Глава 6

Создание объектов

Любая объектно-ориентированная система создает объекты или объектные структуры. В то же время могут возникать ситуации, при которых разработанный код содержит части, дублирующие друг друга. Кроме того, он может оказаться недостаточно простым и наглядным или не будет максимально слабо связанным с клиентским кодом. Шесть рефакторингов, представленные в этой главе, предназначены для решения проблем проектирования во всех областях — начиная от конструкторов с излишне усложненной логикой и заканчивая ненужными синглтонами [12]. Описанные ниже рефакторинги не охватывают всех проблем, которые могут возникнуть при создании проекта. Тем не менее они помогают решить самые распространенные из них.

В том случае, если класс имеет слишком большое количество конструкторов, у клиента могут возникнуть трудности при определении конструктора, который необходимо вызвать. Один из способов решения этой проблемы — сокращение количества конструкторов путем применения таких рефакторингов [15], как Extract Class или Extract Subclass. Если же это невозможно или бесполезно, можно определить назначение конструктора более четко, применив рефакторинг Replace Constructor with Creation Method (с. 84).

Что же такое Creation Method (метод создания)? Это просто статический или нестатический метод, который создает и возвращает экземпляр объекта. При написании этой книги я решил отдельно выделить шаблон Creation Method для того, чтобы отличать его от шаблона Factory Method (метод фабрики) [12]. Последний удобно применять при полиморфной разработке. В отличие от метода создания, он может быть нестатическим и должен реализоваться по меньшей мере в двух классах (обычно это надкласс и подкласс). Если классы иерархии реализуют метод подобным образом (за исключением этапа создания объекта), то, скорее всего, дублирование кода можно устранить, применяя рефакторинг Introduce Polymorphic Creation with Factory Method (с. 116).

Класс, представляющий собой фабрику, реализует один или несколько методов создания. Если данные и/или код, используемые при создании объекта, распределены по многочисленным классам, вам, скорее всего, придется корректировать код во многих местах, что явно указывает на необходимость применения рефакторинга Solution Sprawl (с. 73). Если же применить рефакторинг Move Creation Knowledge to Factory (с. 96), то можно объединить код и данные в одной фабрике, что позволяет снизить “размазанность”.

Другой часто используемый рефакторинг, включающий шаблон Factory, — это Encapsulate Classes with Factory (с. 108). Можно выделить две основные причины для его применения: во-первых, обеспечение связи клиентов с экземплярами классов через общий интерфейс и, во-вторых, сокращение объема сведений о классах, необходимых клиенту при создании экземпляров классов, доступных через фабрику.

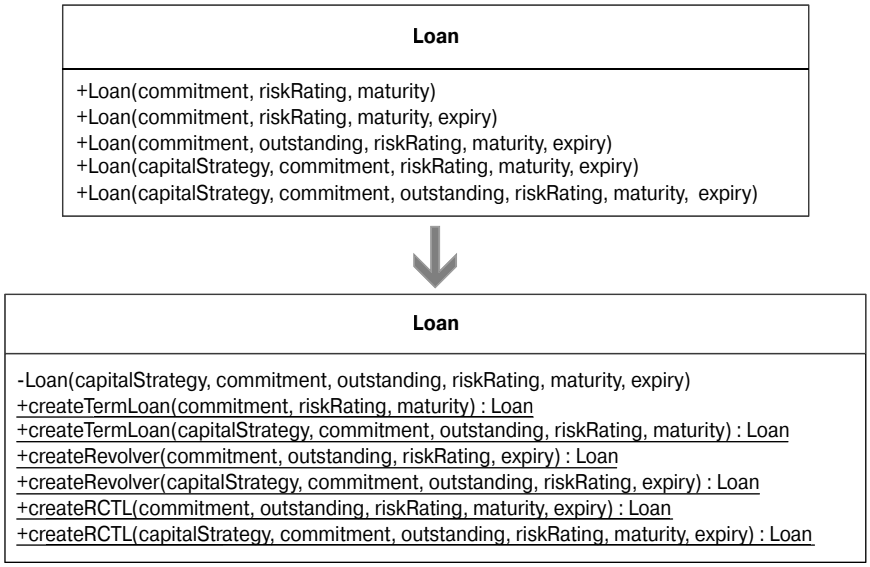
Из всех шаблонов, призванных упростить конструкцию объектной структуры, лучшим является Builder [12]. Рефакторинг Encapsulate Composite with Builder (с. 124) иллюстрирует, как с помощью этого шаблона можно обеспечить более простой и менее подверженный ошибкам способ построения шаблона Composite [12].

Последним в данной главе описывается рефакторинг Inline Singleton (с. 143). Его создание принесло мне огромное количество положительных эмоций, поскольку я часто сталкиваюсь с синглтонами, существование которых абсолютно ничем не обусловлено. Данный рефакторинг (он показывает, как можно удалить синглтон из кода) появился в результате совместного обсуждения синглтонов Вардом Каннингемом (Ward Cunningham), Кентом Беком (Kent Beck) и Мартином Фаулером (Martin Fowler).

Replace Constructors with Creation Methods

Конструкторы класса создают трудности при выборе правильного конструктора в процессе разработки.

Заменить конструкторы методами создания с понятным назначением, которые возвращают экземпляры объектов.



Мотивация

В ряде языков разрешено присваивать конструктору любое название, не зависящее от названия класса. Но в некоторых языках, например в Java и C++, такой вариант недопустим: каждый конструктор должен быть назван в соответствии с его классом. Если в наличии только один простой конструктор, это может и не создавать проблем. С другой стороны, в тех случаях, когда у класса есть множество конструкторов, программисты оказываются перед выбором: основываясь на информации об ожидаемых параметрах и/или на коде конструктора, следует выбрать, какой конструктор должен быть вызван. Что же неправильно в таком подходе? Многое.

Конструкторы просто не передают свое назначение рационально или эффективно. Чем больше у класса конструкторов, тем больше шансов на то, что программист сделает неправильный выбор. Необходимость выбора конструктора снижает эффективность разработки. Код, в котором возможен вызов одного из множества конструкторов, часто недостаточно полно передает природу создаваемого объекта.

Если необходимо добавить в класс новый конструктор, сигнатура которого совпадает с сигнатурой уже существующего, очень легко зайти в тупик. В таких случаях добавить новый конструктор невозможно, поскольку для этого придется использовать уже занятое название. Поскольку все конструкторы должны иметь одно и то же имя, наличие в одном классе конструкторов с одинаковой сигнатурой невозможно, несмотря на то что эти конструкторы могут создавать разные виды объектов.

Ситуация, когда обнаруживается множество давно не используемых конструкторов, которые, тем не менее, продолжают "жить" в коде, очень распространена. Особенно это характерно для высокоразвитых систем. Почему же такие "мертвые" конструкторы все еще существуют в коде? В большинстве случаев это происходит из-за того, что программистам просто не известно, что данный конструктор не вызывается. Это происходит потому, что они либо не выполняют проверку вызываемых процедур (возможно, тому виной необходимость составлять слишком сложное поисковое выражение), либо не используют среду разработки, которая автоматически выделяет невызываемый код. Но в любом случае "мертвые" конструкторы только необоснованно увеличивают объем класса и делают его более сложным, чем необходимо.

Для устранения этой проблемы можно использовать Creation Method — статический или нестатический метод класса, который инстанцирует новые экземпляры класса. Для таких методов нет никаких ограничений на название. Поэтому при выборе названия можно четко выразить, что именно создается (например, метод `createTermLoan()` или `createRevolver()`). Такая гибкость именования позволяет иметь два метода создания с разными именами, но с одинаковым числом и типами аргументов. Кроме того, программистам, не имеющим доступа к современным средам разработки, обычно легче обнаружить неиспользуемый метод, чем неиспользуемый конструктор. Это происходит потому, что поисковые выражения для специфически

названных методов проще для составления, чем поисковые выражения для конструктора или группы конструкторов.

Единственная неприятность данного рефакторинга в том, что он может приводить к нестандартным способам создания объектов. Если большинство классов инстанцируются с помощью оператора `new`, и одновременно некоторые классы инстанцируются с помощью метода создания, то программисту приходится изучать, каким из способов создан каждый класс. Однако этот нестандартный метод создания может оказаться меньшим злом, чем наличие класса со слишком большим количеством конструкторов.

Если было определено, что класс имеет большое количество конструкторов, то перед применением рассматриваемого рефакторинга лучше всего применить рефакторинг Extract Class или Extract Subclass [15]. Рефакторинг Extract Class будет хорош в том случае, если класс, о котором идет речь, просто выполняет слишком большой объем работы (т.е. на него накладывается слишком много обязательств). Что же касается рефакторинга Extract Subclass, то его применение будет полезно в том случае, если экземпляры класса используют лишь небольшую часть своих переменных.

Creation Method и Factory Method

Что в нашей индустрии называется методом, создающим объекты? Многие программисты, следуя названию, используемому для создающих шаблонов [12], могут ответить: "Метод фабрики". Но все ли методы, с помощью которых создаются объекты, действительно являются фабриками? Если определить этот термин достаточно широко (т.е. как метод, который просто создает объекты), то, безусловно, ответ будет положительным. Но если следовать определению шаблона Factory Method, данному его авторами в 1995 году, то становится понятно, что не каждый метод, создающий объекты, обеспечивает слабую связь в том виде, в каком она предусмотрена в истинном Factory Method (например, рефакторинг Introduce Polymorphic Creation with Factory Method на с. 116).

Для того чтобы читателю было легче разобраться с рассматриваемыми проектами или рефакторингами, связанными с созданием объектов, я ввожу термин Creation Method, которым определяю статический или нестатический метод, создающий экземпляры класса. Таким образом, каждый метод фабрики является методом создания, но не обязательно наоборот. Это также означает, что термин Creation Method можно использовать там, где Мартин Фаулер использует термин "factory method" [15], а Джошуа Блок (Joshua Bloch) — термин "static factory method" [10].

Преимущества и недостатки

+ Лучше, чем конструктор, информирует о доступных видах экземпляров класса.
+ Обходит ограничения на конструкторы, в частности невозможность наличия двух конструкторов с одинаковыми количеством и типами аргументов.
+ Упрощает процедуру поиска неиспользуемого кода создания объектов классов.
– Делает создание объектов класса нестандартным: одни классы инстанцируются при помощи оператора `new`, а другие используют методы создания.

Механика

Перед началом реорганизации кода необходимо определить *всеохватывающий конструктор* (catch-all constructor). Это полнофункциональный конструктор, которому делегируют свою работу другие конструкторы. Если такого конструктора нет в наличии, его можно создать с помощью рефакторинга Chain Constructors (с. 380).

1. Найдите клиента, вызывающего конструктор класса, для того чтобы создать *вид* экземпляра. Для построения открытого статического метода примените к вызову конструктора рефакторинг Extract Method [15]. Полученный метод и будет *методом создания*. После этого примените рефакторинг Move Method [15] для перемещения созданного метода в класс, содержащий выбранный конструктор.

   ✓ Скомпилируйте и протестируйте.

2. Найдите для выбранного конструктора все вызывающие функции, которые инстанцируют тот же вид объекта, что и метод создания, и измените их таким образом, чтобы они вызывали этот метод создания.

   ✓ Скомпилируйте и протестируйте.

3. Если выбранный конструктор по цепочке вызывает другой конструктор, ваш метод создания должен вызывать вместо выбранного конструктора вызываемый им конструктор. Это можно сделать с помощью встраивания конструктора, что похоже на рефакторинг Inline Method [15].

   ✓ Скомпилируйте и протестируйте.

4. Повторите шаги 1–3 для каждого из конструкторов класса, которые вы намерены преобразовать в методы создания.

5. Если конструктор класса не вызывается за пределами класса, сделайте его закрытым.

   ✓ Скомпилируйте.

Пример

В свое время я потратил несколько лет на написание, усовершенствование и поддержку некоторого калькулятора, подсчитывающего риски при ссудах. На основе этой работы и возник описанный ниже пример. Как показано в приведенном коде, класс `Loan` содержит множество конструкторов.

```
public class Loan...
  public Loan(double commitment, int riskRating,
              Date maturity, Date expiry) {
    this(commitment, 0.00, riskRating, matuity, null);
  }

public Loan(double commitment, int riskRating,
            Date maturity, Date expiry) {
  this(commitment, 0.00, riskRating, matuity, expiry);
}

public Loan(double commitment, double outstanding,
            int customerRating, Date maturity, Date expiry){
  this(null, commitment, outstanding,
       customerRating, matuity, expiry);
}

public Loan(CapitalStrategy capital Strategy,
            double commitment, int riskRating,
            Date maturity, Date expiry) {
  this(capitalStrategy, commitment, 0.00,
       riskRating, matuity, expiry);
}

public Loan(CapitalStrategy capital Strategy,
            double commitment, double outstanding,
            int riskRating, Date maturity, Date expiry){
  this.commitment=commitment;
  this.outstanding=outstanding;
  this.riskRating=riskRating;
  this.maturity=maturity;
  this.expiry=expiry;
  this.capitalStrategy=capitalStrategy;

  if (capitalStrategy=null){
    if (expiry==null)
      this.capitalStrategy=new CapitalStrategyTermLoan();
    else if (maturity==null)
      this.capitalStrategy=new CapitalStrategyRevolver();
    else
      this.capitalStrategy=new CapitalStrategyRCTL();
  }
}
```

Класс `Loan` может использоваться для представления семи видов ссуд. В этой книге описываются только три их них. Срочный кредит (Term Loan) означает ссуду, которая должна быть полностью выплачена до определенной даты (maturity date). Оборотная ссуда для кредитных карт — это ссуда, означающая "возобновляемый кредит" ("revolving credit"): существует лимит на расходы и дата платежа. Срочный кредит с возможностью возобновления (Revolving Credit Term Loan — RCTL) представляет собой возобновляемую ссуду, которая после истечения срока платежа трансформируется в срочную ссуду.

Итак, калькулятор поддерживает семь типов ссуд. Может возникнуть вопрос: почему класс `Loan` не представлен в виде абстрактного надкласса с подклассами для каждого типа ссуды. Может показаться, что это должно сократить количество конструкторов, необходимых для класса `Loan` и для его подклассов. Такое представление не будет удачным по двум причинам.

1. Основные различия между видами ссуд заключаются не в полях класса, а в том, каким образом производятся подсчеты, например как определяется капитал, доходы или продолжительность ссуды. Для того чтобы обеспечить три разных способа подсчета капитала при срочном кредите, совершенно не обязательно создавать три разных подкласса в классе `Loan`. Проще поддерживать один класс `Loan` и иметь три разных класса `Strategy` для срочного кредита (см. соответствующий пример для Replace Conditional Logic with Strategy на с. 158).
2. Приложению, использующему экземпляры класса `Loan`, необходимо иметь возможность преобразования одного вида ссуды в другой. Выполнить такое преобразование было бы легче, если бы оно включало в себя изменение лишь нескольких полей одного экземпляра класса `Loan`, а не полную замену одного экземпляра подкласса класса `Loan` другим.

Если посмотреть на исходный код класса `Loan`, представленный выше, можно увидеть, что класс имеет пять конструкторов, последний из которых и есть его всеохватывающий конструктор (см. Chain Constructors на с. 380). Не имея в запасе специальных знаний, довольно сложно определить, какой из конструкторов создает срочные кредиты, а какой — оборотные ссуды или ссуды RCTL.

Мне известно, что для ссуды RCTL необходимо определить и дату окончания ссуды (expiry), и срок платежа (maturity). Следовательно, я знаю, что для создания ссуды RCTL необходимо вызвать конструктор, который позволит передать обе эти даты. Но известно ли это вам? И можно ли предполагать, что это известно программисту, который будет читать код?

Что еще вложено в конструкторы класса `Loan` в качестве неявных сведений? На самом деле таких сведений довольно много. Если вызвать первый конструктор, который принимает три параметра, вы получите срочный кредит. Но, если необходима оборотная ссуда, следует вызвать один из тех конструкторов, которые принимают

две даты, а затем обнулить срок платежа. Я бы удивился, если бы это было известно всем пользователям данного кода. Скорее, они узнают это, лишь столкнувшись с какими-нибудь неприятностями.

Давайте посмотрим, что получится, если применить рефакторинг Replace Constructors with Creation Method.

1. Мой первый шаг — найти клиента, вызывающего один из конструкторов класса `Loan`. Ниже приведена одна из возможных вызывающих функций из набора функций тестирования.

```
public class CapitalCalculationTests...
  public void testTermLoanNoPayments() {
    ...
    Loan termLoan=new Loan(commitment, riskRating,
                           maturity);
    ...
  }
```

В этом случае вызов упомянутого ранее конструктора класса `Loan` создает срочный кредит. Я применяю к этому вызову рефакторинг Extract Method [15], чтобы получить открытый статический метод `createTermLoan`.

```
public class CapitalCalculationTests...
  public void testeTermLoanPayment(){
    ...
    Loan termLoan = createTermLoan(commitment, riskRating,
                                   maturity);
    ...
  }

  public static Loan
    createTermLoan(double commitment, int riskRating,
                   Date maturity){
    return new Loan(commitment, riskRating, maturity);
  }
```

Теперь я применяю к методу создания `createTermLoan` рефакторинг Move Method [15], для того чтобы переместить его в класс `Loan`. Это приводит к следующим изменениям:

```
public class Loan...
  public static Loan
    createTermLoan(double commitment, int riskRating,
                   Date maturity){
    return new Loan(commitment, riskRating, maturity);
  }

public class CapitalCalculationTest...
  public void testTermLoanNoPayment(){
  ...
```

```
  Loan termLoan=Loan.createTermLoan(commitment, riskRating,
                                    maturity);
  ...
}
```

Я компилирую и тестирую код, чтобы проверить, что все корректно работает.

