Паттерны проектирования

Наумов Д.А., доц. каф. КТ

Основы программной инженерии, 2019

Содержание лекции

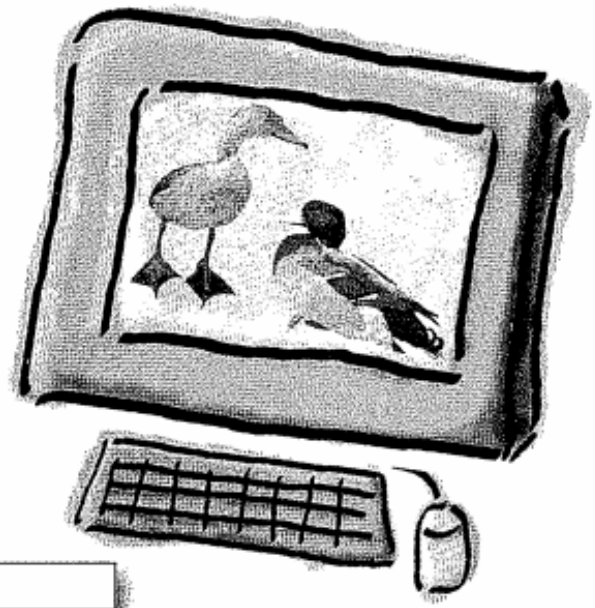
Паттерны проектирования

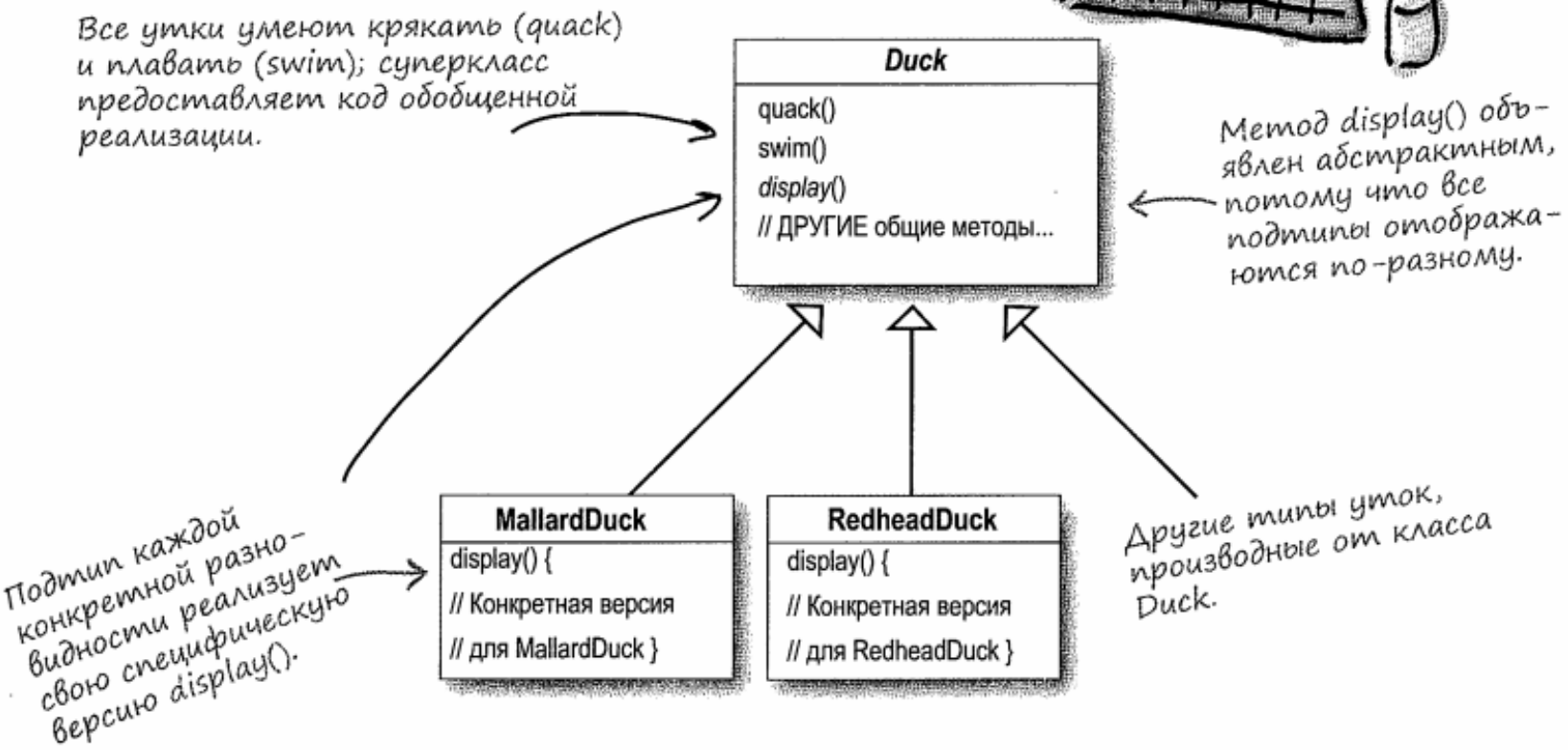
Эрик Фримен
Элизабет Фримен

при участии
Кэтти Сьерра
и Берта Бейтса

Все началось с простого приложения SimUDuck

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.





Теперь утки будут ЛЕТАТЬ

Начальство решило, что летающие утки – именно та «изюминка», которая сокрушит всех конкурентов. И конечно, начальство Джо пообещало, что Джо легко соорудит что-нибудь этакое в течение недели. «В конце концов, он ООП-программист... *Какие могут быть трудности?*»

