Базовые паттерны в многослойной архитектуре

По Мартину Фаулеру (Martin Fowler)

Список паттернов

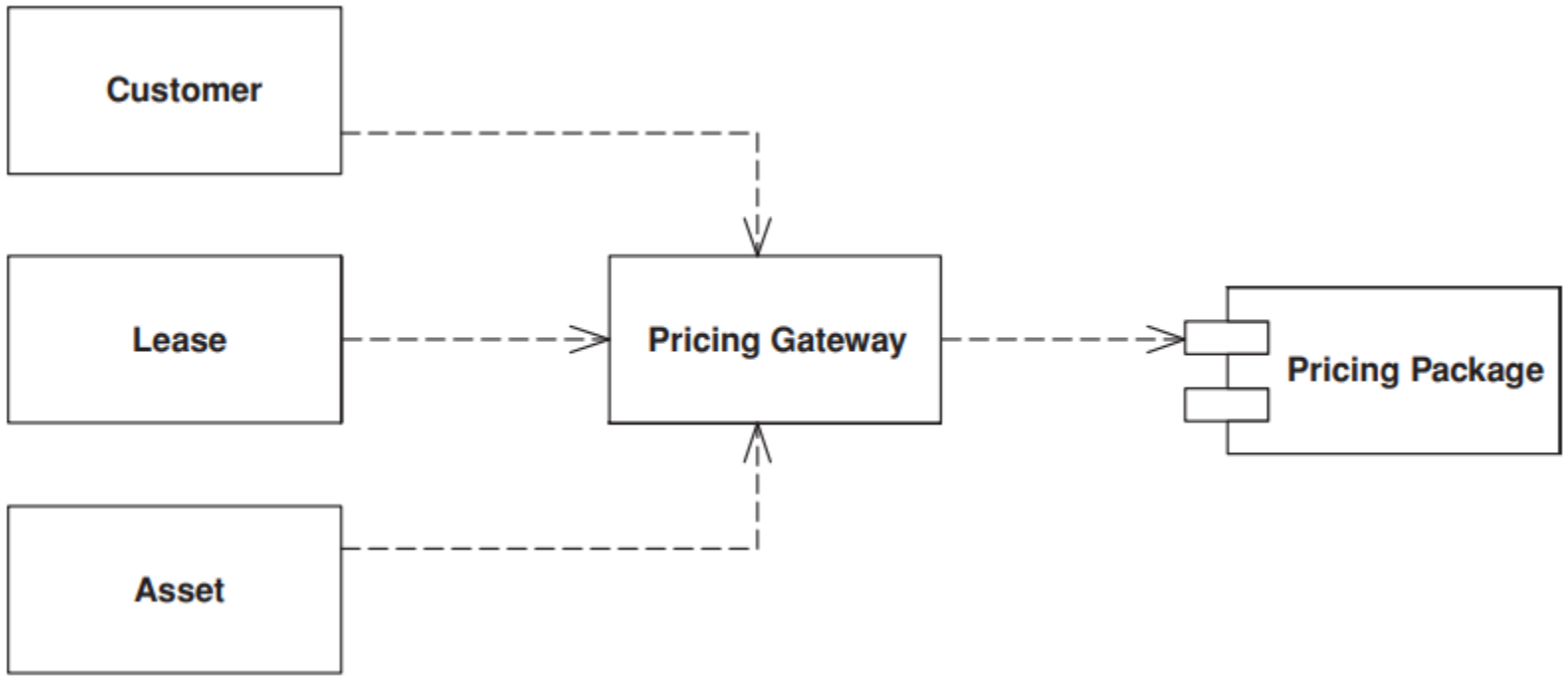
- Gateway
- Mapper
- Layer Supertype
- Separated Interface
- Registry
- Value Object
- Special Case
- Plugin
- Service Stub
- Record Set

Gateway

Объект, который инкапсулирует доступ к внешней системе и ресурсу

Похож на Фасад, Адаптер, Посредник

Gateway



Gateway. Когда использовать?

Яркий маркер для использования Gateway – наличие неудобного (странного, нелогичного) интерфейса для компонента, который можно расценить как внешний

