O'REILLY®

Второе издание

Head First

Паттерны проектирования

Легко масштабировать
Легко поддерживать

Эрик Фримен
Элизабет Робсон
Кэти Сьерра и Берт Бейтс

ПОДАРОК ДЛЯ МОЗГА

ББК 32.973.2-018-02
УДК 004.42
Х99

Фримен Эрик, Робсон Элизабет, Сьерра Кэти, Бейтс Берт
Х99 Head First. Паттерны проектирования. 2-е изд. — СПб.: Питер, 2022. — 640 с.: ил. — (Серия «Head First O'Reilly»).

ISBN 978-5-4461-1819-9

Не имеет смысла каждый раз изобретать велосипед, лучше сразу освоить приемы проектирования, которые уже созданы людьми, сталкивавшимися с аналогичными задачами. В этой книге рассказано, какие паттерны действительно важны, когда и при каких условиях ими необходимо пользоваться, как применить их в ваших проектах и на каких принципах объектно-ориентированного проектирования они построены. Присоединяйтесь к сотням тысяч разработчиков, которые повысили свою квалификацию объектно-ориентированного проектирования благодаря книге «Head First. Паттерны проектирования».

Если вы уже читали книги из серии Head First, то знаете, что вас ждет визуально насыщенный формат, разработанный с учетом особенностей работы мозга. В книге «Head First. Паттерны проектирования» принципы и паттерны проектирования представлены так, чтобы вы не заснули, читая книгу, научились решать реальные задачи проектирования программных продуктов и общаться на языке паттернов с другими участниками вашей команды.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018-02
УДК 004.42

ISBN 978-1492078005 англ.

ISBN 978-5-4461-1819-9

Authorized Russian translation of the English edition of Head First Design Patterns 2E
ISBN 9781492078005 © 2020 Eric Freeman & Elisabeth Robson
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Содержание (сводка)

Содержание (настоящее)

Введение

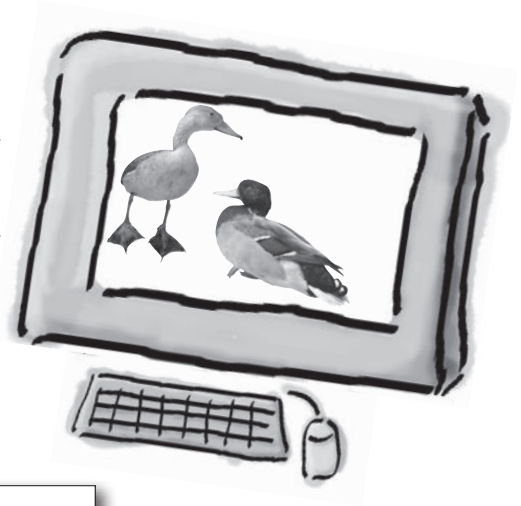
Настройте свой мозг на дизайн паттернов. Вот что вам понадобится, когда вы пытаетесь что-то выучить, в то время как ваш мозг не хочет воспринимать информацию. Ваш мозг считает: «Лучше уж я подумаю о более важных вещах, например об опасных диких животных или почему нельзя голышом прокатиться на сноуборде». Как же заставить свой мозг думать, что ваша жизнь зависит от овладения дизайном паттернов?

