Class Coupling

When one class depends on another

- **Coupling** (**dependency**) between classes creates some challenges
  - **Maintenance Issues**
    - Changes to one class requires changes to be made in the other class
  - **Testing Issues**
    - It is difficult to test the two classes in insolation

Coupling in Applications

```
public class OrderProcessor {
        private AccountingServiceIntlRules  acctService;

        public OrderProcessor() {
                acctSrvc = new AccountingServiceIntlRules ();
        }

        public void recordNewOrder(Order order) {
                acctService.recordNewOrder(order);
                inventoryService.adjustInventory(order);
        }
}
```

```
public class ExpenseHandler {
        private AccountingServiceIntlRules  acctSrvc;

        public ExpenseHandler() {
                acctSrvc = new AccountingServiceIntlRules ();
        }

        public void recordNewExpense(Category cat, double amt) {
                acctSrvc.recordExpense(cat, amt);
                departmentBudget.deductExpense(amt);
        }
}
```

Maintainability: Our application is heavily dependent on the International accounting rules.

- How much work is involved to convert to U.S. Accounting Rules (a different Accounting Service implementation)?

Testing Complications

```
public class OrderProcessor {
        private AccountingServiceIntlRules  acctService;

        public OrderProcessor() {
                acctService = new AccountingServiceIntlRules ();
        }

        public void recordNewOrder(Order order) {
                acctService.recordNewOrder(order);
                inventoryService.adjustInventory(order);
        }
}
```

For Unit Testing purposes we may want to "Mock" the AccountingService

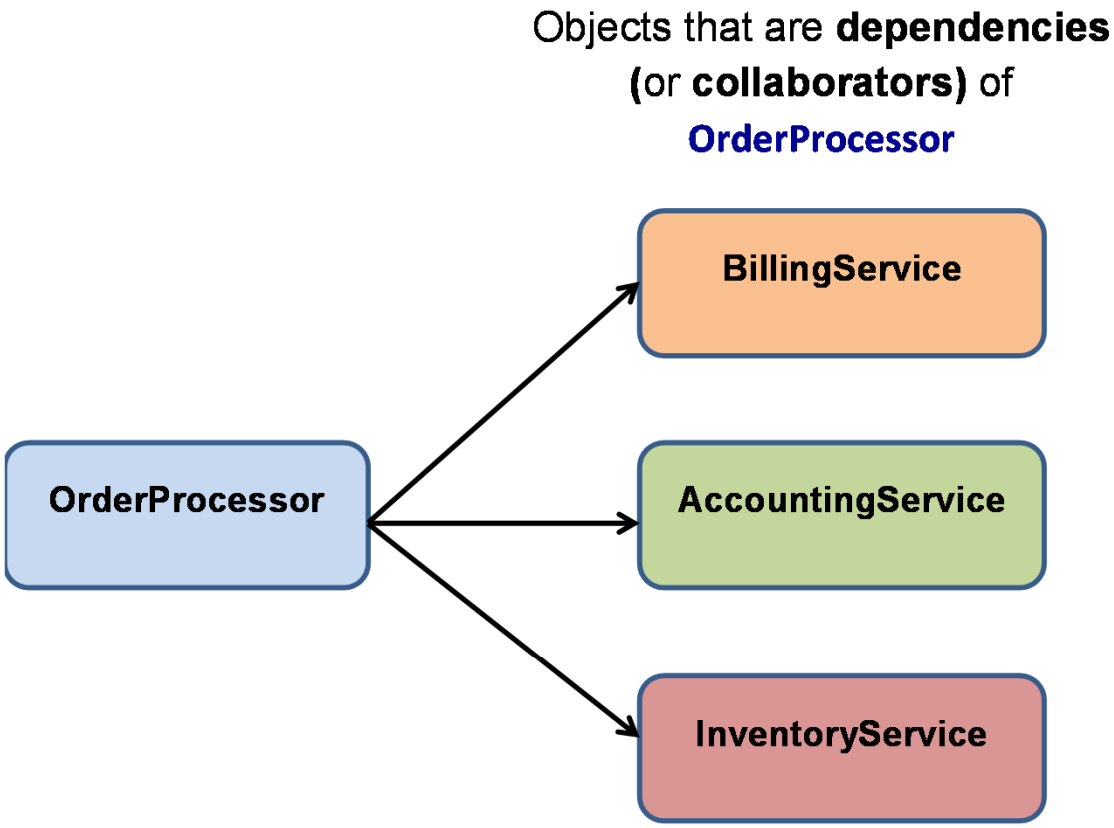
- We want to test if the OrderProcessor is doing the right thing, not the Accounting Service.
- What if the Accounting Service object changes our corporate database (testing that has bad side effects!)?

