

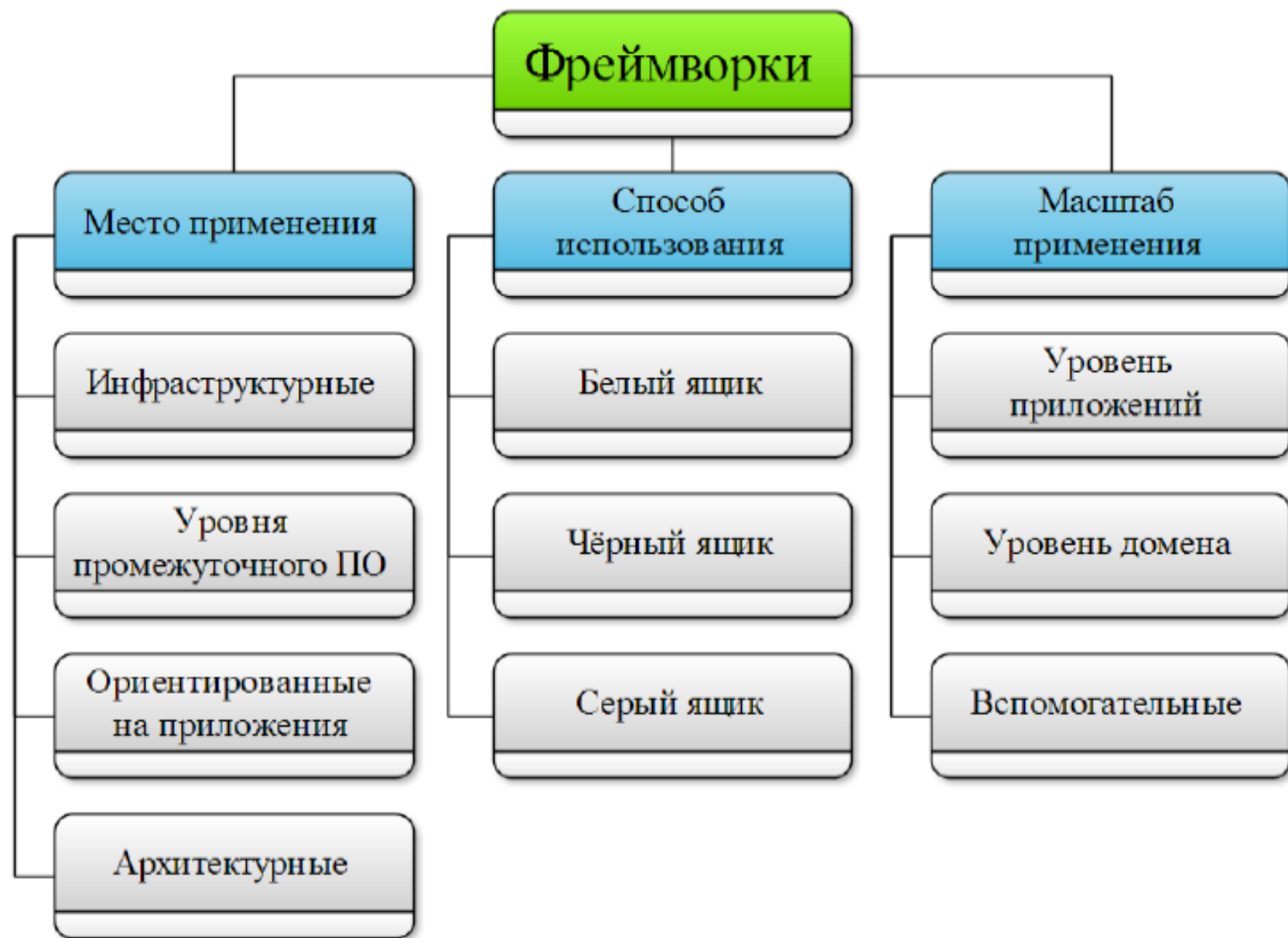
Паттерны проектирования

Общие принципы эффективного программирования

- Повторное использование зарекомендовавшего себя эффективного кода
- Нижний уровень – модули, библиотеки
- Верхний уровень – фреймворки
- Средний – паттерны проектирования

Фреймворки (каркасы)

- Общепринятые архитектурно-структурные решения и подходы к проектированию.
- Фреймворк представляет собой общее решение сложной задачи.



Паттерны проектирования

- Паттерны проектирования это готовые эффективные решения для задач, которые часто встречаются в практике программиста.
- Паттерн проектирования — это часто встречаемое решение определённой проблемы при проектировании архитектуры программ
- Это шаблон решения, по которому вы сможете решить возникшую проблему, внося небольшие частные изменения в существующий паттерн
- Шаблоны проектирования не зависят от языка программирования

Зачем знать паттерны?

- Проверенные решения. Вы тратите меньше времени, используя готовые решения, вместо повторного изобретения велосипеда. До некоторых решений вы смогли бы додуматься и сами, но многие могут быть для вас открытием.
- Стандартизация кода. Вы делаете меньше просчётов при проектировании, используя типовые унифицированные решения, так как все скрытые проблемы в них уже давно найдены.
- Общий программистский словарь. Вы произносите название паттерна, вместо того, чтобы час объяснять другим программистам какой крутой дизайн вы придумали и какие классы для этого нужны

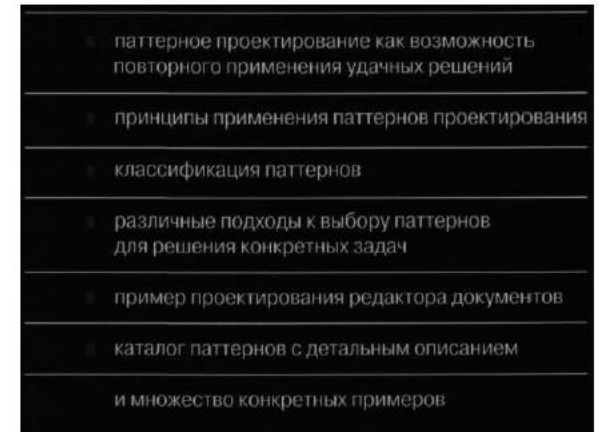
История появления паттернов проектирования

- 70-е годы. Программисты столкнулись с необходимостью разрабатывать большие программы, над которыми должны были трудиться целые команды разработчиков. Были испробованы различные методы организации работы, но сильнее всего на разработку повлияла строительная сфера.
- Концепцию паттернов впервые описал Кристофер Александер в книге “Язык шаблонов. Города. Здания. Строительство”. В этой книге для описания процессов проектирования городов был использован специальный язык – паттерны.
- Паттерны в строительстве описывали типичные проверенные временем решения: какой высоты сделать окна, сколько этажей должно быть в здании, сколько площади в микрорайоне отвести под деревья и газоны.

История появления паттернов проектирования

- 1994 год. Вышла книга “Приемы объектно-ориентированного проектирования. Паттерны проектирования”, в которую вошли 23 паттерна, решающие различные проблемы объектно-ориентированного дизайна.
- Книгу написали 4 автора: Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес. Вскоре все стали называть её “book by the gang of four”, а затем и вовсе “GoF book”.
- И с тех пор были открыты еще другие паттерны проектирования. “Паттерновый” подход стал популярен во всех областях программирования, поэтому сейчас можно встретить всевозможные паттерны и за пределами объектного проектирования.
- Важно! Паттерны — это не какие-то супер-оригинальные решения, а наоборот, часто встречающиеся, типовые решения одной и той же проблемы. Хорошие проверенные временем решения.

Э. Гамма Р. Хелм Р. Джонсон Дж. Влиссидес



Из чего состоит паттерн?

- проблема, которую решает паттерн;
- мотивация к решению проблемы способом, который предлагает паттерн;
- структура классов, составляющих решение;
- пример на одном из языков программирования;
- особенность реализации в различных контекстах;
- связь с другими паттернами.

Список паттернов

- Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы. Проводя аналогию со строительством, можно повысить безопасность перекрестка, поставив светофор, а можно заменить перекресток целой автомобильной развязкой с подземными переходами.
- Самые низкоуровневые и простые паттерны — идиомы. Они не универсальны, поскольку применимы только в рамках одного языка программирования.
- Самые универсальные — архитектурные паттерны, которые можно реализовать практически на любом языке. Они нужны для проектирования всей программы, а не отдельных ее элементов.
- Главное — паттерны отличаются назначением.

Классификация паттернов

- Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы
- Самые низкоуровневые и простые паттерны — *идиомы*. Они не очень универсальные, так как применимы только в рамках одного языка программирования
- Самые универсальные — *архитектурные паттерны*, которые можно реализовать практически на любом языке. Они нужны для проектирования всей программы, а не отдельных её элементов

Группы шаблонов проектирования

- Группы шаблонов проектирования:
 - Порождающие (Creational)
 - Порождающие шаблоны проектирования связаны с механизмом создания новых объектов.
 - Они призваны обеспечить ситуативную гибкость при создании объекта
 - Структурные (Structural)
 - Структурные шаблоны проектирования определяют метод сборки объектов и [классов](#) в более сложные структуры, сохраняя при этом гибкость и эффективность
 - Поведенческие (Behavioral)
 - определяют взаимодействие между классами и объектами, их обязанностями и алгоритмы поведения

B — поведенческие (behavioral);

C — порождающие (creational);

S — структурные (structural).

И наконец список из 23-х паттернов проектирования:

C — Абстрактная фабрика

S — Адаптер

S — Мост

C — Строитель

B — Цепочка обязанностей

B — Команда

S — Компоновщик

S — Декоратор

S — Фасад

C — Фабричный метод

S — Приспособленец

B — Интерпретатор

B — Итератор

B — Посредник

B — Хранитель

C — Прототип

S — Прокси

B — Наблюдатель

C — Одиночка

B — Состояние

B — Стратегия

B — Шаблонный метод

B — Посетитель

Порождающие шаблоны проектирования

Абстрактная фабрика	Abstract Factory	Позволяет создавать семейства взаимосвязанных или взаимозависимых объектов, без указания их конкретных классов.
Строитель	Builder	Интерфейс для пошагового создания сложных объектов.
Фабричный метод	Factory Method	Общий интерфейс для создания объектов в суперклассе, позволяющий подклассам определять тип создаваемого объекта.
Объектный пул	Object Pool	Позволяет использовать уже созданный объект вместо создания нового, в ситуации, когда создание нового объекта требует большого количества ресурсов.
Прототип	Prototype	Позволяет копировать объекты без необходимости учитывать особенности их реализации.
Одиночка	Singleton	Гарантирует, что у класса есть только один экземпляр и предоставляет глобальную точку доступа к нему.
Отложенная инициализация	Lazy initialization	Создание объекта, непосредственно перед его использованием.
Мультитон	Multiton	Шаблон позволяющий создавать несколько одиночек (Singleton), доступ и управление которыми производится через ассоциативную таблицу, например словарь.

Структурные шаблоны проектирования

Адаптер	Adapter	Создание объекта-посредника, который позволит взаимодействовать двум несовместимым объектам.
Мост	Bridge	Разделяет класс на отдельные части: внешнюю абстракцию и внутреннюю реализацию.
Компоновщик	Composite	Идея состоит в том, что группа объектов (контейнер) и сам объект (содержимое контейнера) обладают тем же набором свойств, что позволяет работать с группой как с целым объектом.
Декоратор	Decorator	Добавляет, убирает или изменяет поведение декорированного объекта.
Фасад	Facade	Обертка сложной системы, модуля, пакета в простой интерфейс.
Приспособленец	Flyweight	Использование совместных ресурсов для похожих объектов, вместо выделения ресурсов для каждого объекта по отдельности.
Прокси	Proxy	Создание объекта-подложки для реального объекта, чтобы контролировать обращения к нему, изменять или перенаправлять их.

Поведенческие шаблоны

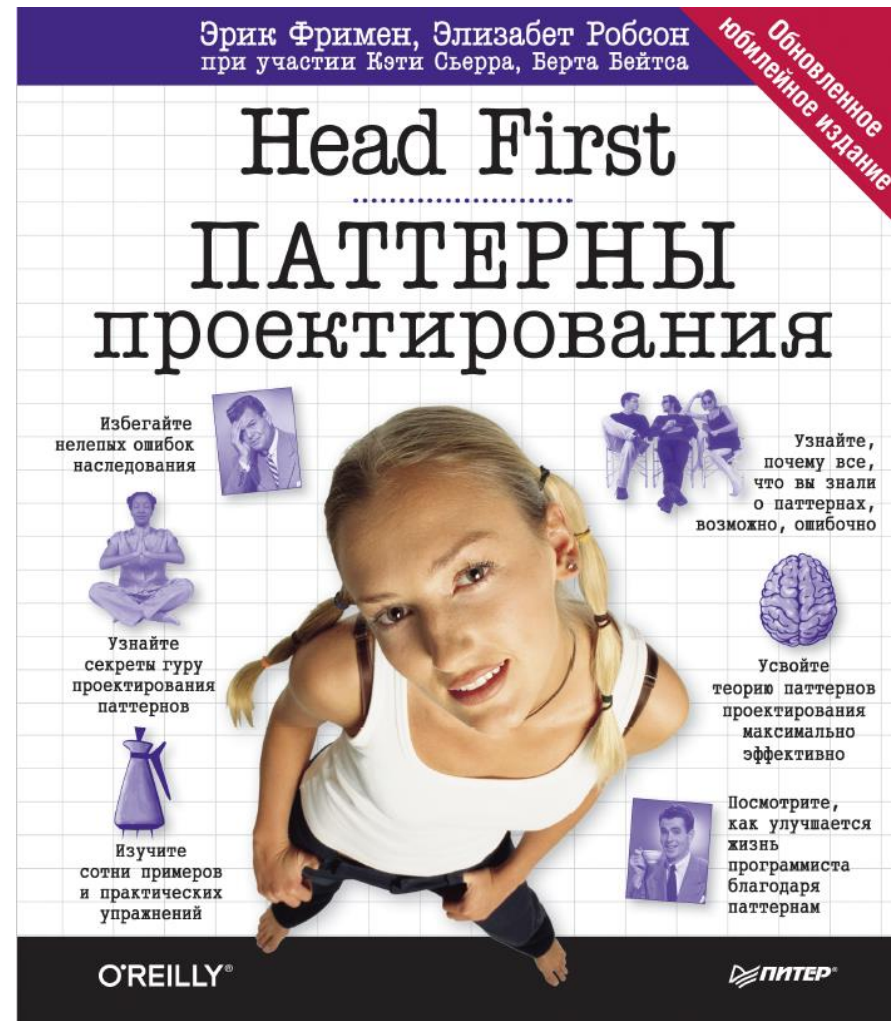
Цепочка обязанностей	Chain of responsibility	Последовательная передача запросов по списку объектов, которые эти запросы обрабатывают и/или передают дальше по цепочке.
Итератор	Iterator	Позволяет последовательно получать объекты из контейнера, не раскрывая особенности реализации контейнера. В Python доступен на встроенном уровне.
Команда	Command	Добавляет слой абстракции между действием и объектом, который это действие вызывает, например, кнопка и действие, которое выполняется при нажатии на эту кнопку.
Посредник	Mediator	Создание такой структуры, в которой объекты не общаются друг с другом, а используют для этого объект-посредник.
Хранитель	Memento	Сохраняет состояние объекта на определенный момент для того, чтобы при необходимости к нему можно было вернуться.
Null Object	Null Object	Объект который может использоваться в случае отсутствия нужного объекта или объект по умолчанию.
Наблюдатель	Observer	Объект "наблюдающий" за состоянием других объектов, информирующий систему / пользователя про изменения состояния наблюдаемого объекта, например пуш-извещения.
Состояние	State	Позволяет изменять поведение объекта в зависимости от его состояния.
Стратегия	Strategy	Позволяет объединить несколько алгоритмов в группу. Порядок применения алгоритмов может изменяться, благодаря чему достигается гибкость всей системы.
Шаблонный метод	Template method	Создание базовых методов и алгоритма их применения в абстрактном родительском классе с тем, чтобы определить конкретные методы в дочерних классах.
Посетитель	Visitor	Шаблон, позволяющий выполнять операции над другими объектами, без необходимости изменять эти объекты.

