Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Пермская государственная сельскохозяйственная академия
имени академика Д.Н. Прянишникова»

ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

направление 230700 «Прикладная информатика»

ЛАБОРАТОРНОЕ ЗАНЯТИЕ № 9

Тема: **МОДЕЛЬ РЕАЛИЗАЦИИ: СОЗДАНИЕ ПРИЛОЖЕНИЯ**

Учебные вопросы:

1. Программирование и процесс разработки.
2. Преобразование результатов проектирования в программный код.
3. Подключение уровня интерфейса пользователя к уровню предметной области

Вопрос 1. Программирование и процесс разработки

Выбор языка программирования

Рассматриваемые примеры написаны на языке Java. Такой выбор объясняется популярностью и широким распространением этого языка. Однако автор не выделяет язык Java среди других языков. Языки C#, Visual Basic, C++, Smalltalk, Python и многие другие тоже удовлетворяют принципам объектного проектирования и могут использоваться для преобразования разработанной модели в исходный код.

Выполнение предварительного проектирования совсем не означает, что в процессе программирования нельзя выполнять макетирование и проектирование. Современные средства разработки предоставляют прекрасную среду для быстрого изучения альтернативных подходов, а некоторые из них (или даже многие) позволяют сочетать процессы проектирования и программирования.

Однако некоторые разработчики считают, что до начала этапа программирования желательно разработать базовые визуальные модели. Это особенно полезно тем разработчикам, которые обладают "визуальным стилем мышления" и лучше воспринимают информацию, представленную в виде диаграмм.

Написание кода на объектно-ориентированном языке программирования, наподобие Java или C#, не относится к процессу анализа или проектирования системы – **это конечная цель проектирования**. Артефакты, создаваемые в контексте RUP в рамках модели проектирования, предоставляют часть информации, необходимой для генерирования кода.

Преимущество объектно-ориентированного подхода к анализу, проектированию и программированию в рамках RUP состоит в том, что он обеспечивает полный цикл разработки системы – от формулировки требований до программной реализации. Артефакты последовательно трансформируются в артефакты следующей стадии разработки, постепенно обеспечивая превращение системы в работающее приложение.

Внесение изменений на стадии реализации

Значительная часть усилий и творческого потенциала была задействована на стадии проектирования. Как станет видно из последующего обсуждения, генерация программного кода является относительно механическим процессом преобразования.

Тем не менее, процесс программирования не сводится к примитивной генерации кода. Совсем наоборот: результаты, полученные на стадии проектирования, оказываются далеко не совершенными. В процессе программирования и тестирования наверняка потребуется внести многочисленные изменения, а также выявить и разрешить возникшие проблемы.

Артефакты проектирования будут составлять эластичное ядро, которое можно масштабировать, сохраняя при этом изящность и устойчивость, а также обеспечивая решение новых возникающих в процессе программирования проблем. Поэтому на стадии построения и тестирования будьте готовы к изменению проектных решений.

Модификация кода и итеративный процесс

Преимущество итеративного и инкрементального процесса разработки заключается в том, что результаты предыдущей итерации могут служить начальными данными для последующей итерации (рис.1.1). Таким образом, результаты последовательного анализа и проектирования непрерывно совершенствуются на последующих итерациях разработки. Например, если код, созданный на итерации N, отклоняется от результатов проектирования этой итерации (что почти наверняка произойдет), то проектные решения, основанные на этой реализации, могут использоваться в качестве входных данных для моделей анализа и проектирования итерации N+1.

