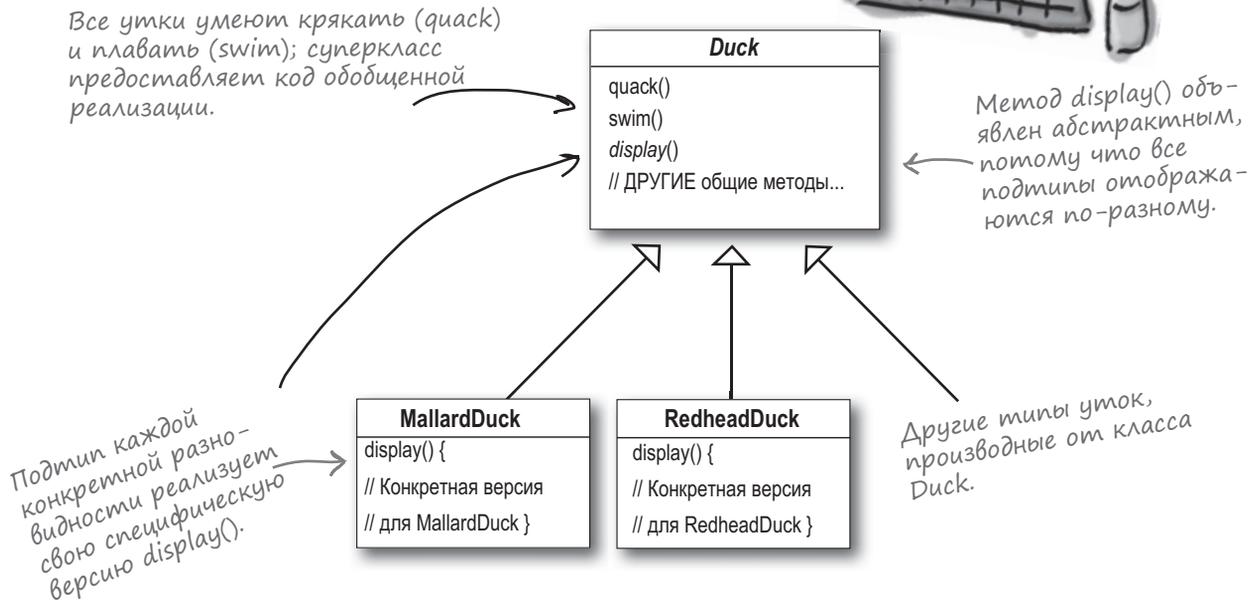


## Все началось с простого приложения SimUDuck

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

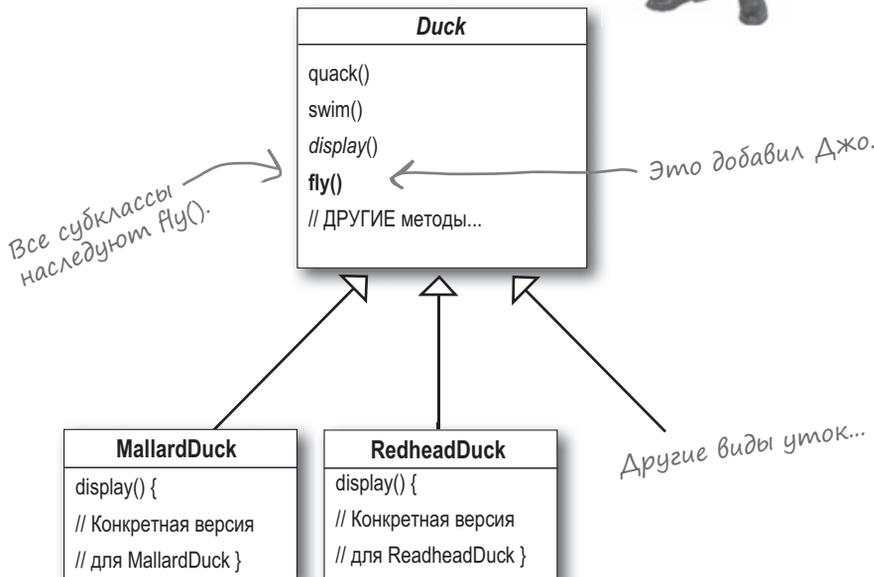


За последний год компания испытывала жесткое давление со стороны конкурентов. Через неделю долгих выездных совещаний за игрой в гольф руководство компании решило, что пришло время серьезных изменений. Нужно сделать что-то действительно впечатляющее, что можно было бы продемонстрировать на предстоящем собрании акционеров *на следующей неделе*.

