Design Patterns & iOS

From abstraction to real value

Marcin Iwanicki

@marciniwanicki - marciniwanicki.com - marciniwanicki.dev (skype)
https://github.com/marciniwanicki/OCDesignPatterns

Overture – Software Development Philosophies

Done is better than perfect.

LIM: Less is more.

KISS: Keep it simple, stupid.

DRY: Don't repeat yourself.

DIE: Duplication is evil.

YAGNI: You aren't gonna need it.

Overture – Object-oriented programming

Object-oriented programming is hard, and designing reusable object-oriented software is even harder [3].

Before your code will be reusable it must actually be usable.

Overture - Gang of Four

Erich Gamma realized the importance of recurring design patterns while working on his PhD thesis and prior to the 1991 European Conference on Object-Oriented Programming he and Richard Helm stared to catalog patterns.

In 1991 Gamma and Helm were joined by Ralph Johnson and John Vlissides.

Contents

What is a Design Pattern?

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [4].

Design Pattern is a general reusable solution to a commonly occurring problem.

What is a Design Pattern?

Patterns are abstract design, not code.

Patterns that imply object-orientation or generally mutable state, are not as applicable in functional programming languages.

What is a Design Pattern?

Design Patterns required neither unusual language features nor amazing programming tricks. **They might take a little more work than ad hoc solutions** but the extra effort invariably pays dividends in **increased flexibility** and **reusability** [3].

What is a Design Pattern?

Design Patterns == Wzorce Projektowe

Design Patterns < Wzorce Projektowe

Elements of a Design Pattern

In general, a pattern has four essential elements:

1. Name
2. Problem
3. Solution
4. Consequences

Types of Design Patterns

Design patterns were grouped into the categories:

1. Creational patterns
2. Structural patterns
3. Behavioral patterns
4. Concurrency patterns

Design Patterns in Cocoa

Start Developing iOS Apps Today

1. Model-View-Controller (MVC)
2. Target-action
3. Delegation

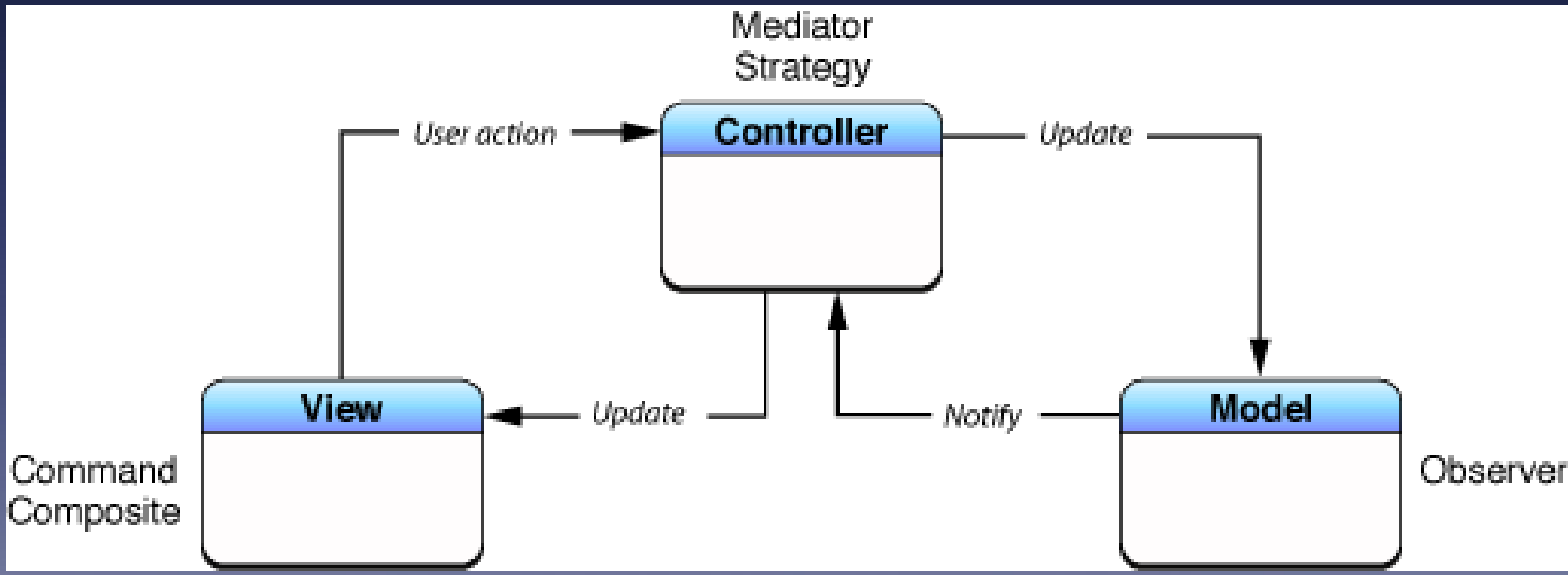
4. Observer
5. DAO (Facade)

Model-View-Controller (MVC)

The Model-View-Controller (MVC) design pattern assigns objects in an application one of three roles: model, view, or controller. The pattern defines not only the roles objects play in the application, it defines the way objects communicate with each other [6].

Model-View-Controller (MVC)

Cocoa version of MVC



Model-View-Controller (MVC)

View objects

A view object is an object in an application that users can see. A view object knows how to draw itself and can respond to user actions. A major purpose of view objects is to display data from the application's model objects and to enable the editing of that data [6].

Model-View-Controller (MVC)

Model objects

Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data.
Ideally, a model object should have no explicit connection to the view objects that present its data and allow users to edit that data [6].

Model-View-Controller (MVC)

