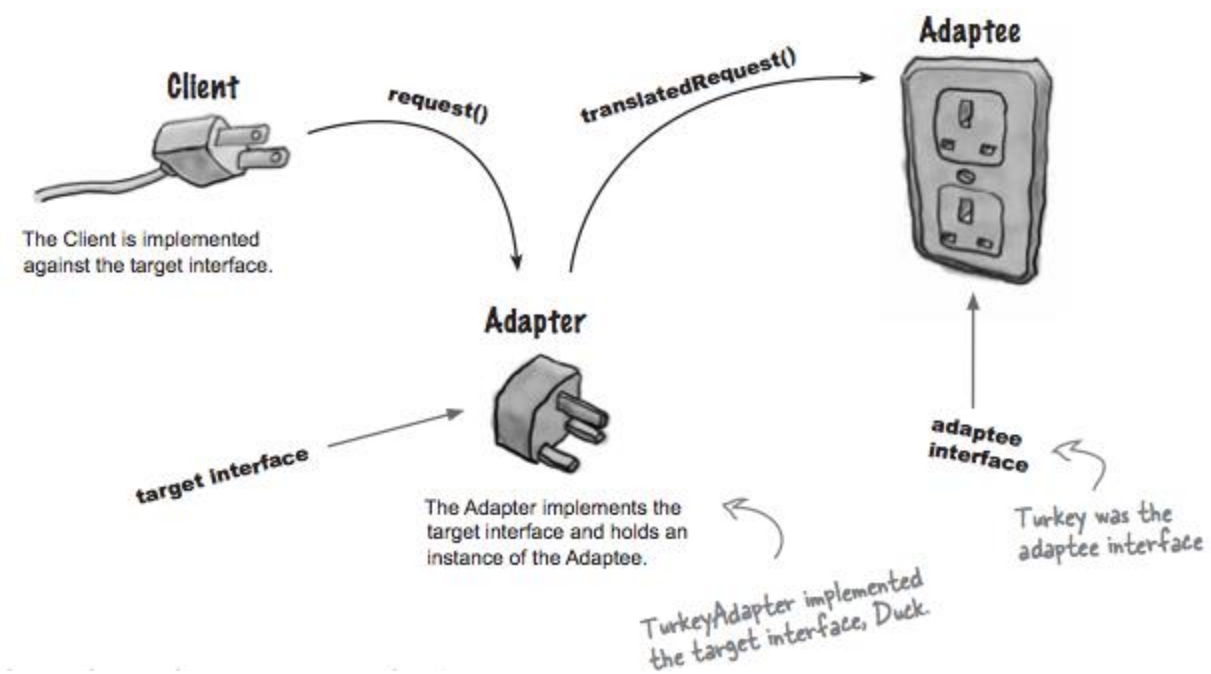
- **Creator relies on its subclasses to implement the factory method so that it returns an instance of the appropriate ConcreteProduct.**

Consequences

- **Advantages**
  - Code is made more flexible and reusable by the elimination of instantiation of application-specific classes.
  - Code deals only with the interface of the Product class and can work with any ConcreteProduct class that supports this interface.
- **Disadvantages**
  - Clients might have to subclass the Creator class just to instantiate a particular ConcreteProduct.

Unit 3.
Adapter

- **Convert the interface of a class into another interface clients expect.**
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

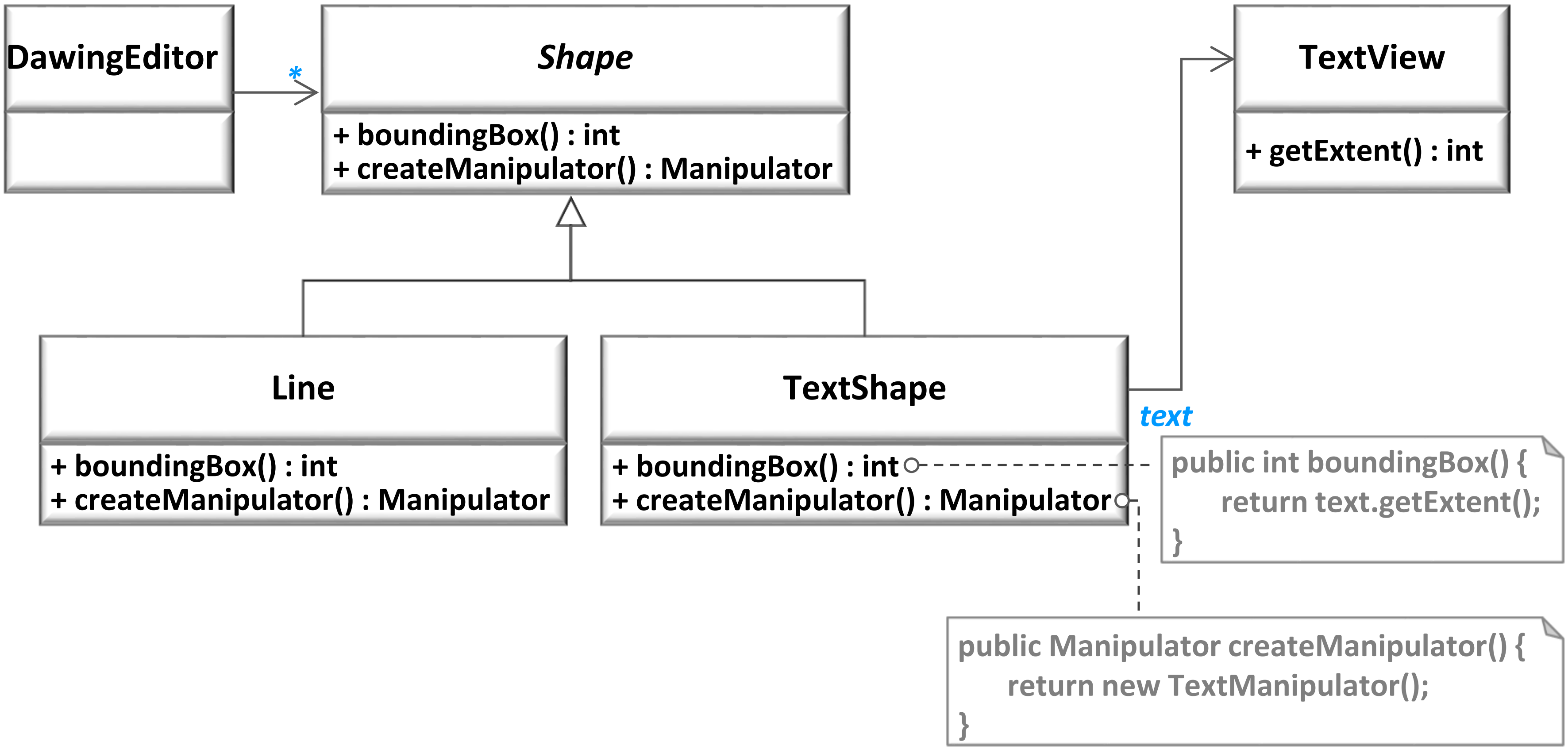


Motivation (1)

- **Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application.**
  - We can not change the library interface, since we may not have its source code.
  - Even if we did have the source code, we probably should not change the library for each domain-specific application.
- **Two Approaches**
  - Class Adapter
    - Inherit an adapter and an adaptee.
  - Object Adapter
    - Compose an adaptee instance within an dapter and implement the adapter in terms of the adaptee’s interface.

Motivation (2)

- A solution using an object adapter:



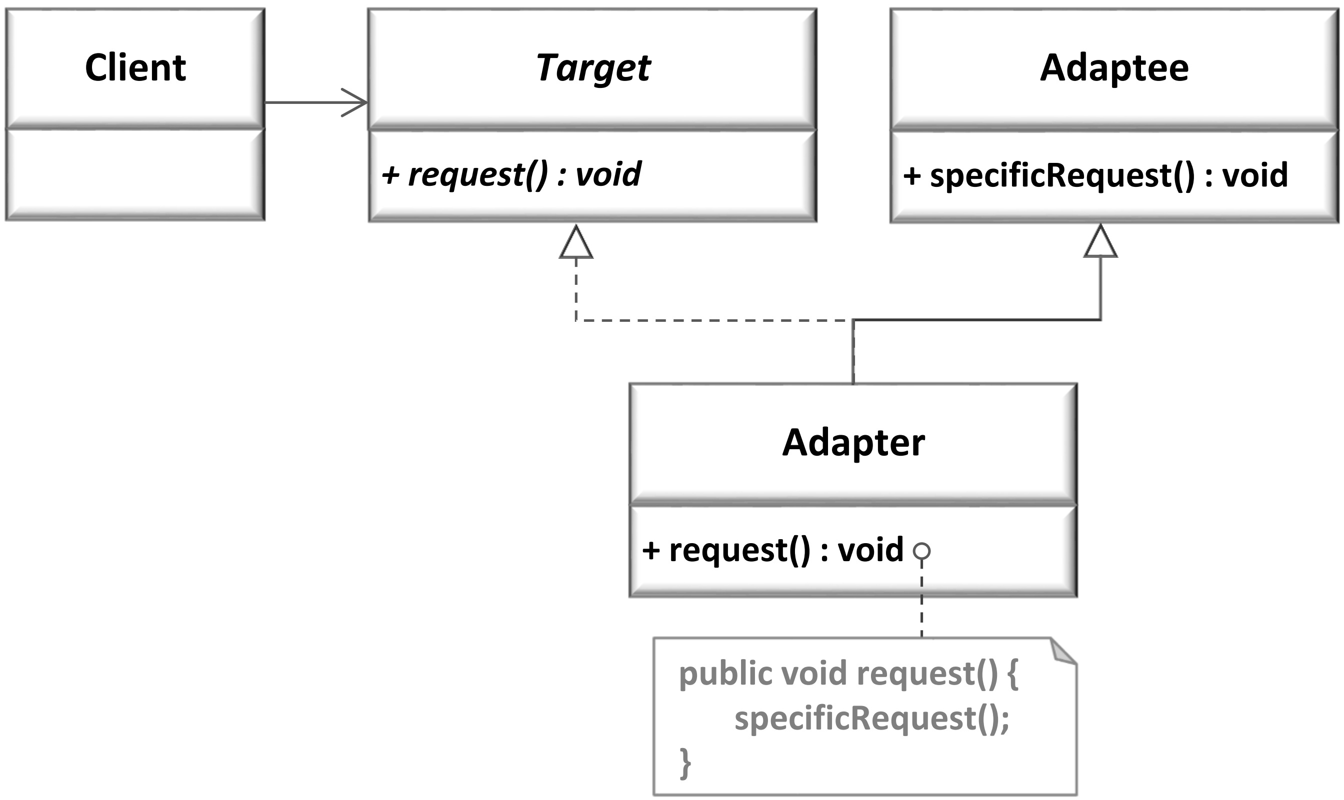
boundingBox() in TextShape are incompatible with getExtent() in TextView.

Applicability

- **Use the Adapter pattern when:**
  - To use an existing class, and its interface does not match the one you need
  - To create a reusable class that cooperates with unrelated classes with incompatible interfaces
  - To use several existing subclasses, but it's impractical to adapt interface by subclassing every one
    - Only for object adapters

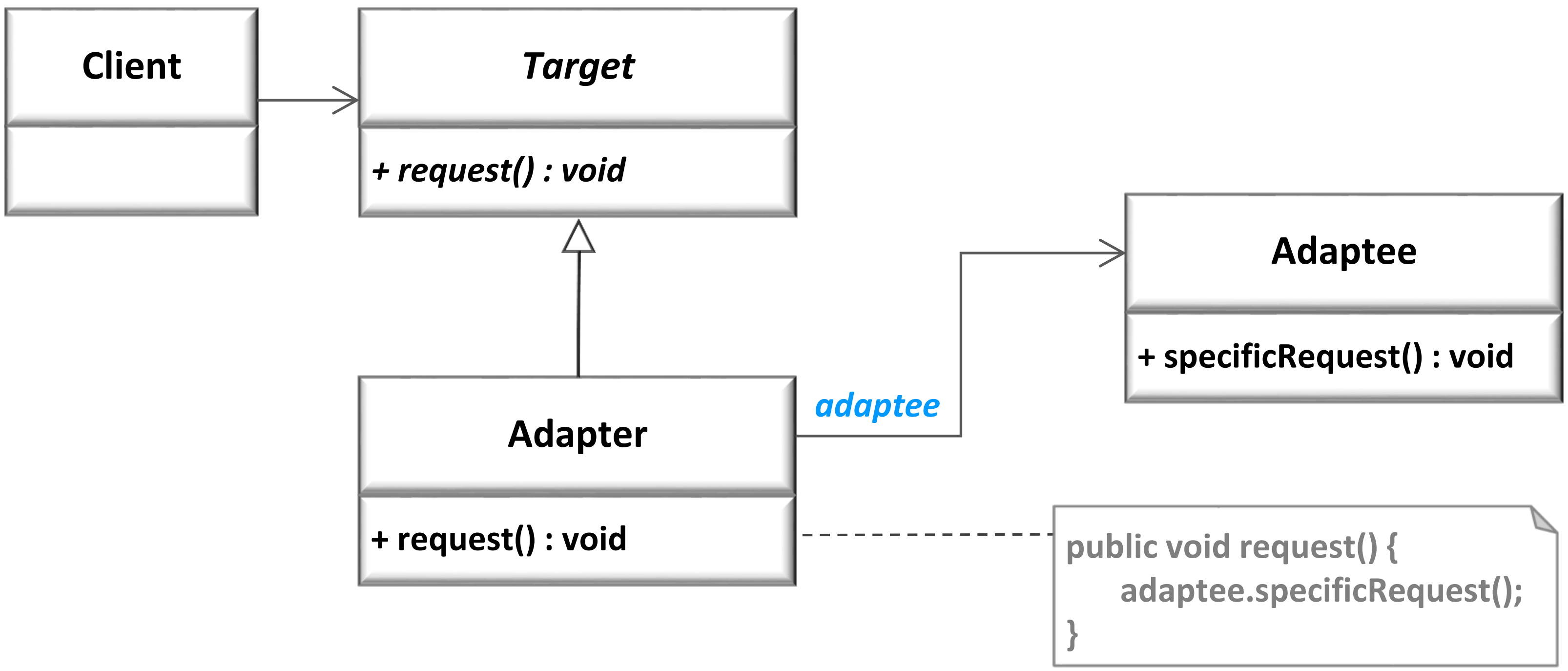
Structure: Class Adapter

- A class adapter uses multiple inheritance to adapt one interface to another:



Structure: Object Adapter

- **An object adapter relies on object composition:**



Participants

- **Target (Shape)**
  - To define the domain-specific interface that Client uses
- **Client (DrawingEditor)**
  - To collaborate with objects conforming to the Target interface
- **Adaptee (TextView)**
  - To define an existing interface that needs adapting
- **Adapter (TextShape)**
  - To adapt the interface of Adaptee to the Target interface

Collaborations

- **Clients call operations on an Adapter instance.**
- **The adapter calls Adaptee operations that carry out the request.**

Consequences: Class Adapter

- **Advantages**
  - Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
  - Introduces only one object, and no additional pointer indirection is needed to get to the adaptee.
- **Disadvantages**
  - Will not work to adapt a class and all its subclasses.