Также используется в паттернах более высокого уровня

Gateway. Преимущества

- Систему проще тестировать
- Легко подменять компонент, скрываемый за Gateway

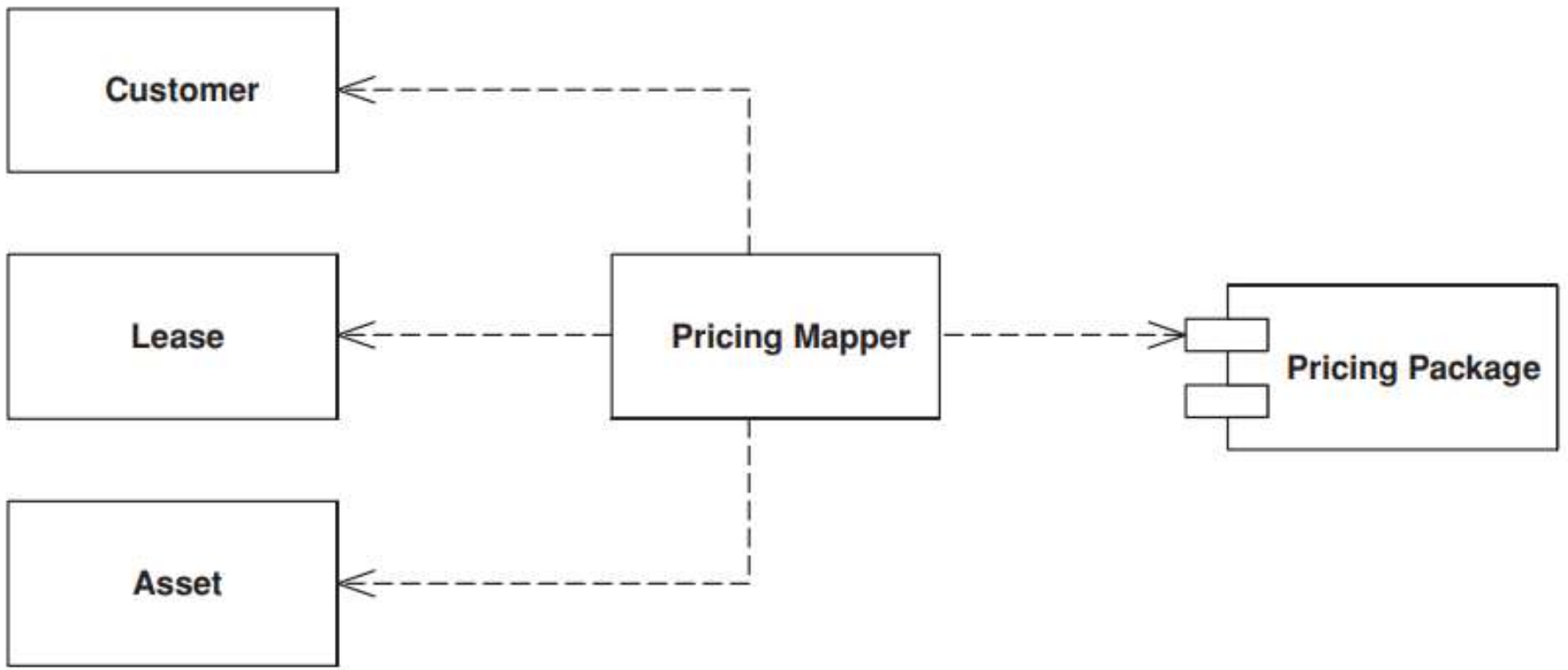
Mapper

Объект, который управляет сообщением между независимыми друг от друга объектами

Очень похож на Посредник, но подсистемы не в курсе о существовании Mapper

Mapper

An object that sets up a communication between two independent objects.



Mapper. Когда использовать?

Спектр применений ограничен, так как сложно сделать так, чтобы компоненты не знали о Mapper и он выполнял свои функции. Mapper может выполнять роль Observer для одного из компонентов

Используется, когда нужно полностью разделить связанные компоненты

Layer Supertype

Тип (класс), который является базовым для всех типов (классов) своего слоя

Если хотя бы несколько классов (в идеале все классы) одного слоя имеют в своём составе однотипные члены, то эти члены целесообразно вынести в базовый класс. Что позволяет избежать повторений в коде и в некоторой степени даже повысить гибкость архитектуры.

Layer Supertype. Пример на Java

```java
@Entity
@Table(name = " Products")
public class Product {
    private int id;
    private String barCode;
    private String name;
    private double cost;
    private Set<Bill> bills;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    public int getId() {
        return id;
    }
    public void setBills(Set<Bill> bills) {
        this.bills = bills;
    }
    @OneToMany(mappedBy = "product")
    public Set<Bill> getBills() {
        return bills;
    }
    @Column(name = "BarCode")
    public String getBarCode() {
        return barCode;
    }
    @Column(name = "Product")
    public String getName() {
        return name;
    }
    @Column(name = "Cost")
    public double getCost() {
        return cost;
    }
    public void setId(int id) {
        this.id = id;
    }
    public void setBarCode(String barCode) {
        this.barCode = barCode;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setCost(double cost) {
        this.cost = cost;
    }
    public Product() {
    }
}
```

Layer Supertype. Пример на Java

```java
@Entity
@Table(name = "Bills")
public class Bill {
    private int id;
    private int crNo;
    private int billNo;
    private Date billDate;
    private int goodNumber;
    private double goodCost;
    private double totalCost;
   private Product good;
    @ManyToOne
    @JoinColumn(name = "productId")
    public Product getProduct() {
        return product;
    }
    public void setProduct(Product product) {
        this.product = product;
    }
    public Bill() {
    }
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    public int getId() {
        return id;
    }
    @Column(name = "crNo")
    public int getCrNo() {
        return crNo;
    }
    @Column(name = "billNo")
    public int getBillNo() {
        return billNo;
    }
    @Column(name = "billDate")
    public Date getBillDate() {
        return billDate;
    }
    @Column(name = "goodNumder")
    public int getGoodNumber() {
        return goodNumber;
    }
    @Column(name = "goodCost")
    public double getGoodCost() {
        return goodCost;
    }
    @Column(name = "totalCost", insertable = false, updatable = false)
    public double getTotalCost() {
        return totalCost;
    }
    public void setId(int id) {
        this.id = id;
    }
    public void setCrNo(int crNo) {
        this.crNo = crNo;
    }
    public void setBillNo(int billNo) {
        this.billNo = billNo;
    }
    public void setBillDate(Date billDate) {
        this.billDate = billDate;
    }
    public void setGoodNumber(int goodNumber) {
        this.goodNumber = goodNumber;
    }
    public void setGoodCost(double goodCost) {
        this.goodCost = goodCost;
    }
    public void setTotalCost(double totalCost) {
        return;
    }
}
```

