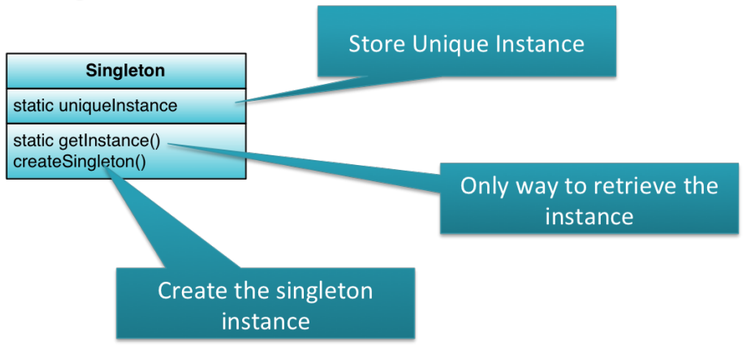
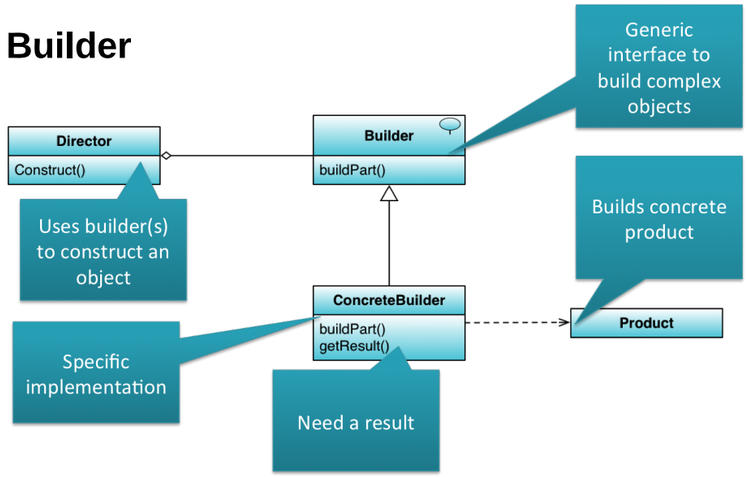
Singleton



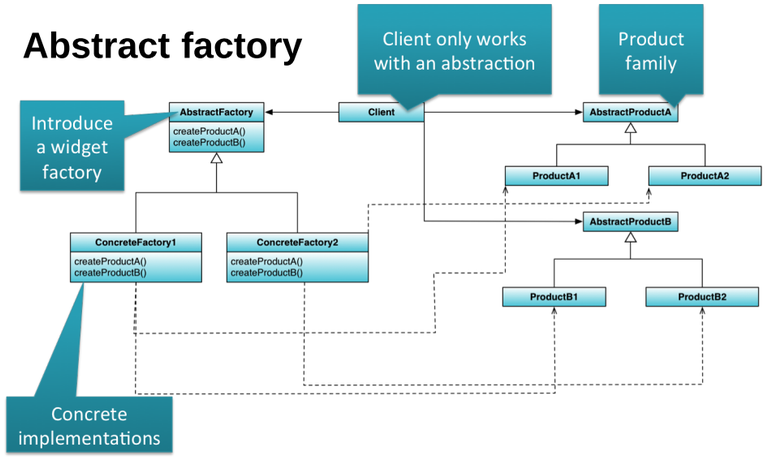
- **When**
  - Only **one instance** of class **required**
  - Must be **one access point**
  - Need to **manage object instances**

Builder



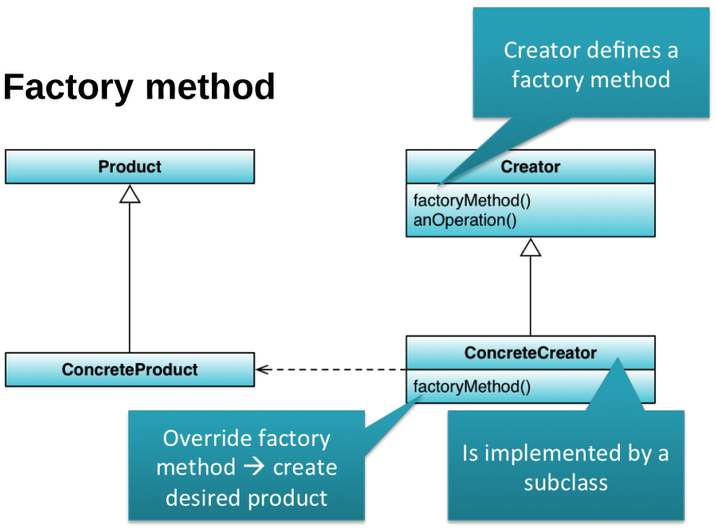
- **Use**
  - **Separate construction** with internal representation
  - One process → **multiple object representation**
  - Object construction <> object assembling

Abstract factory



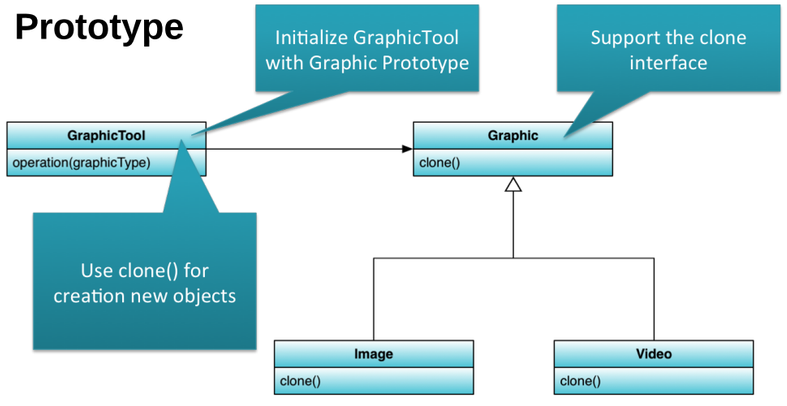
- **Use**
  - **Creation** of products **independent** from the **application**
  - Configuration **of product families** is required
  - Hide product implementation → only provide interface

Factory method



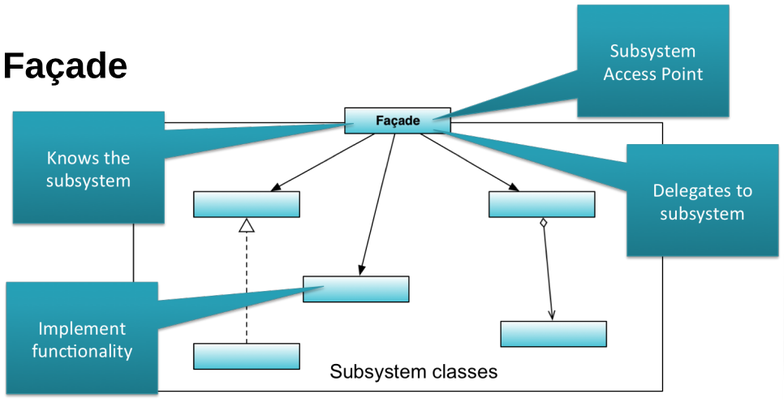
- **When**
  - Class can't **expect** the type of object it must **create**
  - **Subclasses** must decide what types of **objects are created**

Prototype



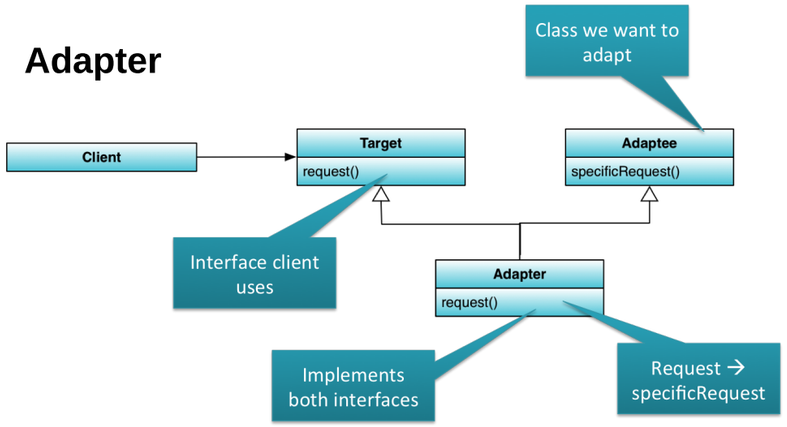
- **When**
  - **Classes** to instantiate are specific at **run-time**
  - Avoid building class hierarchies (abstract factory pattern)
  - A class can have **limited instances** of state
    - Cloning is more efficient

Façade



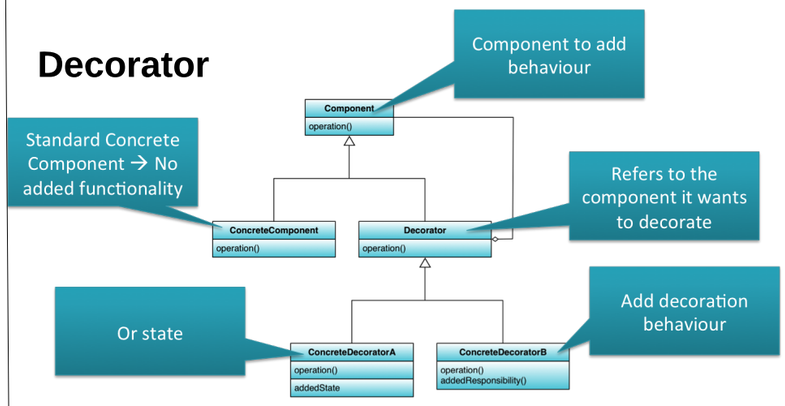
- Use
  - **Decouple** clients from subsystems
  - Provide **simple interface**
  - Subsystem layering (business, data and client services)

Adapter



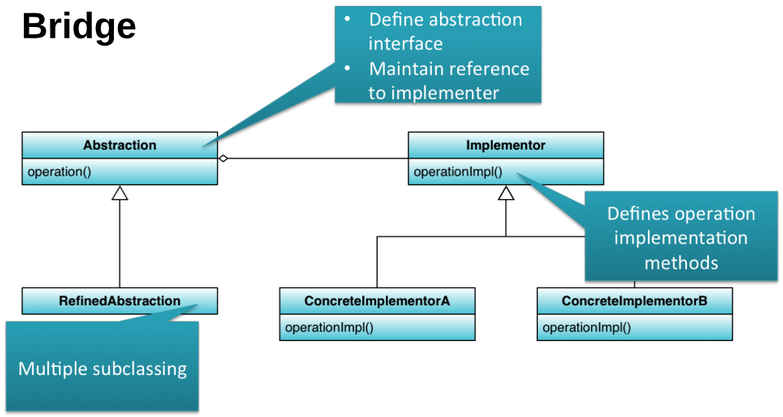
- Use
  - **Re-use** an existing class
  - **Combine** unrelated **classes** with an incompatible interface

