DP4Dummies – Chương 7: Template, Builder

CHƯƠNG 7:

TẠO HÀNG LOẠT ĐỐI TƯỢNG VỚI MẪU TEMPLATE (Khuôn Mẫu)
VÀ MẪU BUILDER (Thợ Xây)

Trong chương này, chúng ta đi qua một số nội dung sau:

- Sử dụng mẫu Template
- Tạo rô bốt sử dụng mẫu template
- Kế thừa mẫu template
- Hiểu biết sự khác biệt giữa mẫu Template và mẫu Builder
- Sử dụng mẫu Builder

"Tin tốt", giám đốc GigundoCorp – một công ty mới mà bạn đang nhận trách nhiệm tư vấn – vừa nói trong khi chạy vào phòng họp "Chúng ta đã nhận được hợp đồng đó"

"Hợp đồng nào?", mọi người hỏi

"Hợp đồng về những con rô bốt tự động lắp ráp xe hơi", vị giám đốc nói.

"Ồ, thì ra là hợp đồng đó" Mọi người nói.

"Giờ thì về phòng và viết chương trình thôi", vị giám đốc vừa nói và xua đuổi mọi người ra khỏi phòng họp

"Chờ một lát", bạn nói "Chúng ta có nên dành chút thời gian cho vấn đề thiết kế không? Ví dụ: có khả năng chúng ta sẽ tạo một loại khác của rô bốt trong tương lai chẳng hạn"

"Chắc chắn rồi", vị giám đốc nói. "Chúng ta có một tá hồ sơ dự thầu ngoài đó. Nhưng không có thời gian nghĩ về nó đâu. Chúng ta cần phải bắt đầu tạo những con rô bốt tự động trước"

"Vâng", các lập trình viên rên rỉ và mọi người trở về phòng của mình.

“Có điều gì đó mách bảo với tôi rằng họ đang mắc phải sai lầm”, bạn tự nhủ trong căn phòng trống rỗng, rải rác những ly Styrofoam trống rỗng lăn lóc khắp sàn.

Chương này nói này về hai mẫu thiết kế giúp bạn có một cách thức khéo léo hơn trong việc tạo dựng các đối tượng: mẫu Template Method và mẫu Builder. Mẫu Template Method cho phép các lớp con định nghĩa lại các bước tạo đối tượng, rất thích hợp cho việc tạo ra các chủng loại rô bốt khác nhau. Mẫu Builder giúp bạn uyển chuyển hơn trong việc tạo đối tượng vì nó tách rời quá trình khởi tạo ra khỏi bản thân đối tượng. Cả hai mẫu sẽ được thảo luận trong chương này

Tạo con rô bốt đầu tiên

Các lập trình viên của GigundoCorp đã xào nấu ra phần mềm của họ trong vài ngày và nó vừa đủ đơn giản. Lớp robot bắt đầu với một hàm khởi tạo như sau:

```
public class Robot
{
  public Robot()
  {
  }
    .
    .
    .
}
```

Và có một số hành động mà robot có thể thực hiện, ví dụ như, để khởi động robot, bạn gọi hàm bắt đầu Start, để robot làm việc, bạn gọi hàm lắp ráp assemble, để kiểm tra sản phẩm, bạn gọi hàm kiểm tra test, và vân vân…

```
public class Robot
{
  public Robot()
  {
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting a carburetor....");
  }

  public void assemble()
  {
    System.out.println("Installing the carburetor....");
  }

  public void test()
  {
    System.out.println("Revving the engine....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }
}
```

Và tất cả những gì bạn cần là một phương thức, tên là *go* , nó sẽ làm cho robot làm việc bằng cách gọi các hàm *start, getParts, assemble, test* và *stop* như sau:

```
public class Robot
{
  public Robot()
  {
  }

  public void go()
  {
    start();
    getParts();
    assemble();
    test();
    stop();
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting a carburetor....");
  }

  public void assemble()
  {
    System.out.println("Installing the carburetor....");
  }

  public void test()
  {
    System.out.println("Revving the engine....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }
}
```

Bạn có thể nhanh chóng viết chương trình kiểm tra. Đầu tiên tạo một robot và gọi hàm *go* như sau:

```
public class TestRobot
{
  public static void main(String args[])
  {
    Robot robot = new Robot();

    robot.go();
  }
}
```

Và khi chạy chương trình, bạn nhận được kết quả:

```
Starting....
Getting a carburetor....
Installing the carburetor....
Revving the engine....
Stopping....
```

“Tuyệt vời”, giám đốc điều hành phấn khích. “Phần thưởng luôn ở xung quanh. Tôi đã nói với anh rằng họ không cần cái thứ mẫu thiết kế vớ vẩn”. Các lập trình viên của công ty ném cho bạn một ánh nhìn dè bỉu.

Tạo Robot với Mẫu thiết kế Template Method

Ngày tiếp theo, “Tin tốt”, giám đốc điều hành của GigundoCorp la lớn, trong khi phóng vào phòng họp. “Chúng ta kí được hợp đồng khác!”

“Hợp đồng khác nào?” Mọi người hỏi

“Hợp đồng cho robot nướng bánh” Vị giám đốc nói “Giờ thì ra khỏi đây và viết phần mềm cho nó”

Các lập trình viên nhìn vào trong ly cà phê của họ “Chúng ta phải viết lại tất cả phần mềm từ đầu”, họ nói

Vị giám đốc liếc mắt nhìn bạn và hỏi “Có tốn nhiều chi phí không?”

