Объектно-ориентированное проектирование

RDD и GRASP на основе книги К. Лармана

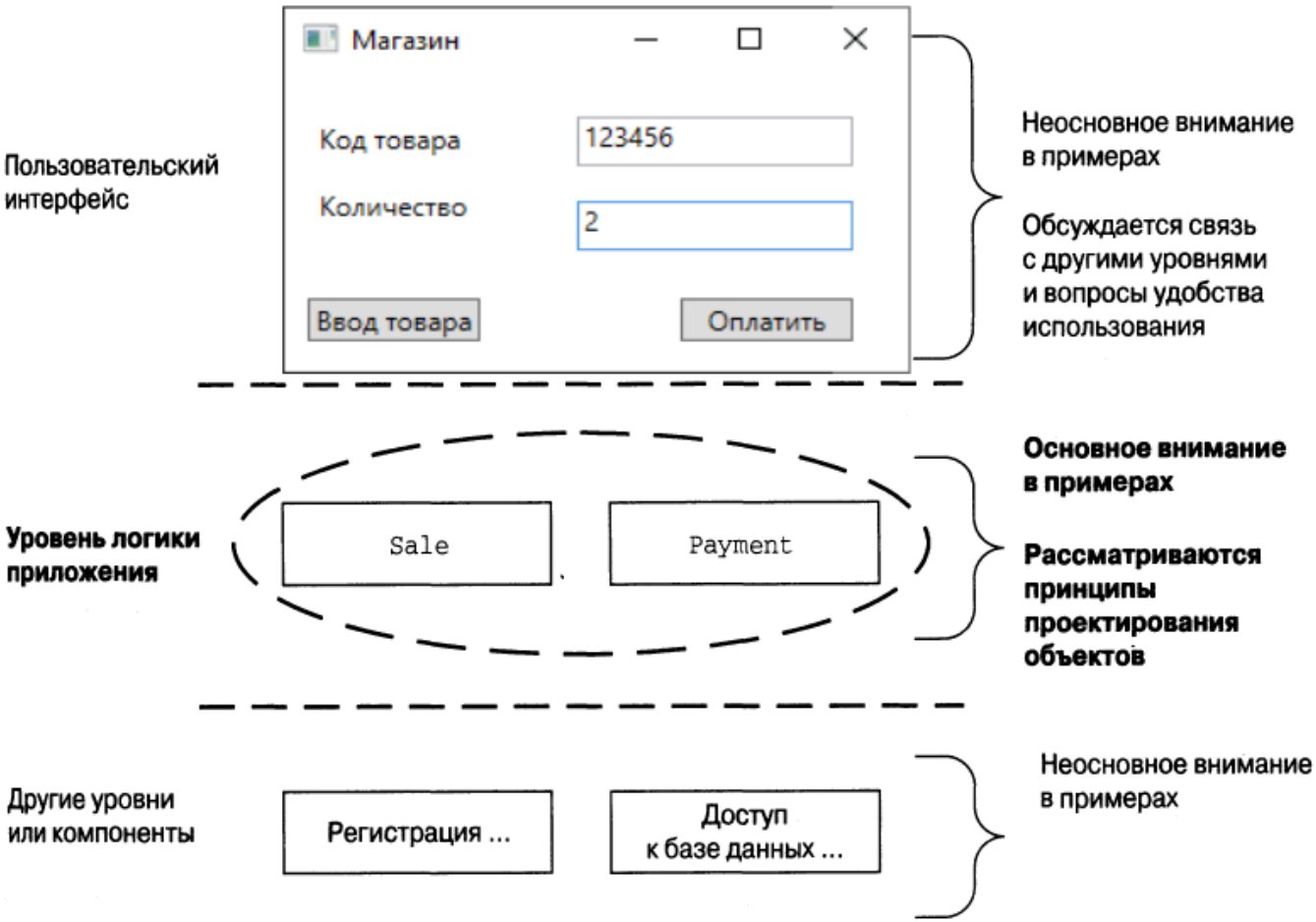
Объектно-ориентированное проектирование

- Процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы или ее компонентов называется **проектированием**
- Анализ -> **Проектирование** -> Программирование.
- Результат процесса проектирования – **дизайн**.
  - должен описывать архитектуру программного обеспечения, то есть представлять декомпозицию программной системы в виде организованной структуры компонент и интерфейсов между компонентами.
  - Важнейшей характеристикой готовности дизайна является тот уровень детализации компонентов, который позволяет заняться их конструированием.
  - Термины **дизайн** и **архитектура** могут использоваться взаимозаменяемым образом, но чаще говорят о дизайне как о целостном взгляде на архитектуру системы.

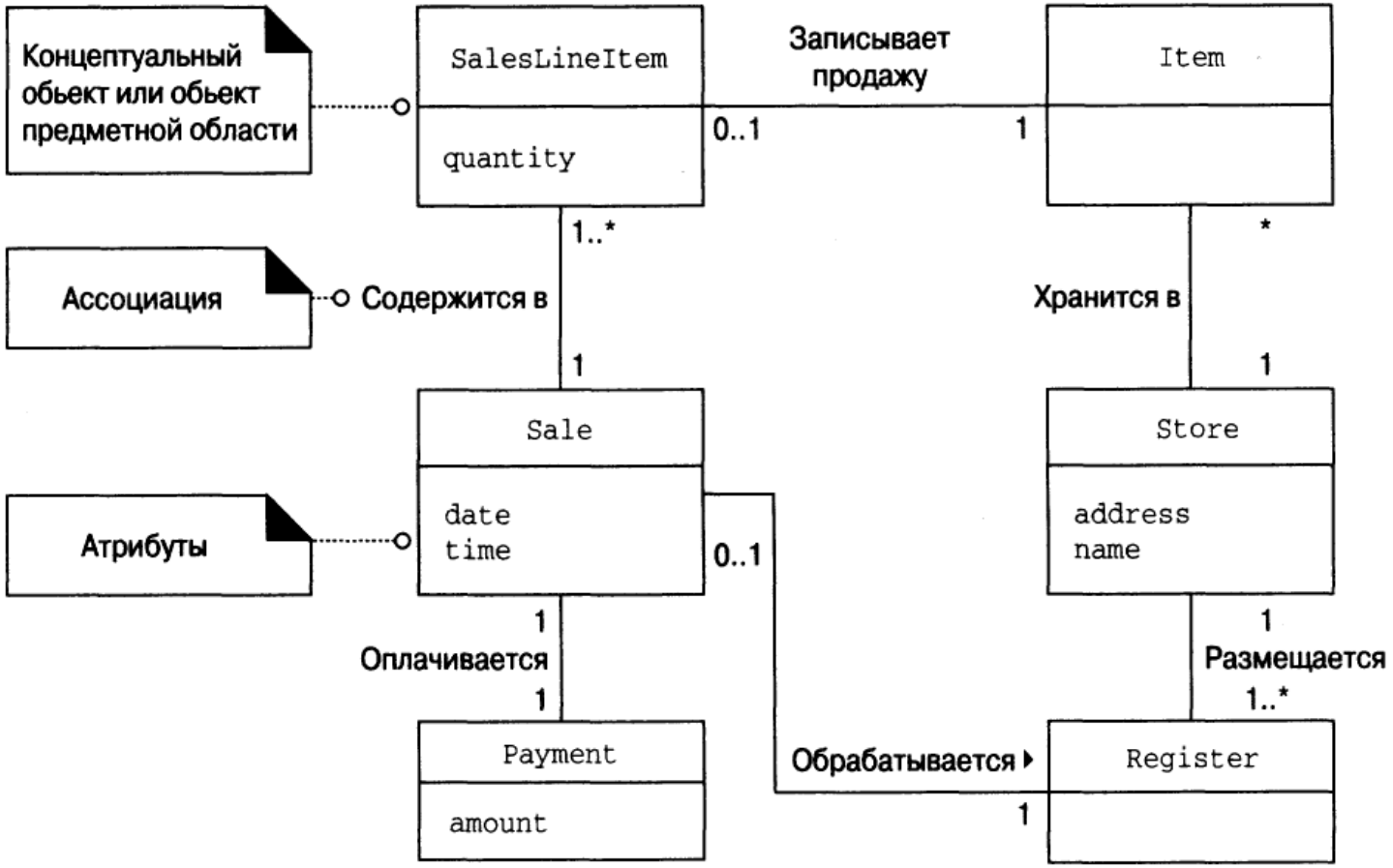
Пример: POS-система

- Первым примером, рассматриваемым в этой книге, является система автоматизации торговой точки — **POS-система NextGen**.
- POS-система — это компьютеризированное приложение, предназначенное для организации **товарооборота и обработки платежей** в обычных магазинах.
- Система автоматизации торговли включает **аппаратные компоненты** (компьютер и устройство считывания штрихкода), а также **программное обеспечение**, выполняющее основные задачи системы. Это приложение **связано с различными служебными программами**, например с программой вычисления налогов, разработанной сторонними производителями, или с системой складского учета товаров.
- Подобные системы должны быть **устойчивыми к сбоям**, т.е. работоспособными при временном выходе из строя удаленных служб (например, системы складского учета товаров). В критических ситуациях они должны обслуживать продажу товаров и обеспечивать обработку хотя бы платежей наличными (чтобы хозяин магазина не обанкротился).
- POS-система должна поддерживать **различные типы клиентских терминалов и интерфейсов**, в том числе клиентский терминал с Web-браузером (“тонкого” клиента), обычный персональный компьютер с графическим интерфейсом пользователя, сенсорный ввод информации, беспроводный интерфейс и т.п.
- Более того, коммерческие POS-системы должны уметь работать с **различными категориями потребителей**, для которых определены **отдельные бизнес-правила**. Для каждого потребителя может быть предусмотрена своя логика выполнения отдельных операций в рамках сценария использования системы, например, специфические действия при добавлении нового товара или создании новой продажи. Следовательно, необходимо предусмотреть механизм обеспечения этой гибкости и настройки системы.

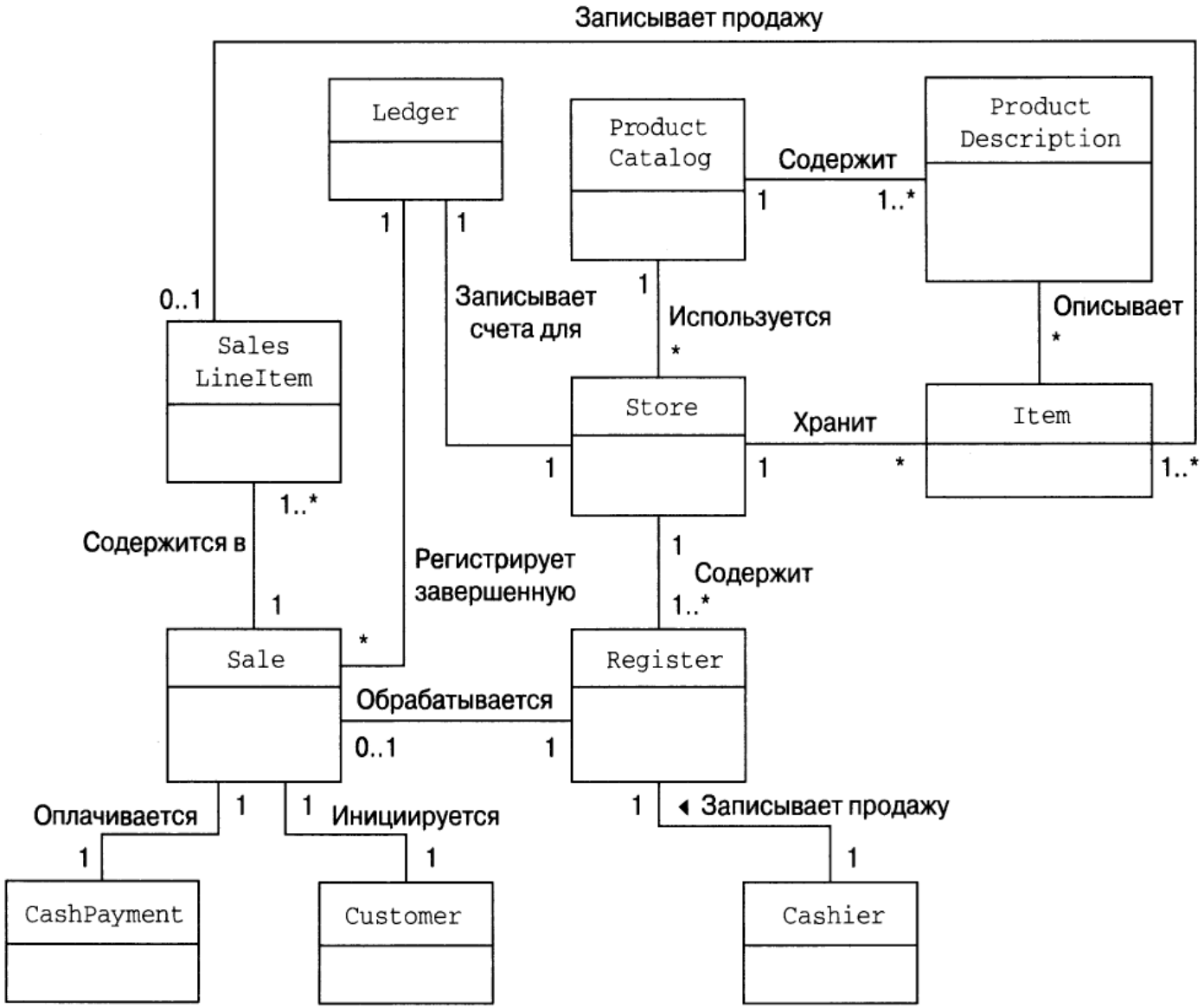
Локализация рассматриваемых в этом курсе подходов