Но тут все пошло наперекосяк ...

Джо, я на собрании акционеров. В демо-версии по экрану летают **резиновые утки**. Это что, шутка такая? На премию можешь не рассчитывать...

Что произошло?

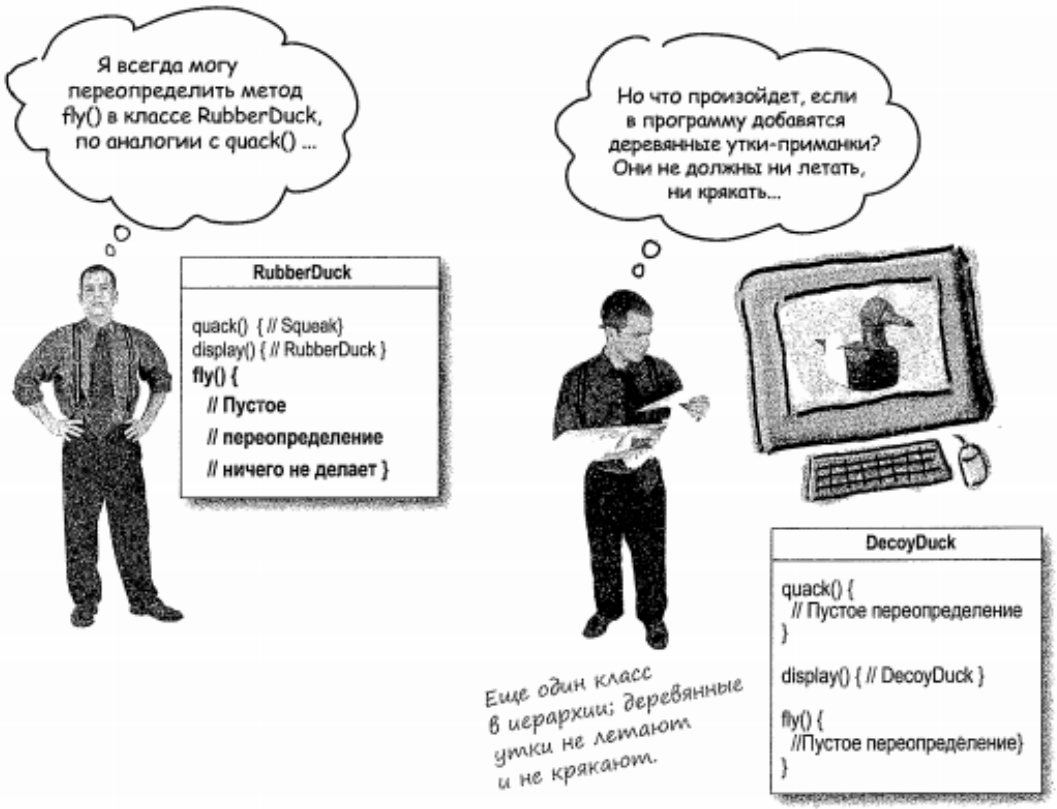
Джо не сообразил, что *летать* должны *не все* субклассы Duck. Новое поведение, добавленное в суперкласс Duck, оказалось *неподходящим* для некоторых субклассов. И теперь в программе начали летать неодушевленные объекты.

Локальное изменение кода привело к нелокальному побочному эффекту (летающие резиновые утки!)

В моей иерархии есть небольшой просчет. А вообще симпатично получилось... Может, сделать вид, что так и было задумано?