Consequences: Object Adapter

- **Advantages**
  - Lets a single Adapter work with many Adaptees.
    - Can also add functionality to all Adaptees at once.
- **Disadvantages**
  - Makes it harder to override Adaptee behavior.
    - Requires subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

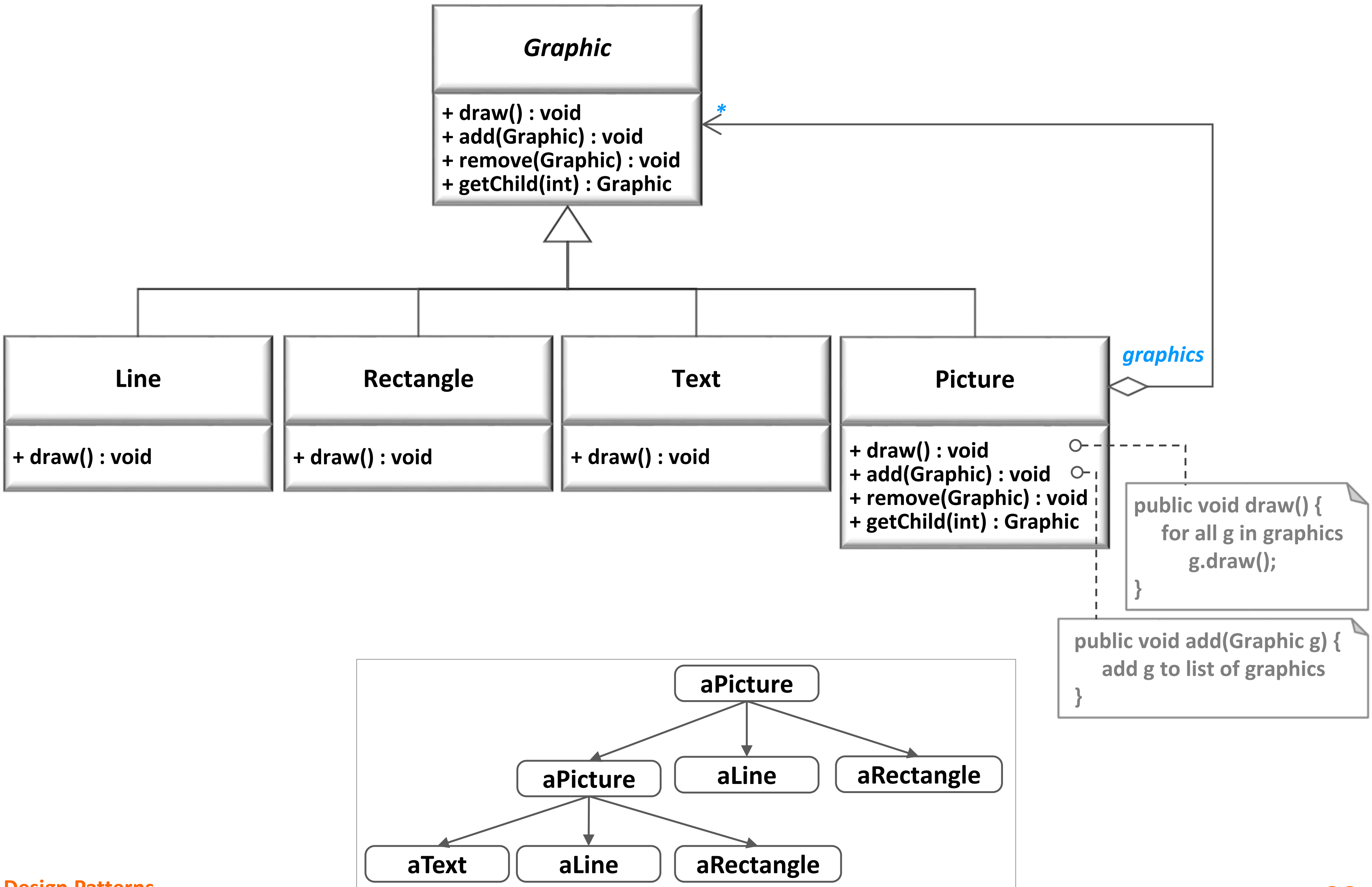
Unit 4. Composite

Intent

- **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.

Motivation (1)

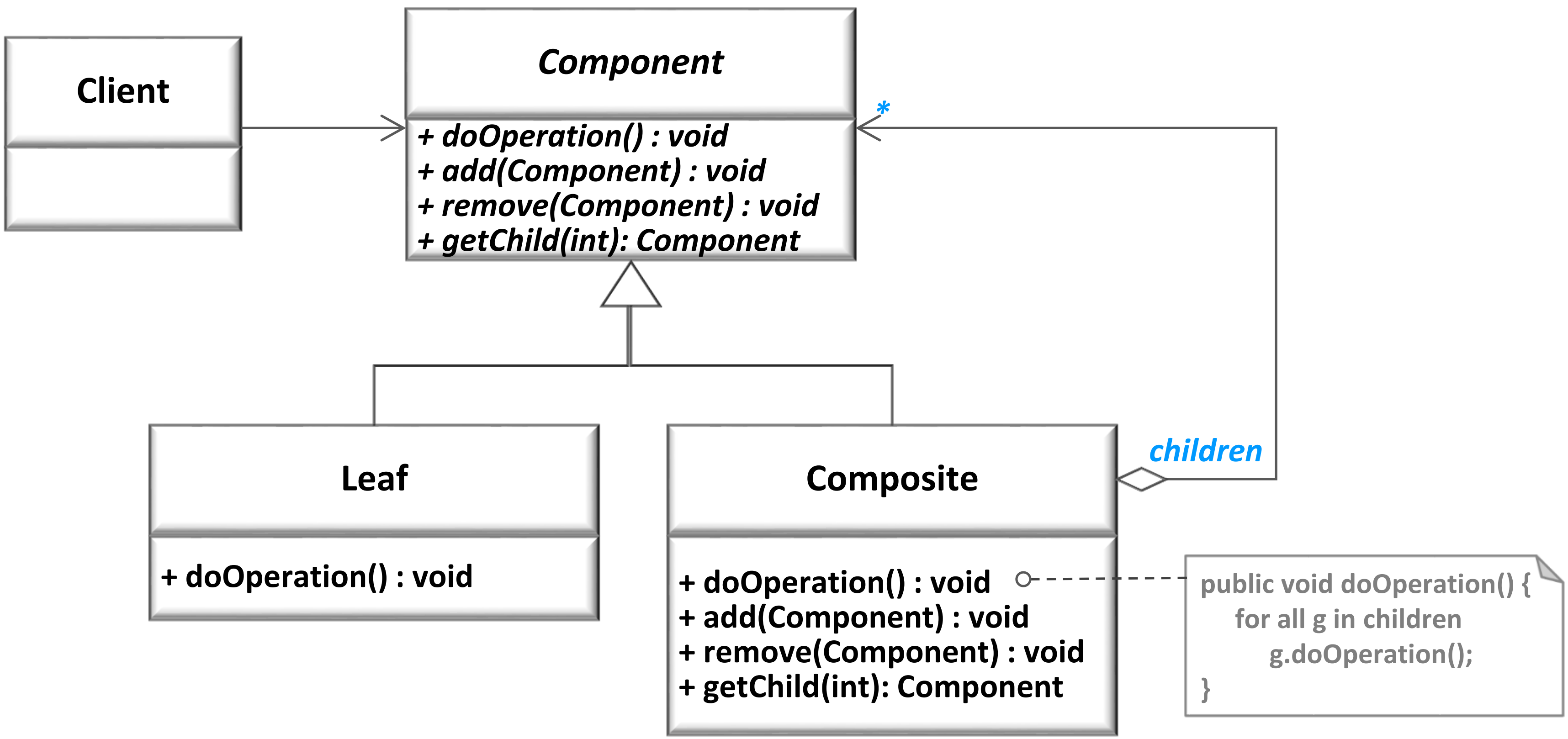
- **Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components.**
  - The user can group components to form larger components, which in turn can be grouped to form still larger components.
- **Problem**
  - Code that uses these classes must treat primitive and container objects differently.
    - Even if most of the time the user treats them identically.
    - Having to distinguish these objects make the application more complex.

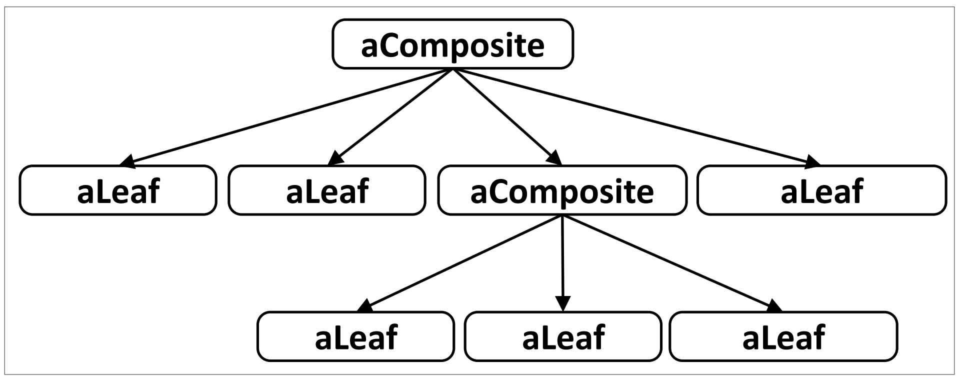

Graphic
+ draw() : void
+ add(Graphic) : void
+ remove(Graphic) : void
+ getChild(int) : Graphic
*
graphics
Line
+ draw() : void
Rectangle
+ draw() : void
Text
+ draw() : void
Picture
+ draw() : void
+ add(Graphic) : void
+ remove(Graphic) : void
+ getChild(int) : Graphic
public void draw() {
for all g in graphics
g.draw();
}
public void add(Graphic g) {
add g to list of graphics
}
aPicture
aPicture
aLine
aRectangle
aText
aLine
aRectangle

Applicability

- **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.
    - Clients will treat all objects in the composite structure <u>uniformly</u>.