Layer Supertype. Пример на Java

```java
@MappedSuperclass
public class Model {
    private int id;
    public Model() {
    }
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

Layer Supertype. Пример на Java

```java
@Entity
@Table(name = "Products")
public class Product extends Model{
    private String barCode;
    private String name;
    private double cost;
    private Set<Bill> bills;
    public Product() {
        super();
    }
    public void setBills(Set<Bill> bills) {
        this.bills = bills;
    }
    @OneToMany(mappedBy = "product")
    public Set<Bill> getBills() {
        return bills;
    }
    @Column(name = "BarCode")
    public String getBarCode() {
        return barCode;
    }
    @Column(name = "Product")
    public String getName() {
        return name;
    }
    @Column(name = "Cost")
    public double getCost() {
        return cost;
    }
    public void setBarCode(String barCode) {
        this.barCode = barCode;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setCost(double cost) {

        this.cost = cost;

    }
}
```

Layer Supertype. Пример на Java

```java
@Entity
@Table(name = "Bills")
public class Bill extends Model {
    private int crNo;
    private int billNo;
    private Date billDate;
    private Product product;
    private int goodNumber;
    private double goodCost;
    private double totalCost;
    public Bill() {
        super();
    }
    @Column(name = "crNo")
    public int getCrNo() {
        return crNo;
    }
    @Column(name = "billNo")
    public int getBillNo() {
        return billNo;
    }
    @Column(name = "billDate")
    public Date getBillDate() {
        return billDate;
    }
    @ManyToOne
    @JoinColumn(name = "goodId")
    public Product getProduct() {
        return product;
    }
    public void setProduct(Product product) {
        this.product = product;
    }
    @Column(name = "goodNumder")
    public int getGoodNumber() {
        return goodNumber;
    }
    @Column(name = "goodCost")
    public double getGoodCost() {
        return goodCost;
    }
    @Column(name = "totalCost", insertable = false, updatable = false)
    public double getTotalCost() {
        return totalCost;
    }
    public void setCrNo(int crNo) {
        this.crNo = crNo;
    }
    public void setBillNo(int billNo) {
        this.billNo = billNo;
    }
    public void setBillDate(Date billDate) {
        this.billDate = billDate;
    }
    public void setGoodNumber(int goodNumber) {
        this.goodNumber = goodNumber;
    }
    public void setGoodCost(double goodCost) {
        this.goodCost = goodCost;
    }
    public void setTotalCost(double totalCost) {
        return;
    }
}
```

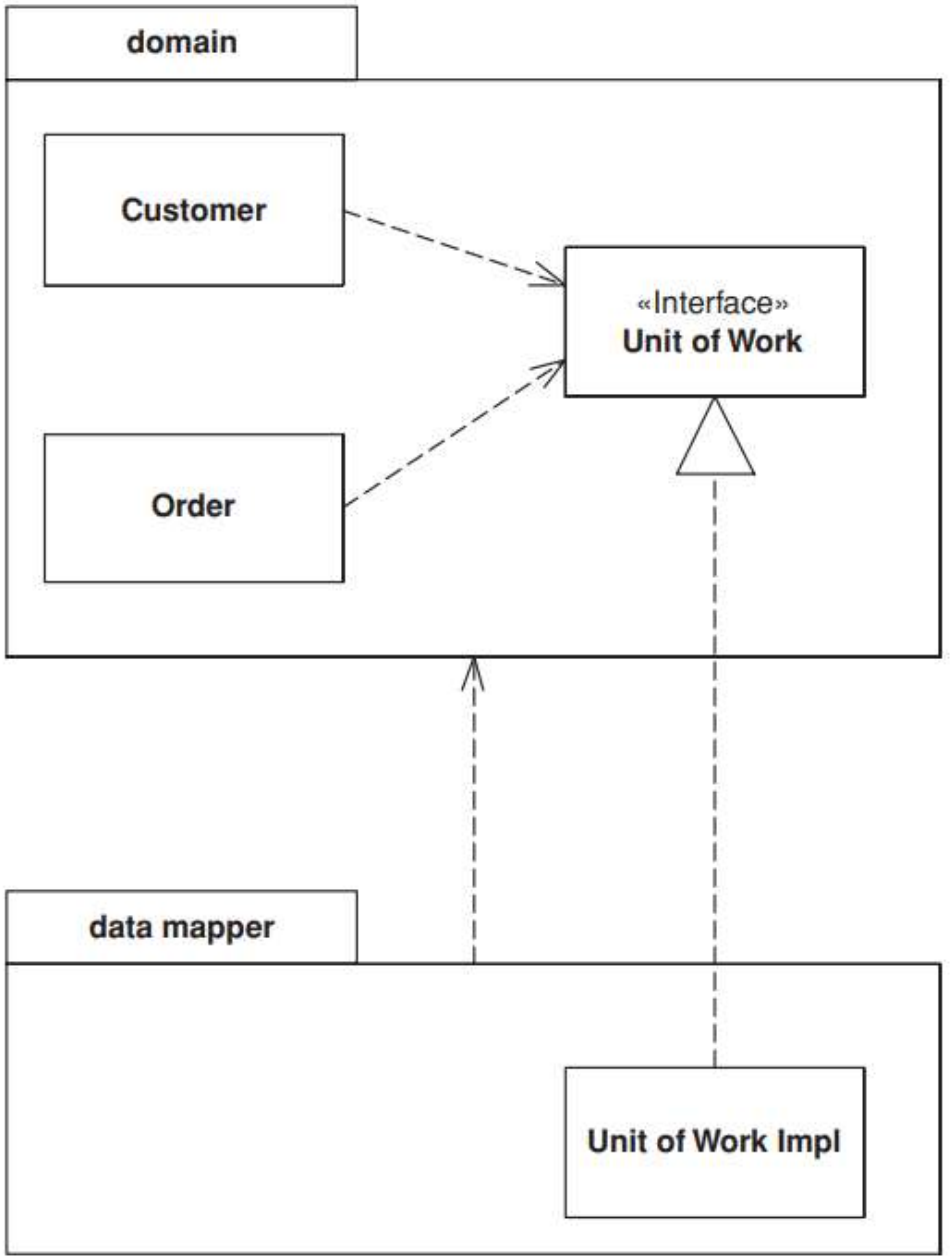
Separated Interface

Выделение какого-либо интерфейса к объекту в отдельный от объекта пакет

