Основные J2EE-паттерны
Service Locator (локатор службы)

Контекст

При поиске и создании службы применяются сложные интерфейсы и сетевые операции.

Проблема

J2EE-клиенты взаимодействуют с компонентами службы, такими как компоненты Enterprise JavaBeans (EJB) и Java Message Service (JMS), предоставляющими бизнес-службы и возможность сохранения данных. Для взаимодействия с этими компонентами клиенты должны либо найти компонент службы (так называемая операция поиска), либо создать новый компонент. Например, EJB-клиент должен найти домашний объект корпоративного компонента и затем использовать его либо для поиска объекта, либо для создания или удаления одного и более корпоративных компонентов. Подобным же образом JMS-клиент должен сначала найти центр соединений JMS для получения JMS-соединения или JMS-сессии.

Все клиенты J2EE-приложения для поиска и создания EJB-компонентов и JMS-компонентов используют стандартные функции JNDI. JNDI API предоставляет клиентам возможность получить исходный объект контекста, который содержит имя компонента для связывания объектов. Исходный компонент остается действительным до завершения клиентской сессии. Для получения ссылки на управляемый объект клиент предоставляет его зарегистрированное JNDI-имя. В контексте EJB-приложения типичным управляемым объектом является домашний объект корпоративного компонента. В JMS-приложениях управляемым объектом может быть центр соединений JMS (для Topic или Queue), либо пункт назначения JMS (Topic или Queue).

Итак, поиск управляемого JNDI-объекта службы является общим действием для всех клиентов, которым необходимо получить доступ к объекту службы. И это действительно так. Легко увидеть, что многие типы клиентов периодически используют JNDI-службу и для них периодически выполняется JNDI-код. Это приводит к лишнему дублированию кода в клиентских приложениях, выполняющих поиск служб.

Кроме того, при создании исходного JNDI-объекта контекста и выполнении операции поиска домашнего EJB-объекта расходуются значительные ресурсы. Если несколько клиентов периодически нуждаются в одном и том же домашнем объекте компонента, такие одновременные действия могут негативно отразиться на производительности приложения.

Рассмотрим процессы поиска и создания различных J2EE-компонентов.

1. Поиск и создание корпоративных компонентов основывается на следующем:

    - Правильно установить JNDI-среду для соединения с службой имен и каталогов, используемой приложением. Эта установка определяет местонахождение службы имен и необходимые данные аутентификации для доступа к этой службе.
    - Затем JNDI-служба может предоставить клиенту исходный контекст, действующий в качестве контейнера для связей имя-объект компонента. Клиент запрашивает этот исходный контекст для поиска объекта EJBHome требуемого корпоративного компонента, указывая JNDI-имя для этого объекта EJBHome.
    - Найти объект EJBHome при помощи имеющегося в исходном контексте механизма поиска.
    - После получения объекта EJBHome создать, удалить или найти корпоративный компонент при помощи методов create, move и find объекта EJBHome (только для компонентов управления данными).
2. Поиск и создание JMS-компонентов (Topic, Queue, QueueConnection, QueueSession, TopicConnection, TopicSession и т.д.) включает в себя следующие шаги. Обратите внимание, что здесь Topic относится к модели обмена сообщений «публикация-подписка», а Queue - к модели «точка-точка».
    - Установить JNDI-среду для службы имен, используемой приложением. Эта установка определяет местонахождение службы имен и необходимые данные аутентификации для доступа к этой службе.
    - Получить исходный контекст для провайдера JMS-службы из службы JNDI-имен.
    - Использовать исходный контекст для получения Topic или Queue, предоставляя JNDI-имя темы или очереди. Topic и Queue представляют собой объекты JMSDestination.
    - Использовать исходный контекст для получения TopicConnectionFactory или QueueConnectionFactory, предоставив JNDI-имя центра соединений для темы или очереди.
    - Использовать TopicConnectionFactory для получения TopicConnection или QueueConnectionFactory для получения QueueConnection.
    - Использовать TopicConnection для получения TopicSession или QueueConnection для получения QueueSession.
    - Использовать TopicSession для получения TopicSubscriber или TopicPublisher требуемого Topic. Использовать QueueSession для получения QueueReceiver или QueueSender для требуемого Queue.

Процесс поиска и создания компонентов зависит от реализации генератора контекста сторонних производителей. Это вводит элемент зависимости от поставщика в прикладных клиентах, которым необходимо использовать функцию JNDI-поиска для определения корпоративных компонентов и JMS-компонентов, таких как темы, очереди и объекты центра соединений.

Ограничения

- EJB-клиенты должны использовать JNDI API для поиска объектов EJBHome, применяя зарегистрированное JNDI-имя корпоративного компонента.
- JMS-клиенты должны использовать JNDI API для поиска JMS-компонентов, применяя зарегистрированные для JMS-компонентов JNDI-имена, такие как центры соединений, очереди и темы.
- Используемый для создания исходного JNDI-контекста генератор контекста поставляется сторонними поставщиками и, следовательно, является зависящим от поставщика. Генератор контекста зависит также от типа искомого объекта. Контекст для JMS отличается от контекста для EJB у различных поставщиков.
- Операции поиска и создания компонентов служб могут быть сложными и могут использоваться периодически в различных клиентах приложения.
- Создание исходного контекста и операции поиска объекта службы при частом их вызове могут потреблять много ресурсов и влиять на производительность приложения. Это особенно проявляется в тех случаях, когда клиенты и службы расположены на различных уровнях.
- Для EJB-клиентов может потребоваться повторное установление соединения с первоначально использовавшимся экземпляром корпоративного компонента при наличии только его объекта Handle.

