DP4Dummies – Chương 5: Singleton, Flyweight

CHƯƠNG V: TỪ MỘT CHO TỚI NHIỀU – MẪU DUY NHẤT SINGLETON VÀ MẪU FLYWEIGHT

Trong chương này, chúng ta sẽ đi qua các nội dung sau:

- Sử dụng mẫu duy nhất Singleton
- Ví dụ về Singleton
- Đồng bộ hóa để loại bỏ các vấn đề rắc rối trong đa luồng.
- Một cách tốt hơn để xử lý đa luồng
- Sử dụng mẫu "hạng ruồi" flyweight

Là một nhà tư vấn được trả lương hậu hĩnh tại MegaGigaco, bạn phải xử lý các sự cố về hiệu năng hệ thống. "Hệ thống hình như ngày càng chậm chạp hơn." Các lập trình viên nói:

"Hmm," bạn nói, "Tôi lưu ý các bạn rằng chúng ta đang có một đối tượng dữ liệu có kích thước khá lớn, khoảng 20Mb"

"Vâng", họ nói.

"Cùng một thời điểm, các bạn sử dụng bao nhiêu đối tượng này?"

"Khoảng 219", các lập trình viên nói

"Trời, vậy các bạn sử dụng 219 đối tượng 20Mb trong lúc chương trình hoạt động?" Bạn nói. "Chẳng lẽ không ai thấy được vấn đề ở đây à?"

"Không", họ đồng thanh nói.

Bạn nói với họ "Các bạn sử dụng quá nhiều tài nguyên hệ thống. Các bạn có hàng trăm đối tượng to lớn mà máy tính phải xử lý. Các bạn có thật sự cần tất cả chúng?"

"Vâng..." họ nói.

“Tôi nghĩ là không,” bạn nói. “Tôi sẽ sửa chữa vấn đề này bằng cách sử dụng mẫu duy nhất Singleton.”

Chương này nói về việc kiểm soát số lượng đối tượng mà bạn phải tạo ra trong mã nguồn của mình. Có hai mẫu thiết kế đặc biệt giúp ích cho bạn: mẫu duy nhất Singleton và mẫu “hạng ruồi” flyweight.

Với mẫu duy nhất Singleton, bạn luôn đảm bảo rằng chỉ có duy nhất một đối tượng cho một lớp cụ thể trong suốt ứng dụng. Với mẫu “hạng ruồi” flyweight, bạn cũng có duy nhất một đối tượng cho một lớp, nhưng khi nhìn vào mã của bạn, ta có thể thấy giống như đang có nhiều đối tượng vậy. Đây là một thủ thuậ t khéo léo.

Tạo một đối tượng duy nhất với mẫu duy nhất Singleton.

Tôi bắt đầu với mẫu Singleton và xử lý rắc rối mà lập trình viên MegaGigaCo gặp phải. Họ muốn chắn chắc rằng chỉ tạo duy nhất một đối tượng cho một lớp cụ thể mặc cho người khác có cố gắng tạo bao nhiêu đối tượng đi nữa.

Các lập trình viên đang tạo ra hàng trăm đối tượng Database trong mã nguồn, và rắc rối là từng đối tượng này có kích thước rất lớn. Đâu là giải pháp? Mẫu duy nhất Singeton là câu trả lời.

Mẫu duy nhất Singleton chắc chắn rằng bạn có thể khởi tạo chỉ duy nhất một đối tượng cho một lớp. Nếu bạn không sử dụng mẫu thiết kế này, toán tử new như thường sử dụng, sẽ tạo ra liên tiếp nhiều đối tượng mới như sau:

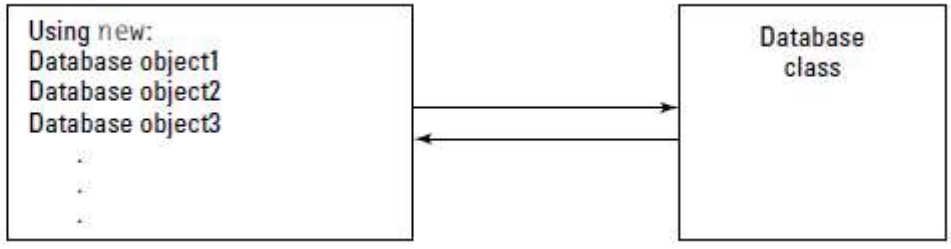

Figure 5-1: Creating object1, object2, object3, and many more.

Ghi nhớ: Để chắc chắn rằng bạn chỉ có duy nhất một đối tượng, mặc cho người khác có hiện thực bao nhiêu phiên bản đi nữa, hãy sử dụng mẫu duy nhất Singleton. Cuốn sách GoF nói rằng, mẫu Singleton “Đảm bảo rằng một lớp chỉ có duy nhất một thể hiện và cung cấp một biến toàn cục để truy cập nó”

Bạn sử dụng mẫu Singleton khi bạn muốn hạn chế việc sử dụng tài nguyên (thay vì việc tạo không hạn chế số lượng đối tượng) hoặc khi bạn cần phải xử lý một đối tượng nhạy cảm, mà dữ liệu của nó không thể chia sẻ cho mọi thể hiện, như registry của Windows chẳng hạn.

