

Объектно-ориентированный Анализ и Дизайн

Часть 3



Шаблоны проектирования

Архитектурные шаблоны

Rational Unified Process

Документирование бизнес-процессов на UML

Управление конфигурацией ИТ проекта



7. Шаблоны проектирования

✓ Шаблон проектирования – стандартное решение стандартной проблемы.

- Название
- Проблема – описание ситуаций, в которых применяется шаблон
- Решение
- Последствия



Виды шаблонов

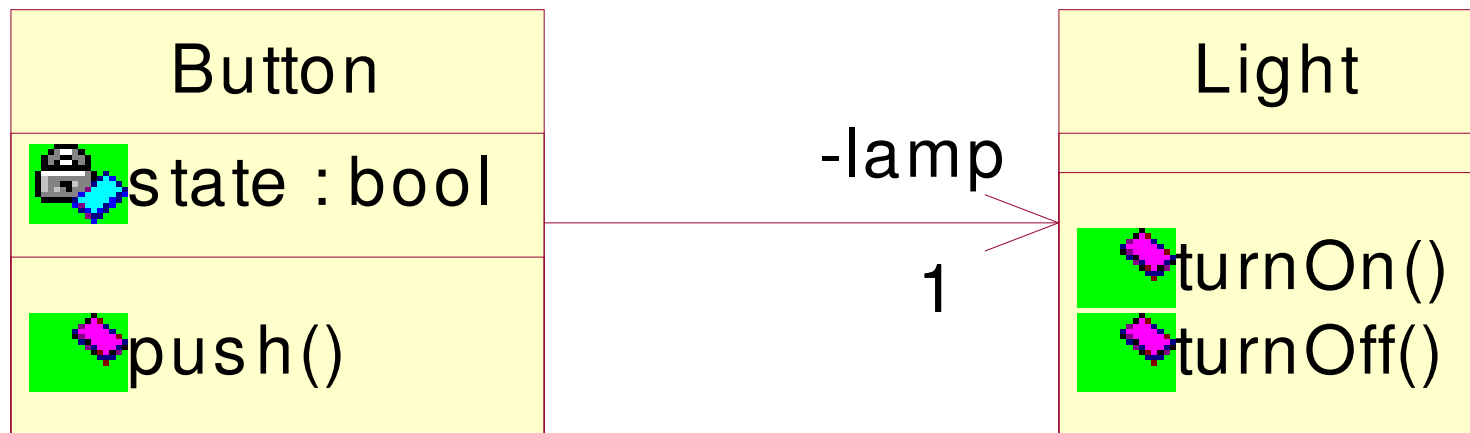
Е. Gamma:

- Конструкционные шаблоны – абстрагируют создание объектов
 - Abstract Factory, Factory Method, Singleton

- Структурные шаблоны – решают проблемы композиции
 - Adapter, Bridge, Decorator, Proxy

- Поведенческие шаблоны – алгоритмы и распределение ответственности между объектами
 - Command, Iterator, Observer, State, Strategy

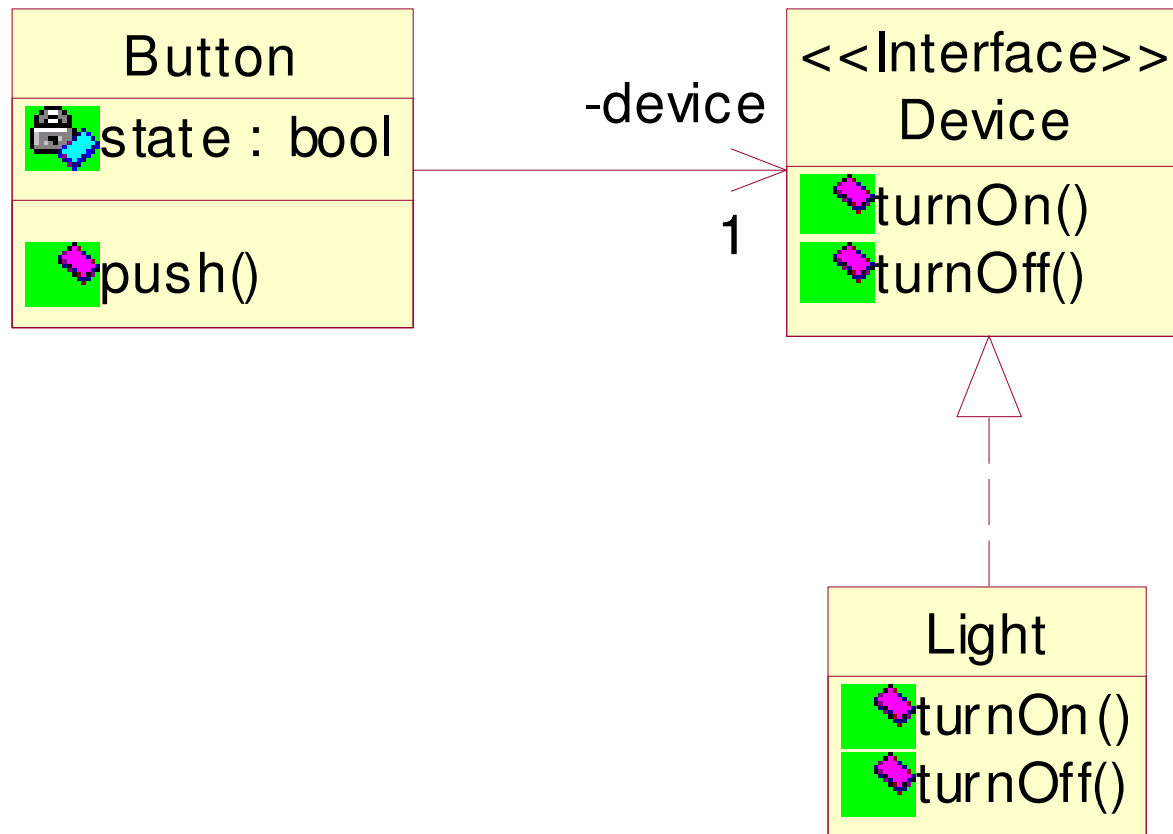
Abstract Server



Проблема:

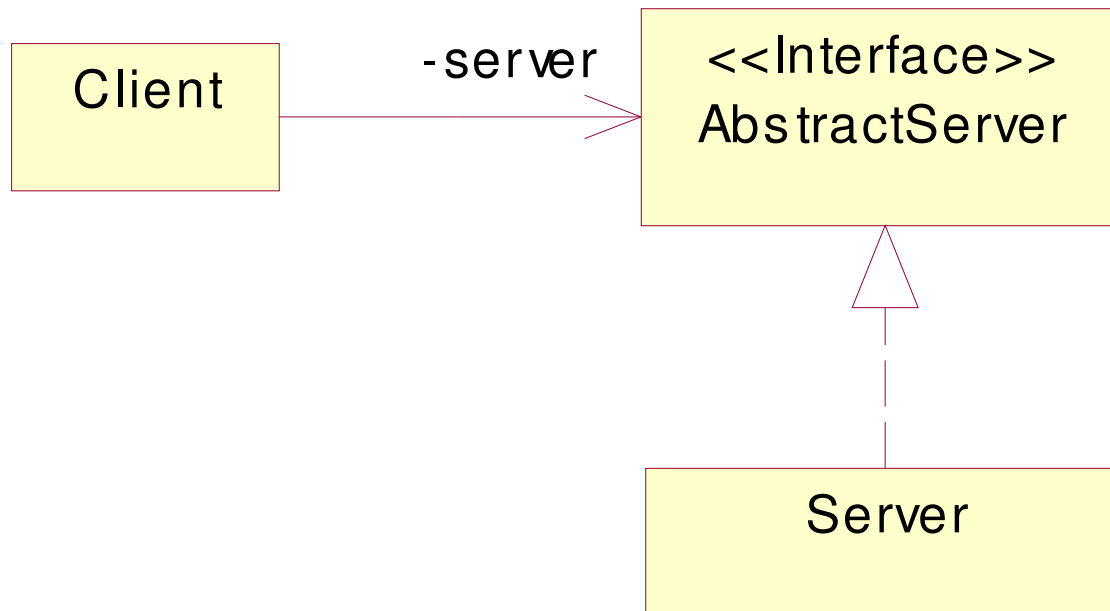
- Button нельзя использовать в контексте не использующем Light

Решение



Решение: разорвать зависимость между Button and Light путем вставки интерфейса

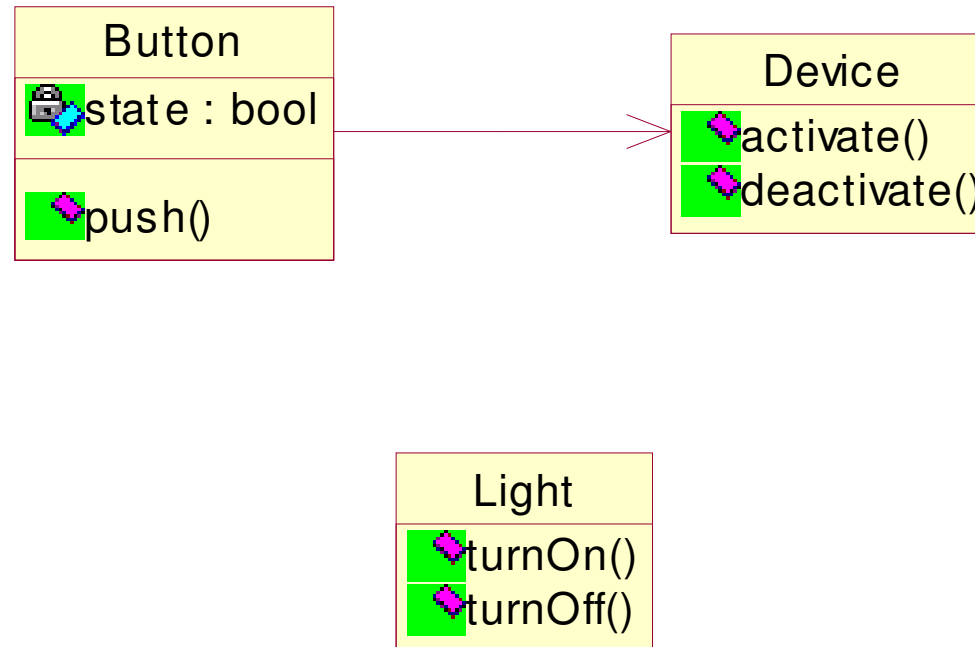
Abstract Server Pattern



Плюсы:

- устраняет зависимость клиента от сервера
- сервера могут изменяться не влияя на клиентов
- устраняет нарушение DIP

