Методики «Inversion of Control» и «Dependency Injection». Применение в Spring.

Малышкин Фёдор (fedor.malyshkin@magnetosoft.ru)

27 июня 2008

Основы. IoC.

Magnetosoft

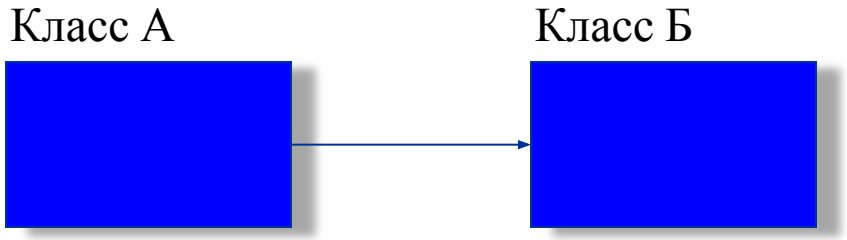
- Концепция, лежащая в основе инверсии управления, часто выражается "голливудским принципом": **"Не звоните мне, я вам сам позвоню"**. IoC переносит ответственность за выполнение действий с кода приложения на фреймворк.
- В отношении конфигурирования это означает, что если в традиционных контейнерных архитектурах наподобие EJB, компонент может вызвать контейнер и спросить: "где объект X, нужный мне для работы?", то в IoC сам контейнер выясняет, что компоненту нужен объект X, и предоставляет его компоненту во время исполнения.

Основы. DI

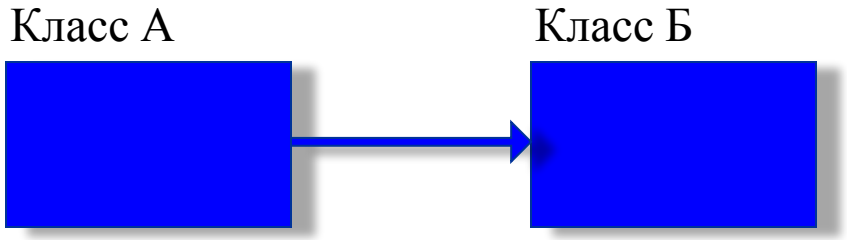
Magnetosoft

- В самом названии закладывается смысл – зависимости не создаются вашим кодом: они внедряются контейнером.
- Контейнер, среди прочего, может подставлять свои реализации, удовлетворяющие описанным интерфейсам. Это могут быть альтернативные реализации, прокси реальных объектов или иначе сконфигурированные старые реализации.
- Со стороны потребителя необходимо лишь приготовить пространство для зависимости (property).

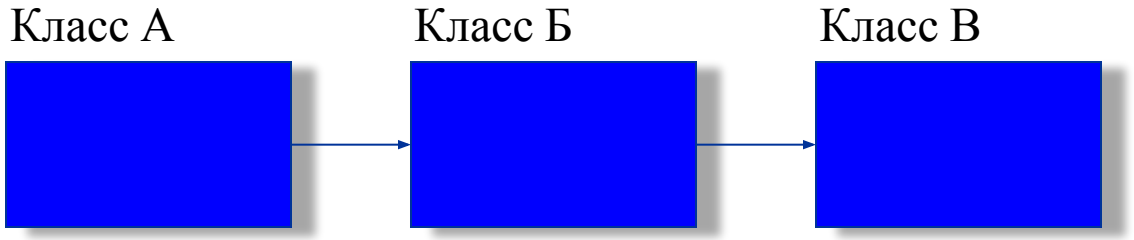
Зависимости.



Поля, параметры



Наследование, реализация



Транзитивные зависимости

Пример 1. Login Manager.

Magnetosoft

```
public class LoginManager
{
  private UserList myUserList = new UserList();
  .....
  public boolean authenticateUser(String theUserName, String
    thePassword)
  {
    User aUser = myUserList.getUserByName(theUserName);
    return thePassword.equals(aUser.getPassword());
  }
  ....
}
```

Пример 1. Недостатки.