Казалось бы, в этой ситуации наследование идеально подходит для повторного использования кода — но с сопровождением возникают проблемы.

Джо думает о наследовании...



Упражнение

Возьми в руку карандаш

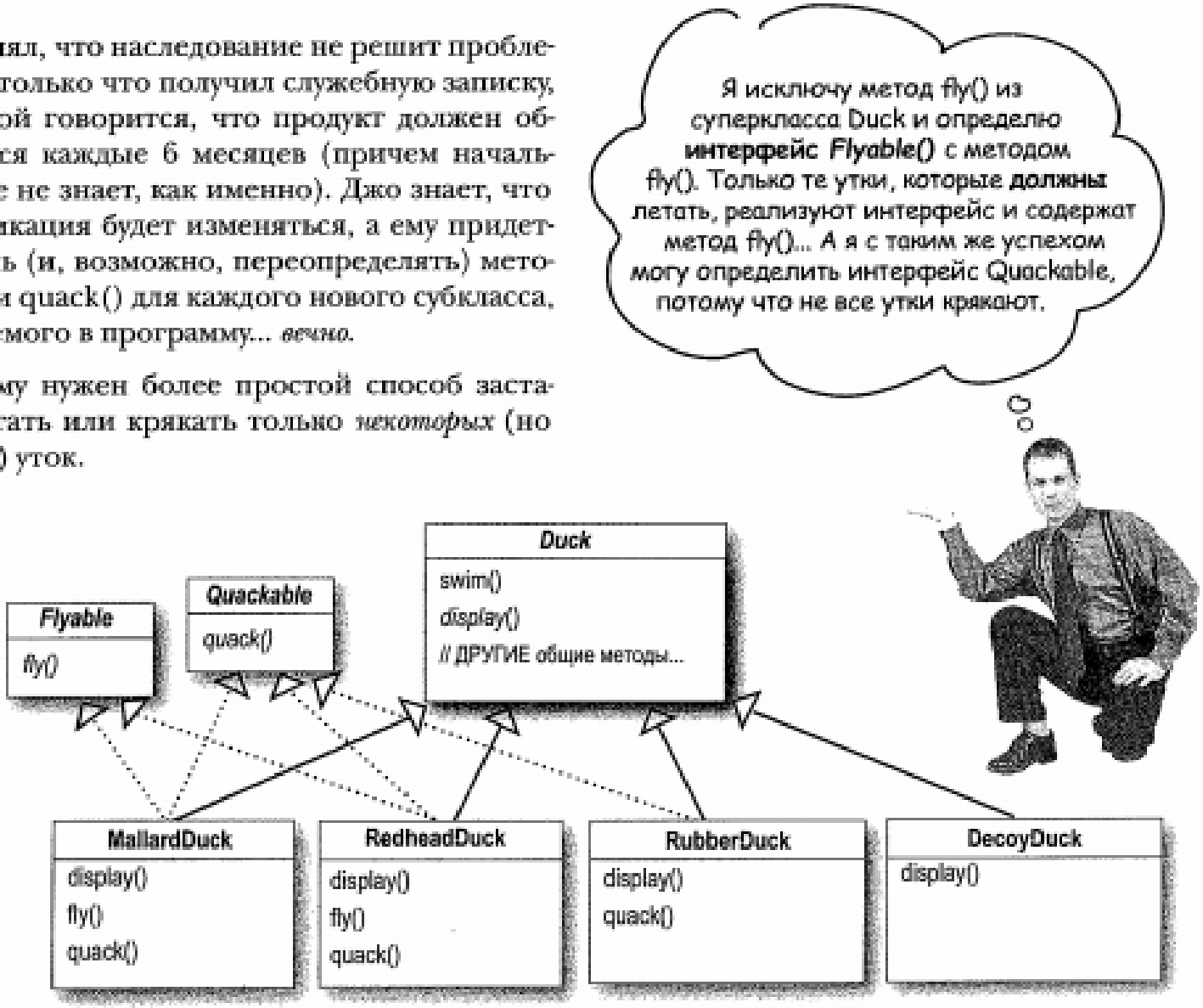
Какие из перечисленных недостатков относятся к применению *наследования* для реализации Duck? (Укажите все варианты.)

- ❑ A. Дублирование кода в субклассах.
- ❑ 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?

