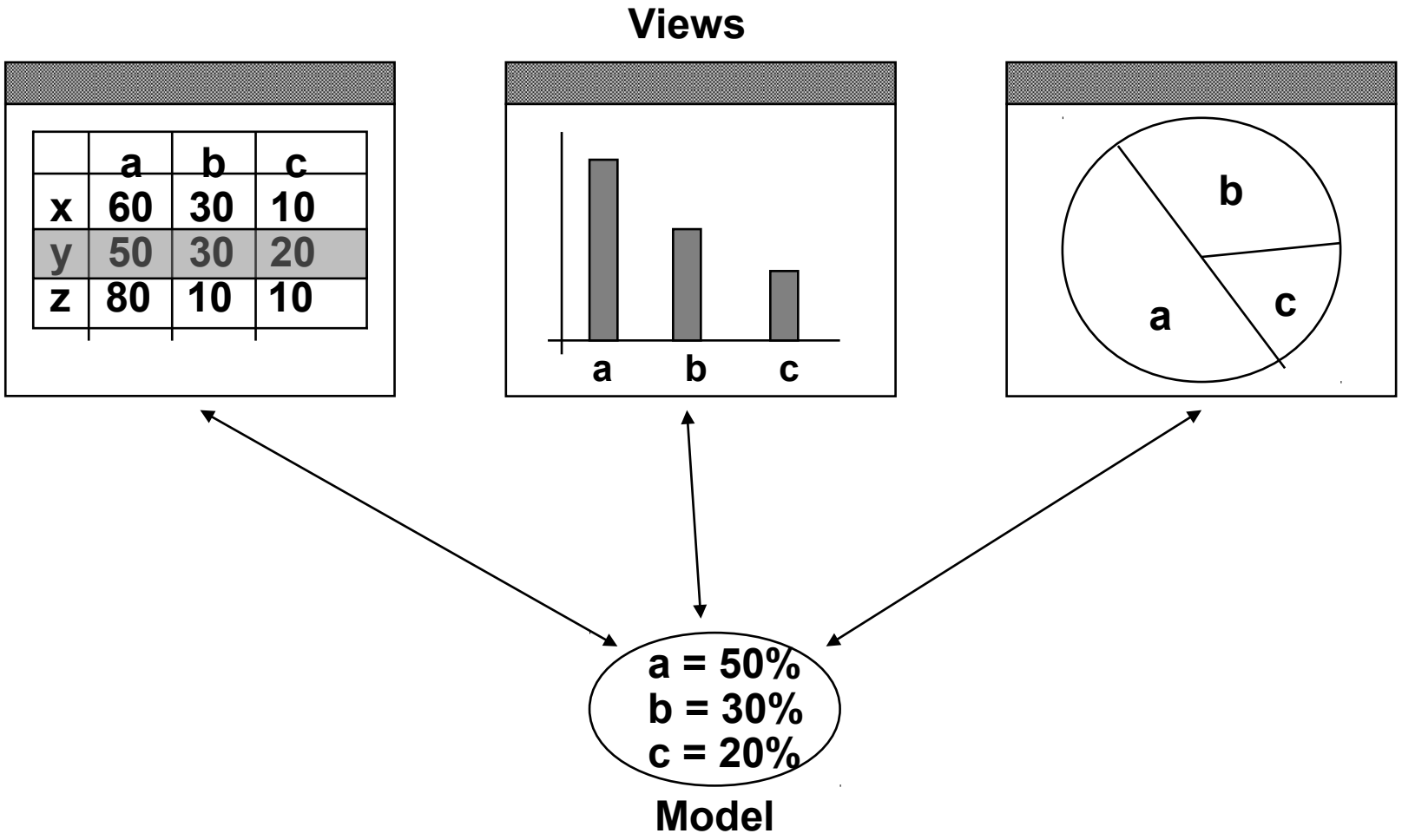
# Design Patterns

- references:
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma *et al* (D)
  - *Patterns in Java,* vol. 1, M. Grand (G)
  - *Understanding Object-Oriented Programming with Java*, Budd: ch.15
- capture structure of successful solutions to common problems
- essential elements
  - pattern name
  - problem: when to apply the pattern
  - solution: general arrangement of elements
  - consequences: results and tradeoffs

# Model-View-Controller (MVC) (D p4)

- one of earliest examples of a design pattern
- user-interface pattern (Smalltalk-80)
- separate data (model) from view (presentation) and controller (user interaction)
- decoupling of objects to increase flexibility and reuse
- subscribe/notify protocol between views and models
- view uses controller to implement response strategy

# Model-View-Controller (D p5)



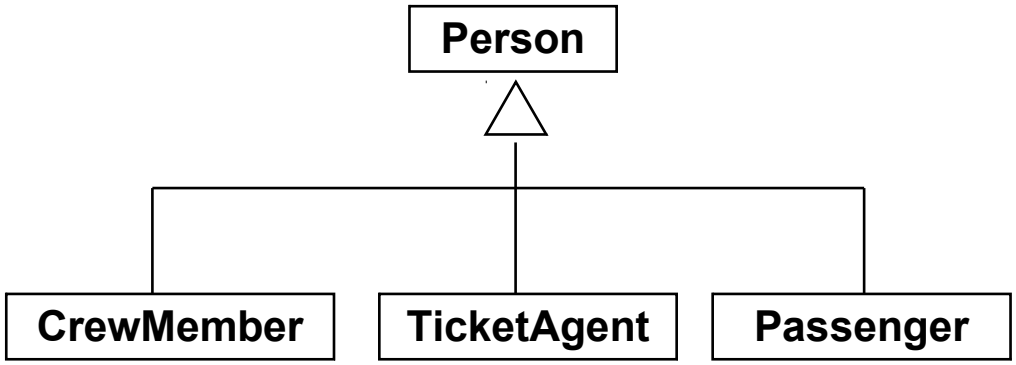
# Using Patterns (D1.7,1.8)

- organization
  - purpose – fundamental (G), creational, structural, behavioral
  - scope
    - class: focus on relationships between classes and subclasses (static)
    - object: focus on relationships between objects (dynamic)
- selection
  - consider how the patterns solve the problems
  - review the Intent sections
  - consider what is variable in your design
- application
  - choose application-specific names for pattern participants
  - define the classes & interfaces
  - choose application-specific names for operations in the pattern
  - implement the operations

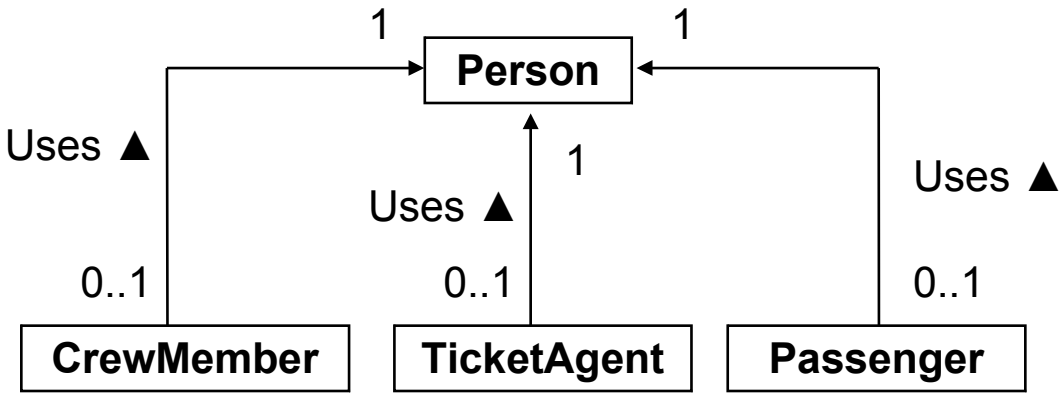
# Delegation (D p20, G p53)

- a fundamental pattern (G)
- intent: extend the behavior of a class in a more flexible way than inheritance
- motivation: sometimes different objects of a class play different roles or the same object plays different roles at different times
- e.g.
  - `CrewMember`, `TicketAgent`, `Passenger` are roles of ("kind-of"?) `Person`, however a `CrewMember` may be a `Passenger` at another time etc.
  - to accommodate possible combinations of roles, could create many subclasses, e.g. `CrewMemberAndPassenger`
  - delegation can simplify as each role delegates to the common class: `Person`

# Airline Roles (G4.1, 4.3)



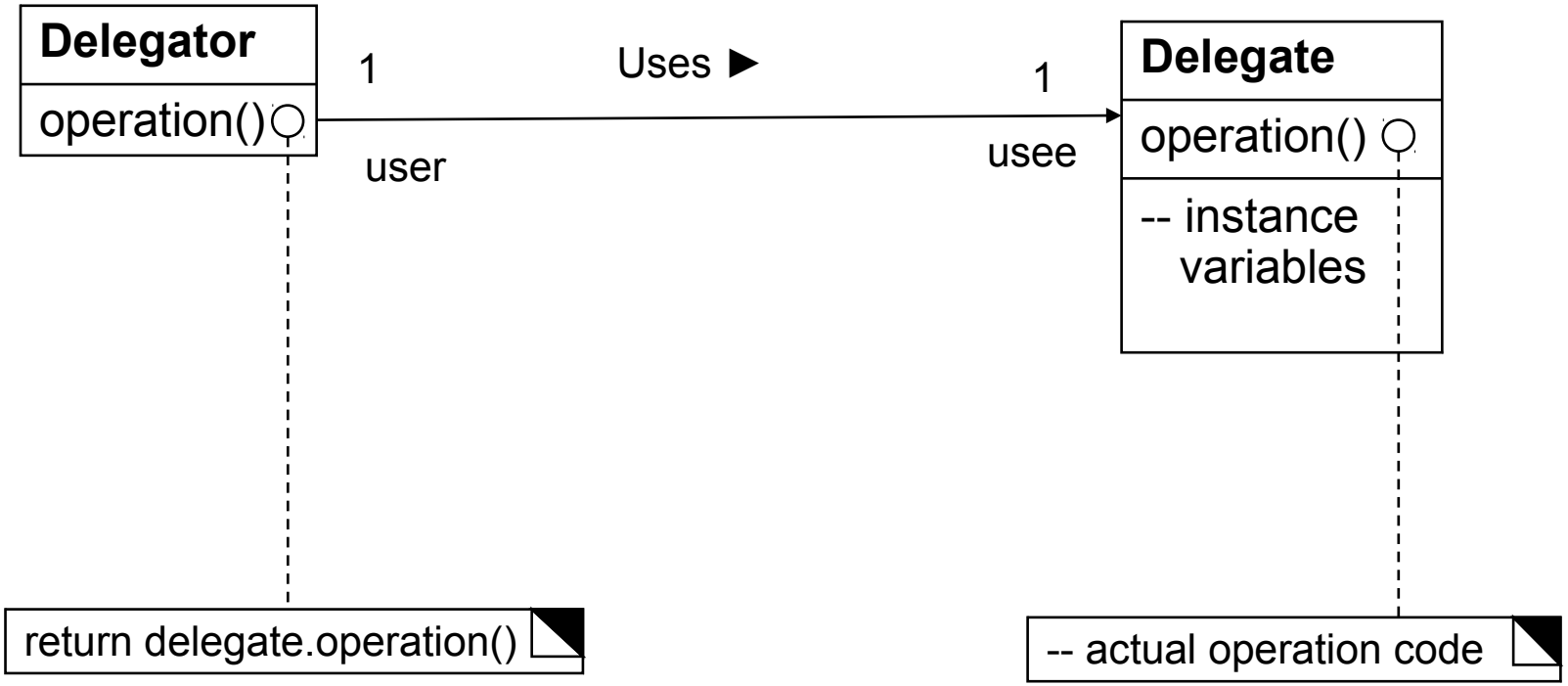

Inheritance




Delegation

- use:
  - class needs to be subclass of different classes over time
  - class attempts to hide methods or variables inherited from superclass
  - class related to a program's problem domain should not be a subclass of a utility class
- model
- consequences:
  - delegation is less convenient
  - delegation imposes less structure than inheritance
- Java API
  - event sources delegate handling of events to listeners

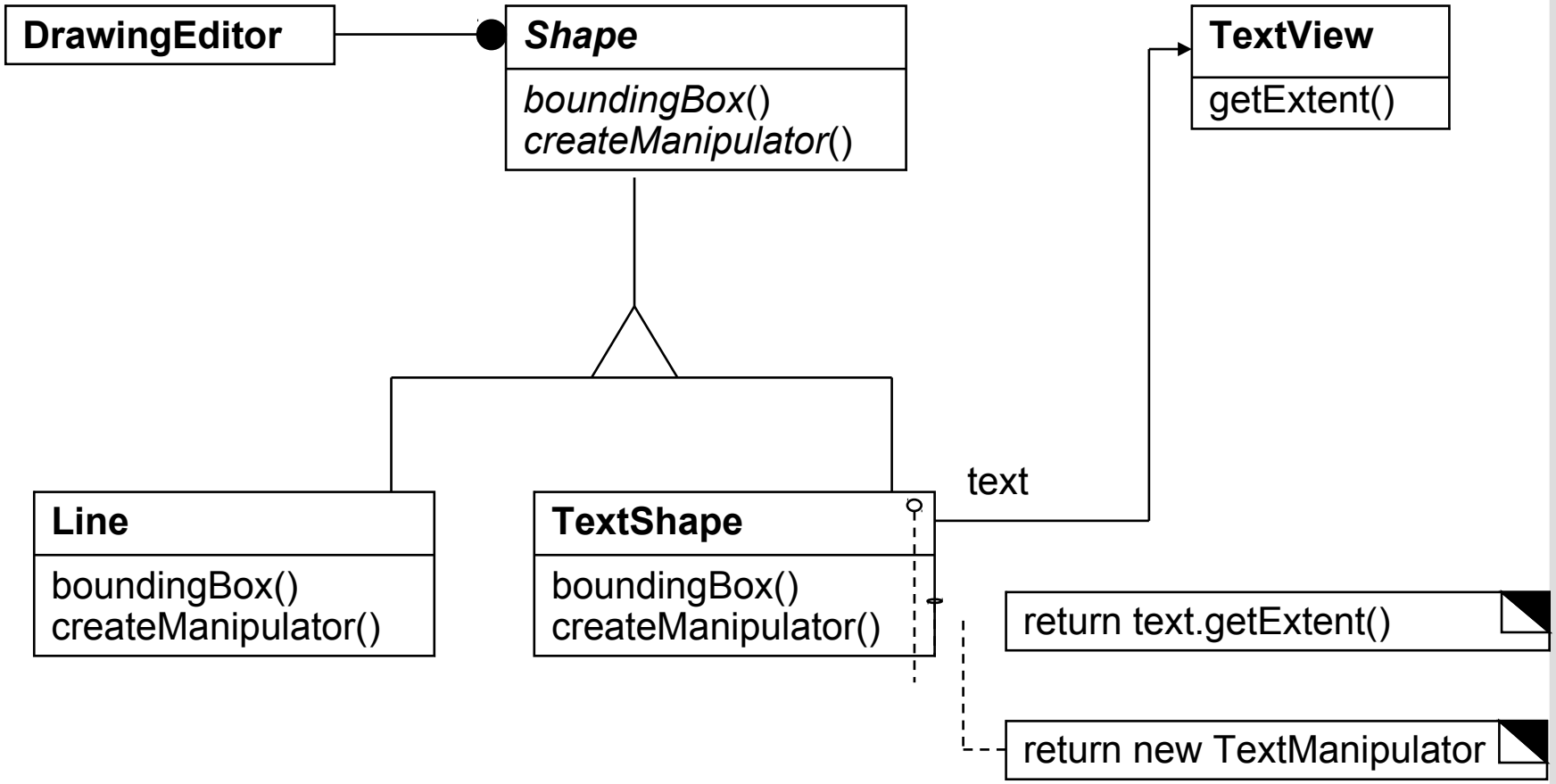
# Delegation (G4.4)



# Adapter (D p139, G p209)

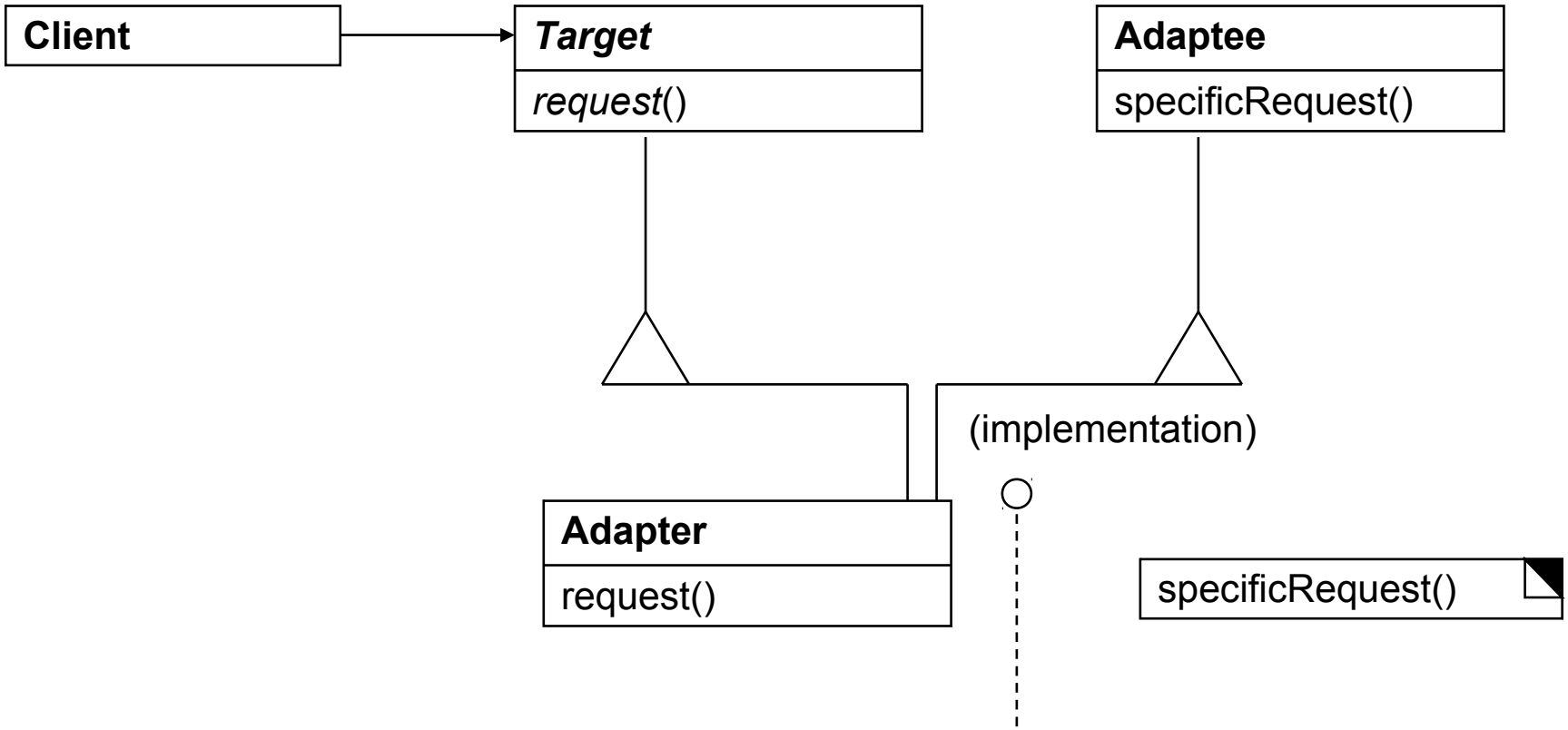
- aka Wrapper
- a structural pattern
- intent: convert interface of a class into another interface clients expect
- motivation:
  - sometimes object provides appropriate behavior but uses a different interface than is required
- e.g.
  - `TextShape` as adapted `TextView` in a drawing editor
  - `Shape` hierarchy
  - existing `TextView` class
    - modify `TextView` class?
  - `TextShape` as adapter
    - via inheritance (class adapter)
    - via composition (object adapter)

# TextShape as Adapter (D p140)

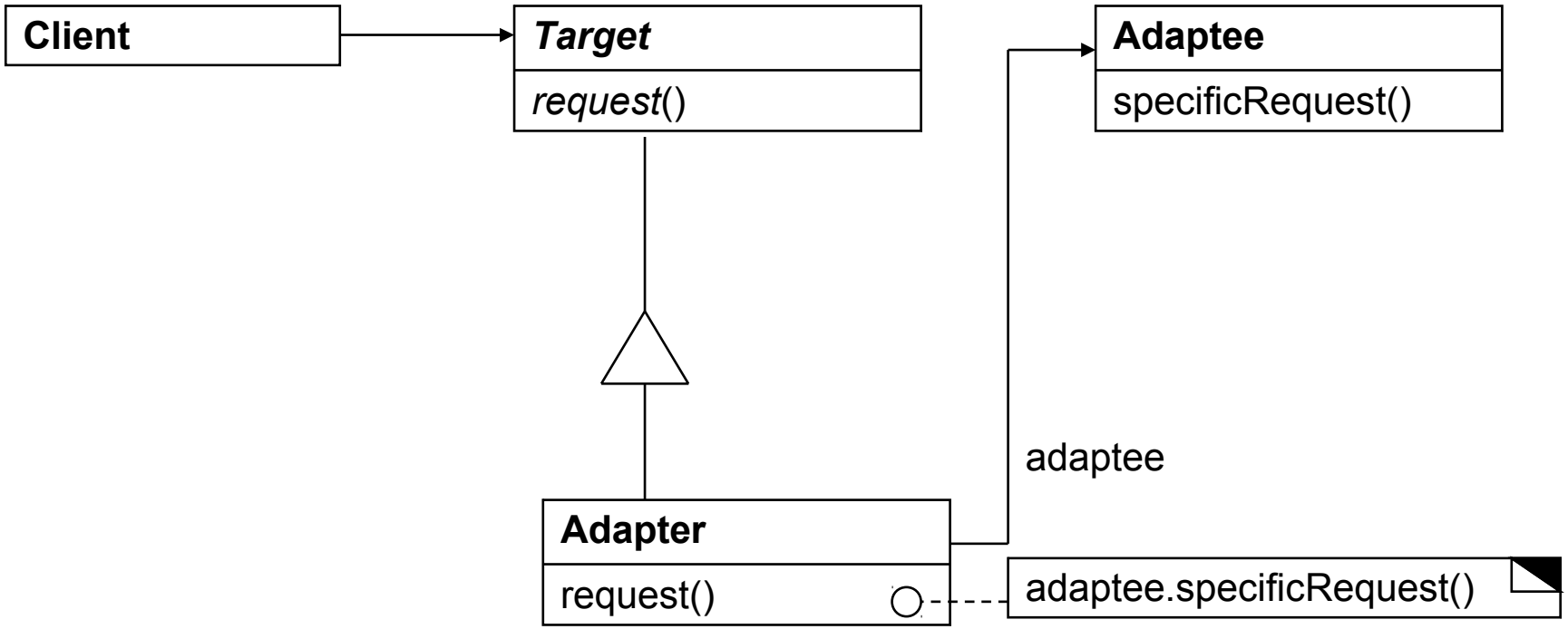


- use:
  - want to use existing class without proper interface
  - want to create reusable class that cooperates with unrelated or unforeseen classes
  - want to use several existing subclasses but impractical to adapt interface by subclassing every one
- model
- consequences:
  - class adapter (inheritance)
    - commits to a concrete `Adaptee` class, can't adapt class and all subclasses
    - `Adapter` can override behavior (inheritance)
    - only one object created
  - object adapter (composition)
    - `Adapter` can work with multiple `Adaptee`s (`Adaptee` class and any subclass), adding functionality to all `Adaptee`s at once
    - harder to override `Adaptee` behavior, i.e. the subclasses may also override so must adapt each subclass as well

# Class Adapter (D p141)



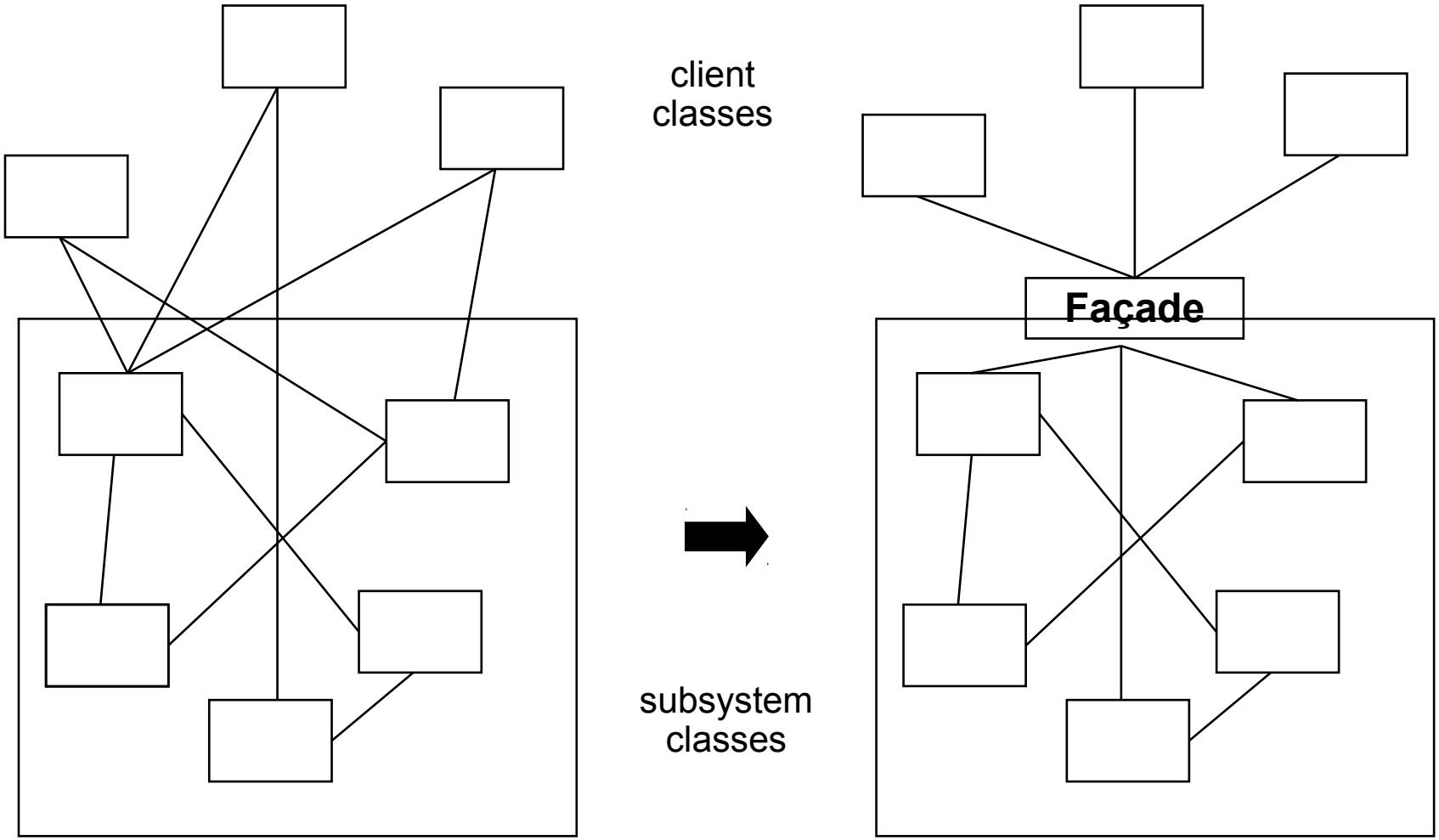
# Object Adapter (D p141)



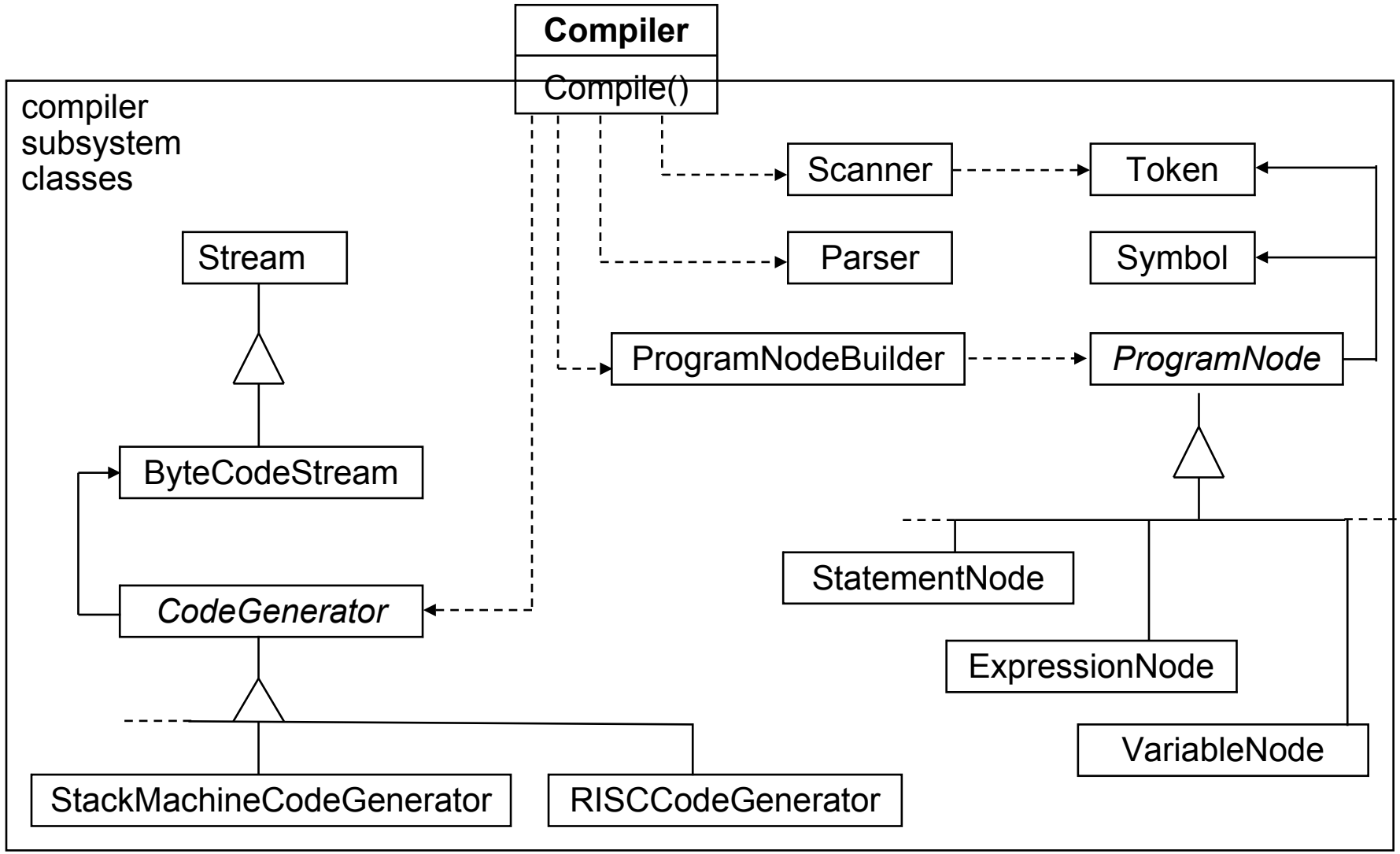
# Façade (D p 185, G p235)

- a structural pattern
- intent:
  - provide a unified interface to a set of interfaces in a subsystem
- motivation:
  - most clients don’t care about details
  - powerful, low-level interfaces complicate task
  - e.g. Compiler is façade for Scanner, Parser, etc.

# Façade Motivation (D p185)



# Compiler as Façade (D p186)

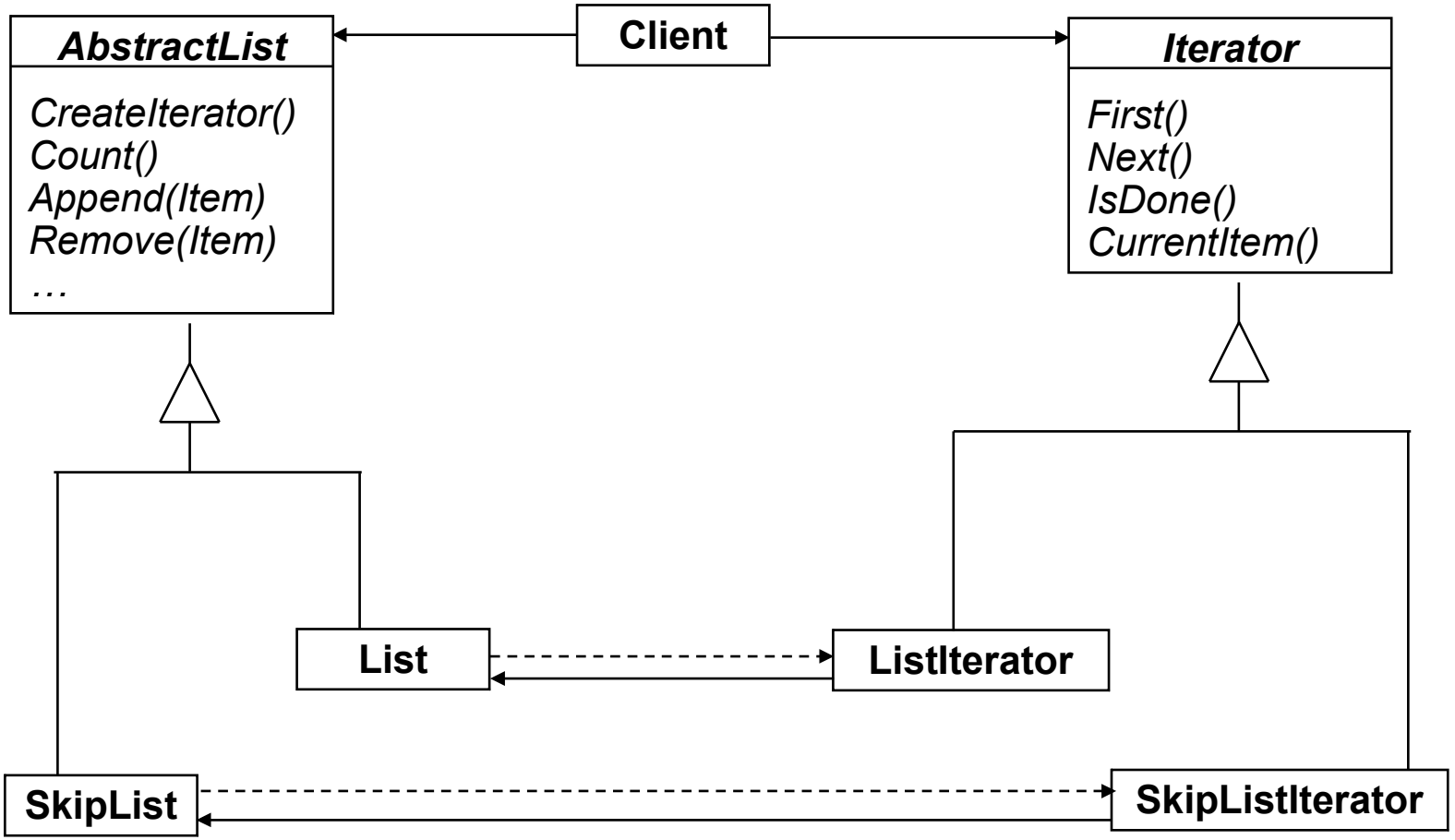


- use:
  - want to provide a simple interface to a complex subsystem
  - there are many dependencies between clients and implementation classes
- model
  - Façade
    - common interface to all subsystem functions
    - delegates to appropriate subsystem object
  - subsystem classes
    - implement subsystem functionality
    - have no knowledge of Façade
- consequences:
  - shields clients from subsystem components, making subsystem easier to use
  - promotes weak coupling between subsystem and clients
  - eliminate complex dependencies
- Java API
  - `java.net URL` class allows access to URLs without knowing the classes which support URLs

# Iterator (D p 257, G p217)

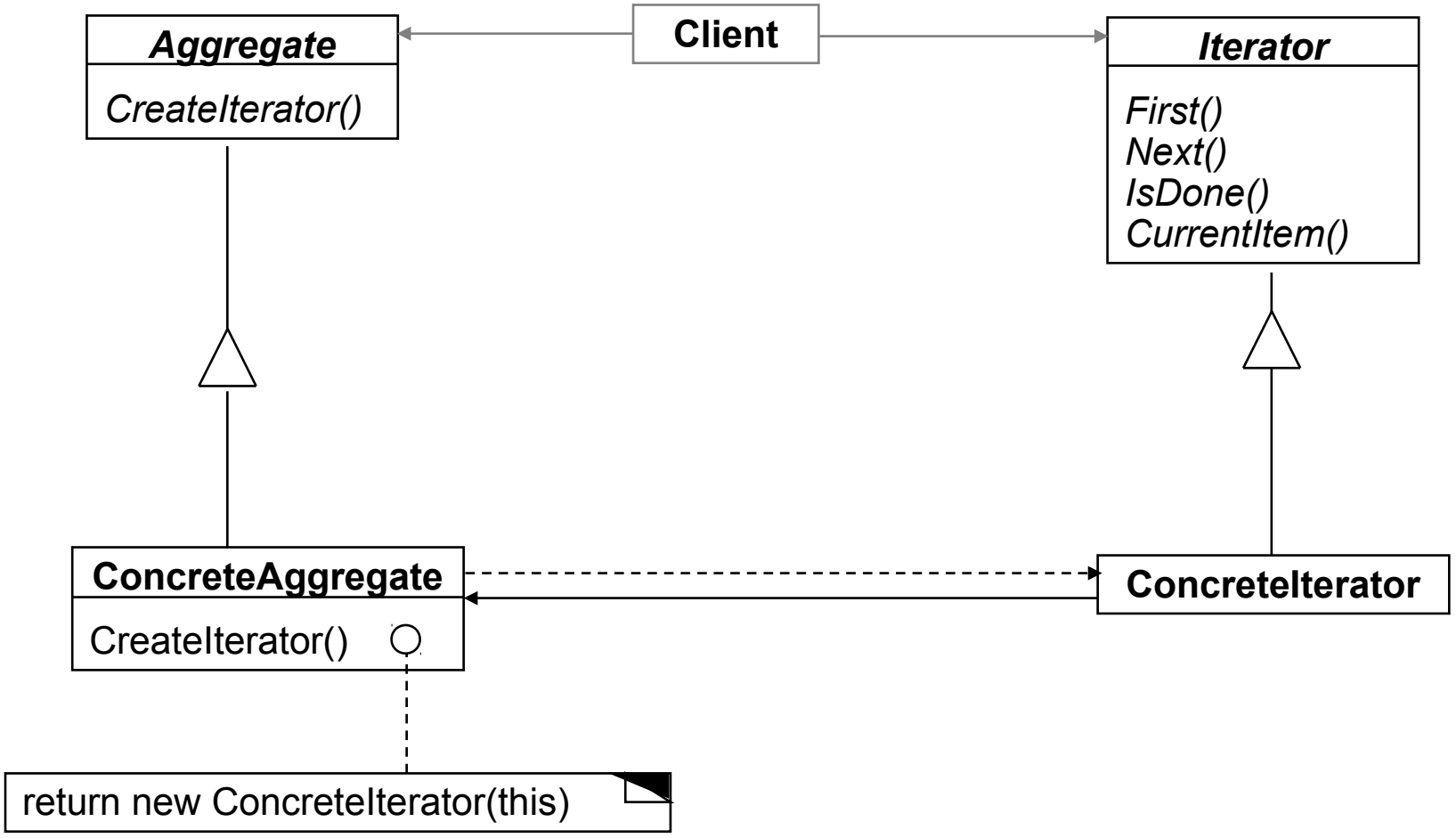
- aka Cursor
- a behavioural pattern
- intent:
  - allow sequential access to elements of an aggregate without exposing underlying representation
- motivation:
  - want to be able to access elements of an aggregate object without exposing internal structure
  - want to use several different traversals
  - want different traversals pending on same list
  - e.g. iterators for `List`, `SkipList`

# List Iterators (D p258)



- use:
  - access aggregate object's contents without exposing internal representation
  - support multiple traversals of aggregate objects
  - provide uniform interface for traversing different structures
- model
- consequences:
  - supports variations in traversal of aggregate
  - simplified aggregate interface
  - more than one traversal can be pending on an aggregate
- considerations:
  - internal vs external iterators
  - who defines traversal algorithm
- Java API
  - `Collection` and `Iterator` interfaces

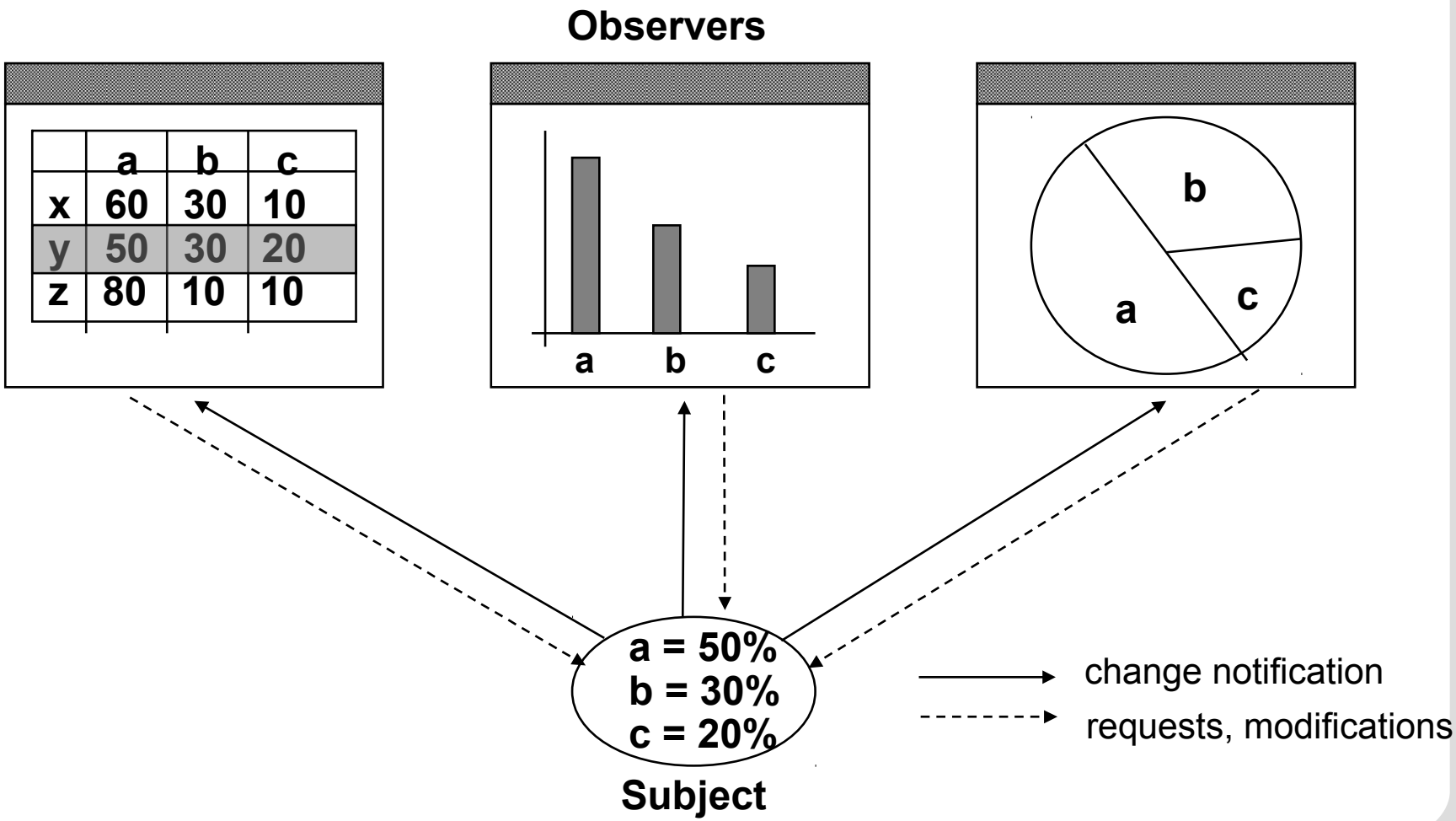
# Iterator (D p259)



# Observer (D p 293, G p387)

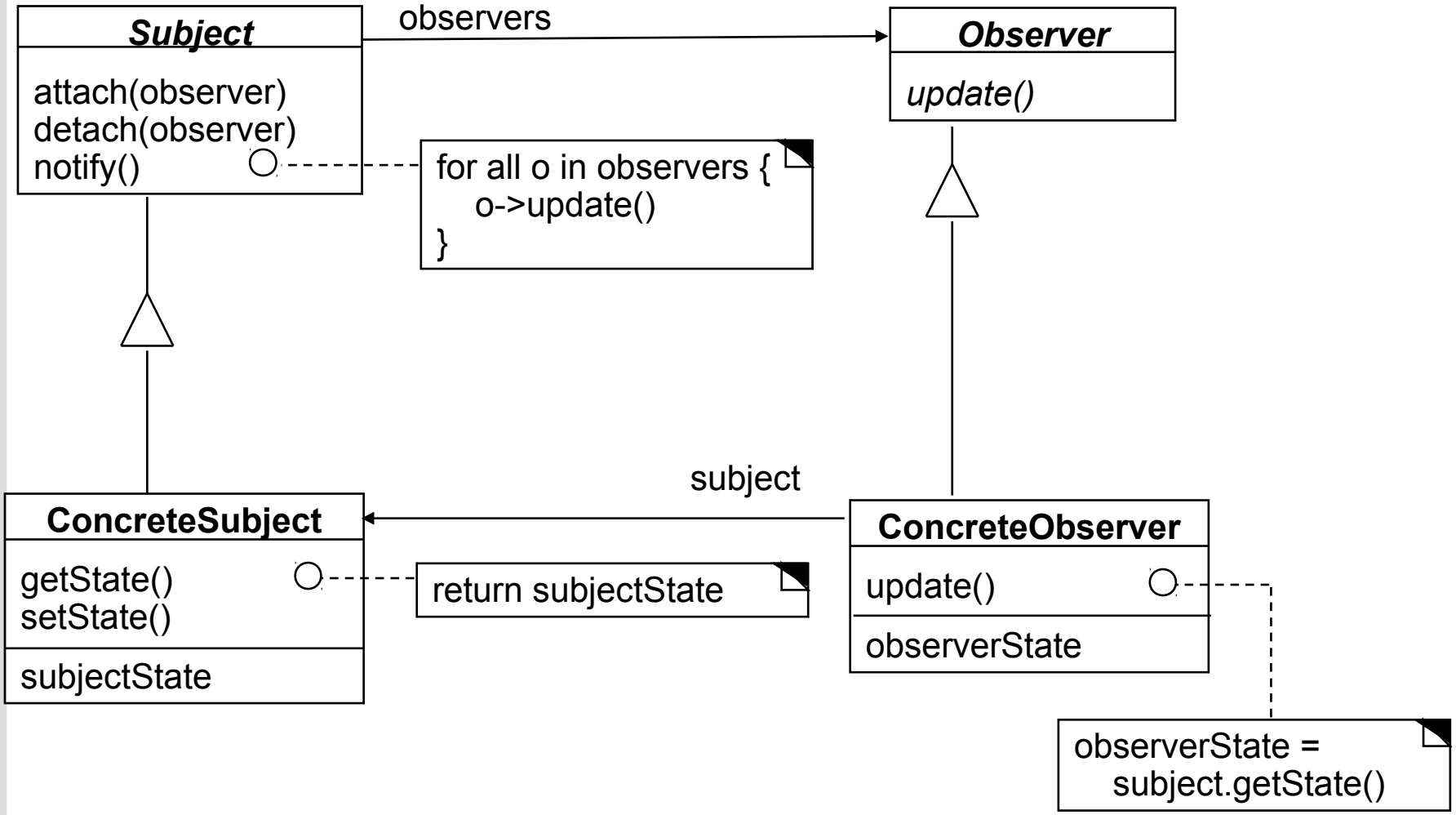
- aka Dependents, Publish-and-Subscribe
- a behavioral pattern
- intent:
  - define dependencies between objects
  - when an object changes state, ensure all dependents are notified and updated
- motivation:
  - need to maintain consistency among cooperating classes
  - e.g. MVC GUI model
    - multiple views of same data
      - multiple windows on same text file
    - subject
    - observers

# Model-View-Controller (D p293)

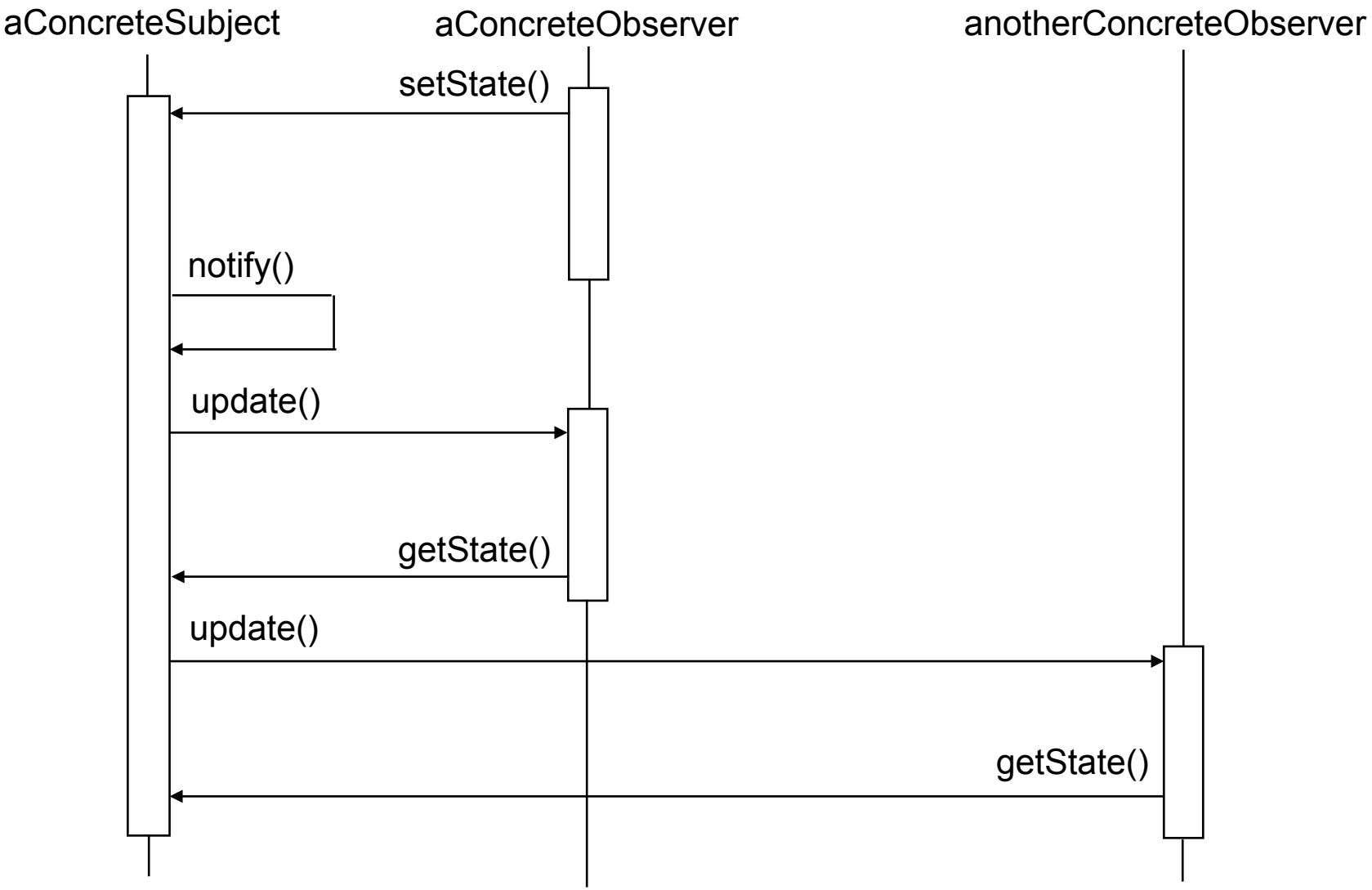


- use:
  - abstraction has two aspects, one dependent on the other
  - change to an object requires changing an unknown number of others
  - object needs to notify other objects without knowing details about them
- model
  - structure
  - interaction

# Observer (D p294)



# Observer – Interaction (D p 295)

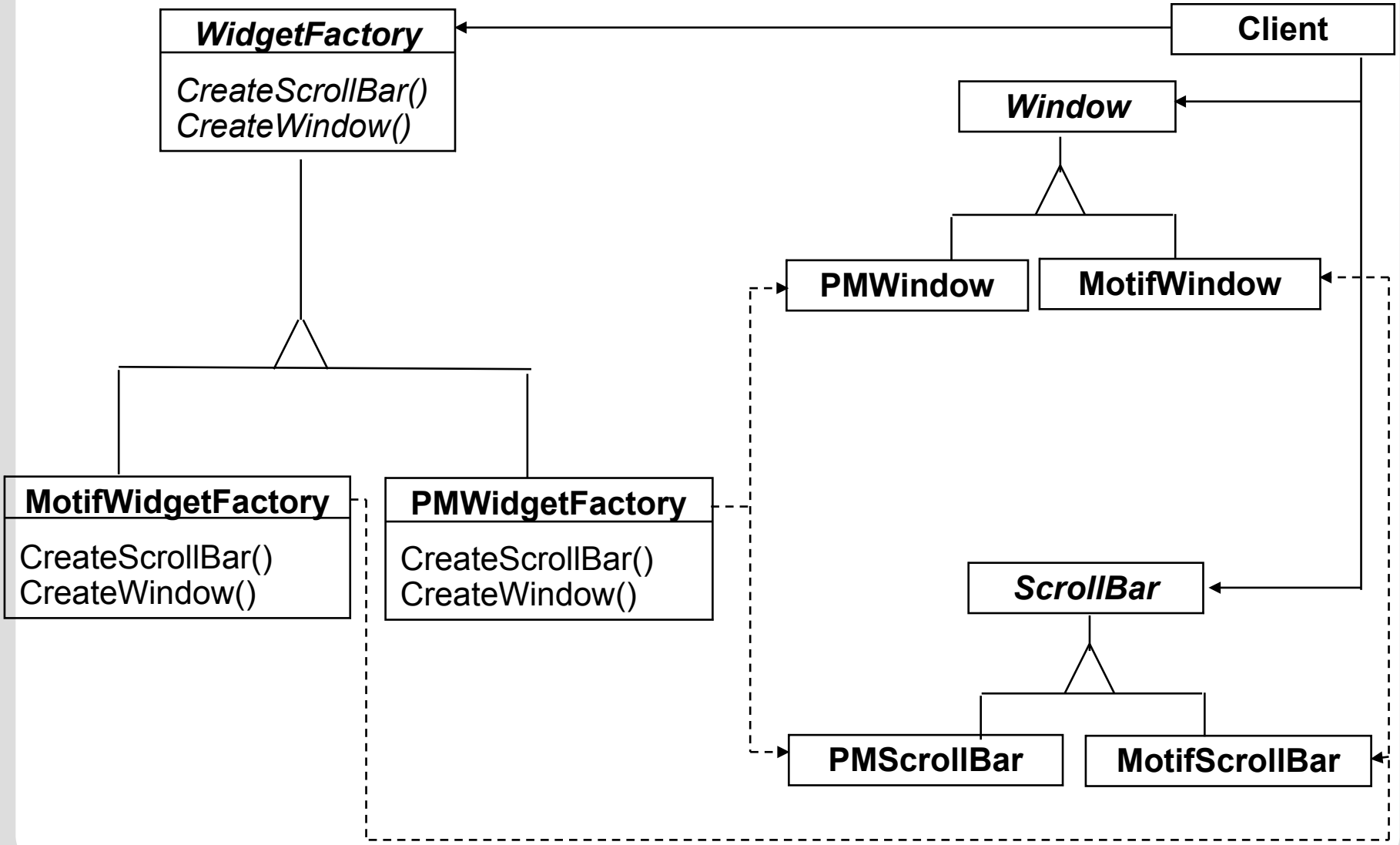


- consequences
  - abstract coupling between `Subject` and `Observer`
  - broadcast communication
  - unexpected updates
- considerations
  - mapping subjects to observers
  - observing more than one subject
  - triggering the update
  - deleted subjects
  - self-consistent state in subject
  - how much information to send on update
- Java API
  - `Observer` and `Observable` interfaces
  - event listeners are observers, event sources are subjects

# Abstract Factory (D p 87, G p119)

- aka Kit
- a creational pattern
- intent:
  - provide interface for creating families of related objects without specifying concrete representation
- motivation:
  - toolkit that supports multiple standards
  - e.g. look and feel of widgets
    - define `WidgetFactory` that declares interface for each kind of widget
    - concrete subclasses implement widgets for different look and feel standards
    - clients call operations in `WidgetFactory`, remaining independent of actual look and feel

# Widgets (D p87)



- use:
  - system should be independent of how products are created, composed and represented
  - family of products designed to be used together
  - want to provide library of products and reveal only interfaces not implementation
- model
- consequences
  - isolates concrete classes
  - easy to exchange product “families”
  - promotes consistency among products
  - difficult to support new kinds of products
- Java API
  - AWT and Swing classes

# Abstract Factory (D p88)