Motivation

Sam Berlin edited this page on Apr 20, 2015 · 4 revisions

Motivation

Wiring everything together is a tedious part of application development. There are several approaches to connect data, service, and presentation classes to one another. To contrast these approaches, we'll write the billing code for a pizza ordering website:

```
public interface BillingService {

  /**
   * Attempts to charge the order to the credit card. Both successful and
   * failed transactions will be recorded.
   *
   * @return a receipt of the transaction. If the charge was successful, the
   *      receipt will be successful. Otherwise, the receipt will contain a
   *      decline note describing why the charge failed.
   */
  Receipt chargeOrder(PizzaOrder order, CreditCard creditCard);
}
```

Along with the implementation, we'll write unit tests for our code. In the tests we need a `FakeCreditCardProcessor` to avoid charging a real credit card!

Direct constructor calls

Here's what the code looks like when we just `new` up the credit card processor and transaction logger:

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    CreditCardProcessor processor = new PaypalCreditCardProcessor();
    TransactionLog transactionLog = new DatabaseTransactionLog();

    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
     } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

This code poses problems for modularity and testability. The direct, compile-time dependency on the real credit card processor means that testing the code will charge a credit card! It's also

awkward to test what happens when the charge is declined or when the service is unavailable.

Factories

A factory class decouples the client and implementing class. A simple factory uses static methods to get and set mock implementations for interfaces. A factory is implemented with some boilerplate code:

```
public class CreditCardProcessorFactory {

  private static CreditCardProcessor instance;

  public static void setInstance(CreditCardProcessor processor) {
    instance = processor;
  }

  public static CreditCardProcessor getInstance() {
    if (instance == null) {
      return new SquareCreditCardProcessor();
    }

    return instance;
  }
}
```

In our client code, we just replace the `new` calls with factory lookups:

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    CreditCardProcessor processor = CreditCardProcessorFactory.getInstance();
    TransactionLog transactionLog = TransactionLogFactory.getInstance();

    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
     } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

The factory makes it possible to write a proper unit test:

```
public class RealBillingServiceTest extends TestCase {

  private final PizzaOrder order = new PizzaOrder(100);
  private final CreditCard creditCard = new CreditCard("1234", 11, 2010);

  private final InMemoryTransactionLog transactionLog = new InMemoryTransactionLog()
  private final FakeCreditCardProcessor processor = new FakeCreditCardProcessor();

  @Override public void setUp() {
    TransactionLogFactory.setInstance(transactionLog);
    CreditCardProcessorFactory.setInstance(processor);
  }

  @Override public void tearDown() {
    TransactionLogFactory.setInstance(null);
    CreditCardProcessorFactory.setInstance(null);
```

```
    }

    public void testSuccessfulCharge() {
      RealBillingService billingService = new RealBillingService();
      Receipt receipt = billingService.chargeOrder(order, creditCard);

      assertTrue(receipt.hasSuccessfulCharge());
      assertEquals(100, receipt.getAmountOfCharge());
      assertEquals(creditCard, processor.getCardOfOnlyCharge());
      assertEquals(100, processor.getAmountOfOnlyCharge());
      assertTrue(transactionLog.wasSuccessLogged());
    }
  }
```

This code is clumsy. A global variable holds the mock implementation, so we need to be careful about setting it up and tearing it down. Should the `tearDown` fail, the global variable continues to point at our test instance. This could cause problems for other tests. It also prevents us from running multiple tests in parallel.

But the biggest problem is that the dependencies are *hidden in the code*. If we add a dependency on a `CreditCardFraudTracker`, we have to re-run the tests to find out which ones will break. Should we forget to initialize a factory for a production service, we don't find out until a charge is attempted. As the application grows, babysitting factories becomes a growing drain on productivity.

Quality problems will be caught by QA or acceptance tests. That may be sufficient, but we can certainly do better.