Magnetosoft

- Если захочется каким-то образом изменить способ хранения пользователей, например, использовать базу данных или LDAP, придется переписать LoginManager, чтобы он создавал соответствующий класс для работы со списком пользователей.
- Если предположить, что класс UserList зависим от платформы, например, использует JNI или пользуется каким-либо API, специфичным для какой-то платформы, то LoginManager будет также платформенно-зависимым.
- Таким образом, из-за зависимости компонентов страдает **переносимость** и возможность их **повторного использования**.

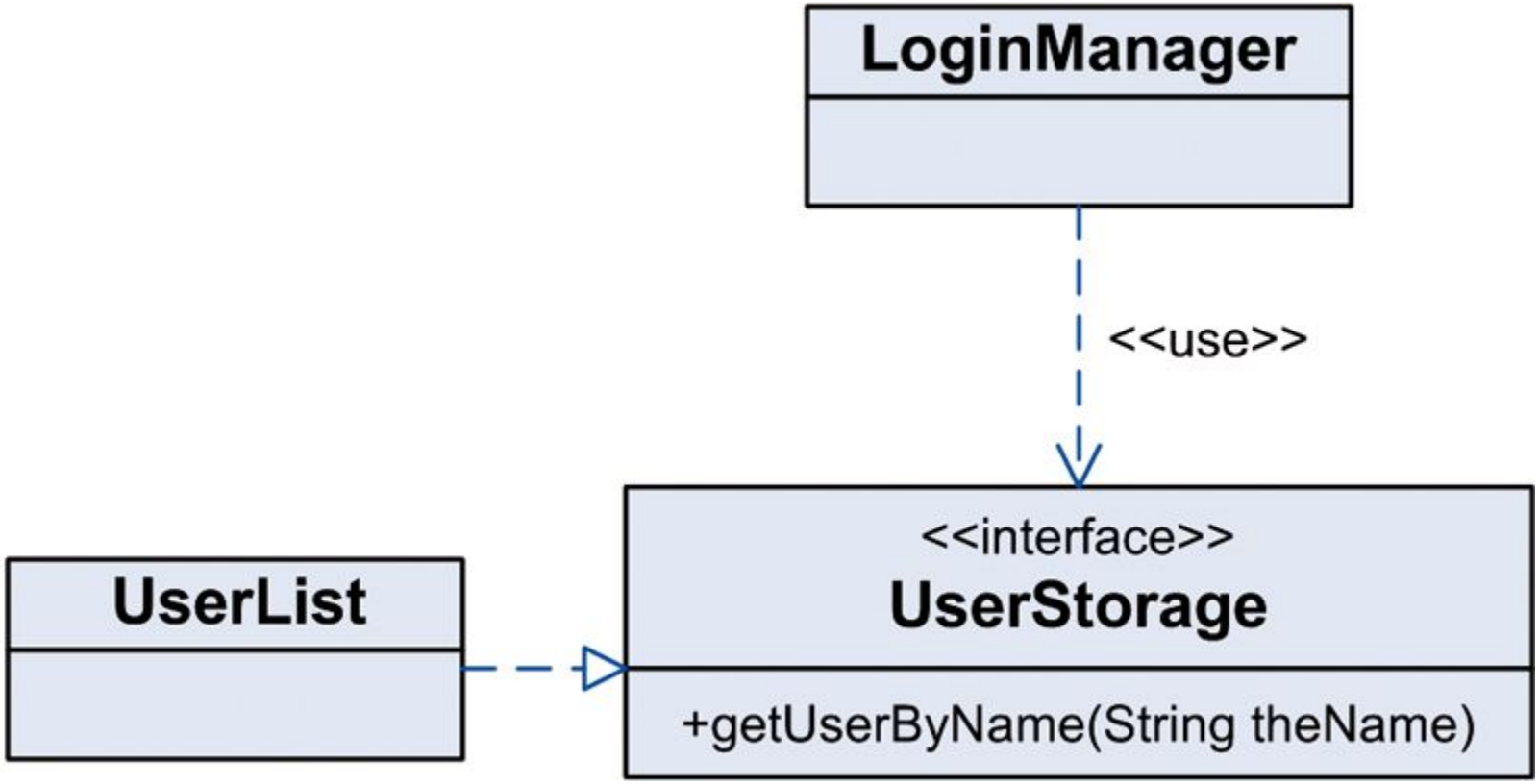
Пример 1. Улучшения.