“Rất nhiều”, các lập trình viên nói. Và bạn thì đang chống lại sự thúc giục để nói rằng “Tôi đã nói với các anh từ trước”

Đây là thời điểm thích hợp để nói về mẫu thiết kế Template Method. Có một rắc rối mà lập trình viên GigundoCopr đối mặt, họ có một con robot tự động như hình sau:

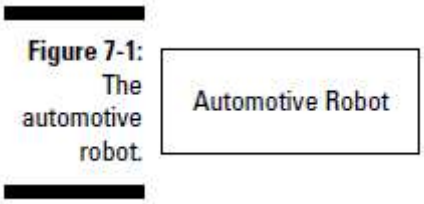

Figure 7-1: The automotive robot.

Nhưng bây giờ họ cần một con robot nướng bánh như hình sau, và thế là phải viết lại mã nguồn từ đầu:

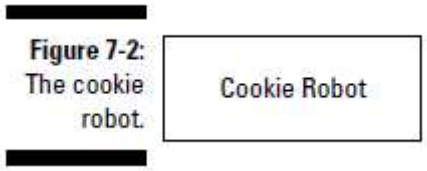

Figure 7-2: The cookie robot.

Con robot nướng bánh có một số chức năng giống như con robot lắp ráp ô tô, như là hàm *start*, *stop*, tuy nhiên nó có những sự khác biệt như lắp ráp *assemble* sẽ không hiển thị "*Getting a carburetor*" mà thay vào đó là "*Getting flour and sugar…*"

Đó là nơi mà mẫu thiết kế Template Method được áp dụng. Mẫu này nói rằng, bạn có thể viết một phương thức, dùng để xác định một loạt các thuật toán, giống như hàm go mà bạn thấy trước đây, để chạy một loạt các chức năng cho robot như hình:

```
public void go()
{
  start();
  getParts();
  assemble();
  test();
  stop();
}
```

Sau đó bạn đưa hàm này vào một bộ khuôn template bằng cách cho phép các lớp con định nghĩa lại các bước thuật toán theo cách cần thiết. Trong trường hợp này, để làm một con robot nướng bánh, bạn sẽ viết lại các hàm getParts, assemble, và test.

Theo định nghĩa chính thức của sách GoF, mẫu Template Method như sau: "Định nghĩa một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con. Mẫu Template Method cho phép lớp con định nghĩa lại cách thực hiện của một thuật toán, mà không phải thay đổi cấu trúc thuật toán."

Điều này có nghĩa là bạn nên sử dụng mẫu Template Method khi bạn có một thuận toán được tạo bởi nhiều bước, và bạn muốn thể tùy chỉnh một số bước trong đó. Chú ý rằng nếu bạn muốn viết lại mọi thứ từ đầu – khi mọi bước đều phải tùy chỉnh lại – thì bạn không cần dùng template.

Tạo robot bằng bộ khuôn Template

Nếu bạn có một bộ khuôn Template dựa trên robot, bạn có thể cho nó kế thừa như hình sau:

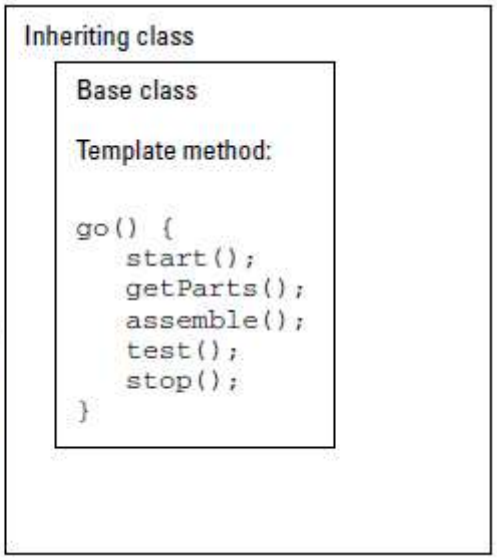

Figure 7-3: Inheriting from the template base class.

Bằng cách gọi hàm go, tập hợp các thuật toán sẽ được thực hiện. Để tùy chỉnh trong lớp kế thừa, bạn chỉ cần viết lại một số bước nào bạn muốn, trong trường hợp robot nướng bánh sẽ như hình sau:

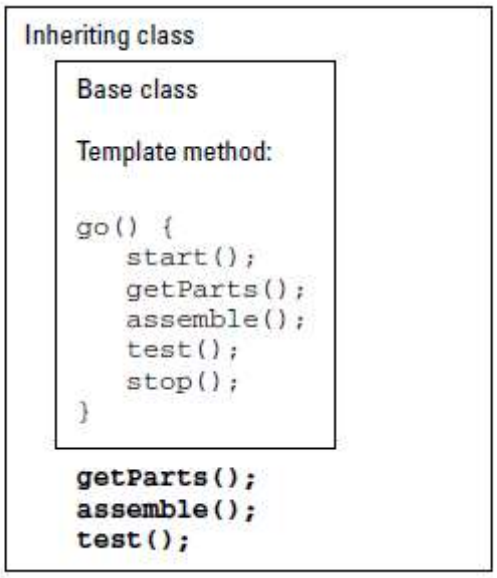

Figure 7-4: Modifying the inherited methods.