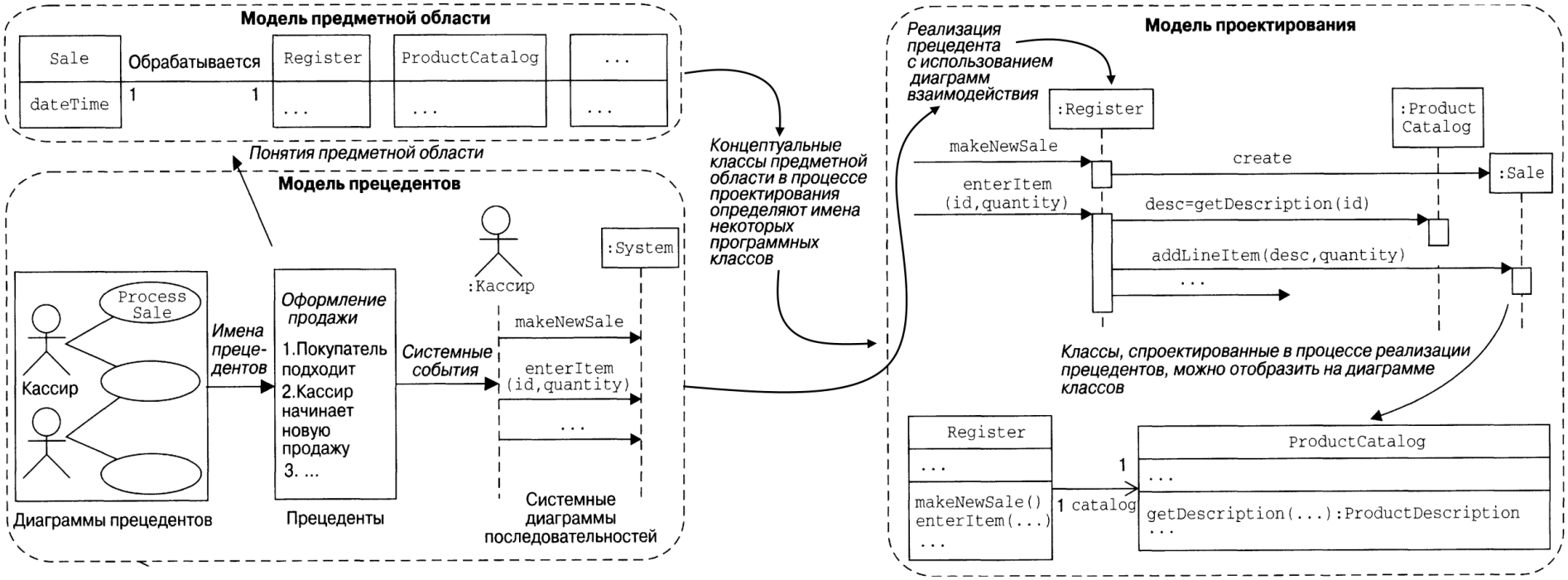
Фрагмент модели предметной области



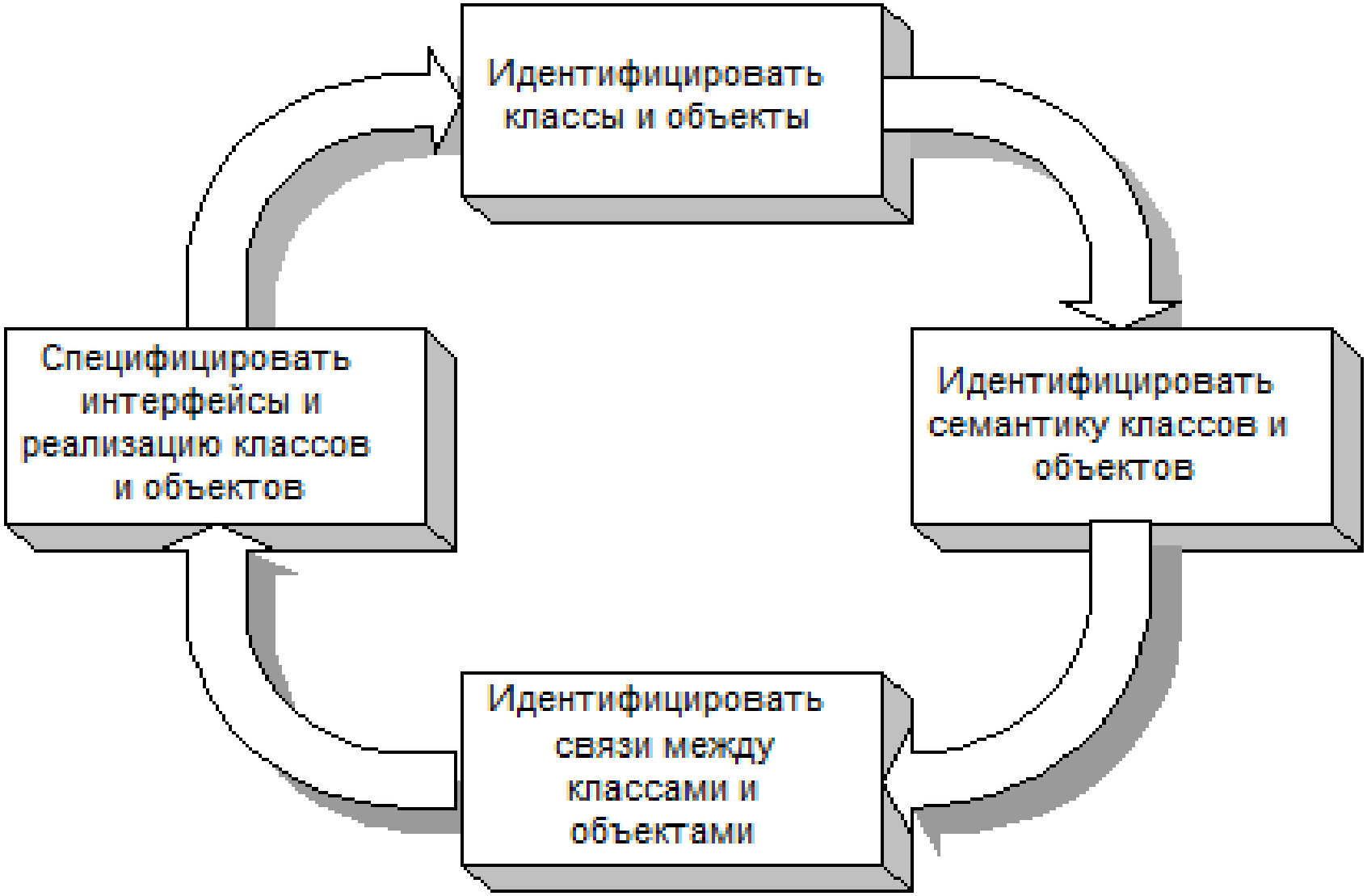
Фрагмент модели предметной области



Взаимосвязи между элементами модели и процесс проектирования



Микропроцесс проектирования



Проектирование на основе обязанностей (RDD)

- Данный подход подразумевает описание проектирования программных объектов и крупномасштабных компонентов в терминах **обязанностей, ролей и кооперации**.
- В RDD считается, что программные объекты имеют определенные **обязанности** —абстракции, реализуемые ими.
  - **Обязанность** может определяться как **контракт** или **обязательство** класса.
  - **Обязанности** описывают поведение объекта в терминах его **ролей**.
- В общем случае можно выделить два типа обязанностей:
  - **знание**
    - Наличие информации о закрытых инкапсулированных данных.
    - Наличие информации о связанных объектах.
    - Наличие информации о следствиях или вычисляемых величинах.
  - **действие**
    - выполнение некоторых действий самим объектом, например создание экземпляра или выполнение вычислений.
    - Инициирование действий других объектов.
    - Управление действиями других объектов и их координирование.

Проектирование на основе обязанностей (RDD)

- **Обязанности присваиваются объектам** в процессе ОО-проектирования.
  - Например, можно сказать, что объект **Sale** отвечает за создание экземпляра **SalesLineItems** (действие) или что объект **Sale** отвечает за наличие информации о **стоимости** покупки (знание).
  - Обязанности, относящиеся к разряду “знаний”, зачастую вытекают из **модели предметной области**, поскольку она иллюстрирует **атрибуты** и **ассоциации**.
  - Например, если в модели предметной области класс **Sale** содержит атрибут **time**, то с целью уменьшения разрыва в представлениях программный класс **Sale** тоже должен “знать” **время** соответствующей продажи.
- **Реализация сложных обязанностей** требует определения **множества** классов и методов. Для простых обязанностей достаточно одного метода.
  - Например, реализация обязанности “обеспечения доступа к реляционным базам данных” может потребовать создания десятков классов и сотен методов, а для реализации обязанности “создания экземпляра объекта Sale” достаточно одного метода одного класса.
  - Между методами и обязанностями нельзя ставить знак равенства, однако можно утверждать, что реализация метода обеспечивает выполнение обязанностей.

Проектирование на основе обязанностей (RDD)

- В RDD существует идея **кооперации**.
  - **Обязанности** реализуются посредством **методов**, действующих либо отдельно, либо во взаимодействии с другими методами и объектами.
  - Например, для класса **Sale** можно определить один или несколько методов **вычисления стоимости** (скажем, метод **getTotal**). Для выполнения этой обязанности объект **Sale** должен взаимодействовать с другими объектами, в том числе передавать сообщения **getSubtotal** каждому объекту **SalesLineItem** о необходимости предоставления соответствующей информации этими объектами.
- **Основной принцип RDD**
  - RDD — это общий подход к проектированию программных объектов. Программные объекты рассматриваются **как люди**, имеющие свои **обязанности** и **сотрудничающие с другими людьми** для их **выполнения**.
  - Согласно основному принципу RDD, **объектное проектное решение** представляет собой **сообщество взаимодействующих объектов**, имеющих свои **обязанности**.

General **R**esponsibility **A**ssignment **S**oftware **P**rinciples/**P**atterns

- **Общие Принципы/Шаблоны Распределения Обязанностей** — принципы, используемые в объектно-ориентированном проектировании для решения общих задач по назначению обязанностей классам и объектам.
- Принципы (шаблоны) GRASP:
  - **Базовые**
    - **Information Expert** (Информационный эксперт)
    - **Creator** (Создатель)
    - **Controller** (Контроллер)
    - **Low Coupling** (Слабое зацепление)
    - **High Cohesion** (Сильная связность)
  - **Продвинутые**
    - **Protected Variations** (Сокрытие реализации)
    - **Polymorphism** (Полиморфизм)
    - **Pure Fabrication** (Чистая выдумка)
    - **Indirection** (Посредник)

Шаблоны

- В объектно-ориентированной технологии проектирования **шаблоном** называют именованное описание проблемы и ее решения, которые можно применить при разработке других систем.
  - В идеале шаблон должен содержать советы по поводу его применения в различных ситуациях, а также описание его преимуществ и недостатков
  - Многие шаблоны содержат рекомендации по распределению обязанностей между объектами с учетом специфики задачи.
- **Шаблоны имеют осмысленные имена**, например Information Expert или Factory.
  - Позволяют зафиксировать понятие в памяти
  - Облегчают общение
- **Шаблоны не содержат новых идей**, а лишь формулируют широко используемые базовые принципы.