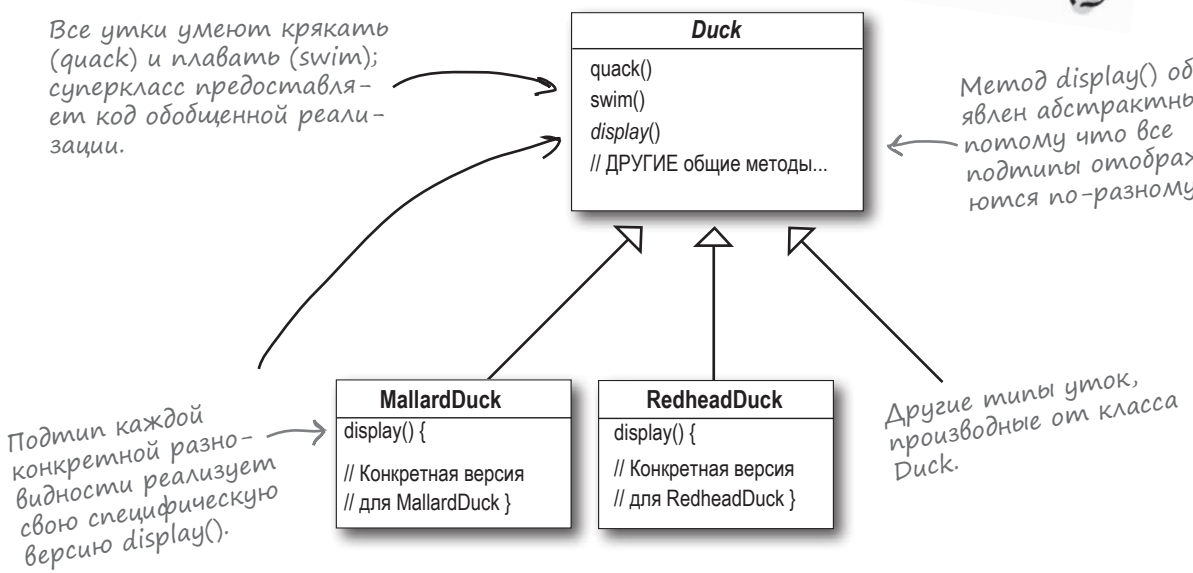
Добро пожаловать в мир паттернов

Наверняка вашу задачу кто-то уже решал. В этой главе вы узнаете, почему (и как) следует использовать опыт других разработчиков, которые уже сталкивались с аналогичной задачей и успешно решили ее. Заодно мы поговорим об использовании и преимуществах паттернов проектирования, познакомимся с ключевыми принципами объектно-ориентированного (ОО) проектирования и разберем пример одного из паттернов. Лучший способ использовать паттерны — *запомнить их*, а затем научиться *распознавать* те места ваших архитектур и существующих приложений, где их уместно *применить*. Таким образом, вместо программного кода вы повторно используете чужой *опыт*.

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.





За последний год компания испытывала жесткое давление со стороны конкурентов. Через неделю долгих выездных совещаний за игрой в гольф 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*.

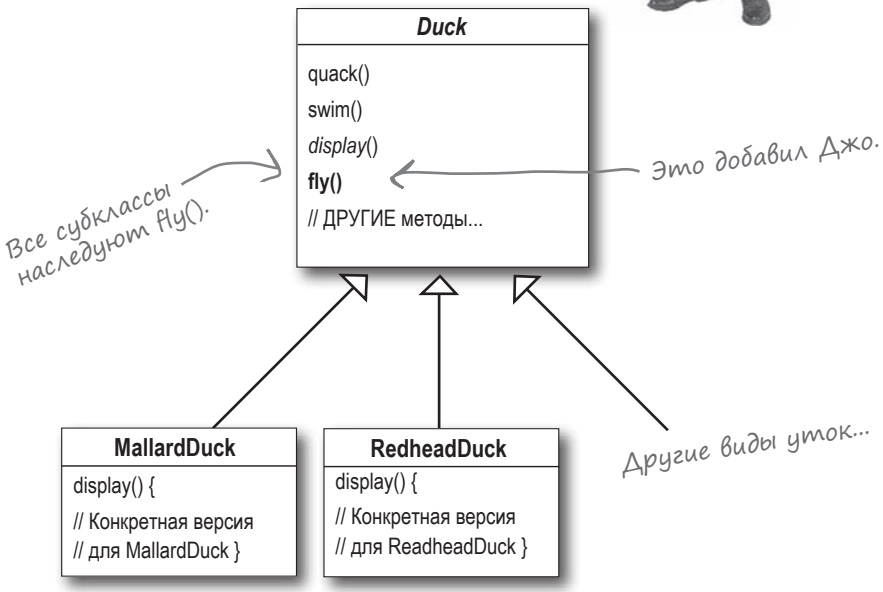
Теперь утки будут ЛЕТАТЬ

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

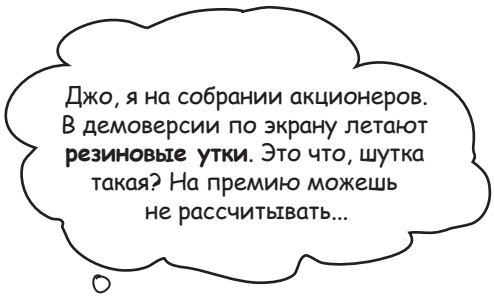
Чего мы добиваемся.

Я добавлю метод fly() в класс Duck, и он будет унаследован всеми производными классами. Пора продемонстрировать мои таланты в области ООП.

Джо



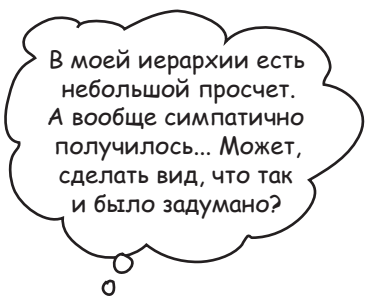
Но тут все пошло наперекосяк...



Что произошло?

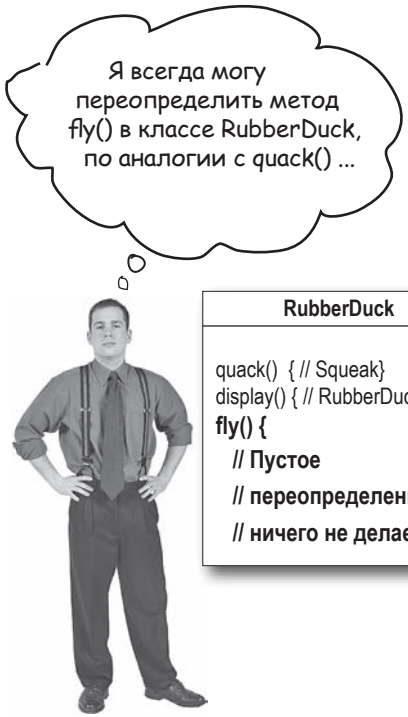
Джо не сообразил, что *летать* должны *не все* субклассы Duck. Новое поведение, добавленное в суперкласс Duck, оказалось *неподходящим* для некоторых субклассов. И теперь в программе начали летать неодушевленные объекты.