Gợi ý: Ngoài đối tượng bản ghi registry, bạn có thể sử dụng mẫu Singleton khi bạn muốn hạn chế số lượng các thể hiện được tạo bởi vì bạn muốn chia sẻ dữ liệu của các đối tượng này. Ví dụ như khi bạn có một đối tượng cửa sổ window hay hộp thoại dialog, cần phải hiện thị và thay đổi dữ liệu, bạn sẽ không muốn tạo nhiều thể hiện của đối tượng này, vì bạn sẽ bị bối rối trong việc phải truy cập dữ liệu của thể hiện nào.

Việc tạo một đối tượng duy nhất cũng rất quan trọng khi bạn sử dụng đa luồng và khi bạn không muốn sự đụng độ dữ liệu xảy ra. Ví dụ bạn đang làm việc với một đối tượng cơ sở dữ liệu, và các thể hiện khác cũng làm việc trên cùng cơ sở dữ liệu đó, việc đụng độ có thể gây ra các vấn đề nghiêm trọng. Tôi sẽ thảo luận cách làm việc với mẫu Singleton và đa luồng trong chương này.

Bất cứ khi nào bạn thật sự cần duy nhất một thể hiện của một lớp, hãy nghĩ tới mẫu Singleton (thay vì dùng toán tử new).

Tạo lớp cơ sở dữ liệu Database dựa trên kiểu Singleton

Giờ là lúc bạn bắt tay vào viết mã nguồn. Bạn sẽ tạo một lớp tên Database mà các lập trình viên trong công ty sẽ sử dụng. Lớp này có một hàm khởi dựng đơn giản, như mã sau:

```
public class Database
{
  private int record;
  private String name;

  public Database(String n)
  {
    name = n;
    record = 0;
  }
    .
    .
    .
}
```

Bạn cần phải thêm vào hai hàm editRecord, cho phép bạn chỉnh sửa một bản ghi, và hàm getName, trả về tên gọi của Database.

```
public class Database
{
  private int record;
  private String name;

  public Database(String n)
  {
    name = n;
    record = 0;
  }

  public void editRecord(String operation)
  {
    System.out.println("Performing a " + operation +
      " operation on record " + record +
      " in database " + name);
  }

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

Tới giờ mọi việc vẫn tốt đẹp. Bất cứ khi nào bạn tạo một đối tượng bằng toán tử new, một đối tượng mới sẽ được tạo ra. Nếu bạn tạo 3 database, bạn sẽ có 3 đối tượng như sau:

```
Database dataOne = new Database("Products");
        .
        .
        .
Database dataTwo = new Database("Products Also");
        .
        .
        .
Database dataThree = new Database("Products Again");
        .
        .
        .
```

Làm sao để bạn có thể tránh việc tạo một đối tượng mới khi sử dụng toán tử new? Đây là một giải pháp – làm cho hàm khởi dụng từ toàn cục (public) trở thành cục bộ (private)

```
  private Database(String n)
  {
    name = n;
    record = 0;
  }
```

Điều này ngăn cản mọi người sử dụng hàm khởi dựng, ngoài trừ chính trong lớp này gọi tới. Nhưng đợi một chút, có gì không ổn ở đây? Ai ở trên trái đất này lại cần có một hàm khởi dựng riêng tư vậy? Làm sao bạn có thể tạo một đối tượng khi bạn không thể gọi hàm khởi tạo nó?

Bạn đã làm cho hàm khởi dựng trở nên riêng tư và cách duy nhất để phần còn lại của thế giới khởi tạo đối tượng đó là thêm vào một hàm tạo đối tượng, và gọi nó khi bạn chắn chắn muốn tạo một đối tượng duy nhất cho lớp này.

Hãy xem đoạn mã sau:

```
public class Database
{
  private int record;
  private String name;
  private Database(String n)
  {
    name = n;
    record = 0;
  }
    .
    .
    .
```

OK. Đầu tiên bạn ngăn chặn việc khởi tạo bằng toán tử new. Và bây giờ cách duy nhất là tạo một hàm để gọi việc khởi tạo đối tượng, thông thường hàm này có tên getInstance (hay createInstance hoặc một cái tên cụ thể như createDatabase cũng được). Chú ý rằng hàm này được gán phạm vi công cộng và là một phương thức tĩnh để bạn có thể truy cập tới nó thông qua tên lớp (ví dụ như Database.getInstance()) (ND: public và static là hai khái niệm trong OOP. Public giúp hàm có thể được sử dụng từ bất kì lớp khác, static giúp ta có thể sử dụng hàm trực tiếp từ tên lớp, không cần thông qua một đối tượng lớp.)

```
public class Database
{
  private int record;
  private String name;

  private Database(String n)
  {
    name = n;
    record = 0;
  }

  public static Database getInstance(String n)
  {
  }
    .
    .
    .
}
```

Hàm này sẽ trả về một đối tượng Database, nhưng hàm chỉ hoạt động khi có ít nhất một đối tượng đã tồn tại. Vì thế đầu tiên ta cần kiểm tra đối tượng này, tôi gọi nó là singleObject, xem nó đã tồn tại chưa? Nếu chưa, tôi sẽ tạo nó. Và sau đó trả giá trị nó về cho hàm.