2. Теперь я нахожу все вызывающие функции для конструктора, который вызывается методом `createTermLoan`, и изменяю их таким образом, чтобы они вызывали метод `createTermLoan`. Например, это можно сделать так:

```
public class CapitalCalculationTest...
  public void testTermLoanOnePayment() {
    ...
    Loan termLoan=new Loan(commitment, riskRating, maturity);
    Loan termLoan= Loan.cteateTermLoan(commitment, riskRating,
                                       maturity);
    ...
  }
```

Я еще раз компилирую и тестирую код, чтобы убедиться, что все нормально работает.

3. На данном этапе метод `createTermLoan` — единственная функция, вызывающая данный конструктор. Поскольку этот конструктор по цепочке вызывает другой конструктор, я могу его удалить, применив рефакторинг Inline Method [15] (в данном случае он фактически является "встроенным конструктором"). Это приводит к следующим изменениям:

```
public class Loan...
  public Loan(double commitment, int riskRating,
              Date maturity) {
    this(commitment, 0.00, riskRating, maturity, null);
  }

  public static Loan
    createTermLoan(double commitment, int riskrating,
                   Date maturity) {
    return new Loan(commitment, 0.00, riskRating,
                    maturity, null);
  }
```

Я снова компилирую и тестирую код, чтобы проверить, что после выполненных изменений программа продолжает работать.

4. Теперь я повторяю шаги 1–3 для построения в классе `Loan` дополнительных методов создания. Ниже приведен пример кода, вызывающего всеохватывающий конструктор.

```
public class CapitalCalculationTest...
  public void testTermLoanWithRiskAdjustedCapitalStrategy(){
    ...
    Loan termLoan =
      new Loan(riskAdjustedCapitalStrategy, commitment,
               outstanding, riskRating, maturity, null);
  ...
  }
```

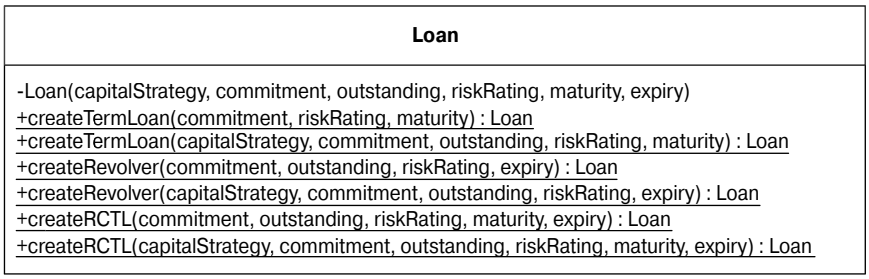
Обратите внимание на нулевое значение, которое передается конструктору в качестве последнего параметра. Передача конструктору нулевых значений — плохая практика, приводящая к снижению читаемости кода. Обычно данная проблема возникает из-за того, что программисты не могут отыскать именно тот конструктор, который им необходим, и вместо создания еще одного конструктора вызывают другой конструктор с более широкими возможностями. Для реорганизации этого кода с использованием Creation Method я повторяю шаги 1 и 2. Шаг 1 приводит к другому методу `createTermLoan` в классе `Loan`.

```
public class CapitalCalculationTests...
  public void testTermLoanwithRiskAdjustedCapitalStrategy(){
    ...
    Loan termLoan =
      Loan.createTermLoan(riskAdjustedCapitalStrategy,
                          commitment, outstanding,
                          riskRating, maturity~~, null~~);
    ...
  }

public class Loan...
  public static Loan
    createTermeLoan(double commitment, int riskRating,
                    Date maturity) {
    return new Loan(commitment, 0.00, riskRating,
                    maturity, null);
  }

 public static Loan
    createTermeLoan(
      CapitalStrategy riskAdjustedCapitalStrategy,
      double commitment, double outstanding, int riskRating,
      Date maturity) {
      return new Loan(riskAdjustedCapitalStrategy,
                      commitment, outstanding, riskRating,
                      maturity, null);
  };
```

Может возникнуть вопрос, почему я решил перегрузить метод `createTermLoan(...)`, вместо того чтобы создать Creation Method со своим собственным названием, например `createTermLoanWithStrategy(...)`? Я принял такое решение, так как считаю, что наличие параметра `CapitalStrategy` достаточно ясно передает разницу между двумя перегруженными версиями метода `createTermLoan(...)`.

Переходим ко второму шагу рассматриваемого рефакторинга. Поскольку новый метод `createTermLoan(...)` вызывает всеохватывающий конструктор класса `Loan`, я должен отыскать остальные места вызова всеохватывающего конструктора для инстанцирования того же вида объектов класса `Loan`, что и создаваемые методом `createTermLoan(...)`. Эта работа требует аккуратности, потому что некоторые функции, вызывающие всеохватывающий конструктор, создают экземпляры класса `Loan` для оборотной ссуды или для ссуды RCTL. Поэтому я изменяю только ту часть кода клиента, в которой создаются экземпляры класса `Loan` для срочного кредита.

Поскольку всеохватывающий конструктор не вызывает по цепочке никакого другого конструктора, нет необходимости выполнять шаг 3. Я продолжаю рефакторинг, выполняя шаг 4, который заключается в повторении шагов 1–3. На этом этапе я получаю методы создания, показанные ниже.



5. Осталось выполнить последний шаг: изменить видимость единственного оставшегося открытого конструктора, а именно всеохватывающего конструктора класса `Loan`. Поскольку у класса `Loan` нет подклассов и теперь конструктор не вызывается за пределами класса, я делаю его закрытым.

```
public class Loan...
  private Loan(CapitalStrategy capitalStrategy,
               double commitment,
               double outstanding, int riskRating,
               Date maturity, Date expiry)...
```

Я еще раз компилирую и тестирую код, чтобы проверить, что все продолжает работать. Реорганизация кода выполнена.

Теперь должно быть очевидно, как получать разные виды экземпляров класса `Loan`. Неоднозначности устранены, и все неявное знание сделано явным. Что же еще осталось сделать? В методах создания задействовано много параметров, поэтому может иметь смысл применить рефакторинг Introduce Parameter Object [15].

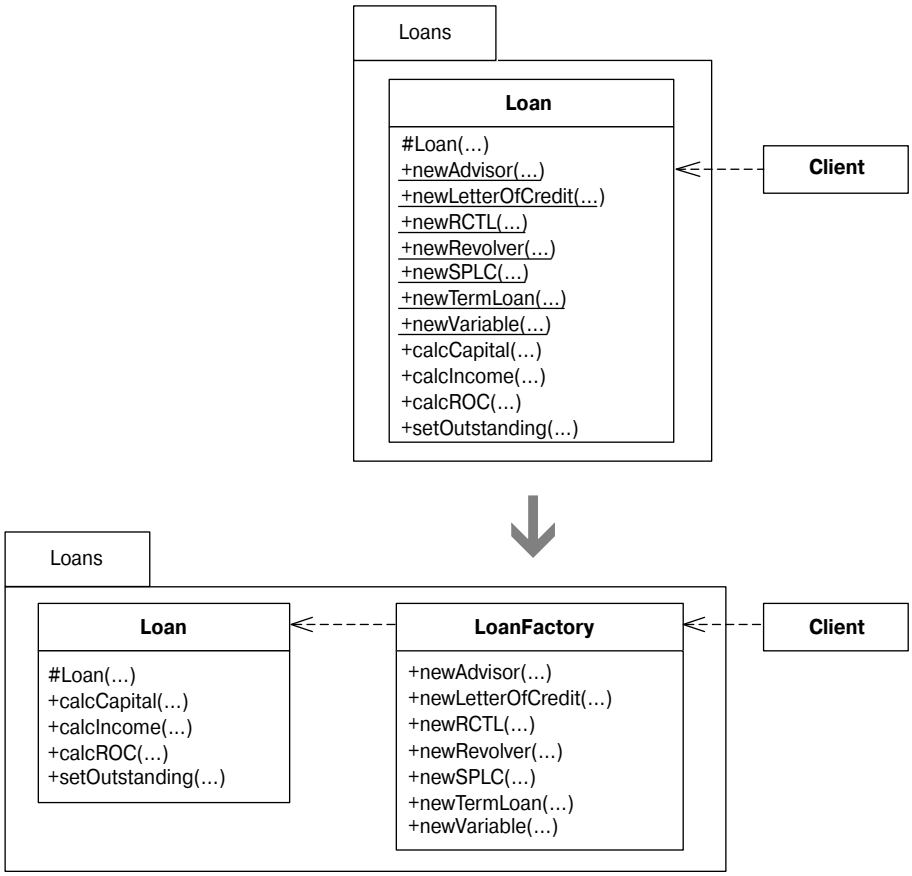
Разновидности

Parameterized Creation Methods

Если вы внимательно посмотрите на реализацию рефакторинга Replace Constructors with Creation Methods, то сможете подсчитать, что для каждой конфигурации объекта, поддерживаемой классом, необходимо около 50 методов создания. Перспектива построения 50 методов не очень вдохновляет, поэтому вы можете решить, что применять этот рефакторинг не стоит. Но имейте в виду, что существуют и другие решения этой проблемы. Во-первых, нет необходимости разрабатывать метод создания для каждой конфигурации объекта. Его можно написать только для самых популярных конфигураций, а для обработки остальных случаев оставить несколько открытых конструкторов. Кроме того, для снижения количества необходимых методов создания имеет смысл рассмотреть использование параметров.

Extract Factory

Может ли такое количество методов создания затенить исходное назначение? В действительности это лишь дело вкуса. Некоторые считают, что когда создание объектов начинает доминировать в открытом интерфейсе класса, то такой класс больше не может отвечать своему основному назначению. Если вы работаете с классом, в котором имеются методы создания, и считаете, что эти методы отвлекают вас от основного назначения класса, можете реорганизовать связанные методы в единую фабрику, например как показано ниже.

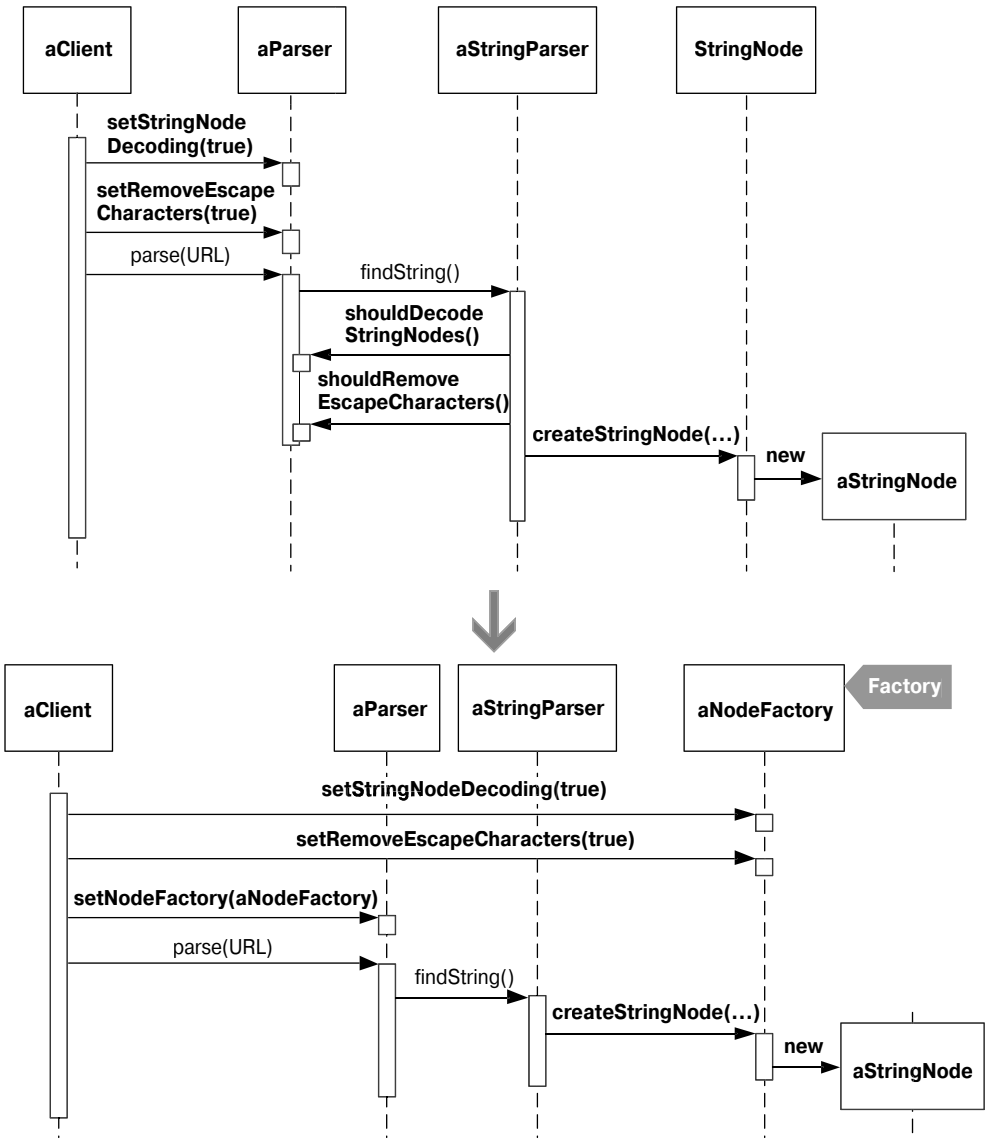


Нет ничего страшного в том, что класс `LoanFactory` не представляет собой Abstract Factory [12]. Классы Abstract Factory могут быть заменены во время выполнения программы. Вы можете определить различные абстрактные фабрики, каждая из которых знает, как возвращать семейство продуктов, а также снабдить систему или клиента конкретным экземпляром Abstract Factory. Обычно фабрики тяготеют к снижению сложности. Они зачастую реализуются как отдельные классы, не являющиеся частью какой-либо иерархии.

Move Creation Knowledge to Factory

Данные и код, используемые для создания класса, распределены по многочисленным классам.

Собрать сведения о создании в отдельный класс фабрики.

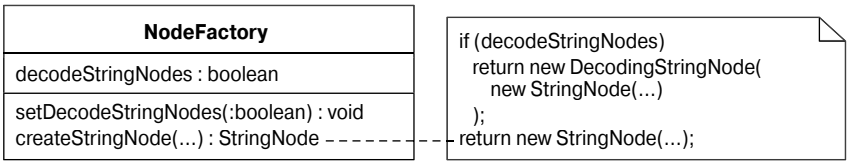


Мотивация

Когда информация, необходимая для создания объекта, рассредоточена по множеству классов, мы получаем признак "расплывшегося решения": размещение ответственности за создание объекта в классе, который не имеет никакого отношения к классу создаваемого объекта. "Расплывшееся создание" представляет собой частный случай признака плохого кода Solution Sprawl (с. 73) и обычно является результатом более ранних проблем проектирования. Например, клиент должен сконфигурировать объект согласно некоторым требованиям, но для него не обеспечен доступ к коду создания объекта. Если клиент не может получить легкий доступ к такому коду (например, если объект находится на системном уровне, значительно удаленном от клиента), то как он сможет сконфигурировать объект?

Обычно эта проблема решается методом грубой силы. Клиент передает требования к конфигурации некоторому объекту, тот, в свою очередь, передает их следующему объекту, который и хранит их до того момента, пока код создания объекта не обратится к ним посредством еще одной цепочки промежуточных объектов. Такие действия приводят к значительному рассредоточению кода и данных, необходимых для создания объекта.

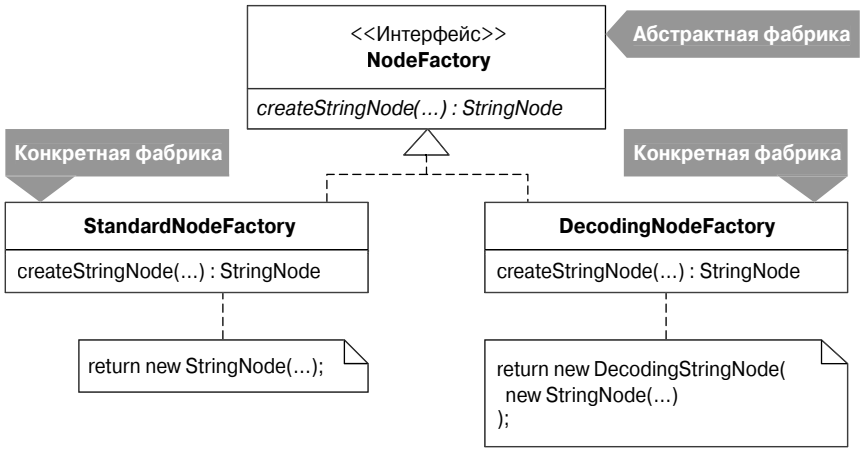
В этом контексте полезен шаблон Factory. С его помощью в одном классе инкапсулируется логика создания и требования клиента к инстанцированию и конфигурации. Клиент может передать в экземпляр Factory информацию о том, как следует инстанцировать или настраивать объект, после чего во время выполнения программы этот экземпляр Factory можно использовать для выполнения инстанцирования/конфигурации. Например, класс `NodeFactory` создает экземпляры класса `StringNode` и может быть сконфигурирован клиентами таким образом, чтобы усовершенствовать эти экземпляры с помощью шаблона Decorator, реализованного как класс `DecodingStringNode`.



Фабрика не обязательно должна быть реализована исключительно конкретным классом. Для определения фабрики можно использовать интерфейс и заставить существующий класс реализовать его. Такой подход полезен в том случае, если вы хотите, чтобы прочие области системы связывались с экземпляром существующего класса исключительно посредством интерфейса фабрики.