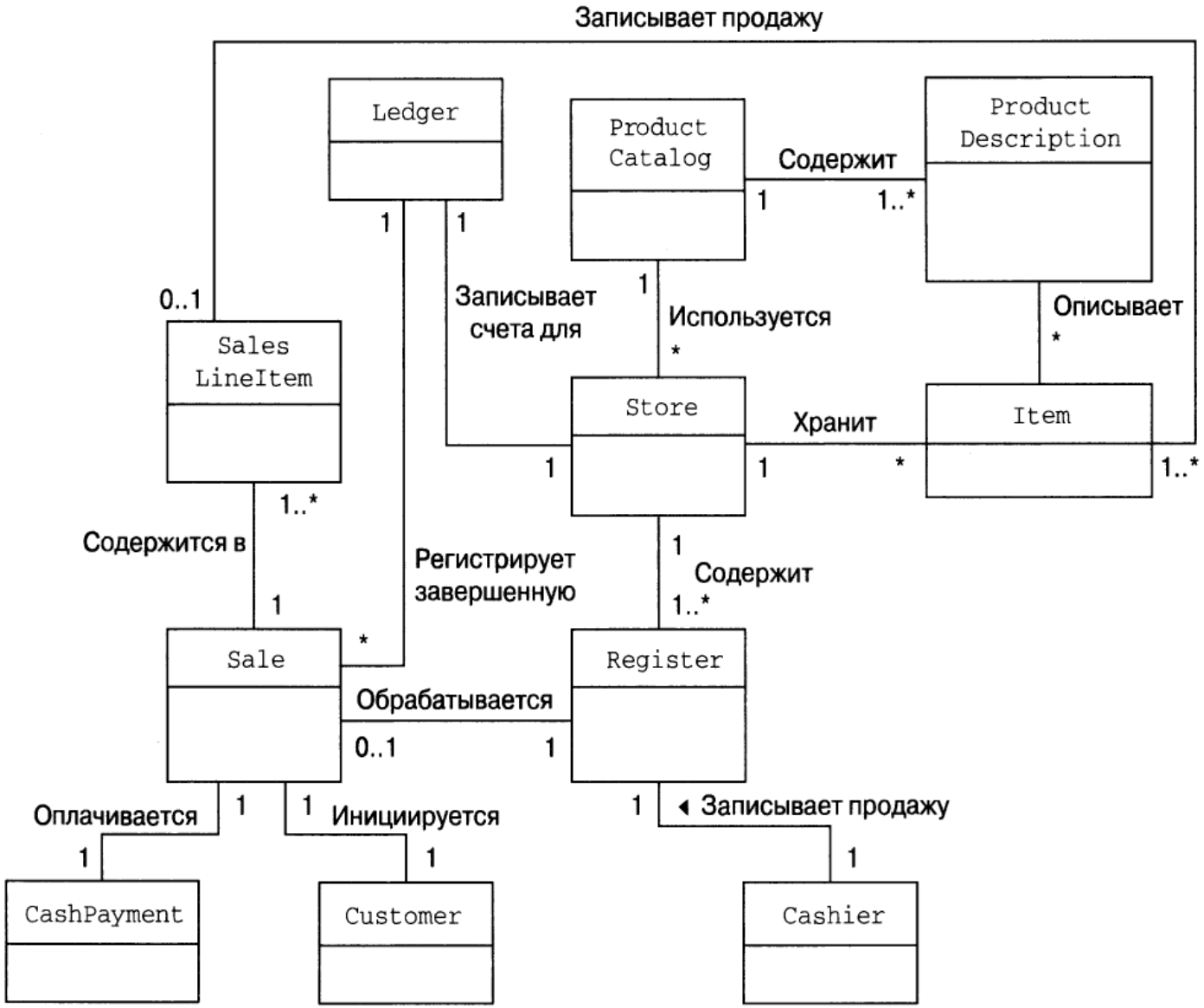
GRASP: Information Expert (Информационный эксперт)

- Шаблон Information Expert определяет базовый принцип назначения обязанностей. Он утверждает, что **обязанности должны быть назначены объекту, который владеет максимумом необходимой информации для выполнения обязанности.** Такой объект называется информационным экспертом.
- Возможно, этот шаблон является самым очевидным из девяти, но вместе с тем и самым важным.
- Если дизайн не удовлетворяет этому принципу, то при программировании получается **спагетти-код**, в котором очень трудно разбираться. Локализация обязанностей позволяет повысить уровень инкапсуляции и уменьшить уровень зацепления. Кроме читабельности кода повышается уровень готовности компонента к повторному использованию.

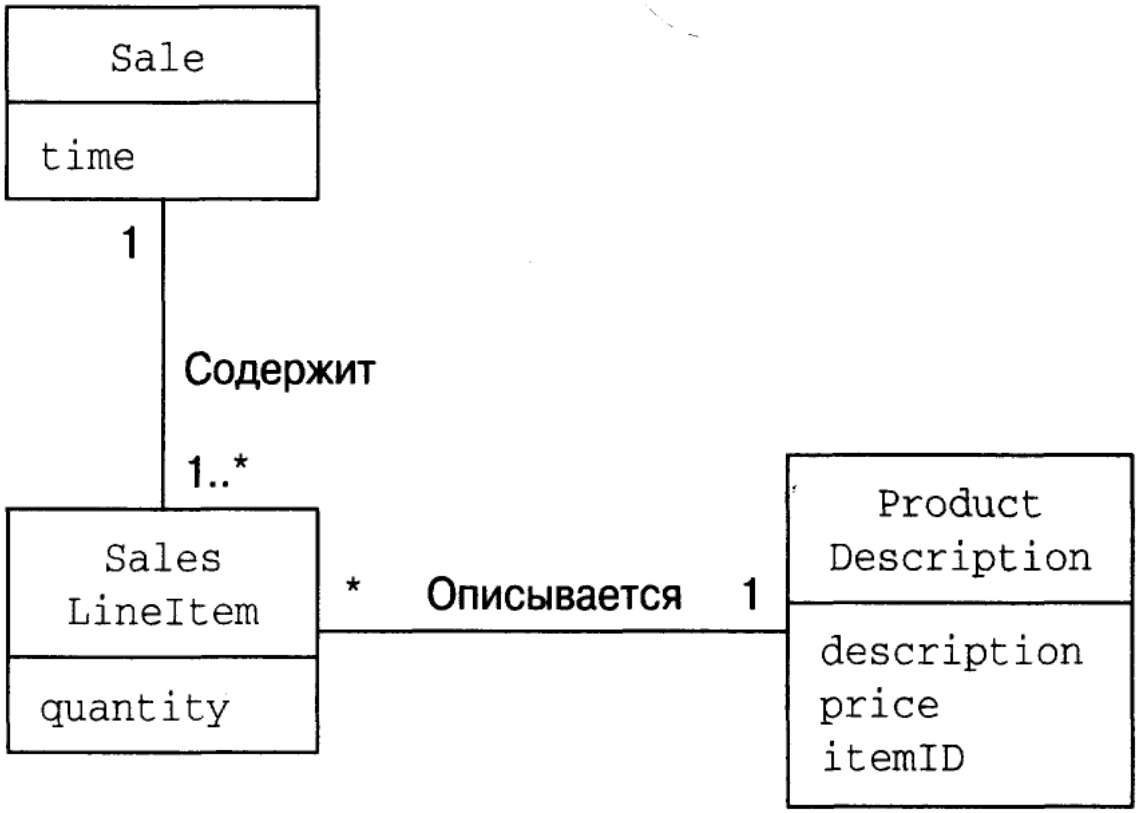
GRASP: Информационный эксперт

- **Проблема**:
  - каков **наиболее общий принцип распределения обязанностей** между объектами при объектно-ориентированном проектировании? В модели системы могут быть определены десятки или сотни программных классов, а в приложении может потребоваться выполнение сотен или тысяч обязанностей. Во время объектно-ориентированного проектирования при формулировке принципов взаимодействия объектов необходимо **распределить обязанности** между классами. При правильном выполнении этой задачи система становится гораздо проще для понимания, поддержки и расширения. Кроме того, появляется возможность повторного использования уже разработанных компонентов в последующих приложениях.
- **Решение**:
  - назначить обязанность **информационному эксперту** — классу, у которого имеется информация, требуемая для выполнения обязанности.
- **Пример**.
  - В приложении POS-системы NextGen некоторому классу необходимо знать **общую сумму продажи**. Какой класс это должен быть?

Фрагмент модели предметной области

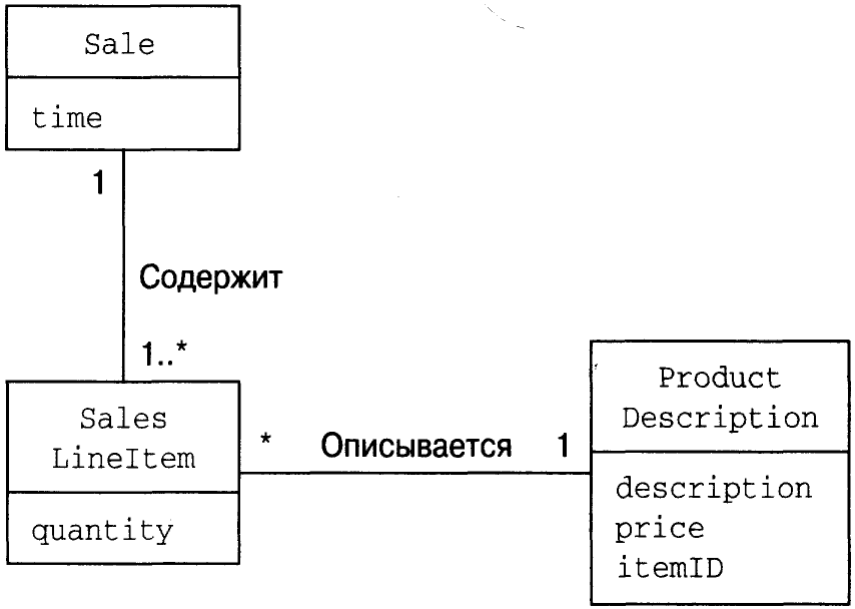


Пример - GRASP: Информационный эксперт

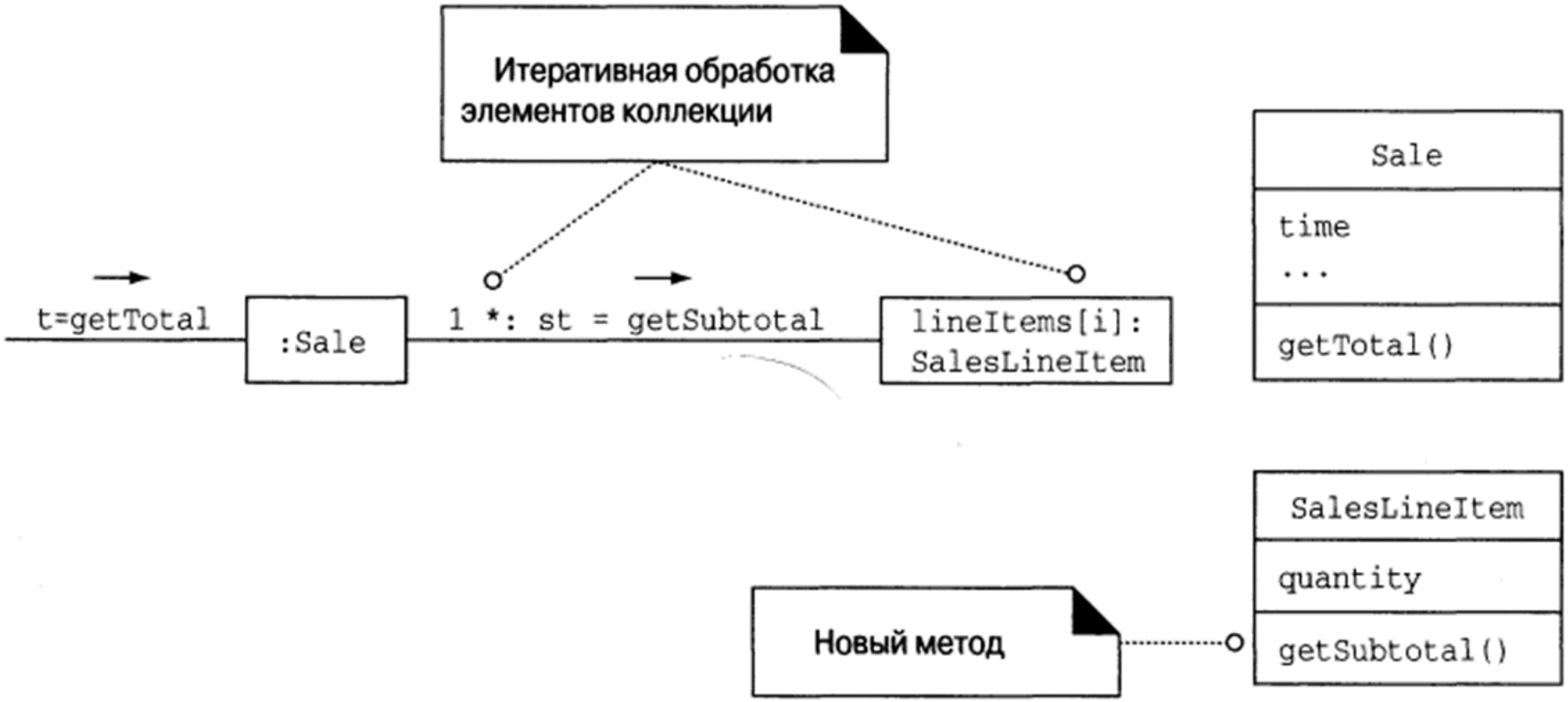


Необходимо рассчитать общую сумму продажи.