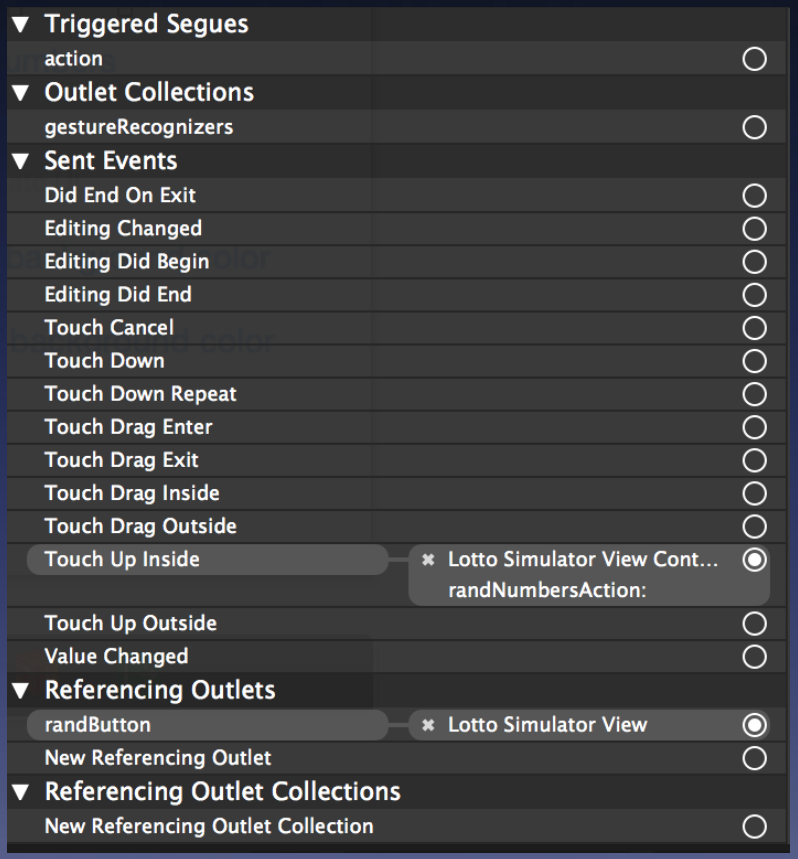
Controller objects

A controller object acts as an intermediary between one or more of an application's view objects and one or more of its model objects [6].

Target-action

Target-action is a conceptually simple design in which one object sends a message to another object when a specific event occurs [5].

Target-action




```
- (IBAction)randNumbersAction:
(id)sender
{
    // Do something
}
```

Target-action

```
 [self.button addTarget:self
action:@selector(doSomething:)
forControlEvents:UIControlEventTouchUpInside];

...

- (void)doSomething:(id)sender
{
    // Do something
}
```

Delegation

Delegation is a simple and powerful pattern in which one object in an app acts on behalf of, or in coordination with, another object. The delegating object keeps a reference to the other object—the delegate—and at the appropriate time sends a message to it [5].

Delegation

```
@protocol UIScrollViewDelegate<NSObject>

...
- (void)scrollViewDidScroll:(UIScrollView *)scrollView;
                                    // any offset changes
- (void)scrollViewDidZoom:(UIScrollView *)scrollView
NS_AVAILABLE_IOS(3_2); // any zoom scale changes

...
- (BOOL)scrollViewShouldScrollToTop:(UIScrollView *)scrollView;   //
return a yes if you want to scroll to the top. if not defined, assumes
YES
- (void)scrollViewDidScrollToTop:(UIScrollView *)scrollView;      //
called when scrolling animation finished. may be called immediately if
already at top

...
```

Delegation

```
@protocol UITableViewDataSource<NSObject>

@required

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
(NSInteger)section;

// Row display. Implementers should *always* try to reuse cells by
setting each cell's reuseIdentifier and querying for available
reusable cells with dequeueReusableCellWithIdentifier:
// Cell gets various attributes set automatically based on table
(separators) and data source (accessory views, editing controls)

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath;

...
```

Delegation

```
@protocol UITableViewDelegate<NSObject, UIScrollViewDelegate>

...
// Variable height support
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:
(NSIndexPath *)indexPath;
- (CGFloat)tableView:(UITableView *)tableView
heightForHeaderInSection:(NSInteger)section;
- (CGFloat)tableView:(UITableView *)tableView
heightForFooterInSection:(NSInteger)section;

...
// Called after the user changes the selection.
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
(NSIndexPath *)indexPath;
- (void)tableView:(UITableView *)tableView didDeselectRowAtIndexPath:
(NSIndexPath *)indexPath NS_AVAILABLE_IOS(3_0);

...
```

Observer

1. Notifications
2. KVO

Data Access Object (DAO)

DAO is an object that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, DAOs provide some specific data operations without exposing details of the database.

Other popular Design Patterns

Creational patterns:

1. Abstract factory
2. Builder
3. Factory method
4. Lazy initialization
5. Object pool
6. Prototype
7. Singleton

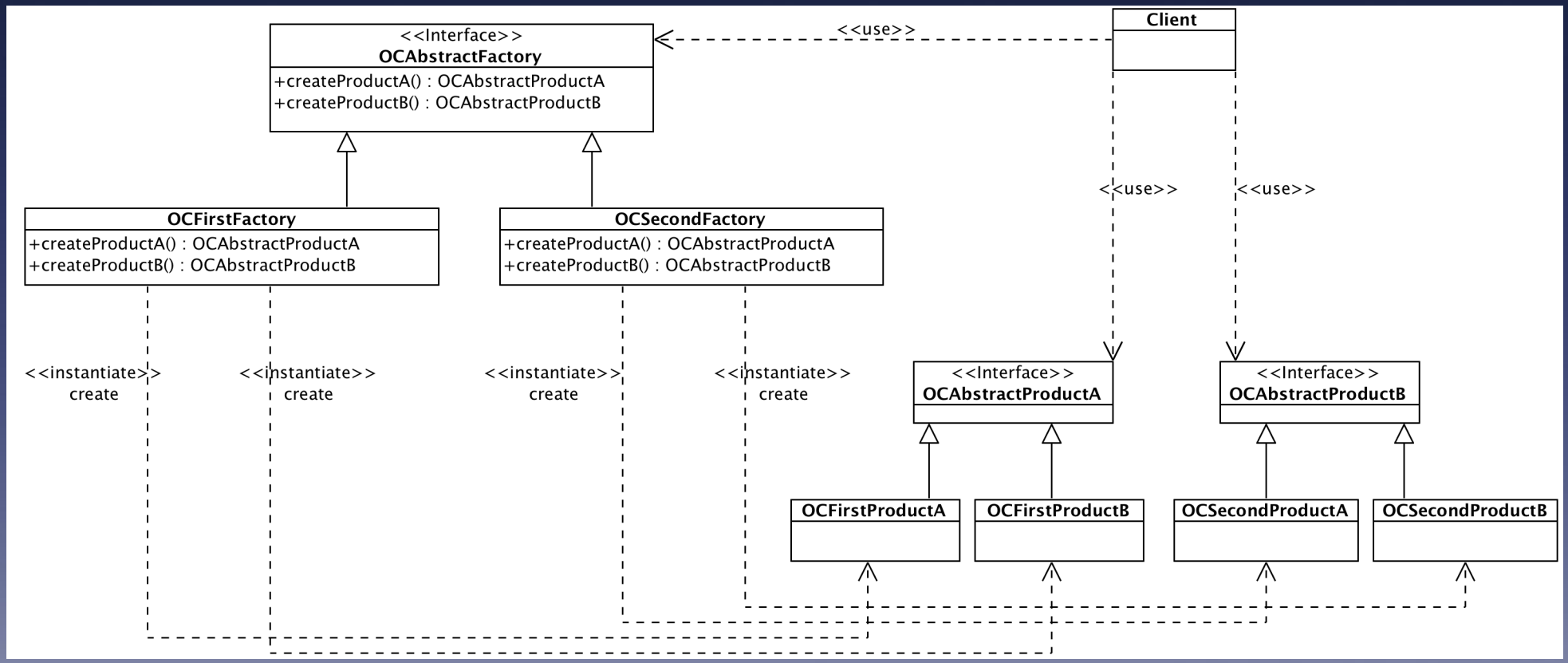
Abstract factory

Abstract factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes [3].