Structure


Client
Component
+ doOperation() : void
+ add(Component) : void
+ remove(Component) : void
+ getChild(int): Component
*
Leaf
+ doOperation() : void
Composite
+ doOperation() : void
+ add(Component) : void
+ remove(Component) : void
+ getChild(int): Component
children
public void doOperation() {
for all g in children
g.doOperation();
}


aComposite
aLeaf
aLeaf
aComposite
aLeaf
aLeaf
aLeaf
aLeaf

Participants (1)

- **Component (Graphic)**
  - To declare the interface for objects in the composition
  - To implement default behavior for the interface common to all classes, as appropriate
  - To declare an interface for accessing and managing its child components
  - (optional) To defines an interface for accessing a component's parent in the recursive structure, and implement it if that's appropriate
- **Leaf (Rectangle, Line, Text, etc.)**
  - To represent leaf objects in the composition
    - A leaf has no children.
  - To define behavior for primitive objects in the composition

Participants (2)

- **Composite (Picture)**
  - To define behavior for components having children
  - To store child components
  - To implement child-related operations in the Component interface
- **Client**
  - To manipulate objects in the composition through the 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.**

Consequences

- **Advantages**
  - To be able to define class hierarchies consisting of primitive objects and composite objects
  - To make clients simpler, since they do not have to know if they are dealing with a leaf or a composite component
  - To make it easy to add new kinds of components
- **Disadvantages**
  - To make it harder to restrict the type of components of a composite

Unit 5.
Facade

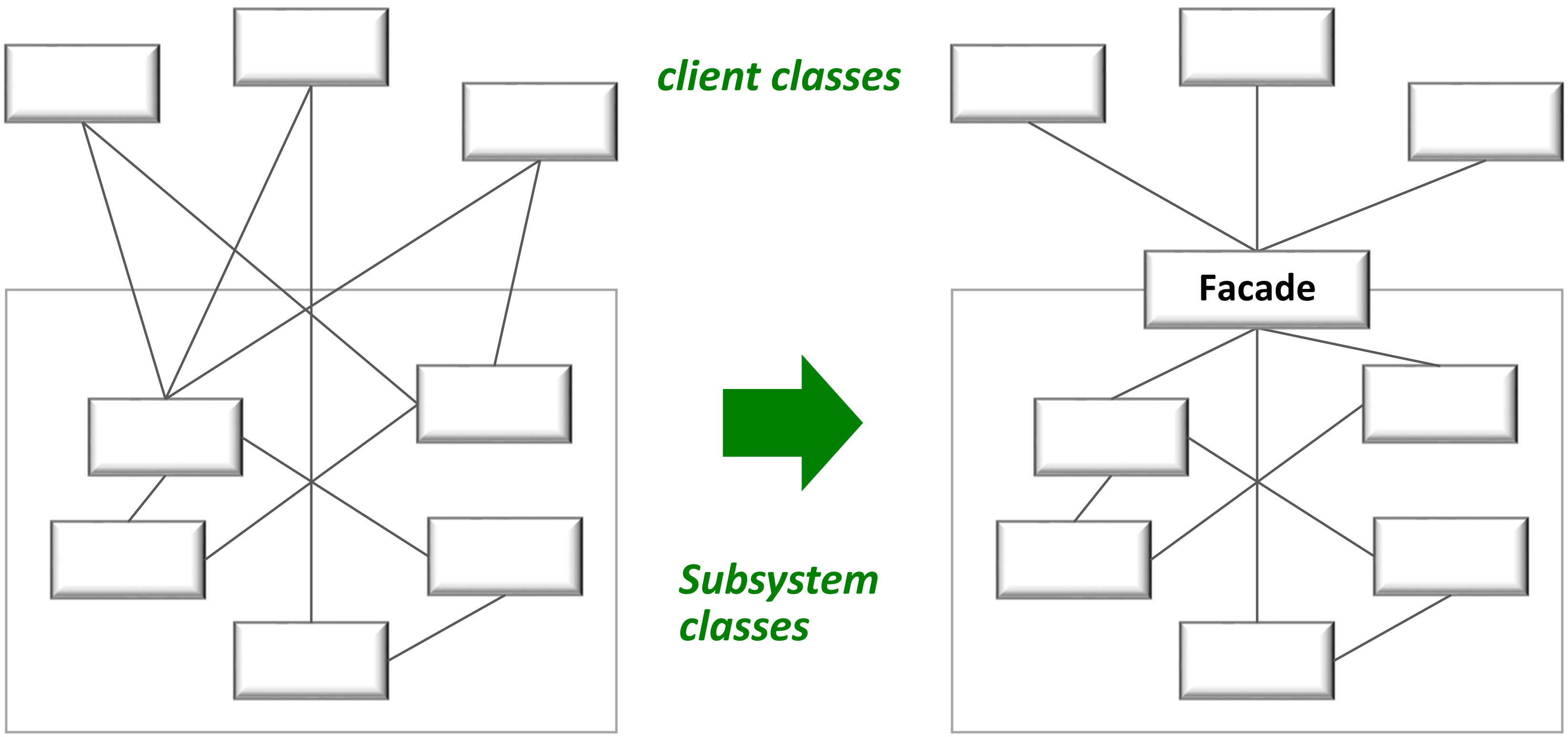
Intent

- To provide a unified interface for a set of interfaces in a subsystem
- To define a higher-level interface that makes the subsystem easier to use

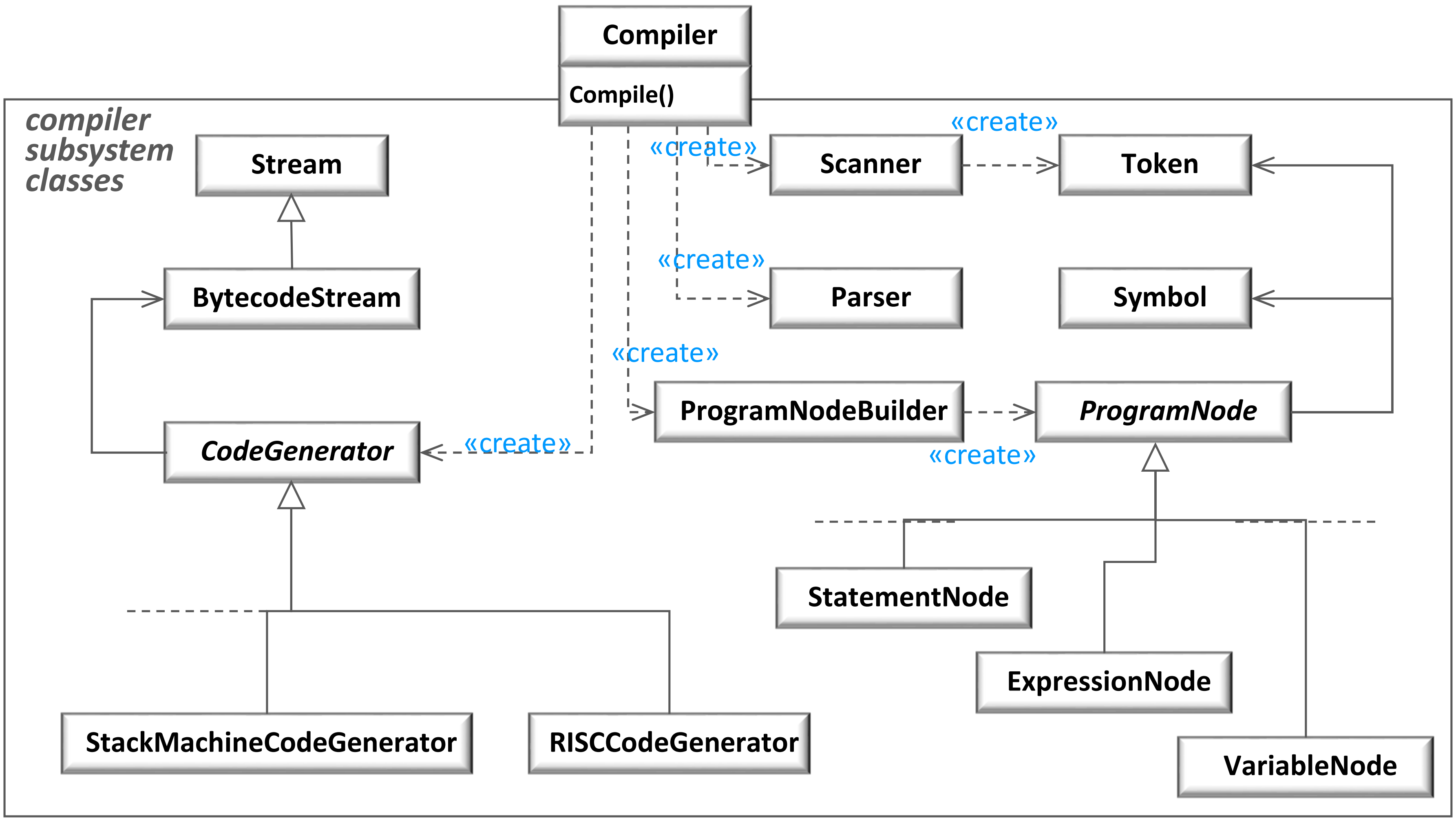
Motivation (1)

- **Structuring a system into subsystems helps reduce complexity.**
  - To minimize the communication and dependencies between subsystems
- **One way to reduce complexity**
  - To introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem

Motivation (2)



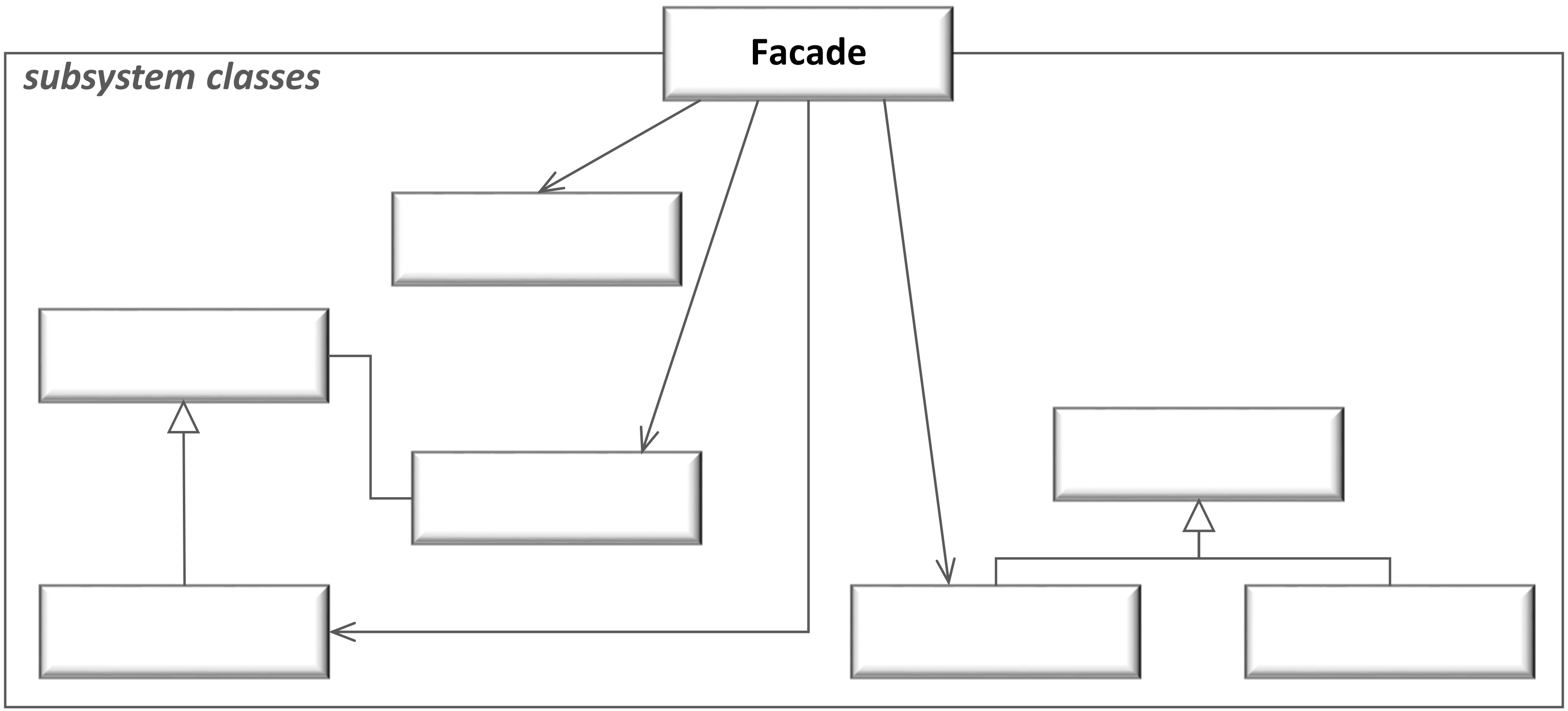
- **Example of Compiler Subsystem**



Applicability

- **Applicable Situations:**
  - To provide a simple interface for a complex subsystem
    - Providing a simple default view of the subsystem that is good enough for most clients
    - Only clients needing more customizability will need to look beyond the façade.
  - To decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability
  - To layer the subsystems
    - Using a façade to define an entry point to each subsystem level
    - If subsystems are dependent, then the dependencies between subsystems can be simplified by making them communicate with each other solely through their façades.