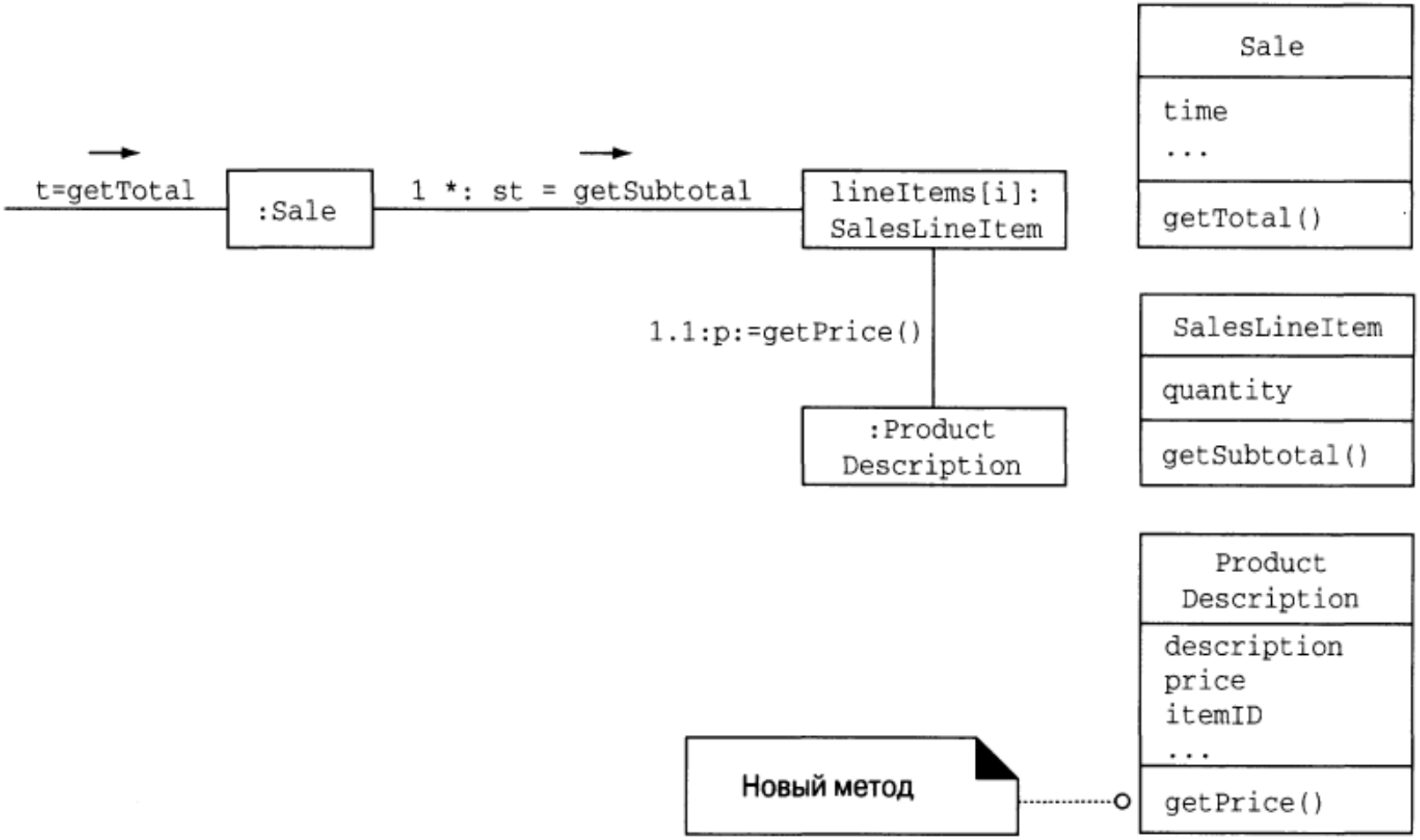
Пример - GRASP: Информационный эксперт



Пример - GRASP: Информационный эксперт



Пример - GRASP: Информационный эксперт



GRASP: Информационный эксперт

- **Обсуждение**:
  - Интуитивно понятный принцип
  - «Частичные эксперты», которым делегируется часть обязанностей.
  - Аналогия с реальным миром («живые объекты») – кому поручить задачу?
- **Ограничения**:
  - Применение данного шаблона не должно ухудшать связывание и зацепление (описаны далее) и нарушать SRP (следующая лекция). Пример – сохранение в БД.
- **Преимущества**.
  - Поддержка инкапсуляции.
  - Обеспечение простой и понятной структуры классов.

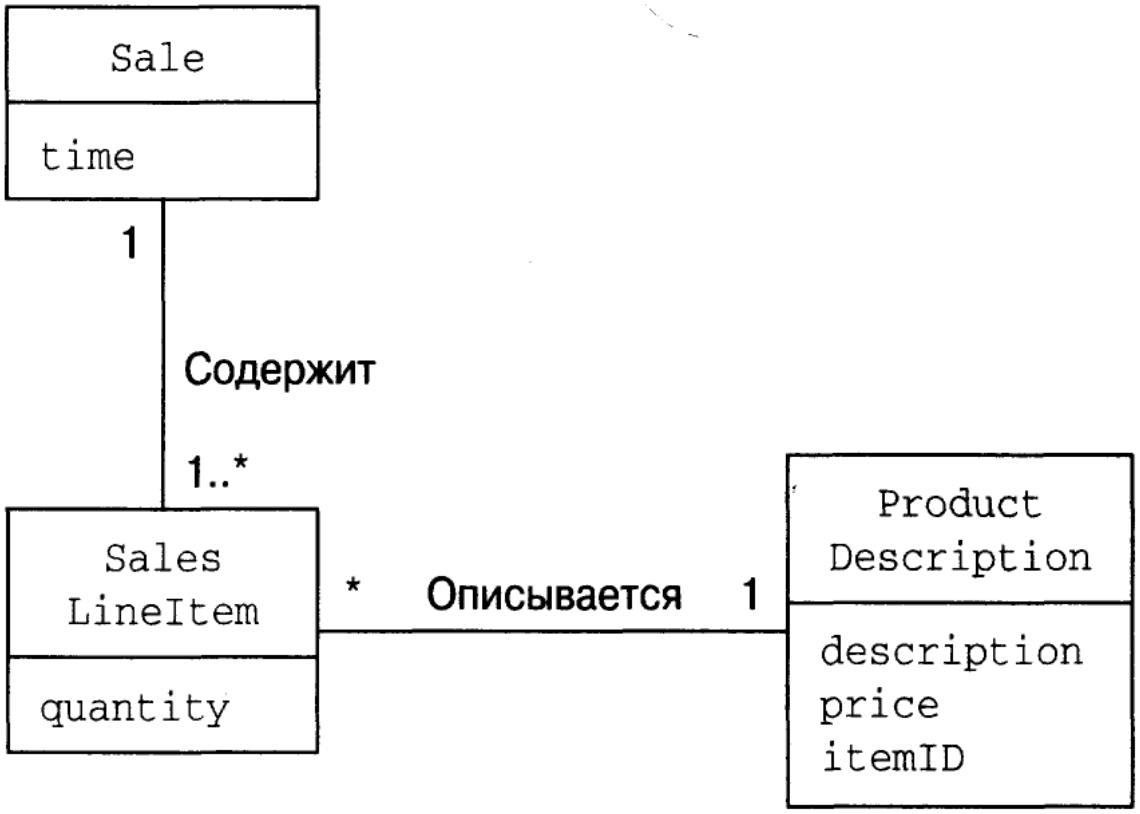
GRASP: Creator (Создатель)

- Шаблон **Creator** решает, кто должен **создавать объект**.
  - **Создание объектов** в объектно-ориентированной системе является одним из наиболее стандартных видов деятельности. Следовательно, при назначении обязанностей, связанных с созданием объектов, полезно руководствоваться некоторым основным принципом.
  - Правильно распределив обязанности при проектировании, можно создать слабо связанные независимые компоненты с возможностью их дальнейшего использования, упростить их, а также обеспечить инкапсуляцию данных и их повторное использование.
- Фактически, это применение шаблона **Information Expert** к проблеме создания объектов.
- Альтернативой создателю является шаблон проектирования **Фабрика**. В этом случае создание объектов концентрируется в отдельном классе.

GRASP: Создатель

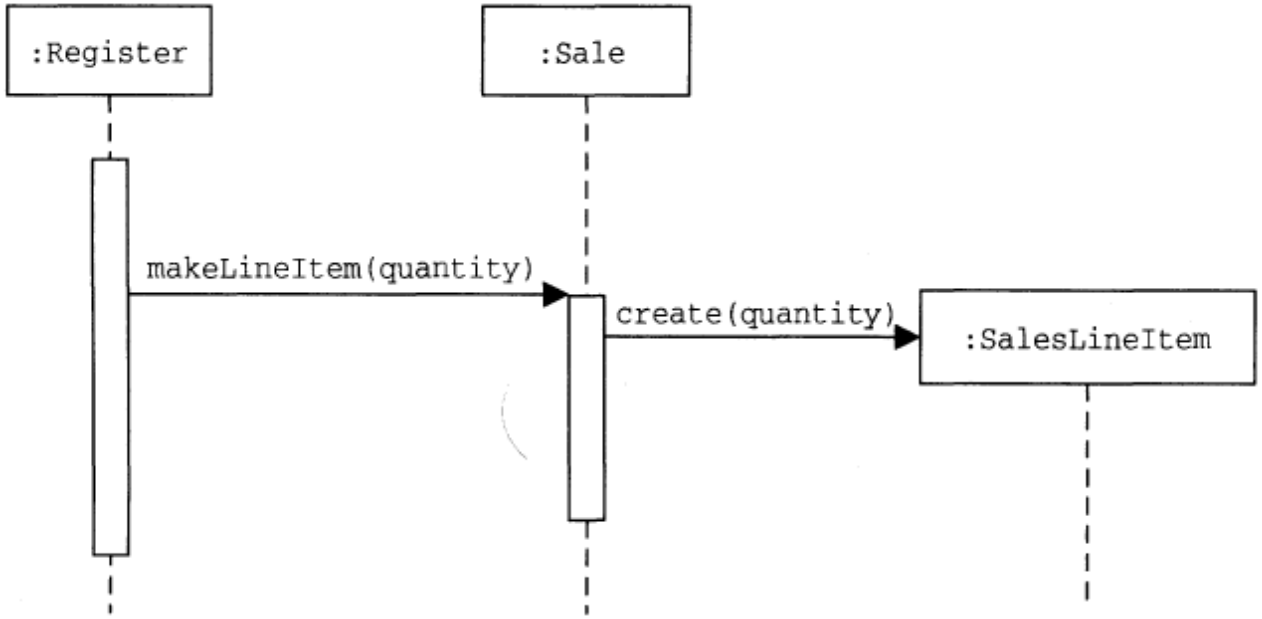
- **Проблема**:
  - кто должен отвечать за создание нового экземпляра некоторого класса?
- **Решение**:
  - назначить классу B обязанность создавать экземпляры класса A, если выполняется одно из следующих условий.
    - Класс B **содержит или агрегирует** объекты A
    - Класс B **записывает экземпляры** объектов A
    - Класс B **активно использует** объекты A.
    - Класс B обладает **данными инициализации**, которые будут передаваться объектам A при их создании (т.е. при создании объектов A класс B является **экспертом**).
  - Класс B при этом — **создатель** (creator) объектов A.
  - Если выполняется сразу несколько из этих условий, то возможно имеет смысл использовать класс B, агрегирующий или содержащий **класс** A (вложенный класс).
- Пример.
  - Кто в POS-системе должен отвечать за создание **нового экземпляра** объекта **SalesLineItem**?

Пример - GRASP: Создатель



- В соответствии с шаблоном Creator, необходимо найти класс, **агрегирующий**, **содержащий** и т.д. экземпляры объектов **SalesLineItem**.

