Introduction to Software Engineering

ECSE-321

Unit 16 – Design Patterns (Part 2)

Generator Patterns

- Have a client who needs a new instance of a product, and a *generator* class that supplies the instance
  - Factory patterns
  - Abstract factory patterns
  - Singleton patterns
  - Prototype patterns

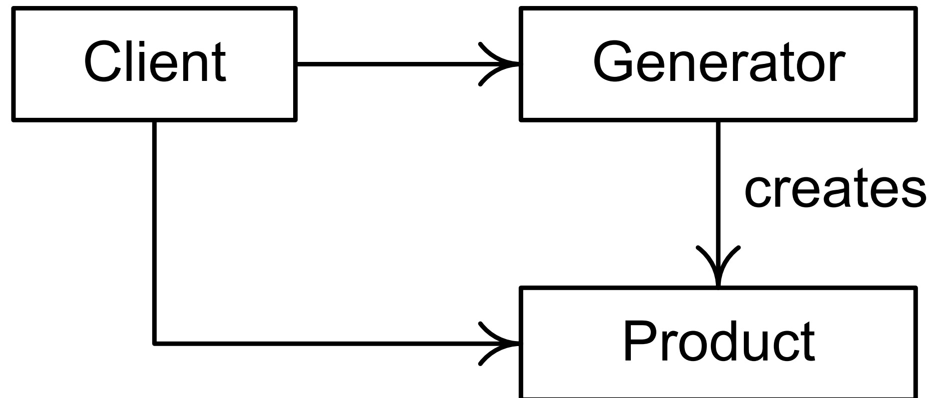
For generator patterns…

- To present the structure, behavior, and characteristics of generator patterns
- To discuss the uses of factory methods
- To present the Factory Method and Abstract Factory design patterns

Instance Creation

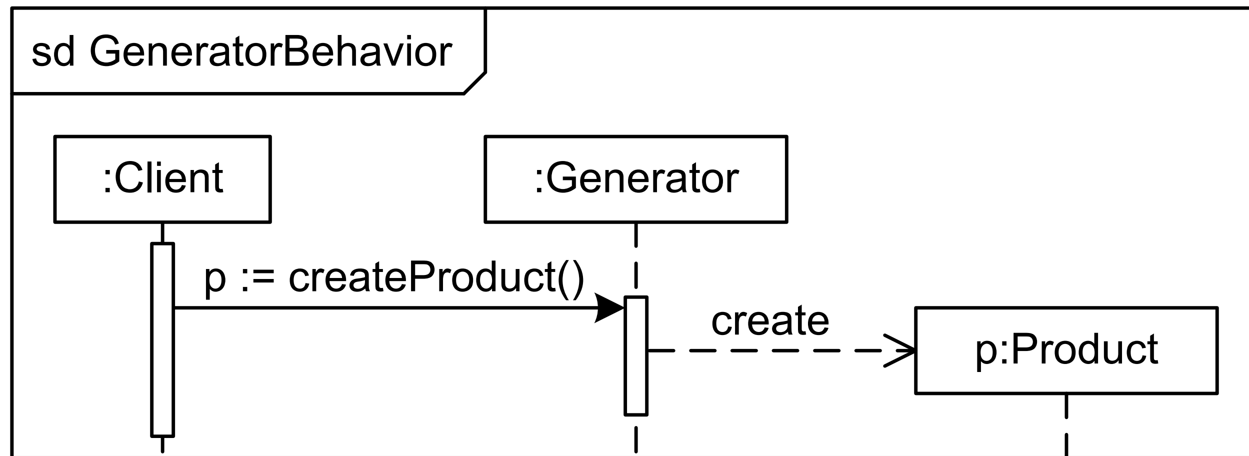
- There are two ways to create objects:
  - Instantiating a class using one of its constructors
  - Cloning an existing object
- Clients may use another class to create an instance on their behalf; this is the essence of the generator pattern category.
- Analogy: a tailor

Generator Pattern Structure



The Client must access the Generator that creates an instance of the Product and provides it to the Client

Generator Pattern Behavior



Generator Pattern Advantages

- *Product Creation Control*—A generator can mediate access to constructors so that only a certain number or type of product instances are created.
- *Product Configuration Control*—A generator can take responsibility for configuring product instances.
- *Client and Product Decoupling*—A generator can determine how to create product instances for a client.

Factory Methods

A Generator must have an operation that creates and returns Product instances.

A **factory method** is a non-constructor operation that creates and returns class instances.

Factory Method Capabilities

- Access to product constructors can be restricted.
- Private data can be provided to new product objects.
- Product objects can be configured after creation.
- Product class bindings can be deferred until runtime.

The Factory Patterns

- Factory patterns configure participating classes in certain ways to decouple the client from the product.
- Interfaces are used to
  - Change the generator
  - Change the product instances
- Analogy: automobile factories

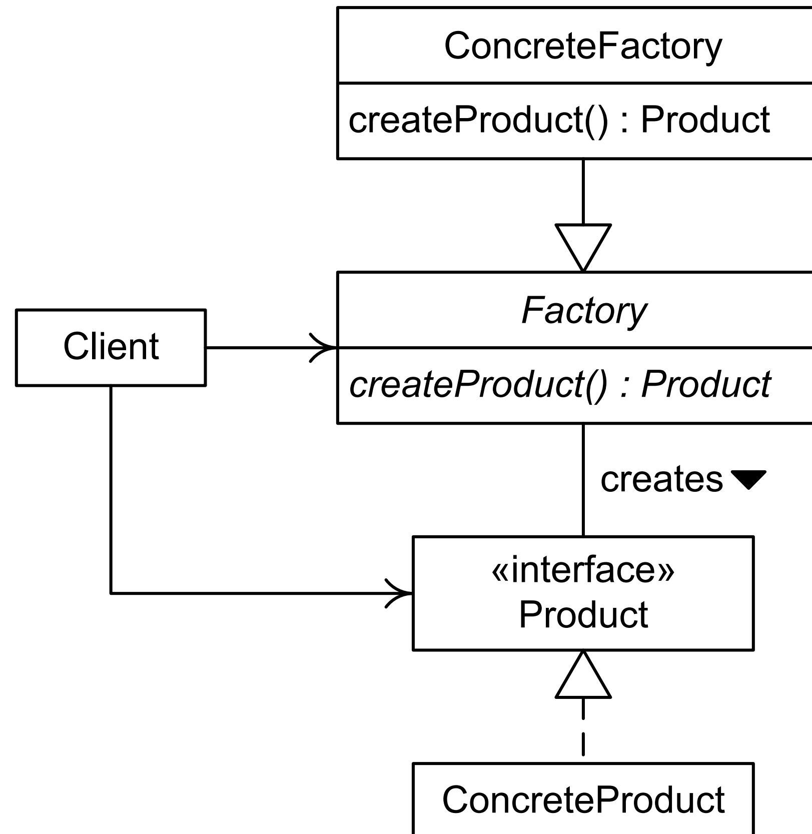
The Factory Patterns

- **Factory Method**—Uses interfaces and abstract classes to decouple the client from the generator class and the resulting products.
- **Abstract Factory**—Has a generator that is a container for several factory methods, along with interfaces decoupling the client from the generator and the products.

