

# Object Oriented Programming - Lecture 13

Diana Borza - [diana.borza@ubbcluj.ro](mailto:diana.borza@ubbcluj.ro)

May 23, 2022

- Design patterns
  - Composite
  - Observer
  - Factory method
  - Iterator

# Structural design patterns

**Structural patterns** *are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations.*

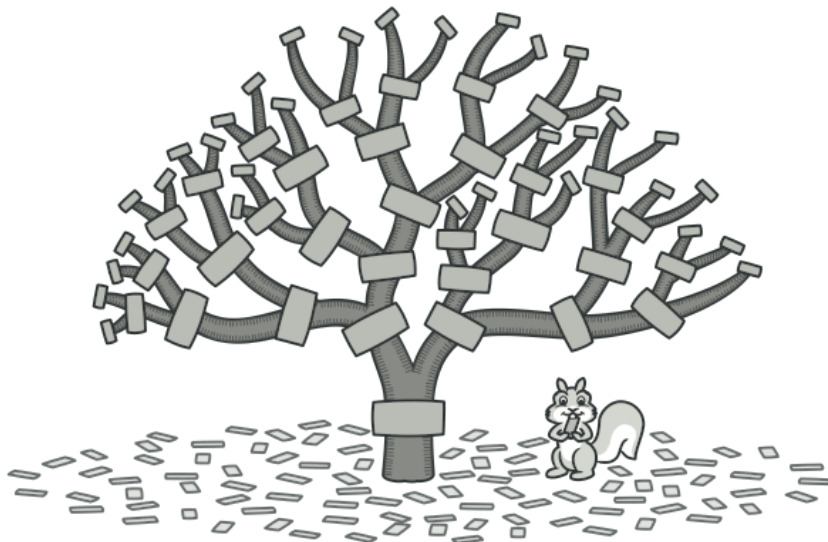
- **Intent**

- Compose objects into tree structures to represent **part-whole hierarchies**. Composite lets clients treat individual objects and compositions of objects uniformly.

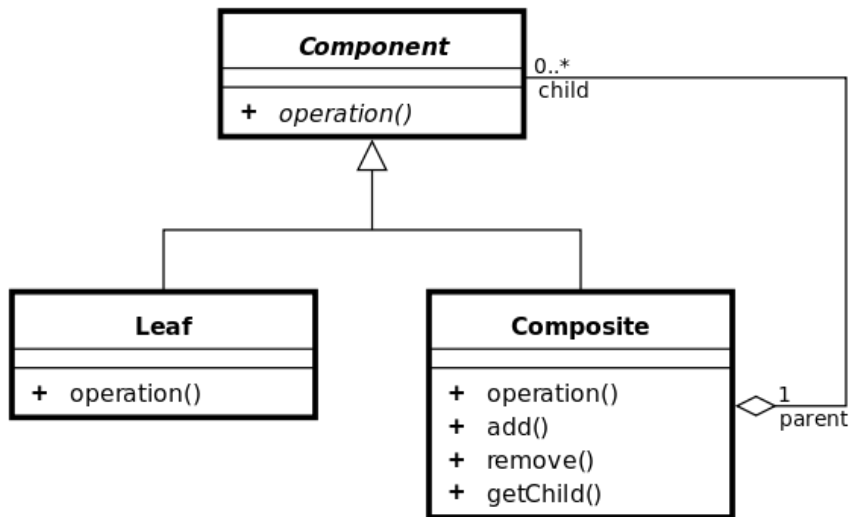
- **Applicability**

- 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.

# Composite



# Composite - structure



- **Client**

- Manipulates objects in the composition through the Component interface.