Решение

Используйте объект Service Locator для полного абстрагирования от использования JNDI и скрытия сложностей создания исходного контекста, поиска домашнего EJB-объекта и повторного создания EJB-объекта. Объект Service Locator может повторно использоваться несколькими клиентами, что уменьшает сложность кода, обеспечивает простое управление и улучшает производительность, предоставляя функции кэширования.

Этот паттерн уменьшает сложность клиента, являющуюся результатом его зависимостей и необходимости выполнять ресурсоемкие процессы поиска и создания. Для устранения этих проблем данный паттерн предоставляет механизм абстракции всех зависимостей и деталей сетевых взаимодействий в Service Locator.

Структура

На рисунке 8.31 показана диаграмма классов, представляющая взаимосвязи паттерна Service Locator.

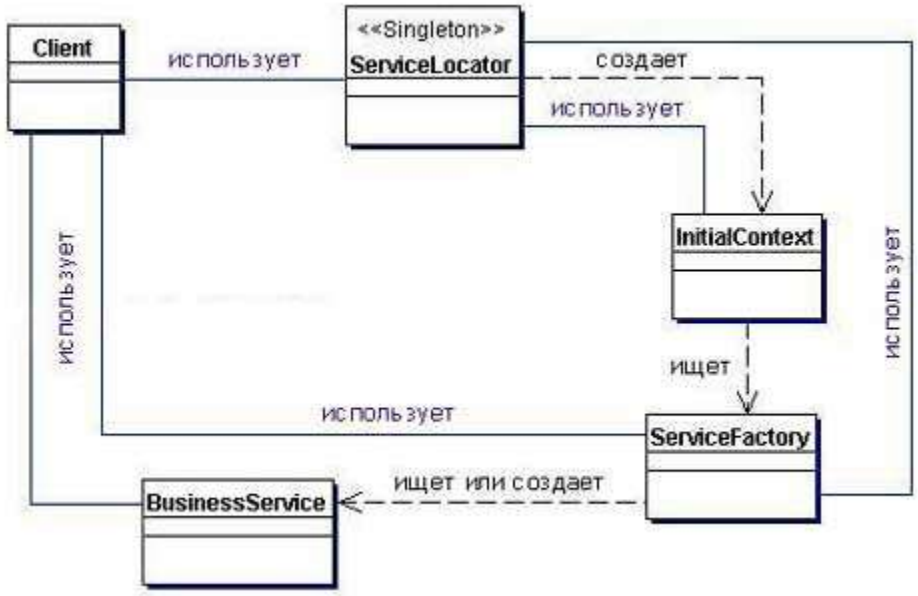

Рисунок 8.31 Диаграмма классов Service Locator

Участники и обязанности

На рисунке 8.32 представлена диаграмма последовательности действий, показывающая взаимодействия между различными участниками Service Locator.

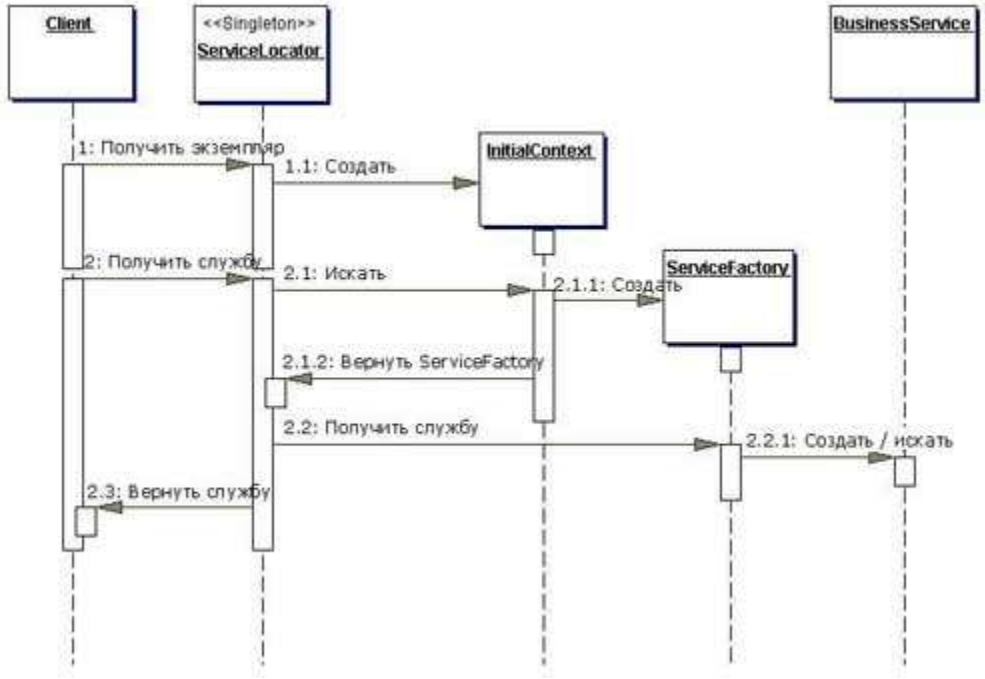

Рисунок 8.32 Диаграмма последовательности действий Service Locator

Client

Это клиент паттерна Service Locator. Клиент представляет собой объект, требующий обычно доступ к бизнес-объектам, таким как Business Delegate (см. раздел "Business Delegate" на странице 248).