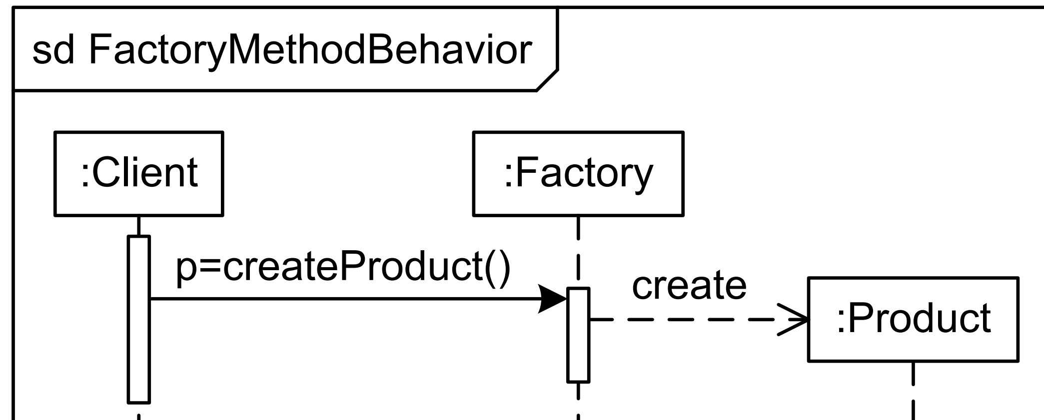
The Factory Method Pattern

- The generator usually contains both factory methods and other methods.
- Analogy: different auto factories producing the same kind of automobile (SUVs for example).

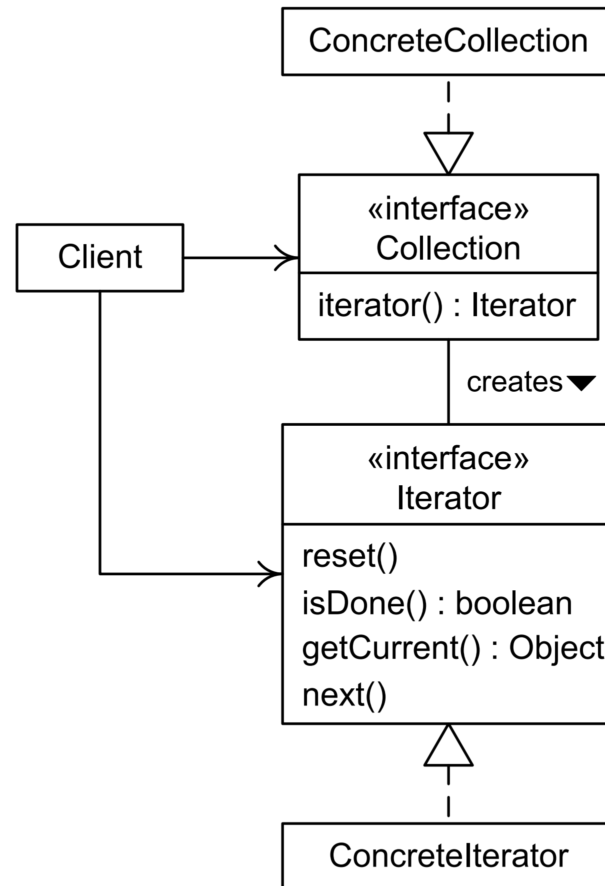
Factory Pattern Structure



Factory Method Behavior



The Iterator and Factory Method Patterns



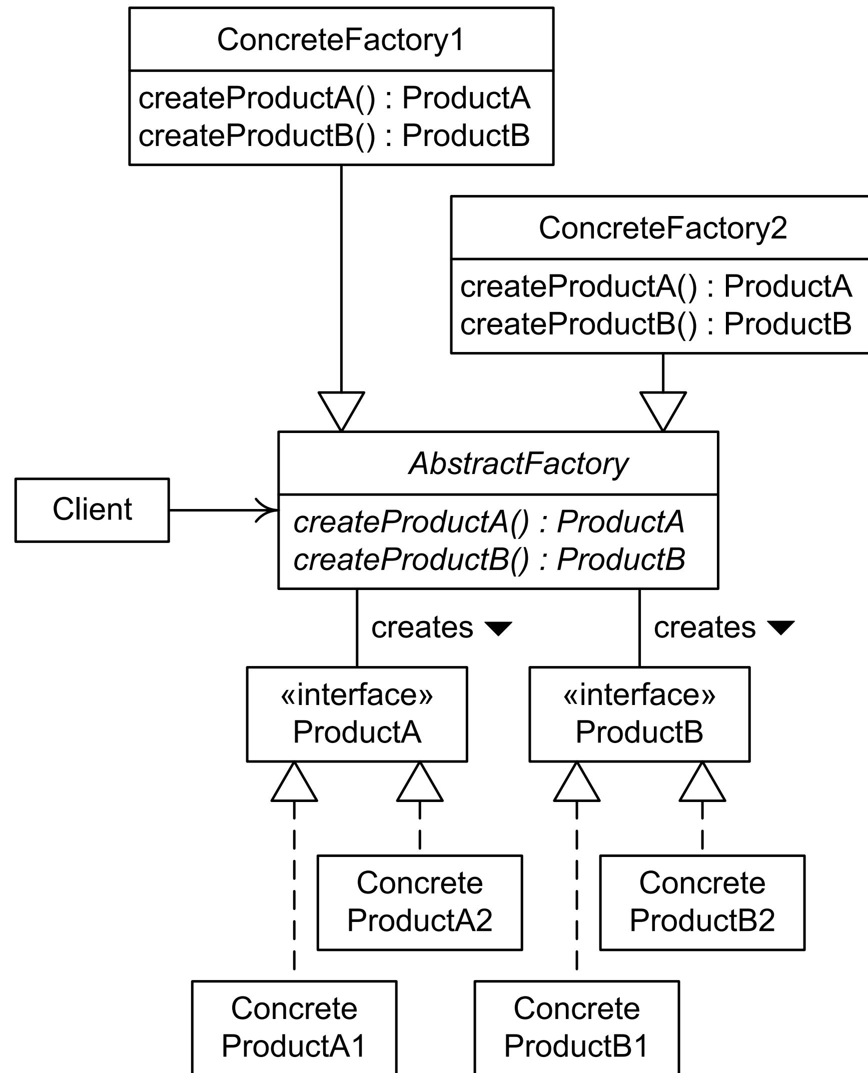
When to Use the Factory Method Pattern

- Use the Factory Method pattern when there is a need to decouple a client from a particular product that it uses.
- Use the Factory Method to relieve a client of responsibility for creating and configuring instances of a product.
- With Iterator, client does not need to know how the collection is organized

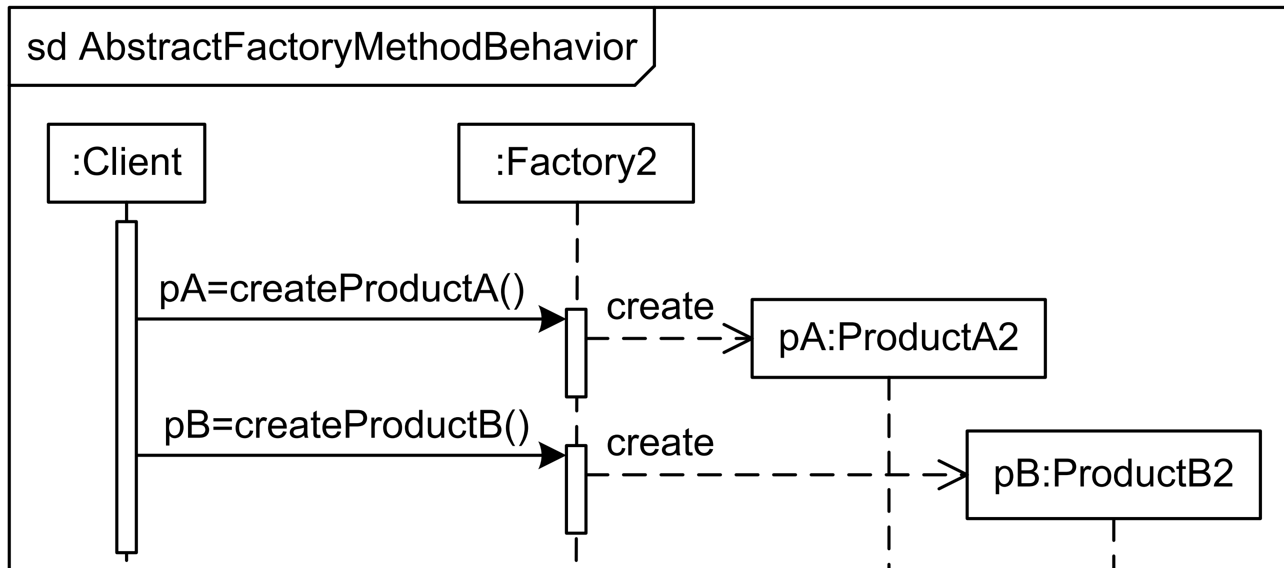
The Abstract Factory Pattern

- A **factory class** is one that contains only factory methods for different (though usually related) products.
- The Abstract Factory generator class is a factory class.
  - Restricts the Factory Method pattern because the generator holds only factory methods
  - Generalizes the Factory Method pattern because the generator creates several different kinds of product
- Analogy: auto factory with assembly lines for different kinds of vehicles