При разработке какой-либо системы, можно добиться улучшения её архитектуры, уменьшая связанность между её частями. Это можно сделать так - распределив классы по отдельным пакетам и контролировать зависимости этими пакетами. Тогда можно следовать правилам о том, как классы из одного пакета могут обращаться к классам из другого пакета. Например, то, которое запрещает классам с уровня данных обращаться к классам с уровня представления.

Defines an interface in a separate package from its implementation.

Separated Interface



Separated Interface. Когда применять?

Отделенный интерфейс применяется для того, чтобы избежать возникновения зависимостей между двумя частями системы

- Если вы разместили в пакете инфраструктуры абстрактный код для обработки стандартных случаев, который должен вызывать конкретный код приложения
- Если код одного слоя приложения должен обратиться к коду другого слоя, о существовании которого он не знает
- Если нужно вызвать методы, разработанные кем-то другим, но вы не хотите, что бы ваш код зависел от интерфейсов API этих разработчиков

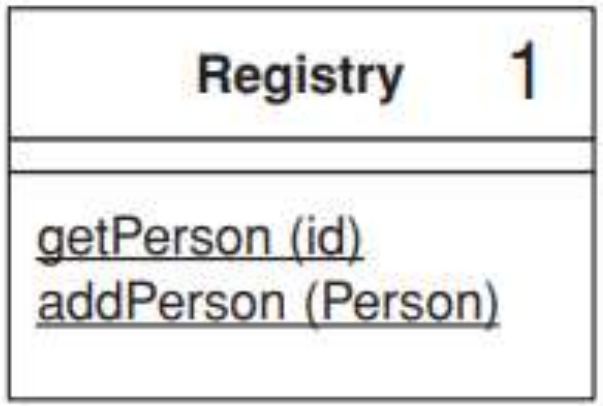
Registry

Хорошо известный объект, который используется другими объектами для получения общих объектов и сервисов

Когда нужно найти какой-нибудь объект, обычно начинают с другого объекта, связанного с целевым. Например, если нужно найти все счета для покупателя, начинают, как раз с покупателя и используют его метод получения счетов. Тем не менее, в некоторых случаях нет подходящего объекта, с которого начать. В таких случаях применяется Registry

Registry

A well-known object that other objects can use to find common objects and services.



Registry

Registry – так или иначе глобальный объект

Поэтому можно использовать Singleton для него

Value Object

Небольшой простой объект (например, деньги или временной интервал), чья идентичность основана на содержащихся в нём данных

Яркие примеры, встроенные в .NET:

- TimeSpan
- DateTime

Value Object

Значение можно считать Value Object, если:

- Оно измеряет, оценивает или описывает объект предметной области
- Его можно считать неизменяемым
- Оно моделирует нечто концептуально целостное, объединяет связанные атрибуты
- Его можно сравнить с другими такими объектами
- Методы, связанные с VO, должны быть без побочных эффектов

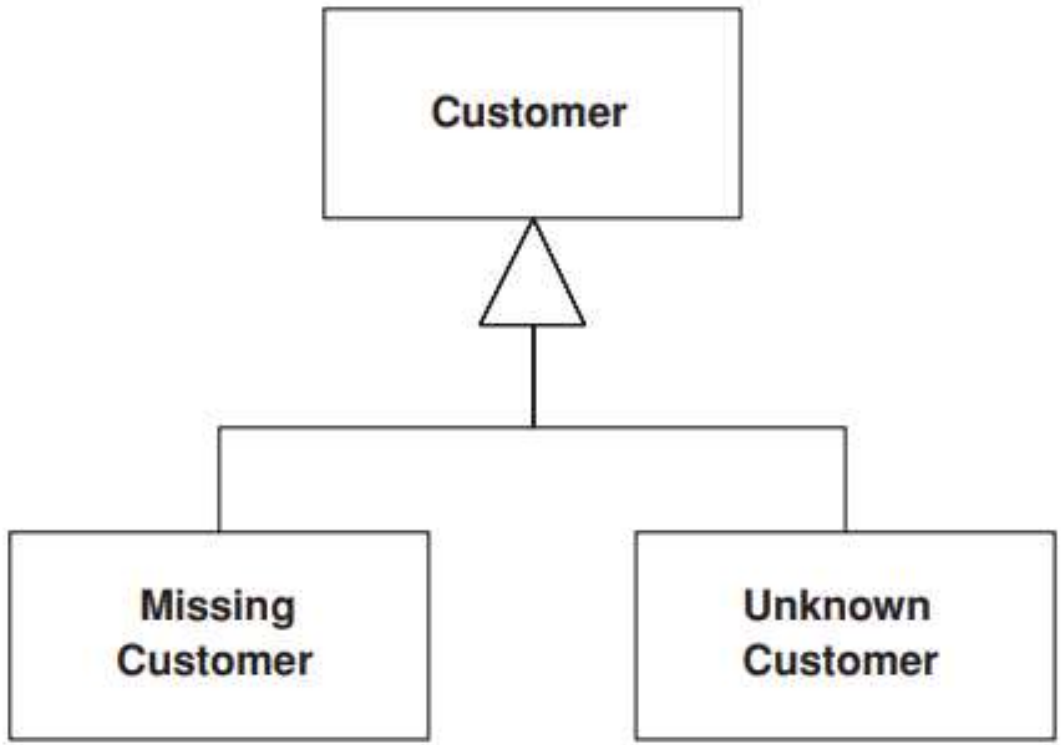
Special Case

Производный класс, описывающий поведение объекта в особых ситуациях

Так как зачастую не нужно больше одного экземпляра такого класса , то можно воспользоваться паттерном Легковес

Special Case

A subclass that provides special behavior for particular cases.



Special Case. Когда применять?

Самая популярная ситуация – поведение несуществующего (ненайденного, несозданного) объекта вашего класса. Другими словами, если из метода, который возвращает экземпляры вашего класса, вы хотите вернуть NULL – вместо этого вы создаете специальную NULL версию вашего класса и возвращаете её экземпляр

А так – это любой конкретный частный случай вашего класса, который ведёт себя по-особому

Special Case. Пример с NULL