Джо понял, что наследование не решит проблему — он только что получил служебную записку, в которой говорится, что продукт должен обновляться каждые 6 месяцев (причем начальство еще не знает, как именно). Джо знает, что спецификация будет изменяться, а ему придется искать (и, возможно, переопределять) методы fly() и quack() для каждого нового субкласса, включаемого в программу... *вечно.*

Итак, ему нужен более простой способ заставить летать или крякать только *некоторых* (но не всех!) уток.



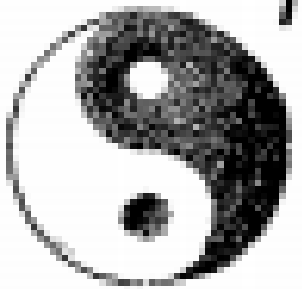
Упражнение

Возьми в руку карандаш

Изменения могут быть обусловлены многими факторами. Укажите некоторые причины для изменения кода в приложениях (чтобы вам было проще, мы привели пару примеров).

Клиенты или пользователи требуют реализации новой или расширенной функциональности.

Компания переходит на другую СУБД, а данные будут приобретаться у другого поставщика в новом формате. Ужас!



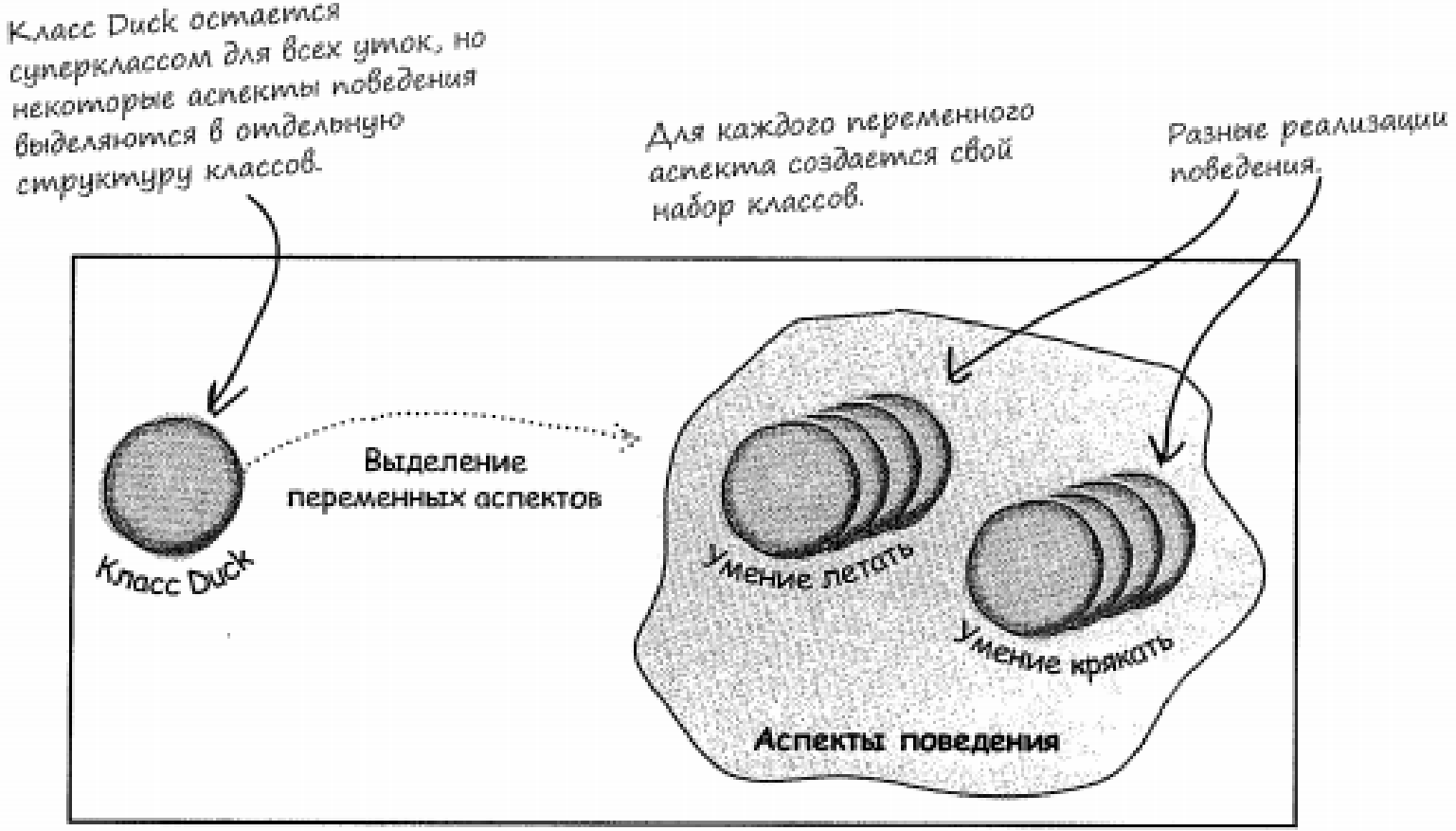
Принцип проектирования

Выделите аспекты приложения, которые могут изменяться, и отделите их от тех, которые всегда остаются постоянными.