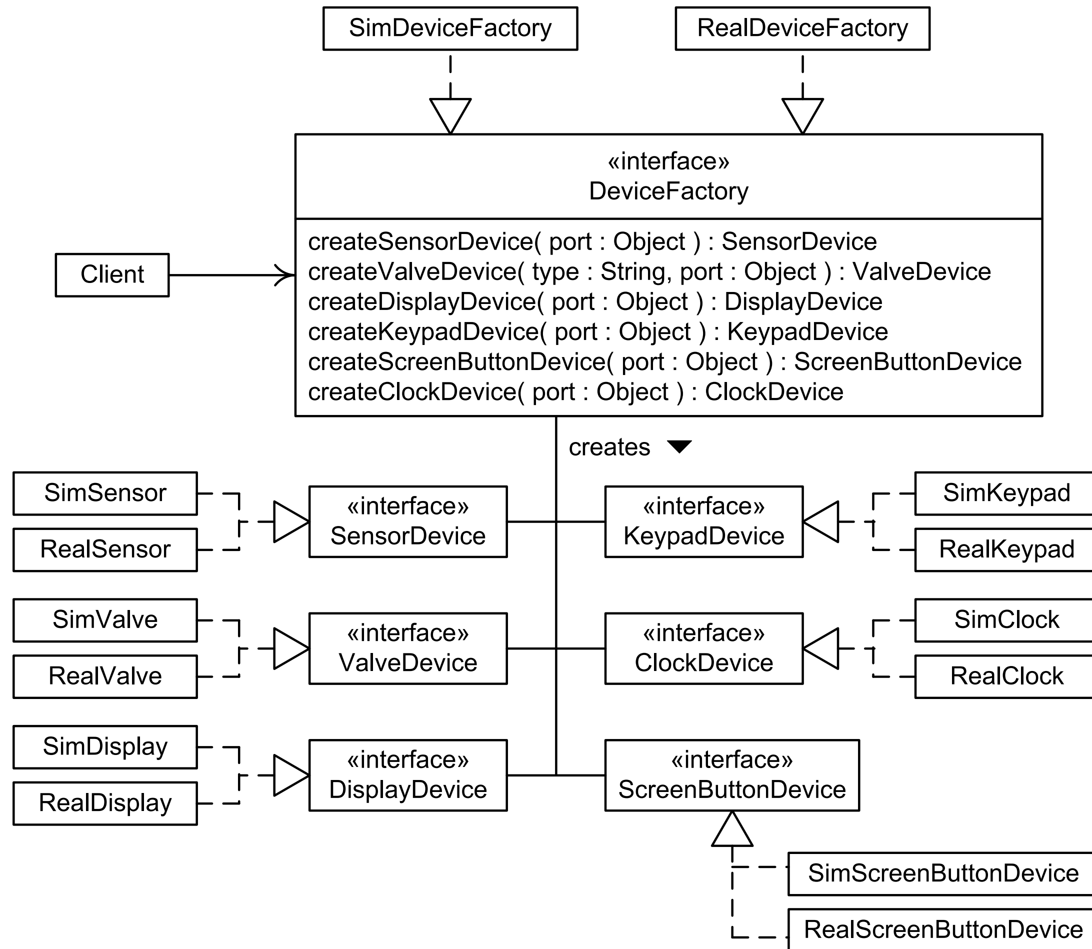
Abstract Factory Pattern Structure



Abstract Factory Pattern Behavior



Abstract Factory Pattern Example



When to Use the Abstract Factory Pattern

- Use the Abstract Factory pattern when clients must be decoupled from product classes.
  - Especially useful for program configuration and modification
- The Abstract Factory pattern can also enforce constraints about which classes must be used with others.
- It may be a lot of work to make new concrete factories.

Singletons

Often there is a need to guarantee that a class has only a single (or a few) instances.

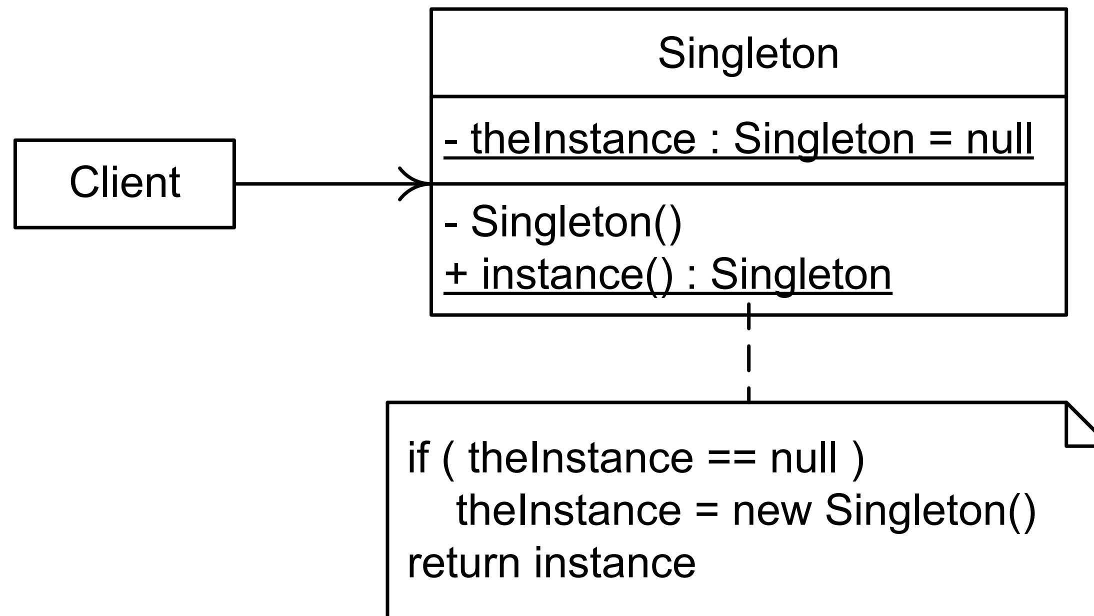
- Servers
- Controllers
- Managers

A **singleton** is a class that can have only one instance.