Decorator



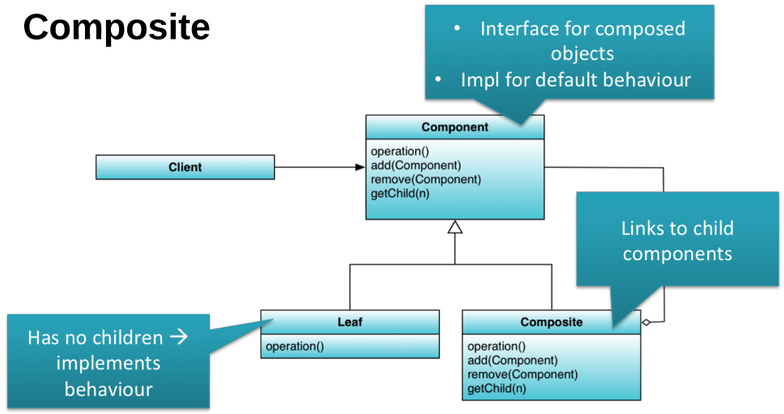
- Use
  - **Add functionality** to objects, without affecting other objects
  - Functionalities can be taken away in the future
  - **Extension** by subclassing **is difficult**

Bridge



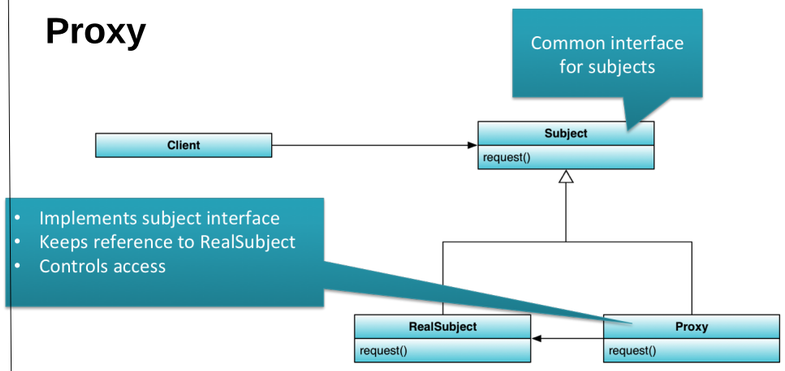
- Use
  - **Avoid binding** between interface and implementation
  - Possible subclasses for abstraction and implementation
  - Must be possible to **change implementation at runtime** without affecting clients

Composite



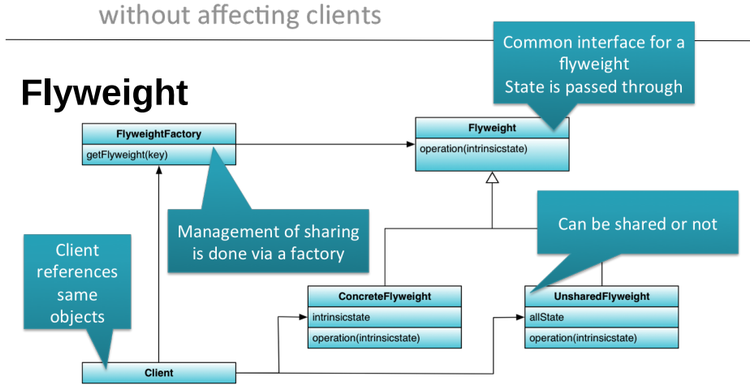
- Use
  - **Ignore differences** between **compositions** and **individual items**
  - **Represent** part-whole **hierarchies of objects**

Proxy



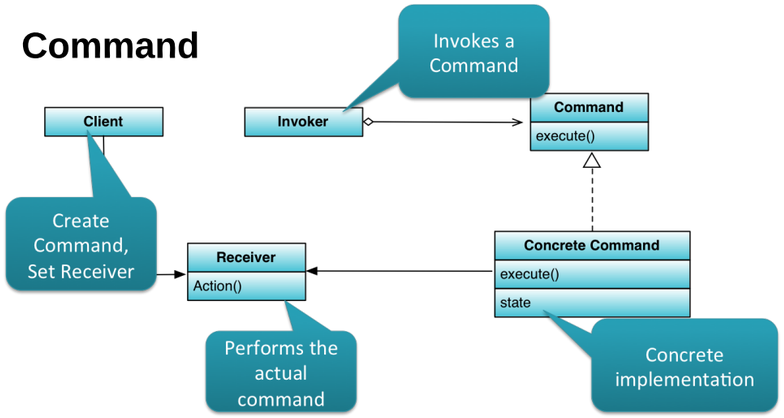
- Use
  - Extra functionality is required
    - **Transparancy**
    - More than just a reference

Flyweight



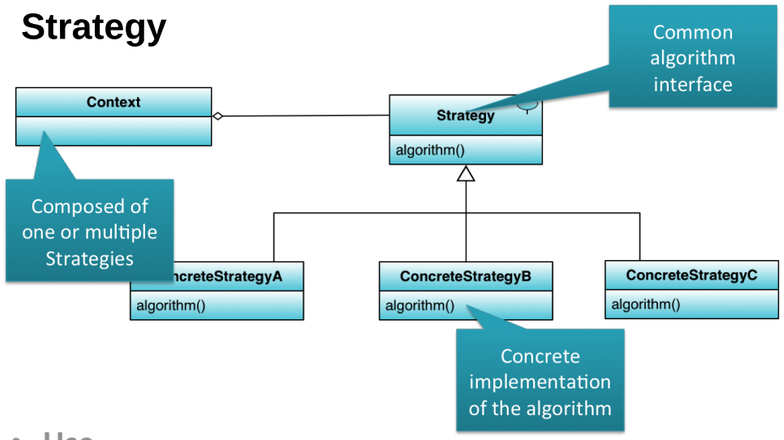
- Use
  - **Large number** of objects
  - High **storage costs**
  - Extrinsic state (**shareable**)
  - Many objects → replaced by few objects
  - Object identity isn't necessary

Command



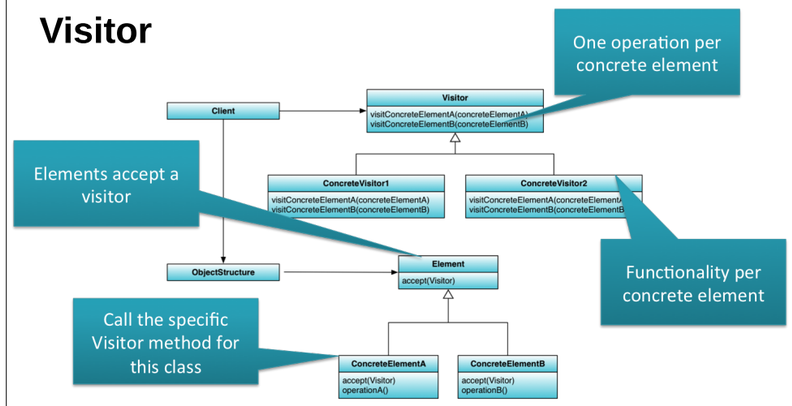
- **Use**
  - Command as **parameter**
  - Pass command like general object
  - **Queue Request**
  - **Save** request state
  - Undo functionality
  - Provide an execute and undo method
  - Support Logging
  - **Re-execute** code in case of failure

Strategy



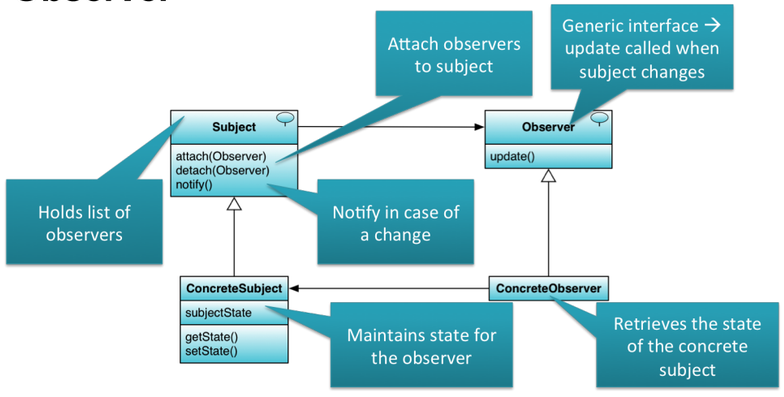
- **Use**
  - Classes only change in behavior
  - Different variants of an **algorithm**
  - Algorithms that **use complex data** that clients shouldn't be aware of

Visitor



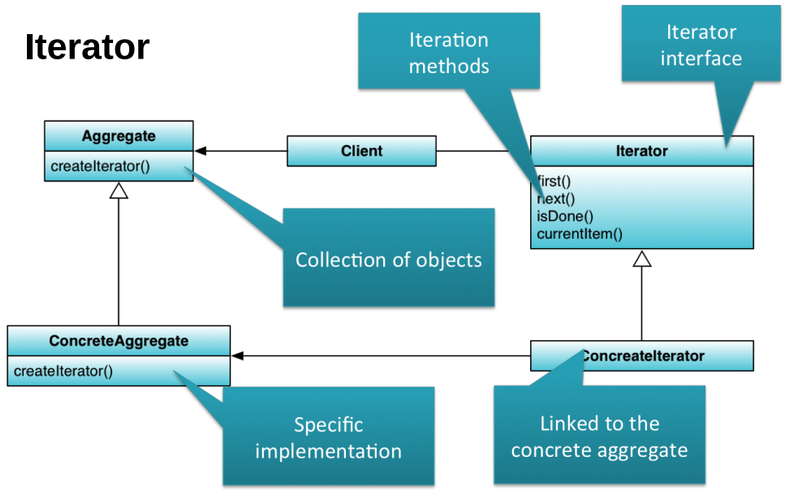
- **Use**
  - **Visit complex object structure** (inheritance)
    - Perform operations based upon concrete classes
  - **Avoid pollution** of concrete classes with many different operations
    - Visitor groups this functionality
  - Ability **to easily define new operations** without changing concrete classes.