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



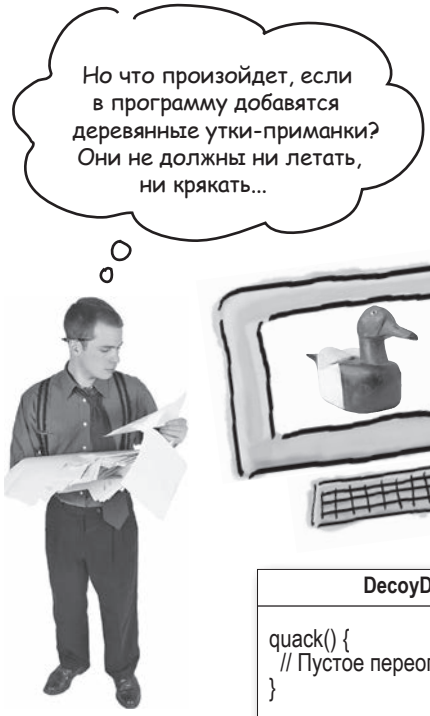
Казалось бы, в этой ситуации наследование идеально подходит для повторного использования кода — но с сопровождением возникают проблемы.

Джо думает о наследовании...



RubberDuck

```
quack()  { // Squeak}
display() { // RubberDuck }
fly() {
  // Пустое
  // переопределение
  // ничего не делает }
```



Еще один класс в иерархии; деревянные утки не летают и не крякают.

DecoyDuck

```
quack() {
  // Пустое переопределение
}

display() { // DecoyDuck }

fly() {
  //Пустое переопределение}
}
```

Возьми в руку карандаш

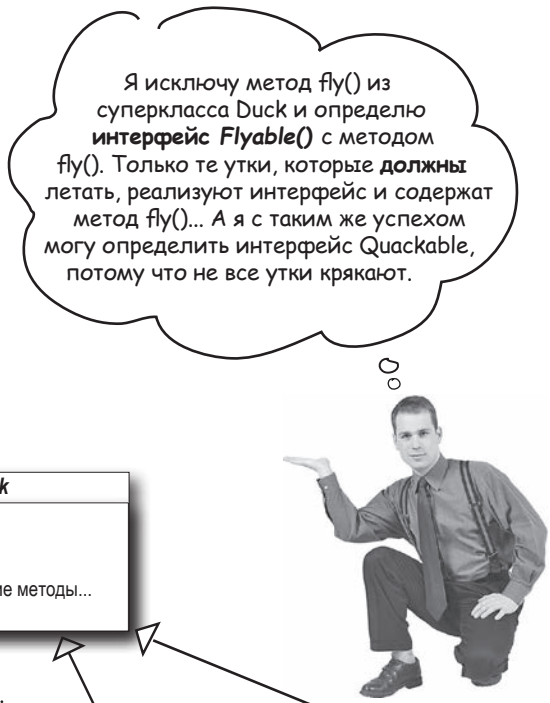
Какие из перечисленных недостатков относятся к применению *наследования* для реализации Duck? (Укажите все варианты.)

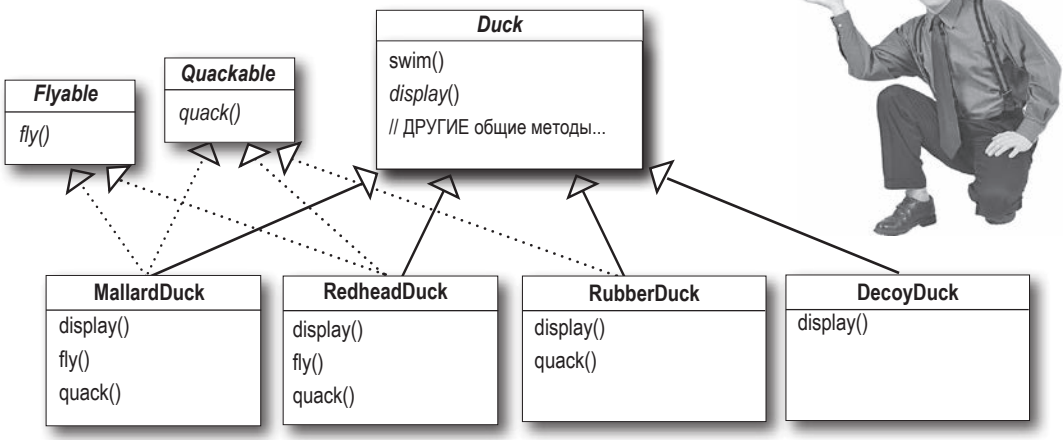
- ❑ A. Дублирование кода в субклассах.
- ❑ B. Трудности с изменением поведения на стадии выполнения.
- ❑ C. Уток нельзя научить танцевать.
- ❑ D. Трудности с получением информации обо всех аспектах поведения уток.
- ❑ E. Утки не могут летать и крякать одновременно.
- ❑ F. Изменения могут оказать непредвиденное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

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





А что ВЫ думаете об этой архитектуре?

По-моему, это самая дурацкая из твоих идей. **Как насчет дублирования кода?** Тебе не хочется переопределять несколько методов, но как тебе понравится вносить маленькое изменение в поведении fly()... во всех 48 «летающих» субклассах Duck?!

А как бы ВЫ поступили на месте Джо?

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 *другой* кошмар из области сопровождения. Не говоря уже о том, что даже *летающие* утки могут летать по-разному...