Первый из многих принципов проектирования, которые встречаются в этой книге.

Мы знаем, что fly() и quack() — части класса Duck, изменяющиеся в зависимости от субкласса.

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.

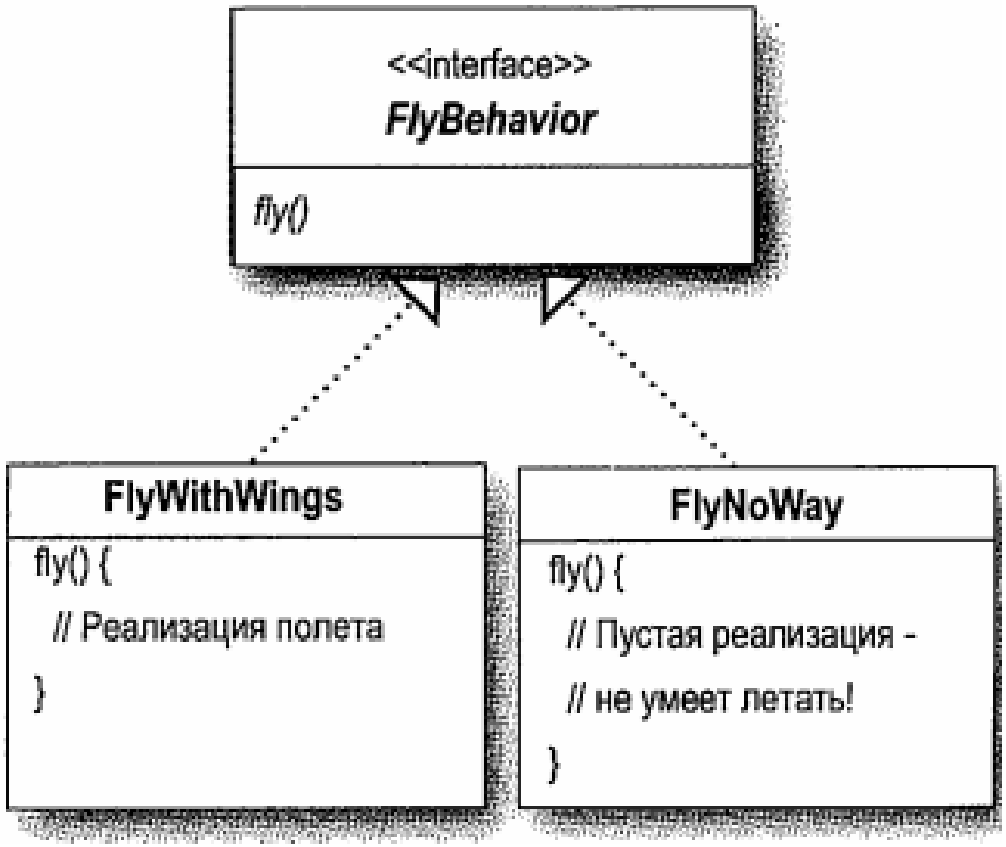


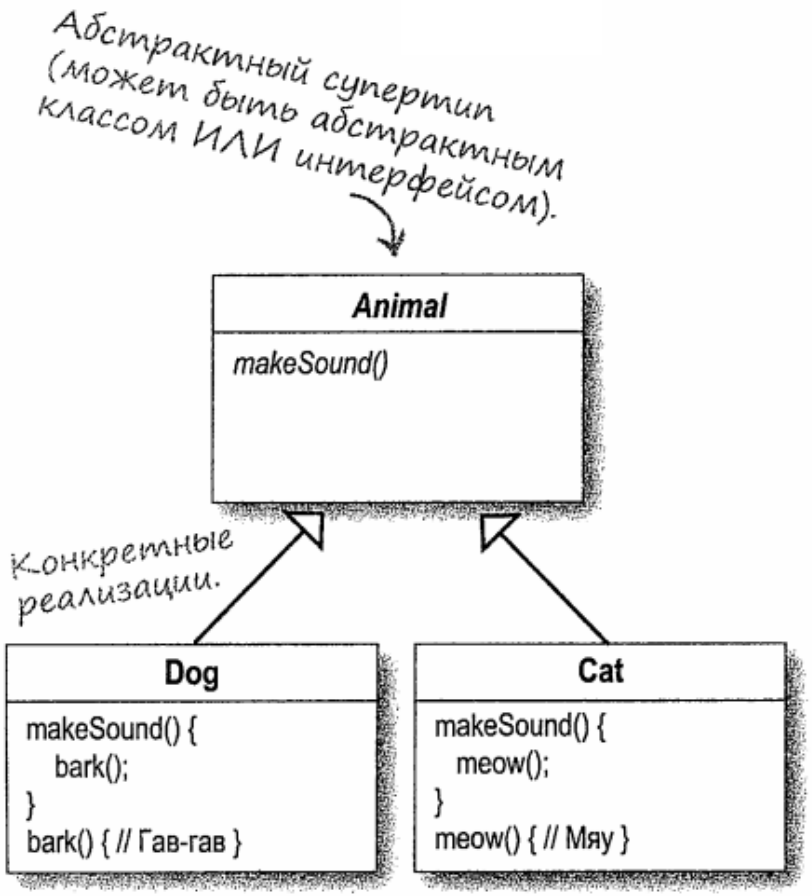
Проектирование переменного поведения

Как же спроектировать набор классов, реализующих переменные аспекты поведения?

Отныне аспекты поведения Duck будут находиться в отдельных классах, реализующих интефейс конкретного аспекта.