Observer



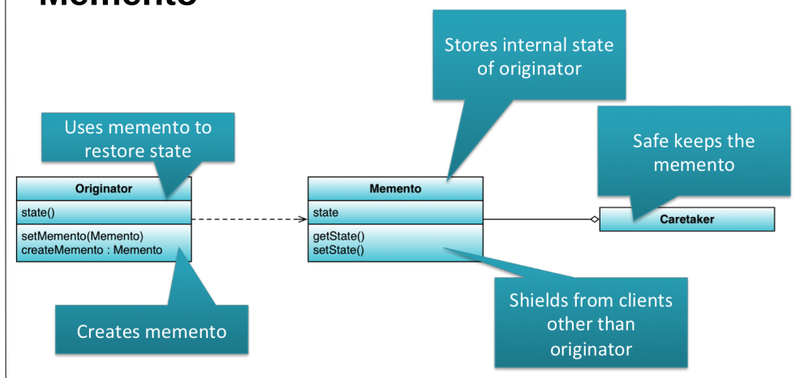
- **Use**
  - **Change one** object → **changes others**
  - No idea how many objects need to be changed
  - Object change notification
    - With preserving loose coupling
  - One object may notify another without knowing them directly

Iterator



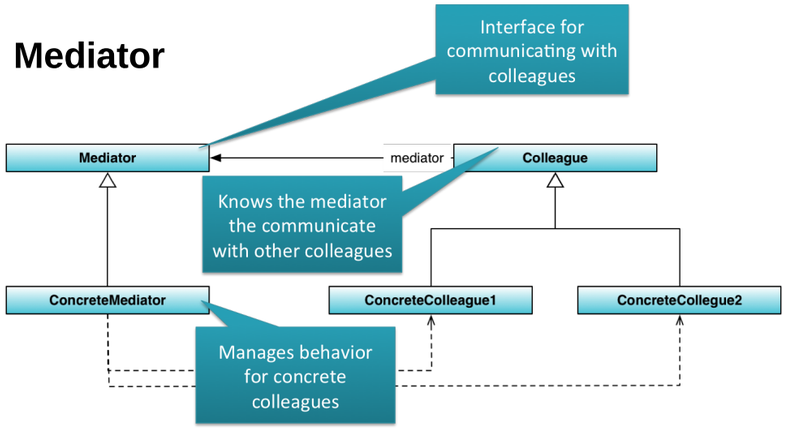
- **Use**
  - **Access aggregated object's** contents **without exposing** its representation
  - Support multiple traversal of aggregated objects
  - Provide **uniform interface** to
    - Traverse aggregated objects
    - Might be of different classes

Memento



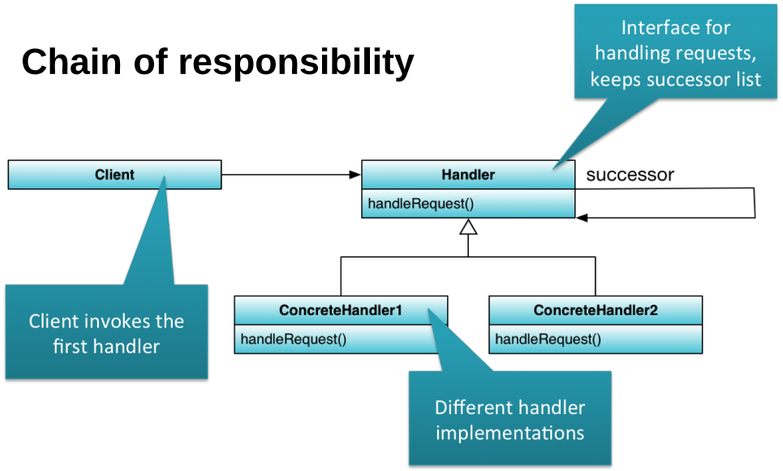
- **Use**
  - **Save** a **snapshot** of an objects state
  - Direct interface to object state would violate encapsulation

Mediator



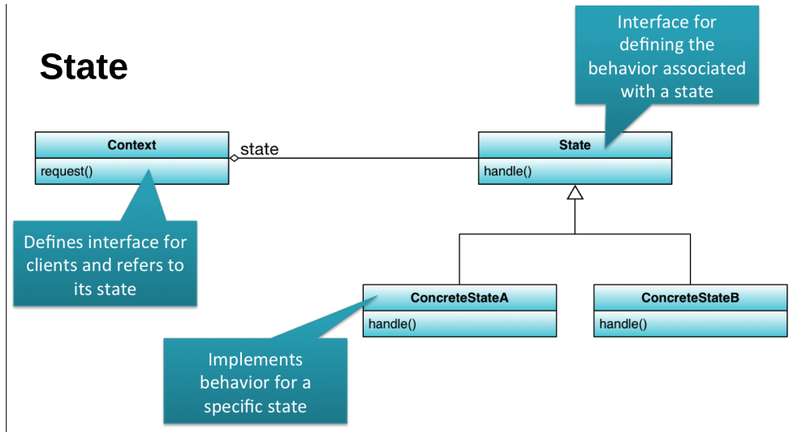
- **Use**
  - Objects have **complex communication**, but it's well defined
  - Hard to identify how the communication actually works
  - Object **re-use** is **difficult**
    - These objects require a set of different objects (waterfall of dependencies)
  - **Centralize behavior** between classes.

Chain of responsibility



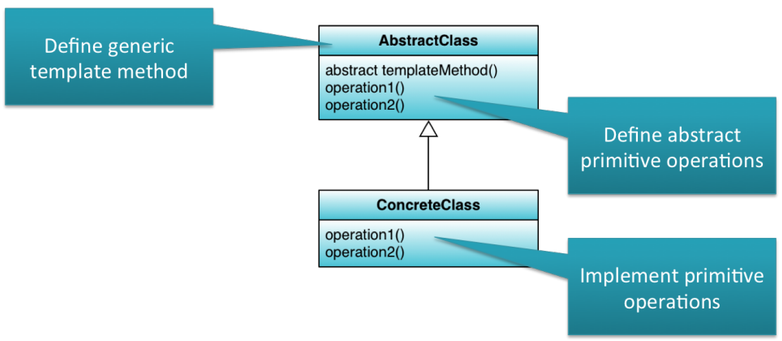
- **Use**
  - **Multiple objects** need to **handle a request**
  - Isn't clear upfront **who will handle it**
  - Who can handle the request should be dynamic

State



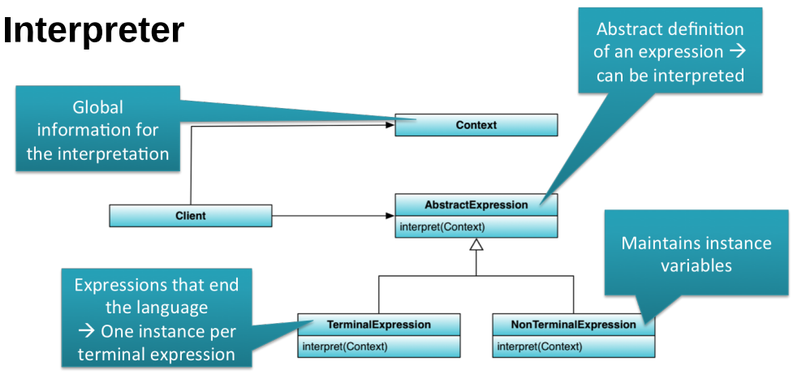
- **Use**
  - Object **behavior depends** on **object state**.
  - **Avoid complex if-else-structures**
    - When state changes, simply change the state object
    - Implementation is done in the state object

Template method



- **Use**
  - **Implement** an **algorithm once**
    - Subclasses can provide different implementation
  - **Avoid code duplication**
  - Define how a class should be extended
    - Implementing hooks

Interpreter

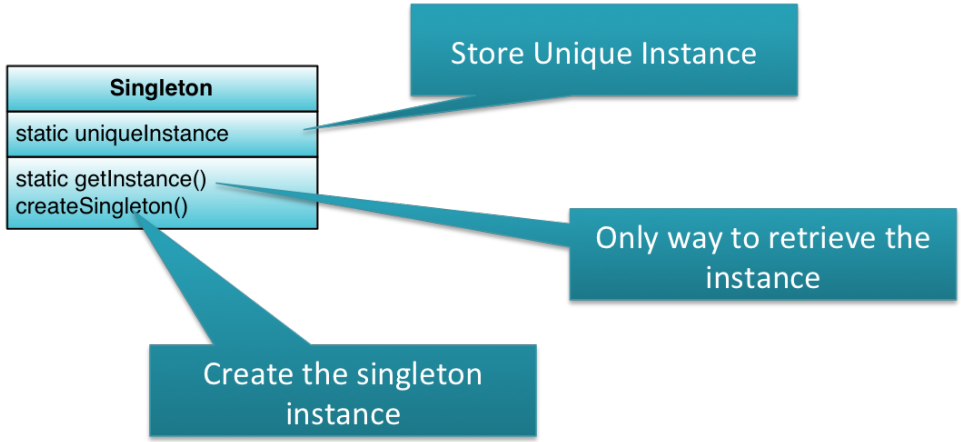


- **Use**
  - **A language needs to be interpreted**
    - The language is simple
  - **Efficiency** is **not critical**
    - Note: languages are normally translated into a state machine

Singleton pattern

Intent

Ensure a class only has **one instance**, and provide a **global point of access to it**.