Đó là ý tưởng đằng sau mẫu thiết kế Template Method – Một chức năng bao gồm nhiều bước sẽ được tùy chỉnh bởi lớp con. Trong trường hợp bạn cần hai robot, một robot lắp ráp ô tô, một robot nướng bánh, mọi việc sẽ như thế nào?

Bạn bắt đầu bằng cách tạo một bộ khuôn Template trong một lớp trừu tượng abstract (để lớp khác có thể kế thừa nó), gọi là RobotTemplate

```
public abstract class RobotTemplate
{
  public final void go()
  {
    start();
    getParts();
    assemble();
    test();
    stop();
  }
  .
  .
  .
}
```

Và lớp này cũng cài đặt việc thực hiện mặc định cho từng chức năng trong hàm *algorithm, start, getParts, assemble, test* và *stop*.

```
public abstract class RobotTemplate
{
  public final void go()
  {
    start();
    getParts();
    assemble();
    test();
    stop();
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting parts....");
  }

  public void assemble()
  {
    System.out.println("Assembling....");
  }

  public void test()
  {
    System.out.println("Testing....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }
}
```

Nếu một con robot sử dụng đúng các phương thức này, ví dụ như hàm *start* và *stop*, chúng ta không cần phải viết lại chúng. Ngược lại bạn có thể thay đổi các phương thức này trong các lớp con.

Ví dụ, bạn có thể sử dụng RobotTemplate để tạo một con robot lắp ráp ô tô. Bạn có thừa kế từ lớp trừu tượng RobotTemplate trong một lớp mới, lớp AutomotiveTobot.

```
public class AutomotiveRobot extends RobotTemplate
{
    .
    .
    .
}
```

Robot lắp ráp ô tô này phải viết lại một số hàm của RobotTemplate như hàm *getParts* sẽ thông báo *"Getting a carburetor..."*, hàm *assemble* sẽ thông báo *"Installing the carburetor..."*, và hàm *test* sẽ thông báo *"Revving the engine..."*. Bạn thấy đó, bạn có thể tùy chỉnh các bước trong một thuật toán được cung cấp bởi một bộ khuôn template:

```
public class AutomotiveRobot extends RobotTemplate
{
  public void getParts()
  {
    System.out.println("Getting a carburetor....");
  }

  public void assemble()
  {
    System.out.println("Installing the carburetor....");
  }

  public void test()
  {
    System.out.println("Revving the engine....");
  }
}
```

Bạn cũng có thể tùy chỉnh mã nguồn dựa trên template bằng cách thêm vào một số hàm, ví dụ như hàm khởi tạo sẽ nhận tên của con robot, và hàm *getName* sẽ trả về tên này.

```
public class AutomotiveRobot extends RobotTemplate
{
  private String name;

  public AutomotiveRobot(String n)
  {
    name = n;
  }

  public void getParts()
  {
    System.out.println("Getting a carburetor....");
  }

  public void assemble()
  {
    System.out.println("Installing the carburetor....");
  }

  public void test()
  {
    System.out.println("Revving the engine....");
  }

  public String getName()
  {
    return name;
  }
}
```

Tuyệt vời. Bạn đã kế thừa phương thức *go* từ template, và tùy chỉnh nó cho robot lắp ráp ô tô.

Bạn cũng có thể tùy chỉnh hàm *go* kế thừa từ template, trong trường hợp tạo robot nướng bánh. Bạn tạo lớp mới CookieRobot, kế thừa từ lớp RobotTemplate. Bạn có thể viết lớp CookieRobot bằng cách làm cho hàm *getParts* thông *báo "Getting flour and sugar..."*, hàm *assemble* thông báo *"Baking a cookie..."*, và hàm *test* thông báo *"Crunching a cookie..."*

```
public class CookieRobot extends RobotTemplate
{
  private String name;

  public CookieRobot(String n)
  {
     name = n;
  }

  public void getParts()
  {
    System.out.println("Getting flour and sugar....");
  }

  public void assemble()
  {
    System.out.println("Baking a cookie....");
  }

  public void test()
  {
    System.out.println("Crunching a cookie....");
  }

  public String getName()
  {
    return name;
  }
}
```

Tới giờ, bạn đã sử dụng hàm *go* từ bộ khuôn template để tạo hai lớp mới, AutomotiveRobot và CookieRobot, và bạn đã viết lại một số bước trong thuật toán tùy thuộc vào hai loại robot khác nhau. Bạn đã không phải viết lại hai lớp này từ đầu.