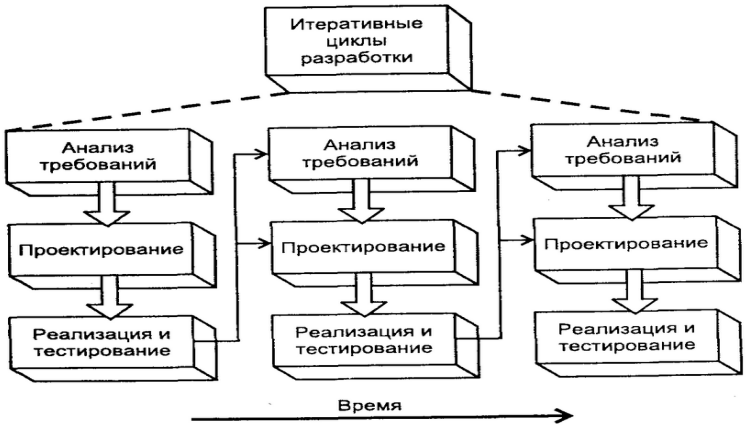

Рисунок 1.1 – Влияние программной реализации на процесс проектирования последующих итераций

Именно поэтому одним из самых ранних видов деятельности на каждой итерации разработки является синхронизация артефактов. Диаграммы итерации N могут не соответствовать результирующему программному коду итерации N. Прежде чем переходить к новому этапу анализа и проектирования, нужно выполнить синхронизацию.

Изменение кода, CASE-средства и обратное проектирование

Очень желательно, чтобы диаграммы, полученные на стадии проектирования, были полуавтоматически обновлены и отражали изменения, внесенные на соответствующей стадии кодирования. В идеале это делается с использованием CASE-средств (например, CaseBerry), с помощью которых можно считывать исходный код и автоматически генерировать, например, диаграммы пакетов, классов и последовательностей. Это пример *обратного проектирования* (reverse engineering), когда логические модели генерируются на основе исходного (или даже исполняемого) кода.

Вопрос 2. Преобразование результатов проектирования в программный код

Реализация на объектно-ориентированном языке программирования требует написания исходного кода для:

- **определений классов и интерфейсов;**
- **методов.**

В последующих подпунктах обсуждается генерация таких определений на языке Java (как типичный случай).

Создание определений классов на основе диаграмм классов

Коротко можно сказать, что на диаграммах классов отображаются имена классов и интерфейсов, суперклассов, сигнатуры методов и простые атрибуты классов. Этого вполне достаточно для создания базового определения класса на объектно-ориентированном языке программирования. Ниже будет рассмотрен процесс добавления информации об интерфейсе и пространстве имен (или пакете), а также других данных.

Определение класса с методами и простыми атрибутами

Как видно из рис. 2.1, преобразование диаграммы классов в базовые определения атрибутов (простые экземпляры переменных-членов на Java) и сигнатуры методов для определения класса SalesLineItem на языке Java выполняется очень просто.

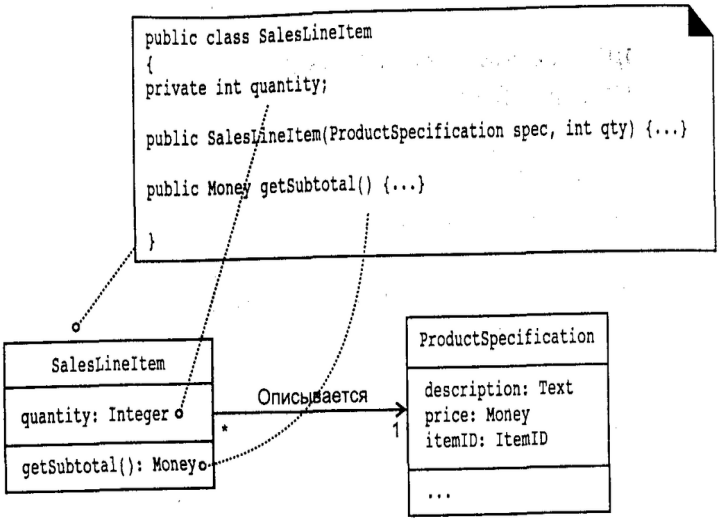

Рисунок 2.1 – Класс SalesLineItem на языке Java

Обратите внимание на добавление конструктора SalesLineItem {. . .}. Это сделано на основании того, что в диаграмме взаимодействия для системной операции enterItem объекту SalesLineItem передается сообщение create (spec, qty). При переходе к языку Java это означает необходимость использования конструктора с двумя указанными параметрами. Метод create зачастую исключается из диаграммы классов, поскольку он является стандартным и имеет несколько интерпретаций, зависящих от используемого языка программирования.