Э. Гамма Р. Хелм Р. Джонсон Дж. Влиссидес

Приемы объектно-ориентированного проектирования

паттерны проектирования

- паттерное проектирование как возможность повторного применения удачных решений
- принципы применения паттернов проектирования
- классификация паттернов
- различные подходы к выбору паттернов для решения конкретных задач
- пример проектирования редактора документов
- каталог паттернов с детальным описанием и множество конкретных примеров



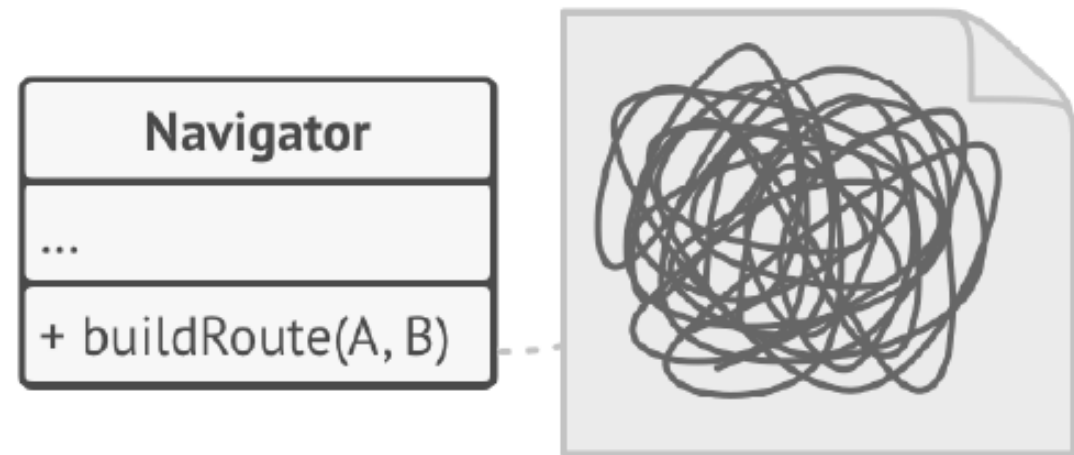
Паттерн Стратегия

Стратегия — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс. После чего, алгоритмы можно взаимозаменять прямо во время исполнения программы

Паттерн Стратегия

- **Задача**
- Нужно написать приложение-навигатор для путешественников. Он должен показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.
- Одной из самых востребованных функций был поиск и прокладка маршрута, поэтому вы планировали посвятить ей особое внимание.
- Пребывая в неизвестном ему городе, пользователь должен иметь возможность указать начальную точку и пункт назначения. А навигатор — проложит оптимальный путь.

- Первая версия: прокладка маршрута лишь по дорогам, отлично подходит для путешествий на автомобиле.
- Следующий шаг: добавление в навигатор прокладки пеших маршрутов.
- Через некоторое время выяснилось, что некоторые люди предпочитают ездить по городу на общественном транспорте, поэтому добавлена и такая опции прокладки пути.
- В ближайшей перспективе будет необходимо добавить прокладку маршрутов по велодорожкам. А в отдалённом будущем — интересные маршруты посещения достопримечательностей.



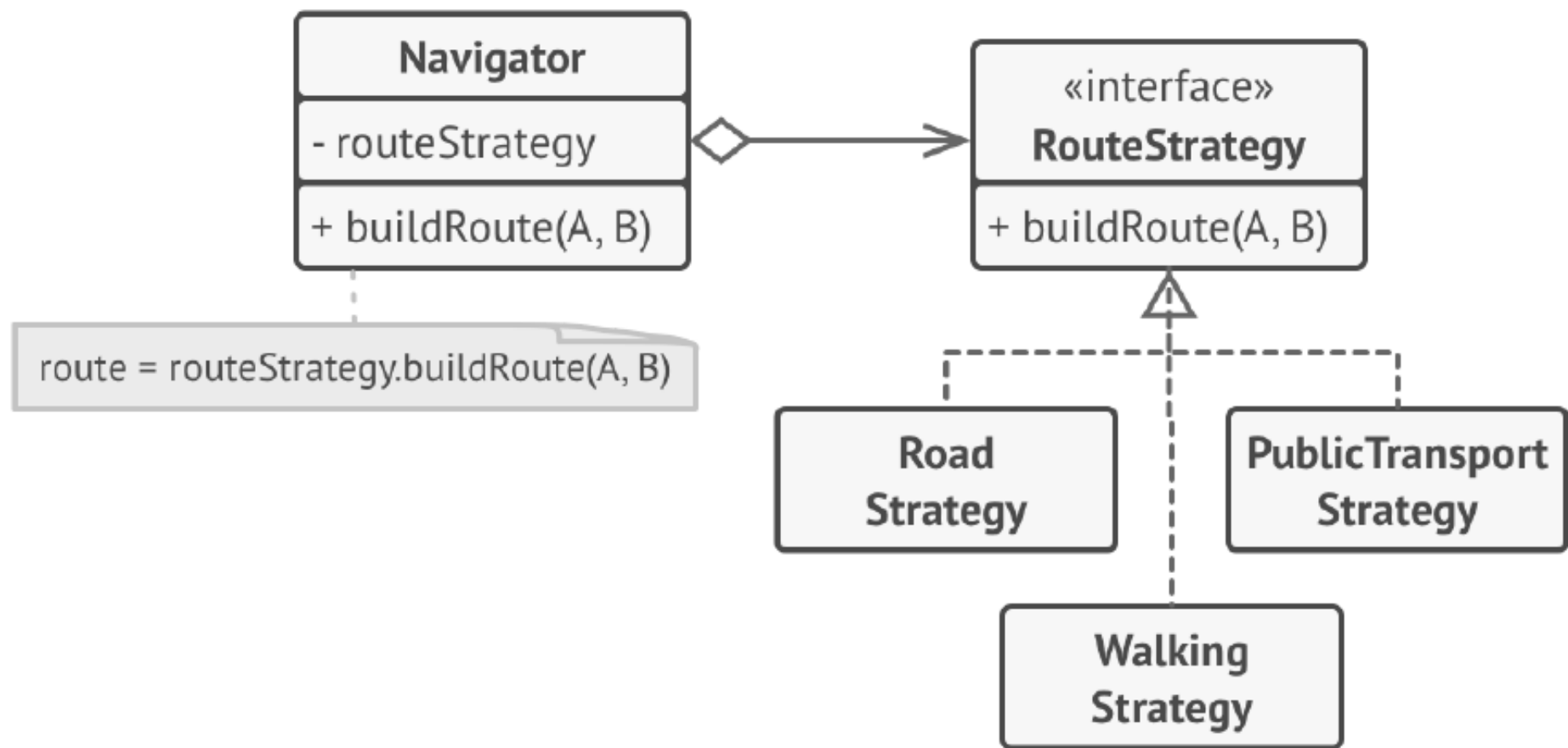
Код навигатора становится слишком раздутым.

Проблема

- С каждым новым алгоритмом, код основного класса навигатора увеличивался вдвое. В таком большом классе стало довольно трудно ориентироваться.
- Любое изменение алгоритмов поиска, будь то исправление багов или добавление нового алгоритма, затрагивало основной класс. Это повышало риск сделать ошибку, случайно задев остальной работающий код.
- Кроме того, осложнялась командная работа с другими программистами, которых наняли после успешного релиза навигатора. Ваши изменения нередко затрагивали один и тот же код, создавая конфликты, которые требовали дополнительного времени на их разрешение.

Решение

- Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями.
- Вместо того чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы.
- Для смены алгоритма будет достаточно подставить в контекст другой объект-стратегию.
- Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.



Стратегии построения пути.

- Каждый алгоритм поиска пути перейдет в свой собственный класс с единственным методом, принимающим в параметрах начальную и конечную точку маршрута и возвращающий массив точек маршрута.
- Хотя каждый класс будет прокладывать маршрут по-своему, для навигатора это не будет иметь никакого значения, так как его работа заключается только в отрисовке маршрута.
- Навигатору достаточно подать в стратегию данные о начале и конце маршрута, чтобы получить массив точек маршрута в оговорённом формате.
- Класс навигатора будет иметь метод изменения стратегии, позволяющий изменять стратегию поиска пути на лету. Им сможет воспользоваться клиентский код навигатора, например, кнопки-переключатели типов маршрутов в пользовательском интерфейсе.

Аналогия из жизни

- Вам нужно добраться до аэропорта. Можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией.
- Вы выбираете конкретную стратегию в зависимости от контекста (например, наличия денег или времени до отлёта).

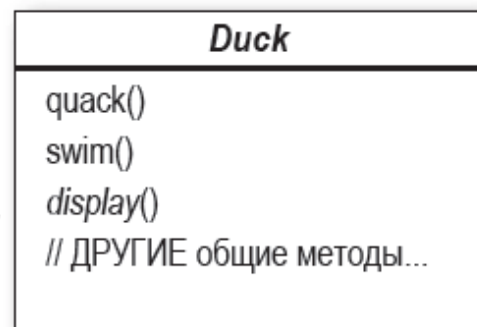


Проектирование SimUDuck

Ситуация

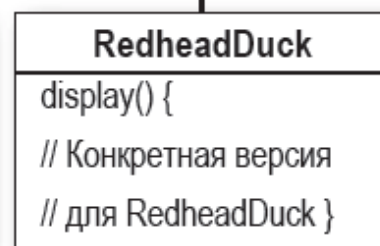
- Джо работает на компанию, выпустившую чрезвычайно успешный имитатор утиного пруда. В этой игре представлен пруд, в котором плавают и крякают утки разных видов.
- Проектировщики системы воспользовались стандартным приемом ООП и определили суперкласс Duck, на основе которого объявляются типы конкретных видов уток.

Все утки умеют крякать (*quack*) и плавать (*swim*); суперкласс предоставляет код обобщенной реализации.



Метод *display()* объявлен абстрактным, потому что все подтипы отображаются по-разному.

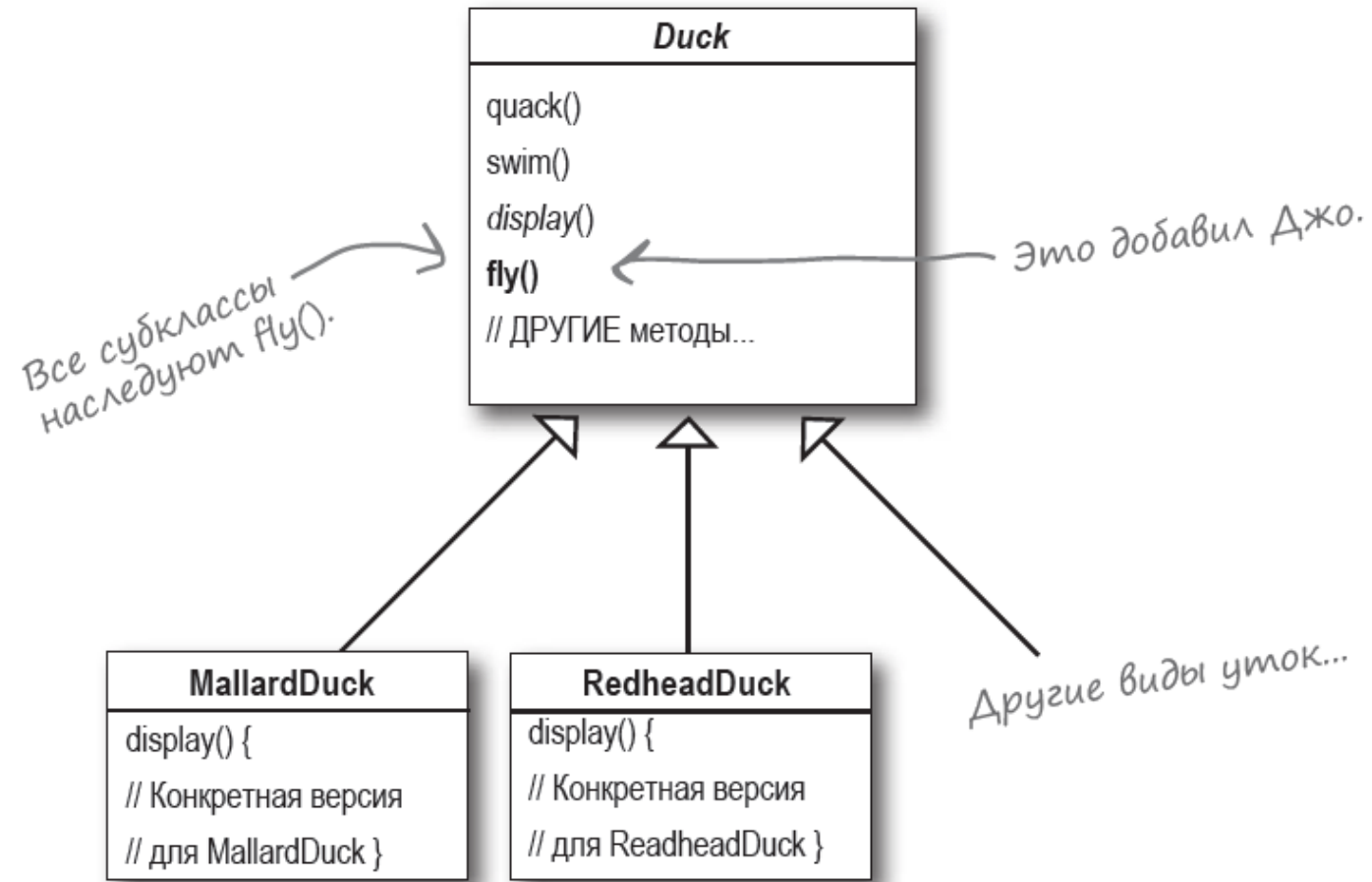
Подтип каждой конкретной разновидности реализует свою специфическую версию *display()*.