A Mocked Class

```
public class AccountingServiceMockImpl {
     public void recordNewOrder(Order order) {
           System.out.println("Accounting system called for: " + order);
       }
}
```

Dependencies

- Enterprise Applications can have many of these dependencies



Ways of Reducing Coupling

- **Interfaces**
- **Factory Pattern**
- **Service Locator Pattern**
- **Dependency Injection**

Using Interfaces

```
public interface AccountingService {
        public void recordNewOrder(Order order);
}
```

```
public class OrderProcessor {
        private AccountingService  acctService;

        public OrderProcessor(AccountingService  acctSrvc) {
                this.acctService = acctSrvc;
        }

        public void recordNewOrder(Order order) {
                acctService.recordNewOrder(order);
                inventoryService.adjustInventory(order);
        }
}
```

```
// Order Processor Client
AccountingService acctSrvc = new AccountingServiceIntlRules();
OrderProcessor orderProc = new OrderProcessor(acctSrvc);
```

Using an Interface (AccountingService) reduces the coupling

- The OrderProcessor class is no longer dependent on any specific AccountingService implementation
- Still may have many uses of **new** throughout the code instantiating a specific implementation

Inversion of Control

- Normal Flow of Program
  - An object requiring collaborators **instantiates** them (with new)
  - The object manages the collaborator lifecycles (initializing them, destroying them, etc)

- Inverted Control
  - An object requiring collaborators is **given** them – it no longer knows about their specific implementation
  - No use of new or lifecycle management of the collaborators

- **We are decoupling by inverting control**
  - **OrderProcessor** no longer instantiates its dependencies
  - The client does the new operation and gives the AccountingService to OrderProcessor

Factory Pattern

- One class can instantiate different implementations of an interface
  - Use of **new** is consolidated in one class for all implementations of that interface

```
public class OrderProcessor {
        private AccountingService  acctSrvc;

        public OrderProcessor(AccountingService  acctSrvc) {
                this.acctSrvc = acctSrvc;
        }

        public void recordNewOrder(Order order) {
                acctSrvc.recordNewOrder(order);
                inventoryManager.adjustInventory(order);
        }
}
```

```
public class AccountingServiceFactory {
   static public AccountingService getAcctService(String type, boolean mock ) {
        if (type.equals("Intl")) {
             if (mock) {
                  return new AccountingServiceIntlMock();
             } else {
                  return new AccountingServiceIntlRules();
             }
        } else if (type.equals("US")) {
                  ...
    }
}
```

```
// Order Processor Client – no longer knows about a specific implementation
boolean doingTesting = false;
AccountingService acctSrvc;
acctSrvc = AccountingServiceFactory.getAcctService("Intl", doingTesting);
OrderProcessor orderProc = new OrderProcessor(acctSrvc);
```

Service Locator Pattern

Commonly used in the traditional JEE (especially the **JNDI** API – **J**ava **N**aming and **D**irectory **I**nterface)

- A registry maps names to specific objects
- Related objects to be looked up are stored in the same registry

```
// Client Code
try {
    acctService = (AccountingService) ServiceLocator.lookup("IntlAccounting");
    OrderProcessor  orderProc = new OrderProcessor(acctService);
}  catch (NamingException) {
     System.out.println("Unknown Service: IntlAccounting");
}
```

- Somewhere else the AccountingServiceIntlRules object is created and associated with the name IntlAccounting

```
// Perhaps a special function that initializes and gives names to all
// objects that are to be stored in the registry
public void configure() {
    AccountingService acctSrvc = new AccountingServiceIntlRules();
    ServiceLocator.load("IntlAccounting", acctSrvc);
    ...  // add other objects to the registry
}
```

Improvements?