```
class Employee...

      public virtual String Name {
         get {return _name;}
         set {_name = value;}
      }
      private String _name;
      public virtual Decimal GrossToDate {
         get {return calculateGrossFromPeriod(0);}
      }
      public virtual Contract Contract {
         get {return _contract;}
      }
      private Contract _contract;
```

```
class NullEmployee : Employee, INull...

      public override String Name {
         get {return "Null Employee";}
         set {}
      }
      public override Decimal GrossToDate {
         get {return 0m;}
      }
      public override Contract Contract {
         get {return Contract.NULL;}
      }
```

Plugin

Связывание классов друг с другом происходит не на этапе компиляции кода, а во время его выполнения. Plugin позволяет делать это централизованно, без необходимости редактирования и сборки исходного кода для изменения конфигурации

Plugin

Links classes during configuration rather than compilation.

Plugin. Схема работы

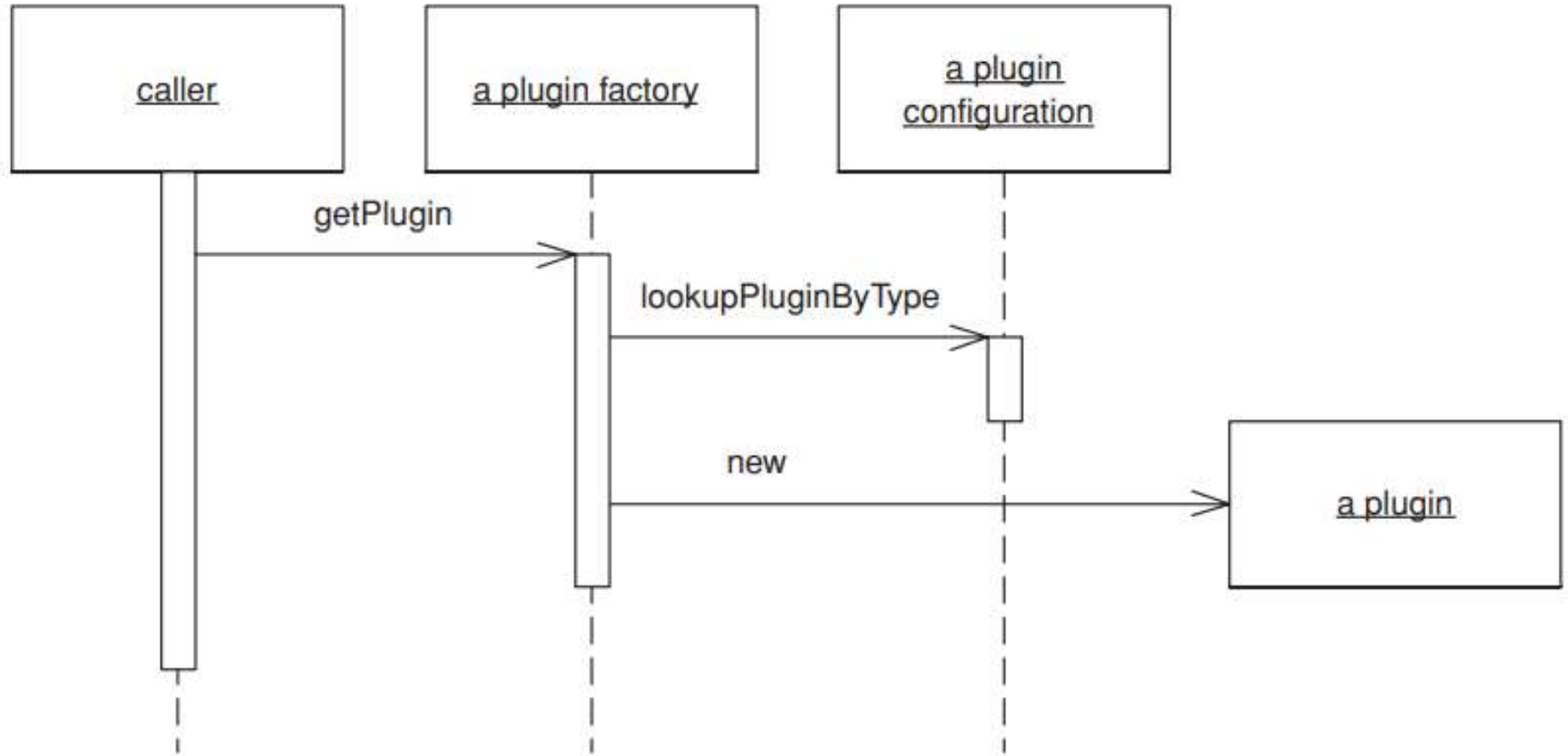


Figure 18.2 *A caller obtains a Plugin implementation of a separated interface.*

Plugin. Пример на Java

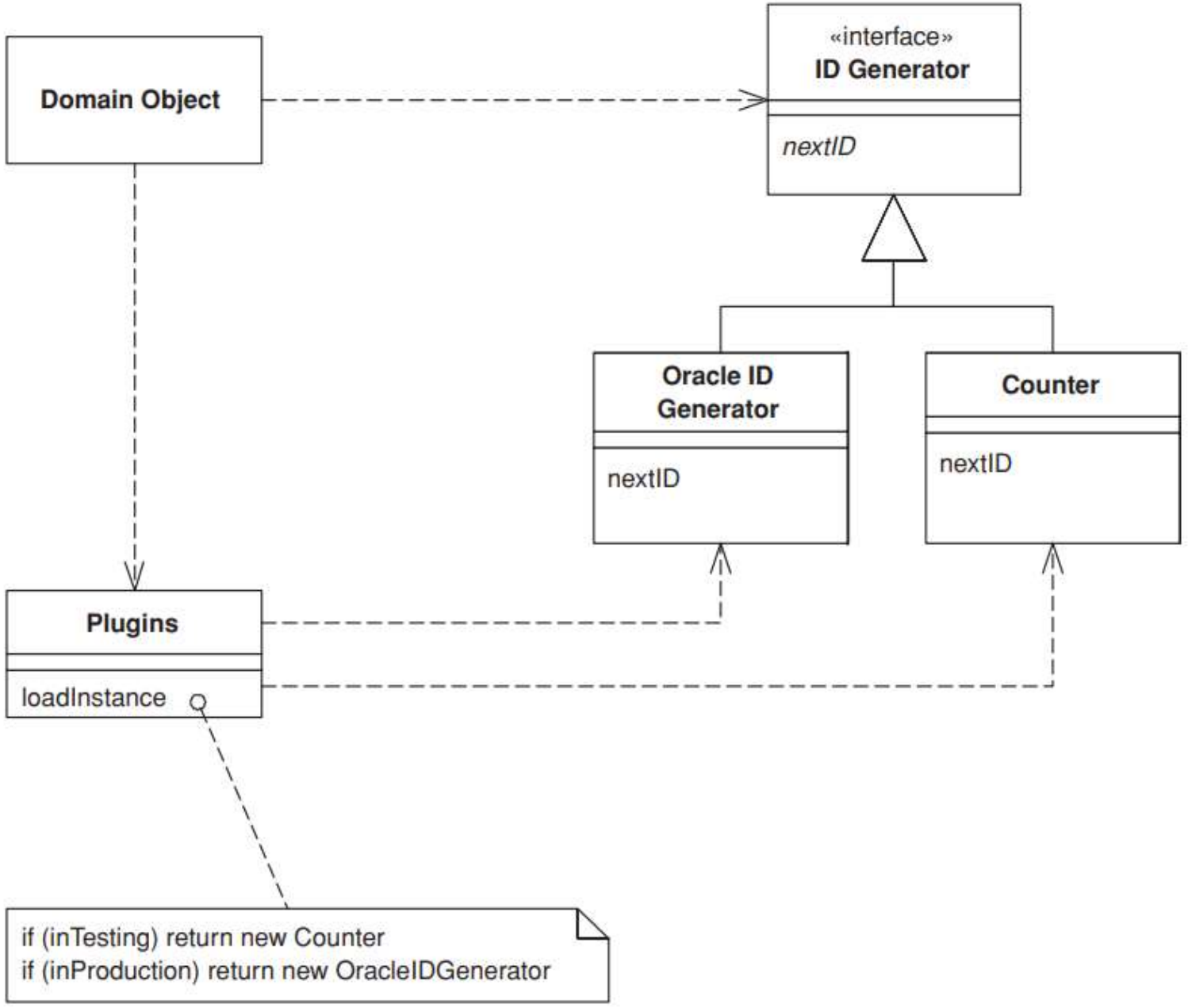


Figure 18.3 *Multiple ID generators.*

Plugin. Пример на Java