Другие типы уток, производные от класса Duck.

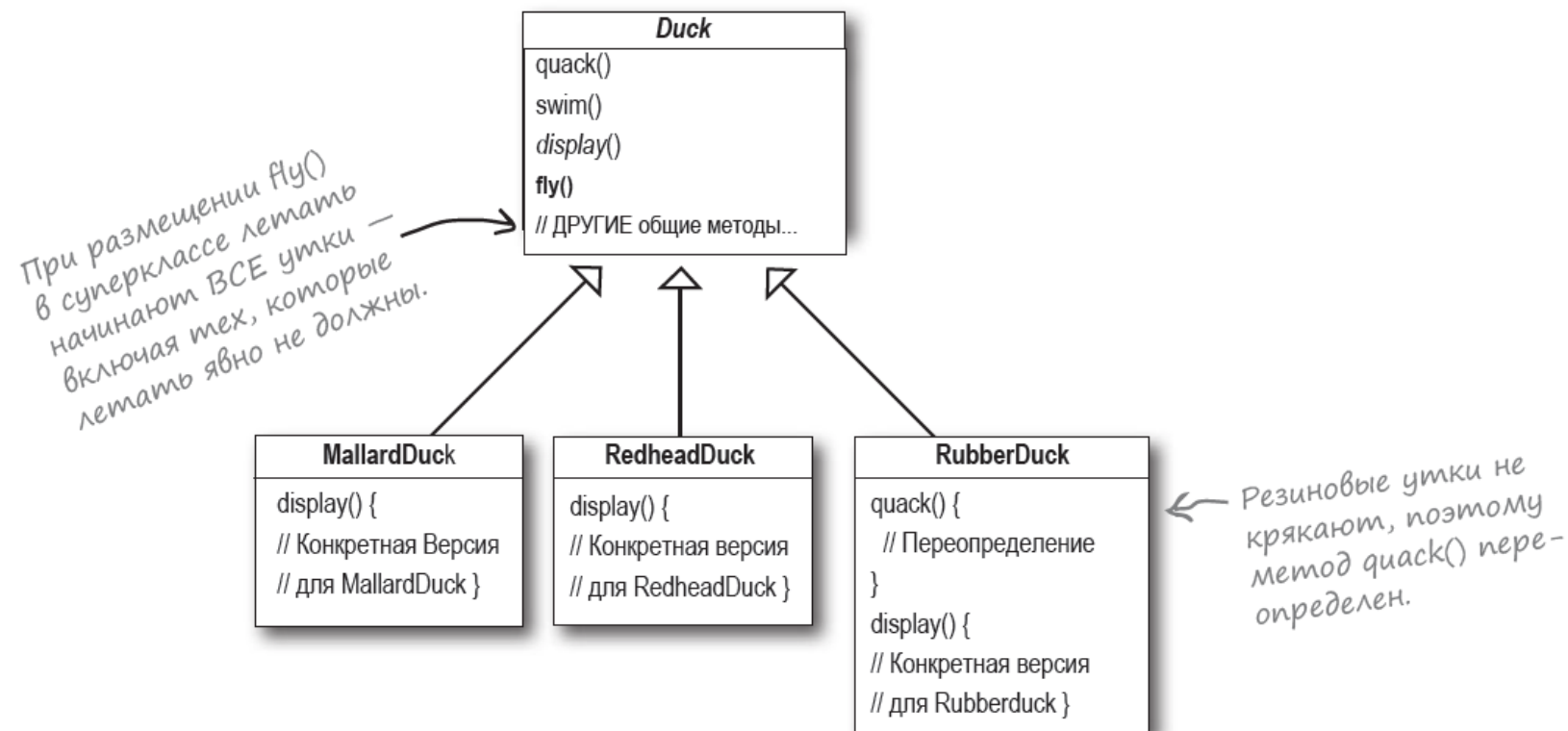
Теперь утки будут ЛЕТАТЬ

- Руководство компании решило, что пришло время серьезных изменений. Нужно сделать что-то действительно впечатляющее, что можно было бы продемонстрировать на предстоящем собрании акционеров на следующей неделе.
- Начальство решило, что летающие утки — именно та «изюминка», которая сокрушит всех конкурентов.



Летающие резиновые утки

- Локальное изменение кода привело к нелокальному побочному эффекту (летающие резиновые утки!)
- Казалось бы, в этой ситуации наследование идеально подходит для повторного использования кода — но с сопровождением возникают проблемы.



- Можно переопределить метод fly() в классе RubberDuck, по аналогии с quack() ...
- Но что произойдет, если в программу добавятся деревянные утки-приманки? Они не должны ни летать, ни крякать...

RubberDuck
<pre>quack() { // Squeak} display() { // RubberDuck } fly() { // Пустое // переопределение // ничего не делает }</pre>

DecoyDuck
<pre>quack() { // Пустое переопределение } display() { // DecoyDuck } fly() { //Пустое переопределение} }</pre>

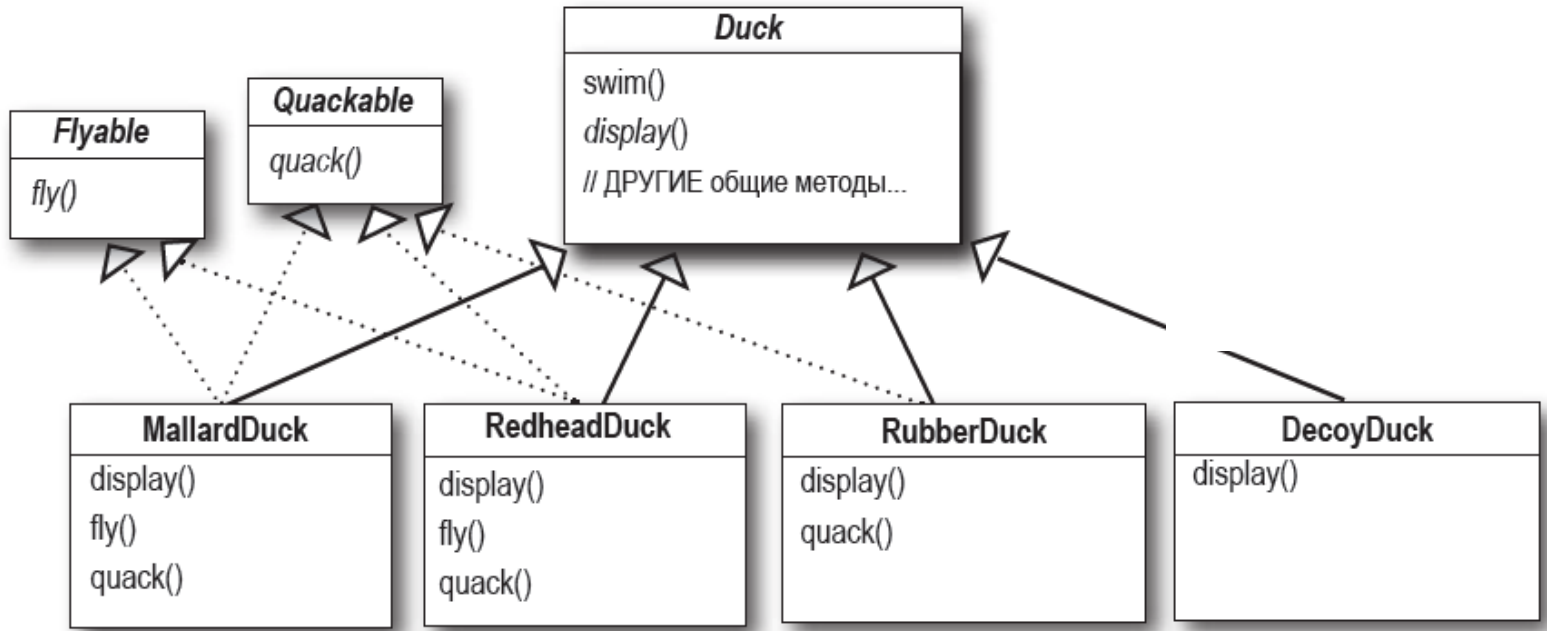


Какие из перечисленных недостатков относятся к применению наследования для реализации Duck? (Укажите все варианты.)

- ☐ A. Дублирование кода в subclasses.
- ☐ B. Трудности с изменением поведения на стадии выполнения.
- ☐ C. Уток нельзя научить танцевать.
- ☐ D. Трудности с получением информации обо всех аспектах поведения уток.
- ☐ E. Утки не могут летать и крякать одновременно.
- ☐ F. Изменения могут оказать непредвиденное влияние на другие классы.

Как насчет интерфейса?

- Наследование не решит проблему
- Нужен более простой способ заставить летать или крякать только некоторых (но не всех!) уток.
- Исключим метод `fly()` из суперкласса `Duck` и определим интерфейс `Flyable()` с методом `fly()`. Только те утки, которые должны летать, реализуют интерфейс и содержат метод `fly()`...
- Аналогично можно определить интерфейс `Quackable`, потому что не все утки крякают.



- Мы знаем, что не все субклассы должны реализовывать методы `fly()` или `quack()`, так что наследование не является оптимальным решением.
- С другой стороны, реализация интерфейсов `Flyable` и (или) `Quackable` решает проблему частично (резиновые утки перестают летать), но полностью исключает возможность повторного использования кода этих аспектов поведения — а следовательно, создает другую проблему из области сопровождения. Кроме того, летающие утки могут летать по-разному...

Изменения

- Как бы вы ни спроектировали свое приложение, со временем оно должно развиваться и изменяться — иначе оно умрет.
- Изменения могут быть обусловлены многими факторами. Укажите некоторые причины для изменения кода в приложениях (Продолжите приведенные случаи).
 - Клиенты или пользователи требуют реализации новой или расширенной функциональности.
 - Компания переходит на другую СУБД, а данные будут приобретаться у другого поставщика в новом формате.

Принцип проектирования

- **Выделите аспекты приложения, которые могут изменяться, и отделите их от тех, которые всегда остаются постоянными.**
- Другая формулировка: выделите переменные составляющие и инкапсулируйте их, чтобы позднее их можно было изменять или расширять без воздействия на постоянные составляющие.

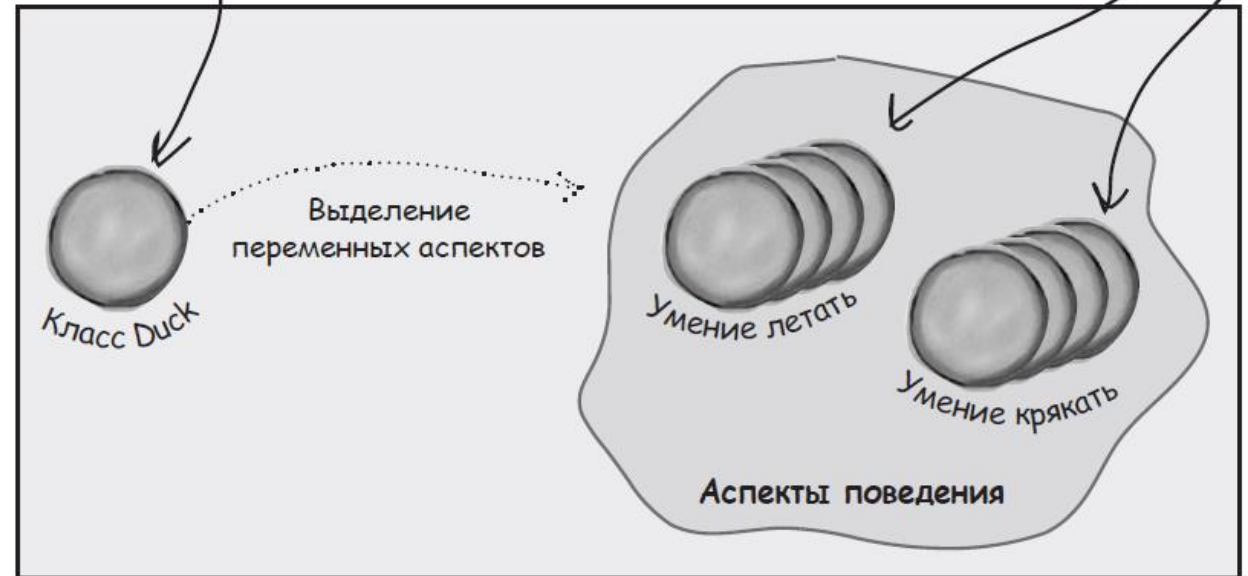
Отделяем переменное от постоянного

- Чтобы отделить «переменное от постоянного», мы создадим два набора классов (совершенно независимых от Duck): один для fly, другой для quack.
- Каждый набор классов содержит реализацию соответствующего поведения.
- Мы знаем, что fly() и quack() — части класса Duck, изменяющиеся в зависимости от subclasses.
- Чтобы отделить эти аспекты поведения от класса Duck, мы выносим оба метода за пределы класса Duck и создаем новый набор классов для представления каждого аспекта.

Класс Duck остается суперклассом для всех уток, но некоторые аспекты поведения выделяются в отдельную структуру классов.

Для каждого переменного аспекта создается свой набор классов.

Разные реализации поведения.