Если же логика создания внутри фабрики становится слишком сложной (что, возможно, обусловлено поддержкой слишком большого количества опций создания), может иметь смысл выделить ее в Abstract Factory [12]. После того как эта процедура будет выполнена, клиент сможет конфигурировать систему, используя

определенную конкретную фабрику (т.е. конкретную реализацию Abstract Factory) или позволить системе использовать конкретную фабрику по умолчанию. Хотя ранее упомянутый класс `NodeFactory` не настолько усложнен и нет реальной необходимости в его усовершенствовании, далее представлена диаграмма, показывающая, как будет выглядеть данный класс, если преобразовать его в Abstract Factory.



Что же такое фабрика?

Одно из наиболее злоупотребляемых и наименее точных понятий в нашей области знаний — это слово “Фабрика”. Одни, используя термин “шаблон Factory”, подразумевают под этим словом Factory Method [12], другие применяют этот термин для обозначения Abstract Factory [12], а третьи могут подразумевать и то и другое. Кроме того, некоторые могут таким образом обозначать *любой* код, создающий объекты.

Отсутствие общепринятого и понятного определения для слова “фабрика” ограничивает наши возможности в выяснении того, будет ли полезна фабрика для данного проекта? Поэтому я предлагаю свое определение, которое одновременно является и достаточно полным, и довольно-таки узким: *фабрика* — это класс, реализующий один или несколько методов создания.

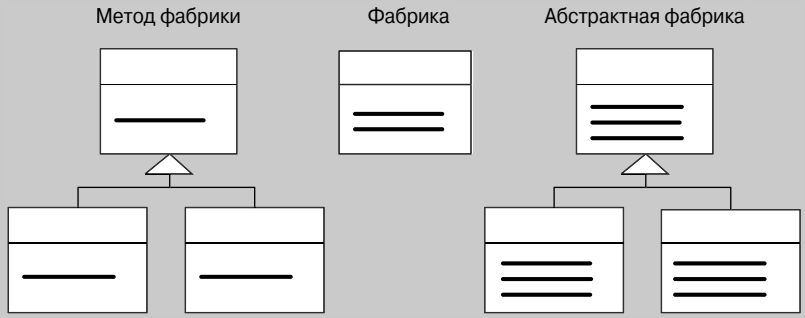
Это определение корректно и для статических, и для нестатических методов создания. Оно корректно, как в том случае, если возвращаемый тип метода создания является интерфейсом, абстрактным классом или конкретным классом, так и в том случае, если класс, реализующий метод создания, кроме этого, выполняет действия, не связанные с созданием.

Factory Method [12] — это нестатический метод, возвращающий тип базового класса или интерфейса и реализованный в иерархии классов для обеспечения возможности полиморфного создания (см. Introduce Polymorphic Creation with

Factory Method на с. 116). Метод фабрики должен быть определен и реализован в некотором классе и в одном или нескольких его подклассах. Такой класс и каждый из его подклассов выступают в качестве фабрики. Однако нельзя утверждать, что метод фабрики является фабрикой.

Abstract Factory — это "интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов" [12, с. 87]. Абстрактные фабрики разрабатываются таким образом, чтобы их можно было подставить в код во время выполнения программы, так что система должна быть сконфигурирована для использования определенной конкретной реализации абстрактной фабрики. Каждая абстрактная фабрика является фабрикой, но не каждая фабрика абстрактна. Классы фабрик, которые не являются абстрактными фабриками, могут превращаться в абстрактные фабрики, когда возникает необходимость в поддержке создания нескольких семейств связанных или зависимых объектов.

Приведенная далее диаграмма, в которой полужирные линии используются для обозначения методов, создающих объекты, иллюстрирует основные различия между методом фабрики, фабрикой и абстрактной фабрикой.



Мне много раз приходилось сталкиваться с системами, перегруженными шаблонами Factory. Например, избыток фабрик в системе может возникнуть в том случае, если каждый ее объект вместо прямого инстанцирования (например, `new StringNode(...)`) создается с помощью фабрики. Перегруженность этим шаблоном часто возникает, если программисты *всегда* отделяют код клиента от кода, в котором выбираются инстанцируемые классы или способ их инстанцирования. Например, следующий метод `createQuery()` выбирает, какой из двух запрошенных классов будет инстанцирован.

```
public class Query...
  public void createQuery() throws QueryException...
    if (usingSDVersion52()) {
      query = new QuerySD52();
```

```
    ...
  } else {
    query = new QuerySD51();
    ...
  }
```

Для того чтобы устранить в приведенном коде условную логику, можно было бы выполнить его реорганизацию с использованием класса `QueryFactory`.

```
public class Query...
  public void createQuery() throws QueryException...
    query = queryFactory.createQuery();
    ...
```

В данном случае метод `QueryFactory` инкапсулирует выбор конкретного класса запроса для инстанцирования. Однако улучшится ли проект, если использовать такой метод? Он определенно не сумеет собрать рассредоточенное решение воедино, и если этот метод всего лишь отделяет класс `Query` от кода, который инстанцирует один из двух конкретных запросов, то он точно не будет настолько полезным, чтобы оправдать свое существование. Из сказанного можно сделать вывод, что реализовать фабрику следует только тогда, когда она либо действительно улучшает проект, либо позволяет создавать и/или конфигурировать объекты в тех случаях, когда это невозможно при непосредственном инстанцировании.

Преимущества и недостатки

+ Объединяет логику создания и требования инстанцирования/конфигурации.
+ Отделяет клиента от логики создания.
– Усложняет проект в тех случаях, когда возможно прямое инстанцирование.

Механика

Инструкции, представленные в данном разделе, предполагают, что фабрика будет реализована в виде класса, а не интерфейса, реализуемого классом. В том случае, когда требуется интерфейс фабрики, реализуемый классом, в приведенные инструкции необходимо внести лишь небольшие изменения.

1. *Инстанциатор* — это класс, который взаимодействует с *другими классами* при инстанцировании *продукта* (т.е. экземпляра некоторого класса). Если *инстанциатор* не инстанцирует *продукт* при помощи метода создания, измените его, а при необходимости измените также класс *продукта* таким образом, чтобы инстанцирование выполнялось методом создания.

   ✓ Скомпилируйте и протестируйте.

2. Создайте новый класс, который станет вашей *фабрикой*. Назовите его в соответствии с создаваемыми им объектами (например, `NodeFactory` или `LoanFactory`).

   ✓ Скомпилируйте.

3. Примените рефакторинг Move Method [15] для переноса метода создания в фабрику. Если метод создания статический, то после этого перемещения можно сделать его нестатическим.

   ✓ Скомпилируйте.

4. Измените инстанциатор таким образом, чтобы он инстанцировал фабрику и вызывал ее для получения экземпляра класса.

   ✓ Скомпилируйте, протестируйте корректность работы инстанциатора.

   Повторите этот шаг для всех инстанциаторов, которые из-за изменений, выполненных на третьем шаге, могут больше не компилироваться.

5. В настоящее время при инстанцировании все еще используются данные и методы из других классов. Перенесите в фабрику все, что имеет смысл перенести, чтобы она выполняла как можно больше работы по созданию объектов. Это действие может включать перенос инстанцирования самой фабрики и того, кто инстанцирует ее.

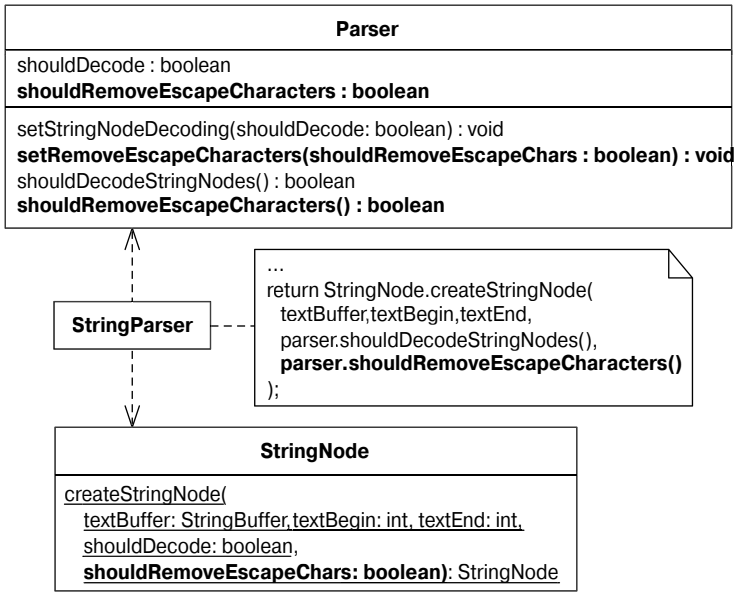
   ✓ Скомпилируйте и протестируйте.

Пример

Идея этого примера возникла из проекта синтаксического анализатора HTML. Как описывается в Move Embellishment to Decorator (с. 174), пользователь анализатора может настраивать анализатор таким образом, чтобы он по-разному обрабатывал строки. Если пользователь не желает, чтобы анализируемые строки содержали закодированные символы, например символ `&` (он обозначает амперсанд — `&`) или символ `<` (обозначает знак <, т.е. меньше), он может вызвать метод анализатора `setStringNodeDecoding(shouldDecode:boolean)`, который включает или выключает опцию кодирования строк. Как показано в схеме, приведенной в начале этого раздела, метод анализатора `StringParser` фактически создает объекты класса `StringNode`, а затем настраивает созданные объекты, определяя, требуется кодирование или нет, на основе значения поля кодирования в классе `Parser`.

Когда я приступил к работе над данным кодом, сведения о создании класса `StringNode` были распределены по классам `Parser`, `StringParser` и `StringNode`. Проблема усугубляется при добавлении в класс `Parser` новых опций анализа строки. Каждая новая опция требует создания нового поля в классе `Parser` с соответствующими функциями установки и получения значения поля, а также нового кода в классах `StringParser` и `StringNode` для работы с новой опцией. В приведенной

далее диаграмме полужирным шрифтом обозначены некоторые изменения, происходящие в классах при добавлении опции удаления специальных символов (например, `\n` или `\r`).



Те поля и функции для их у становки и получения их значений, которые были добавлены в класс `Parser` для поддержки разных опций анализа в классе `StringNode`, не принадлежат собственно этому классу. Почему? Потому что `Parser` отвечает за выполнение анализа, а не за управление тем, каким образом должен быть проанализирован класс `StringNode`, представляющий только один из множества типов `Node` или `Tag`. К тому же нет никаких причин для того, чтобы в классе `StringNode` имелись какие-либо сведения о кодировке или опциях удаления специальных символов, которые уже смоделированы с помощью шаблона Decorator (см. пример для Move Embellishment to Decorator на с. 174).

Если воспользоваться введенным мною ранее определением, то можно сказать, что класс `StringNode` уже представляет собой фабрику, поскольку он реализует метод создания. Проблема в том, что класс `StringNode` не помогает объединить все сведения, используемые при инстанцировании или конфигурации класса `StringNode`. Но это и не нужно, поскольку лучше всего оставить класс `StringNode` простым и небольшим. Новый класс фабрики лучше справится с объединением инстанцирования/конфигурации, поэтому я реорганизую код, создав такой класс. Для простоты в следующем коде содержится только одна опция анализа — опция декодирования — и не содержатся опции для удаления специальных символов.

1. Класс `StringParser` инстанцирует объекты класса `StringNode`. Первый шаг рефакторинга Move Creation Knowledge to Factory заключается в изменении класса `StringParser` таким образом, чтобы он инстанцировал объекты класса `StringNode`, используя для этого метод создания. Как показано в следующем коде, это уже сделано.

```
public class StringParser...
  public Node find(...) {
    ...
    return StringNode.createStringNode(
        textBuffer, textBegin, textEnd,
        parser.shouldDecodeNodes()
    );
  }

public class StringNode...
  public static Node createStringNode(
    StringBuffer textBuffer, int textBegin, int textEnd,
    boolean shouldDecode) {
    if (shouldDecode)
      return new DecodingStringNode(
        new StringNode(textBuffer, textBegin, textEnd)
      );
    return new StringNode(textBuffer, textBegin, textEnd);
  }
```

2. Теперь я создаю новый класс, который станет новой фабрикой для объектов класса `StringNode`. Поскольку класс `StringNode` — это тип `Node`, я называю новый класс `NodeFactory`.

```
public class NodeFactory {
}
```

3. На следующем этапе я применяю рефакторинг Move Method [15] для переноса метода создания в класс `NodeFactory`. Я делаю перенесенный метод нестатическим, поскольку не желаю, чтобы код клиента был статически связан с реализацией фабрики. Кроме этого, я удаляю метод создания из класса `StringNode`.

```
public class NodeFactory {
 public ~~static~~ Node createStringNode(
    StringBuffer textBuffer, int textBegin, int textEnd,
    boolean shouldDecode) {
    if (shouldDecode)
      return new DecodingStringNode(
        new StringNode(textBuffer, textBegin, textEnd));
    return new StringNode(textBuffer, textBegin, textEnd);
  }
```

```
}

public class StringNode...
  public static Node createStringNode(...
  }
```

После этого шага класс `StringParser` и другие клиенты, которые использовали вызов метода создания класса `StringNode`, больше не компилируются. Разберемся с этим позже.

4. Теперь я изменяю класс `StringParser` таким образом, чтобы он инстанцировал класс `NodeFactory` и вызывал его для создания объекта класса `StringNode`.

```
public class StringParser...
  public Node find(...) {
    ...
    NodeFactory nodeFactory = new NodeFactory();
    return nodeFactory.createStringNode(
      textBuffer, textBegin, textEnd,
      parser.shouldDecodeNodes()
    );
  }
```

Аналогичные шаги я выполнил и для остальных клиентов, которые перестали компилироваться после третьего шага.

5. Теперь приступаем к самой интересной части: удаляем или сокращаем рассредоточенность путем перемещения соответствующего кода создания в класс `NodeFactory` из других классов. В данном случае роль других классов выполняет класс `Parser`, который в процессе создания объекта `StringNode` вызывался классом `StringParser` для передачи аргументов в класс `NodeFactory`.

```
public class StringParser...
  public Node find(...) {
    ...
    NodeFactory nodeFactory = new NodeFactory();
      return nodeFactory.createStringNode(
        textBuffer, textBegin, textEnd,
        parser.shouldDecodeNodes()
      );
  }
```

Я считаю, что в класс `NodeFactory` необходимо перенести следующий код класса `Parser`:

```
public class Parser...
  private boolean shouldDecodeNodes = false;
```

```
  public void setNodeDecoding(boolean shouldDecodeNodes) {
    this.shouldDecodeNodes = shouldDecodeNodes;
  }

  public boolean shouldDecodeNodes() {
    return shouldDecodeNodes;
}
```

Однако я не могу просто так перенести этот код в класс `NodeFactory`, поскольку клиенты этого кода являются клиентами анализатора, который вызывает методы класса `Parser` (в частности, `setNodeDecoding(...)`), чтобы настроить анализатор для выполнения данного анализа. Между тем класс `NodeFactory` не будет даже видим клиентам анализатора: он инстанцирован классом `StringParser`, который также для них невидим. Из всего этого можно сделать вывод, что экземпляр класса `NodeFactory` должен быть доступным и для клиентов класса `Parser`, и для класса `StringParser`. С этой целью я выполняю ряд действий.

а) Сначала применяю рефакторинг Extract Class [15] к коду `Parser`, который я в конечном итоге хочу объединить с классом `NodeFactory`. Это приводит к созданию класса `StringNodeParsingOption`.

```
public class StringNodeParsingOption {
  private boolean decodeStringNodes;

  public boolean shouldDecodeStringNodes() {
    return decodeStringNodes;
  }

  public void setDecodeStringNodes(boolean decodeStringNodes) {
    this.decodeStringNodes = decodeStringNodes;
  }
}
```

Этот новый класс заменяет поле `shouldDecodeNodes`, а также функции установки и получения его значения полем `StringNodeParsingOption`, а также соответствующими функциями установки и получения его значения.

```
public class Parser....
 private StringNodeParsingOption stringNodeParsingOption =
    new StringNodeParsingOption();

  private boolean shouldDecodeNodes = false;

  public void setNodeDecoding(boolean shouldDecodeNodes) {
    this.shouldDecodeNodes = shouldDecodeNodes;
  }
```

```
  public boolean shouldDecodeNodes() {
    return shouldDecodeNodes;
  }

 public StringNodeParsingOption
    getStringNodeParsingOption() {
      return stringNodeParsingOption;
  }

  public void setStringNodeParsingOption(
    StringNodeParsingOption option) {
    stringNodeParsingOption = option;
  }
```

Теперь клиенты класса `Parser` включают декодирование в классе `StringNode`, инстанцируя и конфигурируя экземпляр класса `StringNodeParsingOption` и передавая его анализатору.

```
class DecodingNodeTest...
  public void testDecodeAmpersand() {
    ...
    StringNodeParsingOption decodeNodes =
      new StringNodeParsingOption();
    decodeNodes.setDecodeStringNodes(true);
    parser.setStringNodeParsingOption(decodeNodes);
    parser.setNodeDecoding(true);
    ...
  }
```

Теперь класс `StringParser` получает состояние опции кодирования `StringNode` через новый класс.

```
public class StringParser...
  ...
  public Node find(...) {
    NodeFactory nodeFactory = new NodeFactory();
    return nodeFactory.createStringNode(
      textBuffer, textBegin, textEnd,
      parser.getStringNodeParsingOption()
      .shouldDecodeStringNodes()
    );
  }
```

б) На этом этапе я применяю рефакторинг Inline Class [15] для объединения класса `NodeFactory` с классом `StringNodeParsingOption`. Это приводит к следующим изменениям в классе `StringParser`:

```
public class StringParser...
  public Node find(...) {
    ...
```

```
    return
      parser.getStringNodeParsingOption().createStringNode(
             textBuffer, textBegin, textEnd,
             parser.getStringNodeParsingOption()
             .shouldDecodeStringNodes()
      );
  }
```

и к изменениям в классе StringNodeParsingOprion:

```
public class StringNodeParsingOption...
  private boolean decodeStringNodes;
  public Node createStringNode(
    StringBuffer textBuffer, int textBegin, int textEnd,
    boolean shouldDecode) {
    if (decodeStringNodes)
      return new DecodingStringNode(
        new StringNode(textBuffer, textBegin, textEnd));
    return new StringNode(textBuffer, textBegin, textEnd);
    }
..}
```

в) Последний этап — переименовать класс StringNodeParsingOprion в NodeFactory, после чего выполнить аналогичное переименование поля NodeFactory, а также функций установки и получения его значения в классе Parser.