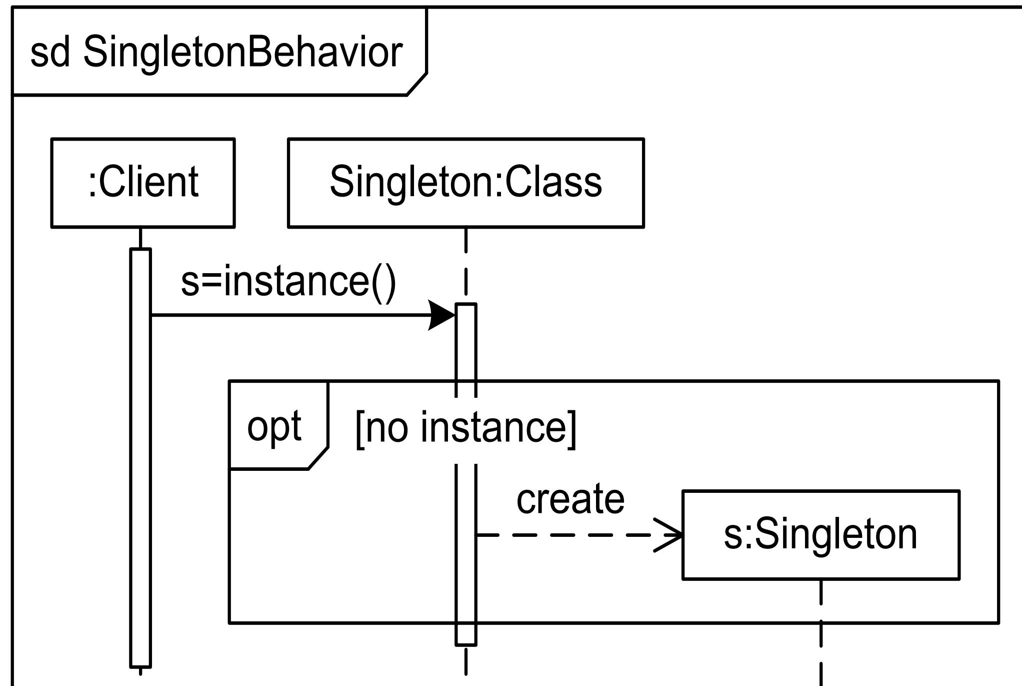
The Singleton Pattern

- Guarantees that a class is a singleton
  - Can be modified to provide any set number of instances
- Provides wide (often global) access to the single instance
  - Can be modified to provide more restricted access

Singleton Pattern Structure



Singleton Pattern Behavior



Singleton Examples

Examples of the need for global unique entities in a program abound:

- Subsystems (or their façade objects)
- Communication streams
- Major containers or aggregates
- OS or windowing system proxies

When to Use Singletons

- Use the Singleton pattern to guarantee that there is only one or a small number of instances of some class.
- Sometimes an abstract class with static attributes and operations can be used instead

Cloning

An alternative to class instantiation is making a copy of an object.

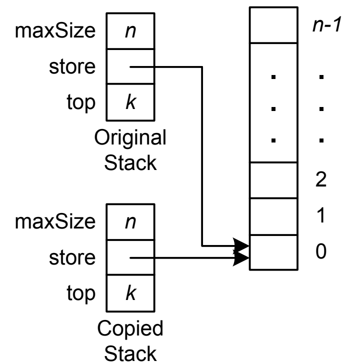
A **clone** is a copy of an object.

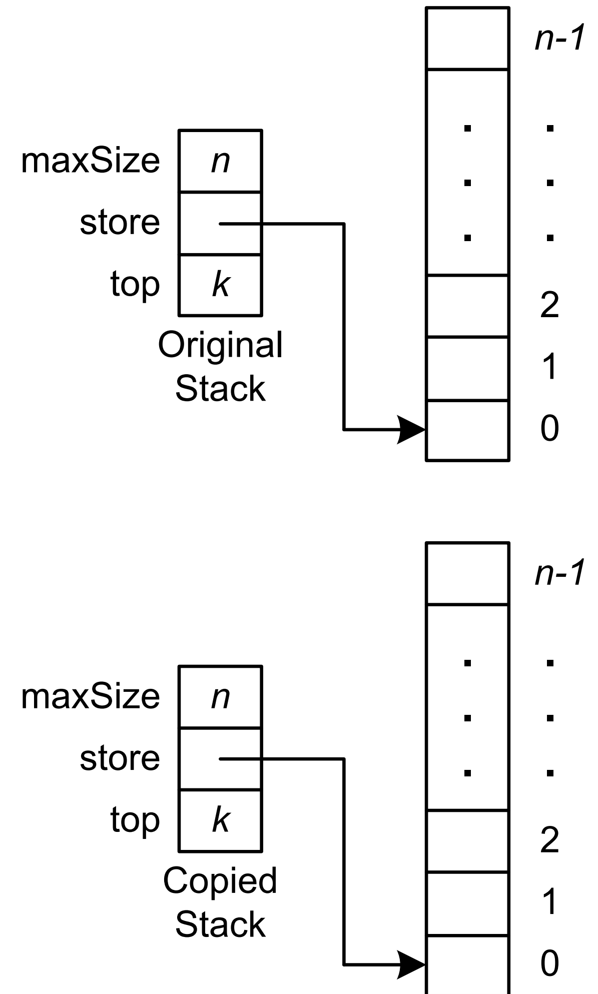
A clone has the state of its source at the moment of creation.

Shallow and Deep Copies

- Copying objects that hold references to other entities raises the question "Should the references or the referenced entities be copied?"
  - *Shallow copy*: Copy references when an entity is copied.
  - *Deep copy*: Copy referenced entities (and any entities they reference as well) when an entity is copied.
- Sometimes a shallow copy is the right thing to do, and sometimes a deep copy is.

Shallow vs. Deep Copy Example





Shallow or Deep Copy in Cloning?

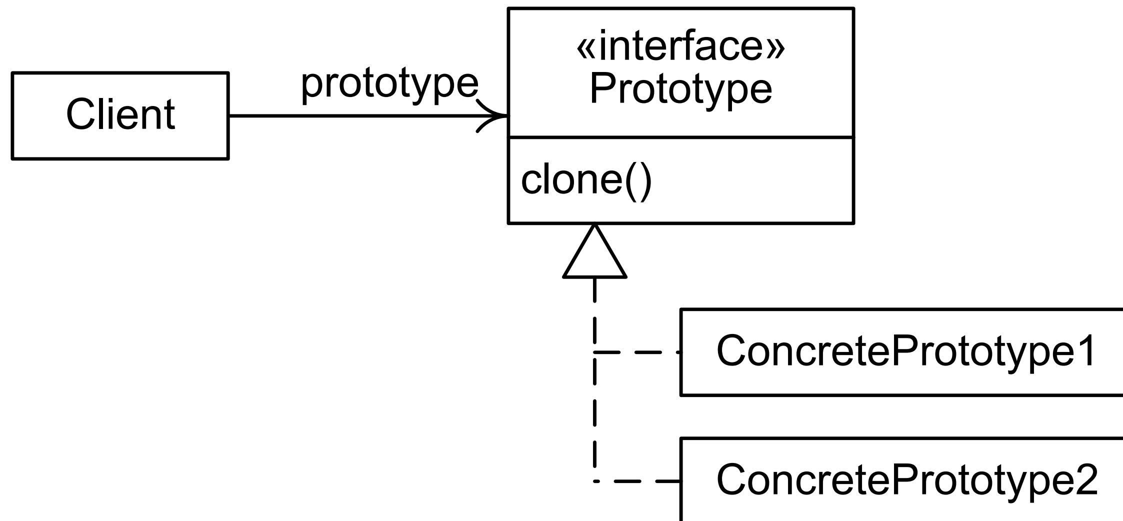
Should a clone be made using a shallow or a deep copy?

- It does not matter if no attributes hold references.
- There is not accepted practice.
- What ought to be done depends on the particular case at hand.