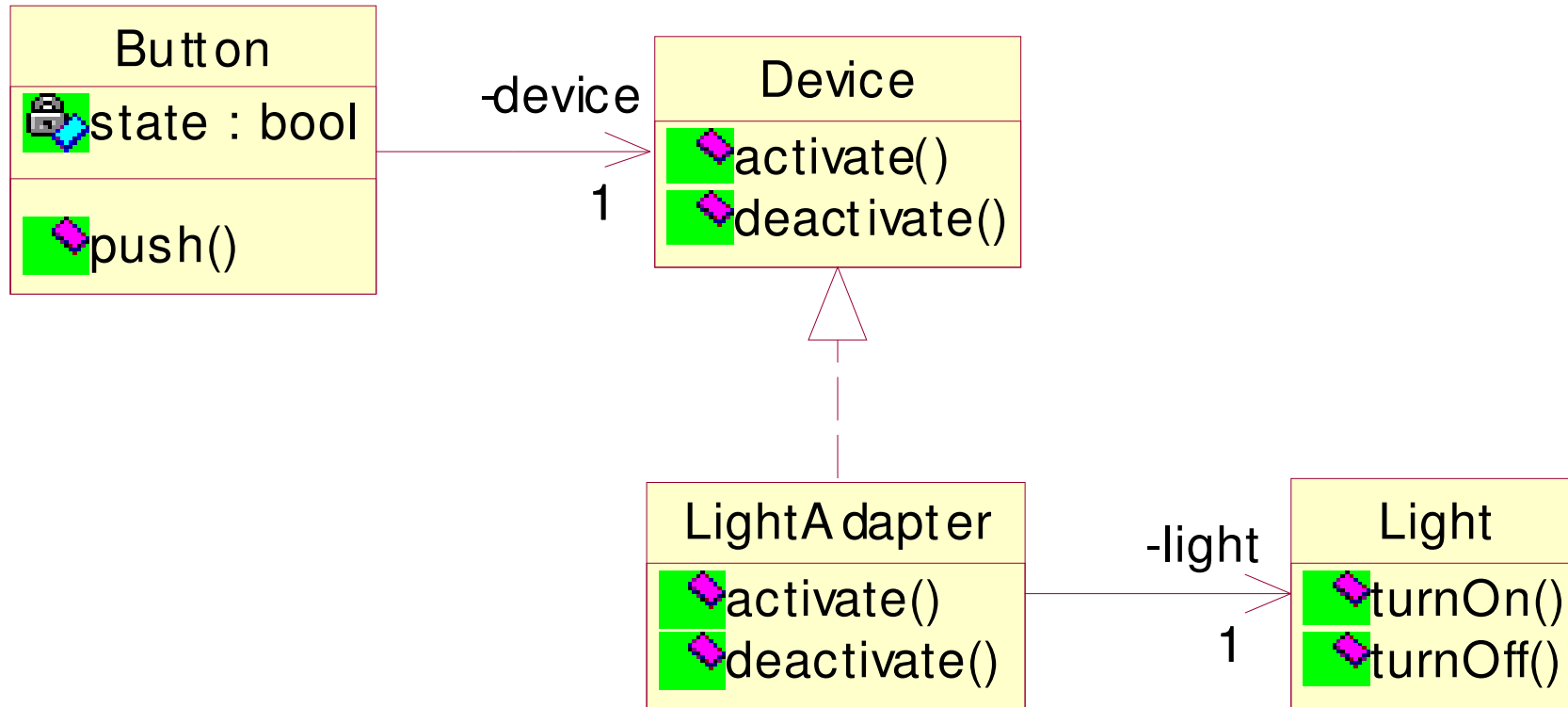
Adapter



Проблема:

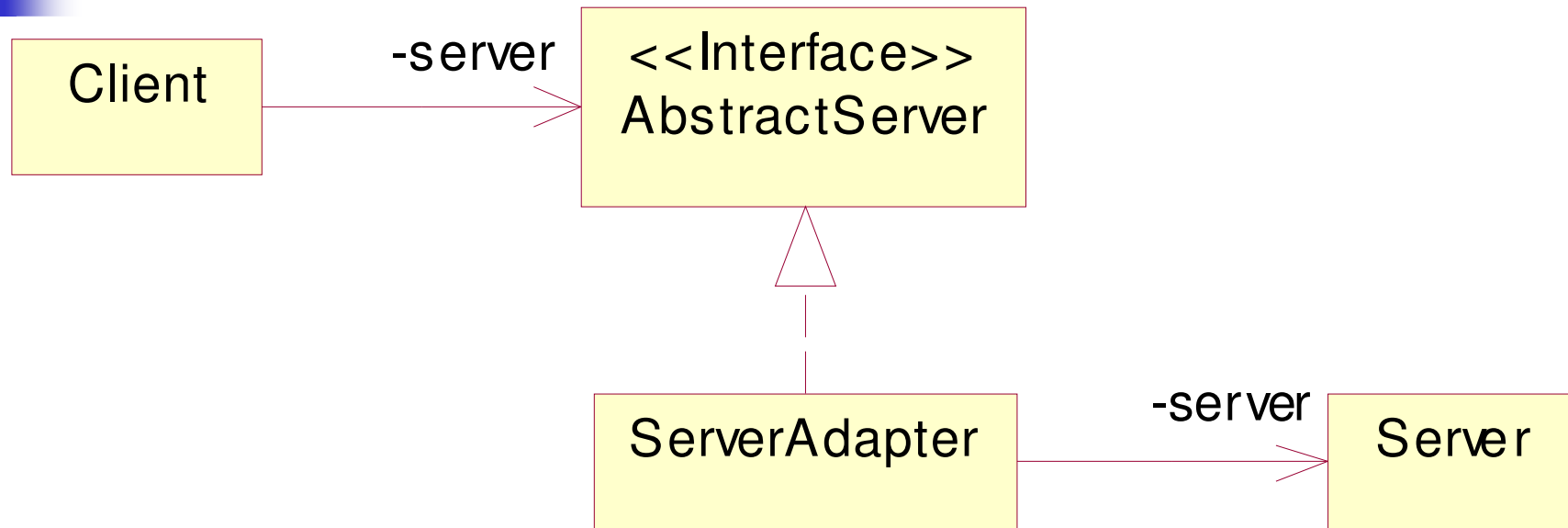
- Light уже существует и не может наследовать Device
- Device может уже иметь методы activate/deactivate

Решение



- Адаптер конвертирует один интерфейс в другой

Pattern: Adapter

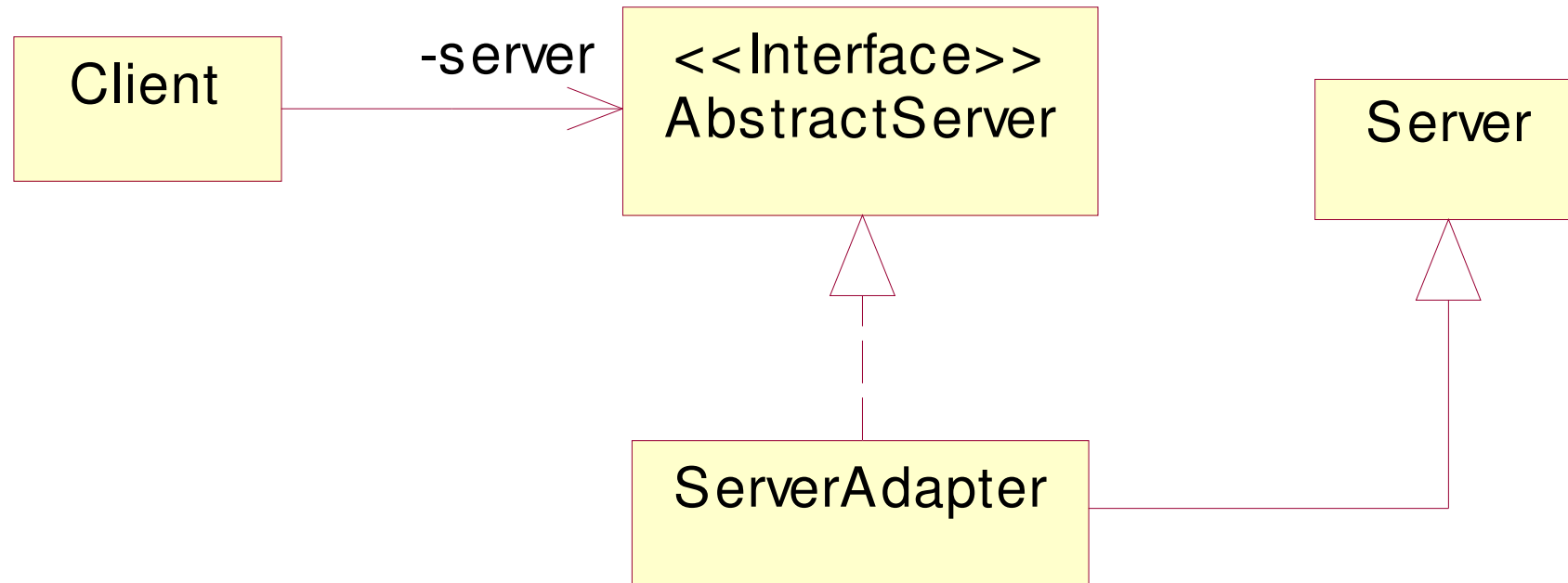


Также известен как: Wrapper

Плюсы:

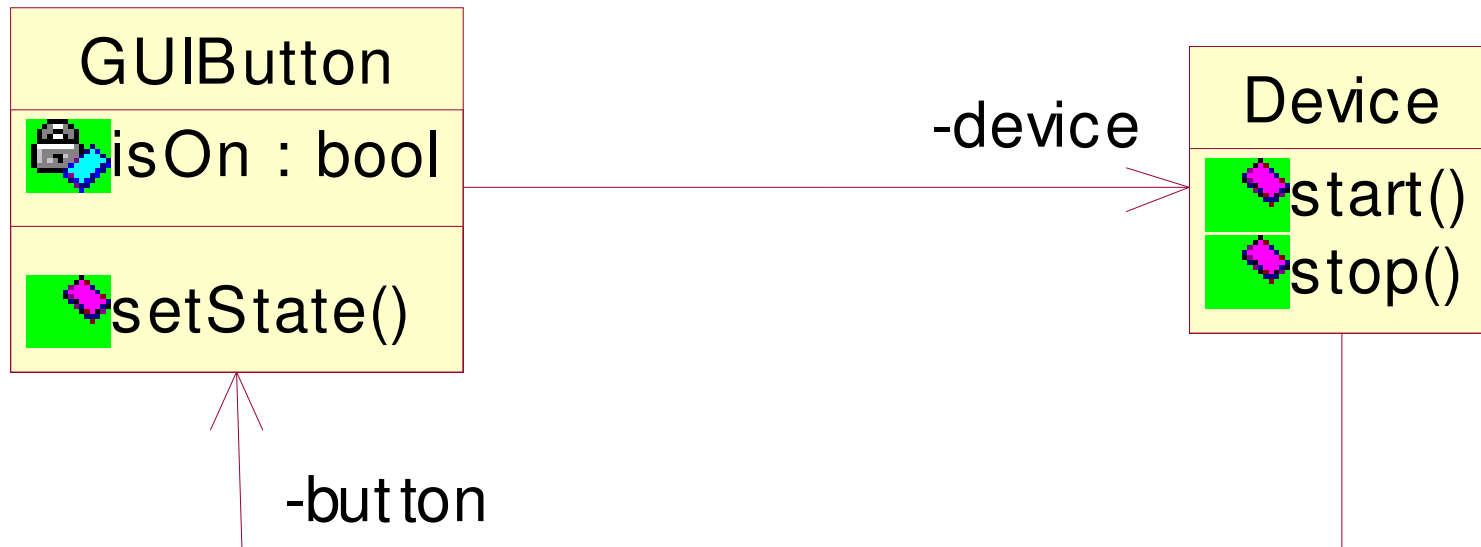
- разрыв зависимости между client and server когда сервер уже существует
- позволяет подменять сервера