Abstract factory

Class diagram



Abstract factory

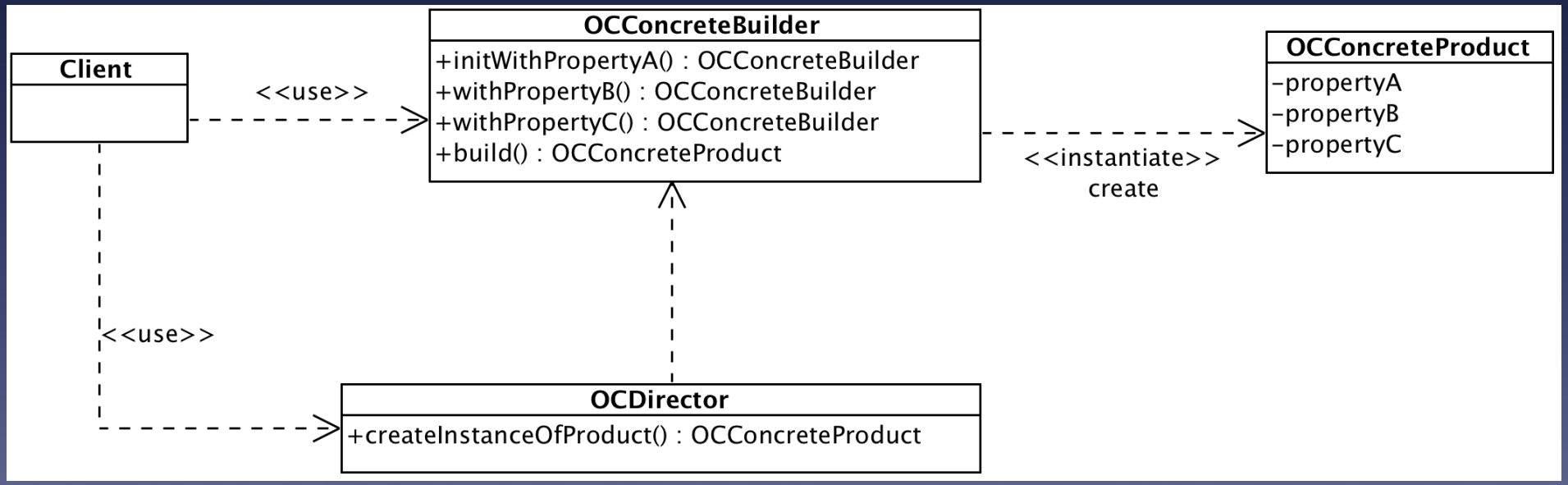
1. It isolates concrete classes.
2. It makes exchanging product families easy.
3. It promotes consistency among products.
4. Supporting new kinds of products is difficult.

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations [3].

Builder

Class diagram



Builder

Classical Builder Pattern:

```
OCConcreteBuilder *builder = [[OCConcreteBuilder alloc]
initWithPropertyA:1];

OCConcreteProduct *product = [[[builder withPropertyB:2]
withPropertyC: 3] build];
```

Builder

Using category:

```
OCConcreteProductB *productB = [[[[OCConcreteProductB
alloc] initWithPropertyA:1] withPropertyB:2]
withPropertyC:3];
```

Builder

1. It lets you vary a product's internal representation.
2. It isolates code for construction and representation.
3. It gives you finer control over the construction process.

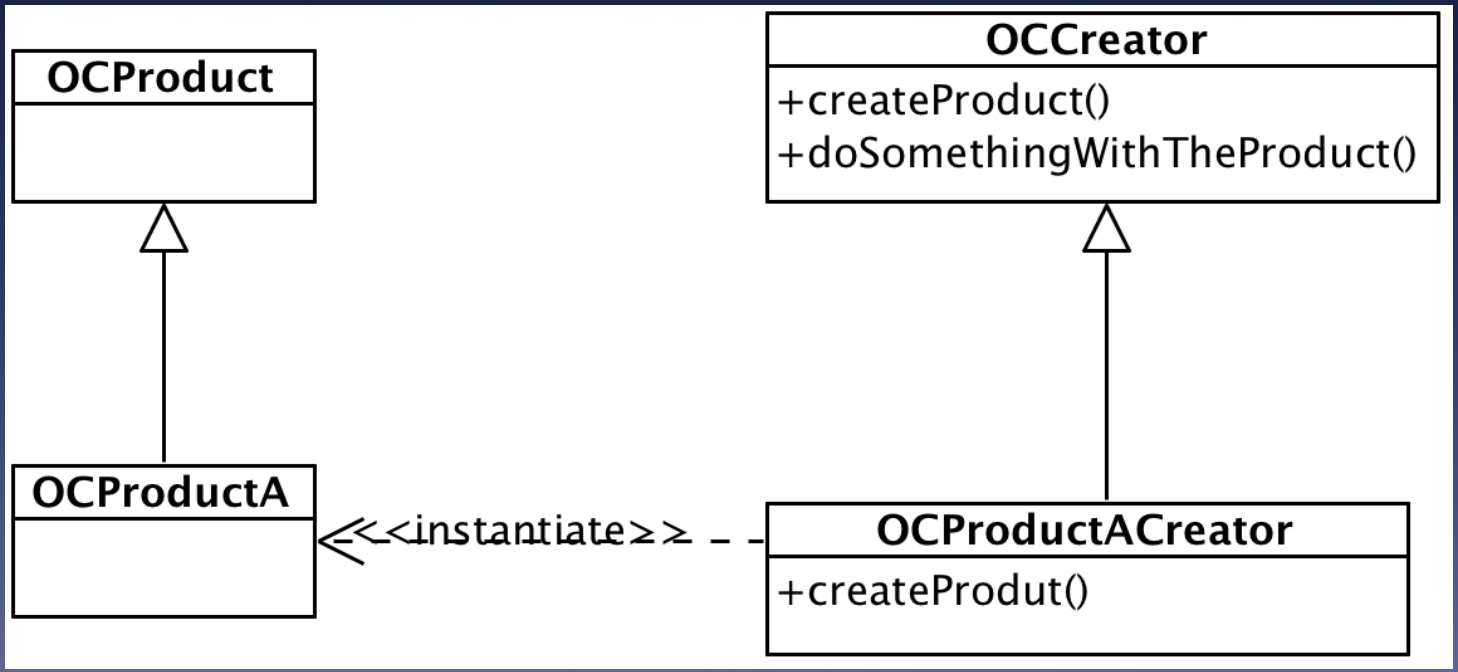
Factory method

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory Method lets a class defer instantiation to subclasses.