В этом случае классам Duck не нужно знать подробности реализации своих аспектов поведения.





Программирование на уровне реализации выглядит так:

```
Dog d = new Dog();
d.bark();
```

Объявление «d» с типом Dog требует программирования на уровне конкретной реализации Animal.

Программирование на уровне интерфейса/супертипа:

```
Animal animal = new Dog();
animal.makeSound();
```

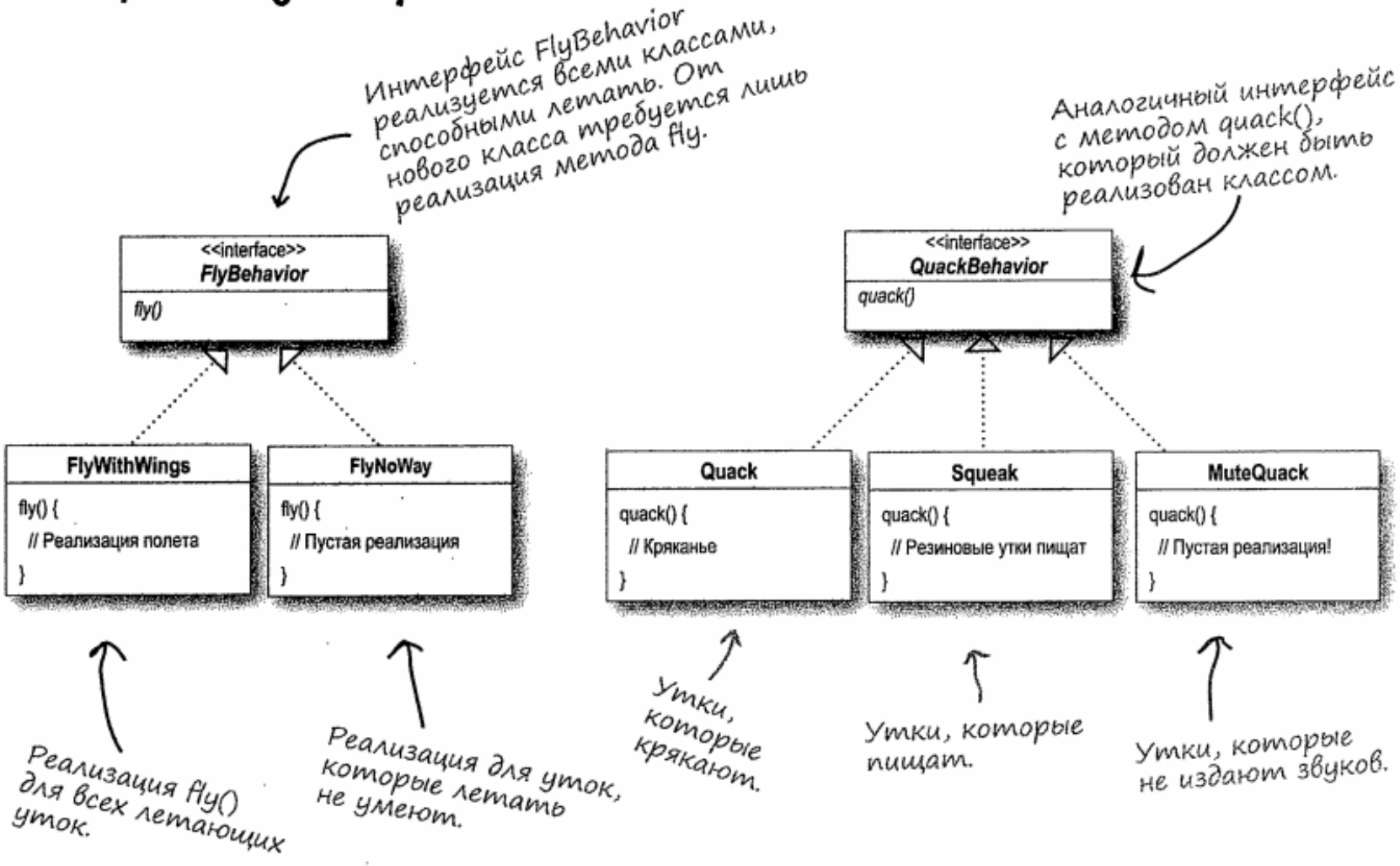
Полиморфное использование ссылки.