Вероятно, вы ждете, что сейчас паттерн проектирования явится на белом коне и всех спасет. Но какой интерес в готовом рецепте? Нет, мы самостоятельно вычислим решение, *руководствуясь канонами ОО-проектирования.*

А как было бы хорошо, если бы программу можно было написать так, чтобы вносимые изменения оказывали минимальное влияние на существующий код... Мы тратили бы меньше времени на **переработку** и больше — на всякие интересные вещи...

Единственная константа в программировании

На что всегда можно рассчитывать в ходе работы над проектом?

В какой бы среде, над каким бы проектом, на каком угодно языке вы ни работали — что всегда будет неизменно присутствовать в вашей программе?

RNHƎHƎMƐN

(ответ можно прочитать в зеркале)

Как бы вы ни спроектировали свое приложение, со временем оно должно развиваться и изменяться — иначе оно *умрет*.

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). Сверьтесь с ответами в конце главы, прежде чем двигаться дальше.

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!

Захожу на цель...

Итак, наследование нам не подошло, потому что утиное поведение изменяется в субклассах, а некоторые аспекты поведения присутствуют *не во всех* субклассах. Идея с интерфейсами Flyable и Quackable на первый взгляд выглядит заманчиво, но интерфейсы Java не имеют реализации, что исключает повторное использование кода. И если когда-нибудь потребуется изменить аспект поведения, вам придется искать и изменять его во всех субклассах, где он определяется, – скорее всего, с внесением *новых* ошибок!

К счастью, для подобных ситуаций существует полезный принцип проектирования.

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.

Иначе говоря, если некий аспект кода изменяется (допустим, с введением новых требований), то его необходимо отделить от тех аспектов, которые остаются неизменными.

Другая формулировка того же принципа: ***выделите переменные составляющие и инкапсулируйте их, чтобы позднее их можно было изменять или расширять без воздействия на постоянные составляющие.***

При всей своей простоте эта концепция лежит в основе почти всех паттернов проектирования. *Все паттерны обеспечивают возможность изменения некоторой части системы независимо от других частей.*

Итак, пришло время вывести утиное поведение за пределы классов Duck!

Выделите то, что изменяется, и «инкапсулируйте» эти аспекты, чтобы они не влияли на работу остального кода.

Результат? Меньше непредвиденных последствий от изменения кода, бóльшая гибкость ваших систем!

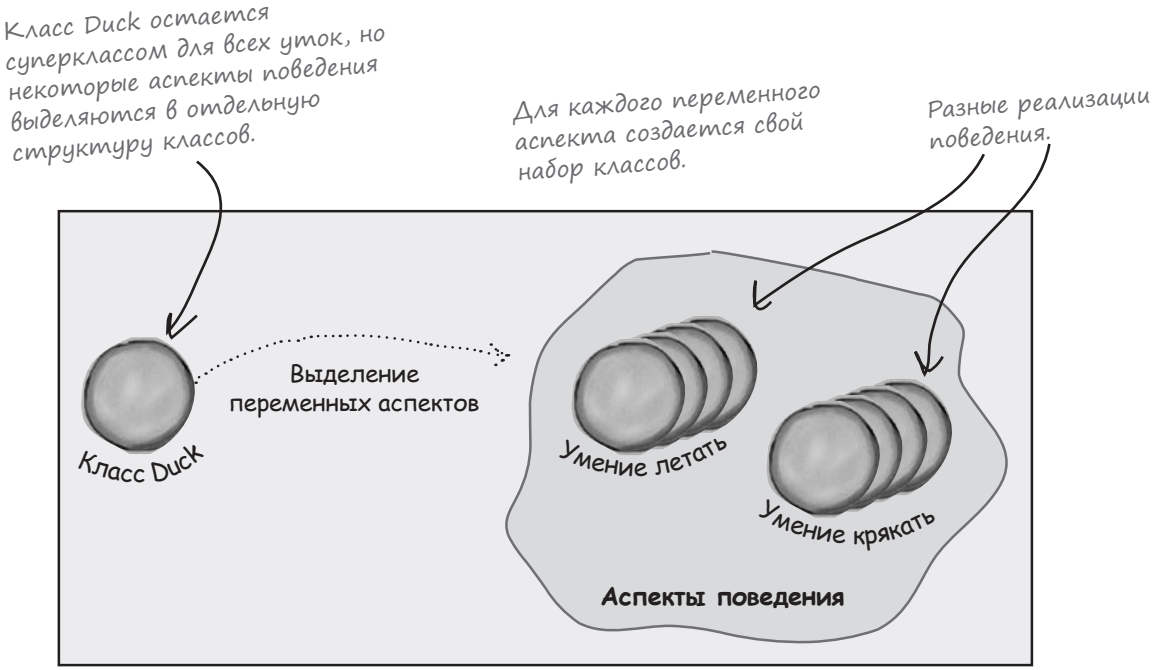
Отделяем переменное от постоянного

С чего начать? Если не считать проблем с fly() и quack(), класс Duck работает хорошо, и другие его аспекты вряд ли будут часто изменяться. Таким образом, если не считать нескольких второстепенных модификаций, класс Duck в целом остается неизменным.

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 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 ?

Нам хотелось бы сохранить максимальную гибкость; в конце концов, все неприятности возникли именно из-за отсутствия гибкости в поведении Duck. Например, желательно иметь возможность создать новый экземпляр MallardDuck и инициализировать его с конкретным *типом* поведения fly(). И раз уж на то пошло, 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.*

Так мы переходим ко второму принципу проектирования.