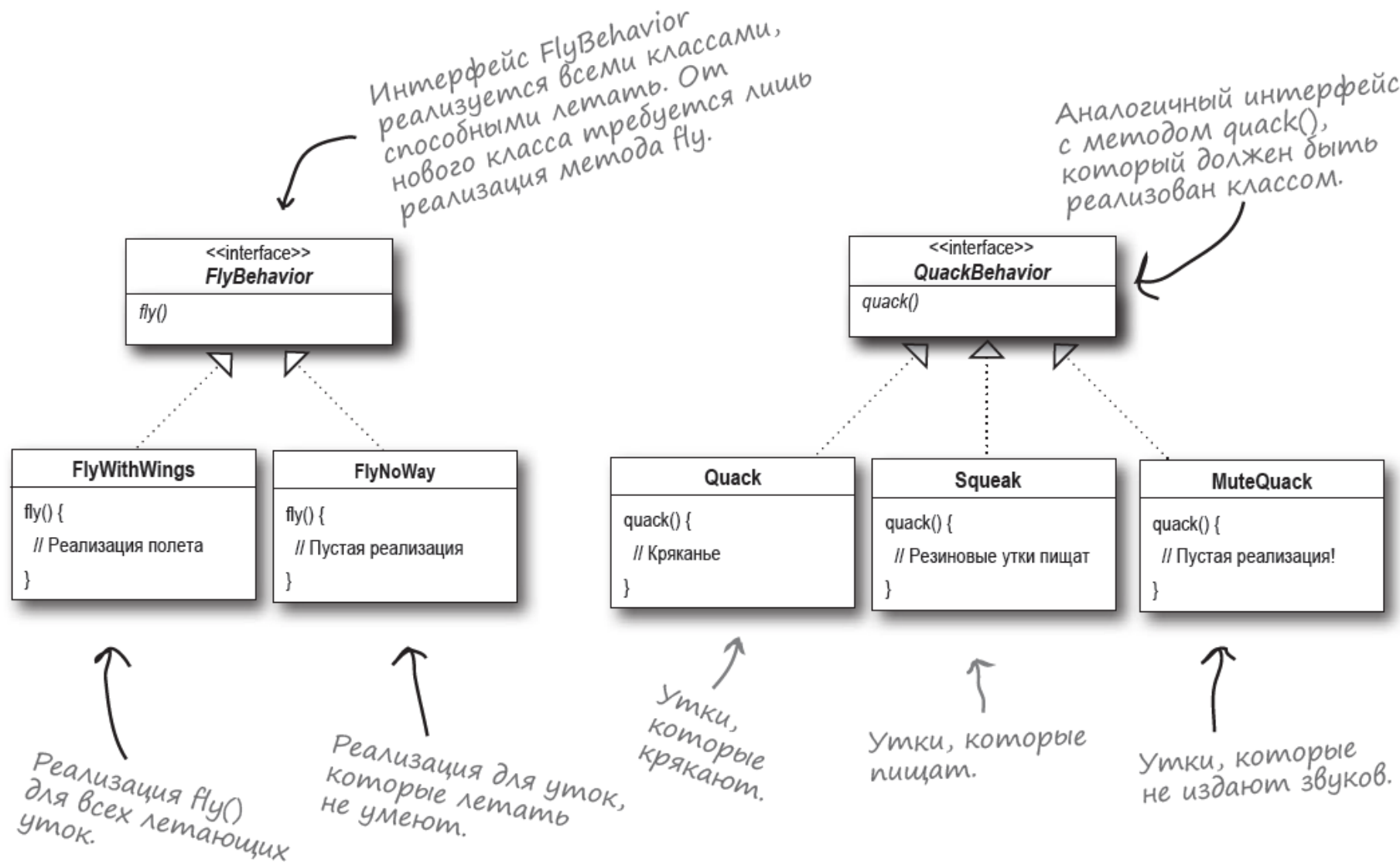


Принцип проектирования

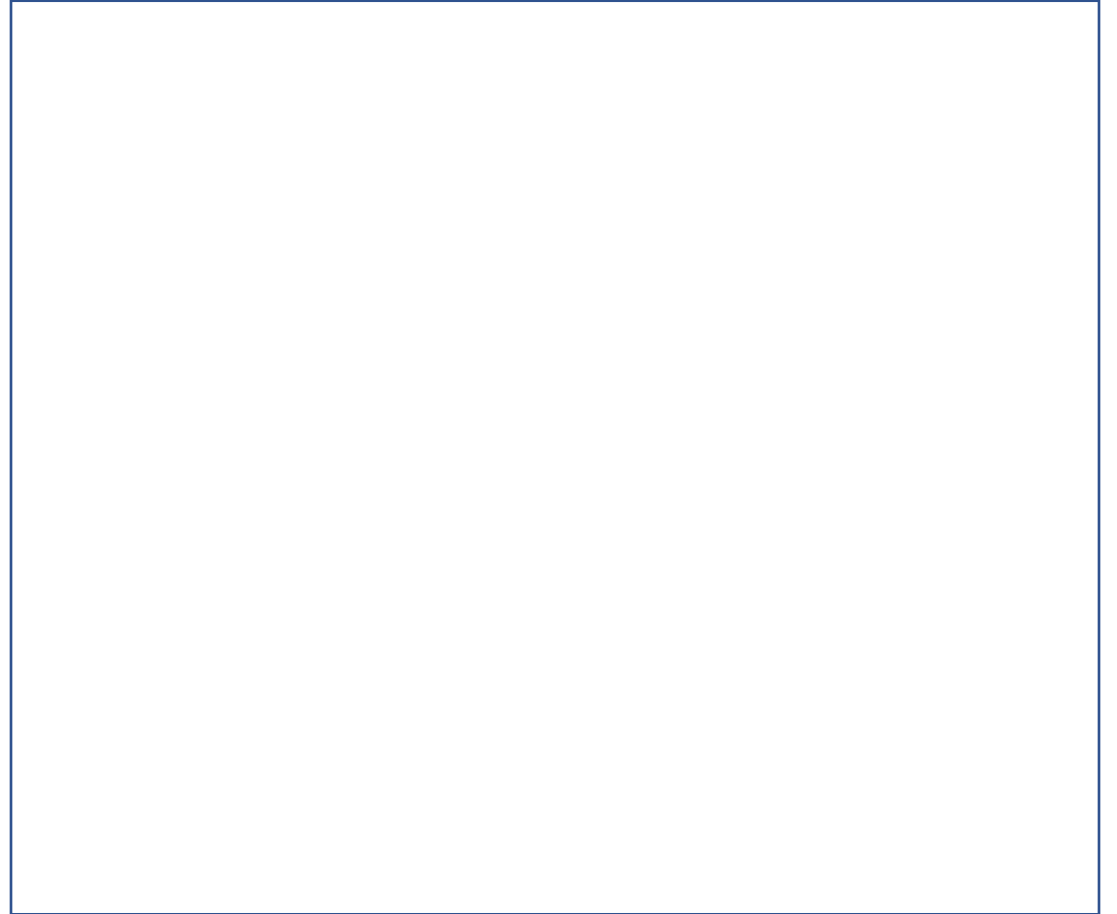
- **Программируйте на уровне интерфейса, а не на уровне реализации.**
- Желательно иметь возможность создать новый экземпляр `MallardDuck` и инициализировать его с конкретным типом поведения `fly()`.
- Почему бы не предусмотреть возможность динамического изменения поведения?
- Иначе говоря, в классы `Duck` следует включить методы выбора поведения, чтобы способ полета `MallardDuck` можно было изменить во время выполнения.

Реализация поведения уток

- Такая архитектура позволяет использовать поведение `fly()` и `quack()` в других типах объектов, потому что это поведение не скрывается в классах `Duck`!
- Мы можем добавлять новые аспекты поведения без изменения существующих классов поведения, и без последствий для классов `Duck`, использующих существующее поведение.

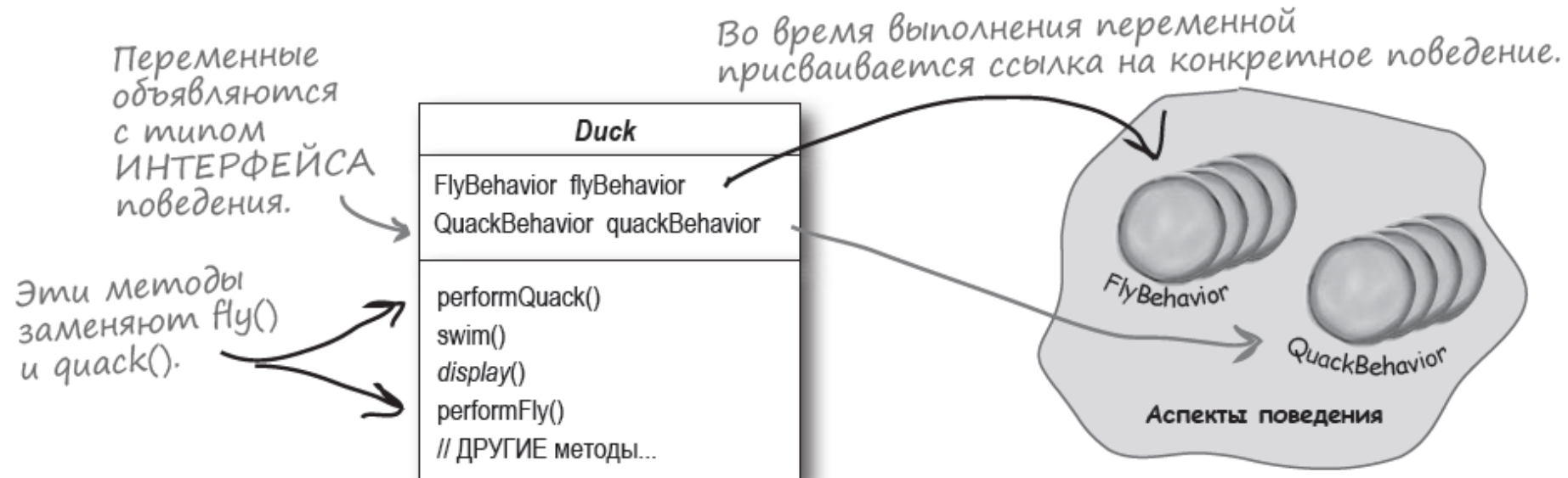


1. Как бы вы поступили в новой архитектуре, если бы вам потребовалось включить в приложение SimUDuck полеты на реактивной тяге?
2. Какой класс мог бы повторно использовать поведение `quack()`, не являясь при этом уткой?



Интеграция поведения с классом Duck

- Класс Duck теперь делегирует свои аспекты поведения (вместо простого использования методов, определенных в классе Duck или его subclasses)
- В класс Duck включаются две переменные экземпляров flyBehavior и quackBehavior, объявленные с типом интерфейса. Каждый объект на стадии выполнения присваивает этим переменным полиморфные значения, соответствующие конкретному типу поведения (FlyWithWings, Squeak и т. д.).
- Методы fly() и quack() удаляются из класса Duck (и всех subclasses), потому что это поведение перемещается в классы FlyBehavior и QuackBehavior.
- В классе Duck методы fly() и quack() заменяются двумя аналогичными методами: performFly() и performQuack()



Реализация performQuack()

```
public class Duck {  
    QuackBehavior quackBehavior;  
    // ...  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
}
```

← • Каждый объект Duck содержит ссылку на реализацию интерфейса QuackBehavior.

← • Объект Duck делегирует поведение объекту, на который ссылается quackBehavior.

Как присваиваются значения переменных flyBehavior и quackBehavior

```
public class MallardDuck extends Duck {  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
    public void display() {  
        System.out.println("I'm a real  
                             Mallard duck");  
    }  
}
```

- MallardDuck наследует переменные quackBehavior и flyBehavior от класса Duck.
- MallardDuck использует класс Quack для выполнения действия, так что при вызове performQuack() ответственность за выполнение возлагается на объект Quack.
- В качестве реализации FlyBehavior используется тип FlyWithWings.
- При создании экземпляра MallardDuck конструктор инициализирует **унаследованную переменную экземпляра quackBehavior** новым экземпляром типа Quack (класс конкретной реализации QuackBehavior).
- То же самое происходит и с другим аспектом поведения: конструктор MallardDuck инициализирует переменную flyBehavior экземпляром типа FlyWithWings (класс конкретной реализации FlyBehavior).

Тестирование кода Duck

```
# Утка Mallard
mallard = MallardDuck()
mallard.display()
mallard.swim()

mallard.performQuack()
mallard.performFly()
```

```
>>>
```

I'm a real Mallard duck

All ducks float, even decoys!

Quack

I'm flying!!

Динамическое изменение поведения

- Тип поведения может задаваться set-методом подкласса (вместо создания экземпляра в конструкторе)
- Создать новый субкласс Duck (ModelDuck)
- Определить новый тип FlyBehavior (FlyRocketPowered)
- Внести изменения в симулятор MiniDuckSimulator.
- Добавить экземпляр ModelDuck и перевести его на реактивную тягу

<i>Duck</i>
FlyBehavior flyBehavior; QuackBehavior quackBehavior;
swim() display() performQuack() performFly() setFlyBehavior() setQuackBehavior() // ДРУГИЕ методы...

Поведение утки во время выполнения изменяется простым вызовом set-метода.

Утка-модель

```
model = ModelDuck()  
model.display()  
model.performQuack()  
model.performFly()  
model.setFlyBehavior(FlyRocketPowered)  
model.performFly()
```

>>>

I'm a model duck

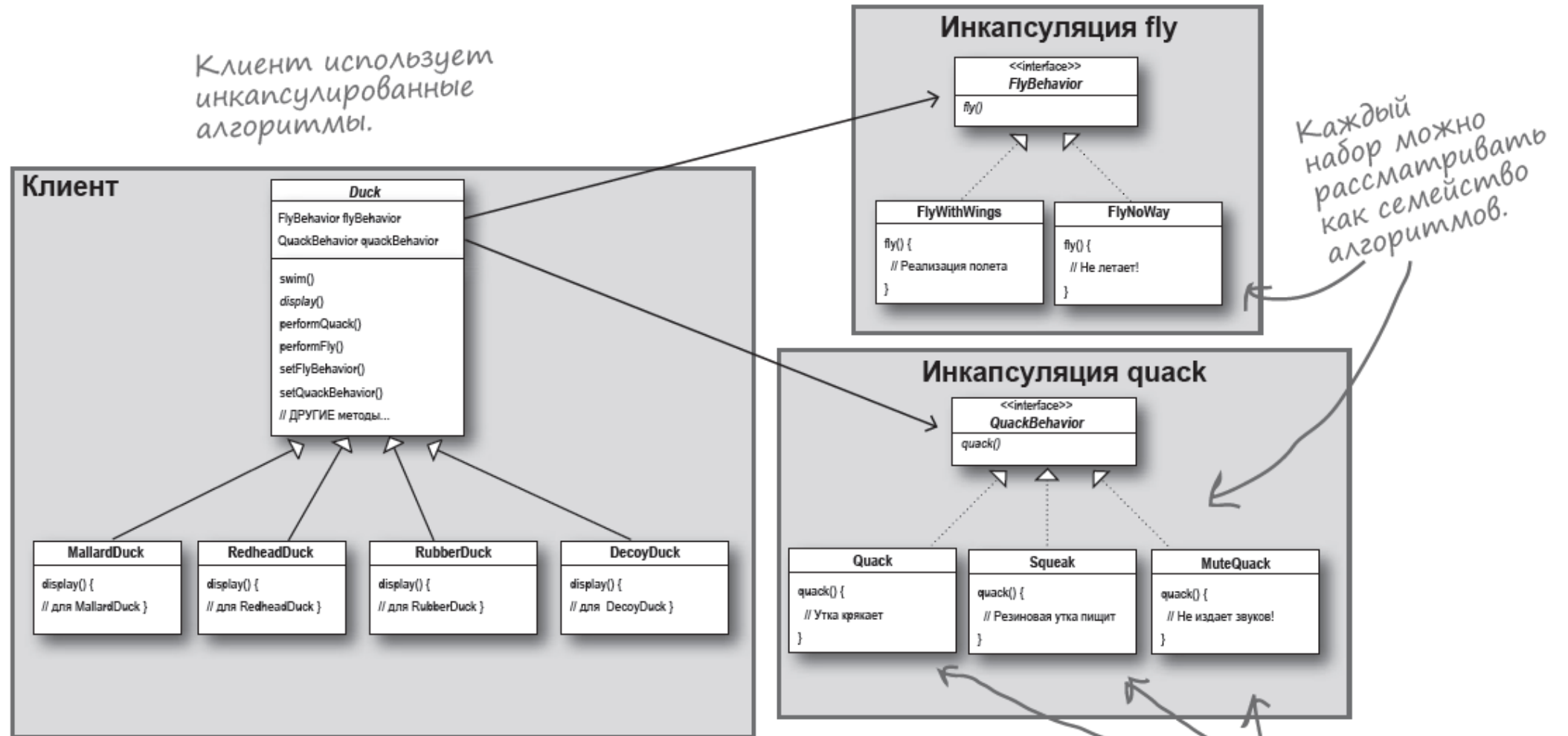
Quack

I can't fly

I'm flying with a rocket!