Kiểm tra việc tạo Robot

Bạn hãy tạo hai đối tượng của hai lớp AutomotiveRobot và CookieRobot, và gọi hàm *go* như sau:

```
public class TestTemplate
{
  public static void main(String args[])
  {
    AutomotiveRobot automotiveRobot =
      new AutomotiveRobot("Automotive Robot");

    CookieRobot cookieRobot = new CookieRobot("Cookie Robot");

    System.out.println(automotiveRobot.getName() + ":");
    automotiveRobot.go();
    System.out.println();
    System.out.println(cookieRobot.getName() + ":");
    cookieRobot.go();
  }
}
```

Khi bạn chạy thử chương trình, bạn có thể thấy rằng bạn thật sự có thể tùy chỉnh một số bước trong thuật toán của hai loại robot khác nhau.

```
Automotive Robot:
Starting....
Getting a carburetor....
Installing the carburetor....
Revving the engine....
Stopping....

Cookie Robot:
Starting....
Getting flour and sugar....
Baking a cookie....
Crunching a cookie....
Stopping....
```

Thêm vào một "hook" (ND: Hook – móc câu – một kỹ thuật chặn bắt thông điệp chương trình)

Bạn cũng có thể thêm vào một hook trong thuật toán. Một hook là phương pháp kiểm soát một số khía cạnh của thuật toán. Ví dụ, nếu bạn muốn phần kiểm tra testing trong thuật toán Robot có trả về kết quả đúng không, bạn có thể thêm vào một điều kiện, một hàm hook có tên *testOK* như sau:

```
public abstract class RobotHookTemplate
{
  public final void go()
  {
    start();
    getParts();
    assemble();
    if (testOK()){
      test();
    }
    stop();
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting parts....");
  }

  public void assemble()
  {
    System.out.println("Assembling....");
  }

  public void test()
  {
    System.out.println("Testing....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }

  public boolean testOK()
  {
    return true;
  }
}
```

Mặc định, bạn có thể bỏ qua hàm hook *testOK* – nếu không làm gì khác, thuật toán Robot sẽ gọi đầy đủ các bước, bao gồm cả hàm test. Tuy nhiên bạn có thể "câu móc" vào thuật toán bằng cách viết lại hàm testOK trong một lớp con, lớp CookieHookRobot, nơi mà hàm *testOK* sẽ trả về giá trị false, không phải là true.

```
public class CookieHookRobot extends RobotHookTemplate
{
  private String name;

  public CookieHookRobot(String n)
  {
     name = n;
  }

  public void getParts()
  {
    System.out.println("Getting flour and sugar....");
  }

  public void assemble()
  {
    System.out.println("Baking a cookie....");
  }

  public String getName()
  {
    return name;
  }

  public boolean testOK()
  {
    return false;
  }

}
```

Bởi vì hàm hook *testOK* trả về giá trị false, thuật toán Robot sẽ không gọi hàm test từ hàm *go*, bạn có thể xem mã sau:

```
  public final void go()
  {
    start();
    getParts();
    assemble();
    if (testOK()){
      test();
    }
    stop();
  }
```

Kiểm tra hàm hook:

Bây giờ tạo chương trình, và gọi hàm *cookieHookRobot.go:*

```
public class TestHookTemplate
{
  public static void main(String args[])
  {
    CookieHookRobot cookieHookRobot =
      new CookieHookRobot("Cookie Robot");

    System.out.println(cookieHookRobot.getName() + ":");
    cookieHookRobot.go();
  }
}
```

Bạn sẽ thấy thuật toán Robot thực hiện, trừ bước test:

```
Cookie Robot:
Starting....
Getting flour and sugar....
Baking a cookie....
Stopping....
```

Bạn thấy đó, bạn đã không phải làm bất cứ thứ gì với hàm hook, tuy nhiên nếu bạn muốn, bạn có thể tác động lên việc thực hiện của thuật toán. Nếu bạn xây dựng một thuật toán sử dụng nhiều hàm trừu tượng, từng hàm này sẽ được viết lại ở lớp con, mặc khác, hàm hook sẽ không phải viết lại, trừ khi bạn muốn thay đổi việc thực thi mặc định của thuật toán.

Bạn sử dụng mẫu thiết kế Template Method khi bạn có một thuật toán với nhiều bước và bạn muốn cho phép tùy chỉnh chúng trong lớp con. Thật dễ dàng. Bằng cách viết lại các hàm đã được khai báo trong lớp trừu tượng, bạn sẽ thay đổi được theo cách bạn muốn.

Mẫu thiết kế Template Method là một ý tưởng tuyệt vời khi bạn có một thuật toán nhiều bước mà chính bạn có thể tùy chỉnh nó. Có một mẫu thiết kế khác cũng làm việc giống vậy, mà tôi sẽ thảo luận trong phần tới của chương, nó là mẫu Builder.

Xây dựng Robots với mẫu Builder

"Tin tốt!" Giám đốc điều hành của GigundoCorp reo lên, trong khi phóng như bay vào phòng họp. "Khách hàng của chúng ta nói rằng họ muốn kiểm soát nhiều hơn tính năng của Robot, vì vậy chúng ta không thể sử dụng những bộ khuôn Template đã viết sẵn được nữa. Bây giờ họ muốn họ có thể chọn hành động mà robot sẽ thực hiện"

"Để tôi làm rõ chỗ này", bạn nói "Đầu tiên, chúng ta thiết lập mọi thứ , robot khởi động, nhận nguyên liệu, lắp ráp, kiểm tra và dừng. Nhưng bây giờ khách hàng lại muốn kiểm

soát trình tự này và chọn lựa những chức năng họ muốn? Có thể là robot khởi động, rồi kiểm tra, rồi lắp ráp, rồi dừng?"

"Đúng vậy", Giám đốc nói

"Đây là thời điểm để sử dụng một mẫu thiết kế mới", bạn nói

"Tôi e rằng phải làm như vậy", Giám đốc nói.