Service Locator

Паттерн Service Locator абстрагирует API-службы поиска (именования), зависимости от поставщика, сложности операции поиска и создания бизнес-объекта, и предоставляет клиентам простой интерфейс. Это уменьшает сложность клиента. Кроме того, один и тот же клиент, или другие клиенты, могут использовать Service Locator повторно.

InitialContext

Объект InitialContext представляет собой точку отсчета для процессов поиска и создания. Поставщики служб обеспечивают объект контекста, который зависит от типа бизнес-объекта, предоставленного службами поиска и создания паттерна Service Locator. Service Locator, обеспечивающий службы для различных типов бизнес-объектов (таких как корпоративные компоненты, JMS-компоненты и др.) использует несколько типов объектов контекста, каждый из которых получен от различных поставщиков (например, провайдер контекста сервера EJB-приложения может отличаться от провайдера контекста JMS-службы).

ServiceFactory

ServiceFactory представляет собой объект, обеспечивающий управление циклом жизни объектов BusinessService. Объектом ServiceFactory для корпоративных компонентов является объект EJBHome. Объектом ServiceFactory для JMS-компонентов может быть JMS-объект ConnectionFactory, например TopicConnectionFactory (для модели обмена сообщениями «публикация-подписка»), либо QueueConnectionFactory (для модели обмена сообщениями «точка-точка»).

BusinessService

BusinessService представляет собой роль, выполняющуюся службой, доступ к которой ищет клиент. Объект BusinessService создается, ищется или удаляется при помощи ServiceFactory. Объектом BusinessService в контексте EJB-приложения является корпоративный компонент. Объектом BusinessService в контексте JMS-приложения может быть либо TopicConnection, либо QueueConnection. TopicConnection и QueueConnection могут затем использоваться для создания объекта JMSSession, т.е. TopicSession или QueueSession соответственно.

Стратегии

Стратегия EJB Service Locator

Service Locator для корпоративных компонентов использует объект EJBHome, представленный как BusinessHome в роли ServiceFactory. Как только объект

EJBHome получен, он может быть кэширован в ServiceLocator для дальнейшего использования. Это позволяет избежать повторных операций JNDI-поиска в тех случаях, когда клиенту снова понадобится домашний объект. В зависимости от реализации домашний объект может быть возвращен клиенту, который может затем использовать его для поиска, создания и удаления корпоративных компонентов. ServiceLocator может запомнить (кэшировать) домашний объект и выступать в качестве прокси-объекта для всех запросов клиента домашнему объекту. Диаграмма классов стратегии EJB Service Locator показана на рисунке 8.33.

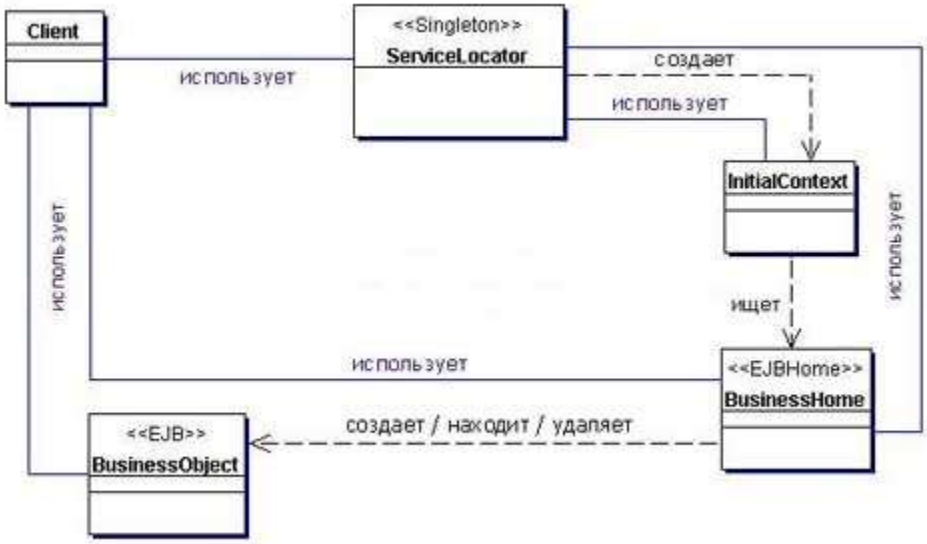

Рисунок 8.33 Диаграмма классов стратегии EJB Service Locator

Взаимодействия между участниками в Service Locator для корпоративного компонента показаны на рисунке 8.34.