Magnetosoft

```
public interface UserStorage {
  User getUserByName(String theUserName);
}
public class UserList implements UserStorage { .... }
public class LoginManager
{
  private UserStorage myUserList = new UserList();
  .....
  public boolean authenticateUser(String theUserName, String
    thePassword)
  {
    User aUser = myUserList.getUserByName(theUserName);
    return thePassword.equals(aUser.getPassword());
  }
  ...
```

Пример 1. UML. Начальная диаграмма классов.

Пример 1. UML. Диаграмма классов с вынесением зависимости.



Пример 1. Итоги.

Magnetosoft

- Итак, мы имеем прекрасные переносимые компоненты LoginManager, UserList, JdbcUserStorage, LdapUserStorage. Не стоит думать, что мы избавились от необходимости соединять их вместе.

Сборка.

Magnetosoft

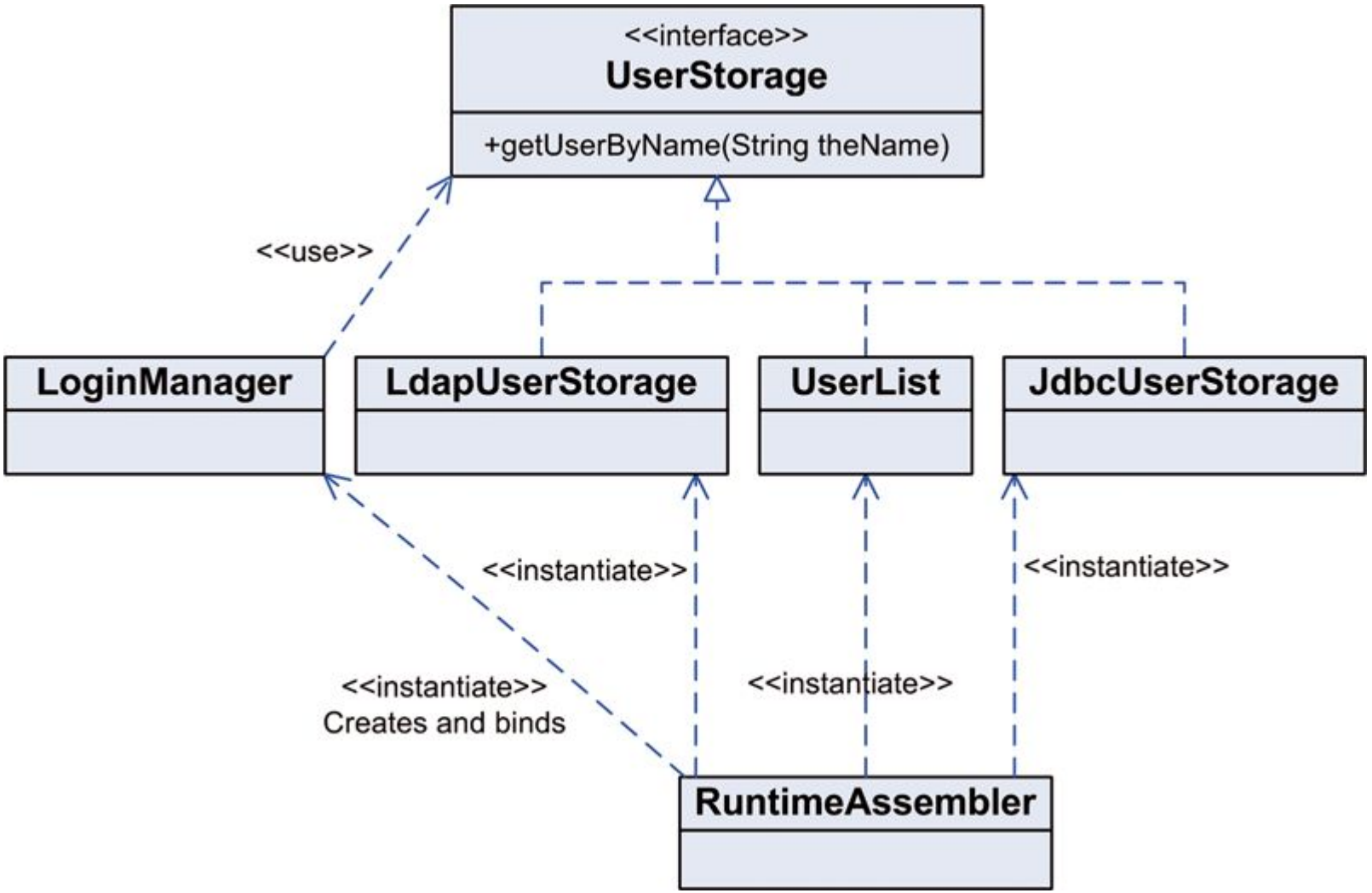
- Для использования созданных нами компонентов необходим некий класс RuntimeAssembler, который будет делать грязную работу по соединению компонентов в единую систему.