Structure



Participants

- **Facade (Compiler)**
  - To know which subsystem classes are responsible for a request
  - To delegate client requests to appropriate subsystem objects
- **Subsystem classes (Scanner, Parser, ProgramNode, etc.)**
  - To Implement subsystem functionality
  - To handle work assigned by the Façade object
  - To have no knowledge of the façade;
    - Subsystem classes keep no references to the façade

Collaborations

- **Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem object(s).**
  - Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- **Clients that use the facade don't have to access its subsystem objects directly.**

Consequences

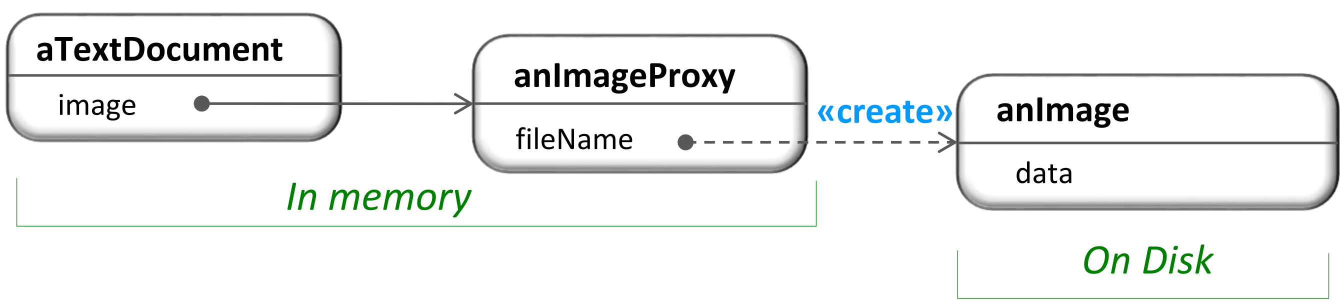
- **Advantages**
  - To hide implementations of a subsystem from its clients
    - Reducing the number of objects that clients deal with
    - Making the subsystem easier to use
  - To promote weak coupling between the subsystem and its clients
    - To allow changing the subsystem classes without affecting its clients.
  - To layer a system and the dependencies between objects
    - Eliminating complex or circular dependencies
  - To reduce compilation dependencies in large software systems
  - To simplify porting systems to other platforms
  - Not to prevent sophisticated clients from accessing subsystem classes

Unit 6.
Proxy

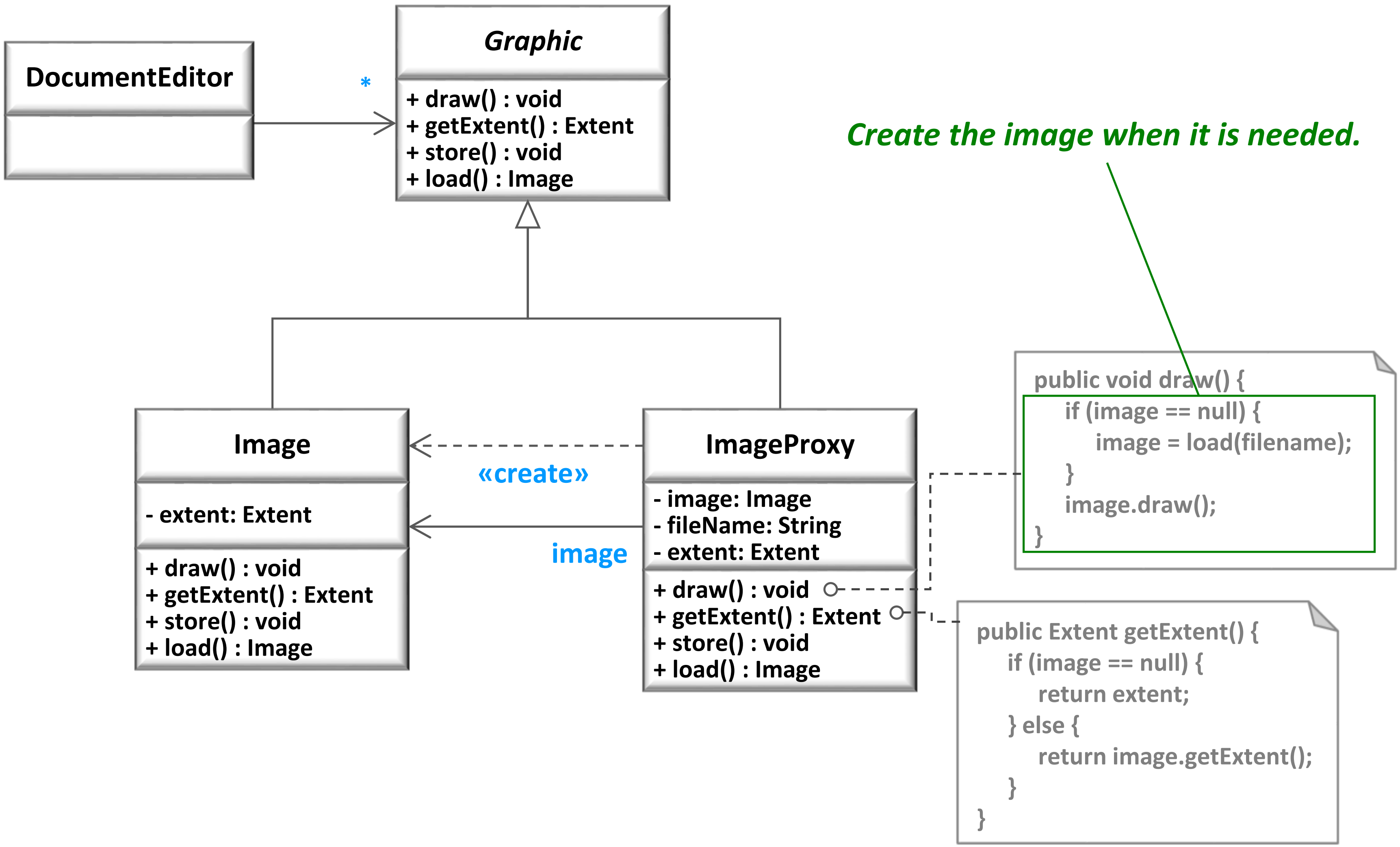
Intent

- **Provide a surrogate or placeholder for another object to control access to it.**
- **A proxy is**
  - a person authorized to act for another person
  - an agent or substitute
  - the authority to act for another
- **There are situations in which a client does not or can not reference an object directly, but wants to still interact with the object.**
- **A proxy object can act as the intermediary between the client and the target object.**
  - The proxy object has the same interface as the target object.
  - The proxy holds a reference to the target object and can forward requests to the target as required (delegation!).
  - In effect, the proxy object has the authority the act on behalf of the client to interact with the target object.

- **Consider a document editor that can embed graphical objects in a document.**
  - Creating larger graphic objects can be expensive to create, but opening a document should be fast.
  - Need to defer the full cost of its creation and initialization until we actually need to use it
  - A solution for this is to use proxy acting as a stand-in for the real image.
    - The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation.



Motivation (2)

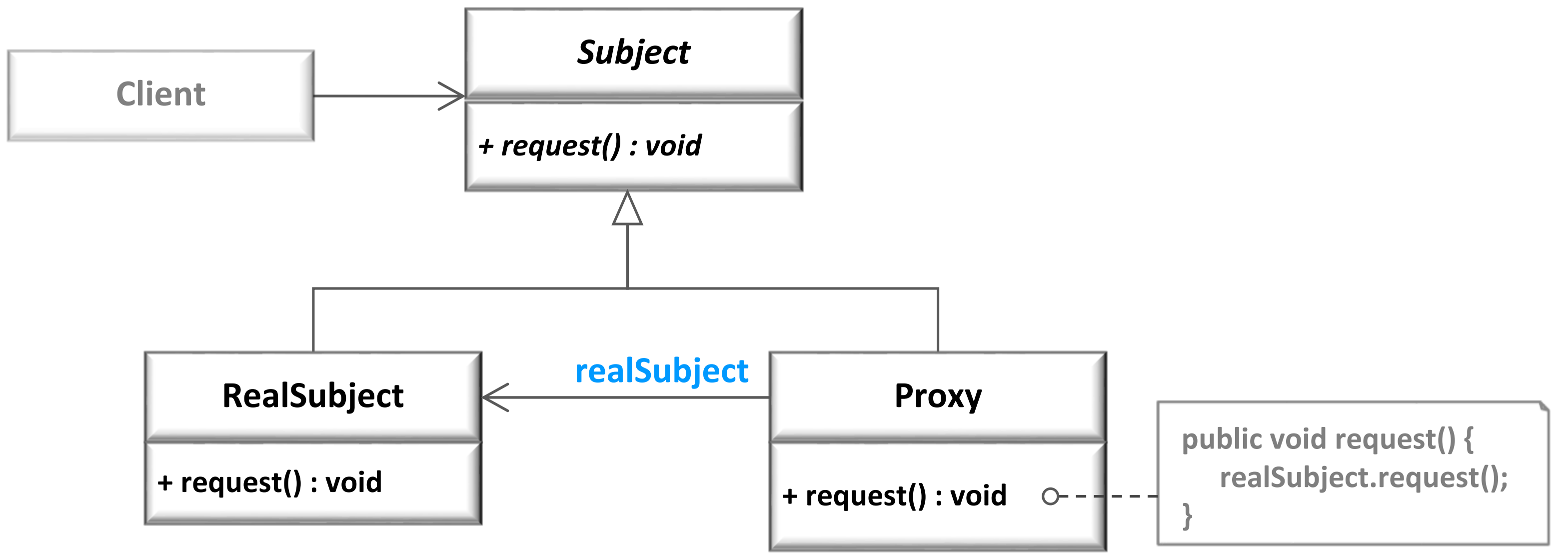


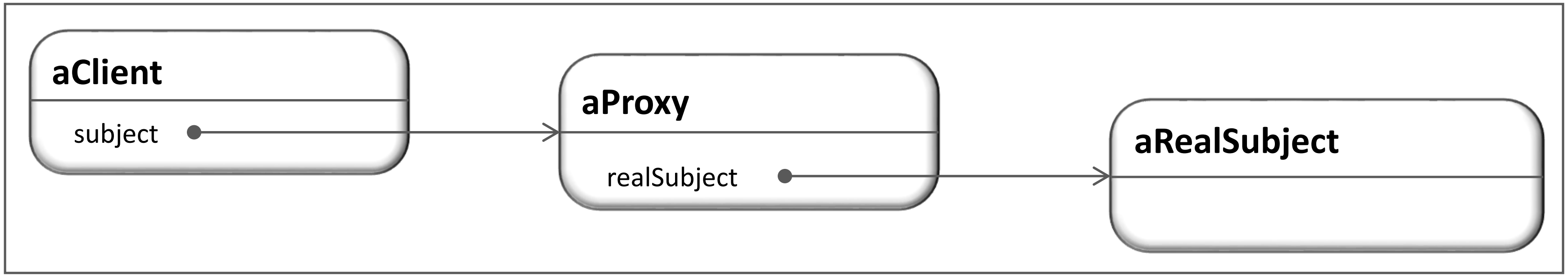
Applicability (1)

- **Proxies are useful wherever there is a need for a more sophisticated reference to a object than a simple pointer or simple reference can provide.**

Applicability (2)

- **Common situations in which the Proxy pattern is applicable**
  - Remote Proxy
    - To provide a reference to an object located in a different address space on the same or different machine
  - Virtual Proxy
    - To allow creation of a memory intensive object on demand
    - The object will not be created until it is really needed.
  - Protection (Access) Proxy
    - To provide different clients with different levels of access to a target object
    - Useful when objects should have different access rights.
  - Smart Reference Proxy
    - To provide additional actions whenever a target object is referenced such as;
      - Counting the number of references to the object
      - Loading a persistent object into memory when it's first referenced
      - Checking that the real object is locked before it's accessed to ensure that no other object can change it.


Client
Subject
+ request() : void
RealSubject
+ request() : void
realSubject
Proxy
+ request() : void
public void request() {
realSubject.request();
}


aClient
subject
aProxy
realSubject
aRealSubject

Participants (1)

- **Proxy (ImageProxy)**
  - To maintain a reference that lets the proxy access the real subject
    - Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - To provide an interface identical to Subject's so that a proxy can by substituted for the real subject
  - To control access to the real subject and may be responsible for creating and deleting it
  - Other responsibilities depending on the kind of proxy
    - *Remote Proxies* – To encode a request and its arguments and send the encoded request to the real subject in a different address space
    - *Virtual Proxies* – To cache additional information about the real subject so that they can postpone accessing it
    - *Protection Proxies* – To check that the caller has the access permissions required to perform a request

Participants (2)

- **Subject (Graphic)**
  - To define the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
- **RealSubject (Image)**
  - To define the real object that the proxy represent

Collaborations

- **Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.**

Consequences

- **To introduce a level of indirection when accessing an object**
  - A remote proxy can hide the fact that an object resides in a different address space.
  - A virtual proxy can perform optimizations such as creating an object on demand.
  - Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Unit 7.
Mediator

Intent

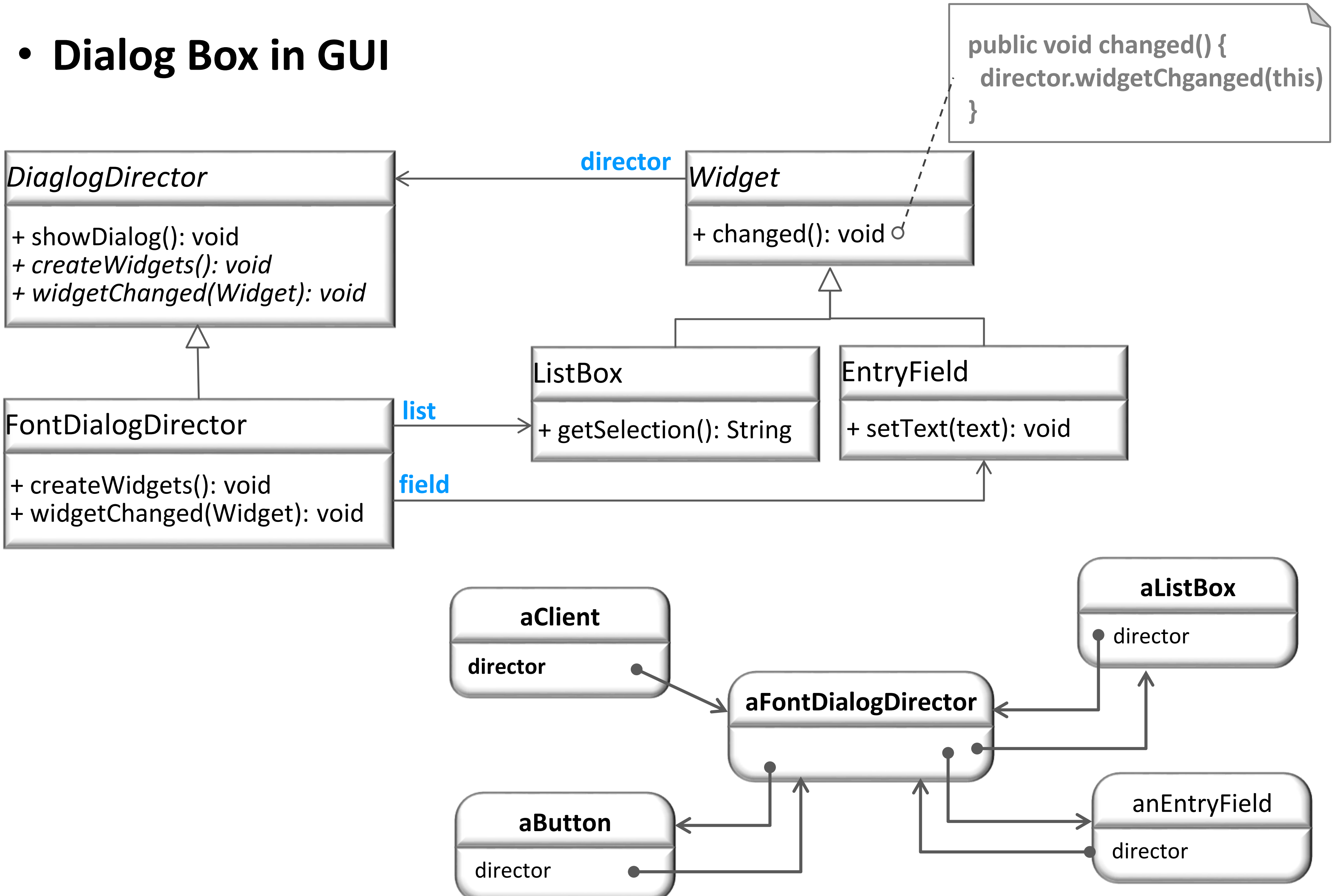
- **To define an object that encapsulates how a set of objects interact**
- **To promote loose coupling by keeping objects from referring to each other explicitly**
- **To allow multiple objects interaction independently**