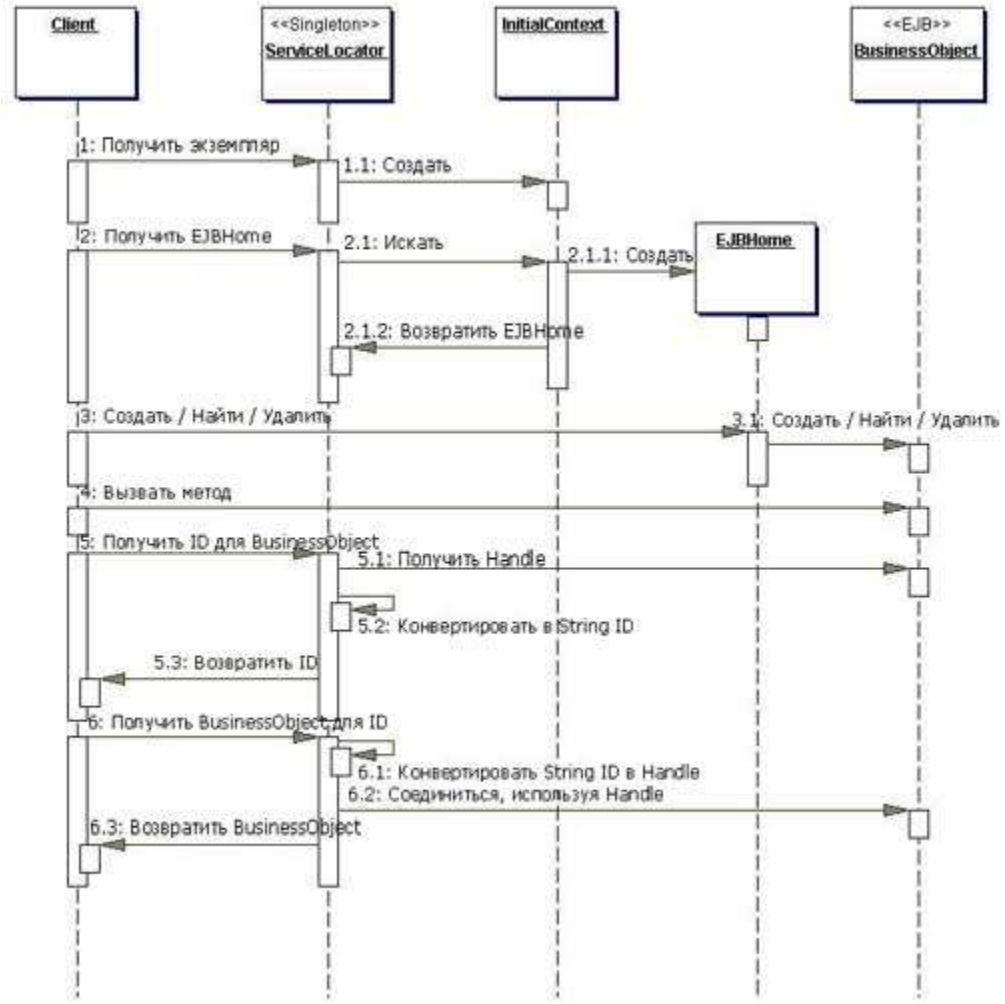

Рисунок 8.34 Диаграмма последовательности действий стратегии EJB Service Locator

Стратегия JMS Queue Service Locator

Данная стратегия применяется для модели обмена сообщениями «точка-точка». Service Locator JMS-компонентов в роли ServiceFactory использует объекты QueueConnectionFactory. QueueConnectionFactory ищется по его JNDI-имени. QueueConnectionFactory может быть кэширован в ServiceLocator для дальнейшего использования. Это устраняет повторяющиеся JNDI-вызовы от клиента. В противоположность этому ServiceLocator может передать контроль над QueueConnectionFactory клиенту. Клиент может использовать его для создания QueueConnection. QueueConnection необходим для получения QueueSession или создания Message, QueueSender (для передачи сообщений в очередь) или QueueReceiver (для приема сообщений из очереди). Диаграмма классов стратегии JMS Queue Service Locator показана на рисунке 8.35. На этой диаграмме Queue является объектом JMS Destination, зарегистрированным как управляемый JNDI объект очереди. Объект Queue может быть получен непосредственно из контекста путем поиска по его JNDI-имени.