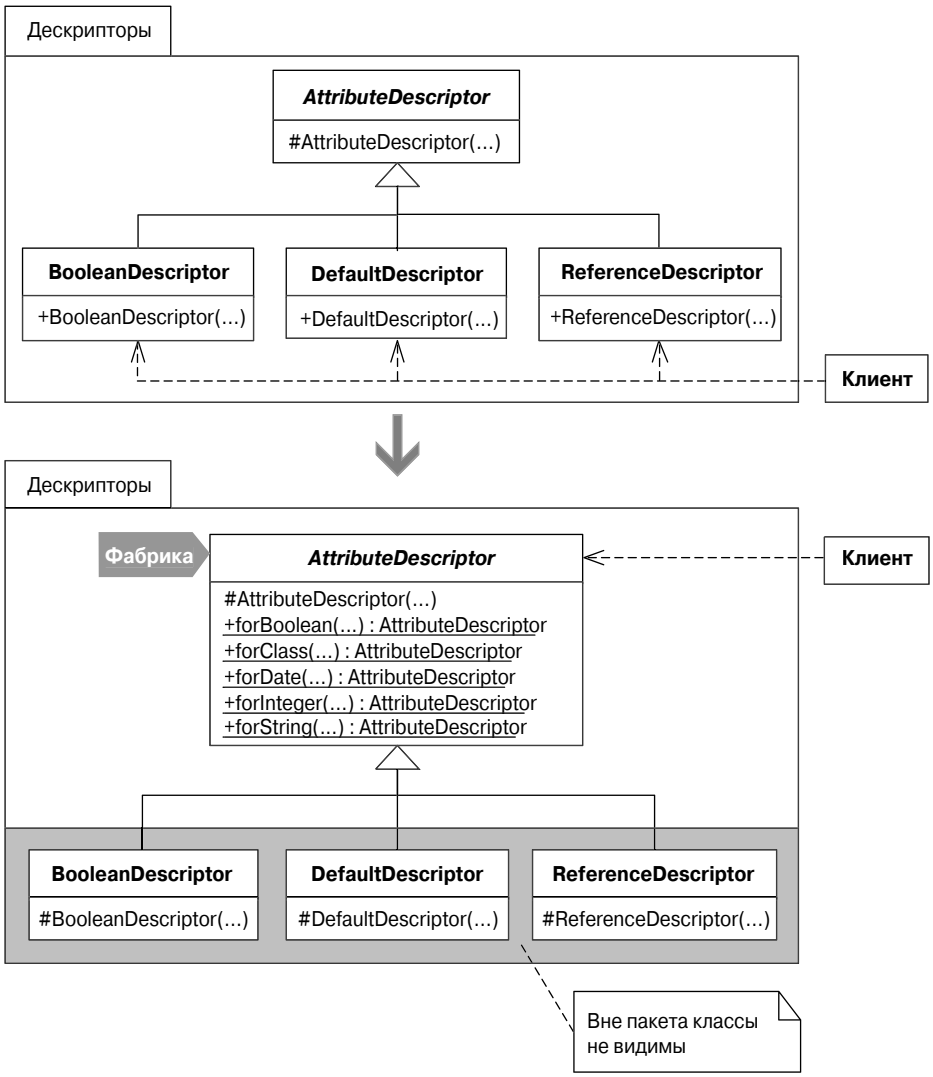
```
public class StringNodeParsingOption NodeFactory...

public class Parser...
  private NodeFactory nodeFactory = new NodeFactory();

  public NodeFactory getNodeFactory() {
    return nodeFactory;
  }

  public void setNodeFactory(NodeFactory nodeFactory) {
    this.nodeFactory = nodeFactory;
  }
```

Реорганизация кода выполнена. Класс NodeFactory помог преодолеть "размазанность" решения, беря на себя работу, связанную с реализацией и конфигурацией объектов StringNode.

Encapsulate Classes with Factory

Клиенты непосредственно инстанцируют классы,
которые находятся в одном пакете и реализуют общий интерфейс.

Сделать конструкторы классов закрытыми и разрешить клиентам создавать экземпляры классов с использованием фабрики.



Мотивация

Способность клиента непосредственно инстанцировать классы полезна лишь при том условии, что ему действительно необходимо знать о точном существовании этих классов. Но что, если клиент в действительности не нуждается в этих сведениях? Что делать, если эти классы "живут" в некотором пакете и реализуют один интерфейс и эти условия не собираются меняться? В этом случае классы, содержащиеся в пакете, можно сделать скрытыми от внешних клиентов, передавая фабрике ответственность за создание и возврат экземпляров, которые реализуют общий интерфейс.

В пользу данного рефакторинга существует несколько аргументов. Во-первых, таким образом обеспечивается строгое выполнение принципа "программировать интерфейсы, а не реализации" [12], поскольку гарантируется, что клиенты будут взаимодействовать с классами через общий интерфейс. Во-вторых, обеспечивается уменьшение "концептуального веса" [10] пакета путем сокрытия классов, которые не должны быть видимыми за пределами своего пакета (т.е. клиенты не обязаны знать о существовании таких классов). И в-третьих, упрощается создание доступных *видов* экземпляров путем построения множества методов создания с ясным назначением, доступного посредством фабрики.

Основная проблема, возникающая в связи с применением данного рефакторинга — это циклическая зависимость: каждый раз, создавая новый подкласс или добавляя/изменяя конструктор существующего подкласса, необходимо также добавлять новый метод создания в фабрику. Если создание нового подкласса или добавление/изменение конструктора в существующем подклассе происходит не часто, то проблемы нет. Но в противном случае у вас может возникнуть желание уклониться от применения данного рефакторинга или перейти к дизайну, позволяющему клиентам непосредственно инстанцировать подклассы, которые им нужны. Кроме того, можно рассмотреть гибридный подход, при котором вы создаете фабрику для некоторых наиболее распространенных видов экземпляров и не полностью инкапсулируете все подклассы, что позволяет клиентам при необходимости инстанцировать классы.

Программисты, предоставляющие доступ к бинарному коду, а не к исходному тексту, также могут избегать применения данного рефакторинга, поскольку он не позволяет клиентам изменять инкапсулированные классы или методы создания в фабрике.

С помощью этого рефакторинга можно получить класс, который ведет себя и как фабрика, и как класс реализации (т.е. класс, реализующий методы, не основанные на создании). Для некоторых, хотя и не для всех, такая композиция удобна. Если вы считаете, что данная композиция лишь скрывает основную ответственность класса, рекомендую рассмотреть применение рефакторинга Extract Factory (с. 94).

Схема, представленная в начале этого раздела, призвана прояснить отображение объектов в реляционные базы данных. До применения данного рефакторинга некоторые программисты (включая меня) случайно инстанцируют ошибочный подкласс или правильный подкласс с некорректными аргументами (например, вызывается конструктор, получающий `int`, в то время как на самом деле необходимо вызвать

конструктор, который получает `Integer`). Рассматриваемый рефакторинг сокращает количество ошибок, инкапсулируя сведения о подклассах и создавая единое место для получения множества экземпляров подкласса с поясняющими именами.

Преимущества и недостатки

+ Упрощает создание видов экземпляров, создавая множество доступных экземпляров через методы с ясным назначением.
+ Сокращает "концептуальный вес" пакета [10], скрывая классы, которые не должны быть открытыми.
+ Помогает выполнять принцип "программировать интерфейс, а не реализацию" [12].
– Требует новых или обновления старых методов создания каждый раз, когда необходимо создать новый вид экземпляра.
– Ограничивает настройку в том случае, если клиент может получить доступ только к бинарному (а не к исходному) коду фабрики.

Механика

Обычно необходимость в применении этого рефакторинга возникает в тех случаях, когда классы совместно используют общий открытый интерфейс, произведены от одного надкласса и располагаются в одном пакете.

1. Найдите клиент, который вызывает конструктор класса для создания экземпляра некоторого вида. Примените рефакторинг Extract Method [15] к вызову конструктора для создания открытого статического метода. Такой новый метод представляет собой *метод создания*. Теперь примените рефакторинг Move Method [15] для переноса созданного метода в надкласс класса с выбранным конструктором.
   ✓ Скомпилируйте и протестируйте.
2. Найдите все вызывающие функции для выбранного конструктора, которые инстанцируют тот же вид экземпляров, что и метод создания. Измените их таким образом, чтобы они вызывали метод создания.
   ✓ Скомпилируйте и протестируйте.
3. Повторите шаги 1 и 2 для остальных видов экземпляров, которые могут быть созданы конструктором класса.
4. Объявите конструктор класса закрытым.
   ✓ Скомпилируйте.
5. Повторите шаги 1–4 для всех классов, которые вы хотите инкапсулировать.

Пример

Следующий пример основан на коде отображения объектов, которые используются при записи и считывании объектов из реляционных баз данных.

1. Я начинаю с небольшой иерархии классов, которые располагаются в пакете с именем `descriptors`. Эти классы принимают участие в отображении атрибутов базы данных в экземпляры переменных объектов.

```
package descriptors;

public abstract class AttributeDescriptor...
  protected AttributeDescriptor(...)

public class BooleanDescriptor extends AttributeDescriptor...
  public BooleanDescriptor(...) {
    super(...);
  }

public class DefaultDescriptor extends AttributeDescriptor...
  public DefaultDescriptor(...) {
    super(...);
  }

public class ReferenceDescriptor
  extends AttributeDescriptor...
  public ReferenceDescriptor(...) {
    super(...);
  }
```

Абстрактный конструктор класса `AttributeDescriptor` защищенный, а конструкторы трех его подклассов открыты. В данном примере я показываю только три подкласса класса `AttributeDescriptor`, хотя в действительности их около десяти.

Я концентрируюсь на подклассе `DefaultDescriptor`. Первый шаг — определить вид экземпляра, который может быть создан конструктором подкласса `DefaultDescriptor`. Для этого я рассматриваю некоторый код клиента.

```
protected List createAttributeDescriptors() {
  List result = new ArrayList();
  result.add(new DefaultDescriptor("remoteId",
                   getClass(), Integer.TYPE));
  result.add(new DefaultDescriptor("createdDate",
                   getClass(), Date.class));
  result.add(new DefaultDescriptor("lastChangedDate",
                   getClass(), Date.class));
  result.add(new ReferenceDescriptor("createdBy",
```

```
                    getClass(), User.class,
                    RemoteUser.class));
  result.add(new ReferenceDescriptor("lastChangedBy",
                    getClass(), User.class,
                    RemoteUser.class));
  result.add(new DefaultDescriptor("optimisticLockVersion",
                    getClass(), Integer.TYPE));
  return result;
}
```

В рассмотренном коде класс `DefaultDescriptor` используется для представления отображений типов `Integer` и `Date`. В то же время его можно применять и для отображения других типов, так что я должен концентрироваться на видах экземпляров по одному. Я решаю построить метод создания, который будет создавать дескрипторы атрибутов для типов `Integer`. Для этого я сначала применяю рефакторинг Extract Method [15], с помощью которого создаю открытый статический метод `forInteger(…)`.

```
protected List createAttributeDescriptors()...
  List result = new ArrayList();
  result.add(forInteger("remoteId", getClass(),
                        Integer.TYPE));
  ...

public static DefaultDescriptor forInteger(...) {
  return new DefaultDescriptor(...);
}
```

Поскольку метод `forInteger(…)` всегда создает объекты класса `AttribudeDescriptor` типа `Integer`, нет необходимости передавать ему значение `Integer.TYPE`.

```
protected List createAttributeDescriptors()...
  List result = new ArrayList();
  result.add(forInteger("remoteId", getClass(),
                        Integer.TYPE));
  ...

public static DefaultDescriptor forInteger(...) {
  return new DefaultDescriptor(..., Integer.TYPE);
}
```

Кроме этого, я изменяю возвращаемый тип метода `forInteger(…)` с `DefaultDescriptor` на `AttributeDescriptor`, поскольку хочу, чтобы клиенты взаимодействовали со всеми подклассами класса `AttributeDescriptor` через интерфейс класса `AttributeDescriptor`.

```
public static AttributeDescriptor DefaultDescriptor-
              forInteger(...)...
```

После этого я переношу метод `forInteger(…)` в класс `AttributeDescriptor`, применяя для этого рефакторинг Move Method [15].

```
public abstract class AttributeDescriptor {
  public static AttributeDescriptor forInteger(...) {
    return new DefaultDescriptor(...);
  }
```

Теперь код клиента выглядит так:

```
protected List createAttributeDescriptors()...
  List result = new ArrayList();
  result.add(AttributeDescriptor.forInteger(...));
  ...
```

Я компилирую и тестирую код, проверяя, все ли работает правильно.

2. Теперь я нахожу остальные вызывающие функции для конструктора подкласса `DefaultDescriptor`, которые создают класс `AttributeDescriptor` для типа `Integer`. Я изменяю эти функции таким образом, чтобы они вызывали новый метод создания.

```
protected List createAttributeDescriptors() {
  List result = new ArrayList();
  result.add(AttributeDescriptor.forInteger("remoteId",
                                            getClass()));
  ...
  result.add(AttributeDescriptor.forInteger(
                    "optimisticLockVersion", getClass()));
  return result;
}
```

Я компилирую и тестирую код. Все работает правильно.

3. Теперь я повторяю шаги 1 и 2 по мере того, как продолжаю строить методы создания для остальных видов экземпляров, которые может создавать конструктор подкласса `DefaultDescriptor`. Это приводит к еще двум методам создания.

```
public abstract class AttributeDescriptor {
  public static AttributeDescriptor forInteger(...) {
    return new DefaultDescriptor(...);
  }

  public static AttributeDescriptor forDate(...) {
    return new DefaultDescriptor(...);
```

```
  }

  public static AttributeDescriptor forString(...) {
    return new DefaultDescriptor(...);
  }
```

4. Теперь я объявляю конструктор `DefaultDescriptor` защищенным.

```
public class DefaultDescriptor extends AttributeDescriptor {
  protected DefaultDescriptor(...) {
    super(...);
  }
```

   Я компилирую код. Пока все идет по плану.

5. Я повторяю шаги 1–4 для остальных подклассов класса `AttributeDescriptor`. После выполнения этих действий новый код делает следующее:

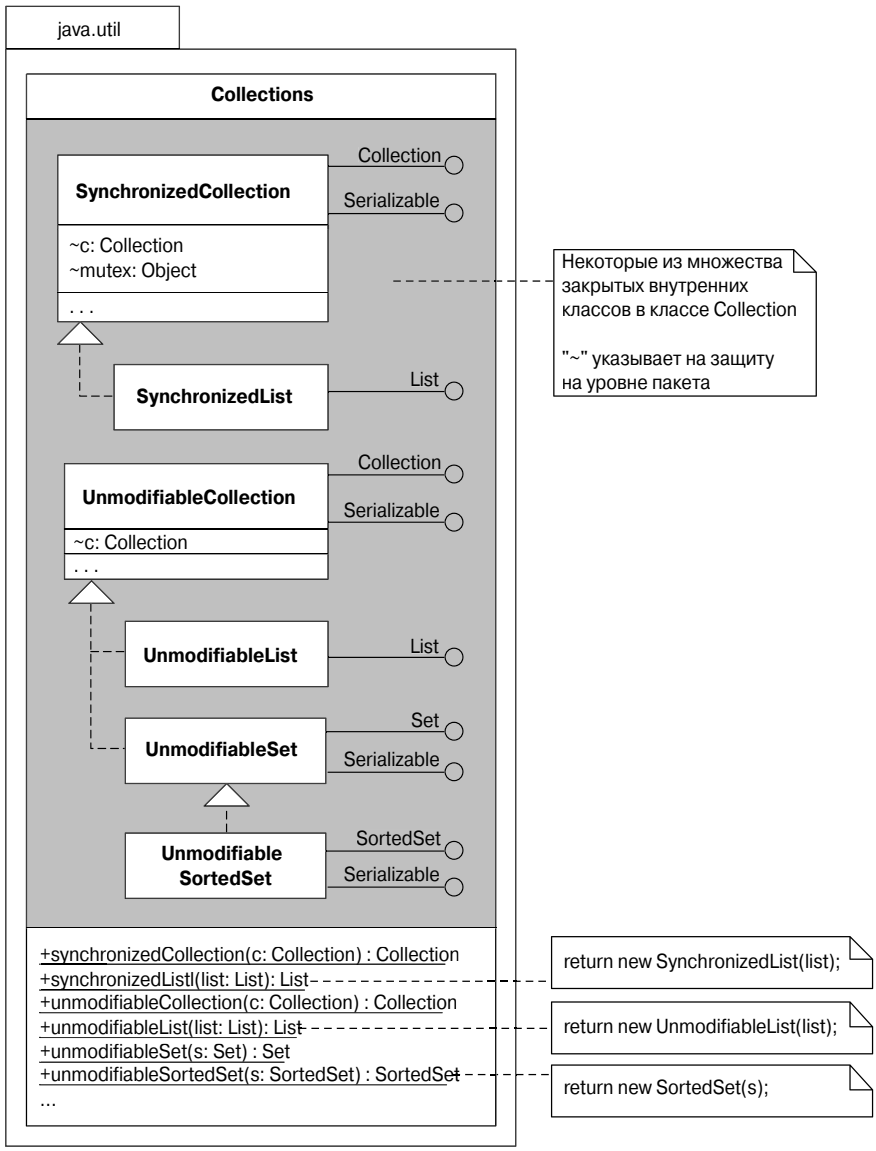
   - предоставляет доступ к подклассам `AttributeDescriptor` через их надкласс;
   - гарантирует, что клиент получает экземпляры подкласса через интерфейс `AttributeDescriptor`;
   - предотвращает непосредственное инстанцирование клиентами подклассов класса `AttributeDescriptor`;
   - сообщает остальным программистам о том, что подклассы класса `AttributeDescriptor` не должны быть открытыми. Клиенты взаимодействуют с экземплярами подкласса через общий интерфейс.