Пример 2. Сборщик (Composer, Assembler).

Magnetosoft

```
public final class SimpleSystemAssembler
{
  public void main(String[] args)
  {
    LoginManager aManager = new LoginManager();
    aManager.setUserStorage(new UserList());
    aManager.authenticateUser("user", "test");
  }
}
public final class ComplexSystemAssembler
{
  public void main(String[] args)
  {
    LoginManager aManager = new LoginManager();
    aManager.setUserStorage(
      new JdbcUserStorage("jdbc:mysql:...", "mysql", "mysql"));
    aManager.authenticateUser("user", "test");
  }
```

Пример 2. UML.


<<interface>>
UserStorage
+getUserByName(String theName)
<<use>>
LoginManager
LdapUserStorage
UserList
JdbcUserStorage
<<instantiate>>
<<instantiate>>
<<instantiate>>
<<instantiate>>
Creates and binds
RuntimeAssembler

Сборка. Итоги.

Magnetosoft

- RuntimeAssembler-классы не предназначены для повторного использования или наследования от них. В больших системах эти классы максимально запутаны.
- IoC-контейнеры как раз и предназначены для упрощения соединения компонентов. Все они позволяют использовать API и создавать RuntimeAssember-классы, при этом XML-конфигурация системы и RuntimeAssembler не понадобятся.