Добавление атрибутов-ссылок

Атрибут-ссылка (reference attribute) – это атрибут, ссылающийся на другой сложный объект, а не на простой тип, такой как String, Number и т.д. (На диаграмме классов атрибуты-ссылки представлены ассоциациями и связанным с ними направлением перемещения).

Например, класс SalesLineItem имеет ассоциацию, направленную к классу productSpecification. Обычно эта ассоциация интерпретируется как атрибут-ссылка класса SalesLineItem, ссылающаяся на экземпляр ProductSpecification (рис. 2.2).

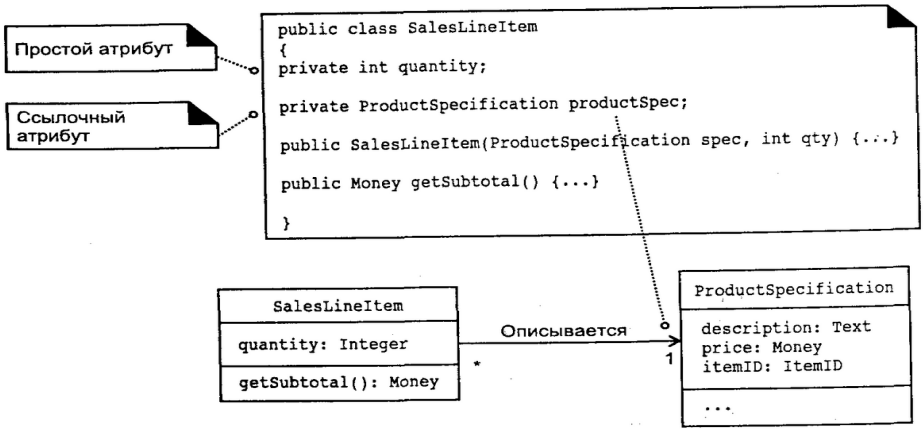

Рисунок 2.2 – Добавление атрибутов ссылок

На языке Java это означает добавление поля, ссылающегося на экземпляр ProductSpecification.

(Обратите внимание, что атрибуты-ссылки классов зачастую косвенно присутствуют, а не явно определяются на диаграмме классов).

Например, хотя в определение класса SalesLineItem на языке Java был добавлен экземпляр переменной, указывающей на экземпляр ProductSpecification, на диаграмме классов в разделе атрибутов явно объявленный атрибут отсутствует. Это объясняется предполагаемой видимостью атрибута, задаваемой ассоциацией и ее направлением. На стадии генерации кода эта ассоциация явно определяется как атрибут.

Атрибуты-ссылки и имена ролей

Рассмотрим имена ролей на статической структурной диаграмме. Каждый конец линии ассоциации называется ролью. Если говорить кратко, *имя роли* (role name) – это имя,

идентифицирующее роль и обеспечивающее некоторый семантический контекст, которой иллюстрирует его природу.

Если имя роли присутствует на диаграмме классов, то при генерации кода его нужно использовать в качестве основы имени атрибута-ссылки (рис. 2.3).

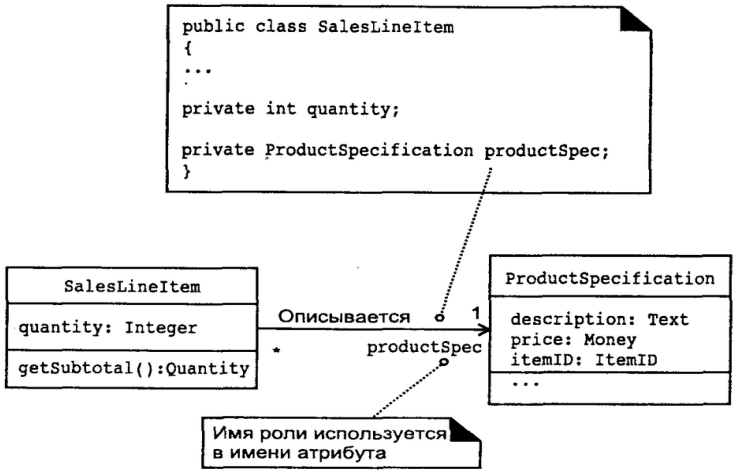

Рисунок 2.3 – Имена ролей можно использовать для генерации имен экземпляров переменных

Отображение атрибутов

На примере класса Sale можно удостовериться в неоднозначности отображения атрибутов диаграммы проектирования в исходный код. На рис. 2.4 показаны возможные проблемы, связанные с таким преобразованием.