Motivation (1)

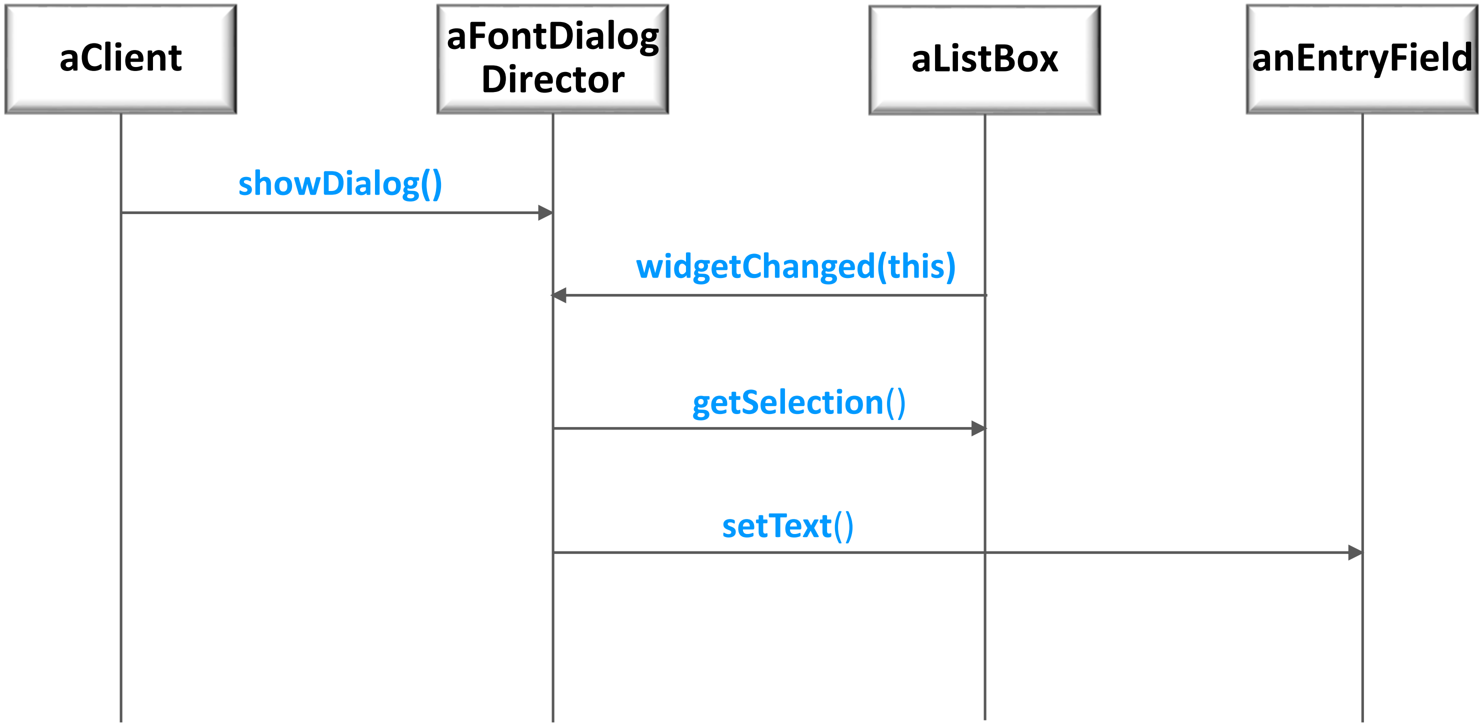
- **Object-oriented design encourages the distribution of behavior among objects.**
  - Such distribution can result in an object structure with many connections between objects.
- **Partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again.**

Motivation (2)

- **Dialog Box in GUI**



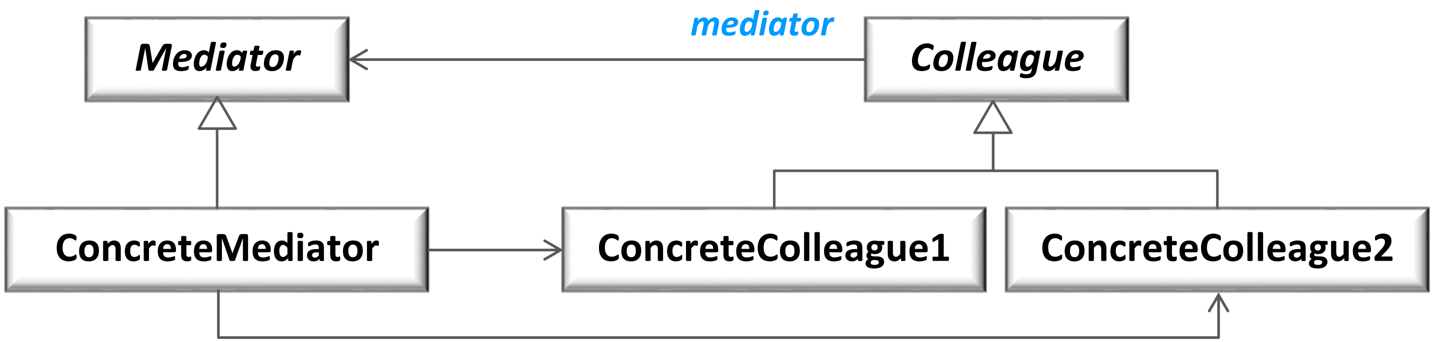
Motivation (3)



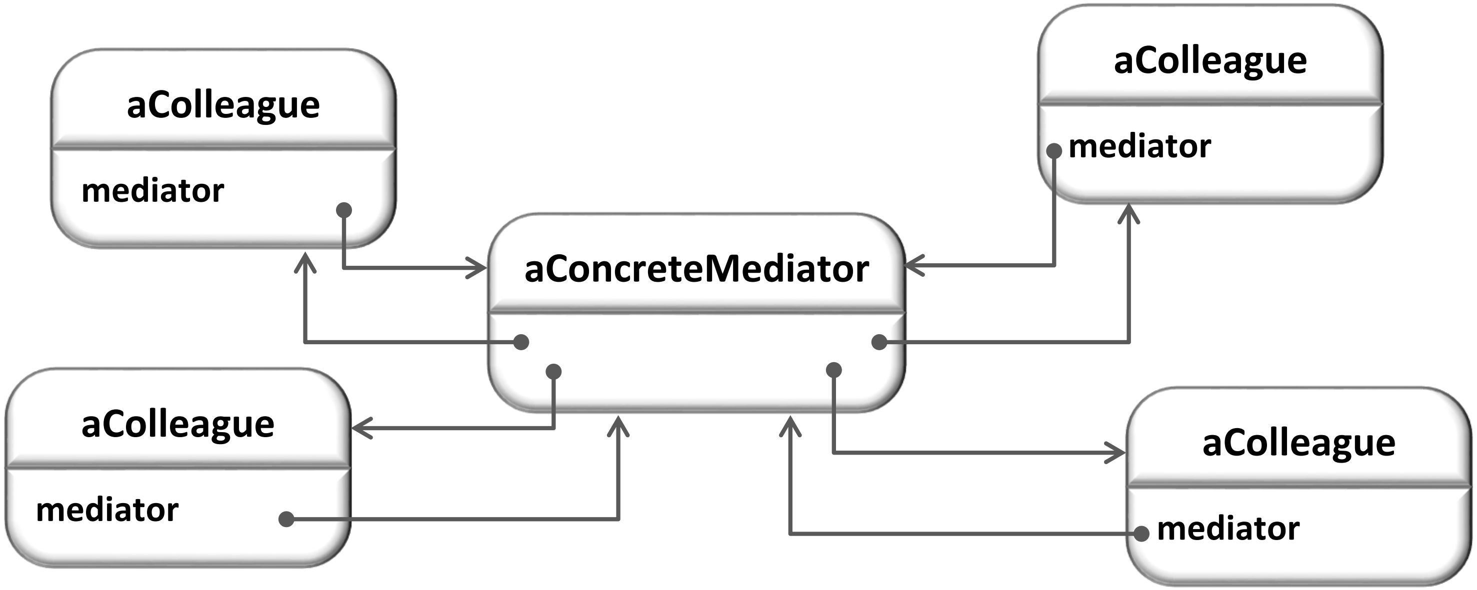
Applicability

- **Applicable Situations:**
  - A set of objects communicate in well-defined but complex ways.
    - The resulting interdependencies are unstructured and difficult to understand.
  - Reusing an object is difficult because it refers to and communicates with many other objects.
  - A behavior that's distributed between several classes should be customizable without a lot of subclassing.

Structure



- **A typical object structure**



Participants

- **Mediator (DialogDirector)**
  - To define an interface for communicating with Colleague objects
- **ConcreteMediator (FontDialogDirector)**
  - To implement cooperative behavior by coordinating Colleague objects
  - To know and maintain its colleagues
- **Colleague classes (ListBox, EntryField)**
  - Each Colleague class knows its Mediator object.
  - Each colleague communicates with its mediator whenever it would have communicated with another colleague.

Collaborations

- **Colleagues send and receive requests from a Mediator object.**
- **The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).**

Consequences

- To limit subclassing
- To decouple colleagues
- To simplify object protocols
- To abstract how objects cooperate
- To centralize controls

Unit 8.
Memento

Intent

- **To capture and externalize an object's internal state so that the object can be restored to this state later without violating encapsulation**