Factory method

Class diagram



Factory method

```
id<OCProduct> productA = [OCFactory
createProduct:ProductTypeA];

id<OCProduct> productB = [OCFactory
createProduct:ProductTypeB];
```

Factory method

1. Factory methods eliminate the need to bind application-specific classes into your code.
2. Connects parallel class hierarchies.

A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object.

Lazy initialization

Lazy initialization is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed [9].

Lazy initialization

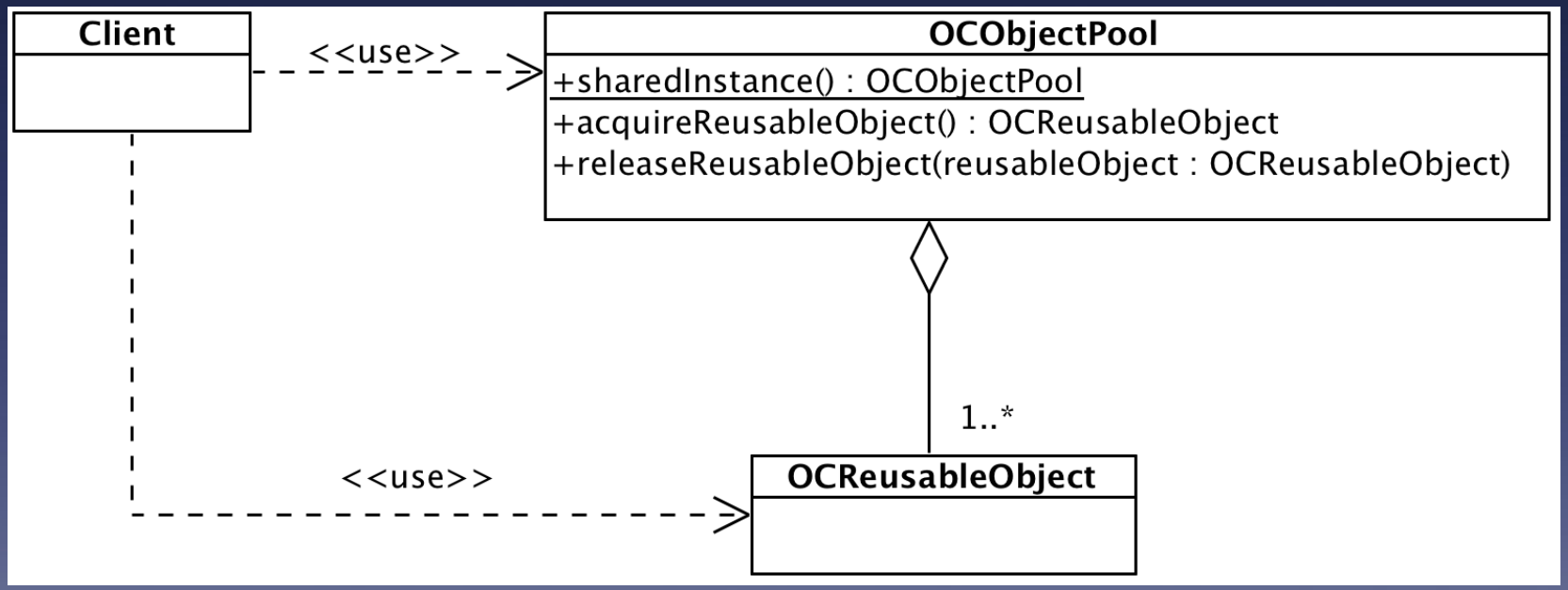
```
@property (readonly) OCElement *element;

...
// @synthesize
...

- (OCElement *)element
{
    if (_element == nil) {
        _element = [OCElement new];
    }
    return _element;
}
```

Object pool

Object pool uses a set of initialized objects kept ready to use, rather than allocating and destroying them on demand.

Object pool

Class diagram



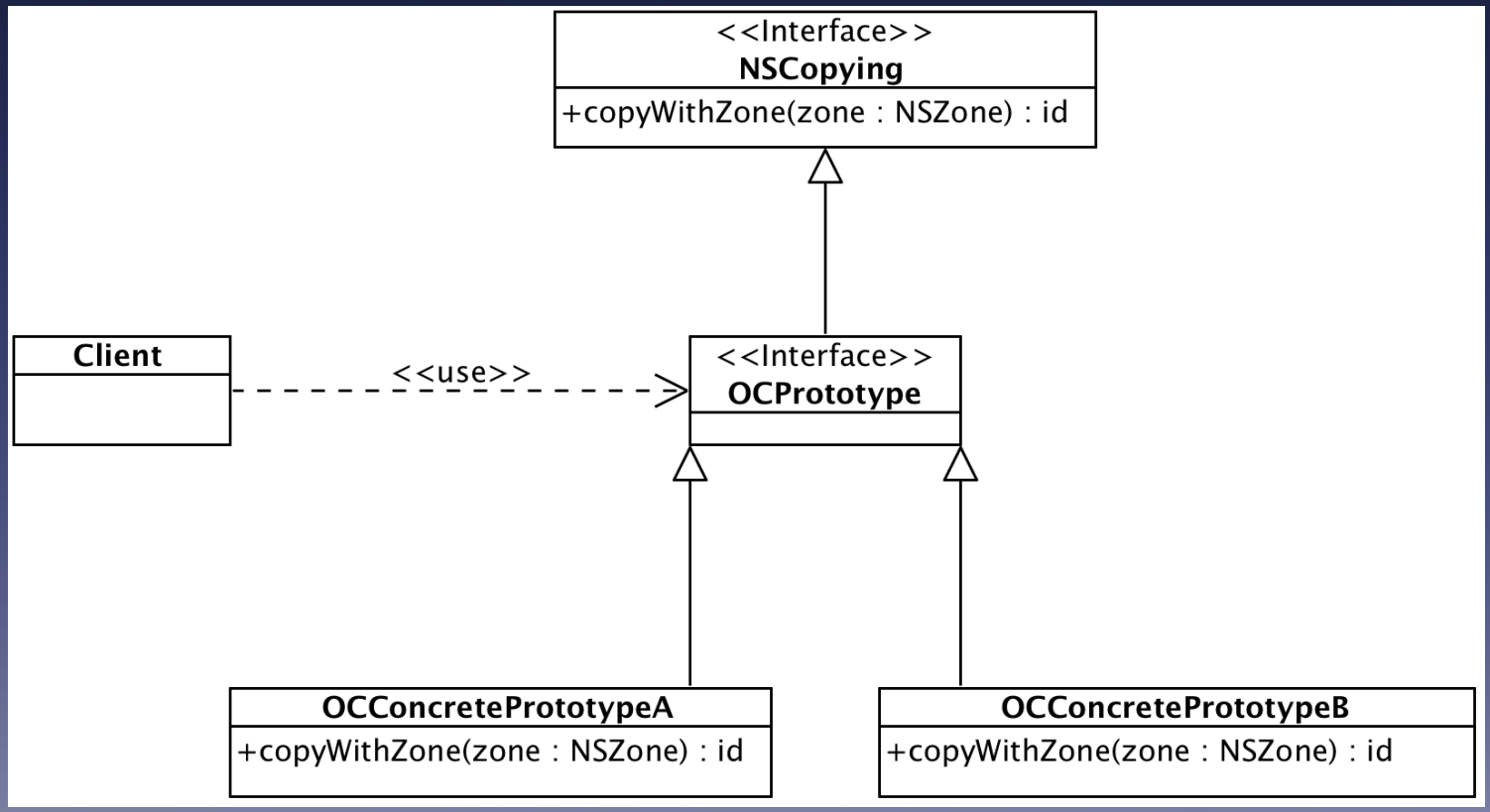
Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype [3].

```
@property (nonatomic, copy) OCConcretePrototypeA *prototype;

...

id<OCPrototype> prototype = [OCConcretePrototypeA new];
id<OCPrototype> prototypeCopy = [prototype
copyWithZone:NULL];
```