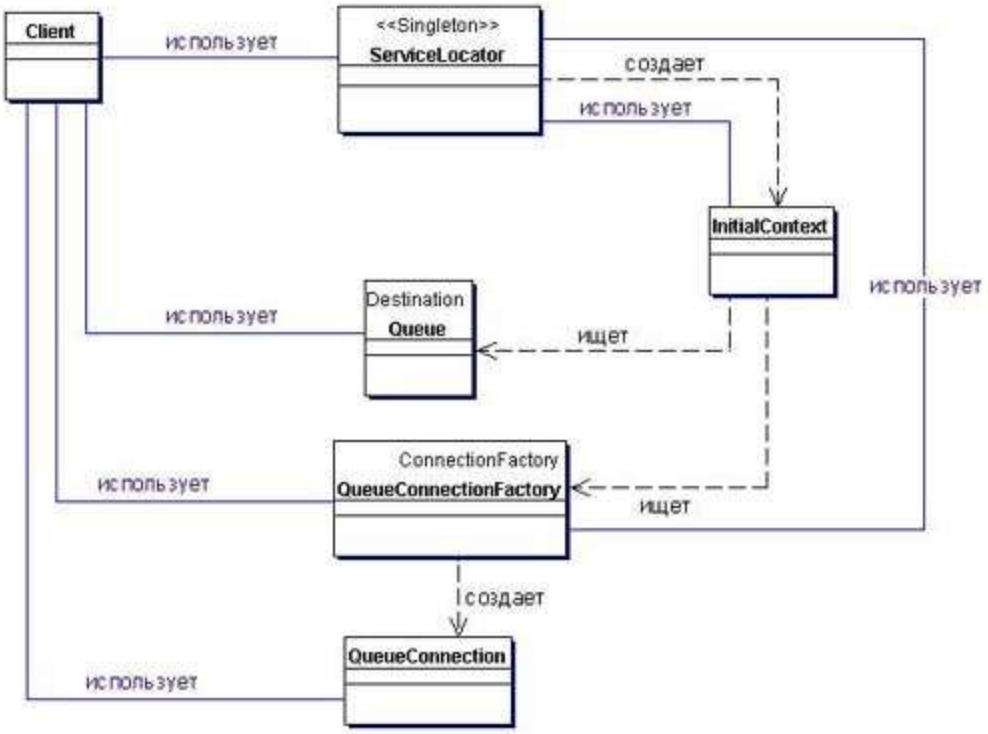

Рисунок 8.35 Диаграмма классов стратегии JMS Queue Service Locator

Взаимодействия между участниками в Service Locator для модели обмена сообщениями «точка-точка» показаны на рисунке 8.36.

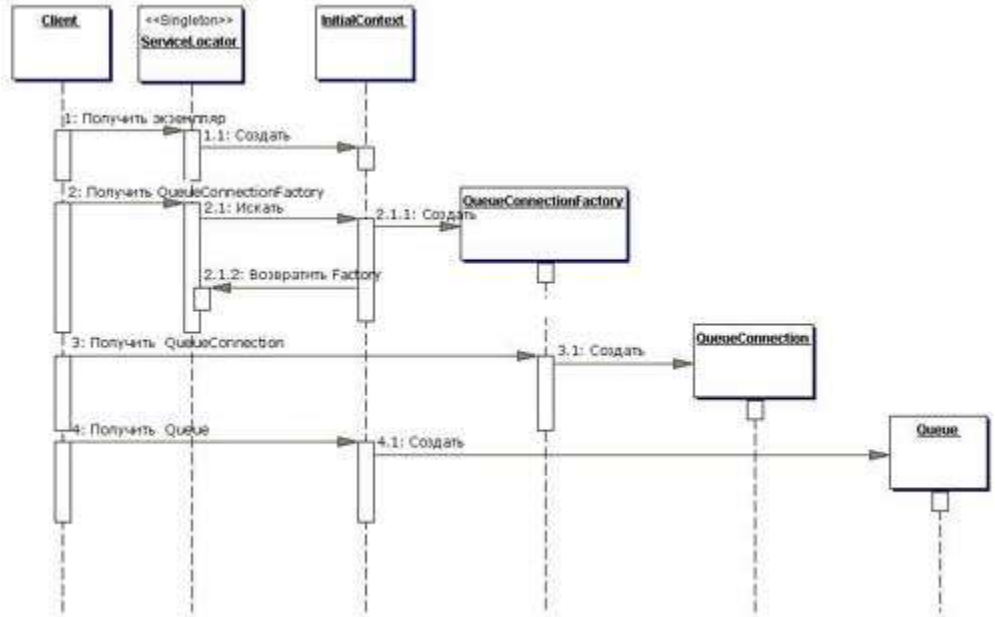

Рисунок 8.36 Диаграмма последовательности действий стратегии JMS Queue Service Locator

Стратегия JMS Topic Service Locator

Данная стратегия применяется для модели обмена сообщениями «публикация-подписка». Service Locator для JMS-компонентов в роли ServiceFactory использует объекты TopicConnectionFactory. TopicConnectionFactory ищется по его JNDI-имени. TopicConnectionFactory может быть кэширован в ServiceLocator для дальнейшего использования. Это устраняет повторяющиеся JNDI-вызовы от клиента. В противоположность этому ServiceLocator может передать контроль над TopicConnectionFactory клиенту. Клиент может использовать его для создания TopicConnection. TopicConnection необходим для получения TopicSession или создания Message, TopicPublisher (для публикации сообщений в теме) или TopicSubscriber (для подписки темы). Диаграмма классов стратегии JMS Topic Service Locator показана на рисунке 8.37. На этой диаграмме Topic является объектом JMS Destination, зарегистрированным как управляемый JNDI объект темы. Объект Topic может быть получен непосредственно из контекста путем поиска по его JNDI-имени