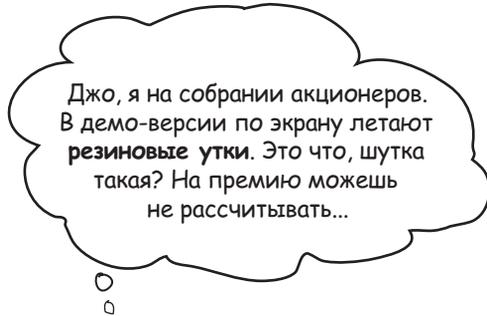
## Теперь утку бугум ЛЕТАТЬ

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?»

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



## Но тут все пошло наперекосяк ...

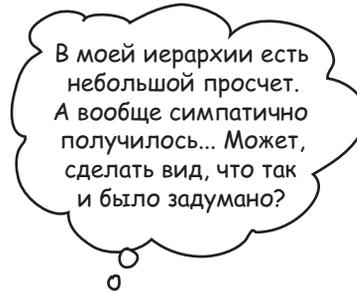


### Что произошло?



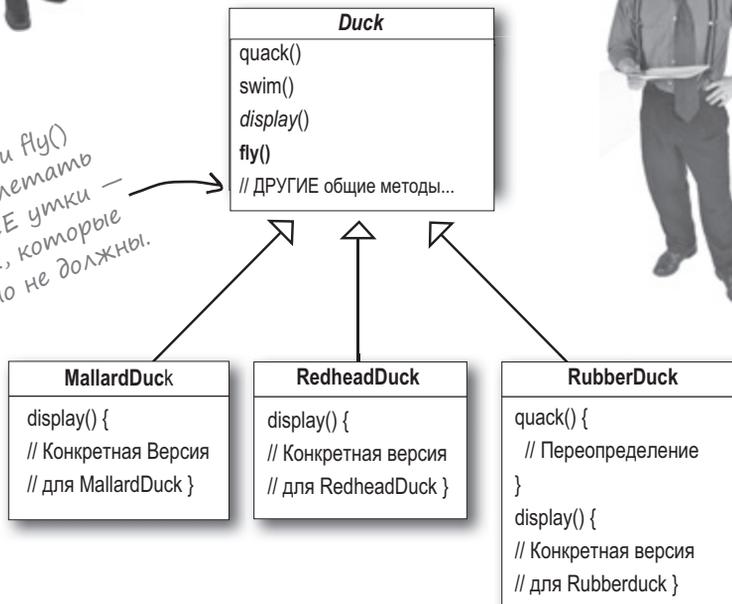
Джо не сообразил, что *летать* должны не все subclasses Duck. Новое поведение, добавленное в суперкласс Duck, оказалось *неподходящим* для некоторых subclasses. И теперь в программе начали летать неодушевленные объекты.

*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.

*При размещении fly() в суперклассе летать начинают ВСЕ утки — включая тех, которые летать явно не должны.*