Разновидности

Инкапсулированные внутренние классы

В классе Java `java.util.Collections` содержится замечательный пример, описывающий суть процесса инкапсуляции классов с Creation Method. Автор этого класса Джошуа Блок должен был предоставить программистам способ создания неизменяемых и/или синхронизируемых коллекций, списков, множеств и отображений. Для выполнения этого задания он благоразумно выбрал использование защищенной формы шаблона Proxy [12]. Впрочем, вместо того чтобы создавать общедоступные прокси-классы `java.util` (для управления синхронизацией и неизменяемостью), а затем ожидать, что программисты самостоятельно защитят свои коллекции, он определил в классе `Collections` прокси-классы в виде закрытых внутренних классов. После такого определения он предоставил в классе `Collections` множество методов создания, из которых каждый программист может получить необходимый ему вид прокси-класса. В приведенной далее схеме показаны некоторые из внутренних классов и методов создания, определенных в классе `Collections`.

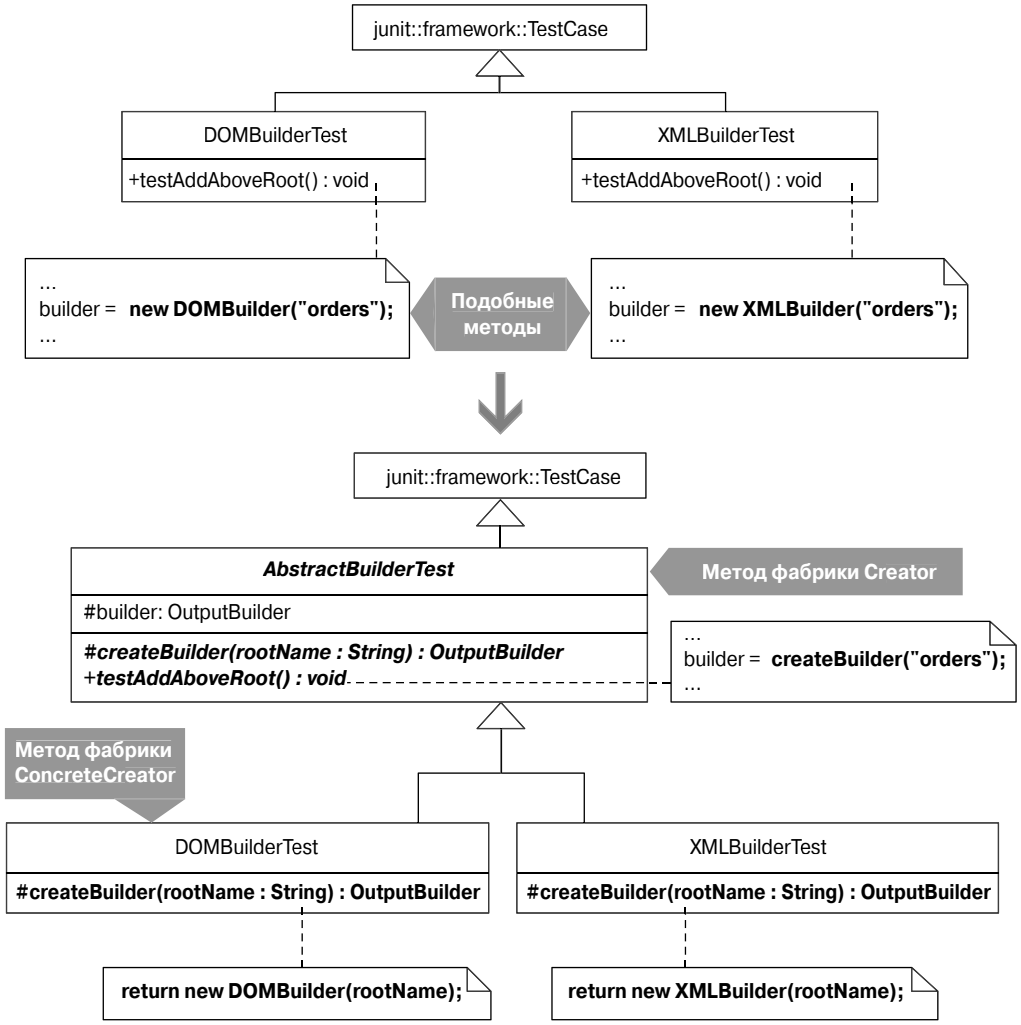
Следует отметить, что класс `java.util.Collections` содержит даже небольшую иерархию внутренних классов, каждый из которых является закрытым. Каждый внутренний класс имеет соответствующий метод, который принимает коллекцию, защищает ее и после этого возвращает уже защищенный экземпляр, используя для этого общий тип интерфейса (такой, как `List` или `Set`). С помощью такого решения уменьшается количество классов, про которые должны знать программисты, и в то же время обеспечивается необходимая функциональность. Класс `java.util.Collections` является также примером фабрики.



Introduce Polymorphic Creation with Factory Method

Классы иерархии одинаково реализуют метод,
за исключением этапа создания объекта.

*Создать единственную версию метода в надклассе,
которая вызывает метод фабрики для создания объектов.*



Мотивация

Для построения метода создания (см. Replace Constructors with Creation Methods на с. 84) класс должен реализовать статический или нестатический метод, который инстанцирует и возвращает объект. С другой стороны, для того чтобы построить метод фабрики, необходимо следующее:

- тип (определяемый интерфейсом, абстрактным классом или классом), указывающий множество классов, которое могут инстанцироваться и возвращаться реализациями метода фабрики;
- множество классов, реализующих данный тип;
- несколько классов, реализующих метод фабрики и локально принимающих решение о том, какие классы из заданного множества будут инстанцироваться, инициализироваться и возвращаться.

Может показаться, что все это слишком сложно, хотя на самом деле из всех объектно-ориентированных программ самые распространенные — это как раз методы фабрик. Причина в том, что они предоставляют способ полиморфного создания объектов.

На практике методы фабрик обычно реализованы внутри иерархии классов, однако они также могут быть реализованы классами, которые просто совместно используют общий интерфейс. Обычно абстрактный класс или объявляет метод фабрики и заставляет подклассы перекрывать его, или обеспечивает реализацию по умолчанию метода фабрики и позволяет подклассам наследовать или перекрывать ее.

Поскольку сигнатура метода фабрики должна быть одинаковой для всех реализаций, может возникнуть необходимость передачи лишних параметров некоторым реализациям метода фабрики. Например, если один подкласс запрашивает для создания объектов аргументы `int` и `double`, в то время как другой подкласс запрашивает только `int`, то метод фабрики, реализованный этими двумя подклассами, должен получать как аргумент типа `int`, так и аргумент типа `double`. Поскольку в одном из подклассов тип `double` не нужен, внешний вид кода может оказаться несколько запутанным.

Методы фабрики часто вызываются шаблоном Template Method [12]. Сотрудничество этих двух шаблонов часто превращается в иерархии классов, по мере того как выполняется реорганизация, устраняющая дублирование кода.

Можно ли утверждать, что шаблон Factory Method — это более простой путь, чем вызов оператора `new` или метода создания? Реализация такого шаблона определенно не проще указанных методов. Тем не менее обычно окончательный код, получаемый при помощи шаблона Factory Method, оказывается проще, чем код, дублирующий методы в нескольких классах только для того, чтобы выполнить создание пользовательского объекта.

Преимущества и недостатки

+ Уменьшает дублирование кода, возникающее на шаге создания пользовательских объектов.
+ Эффективно указывает, где именно выполняется создание и как оно может быть перекрыто.
+ Указывает, какой тип класса должен быть реализован для использования методом фабрики.
– Может возникнуть необходимость в передаче лишних параметров в некоторые реализации метода фабрики.

Механика

Рассматриваемый рефакторинг чаще всего используется в следующих случаях:

- если родственные подклассы[1] реализуют метод одинаковым способом, за исключением этапа создания объекта;
- если надкласс и подкласс реализуют метод одинаковым способом, за исключением этапа создания объекта.

В механике, представленной в данном разделе, рассмотрен сценарий родственных подклассов. Этот сценарий легко адаптируется для варианта надкласса и подкласса. При описании механики метод, который реализуется одинаковым способом, за исключением этапа создания объекта, будет называться *подобным методом* (similar method).

1. В подклассе, который содержит подобный метод, измените последний таким образом, чтобы при создании пользовательского объекта вызывался *метод инстанцирования* (instantiation method). Вы всегда можете это сделать, применив рефакторинг Extract Method [15] к коду создания или же реорганизовав код таким образом, чтобы он вызывал ранее выделенный метод инстанцирования.

   Используйте для метода инстанцирования обобщенное имя (например, `createBuilder` или `newProduct`), поскольку то же имя метода будет использоваться в подобных методах родственных подклассов. Измените возвращаемый тип для метода инстанцирования таким образом, чтобы этот тип был общим для логики пользовательского инстанцирования в подобных методах родственных подклассов.

   ✓ Скомпилируйте и протестируйте.

[1] Под этим термином подразумеваются подклассы одного надкласса, т.е. можно говорить об отношении не просто родства, но конкретно “братства”. Поскольку термин “братские кассы” менее благозвучен, в этом разделе он заменен термином “родственные классы”. — *Примеч. ред.*

2. Повторите шаг 1 для подобного метода в родственных подклассах. Это приведет к созданию метода инстанцирования для каждого из родственных подклассов, причем сигнатура метода инстанцирования в каждом родственном подклассе должна быть одинаковой.

   ✓ Скомпилируйте и протестируйте.

3. Теперь измените надкласс родственных подклассов. Если вы не можете изменить этот класс или его изменение нежелательно, примените рефакторинг Extract Superclass [15] для получения надкласса, который будет наследником исходного надкласса и родительским по отношению к исходным подклассам.

   ✓ Скомпилируйте и протестируйте.

4. Примените к подобному методу рефакторинг Form Template Method [15]. Это, в свою очередь, вызовет применение рефакторинга Pull Up Method [15]. Выполняя такой рефакторинг, убедитесь, что вы реализуете следующий совет из раздела *Механика* рефакторинга Pull Up Method:

   > Если вы работаете со строго типизированным языком и [метод, с которым вы работаете] вызывает другой метод, который имеется во всех подклассах, но отсутствует в надклассе, объявите в надклассе абстрактный метод [15, с. 323].

   Один такой абстрактный метод, который вы объявите в надклассе, будет вашим методом инстанцирования. Объявив этот абстрактный метод, вы реализуете *метод фабрики*. Теперь каждый из родственных подклассов представляет собой Factory Method: ConcreteCreator [12].

   ✓ Скомпилируйте и протестируйте.

5. Повторите шаги 1–4, если в родственных подклассах содержатся подобные дополнительные методы, которые могут получить преимущества от вызова ранее созданного метода фабрики.

6. Если метод фабрики в большинстве классов содержит одинаковый код инстанцирования, переместите этот код в надкласс, изменив объявление абстрактного метода фабрики в надклассе на конкретный метод, который выполняет действия по созданию объектов, выполняемые в большинстве подклассов (“основной случай”).

   ✓ Скомпилируйте и протестируйте.

Пример

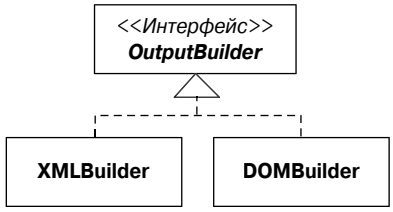
В одном из своих проектов я использовал управляемую тестами разработку для программы `XMLBuilder`, которая позволяет клиентам легко создавать XML-код. После этого я подумал, что стоит создать класс `DOMBuilder`, который ведет себя

подобно XMLBuilder, но генерирует XML путем создания DOM (Document Object Model — объектная модель документа) и предоставляя клиентам доступ к ней.

Для создания класса DOMBuilder я использовал уже написанные для XMLBuilder тесты. Необходимо было внести только одно изменение в каждом тесте: инстанцировать класс DOMBuilder вместо класса XMLBuilder.

```
public class DOMBuilderTest extends TestCase...
  private OutputBuilder builder;

  public void testAddAboveRoot() {
    String invalidResult =
    "<orders>" +
      "<order>" +
      "</order>" +
    "</orders>" +
    "<customer>" +
    "</customer>";
    builder = new DOMBuilder("orders"); //Используется
                        // вместо XMLBuilder("orders")
    builder.addBelow("order");
    try {
      builder.addAbove("customer");
      fail("expecting java.lang.RuntimeException");
    } catch (RuntimeException ignored) {}
}
```

Ключевая цель заключалась в том, чтобы классы DOMBuilder и XMLBuilder совместно использовали один и тот же тип — OutputBuilder, как показано в следующей диаграмме.



После написания класса DOMBuilder у меня было девять тестовых методов, практически идентичных для классов XMLBuilder и DOMBuilder. Кроме того, в классе DOMBuilder имелись и свои особые тесты, которые проверяли доступ и содержание DOM. Меня абсолютно не радовало существование такого дублирования тестов, поскольку, решив внести изменения в класс XMLBuilder, я должен был бы вносить аналогичные изменения и в класс DOMBuilder. Я понял, что наступил момент для выполнения рефакторинга с помощью Factory Method. Далее приводится описание проделанной мною работы.

1. Первым подобным методом был тестирующий метод `testAddAboveRoot()`. Я перенес его логику инстанцирования в метод инстанцирования.

```
public class DOMBuilderTest extends TestCase...
  protected OutputBuilder createBuilder(String rootName) {
    return new DOMBuilder(rootName);
  }

  public void testAddAboveRoot() {
    String invalidResult =
    "<orders>" +
     "<order>" +
     "</order>" +
    "</orders>" +
    "<customer>" +
    "</customer>";
    builder = createBuilder("orders");
    builder.addBelow("order");
    try {
      builder.addAbove("customer");
      fail("expecting java.lang.RuntimeException");
    } catch (RuntimeException ignored) {}
  }
```

Обратите внимание, что возвращаемый тип нового метода `createBuilder(…)` — класс `OutputBuilder`. Я использую этот возвращаемый тип, поскольку в рассматриваемом родственном подклассе `XMLBuilderTest` потребуется определять собственный метод `createBuilder(…)` (он будет создан на следующем шаге) и сигнатура метода инстанцирования в этих двух классах одинакова.

Я компилирую код и запускаю тесты, проверяя, что после изменений все продолжает работать.

2. Теперь я повторяю шаг 1 для всех остальных родственных подклассов. В данном случае это только класс `XMLBuilderTest`.

```
public class XMLBuilderTest extends TestCase...
  private OutputBuilder createBuilder(String rootName) {
    return new XMLBuilder(rootName);
  }
  public void testAddAboveRoot() {
  String invalidResult =
  "<orders>" +
    "<order>" +
    "</order>" +
  "</orders>" +
  "<customer>" +
  "</customer>";
```

```
  builder = createBuilder("orders");
  builder.addBelow("order");
  try {
    builder.addAbove("customer");
    fail("expecting java.lang.RuntimeException");
  } catch (RuntimeException ignored) {}
}
```

Я компилирую и тестирую код, проверяя, что тесты продолжают работать.

3. На этом этапе мне нужно изменить надкласс для тестов. Но рассматриваемый надкласс — это класс `TestCase`, который представляет собой часть инструментария JUnit. Я не хочу изменять этот надкласс, поэтому применяю рефакторинг Extract Superclass [15], создавая класс `AbstractBuilderTest` — новый надкласс для подклассов тестов.

```
public class AbstractBuilderTest extends TestCase {
}

public class XMLBuilderTest extends AbstractBuilderTest...

public class DOMBuilderTest extends AbstractBuilderTest...
```

4. Теперь я могу применить рефакторинг Form Template Method (с. 239). Поскольку на данном этапе подобные методы в классах `XMLBuilderTest` и `DOMBuilderTest` идентичны, в соответствии с механикой этого рефакторинга я применяю рефакторинг Pull Up Method [15] к методу `testAddAboveRoot()`. В свою очередь, это приводит меня к рефакторингу Pull Up Field [15] для поля `builder`.

```
public class AbstractBuilderTest extends TestCase {
  protected OutputBuilder builder;
}

public class XMLBuilderTest extends AbstractBuilderTest...
  private OutputBuilder builder;

public class DOMBuilderTest extends AbstractBuilderTest...
  private OutputBuilder builder;
```

Продолжая следовать механике рефакторинга Pull Up Method [15], применяемого к методу `testAddAboveRoot()`, я должен объявить в качестве абстрактного метода надкласса каждый метод, который вызывается методом `testAddAboveRoot()` и имеется в классах `XMLBuilderTest` и `DOMBuilderTest`. Таким методом является `createBuilder(...)`, поэтому я исправляю его описание, объявляя этот метод абстрактным.

```
public abstract class AbstractBuilderTest extends TestCase{
  protected OutputBuilder builder;

  protected abstract OutputBuilder
        createBuilder(String rootName);
}
```

Теперь я могу продолжить, перенося метод `testAddAboveRoot()` в класс `AbstractBuilderTest`.

```
public abstract class AbstractBuilderTest extends TestCase...
  public void testAddAboveRoot() {
    String invalidResult =
    "<orders>" +
      "<order>" +
      "</order>" +
    "</orders>" +
    "<customer>" +
    "</customer>";
    builder = createBuilder("orders");
    builder.addBelow("order");
    try {
      builder.addAbove("customer");
      fail("expecting java.lang.RuntimeException");
    } catch (RuntimeException ignored) {}
  }
```

На этом этапе метод `testAddAboveRoot()` удаляется из классов `XMLBuilderTest` и `DOMBuilderTest`. Метод `createBuilder(…)`, который теперь объявлен в классе `AbstractBuilderTest` и реализован в классах `XMLBuilderTest` и `DOMBuilderTest`, реализует шаблон Factory [15].