Service Locator

Magnetosoft

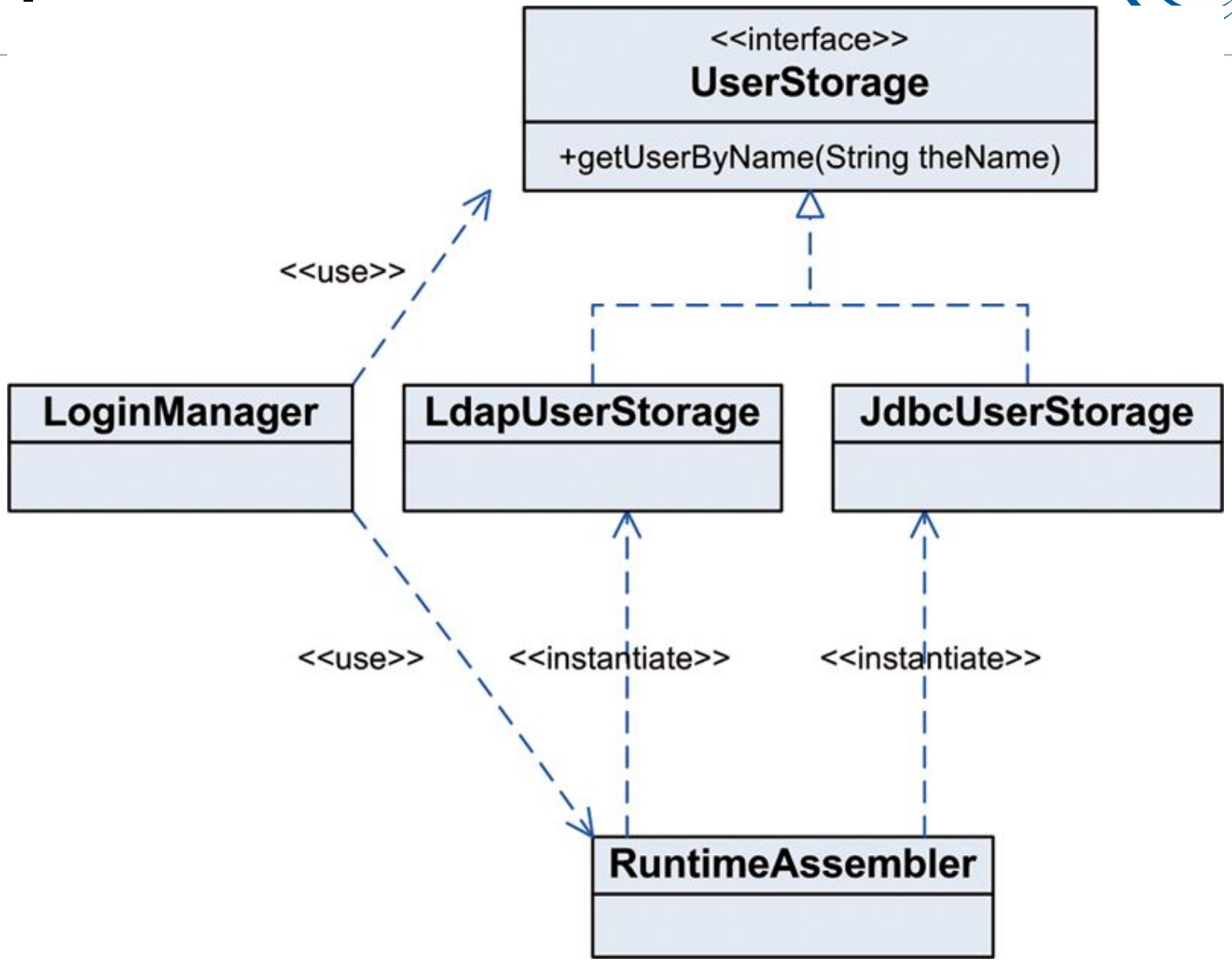
- Альтернативой паттерну вынесения зависимости (Dependency Injection) является паттерн Service Locator. Он широко используется в J2EE. Так, ServiceLocator может инкапсулировать все JNDI-вызовы J2EE-системы или создание (получение) UserStorage-реализации в нашем примере. Основным отличием Dependency Injection и Service Locator является то, что в Service Locator для получения реализации UserStorage используется вызов объекта ServiceLocator, в то время как в Dependency Injection ассемблер создает связь автоматически.

Пример 3. Service Locator

Magnetosoft

```
// Service Locator
public LoginManager()
{
  myUserList = ServiceLocator.getUserStorage();
}
```

Пример 3. UML.


<<interface>>
UserStorage
+getUserByName(String theName)
<<use>>
LoginManager
LdapUserStorage
JdbcUserStorage
<<use>>
<<instantiate>>
<<instantiate>>
RuntimeAssembler