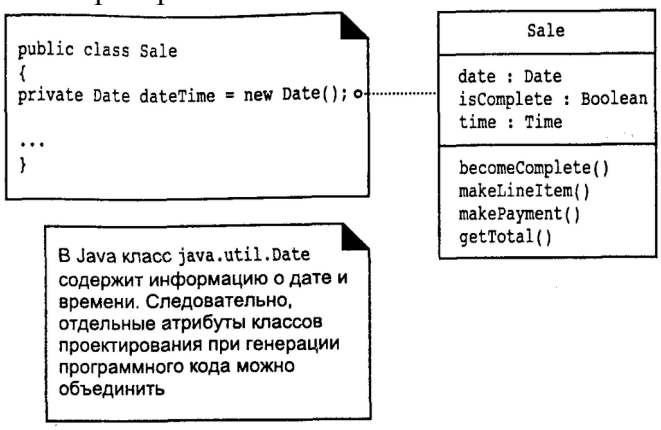

Рисунок 2.4 – Преобразование в исходный код на языке Java атрибутов даты и времени

Создание методов на основе диаграмм взаимодействия

На диаграммах взаимодействия представлены сообщения, которые передаются в ответ на вызов метода. Последовательность этих сообщений преобразуется в серию операторов в определении метода. Для иллюстрации определения метода enterItem на языке Java можно использовать диаграмму взаимодействия системной операции enterItem (рис. 2.5).

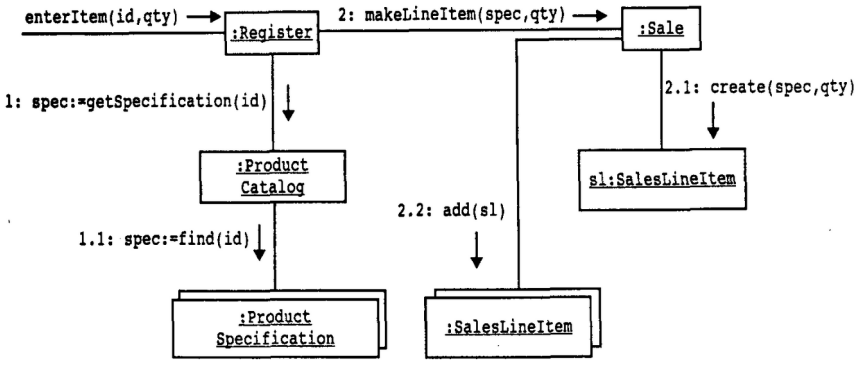

Рисунок 2.5 – Диаграмма взаимодействия для системной операции enterItem

В приведенном примере будет использован класс Register. Его определение на языке Java представлено на рис. 2.6.

Метод Register- -enterItem

Сообщение enterItem передается экземпляру объекта Register. Следовательно, метод enterItem определяется в этом классе.

```
public void enterItem(ItemID itemID, int qty)
```

Сообщение 1. Для получения объекта ProductSpecification объекту ProductCatalog передается сообщение getSpecification.

```
ProductSpecification spec =
        catalog.getSpecification(itemID);
```