Принцип проектирования

Программируйте на уровне интерфейса, а не на уровне реализации.

Для представления каждого аспекта поведения (например, FlyBehavior или QuackBehavior) будет использоваться интерфейс, а каждая реализация аспекта поведения будет представлена реализацией этого интерфейса.

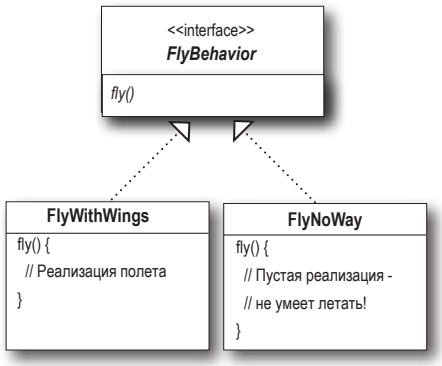
Итак, на этот раз интерфейсы реализуются не классами *Duck*. Вместо этого мы создаем набор классов, единственным смыслом которых является представление некоторого поведения. И теперь интерфейс поведения реализуется *классом поведения*, а не классом Duck.

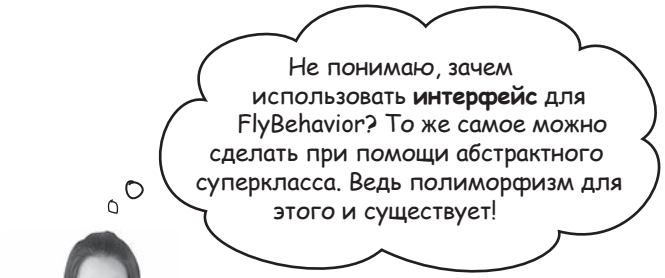
Такой подход отличается от того, что делалось прежде, когда поведение предоставлялось либо конкретной реализацией в суперклассе Duck, либо специализированной реализацией в самом субклассе. В обоих случаях возникала зависимость от *реализации*. Мы были вынуждены использовать именно эту реализацию, и изменить поведение было невозможно (без написания дополнительного кода).

В новом варианте архитектуры субклассы Duck используют поведение, представленное *интерфейсом* (FlyBehavior или QuackBehavior), поэтому фактическая *реализация* этого поведения (то есть конкретное поведение, запрограммированное в классе, реализующем FlyBehavior или QuackBehavior) не привязывается к субклассу Duck.

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

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

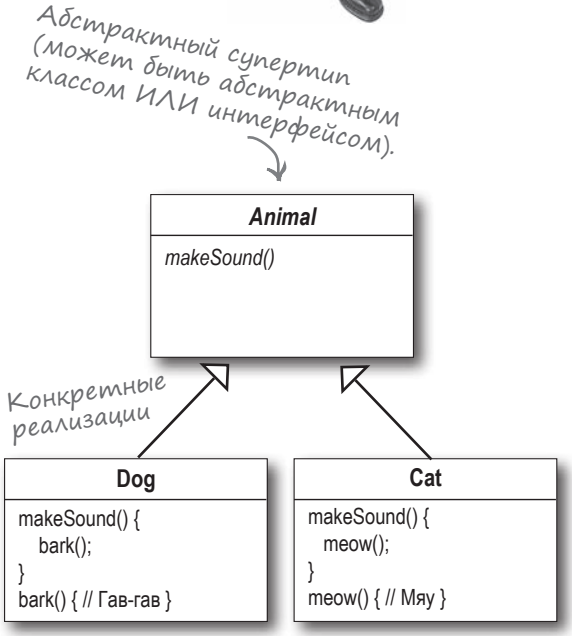




«Интерфейс» в данном случае означает «супертип».

Слово *интерфейс* имеет несколько смыслов. Наряду с *концепцией* интерфейса существует конструкция Java **`interface`**. Программирование на уровне интерфейса может не использовать Java-конструкцию **`interface`**. Собственно, главной целью применения полиморфизма посредством программирования на уровне супертипа является как раз отсутствие жесткой привязки к конкретному объекту во время выполнения. Или, другими словами, «переменные должны объявляться с супертипом (обычно абстрактным классом или интерфейсом), чтобы присваиваемые им объекты могли относиться к любой конкретной реализации супертипа».

Вероятно, вам все это хорошо известно, но просто для того, чтобы убедиться в общности наших представлений, рассмотрим пример использования полиморфного типа. Допустим, имеется абстрактный класс Animal с двумя конкретными реализациями: Dog и Cat.



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.