Пример 3. Итоги

Magnetosoft

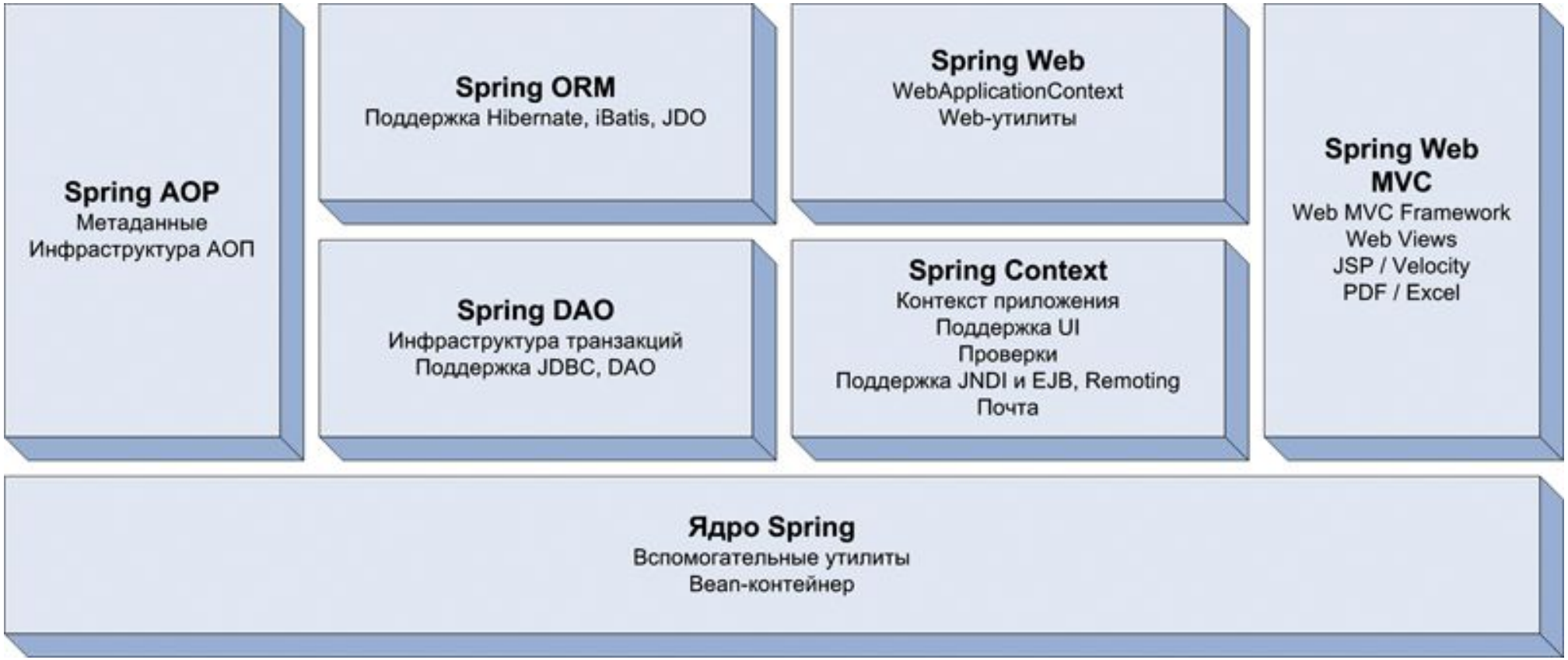
- Очевидно, что в паттерне ServiceLocator есть зависимость между LoginManager и ServiceLocator, в то время как LoginManager не зависит от RuntimeAssembler в Dependency Injection. Каждый может выбирать один из этих IoC-паттернов, в зависимости от своих нужд и задач.
- Далее будет рассмотрен IoC-контейнер, реализованный в Spring Framework.

Spring

- SpringFramework Spring Framework представляет собой набор готовых решений для использования всех основных Enterpise Java технологий — JDBC, ORM, JTA, Servlets/JSP, JMX и многих других. Абстрактные классы, фабрики и бины разработаны таким образом, чтобы программисту оставалось написать только свою логику

Spring. Элементы.

Magnetosoft



Пример 4. Spring Container.

- XML файл контейнера может быть таким:

```
<beans>
    <bean id="helloWorld" class="samples.HelloWorldImpl">
     <property name="message">
          <value>Sergei</value>
     </property>
    </bean>
</beans>
```

Пример 4. Spring Container.

Magnetosoft

```
public class HelloWorldImpl {
    private String myMessage;
    public void setMessage(String theMessage) {
      myMessage = theMessage;
    }
    public void sayMessage() {
      System.out.println("Hello world!Hello " + myMessage + "!");
    }
}

public class Application {
    public static void main(String[] args) {
      BeanFactory aBeanFactory = new XmlBeanFactory("sample-beans.xml");
      HelloWorld aHelloWorld = (HelloWorld) aBeanFactory.getBean("helloWorld");
      // выводит "Hello world!Hello Sergei!" в System.out
      aHelloWorld.sayMessage();
    }
}
```

Создание объектов

Magnetosoft

- В приведенном примере за конструирование объекта helloWorld отвечает контейнер – атрибут class элемента bean соответствует вызову конструктора без параметров.
- Spring поддерживает широкий спектр механизмов создания объектов – вызов конструктора с параметрами или без, использование фабрик классов или фабричных методов.