Prototype

Class diagram



Prototype

Your options for implementing NSCopying protocol are as follows:

1. Implement NSCopying using alloc and init.
2. Implement NSCopying by invoking the superclass's copyWithZone.
3. Implement NSCopying by retaining the original object.

Singleton

Common singletons:

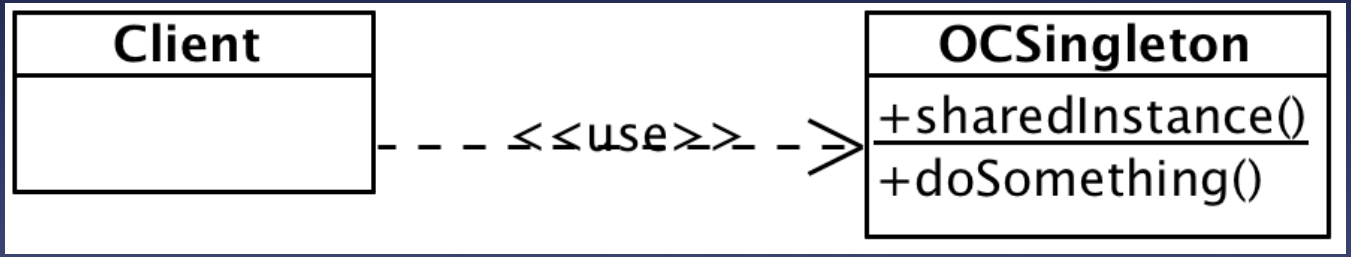
```
[UIApplication sharedApplication];
[NSFileManager defaultManager];
[NSNotificationCenter defaultCenter];
[UIDevice currentDevice];
```

Singleton

Use the Singleton pattern when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point [3].

Singleton

Class diagram



Other popular Design Patterns

Structural patterns:

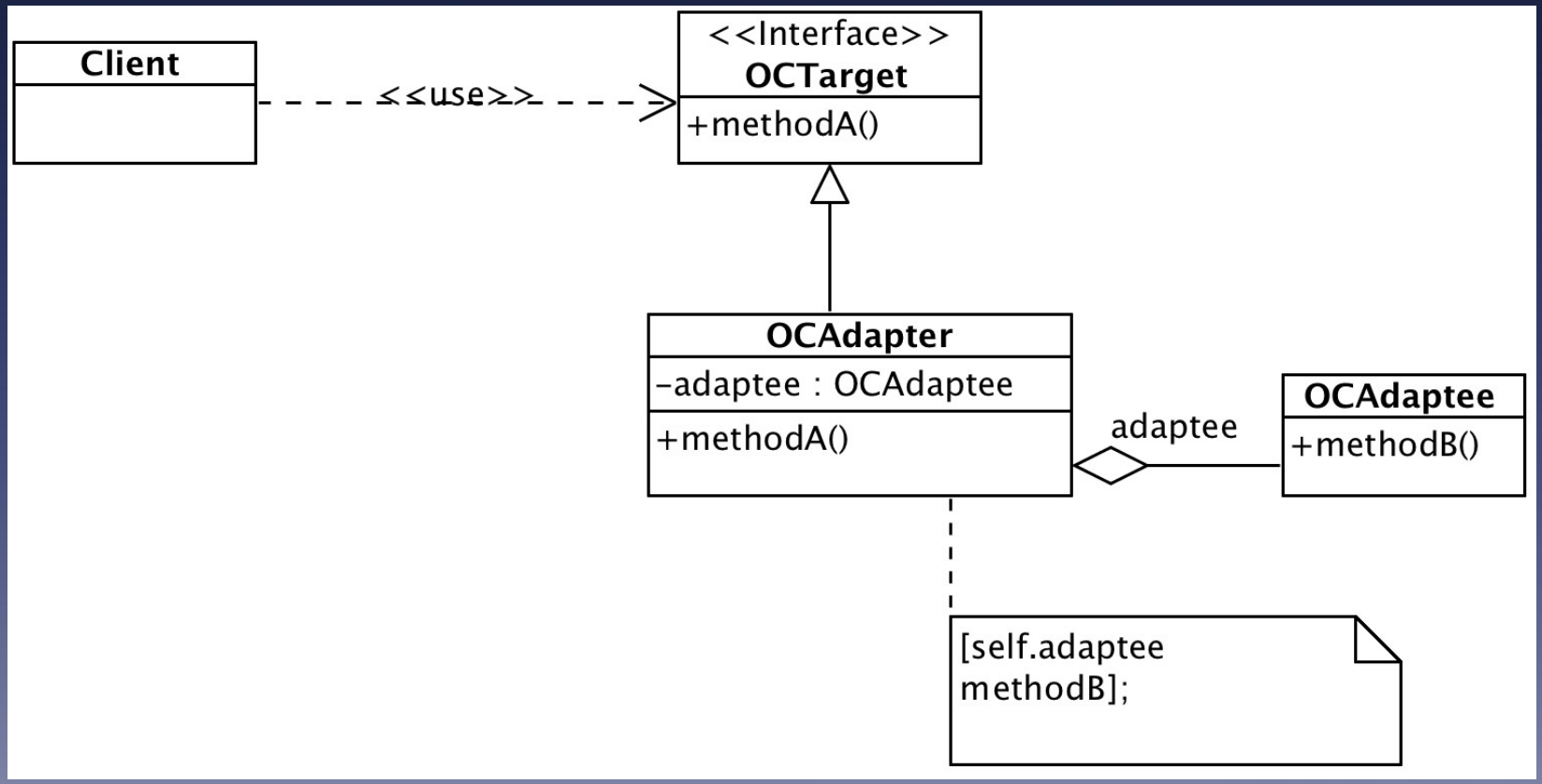
1. Adapter (Wrapper)
2. Bridge
3. Composite
4. Decorator
5. Facade
6. Proxy

Adapter (Wrapper)

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces [3].