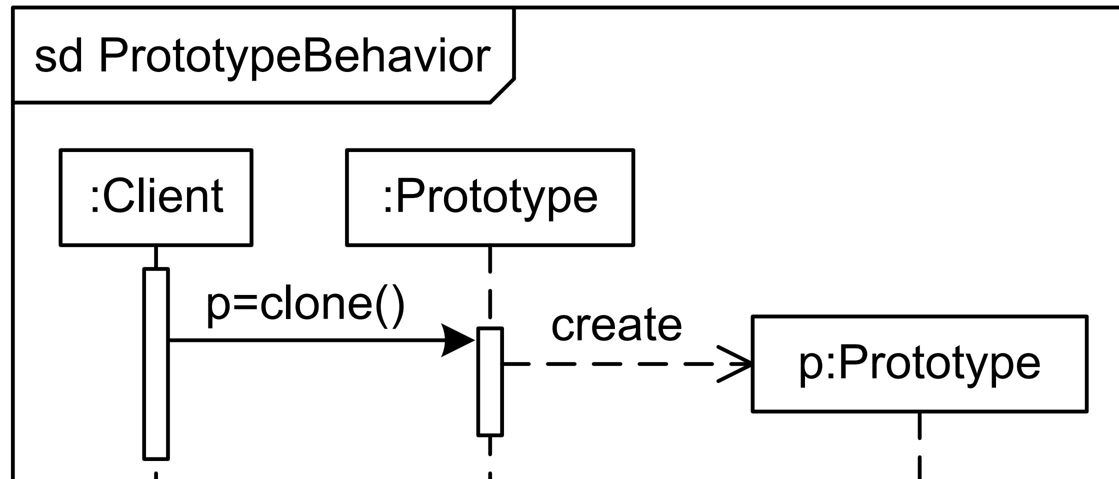
The Prototype Pattern

- Uses cloning implemented by a clone() factory method
- Uses interfaces to decouple the client from the cloned product
- Does not specify whether cloning is deep or shallow—use the right one for the case at hand
- Analogy: using an instance of something to get a new one

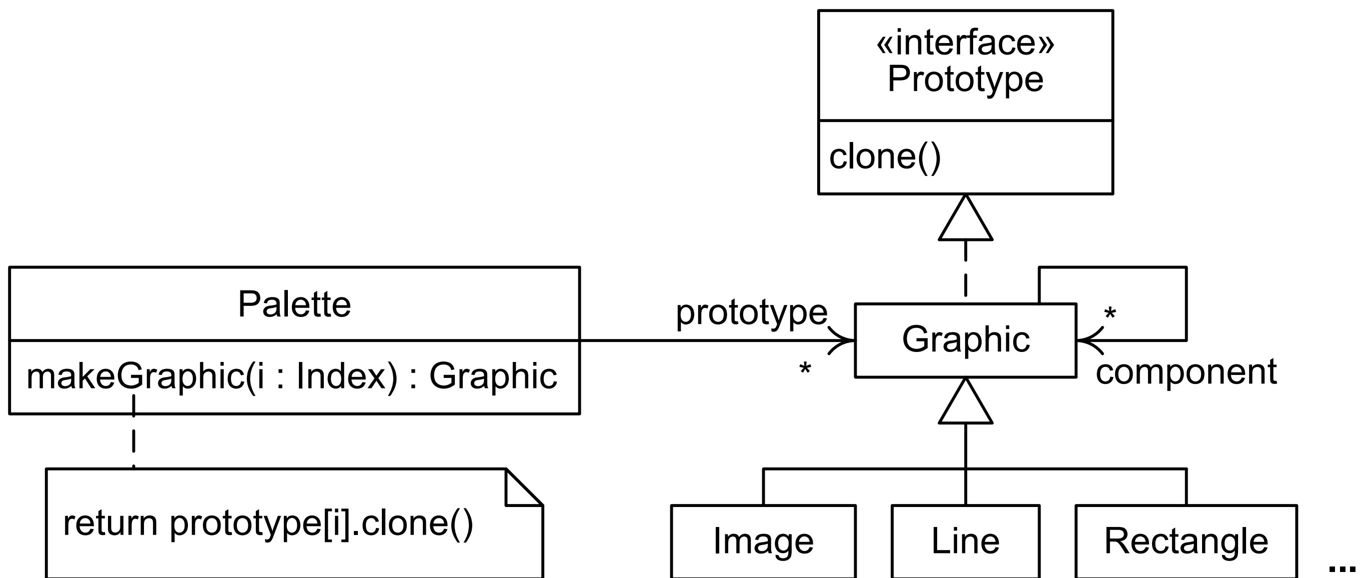
Prototype Pattern Structure



Prototype Pattern Behavior



Example: Graphic Prototypes



When to Use the Prototype Pattern

- Use the Prototype pattern when clients need to be decoupled from products, and the products need to be set at runtime.
- The main problem with the prototype pattern is that it relies on cloning, which presents problems about shallow and deep copies.

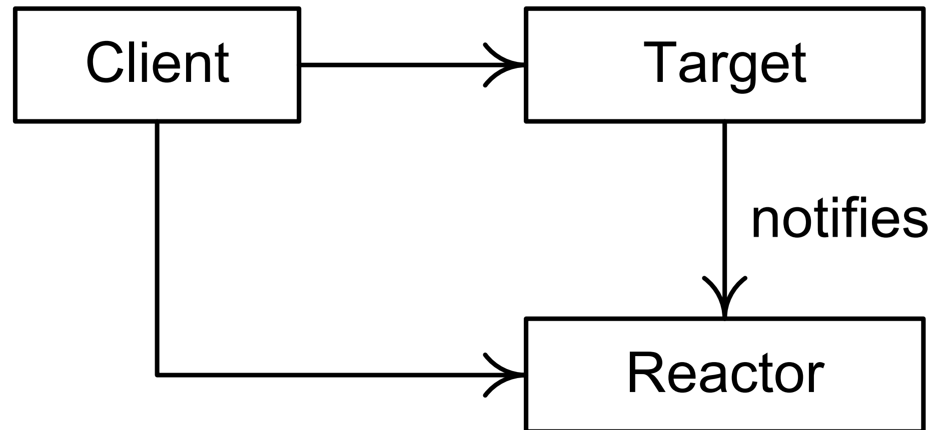
Summary

- The Singleton pattern is used to guarantee that only one or a set number of instances of some class exist.
- The Prototype pattern is used to create instances of classes (really copies of objects) determined at run time.

Reactor Patterns

- Have a client that needs to respond to an event in a target. The client delegates this responsibility to a *reactor*.
  - Command patterns
  - Observer patterns

Reactor Pattern Structure



The Client must access the Target and the Reactor so it can register the Reactor with the Target.

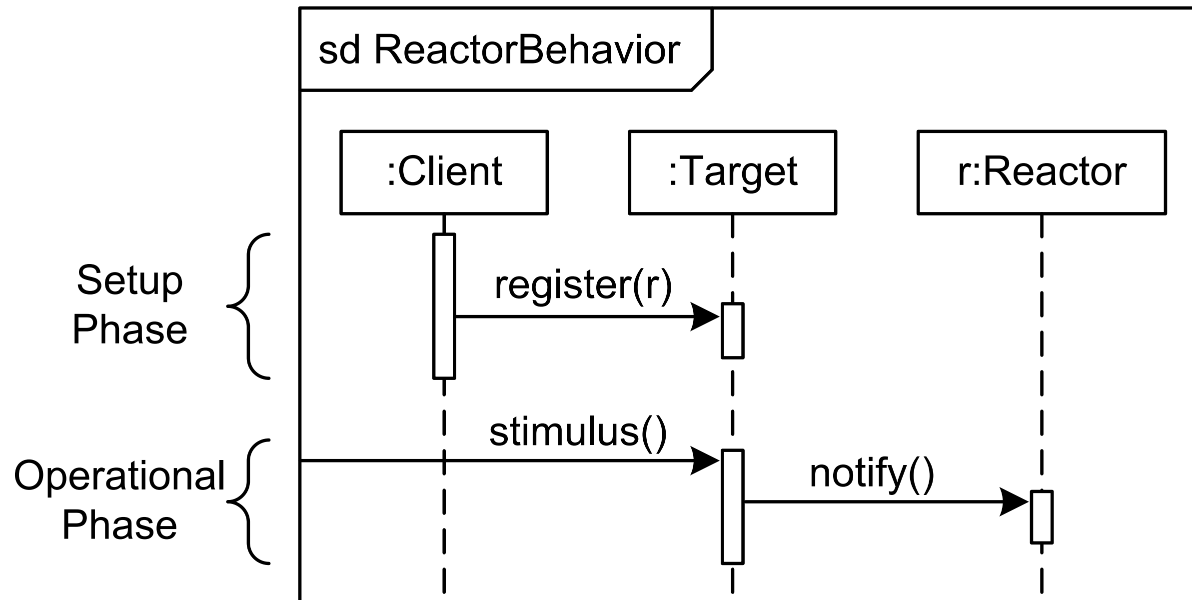
Event-Driven Design

- **Event-driven design** is an approach to program design that focuses on events to which a program must react.
  - An event is a significant occurrence
  - Contrasts with stepwise refinement
- Event handlers are components that react to or respond to events.
- Reactor patterns assist in event-driven design.