```
interface IdGenerator...

   public Long nextId();
```

```
class Counter implements IdGenerator...

   private long count = 0;
   public synchronized Long nextId() {
      return new Long(count++);
   }
```

```
class OracleIdGenerator implements IdGenerator...

   public OracleIdGenerator() {
      this.sequence = Environment.getProperty("id.sequence");
      this.datasource = Environment.getProperty("id.source");
   }
```

Plugin. Пример на Java

```
class PluginFactory...
   private static Properties props = new Properties();

   static {
      try {
         String propsFile = System.getProperty("plugins");
         props.load(new FileInputStream(propsFile));
      } catch (Exception ex) {
         throw new ExceptionInInitializerError(ex);
      }
   }

   public static Object getPlugin(Class iface) {

      String implName = props.getProperty(iface.getName());
      if (implName == null) {
         throw new RuntimeException("implementation not specified for " +
                                    iface.getName() + " in PluginFactory propeties.");
      }
      try {
         return Class.forName(implName).newInstance();
      } catch (Exception ex) {
         throw new RuntimeException("factory unable to construct instance of " +
                                    iface.getName());
      }
   }
```

Plugin. Пример на Java

config file test.properties...

```
# test configuration
IdGenerator=TestIdGenerator
```

config file prod.properties...

```
# production configuration
IdGenerator=OracleIdGenerator
```

Plugin. Пример на Java

```
interface IdGenerator...

   public static final IdGenerator INSTANCE =
      (IdGenerator) PluginFactory.getPlugin(IdGenerator.class);


      class Customer extends DomainObject...

         private Customer(String name, Long id) {
            super(id);
            this.name = name;
         }
         public Customer create(String name) {
            Long newObjId = IdGenerator.INSTANCE.nextId();
            Customer obj = new Customer(name, newObjId);
            obj.markNew();
            return obj;
         }
```

Plugin. Пример на Java

```
interface IdGenerator...

   public static final IdGenerator INSTANCE =
      (IdGenerator) PluginFactory.getPlugin(IdGenerator.class);


      class Customer extends DomainObject...