Пример - GRASP: Создатель



- При таком распределении обязанностей требуется, чтобы в объекте Sale был определен метод makeLineItem.
- Рассмотрение и распределение обязанностей выполнялись в процессе создания диаграммы взаимодействий.
- Затем полученные результаты могут быть реализованы в конкретных методах, помещенных в раздел методов диаграммы классов.

GRASP: Создатель

- **Обсуждение**:
  - выявление объекта-создателя, который должен быть **связан** со всеми созданными им объектами. При таком подходе обеспечивается низкая степень связанности.
  - внешний контейнер или класс-регистратор — это хорошие **кандидаты** на выполнение обязанностей, связанных с созданием сущностей, которые они будут содержать или регистрировать.
  - Создатель, обладающий **данными инициализации == эксперт**
- **Ограничения**:
  - Зачастую создание экземпляра — это достаточно сложная операция, выполняемая при реализации некоторого условия на основе каких-либо внешних свойств. В этом случае предпочтительнее использовать шаблон **Factory (Фабрика)** и делегировать обязанность создания экземпляров вспомогательному классу.
- **Преимущества**.
  - Обеспечение слабого зацепления – при таком подходе не появляется новых ассоциаций между классами.

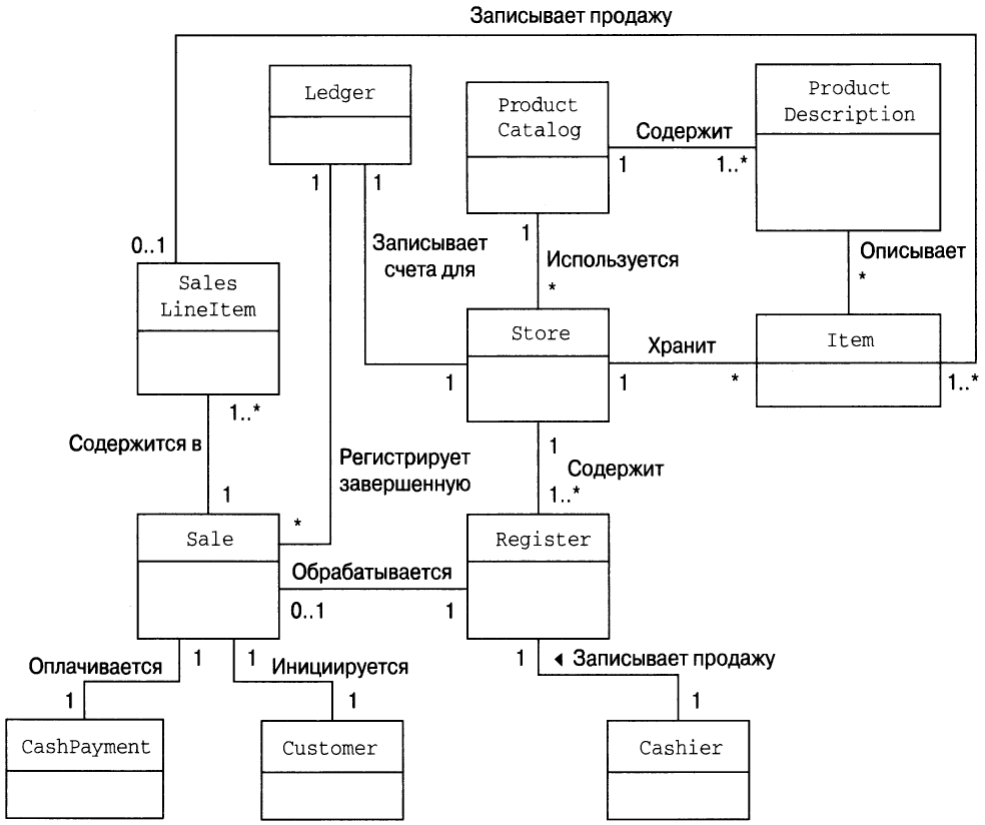
GRASP: Controller (Контроллер)

- **Контроллер** берёт на себя ответственность за выполнение **операций, инициируемых пользователем** и часто выполняет сценарий одного или нескольких **вариантов использования** (например, один контроллер может обрабатывать сценарии создания и удаления пользователя).
- Контроллер **не относится** к интерфейсу пользователя
- Как правило, контроллер не выполняет работу самостоятельно, а **делегирует** обязанности компетентным объектам.
- Иногда класс-контроллер представляет всю систему в целом, корневой объект, устройство или важную подсистему (внешний контроллер).

GRASP: Контроллер

- **Проблема**:
  - кто должен отвечать за **получение** и **координацию выполнения системных операций**, поступающих на уровне интерфейса пользователя?
    - **Системные операции** являются основными входными событиями в системе. Например, когда кассир в POS-системе щелкнет на кнопке Оплатить, он генерирует системное событие, свидетельствующее о завершении торговой операции.
- **Решение:**
  - **Делегирование обязанностей по обработке системных сообщений** классу, удовлетворяющему одному из следующих условий:
    - Класс представляет всю систему в целом, корневой объект, устройство или подсистему (**внешний контроллер**).
    - Класс представляет сценарий некоторого варианта использования, в рамках которого выполняется обработка всех системных событий, и обычно называется <Имя_ВИ>Handler, <Имя_ВИ>Coordinator или <Имя_ВИ>Session (**контроллер варианта использования, или контроллер сеанса**).
      - Для всех системных событий в рамках одного сценария варианта использования используется один и тот же класс-контроллер.
      - Неформально сеанс — это экземпляр взаимодействия с исполнителем. Сеансы могут иметь произвольную длину, но зачастую они организованы в рамках варианта использования (сеансы варианта использования).
  - **Контроллер** (controller) — это объект, не относящийся к интерфейсу пользователя и отвечающий за получение или обработку системных сообщений.
- **Пример**:
  - Какой класс должен выступать в роли контроллера для системных событий типа: enterItem или endSale?

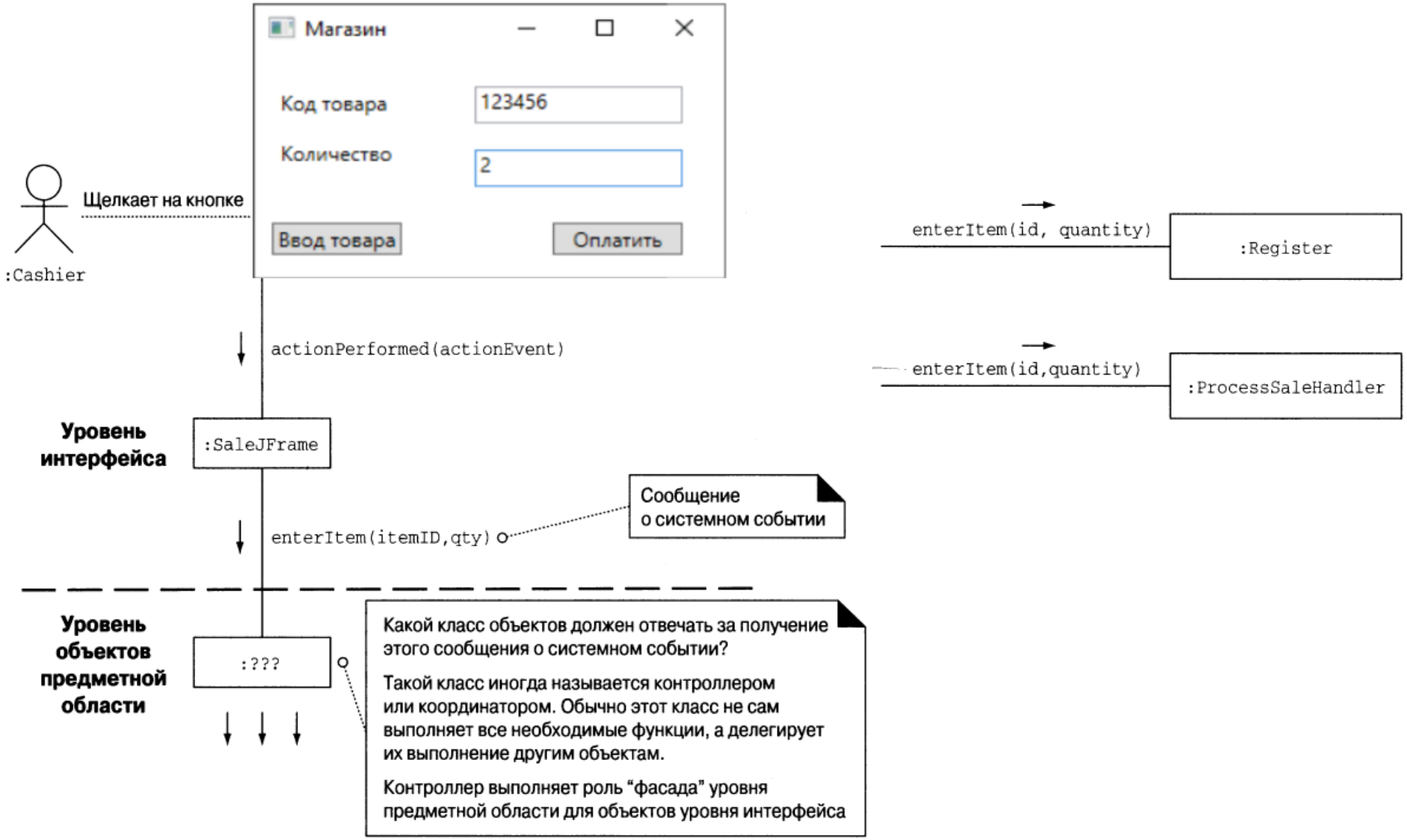
Пример - GRASP: Контроллер



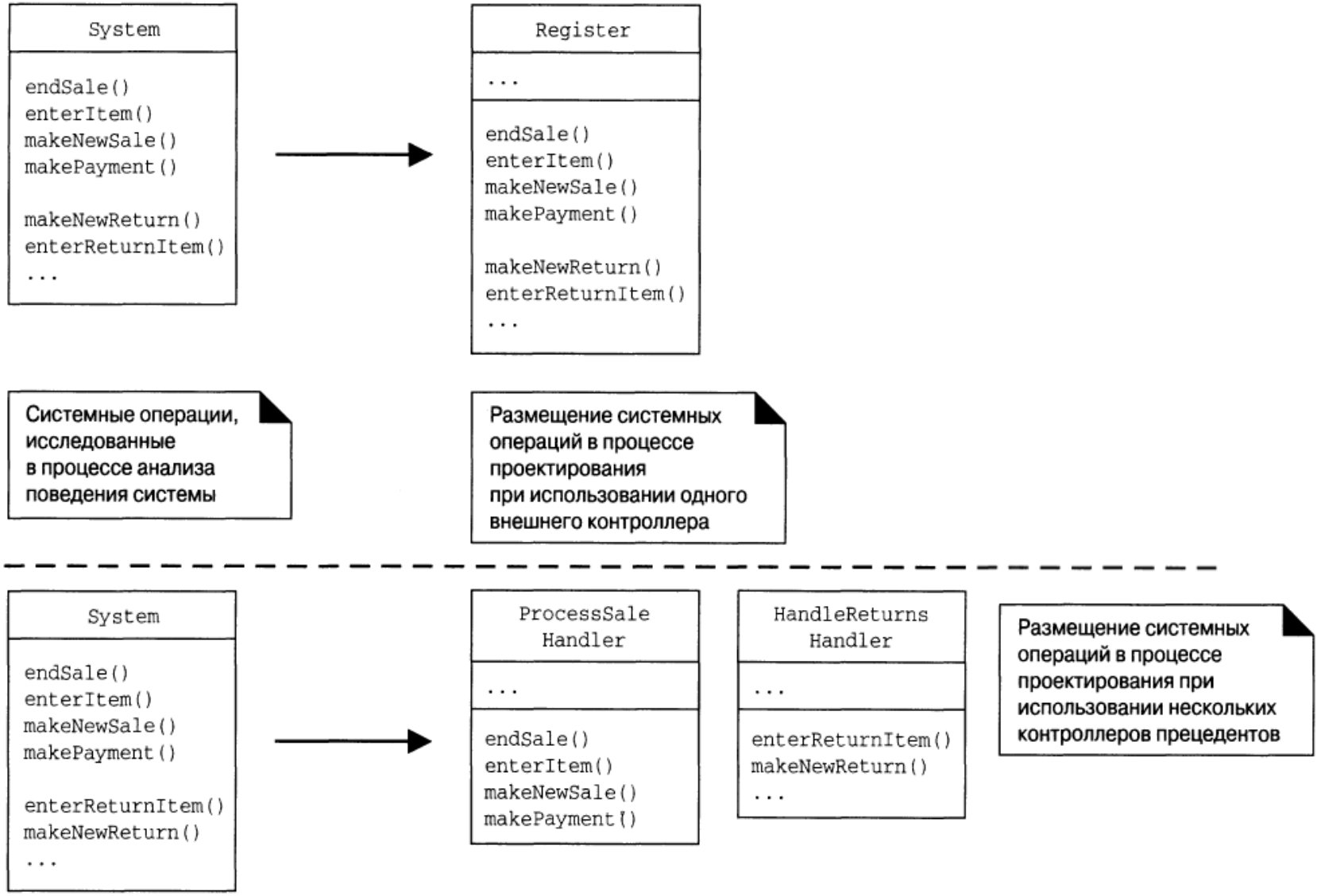
Согласно шаблону **Controller**, возможны следующие варианты:

- Класс, представляющий всю систему в целом, устройство или подсистему:
  - **Register**, **POSSystem**.
- Класс, представляющий получателя или искусственный обработчик всех системных событий некоторого сценария прецедента
  - **ProcessSaleHandler**, **ProcessSaleSession**.