Behavioral Phases

- *Setup Phase*—The Client registers the Reactor with the Target.
  - Client interacts with the Target
- *Operational Phase*—The Reactor responds to events in the Target on behalf of the Client.
  - Client is not involved

Reactor Pattern Behavior



Reactor Pattern Advantages

- *Client and Target Decoupling*—Once the client registers the reactor with the target, the client and target need not interact again.
- *Low Reactor and Target Coupling*—The target only knows that the reactor is a receiver of event notifications.
- *Client Decomposition*—The reactor takes over client responsibilities for reacting to target events.
- *Operation Encapsulation*—The event handler in a reactor is an encapsulated operation that can be invoked in other ways for other reasons.

Event-Driven Architectures vs. Reactor Patterns

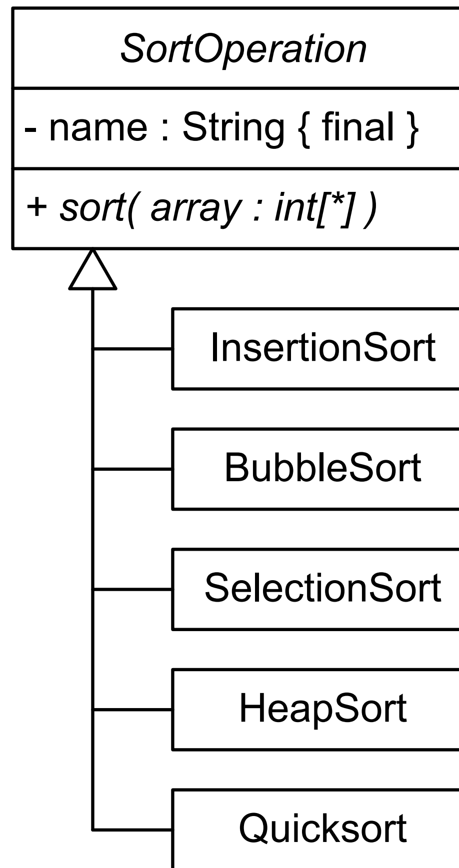
- Commonalities
  - Support event-driven design
  - Event announcement and handling
  - Two-phase behavior
- Differences
  - Level of abstraction
  - Event dispatcher completely decouples targets and reactors
  - Event dispatchers are complex and harder to use
  - Event dispatch may damage performance

Function Objects

- In some languages operations can be treated as values (stored, passed, etc.).
- This is not usually true in object-oriented languages—use objects to encapsulate operations.

A **function object** or **functor** is an object that encapsulates an operation; the encapsulating class is a **function class**.

Function Object Example 1



Function Object Example 2

```
Collection sortCollection = new ArrayList()
sortCollection.add( new InsertionSort() )
sortCollection.add( new BubbleSort() )
...
print "Sort Time1 Time2 Time3 ... Timek"
for each element sorter of sortCollection
   print sorter.toString()
   for each array a
      startTime = now()
      sorter.sort(a)
      endTime = now()
      print( endTime - startTime )
   println
```

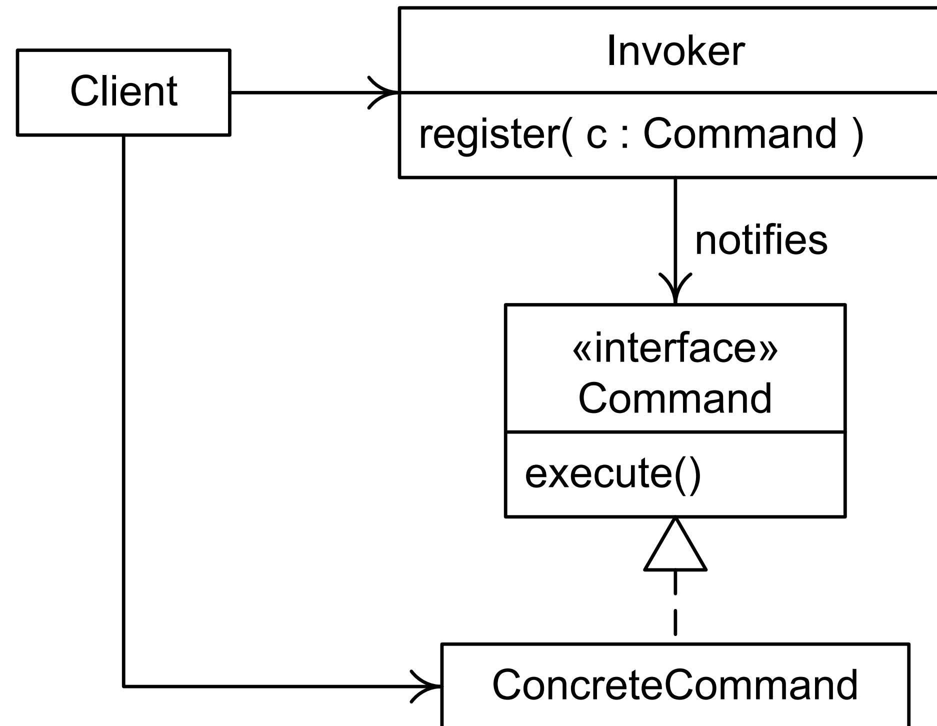
Function Object Advantages

- Additional features can be added to the function class besides the encapsulated operation.
- The function class can include operations and data that the encapsulated operation needs.