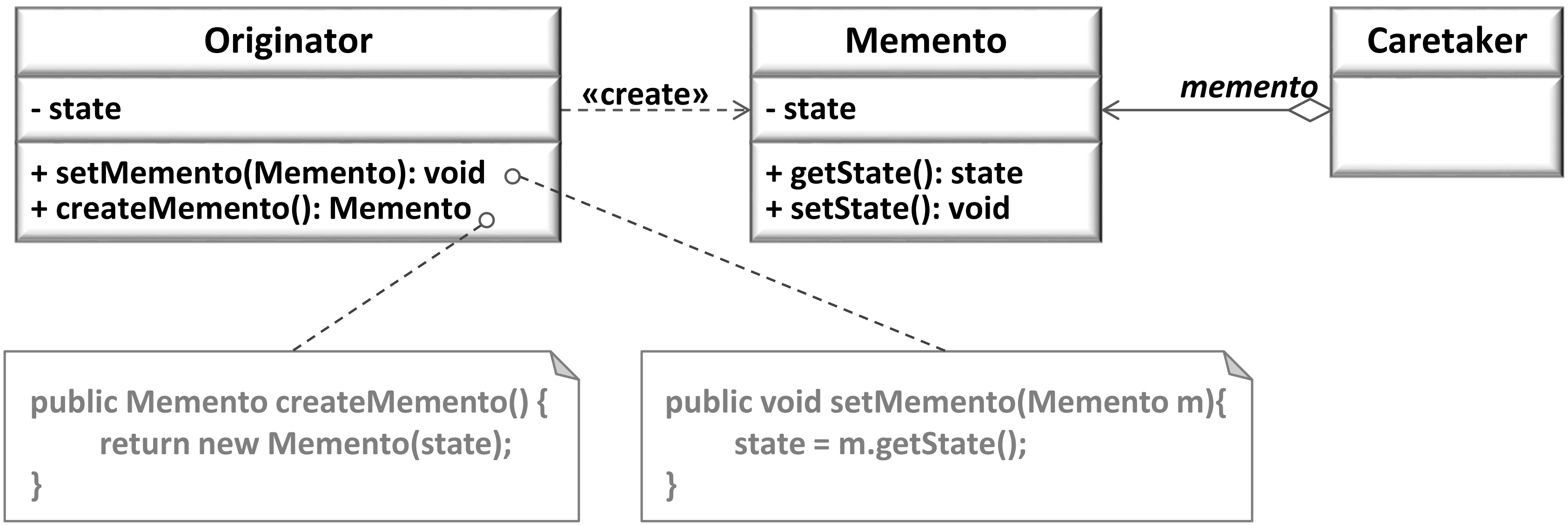
Motivation

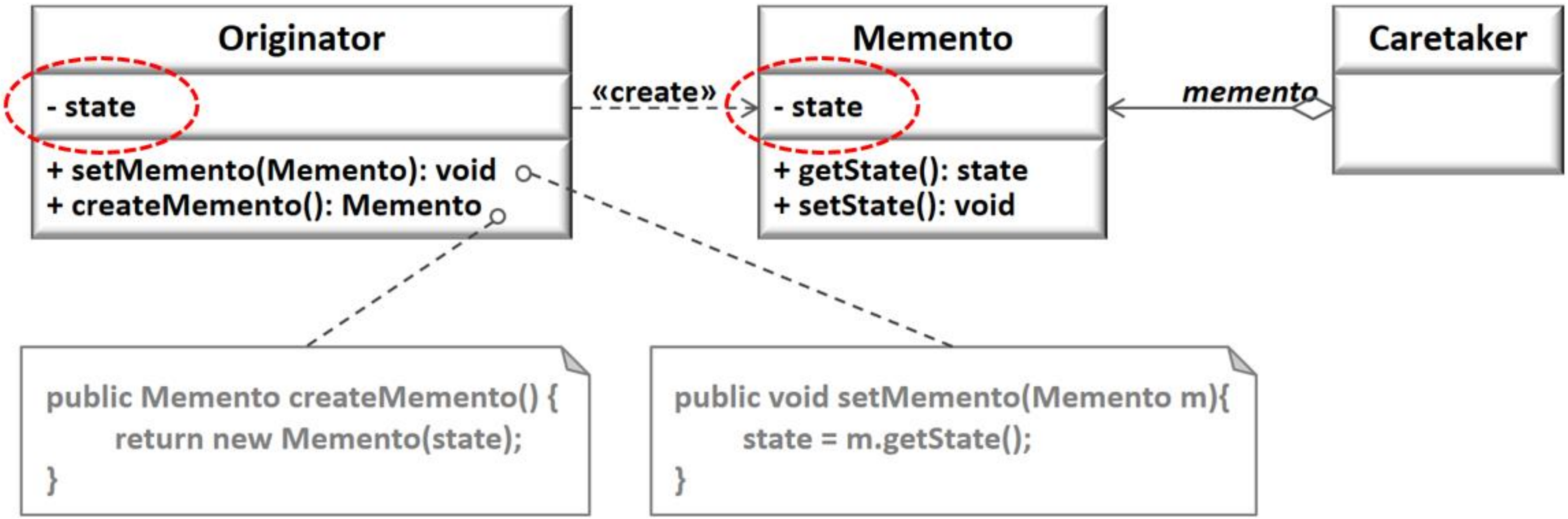
- **To record the internal state of an object**
  - To save state information somewhere so that you can restore objects to their previous states
- **Objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally.**
- **Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility.**

Applicability

- **Applicable Situations:**
  - A snapshot of an object's state must be saved so that it can be restored to that state later.
  - A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure


Originator
- state
+ setMemento(Memento): void
+ createMemento(): Memento
«create»
Memento
- state
+ getState(): state
+ setState(): void
memento
Caretaker
public Memento createMemento() {
return new Memento(state);
}
public void setMemento(Memento m){
state = m.getState();
}


Originator
- state
+ setMemento(Memento): void
+ createMemento(): Memento
«create»
Memento
- state
+ getState(): state
+ setState(): void
memento
Caretaker
public Memento createMemento() {
return new Memento(state);
}
public void setMemento(Memento m){
state = m.getState();
}

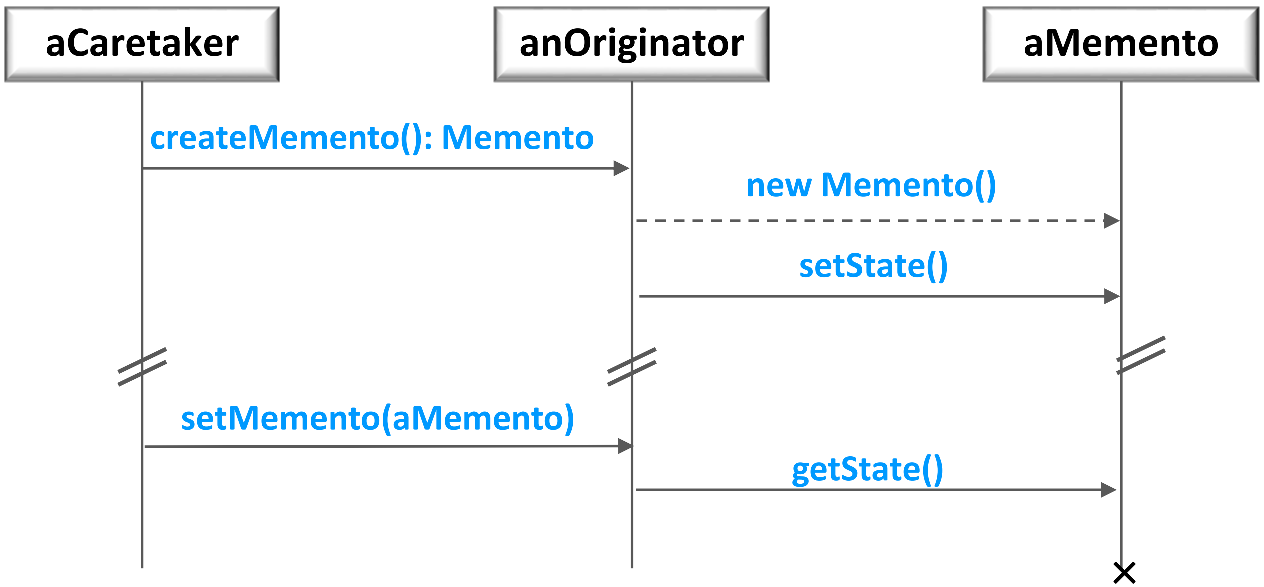
Participants (1)

- **Memento (SolverState)**
  - To store internal state of the Originator object
    - The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - To protect against access by objects other than the originator. Mementos have effectively <u>two interfaces</u>;
    - Caretaker sees a *narrow* interface to the Memento.
      - it can only pass the memento to other objects.
    - Originator sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state.
      - Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

Participants (2)

- **Originator (ConstraintSolver)**
  - To create a memento containing a snapshot of its current internal state
  - To use the memento to restore its internal state
- **Caretaker (undo mechanism)**
  - To be responsible for the memento's safekeeping
  - Never to operate on or examine the contents of a memento

- **A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator, as the following interaction diagram illustrates.**



- **Mementos are passive.**
  - Only the originator that created a memento will assign or retrieve its state.

Consequences

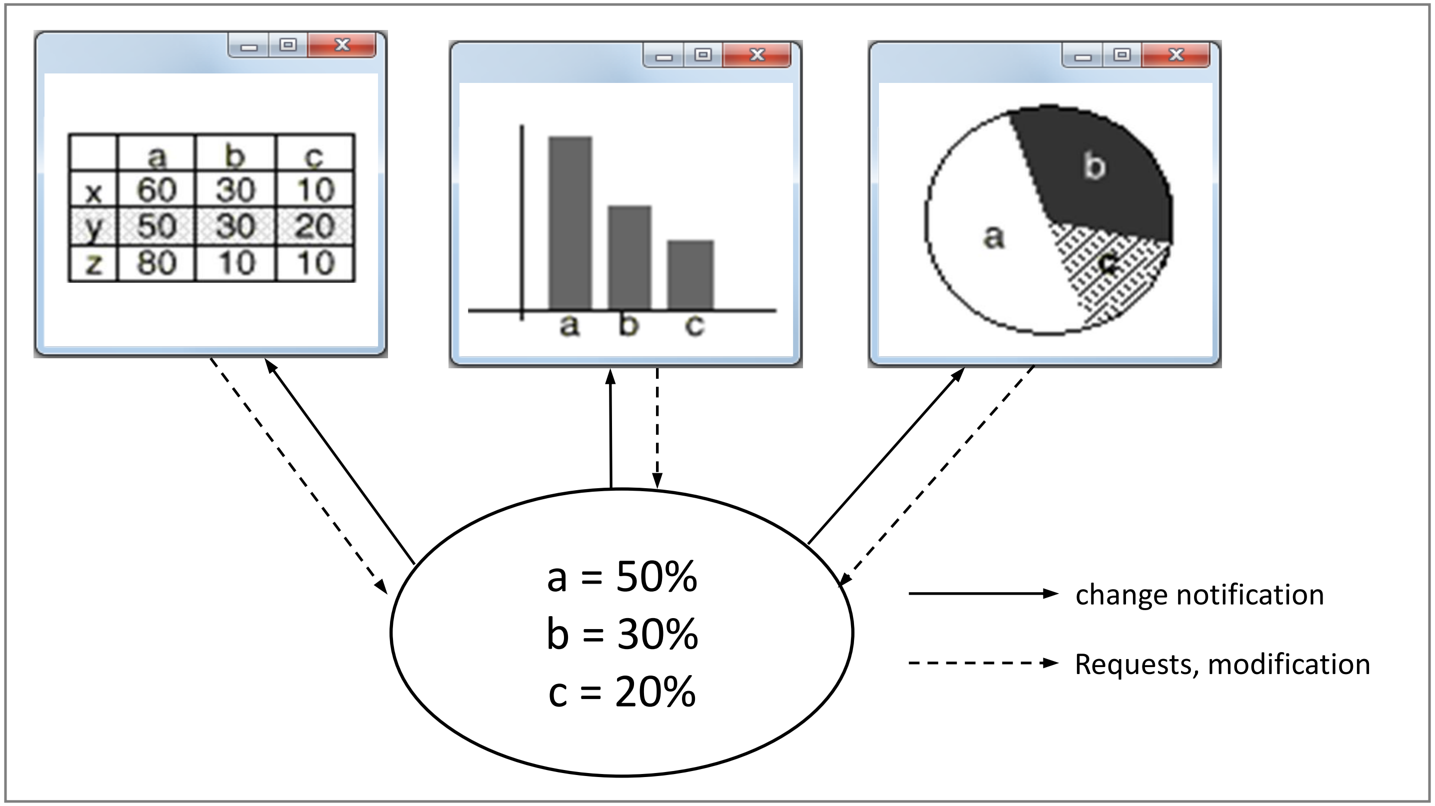
- To preserve encapsulation boundaries
- To simplify Originator
- Using mementos might be expensive.
- To define narrow and wide interfaces
- To hide costs in caring for mementos

Unit 9.
Observer

Intent

- To define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

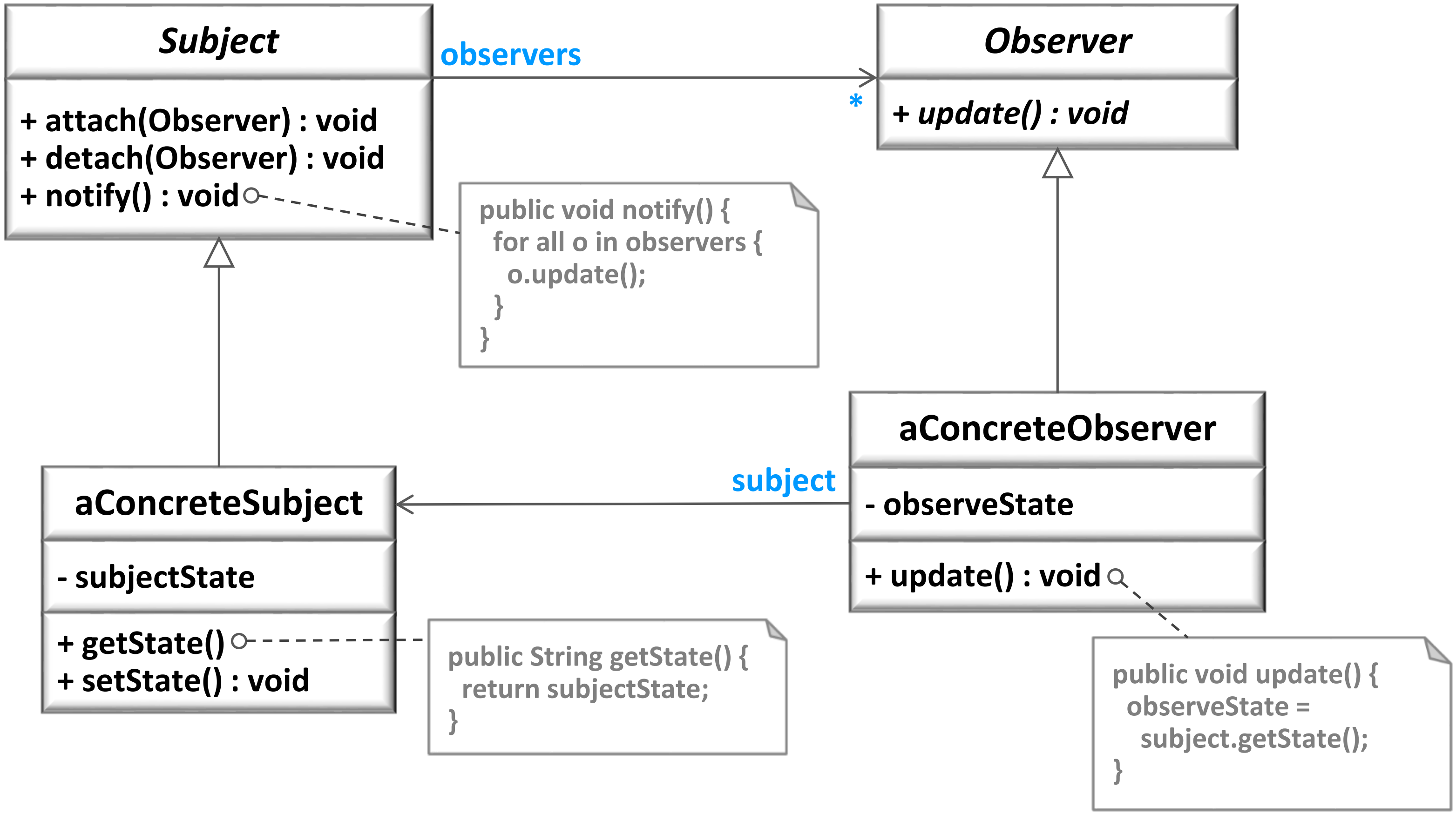
- **Consider graphical user interface toolkits which separates the presentational aspects of the user interface from the underlying application data**
  - The need to maintain consistency between related objects without making classes tightly coupled



