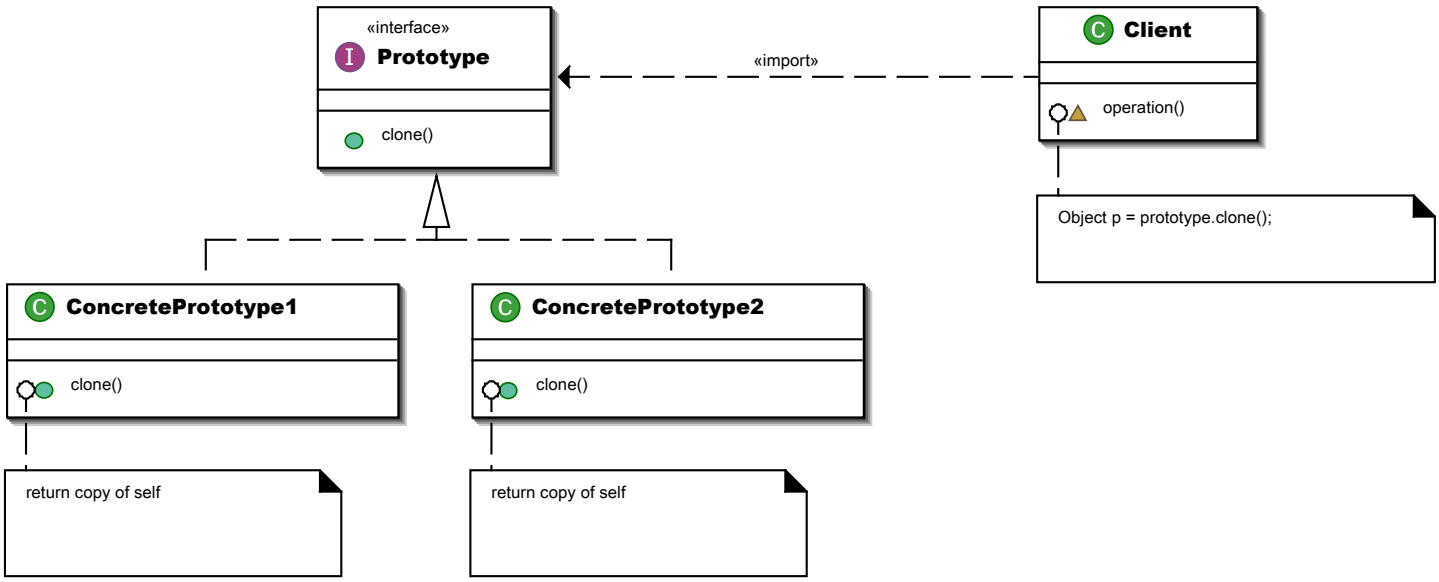
设计模式

创建范例：关于如何创建实例
结构范例：关于类及对象复合关系
行为范例：关于对象之间如何通讯

创建范例	结构范例	行为范例
抽象工厂（ Abstract Factory ） 生成器（ Builder ） 工厂方法（ Factory Method ） 原型（ Prototype ） 单例（ Singleton ）	适配器（ Adapter ） 桥接（ Bridge ） 组合（ Composite ） 修饰（ Decorator ） 外观（ Facade ） 享元（ Flyweight ） 代理（ Proxy ）	责任链（ Chain of Responsibility ） 命令（ Command ） 解释器（ Interpreter ） 迭代器（ Iterator ） 中介者（ Mediator ） 备忘录（ Memento ） 观察者（ Observer ） 状态机（ State ） 策略（ Strategy ） 模版方法（ Template Method ） 访问者（ Visitor ）

原型模式

原型模式：通过“复制”一个已经存在的实例来返回新的实例，而不是新建实例。




```objc
@interface Worker : NSObject <NSCopying>

@property (nonatomic) NSString *name;
@property (nonatomic) NSInteger age;

@end
```

```objc
@implementation Worker

- (id)copyWithZone:(NSZone *)zone {
    Worker *a = [[[self class] allocWithZone:zone] init];
    a.name = [self.name copyWithZone:zone];
    a.age = self.age;
    return a;
}

@end
```

```objc
    Worker *a = [Worker new];
    a.name = @"Tom";
    a.age = 12;
    Worker *b = [a copy];
```

在 iOS 中可简单认为 copy 就是原型模式。

单例模式

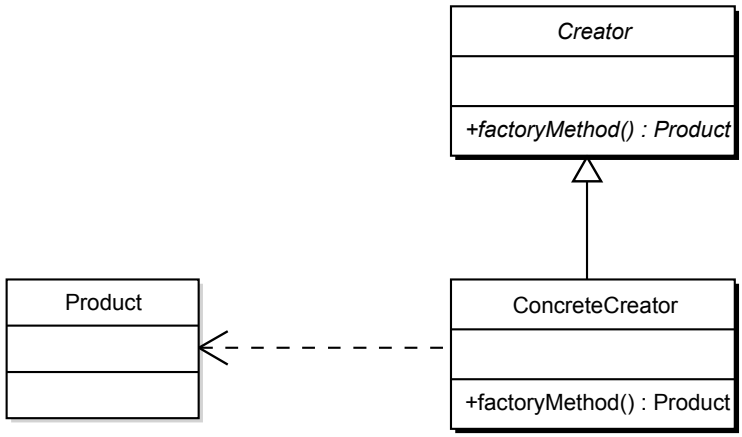
单例模式：单例对象的类必须保证只有一个实例存在。

```
@interface Worker : NSObject <NSCopying, NSMutableCopying>

+ (instancetype)shareInstance;

@end
```

```
@implementation Worker
static Worker *_instance = nil;

+ (instancetype)shareInstance {
   return [[self alloc] init];
}

+ (instancetype)allocWithZone:(struct _NSZone *)zone {
   @synchronized (self) {
      if (_instance == nil) {
         _instance = [super allocWithZone:zone];
      }
      return _instance;
   }
}

- (id)copyWithZone:(NSZone *)zone {
   return _instance;
}

- (id)mutableCopyWithZone:(NSZone *)zone {
   return _instance;
}

@end
```

工厂方法模式

工厂方法模式：定义一个创建对象的接口，让实现这个接口的类来决定实例化哪个类。




```objc
// Worker本身也是使用工厂方法模式
@interface Worker : NSObject

// 定义创建对象的接口。（也可以定义在@protocol中）
+ (instancetype)createInstance;

@end
```

```objc
@implementation Worker

+ (instancetype)createInstance {
    return [Worker new];
}

@end
```

```objc
@interface Doctor : Worker

@end

@implementation Doctor

+ (instancetype)createInstance {
    return [Doctor new];
}

@end
```