Способность утки-приманки к полету переключается динамически! Если бы реализация находилась в иерархии Duck, ТАКОЕ было бы невозможно.

Инкапсуляция поведения: общая картина



ЯВЛЯЕТСЯ

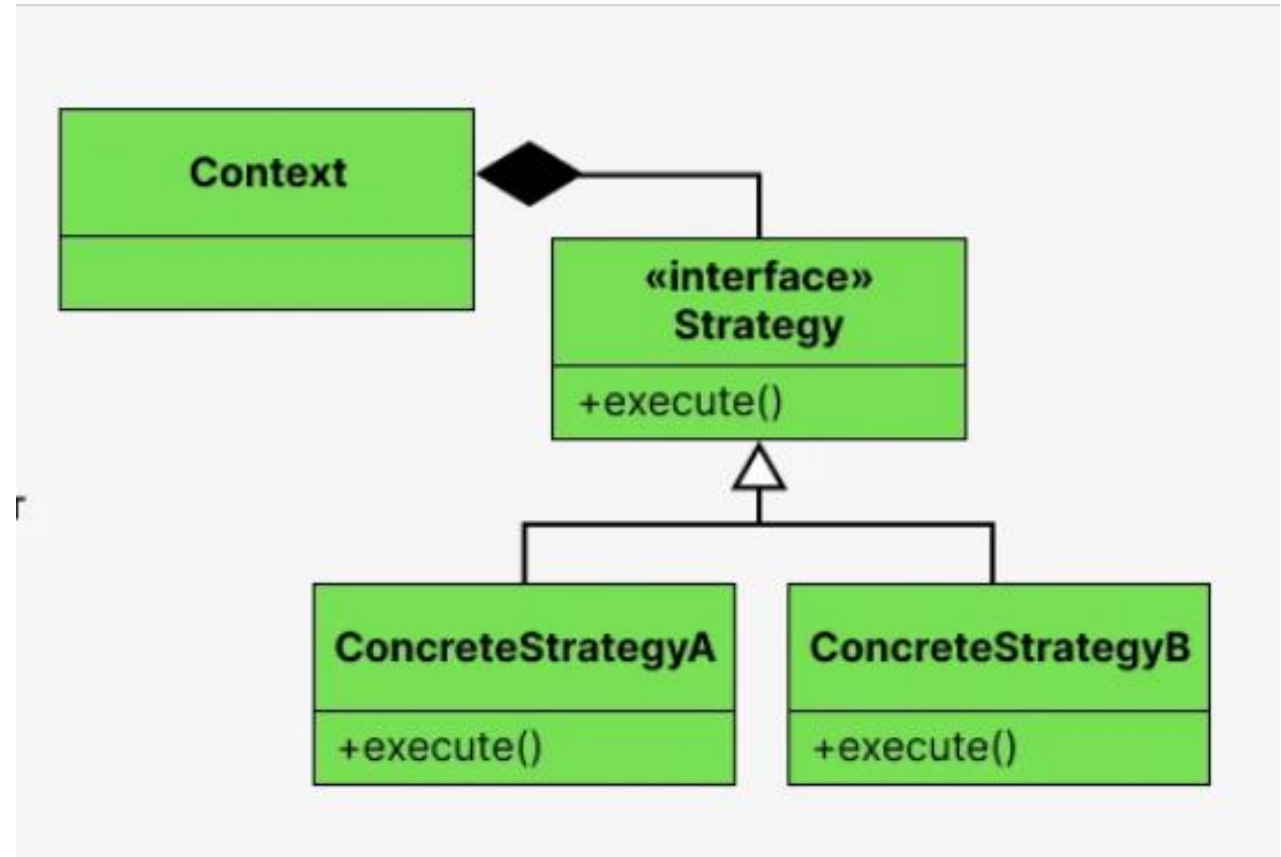
СОДЕРЖИТ

РЕАЛИЗУЕТ

Эти ~~алгоритмы~~ алгоритмы взаимозаменяемы.

Паттерн Стратегия - Определение

- Паттерн Стратегия определяет семейство алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость.
- Позволяет модифицировать алгоритмы независимо от их использования на стороне клиента.
- **Сильные стороны:**
 - инкапсуляция реализации различных алгоритмов, система становится независимой от возможных изменений бизнес-правил;
 - вызов всех алгоритмов одним стандартным образом;
 - отказ от использования переключателей и/или условных операторов.



Применимость

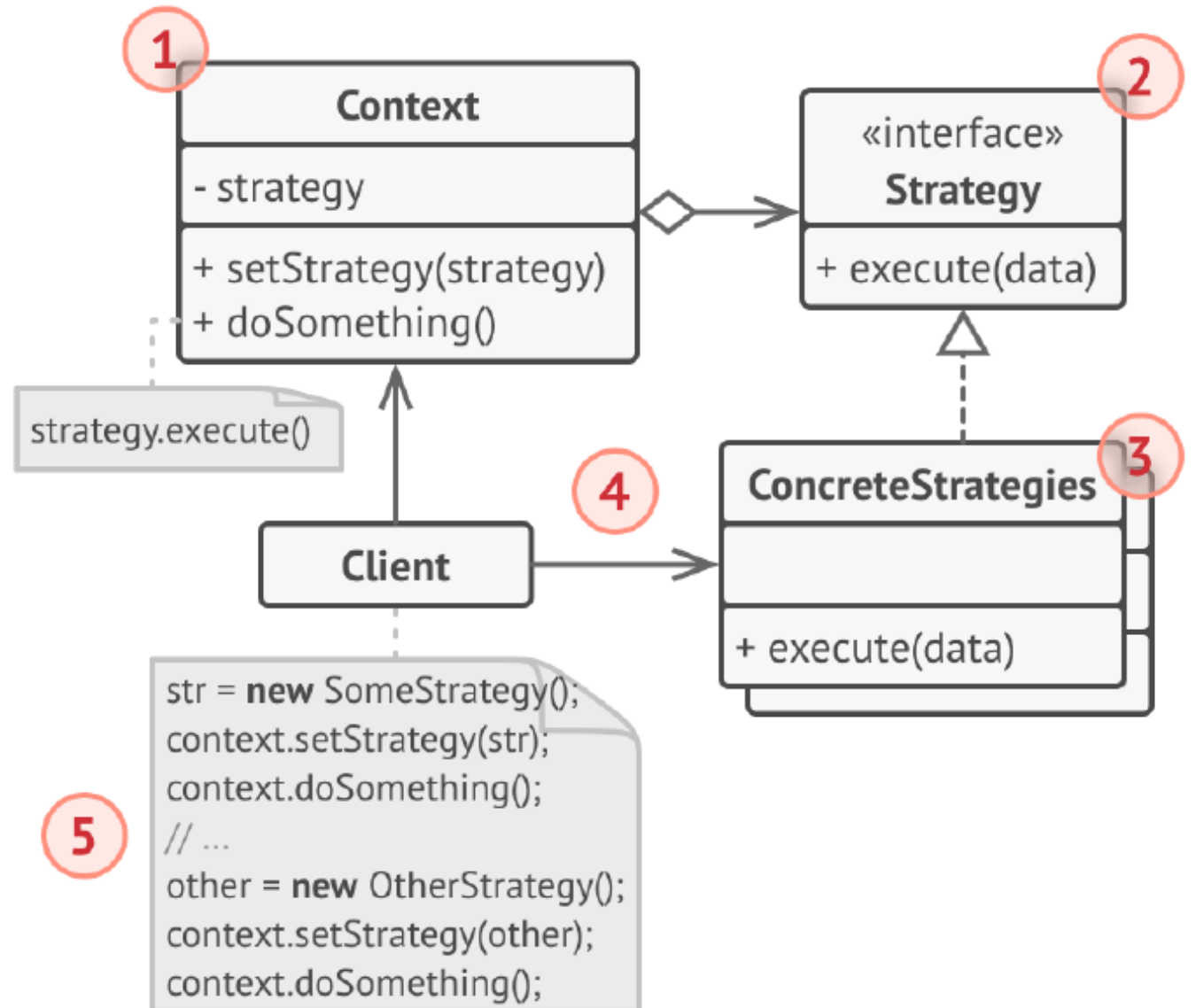
- Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
- Имеется много похожих классов, отличающихся только поведением. Стратегия позволяет сконфигурировать класс, задав одно из возможных поведений;
- Когда вы не хотите обнажать детали реализации алгоритмов для других классов.
- Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет вариацию алгоритма.

Шаги реализации

1. Определите алгоритм, который подвержен частым изменениям. Также подойдёт алгоритм, имеющий несколько вариаций, которые выбираются во время выполнения программы.
2. Создайте интерфейс стратегий, описывающий этот алгоритм. Он должен быть общим для всех вариантов алгоритма.
3. Поместите вариации алгоритма в собственные классы, которые реализуют этот интерфейс.
4. В классе контекста создайте поле для хранения ссылки на текущий объект-стратегию, а также метод для её изменения. Убедитесь в том, что контекст работает с этим объектом только через общий интерфейс стратегий.
5. Клиенты контекста должны подавать в него соответствующий объект-стратегию, когда хотят, чтобы контекст вёл себя определённым образом.

Структура

- 1. Контекст хранит ссылку на объект конкретной стратегии, работая с ним объектом через общий интерфейс стратегий.
- 2. Стратегия определяет интерфейс, общий для всех вариаций алгоритма. Контекст использует этот интерфейс для вызова алгоритма.
- Для контекста не важно, какая именно вариация алгоритма будет выбрана, так как все они имеют одинаковый интерфейс.
- 3. Конкретные стратегии реализуют различные вариации алгоритма.
- 4. Во время выполнения программы, контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.
- 5. Обычно, клиент должен создать объект конкретной стратегии и передать его в контекст: либо через конструктор, либо в какой-то другой решающий момент, используя сеттер. Благодаря этому, контекст не знает о том, какая именно стратегия сейчас выбрана.



Преимущества и недостатки

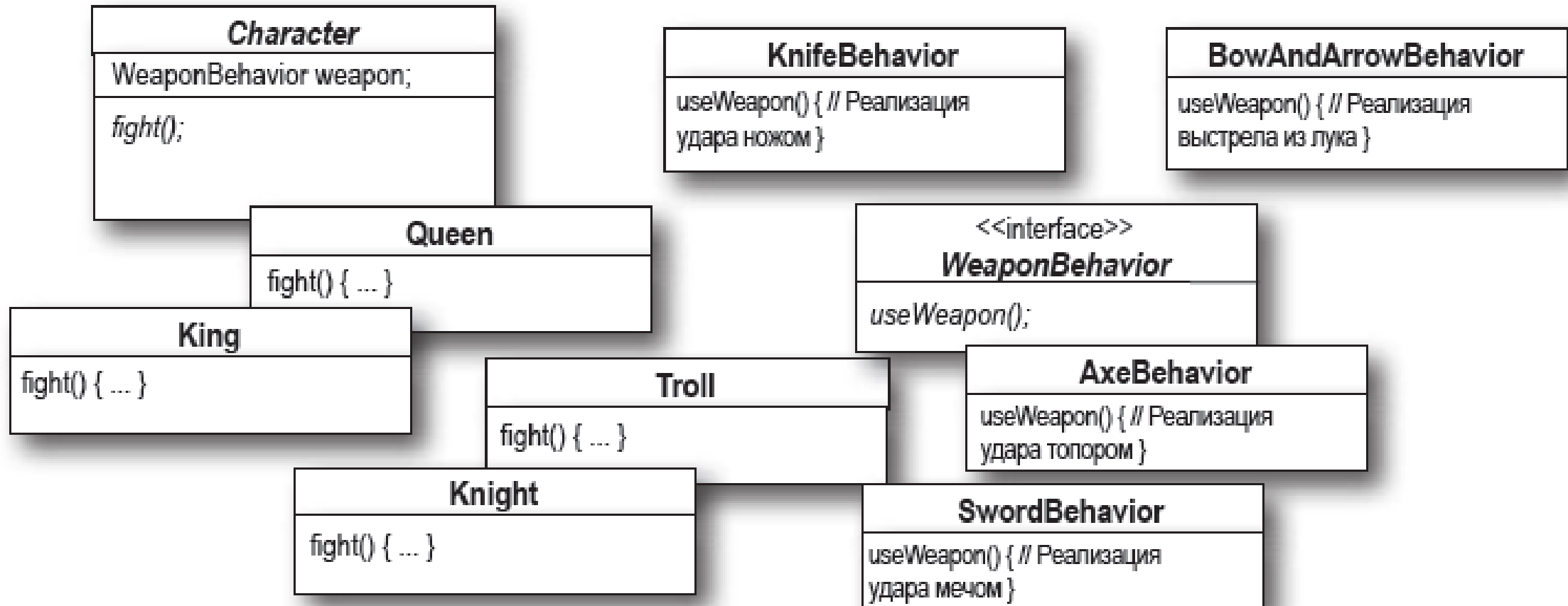
- Горячая замена алгоритмов на лету.
- Изолирует код и данные алгоритмов от остальных классов.
- Уход от наследования к делегированию.
- Реализует принцип открытости/закрытости.
- Усложняет программу за счёт дополнительных классов.
- Клиент должен знать, в чём разница между стратегиями, чтобы выбрать подходящую.

Выводы

- Концепции ООП
 - Абстракция
 - Инкапсуляция
 - Полиморфизм
 - Наследование
- Принципы
 - Инкапсулируйте то, что изменяется.
 - Отдавайте предпочтение композиции перед наследованием.
 - Программируйте на уровне интерфейсов, а не реализации.