```
public class Database
{
  private static Database singleObject;
  private int record;
  private String name;

  private Database(String n)
  {
    name = n;
    record = 0;
  }

  public static Database getInstance(String n)
  {
      if (singleObject == null){
        singleObject = new Database(n);
      }

      return singleObject;
  }
    .
    .
    .
}
```

Vấn đề đã được giải quyết. Bây giờ chỉ có duy nhất một đối tượng Database tồn tại trong cùng một thời điểm. (Vấn đề đa luồng ta sẽ giải quyết trong phần sau của chương). Việc gọi hàm getInstance sẽ cho ta một đối tượng như hình sau:

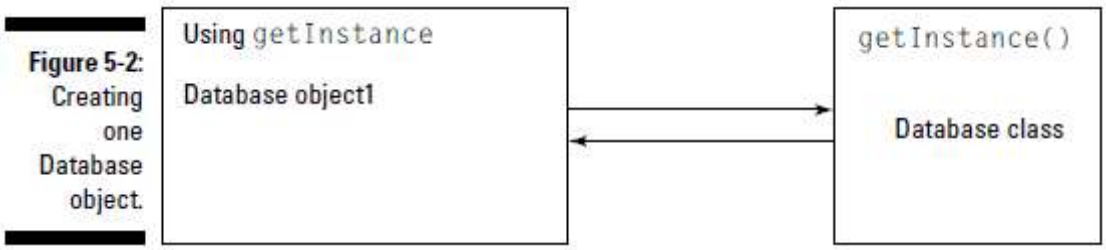

Figure 5-2: Creating one Database object.

Khi bạn gọi getInstance lần nữa, bạn sẽ nhận được cùng một đối tượng như lần đầu.

Không quan tâm đến việc bạn gọi bao nhiêu lần getInstance, bạn luôn nhận được cùng một đối tượng. Đó chính là cách bạn phải làm với mẫu singleton.

Chạy thử ví dụ với mẫu Singleton

Bắt đầu bằng việc tạo một đối tượng Database với tên là products, sau đó gọi hàm getName:

```
public class TestSingleton
{
  public static void main(String args[])
  {
    Database database;

    database = Database.getInstance("products");

    System.out.println("This is the " +
      database.getName() + " database.");
        .
        .
        .
  }
}
```

Sau đó bạn tiếp tục tạo một đối tượng Database với tên là employees, và gọi lại hàm getName để kiểm tra:

```
public class TestSingleton
{
  public static void main(String args[])
  {
    Database database;

    database = Database.getInstance("products");

    System.out.println("This is the " +
      database.getName() + " database.");

    database = Database.getInstance("employees");

    System.out.println("This is the " +
      database.getName() + " database.");
  }
}
```

Tuy nhiên đối tượng Database đã được tạo, vì vậy trong lần thứ hai, hàm getInstance vẫn trả về đối tượng Database cũ, và kết quả là bạn nhận được thông báo:

```
This is the products database.
This is the products database.
```

Quá rõ ràng. Bạn đã nhận được duy nhất một đối tượng cho dù đã thực hiện việc tạo hai lần. Cách thức bạn làm việc như sau: ngăn cản việc khởi tạo bằng toán tử new, và tạo một hàm mới để tạo đối tượng theo ý bạn. Đó chính là cách mẫu Singleton hoạt động.

Đừng quên vấn đề đa luồng

Hãy xem hàm getInstance trong ví dụ trên:

```
public static Database getInstance(String n)
{
    if (singleObject == null){
      singleObject = new Database(n);
    }

    return singleObject;
}
```

Có một lỗ hổng tiềm tàng ở đây, tuy nhỏ nhưng là một lỗ hổng rõ ràng, đó là khi làm việc với đa luồng. Hãy nhớ rằng, bạn muốn đảm bảo rằng chỉ có duy nhất một đối tượng Database tồn tại. Nhưng khi bạn có nhiều luồng chương trình chạy cùng lúc, bạn sẽ gặp rắc rối. Cụ thể là, hãy chú ý đoạn mã kiểm tra sự tồn tại của đối tượng Database:

```
public static Database getInstance(String n)
{
    if (singleObject == null){
      singleObject = new Database(n);
    }

    return singleObject;
}
```

Nếu có hai luồng cùng thực hiện hàm kiểm tra này một lúc, hai luồng này đều thỏa điều kiện của hàm if (tức chưa có đối tượng nào được tạo), và điều này có nghĩa là cả hai luồng đều tạo ra một đối tượng Database.

Làm sao để chỉnh sửa chỗ này? Một cách dễ dàng là sử dụng từ khóa synchronized (đồng bộ) trong Java, xem đoạn mã sau:

```
public class DatabaseSynchronized
{
  private static DatabaseSynchronized singleObject;
  private int record;
  private String name;
  private DatabaseSynchronized(String n)
  {
    name = n;
    record = 0;
  }

  public static synchronized DatabaseSynchronized getInstance(String n)
  {
      if (singleObject == null){
        singleObject = new DatabaseSynchronized(n);
      }

      return singleObject;
  }

  public void editRecord(String operation)
  {
    System.out.println("Performing a " + operation +
      " operation on record " + record +
      " in database " + name);
  }

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

Sử dụng từ khóa synchronized sẽ khóa việc truy cập vào hàm getInstance, trong khi hàm getInstance được chạy. Bất cứ luồng nào muốn gọi hàm getInstance, đều phải đợi hàm này hoạt động xong. Sử dụng kỹ thuật đồng bộ hóa synchronized là cách dễ nhất để thực thi việc đơn luồng trong gọi hàm, và kỹ thuật này đã giải quyết được vấn đề đa luồng.

Chạy thử chương trình với giải pháp đồng bộ hóa:

Bởi vì việc gọi hàm getInstance đã được đồng bộ hóa, bạn có gọi hàm từ nhiều luồng khác nhau. Xem mã sau:

```
public class TestSingletonSynchronized implements Runable
{
  public static void main(String args[])
  {
    TestSingletonSynchronized t = new TestSingletonSynchronized();
  }

  public TestSingletonSynchronized()
  {
    DatabaseSynchronized database;

    database = DatabaseSynchronized.getInstance("products");
        .
        .
        .
}
```

Đoạn mã cũng cho phép chạy một tiến trình mới để cố gắng tạo mới một đối tượng DatabaseSynchronized:

```
public class TestSingletonSynchronized implements Runable
{
  Thread thread;

  public static void main(String args[])
  {
    TestSingletonSynchronized t = new TestSingletonSynchronized();
  }

  public TestSingletonSynchronized()
  {
    DatabaseSynchronized database;

    database = DatabaseSynchronized.getInstance("products");

    thread = new Thread(this, "second");
    thread.start();

    System.out.println("This is the " +
      database.getName() + " database.");
  }
    .
    .
    .
}
```

Tiến trình mới cố gắng tạo một đối tượng mới DatabaseSynchronized với tên employees.

```
public class TestSingletonSynchronized implements Runable
{
  Thread thread;

  public static void main(String args[])
  {
    TestSingletonSynchronized t = new TestSingletonSynchronized();
  }

  public TestSingletonSynchronized()
  {
    DatabaseSynchronized database;
    database = DatabaseSynchronized.getInstance("products");

    thread = new Thread(this, "second");
    thread.start();

    System.out.println("This is the " +
      database.getName() + " database.");
  }

    public void run()
    {
      DatabaseSynchronized database =
        DatabaseSynchronized.getInstance("employees");

      System.out.println("This is the " +
        database.getName() + " database.");
    }
}
```

Nhưng như bạn có thể nhìn thấy, khi chương trình thực thi, chỉ duy nhất một đối tượng DatabaseSynchronized tồn tại. Đó là đối tượng products.

```
This is the products database.
This is the products database.
```

Kể từ khi bạn sử dụng kỹ thuật đồng bộ hóa trên hàm getInstance, bạn không còn lo lắng về vấn đề đa luồng nữa. Chỉ duy nhất một luồng được gọi hàm getInstance. Nó ngăn chặn việc tạo đối tượng bằng một bức tường an toàn, việc kiểm tra ở trên cho thấy, nếu đối tượng muốn tạo đã tồn tại, hàm sẽ không tạo, ngược lại, sẽ tạo đối tượng cho lớp.

Thoạt nhìn, điều này thật tuyệt vời, đồng bộ hóa hàm getInstance và giải quyết được vấn đề đa luồng, bảo vệ mã nguồn chống lại việc xung đột khi có nhiều tiến trình cùng tạo một đối tượng.

Tuy nhiên, vẫn còn một câu hỏi, việc đồng bộ hóa đã giải quyết vấn đề, nhưng đó có phải là cách tốt nhất. Việc đồng bộ hóa gây tốn tài nguyên hệ thống, Java buộc phải theo dõi từng tiến trình để cho xử lý khi nào thì cho phép tiến trình truy cập hàm getInstance, khi nào thì không cho phép.

Đồng bộ hóa hàm getInstance đã làm việc, nhưng với chi phí đáng kể. Có cách nào tốt hơn để giải quyết vấn đề này?

Tối ưu hóa việc xử lý đa luồng

Vấn đề khi bạn cố gắng xử lý từ khóa synchronized là đoạn mã kiểm tra việc tạo đối tượng sẽ bị phá hỏng bởi các tiến trình khác. Một cách tốt hơn để làm việc này là đảm bảo rằng đoạn mã kiểm tra không còn quan trọng nữa.

"Mọi việc thế nào?"Các lập trình viên hỏi trong kinh ngạc. "Nếu bạn không kiểm tra việc tạo đối tượng, làm sao bạn có thể chắc chắn là bạn không tạo thêm một đối tượng mới?"