Dependency Injection

Like the factory, dependency injection is just a design pattern. The core principle is to *separate behaviour from dependency resolution*. In our example, the `RealBillingService` is not responsible for looking up the `TransactionLog` and `CreditCardProcessor`. Instead, they're passed in as constructor parameters:

```
  public class RealBillingService implements BillingService {
    private final CreditCardProcessor processor;
    private final TransactionLog transactionLog;

    public RealBillingService(CreditCardProcessor processor,
        TransactionLog transactionLog) {
      this.processor = processor;
      this.transactionLog = transactionLog;
    }

    public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
      try {
        ChargeResult result = processor.charge(creditCard, order.getAmount());
        transactionLog.logChargeResult(result);

        return result.wasSuccessful()
            ? Receipt.forSuccessfulCharge(order.getAmount())
            : Receipt.forDeclinedCharge(result.getDeclineMessage());
       } catch (UnreachableException e) {
        transactionLog.logConnectException(e);
        return Receipt.forSystemFailure(e.getMessage());
      }
    }
  }
```

We don't need any factories, and we can simplify the testcase by removing the `setUp` and `tearDown` boilerplate:

```
public class RealBillingServiceTest extends TestCase {

  private final PizzaOrder order = new PizzaOrder(100);
  private final CreditCard creditCard = new CreditCard("1234", 11, 2010);

  private final InMemoryTransactionLog transactionLog = new InMemoryTransactionLog()
  private final FakeCreditCardProcessor processor = new FakeCreditCardProcessor();

  public void testSuccessfulCharge() {
    RealBillingService billingService
        = new RealBillingService(processor, transactionLog);
    Receipt receipt = billingService.chargeOrder(order, creditCard);

    assertTrue(receipt.hasSuccessfulCharge());
    assertEquals(100, receipt.getAmountOfCharge());
    assertEquals(creditCard, processor.getCardOfOnlyCharge());
    assertEquals(100, processor.getAmountOfOnlyCharge());
    assertTrue(transactionLog.wasSuccessLogged());
  }
}
```

Now, whenever we add or remove dependencies, the compiler will remind us what tests need to be fixed. The dependency is *exposed in the API signature*.

Unfortunately, now the clients of `BillingService` need to lookup its dependencies. We can fix some of these by applying the pattern again! Classes that depend on it can accept a `BillingService` in their constructor. For top-level classes, it's useful to have a framework. Otherwise you'll need to construct dependencies recursively when you need to use a service:

```
  public static void main(String[] args) {
    CreditCardProcessor processor = new PaypalCreditCardProcessor();
    TransactionLog transactionLog = new DatabaseTransactionLog();
    BillingService billingService
        = new RealBillingService(processor, transactionLog);
    ...
  }
```

Dependency Injection with Guice

The dependency injection pattern leads to code that's modular and testable, and Guice makes it easy to write. To use Guice in our billing example, we first need to tell it how to map our interfaces to their implementations. This configuration is done in a Guice module, which is any Java class that implements the `Module` interface:

```
public class BillingModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(TransactionLog.class).to(DatabaseTransactionLog.class);
    bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
    bind(BillingService.class).to(RealBillingService.class);
  }
}
```

We add `@Inject` to `RealBillingService` 's constructor, which directs Guice to use it. Guice will inspect the annotated constructor, and lookup values for each parameter.

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;
```

```
    @Inject
    public RealBillingService(CreditCardProcessor processor,
        TransactionLog transactionLog) {
      this.processor = processor;
      this.transactionLog = transactionLog;
    }

    public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
      try {
        ChargeResult result = processor.charge(creditCard, order.getAmount());
        transactionLog.logChargeResult(result);

        return result.wasSuccessful()
            ? Receipt.forSuccessfulCharge(order.getAmount())
            : Receipt.forDeclinedCharge(result.getDeclineMessage());
       } catch (UnreachableException e) {
        transactionLog.logConnectException(e);
        return Receipt.forSystemFailure(e.getMessage());
      }
    }
  }
```

Finally, we can put it all together. The `Injector` can be used to get an instance of any of the bound classes.

```
    public static void main(String[] args) {
      Injector injector = Guice.createInjector(new BillingModule());
      BillingService billingService = injector.getInstance(BillingService.class);
      ...
    }
```

Getting started explains how this all works.