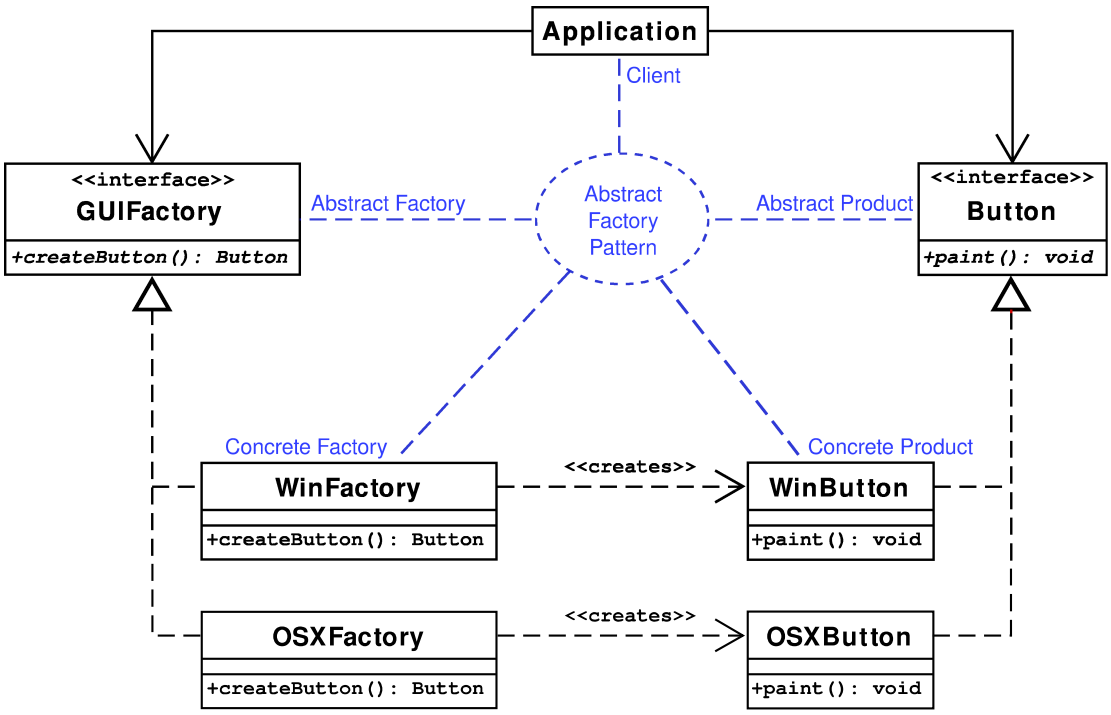
```objc
@interface Nurse : Worker
@end

@implementation Nurse

+ (instancetype)createInstance {
    return [Nurse new];
}

@end
```

```objc
Worker *w = [Worker createInstance];
Doctor *d = [Doctor createInstance];
Nurse *n = [Nurse createInstance];
```

工厂方法模式需要知道具体的子类

抽象工厂模式

抽象工厂模式：提供接口，创建一系列相关或独立的对象，而不指定这些对象的具体类。




```objc
// 定义抽象接口
@interface Worker : NSObject

+ (instancetype)createDoctor;

+ (instancetype)createNurse;

- (void)work;

@end
```

```objc
@implementation Worker

+ (instancetype)createDoctor {
    return [Doctor new];
}

+ (instancetype)createNurse {
    return [Nurse new];
}

@end
```

```objc
@interface Doctor : Worker
@end

@implementation Doctor

- (void)work {
    printf(">>> Doctor work\n");
}

@end
```

```objc
@interface Nurse : Worker
@end

@implementation Nurse

- (void)work {
    printf(">>> Nurse work\n");
}

@end
```

```objc
    Worker *w = [Worker createNurse];
    [w work];
```

抽象工厂模式不需要知道具体的子类

生成器模式

生成器模式：将复杂对象的建造过程抽象出来（抽象类别），使这个抽象过程的不同实现方法可以构造出不同表现（属性）的对象。

```
@interface Worker : NSObject

@property (readonly, nonatomic) NSString *name;
@property (readonly, nonatomic) NSInteger age;

- (WorkerBuilder *)builder;

@end
```

```
@implementation Worker

- (WorkerBuilder *)builder {
    return [[WorkerBuilder alloc] initWithWorker:self];
}

@end
```

```
@interface WorkerBuilder : NSObject

- (instancetype)setAge:(NSInteger)age;

- (instancetype)setName:(NSString *)name;

- (instancetype)initWithWorker:(Worker *)worker;

- (Worker *)build;

@end
```

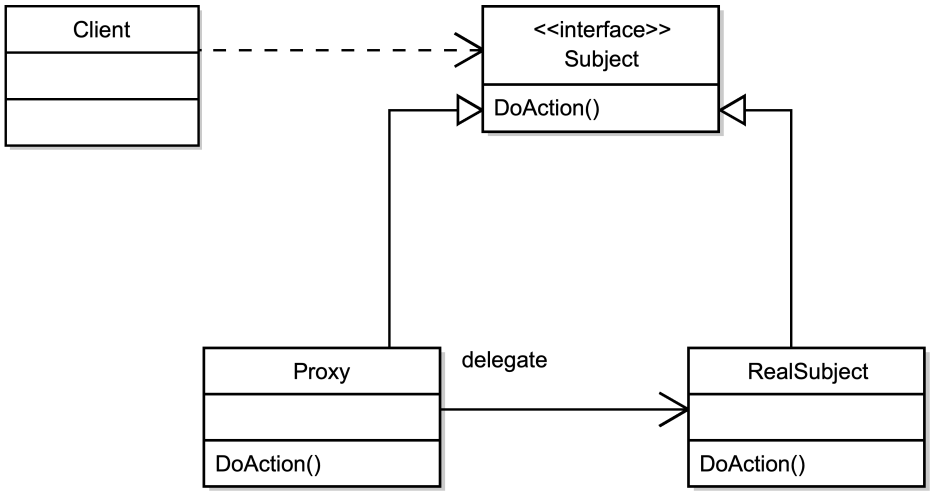
```
@implementation WorkerBuilder

- (instancetype)initWithWorker:(Worker *)worker {
    self = [super init];
    if (self) {
        _worker = worker;
    }
    return self;
}

- (Worker *)build {
    return _worker;
}

- (instancetype)setAge:(NSInteger)age {
    [_worker setValue:@(age) forKey:@"age"];
    return self;
}

- (instancetype)setName:(NSString *)name {
    [_worker setValue:name forKey:@"name"];
    return self;
}

@end
```

```
Worker *a = [Worker new];
  [[[a.builder
    setAge:12]
   setName:@"Hanks"] build];
```

生成器模式的构造与表现分离

代理模式

代理模式：一个类别可以作为其他东西的接口。




```objc
@interface Worker : NSObject

@property (weak) id<WorkerDelegate>
delegate;

@property (nonatomic) NSString *name;
@property (nonatomic) NSInteger age;

@end

@protocol WorkerDelegate <NSObject>

- (void)worker:(Worker *)worker
didChangeAge:(NSInteger)age;

@end


@implementation Worker

- (void)setAge:(NSInteger)age {
    _age = age;
    [self.delegate worker:self
didChangeAge:age];
}

@end
```

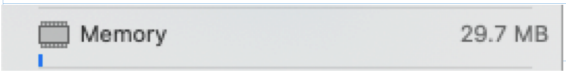
```objc
@interface ViewController ()
<WorkerDelegate> {
    Worker *_worker;
}

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    _worker = [Worker new];
    _worker.delegate = self;
    _worker.age = 13;
}

- (void)worker:(Worker *)worker
didChangeAge:(NSInteger)age {
    puts(__func__);
}

@end
```

享元模式

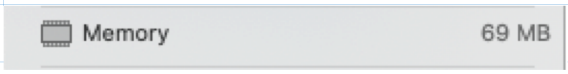
享元模式：使用物件尽可能减少内存使用量，于相似物件中分享尽可能多的资讯。

```objc
@interface Worker : NSObject

@property (nonatomic) NSInteger ID;
@property (nonatomic) NSString *name;
@property (nonatomic) NSInteger age;

@end

@implementation Worker
@end
```

```objc
Worker *a = [Worker new];
a.ID = 123;
NSMutableArray *array = [NSMutableArray new];
for (int i = 0; i < 1e6; i++) {
    // 同一个对象，享元模式
    [array addObject:a];
}
```