*Резиновые утки не крякают, поэтому метод quack() переопределен.*

## Джо думает о наследовании...

Я всегда могу переопределить метод fly() в классе RubberDuck, по аналогии с quack() ...



```

RubberDuck

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

Но что произойдет, если в программу добавятся деревянные утки-приманки? Они не должны ни летать, ни крякать...



*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.)

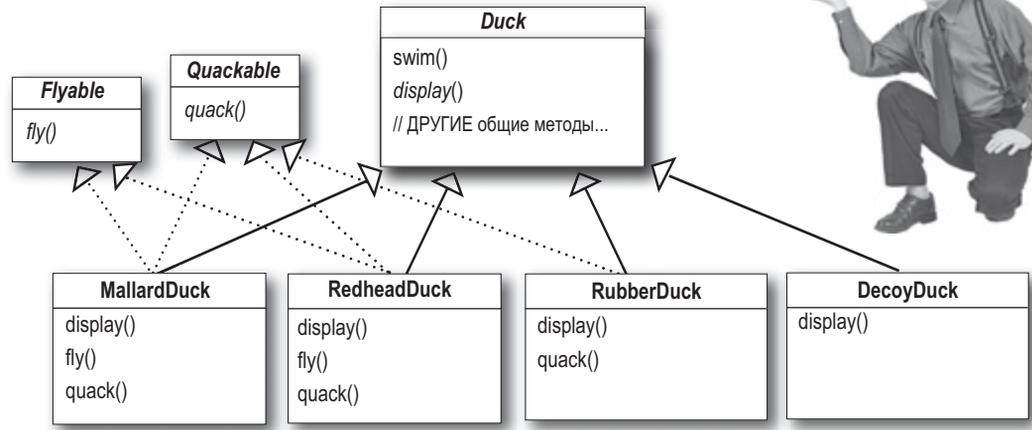
- A. Дублирование кода в subclasses.
- B. Трудности с изменением поведения на стадии выполнения.
- C. Уток нельзя научить танцевать.
- D. Трудности с получением информации обо всех аспектах поведения уток.
- E. Утки не могут летать и крякать одновременно.
- F. Изменения могут оказать непредвиденное влияние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

## Как насчет интерфейса?

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 subclasses, включаемого в программу... *вечно*.

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

Я исключу метод fly() из суперкласса Duck и определю интерфейс **Flyable()** с методом fly(). Только те утки, которые **должны** летать, реализуют интерфейс и содержат метод fly()... А я с таким же успехом могу определить интерфейс **Quackable**, потому что не все утки крикают.



## 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» subclasses Duck?!



## А как бы ВЫ поступили на месте Джо?

Мы знаем, что *не все* subclasses 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?

RNHЭHЭMEN

(ответ можно прочесть 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).

*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!*

---

---

---

---

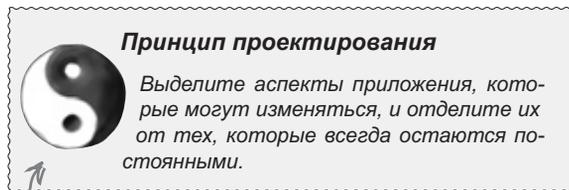
---

---

## Захожу на цель...

Итак, наследование нам не подошло, потому что утиное поведение изменяется в subclasses, а некоторые аспекты поведения присутствуют *не во всех* subclasses. 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 subclasses, где он определяется, — скорее всего, с внесением *новых* ошибок!

К счастью, для подобных ситуаций существует полезный принцип проектирования.



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 неизменным.

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.

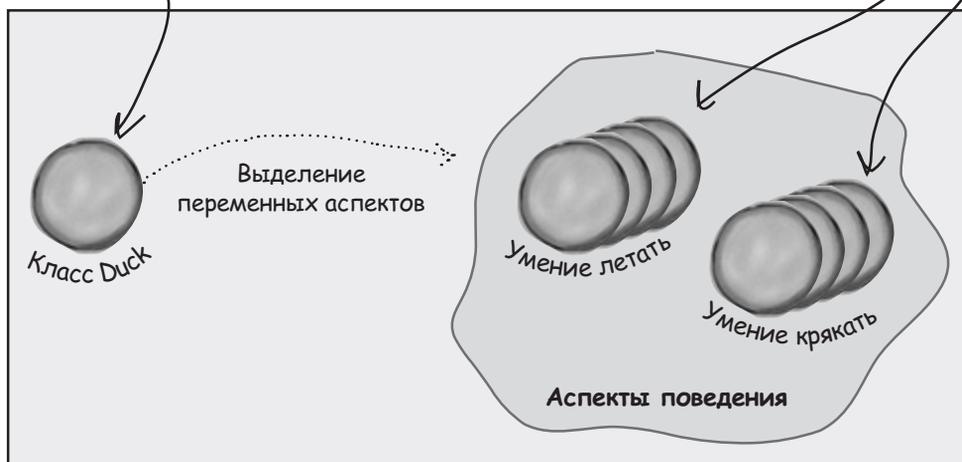
**Мы знаем, что `fly()` и `quack()` — части класса `Duck`, изменяющиеся в зависимости от subclasses.**

**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.

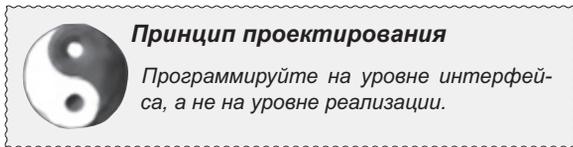


## Проектирование переменного поведения

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.



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 subclasses. 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 subclasses 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 subclassу 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.