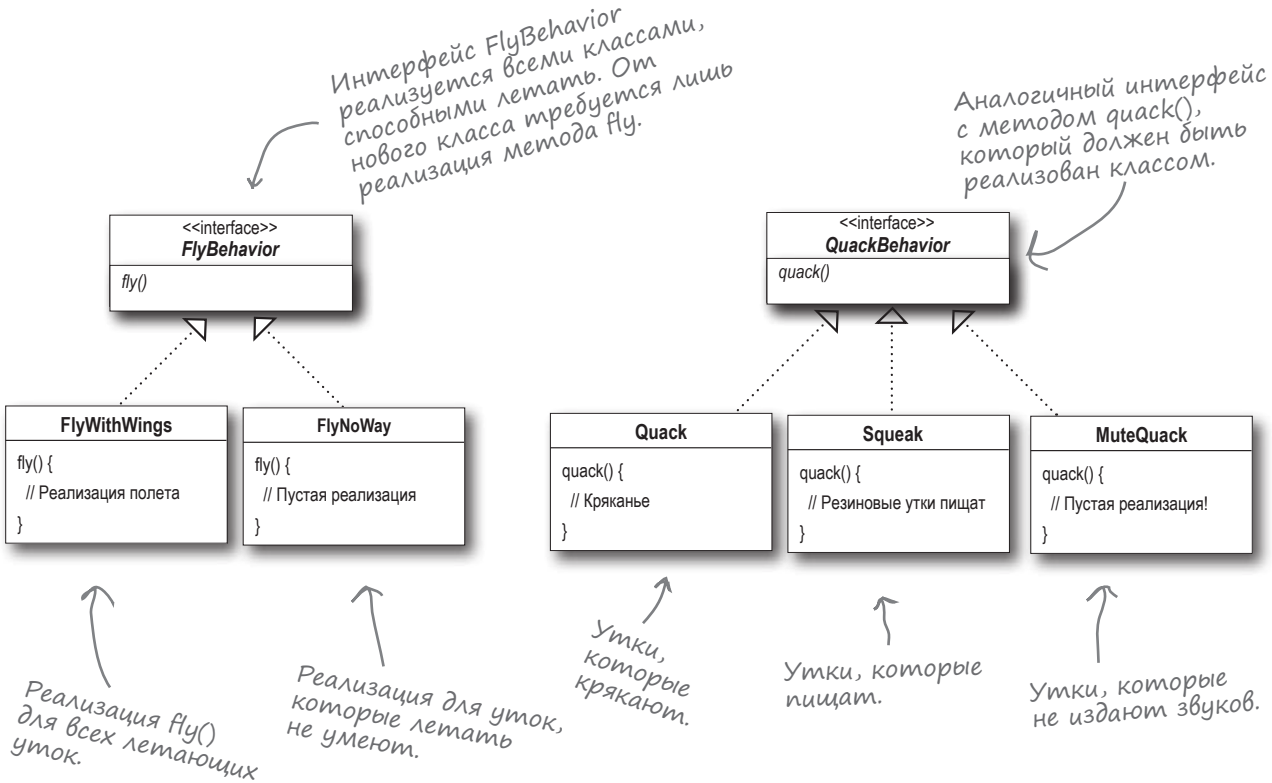
Или еще лучше, вместо жесткой фиксации подтипа в коде (new Dog()), **объект конкретной реализации присваивается во время выполнения**:

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

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

Реализация поведения уток

Интерфейсы FlyBehavior и QuackBehavior вместе с соответствующими классами, реализующими каждое конкретное поведение.



Такая архитектура позволяет использовать поведение fly() и quack() в других типах объектов, потому что это поведение не скрывается в классах Duck!

Кроме того, мы можем добавлять новые аспекты поведения без изменения существующих классов поведения и без последствий для классов Duck, *использующих* существующее поведение.

Все преимущества ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ без недостатков, присущих наследованию!

Часто Задаваемые Вопросы

В: Так я всегда должен сначала реализовать приложение, посмотреть, что в нем изменяется, а затем вернуться, выделить и инкапсулировать переменные составляющие?

О: Не всегда; в ходе проектирования приложения часто удается заранее выявить изменяющиеся аспекты и включить гибкие средства для работы с ними в программный код. Общие принципы и паттерны применимы на любой стадии жизненного цикла разработки.

В: Может, Duck тоже стоит преобразовать в интерфейс?

О: Не в этом случае. Структура, в которой Duck не является интерфейсом, имеет свои преимущества: она позволяет конкретным подклассам уток (например, MallardDuck) наследовать общие свойства и методы. После исключения переменных аспектов из иерархии Duck мы пользуемся преимуществами этой структуры без всяких проблем.

В: Класс, представляющий поведение, выглядит немного странно. Разве классы не должны представлять *сущности*? И разве классы не должны обладать *состоянием* И *поведением*?

О: Действительно, в ОО-системах классы представляют сущности, которые обычно обладают как состоянием (переменными экземпляров), так и методами. И в данном случае сущностью оказывается поведение. Однако даже поведение может обладать состоянием и методами; скажем, поведение полета может использовать переменные экземпляров, представляющие атрибуты полета (количество взмахов крыльев в минуту, максимальная высота и скорость и т. д.).

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?

Ответы:

1. Создайте класс FlyRocketPowered, реализующий интерфейс FlyBehavior.

2. Например, утиный манок (охотничье устройство, подражающее кряканью).