- **Component**

- declares the interface for objects in the composition.
- implements default behavior for the interface common to all classes, as appropriate.
- declares an interface for accessing and managing its child components.
- (*optional*) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

- **Leaf**

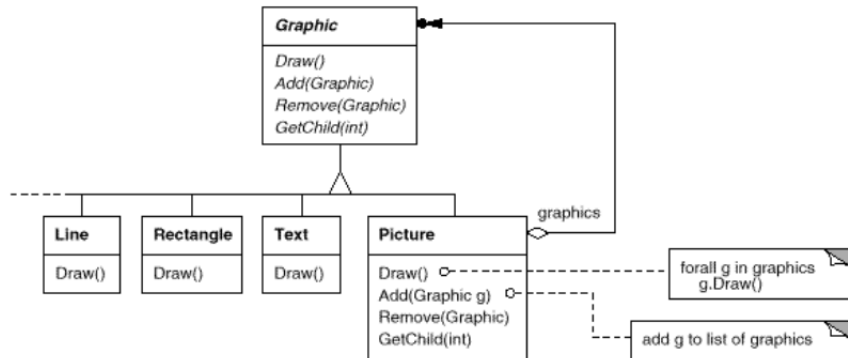
- represents leaf objects in the composition. A leaf has no children.
- defines behavior for primitive objects in the composition.

- **Composite**

- defines behavior for components having children.
- stores child components.
- implements child-related operations in the Component interface.



# Composite - demo



## Demo

### Composite design pattern

## • Advantages

- Makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component.
- You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

## • Disadvantages

- It might be difficult to provide a common interface for classes whose functionality differs too much.
- In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

*Behavioral patterns are concerned with algorithms and the assignment of **responsibilities** between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.*

- **Intent**

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

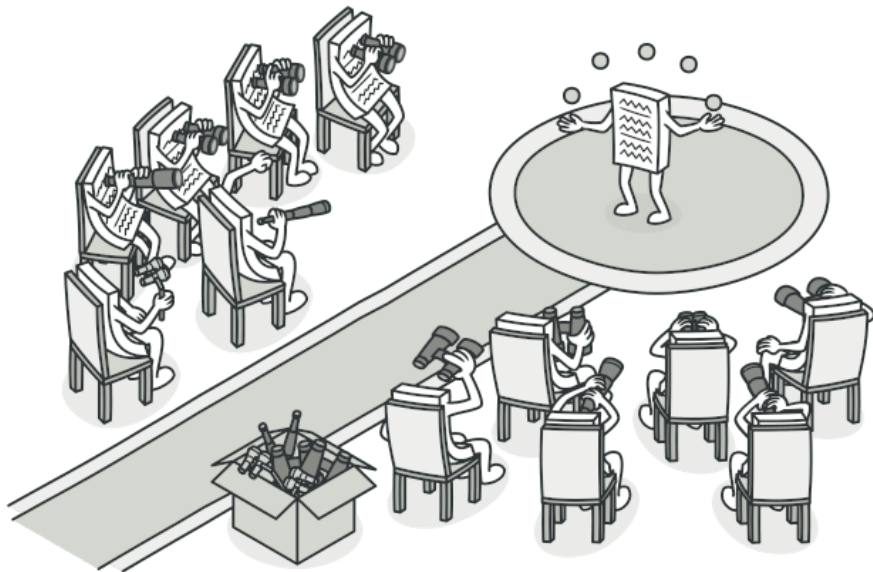
- **Applicability**

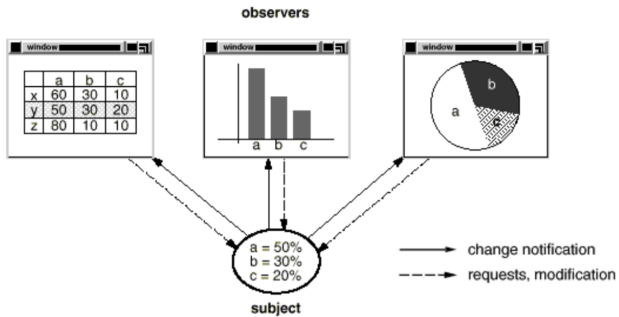
- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are (low coupling between objects)