Или еще лучше, вместо жесткой фиксации подтипа в коде (new Dog()), **объект конкретной реализации присваивается во время выполнения:**

```
a = getAnimal();
a.makeSound();
```

Фактический подтип Animal неизвестен... Важно лишь то, что он умеет реагировать на makeSound().

Реализация поведения уток

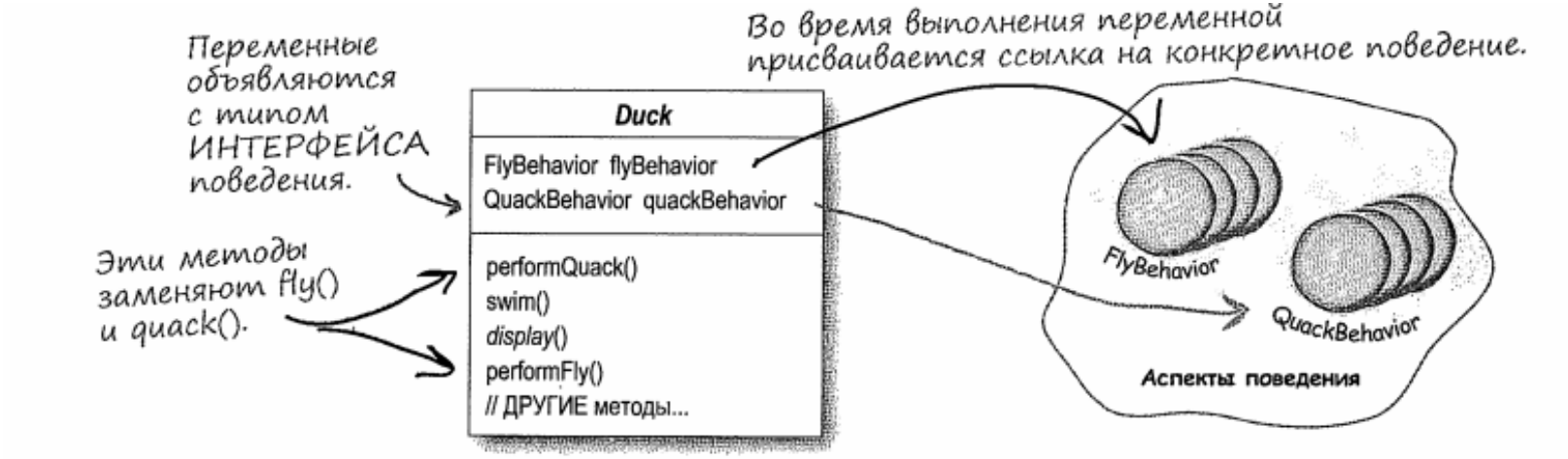


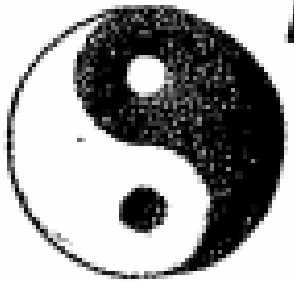
Упражнение

Возьми в руку карандаш

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?