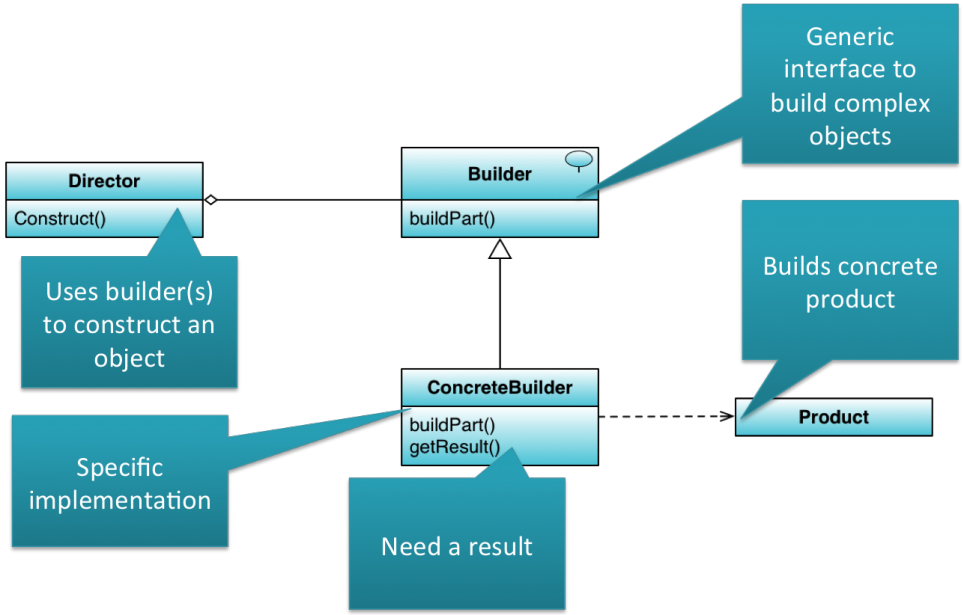
- **When**
  - Only **one instance** of class **required**
  - Must be **one access point**
  - Need to **manage object instances**

- **Benefits**
  - **Controlled access** to one instance
  - Reduce name space → Avoids global variables
  - The ability to subclass the singleton class

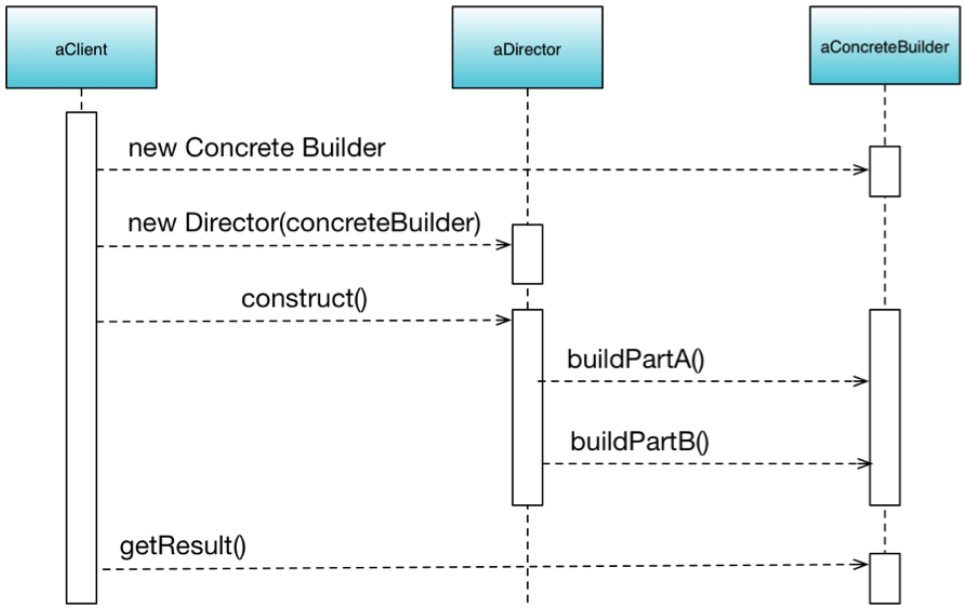
Builder pattern

Intent

Separate the **construction** of a **complex object** from its **representation** so that the **same construction** process can create **different representations**.



- **Use**
  - **Separate construction** with internal representation
  - One process → **multiple object representation**
  - Object construction <> object assembling

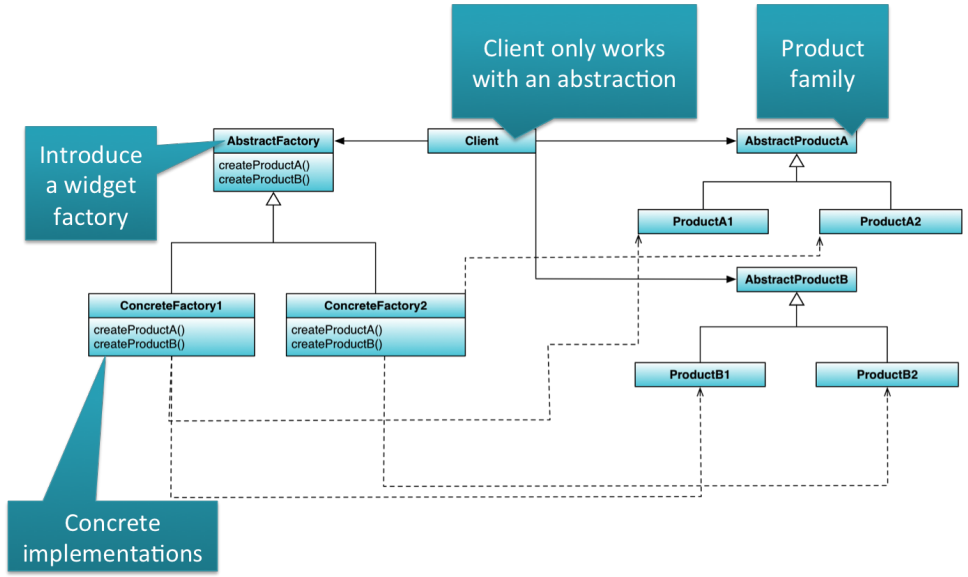


- **Benefits**
  - **Uniform production** creation via an interface
  - Abstract building process
  - **Loose coupling**
    - Construction
    - Representation
  - **Finer control** on the build process → Allow multiple steps

Abstract factory pattern

Intent

Provide an **interface for creating families** of related or dependent objects **without specifying their concrete classes**.



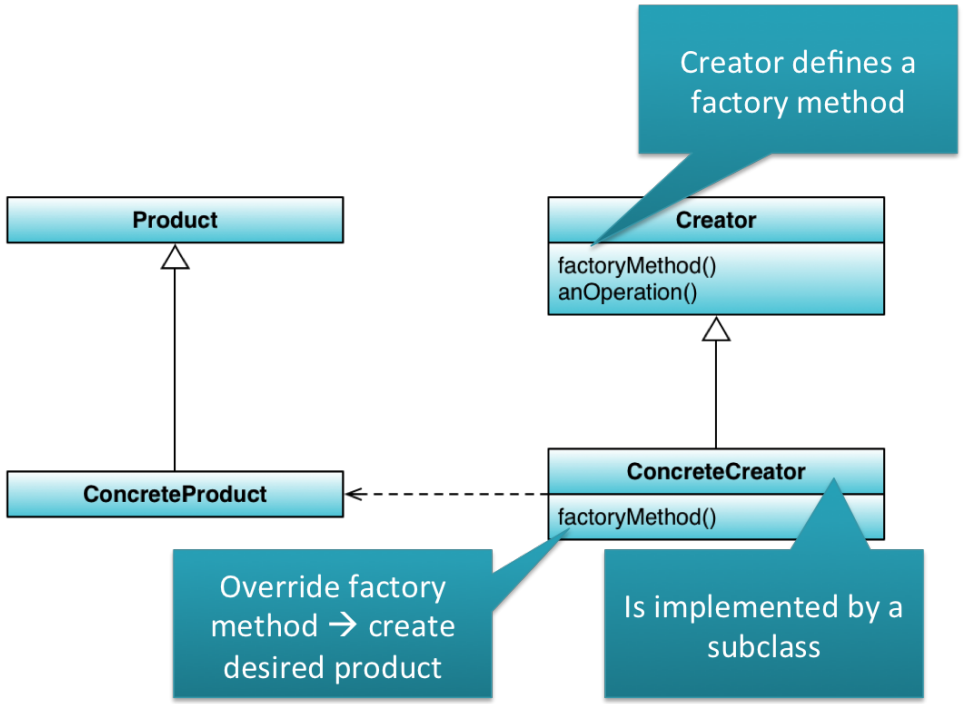
- **Use**
  - **Creation** of products **independent** from the **application**
  - Configuration **of product families** is required
  - Hide product implementation → only provide interface
- **Benefits**
  - **Control** the classes of object to be created
  - Exchanging product families easy
  - Promote **consistency** among products
- **Drawbacks**
  - **Addition** of **new products** is difficult → extend factory interface

Eloo

Factory method pattern

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**.

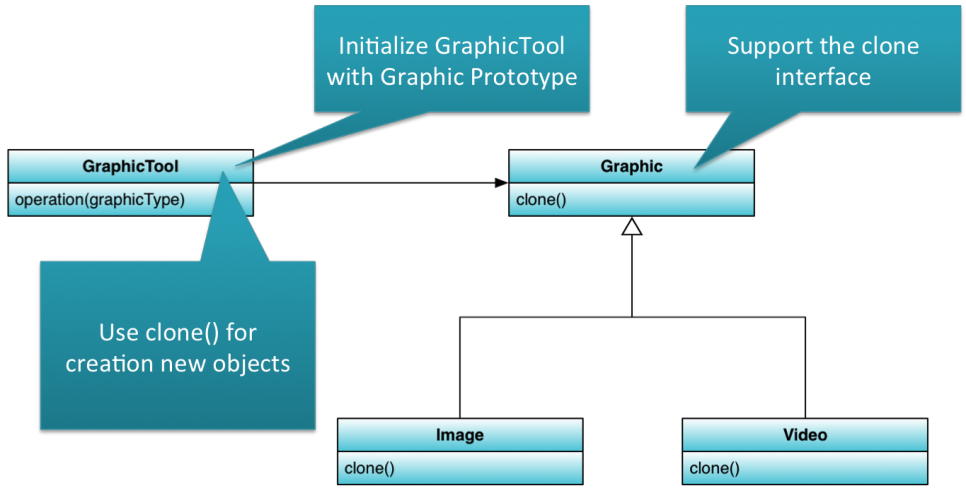


- **When**
  - Class can't **expect** the type of object it must **create**
  - **Subclasses** must decide what types of **objects are created**
- **Benefits**
  - **Delegate** object creation
  - **Hooks** for subclasses
  - Base class can provide a **default implementation**

Prototype pattern

Intent

Specify the kinds of objects to create using a **prototypical instance**, and create new objects by **copying this prototype**.