         private Customer(String name, Long id) {
            super(id);
            this.name = name;
         }
         public Customer create(String name) {
            Long newObjId = IdGenerator.INSTANCE.nextId();
            Customer obj = new Customer(name, newObjId);
            obj.markNew();
            return obj;
         }
```

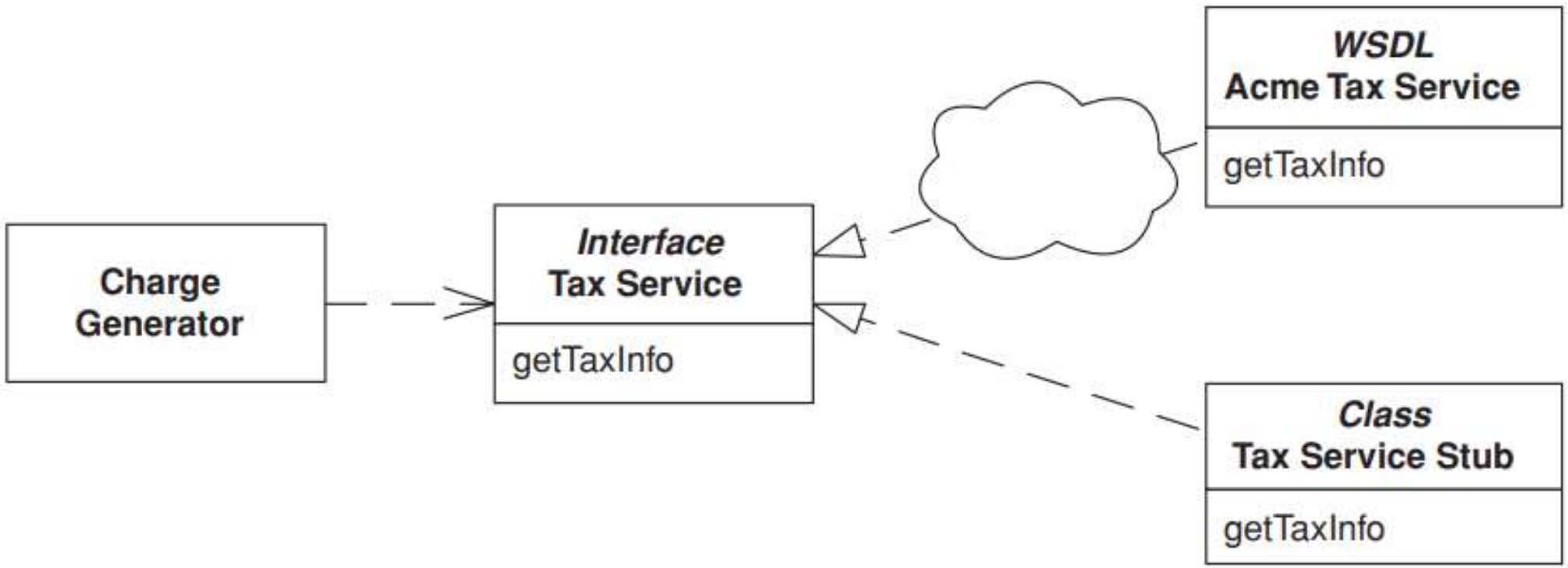
Service Stub

Устраняет зависимость приложения от труднодоступных или проблемных служб на время тестирования

Также можно увидеть под названием *Mock Object*

Service Stub

Removes dependence upon problematic services during testing.



Service Stub

Для организации Service Stub используем *Gateway*, *Separated Interface* и *Plugin*

Отделяем доступ к нашей зависимости через *Gateway*, в нашем пакете *Gateway* представляем в виде *Separated Interface*, требуемую реализацию определяем и подгружаем с помощью *Plugin*

Service Stub. Когда использовать?

Когда выясняете, что зависимость от какого-то конкретного компонента усложняет и замедляет разработку и тестирование. Например, в начале разработки, когда спроектированные сервисы ещё не реализованы

Record Set

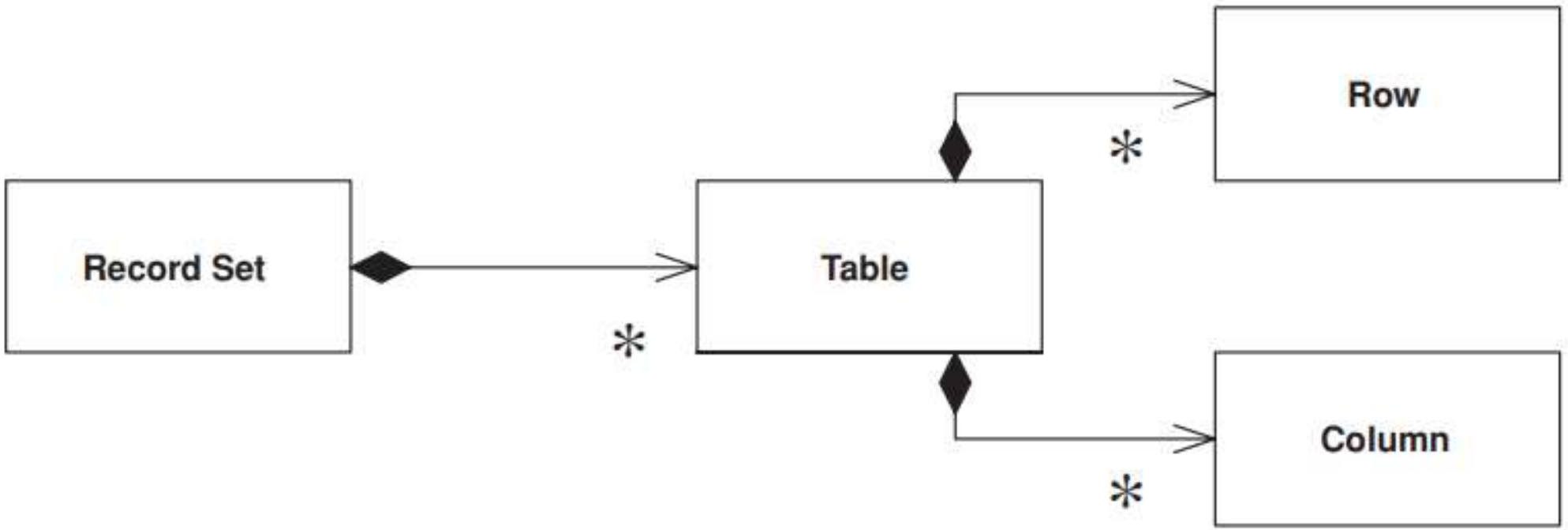
Представление табличных данных (из реляционной БД) в оперативной памяти

Решает проблему вынужденного добавления кода уровня бизнес-логики (валидация данных и т.д.) в уровень представления или в саму базу данных при использовании существующих инструментов, облегчающих интеграцию UI с БД путём имитирования БД

Обычно предоставляется фреймворками (data set в ADO.NET, row set в JDBC 2.0)

Record Set

An in-memory representation of tabular data.



Record Set

Record Set может предоставлять **explicit** или **implicit** интерфейс

- Explicit – `aReservation.passenger`
- Implicit – `aReservation["passenger"]`

Record Set. Когда использовать?

- Работа с фреймворком, ориентированным на реляционные БД
- Table Module (рассмотрим позднее)