Material and some slide content from:
- Krzysztof Czarnecki
- Ian Sommerville
- Head First Design Patterns

Dependency Injection & Design Principles Recap

Reid Holmes

SOLI**D** (Dependency Inversion)

Program to **interfaces** not to **implementations**.

Dependency Inversion

(also called inversion of control)

- Common problem: ‘how can we wire these interfaces together without creating a dependency on their concrete implementations?’
  - This often challenges the ‘program to interfaces, not implementations ’ design principle
    - Would like to reduce (eliminate) coupling between concrete classes
  - Would like to be able to substitute different implementations without recompiling
    - e.g., be able to test and deploy the same binary even though some objects may vary
  - Solution: separate objects from their assemblers

Example Overview

Simple Pizza BillingService API

```
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);
}
```

[Example from: https://code.google.com/p/google-guice/wiki/Motivation]

Example Overview

Charging orders requires a CCProcessor and a TransactionLog

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

    CreditCardProcessor processor = new PaypalCreditCardProcessor();
    TransactionLog transactionLog = new DatabaseTransactionLog();

    …
    }
  }
}
```

BillingService is dependent on the concrete implementations of the processor/log classes rather than their interfaces

Example Overview

Can't test without actually processing the CC data

```
public class RealBillingServiceTest extends TestCase {

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


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

    assertTrue(…);
  }
}
```

Could test with invalid data, but that would not test the success case.

Factory Fix

```
public class CreditCardProcessorFactory {

  private static CreditCardProcessor instance;

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

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

    return instance;
  }
}
```

Factories provide one way to encapsulate object instantiation

Factory Fix

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

    CreditCardProcessor processor = CreditCardProcessorFactory.getInstance();
    TransactionLog transactionLog = TransactionLogFactory.getInstance();

    …
  }
}
```

Instead of depending on the concrete classes, BillingService relies on the factory to instantiate them.

Factory Fix

This enables mock implementations to be returned for testing.

```
public class RealBillingServiceTest extends TestCase {

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

  private final MemoryTransactionLog transactionLog = new MemoryTransactionLog();
  private final FakeCCPro creditCardProcessor = new FakeCCPro();

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

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

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

    assertTrue(…);
  }
}
```

Factories work, but from the BillingService APIs alone, it is impossible to see the CC/Log dependencies.

DI Goal

- Eliminate initialization statements. e.g.,
  - Foo f = new ConcreteFoo();
- In dependency injection a third party (an injector)
- At a high level dependency injection:
  - **Takes** a set of components (classes + interfaces)
  - **Adds** a set of configuration metadata
  - **Provides** the metadata to an injection framework
  - **Bootstraps** object creation with a configured injector

Dependency Injection

```
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) {
  …
 }
}
```

We can hoist the dependencies into the API to make them transparent.

Dependency Injection

```
public class RealBillingServiceTest extends TestCase {

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

  private final MemoryTransactionLog transactionLog = new MemoryTransactionLog();
  private final FakeCCProcessor creditCardProcessor = new FakeCCProcessor();

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

    assertTrue(...);
  }
}
```

This also enables unit test mocking, but as in the initial example, pushes the object instantiations throughout the code.

Guice Injection

Google Guice is a common IoC framework for alleviating some of the boiler plate code associated with this pattern.

```
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);
  }
}
```

Here, the types of classes to their concrete implementations. Guice automatically instantiates the objects as required.

Guice Injection

Deployment Module:

```
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);
  }
}
```

Testing Module:

```
public class MockBillingModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(TransactionLog.class).to(MockTransactionLog.class);
    bind(CreditCardProcessor.class).to(MockCreditCardProcessor.class);
    bind(BillingService.class).to(RealBillingService.class);
  }
}
```

Guice Injection

```
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) {
     …
 }
}
```

@Inject tells Guice to automatically instantiate the correct CC/Log objects. The module will determine what gets injected.

Guice Injection

Guice modules need to be configured with the configuration of the system they are injecting for.

Deployment:

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

Test:

```
public class RealBillingServiceTest extends TestCase {

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

  @BeforeClass
  public final void guiceSetup() {
      Guice.createInjector( new MockBillingModule()).injectMembers(this);
  }

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

    assertTrue(...);
  }
}
```

SOLID (Open/Close)

Classes should be open to **extension** and closed to **modification**.

SOLID (Open/Close)

Which **design** (that we have covered in class) **patterns** support the open/close principle?

(These patterns are a subset of those patterns that help with *encapsulating what varies*. E.g., the ‘extension’ part is often expected to change.)

SOLID (Open/Close)

- Observer (extend set of observers)
  - w/o changing subject behaviour
- Strategy (extend algorithm suite)
  - w/o changing context or other algorithms
- Command (extend command suite)
  - w/o changing invoker
- Visitor (extend model analysis)
  - w/o changing data structure, traversal code, other visitors
- Composite (extend component)
  - w/o changing clients / composites using any component

SOLID (Single Responsibility)

Classes should do one thing and do it well.

SOLID (Single Responsibility)

- Strategy (small, targeted, algorithms)
- Command (invokers should be oblivious to actions)
- Visitor (usually accomplish specific tasks)
- Facade (centralize 3rd party complexity)