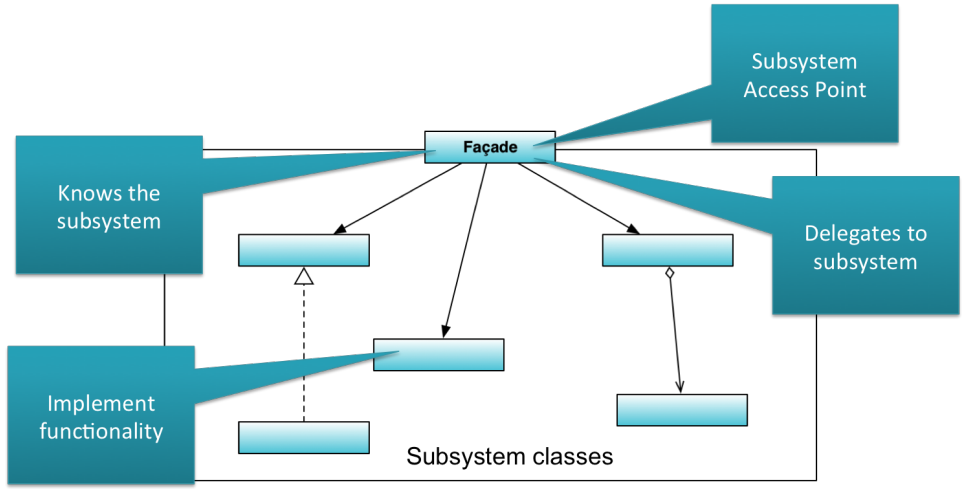


- **When**
  - **Classes** to instantiate are specific at **run-time**
  - Avoid building class hierarchies (abstract factory pattern)
  - A class can have **limited instances** of state
    - Cloning is more efficient
- **Benefits**
  - **Add/remove** products at **runtime**
  - Reduce sub classing (avoid abstract factory)
  - **Configure** application **dynamically**
- **Drawbacks**
  - Requires to create prototypes before other object creation

Façade pattern

Intent

Provide a **unified interface** to a set of interfaces in a subsystem. Façade defines a **higher-level interface** that makes the **subsystem easier to use**.



- **Use**
  - **Decouple** clients from subsystems
  - Provide **simple interface**
  - Subsystem layering (business, data and client services)

- **Benefits**
  - Subsystem **easier** to use
    - Client don't require specific knowledge
  - Loose coupling
  - Subsystem **can** still be **used directly** (if necessary)
- **Drawbacks**
  - Façade introduces an extra programming layer

Eloo

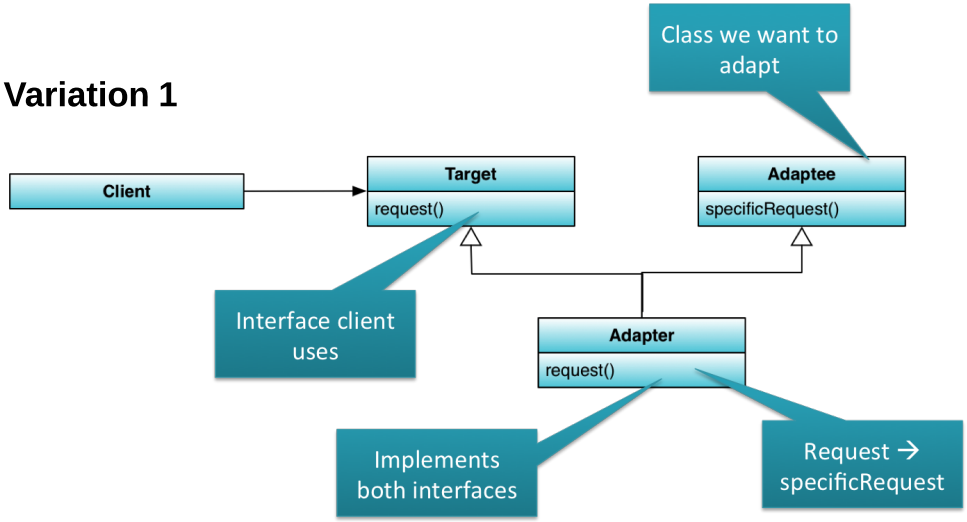
Adapter pattern

Intent

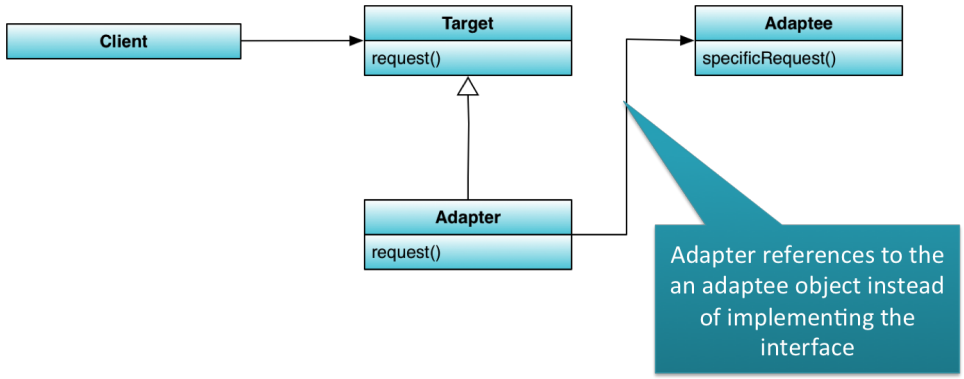
Provide a **unified interface** to a set of interfaces in a subsystem. Façade defines a **higher-level interface** that makes the **subsystem easier to use**.

- **Use**
  - **Re-use** an existing class
  - **Combine** unrelated **classes** with an incompatible interface
- **Benefits**
  - Adapter can **override** adaptee **behaviour** → it is a subclass
  - One adapter → Many adaptees
- **Drawbacks**
  - Adapter doesn't work for
    - Class with many subclasses → can't extend them all
  - Harder to override adaptee behaviour (do we want this?)

Variation 1



Variation 2

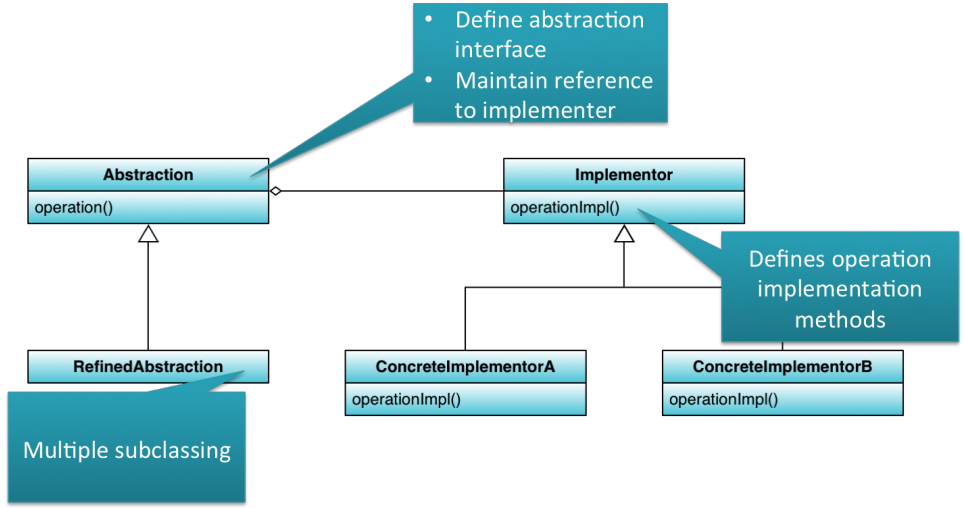


Eloo

Bridge pattern

Intent

Decouple an **abstraction** from its **implementation** so that the two can vary independently.



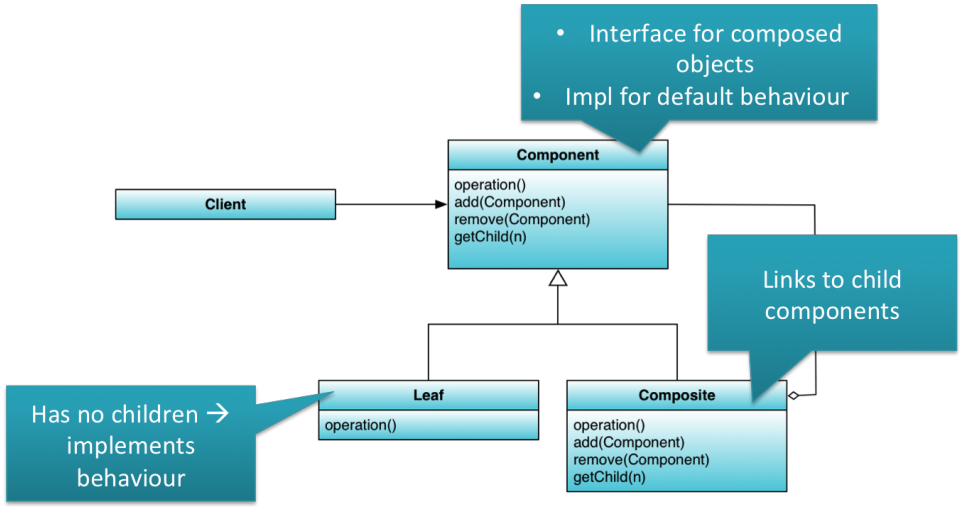
- **Use**
  - **Avoid binding** between interface and implementation
  - Possible subclasses for abstraction and implementation
  - Must be possible to **change implementation at runtime** without affecting clients
- **Benefits**
  - **Decouple** interface and implementation
  - Improve **extensibility**
  - Hide implementation details

Eloo

Composite pattern

Intent

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



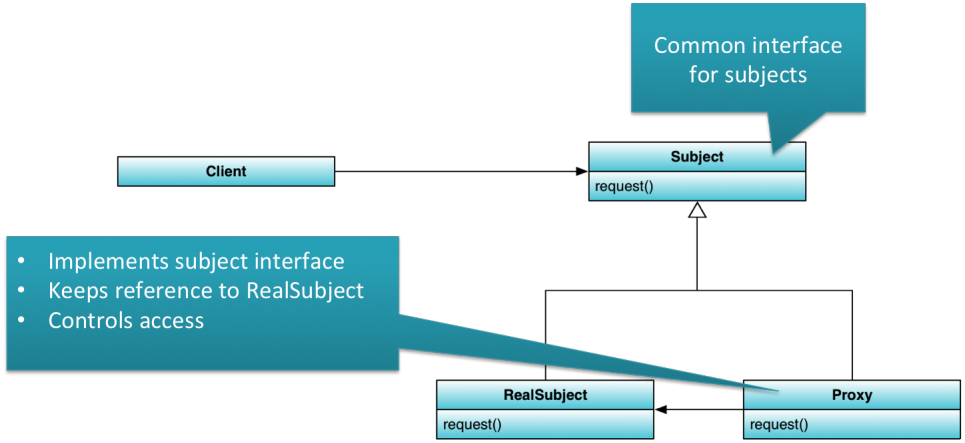
- **Use**
  - **Ignore differences** between **compositions** and **individual items**
  - **Represent** part-whole **hierarchies of objects**
- **Benefits**
  - **Define class hierarchies** of consistent objects
  - Client is simplified
    - **No distinction** between child and composite objects
  - Easy to add new components
- **Drawbacks**
  - Can be hard for new components with different expectations

Eloo

Proxy pattern

Intent

Provide a **surrogate** or **placeholder** for another object to control access to it.