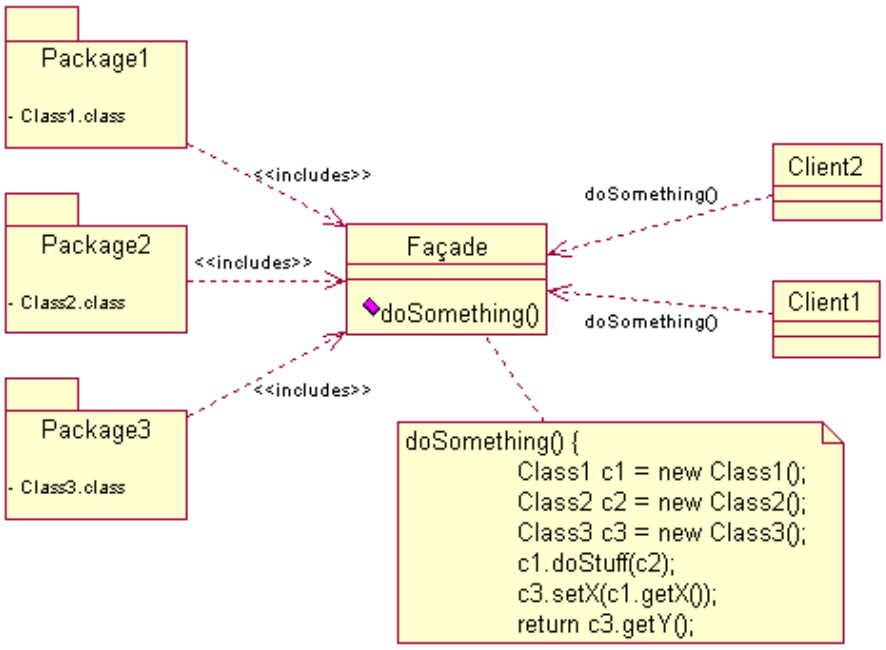

```objc
NSMutableArray *array = [NSMutableArray new];
for (int i = 0; i < 1e6; i++) {
    // 不同对象
    Worker *a = [Worker new];
    a.ID = 123;
    [array addObject:a];
}
```



UITableViewCell 的重用也是享元模式

外观模式

外观模式：为子系统中的一组接口提供一个统一的高层接口，使得子系统更容易使用。




```objc
@interface Worker : NSObject

@property (nonatomic) NSInteger ID;
@property (nonatomic) NSString *name;
@property (nonatomic) NSInteger age;

- (void)work;

@end

@implementation Worker

- (void)work {
    printf(">>> worker-%s work\n", [self.name
UTF8String]);
}

@end
```

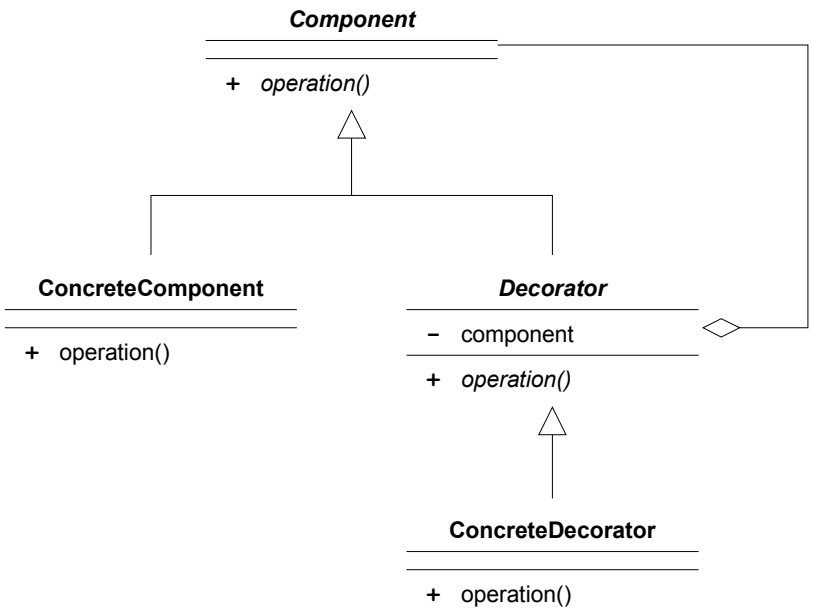
```objc
@interface Boss : NSObject

@property (nonatomic) NSString *name;
@property (nonatomic) NSInteger age;

- (void)manage;

@end

@implementation Boss

- (void)manage {
    printf(">>> boss-%s manage\n", [self.name
UTF8String]);
}

@end
```

```objc
@interface Company : NSObject

@property (nonatomic) Boss *boss;
@property (nonatomic) NSArray<Worker *>
*workers;

//  外观模式
- (void)produce;

@end

@implementation Company

- (void)produce {
    [self.boss manage];
    for (Worker *worker in self.workers) {
        [worker work];
    }
}

@end
```

```objc
    Boss *b = [Boss new];
    b.name = @"老板";

    Worker *w1 = [Worker new];
    w1.name = @"员工A";

    Worker *w2 = [Worker new];
    w2.name = @"员工B";

    Worker *w3 = [Worker new];
    w3.name = @"员工C";

    Company *c = [Company new];
    c.boss = b;
    c.workers = @[w1, w2, w3];

    [c produce];
```

修饰模式

修饰模式：动态地往一个类中添加新的行为。修饰模式相比生成子类更为灵活，这样可以给某个对象而不是整个类添加一些功能。




```
@interface Worker : NSObject

@property (nonatomic) NSInteger ID;
@property (nonatomic) NSString *name;
@property (nonatomic) NSInteger age;

@end

@implementation Worker

@end
```

```
@protocol Work <NSObject>

- (void)work;

@end
```

```
@interface WorkerDecorator : NSObject

+ (void)decorate:(Worker *)worker;

@end
```

```
    Worker *w = [Worker new];
//
[WorkerDecorator decorate:w];
//
[(Worker<Work> *)w work];
```

```
@implementation WorkerDecorator

+ (void)decorate:(Worker *)worker {
    Class cls2 = [WorkerDecorator TempWorker];
    //
    object_setClass(worker, cls2);
    //
    [WorkerDecorator addWork:cls2];
}

//  生成TempWorker
+ (Class)TempWorker {
    Class cls = [Worker class];
    Class cls2 = objc_allocateClassPair(cls,
"TempWorker", 0);
    objc_registerClassPair(cls2);
    return cls2;
}

//  添加work方法
+ (void)addWork:(Class)cls {
    //  "v16@0:8"
    SEL sel = @selector(work);
    IMP imp =
imp_implementationWithBlock(^void(id sender) {
        puts(__func__);
    });
    class_addMethod(cls, sel, imp, "v16@0:8");
}

@end
```