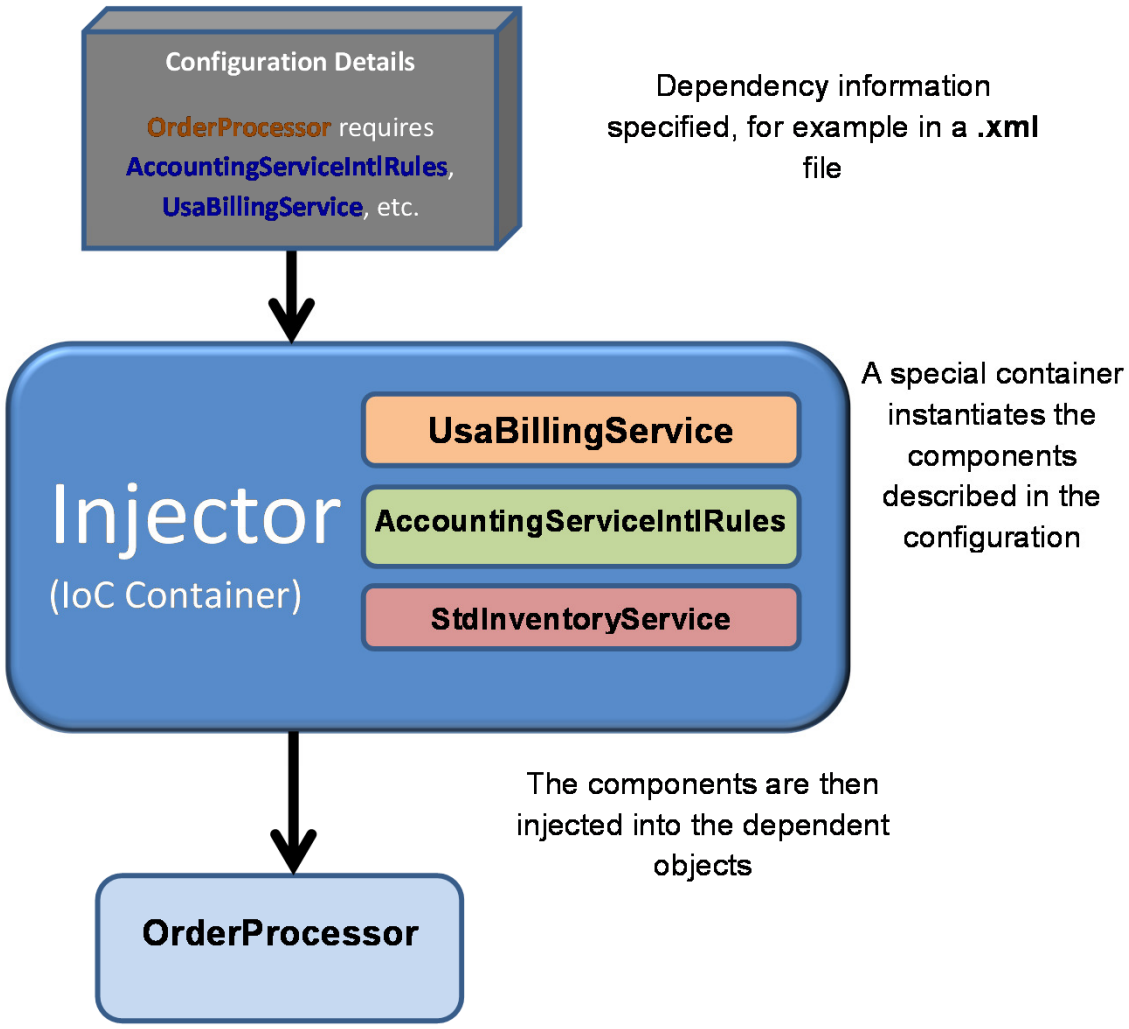
- The design patterns we've seen help to decouple

- Issue – The client still includes code for configuring the dependencies

  - **Factory** pattern still requires use of new and knowledge of specific implementations in the Factory
  - **Service Locator** requires client to know the name of the object it needs
  - Both require extra code tying the client to the specific design pattern

Dependency Injection

- One way delivery of collaborator objects
  - Give the object what it needs instead of making the object get it by itself



IoC Container

- Inversion of Control
  - Instead of an object creating a collaborator and managing its lifecycle, the object is given the collaborator
  - Dependency management is “externalized”