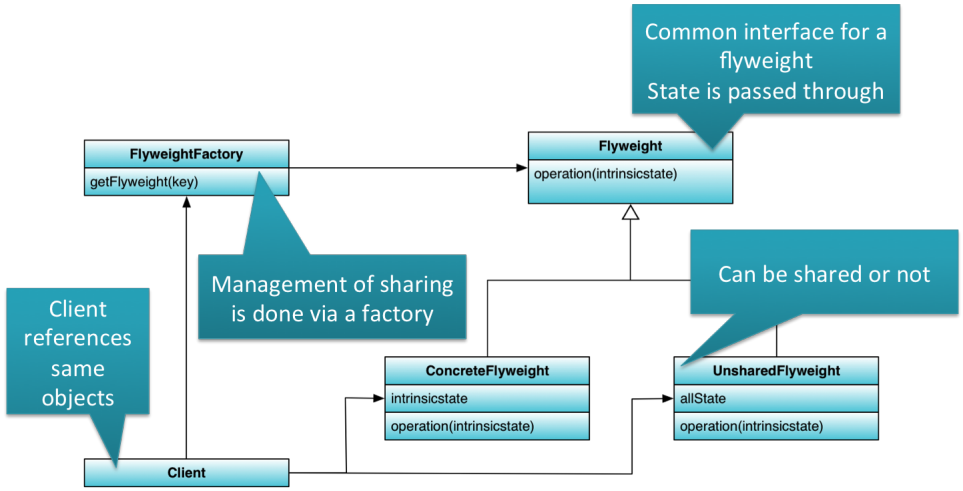
- **Use**
  - Extra functionality is required
    - **Transparancy**
    - More than just a reference
- **Benefits**
  - Adds **indirection**
  - Can be beneficial in many cases
    - Remote proxy
    - Virtual proxy
    - Smart references

Eloo

Flyweight pattern

Intent

Use Sharing to support large numbers of fine-grained objects efficiently.

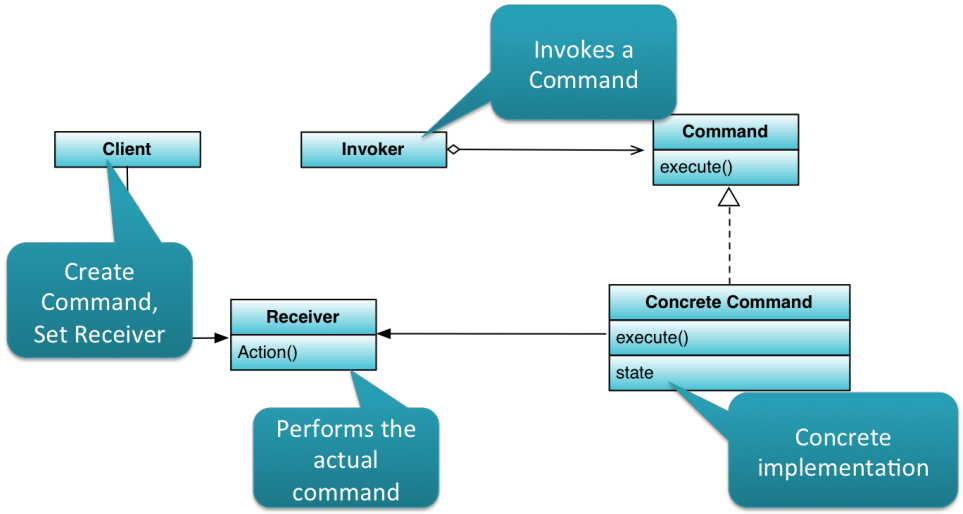


- **Use**
  - **Large number** of objects
  - High **storage costs**
  - Extrinsic state (**shareable**)
  - Many objects → replaced by few objects
  - Object identity isn't necessary
- **Benefits**
  - **Reduce** the total number of **instances**
  - Share intrinsic state per object
- **Drawbacks**
  - Factory needs to manage shared instances → run-time overhead

Command pattern

Intent

Encapsulate a request as an object, thereby letting you **parameterize clients** with different **requests**, **queue** or **log requests**, and support **undoable operations**



- **Use**
  - Command as **parameter**
  - Pass command like general object
  - **Queue Request**
  - **Save** request state
  - Undo functionality
  - Provide an execute and undo method
  - Support Logging
  - **Re-execute** code in case of failure

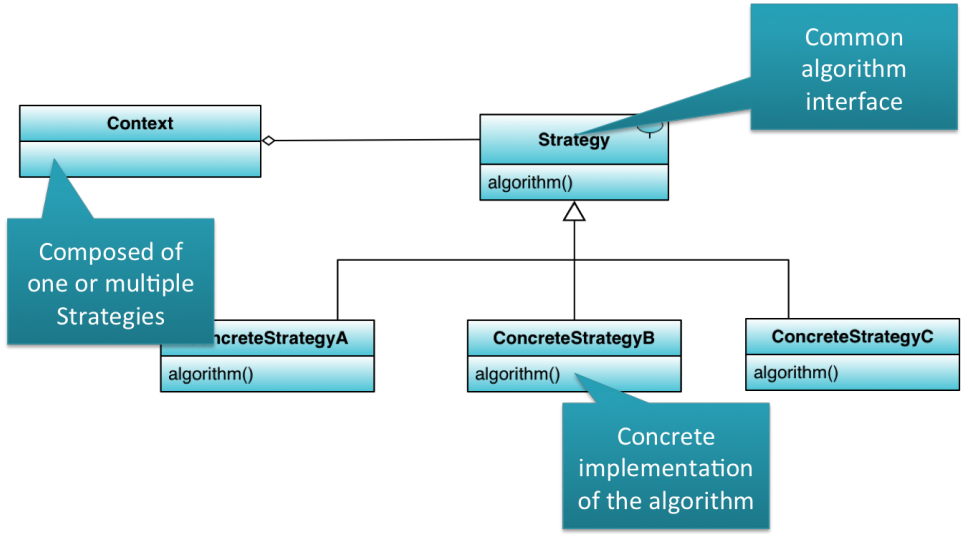
- **Benefits**
  - **Decoupling** between
    - Object that invokes the operation
    - Object that knows how to perform it
  - Commands are **Objects**
    - All OO rules apply to them
  - Creating **new Commands** is easy
    - Just add a new class

Eloo

Strategy pattern

Intent

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



- **Use**
  - Classes only change in behavior
  - Different variants of an **algorithm**
  - Algorithms that **use complex data** that clients shouldn't be aware of

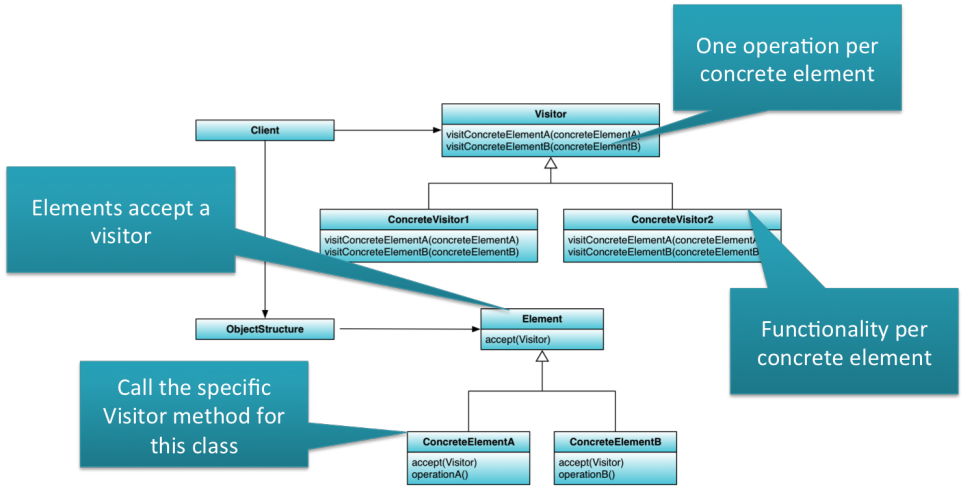
- **Benefits**
  - Algorithm families use inheritance for common parts
  - Avoid conditional statements using this pattern
  - **Clients** can **choose** the required **behavior**
- **Drawbacks**
  - Pattern increases nr of objects in application
    - Share strategies between objects.

Eloo

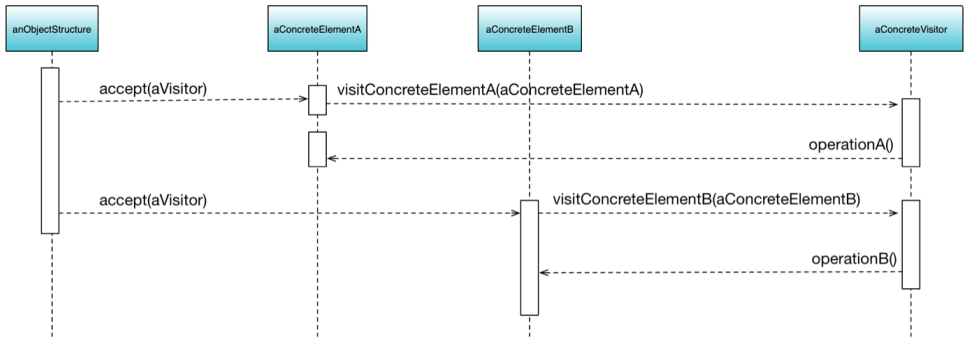
Visitor pattern

Intent

Represent an **operation** to be **performed** on the **elements** of **an object structure**. Visitor lets you **define a new operation** without changing the classes of the elements on which it operates.



- **Use**
  - **Visit complex object structure** (inheritance)
    - Perform operations based upon concrete classes
  - **Avoid pollution** of concrete classes with many different operations
    - Visitor groups this functionality
  - Ability **to easily define new operations** without changing concrete classes.