组合模式

组合模式：将对象组合成树形结构以表示“部分 - 整体”的层次结构。组合使得用户对单个对象和组合对象的使用具有一致性。

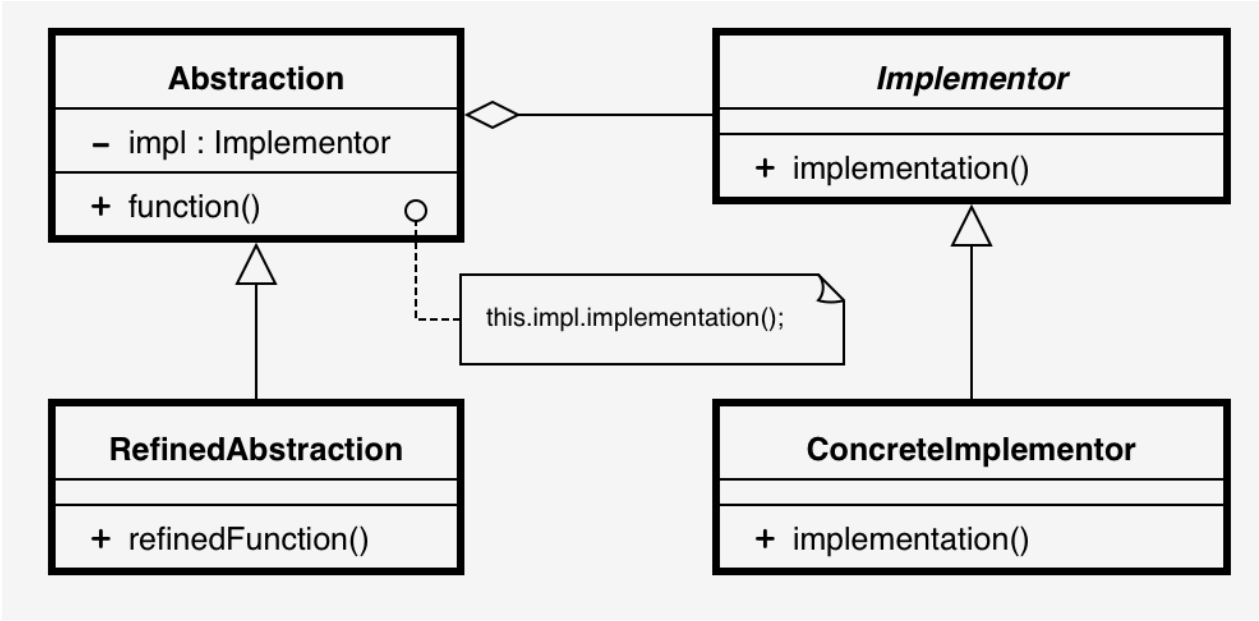
```objc
@interface Worker : NSObject

@property Worker *leftHand;
@property Worker *rightHand;
@property Worker *mouth;

@end

@implementation Worker

@end
```

```objc
Worker *leftHand = [Worker new];
Worker *rightHand = [Worker new];
Worker *mouth = [Worker new];

Worker *boss = [Worker new];
boss.leftHand = leftHand;
boss.rightHand = rightHand;
boss.mouth = mouth;
```

UIView.addSubView 也是组合模式

桥接模式

桥接模式：把事物对象的具体行为、具体特征分离开来，使它们可以各自独立的变化。




```objc
@protocol Work <NSObject>

- (void)work:(NSString *)name age:(NSInteger)age;

@end
```

```objc
@interface DesignWork : NSObject <Work>

@end

@implementation DesignWork

- (void)work:(NSString *)name age:(NSInteger)age {
    printf(">>> %s-%ld, 开始设计\n", name.UTF8String, age);
}

@end
```

```objc
@interface CodingWork : NSObject <Work>

@end

@implementation CodingWork

- (void)work:(NSString *)name age:(NSInteger)age {
    printf(">>> %s-%ld, 开始编码\n", name.UTF8String, age);
}

@end
```

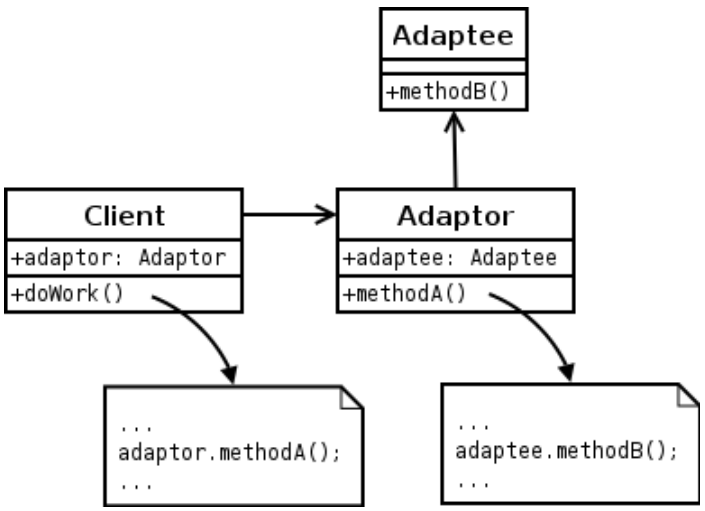
```objc
@interface Worker : NSObject

@property (nonatomic) NSString *name;
@property (nonatomic) NSInteger age;

- (void)work:(id<Work>)work;

@end
```

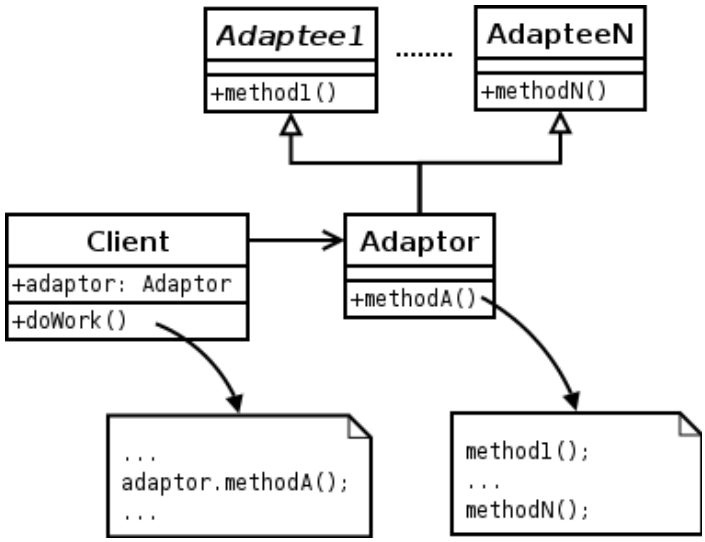
```objc
@implementation Worker

- (void)work:(id<Work>)work {
    [work work:self.name age:self.age];
}

@end
```

```objc
    DesignWork *d = [DesignWork new];
    CodingWork *c = [CodingWork new];

    Worker *w = [Worker new];
    w.name = @"Tom";
    w.age = 33;

    [w work:d];
    [w work:c];
```

适配器模式

适配器模式：将一个类的接口转换成用户所期待的。适配器使得因接口不兼容而不能一起工作的类能在一起工作。






```objc
@interface Worker : NSObject

@property (nonatomic) NSString *name;
@property (nonatomic) NSInteger age;

- (void)work;

@end
```

```objc
@implementation Worker

- (void)work {
    printf(">>> 开始工作\n");
}

@end
```

```objc
// 对象适配器模式
@interface ObjectAdapter : NSObject

@property (nonatomic) Worker *worker;

- (void)run;

@end

@implementation ObjectAdapter

- (void)run {
    [self.worker work];
}

@end
```

```objc
// 类适配器模式
@interface ClassAdapter : Worker

- (void)run;

@end

@implementation ClassAdapter

- (void)run {
    [self work];
}

@end
```

```objc
// 对象适配器模式
Worker *w = [Worker new];
ObjectAdapter *a = [ObjectAdapter new];
a.worker = w;
[a run];
```

```objc
// 类适配器模式
ClassAdapter *c = [ClassAdapter new];
[c run];
```