Упражнение

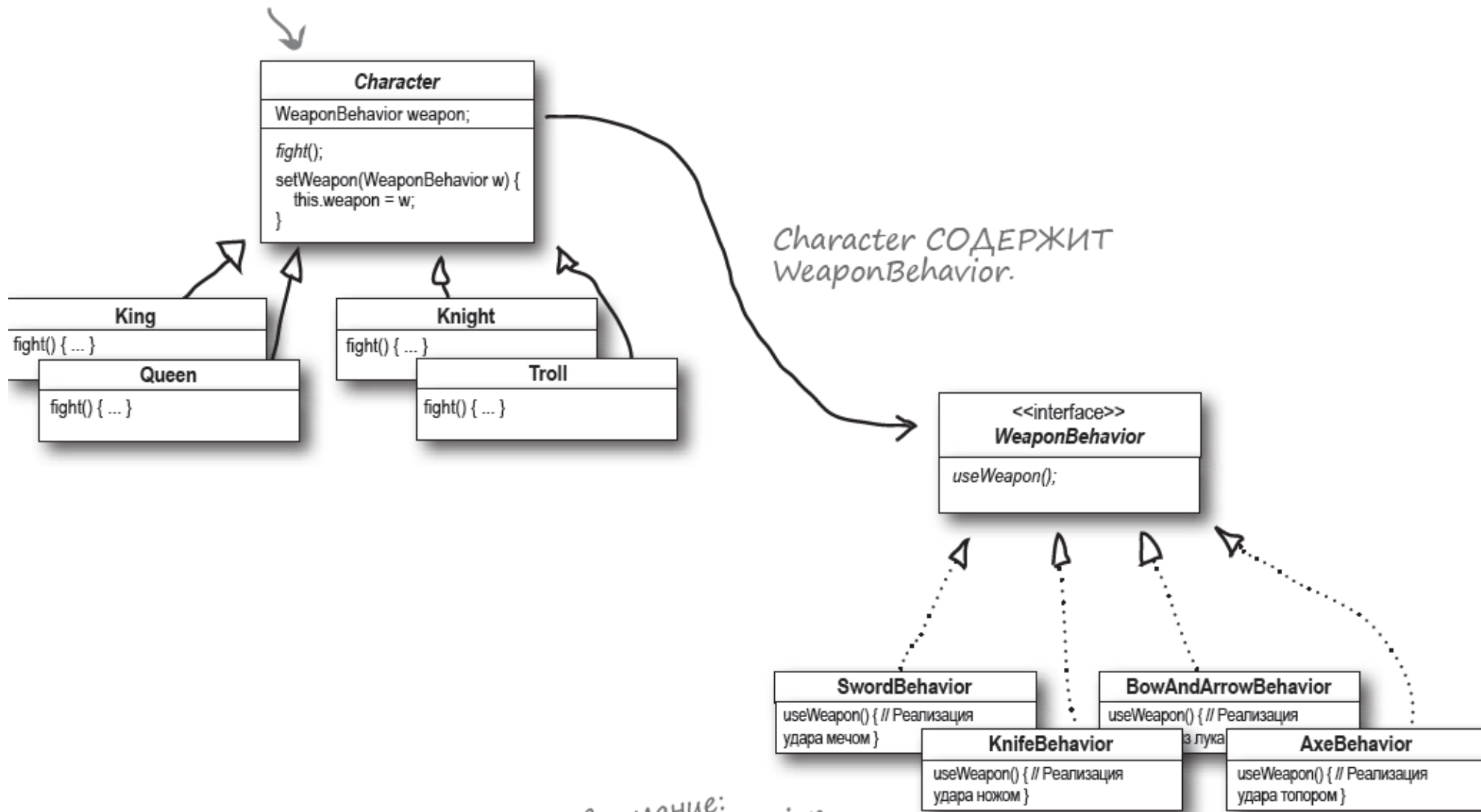
Классы и интерфейсы, используемые для программирования приключенческой игры. В иерархию входят классы игровых персонажей и разных типов вооружения. Каждый персонаж в любой момент времени использует только один вид оружия, но может свободно менять оружие в ходе игры. Восстановите отсутствующие связи.



Решение

1. Организуйте классы в иерархию.
2. Найдите один абстрактный класс, один интерфейс и восемь классов.
3. Соедините классы стрелками.
 1. Отношение наследования («расширяет»):
 2. Отношение реализации интерфейса:
 3. Отношение типа «СОДЕРЖИТ»:
4. Включите метод `setWeapon()` в правильный класс.

абстрактный



Обратите внимание:
интерфейс WeaponBehavior
может быть реализован
ЛЮБЫМ объектом, будь
то скрепка, тюбик зубной
пасты или рыба-мутант.

Паттерн Наблюдатель

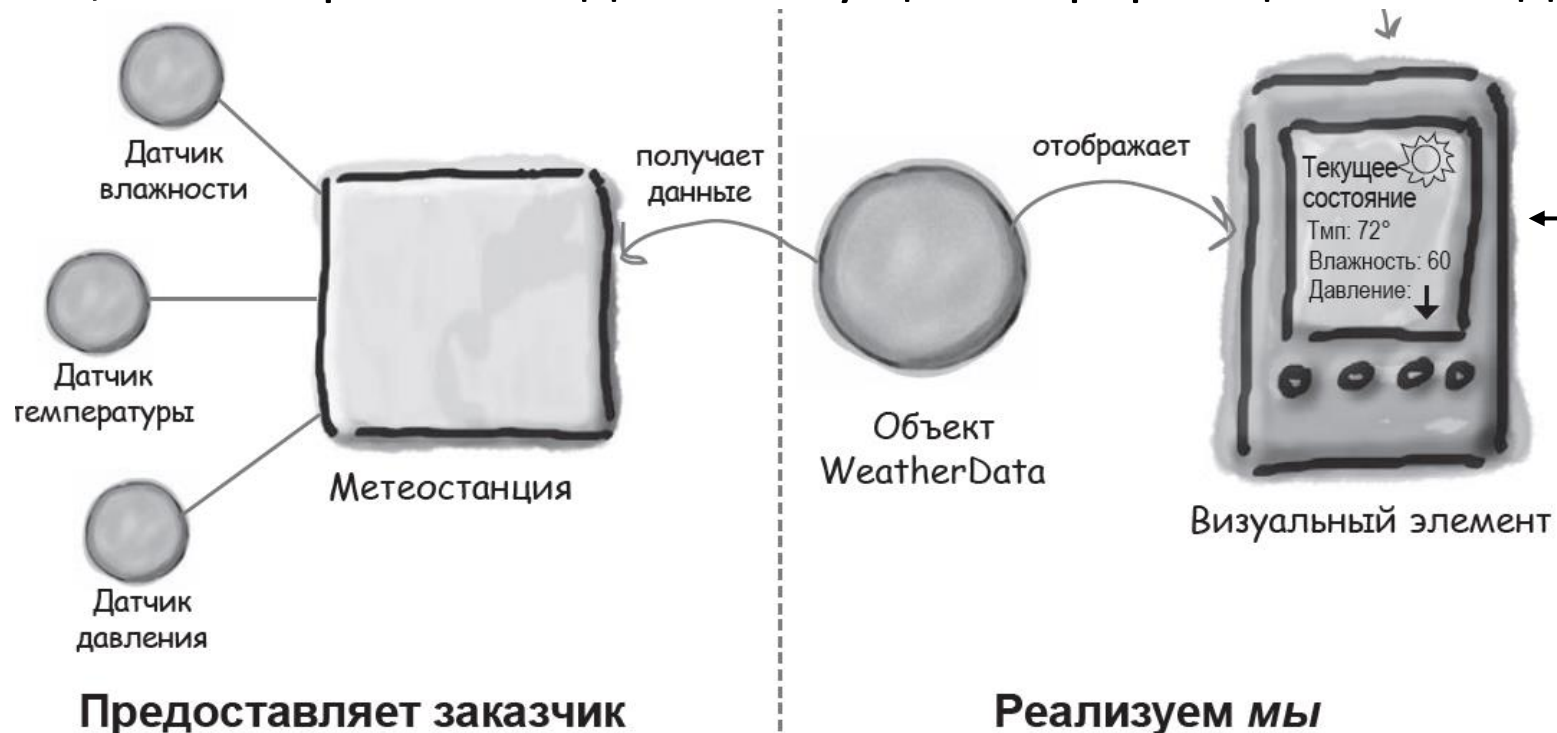
Паттерн оповещает объекты о наступлении неких событий, которые могут представлять для них интерес, — причем объекты даже могут решать во время выполнения, желают ли они и дальше получать информацию

Техническое задание

- Метеостанция работает на базе запатентованного объекта WeatherData, отслеживающего текущие погодные условия (температура, влажность, атмосферное давление).
- Вы должны создать приложение, которое изначально отображает три визуальных элемента: текущую сводку, статистику и простой прогноз. Все данные обновляются в реальном времени, по мере того как объект WeatherData получает данные последних измерений.
- Кроме того, необходимо предусмотреть возможность расширения программы. Определите API, чтобы другие разработчики могли писать собственные визуальные элементы для отображения погодной информации и подключать их к приложению.

Обзор приложения Weather Station

- Система состоит из трех компонентов:
 - метеостанции (физического устройства, занимающегося сбором данных),
 - объекта WeatherData (отслеживает данные, поступающие от метеостанции, и обновляет отображаемую информацию), и
 - экрана, на котором выводится текущая информация о погоде.

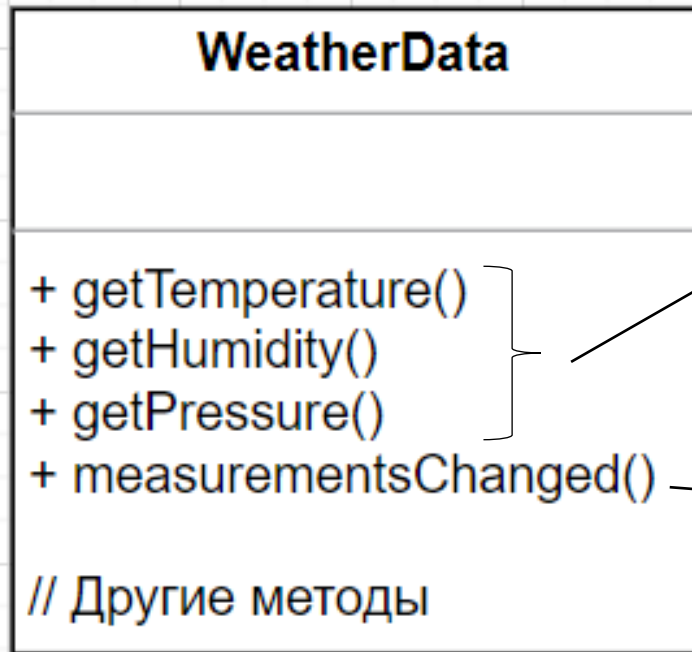


Текущее состояние — один из трех режимов вывода. Пользователь также может запросить статистику и прогноз погоды.

Задача

- Объект `WeatherData` умеет получать от физической метеостанции обновленные данные. Затем объект `WeatherData` обновляет изображение для трех основных элементов: текущего состояния (температура, влажность и давление), статистики и прогноза.
- Текущее состояние — один из трех режимов вывода.
- Пользователь также может запросить статистику и прогноз погоды.
- Наша задача — создать приложение, которое использует данные объекта `WeatherData` для обновления текущих условий, статистики и прогноза погоды.

Класс WeatherData



Эти три метода возвращают новейшие значения температуры, влажности и атмосферного давления соответственно.

Нас не интересует, КАК задаются их значения; объект `WeatherData` знает, как получить обновленную информацию от метеостанции.

- Метод `measurementsChanged()` необходимо реализовать так, чтобы он обновлял изображение для трех элементов: текущего состояния, статистики и прогноза.

```
/*
 *
 * Метод вызывается при каждом
 * обновлении показаний датчиков
 *
 */
public void measurementsChanged() {
    // Здесь размещается ваш код
}
```

Что нам известно?

- Класс `WeatherData` содержит `get`-методы для показаний трех датчиков: температуры, влажности и атмосферного давления.
- Метод `measurementsChanged()` вызывается при появлении новых метеорологических данных. (Мы не знаем, как он вызывается, да это и неважно; достаточно знать, что он вызывается.)
- Необходимо реализовать три экрана вывода, использующих метеорологические данные: экран текущего состояния, экран статистики и экран прогноза. Эти экраны должны обновляться каждый раз, когда у объекта `WeatherData` появляются новые данные.
- Система должна быть расширяемой — другие разработчики будут создавать новые экраны вывода, а пользователи могут добавлять и удалять их в своих приложениях. В настоящее время определены всего три вида экранов (текущее состояние, статистика и прогноз).

Первый вариант реализации

```
public class WeatherData {  
    // Объявления переменных экземпляров  
    public void measurementsChanged() {  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
        currentConditionsDisplay.update(temp, humidity, pressure);  
        statisticsDisplay.update(temp, humidity, pressure);  
        forecastDisplay.update(temp, humidity, pressure);  
    }  
    // Другие методы WeatherData  
}
```

Чтобы получить обновленные данные, мы вызываем (уже реализованные) get-методы класса WeatherData

Запрос на перерисовку каждого элемента с передачей обновленных значений.

Какие из следующих утверждений относятся к первой реализации? (Укажите все варианты.)

- ☐ А. Мы программируем на уровне реализаций, а не интерфейсов.
- ☐ В. Для каждого нового элемента придется изменять код.
- ☐ С. Элементы не могут добавляться (или удаляться) во время выполнения.
- ☐ D. Элементы не реализуют единый интерфейс.
- ☐ Е. Переменные аспекты архитектуры не инкапсулируются.
- ☐ F. Нарушается инкапсуляция класса WeatherData.

Первый вариант реализации

```
public class WeatherData {  
    // Объявления переменных экземпляров  
    public void measurementsChanged() {  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        currentConditionsDisplay.update(temp, humidity, pressure);  
        statisticsDisplay.update(temp, humidity, pressure);  
        forecastDisplay.update(temp, humidity, pressure);  
    }  
    // Другой вариант реализации  
}
```

Переменная область — необходимо инкапсулировать.