Bạn giải thích "Bằng cách loại bỏ toàn bộ mã tạo đối tượng ra khỏi hàm getInstance. Tôi sẽ viết lại mã để chắc chắn chỉ một đối tượng được tạo ra. Và đối tượng sẽ được tạo ra trước khi bất cứ luồng chương trình nào có thể nắm bắt được nó.

"Hmm", các lập trình viên nói "Nghe có vẻ nó sẽ hoạt động"

Ý tưởng như sa:– Tạo đối tượng duy nhất mà bạn muốn ngay khi mã nguồn được nạp lần đầu tiên vào Java Virtual Machine (máy ảo Java, bộ máy biên dịch và thi hành Java). Không để hàm getInstance tạo đối tượng nữa. Chỉ cho phép hàm trả về đối tượng vừa tạo. Mã như sau:

```
public class DatabaseThreaded
{
  private static DatabaseThreaded singleObject =
    new DatabaseThreaded("products");
  private int record;
  private String name;

  private DatabaseThreaded(String n)
  {
    name = n;
    record = 0;
  }

        .
        .
        .
}
```

Tốt, giờ bạn đã tạo đối tượng duy nhất rồi. Tất cả những gì cần làm là cho hàm getInstance trả về đối tượng này

```
public class DatabaseThreaded
{
  private static DatabaseThreaded singleObject =
    new DatabaseThreaded("products");
  private int record;
  private String name;

  private DatabaseThreaded(String n)
  {
    name = n;
    record = 0;
  }

  public static synchronized DatabaseThreaded getInstance(String n)
  {
      return singleObject;
  }

  public void editRecord(String operation)
  {
    System.out.println("Performing a " + operation +
      " operation on record " + record +
      " in database " + name);
  }

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

Như các bạn có thể thấy, thật đơn giản, đối tượng singleton đã được tạo ra trước khi bất cứ luồng chương trình nào có thể nắm bắt được. Tuyệt vời.

Cách làm việc của giải pháp xử lý "tiền tiến trình"

Chương trình trên có hoạt động không? Như với giải pháp đồng bộ hóa, bạn có thể để phiên bản này làm việc bằng cách tạo đối tượng DatabaseThreaded bằng cách gọi hàm getInstance.

```
public class TestSingletonThreaded implements Runable
{
  public static void main(String args[])
  {
    TestSingletonThreaded t = new TestSingletonThreaded();
  }

  public TestSingletonThreaded()
  {
    DatabaseThreaded database;

    database = DatabaseThreaded.getInstance("products");
      .
      .
      .
}
```

Và bạn có thể sử dụng một tiến trình khác để cố gắng tạo ra một đối tượng DatabaseThreaded khác.

```
public class TestSingletonThreaded implements Runable
{
  Thread thread;

  public static void main(String args[])
  {
    TestSingletonThreaded t = new TestSingletonThreaded();
  }

  public TestSingletonThreaded()
  {
    DatabaseThreaded database;

    database = DatabaseThreaded.getInstance("products");

    thread = new Thread(this, "second");
    thread.start();

    System.out.println("This is the " +
      database.getName() + " database.");
  }

    public void run()
    {
      DatabaseThreaded database;

      database = DatabaseThreaded.getInstance("employees");

      System.out.println("This is the " +
        database.getName() + " database.");
    }
}
```

Và khi chương trình hoạt động. Bạn thấy rằng bạn đang nhận được cùng một đối tượng.