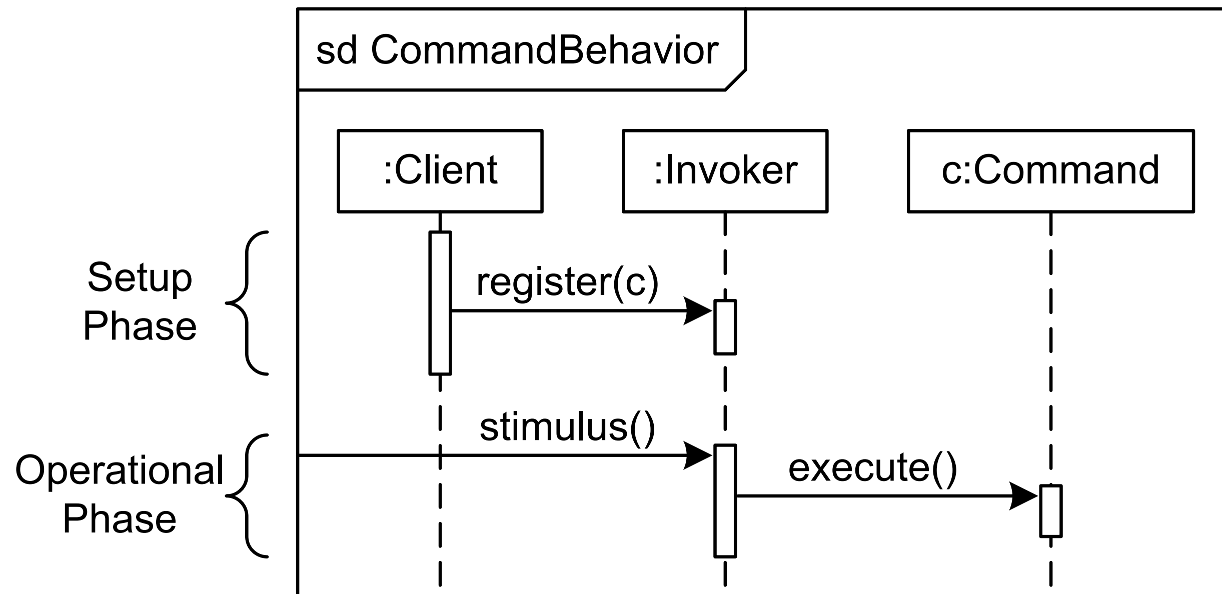
The Command Patterns

- The reactor is a function object
- Simple and very widely used way to implement *callback functions* in user interfaces

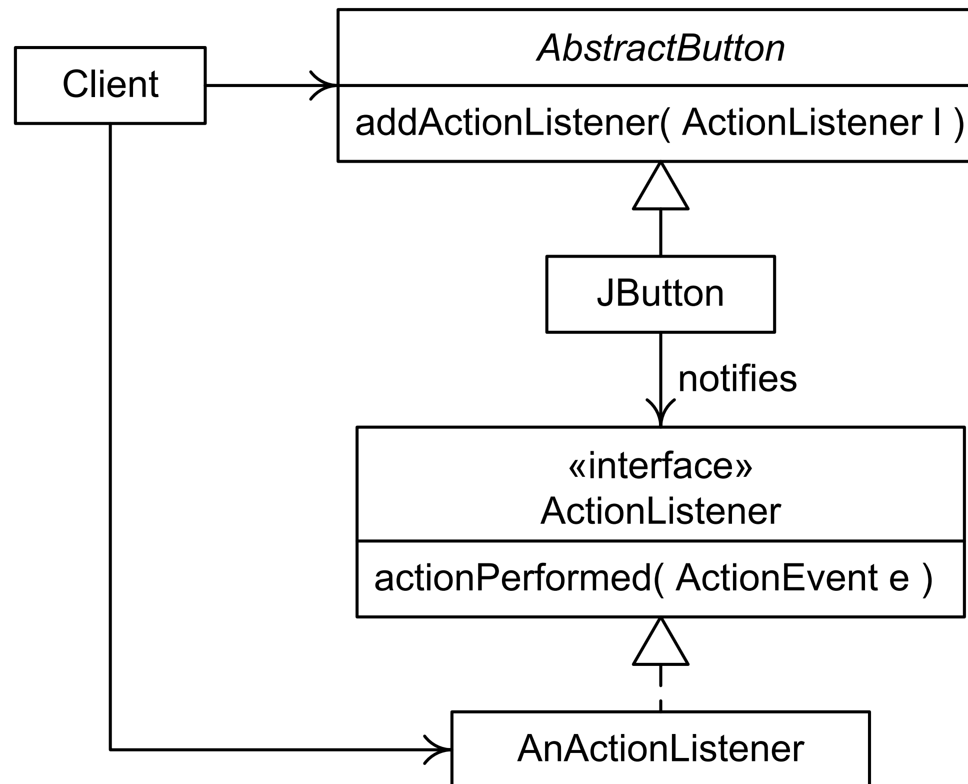
Command Pattern Structure



Command Pattern Behavior



Command Pattern Example



When to Use the Command Pattern

- Use the Command pattern to delegate a client's response to events from an invoker class to another class that encapsulates the reaction.
- Use the Command pattern to decompose clients, lower coupling between clients and invokers, and to encapsulate event-handling code.

The Observer Pattern

- Reduces coupling between classes while preserving their ability to interact
- Can be used whenever one or more objects (observers) must react to events in another object (subject)
- Analogy: current awareness service

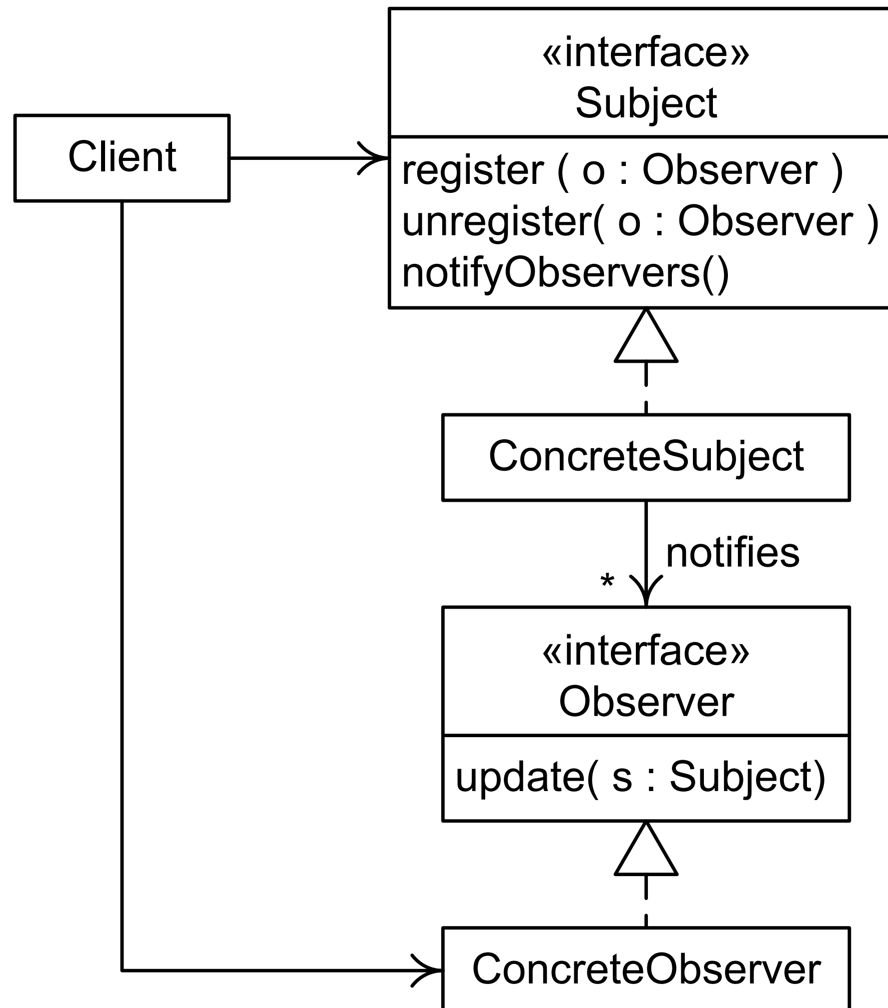
Observer in the MVC Architecture

- A model in an MVC architecture can keep track of its views and controller
  - Strongly couples the model to its views and controllers
  - Changing the UI forces changes in the model
- The model can be a subject and the views and controllers can be observers
  - Decouples the model from its views and controllers
  - Changing the UI has no effect on the model