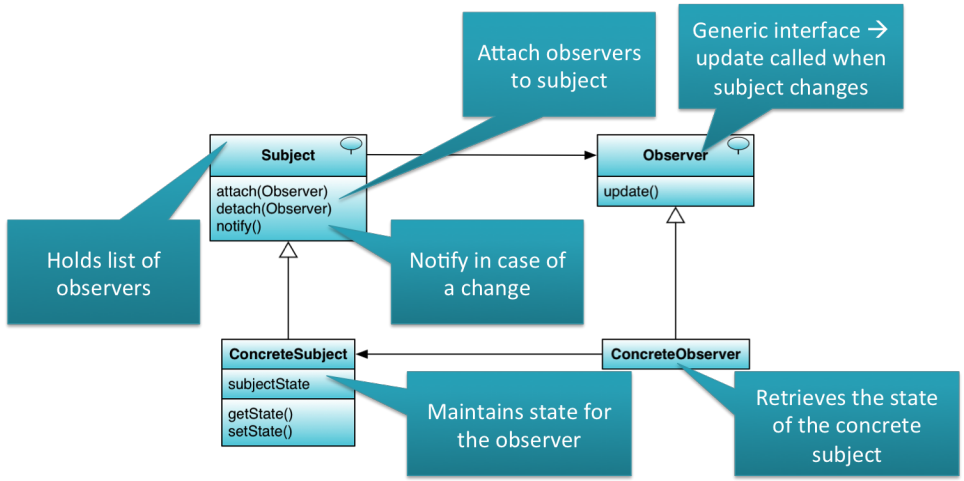
- **Benefits**
  - Adding **new operations** is **easy**
  - Visitor separates operations that don't belong together
  - **Accumulate state**
    - Visitors can maintain state across the hierarchy
- **Drawbacks**
  - **Adding** new **concrete elements** is **hard**
    - Requires a new method for all concrete visitors
  - Visitors **break encapsulation**
    - They rely on the interface of the concrete element

Eloo

Observer pattern

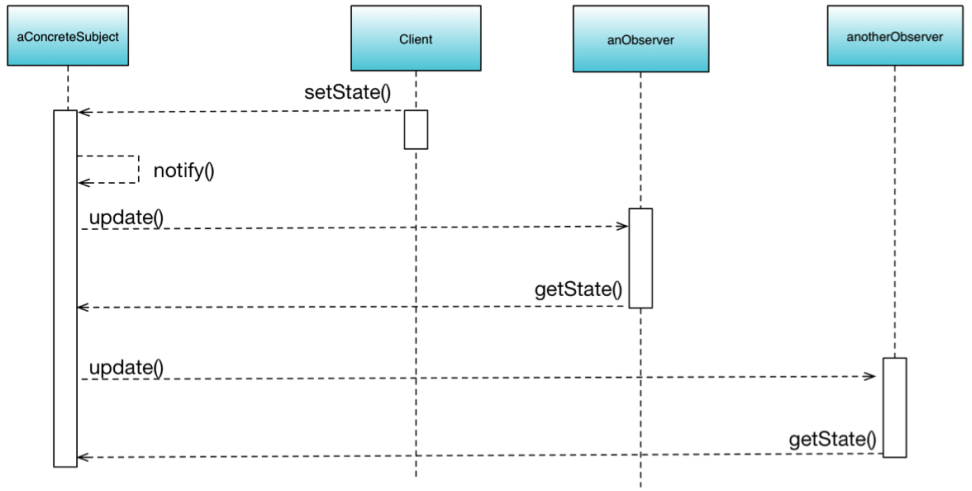
Intent

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



- **Use**
  - **Change one** object → **changes others**
  - No idea how many objects need to be changed
  - Object change notification
    - With preserving loose coupling
  - One object may notify another without knowing them directly

- **Benefits**
  - **Loose coupling** between observers and subjects
  - Supporting a **broadcast model**
- **Drawbacks**
  - One change can result in multiple **unnecessary notifications**
  - Clients don't know the **ripple effects**

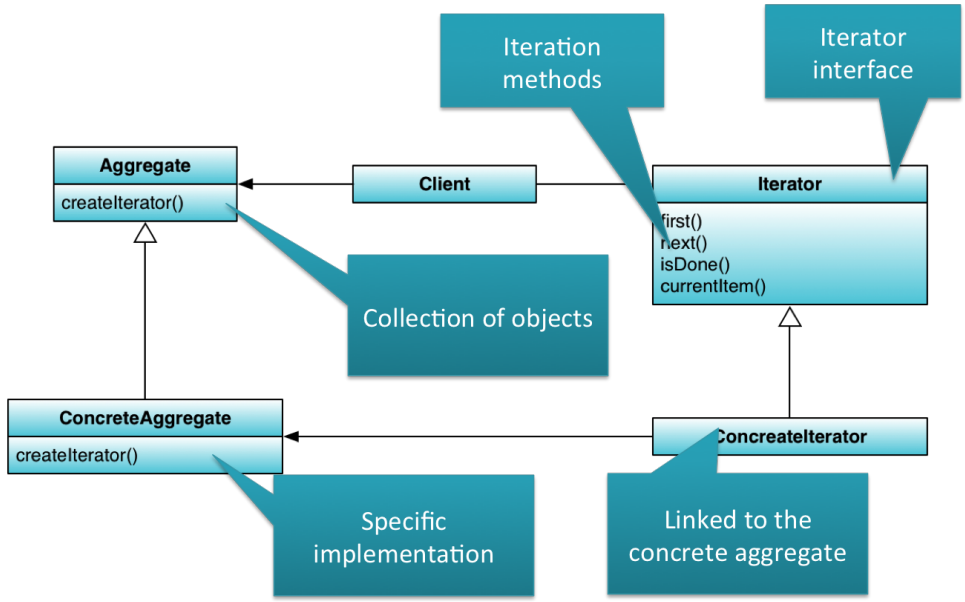


Eloo

Iterator pattern

Intent

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

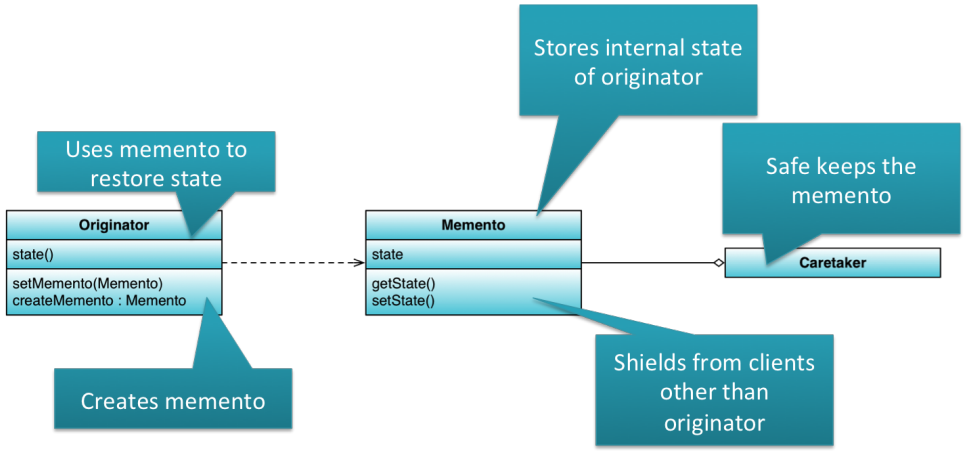


- **Use**
  - **Access aggregated object's** contents **without exposing** its representation
  - Support multiple traversal of aggregated objects
  - Provide **uniform interface** to
    - Traverse aggregated objects
    - Might be of different classes
- **Benefits**
  - **Multiple iteration** variations are possible
  - Aggregate or Collection **interface is simplified**
  - Each iterator keeps track where he is
    - Use **different iterators** to traverse an object **at same time**

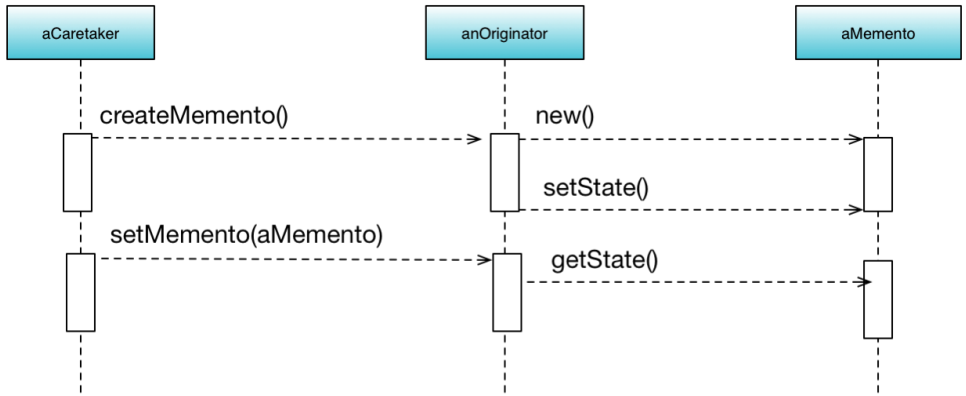
Memento pattern

Intent

Without violating encapsulation, **capture and externalize an object's internal state** so that the object can be **restored** to this state later.



- **Use**
  - **Save** a **snapshot** of an objects state
  - Direct interface to object state would violate encapsulation



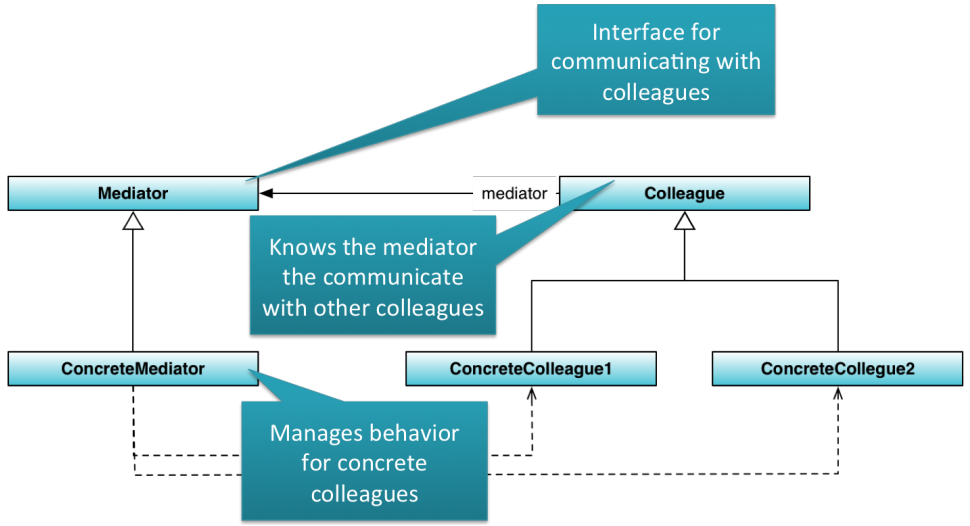
- **Benefits**
  - **Preserve encapsulation**
  - Simplifies originator
- **Drawbacks**
  - Might be **expensive**
    - **Object creation**
  - Language must facilitate that only the originator can access the memento's state
  - **Extra management** for caretaking the mementos

Eloo

Mediator pattern

Intent

Define an object that **encapsulates** how a **set of objects interact**. Mediator **promotes loose coupling** by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.