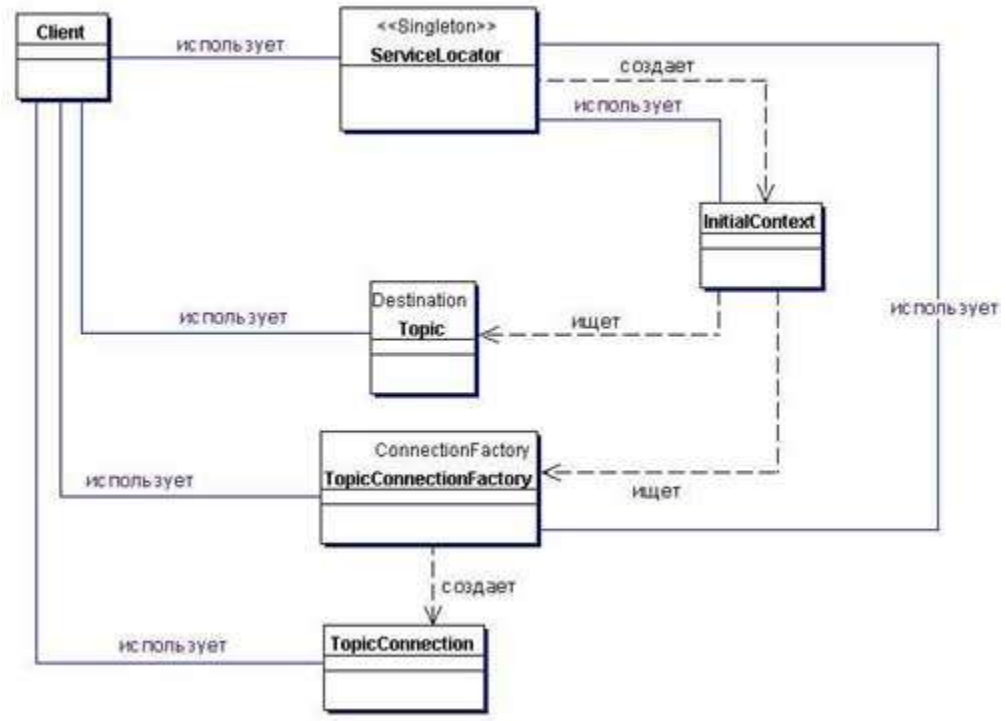

Рисунок 8.37 Стратегия JMS Topic Service Locator

Взаимодействия между участниками в Service Locator для модели обмена сообщениями «публикация-подписка» с использованием JMS Topic показаны на рисунке 8.38.

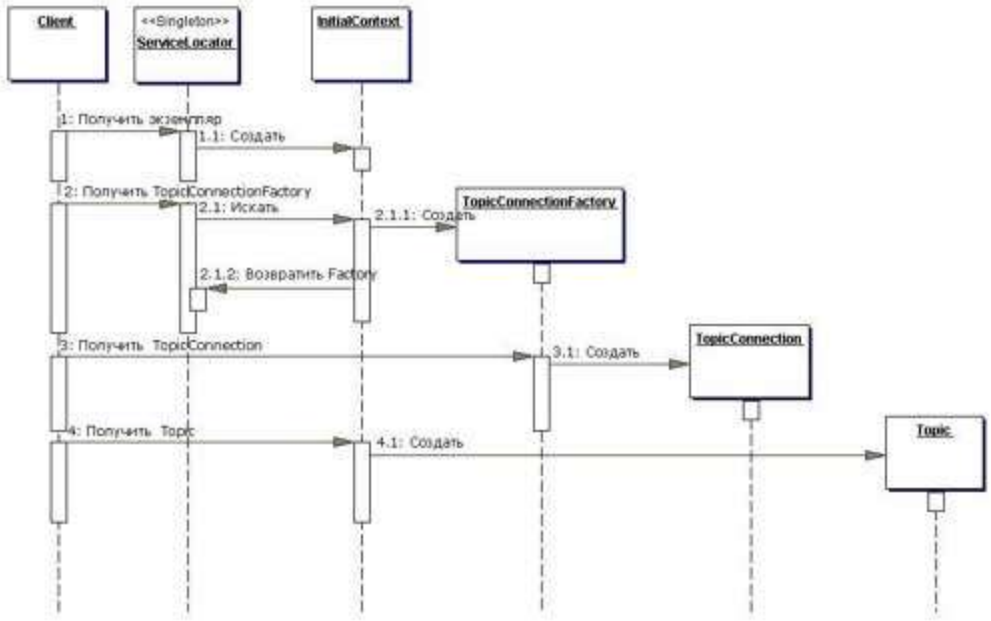

Рисунок 8.38 Диаграмма последовательности действий стратегии JMS Topic Service Locator

Комбинированная стратегия EJB и JMS Service Locator

Стратегии для EJB и JMS могут быть использованы для предоставления отдельных реализаций Service Locator, поскольку клиенты для EJB и JMS вероятнее всего являются взаимоисключающими. Однако, при необходимости скомбинировать эти стратегии есть возможность это сделать путем предоставления Service Locator для всех объектов - корпоративных компонентов и JMS-компонентов.

Стратегия Type Checked Service Locator (стратегия с проверкой типов)

Диаграммы на рисунках 8.37 и 8.38 обеспечивают функции поиска путем передачи службе искомого имени. Для поиска корпоративного компонента в Service Locator необходимо передать класс в качестве параметра метода PortableRemoteObject.narrow(). Service Locator может предоставить метод getHome (), который принимает в качестве аргументов JNDI-имя службы и объект класса EJBHome корпоративного компонента. Использование такого метода передачи JNDI-имени службы и объектов класса EJBHome может привести к ошибкам. Другой подход - статически определять службы в ServiceLocator. Вместо передачи строковых имен клиент передает константы. Эта стратегия применяется в примере 8.34.

Данная стратегия имеет недостатки. Она уменьшает гибкость поиска по сравнению со стратегией Services Property Locator, но добавляет проверку типов при передаче констант в метод ServiceLocator.getHome().