Những quy định của khách hàng

Trong mẫu thiết kế Template Method, vấn đề chính là những thuật toán nhiều bước – bạn có thể cài đặt nó theo cách bạn muốn, và những lớp con sử dụng theo cách bạn đã thiết lập (Mặc dù bạn có thể viết lại một số bước, nhưng quy trình vẫn không thay đổi). Nhưng bây giờ tình hình đã khác – khách hàng muốn họ thiết lập trình tự hoạt động và số lượng các bước của thuật toán. Vì vậy mã nguồn mà bạn đã phát triển không còn là trung tâm chính nữa, bạn phải đóng gói nó trong một lớp mới, lớp builder.

Mẫu Template Method mà ta đã được làm quen trong phần trước cho phép bạn tùy chỉnh các bước của một thuật toán bằng cách viết lại các bước trong thuật toán như hình sau:

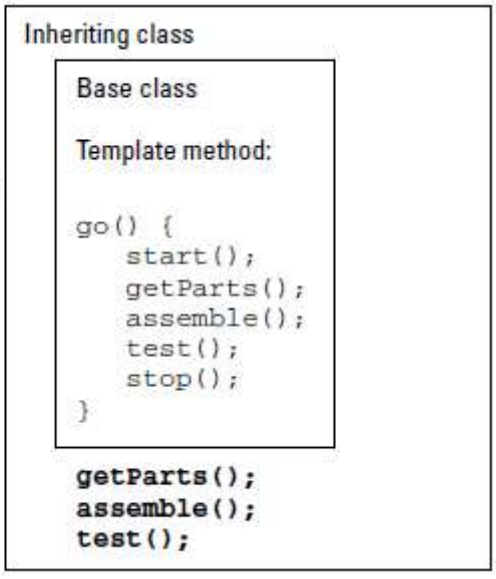

Figure 7-5: The Template Method pattern allows you to customize the multi-step algorithm.

Mọi chức năng đều dựa trên khuôn mẫu Template trong mẫu thiết kế này, và bạn có thể tùy chỉnh template theo cách bạn muốn. Nhưng bây giờ bạn không còn điều khiển thuật toán nữa, thay vào đó chính khách hàng thực hiện. Họ tạo robot với những chức năng và trình tự họ muốn. Ví dụ để thêm hành động khởi động, khách hàng có thể gọi hàm

addStart. Để thêm hành động kiểm tra, họ gọi hàm *addTest* và vân vân. Hình minh họa như sau:

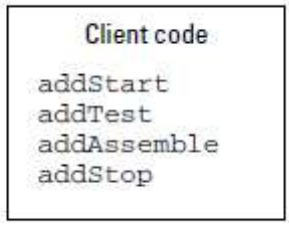

Figure 7-6: Customizing client code with an addStart method.

Để có thể đáp ứng yêu cầu kiếm soát hành động robot của khách hàng GigundoCorp, bạn phải chuyển mã nguồn cũ qua một lớp mới, lớp CookieRobotBuilder, lớp này hỗ trợ các hàm addStart, addTest, addAssemble và hàm addStop, như hình sau:

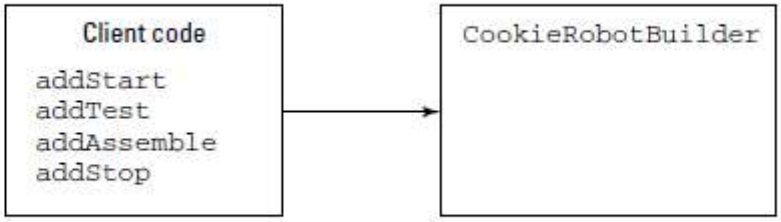

Figure 7-7: Separating out the construction code into the Cookie Robot Builder class.

Nhờ đó, khách hàng có thể sử dụng CookieRobotBuilder để tạo robot nướng bánh. Khi khách hàng tạo xong robot, mã nguồn sẽ gọi hàm getRobot của đối tượng CookieRobotBuilder để nhận về một robot mới, như hình vẽ sau:

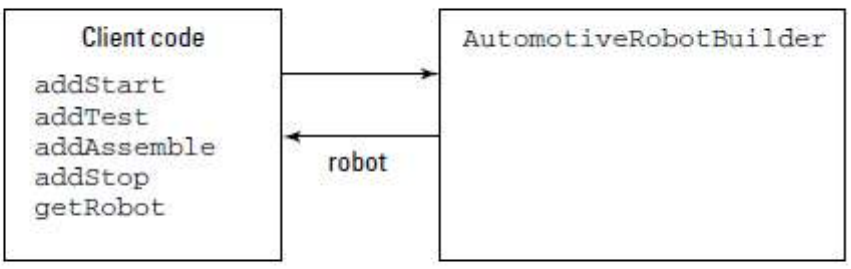

Figure 7-9: Using the Automotive Robot Builder class for construction instead of the Cookie Robot Builder class.

Và bây giờ khách hàng đã nắm quyền kiểm soát các thuật toán, bạn không phải kế thừa một template mẫu nữa.Thay vì vậy, để tạo một loại khác của robot, bạn cho phép khách hàng sử dụng những đối tượng builder khác nhau.