Пример - GRASP: Контроллер



Пример - GRASP: Контроллер



Пример - GRASP: Контроллер

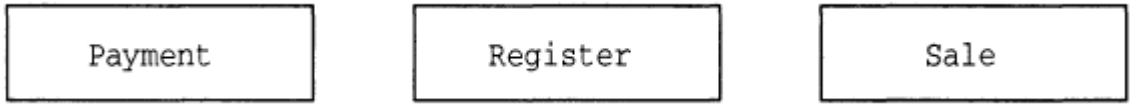
- **Обсуждение**:
  - Шаблон **делегирования** обязанностей.
    - Поскольку объекты интерфейса пользователя не должны реализовывать логику приложения, они делегируют эти обязанности объектам следующего уровня.
  - Чтобы обеспечить возможность **координации исполнения** варианта использования, для обработки всех системных событий в рамках него должен использоваться один и тот же класс контроллера, хранящий информацию о ходе взаимодействия.
    - Такая информация может понадобиться, например, для идентификации момента нарушения последовательности системных событий (например, выполнение операции makePayment перед выполнением операции endSale).
    - Для различных вариантов использования можно использовать разные контроллеры.
  - Типичной ошибкой при создании контроллеров является возложение на них слишком большого числа обязанностей.
    - Обычно контроллер должен лишь делегировать функции другим объектам и координировать их деятельность, а не выполнять эти действия самостоятельно.
- **Ограничения**
  - **Раздутый контроллер: единственный, сам выполняет все функции, содержит или дублирует информацию.**
    - **высокое зацепление и низкая связность**
- **Преимущества**.
  - Улучшение условий для **повторного использования** компонентов и подключения интерфейсов.
  - **Координация исполнения** варианта использования.

GRASP: Low Coupling (Слабое зацепление)

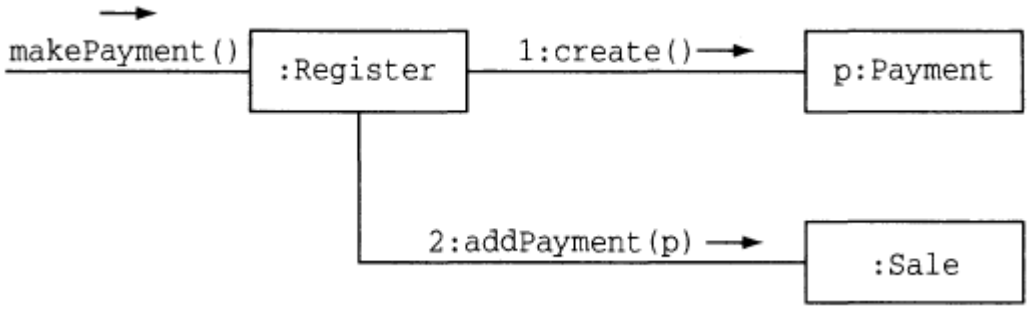
- **Проблема**:
  - как обеспечить незначительное влияние изменений и повысить возможность повторного использования?
- **Зацепление**, или **Степень связанности (coupling)** — это мера, определяющая насколько жестко один элемент **связан** с другими элементами, либо каким количеством **данных** о других элементах он обладает.
- Элемент с **низкой степенью связанности** (или **слабым зацеплением**) зависит от не очень большого числа других элементов.
  - Выражение “очень много” зависит от контекста, однако необходимо провести его оценку.
- Класс с **высокой степенью связанности** (или жестко **сцепленный** с другими) зависит от множества других классов. Наличие таких классов нежелательно, поскольку оно приводит к возникновению следующих проблем.
  - **Изменения** в связанных классах приводят к локальным изменениям в данном классе.
  - Затрудняется **понимание** каждого класса в отдельности.
  - Усложняется **повторное использование**, поскольку для этого требуется дополнительный анализ классов, с которыми связан данный класс.
- **Решение**:
  - распределить обязанности таким образом, чтобы степень связанности оставалась низкой.

GRASP: Слабое зацепление

- **Пример**.
  - Рассмотрим следующий фрагмент диаграммы классов для приложения NextGen.



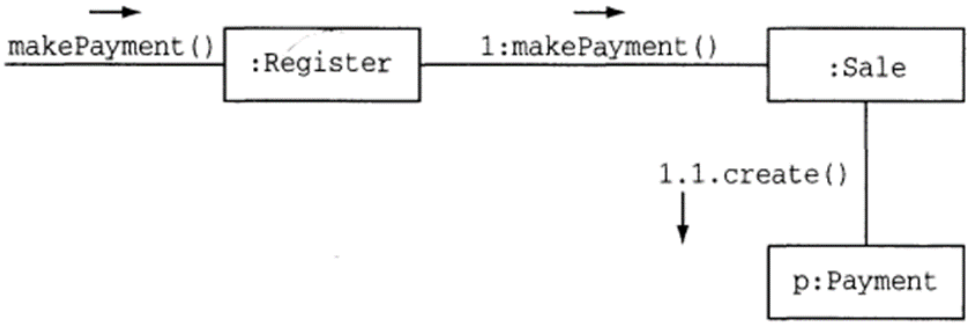
  - Необходимо создать экземпляр класса **Payment** и связать его с объектом **Sale**.
  - Какой класс должен отвечать за выполнение этой операции?
  - Поскольку в реальной предметной области регистрация объекта **Payment** выполняется объектом **Register**, в соответствии с шаблоном **Creator**, объект **Register** является хорошим кандидатом для создания объекта **Payment**. Затем экземпляр объекта **Register** должен передать сообщение **addPayment** объекту **Sale**, указав в качестве параметра новый объект **Payment**.



  - Такое распределение обязанностей предполагает, что **класс Register обладает знаниями о данных класса Payment** (т.е. **связывается** с ним).

GRASP: Слабое зацепление

- **Пример**.
  - Альтернативный способ создания объекта **Payment** и его связывания с объектом **Sale**



  - В обоих случаях предполагается, что в конечном итоге объекту **Sale** должно быть известно о существовании объекта **Payment**.
  - При использовании первого способа, когда объект **Payment** создается с помощью объекта **Register**, между этими двумя объектами добавляется **новая связь**, тогда как второй способ **степень связывания** объектов не усиливает.
  - С точки зрения числа связей между объектами, более предпочтительным является второй способ, поскольку в этом случае обеспечивается низкое **зацепление**.
  - Приведенная иллюстрация является примером того, как при использовании двух различных шаблонов — **Low Coupling** и **Creator** — можно прийти к двум различным решениям.

GRASP: Слабое зацепление

- **Обсуждение**.
  - Low Coupling – это **средство оценки всех принимаемых в процессе проектирования решений** на протяжении всех стадий работы над проектом.
  - В объектно-ориентированных языках программирования, таких как C++, Java и C#, имеются следующие стандартные способы зацепления объектов TypeX и TypeY.
    - TypeX **содержит атрибут** (переменную-член), который ссылается на экземпляр TypeY или сам TypeY.
    - TypeX **вызывает методы** объекта TypeY
    - TypeX **содержит метод**, **который каким-либо образом ссылается** на экземпляр TypeY или сам TypeY (TypeY в качестве типа параметра, локальной переменной или возвращаемого значения).
    - TypeX является прямым или непрямым **подклассом** объекта TypeY.
    - TypeY является интерфейсом, а TypeX **реализует этот интерфейс**.
  - Шаблон Low Coupling подразумевает такое распределение обязанностей, которое не влечет за собой **чрезмерное** повышение степени зацепления, приводящее к **отрицательным** результатам.
  - Шаблон Low Coupling **нельзя рассматривать изолированно** от других шаблонов, таких как Expert и High Cohesion. Скорее, он обеспечивает **один из** основных принципов проектирования, применяемых при распределении обязанностей.

GRASP: Слабое зацепление

- **Обсуждение**
  - **Подкласс жестко связан со своим суперклассом**. Это следует учитывать, принимая решение о наследовании.
    - Например, предположим, объекты необходимо постоянно хранить в базе данных. В этом случае, зачастую, создают абстрактный суперкласс **PersistentObject**, от которого наследуют свои свойства другие классы. Недостатком такого подхода является жесткое зацепление объектов с конкретной службой, а преимуществом — автоматическое наследование поведения.
  - **Не существует абсолютной меры для определения слишком высокой степени зацепления**.
    - Важно лишь понимать степень связанности объектов на текущий момент и не упустить тот момент, когда дальнейшее повышение степени связанности может привести к возникновению проблем.
    - В целом, следует руководствоваться таким принципом: классы, которые являются достаточно общими по своей природе и с высокой вероятностью будут повторно использоваться в дальнейшем, должны иметь минимальную степень связанности с другими классами.
  - **Крайним случаем при реализации шаблона Low Coupling является полное отсутствие зацепления между классами**.
    - Такая ситуация тоже **нежелательна**, поскольку базовой идеей объектного подхода является система связанных объектов, которые “общаются” между собой посредством передачи сообщений.
    - При слишком частом использовании принципа слабого зацепления система будет состоять из нескольких изолированных сложных активных объектов, самостоятельно выполняющих все операции, и множества пассивных объектов, основная функция которых сводится к хранению данных.
    - Поэтому при создании объектно-ориентированной системы должна присутствовать некоторая **оптимальная степень зацепления** между объектами, позволяющая выполнять основные функции посредством взаимодействия этих объектов.

GRASP: Слабое зацепление

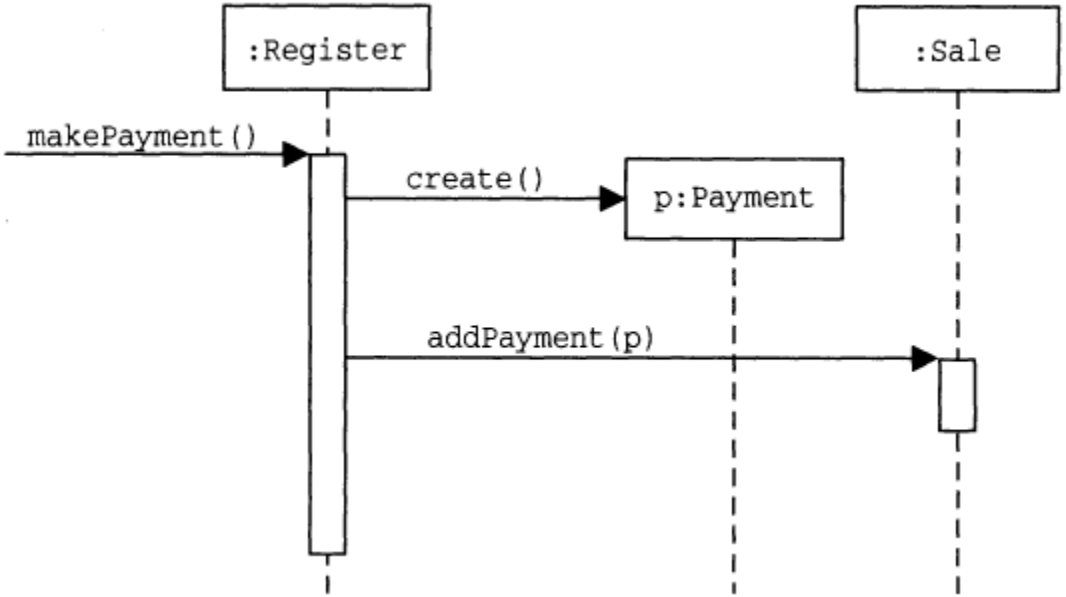
- **Когда не следует применять шаблон**.
  - **Высокая степень зацепления с устойчивыми элементами не представляет проблемы**. Например, приложение может быть связано с библиотеками фреймворка, поскольку эти библиотеки широко распространены и стабильны.
- Проблемой является жесткое зацепление с **неустойчивыми** в некотором отношении элементами.
  - Например, в проекте NextGen к системе планируется подключать различные программы вычисления налоговых платежей (с общим интерфейсом). Следовательно, в этой части системы нужно обеспечить низкую степень зацепления.
- **Преимущества**
  - Изменения компонентов мало сказываются на других объектах.
  - Принципы работы и функции компонентов можно понять, не изучая другие объекты.
  - Удобство повторного использования.