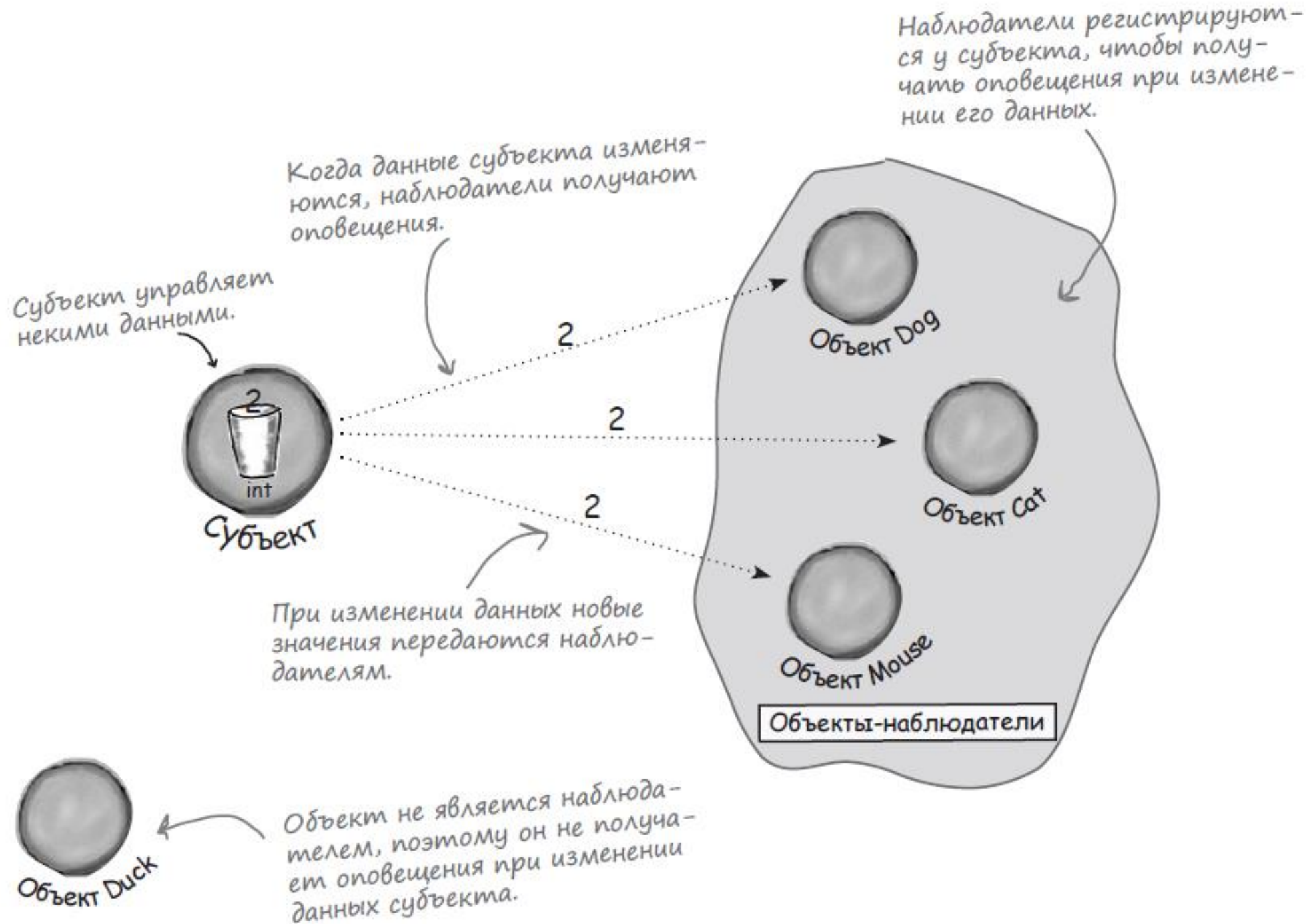
```
currentConditionsDisplay.update(temp, humidity, pressure);  
statisticsDisplay.update(temp, humidity, pressure);  
forecastDisplay.update(temp, humidity, pressure);
```

Программируя на уровне конкретной реализации, мы не сможем добавлять и удалять визуальные элементы без внесения изменений в программу.

Нечто похожее на общий интерфейс взаимодействия с экранными элементами... Каждый элемент имеет метод `update()`, которому передаются значения температуры, влажности и давления.

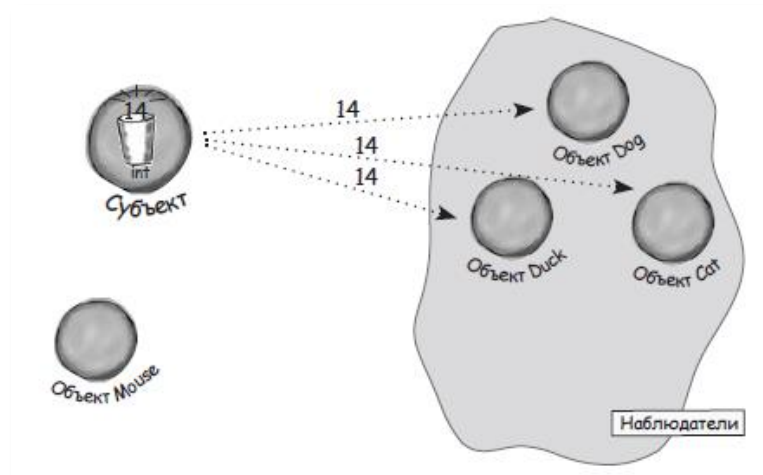
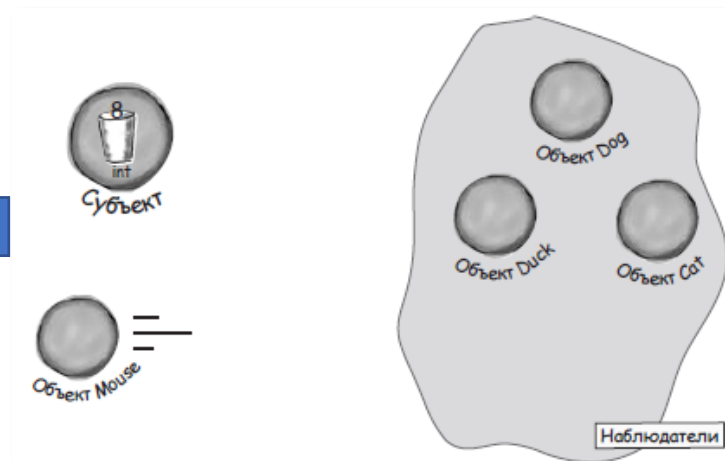
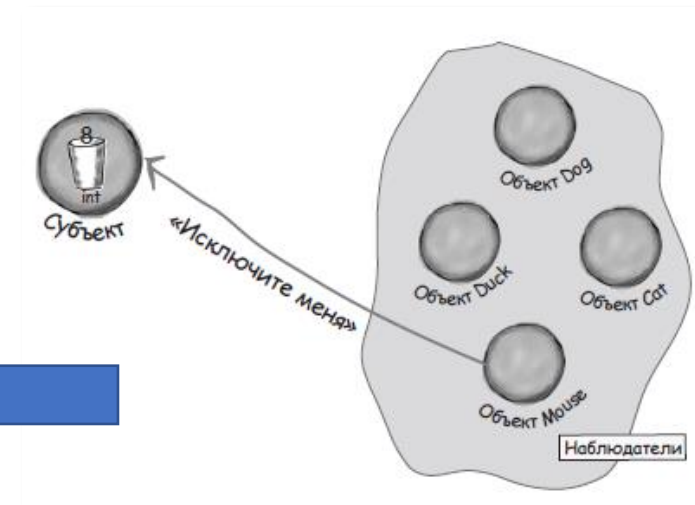
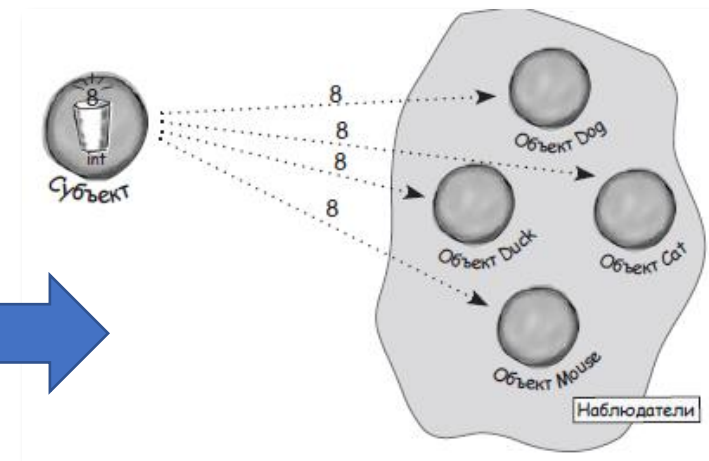
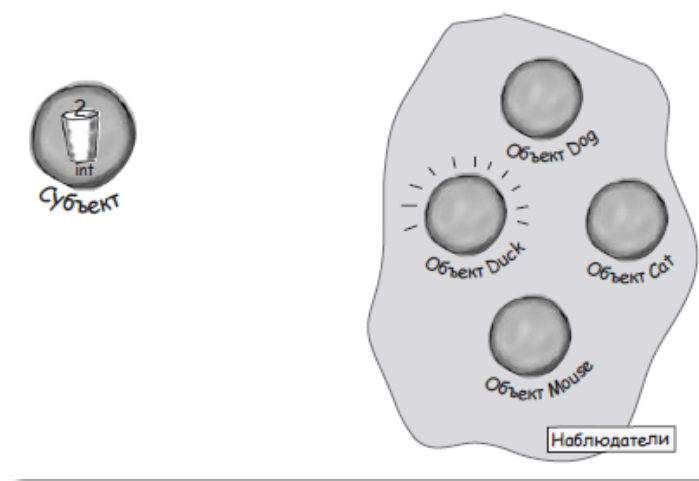
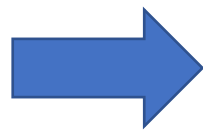
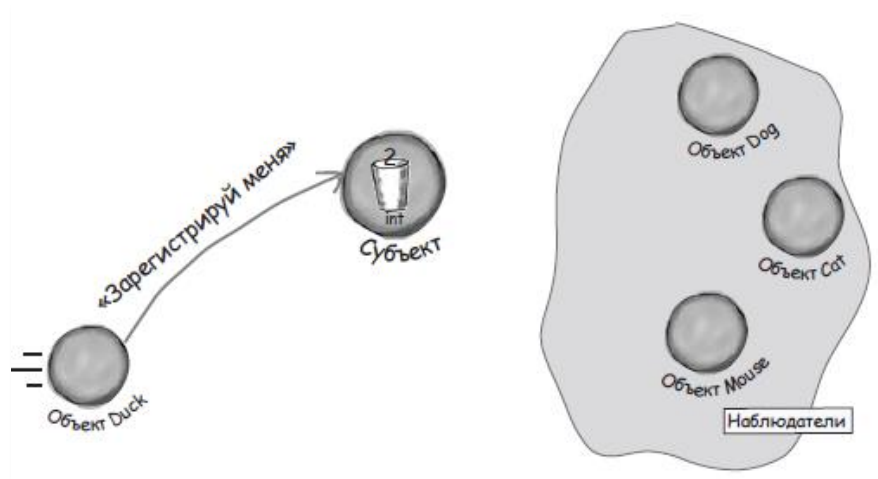
Знакомство с паттерном Наблюдатель

- Всем известно, как работает подписка на газету или журнал
- Если вы понимаете, как работает газетная подписка, вы в значительной мере понимаете и паттерн Наблюдатель — в данном случае издатель называется СУБЪЕКТОМ, а подписчики — НАБЛЮДАТЕЛЯМИ.



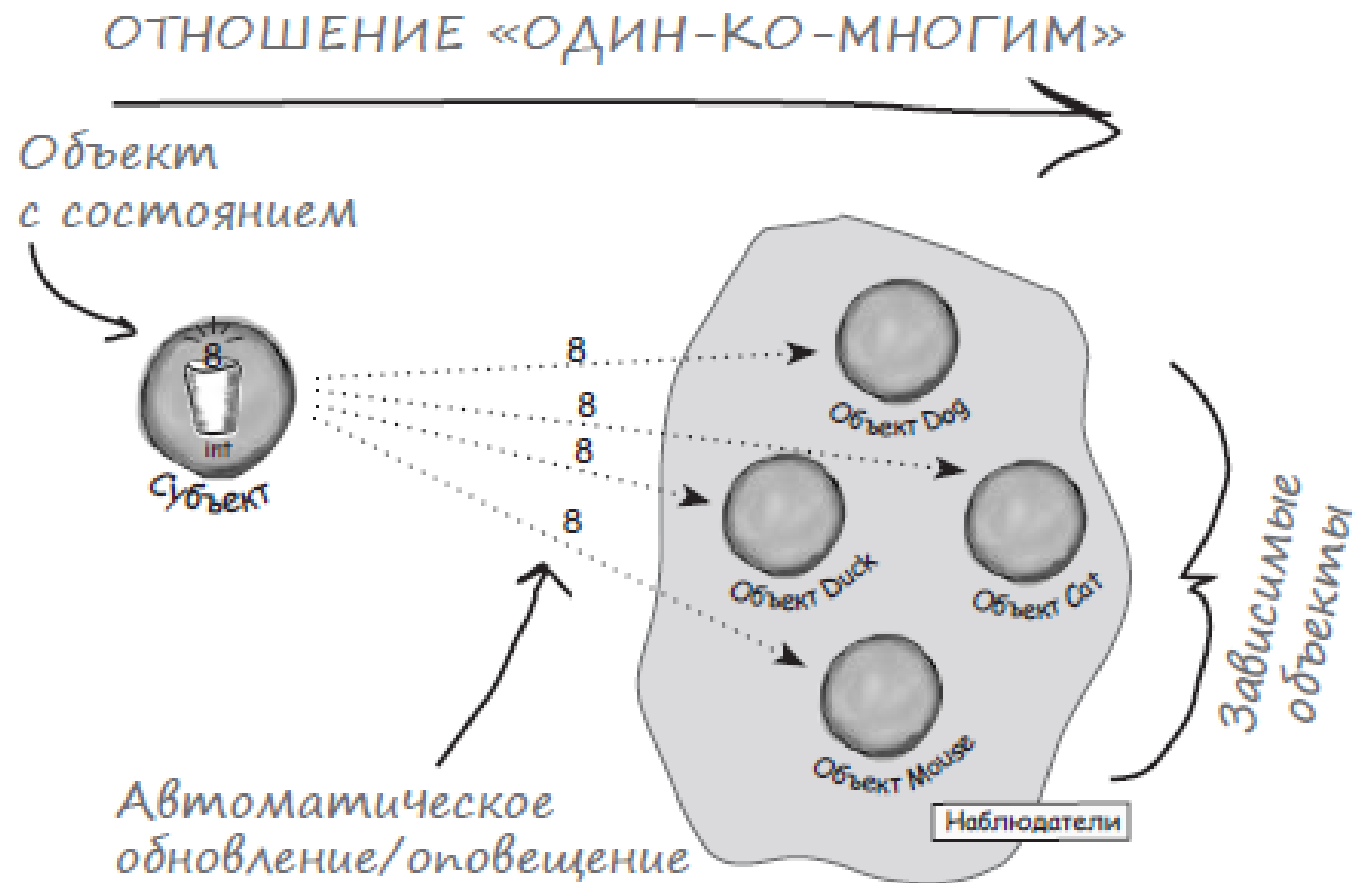
Как работает паттерн Наблюдатель

- Объект Duck сообщает субъекту, что он хочет стать наблюдателем.
 - Объект Duck хочет быть в курсе дела; эти значения `int`, которые субъект рассылает при изменении состояния, выглядят так интересно...
- Объект Duck стал официальным наблюдателем.
 - Объект Duck включен в список... Теперь он с нетерпением ждет следующего оповещения, с которым он получит интересующее его значение `int`.
- У субъекта появились новые данные!
 - Duck и все остальные наблюдатели оповещаются об изменении состояния субъекта.
- Объект Mouse требует исключить его из числа наблюдателей.
 - Объекту Mouse надоело получать оповещения, и он решил, что пришло время выйти из числа наблюдателей.
- Mouse уходит!
 - Субъект принимает запрос объекта Mouse и исключает его из числа наблюдателей.
- У субъекта появилось новое значение `int`.
 - Все наблюдатели получают очередное оповещение — кроме объекта Mouse, который исключен из списка. Не говорите никому, но он тайно скучает по этим оповещениям... и, возможно, когда-нибудь снова войдет в число наблюдателей.