Ý tưởng chính như sau: bây giờ khách hàng có thể thiết lập trình tự và số lượng các bước trong thuật toán, và chọn lựa đúng đối tượng builder để tạo ra robot mà họ muốn.

Sách của GoF nói rằng, mẫu thiết kế Builder "Tách rời việc tạo dựng một đối tượng phức tạp ra khỏi bản thân đối tượng vì vậy cho phép cùng một quá trình tạo dựng có thể tạo ra nhiều loại đối tượng khác nhau"

Khác biệt lớn nhất giữa mẫu Template Method và mẫu Builder là ai sẽ tạo ra trình tự các bước trong thuật toán. Trong mẫu Template, bạn là người tạo ra trình tự, và các lớp con sẽ hiện thực chúng. Trong mẫu Builder, khách hàng sẽ thiết lập trình tự và số lượng các bước trong thuật toán, và hoán đổi giữa các builder mà phải cung cấp để tạo ra các đối tượng khác nhau thể hiện thuật toán đó.

Sử dụng mẫu thiết kế Builder khi bạn muốn khách hàng kiểm soát được quá trình tạo dựng. Ví dụ, đây là mẫu thiết kế mà bạn muốn khi bạn xây dựng robot sử dụng cùng một quá trình khởi tạo nhưng muốn có thể tạo ra những con robot khác nhau. Tất cả những gì khách hàng cần là gọi những builder khác nhau – quá trình xây dựng vẫn như cũ. Đây là một ví dụ khác, bạn muốn đọc một đoạn văn bản và xây dựng một tài liệu, nhưng bạn lại không biết định dạng chính xác của nó là RTF, Microsoft Word, hay văn bản đơn giản… Mặc dù quá trình tạo dựng là giống nhau cho từng tài liệu, bạn có thể sử dụng những builder khác nhau để tạo dựa vào kiểu của loại tài liệu.

Nói cách khác, khi khách hàng muốn kiểm soát quá trình tạo dựng, nhưng bạn vẫn muốn có thể tạo ra nhiều đối tượng khác nhau, mẫu Builder sẽ giúp bạn thực hiện điều đó.

Ghi nhớ: Mẫu thiết kế này tương tự với mẫu Factory, nhưng mẫu Factory là trung tâm trong quá trình khởi tạo một bước, chứ không cài đặt nhiều bước như ở đây.

Cho phép khách hàng tạo Robot

Khi bạn sử dụng mẫu Builder, khách hàng sẽ phụ trách quá trình tạo dựng. Khách hàng sử dụng đối tượng xây dựng builder của bạn để làm những gì họ muốn. Để cho phép khách hàng tạo robot thể hiện một loạt các hành động – khởi động, lắp ráp, ngừng… – Tôi tạo ra một giao diện interface Robot Builder hỗ trợ các hàm như sau: addStart, addGetParts, addAssemble, addTest và addStop:

Ví dụ để tạo một robot với các hành động start, test, assemble và sau đó là stop, khách hàng chỉ cần gọi hàm addStart, addTest, addAssemble và addStop của đối tượng xây dựng builder theo đúng trình tự đó. Khi robot đã được tạo xong, khách hàng chỉ cần gọi hàm getRobot của Builder để nhận về một robot mới. Và đối tượng robot mới này có hỗ trợ hàm go, hàm này sẽ thực hiện hàng loạt hành động mà bạn đã tạo dựng trước đó.

Bởi vì bạn có nhiều loại đối tượng builder để tạo nhiều loại robot khác nhau – ví dụ như builder xây dựng robot làm bánh, builder xây dựng robot lắp ráp ô tô – Tôi sẽ tạo một giao diện interface RobotBuilder mà tất cả các builder sẽ hiện thực giao diện này. Đây là những hàm mà các robot builder phải hiện thực, từ hàm addStart tới hàm addStop, kể cả hàm getRobot. Xem mã sau:

```
public interface RobotBuilder
{
  public void addStart();
  public void addGetParts();
  public void addAssemble();
  public void addTest();
  public void addStop();
  public RobotBuildable getRobot();
}
```

Tôi bắt đầu tạo builder cho robot làm bánh, CookieRobotBuilder, cũng giống như tất cả các builder khác, builder này cần phải hiện thực giao diện RobotBuilder

```
public class CookieRobotBuilder implements RobotBuilder
{
    .
    .
    .
}
```

Đối tượng robot trong mã nguồn trên, dựa trên lớp CookieRobotBuildable, sẽ được nói tới trong phần "Tạo một vài robot tương thích". Đối tượng robot được tạo sẽ là một đối tượng CookieRobotBuildalbe. Vì thế chúng ta cần một biến để lưu trữ đối tượng này, mã như sau:

```
public class CookieRobotBuilder implements RobotBuilder
{
  CookieRobotBuildable robot;

  public CookieRobotBuilder()
  {
    robot = new CookieRobotBuildable();
  }
    .
    .
    .
}
```

Khách hàng có thể cài đặt các hành động cho robot như start, stop, test, assemble, getParts ... theo trình tự bất kì. Để lưu lại trình tự này, tôi sử dụng kiểu ArrayList, với đối tượng actions như sau:

```
import java.util.*;

public class CookieRobotBuilder implements RobotBuilder
{
  CookieRobotBuildable robot;
  ArrayList<Integer> actions;

  public CookieRobotBuilder()
  {
    robot = new CookieRobotBuildable();
    actions = new ArrayList<Integer>();
  }
    .
    .
    .
}
```