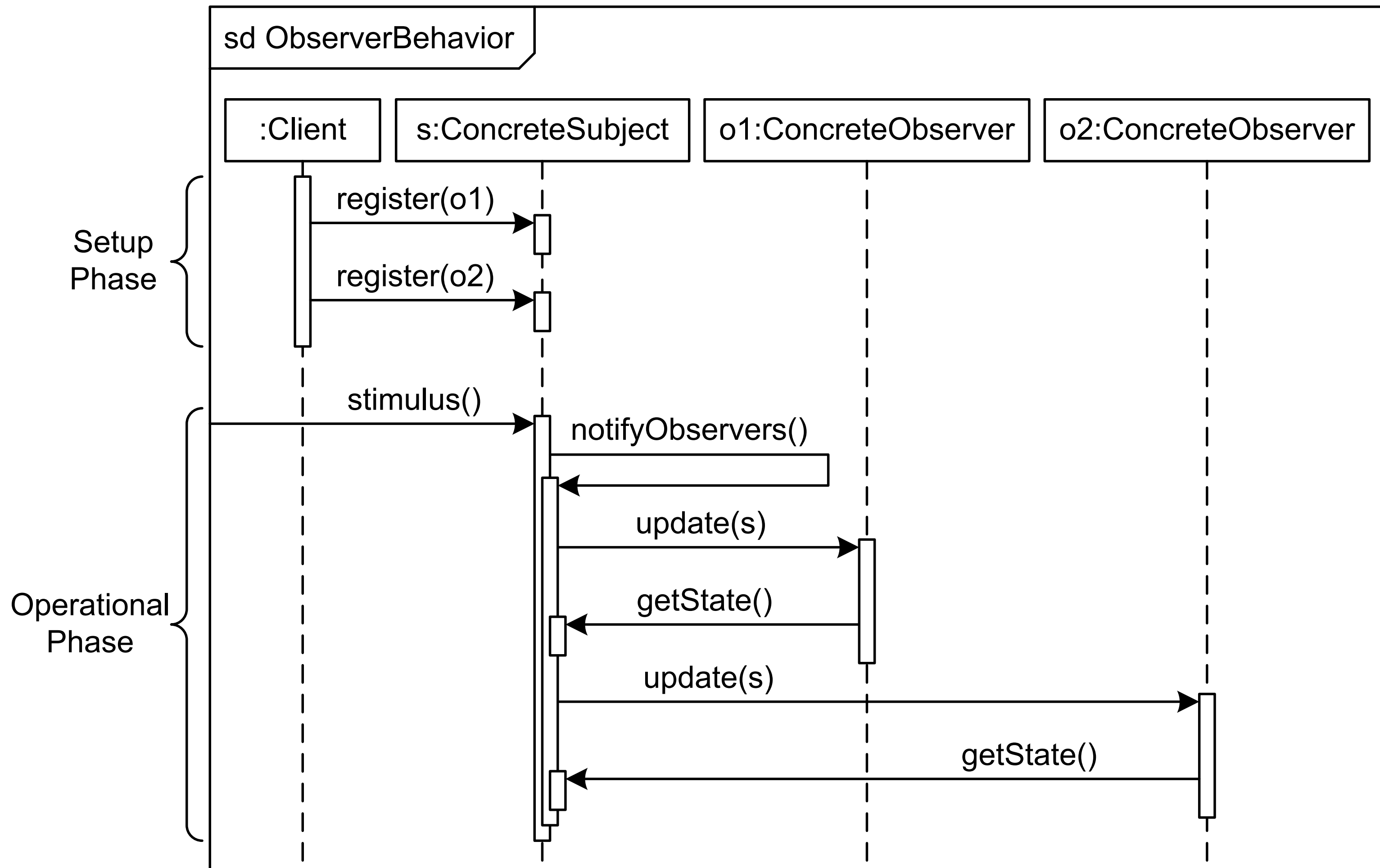
Subject and Observer Operations

- Subject
  - Registration operations
  - Notification control operations
  - Query operations
- Observer
  - Notification operation

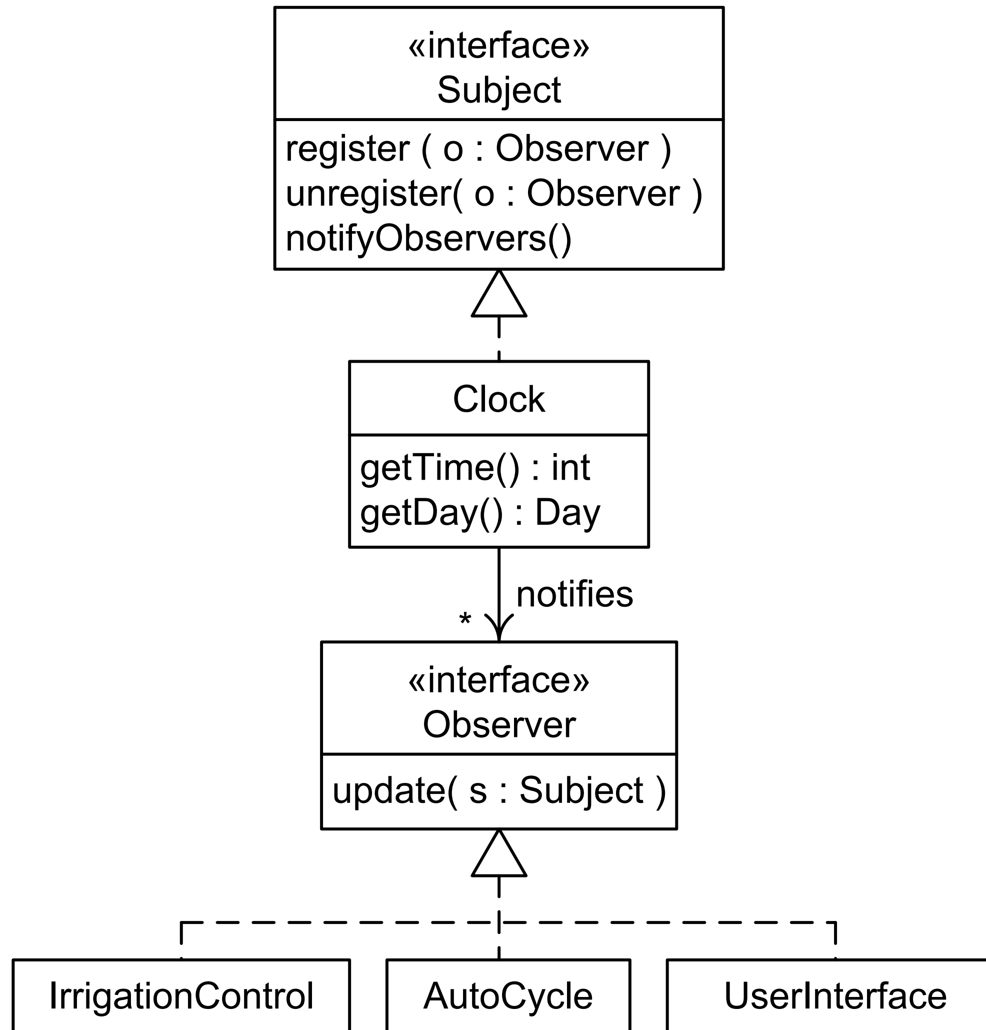
Observer Pattern Structure



Observer Pattern Behavior



Observer Pattern Example



When to Use the Observer Pattern

- Use the Observer pattern whenever one object must react to changes in another object, especially if the objects must be loosely coupled.
  - User interface components
  - Clock-driven components
- The main drawback of the Observer pattern is that notification may be expensive.
  - This can be dealt with in several ways depending on the situation.

Summary 1

- Reactor patterns use a Reactor to which a Client delegates responsibility for responding to events in a Target.
- Reactor patterns help realize event-driven designs in a cheaper and easier way than event-driven architectures at the expense of slightly higher component coupling.
- The reactor patterns help decouple targets from their both clients and reactors.

Summary 2

- The Command pattern uses a function object as a reactor; the function object encapsulates the reaction and can be used for other purposes as well.
- The Observer pattern has a subject with which observers register; the subject then notifies its observers of changes, and the observers query the subject to determine how to react.