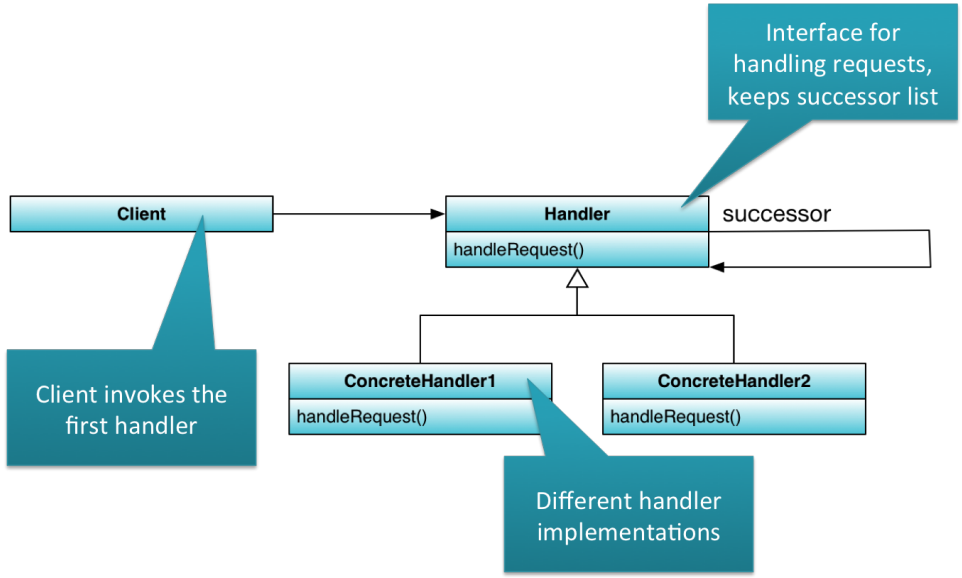
- **Use**
  - Objects have **complex communication**, but it's well defined
  - Hard to identify how the communication actually works
  - Object **re-use** is **difficult**
    - These objects require a set of different objects (waterfall of dependencies)
  - **Centralize behavior** between classes.

- **Benefits**
  - **Centralize behavior**
    - Would otherwise be distributed among other objects
  - **Decoupling** of colleagues
  - Changes **many-to-many** interaction **to one-to-many** interaction
  - Object collaboration is abstracted
  - **Centralized control**

Chain of responsibility pattern

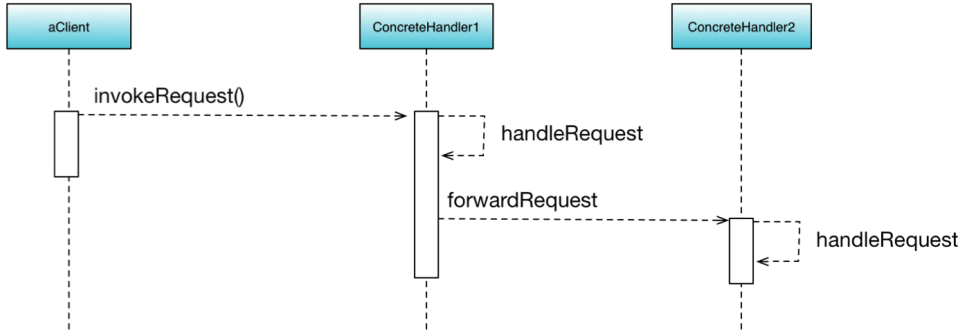
Intent

Avoid coupling the **sender** of a **request** to its **receiver** by giving **more than one object** a change to **handle the request**. Chain the receiving objects and pass the request along the chain until an object handles it.



- **Use**
  - **Multiple objects** need to **handle a request**
  - Isn't clear upfront **who will handle it**
  - Who can handle the request should be dynamic

- **Benefits**
  - **Loose coupling** between **requester** and **receiver**
  - Objects can **spread responsibility** in handling requests.
- **Drawbacks**
  - No guarantee the **request** will be **handled**

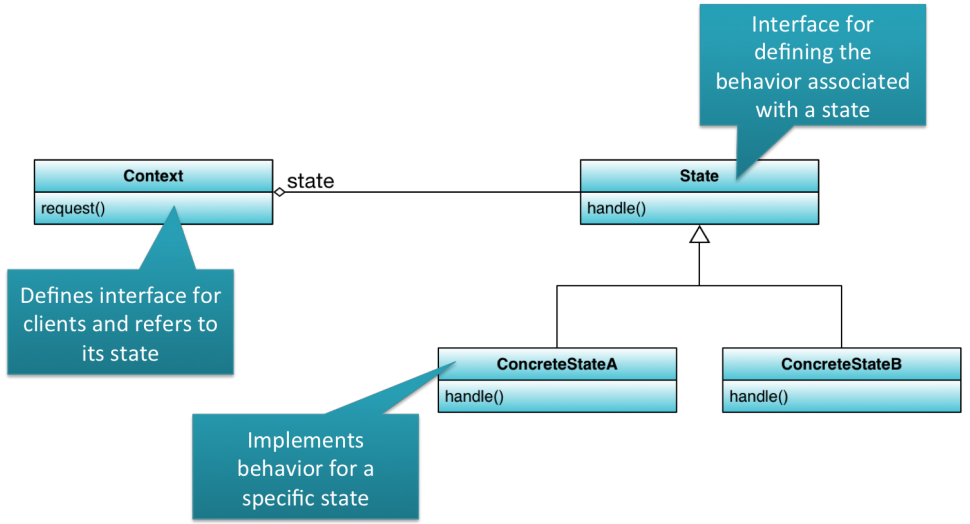


Eloo

State pattern

Intent

Allow an object **to alter its behavior** when its internal **state changes**. The object will appear to change its class.



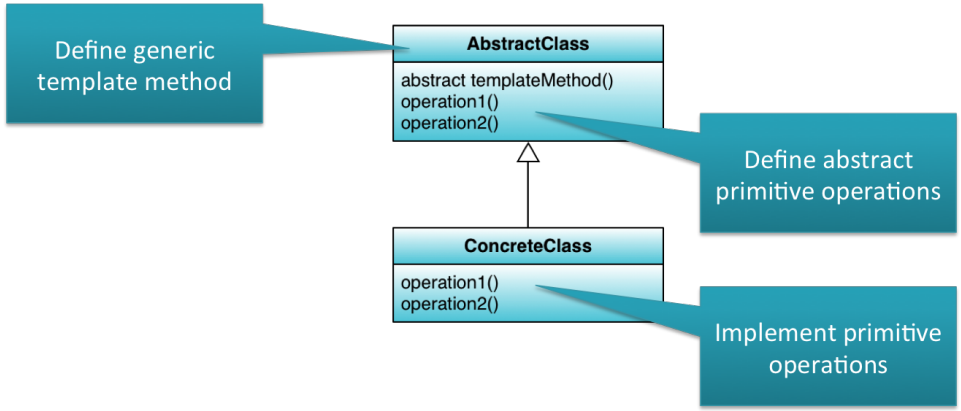
- **Use**
  - Object **behavior depends** on **object state**.
  - **Avoid complex if-else-structures**
    - When state changes, simply change the state object
    - Implementation is done in the state object
- **Benefits**
  - State related **behavior** is **centralized**
  - State **transitions** are **explicit** → state object must be changed
  - State **changes** can be **atomic** → only one variable in the Context
- **Drawbacks**
  - State objects must be **shareable**

Eloo

Template method pattern

Intent

Define the **skeleton of an algorithm** in an operation, deferring some steps to subclasses. Template Method **lets subclasses redefine certain steps** of an algorithm without changing the algorithm's structure.



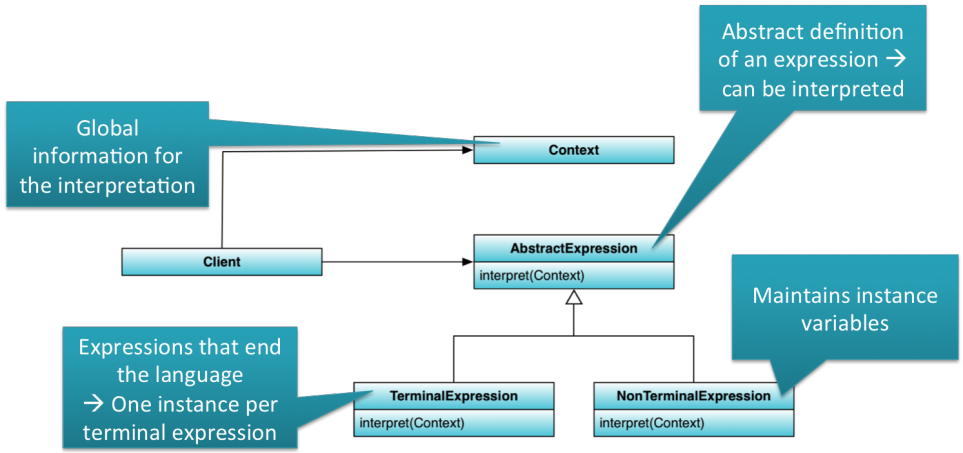
- **Use**
  - **Implement** an **algorithm once**
    - Subclasses can provide different implementation
  - **Avoid code duplication**
  - Define how a class should be extended
    - Implementing hooks

- **Benefits**
  - Ultimate **code re-use**
    - Important for class libraries
  - **Provide hooks**
  - The abstract base class generally doesn't implement a hook → forces clients to have an implementation

Interpreter pattern

Intent

Given a **language**, **define a representation** for its grammar along with an interpreter that uses the representation to **interpret sentences in the language**



- **Use**
  - **A language needs to be interpreted**
    - The language is simple
  - **Efficiency** is **not critical**
    - Note: languages are normally translated into a state machine
- **Benefits**
  - **Easy to change** or **extend** the grammar
  - New interpretation expressions can be added easily
    - E.g. a pretty print
- **Drawbacks**
  - Hard for complex grammar

Eloo