Adapter (Wrapper)

Class diagram



Adapter (Wrapper)

1. How much adapting does Adapter do?

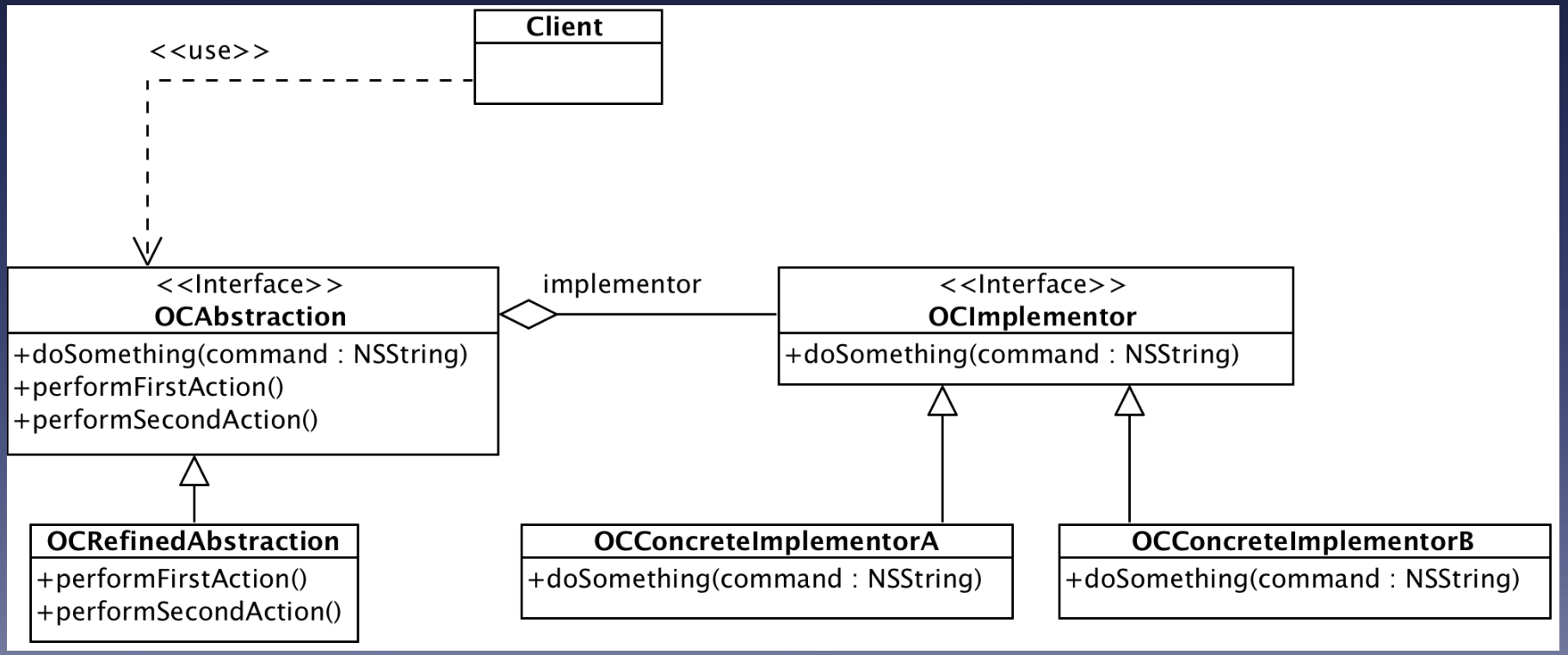
2. An adapter class is more reusable when you minimize the assumptions other classes must make to use it.

Bridge

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

Bridge

Class diagram

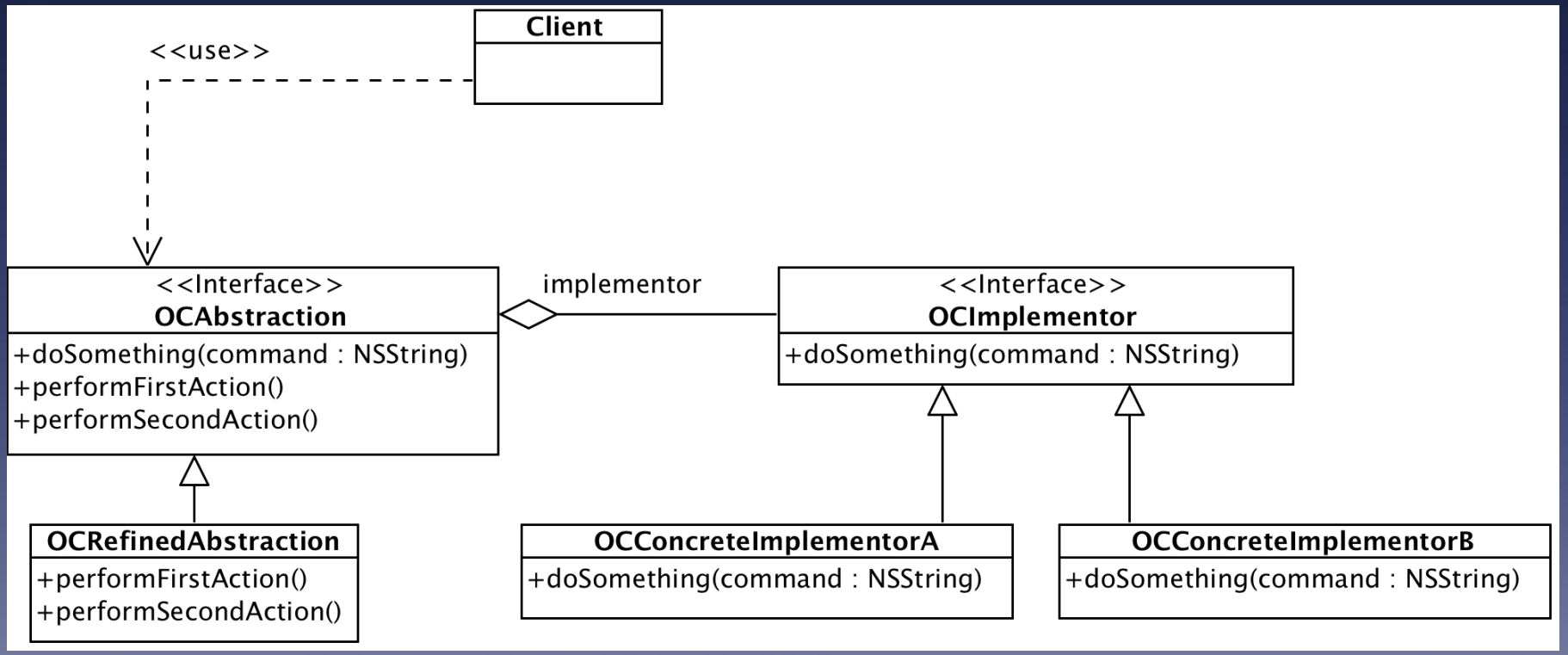


Bridge

1. Improved extensibility.
2. Hiding implementation details from clients.

Bridge

Class diagram



Composite

How to create XML, HTML, or PDF documents dynamically?