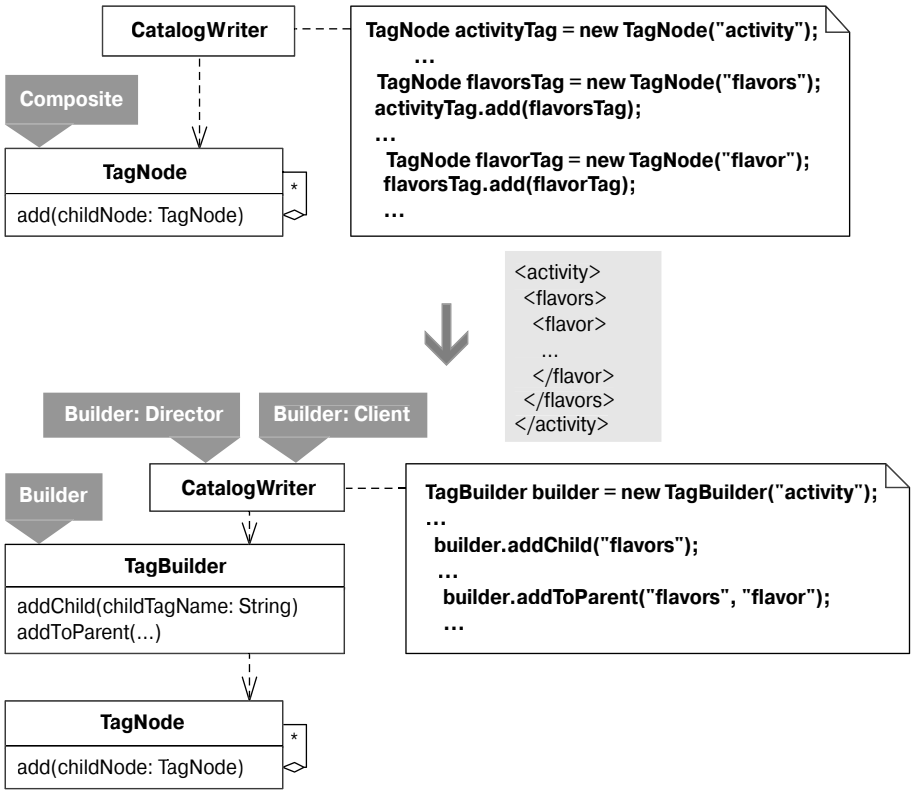
Как обычно, я компилирую код и проверяю тесты, убеждаясь, что все продолжает работать.

5. Я повторяю шаги 1–4 для каждого подобного метода до тех пор, пока такие методы присутствуют в классах `XMLBuilderTest` и `DOMBuilderTest`.
6. На этом этапе я рассматриваю создание в классе `AbstractBuilderTest` реализации метода `createBuilder(…)` по умолчанию. Необходимость в таком создании есть только в одном случае: если это поможет удалить дублирующийся код в многократных реализациях метода `createBuilder(…)` в подклассах. В данном случае, поскольку каждый из классов `XMLBuilderTest` и `DOMBuilderTest` инстанцирует свой собственный вид класса `OutputBuilder`, такой необходимости нет. Следовательно, на этом реорганизация кода завершена.

Encapsulate Composite with Builder

Построение сложного объекта Composite приводит к повторениям, усложнению или подвержено ошибкам.

Упростить построение, позволив шаблону Builder *управлять деталями.*



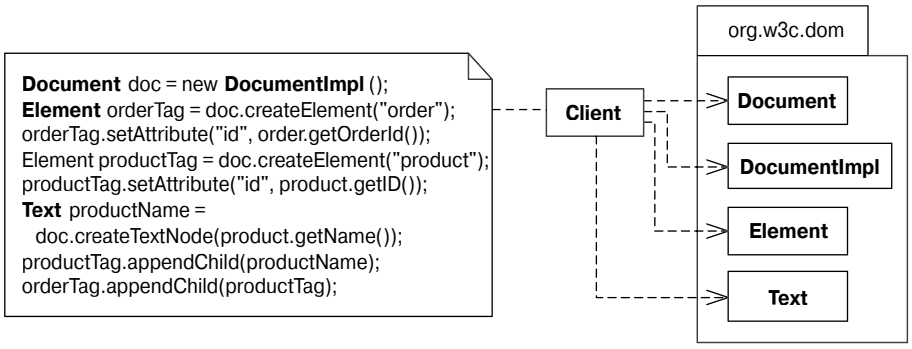
Шаблон Builder [12] выполняет трудоемкие и сложные этапы построения от лица клиента. Основная причина рефакторинга к шаблону Builder заключается в упрощении клиентского кода, который создает сложные объекты. Если сложные и трудоемкие этапы создания выполняются с помощью шаблона Builder, то клиент может управлять этим процессом, не имея сведений о том, как именно он происходит.

Построители часто инкапсулируют классы Composite [12], потому что их создание обычно приводит к повторениям и представляет собой сложный процесс, подверженный ошибкам. Например, для того чтобы добавить дочерний узел в родительский, клиент должен выполнить следующие действия:

- инстанцировать новый узел;
- инициализировать этот новый узел;
- корректно добавить новый узел в правильный родительский узел.

Этот процесс подвержен ошибкам, поскольку можно легко забыть добавить новый узел в родительский или добавить его в неправильный родительский узел. Кроме того, он подвержен повторениям, поскольку снова и снова требует выполнения одинаковых этапов построения. Поэтому было бы неплохо реорганизовать этот код к шаблону Builder, что может уменьшить количество ошибок или минимизировать и упростить этапы создания.

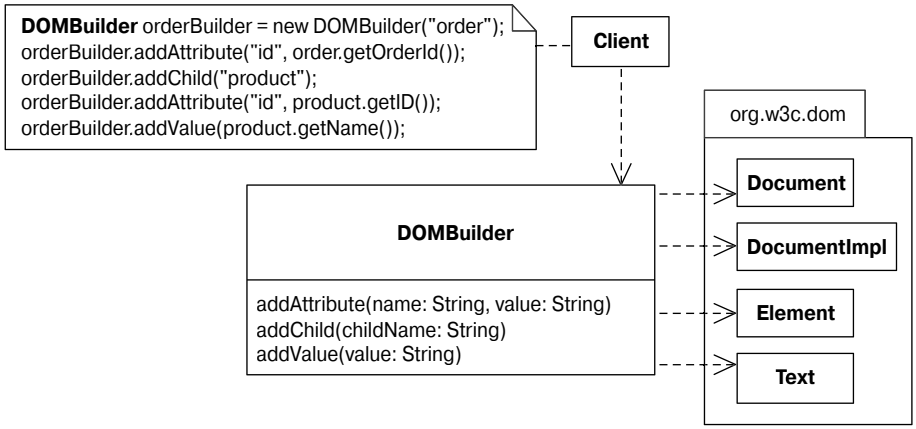
Другой причиной для инкапсуляции Composite шаблоном Builder может быть отделение кода клиента от кода Composite. Например, в коде, представленном в приведенной ниже диаграмме, показано, как тесно связано создание объекта `orderTag` с классами и интерфейсами DOM `Document`, `DocumentImpl`, `Element` и `Text`.



Вследствие такой тесной связи возникают трудности при изменении реализации шаблона Composite. Например, в некоем проекте необходимо усовершенствовать систему таким образом, чтобы в ней использовалась более новая версия стандарта DOM, а эта версия содержит некоторые отличия от версии 1.0, которая использовалась до этого. Такое усовершенствование влечет за собой изменение множества строк создающего Composite кода, разбросанных по всей системе. Частью этого усовершенствования будет инкапсуляция кода нового стандарта DOM в классе `DOMBuilder`, что показано в приведенной ниже диаграмме.

Все методы класса `DOMBuilder` принимают в качестве аргумента строку и возвращают пустой тип. В интерфейсе класса `DOMBuilder` нет упоминания об интерфейсах или классах DOM, однако класс `DOMBuilder` во время выполнения программы проводит внутреннюю сборку объектов DOM. Код клиента, использующий класс `DOMBuilder`, слабо связан с кодом DOM. Это хорошо, поскольку, когда выйдет новая версия DOM либо будут использоваться JDOM или наши собственные объекты `TagNode`, можно легко создать новый шаблон Builder, который определяет интерфейс, идентичный интерфейсу `DOMBuilder`. Кроме того, наличие некоторого обобщенного

интерфейса Builder позволяет настраивать систему таким образом, чтобы в ней можно было использовать любую реализацию Builder, необходимую в данном контексте.



В книге *Design Patterns* назначение шаблона Builder описывается так: "Отделить создание сложного объекта от его внутреннего представления, таким способом позволяя одному и тому же процессу создавать разные представления" [12, с. 97].

"Создание разных представлений" сложного объекта — это полезный, но не единственный сервис, который предоставляет шаблон Builder. Как уже упоминалось, причины, одинаково достаточные для использования шаблона Builder, — это упрощение конструирования или отделение кода клиента от сложного объекта.

От интерфейса Builder ожидается, что он будет показывать свое назначение настолько отчетливо, чтобы любой, кто читает данный код, мог немедленно понять, что он делает. На практике интерфейс шаблона Builder (или часть его интерфейса) может и не быть достаточно прозрачным, поскольку для упрощения конструирования большую часть своей работы шаблоны Builder выполняют "за сценой". Это означает, что для полного понимания возможностей шаблона вам, вероятно, придется изучить его реализацию либо ознакомиться с соответствующим тестовым кодом или документацией.

Преимущества и недостатки

+ Упрощает код клиента для построения сложного объекта Composite.
+ Снижает количество повторений и склонность к ошибкам при создании Composite.
+ Создает слабую связь между клиентом и Composite.
+ Допускает разные представления инкапсулированных Composite или сложного объекта.
– Может не иметь интерфейса с явно выраженным назначением.

Механика

Существует множество способов создания шаблона Builder для построения Composite. Поэтому невозможно указать единственный набор инструкций для данной реорганизации кода. Какой бы проект вы не выбрали для шаблона Builder, советую использовать управляемую тестами разработку [7].

Инструкции, представленные в этом разделе, предполагают, что уже имеется код построения Composite и стоит задача инкапсулировать этот код с помощью шаблона Builder.

1. Создайте *построитель* (builder) — новый класс, который в конце реорганизации кода станет шаблоном Builder. Предоставьте созданному построителю возможность производить одноузловой Composite [12]. Добавьте в построитель метод для получения результата построения.

   ✓ Скомпилируйте и протестируйте.

2. Обеспечьте построитель возможностью построения дочерних узлов. Эта процедура часто влечет за собой создание множества методов, необходимых для того, чтобы позволить клиенту легко управлять созданием и размещением дочерних узлов.

   ✓ Скомпилируйте и протестируйте.

3. Если заменяемый вами код, создающий Composite, устанавливает атрибуты или значения узлов, обеспечьте построитель возможностью устанавливать эти атрибуты и значения.

   ✓ Скомпилируйте и протестируйте.

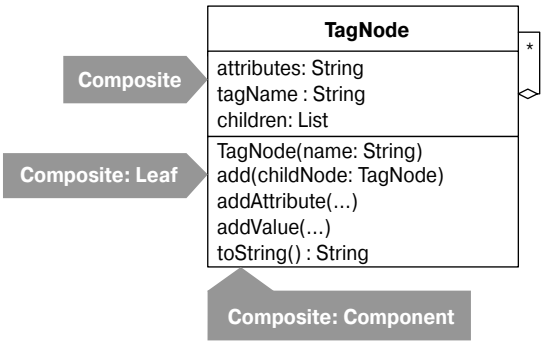
4. Продумайте, достаточна ли проста структура построителя для использования клиентами. Если возможно, еще больше упростите эту структуру.

5. Реорганизуйте код, создающий Composite, таким образом, чтобы он использовал новый построитель. Это влечет за собой создание клиентского кода, известного как Builder: Client and Builder: Director [12].

   ✓ Скомпилируйте и протестируйте.

Пример

Класс Composite, который я хочу инкапсулировать с помощью шаблона Builder, называется `TagNode`. Этот класс возникает в рефакторинге Replace Implicit Tree with Composite (с. 210). Класс `TagNode` облегчает создание документов XML. Он выполняет все три роли Composite, поскольку является компонентом, который во время выполнения может представлять собой либо Leaf, либо Composite, как показано ниже.



Метод `toString()` класса `TagNode` выдает XML-представление для всех объектов, содержащихся в классе `TagNode`. Класс `TagBuilder` инкапсулирует класс `TagNode`, обеспечивая клиентам способ создания сложных объектов класса `TagNode`, менее подверженный повторениям и ошибкам.

1. Мой первый шаг — создание построителя, который сможет успешно построить один узел. В нашем случае я хочу создать класс `TagBuilder`, который генерирует правильный документ XML для дерева, содержащего единственный объект класса `TagNode`. Я начинаю этот процесс с написания недостающего теста, который использует `assertXmlEquals` — метод, написанный мною для сравнения двух частей документа XML.

```
public class TagBuilderTest...
  public void testBuildOneNode() {
    String expectedXml =
      "<flavors/>";
    String actualXml = new TagBuilder("flavors").toXml();
    assertXmlEquals(expectedXml, actualXml);
  }
```

Пройти этот тест легко. Вот код, который я пишу:

```
public class TagBuilder {
  private TagNode rootNode;

  public TagBuilder(String rootTagName) {
    rootNode = new TagNode(rootTagName);
  }

  public String toXml() {
    return rootNode.toString();
  }
}
```

Компилятор и тестовый код успешно справляются с этим новым кодом.

2. Теперь я должен предоставить классу `TagBuilder` возможности для обработки дочерних узлов. Я хочу рассмотреть несколько сценариев, каждый из которых требует написания различных методов класса `TagBuilder`.

   Я начинаю со сценария добавления дочернего узла в корневой узел. Поскольку я хочу, чтобы класс `TagBuilder` как создавал дочерний узел, так и корректно размещал его внутри инкапсулированного композита, я решаю создать метод `addChild()`, который будет выполнять только эти действия. Этот метод используется в следующем тесте:

```
public class TagBuilderTest...
  public void testBuildOneChild() {
    String expectedXml =
      "<flavors>"+
        "<flavor/>" +
      "</flavors>";

    TagBuilder builder = new TagBuilder("flavors");
    builder.addChild("flavor");
    String actualXml = builder.toXml();

    assertXmlEquals(expectedXml, actualXml);
  }
```

   Дальше показаны изменения, выполненные мною для того, чтобы этот тест был пройден.

```
public class TagBuilder {
  private TagNode rootNode;
  private TagNode currentNode;

  public TagBuilder(String rootTagName) {
    rootNode = new TagNode(rootTagName);
    currentNode = rootNode;
}

  public void addChild(String childTagName) {
    TagNode parentNode = currentNode;
    currentNode = new TagNode(childTagName);
    parentNode.add(currentNode);
  }

  public String toXml() {
    return rootNode.toString();
  }
}
```

   Это было несложно. Для того чтобы полностью проверить новый код, я создаю более строгий тест и проверяю, все ли работает правильно.

```
public class TagBuilderTest...
  public void testBuildChildrenOfChildren() {
    String expectedXml =
      "<flavors>"+
        "<flavor>" +
          "<requirements>" +
            "<requirement/>" +
          "</requirements>" +
        "</flavor>" +
      "</flavors>";

    TagBuilder builder = new TagBuilder("flavors");
    builder.addChild("flavor");
      builder.addChild("requirements");
        builder.addChild("requirement");
    String actualXml = builder.toXml();

    assertXmlEquals(expectedXml, actualXml);
  }
```

Код проходит и этот тест. Теперь наступает время для обработки другого сценария — добавления родственного ("братского") узла дерева. Я снова создаю недостающий тест.

```
public class TagBuilderTest...
  public void testBuildSibling() {
    String expectedXml =
      "<flavors>"+
        "<flavor1/>" +
        "<flavor2/>" +
      "</flavors>";

    TagBuilder builder = new TagBuilder("flavors");
    builder.addChild("flavor1");
    builder.addSibling("flavor2");
    String actualXml = builder.toXml();

    assertXmlEquals(expectedXml, actualXml);
  }
```

Добавление братских узлов к существующим дочерним узлам подразумевает, что класс `TagBuilder` может определить, какой узел является общим родительским узлом для данных дочернего и братского узлов. В настоящий момент определить это невозможно, поскольку экземпляр класса `TagNode` не хранит ссылку на свой родительский узел. Поэтому я создаю очередной тест для проведения управляемой тестами разработки необходимой функциональности.

```
public class TagNodeTest...
  public void testParents() {
    TagNode root = new TagNode("root");
    assertNull(root.getParent());

    TagNode childNode = new TagNode("child");
    root.add(childNode);
    assertEquals(root, childNode.getParent());
    assertEquals("root", childNode.getParent().getName());
  }
```

Чтобы этот тест был пройден, я добавляю в класс `TagNode` следующий код:

```
public class TagNode...
  private TagNode parent;

  public void add(TagNode childNode) {
    childNode.setParent(this);
    children().add(childNode);
  }

  private void setParent(TagNode parent) {
    this.parent = parent;
  }

  public TagNode getParent() {
    return parent;
  }
```

После размещения новой функциональности я могу переключиться на написание кода для прохождения приведенного ранее теста `testBuildSibling()`. Вот код, который я пишу:

```
public class TagBuilder...
  public void addChild(String childTagName) {
    addTo(currentNode, childTagName);
  }

  public void addSibling(String siblingTagName) {
    addTo(currentNode.getParent(), siblingTagName);
  }

  private void addTo(TagNode parentNode, String tagName) {
    currentNode = new TagNode(tagName);
    parentNode.add(currentNode);
}
```

Новый код успешно проходит проверку компилятором и тестами. Для подтверждения того, дочерние и братские узлы корректно ведут себя в разных условиях, я пишу дополнительные тесты.