Cách dễ dàng nhất để lưu trữ trình tự các hành động của robot trong mảng danh sách actions là gán từng giá trị số nguyên cho từng hành động, như hình sau:

- start = 1
- getParts = 2
- assemble = 3, and so on

Tôi lưu đối tượng số nguyên trong một mảng danh sách ArrayList. Ví dụ, khi khách hàng muốn thêm hành động start, chương trình gọi hàm addStart, và robot builder sẽ thêm một đối tượng số nguyên có giá trị 1 vào mảng danh sách actions, và cứ thế tiếp tục... Đây là tất cả các hàm để thêm chức năng cho robot trong builder:

```
import java.util.*;

public class CookieRobotBuilder implements RobotBuilder
{
  CookieRobotBuildable robot;
  ArrayList<Integer> actions;

  public CookieRobotBuilder()
  {
    robot = new CookieRobotBuildable();
    actions = new ArrayList<Integer>();
  }

  public void addStart()
  {
    actions.add(new Integer(1));
  }

  public void addGetParts()
  {
    actions.add(new Integer(2));
  }

  public void addAssemble()
  {
    actions.add(new Integer(3));
  }

  public void addTest()
  {
    actions.add(new Integer(4));
  }

  public void addStop()
  {
    actions.add(new Integer(5));
  }
    .
    .
    .
}
```

Khi khách hàng muốn tạo một đối tượng robot, họ sẽ gọi hàm getRobot của builder này. Khi hàm này được gọi, bạn biết rằng quá trình khởi tạo đã hoàn tất, vì vậy bạn có thể cài đặt robot bằng cách chuyển giao cho nó tham số mảng danh sách actions mà nó sẽ thực thi. Trong ví dụ này, từng robot sẽ được cài đặt bằng cách chuyển tham số actions thông qua hàm loadActions. Mã như sau:

```
import java.util.*;

public class CookieRobotBuilder implements RobotBuilder
{
  CookieRobotBuildable robot;
  ArrayList<Integer> actions;

  public CookieRobotBuilder()
  {
    robot = new CookieRobotBuildable();
    actions = new ArrayList<Integer>();
  }

  public void addStart()
  {
    actions.add(new Integer(1));
  }
     .
     .
     .
  public void addStop()
  {
    actions.add(new Integer(5));
  }

  public RobotBuildable getRobot()
  {
    robot.loadActions(actions);
    return robot;
  }

}
```

Vậy là hoàn thành phần đối tượng xây dựng builder, nó cho phép khách hàng tự cài đặt robot theo trình tự họ muốn. Vậy làm sao để tạo lớp Robot mà ta đã sử dụng ở trên?

Tạo một số robot thích hợp:

Từng loại thợ xây builder sẽ tạo ra một loại robot khác nhau, từng robot lại được tạo từ lớp cơ sở của nó, như lớp CookieRobotBuildable hay AutomotiveRobotBuildalble. Tất cả các robot phải có cùng một hàm go để thực hiện các chức năng. Vì vậy bạn có tạo một giao diện interface với tên RobotBuildable để chắc chắn rằng mọi robot đều phải hiện thực giao diện này. Mã như sau:

```
public interface RobotBuildable
{
  public void go();
}
```

Bây giờ tất cả Robot đều phải hiện thực giao diện này. Đây là cách lớp RobotBuildable hoạt động. Bạn có thể nạp robot với mảng danh sách actions, thông qua hàm loadActions, với tham số actions đã được đối tượng builder tạo trước. Xem mã sau:

```
import java.util.*;

public class CookieRobotBuildable implements RobotBuildable
{
  ArrayList<Integer> actions;

  public CookieRobotBuildable()
  {
  }

  public void loadActions(ArrayList a)
  {
    actions = a;
  }
    .
    .
    .
}
```

Khi khách hàng muốn robot thực hiện các hành động được cài đặt sẵn, họ gọi hàm go. Trong hàm go, bạn có thể duyệt qua mảng actions và gọi từng hàm tương ứng với chức năng đó. Ví dụ bạn duyệt qua đối tượng số nguyên "1" trong actions, bạn sẽ gọi hàm start, duyệt tới số "2" bạn gọi hàm getParts và vân vân..Bạn có thể sử dụng một đối tượng Iterator và phát biểu switch trong hàm go như sau:

```
import java.util.*;

public class CookieRobotBuildable implements RobotBuildable
{
  ArrayList<Integer> actions;

  public CookieRobotBuildable()
  {
  }

  public final void go()
  {
    Iterator itr = actions.iterator();
    while(itr.hasNext()) {
      switch ((Integer)itr.next()){
        case 1:
          start();
          break;
        case 2:
          getParts();
          break;
        case 3:
          assemble();
          break;
        case 4:
          test();
          break;
        case 5:
          stop();
          break;
      }
    }
  }
    .
    .
    .
  public void loadActions(ArrayList a)
  {
    actions = a;
  }
}
```