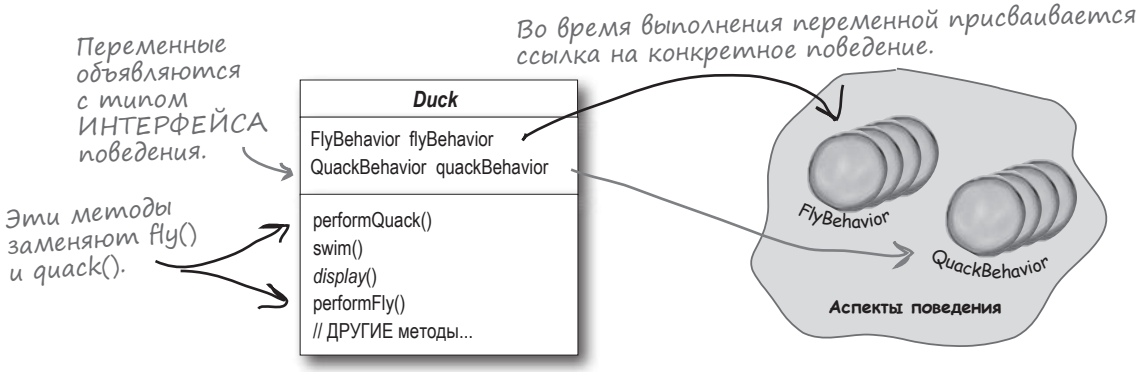
Интеграция поведения с классом Duck

У новой структуры есть одна принципиальная особенность: класс Duck теперь *делегирует* свои аспекты поведения (вместо простого использования методов, определенных в классе Duck или его субклассах). Вот как это делается:

❶ **Начнем с добавления двух переменных экземпляров с типами FlyBehavior и QuackBehavior** – назовем их *flyBehavior* и *quackBehavior*. Каждый конкретный объект утки будет присваивать значения этих переменных в определенный момент выполнения – например, FlyWithWings для полета и Squeak для кряканья.

Методы fly() и quack() удаляются из класса Duck (и всех субклассов), потому что это поведение перемещается в классы FlyBehavior и QuackBehavior.

В классе Duck методы fly() и quack() заменяются двумя аналогичными методами: performFly() и performQuack(); вскоре вы увидите, как они работают.



❷ **Реализация performQuack()**

```
public abstract 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.

Все просто, верно? Вместо того чтобы выполнять действие самостоятельно, объект Duck просто поручает эту работу объекту, на который ссылается quackBehavior. В этой части нас совершенно не интересует, что это за объект, – ***важно лишь, чтобы он умел выполнять quack()***!

Подробнее об интеграции...

❸ Пора разобраться с тем, как присваиваются значения переменным **flyBehavior** и **quackBehavior**. Рассмотрим фрагмент класса MallardDuck:

```
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");
   }
}
```

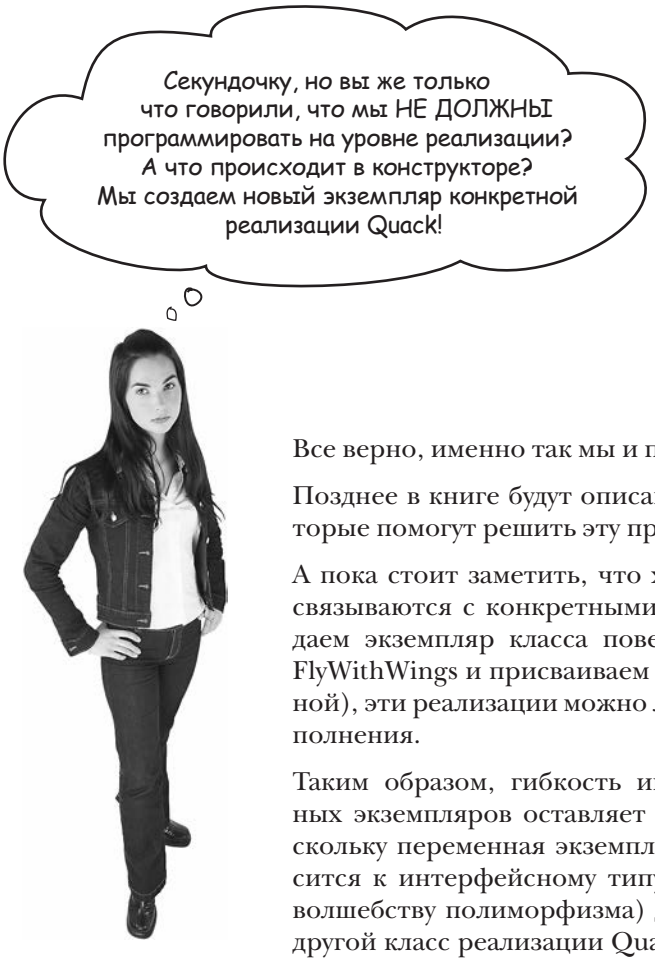
MallardDuck использует класс Quack для выполнения действия, так что при вызове performQuack() ответственность за выполнение возлагается на объект Quack.

А в качестве реализации FlyBehavior используется тип FlyWithWings.

Запомните, что MallardDuck наследует переменные quackBehavior и flyBehavior от класса Duck.

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).



Все верно, именно так мы и поступаем... *пока*.

Позднее в книге будут описаны другие паттерны, которые помогут решить эту проблему.

А пока стоит заметить, что хотя аспекты поведения связываются с конкретными реализациями (мы создаем экземпляр класса поведения типа Quack или FlyWithWings и присваиваем его ссылочной переменной), эти реализации можно *легко* менять во время выполнения.

Таким образом, гибкость инициализации переменных экземпляров оставляет желать лучшего. Но поскольку переменная экземпляра quackBehavior относится к интерфейсному типу, мы можем (благодаря волшебству полиморфизма) динамически присвоить другой класс реализации QuackBehavior во время выполнения.

Остановитесь на минуту и подумайте, как бы *вы* реализовали динамическое изменение поведения. (Пример кода будет приведен через несколько страниц.)

Тестирование кода Duck

1 Введите и откомпилируйте класс Duck (Duck.java, см. ниже) и класс MallardDuck class (MallardDuck.java, приводился две страницы назад).