Стратегия Service Locator Properties (стратегия свойств)

Данная стратегия позволяет устранить недостатки стратегии с проверкой типов.

Здесь предлагается использование файлов свойств и дескрипторов размещения для указания JNDI-имен и имени класса EJBHome. Для клиентов уровня презентации эти свойства могут быть указаны в дескрипторах размещения или файлах свойств. При обращении этого уровня к бизнес-уровню обычно используется паттерн Business Delegate.

Для поиска бизнес-компонентов Business Delegate взаимодействует с Service Locator. Если уровень презентации загружает свойства при инициализации и может предоставлять службе JNDI-имена и имена EJB-классов для требуемого корпоративного компонента, то Business Delegate для их получения мог бы послать запрос к этой службе. Как только Business Delegate получает JNDI-имя и имя класса EJBHome, он может запросить Service Locator для EJBHome путем передачи этих свойств в качестве аргументов.

Service Locator может, в свою очередь, использовать Class.forName(EJBHome ClassName) для получения объекта класса EJBHome и выполнить свою функцию, найдя EJBHome и используя метод Portable RemoteObject.narrow() для приведения объекта, как показано в методе getHome() объекта ServiceLocator в примере 8.33. Единственное изменение – место, откуда берутся JNDI-имя и объекты Class. То есть, данная стратегия устраняет необходимость жесткого кодирования JNDI-имен в программе и обеспечивает гибкость размещения. Однако, из-за отсутствия проверки типов существуют некоторые ограничения для устранения ошибок и несоответствий при указании JNDI-имен в различных дескрипторах размещения.

Выводы

- **Абстрагирует сложность**
  Паттерн Service Locator инкапсулирует сложности процессов поиска и создания (описанных выше в проблеме) и скрывает их от клиента. Клиент не должен иметь дело с поиском промежуточных объектов компонента (среди которых EJBHome, QueueConnectionFactory и TopicConnectionFactory), поскольку эти функции передаются объекту ServiceLocator.
- **Обеспечивает клиентам унифицированный доступ к службе**
  Паттерн Service Locator абстрагирует все сложности, как объяснялось выше. Для этого он предоставляет очень полезный и точный интерфейс, который могут использовать все клиенты. Этот интерфейс дает гарантию того, что все типы клиентов в приложении обращаются к бизнес-объектам единым способом, в терминах поиска и создания. Такая унификация уменьшает накладные расходы при разработке и обслуживании.
- **Облегчает добавление новых бизнес-компонентов**
  Поскольку клиенты корпоративных компонентов не имеют дела с объектами EJBHome, есть возможность добавить новые объекты EJBHome для корпоративных компонентов, разработанных и размещенных позднее, без изменения клиентов. JMS-клиенты непосредственно не сталкиваются с центрами JMS-соединений, то есть новые центры соединений могут быть добавлены без изменения клиентов.
- **Улучшает сетевую производительность**

Клиенты не участвуют в JNDI-поиске и создании объектов-генераторов и домашних объектов. Поскольку эту работу выполняет Service Locator, он может группировать сетевые вызовы, требуемые для поиска и создания бизнес-объектов.

- **Улучшает производительность клиента, используя кэширование**
  Service Locator способен кэшировать исходные объекты контекста и ссылки на объекты-генераторы (EJBHome, центры соединений JMS) для устранения не обязательных JNDI-вызовов при получении исходного контекста и других объектов. Это улучшает производительность приложения.

Пример

Реализация паттерна Service Locator

Возможная реализация паттерна Service Locator приводится в примере 8.33. Возможная реализация стратегии Type Checked Service Locator приводится в примере 8.34.

Пример 8.33 Реализация Service Locator

```
package corepatterns.apps.psa.util;
import java.util.*;
import javax.naming.*;
import java.rmi.RemoteException;
import javax.ejb.*;
import javax.rmi.PortableRemoteObject;
import java.io.*;

public class ServiceLocator {
  private static ServiceLocator me;
  InitialContext context = null;

  private ServiceLocator()
  throws ServiceLocatorException {
    try {
      context = new InitialContext();
    } catch(NamingException ne) {
      throw new ServiceLocatorException(...);
    }
  }

  // Возвращает экземпляр класса ServiceLocator
  public static ServiceLocator getInstance()
  throws ServiceLocatorException {
    if (me == null) {
      me = new ServiceLocator();
    }
    return me;
  }

  // Преобразует сериализованную строку в EJBHandle
  // затем в EJBObject.
  public EJBObject getService(String id)
  throws ServiceLocatorException {
    if (id == null) {
      throw new ServiceLocatorException(...);
```

```
    }
    try {
      byte[] bytes = new String(id).getBytes();
      InputStream io = new
        ByteArrayInputStream(bytes);
      ObjectInputStream os = new
        ObjectInputStream(io);
      javax.ejb.Handle handle =
        (javax.ejb.Handle)os.readObject();
      return handle.getEJBObject();
    } catch(Exception ex) {
      throw new ServiceLocatorException(...);
    }
  }

  // Возвращает объект String, представляющий  идентификатор
  // данного EJBObject в сериализованном формате.
  protected String getId(EJBObject session)
  throws ServiceLocatorException {
    try {
      javax.ejb.Handle handle = session.getHandle();
      ByteArrayOutputStream fo = new
        ByteArrayOutputStream();
      ObjectOutputStream so = new
        ObjectOutputStream(fo);
      so.writeObject(handle);
      so.flush();
      so.close();
      return new String(fo.toByteArray());
    } catch(RemoteException ex) {
      throw new ServiceLocatorException(...);
    } catch(IOException ex) {
      throw new ServiceLocatorException(...);
    }
    return null;
  }

  // Возвращает объект EJBHome запрошенного имени
  // службы. Генерирует ServiceLocatorException при
  // возникновении любой ошибки при поиске
  public EJBHome getHome(String name, Class clazz)
  throws ServiceLocatorException {
    try {
      Object objref = context.lookup(name);
      EJBHome home = (EJBHome)
        PortableRemoteObject.narrow(objref, clazz);
      return home;
    } catch(NamingException ex) {
      throw new ServiceLocatorException(...);
    }
  }
}
```