Applicability

- **Use the Observer pattern in any of the following situations:**
  - When an abstraction has two aspects, one dependent on the other.
    - Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others
  - When an object should be able to notify other objects without making assumptions about those objects

Structure


Subject
+ attach(Observer) : void
+ detach(Observer) : void
+ notify() : void
observers
*
Observer
+ update() : void
public void notify() {
for all o in observers {
o.update();
}
}
aConcreteObserver
- observeState
+ update() : void
subject
aConcreteSubject
- subjectState
+ getState()
+ setState() : void
public String getState() {
return subjectState;
}
public void update() {
observeState =
subject.getState();
}

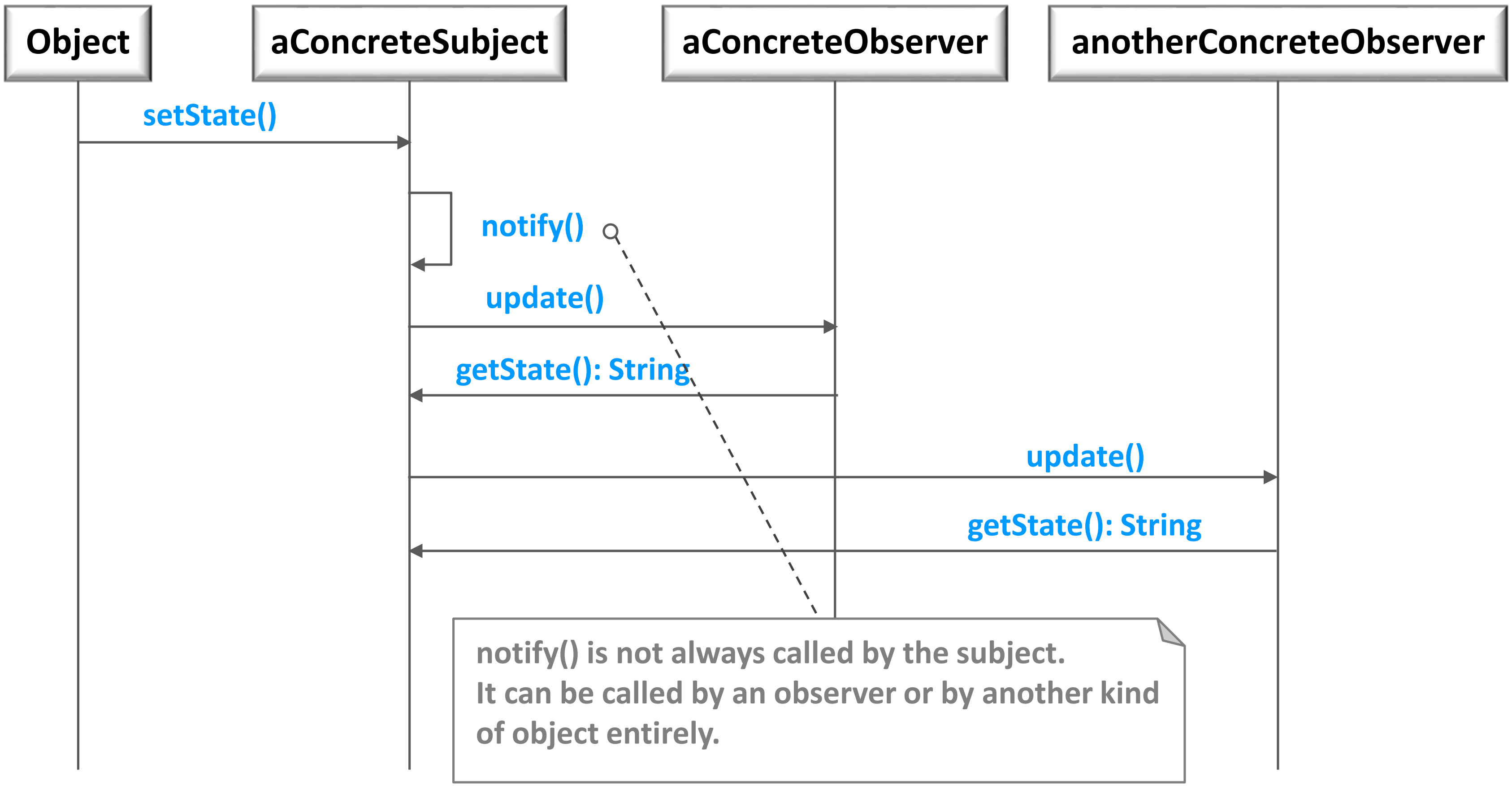
Participants

- **Subject**
  - To keep track of its observers
  - To provide an interface for attaching and detaching Observer objects
- **Observer**
  - To define an interface for update notification
- **ConcreteSubject**
  - The object being observed
  - To store state of interest to ConcreteObserver objects
  - To send a notification to its observers when its state changes
- **ConcreteObserver**
  - The observing object
  - To store state that should stay consistent with the subject's
  - To implement the Observer update interface to keep its state consistent with the subject's

Collaborations (1)

- **Notifying changes to observers**
  - ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- **Reflecting the changes**
  - After being informed of a change in the concrete object, ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

Collaborations (2)



Consequences

- **Minimal coupling between the Subject and the Observer**
  - Can reuse subjects without reusing their observers and vice versa
  - Observers can be added without modifying the subject
  - All subject knows is its list of observers
  - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
  - Subject and observer can belong to different abstraction layers
- **Support for event broadcasting**
  - Subject sends notification to all subscribed observers
  - Observers can be added/removed at any time

Consequences (2)

- **Disadvantages**
  - Possible cascading of notifications
    - Observers are not necessarily aware of each other and must be careful about triggering updates
  - Simple update interface requires observers to deduce changed item

Unit 10.
Strategy

Intent

- **Define a family of algorithms, encapsulate each one, and make them interchangeable.**
- **Strategy lets the algorithm vary independently from clients that use it.**

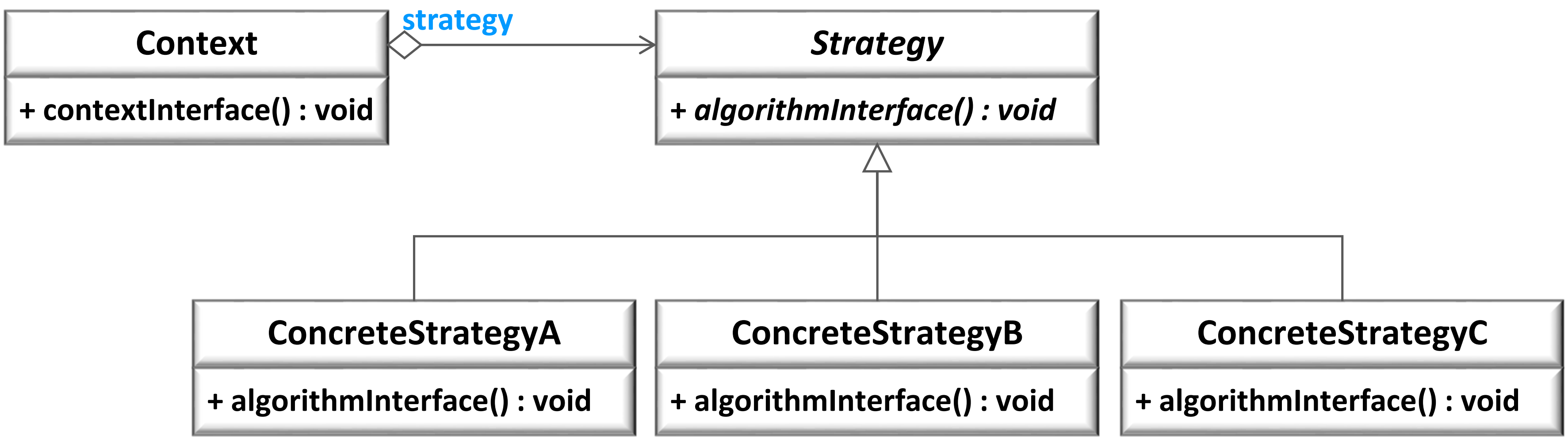
Motivation

- **Typical Problems of Algorithms for Breaking a stream of Text into Lines**
  - To get more complex if client applications include the linebreaking code
    - That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.
  - Not easy to use different algorithms at different times
  - Difficult to add new algorithms and vary existing ones

Applicability

- **Use the Strategy pattern when:**
  - To make many related classes differ only in their behavior
  - To need different variants of an algorithm
  - Not to expose data used by the algorithm to clients
    - Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - To define many behaviors which appear as multiple conditional statements in its operations
    - Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure


Context
+ contextInterface() : void
strategy
Strategy
+ algorithmInterface() : void
ConcreteStrategyA
+ algorithmInterface() : void
ConcreteStrategyB
+ algorithmInterface() : void
ConcreteStrategyC
+ algorithmInterface() : void

Participants

- **Strategy (Compositor)**
  - To declare an interface common to all supported algorithms
  - Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)**
  - To implement the algorithm using the Strategy interface
- **Context (Composition)**
  - To be configured with a ConcreteStrategy object
  - To maintain a reference to a Strategy object
  - May define an interface that lets Strategy access its data.

Collaborations

- **Strategy and Context interact to implement the chosen algorithm.**
  - A context may pass all data required by the algorithm to the strategy when the algorithm is called.
  - Alternatively, the context can pass itself as an argument to Strategy operations.
  - That lets the strategy call back on the context as required.
- **A context forwards requests from its clients to its strategy.**
  - Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively.
  - There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences

- **Advantages**
  - To provide an alternative to subclassing the Context class to get a variety of algorithms or behaviors
  - To eliminate large conditional statements
  - To provide a choice of implementations for the same behavior
- **Disadvantages**
  - To increase the number of objects
  - All algorithms must use the same Strategy interface.

Unit 11.
Builder

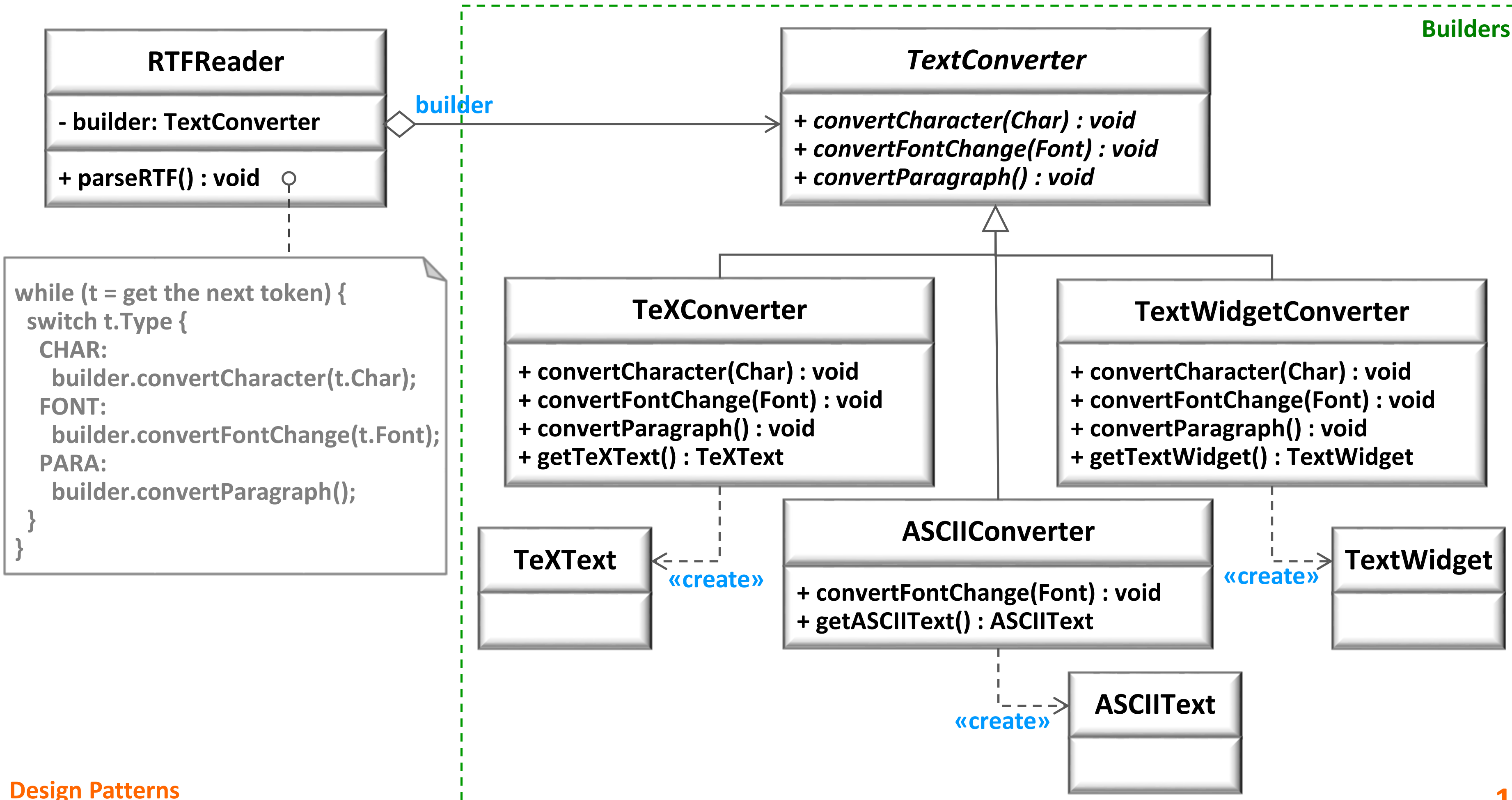
Intent

- **Separate the construction of a complex object from its representation so that the same construction process can create different representations.**
- **Allow for the dynamic creation of objects based upon easily interchangeable algorithms.**

Motivation (1)

- **A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats such as plain ASCII text or text widget.**
  - The number of possible conversions is open-ended.
  - It should be easy to add a new conversion without modifying the reader.