```
public abstract class Duck {

   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;

   public Duck() {
   }

   public abstract void display();

   public void performFly() {
      flyBehavior.fly();
   }

   public void performQuack() {
      quackBehavior.quack();
   }

   public void swim() {
      System.out.println("All ducks float, even decoys!");
   }
}
```

Объявляем две ссылочные переменные с типами интерфейсов поведения. Переменные наследуются всеми субклассами Duck (в том же пакете).

Делегирование операции классам поведения.

2 Введите и откомпилируйте интерфейс FlyBehavior (FlyBehavior.java) и два класса реализации поведения (FlyWithWings.java и FlyNoWay.java).

```
public interface FlyBehavior {
   public void fly();
}
```

Интерфейс реализуется всеми классами.

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("I'm flying!!");
   }
}
```

Реализация поведения для уток, которые УМЕЮТ летать...

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
       System.out.println("I can't fly");
   }
}
```

Реализация поведения для уток, которые НЕ ЛЕТАЮТ (например, резиновых).

Тестирование кода Duck продолжается...

3 Введите и откомпилируйте интерфейс QuackBehavior (QuackBehavior.java) и три класса реализации поведения (Quack.java, MuteQuack.java и Sqeak.java).

```
public interface QuackBehavior {
   public void quack();
}
```

```
public class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Quack");
   }
}
```

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
       System.out.println("<< Silence >>");
   }
}
```

```
public class Squeak implements QuackBehavior {
   public void quack() {
       System.out.println("Squeak");
   }
}
```

4 Введите и откомпилируйте тестовый класс (MiniDuckSimulator.java).

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();
   }
}
```

Вызов метода performQuack(), унаследованного классом MallardDuck; метод делегирует выполнение операции по ссылке на QuackBehavior (то есть вызывает quack() через унаследованную переменную quackBehavior).

Затем то же самое происходит с методом performFly(), также унаследованным классом MallardDuck.

5 Выполните код!

```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator

Quack

I'm flying!!
```

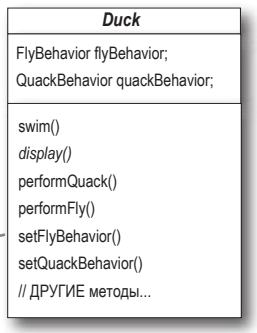
Динамическое изменение поведения

Согласитесь, обидно было бы наделить наших уток возможностями динамической смены поведения и не использовать их! Предположим, вы хотите, чтобы тип поведения задавался set-методом подкласса (вместо создания экземпляра в конструкторе).

❶ Добавьте два новых метода в класс Duck:

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



Вызывая эти методы в любой момент, мы можем изменить поведение утки «на лету».

❷ Создайте новый субкласс Duck (ModelDuck.java).

```
public class ModelDuck extends Duck {
    public ModelDuck() {
       flyBehavior = new FlyNoWay();
       quackBehavior = new Quack();
    }

    public void display() {
       System.out.println("I'm a model duck");
    }
}
```

Утка-приманка изначально летать не умеет...

❸ Определите новый тип FlyBehavior (FlyRocketPowered.java).

Определяем новое поведение — реактивный полет.

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
       System.out.println("I'm flying with a rocket!");
    }
}
```

4 **Внесите изменения в тестовый класс (MiniDuckSimulator.java), добавьте экземпляр ModelDuck и переведите его на реактивную тягу.**

```
public class MiniDuckSimulator {
   public static void main(String[] args) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck model = new ModelDuck();
      model.performFly();
      model.setFlyBehavior(new FlyRocketPowered());
      model.performFly();
   }
}
```

до

Первый вызов performFly() передается реализации, заданной в конструкторе ModelDuck, то есть экземпляру FlyNoWay.

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

Вызываем set-метод, унаследованный классом ModelDuck, и... утка-приманка вдруг взлетает на реактивном двигателе!

после

5 **Поехали!**

```
File Edit Window Help Yabadabadoo
%java MiniDuckSimulator
Quack
I'm flying!!
I can't fly
I'm flying with a rocket
```

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

Инкапсуляция поведения: общая картина

Мы основательно повозились с конкретной архитектурой. Пора сделать шаг назад и взглянуть на картину в целом.

Ниже изображена вся переработанная структура классов. В ней есть все, чего можно ожидать: классы уток, расширяющие Duck, а также классы поведения, реализующие FlyBehavior и QuackBehavior.

Стоит заметить, что мы начинаем рассматривать происходящее с несколько иной точки зрения. Поведение утки уже рассматривается не как *совокупность аспектов поведения*, а как *семейство алгоритмов*. В архитектуре SimUDuck алгоритмы представляют то, что делают утки (как они летают, крякают и т. д.), однако эту методологию с таким же успехом можно применить к набору классов для вычисления налога с продаж в разных штатах.

Обратите особое внимание на *отношения* между классами. А еще лучше — возьмите ручку и подпишите тип отношения (ЯВЛЯЕТСЯ, СОДЕРЖИТ или РЕАЛИЗУЕТ) над каждой стрелкой на диаграмме.

Не ленитесь и сделайте.