Ghi chú: Bạn cũng cần phải thêm các hàm cho từng hành động như : hàm start (hiển thị chữ "Starting..."), hàm getParts (hiển thị chữ Getting flour and sugar...) vân vân.

```
import java.util.*;

public class CookieRobotBuildable implements RobotBuildable
{
  ArrayList<Integer> actions;

  public CookieRobotBuildable()
  {
  }

  public final void go()
  {
    Iterator itr = actions.iterator();

    while(itr.hasNext()) {
      switch ((Integer)itr.next()){
```

```
        case 1:
          start();
          break;
        case 2:
          getParts();
          break;
        case 3:
          assemble();
          break;
        case 4:
          test();
          break;
        case 5:
          stop();
          break;
      }
    }
  }

  public void start()
  {
    System.out.println("Starting....");
  }

  public void getParts()
  {
    System.out.println("Getting flour and sugar....");
  }

  public void assemble()
  {
    System.out.println("Baking a cookie....");
  }

  public void test()
  {
    System.out.println("Crunching a cookie....");
  }

  public void stop()
  {
    System.out.println("Stopping....");
  }

  public void loadActions(ArrayList a)
  {
    actions = a;
  }
}
```

Vậy là hoàn tất lớp CookieRobotBuilable. Giờ bạn đã có đối tượng xây dựng Builder và robot. Giờ là lúc để thử nghiệm chúng

```
import java.io.*;

public class TestRobotBuilder
{
  public static void main(String args[])
  {
    String response = "a";

    System.out.print(
      "Do you want a cookie robot [c] or an automotive one [a]? ");
    BufferedReader reader = new
      BufferedReader(new InputStreamReader(System.in));

    try{
      response = reader.readLine();
    } catch (IOException e){
      System.err.println("Error");
    }
      .
      .
      .
}
```

Tùy thuộc vào loại robot mà user chọn lựa, đối tượng builder dành cho robot làm bánh hay builder dành cho robot lắp ráp ô tô sẽ được tạo ra. Xem mã sau:

```
import java.io.*;

public class TestRobotBuilder
{
  public static void main(String args[])
  {
    RobotBuilder builder;
    String response = "a";

    System.out.print(
      "Do you want a cookie robot [c] or an automotive one [a]? ");
    BufferedReader reader = new
      BufferedReader(new InputStreamReader(System.in));

    try{
      response = reader.readLine();
    } catch (IOException e){
      System.err.println("Error");
    }

    if (response.equals("c")){
      builder = new CookieRobotBuilder();
    } else {
      builder = new AutomotiveRobotBuilder();
    }
      .
      .
      .
  }
}
```

Sau đó khách hàng tạo loại robot mà họ muốn, và sử dụng các hàm addStart, addGetParts, addAssemble, addTest và addStop theo trình tự họ muốn

```
import java.io.*;

public class TestRobotBuilder
{
  public static void main(String args[])
  {
    RobotBuilder builder;
    String response = "a";

    System.out.print(
      "Do you want a cookie robot [c] or an automotive one [a]? ");
    BufferedReader reader = new
      BufferedReader(new InputStreamReader(System.in));
    try{
      response = reader.readLine();
    } catch (IOException e){
      System.err.println("Error");
    }

    if (response.equals("c")){
      builder = new CookieRobotBuilder();
    } else {
      builder = new AutomotiveRobotBuilder();
    }

    //Start the construction process.

    builder.addStart();
    builder.addTest();
    builder.addAssemble();
    builder.addStop();
        .
        .
        .
  }
}
```

Sau khi robot được tạo, khách hàng gọi hàm getRobot, đối tượng robot trả về được lưu trong biến RobotBuildable. Và bạn có thể gọi hàm go của robot. Mã như sau:

```
import java.io.*;

public class TestRobotBuilder
{
  public static void main(String args[])
  {
    RobotBuilder builder;
    RobotBuildable robot;
    String response = "a";

    System.out.print(
      "Do you want a cookie robot [c] or an automotive one [a]? ");
    BufferedReader reader = new
      BufferedReader(new InputStreamReader(System.in));

    try{
      response = reader.readLine();
    } catch (IOException e){
      System.err.println("Error");
    }
    if (response.equals("c")){
      builder = new CookieRobotBuilder();
    } else {
      builder = new AutomotiveRobotBuilder();
    }

    //Start the construction process.

    builder.addStart();
    builder.addTest();
    builder.addAssemble();
    builder.addStop();

    robot = builder.getRobot();

    robot.go();
  }
}
```

Khách hàng có thể tạo robot làm bánh hay robot lắp ráp ô tô một cách đơn giản thông qua việc chọn đúng builder. Đây là kết quả:

```
Do you want a cookie robot [c] or an automotive one [a]? c
Starting....
Crunching a cookie....
Baking a cookie....
Stopping....
```

Và đây là kết qủa việc tạo robot lắp ráp ôtô, sử dụng cùng một quy trình khởi tạo:

```
Do you want a cookie robot [c] or an automotive one [a]? a
Starting....
Revving the engine....
Installing the carburetor....
Stopping....
```

Tuyệt vời. Bạn có thể đưa builder cho khách hàng, giúp khách hàng có thể kiểm soát quá trình tạo dựng đối tượng.

Download source code C sharp tại đây.

Design Patterns For Dummies
Design Patterns
< DP4Dummies – Chương 6: Adapter, Facade
> DP4Dummies – Chương 8: Iterator, Composite