Теперь необходимо обработать последний сценарий построения дочерних узлов — случай, когда методы `addChild` и `addSibling` не будут работать из-за того, что дочерний узел должен быть добавлен к конкретному родительскому узлу. Следующий тест определяет наличие такой проблемы:

```
public class TagBuilderTest...
  public void testRepeatingChildrenAndGrandchildren() {
    String expectedXml =
      "<flavors>"+
        "<flavor>" +
          "<requirements>" +
            "<requirement/>" +
          "</requirements>" +
        "</flavor>" +
        "<flavor>" +
          "<requirements>" +
            "<requirement/>" +
          "</requirements>" +
        "</flavor>" +
      "</flavors>";

    TagBuilder builder = new TagBuilder("flavors");
    for (int i=0; i<2; i++) {
      builder.addChild("flavor");
        builder.addChild("requirements");
          builder.addChild("requirement");
    }

    assertXmlEquals(expectedXml, builder.toString());
  }
```

Этот тест не будет пройден, поскольку объекты в нем не строятся так, как ожидалось. Когда цикл выполняет вторую итерацию, при вызове метода `addChild()` объекта `builder` происходит ошибка, заключающаяся в том, что он добавляет дочерний узел к последнему добавленному узлу, и мы получаем неверный результат.

```
<flavors>
  <flavor>
    <requirements>
      <requirement/>
        <flavor>       ← Ошибка: дескрипторы не на месте
          <requirements>
            <requirement/>
          </requirements>
        </flavor>
    </requirements>
  </flavor>
  <flavors>
```

Для устранения этой проблемы я изменяю тест, обращаясь в нем к методу `addToParent()`, который позволяет клиенту добавлять новый узел к конкретному родительскому узлу.

```
public class TagBuilderTest...
  public void testRepeatingChildrenAndGrandchildren()...
    ...
    TagBuilder builder = new TagBuilder("flavors");
    for (int i=0; i<2; i++) {
      builder.addToParent("flavors", "flavor");
        builder.addChild("requirements");
          builder.addChild("requirement");
  }
  assertXmlEquals(expectedXml, builder.toXml());
```

Этот тест не будет компилироваться до тех пор, пока не будет реализован метод `addToParent()`. Идея данного метода заключается в том, что он будет запрашивать узел `currentNode` класса `TagBuilder`, соответствует ли его имя имени родительского узла (переданному в качестве параметра). Если имя соответствует, то метод добавляет новый дочерний узел к узлу `currentNode`. В противном случае метод находит узел, родительский по отношению к узлу `currentNode`, и продолжает процесс до тех пор, пока не будет найдено соответствующее имя или не будет отсутствовать родительский узел. Соответствующий шаблон называется Chain of Responsibility [12].

Для реализации шаблона Chain of Responsibility я добавляю в класс `TagBuilder` новый код.

```
public class TagBuilder...
  public void addToParent(String parentTagName,
                          String childTagName) {
    addTo(findParentBy(parentTagName), childTagName);
  }

  private void addTo(TagNode parentNode, String tagName) {
    currentNode = new TagNode(tagName);
    parentNode.add(currentNode);
  }

  private TagNode findParentBy(String parentName) {
    TagNode parentNode = currentNode;
    while (parentNode != null) {
      if (parentName.equals(parentNode.getName()))
        return parentNode;
      parentNode = parentNode.getParent();
    }
    return null;
  }
```

Теперь тест оказывается пройденным. Перед тем как идти далее, я хочу сделать так, чтобы в методе `addToParent()` был предусмотрен вариант, при котором переданного имени родительского узла не существует. Поэтому я пишу следующий тест:

```
public class TagBuilderTest...
  public void testParentNameNotFound() {
    TagBuilder builder = new TagBuilder("flavors");
    try {
      for (int i=0; i<2; i++) {
        // В строке ниже требуется "flavors", а не "favors"
        builder.addToParent("favors", "flavor");
        builder.addChild("requirements");
        builder.addChild("requirement");
      }
      fail("Не разрешено добавление к "
           "несуществующему родительскому узлу.");
    } catch (RuntimeException runtimeException) {
      String expectedErrorMessage =
        "Отсутствует родительский дескриптор : favors";
      assertEquals(expectedErrorMessage,
                   runtimeException.getMessage());
    }
  }
```

Внося изменения в класс `TagBuilder`, я добиваюсь того, что тест оказывается успешно пройденным.

```
public class TagBuilder...
  public void addToParent(String parentTagName,
                          String childTagName) {
    TagNode parentNode = findParentBy(parentTagName);
    if (parentNode == null)
      throw new RuntimeException(
        "missing parent tag: " + parentTagName);
    addTo(parentNode, childTagName);
  }
```

3. Теперь я предоставляю классу `TagBuilder` возможность добавления атрибутов и значений в узлы. Выполнить этот шаг не сложно, поскольку инкапсулированный класс `TagNode` уже в состоянии работать с атрибутами и значениями. Далее представлен тест, позволяющий убедиться в правильности работы с атрибутами и значениями.

```
public class TagBuilderTest...
  public void testAttributesAndValues() {
    String expectedXml =
      // Дескриптор с атрибутом:
```

```
    "<flavor name='Test-Driven Development'>" +
      "<requirements>" +
        "<requirement type='hardware'>" +
            // Дескриптор со значением
            "Один компьютер для каждых двух участников" +
          "</requirement>" +
        "<requirement type='software'>" +
            "IDE" +
        "</requirement>" +
      "</requirements>" +
    "</flavor>";
  TagBuilder builder = new TagBuilder("flavor");
  builder.addAttribute("name", "Test-Driven Development");
    builder.addChild("requirements");
      builder.addToParent("requirements", "requirement");
      builder.addAttribute("type", "hardware");
      builder.addValue("Один компьютер для "
                       "каждых двух участников");
      builder.addToParent("requirements", "requirement");
      builder.addAttribute("type", "software");
      builder.addValue("IDE");

  assertXmlEquals(expectedXml, builder.toXml());
}
```

Следующие новые методы позволяют пройти этот тест:

```
public class TagBuilder...
  public void addAttribute(String name, String value) {
    currentNode.addAttribute(name, value);
  }

  public void addValue(String value) {
    currentNode.addValue(value);
  }
```

4. Теперь наступает время задуматься, насколько прост класс `TagBuilder` и насколько легко клиенту его использовать. Есть ли более простой способ создания документов XML? Это не тот вопрос, на который можно сразу же дать правильный ответ. Возможно, более простая идея и появится, но для этого в любом случае потребуется множество экспериментов и часы, дни или даже недели размышлений. Далее, в разделе *Разновидности*, описана более простая реализация, а пока перейдем к последнему этапу.

5. Я завершаю данный рефакторинг, заменяя код построения Composite кодом, в котором используется класс `TagBuilder`. Я не знаю легких способов, позволяющих проделать это: код построения Composite может охватывать большие части системы. Будем надеяться, что у вас есть все необходимые тесты, которые помогут обнаружить все ошибки во время такого преобразования. Далее

представлен метод класса `CatalogWriter`, который нужно изменить таким образом, чтобы вместо класса `TagNode` он использовал класс `TagBuilder`.

```
public class CatalogWriter...
  public String catalogXmlFor(Activity activity) {
    TagNode activityTag = new TagNode("activity");
    ...
    TagNode flavorsTag = new TagNode("flavors");
    activityTag.add(flavorsTag);
    for (int i=0; i < activity.getFlavorCount(); i++) {
      TagNode flavorTag = new TagNode("flavor");
      flavorsTag.add(flavorTag);
      Flavor flavor = activity.getFlavor(i);
      ...
      int requirementsCount =
          flavor.getRequirements().length;
      if (requirementsCount > 0) {
        TagNode requirementsTag =
            new TagNode("requirements");
        flavorTag.add(requirementsTag);
        for (int r=0; r < requirementsCount; r++) {
          Requirement requirement =
              flavor.getRequirements()[r];
          TagNode requirementTag =
              new TagNode("requirement");
          ...
          requirementsTag.add(requirementTag);
        }
      }
    }
    return activityTag.toString();
  }
```

Как показано в приведенной ниже диаграмме, этот код работает с объектами предметной области `Activity`, `Flavor` и `Requirement`.

Вы можете удивиться тому, что этот код создает Composite для объектов класса `TagNode`, передавая данные объекта `Activity` в XML, вместо того чтобы просто потребовать от экземпляров `Activity`, `Flavor` и `Requirement` передать данные в XML с помощью метода `toXml()`. Этот вопрос можно задавать, если объекты предметной области образуют структуру Composite. Тогда может не быть никакого смысла в образовании другой структуры Composite наподобие класса `activityTag` только для того, чтобы передать объекты предметной области в XML. Однако в данном случае имеет смысл создавать

XML внешним образом из объектов предметной области, потому что система, использующая эти объекты, должна создавать для них несколько абсолютно разных XML-представлений. Единственный метод `toXML()` для всех объектов не в состоянии обеспечить хорошую работу в этой ситуации — от каждой реализации этого метода будет требоваться создание многих разных представлений XML объекта.

Вот как будет выглядеть метод `catalogXmlFor(...)` после внесения изменений для использования в нем класса `TagBuilder`:

```
public class CatalogWriter...
  private String catalogXmlFor(Activity activity) {
    TagBuilder builder = new TagBuilder("activity");
    ...
    builder.addChild("flavors");
    for (int i=0; i < activity.getFlavorCount(); i++) {
      builder.addToParent("flavors", "flavor");
      Flavor flavor = activity.getFlavor(i);
      ...
      int requirementsCount =
          flavor.getRequirements().length;
      if (requirementsCount > 0) {
        builder.addChild("requirements");
        for (int r=0; r < requirementsCount; r++) {
          Requirement requirement =
              flavor.getRequirements()[r];
          builder.addToParent("requirements",
                              "requirement");
          ...
        }
      }
    }
    return builder.toXml();
  }
```

Вот теперь данный рефакторинг завершен. Класс `TagNode` полностью инкапсулирован классом `TagBuilder`.

Усовершенствование шаблона Builder

Я не могу удержаться, чтобы не рассказать вам о возможности улучшить производительность класса `TagBuilder`, поскольку таким образом можно продемонстрировать элегантность и простоту шаблона Builder. Несколько моих коллег по компании Evant выполняли ряд профилирований нашей системы и обнаружили, что причиной проблем в работе является класс `StringBuffer`, используемый классом `TagNode` (класс, инкапсулированный классом `TagBuilder`). Класс `StringBuffer` используется как шаблон Collecting Parameter: после своего создания он передается в каждый узел композиции объектов `TagNode`. Таким способом получается результат, возвращаемый вы-

зовом метода `toXml()` класса `TagNode`. Чтобы лучше понять принцип его работы, рекомендую рассмотреть пример для Move Accumulation To Collecting Parameter (с. 352).

Класс `StringBuffer`, который использовался в этой операции, не инстанцировался с каким-либо конкретным размером. Это означает, что данный класс должен автоматически расти, когда он не в состоянии хранить все данные, поступающие в него из-за постоянного добавления нового XML-текста. Здесь все в порядке: класс `StringBuffer` разработан именно так, чтобы при необходимости автоматически расширяться. С другой стороны, таким образом снижается производительность, поскольку класс `StringBuffer` должен выполнять работу, вызванную таким прозрачным расширением и постоянным перемещением данных. Для системы Evant это было не приемлемо.

Эту проблему можно решить, если перед инстанцированием класса `StringBuffer` знать его точный необходимый размер. Каким же образом можно вычислить такой подходящий размер? Это легко. Каждый добавляемый в `TagBuilder` узел, атрибут или значение может увеличить размер буфера на величину, зависящую от размера добавляемого элемента. Окончательно вычисленный размер буфера можно затем использовать для инстанцирования класса `StringBuffer`, получая объект, который никогда не придется расширять.

Для реализации такого повышения производительности мы, как обычно, начнем с создания недостающих тестов. Приведенный далее тест строит дерево XML с помощью вызовов `TagBuilder`, после чего получает размер созданной в результате строки XML, возвращаемой построителем, и в завершение сравнивает размеры этой строки с вычисленным размером буфера.

```
public class TagBuilderTest...
  public void testToStringBufferSize() {
    String expected =
    "<requirements>" +
      "<requirement type='software'>" +
        "IDE" +
      "</requirement>" +
    "</requirements>";
    TagBuilder builder = new TagBuilder("requirements");
    builder.addChild("requirement");
    builder.addAttribute("type", "software");
    builder.addValue("IDE");

    int stringSize = builder.toXml().length();
    int computedSize = builder.bufferSize();
    assertEquals("buffer size", stringSize, computedSize);
  }
```

Для того чтобы пройти этот тест и подобные ему, мы изменяем класс `TagBuilder`, как показано ниже.

```
public class TagBuilder...
  private int outputBufferSize;
  private static int TAG_CHARS_SIZE = 5;
  private static int ATTRIBUTE_CHARS_SIZE = 4;

  public TagBuilder(String rootTagName) {
    ...
    incrementBufferSizeByTagLength(rootTagName);
    }

    private void addTo(TagNode parentNode, String tagName) {
      ...
      incrementBufferSizeByTagLength(tagName);
    }

    public void addAttribute(String name, String value) {
      ...
      incrementBufferSizeByAttributeLength(name, value);
    }

    public void addValue(String value) {
      ...
      incrementBufferSizeByValueLength(value);
    }

    public int bufferSize() {
      return outputBufferSize;
    }

    private void
      incrementBufferSizeByAttributeLength(String name,
                                           String value) {
      outputBufferSize += (name.length() + value.length() +
                           ATTRIBUTE_CHARS_SIZE);
    }

    private void incrementBufferSizeByTagLength(String tag){
      int sizeOfOpenAndCloseTags = tag.length() * 2;
      outputBufferSize += (sizeOfOpenAndCloseTags +
                           TAG_CHARS_SIZE);
    }

    private void
      incrementBufferSizeByValueLength(String value) {
      outputBufferSize += value.length();
    }
```

Такие изменения в классе `TagBuilder` прозрачны для пользователей данного класса, поскольку он инкапсулирует логику, связанную с повышением производительности. Единственное дополнительное изменение требуется внести в метод

toXml() класса TagBuilder, чтобы он мог инстанцировать StringBuffer корректного размера и передавать его корневому узлу TagNode, который накапливает XML-содержимое. Для этого мы изменяем метод toXml()

```
public class TagBuilder...
  public String toXml() {
    return rootNode.toString();
  }
```

следующим образом:

```
public class TagBuilder...
  public String toXml() {
    StringBuffer xmlResult =
        new StringBuffer(outputBufferSize);
    rootNode.appendContentsTo(xmlResult);
    return xmlResult.toString();
  }
```

Вот, собственно, и все. Тест пройден, и теперь класс TagBuilder заработал значительно быстрее.

Разновидности

Шаблон Builder*, основанный на схеме*

Класс TagNode содержит три метода, предназначенные для добавления узлов в Composite:

- addChild(String childTagName);
- addSibling(String siblingTagName);
- addToParent(String parentTagName, String childTagName).

Каждый из этих методов участвует в создании и установке новых узлов с дескрипторами внутри инкапсулированного Composite. Я заинтересовался, возможно ли написать шаблон Builder, который способен использовать только один метод add(String tagName). Для реализации такого проекта необходимо, чтобы шаблон Builder содержал сведения о размещении дескрипторов, добавленных клиентами. Я решил поэкспериментировать с этой идеей. Класс, полученный в результате, я назвал SchemaBasedTreeBuilder. Ниже приведен тест, показывающий, как он работает.

```
public class SchemaBasedTagBuilderTest...
  public void testTwoSetsOfGreatGrandchildren() {
    TreeSchema schema = new TreeSchema(
      "orders" +
      " order" +
      "   item" +
      "     apple" +
```

```
      "     orange"
    );

    String expected =
      "<orders>" +
        "<order>" +
          "<item>" +
            "<apple/>" +
            "<orange/>" +
          "</item>" +
          "<item>" +
            "<apple/>" +
            "<orange/>" +
          "</item>" +
        "</order>" +
      "</orders>";
    SchemaBasedTagBuilder builder =
        new SchemaBasedTagBuilder(schema);
    builder.add("orders");
      builder.add("order");
      for (int i=0; i<2; i++) {
        builder.add("item");
          builder.add("apple");
          builder.add("orange");
      }
    assertXmlEquals(expected, builder.toString());
  }
```

Класс `SchemaBasedTreeBuilder` изучает, где он должен разместить дескрипторы, посредством экземпляра `TreeSchema`. Класс `TreeSchema` — это класс, принимающий строки, с символами табуляции в качестве разделителей, которые определяют дерево, состоящее из имен дескрипторов.

```
"orders" +
"  order" +
"    item" +
"      apple" +
"      orange"
```

Экземпляр `TreeSchema` получает эту строку и преобразует ее в приведенную ниже схему.

Дочерний узел	Родительский узел
orders	`null`
order	orders
item	order
apple	item
orange	item

Во время выполнения программы класс `SchemaBasedTagBuilder` использует схему `TreeSchema` для вычисления позиции нового дескриптора. Например, если я напишу `build.add("orange")`, то построитель из экземпляра `TreeSchema` выяснит, что родительский узел для дескриптора `"orange"` — дескриптор `"item"`, и добавит новый дескриптор `"orange"` к ближайшему дескриптору `"item"`.