- IoC Container
  - Software responsible for creating the collaborator objects and managing their lifecycle
  - Can provide the collaborator objects through either lookup or injection

Traditional Dependency Management

- Traditional Lookup
  - Object requests its dependencies using new. Collaborator object is returned.
  - Object must then manage the collaborator through its lifecycle
    - Initialize the object appropriately
    - Shutdown the object (e.g. free resources) when object's services are no longer needed

IoC Dependency Management

- Dependency Lookup
  - Instead of using new, the object uses some form of lookup (factory, service locator)
  - Creation (new) and management (initialization, destruction) is no longer a responsibility of the dependent object
- Dependency Injection
  - Collaborators are **delivered** by the IoC container without any requests or lookup
  - IoC Container handles creation and lifecycle management of the collaborator objects
- Using an IoC Container
  - IoC Container is a separate piece of software that handles the creation and lifecycle management of the collaborator objects
  - Uses lookup, injection or both styles to provide the collaborator objects

Preparing Objects for DI

- **OrderProcessor** has 3 dependencies that need to be obtained.
- We write **OrderProcessor** as a regular POJO class
- The IoC container can call **OrderProcessor's** set methods to inject the dependency objects

```
public class OrderProcessor {
        private AccountingService  acctService;
        private BillingService  billService;
        private InventoryService  invService;

        public void setAcctService(AccountingService acctService) {
                this.acctService = acctService;
        }

        public void setBillService(BillingService billService) {
                this.billService = billService;
        }

        public void setInvService(InventoryService invService) {
                this.invService = invService;
        }
}
```

Specifying Dependencies in XML

- A **bean** is an object that will be instantiated
  - Each bean has a name and the class used
- Bean **properties** denote dependencies that will be injected (using the bean's set methods)

```
<!-- Spring XML configuration file -->
<beans>
   <bean id="acctServiceIntlRules" class="AccountingServiceIntlRules" />
   <bean id="billingService" class="UsaBillingService" />
   <bean id="inventoryService" class="StdInventoryService" />
   <bean id="orderProcessor" class="OrderProcessor">
        <property name="acctService" ref="acctServiceIntlRules" />
        <property name="billService" ref="billingService" />
        <property name="invService" ref="inventoryService" />
   </bean>
</beans>
```

You write:

We would have **interfaces** for:

- AccountingService
- BillingService
- InventoryService

We would have **implementations** for:

- AccountingServiceIntlRules
- UsaBillingService
- StdInventoryService

You get:

The IoC container will **instantiate** the following beans:

- acctServiceIntlRules
- billingService
- inventoryService
- orderProcessor

The following implementations are **injected** into the orderProcessor bean:

- AccountingServiceIntlRules
- UsaBillingService
- StdInventoryService

Dependency Injection from .xml

- The IoC container instantiates all 4 beans
- As requested by the .xml bean properties, the 3 dependencies are injected into the orderProcessor bean by calling its **set** methods

```
public class OrderProcessor {
    private AccountingService  acctService;
    private BillingService  billService;
    private InventoryService  invService;

    public void setAcctService(AccountingService acctService) {
        this.acctService = acctService;
    }

    public void setBillService(BillingService billService) {
        this.billService = billService;
    }

    public void setInvService(InventoryService invService) {
        this.invService = invService;
    }
}
```

AccountingServiceIntlRules

UsaBillingService

StdInventoryService

Ways of Injecting Dependencies

- Set Methods

```
public class OrderProcessor {
        private AccountingService  acctService;
        private BillingService  billService;

        public void setAcctService(AccountingService acctService) {
                this.acctService = acctService;
        }

        public void setBillService(BillingService billService) {
                this.billService = billService;
        }
}
```

- Constructors

```
public class OrderProcessor {
        private AccountingService  acctService;
        private BillingService  billService;

        public OrderProcessor(AccountingService acctService,
                              BillingService billService)
        {
                this.acctService = acctService;
                this.billService = billService;
        }
}
```

What is a Container?