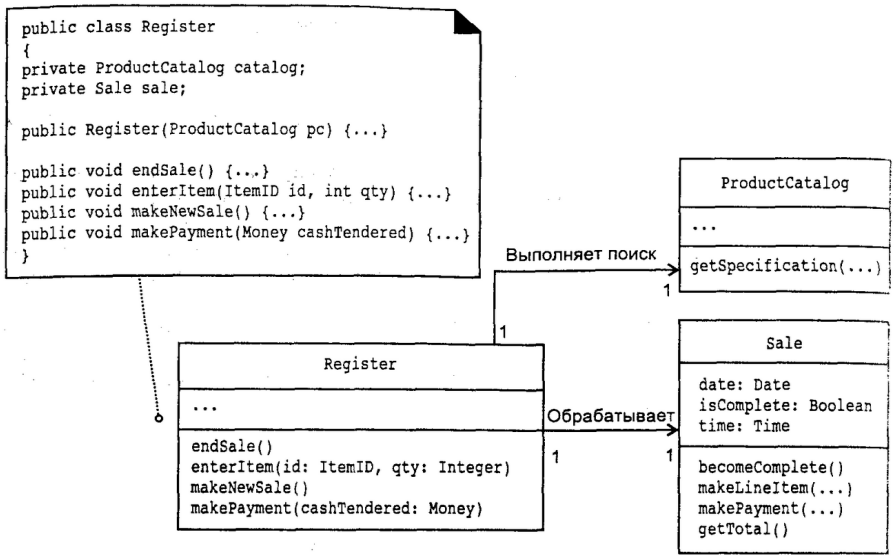

Рисунок 2.6 – Класс Register

Сообщение 2. Объекту Sale передается сообщение sale.makeLineItem(spec, qty).

Подводя итоги, можно еще раз повторить, что каждое сообщение из последовательности внутри метода, как показано на диаграмме взаимодействия, преобразуется в оператор метода на языке Java.

Полученный метод enterItem и его связь с диаграммой взаимодействия представлены на рис. 2.7.

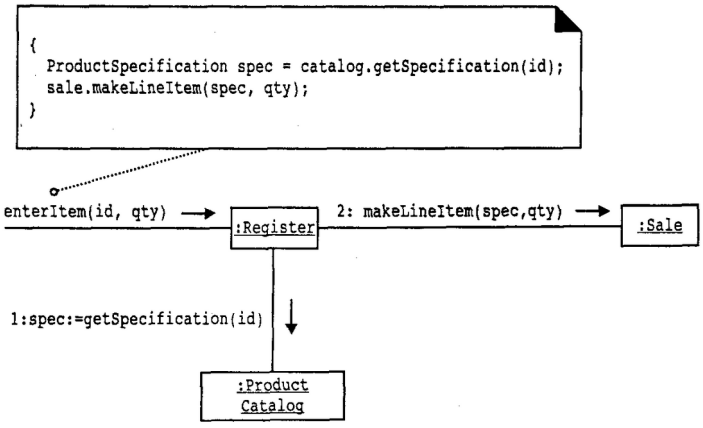

Рисунок 2.7 – Метод enterItem

Классы-контейнеры в программном коде

Зачастую для объекта необходимо обеспечить видимость группы других объектов. Обычно это следует непосредственно из значения кратности, указанного на диаграмме классов. Оно может превышать единицу. Например, для объекта Sale необходимо обеспечить видимость группы экземпляров класса SalesLineItem (рис. 2.8).