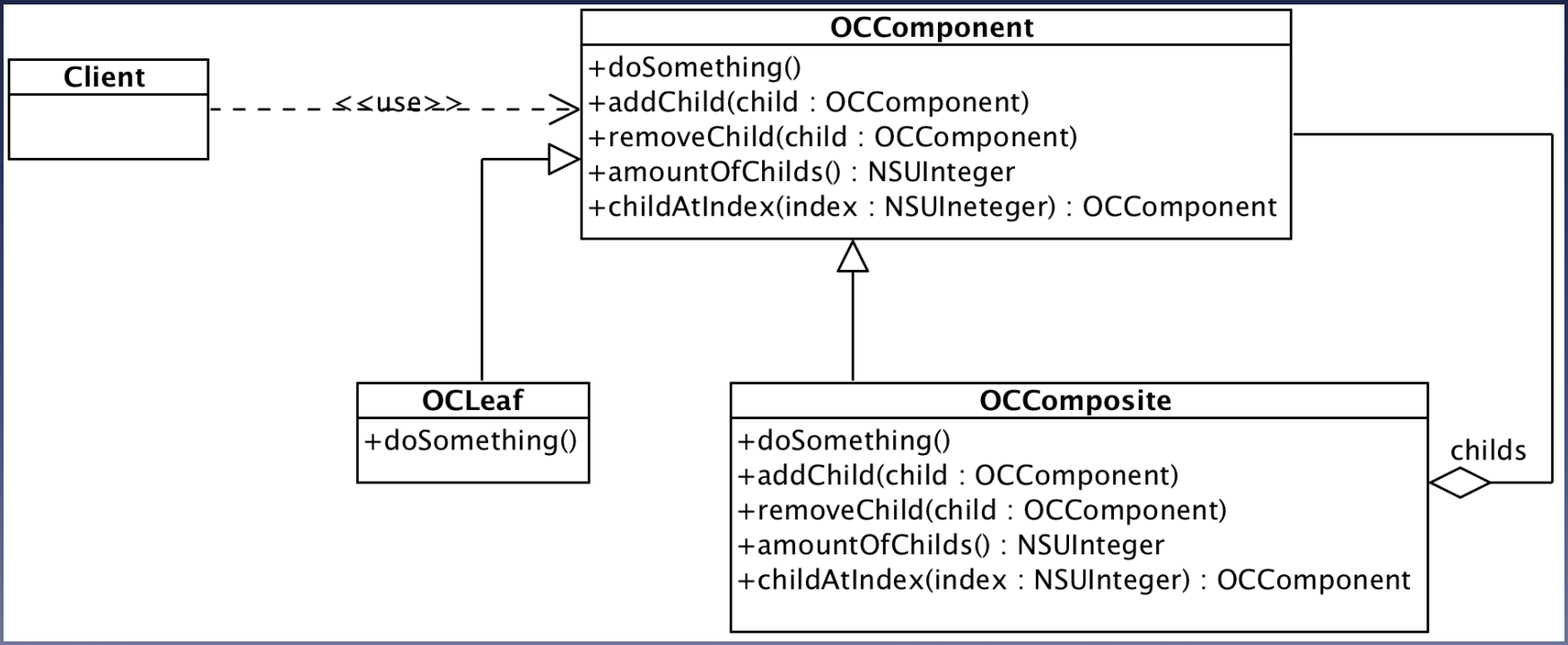
How to create a complex UIView dynamically?

Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly [3].

Composite

Class diagram



Composite

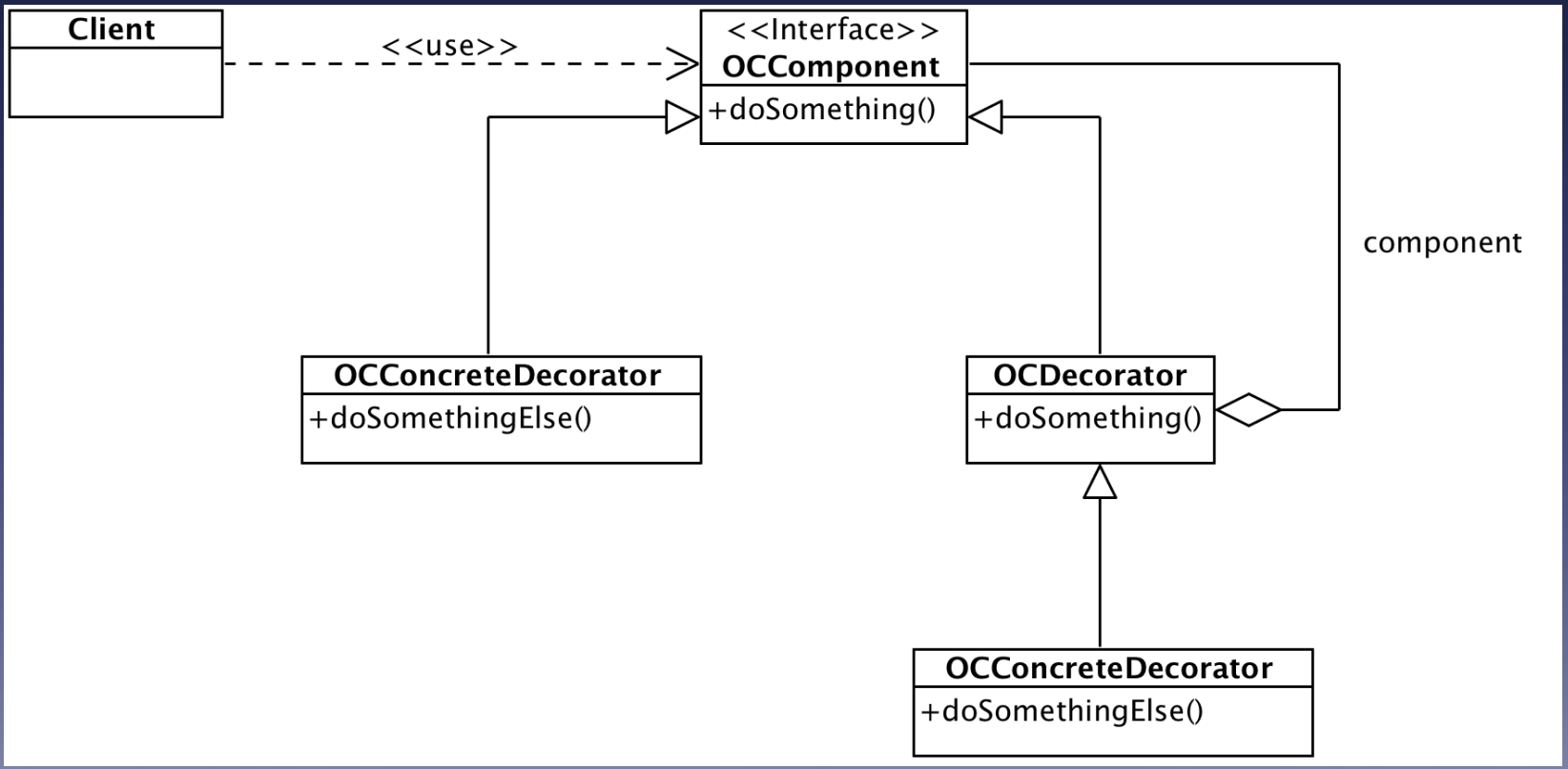
1. Explicit parent references.
2. Sharing components.
3. Maximizing the Component interface.
4. Declaring the child management operations.
5. Should Component implement a list of Components?

Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality [3].

Decorator

Class diagram



Decorator

1. More flexibility than static inheritance. (run-time)

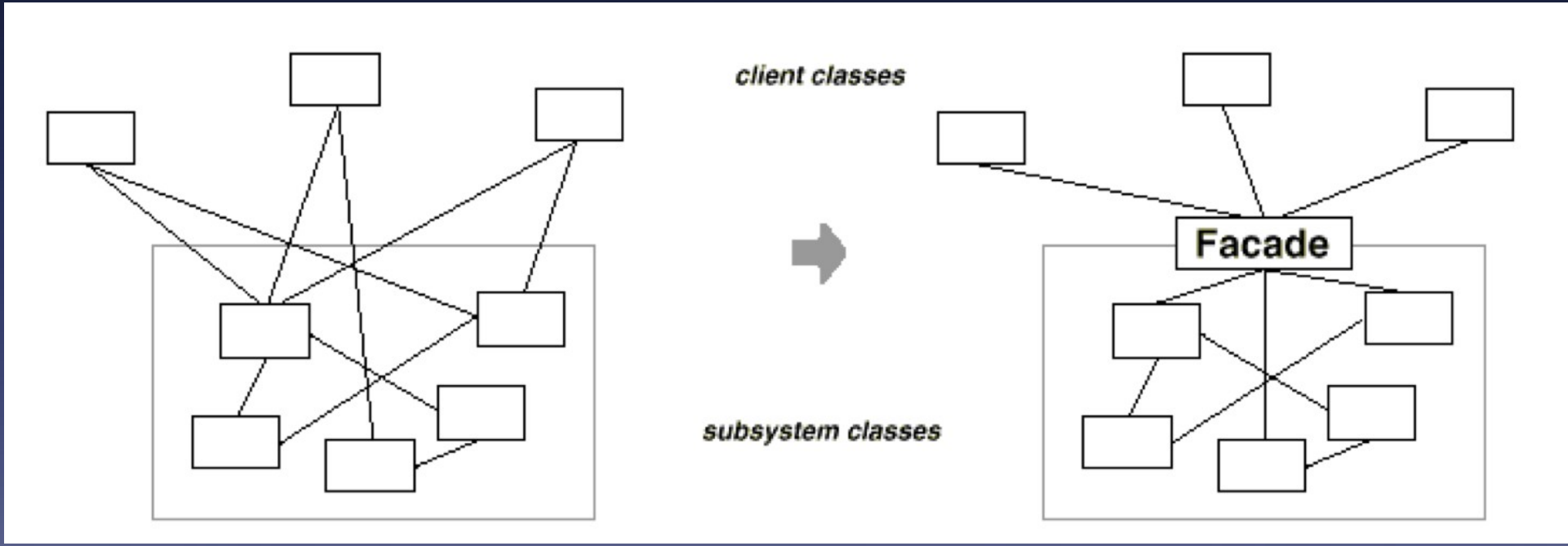
2. Avoids feature-laden classes high up in the hierarchy.

3. The code can be hard to learn and debug.

Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use [3].

Facade



Facade

1. It shields clients from subsystem components.

2. It promotes weak coupling between the subsystem and its clients.

3. It doesn't prevent applications from using subsystem classes.

Facade

1. It shields clients from subsystem components.

2. It promotes weak coupling between the subsystem and its clients.

3. It doesn't prevent applications from using subsystem classes.

Proxy

Provide a surrogate or placeholder for another object to control access to it [3].

Proxy

1. A remote proxy can hide the fact that an object resides in a different address space.

2. A proxy can perform optimizations such as creating an object on demand.

3. Allows to add additional housekeeping tasks when an object is accessed.

Other popular Design Patterns

Behavioral patterns:

1. Chain of responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Null object
8. State
9. Strategy
10. Template method
11. Visitor

Chain of responsibility

TODO

Command

TODO

Interpreter

TODO

Iterator

TODO

Mediator

TODO

Memento

TODO

Null object

TODO

State

TODO

Strategy

TODO

Template method

TODO

Visitor

TODO

Case Study

TODO

Use or not?

TODO

Further resources

[1] http://en.wikipedia.org/wiki/Software_design_pattern

[2] http://en.wikipedia.org/wiki/List_of_software_development_philosophies

[3] Gamma, Helm, Johnson & Vlissides (1994). Design Patterns (the Gang of Four book). Addison-Wesley. ISBN 0-201-63361-2.

[4] Alexander, Christopher (1977). A Pattern Language: Towns, Buildings, Construction. Oxford University Press. ISBN 0-19-501919-9.

[5] https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/DesignPattern

[6] https://developer.apple.com/library/mac/documentation/general/conceptual/devpedia-cocoacore/M

[7] https://developer.apple.com/library/ios/documentation/general/conceptual/CocoaEncyclopedia/Mo

[8] Carlo Chung (2011). Pro Objective-C Design Patterns for iOS. Apress. ISBN 978-1-4302-3330-5.

[9] http://en.wikipedia.org/wiki/Lazy_initialization