Конструктор без параметра

Magnetosoft

```
<bean id="helloWorld" class="sample.helloWorldImpl">
 ....
</bean>
```

Конструктор с параметрами

Magnetosoft

```
<beans>
    <bean id="exampleBean" class="examples.ExampleBean">
     <constructor-arg>
          <ref bean="anotherExampleBean"/>
     </constructor-arg>
     <constructor-arg>
          <ref bean="yetAnotherBean"/>
     </constructor-arg>
     <constructor-arg type="int">
          <value>1</value>
     </constructor-arg>
    </bean>
    <bean id="anotherExampleBean" class="examples.AnotherBean"/>
    <bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
</beans>
```

Фабричный метод

Magnetosoft

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <bean id="exampleBean" class="examples.ExampleBean"
    factory-method="createInstance">
     <constructor-arg>
          <ref bean="anotherExampleBean"/>
     </constructor-arg>
     <constructor-arg>
          <ref bean="yetAnotherBean"/>
     </constructor-arg>
     <constructor-arg>
          <value>1</value>
     </constructor-arg>
    </bean>
    <bean id="anotherExampleBean" class="examples.AnotherBean"/>
    <bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

Фабрика классов

Magnetosoft

```
<beans>
    <bean id="exampleBean" factory-bean="factory"
    factory-method="createInstance">
     <constructor-arg>
          <ref bean="anotherExampleBean"/>
     </constructor-arg>
     <constructor-arg>
          <ref bean="yetAnotherBean"/>
     </constructor-arg>
     <constructor-arg>
          <value>1</value>
     </constructor-arg>
    </bean>
    <bean id="anotherExampleBean" class="examples.AnotherBean"/>
    <bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
    <bean id="factory" class="examples.UserFactory"/>
```

Установка зависимостей и свойств

Magnetosoft

- Возникает вопрос: "Какие типы данных могут быть установлены при помощи элемента property?", - ответ: "Любые".
- Spring использует технологию JavaBeans для установки свойств объектов, эта технология частично используется в JSP для установки свойств типа String и примитивных типов, но в Spring она используется гораздо шире – интерфейс PropertyEditor позволяет устанавливать значения свойств любых типов.
- Поддерживаются стандартные коллекции из java.util: List, Set, Map, Properties, а также ссылки на объекты в контейнере по имени и значение null.

Установка зависимостей и свойств

Magnetosoft

- java.util.Properties – задается элементом props, отдельные свойства добавляются при помощи вложенного элемента prop, где атрибут key задает имя свойства, а текст внутри – значение.
- java.util.Map – задается элементом map, отдельные элементы добавляются при помощи вложенного элемента entry, где атрибут key задает имя свойства, а значение – это значение внутреннего элемента. Внутренний элемент entry может представлять любой объект, в том числе и Map.
- java.util.List, java.util.Set – представляются элементами list и set соответственно. Каждый внутренний элемент представляет собой значение элемента списка (множества).
- ссылка на объект из контейнера – представляется элементом ref, причем ссылка на объект, определенный в том же файле, использует атрибут local, а в любом из конфигурационных

Итоги

Magnetosoft

Избавление от зависимости от конкретной реализации обладает следующими преимуществами:

1. Повышается тестируемость кода. Вместо конкретной реализации всегда можно подсунуть Mock.
2. При написании распределенного приложения достаточно реализовать специфичную для клиента версию контракта (интерфейса).
3. В том случае, когда «клиент» напрямую зависит от «другого» класса, существует большая опасность завязаться на особенности конкретной реализации «другого» класса. Таким образом, изменения в реализации «другого» могут отразится и на «клиенте», и вызвать в нем так же ряд изменений.