责任链模式

责任链模式：包含了一些命令对象和一系列的处理对象，每一个处理对象决定它能处理哪些命令对象，它也知道如何将它不能处理的命令对象传递给该链中的下一个处理对象。该模式还描述了往该处理链的末尾添加新的处理对象的方法。

```objc
typedef NS_ENUM(NSInteger, WorkType) {
    WorkTypeDesign,
    WorkTypeCoding,
    WorkTypeTest,
};

@interface Worker : NSObject

@property WorkType type;
@property Worker *next;

// Worker是处理对象，message、type是命令对象
- (void)work:(NSString *)message type:
(WorkType)type;

@end
```

```objc
@implementation Worker

- (void)work:(NSString *)message type:
(WorkType)type {
    if (type == self.type) {
        // 处理
        NSLog(@">>> %@ - %@ - %ld", [self class],
message, type);
    } else {
        // 继续传递
        [_next work:message type:type];
    }
}

@end
```

```objc
@interface DesignWorker : Worker
@end
```

```objc
@implementation DesignWorker
@end
```

```objc
@interface CodingWorker : Worker
@end
```

```objc
@implementation CodingWorker
@end
```

```objc
@interface TestWorker : Worker
@end
```

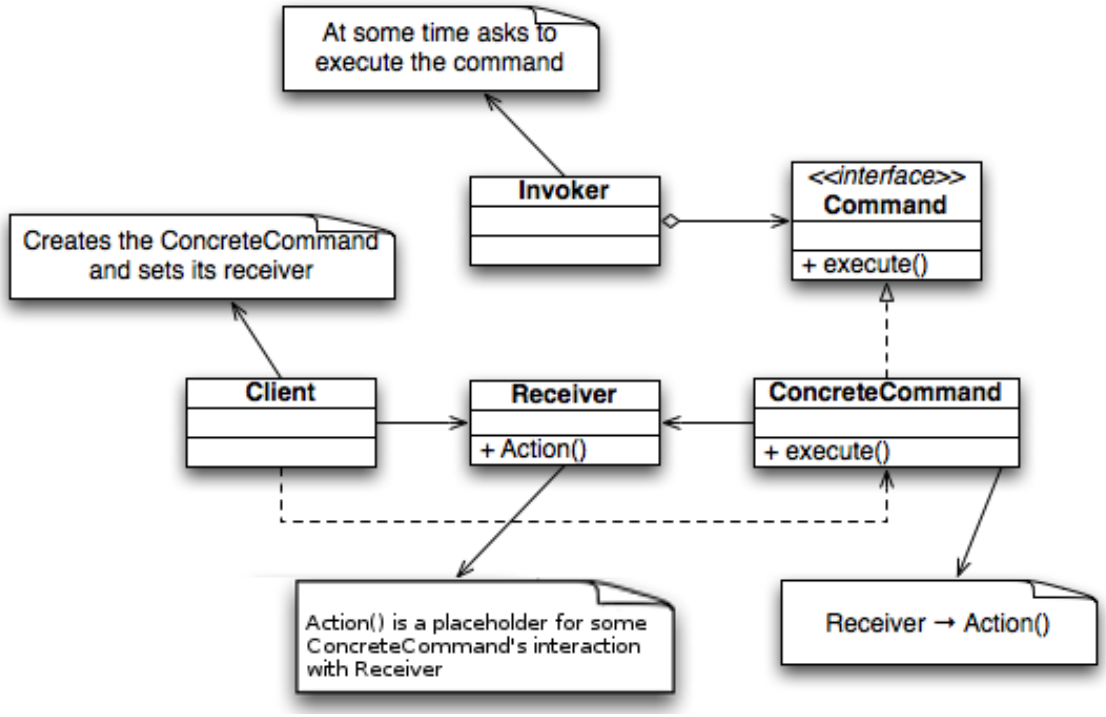
```objc
@implementation TestWorker
@end
```

```objc
DesignWorker *d = [DesignWorker new];
d.type = WorkTypeDesign;
CodingWorker *c = [CodingWorker new];
c.type = WorkTypeCoding;
TestWorker *t = [TestWorker new];
t.type = WorkTypeTest;

[d setNext:c];
[c setNext:t];

[d work:@"Hello World" type:WorkTypeCoding];
```

通常在责任链模式中，如果某个处理对象处理了这个命令对象，那么这个命令对象就不会传递下去

命令模式

命令模式：以对象代表实际行动，命令对象把行动及其参数封装起来。这些行动可以被重复多次、取消、取消后重做。




```objc
@interface Worker : NSObject

// 参数
@property (nonatomic) NSString *name;

// 行动
- (void)work;

@end
```

```objc
@implementation Worker

- (void)work {
    printf(">>> worker-%s work\n", [self.name UTF8String]);
}

@end
```

```objc
Worker *w = [Worker new];
w.name = @"Tom";
[w work];
```

迭代器模式

迭代器模式：透过特定的接口巡访容器中的每一个元素而不用了解底层的实现。

```objc
@protocol Iterator <NSObject>

- (id)first;
- (id)next;
- (BOOL)isDone;
- (id)current;

@end
```

```objc
@interface Worker : NSObject

@property (nonatomic) NSString *name;

@end


@implementation Worker
@end
```

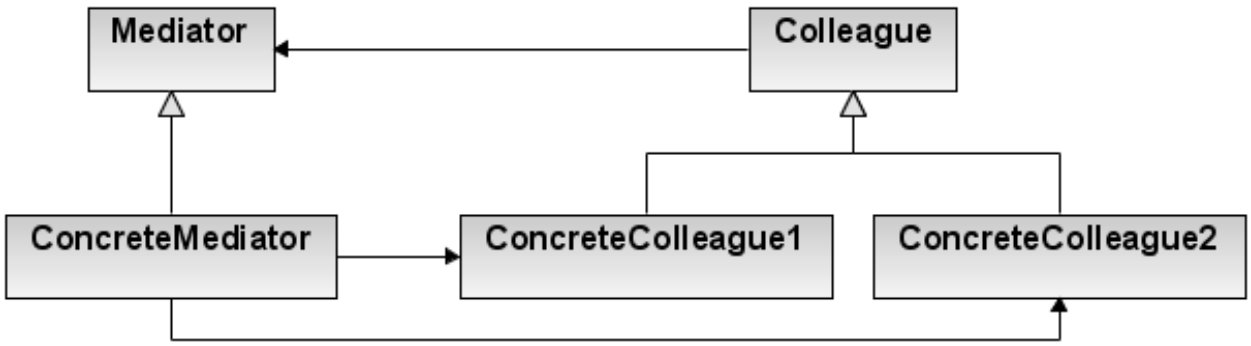
```objc
@interface Company : NSObject

- (void)addWorker:(Worker *)worker;

- (id<Iterator>)iterate;

@end
```

```objc
@interface Company () <Iterator> {
    NSMutableArray<Worker *> *_workers;
    NSInteger _cur;
}

@end

@implementation Company

- (instancetype)init {
    self = [super init];
    if (self) {
        _workers = [NSMutableArray new];
        _cur = 0;
    }
    return self;
}

- (void)addWorker:(Worker *)worker {
    [_workers addObject:worker];
}

- (id<Iterator>)iterate {
    return self;
}

//MARK:-    Iterator

- (id)current {
    return _workers[_cur];
}

- (id)first {
    _cur = 0;
    return _workers.firstObject;
}

- (BOOL)isDone {
    if (_cur >= _workers.count) {
        return YES;
    }
    return NO;
}

- (id)next {
    if (++_cur < _workers.count) {
        return _workers[_cur];
    }
    return nil;
}

@end
```

```objc
    Worker *w1 = [Worker new];
    w1.name = @"AAA";

    Worker *w2 = [Worker new];
    w2.name = @"BBB";

    Worker *w3 = [Worker new];
    w3.name = @"CCC";

    Company *c = [Company new];
    [c addWorker:w1];
    [c addWorker:w2];
    [c addWorker:w3];

    id<Iterator> iterator = [c iterate];
    [iterator first];
    while (![iterator isDone]) {
        Worker *w = [iterator current];
        NSLog(@">>> %@", w.name);
        [iterator next];
    }
```