- **Also known as**

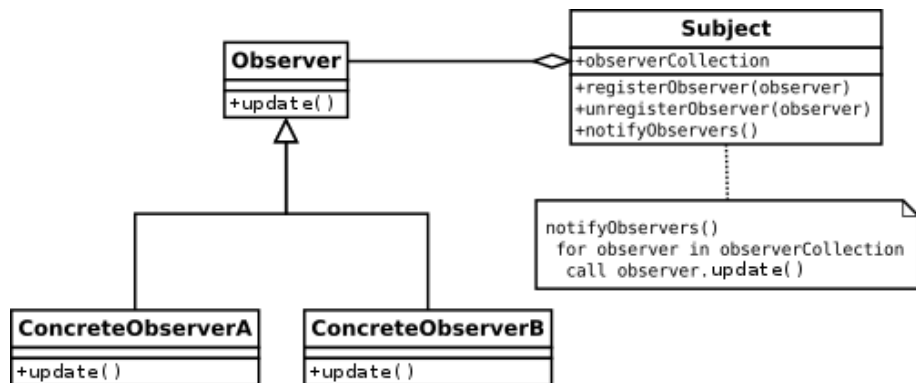
- Publish-Subscribe

# Observer





# Observer - structure



- **Subject**

- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

- **ConcreteSubject**

- stores state of interest to ConcreteObserver objects.
- sends a notification to its observers when its state changes.



- **Observer**

- defines an updating interface for objects that should be notified of changes in a subject.

- **ConcreteObserver**

- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's.

## • Advantages

- Lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa.
- It lets you add observers without modifying the subject or other observers.
- Abstract coupling between Subject and Observer: all a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class (not the concrete class).
- Support for broadcast communication.

## • Disadvantages

- Unexpected updates: observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.
- The simple update protocol provides no details on what changed in the subject. (observers maybe forced to work hard to deduce the changes.)

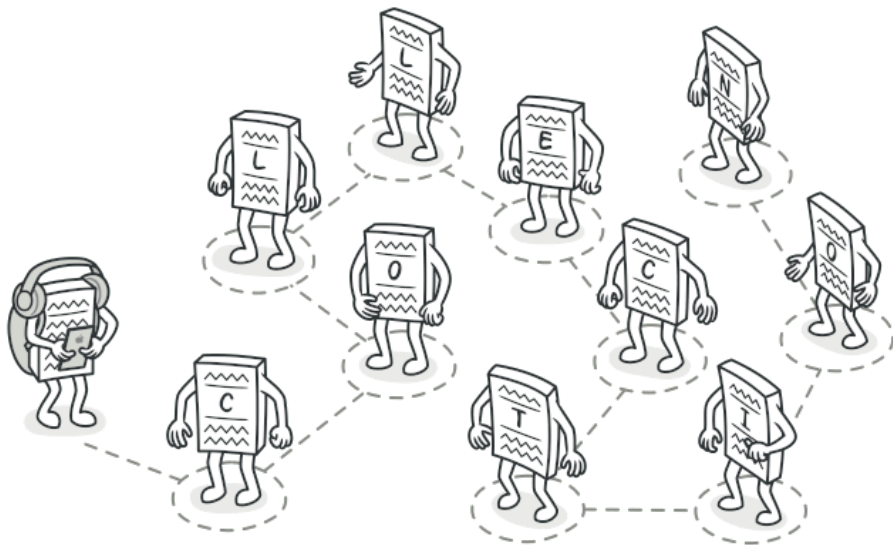
- **Intent**

- Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

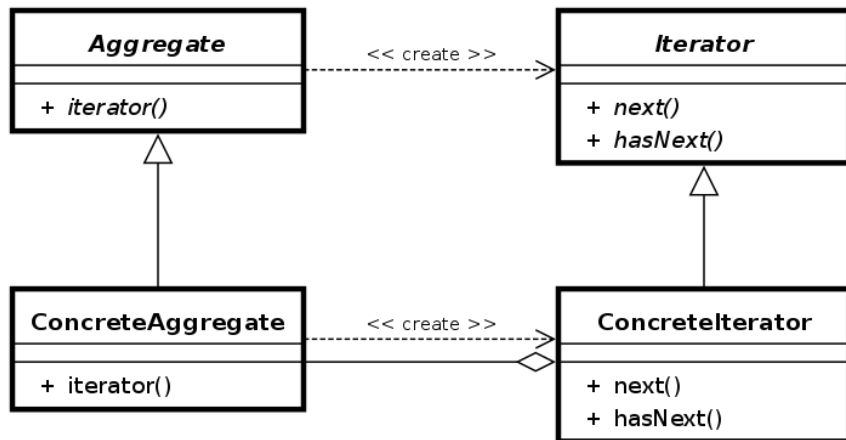
- **Applicability**

- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

# Iterator



# Iterator - structure



- **Iterator**

- defines an interface for accessing and traversing elements.