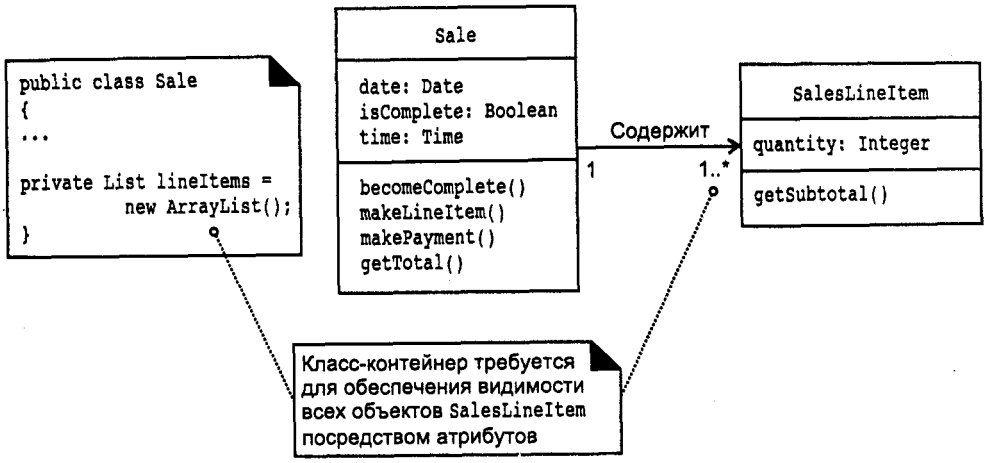

Рисунок 2.8 – Добавление контейнера

В объектно-ориентированных языках программирования такие взаимосвязи в основном реализуются с помощью промежуточного контейнера или коллекции объектов. В классе, с которым связано единичное значение кратности, определяется атрибут-ссылка, содержащий указатель на экземпляр контейнера/коллекции. А в самом контейнере содержатся экземпляры класса, для которого значение кратности превышает единицу.

Например, в библиотеках Java имеются такие контейнерные классы, как ArrayList и HashMap, реализующие интерфейсы List и Map, соответственно. При использовании класса ArrayList в классе Sale можно определить атрибут, указывающий на упорядоченный список экземпляров SalesLineItem.

Выбор подходящего контейнерного класса определяется реальными требованиями. Если планируется применять поиск по ключу, то лучше воспользоваться классом Map; при организации расширяемого упорядоченного списка потребуется класс List и т.д.

Исключения и обработка ошибок

До сих пор обработка ошибок в процессе разработки игнорировалась, поскольку мы рассматривали основные вопросы распределения обязанностей и объектно-ориентированного проектирования. Однако при разработке реального приложения на стадии проектирования, а особенно реализации, не лишним будет рассмотреть также вопросы обработки ошибок.

В языке UML для обозначения исключений используются асинхронные сообщения на диаграммах взаимодействия.

Метод Sale- -makeLineItem

В качестве заключительного примера рассмотрим метод makeLineItem класса Sale, который может быть описан на основе анализа диаграммы взаимодействия системной операции enterItem. Фрагмент диаграммы взаимодействия и соответствующий метод на языке Java представлены на рис. 2.9.

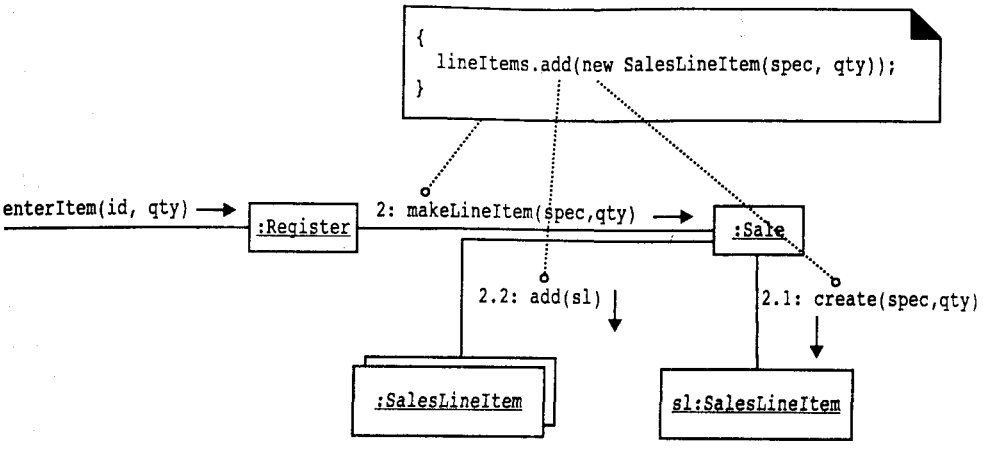

Рисунок 2.9 – Метод Sale-makeLineItem

Вопрос 3. Подключение уровня интерфейса пользователя к уровню предметной области

Как отмечалось выше, приложения делятся на логические уровни, представляющие различные аспекты функционирования приложения, в том числе уровень графического интерфейса пользователя и уровень объектов предметной области (для реализации логики предметной области).