中介者模式

中介者模式：用一个对象封装对象间的交互方式。这种模式避免了显示调用其他类，促进类间的松耦合，并使得类间交互关系本身可以单独修改。




```objc
@interface CodingWorker : NSObject

@property (weak) Mediator *mediator;
@property NSString *name;

- (void)code:(NSInteger)num;

@end
```

```objc
@implementation CodingWorker

- (void)code:(NSInteger)num {
    printf(">>> %s 编码完成，提交测试\n", [self.name UTF8String]);
    [self.mediator test:num];
}

@end
```

```objc
@interface TestWorker : NSObject

@property (weak) Mediator *mediator;
@property NSString *name;

- (void)test:(NSInteger)num;

@end
```

```objc
@implementation TestWorker

- (void)test:(NSInteger)num {
    printf("--- %s 测试完成\n", [self.name UTF8String]);
    if (num != 0) {
        printf("--- 发现bug，反馈给编码人员\n");
        [self.mediator code:num - 1];
    } else {
        printf("--- 没有bug\n");
    }
}

@end
```

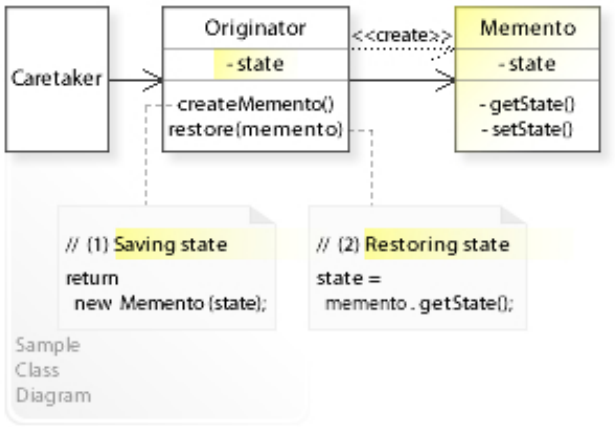
```objc
@interface Mediator : NSObject

- (void)registerCoder:(CodingWorker *)coder;
- (void)registerTester:(TestWorker *)tester;

- (void)code:(NSInteger)num;
- (void)test:(NSInteger)num;

@end
```

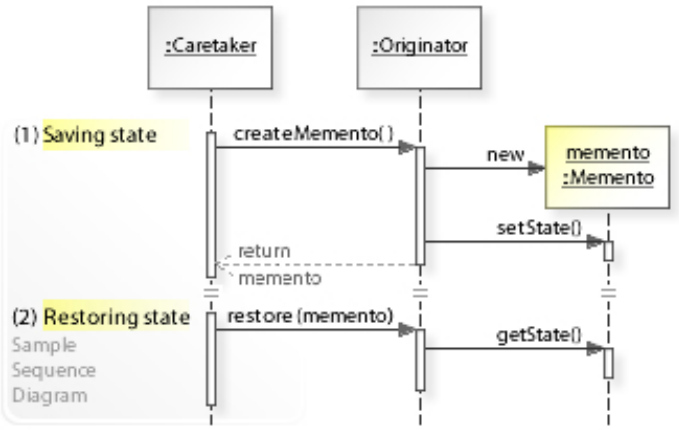
```objc
@interface Mediator () {
    CodingWorker *_coder;
    TestWorker *_tester;
}

@end

@implementation Mediator

- (void)registerCoder:(CodingWorker *)coder {
    _coder = coder;
    _coder.mediator = self;
}

- (void)registerTester:(TestWorker *)tester {
    _tester = tester;
    _tester.mediator = self;
}

- (void)code:(NSInteger)num {
    [_coder code:num];
}

- (void)test:(NSInteger)num {
    [_tester test:num];
}

@end
```

```objc
CodingWorker *c = [CodingWorker new];
c.name = @"张三";

TestWorker *t = [TestWorker new];
t.name = @"赵四";

Mediator *m = [Mediator new];
[m registerCoder:c];
[m registerTester:t];

[c code:1];
```

备忘录模式

备忘录模式：为一个对象提供恢复到它之前的状态的能力。






```objc
//  Originator
@interface Worker : NSObject

@property NSString *name;
@property NSInteger age;

- (void)save;
- (void)restore;

@end
```

```objc
//  Memento通常是磁盘文件、内存
@interface Memento : NSObject

- (instancetype)initWithState:(Worker *)worker;
- (Worker *)state;

@end
```

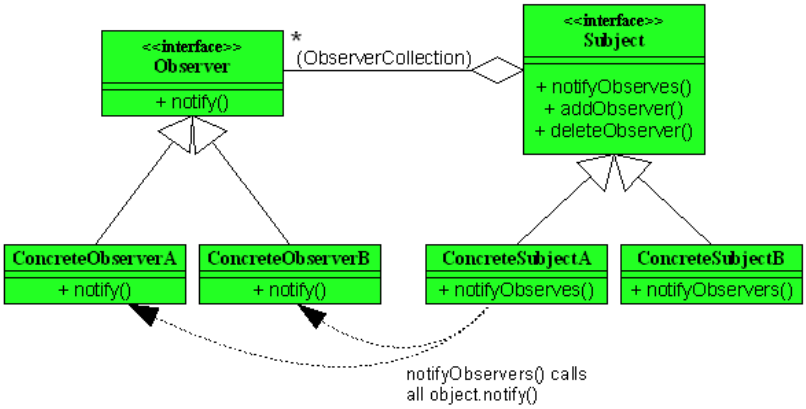
```objc
@interface Worker () {
    Memento *_memento;
}

@end

@implementation Worker

- (void)save {
    Worker *worker = [Worker new];
    worker.name = self.name;
    worker.age = self.age;
    _memento = [[Memento alloc]
initWithState:worker];
}

- (void)restore {
    Worker *worker = [_memento state];
    self.name = worker.name;
    self.age = worker.age;
}

@end
```

```objc
@interface Memento () {
    Worker *_worker;
}
@end

@implementation Memento

- (instancetype)initWithState:(Worker *)worker
{
    self = [super init];
    if (self) {
        _worker = worker;
    }
    return self;
}

- (Worker *)state {
    return _worker;
}

@end
```

```objc
// Caretaker
Worker *worker = [Worker new];
worker.name = @"Tom";
worker.age = 11;
[worker save];
worker.name = @"Hanks";
worker.age = 22;
[worker restore];
```