- Software that is responsible for maintaining the lifecycle of various objects
  - Example: Servlet Container
- Provides services to those objects
- What services are provided and how they are accessed is specified by an API
  - API -- a contract between the container and the objects it supports

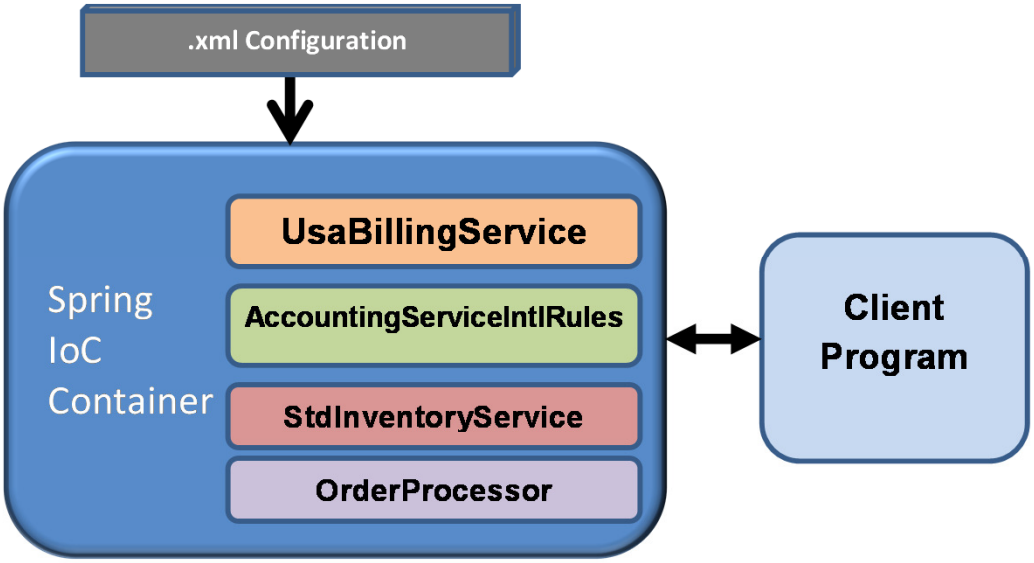
IoC Containers

A variety of IoC Containers are available

- Spring
  - The core of the Spring Framework is an IoC container
- Pico Container
  - picocontainer.com
- Google Guice
  - code.google.com/p/google-guice

Instantiating an IoC Container

- To make all this work, the final step is to instantiate the Spring IoC container
  - It will read the dependency configuration, instantiate, and configure the objects




```
public class OrderApplication {   // Our client program
    public static void main(String args[]) {
        // Instantiate the IoC Container – provide the XML configuration file
        ApplicationContext container =
                new ClassPathXmlApplicationContext("application.xml");
        // Get bean that has already been instantiated and configured by the container
        OrderProcessor orderProc =
                (OrderProcessor) container.getBean("orderProcessor");
        Order order;

        order = new Order("GSX-56789");
        orderProc.newOrder(order);   /*  Use our bean  */
    }
}
```

IoC / Depenency Injection

- Terminology
  - Inversion of Control and Dependency Injection are often used interchangeably
  - Technically speaking, Inversion of Control is a more general term.
    - It implies objects won't have the responsibility of instantiating and maintaining their collaborators (the collaborator objects will be provided).
  - Dependency Injection is a technique to achieve Inversion of Control.
- History
  - Dependency injection is a term coined by Martin Fowler to more accurately describe how the inversion of control is done
  - http://martinfowler.com/articles/injection.html

Benefits of Dependency Injection

- Loose coupling
  - Objects are not expected to create or obtain their dependencies
  - Dependencies are injected into the objects that need them
- Cleaner code
  - All the new statements / creation of dependent objects is removed
  - Centralized configuration – all dependencies could be configured in one place
- Enforces good programming practices
  - Programming to interfaces
  - Loose coupling
- Easier to Unit Test
  - Easy to replace production objects with mock objects with no changes to code or recompilation necessary

Benefits of using Spring

- Lightweight Container
  - Most objects do not contain Spring code – they are simply POJOs
  - Nothing special required to configure POJO objects
- Non-Intrusive
  - One could substitute a different IOC container without any code changes
  - A goal for all of Spring
    - use of Spring should not favor any technology (including Spring)
    - it should not force any particular technology to be used
    - easy to swap in/out different technologies
- Automated Configuration and Wiring of Application Components
  - Collaborator objects are created by Spring, connected to each other (wired) through dependency injection, and managed over their entire lifecycle
- Allows Unit Testing
  - Write code for each object and test it separately (even outside of Spring!)