Определение паттерна Наблюдатель

- Паттерн Наблюдатель определяет отношение «один-ко-многим» между объектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых объектов.

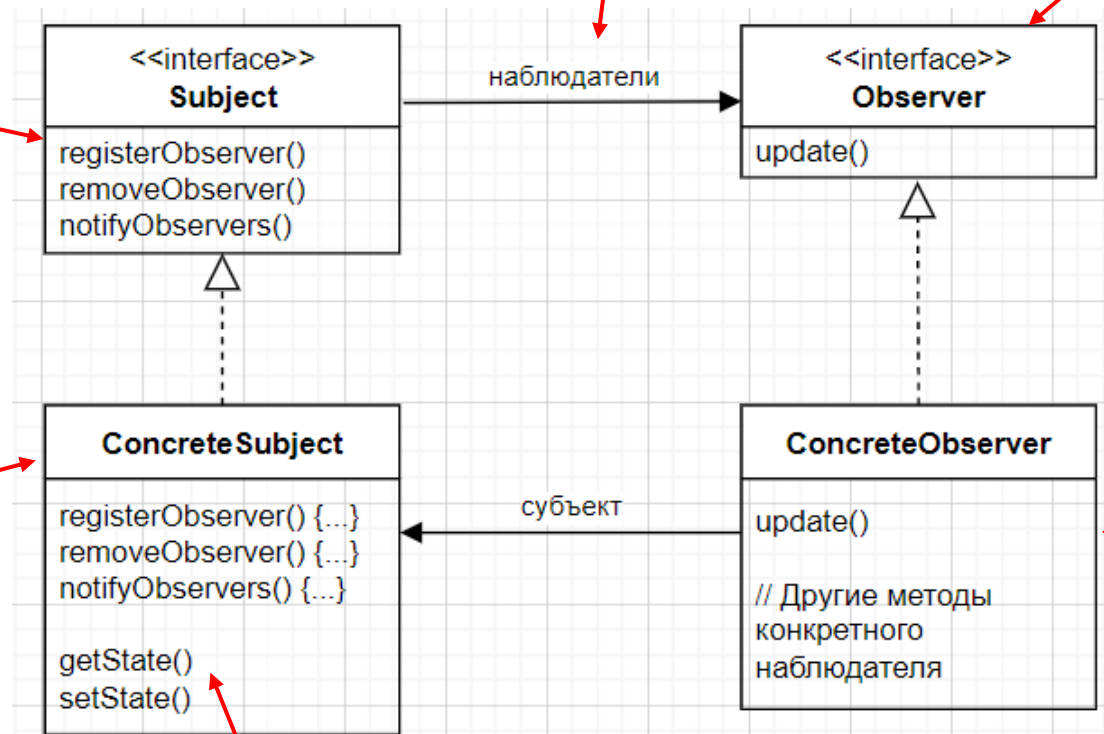


Паттерн Наблюдатель: диаграмма классов

Интерфейс субъекта.
Используется объектами
для регистрации в качестве
наблюдателя, а также
исключения из списка.

Каждый субъект
может иметь много
наблюдателей.

Каждый потенциальный на-
блюдатель должен
реализовать
интерфейс Observer. Интер-
фейс содержит единствен-
ный метод update(), который
вызывается при изменении
состояния субъекта.



Субъект реализует
интерфейс Subject. Кроме
методов регистрации и
исключения, субъект также
реализует метод
`notifyObservers()`,
оповещающий всех текущих
наблюдателей об
изменении состояния.

Субъект также может иметь
get- и set-методы для
изменения состояния

Наблюдатели могут относиться к
любому классу, реализующему
интерфейс Observer. Каждый
наблюдатель регистриру ется у
конкретного субъекта для
получения обновлений.

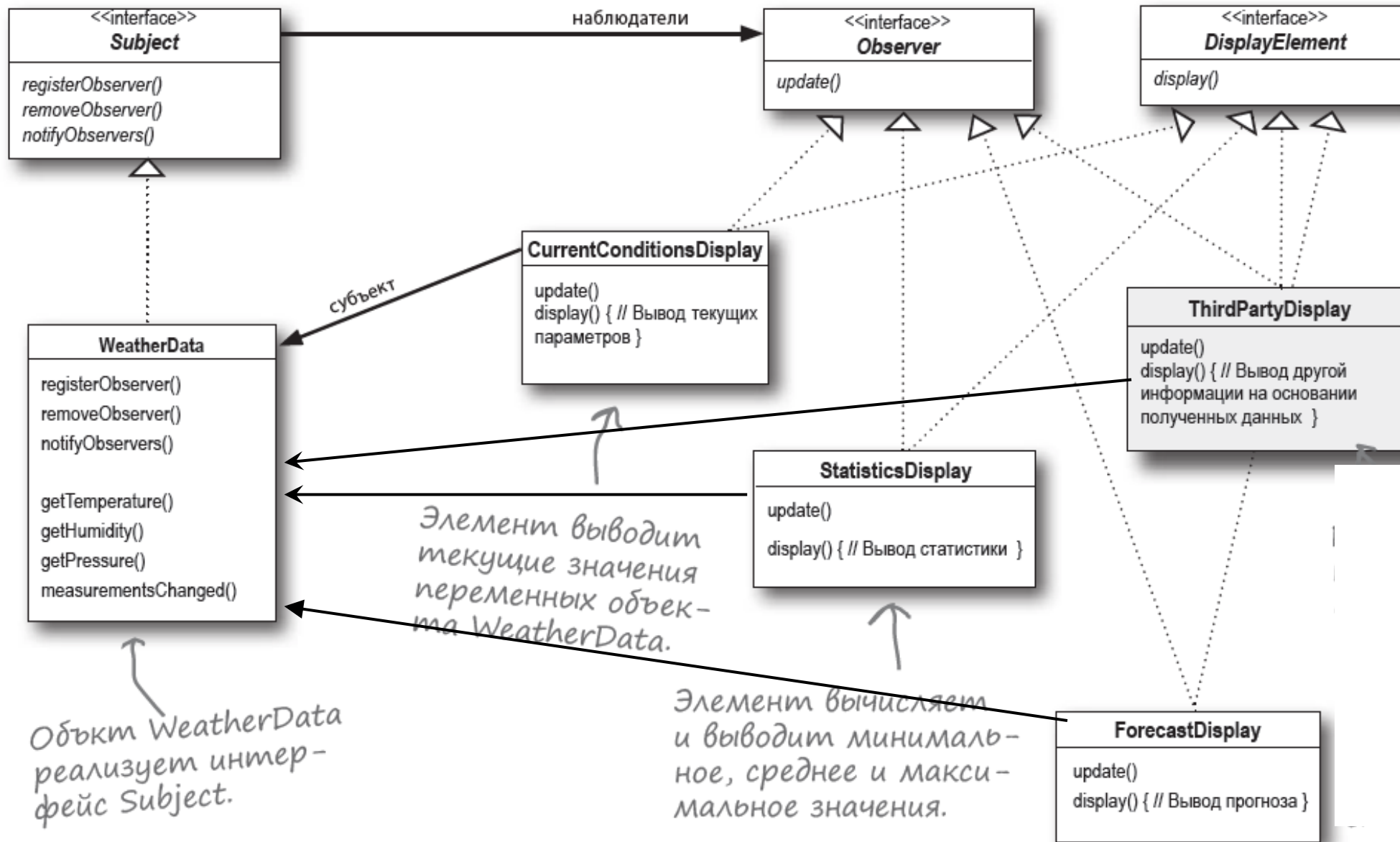
Слабые связи

- Если два объекта могут взаимодействовать, не обладая практически никакой информацией друг о друге, такие объекты называют слабосвязанными.
- В архитектуре паттерна Наблюдатель между субъектами и наблюдателями существует слабая связь.
 - Единственное, что знает субъект о наблюдателе, — то, что тот реализует некоторый интерфейс
 - Новые наблюдатели могут добавляться в любой момент
 - Добавление новых типов наблюдателей не требует модификации субъекта
 - Субъекты и наблюдатели могут повторно использоваться независимо друг от друга
 - Изменения в субъекте или наблюдателе не влияют на другую сторону

Принцип проектирования

- Стремитесь к слабой связанности взаимодействующих объектов.
- На базе слабосвязанных архитектур строятся гибкие ОО-системы, которые хорошо адаптируются к изменениям благодаря минимальным зависимостям между объектами

Проектирование Weather Station



Разработчики реализуют интерфейсы **Observer** и **Display** для создания собственных визуальных элементов.

Реализация Weather Station

```
public interface Subject {  
    public void registerObserver(Observer o);  
    public void removeObserver(Observer o);  
    public void notifyObservers();  
}
```

Оба метода получают в аргументе реализацию *Observer* (регистрируемый или исключаемый наблюдатель).

Этот метод вызывается для оповещения наблюдателей об изменении состояния субъекта.

```
public interface Observer {  
    public void update(float temp, float humidity, float pressure);  
}
```

Данные состояния, передаваемые наблюдателям при изменении состояния субъекта.

Интерфейс *Observer* реализуется всеми наблюдателями; таким образом, все наблюдатели должны реализовать метод *update()*.

```
public interface DisplayElement {  
    public void display();  
}
```

Интерфейс *DisplayElement* содержит всего один метод *display()*, который вызывается для отображения визуального элемента.

Реализация интерфейса Subject в WeatherData

```
public class WeatherData implements Subject {  
    private ArrayList observers;  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherData() {  
        observers = new ArrayList();  
    }  
}
```

Теперь WeatherData реализует интерфейс Subject.

Добавляем контейнер ArrayList для хранения наблюдателей и создаем его в конструкторе.

Реализация интерфейса Subject в WeatherData

Реализация интерфейса Subject.

```
public void registerObserver(Observer o) {  
    observers.add(o);  
}
```

← Новые наблюдатели просто добавляются в конец списка.

```
public void removeObserver(Observer o) {  
    int i = observers.indexOf(o);  
    if (i >= 0) {  
        observers.remove(i);  
    }  
}
```

← Если наблюдатель хочет отменить регистрацию, мы просто удаляем его из списка.

```
public void notifyObservers() {  
    for (int i = 0; i < observers.size(); i++) {  
        Observer observer = (Observer)observers.get(i);  
        observer.update(temperature, humidity, pressure);  
    }  
}
```


← Самое интересное: оповещение наблюдателей об изменении состояния через метод update(), реализуемый всеми наблюдателями.

```
public void measurementsChanged() {  
    notifyObservers();  
}
```

← Оповещение наблюдателей о появлении новых данных.

Реализация интерфейса Subject в WeatherData

```
public void setMeasurements(float temperature, float humidity, float pressure) {  
    this.temperature = temperature;  
    this.humidity = humidity;  
    this.pressure = pressure;  
    measurementsChanged();  
}  
  
// Другие методы WeatherData  
}
```



Приложить метеостанцию к каждому экземпляру книги нам не разрешили, поэтому вместо чтения данных с устройства мы воспользуемся тестовым методом. При желании вы можете написать код для загрузки погодных данных из Интернета.

Визуальные элементы

Элемент реализует Observer, чтобы получать данные от объекта WeatherData.

Также он реализует интерфейс DisplayElement, как и все визуальные элементы в нашем API.

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {  
    private float temperature;  
    private float humidity;  
    private Subject weatherData;
```

```
    public CurrentConditionsDisplay(Subject weatherData) {  
        this.weatherData = weatherData;  
        weatherData.registerObserver(this);  
    }
```

Конструктору передается объект WeatherData, который используется для регистрации элемента в качестве наблюдателя.

```
    public void update(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        display();  
    }
```

При вызове update() мы сохраняем значения температуры и влажности, после чего вызываем display().

```
    public void display() {  
        System.out.println("Current conditions: " + temperature  
            + "F degrees and " + humidity + "% humidity");  
    }  
}
```

Метод display() просто выводит текущие значения температуры и влажности.

Тестирование Weather Station

```
# Моделирование
```

```
def main():  
    weatherData = WeatherData()  
    currentDisplay = CurrentConditionsDisplay(weatherData)  
    statisticsDisplay = StatisticsDisplay(weatherData)  
    forecastDisplay = ForecastDisplay(weatherData)  
    heatIndexDisplay = HeatIndexDisplay(weatherData)  
  
    weatherData.setMeasurements(80, 65, 30.4)  
    weatherData.setMeasurements(82, 70, 29.2)  
    weatherData.setMeasurements(78, 90, 29.2)  
  
if __name__ == '__main__':  
    main()
```

Тестирование Weather Station

>>>

Current conditions:

80 F degrees, 65 percent of humidity and atmospheric pressure 30.4 mm of mercury column
Avg/Max/Min temperature = 80.0 / 80 / 80

Forecast:

Improving weather on the way!

Heat index is 82.95535063710001

Current conditions:

82 F degrees, 70 percent of humidity and atmospheric pressure 29.2 mm of mercury column
Avg/Max/Min temperature = 81.0 / 82 / 80

Forecast:

Watch out for cooler, rainy weather

Heat index is 86.90123306385205

Current conditions:

78 F degrees, 90 percent of humidity and atmospheric pressure 29.2 mm of mercury column
Avg/Max/Min temperature = 80.0 / 82 / 78

Forecast:

More of the same

Heat index is 83.64967139559604