观察者模式

观察者模式：一个目标对象管理所有依赖于它的观察者对象，并且在它本身的状态改变时主动发出通知。




```objc
@interface NotifyCenter : NSObject

- (void)addObserver:(id)observer selector:(SEL)aSelector object:(nullable id)object;
- (void)removeObserver:(id)observer;

- (void)post:(nullable id)object;

@end
```

```objc
@interface NotifyCenter () {
    NSMutableArray<NotifyModel *> *_lists;
}
@end
```

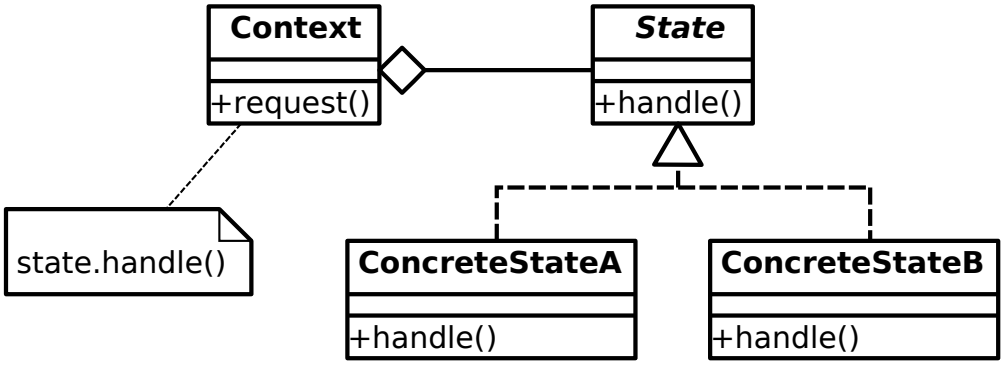
```objc
@interface NotifyModel: NSObject
@property id observer;
@property SEL aSelector;
@property id object;
@end

@implementation NotifyModel
@end
```

```objc
- (void)viewDidLoad {
    [super viewDidLoad];

    _center = [NotifyCenter new];
    // 监听
    [_center addObserver:self selector:@selector(observe:) object:nil];

    // 状态改变代码...

    // 发出
    [_center post:nil];
    // 移除
    [_center removeObserver:self];
}

- (void)observe:(nullable id)object {
    puts(__func__);
}
```

```objc
@implementation NotifyCenter

- (instancetype)init {
    self = [super init];
    if (self) {
        _lists = [NSMutableArray new];
    }
    return self;
}

- (void)addObserver:(id)observer selector:(SEL)aSelector object:(nullable id)object {
    NotifyModel *model = [NotifyModel new];
    model.observer = observer;
    model.aSelector = aSelector;
    model.object = object;
    [_lists addObject:model];
}

- (void)removeObserver:(id)observer {
    int i = 0;
    while (i < _lists.count) {
        NotifyModel *model = _lists[i];
        if (model.observer == observer) {
            [_lists removeObject:model];
        } else {
            i++;
        }
    }
}

- (void)post:(nullable id)object {
    for (NotifyModel *model in _lists) {
        ((void (*)(id, SEL, id))objc_msgSend)(model.observer, model.aSelector, model.object);
    }
}

@end
```

状态机模式

状态机模式：当对象的内部状态改变时允许改变它自身的行为。




```
@protocol State <NSObject>

@property (readonly) BOOL loginingEnabled;
@property (readonly) BOOL loginSuccessEnabled;
@property (readonly) BOOL loginFailedEnabled;

- (void)login:(Context *)ctx;
- (void)loginSuccess:(Context *)ctx;
- (void)loginFailed:(Context *)ctx;

@end
```

```
@interface LoginingState : NSObject <State>
@end
```

```
@interface LoginSuccessState : NSObject <State>
@end
```

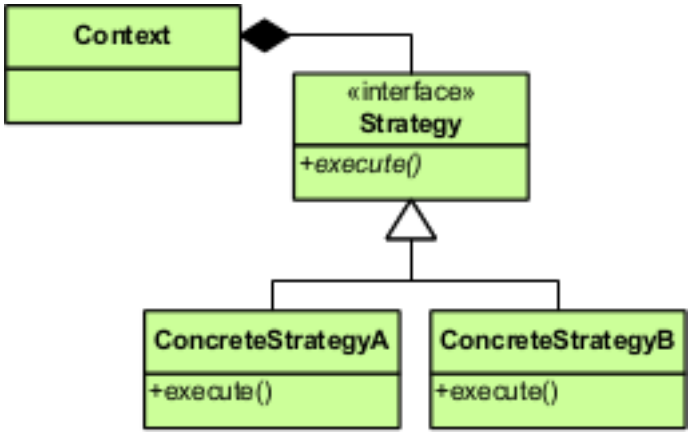
```
@interface LoginFailedState : NSObject <State>
@end
```

```
@interface Context : NSObject

@property (class, readonly) LoginingState *loginingState;
@property (class, readonly) LoginSuccessState
*loginSuccessState;
@property (class, readonly) LoginFailedState
*loginFailedState;

@property id<State> state;

- (void)login;
- (void)loginSuccess;
- (void)loginFailed;

@end
```

```
Context *ctx = [Context new];
ctx.state = Context.loginingState;
[ctx login];
[ctx loginSuccess];
```

策略模式

策略模式：指对象的某个行为，在不同的场景中，该行为有不同的实现算法。




```
@protocol Strategy <NSObject>

- (void)execute;

@end
```

```
@interface StrategyA : NSObject <Strategy>

@end

@implementation StrategyA

- (void)execute {
    puts(">>> StrategyA");
}

@end
```

```
@interface StrategyB : NSObject <Strategy>

@end

@implementation StrategyB

- (void)execute {
    puts(">>> StrategyB");
}

@end
```