Реализация стратегии Type Checked Service Locator

Пример 8.34 Реализация стратегии Type Checked Service Locator

```
package corepatterns.apps.psa.util;
// imports
...
```

```
public class ServiceLocator {
  // private экземпляр singleton
  private static ServiceLocator me;

  static {
    me = new ServiceLocator();
  }

  private ServiceLocator() {}

  // Возвращает экземпляр Service Locator
  static public ServiceLocator getInstance() {
    return me;
  }


  // Внутренний константный класс Services - вспомогательные объекты
  public class Services {
    final public static int PROJECT  = 0;
    final public static int RESOURCE = 1;
  }

  // Константы, относящиеся к EJB-проекту
  final static Class  PROJECT_CLASS = ProjectHome.class;
  final static String PROJECT_NAME  = "Project";

  // Константы, относящиеся к EJB-ресурсу

  final static Class  RESOURCE_CLASS = ResourceHome.class;
  final static String RESOURCE_NAME  = "Resource";

  // Возвращает Class запрошенной службы
  static private Class getServiceClass(int service){
    switch( service ) {
      case Services.PROJECT:
        return PROJECT_CLASS;
      case Services.RESOURCE:
      return RESOURCE_CLASS;
    }
    return null;
  }

  // Возвращает JNDI-имя запрошенной службы
  static private String getServiceName(int service){
    switch( service ) {
      case Services.PROJECT:
        return PROJECT_NAME;
      case Services.RESOURCE:
        return RESOURCE_NAME;
    }
    return null;
  }

  /* получает EJBHome для данной службы при помощи
  ** JNDI-имени и имени Class EJBHome
  */
  public EJBHome getHome( int s )
    throws ServiceLocatorException {
    EJBHome home = null;
    try {
        Context initial  = new InitialContext();
```

```
      // Поиск имени службы из
      // определенной константы
      Object objref =
        initial.lookup(getServiceName(s));

      // Приведение при помощи EJBHome Class из
      // определенной константы
      Object obj = PortableRemoteObject.narrow(
                objref, getServiceClass(s));
      home = (EJBHome)obj;
    }
    catch( NamingException ex ) {
        throw new ServiceLocatorException(...);
    }
    catch( Exception ex ) {
        throw new ServiceLocatorException(...);
    }
    return home;
  }
}
```

Код клиента для этой стратегии может выглядеть так, как показано в примере 8.35.

Пример 8.35 Код клиента, использующего Service Locator

```
public class ServiceLocatorTester {
  public static void main( String[] args ) {
    ServiceLocator serviceLocator =
      ServiceLocator.getInstance();
    try {
      ProjectHome projectHome = (ProjectHome)
        serviceLocator.getHome(
          ServiceLocator.Services.PROJECT );
    }
    catch( ServiceException ex ) {
      // клиент обрабатывает исключительные ситуации
      System.out.println( ex.getMessage( ));
    }
  }
}
```

Данная стратегия использует проверку типов при клиентском поиске. Она инкапсулирует статические значения службы в ServiceLocator и создает внутренний класс Services, который объявляет константы службы (PROJECT и RESOURCE). Клиент Tester получает экземпляр singleton ServiceLocator и вызывает getHome(), передавая PROJECT. ServiceLocator, в свою очередь, получает JNDI-имя входа и класс Home и возвращает их в EJBHome.

Связанные паттерны

- **Business Delegate**
  Паттерн Business Delegate использует Service Locator для получения доступа к объектам бизнес-службы, таким как EJB-объекты, JMS-темы и JMS-очереди. Это отделяет сложность поиска службы от паттерна Business Delegate, что приводит к ослаблению связанности и повышению управляемости.

- **Session Facade**
  Паттерн Session Facade использует Service Locator для получения доступа к корпоративным компонентам, вовлеченным в рабочий процесс. Session Facade мог бы непосредственно использовать Service Locator или делегировать функции в Business Delegate (См. раздел "Business Delegate" на стр. 248.).
- **Transfer Object Assembler**
  Паттерн Transfer Object Assembler использует Service Locator для получения доступа к различным корпоративным компонентам, в которых он нуждается при создании своего составного объекта Transfer Object. Transfer Object Assembler мог бы непосредственно использовать Service Locator или делегировать функции в Business Delegate (См. раздел "Business Delegate" на стр. 248.).