Основные проектные решения, обеспечивающие видимость объектов уровня Предметной области для объектов уровня пользовательского интерфейса, сводятся к следующему.

- Инициализирующая программа (например, метод main Java) создает и объекты уровня пользовательского интерфейса, и объект предметной области, а затем передает этот объект на уровень интерфейса пользователя.
- Объект пользовательского интерфейса получает объект предметной области из стандартного источника, например, от объекта-фабрики, отвечающего за создание объектов предметной области.

Следующий фрагмент кода демонстрирует пример использования первого подхода.

```
public class Main
{
public static void main(String[] args)
{
   Store store = new Store();
   Register register = store.getRegister();
   ProcessSaleJFrame frame = new ProcessSaleJFrame(register);
   ...
}
}
```

Будучи связанным с экземпляром объекта Register (выступающим в данном приложении в роли внешнего контроллера), объект уровня пользовательского интерфейса может направить ему сообщение о системном событии, такое как enterItem или endSale (рис. 3.1).

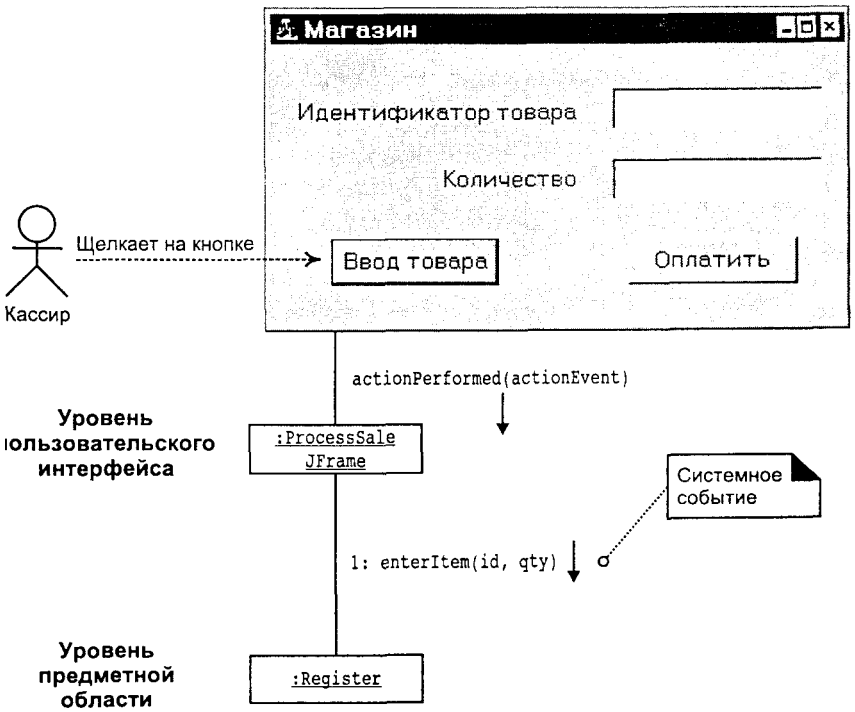

Рисунок 3.1 – Связь уровней предметной области и пользовательского интерфейса

Для передачи сообщения enterItem необходимо иметь диалоговое окно, в котором будет отображаться общая стоимость каждого товара. При этом можно использовать несколько проектных решений:

- Добавить объекту Register метод getTotal. Тогда с уровня пользовательского интерфейса

объекту Register будет передаваться сообщение getTotal, а этот объект, в свою очередь, перенаправит это сообщение объекту Sale. Преимуществом такого подхода является низкая степень связывания между уровнями пользовательского интерфейса и предметной области – пользовательскому интерфейсу известен лишь один объект Register. Однако при этом "раздувается" интерфейс объекта Register, что приводит к слабой степени зацепления этого объекта.