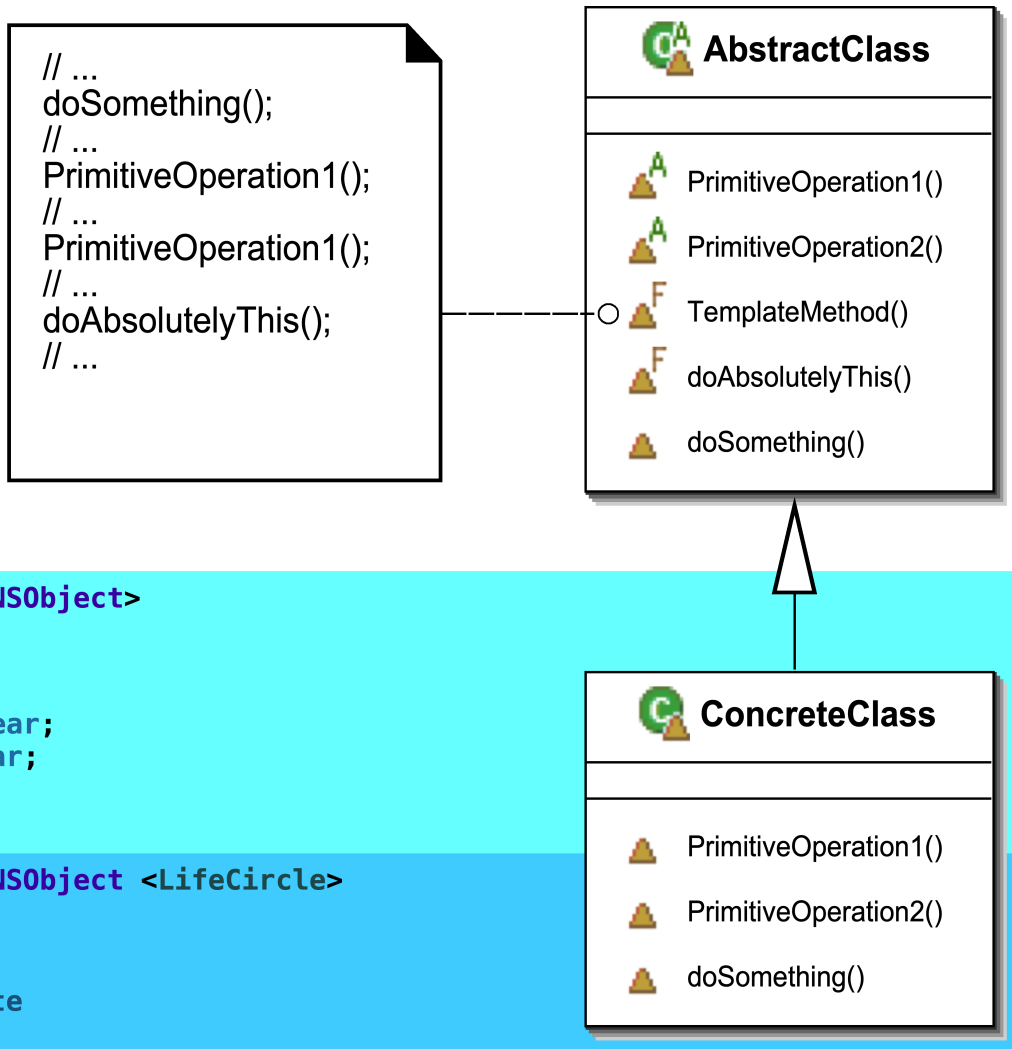
```
@interface Context : NSObject

@property id<Strategy> strategy;

- (void)doSomething;

@end
```

```
@implementation Context

- (void)doSomething {
    [self.strategy execute];
}

@end
```

```
    StrategyA *a = [StrategyA new];
    StrategyB *b = [StrategyB new];

    Context *ctx = [Context new];
    ctx.strategy = a;
    [ctx doSomething];
    ctx.strategy = b;
    [ctx doSomething];
```

模版方法模式

模版方法模式：定义一个操作中算法的骨架，而将一些步骤延迟到子类中，模版方法使子类可以重定义算法的某些特定步骤而不改变算法的结构。



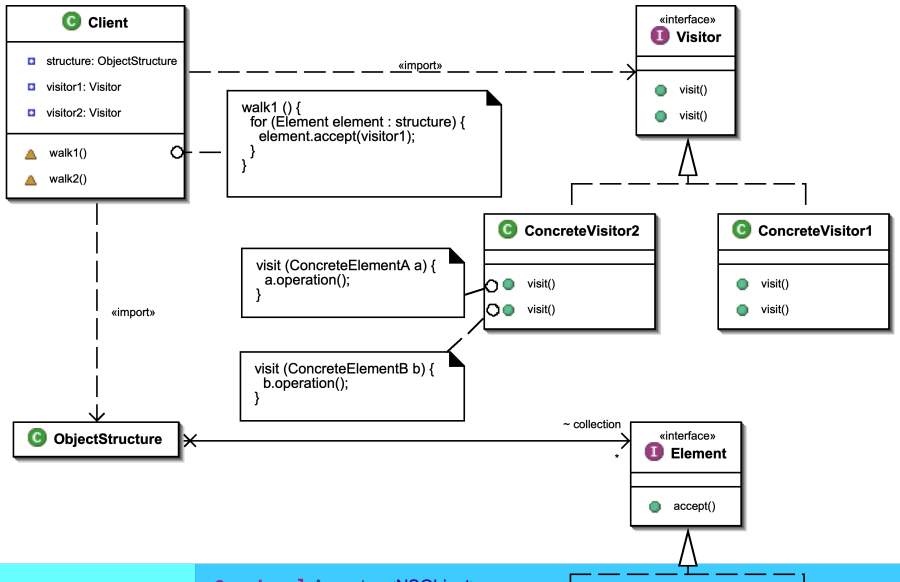

```objc
@protocol LifeCircle <NSObject>

- (void)viewWillAppear;
- (void)viewDidAppear;
- (void)viewWillDisappear;
- (void)viewDidDisappear;

@end
```

```objc
@interface Concrete : NSObject <LifeCircle>

@end

@implementation Concrete

- (void)viewDidAppear {
    puts(">>> viewDidAppear");
}

- (void)viewDidDisappear {
    puts(">>> viewDidDisappear");
}

- (void)viewWillAppear {
    puts(">>> viewWillAppear");
}

- (void)viewWillDisappear {
    puts(">>> viewWillDisappear");
}

@end
```

```objc
Concrete *c = [Concrete new];
[c viewWillAppear];
```

访问者模式

访问者模式：将算法与对象结构分离。




```objc
@protocol Visitor <NSObject>

- (void)visitEngine:(Engine *)engine;
- (void)visitWheel:(Wheel *)wheel;
- (void)visitCar:(Car *)car;

@end
```

```objc
@protocol Acceptor <NSObject>

- (void)accept:(id<Visitor>)visitor;

@end
```

accept (Visitor v) {
v.visit(this);
}

ConcreteElementA
accept()
operation()

ConcreteElementB
accept()
operation()

```objc
@interface Wheel : NSObject <Acceptor>

@property NSString *name;

@end

@implementation Wheel

- (void)accept:(nonnull id<Visitor>)visitor {
    [visitor visitWheel:self];
}

@end
```

```objc
@interface Engine : NSObject <Acceptor>

@end

@implementation Engine

- (void)accept:(nonnull id<Visitor>)visitor {
    [visitor visitEngine:self];
}

@end
```

```objc
@interface Car : NSObject <Acceptor>

@end

@interface Car () {
    Engine *_engine;
    NSMutableArray<Wheel *> *_wheels;
}

@end

@implementation Car

- (instancetype)init
{
    self = [super init];
    if (self) {
        _engine = [Engine new];
        _wheels = [NSMutableArray new];
        NSArray *names = @[@"左前", @"右前", @"左后", @"左后"];
        for (int i = 0; i < names.count; i++) {
            Wheel *wheel = [Wheel new];
            wheel.name = names[i];
            [_wheels addObject:wheel];
        }
    }
    return self;
}

- (void)accept:(nonnull id<Visitor>)visitor {
    [visitor visitCar:self];
    [_engine accept:visitor];
    for (int i = 0; i < _wheels.count; i++) {
        Wheel *wheel = _wheels[i];
        [wheel accept:visitor];
    }
}

@end
```

```objc
@interface PrintVisitor : NSObject <Visitor>

@end

@implementation PrintVisitor

- (void)visitCar:(nonnull Car *)car {
    puts(">>> 访问Car");
}

- (void)visitEngine:(nonnull Engine *)engine {
    puts(">>> 访问发动机");
}

- (void)visitWheel:(nonnull Wheel *)wheel {
    printf(">>> 访问轮胎: %s\n", wheel.name.UTF8String);
}

@end
```

```objc
Car *car = [Car new];
PrintVisitor *visitor = [PrintVisitor new];
[car accept:visitor];
```