Такой подход хорошо работает до тех пор, пока не появляется два дескриптора с одинаковым именем.

```
"organization" +
"  name" +
"  departments" +
"    department" +
"      name"
```

В этом случае схема `TreeSchema` должна содержать два родительских узла для дескриптора `"name"`.

Дочерний узел	Родительский узел
...	...
name	organization, department
...	...

Во время выполнения программы класс `SchemaBasedTagBuilder` ищет имена родительских узлов для нового дескриптора, находит ближайший родительский дескриптор и добавляет новый дескриптор к ближайшему родительскому. Если клиент не желает полагаться на поиск ближайшего дескриптора, а хочет точно указать родительский узел для добавления нового дескриптора, то он может вызвать метод `add()`, который явно указывает соединение.

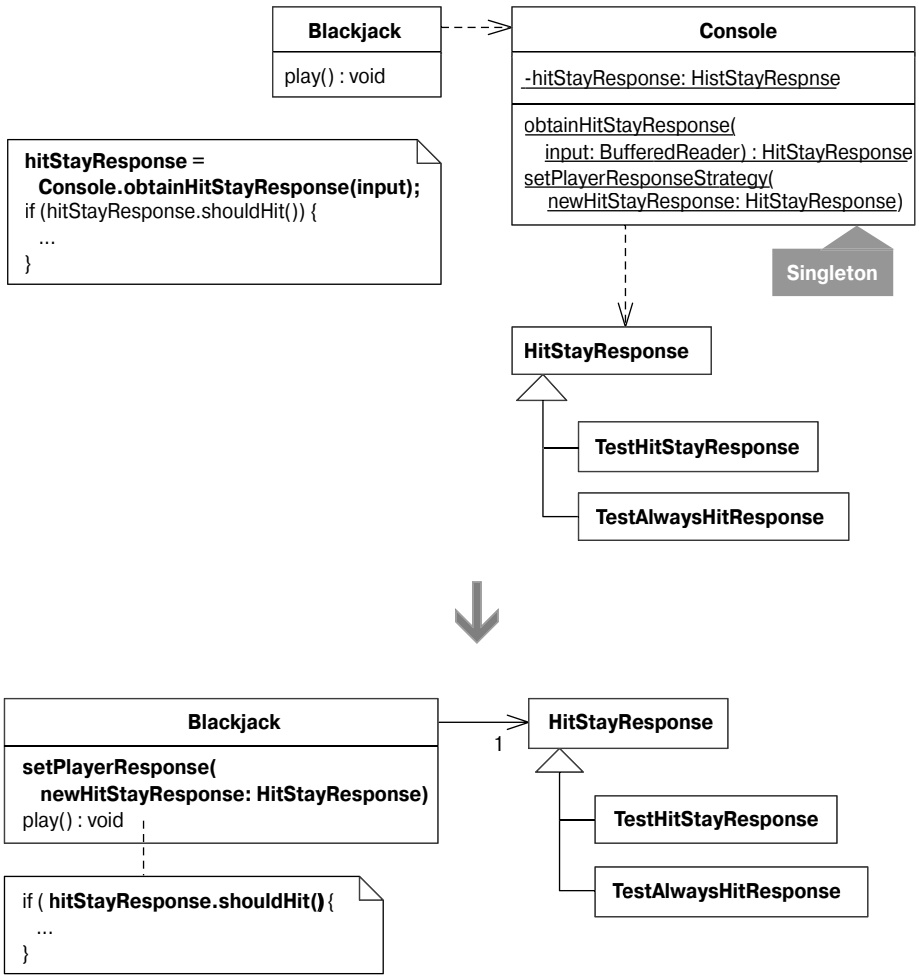
```
builder.add("department","name");   // Явное указание
   // построителю места добавления дескриптора "name"
```

Вот и все, что следовало рассказать про класс `SchemaBasedTagBuilder`. Он и класс `TagBuilder` выполняют похожую работу, но делают это разными способами. Обычно для построения документа XML я использую класс `TagBuilder`. Но для создания большого документа XML я бы применил класс `SchemaBasedTagBuilder`, поскольку в нем можно не беспокоиться о размещении дескрипторов. Кроме того, если большой документ XML имеет связанную с ним схему XML, скорее всего, я бы написал код для преобразования этой XML-схемы в экземпляр класса `TreeSchema`, чтобы ее можно было использовать в классе `SchemaBasedTagBuilder`.

Inline Singleton

В коде требуется доступ к объекту,
но не нужна глобальная точка доступа к нему.

Переместить функциональные возможности синглтона в класс, сохраняющий объект и обеспечивающий к нему доступ. Удалить синглтон.



Мотивация

Для обозначения поистине наркотической зависимости от шаблона Singleton я придумал термин "синглтонизм". Назначение шаблона Singleton — "обеспечить для каждого класса только один экземпляр и предоставить глобальную точку доступа к нему" [12, с. 127]. Вы — синглтоник, если шаблон Singleton так глубоко укоренился в вашем сознании, что начинает царствовать над другими шаблонами и более простыми идеями разработки, тем самым вынуждая вас использовать слишком большое количество синглтонов.

Мне удалось избавиться от этой зависимости, и сейчас я задумываюсь о возможности основания "общества анонимных синглтоников" — места, где восстанавливающиеся после злоупотребления синглтонами смогут поддержать друг друга на непростом пути назад, к использованию простых неглобальных объектов. Применение рефакторинга Inline Singleton — полезный шаг на этом пути. Такой рефакторинг помогает избавить систему от излишних синглтонов. Но возникает очевидный вопрос: в каких случаях синглтон будет излишним?

Краткий ответ: практически во всех случаях.

Подробный ответ: синглтон не нужен, если проще передать ресурс нуждающемуся в нем объекту в виде ссылки, чем обеспечить объекты глобальным доступом к ресурсу. Синглтон не нужен, если он используется только для незначительной экономии памяти или небольшого повышения производительности. Синглтон не нужен, если для доступа к ресурсу требуется "погружение" кода на системный уровень, но код этому уровню не принадлежит. Я могу долго продолжать этот список. Основная идея состоит в том, что синглтоны не нужны, если при проектировании или перепроектировании можно избежать их использования. При написании этого рефакторинга я решил попросить Варда Каннингема (Ward Cunningham) и Кента Бека (Kent Beck) прокомментировать шаблон Singleton.

Вард Каннингем о синглтонах

Частое использование шаблонов Singleton связано с тем, что механизм защиты языка экранируется защитой только одного аспекта — сингулярности. Я согласен, это важно, но пропорции здесь явно нарушены.

Для всего существует свой контекст. Больш*ая* часть объектно-ориентированного программирования имеет дело с созданием контекста и временем жизни переменных, заставляя последние прожить столько, сколько нужно, и корректно уйти, расплатившись со всеми долгами. Я не боюсь небольшой глобальности. Она обеспечивает глобальный контекст, который все должны понимать. Но ее не должно быть слишком много. Чрезмерная глобальность меня пугает.

Кент Бек о синглтонах

Истинная проблема синглтонов заключается в том, что они дают нам довольно хорошее оправдание не заботиться о корректной видимости объекта. Для поддержки гибкости чрезвычайно важно определить разумный баланс между видимостью и защитой объекта.

Однажды мы вместе с Массимо Арнольди (Massimo Arnoldi) работали над системой, в которой синглтон использовался для хранения обменных курсов. Каждый раз, когда мы создавали тест для работы с несколькими валютами, мы должны были сохранять старые курсы обмена, после этого сохранять несколько новых курсов, запускать тест, а затем восстанавливать старые курсы обмена. Однажды нам надоело все время иметь дело с ошибками, которые были вызваны использованием неверного курса, и мы решили отказаться от синглтона.

Мы были озадачены и растеряны — ведь курсы обмена использовались повсеместно во всей системе. Но хорошая идея — это хорошая идея, и нам пришлось найти все места в системе, в которых использовались курсы обмена. После этого мы добавили параметры, необходимые для явной передачи этих курсов. Мы думали, что нам потребуется проделать огромное количество работы, но все это заняло лишь полчаса. Время от времени возникали небольшие трудности, связанные с получением курсов в том месте, где они были нужны, но всегда было очевидно, какой рефакторинг решал нашу проблему. Заодно эти рефакторинги разрешили некоторые хронические проблемы проектирования, которые нам уже порядком надоели, но мы не знали, как с ними справиться.

И вот какие результаты были достигнуты после тридцати минут работы:

- более ясный и гибкий проект;
- стабильные результаты тестирования;
- чувство глубокого облегчения.

Мартин Фаулер также признает необходимость небольшой глобальности, хотя считает, что прибегать к этому стоит лишь в крайних случаях. Его шаблон Registry, описанный в *Patterns of Enterprise Application Architecture*, представляет собой незначительную вариацию шаблона Singleton. Мартин так описывает этот шаблон: “широко известный объект, который может использоваться другими объектами для поиска общих объектов и служб” [16, с. 480]. Свое отношение к использованию этого шаблона он высказывает так:

> Существуют альтернативы использованию шаблона Registry. Один из возможных вариантов — передача необходимых данных в качестве параметров. Но проблема заключается в том, что эти параметры добавляются в вызовы методов, в то время как в самих вызываемых методах они не используются и нужны только методам,

расположенным несколькими слоями ниже в дереве вызовов. Передача параметров, которые для 90% вызовов не нужны, привела меня к использованию шаблона Register...

Так что бывают ситуации, когда необходимо использовать шаблон Register. Но помните, что любые глобальные данные всегда виновны, пока не доказана их невиновность [16, с. 482–483].

Мое довольно тесное знакомство с книгой *Design Patterns* [12] привело меня к заболеванию синглтонизмом. Описание каждого шаблона в этой книге содержит раздел *Related Pattern*, и в этих разделах часто можно найти упоминания о шаблоне Singleton. Например, в соответствующем разделе описания шаблона State автор пишет: "объекты шаблона State часто являются синглтонами" [12, с. 313], а в соответствующем разделе описания шаблона Abstract Factory можно прочитать: "конкретная фабрика часто является синглтоном" [12, с. 95]. В защиту автора стоит отметить: такие высказывания просто констатируют, что классы State и Abstract Factory *часто* бывают синглтонами. В книге не указывается, что они на самом деле ими есть. Если существуют весомые причины, по которым класс стоит сделать синглтоном или реестром, смело делайте это. Достойная причина для реорганизации кода с включением в него синглтона указана в рефакторинге Limit Instantiation with Singleton (с. 335): реальное повышение производительности. В описании этого рефакторинга содержится также предостережение против преждевременной оптимизации.

Одно можно утверждать с уверенностью: перед реализацией шаблона Singleton следует *действительно хорошо* подумать и изучить все за и против. И если вы встретите синглтон, который не должен быть синглтоном, воспользуйтесь описываемым в этом разделе рефакторингом.

Преимущества и недостатки

+ Делает сотрудничество объектов более видимым и явным.
+ Не требует специального кода для защиты единственного экземпляра.
– Усложняет проект, если передача экземпляра объекта через несколько уровней неудобна или затруднена.

Механика

Механика данного рефакторинга полностью совпадает с механикой рефакторинга Inline Class [15]. В представленных далее инструкциях под понятием *поглощающий класс* (absorbing class) подразумевается класс, который принимает на себя дополнительные требования синглтона, размещенного внутри кода.

1. Объявите открытые методы синглтона в поглощающем классе. Создайте новые методы таким образом, чтобы они делегировали функциональность назад в синглтон, и удалите все спецификаторы `static` в этих методах (в поглощающем классе).

   Если поглощающий класс является синглтоном, то спецификаторы `static` для этих методов необходимо сохранить.

2. В коде клиента замените все обращения к синглтону на обращения к поглощающему классу.

   ✓ Скомпилируйте и протестируйте.

3. Примените рефакторинги Move Method и Move Field [15] для переноса функциональности синглтона в поглощающий класс. Выполняйте их до тех пор, пока все элементы не будут перенесены.

   Как и в первом шаге, удалите спецификаторы `static` из объявления перемещаемых методов и полей в том случае, если поглощающий класс не является синглтоном.

   ✓ Скомпилируйте и протестируйте.

4. Удалите синглтон.

Пример

Идея этого примера исходит из раннего усовершенствования консольной версии простой игры в двадцать одно (Blackjack). Ход игры заключается в отображении на экране карт игрока, многократно задаваемого игроку вопроса, продолжать игру или остановиться, в итоговом выводе на экран карт, имеющихся на руках у игрока и дилера, и в проверке, кто выиграл. Тестовый код может запустить эту игру и имитировать ввод пользователя (остановиться или продолжить игру).

Такие смоделированные данные определяются и поступают во время выполнения программы из синглтона `Console`, который связан с некоторым экземпляром класса `HitStayResponse` или с одним из его подклассов.

```
public class Console {
  static private HitStayResponse hitStayResponse =
    new HitStayResponse();

  private Console() {
    super();
  }

  public static HitStayResponse
    obtainHitStayResponse(BufferedReader input) {
    hitStayResponse.readFrom(input);
    return hitStayResponse;
  }
```

```
  public static void
    setPlayerResponse(HitStayResponse newHitStayResponse) {
    hitStayResponse = newHitStayResponse;
  }
}
```

Перед началом игры в `Console` регистрируется определенный объект `HitStayResponse`. Например, ниже предоставлен вариант тестового кода, в котором экземпляр класса `TestAlwaysHitResponse` регистрируется в `Console`.

```
public class ScenarioTest extends TestCase...
  public void testDealerStandsWhenPlayerBusts() {
    Console.setPlayerResponse(new TestAlwaysHitResponse());
    int[] deck = { 10, 9, 7, 2, 6 };
    Blackjack blackjack = new Blackjack(deck);
    blackjack.play();
    assertTrue("dealer wins", blackjack.didDealerWin());
    assertTrue("player loses", !blackjack.didPlayerWin());
    assertEquals("dealer total", 11,
                 blackjack.getDealerTotal());
    assertEquals("player total", 23,
                 blackjack.getPlayerTotal());
  }
```

Далее показан код класса `Blackjack`, вызывающий `Console` для получения зарегистрированного экземпляра класса `HitStayResponse`. Как видите, он совсем несложен.

```
public class Blackjack...
  public void play() {
    deal();
    writeln(player.getHandAsString());
    writeln(dealer.getHandAsStringWithFirstCardDown());
    HitStayResponse hitStayResponse;
    do {
      write("H)it or S)tay: ");
      hitStayResponse =
        Console.obtainHitStayResponse(input);
      write(hitStayResponse.toString());
      if (hitStayResponse.shouldHit()) {
        dealCardTo(player);
        writeln(player.getHandAsString());
      }
    }
    while (canPlayerHit(hitStayResponse));
    // ...
  }
```

Представленный код не "живет" на прикладном уровне, окруженном другими прикладными уровнями, что затрудняло бы передачу некоторого экземпляра HitStayResponse на все уровни, где он необходим. Весь код, обращающийся к классу HitStayResponse, находится внутри класса Blackjack. Зачем же классу Blackjack, чтобы получить доступ к классу HitStayResponse, обращаться к синглтону Console? Вот еще один синглтон, которому незачем быть синглтоном. Настало время выполнить реорганизацию кода.

1. Первый шаг — объявить открытые методы синглтона Console в поглощающем классе Blackjack. После этого нужно изменить каждый метод таким образом, чтобы он делегировал свою функциональность синглтону Console, и удалить спецификатор static из объявлений этих методов.

```
public class Blackjack...
  public static HitStayResponse
    obtainHitStayResponse(BufferedReader input) {
    return Console.obtainHitStayResponse(input);
  }
  public static void
    setPlayerResponse(HitStayResponse newHitStayResponse) {
    Console.setPlayerResponse(newHitStayResponse);
  }
```

2. Теперь я изменяю все функции, вызывающие методы синглтона Console, таким образом, чтобы они вызывали версии этих методов из класса Blackjack. Ниже показаны некоторые из таких изменений.

```
public class ScenarioTest extends TestCase...
  public void testDealerStandsWhenPlayerBusts() {
    Console.setPlayerResponse(new TestAlwaysHitResponse());
    int[] deck = { 10, 9, 7, 2, 6 };
    Blackjack blackjack = new Blackjack(deck);
    blackjack.setPlayerResponse(
                    new TestAlwaysHitResponse());
    blackjack.play();
    assertTrue("dealer wins", blackjack.didDealerWin());
    assertTrue("player loses", !blackjack.didPlayerWin());
    assertEquals("dealer total", 11,
                 blackjack.getDealerTotal());
    assertEquals("player total", 23,
                 blackjack.getPlayerTotal());
  }
```

и

```
public class Blackjack...
  public void play() {
    deal();
```

```
    writeln(player.getHandAsString());
    writeln(dealer.getHandAsStringWithFirstCardDown());
    HitStayResponse hitStayResponse;
    do {
      write("H)it or S)tay: ");
      hitStayResponse =
            Console.obtainHitStayResponse(input);
      write(hitStayResponse.toString());
      if (hitStayResponse.shouldHit()) {
        dealCardTo(player);
        writeln(player.getHandAsString());
      }
    }
    while (canPlayerHit(hitStayResponse));
    // ...
  }
```

На этом этапе я компилирую, запускаю автоматические тесты и запускаю игру, проверяя, что все работает правильно.

3. Теперь можно применять рефакторинги Move Method и Move Field [15] для перемещения функциональности синглтона `Console` в класс `Blackjack`. После этого я снова компилирую и тестирую код для того, чтобы убедиться, что класс `Blackjack` продолжает корректно работать.
4. На этом этапе я удаляю синглтон `Console` и, как рекомендует Мартин при описании рефакторинга Inline Class [15], отслужим короткую, но динамичную поминальную службу по еще одному злосчастному синглтону.