- Интерфейс пользователя запрашивает ссылку на текущий объект Sale, а затем при необходимости вычисления общей стоимости (или для получения любой другой информации о продаже) он передает сообщение напрямую объекту Sale. Такое проектное решение повышает степень связывания между уровнями пользовательского интерфейса и предметной области. Однако, как указывалось при рассмотрении шаблона GRASP Low Coupling, высокая степень связывания сама по себе не является проблемой. Проблема возникает при связывании с неустойчивыми объектами. Объект Sale можно считать устойчивым, поскольку он является неотъемлемой частью проектного решения. Следовательно, связывание с этим объектом не составляет проблемы.

Как видно из рис. 3.2, в проектном решении для данного приложения выбран второй подход.

Заметим, что, согласно этим диаграммам, окно Java (ProcessSaleJFrame), составляющее часть уровня пользовательского интерфейса, не отвечает за реализацию логики приложения. Оно лишь направляет запросы для выполнения системных операций объектам уровня предметной области через объект Register. Отсюда следует один из важных принципов проектирования.

Обязанности объектов уровней пользовательского интерфейса и предметной области: уровень пользовательского интерфейса не должен отвечать за реализацию логики приложения. Его обязанностью является лишь выполнение задач пользовательского интерфейса, например, обновление информации в диалоговых окнах.

Объекты уровня пользовательского интерфейса должны направлять запросы на выполнение всех задач предметной области на уровень объектов предметной области, который и отвечает за их выполнение.

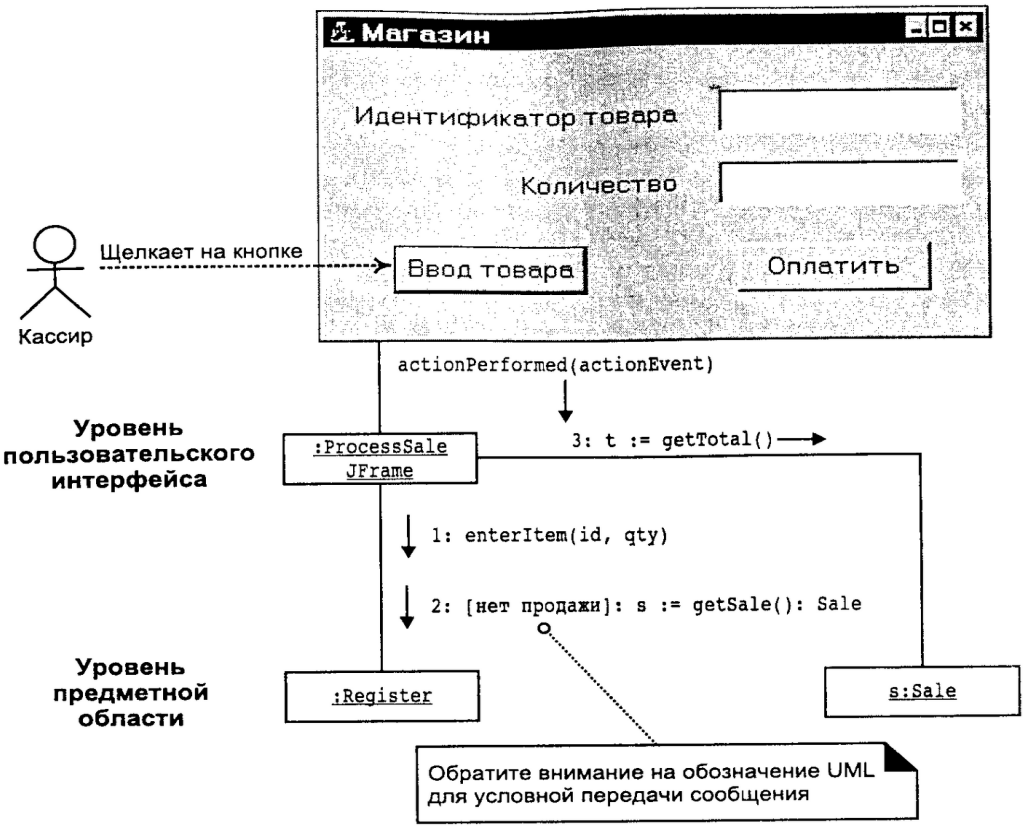

Рисунок 3.2 – Взаимодействие уровней пользовательского интерфейса и предметной области