```
This is the products database.
This is the products database.
```

Đây là giải pháp tốt hơn việc đồng bộ hóa hàm getInstance. Không thể tạo nhiều hơn một đối tượng, nên sẽ tránh được sự xung đột giữa các luồng chương trình. Bạn đã gỡ bỏ mã nguồn đồng bộ hóa bằng cách đưa chúng ra hỏi hàm getInstance.

Chú ý: Nếu bạn sử dụng Java với phiên bản nhỏ hơn 1.2, có một rắc rối với trình thu dọn rác. Nếu không có một tham chiếu với đối tượng singleton, trình thu gom rác sẽ thu gom luôn đối tượng này. Lỗi này đã được sửa lại trên phiên bản 1.2

Có một vấn đề cần chú ý. Nếu bạn sử dụng bộ nạp đa lớp và sử dụng đối tượng singleton, bạn sẽ gặp lỗi. Bởi vì mỗi lớp sử dụng một không gian tên khác nhau, bạn có thể đối mặt

với việc tạo nhiều đối tượng singleton. Vì thế khi bạn sử dụng nhiều lớp, hãy chắc chắn rằng mã nguồn có kiểm tra đối chiếu giữa các lớp, để đảm bảo rằng chỉ có duy nhất một đối tượng singleton tồn tại trong một thời điểm.

Mẫu "hạng ruồi" flyweight giúp cho một đối tượng trông giống nhiều đối tượng.

Mẫu singleton nói về việc tạo một đối tượng duy nhất. Có một mẫu thiết kế khác cũng hạn chế việc tạo đối tượng, nhưng lần này nó sẽ đem đến một cách thức khác trong việc viết mã. Đó là mẫu "hạng ruồi" flyweight.

Mẫu thiết kế này gọi là "hạng ruồi" flyweight nguyên do thay vì phải làm việc với nhiều đối tượng độc lập, to lớn, bạn giảm bớt kích thước chúng bằng việc tạo một tập hợp các đối tượng dùng chung nhỏ hơn, gọi là flyweight mà bạn có thể cài đặt vào lúc thực thi chương trình để chúng trông giống như những đối tượng lớn hơn. Mỗi đối tượng to lớn có thể tiêu tốn nhiều tài nguyên hệ thống, bằng cách tách những điểm giống nhau của các đối tượng này, và dựa trên việc cấu hình thời gian thực để mô phỏng lại các đối tượng lớn, bạn đã làm giảm bớt gánh nặng lên tài nguyên hệ thống.

Bạn có thể đem những phần riêng biệt ra khỏi mã nguồn của những đối tượng to lớn và tạo ra những đối tượng flyweight. Khi làm điều này, bạn đã chấm dứt việc sử dụng nhiều đối tượng có chung các đặc điểm, và giảm xuống việc chỉ sử dụng một đối tượng, có thể cài đặt khi chương trình thực thi, mô phỏng lại cả tập hợp các đối tượng to lớn ban đầu

Ghi nhớ: Sách GoF đã định nghĩa mẫu flyweight như sau: "Sử dụng việc chia sẻ để giúp cho việc xử lý các đối tượng lớn một cách hiệu quả" Họ cũng nói rằng: "Một mẫu flyweight là một đối tượng chia sẻ mà có thể sử dụng trong đồng thời nhiều ngữ cảnh. Mẫu flyweight hoạt động như một đối tượng độc lập trong mỗi thời điểm.

Đây là những gì diễn ra. Bạn bắt đầu với một tập hợp nhiều đối tượng to lớn trong mã nguồn. Bạn gỡ bỏ những phần dùng chung, đóng gói chúng vào một đối tượng chia sẻ, một flyweight, đối tượng này hoạt động như một khuôn mẫu. Đối tượng khuôn mẫu này có thể được cài đặt vào lúc thực thi chương trình bằng cách chuyển các đặc điểm dùng chung vào đối tượng flyweight để nó xuất hiện giống như một hay nhiều đối tượng lớn ban đầu. Bạn có thể thấy như hình vẽ sau:

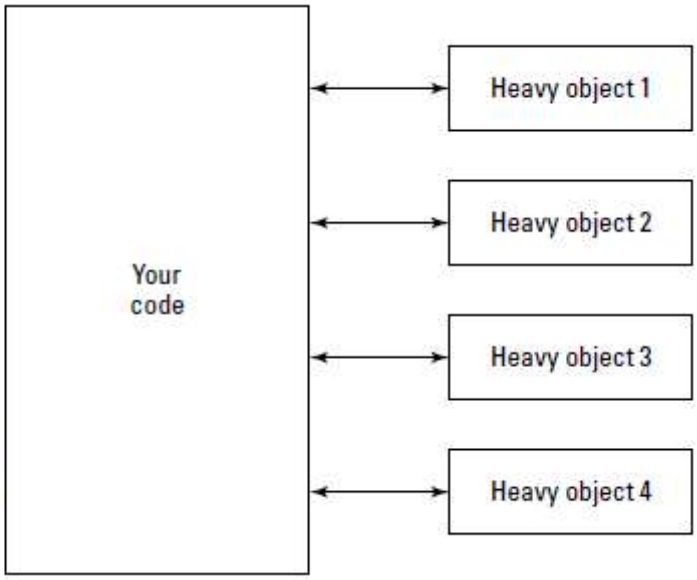

Figure 5-4: Code full of heavy, large objects.

Từ những đối tượng to lớn, bạn tạo một đối tượng nhỏ hơn gọi là flyweight (trong ví dụ này là một flyweight, tùy nhiên tùy thuộc vào ứng dụng mà bạn có thể có nhiều flyweight), mà bạn có thể cài đặt vào lúc chương trình hoạt động như hình sau:

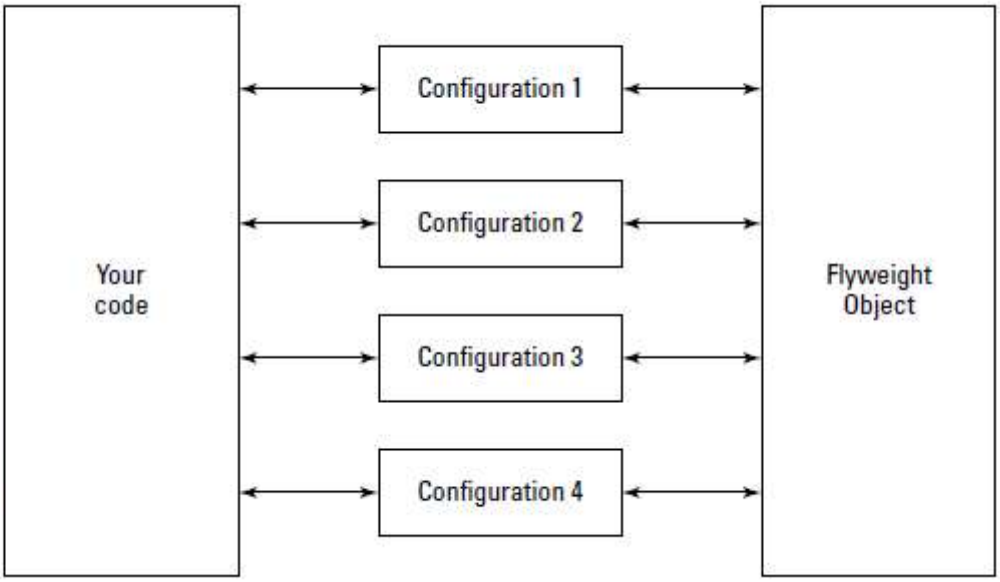

Figure 5-5: Using a flyweight object.

Bất cứ khi nào bạn phải xử lý một lượng lớn các đối tượng, mẫu Flyweight sẽ xuất hiện trong tâm trí bạn. Nếu bạn có thể tách những nội dung giống nhau cần thiết từ những đối tượng này, và tạo một flyweight, hoặc nhiều flyweight, mà hoạt động giống những khuôn mẫu, thì đó chính là cách mẫu flyweight hoạt động

Ví dụ rằng, ở cương vị một chuyên gia thiết kế mẫu, bạn được chọn để giảng dạy về mẫu thiết kế cho một lớp học. Chương trình mà bạn cần có để theo dõi hồ sơ học viên cho

từng học viên có thể là những đối tượng thật sự lớn. Bạn quyết định đã đến lúc tiết kiệm tài nguyên hệ thống. Đó là công việc của mẫu Flyweight.

Tạo một học viên

Để tạo mã nguồn cho một đối tượng học viên, bạn quyết định cài đặt nó như một đối tượng Flyweight với tên Student. Đối tượng này được cấu hình sao cho trông giống nhiều học viên mà bạn muốn. Vì vậy bạn thêm vào các hàm thiết lập thông tin và trả thông tin, chẳng hạn tên học viên, mã số, và điểm.

Bạn cùng có thể muốn so sánh học lực của các học viên với nhau, nên bạn thêm một hàm getStanding, có thể trả về mối tương quan của học lực học viên và điểm trung bình . Mã như sau:

```
public class Student
{
  String name;
  int id;
  int score;
  double averageScore;

  public Student(double a)
  {
    averageScore = a;
  }

  public void setName(String n)
  {
      name = n;
  }

  public void setId(int i)
  {
      id = i;
  }

  public void setScore(int s)
  {
      score = s;
  }

  public String getName()
  {
      return name;
  }

  public int getID()
  {
      return id;
  }

  public int getScore()
  {
      return score;
  }

  public double getStanding()
  {
      return (((double) score) / averageScore - 1.0) * 100.0;
  }
}
```

Lưu ý rằng hàm getStanding sẽ trả về sự khác biệt phần trăm điểm số của sinh viên so với điểm trung bình.

Chạy thử mẫu Flyweight

Để sử dụng mẫu flyweight, bạn phải lưu trữ dữ liệu mà bạn muốn cấu hình cho flyweight. Trong trường hợp này bạn muốn cài đặt cho đối tượng Student giống như một tập hợp các học viên có thật, vì vậy bạn có thể lưu trữ dữ liệu sinh viên (như tên, mã số, điểm) bằng một dãy như sau:

```
public class TestFlyweight
{
  public static void main(String args[])
  {
    String names[] = {"Ralph", "Alice", "Sam"};
    int ids[] = {1001, 1002, 1003};
    int scores[] = {45, 55, 65};
        .
        .
        .
```

Để so sánh học viên này với học viên khác, bạn cần xác định điểm trung bình (tổng điểm chia cho số sinh viên), mã như sau:

```
public class TestFlyweight
{
  public static void main(String args[])
  {
   String names[] = {"Ralph", "Alice", "Sam"};
   int ids[] = {1001, 1002, 1003};
   int scores[] = {45, 55, 65};

   double total = 0;
   for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
     total += scores[loopIndex];
   }

   double averageScore = total / scores.length;
       .
       .
       .
```

Trong ví dụ này, bạn cần duy nhất một đối tượng flyweight Student, bạn sẽ truyền giá trị điểm số trung bình qua hàm khởi dựng của Student như sau:

```
public class TestFlyweight
{
  public static void main(String args[])
  {
    String names[] = {"Ralph", "Alice", "Sam"};
    int ids[] = {1001, 1002, 1003};
    int scores[] = {45, 55, 65};

    double total = 0;
    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      total += scores[loopIndex];
    }

    double averageScore = total / scores.length;

    Student student = new Student(averageScore);
        .
        .
        .
```

Bây giờ bạn phải cài đặt đối tượng flyweight theo ý muốn, thay vì phải tạo từng đối tượng riêng biệt cho từng sinh viên một. Hãy xem cách thức vòng lặp sau thực hiện:

```
public class TestFlyweight
{
  public static void main(String args[])
  {
    String names[] = {"Ralph", "Alice", "Sam"};
    int ids[] = {1001, 1002, 1003};
    int scores[] = {45, 55, 65};

    double total = 0;
    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      total += scores[loopIndex];
    }

    double averageScore = total / scores.length;

    Student student = new Student(averageScore);

    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      student.setName(names[loopIndex]);
      student.setId(ids[loopIndex]);
      student.setScore(scores[loopIndex]);

      System.out.println("Name: " + student.getName());
      System.out.println("Standing: " +
        Math.round(student.getStanding()));
      System.out.println("");
    }
  }
}
```

Chạy mã trên và bạn nhận được kết quả mong muốn. Đối tượng flyweight đã được cấu hình cho từng học viên, thể hiện được tên và xếp hạng:

```
Name: Ralph
Standing: -18

Name: Alice
Standing: 0

Name: Sam
Standing: 18
```

Thay vì sử dụng ba đối tượng đầy đủ, bạn chỉ cần sử dụng một đối tượng. Cũng gần giống mẫu Singleton, tuy nhiên ý tưởng đằng sau mẫu Flyweight là kiểm soát việc tạo dựng đối tượng, và số lượng đối tượng theo ý bạn muốn.

Xử lý vấn đề đa luồng

Mẫu flyweight được sử dụng để kiểm soát việc tạo dựng đối tượng, nhưng bạn lưu ý rằng nó cũng bị chung một vấn đế với mẫu Singleton mà chúng ta đã nhắc tới. Nếu mã nguồn của bạn có sử dụng đa luồng, bạn có thể tránh việc tạo ra quá nhiều đối tượng flyweight bằng cách tách rời quá trình tạo đối tượng ra khỏi toán tử new như đã từng làm với mẫu Singleton. Bạn có thể tạo đối tượng flyweight ngay khi lớp được nạp lần đầu tiên, ngăn cản việc truy xuất hàm khởi dựng bằng cách gán cho nó một truy cập cục bộ, và cho phép việc tạo đối tượng thông qua hàm getInstance.

```
public class StudentThreaded
{
  String name;
  int id;
  int score;
  double averageScore;
  private static StudentThreaded singleObject =
    new StudentThreaded();

  private StudentThreaded()
  {
  }

  public void setAverageScore(double a)
  {
    averageScore = a;
  }

  public void setName(String n)
  {
      name = n;
  }

  public void setId(int i)
  {
      id = i;
  }

  public void setScore(int s)
  {
      score = s;
  }

  public String getName()
  {
      return name;
  }
  public int getID()
  {
      return id;
  }

  public int getScore()
  {
      return score;
  }
  public double getStanding()
  {
      return (((double) score) / averageScore - 1.0) * 100.0;
  }

  public static StudentThreaded getInstance()
  {
      return singleObject;
  }

}
```

Ví dụ sau, cho thấy khi làm việc với phiên bản mới, việc xử lý đa luồng đã được giải quyết

```
public class TestFlyweightThreaded implements Runable
{
  Thread thread;

  public static void main(String args[])
  {
    TestFlyweightThreaded t = new TestFlyweightThreaded();
  }

  public TestFlyweightThreaded()
  {
    String names[] = {"Ralph", "Alice", "Sam"};
    int ids[] = {1001, 1002, 1003};
    int scores[] = {45, 55, 65};

    double total = 0;
    for (int loopIndex = 0; loopIndex < scores.length; loopIndex++){
      total += scores[loopIndex];
    }

    double averageScore = total / scores.length;

    StudentThreaded student = StudentThreaded.getInstance();

    student.setAverageScore(averageScore);
    student.setName("Ralph");
    student.setId(1002);
    student.setScore(45);

    thread = new Thread(this, "second");
    thread.start();

    System.out.println("Name: " + student.getName() +
      ", Standing: " + Math.round(student.getStanding()));
  }

  public void run()
  {
    StudentThreaded student = StudentThreaded.getInstance();

    System.out.println("Name: " + student.getName() +
      ", Standing: " + Math.round(student.getStanding()));
  }
}
```

Chạy đoạn mã trên, kết quả như sau, bạn sẽ nhận được cùng một đối tượng khi chạy cùng lúc hai luồng:

```
Name: Ralph, Standing: -18
Name: Ralph, Standing: -18
```

Chú ý: Vậy sử dụng mẫu flyweight có hạn chế nào không? Vấn đề chính là bạn sẽ mất thêm thời gian để cài đặt một đối tượng flyweight và nếu bạn phải cài đặt bao bọc mọi thứ, bạn có thể sẽ làm giảm hiệu năng hệ thống nhiều hơn mong đợi. Một hạn chế nữa là: bởi vì bạn tách một lớp mẫu chung ra khỏi đối tượng để tạo flyweight, bạn phải thêm vào

một lớp khác trong việc lập trình, và có thể gây ra sự khó khăn trong việc bảo trì và mở rộng.

Download source code Csharp tại đây

Design Patterns For Dummies

Design Patterns

< DP4Dummies – Chương 4: Observer, Chain of Responsibility

> DP4Dummies – Chương 6: Adapter, Facade