- **Concreteliterator**

- implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.

- **Aggregate**

- defines an interface for creating an Iterator object.

- **ConcreteAggregate**

- implements the Iterator creation interface to return an instance of the proper Concreteliterator.

# Iterator - implementation

## Demo

### Iterator design pattern

## • Advantages

- It supports variations in the traversal of an aggregate. Complex aggregates may be traversed in many ways.
- Iterators simplify the Aggregate interface. Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
- More than one traversal can be pending on an aggregate. An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

## • Disadvantages

- Applying the pattern can be an overkill if your app only works with simple collections.
- Using an iterator may be less efficient than going through elements of some specialized collections directly.

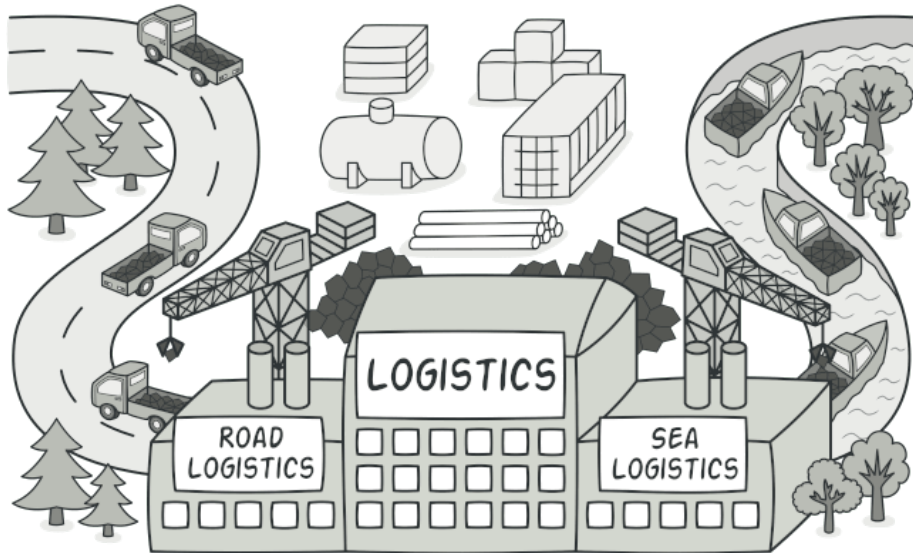


# Creational design patterns

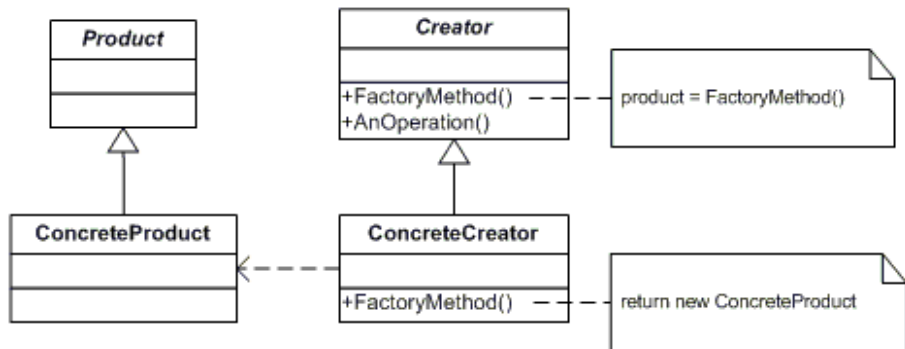
*Creational design patterns abstract the **instantiation process**. They help make a system independent of how its objects are created, composed, and represented.*

- **Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate.
- **Also known as:** Virtual Constructor
- **Applicability**
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

# Factory method



# 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 an object of type Product.
  - may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.

- **ConcreteCreator**

- overrides the factory method to return an instance of a ConcreteProduct.

# Factory method - implementation

Demo

Factory method

## • Advantages

- Eliminates the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.
- *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.
- *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.

## • Disadvantages

- The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern.
- Clients might have to subclass the Creator class just to create a particular ConcreteProduct object.

`https://refactoring.guru/design-patterns`  
`http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf`