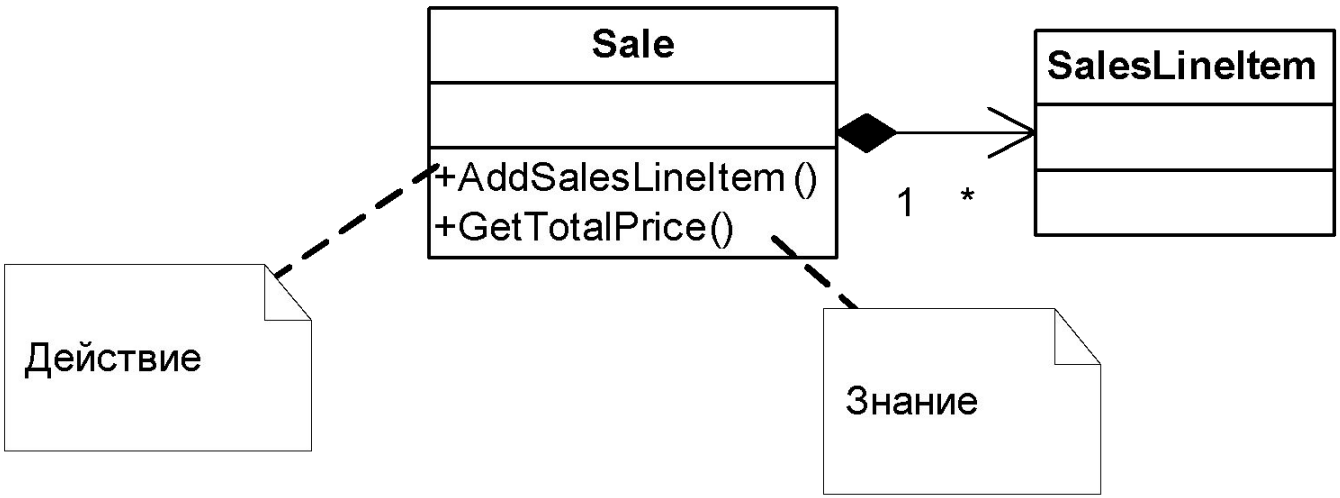
Распределение обязанностей

С использованием UML

Обязанности

- Обязанность – контракт или обязательство классификатора.
  - Действие
    - Выполнение действий другим объектом (создание экземпляра или выполнение вычислений)
    - Инициирование действий других объектов
    - Управление действиями других объектов или их координирование
  - Знание
    - Наличие информации о закрытых инкапсулированных данных
    - Наличие информации о связных объектах
    - Наличие информации о следствиях или вычисляемых величинах

Обязанности



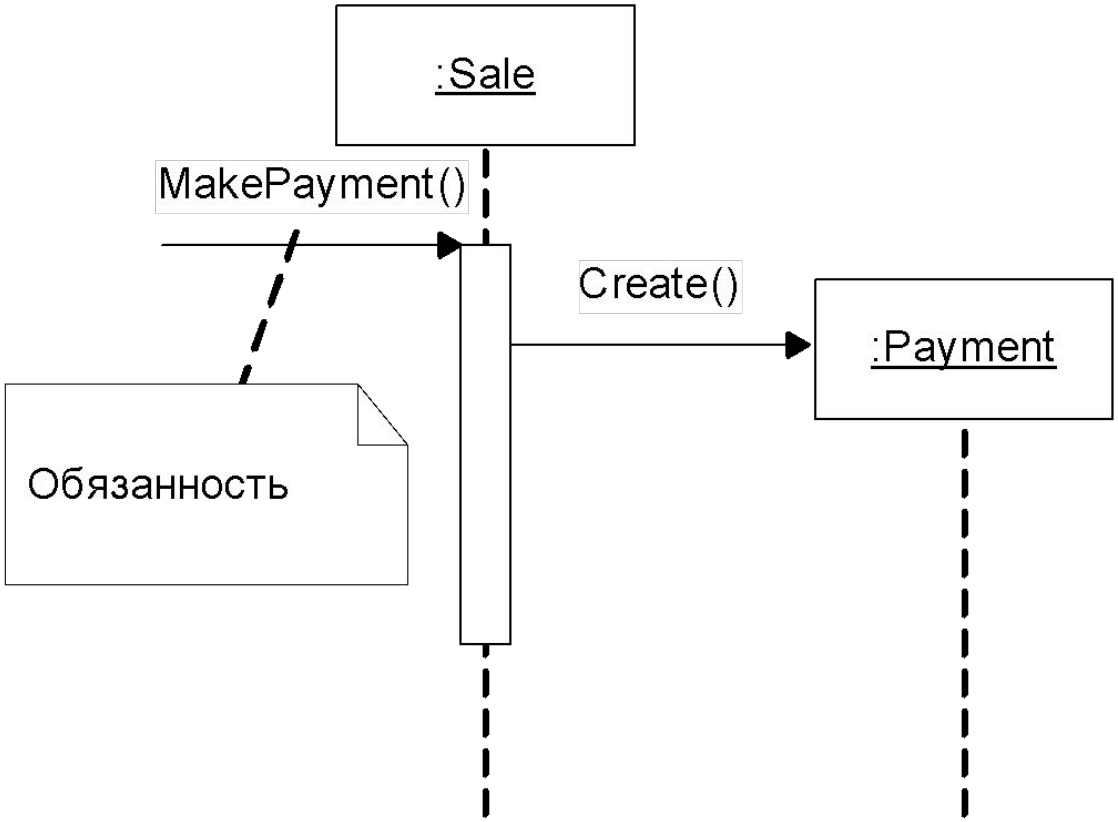
Обязанности

Операция =>
Обязанность

Обязанность ≠>
Операция

Обязанности

на диаграммах взаимодействия



GRASP

- Шаблоны **GRAS** или **GRASP** – **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
  - Информационный эксперт (Information Expert)
  - Создатель (Creator)
  - Высокое зацепление (High Cohesion)
  - Слабое связывание (Low Coupling)
  - Контроллер (Controller)
  - Полиморфизм (Polymorphism)
  - Чистая выдумка (Pure Fabrication)
  - Посредник (Indirection)
  - Сокрытие реализации (Protected Variations)

Информационный эксперт

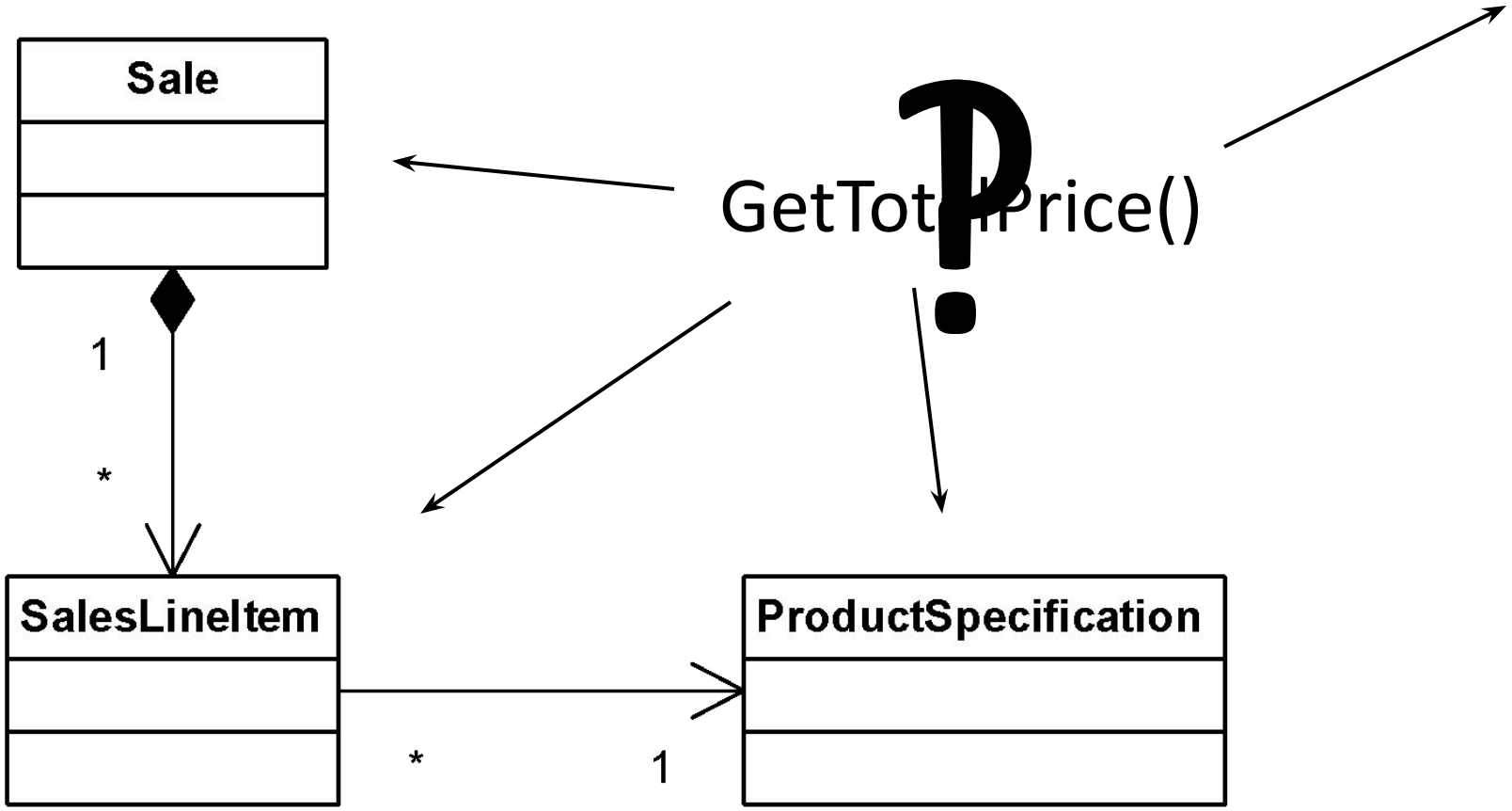
Проблема:

Необходимо выявить наиболее общий принцип распределения обязанностей между объектами.

Решение:

Назначить обязанность информационному эксперту – классу, у которого имеется информация, требуемая для выполнения обязанности.

Информационный эксперт



Информационный эксперт

Информационный эксперт

Требуемая обязанность:

Знать и предоставлять общую сумму продажи.

Итоговое распределение обязанностей:

Класс	Обязанность
Sale	Знание общей суммы продажи
SalesListItem	Знание промежуточной суммы для данного товара
ProductSpecification	Знание цены товара

Информационный эксперт
Выводы

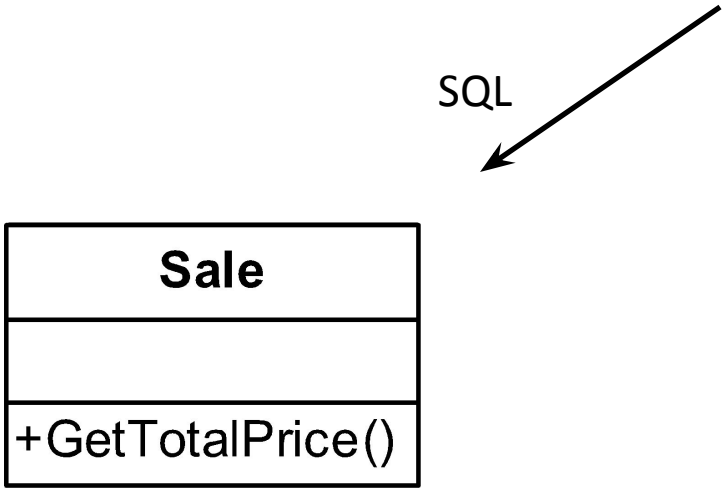
- Самый часто используемый шаблон распределения обязанностей.
- Аналогия с реальным миром.
- Существование «частичных» экспертов.

Информационный эксперт
Преимущества

- Поддерживает инкапсуляцию
- Поведение обеспечивается несколькими классами, содержащими требуемую информацию => простота понимания и поддержки.

Информационный эксперт
не следует применять

В тех случаях, когда это нарушает связывание и сцепление.



Создатель

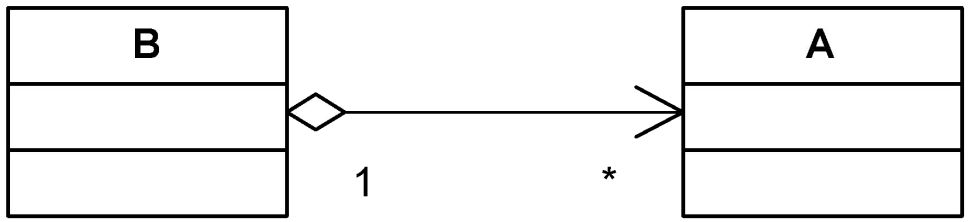
Проблема:

Кто должен отвечать за создание нового экземпляра некоторого класса?

Решение:

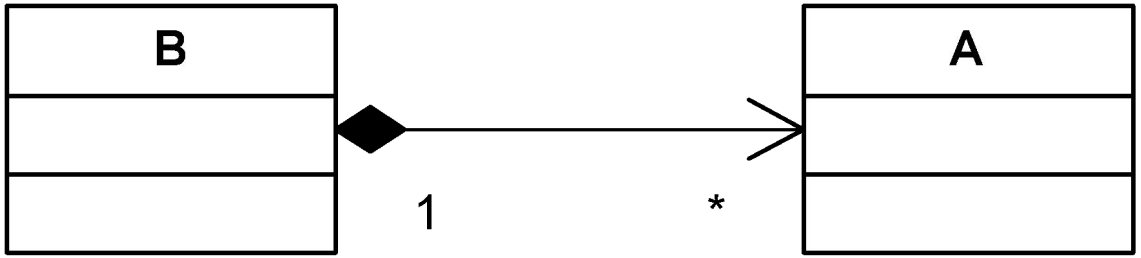
Назначить классу B обязанность создавать экземпляры класса A, если выполняется одно из следующих условий:

– Класс B *агрегирует* объекты A.

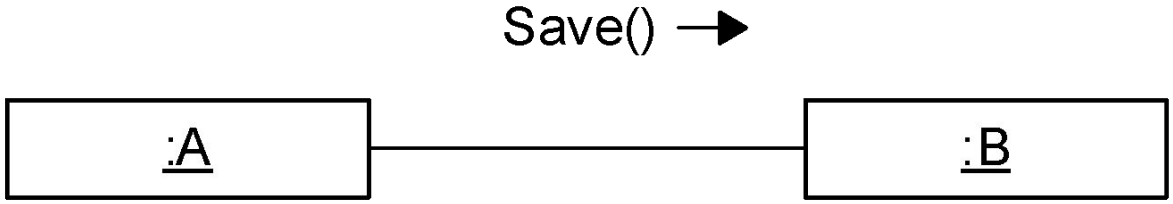


Создатель

– Класс B *содержит* объекты A.

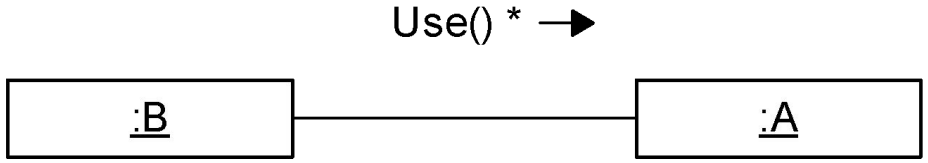


– Класс B *записывает* экземпляры объектов A.

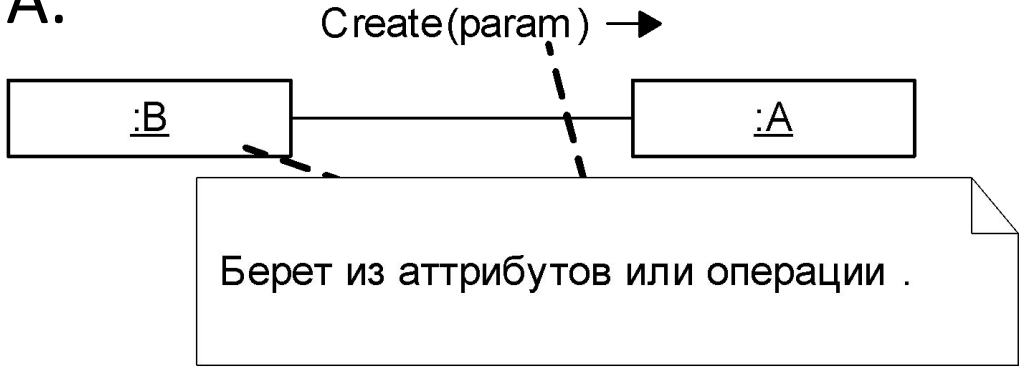


Создатель

– Класс B *активно использует* объекты A.

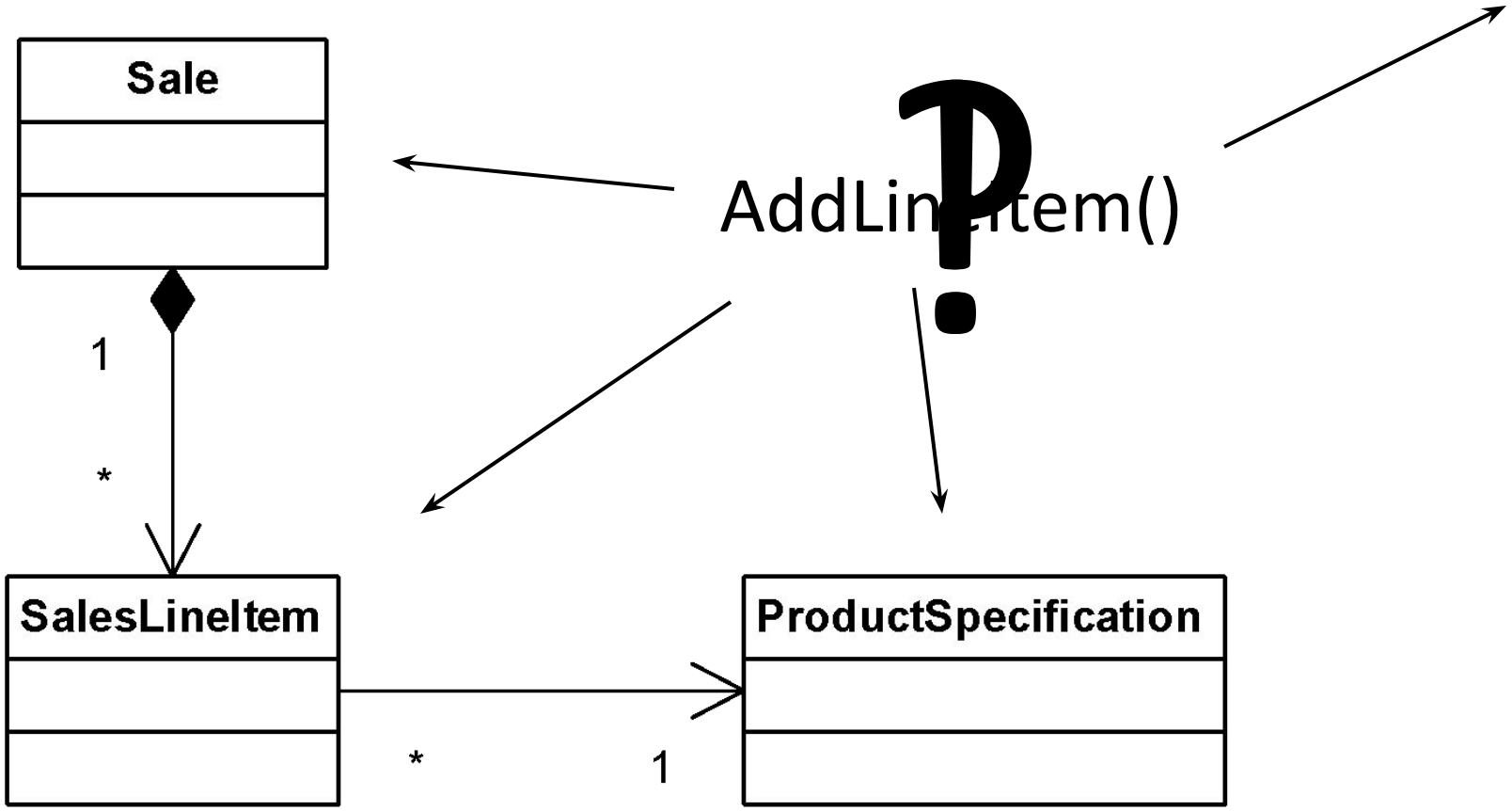


– Класс B *обладает данными инициализации* объектов A.

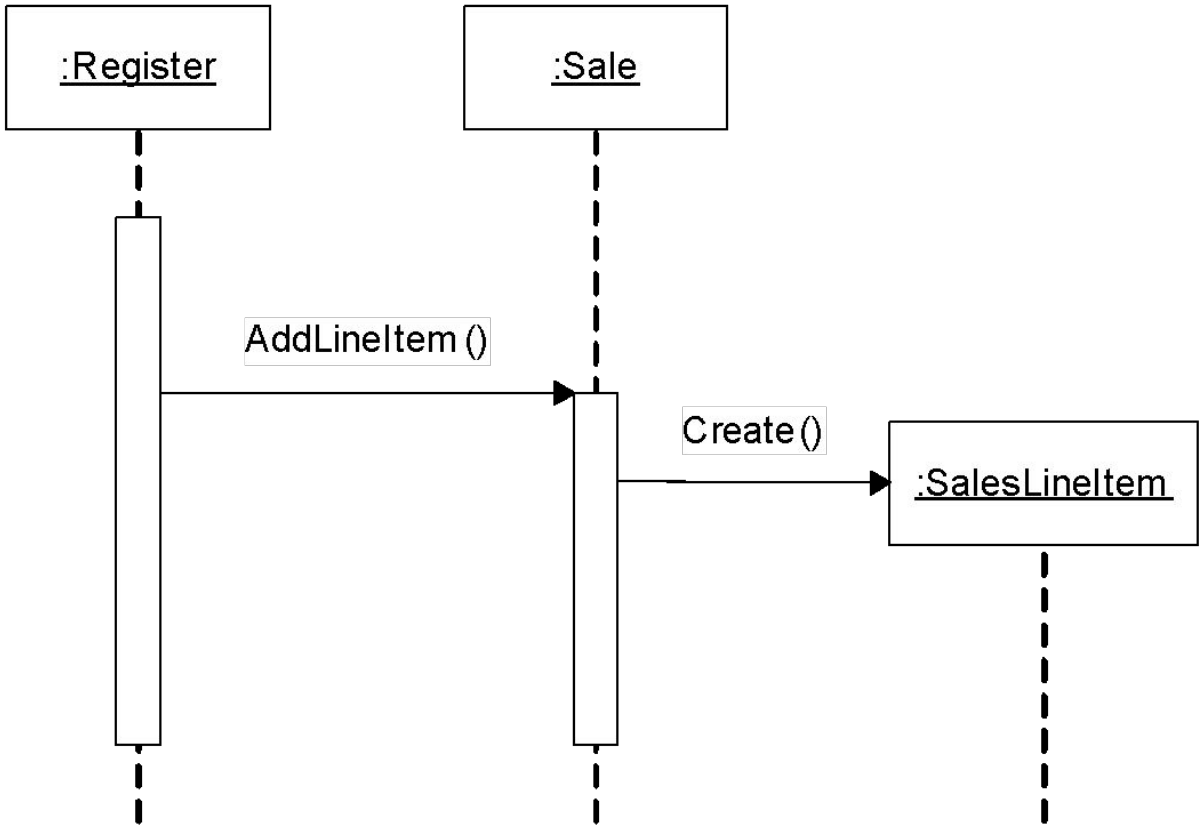


Класс B – *создатель* объектов A.

Создатель


Sale
1
*
AddLineItem()
?
SalesLineItem
ProductSpecification
*
1

Создатель



Создатель
Выводы

- Самый распространенный способ распределения обязанностей, связанных с созданием объектов – простота выявления объекта-создателя, при сохранении низкой степени связанности.
- Создатель, содержащий данные инициализации – шаблон Информационный эксперт.

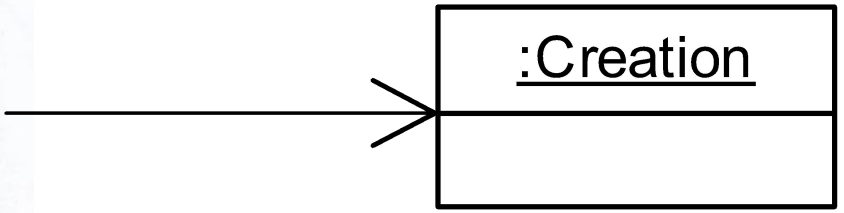
Создатель
не следует применять

Когда создание – сложная операция, выполняемая при реализации некоторого условия на основе каких-либо внешних свойств.

В этом случае предпочтительнее использовать шаблон Фабрика (Factory).

Создатель
Преимущества

- Не повышает степень связанности, так как созданный объект обычно оказывается видимым для объекта-со



Слабое связывание(Low Coupling)

Проблема:

Как обеспечить минимальную зависимость, минимальный риск изменений и повышенное повторное использование?

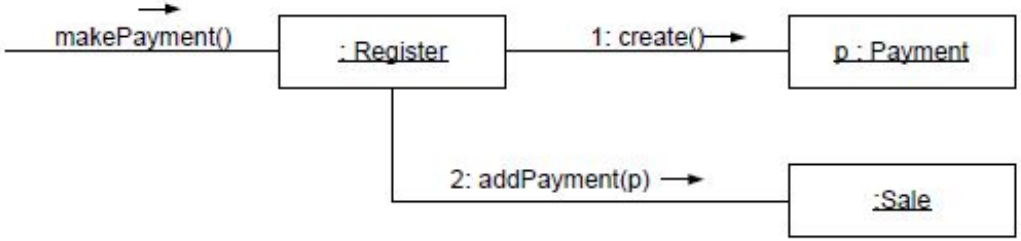
Связывание – мера того, насколько сильно элементы связаны, зависят друг от друга.

Сильное связывание приводит к проблемам:

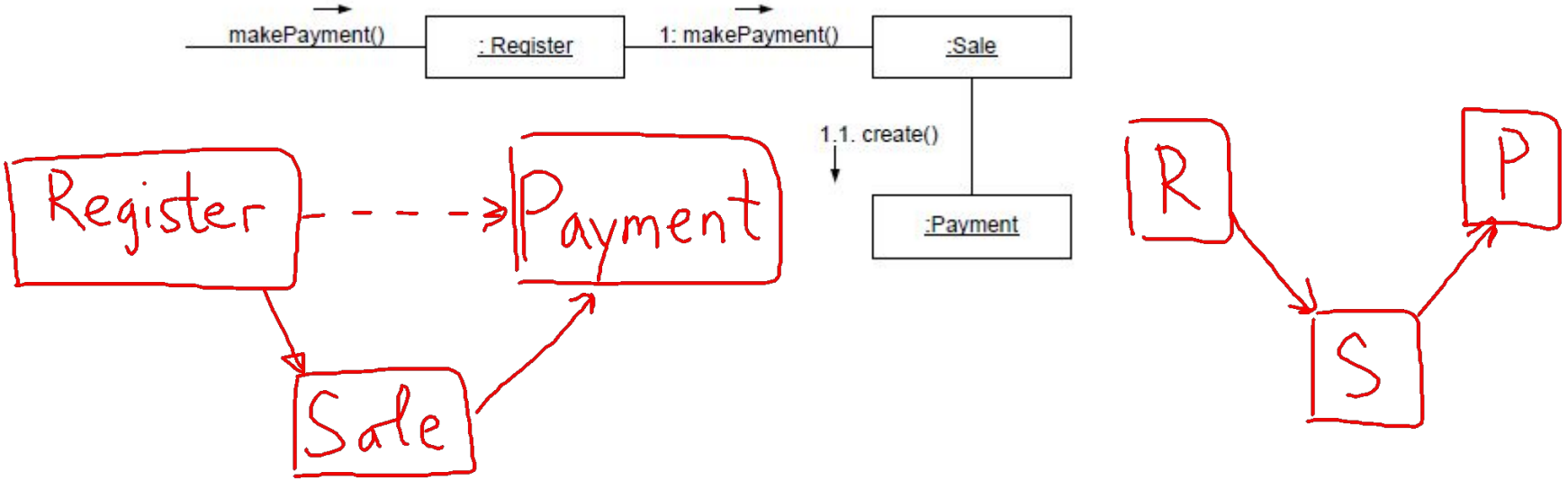
- Изменения в одном классе влекут необходимость в изменении другого
- Сложность понимания элемента в изоляции от других
- Сложность повторного использования

Слабое связывание(Low Coupling)

Паттерн «Создатель» нам диктует:



Но можно поразмыслить:



Слабое связывание(Low Coupling)

Классы ClassX и ClassY могут быть связаны:

- ClassX имеет атрибут, который ссылается на ClassY
- ClassX содержит операцию, которая ссылается на ClassY (использует экземпляр класса в качестве параметра, переменной или возвращаемого значения)
- ClassX – непосредственный потомок ClassY
- ClassY – интерфейс, а ClassX его реализует

Слабое связывание(Low Coupling)

Применение

- Применяется вместе с другими паттернами (такими как *Высокое зацепление* и *Эксперт)*
- Применяется в случае «нестабильности» в эволюционном смысле слова. **Не** применяется в отношении к классам статических библиотек.
- Применяется в разумных пределах

Слабое связывание(Low Coupling)

Результаты

1. Решение проблемы
    1. Классы не зависят от изменений в других классах
    2. Классы легко понимаемы вне контекста
    3. Классы удобны для повторного использования

Высокое зацепление (High cohesion)

Проблема

Как управлять необъятным?

Зацепление – мера того, насколько взаимосвязаны и целенаправленны обязанности, возложенные на элемент.

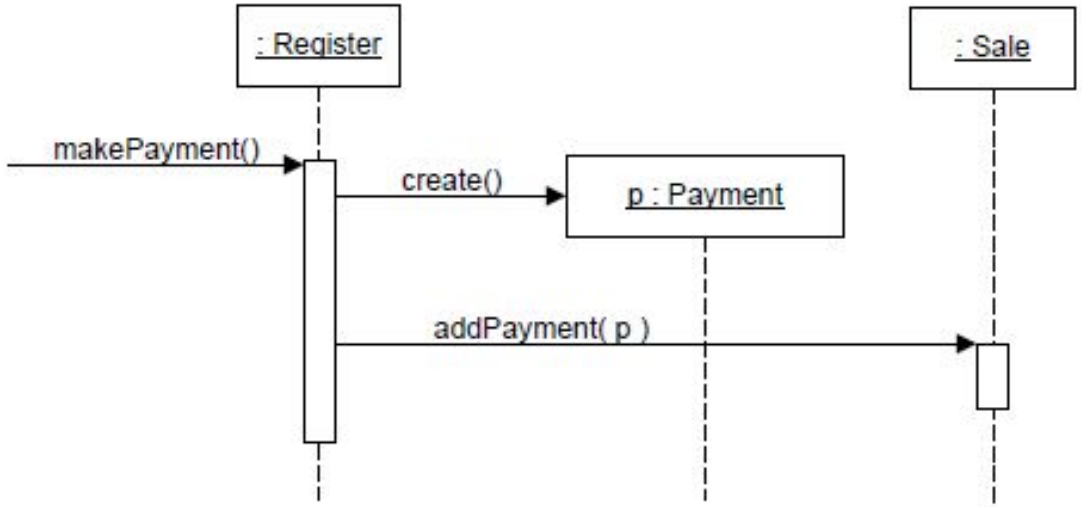
Элемент с взаимосвязанными обязанностями обладает *высоким зацеплением.*

Слабое зацепление приводит к проблемам:

- Класс трудно «познать»
- Класс трудно использовать повторно
- Класс трудно поддерживать
- Класс «нежен»

Высокое зацепление (High cohesion)

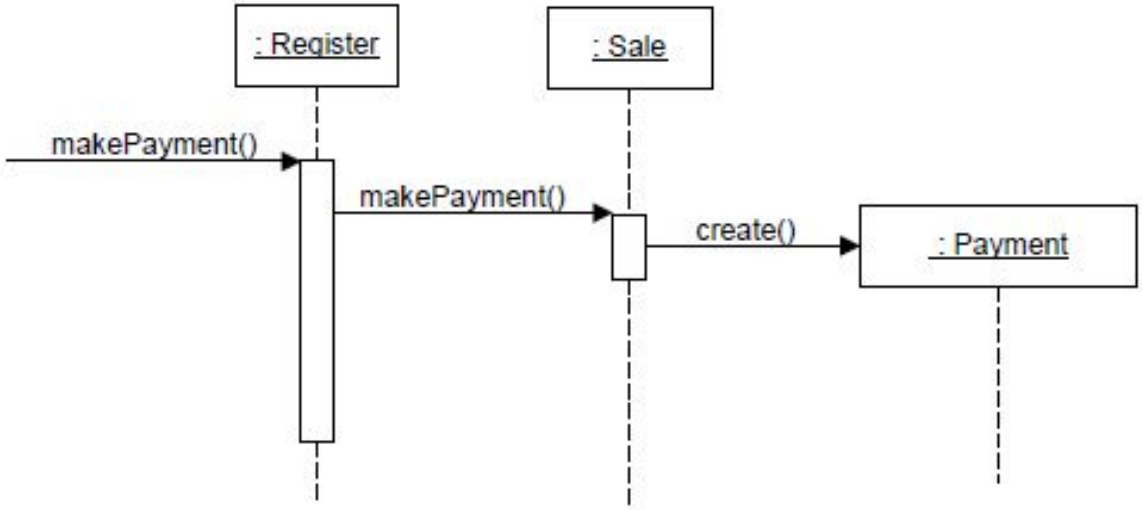
Мы снова доверяемся паттерну «Создатель»



Высокое зацепление (High cohesion)

Но тогда Register может разбухнуть!

Попробуем все же вот так:



Высокое зацепление (High cohesion)

Классификация зацеплений (by Grady Booch)

- Очень низкое зацепление *(RDB-RPC-Interface)*
- Низкое зацепление *(RDB-Interface)*
- Высокое зацепление
- Умеренное зацепление *(Company)*

Высокое зацепление (High cohesion)

Зацепление и связывание

Инь и Янь программной инженерии

Исключения

- Упрощение обслуживания одним человеком *(ICQ Expert)*
- Удаленные операции

Высокое зацепление (High cohesion)

Результаты

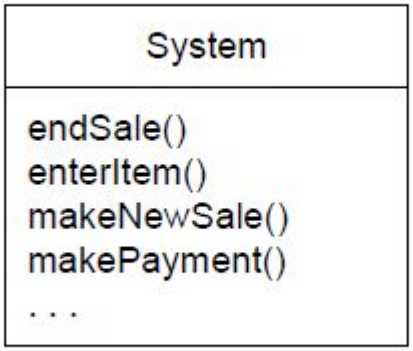
- Ясность и простота понимания дизайна
- Упрощение поддержки и эволюции
- Слабое связывание в подарок
- Повторное использование

Контроллер

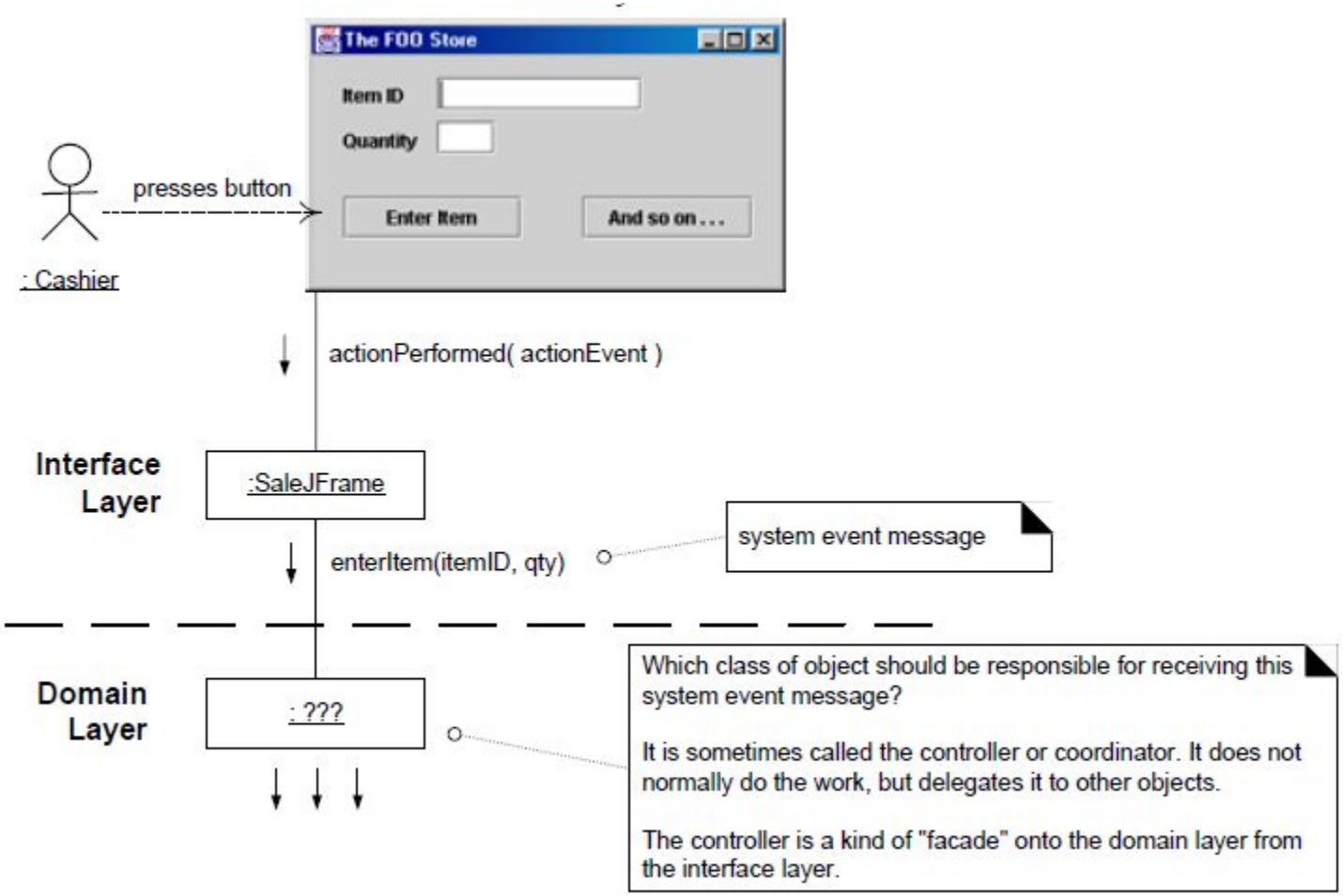
Проблема

Кто должен быть ответственен за обработку системных событий ввода?

Системное сообщение ввода– сообщение, сгенерированное внешним актером. Они ассоциированы с **операциями си** событие.



Контроллер


The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
Interface Layer
:SaleJFrame
enterItem(itemID, qty)
system event message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.
The controller is a kind of "facade" onto the domain layer from the interface layer.

Контроллер

Варианты контроллера

- **Фасадный контроллер -** представляет всю систему(или подсистему) в совокупности
  - *Register*
  - Название, соответствующее системе *(ChessGame)*
- **Контроллер варианта использования** представляет ВИ, в пределах которого возникают сообщения.
  - <UseCaseName>Handler

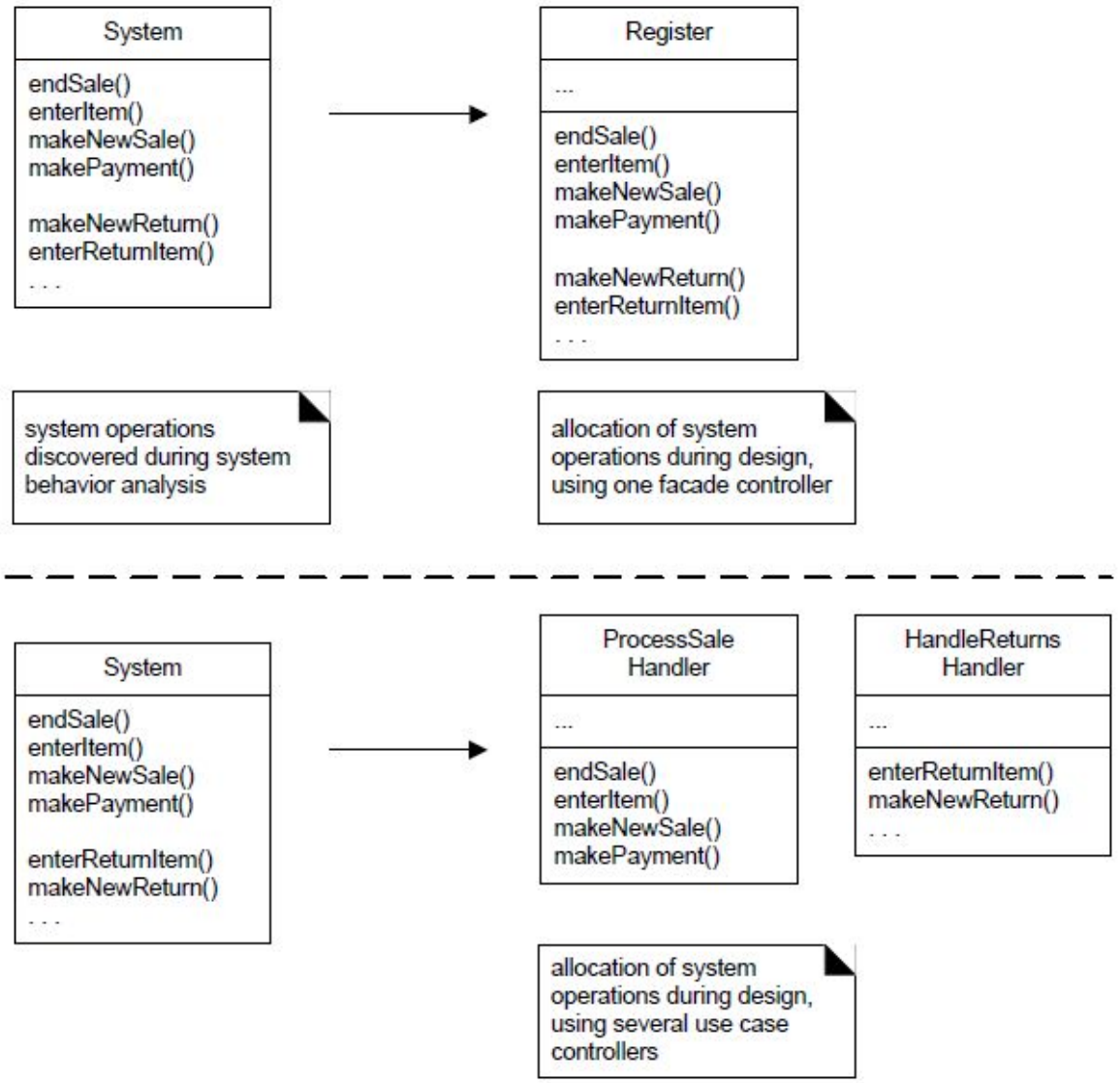
Контроллер

Выбор контроллера

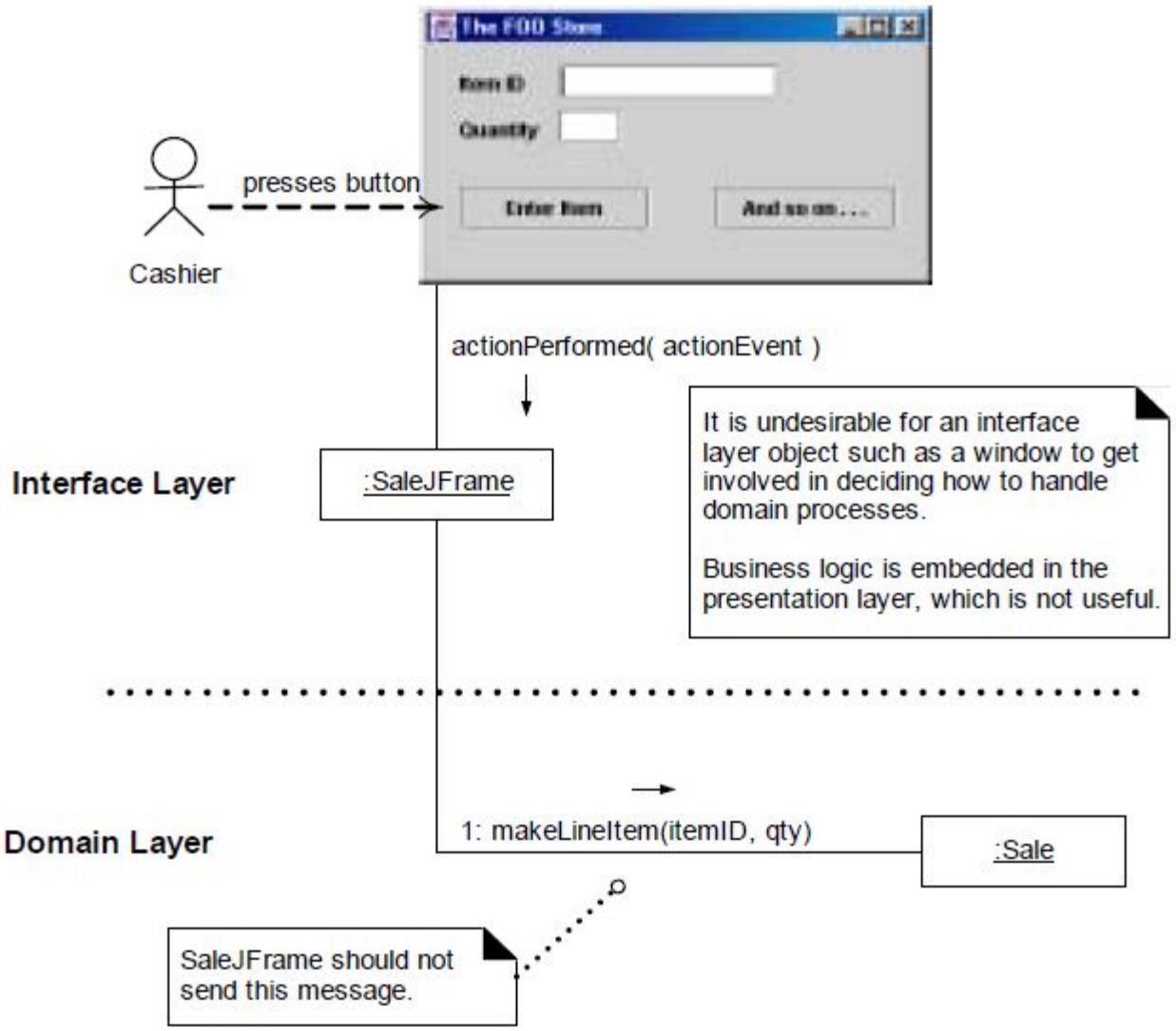
- **Фасадный контроллер**
  - Малое количество событий
  - UI не может перенаправить системные сообщения на переменные контроллеры (как в системах обработки сообщений)
- **Контроллер ВИ**
  - Большое количество событий
  - Необходимо поддерживать логику ВИ

UI не должны быть ответственны за события ввода!

Контроллер


System
endSale()
enterItem()
makeNewSale()
makePayment()
makeNewReturn()
enterReturnItem()
. . .
Register
...
endSale()
enterItem()
makeNewSale()
makePayment()
makeNewReturn()
enterReturnItem()
. . .
system operations discovered during system behavior analysis
allocation of system operations during design, using one facade controller
System
endSale()
enterItem()
makeNewSale()
makePayment()
enterReturnItem()
makeNewReturn()
. . .
ProcessSale Handler
...
endSale()
enterItem()
makeNewSale()
makePayment()
HandleReturns Handler
...
enterReturnItem()
makeNewReturn()
. . .
allocation of system operations during design, using several use case controllers

Не контроллер



Контроллер

Результаты

- Логика приложения отделена от интерфейса
- Повторное использование
- Учет состояния варианта использования

Контроллер

Проблемы

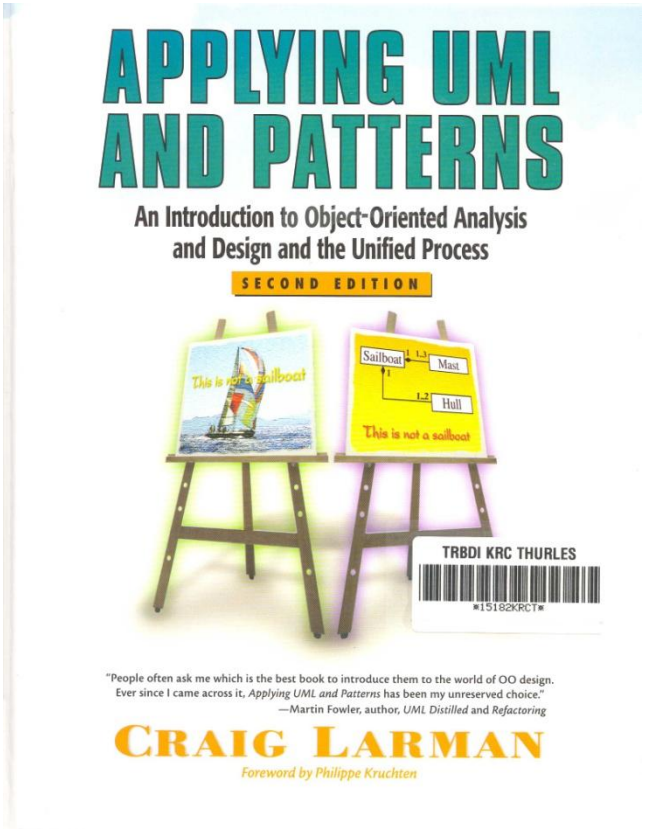
- Один единственный контроллер, принимающий орды сообщений
- Контроллер берет на себя часть обязанностей по обработке сообщений
- Контроллер содержит много аттрибутов
- Контроллер хранит дубликаты информации из системы

Решения

- Увеличить количество контроллеров
- Проектировать контроллер чтобы он только делегировал обязанности другим объектам

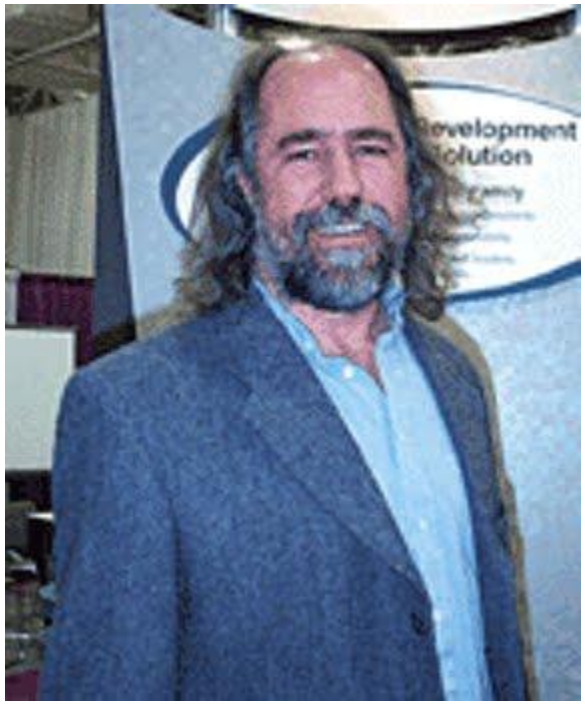
Благодарность

- Creig Larman, за его интереснейшую книгу “Applying UML and patterns”.



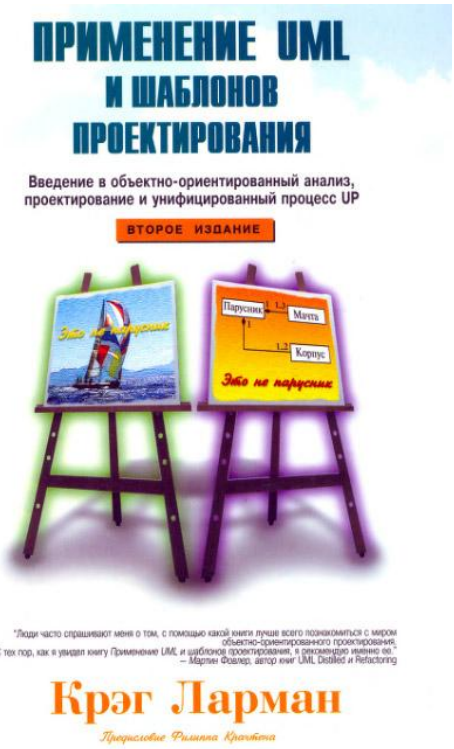
Благодарность

- Гради Бучу, за то, что учил Крэга Лармана. И за то, что научил.



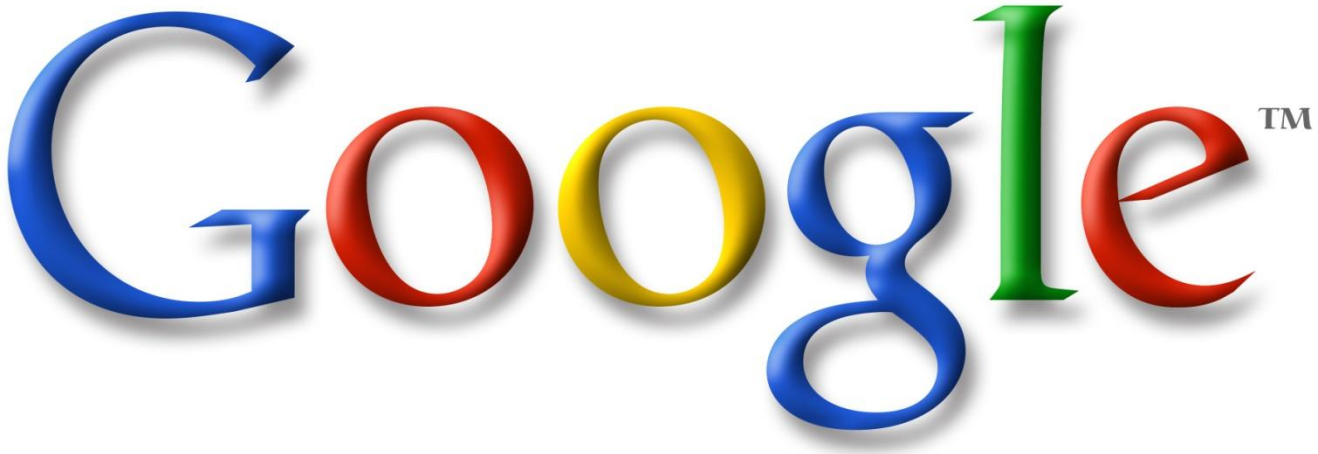
Благодарность

- А.Ю. Шелестову, за перевод замечательной книги Крэга Ларман на русский язык («Применение UML и шаблонов проектирован



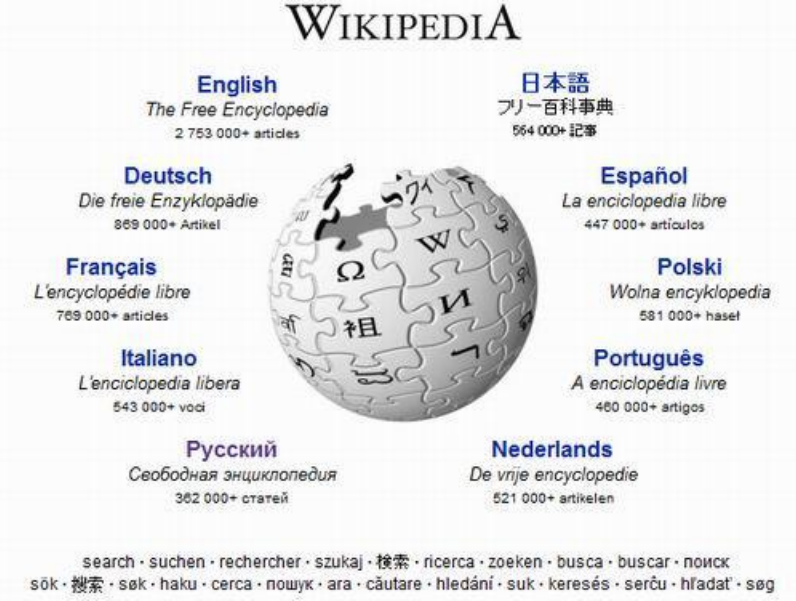
Благодарность

- Google Search, за его релевантную выдачу.



Благодарность

- Ru.wikipedia.org, за понимание темы в целом.



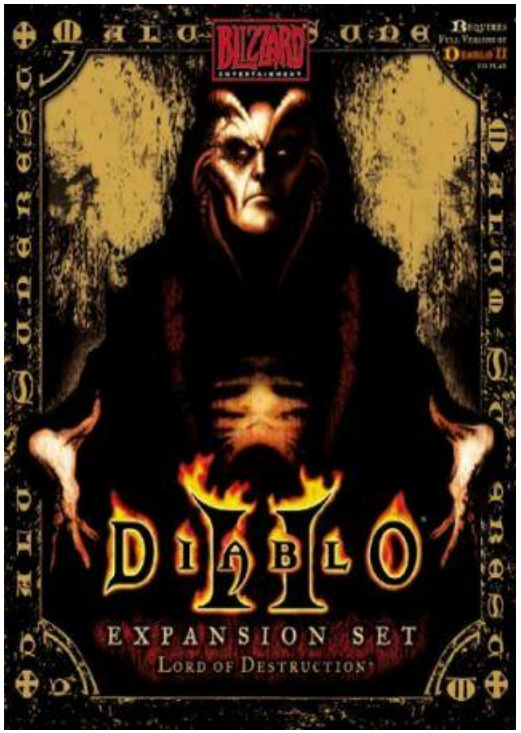
Благодарность

- Пакету Microsoft Office 2007, за прекрасные инструменты создания презентаций

Microsoft® Office

Благодарность

- И, конечно, отдельное спасибо компании Blizzard, за ее бессмертные игры.



Спасибо за внимание