EMBEDDED SYSTEMS PROGRAMMING 2017-18

Design Patterns

DEFINITION

Design pattern: **solution to a recurring problem**

- Describes the key elements of the problem and the solution in an abstract way
- **Applicable outside the realm of software design,** but of course software design is what we will consider in this course

DESIGN PATTERNS

Design patterns capture solutions that

- have been developed over time, so
- they **are not obvious**, and
- often they are **a bit harder** than first-thought or ad-hoc solutions, but
- they have proved good many times, hence
- they will (probably) **pay off in the long term**

ELEMENTS OF A DESIGN PATTERN

- **Name**: makes the pattern recognizable
- **Problem** description (with context)
- **Solution**: describes the key elements to solve the problem, and the relations between them. DPs for OO software design may talk about classes and such
- **Consequences**: pros, cons and tradeoffs of applying the pattern

DESIGN PATTERNS FOR OO SOFTWARE: CLASSIFICATION

- **Behavioral** patterns: discuss algorithms and communication paths to manage the information flow within a system

- **Structural** patterns: show how to compose a large structure out of simpler ones

- **Creational** patterns: aim at making a design independent of how its objects are created, composed and represented

WHAT WE WILL SEE

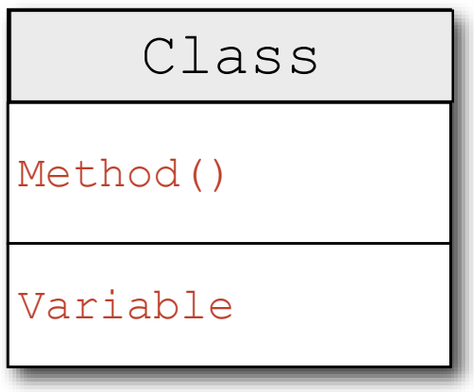
- **Behavioral** patterns: Delegation, Target-Action, Command, Mediator, Observer

- **Structural** patterns: Model-View-Controller, Composite, Façade, Proxy

- **Creational** patterns: Builder, Singleton

NOTATION (1/2)

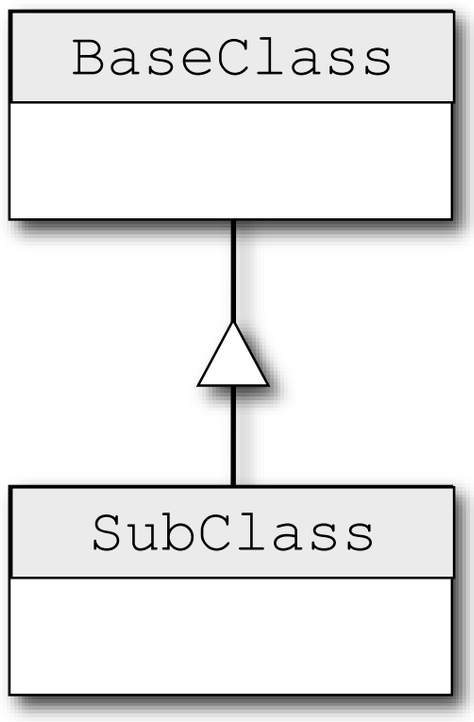
Class



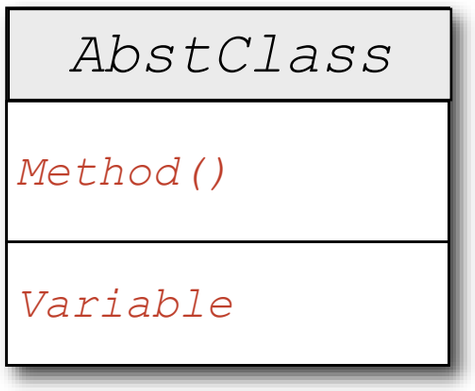
Pseudocode

```
for(...)
{
    /* ... */
}
```

Inheritance



Abstract class



NOTATION (2/2)

"Is part of"
"Is an aggregation of"

"Uses"
"Keeps a reference to"

"Creates"
"Instantiates"

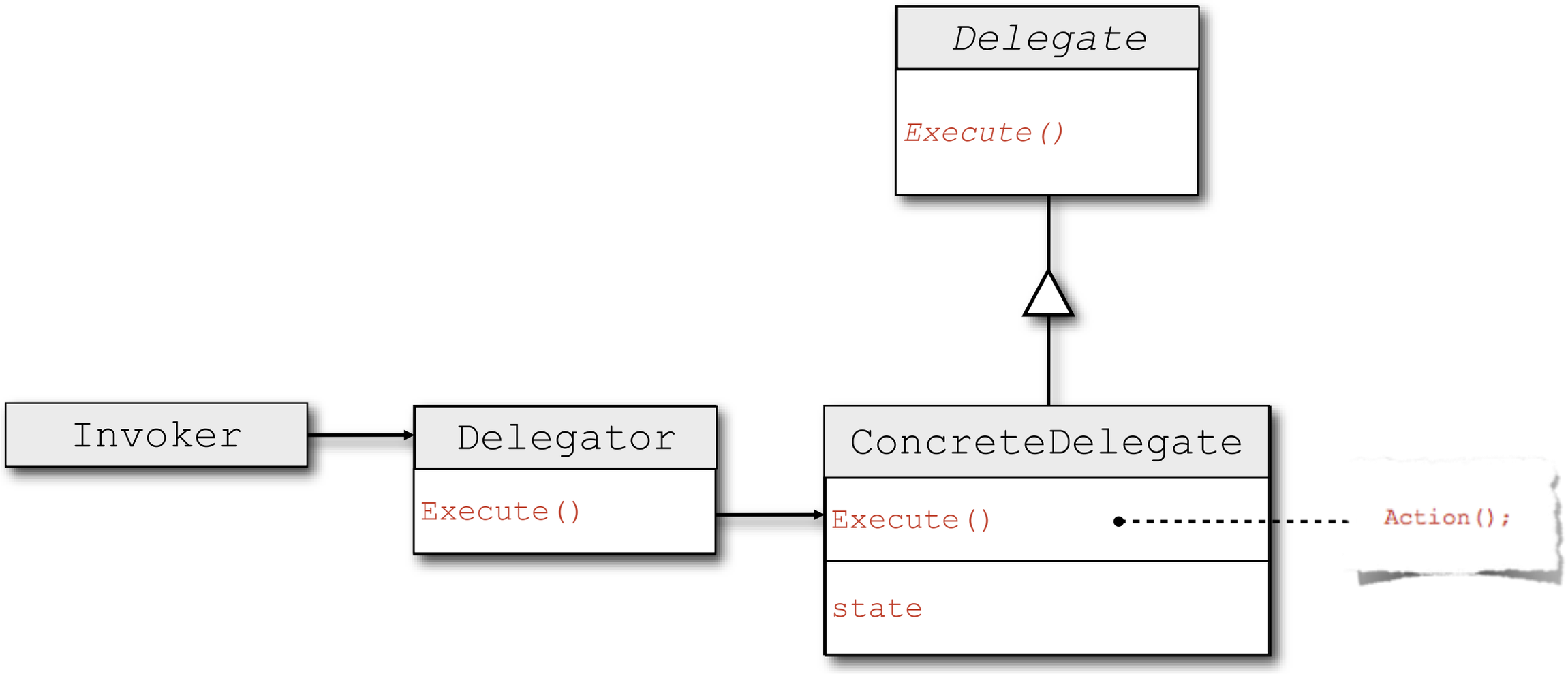
BEHAVIORAL DESIGN PATTERNS

DELEGATION

- Problem: how to modify complex objects without subclassing them

- Solution:
  - custom code for the new behavior is put inside a **separate object (the delegate);**
  - from time to time, the complex object **calls the methods of the delegate** to run its code

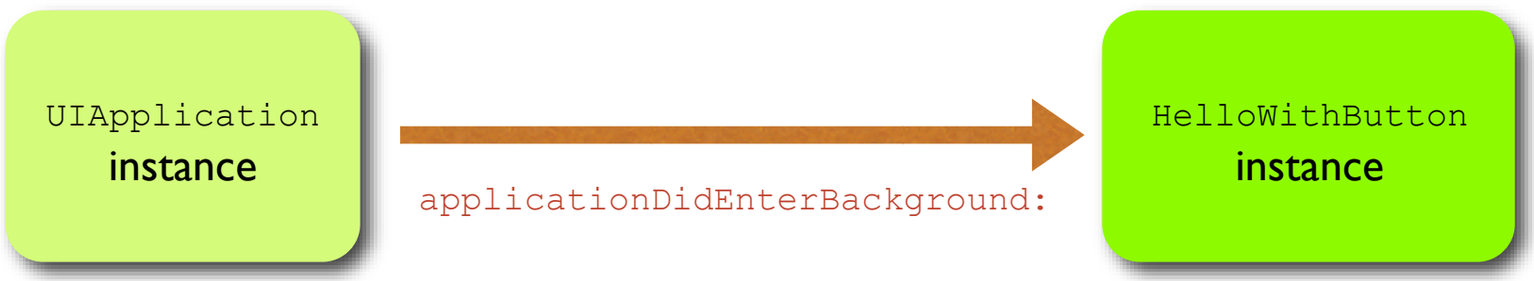
DELEGATION: STRUCTURE



DELEGATION: EXAMPLE (1/2)

- `applicationDidEnterBackground:` is part of the protocol for the so-called **application delegate**, which manages the application lifecycle in iOS

- `applicationDidEnterBackground:` is invoked immediately after the application is moved to the background



DELEGATION: EXAMPLE (2/2)

- In iOS, delegate methods are typically grouped into Objective-C **protocols**, which specify the methods that must be implemented by the delegate

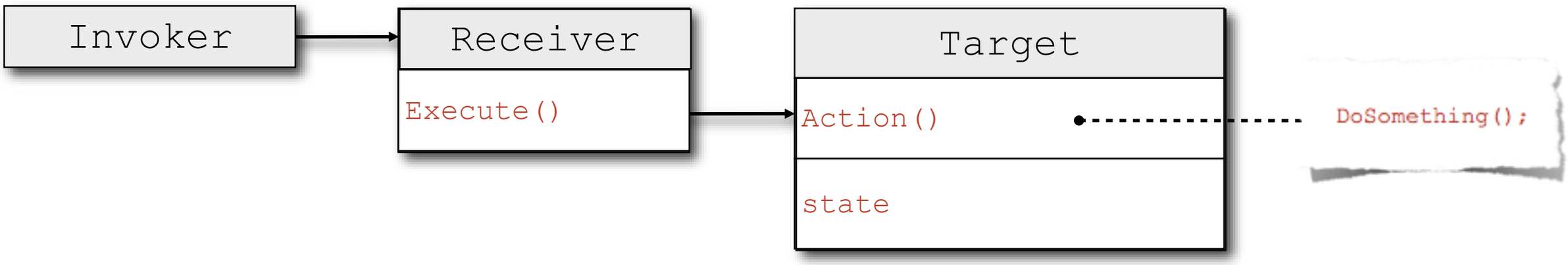
```
@protocol GeometricObject
- (double) Distance;      // Required method
// ...
@optional
- (double) Area;          // Optional method (Obj-C 2.0)
@end

@interface Poiint: NSObject <GeometricObject>
//...                     No need to declare Distance
@end

@implementation Poiint
- (double) Distance { /*... Implementation goes here*/}
//...
@end
```

TARGET-ACTION

- Problem: how to notify an application when the user interacts with a UI object

- Solution: the UI object **sends a message (the action) to another, previously-specified object (the target).**
  The target can then perform the appropriate action

TARGET-ACTION: STRUCTURE



TARGET-ACTION: EXAMPLE

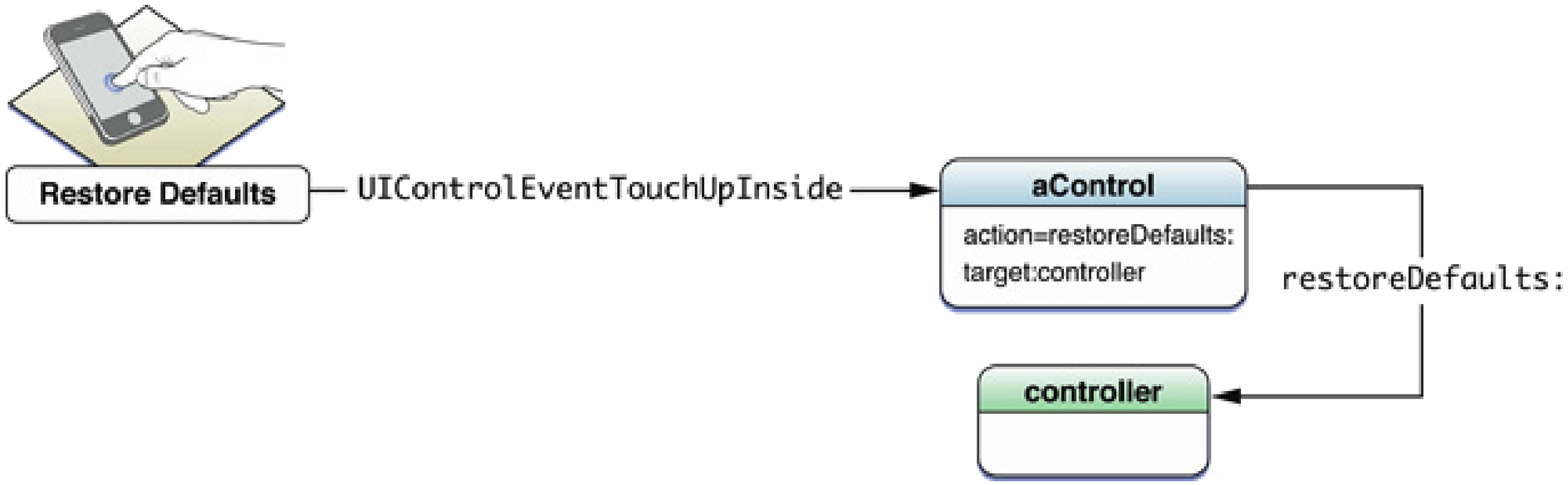


Image from the iOS Developer Library

- In iOS, the `UIControlEventTouchUpInside` event is generated every time a button (`UIButton` class) is pressed. The event can be used to invoke an action method in the view controller (`UIViewController` class)
- The connection between the button and the view controller can be performed in the IDE or in the source code

COMMAND

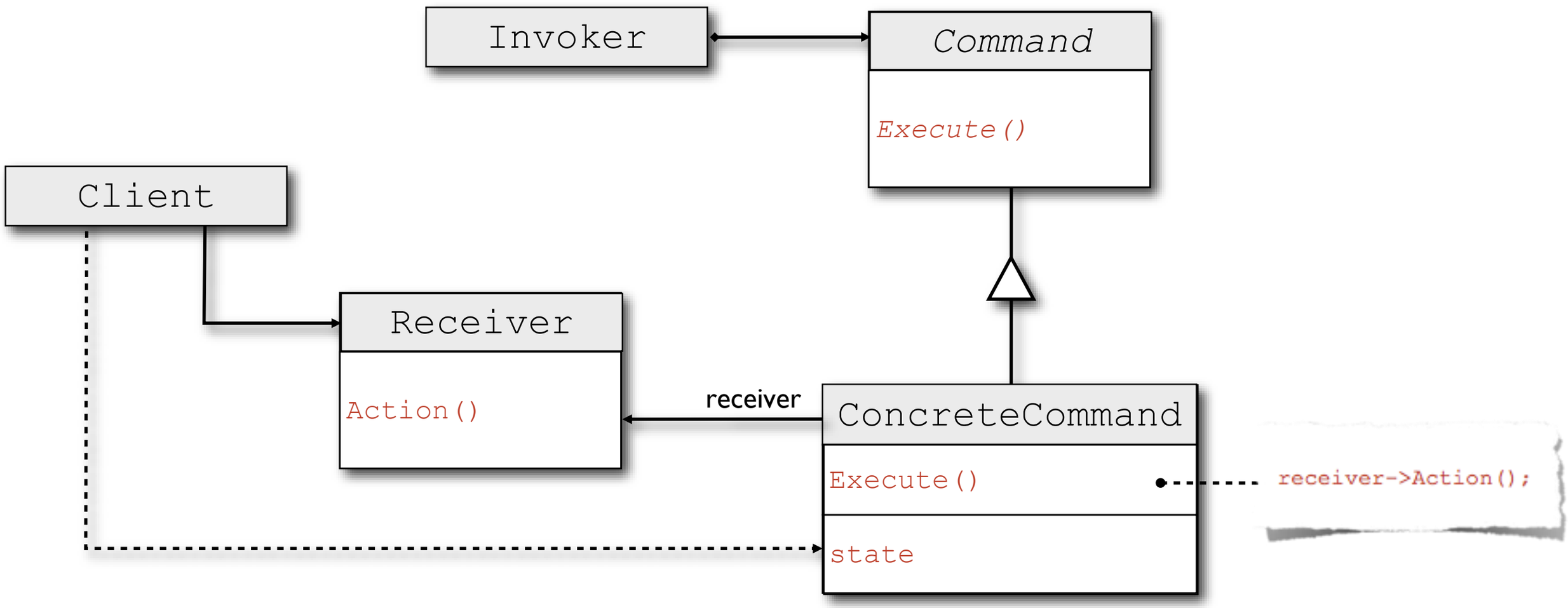
- Problem: how to issue requests to objects without knowing

  1. the details of the operation being requested,

  2. the receiver of the request

- Solution: turn the **request itself** into an **object**. An abstract Command class declares the interface for executing operations

COMMAND: STRUCTURE



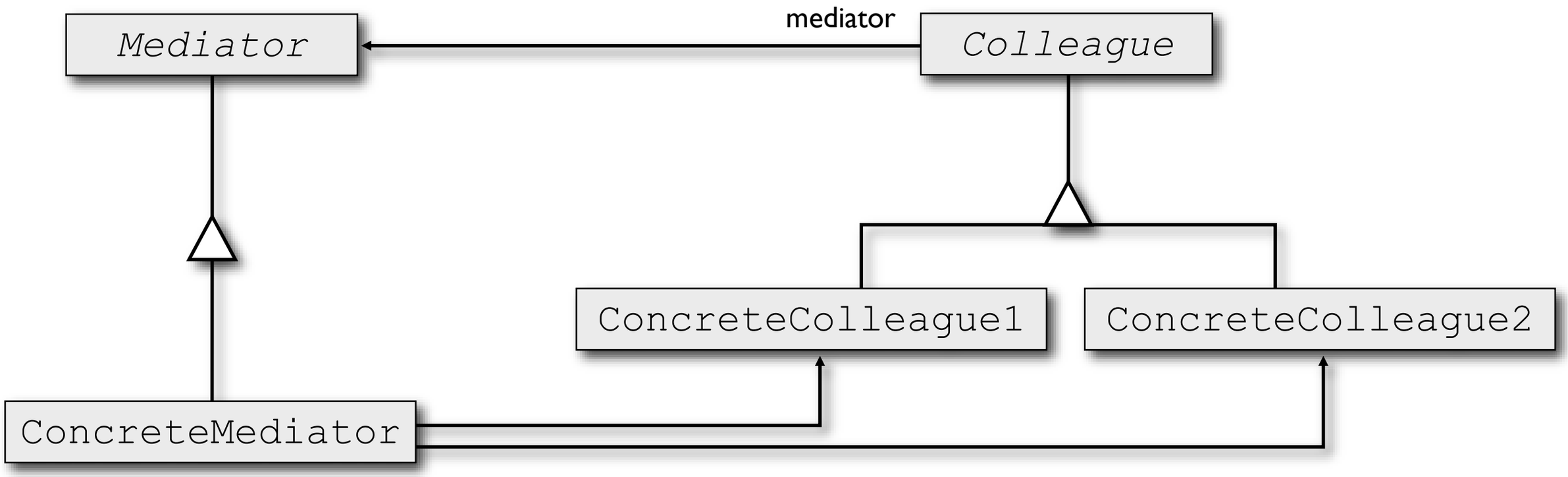
MEDIATOR

- Problem: how to manage the proliferating connections inside a set of objects, connections that arise from distributed behavior

- Solution: centralize communication in a separate **mediator** object. Now the objects communicate through the mediator

- Consequence: many-to-many interactions are replaced with one-to many interactions

MEDIATOR: STRUCTURE



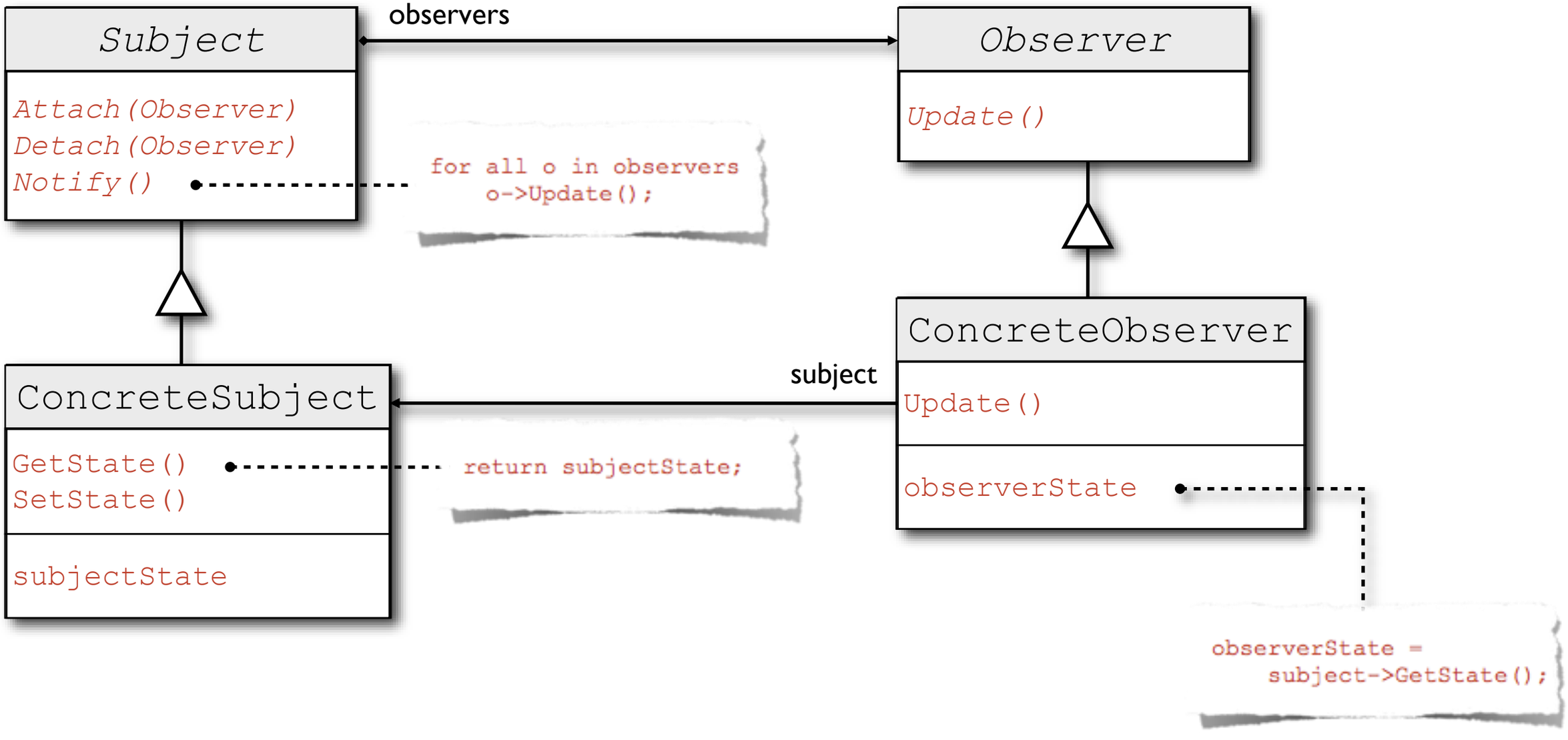
OBSERVER

- Problem: how to maintain consistency between related objects

- Solution: store information of common interest in a new **subject** object. Objects that need to maintain consistency should register with the subject as **observers** to be notified of any change

- Consequences: difficult to evaluate the implications of a state change

OBSERVER: STRUCTURE



STRUCTURAL DESIGN PATTERNS

MODEL-VIEW-CONTROLLER

- Problem: how to design an application that reads, modifies, writes and visualizes data
- Solution: divide the application's classes into **three functional areas**
  - **Model**: contains the application's data
  - **View**: manages the UI, hence it displays data among other things
  - **Controller**: coordinates the application by interacting with both the model and the view: it processes data from the model according to user input received from the view, and it displays data by sending appropriate commands to the view

M-V-C: STRUCTURE

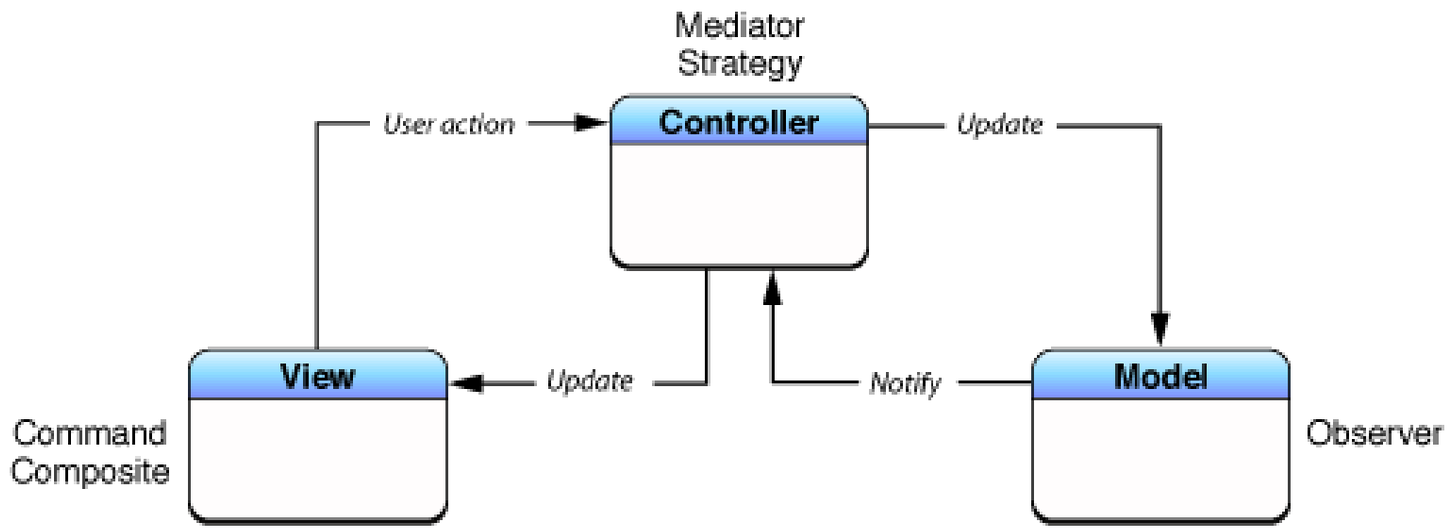
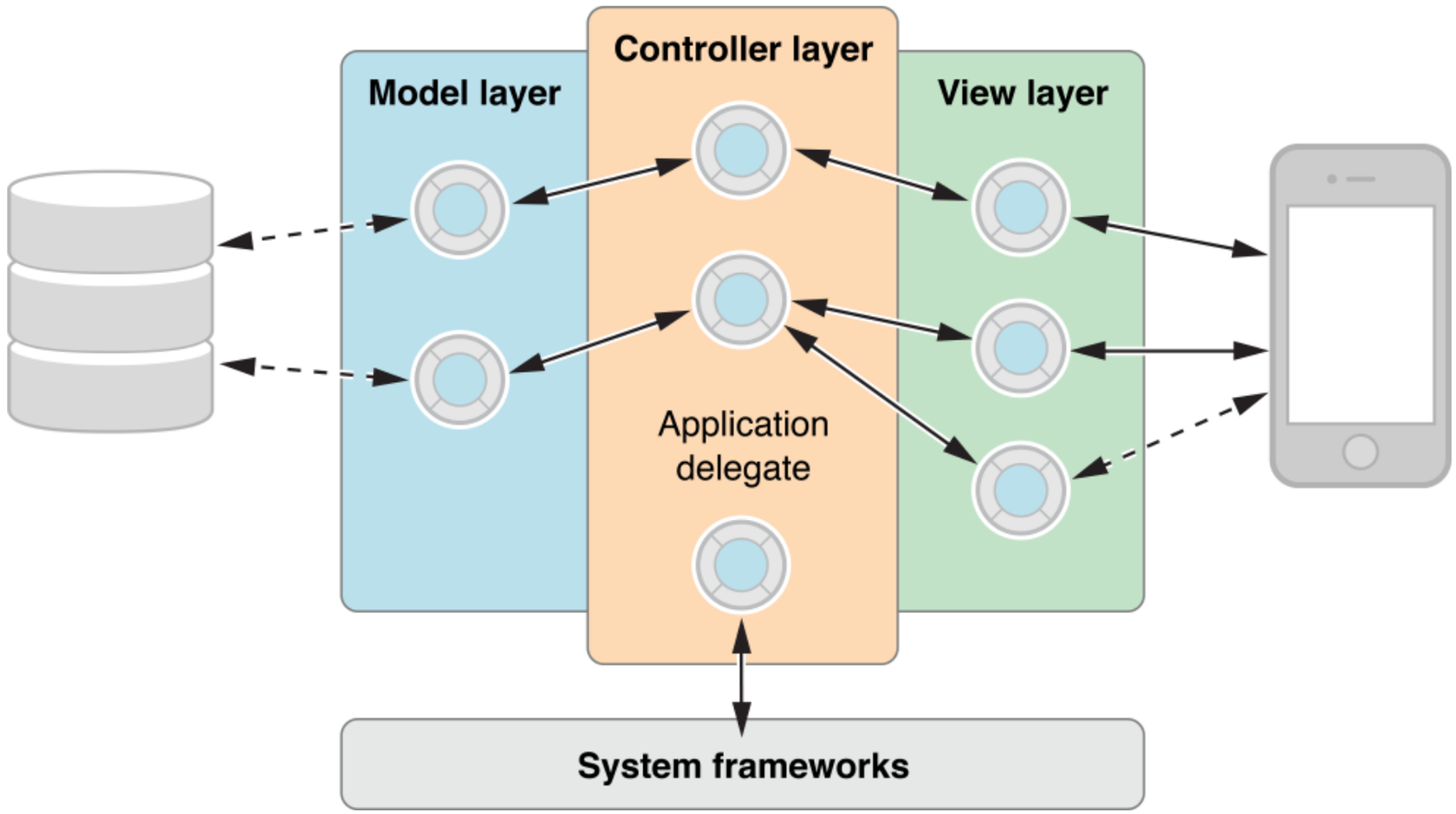


Image from the iOS Developer Library

- The model decides how data can be accessed
- The model knows nothing about the user interface: no direct communication with the view
- Ideally, the model can be reused in several applications

IOS: M-V-C AND DELEGATION



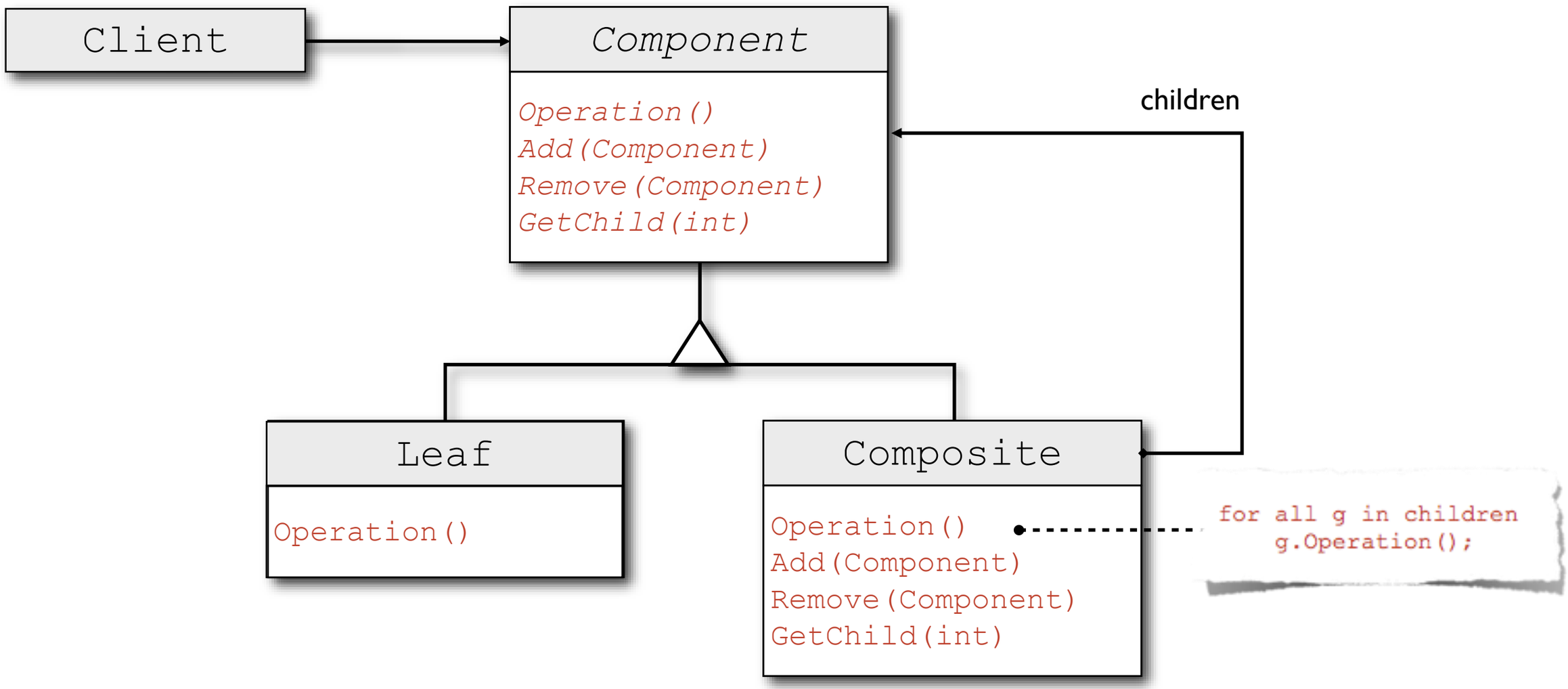
- M-V-C is compulsory in iOS; see also the "Controller Object" page in the iOS Developer Library

COMPOSITE

- Problem: how to compose simple objects to represent part-whole hierarchies, with the possibility of accessing different levels of the hierarchy in a uniform fashion

- Solution: compose objects into tree structures. The root of the tree is an abstract class that define an interface for accessing and managing children. The interface is implemented by the internal nodes of the tree

COMPOSITE: STRUCTURE



COMPOSITE: CONSEQUENCES

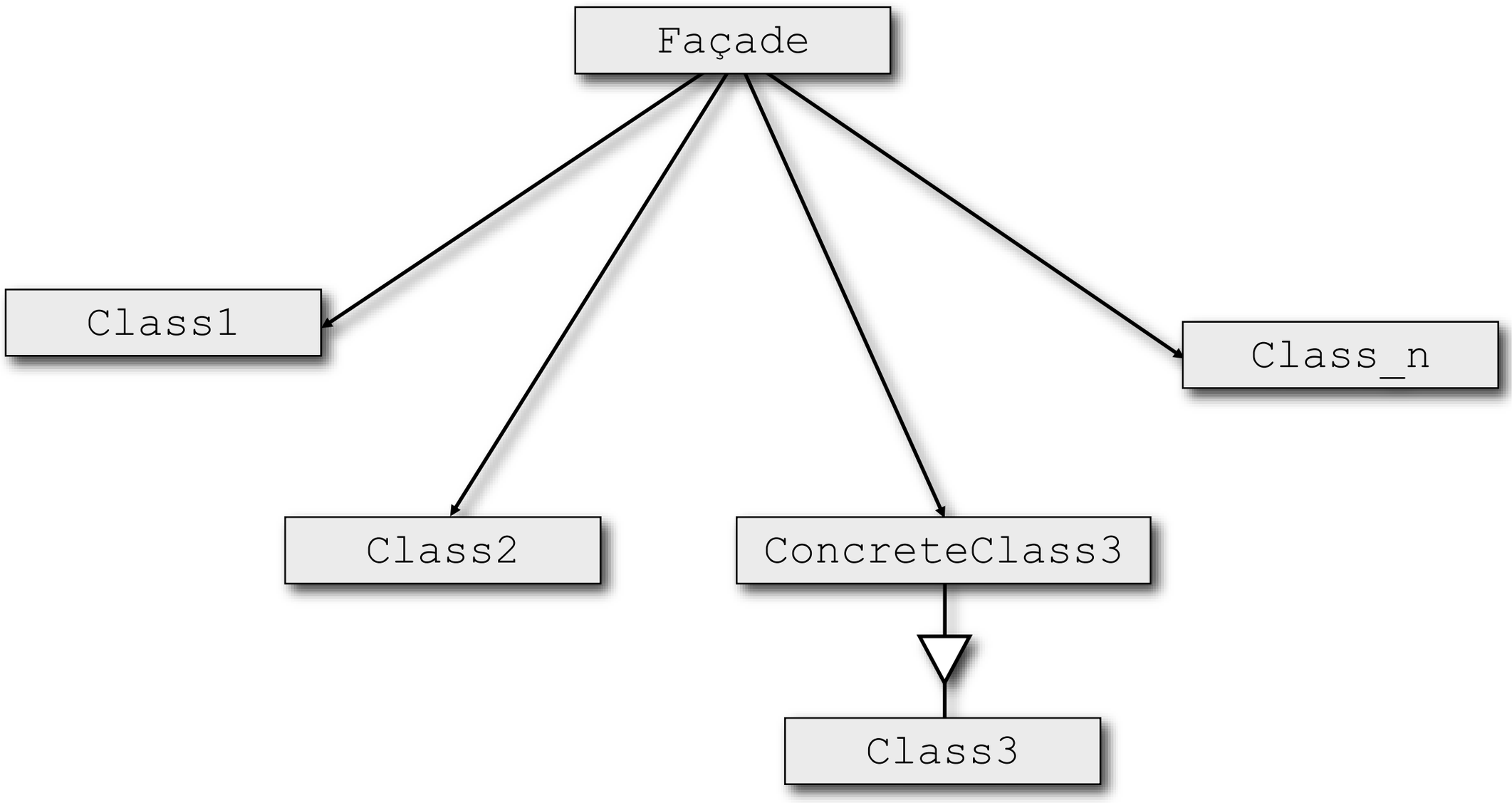
- The design may be overly general
- Difficult to define necessary, common methods in the abstract class, where nothing is known about the children
- Each children can have only one parent
- Issue: traversing the composite structure. Simplified if references from children to parents are maintained
- Possible issue: child ordering

FAÇADE

- Problem: how to provide a unified interface to a set of facilities

- Solution: introduce a **façade** object that provides a single, higher-level interface to the underlying facilities

FAÇADE: STRUCTURE



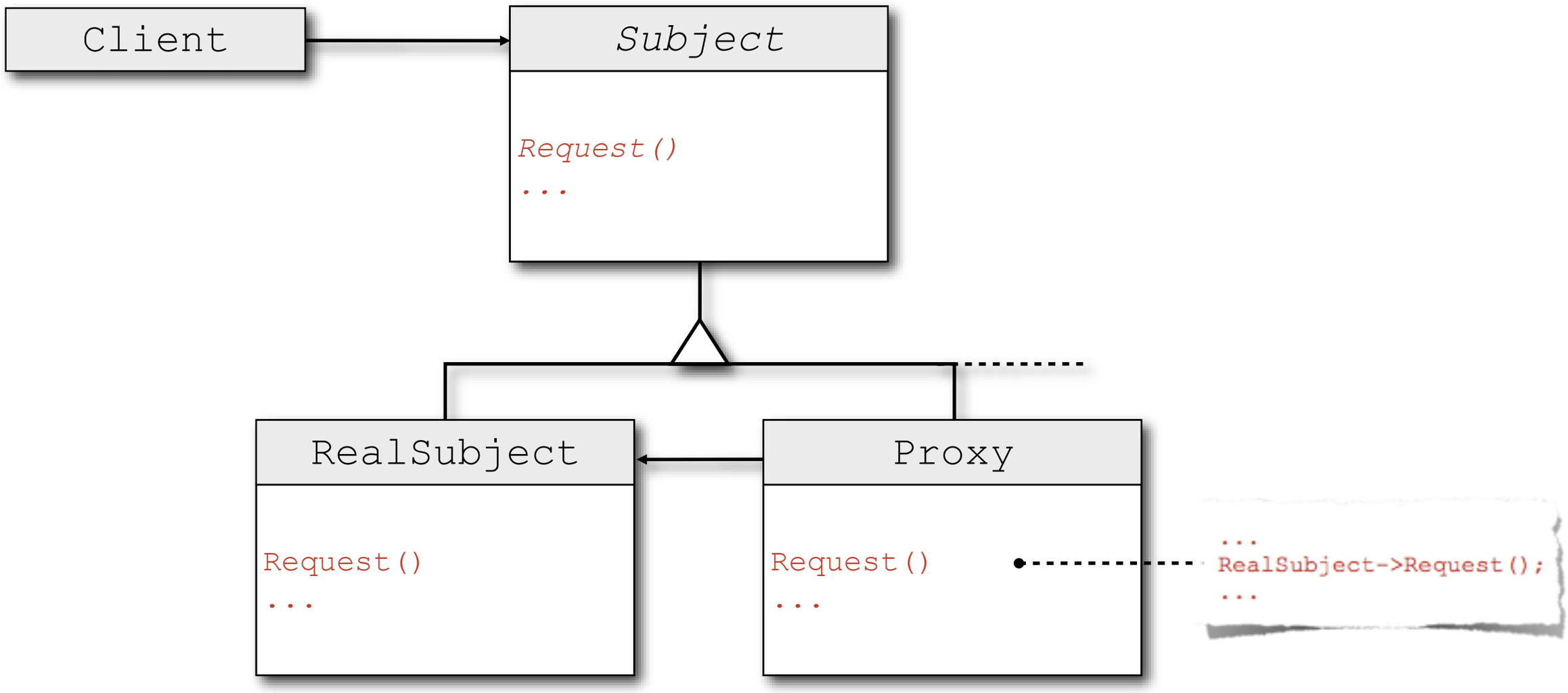
PROXY

- Problem: how make a complex object appear as instantiated and available, while its expensive allocation is actually deferred to the moment when the object is accessed

- Solution: provide a stand-in (**proxy**) for the real object. The proxy takes care of instantiating the real object when needed

- Consequences: a level of indirection is introduced

PROXY: STRUCTURE



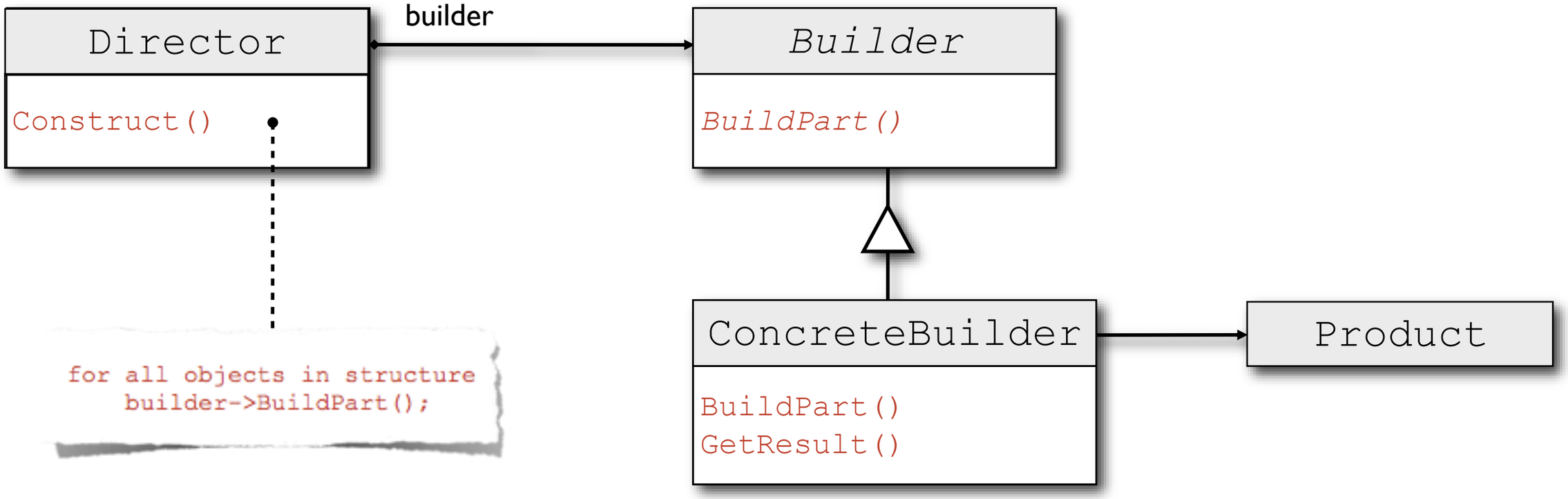
CREATIONAL DESIGN PATTERNS

BUILDER

- Problem: how to create several (the number is unknown *a priori*) representations of an object

- Solution: separate the construction of the object from its representations. Create many **builder** objects that operate under the guidance of a single **director** object

BUILDER: STRUCTURE



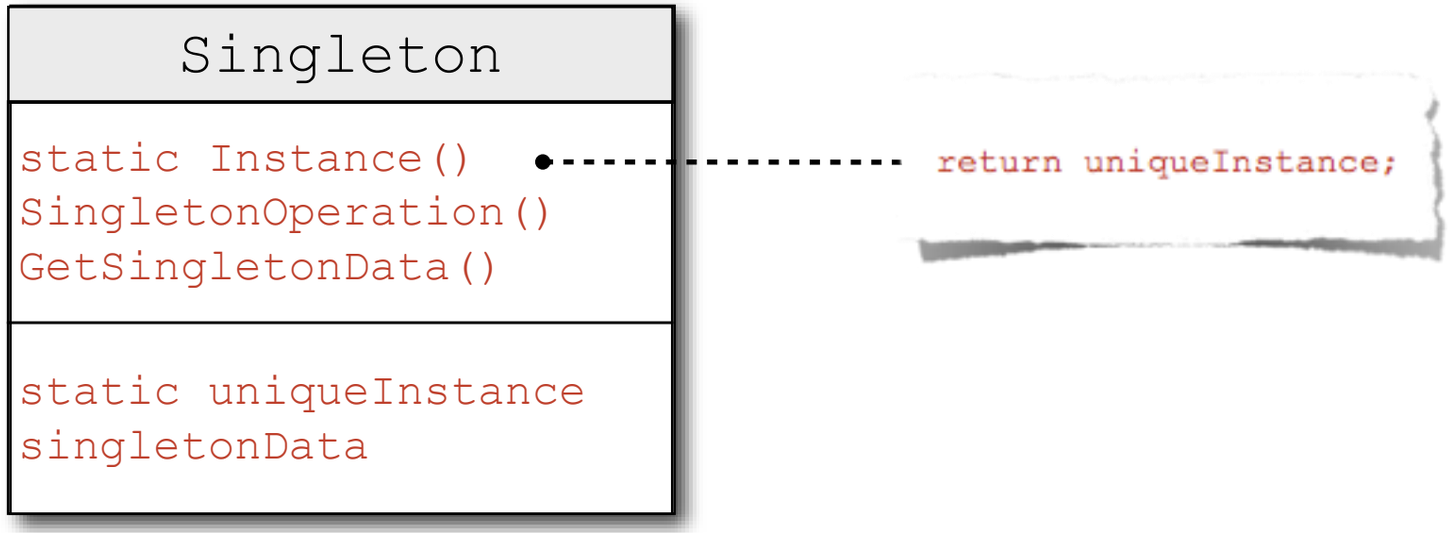
SINGLETON

- Problem: how to ensure that a class has exactly one instance and that the instance is easily accessible

- Solution: **make the class itself responsible for keeping track of its sole instance**. The class also provides a documented way to access the instance

- Consequences: it is easy to allow a fixed, greater-than-one number of instances

SINGLETON: STRUCTURE



MORE ON DESIGN PATTERNS

- Classic **book**: "Design Patterns: Elements of Reusable Object-Oriented Software." ISBN-10: 0201633612

- **Android**: "Android UI Design Patterns", "Design Patterns with Godfrey Nolan." Note that https://developer.android.com/design/patterns/ does not talk about DPs

- **Apple platforms**: "Basic Programming Concepts for Cocoa", "Adopting Cocoa Design Patterns"

LAST MODIFIED: MARCH 6, 2018