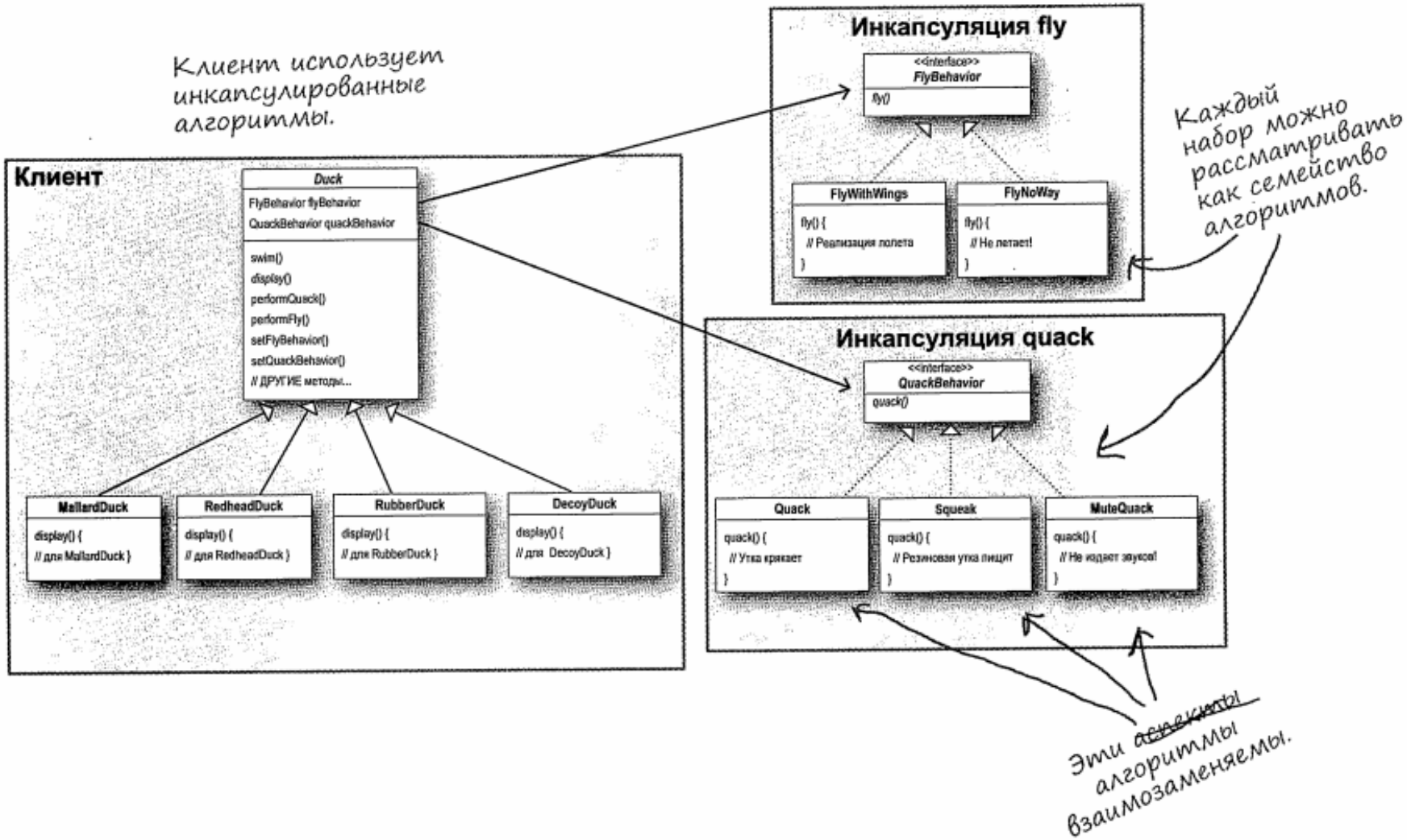




Принцип проектирования

Отдавайте предпочтение композиции перед наследованием.

Упражнение


Клиент использует инкапсулированные алгоритмы.
Клиент
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// ДРУГИЕ методы...
MallardDuck
display() {
// для MallardDuck }
RedheadDuck
display() {
// для RedheadDuck }
RubberDuck
display() {
// для RubberDuck }
DecoyDuck
display() {
// для DecoyDuck }
Инкапсуляция fly
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// Реализация полета
}
FlyNoWay
fly() {
// Не летает!
}
Каждый набор можно рассматривать как семейство алгоритмов.
Инкапсуляция quack
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// Утка крякает
}
Squeak
quack() {
// Резиновая утка пищит
}
MuteQuack
quack() {
// Не издает звуков!
}
Эти алгоритмы взаимозаменяемы.