GRASP: High Cohesion (Сильная связность)

- **Проблема**:
  - как обеспечить **сфокусированность** обязанностей объектов, их **управляемость** и **ясность**, а заодно выполнение принципа Low Coupling?
  - В терминах ООП **связность** (cohesion) — **это мера связанности и сфокусированности обязанностей класса**.
  - Считается, что элемент обладает **высокой степенью связности**, если его **обязанности тесно связаны между собой** и он не выполняет непомерных объемов работы. В роли таких элементов могут выступать классы, подсистемы и т.д.
- **Решение**:
  - распределение обязанностей, поддерживающее **высокую степень связности**.
  - Этот принцип используется для оценки возможных альтернатив. Класс с **низкой степенью связности** выполняет **много разнородных функций** или **несвязанных между собой обязанностей**. Такие классы создавать **нежелательно**, поскольку они приводят к возникновению следующих проблем:
    - Трудность **понимания**.
    - Сложности при **повторном использовании**.
    - Сложности **поддержки**.
    - **Ненадежность**, постоянная подверженность **изменениям**.

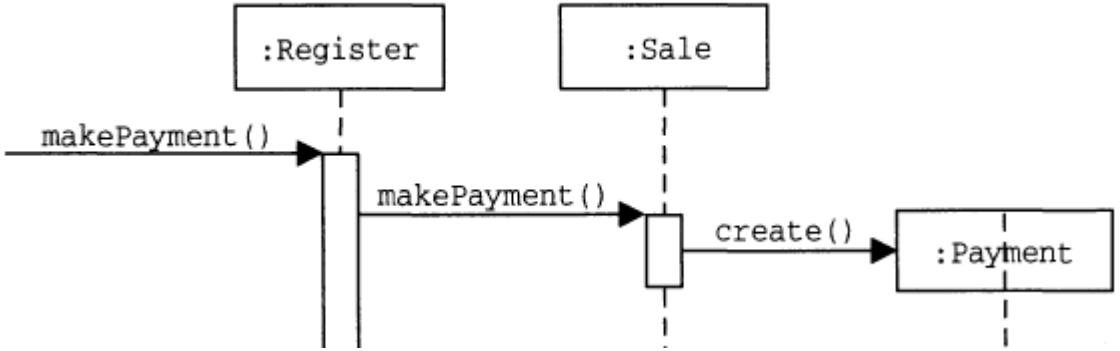
GRASP: Сильная связность

- **Пример**
  - Необходимо создать экземпляр класса **Payment** и связать его с объектом **Sale**.
  - Какой класс должен отвечать за выполнение этой операции?
  - Поскольку в реальной предметной области регистрация объекта **Payment** выполняется объектом **Register**, в соответствии с шаблоном **Creator**, объект **Register** является хорошим кандидатом для создания объекта **Payment**. Затем экземпляр объекта **Register** должен передать сообщение **addPayment** объекту **Sale**, указав в качестве параметра новый объект **Payment**.



GRASP: Сильная связность

- **Пример**
  - При таком распределении обязанностей платежи выполняет объект **Register**, т.е. объект **Register** частично несет **ответственность за выполнение системной операции makePayment**.
  - В данном конкретном примере это приемлемо. Однако если и далее возлагать на класс **Register** обязанности по выполнению все новых и новых функций, связанных с другими системными операциями, то этот класс **будет слишком перегружен и будет обладать низкой степенью связности**.
  - Предположим, приложение должно выполнять пятьдесят системных операций и все они возложены на класс **Register**. Если этот объект будет выполнять все операции, то он станет чрезмерно “раздутым” и не будет обладать свойством связности. И дело не в том, что одна задача создания экземпляра объекта Payment сама по себе снизила степень связности объекта Register; она является частью общей картины распределения обязанностей.



GRASP: Сильная связность

- **Обсуждение**.
  - О высокой степени связности следует помнить в течение **всего процесса проектирования** для **оценки эффективности** каждого проектного решения.
  - Примеры
    - **Очень слабая связность** - один класс отвечает за выполнение множества операций в самых различных функциональных областях. (RDB-RPC-Interface)
    - **Слабая связность** - класс несет единоличную ответственность за выполнение сложной задачи из одной функциональной области. (RDB-Interface)
    - **Сильная связность** - класс имеет среднее количество обязанностей из одной функциональной области и для выполнения своих задач взаимодействует с другими классами. (RDB-Connection)
  - Предпочтительно создавать классы с **высокой степенью связности**, поскольку они весьма просты в **понимании**, **поддержке** и **повторном использовании**.
    - Шаблон High Cohesion, как и другие понятия объектно-ориентированной технологии проектирования, имеет аналогию в реальном мире. Всем известно, что человек, выполняющий большое число разнородных обязанностей, работает не очень эффективно.
  - **Некорректное связывание порождает неправильное зацепление и наоборот**.

GRASP: Сильная связность

- **Когда не следует применять шаблон**.
  - Специфические области кода, требующие особых знаний, могут группироваться в один слабо связный модуль/класс. (SQL в проекте).
  - Распределенные серверные объекты – укрупнение объектов, реализующих удаленные интерфейсы, может повышать производительность.
- **Преимущества**
  - Повышаются **ясность** и **простота** проектных решений.
  - **Упрощаются** поддержка и доработка.
  - Зачастую обеспечивается **слабое зацепление**.
  - Улучшаются возможности **повторного использования**, поскольку класс с высокой степенью связности выполняет конкретную задачу.

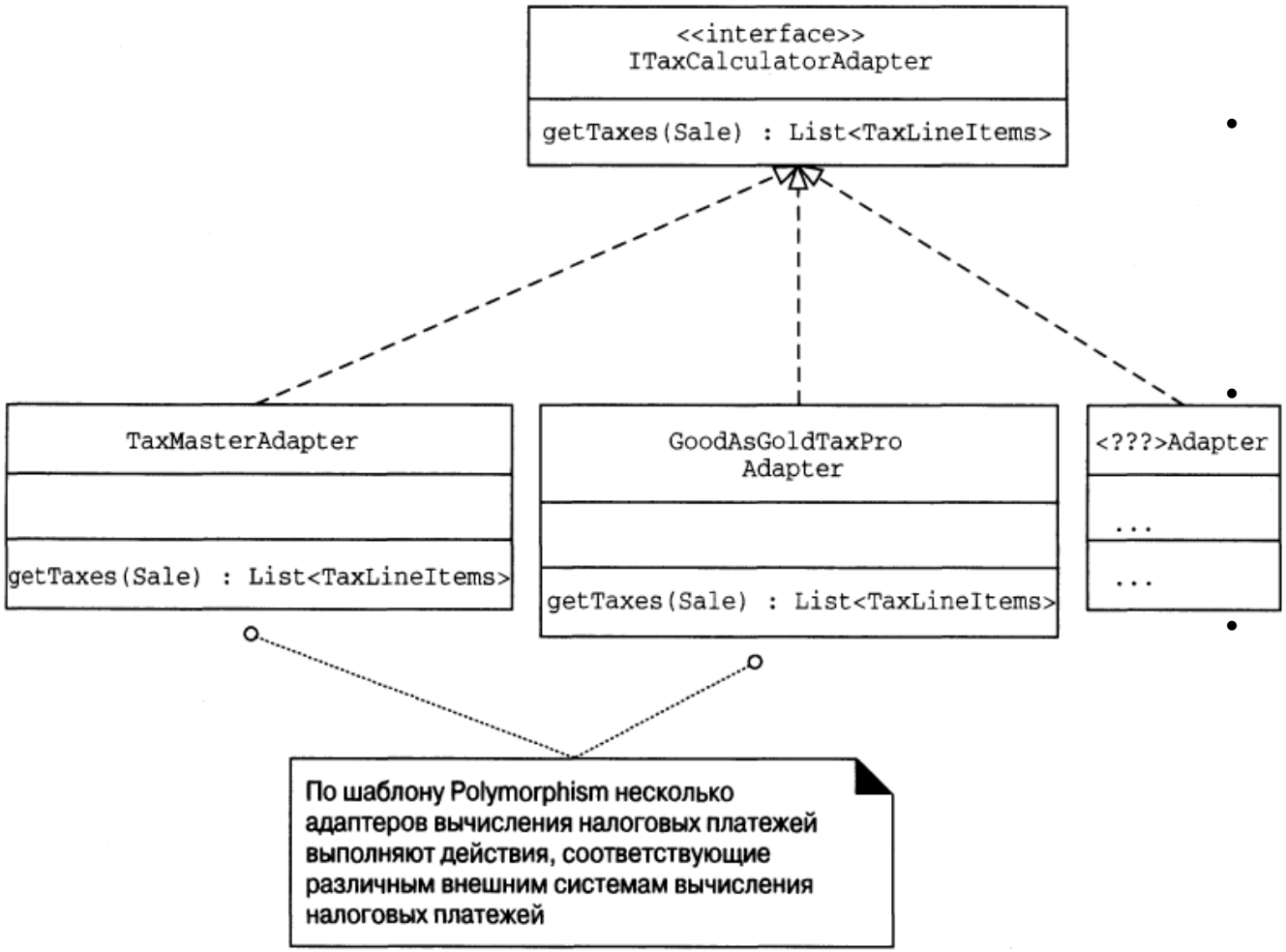
GRASP: Polymorphism (Полиморфизм)

- **Проблема**:
  - как обрабатывать альтернативные варианты поведения на основе типа? Как создавать подключаемые программные компоненты?
  - **Альтернативные варианты поведения на основе типа**.
    - Если программа разработана с использованием условных операторов типа if-then-else или switch-case, то при добавлении новых вариантов поведения приходится модифицировать логику условных операторов во множестве мест. Как этого избежать?
  - **Подключаемые программные компоненты**.
    - Если рассматривать компоненты с точки зрения отношения “клиент/сервер”, то как можно заменить один серверный компонент другим, не затрагивая при этом клиентские компоненты?
- **Решение**:
  - если поведение объектов одного типа (класса) может изменяться, обязанности распределяются для различных вариантов поведения с использованием полиморфных операций для этого класса.

GRASP: Полиморфизм

- **Пример**:
  - Приложение NextGen: как поддерживать работу внешних систем вычисления налоговых платежей?
    - POS-система NextGen должна поддерживать работу **различных** внешних систем вычисления налоговых платежей (в том числе Tax-Master и Good-As-Gold TaxPro) и интеграцию с **другими** внешними системами.
    - Каждая система вычисления налоговых платежей имеет **свой интерфейс** и обладает **собственным поведением** (хотя и аналогичным поведению других систем).
      - Один продукт может поддерживать протокол TCP, другой — интерфейс SOAP, а третий — HTTP REST.
  - **Какие объекты должны отвечать за обработку различных внешних интерфейсов систем вычисления налоговых платежей?**
  - Поскольку поведение службы адаптации внешней системы вычисления налоговых платежей зависит от типа используемой программы вычисления (калькулятора), согласно шаблону Polymorphism, необходимо каким-то образом распределить обязанности по адаптации к различным типам калькуляторов.
  - Для этого можно использовать **полиморфную операцию getTaxes**

GRASP: Полиморфизм



- Объекты, обеспечивающие адаптацию, — это не внешние калькуляторы, а **локальные** программные объекты, их представляющие (**прокси**).
- При отправке сообщения локальному объекту выполняется **обращение** к внешней системе с использованием ее собственного программного интерфейса.
- Каждому методу **getTaxes** в качестве параметра передается объект **Sale**, чтобы система вычисления налоговых платежей могла проанализировать продажу.
- Реализации каждого такого метода отличаются по механизму адаптации к внешней системе

GRASP: Полиморфизм

- **Обсуждение**.
  - **Полиморфизм** — это основной принцип проектирования поведения системы в рамках обработки аналогичных ситуаций.
  - Если обязанности в системе распределены на основе шаблона Polymorphism, то такую **систему** можно **легко модифицировать**, **добавляя в нее новые вариации**.
- **Когда не нужно использовать шаблон**.
  - Иногда разработчики систем **злоупотребляют** добавлением интерфейсов и применением принципа полиморфизма с целью обеспечения дееспособности системы в неизвестных заранее новых ситуациях.
  - Если точка вариаций известна и обоснована, если существует высокая вероятность вариантного поведения, то такие усилия вполне оправданны.
  - Однако зачастую усилия по обеспечению вариантов поведения “на все случаи жизни” затрачиваются впустую.
  - Реально оценивайте вероятность вариантного поведения.
- **Преимущества**
  - С помощью шаблона впоследствии можно легко расширять систему, добавляя в нее новые вариации.
  - Новые реализации можно вводить без модификации клиентской части приложения.