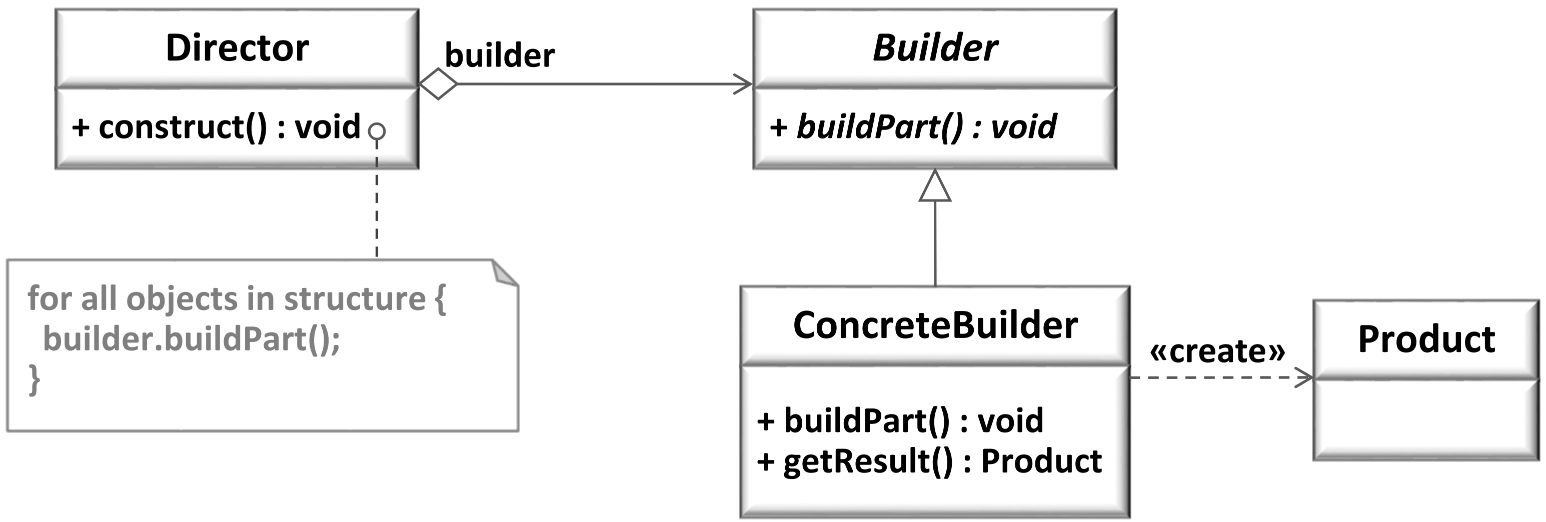
- A solution to configuring the RTFReader class with a TextConverter object that converts RTF to another textual representation:



Applicability

- **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's constructed.
  - The addition of new creation functionality without changing the core code is necessary.
  - Runtime control over the creation process is required.

Structure


Director
+ construct() : void
builder
Builder
+ buildPart() : void
for all objects in structure {
builder.buildPart();
}
ConcreteBuilder
+ buildPart() : void
+ getResult() : Product
«create»
Product

Participants (1)

- **Builder (TextConverter)**
  - To specify an abstract interface for creating parts of a Product object
- **ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter)**
  - To construct and assemble parts of the product by implementing the Builder interface
  - To define and keep track of the representation
  - To provide an interface for retrieving the product
    - e.g., GetASCIIText, GetTextWidget

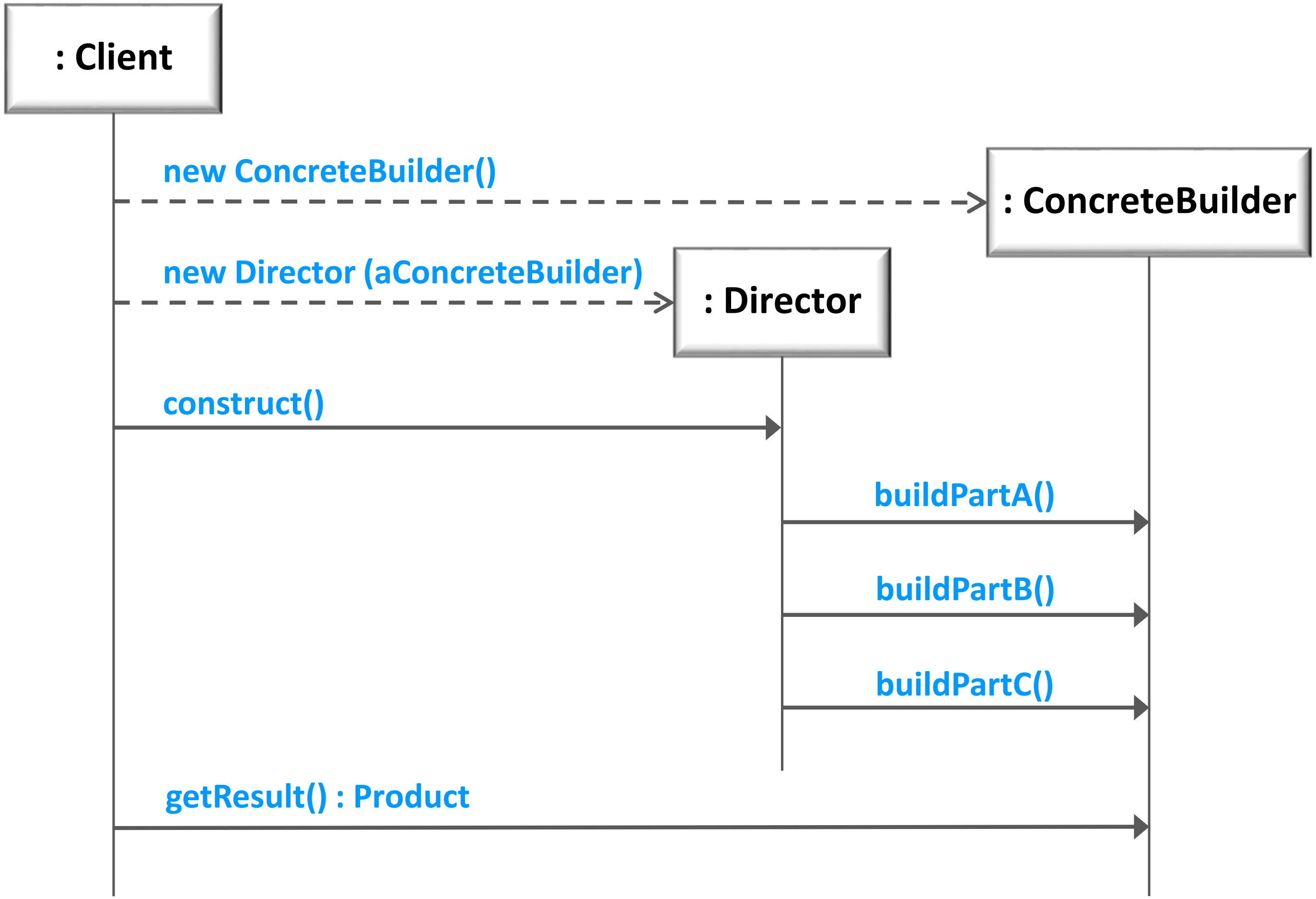
Participants (2)

- **Director (RTFReader)**
  - To construct an object using the Builder interface
- **Product (ASCIIText, TeXText, TextWidget)**
  - To represent the complex object under construction
  - ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
  - To include classes that define the constituent parts, including interfaces for assembling the parts into the final result

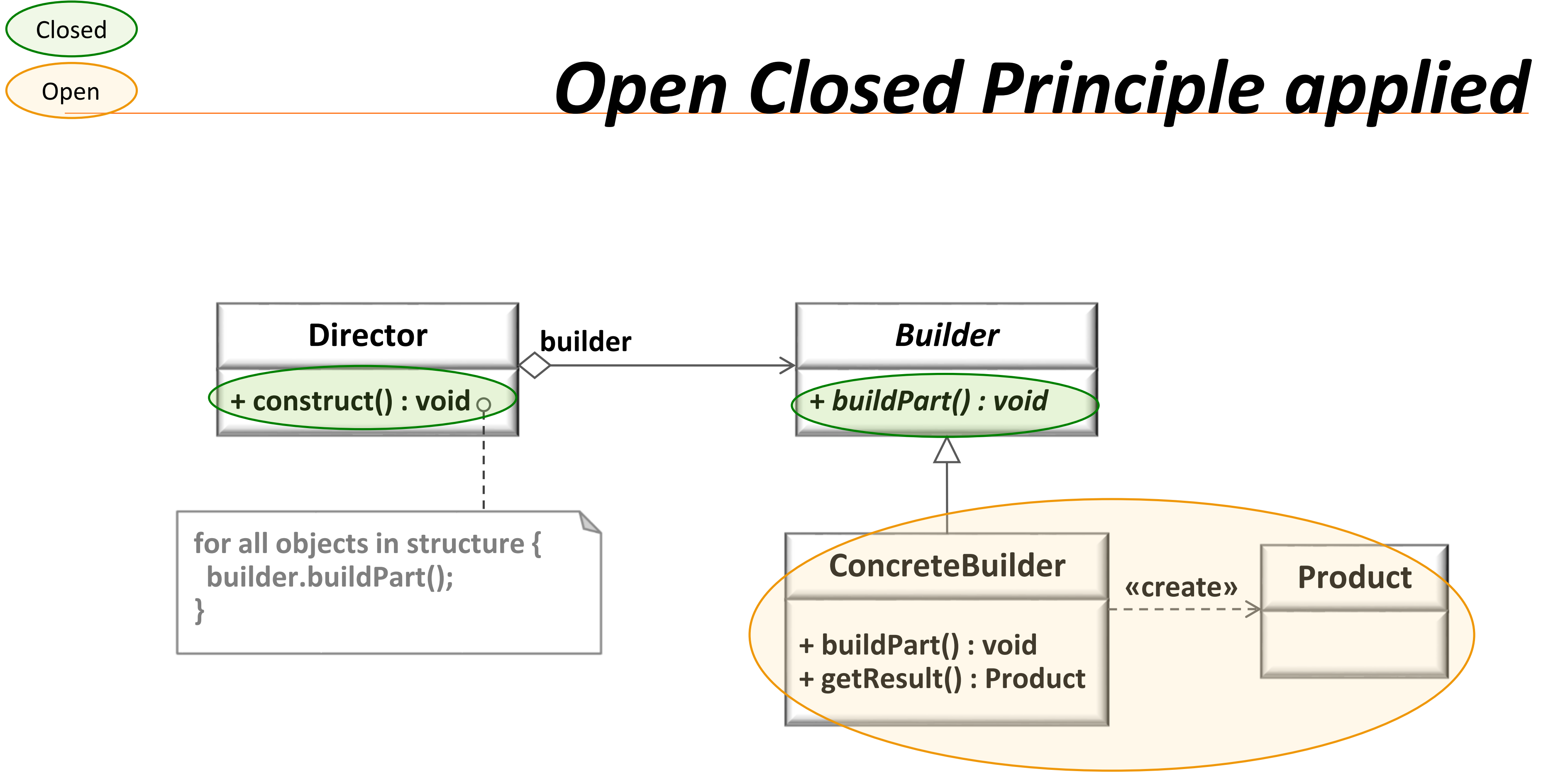
Collaborations (1)

- **The client creates the Director object and configures it with the desired Builder object.**
- **Director notifies the builder whenever a part of the product should be built.**
- **Builder handles requests from the director and adds parts to the product.**
- **The client retrieves the product from the builder.**

Collaborations (2)



Open Closed Principle applied



Consequences

- **To let you vary a product's internal representation**
  - Define a new kind of builder to change the product’s internal representation.
- **To isolate code for construction and representation**
  - Improves modularity by encapsulating the way a complex object is constructed and represented.
- **To give you finer control over the construction process**
  - Supports finer control over the construction process and consequently the internal structure of the resulting product.

END