GRASP: Protected Variations (Сокрытие реализации)

- **Проблема**:
  - как спроектировать объекты, подсистемы и систему, чтобы изменение этих элементов не оказывало нежелательного влияния на другие элементы?
- **Решение**:
  - **идентифицировать точки возможных вариаций или неустойчивости**; распределить обязанности таким образом, чтобы обеспечить **устойчивый интерфейс**.
    - Термин “интерфейс” в данном случае используется для обозначения способа обеспечения доступа и не сводится к понятию интерфейса ЯП.
- **Пример**.
  - Та же проблема взаимодействия с различными системами вычисления налоговых платежей. Решение на основе шаблона Polymorphism одновременно иллюстрирует применение шаблона Protected Variations
    - В данном случае **точкой вариации или неустойчивости являются различные интерфейсы или API внешних систем вычисления налоговых платежей** .
    - POS-система должна обеспечить интеграцию с различными известными (а также еще не существующими) системами вычисления налоговых платежей.
    - Добавив **прокси-интерфейс**, можно разработать разные реализации адаптеров, защитив систему от возможного изменения внешних интерфейсов.
    - Тогда внутренние объекты смогут взаимодействовать с **устойчивым интерфейсом**, а **детали** взаимодействия с внешними системами будут скрыты в **конкретных реализациях** адаптеров.

GRASP: Сокрытие реализации

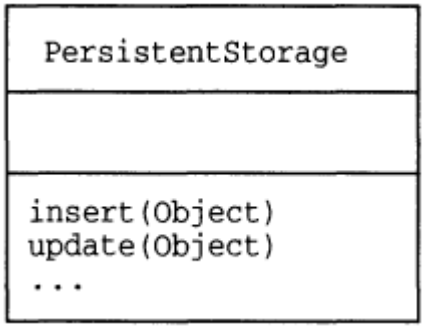
- **Обсуждение**.
  - **Сокрытие реализации** — это важный фундаментальный принцип разработки программных систем, продолжающий на более высоком уровне саму идею **инкапсуляции**.
  - Частный случай – принцип «**Не разговаривайте с незнакомцами» или Закон Деметры**.
- **Когда не следует применять шаблон**.
  - Следует определить два типа особых точек.
    - **Точка вариации** — точка ветвления в существующей на данный момент системе или в требованиях к ней, например, необходимость поддержки нескольких интерфейсов для систем вычисления налоговых платежей.
    - **Точка эволюции** — предполагаемая точка ветвления, которая может возникнуть в будущем, однако не определяемая существующими требованиями.
  - Данный шаблон применим к обоим типам, но к **точкам эволюции – с осторожностью**, т.к. результат, скорее всего, не оправдает затраты.
- **Преимущества**
  - Легкость добавления новых расширений и вариаций.
  - Возможность добавления новых реализаций, не затрагивая клиента.
  - Слабое зацепление.
  - Минимизация влияния изменений.

GRASP: Pure Fabrication (Чистая выдумка)

- **Проблема**:
  - какой класс должен обеспечить реализацию шаблонов **High Cohesion** и **Low Coupling** или других принципов проектирования, если шаблон **Expert** (например) не обеспечивает подходящего решения?
  - Объектно-ориентированные системы отличаются тем, что **программные классы реализуют понятия предметной области**, как, например, классы **Sale** и **Customer**. Однако, существует множество ситуаций, когда распределение обязанностей **только** между такими классами приводит к **проблемам с зацеплением и связностью**, т.е. с невозможностью повторного использования кода.
- **Решение**:
  - Присвоить группу обязанностей с высокой степенью связности **искусственному классу**, не представляющему конкретного понятия предметной области, т.е. синтезировать **искусственную сущность** для поддержки высокой связности, слабого зацепления и повторного использования. Такой класс является продуктом нашего воображения и представляет собой **выдумку** (fabrication).
  - В идеале, присвоенные этому классу обязанности поддерживают высокую степень связности и низкое зацепление, так что структура этого синтетического класса является очень прозрачной, или **чистой** (pure). Отсюда и название: **Pure Fabrication (“чистая выдумка”)**.
    - И наконец, словосочетание “чистая выдумка ” подразумевает создание некоторой сущности в тот момент, когда разработчик “близок к отчаянию”.

GRASP: Pure Fabrication (Чистая выдумка)

- Пример
  - Приложение NextGen: сохранение объекта **Sale** в базе данных
  - Согласно шаблону Information Expert, эту обязанность можно присвоить самому классу **Sale**.
  - Однако следует принять во внимание следующие моменты:
    - Данная задача требует достаточно большого числа специализированных операций, связанных со взаимодействием с базой данных и никак не связанных с понятием самой продажи. Поэтому класс **Sale** получает **низкую степень связности**
    - Класс **Sale** должен быть связан с интерфейсом базы данных. Поэтому **возрастает степень зацепления**, причем даже не с другим объектом предметной области, а с внешней библиотекой
    - Хранение объектов в базе данных — это достаточно общая задача, решение которой необходимо для многих классов. Возлагая эту обязанность на класс **Sale**, разработчик не обеспечивает возможности **повторного использования** этих операций и вынужден дублировать их в других классах.
  - Естественным решением данной проблемы является создание **нового класса**, ответственного за сохранение объектов некоторого вида на постоянном носителе, например в базе данных.
    - Его можно назвать **PersistentStorage** (Постоянное хранилище).
    - Этот класс является **продуктом нашего воображения**, что полностью соответствует шаблону **Pure Fabrication**.



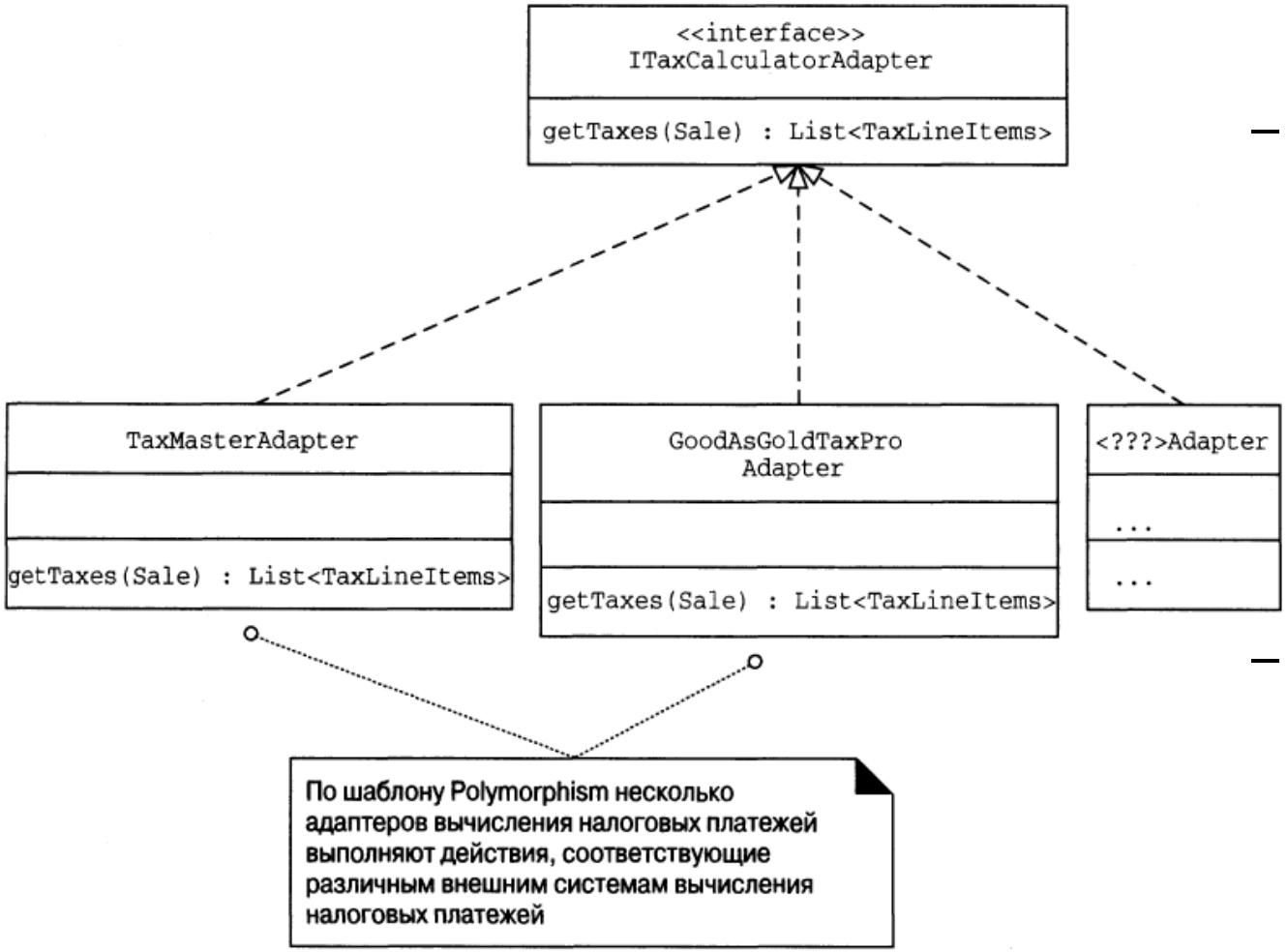
GRASP: Pure Fabrication (Чистая выдумка)

- **Преимущества**
  - При использовании шаблона Pure Fabrication реализуется шаблон High Cohesion, поскольку обязанности передаются отдельному классу, сконцентрированному на решении специфического набора взаимосвязанных задач.
  - Повышается потенциал повторного использования, поскольку чисто синтетические классы можно применять в других приложениях.
- **Когда не нужно использовать шаблон.**
  - Новички в области ООП, имеющие опыт разработки программ в рамках структурного или функционального подхода, зачастую **злоупотребляют** использованием шаблона Pure Fabrication.
  - В их трактовке функции просто превращаются в объекты. Необходимо соблюдать **баланс** между количеством таких объектов и объектов, созданных на основе декомпозиции представления, согласно шаблону Information Expert.
  - Класс Sale тоже имеет право на существование и должен выполнять свои обязанности. Созданные на основе шаблона Information Expert объекты обладают информацией, необходимой для выполнения своих обязанностей, и **соответствуют принципу слабого зацепления объектов**.
  - При **злоупотреблении** применением шаблона Pure Fabrication **нарушаются требования к слабому зацепления объектов**. Типичным симптомом такой ситуации является необходимость передачи данных одного объекта другим объектам для выполнения действий над ними.

GRASP: Indirection (Посредник)

- **Проблема**:
  - как распределить обязанности, чтобы обеспечить отсутствие прямого связывания; снизить уровень зацепления объектов, согласно шаблону Low Coupling, и сохранить высокий потенциал повторного использования?
- **Решение**:
  - присвоить обязанности **промежуточному объекту** для обеспечения связи между другими компонентами или службами, которые не связаны между собой напрямую. При таком подходе связи **перенаправляются** к другим компонентам или службам.
  - Зачастую, такой объект является Чистой выдумкой.
- **Пример**
  - Класс **TaxCalculatorAdapter** (выше)
  - Класс **PersistentStorage** (выше)
- **Обсуждение**.
  - многие вариации шаблона Pure Fabrication зачастую базируются на основном принципе перенаправления связи по шаблону Indirection. Целью такого перенаправления обычно является слабое зацепление, обеспечиваемое за счет отделения друг от друга различных компонентов или служб.
- **Преимущества**.
  - Слабое зацепление между компонентами.

GRASP: Indirection (Посредник)



- **ITaxCalculatorAdapter** – не существующий в предметной области **промежуточный объект** для обеспечения связи между другими компонентами или службами, которые не связаны между собой напрямую.
- Позволяет клиентским объектам знать только об интерфейсе-посреднике, и не знать о конкретных классах адаптеров
- Является Чистой выдумкой.

GRASP: Посредник

- Обсуждение.
  - многие вариации шаблона Pure Fabrication зачастую базируются на основном принципе перенаправления связи по шаблону Indirection. Целью такого перенаправления обычно является слабое связывание, обеспечиваемое за счет отделения друг от друга различных компонентов или служб.
- Преимущества.
  - Слабое связывание между компонентами.