Pattern: Adapter



Вариант, позволяющий переопределить методы сервера

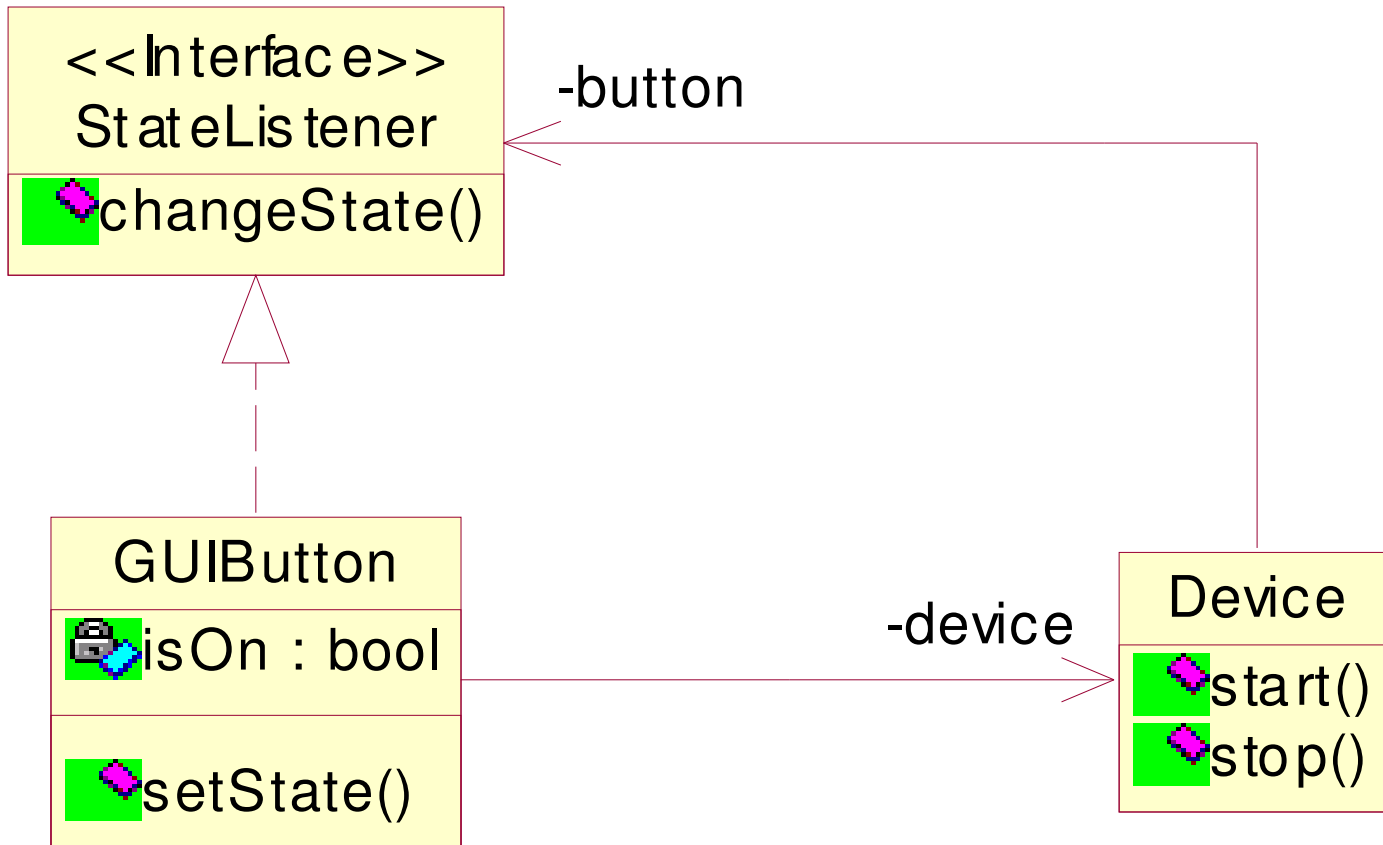
Abstract Client



Проблема:

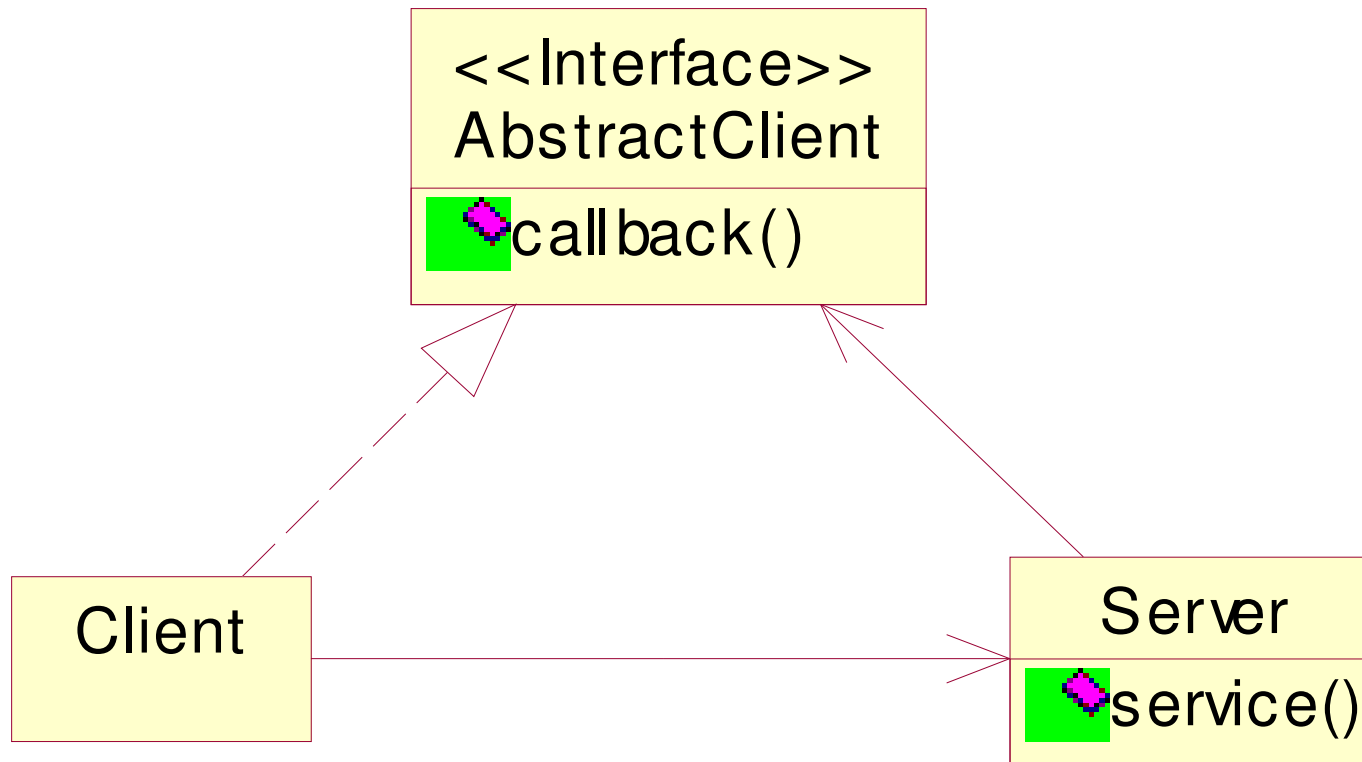
- Server-у нужно послать сообщение GUIButton
- GUI Button невозможно переиспользовать
- Callback метод делает Device также непереиспользуемым

Решение



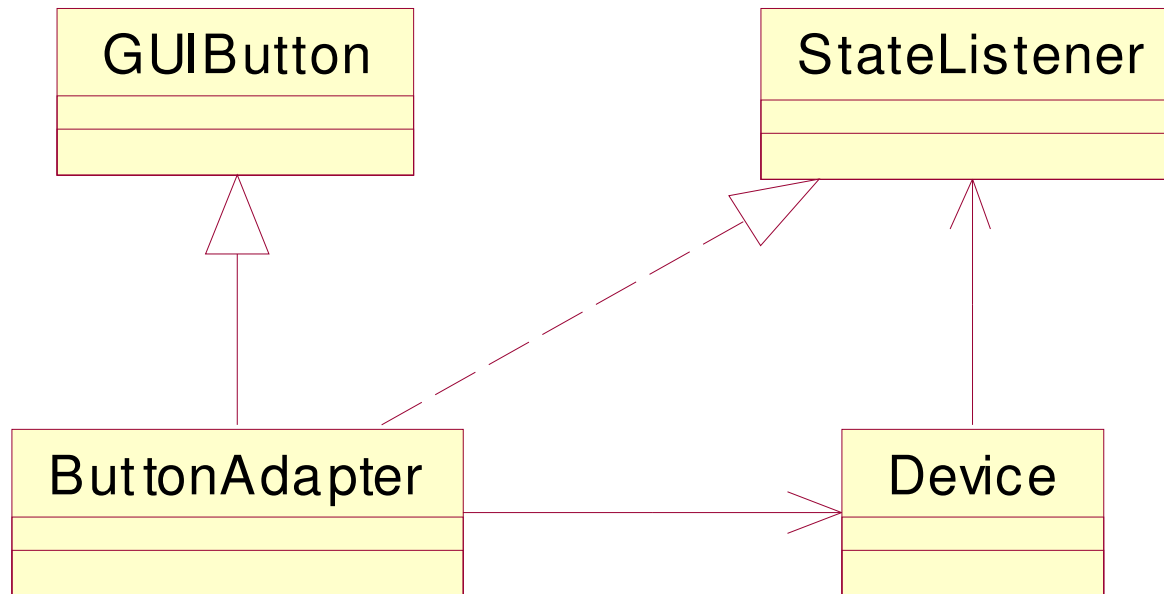
✓ Server (Device) предоставляет клиенту interface для сообщений

Pattern: Abstract Client



- ✓ **Q:** В каком пакете должен находиться `AbstractClient` ?
- ✓ **A:** Пакет `Server`-а

Adapted Client



Проблема:

GUIButton) не реализует StateListener

Решение:

использование Adapter pattern совместно с Abstract Client



Singleton

Проблема:


- Как много объектов класса Database требуется в простой программе?
- Очевидно, один
- Скорее всего, если кто-то создаст еще, и еще один такой объект – будут сюрпризы




Решение: не давать возможность создавать такие объекты

- сделать конструктор приватным.
- сделать сам класс ответственным за создание объектов своего типа

Singleton

CompanyDb

 \$ db : Database

 Instance()
 CompanyDb()
 getEmployee()

```
class CompanyDb {
private:
    static CompanyDb* db;
    CompanyDb();
    ~CompanyDb();
public:
    static CompanyDb* Instance() {
        if( 0 == db ) return db = new CompanyDb();
        return db;
    }
    Employee* getEmployee( const char* name );
};
```

Используйте singleton в случаях, когда:

- в системе должен существовать только один объект класса
- этот объект должен быть доступен из любого места программы:

`CompanyDb::Instance()->getEmployee("Alex");`



Monostate

- ✓ Решает ту же проблему что и Singleton
- ✓ Все поля - static
- ✓ Constructor и destructor - private

```
Class CompanyDb {  
    private:  
        static Db db;  
        CompanyDb();  
    public:  
        static Employee* getEmployee( const char* name );  
};  
Db CompanyDb::db = Db( "CompanyName" );  
Employee* CompanyDb::getEmployee( const char* name ) {  
    return db.find( "Employee", name );  
}  
Usage: Employee* emp = CompanyDb::getEmployee( "Leha" );
```



Singleton vs. Monostate

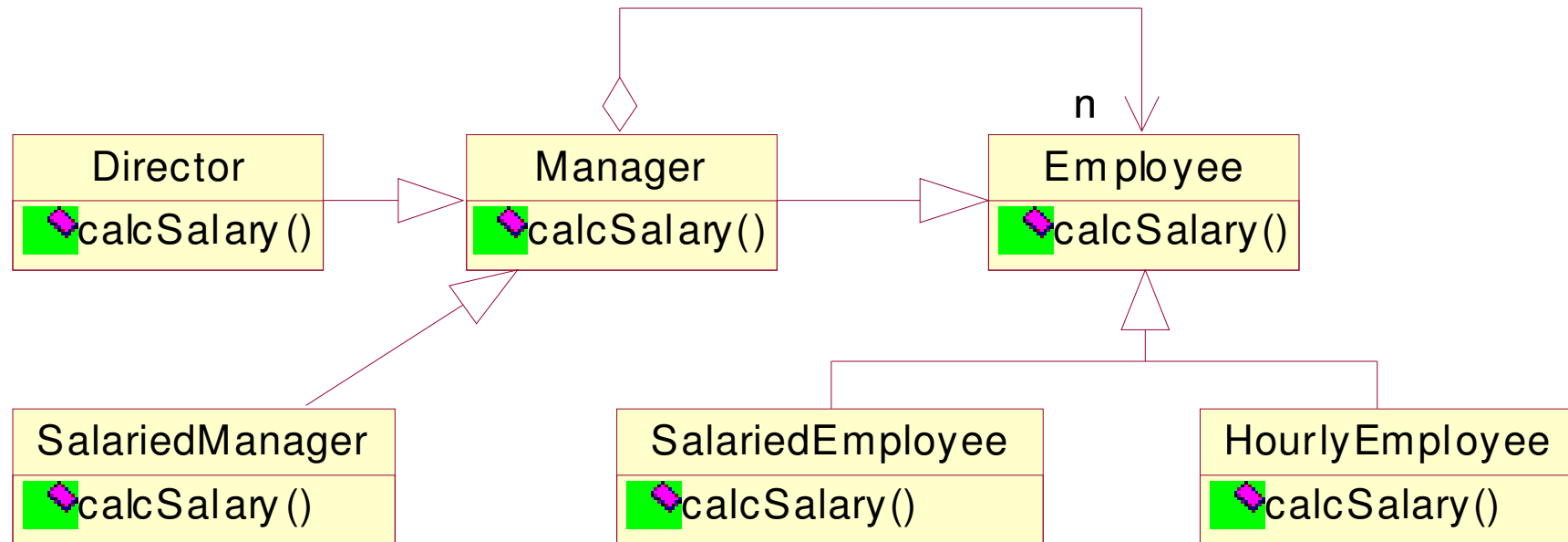
Construction:

- Singleton: отложенное конструирование (lazy construction)
 - Don't pay unless you need it!
- Monostate - сконструирован всегда
- Singleton может иметь нетривиальный конструктор
- Monostate не может иметь сложного конструктора

Destruction:

- Singleton - по запросу
- Monostate – по завершению программы

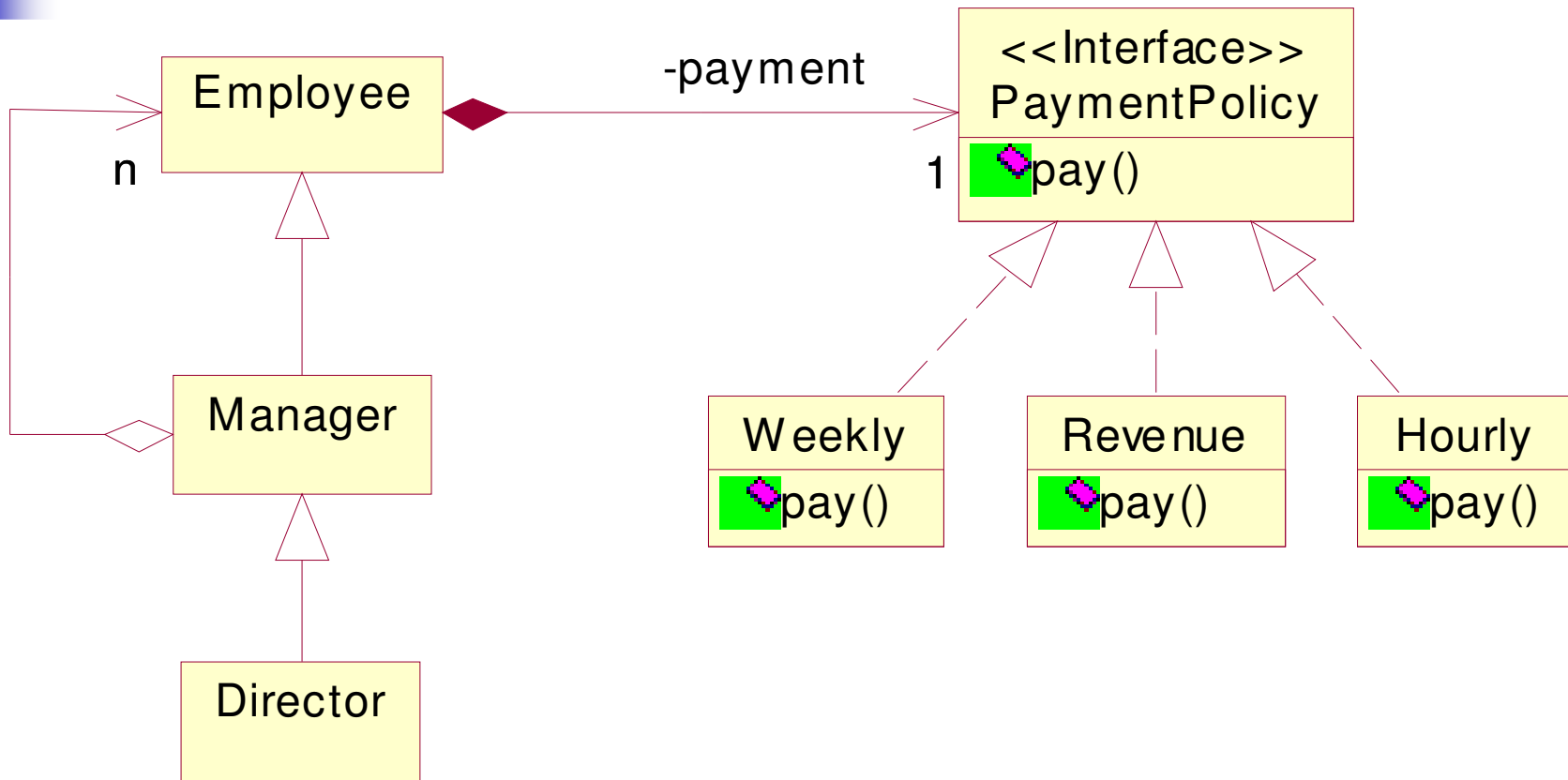
Strategy



Проблема:

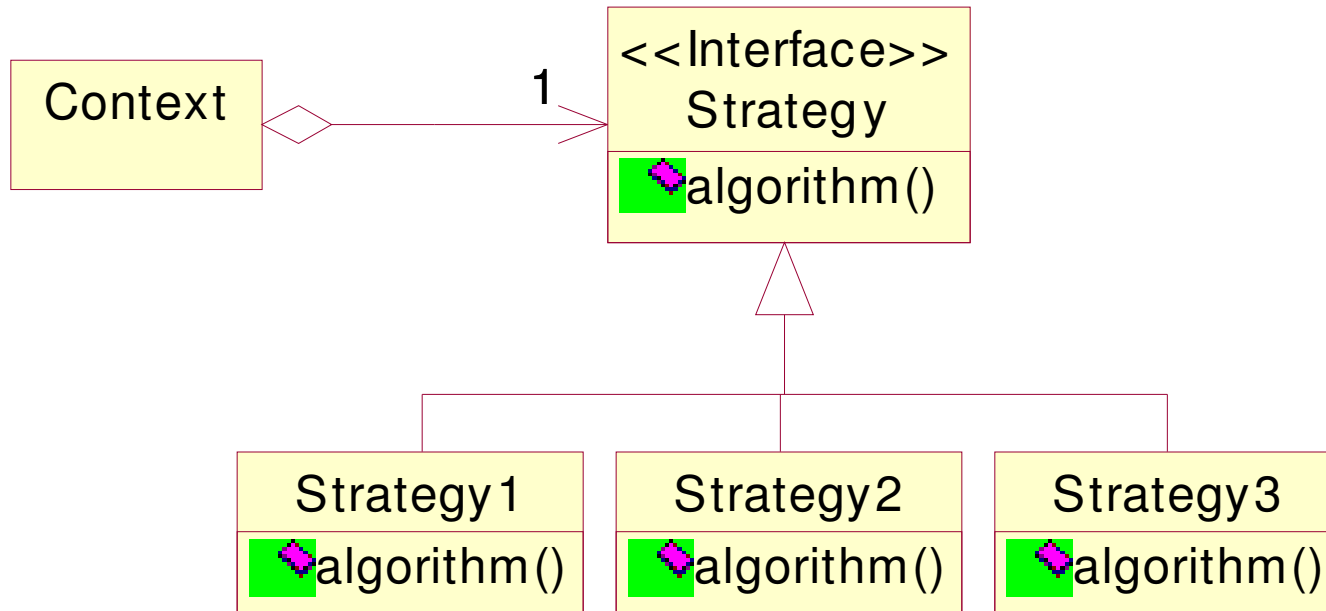
- различные сотрудники оплачиваются по разному
- как добавить hourly manager ?

Решение



✓ Алгоритм начисления зарплаты вынесен в PaymentPolicy

Strategy Pattern



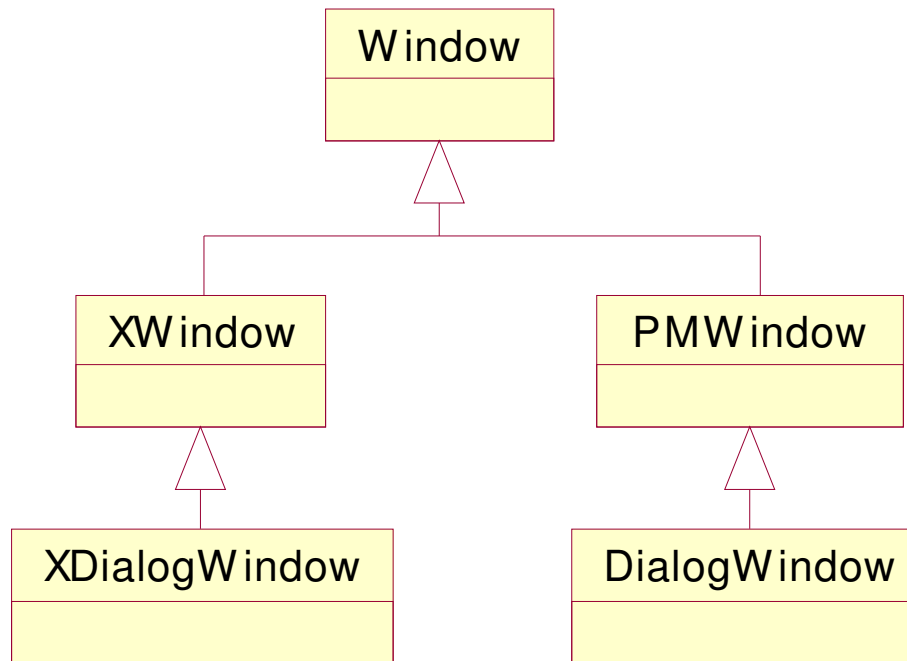
Также известен как: **Policy**

Плюсы:

- можно добавлять различные стратегии
- Context закрыт от модификации стратегий (выполняется OCP)



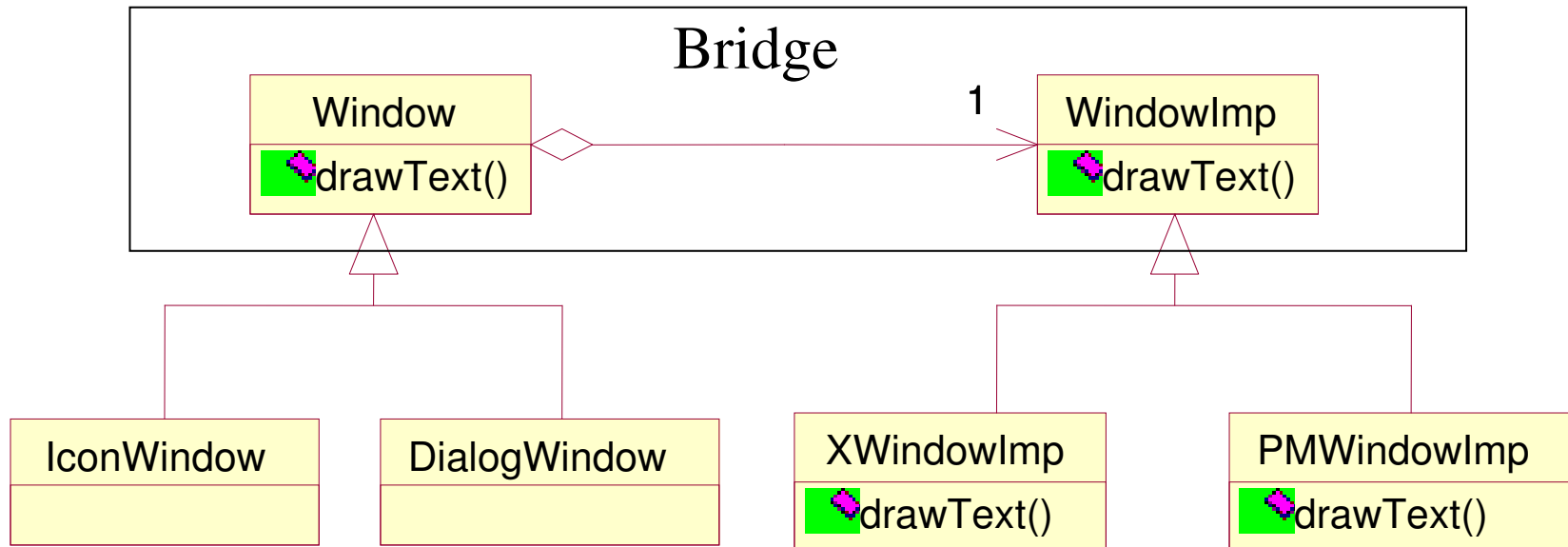
Bridge



Проблема:

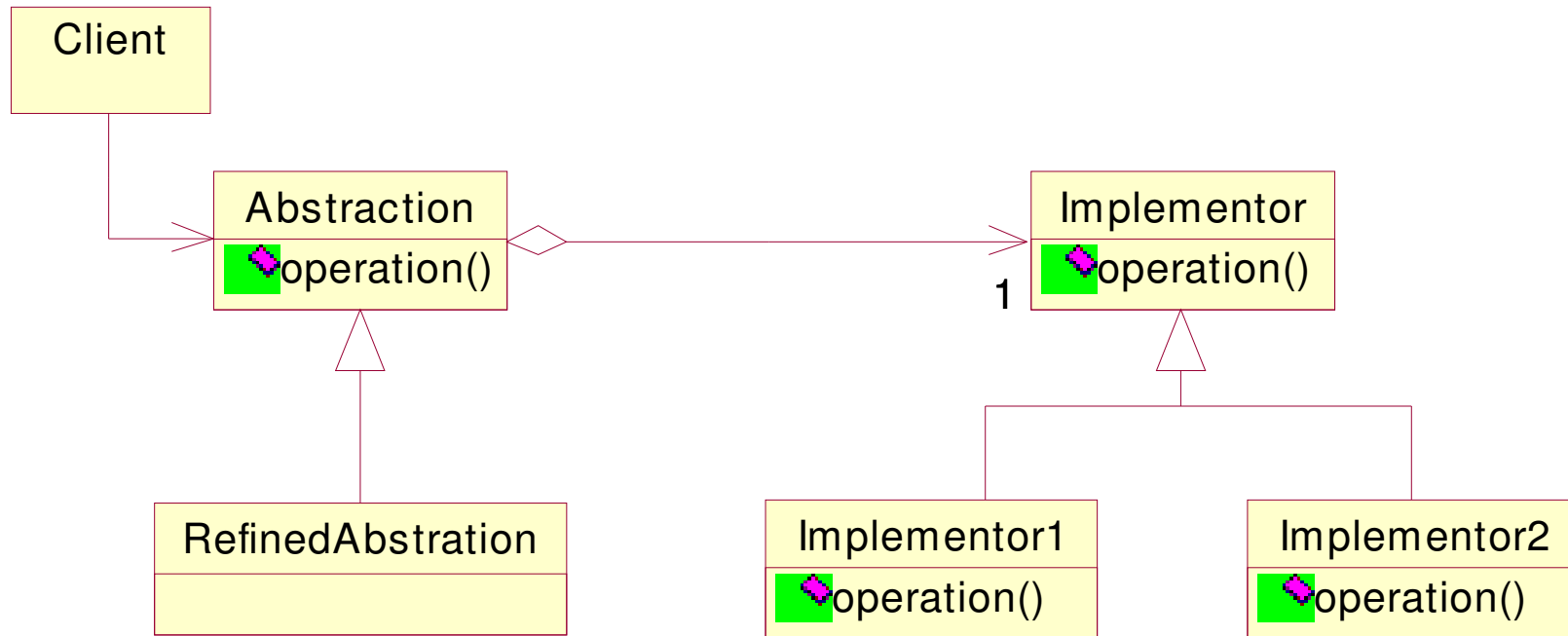
- Код зависит от платформы
- Для поддержки новой платформы надо воспроизвести всю иерархию

Solution



- ✓ Все операции подклассов Window реализованы в терминах операций из WindowImp
 - ✓ Window и его подклассы платформно-независимы
- Q:** Как сделать код независимым от подклассов WindowImp?
- A:** использовать фабрику

Bridge Pattern

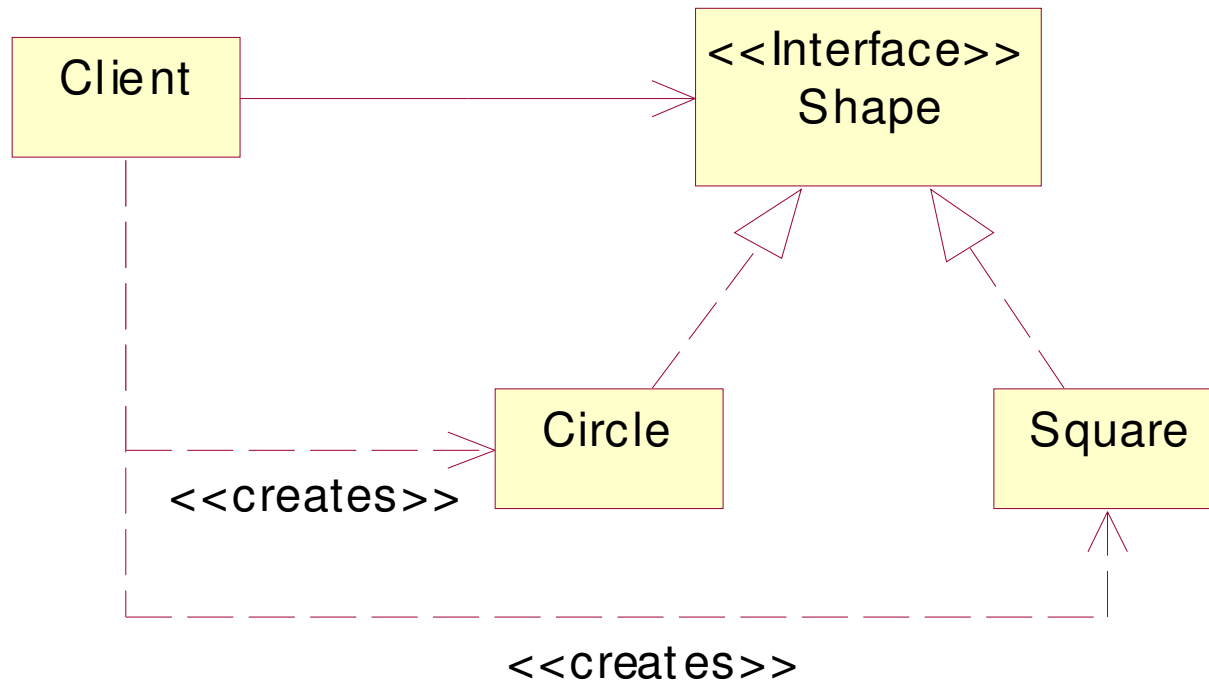


Также известен как: Handle/Body

Плюсы:

- Устраняет нарушение DIP & OCP

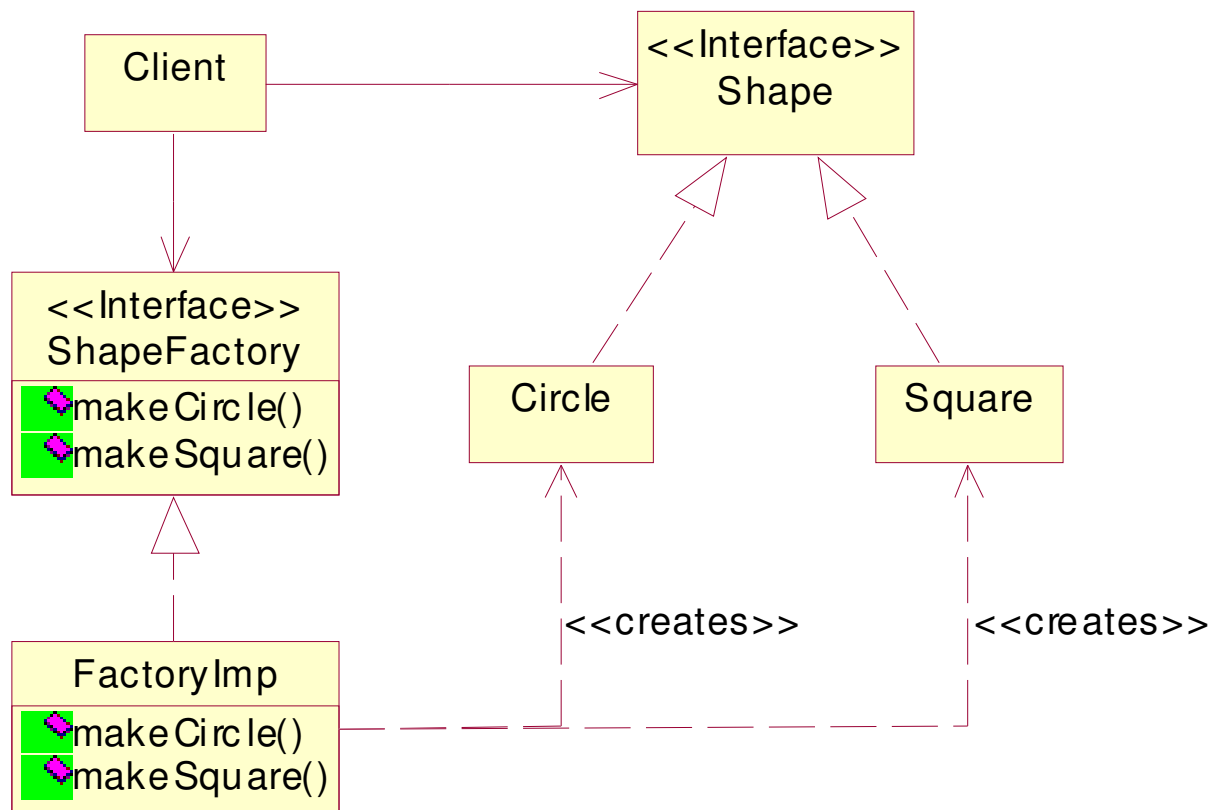
Abstract Factory



Проблема:

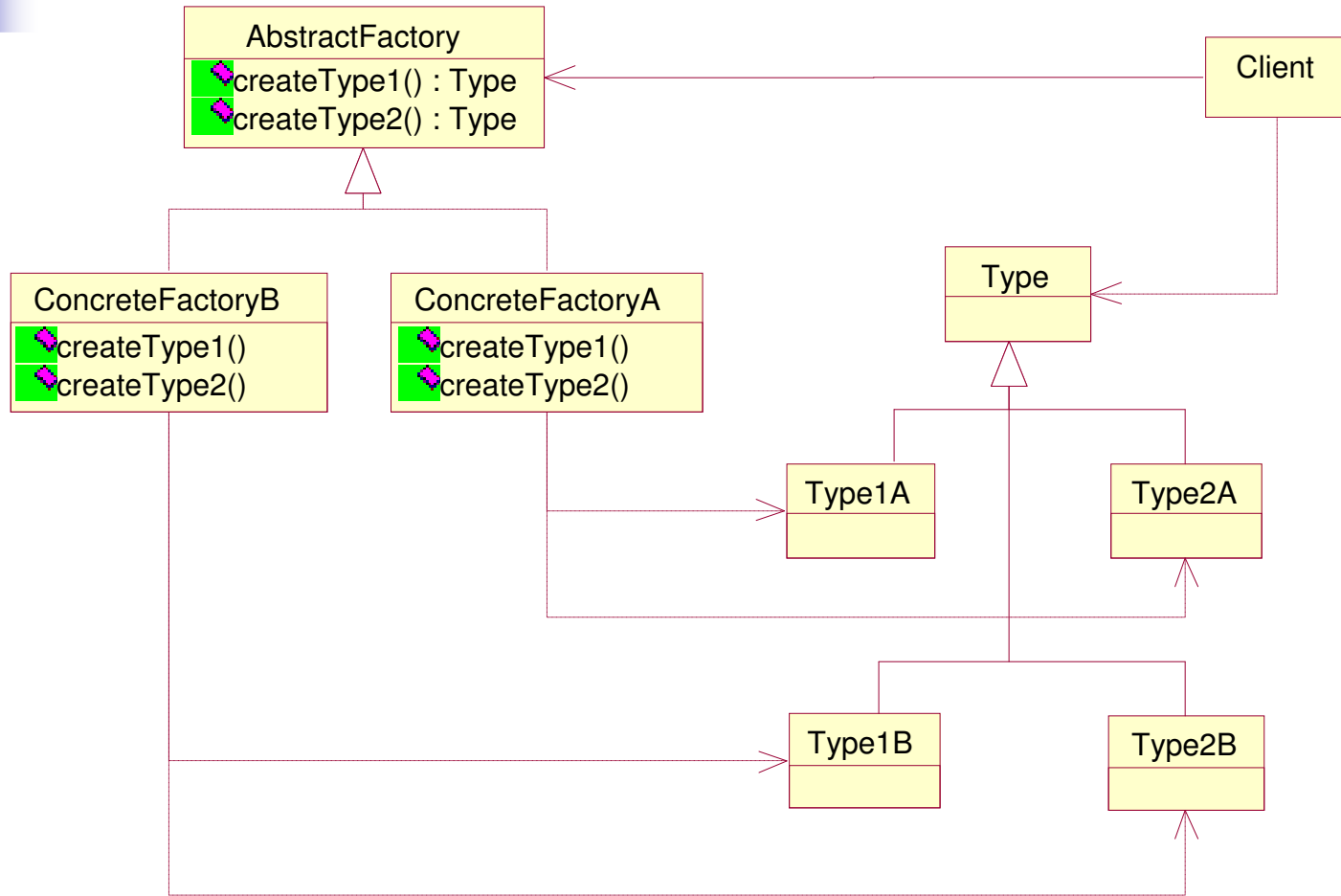
- создание объектов есть зависимость от производных типов
- но все прочие действия делаются через интерфейс
- что мы выиграли от использования интерфейса?

Решение



Создать класс, отвечающий за создание объектов
Избежать создания объектов нельзя, но можно локализовать

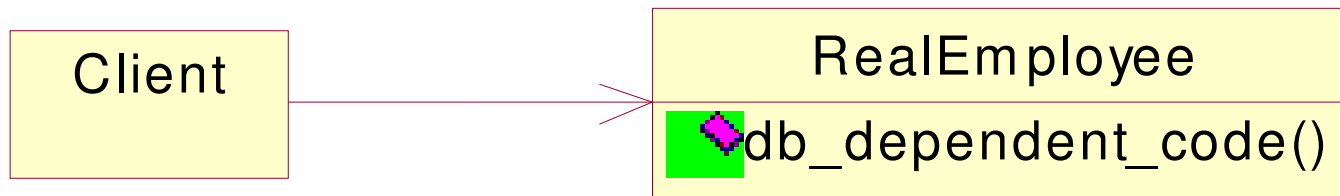
Factory Pattern



Изолирует конкретные типы данных, упрощает внесение изменений



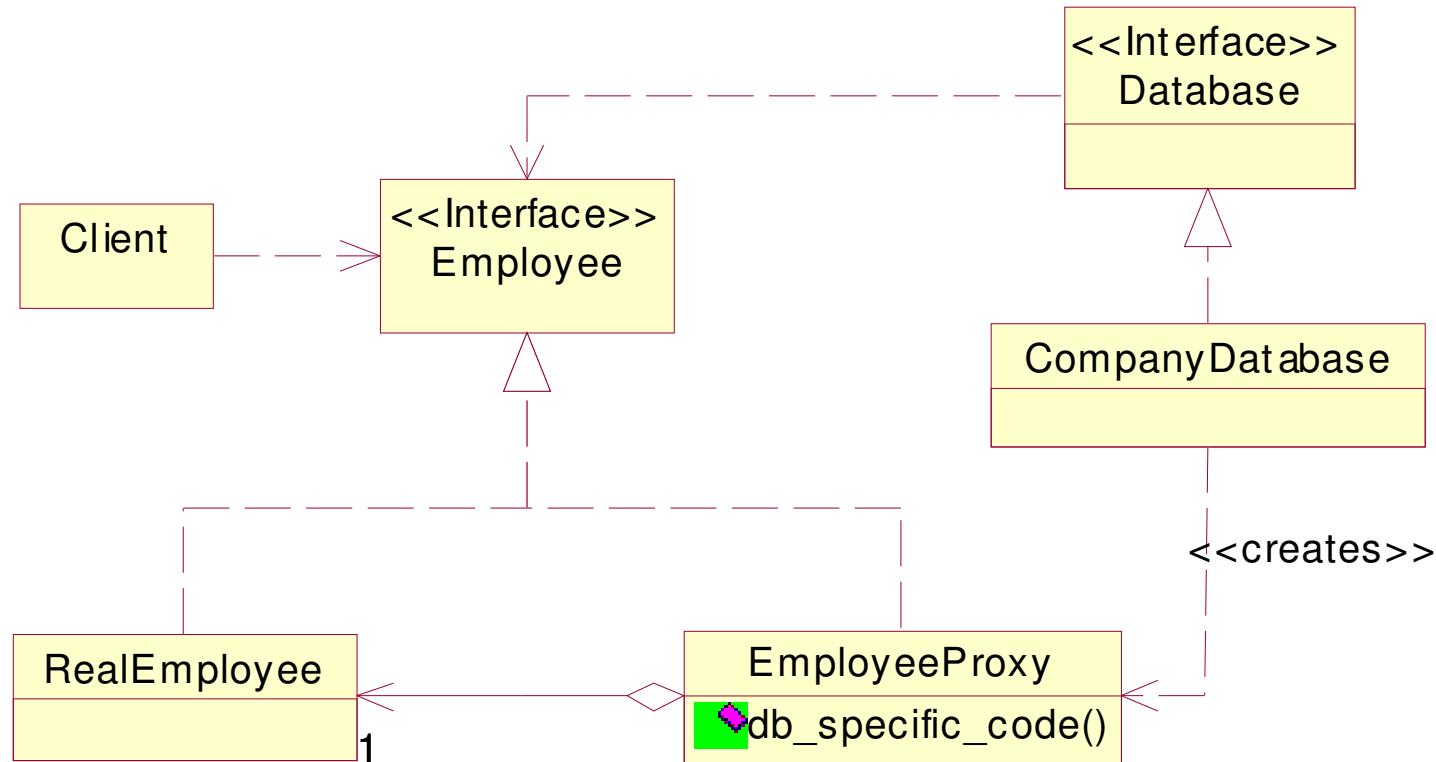
Proxy



Проблема:

Нам нужно хранить объекты в базе данных, но нет желания
зависеть от схемы базы

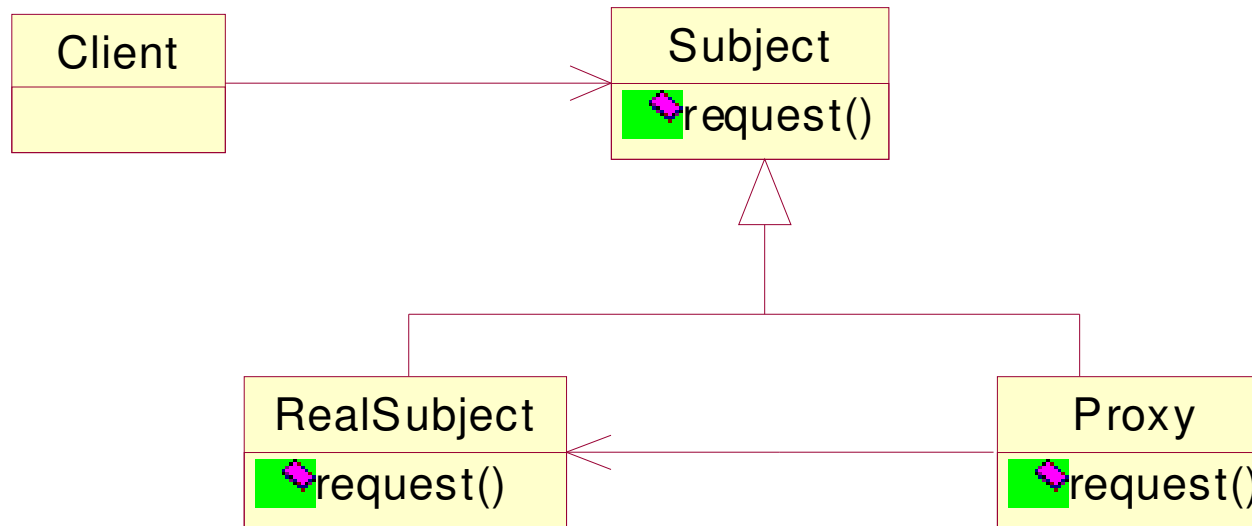
Решение



Решение:

- использовать суррогат который знает о схеме базы
- клиенты могут считать что работают с RealEmployee

Proxy Pattern

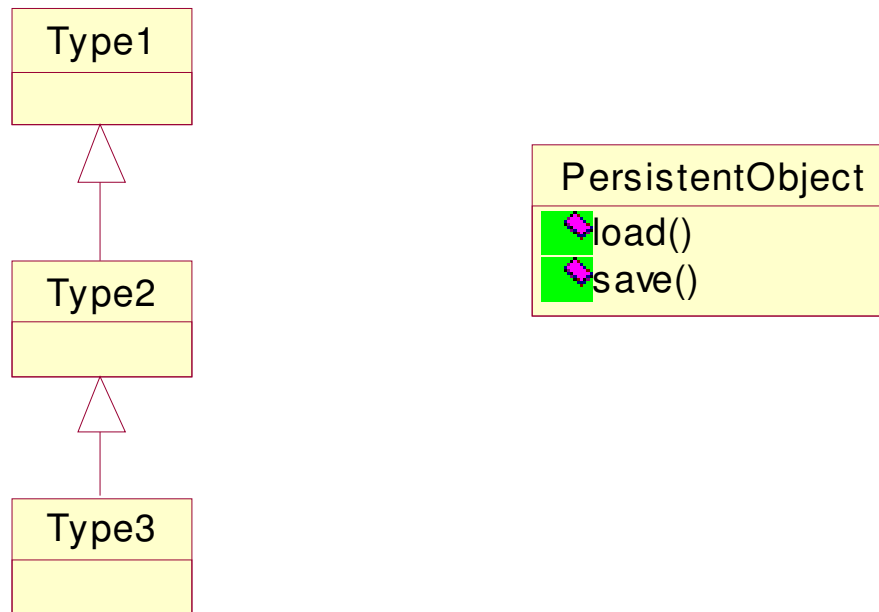


Также известен как: Surrogate

Применимость:

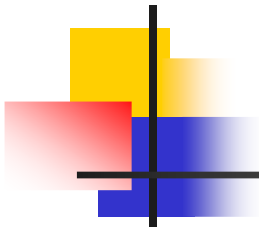
- remote proxy (удаленный доступ к объекту)
- virtual proxy (создание реальных объектов «на лету»)

Stairway to Heaven

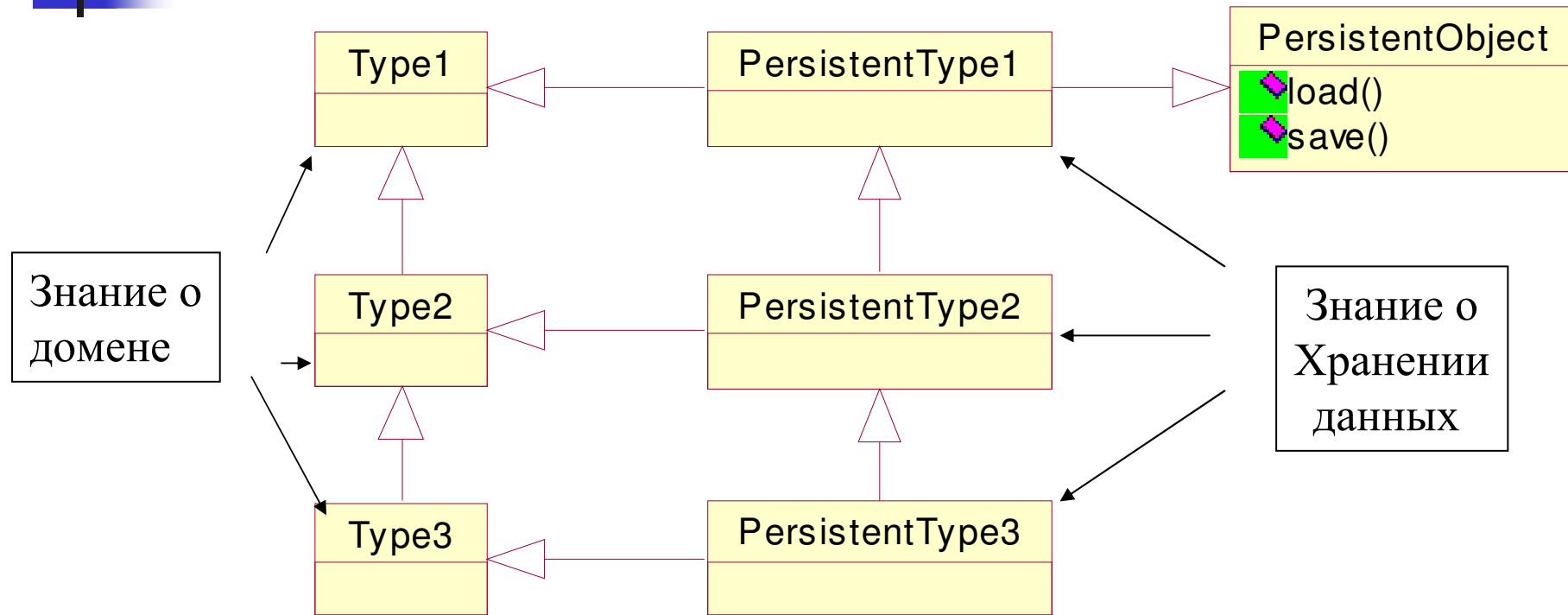


Проблема:

- Нужно сделать иерархию персистентной, но нет желания зависеть от производителя OR-маппера или OODB



Решение



Применить Adaptor к каждому уровню иерархии
❖ Требуется виртуального множественного наследования. Как его избежать?



8. Архитектурный шаблон

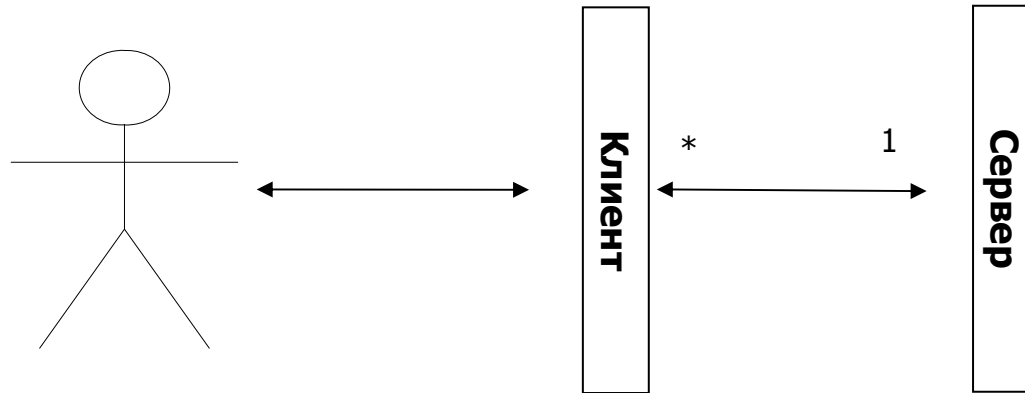
- - хорошо зарекомендовавшее себя решение определенной проблемы в определенном контексте
- Прежде чем изобретать новый «велосипед» – стоит изучить, как устроены и насколько хорошо работают отдельные узлы уже существующих «велосипедов»



Архитектурные шаблоны

- Client/server : Клиент/Сервер
- N-tier : Многоуровневая архитектура
- Peer-to-peer (P2P) : Одноранговая сеть
- Pipes and filters : Каналы и фильтры
- ACL security : Списки контроля доступа
- MVC (Model-View-Controller) : Модель-Представление-Управление

Архитектура клиент/сервер



Характерные черты:

- Единый сервер
- Отсутствие прямого взаимодействия между клиентами

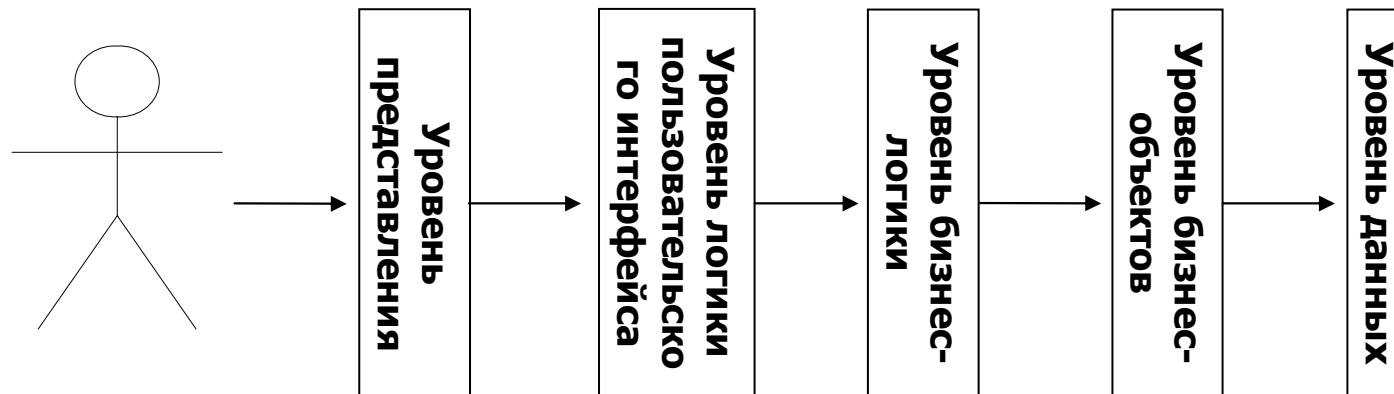
Преимущества:

- Централизованное обслуживание

Недостатки:

- Сервер может оказаться «узким местом» системы

Многоуровневая архитектура



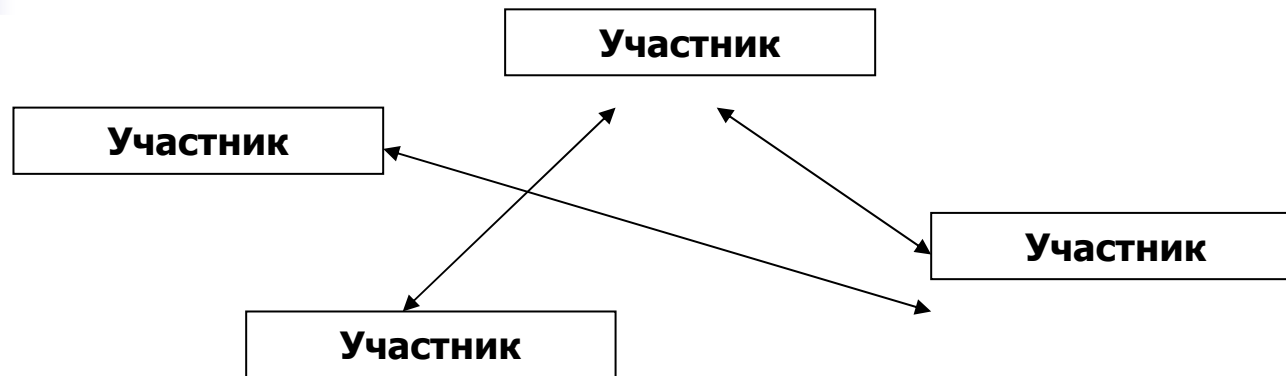
Характерные черты:

- каждый слой предоставляет сервисы следующему и использует сервисы предыдущего
- взаимодействуют только соседние слои
- каждый слой реализует четко определенную часть функциональности

Преимущества:

- возможность независимой разработки
- сужение набора необходимых для создания каждого слоя знаний

Одноранговая сеть



Характерные черты:

- отсутствие центрального сервера
- равные права участников

Преимущества:

- Надежность

Недостатки:

- Сложность распространения изменений (updates)

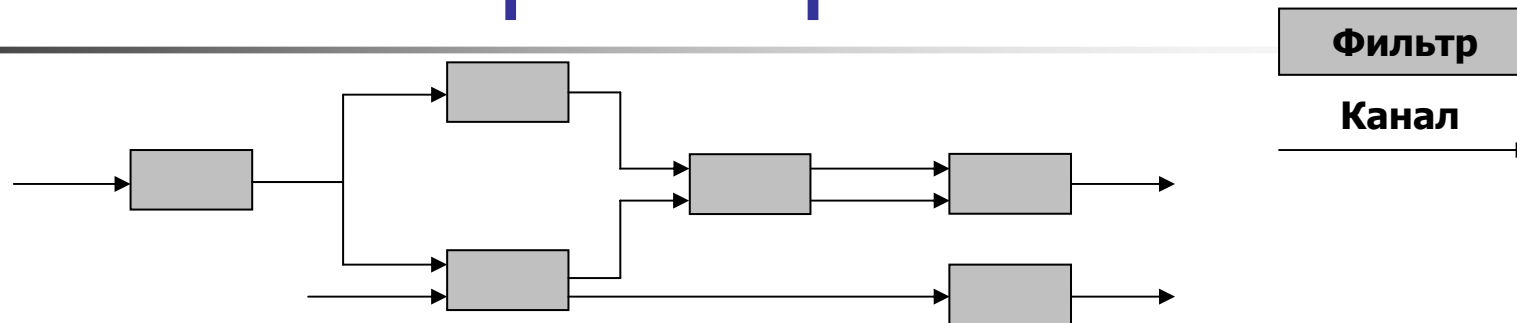


Одноранговая сеть

Применение

- Collaborative Computing
- Instant Messaging (совместно с Client-Server)
- P2P networks (сервера UseNet news, SMTP)
- Simple file sharing (MS Windows network w/o domain)

Каналы и фильтры



Применение: системы обработки потоков данных

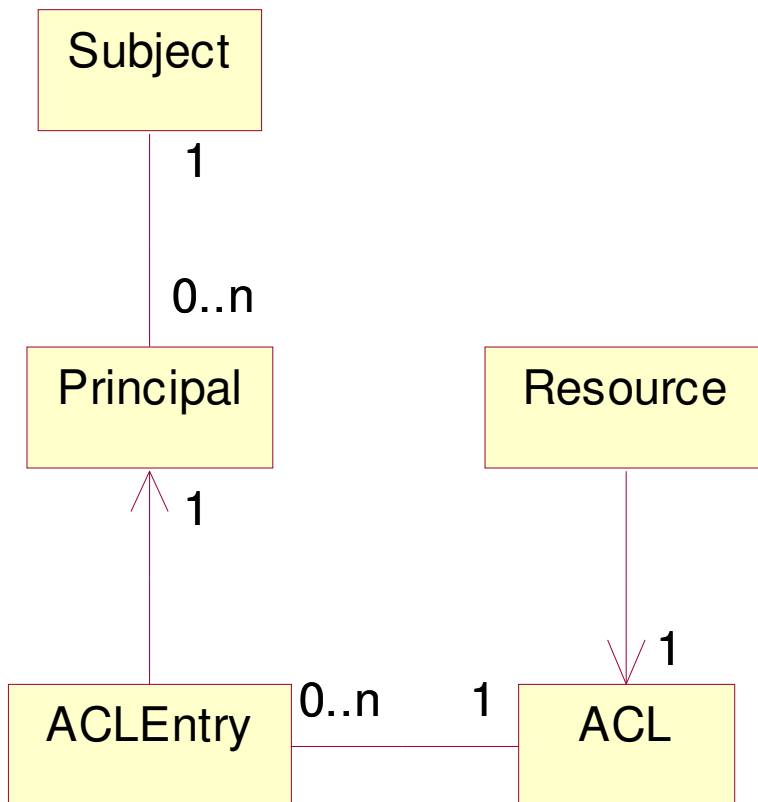
Ситуации:

- система создается разными разработчиками (возможно, часть системы уже существует)
- задача естественно разбивается на независимые этапы обработки
- задачи могут меняться, требуя декомпозиции шагов обработки

Характерные черты:

- фильтры – модули, получающие на вход поток(и) данных и выдающие поток(и) данных
- Простота конфигурирования системы фильтров (соединение их с помощью каналов)
- фильтры работают параллельно

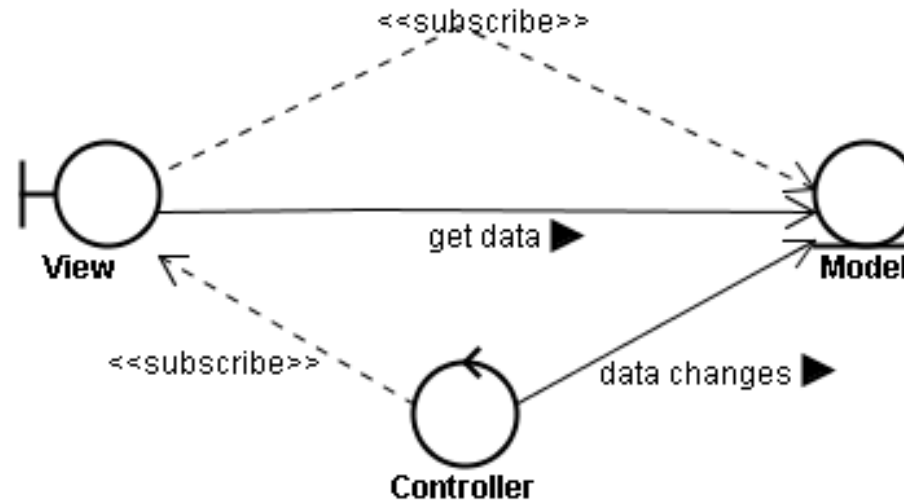
Списки контроля доступа (ACL или ACE)



- Subject – соответствует сессии пользователя
- Principal – некоторый идентификатор лица (пользователя) которому можно поставить в соответствие права доступа (например UUID), может быть не один, если у пользователя несколько ролей в системе
- Resource – ресурс, доступ к которому ограничен
- ACL – список идентификаторов лиц, которым разрешен доступ к ресурсу

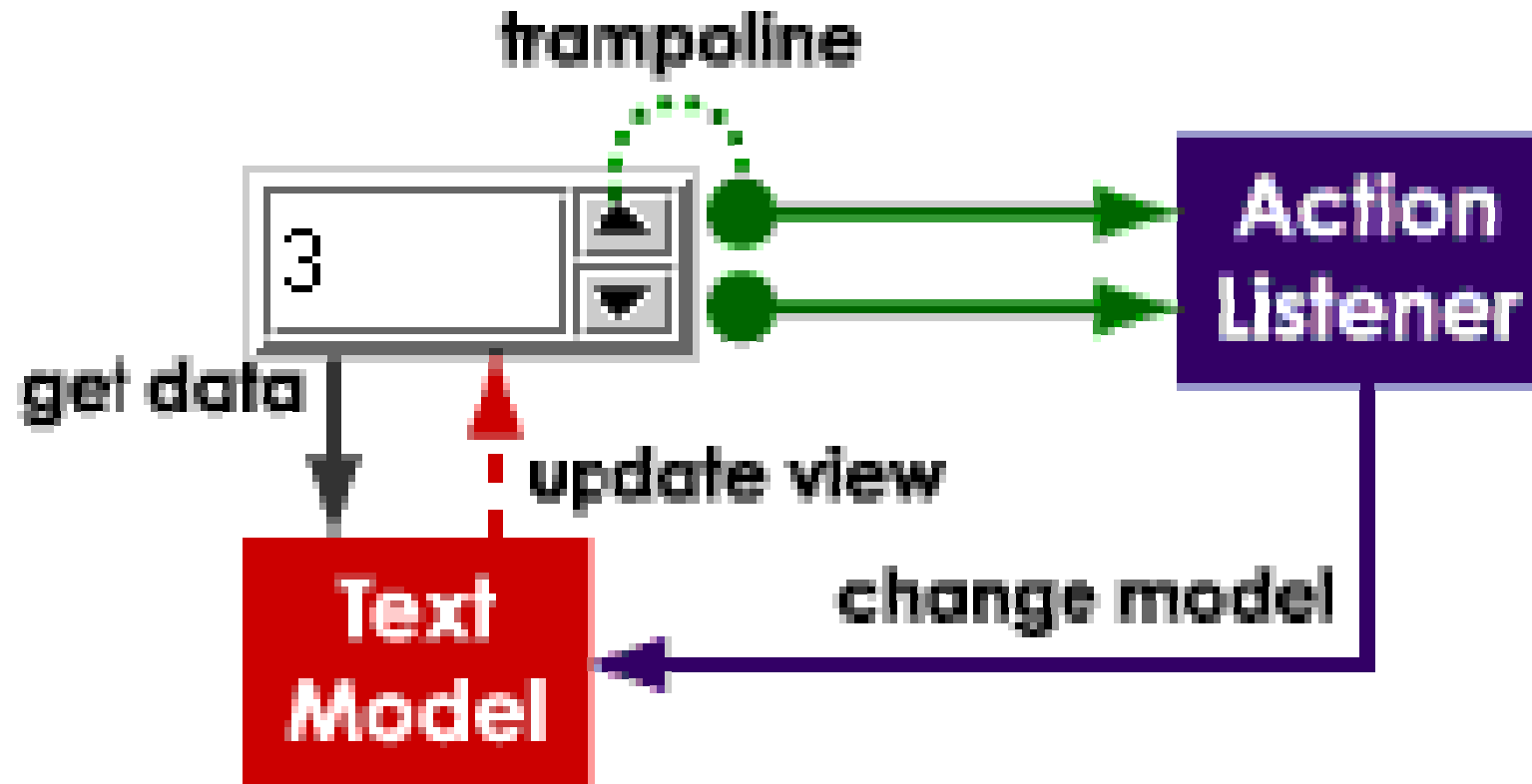
При каждом обращении к ресурсу производится проверка, есть ли у обращающегося subject'a principal, которому разрешено данное действие

Model-View-Controller



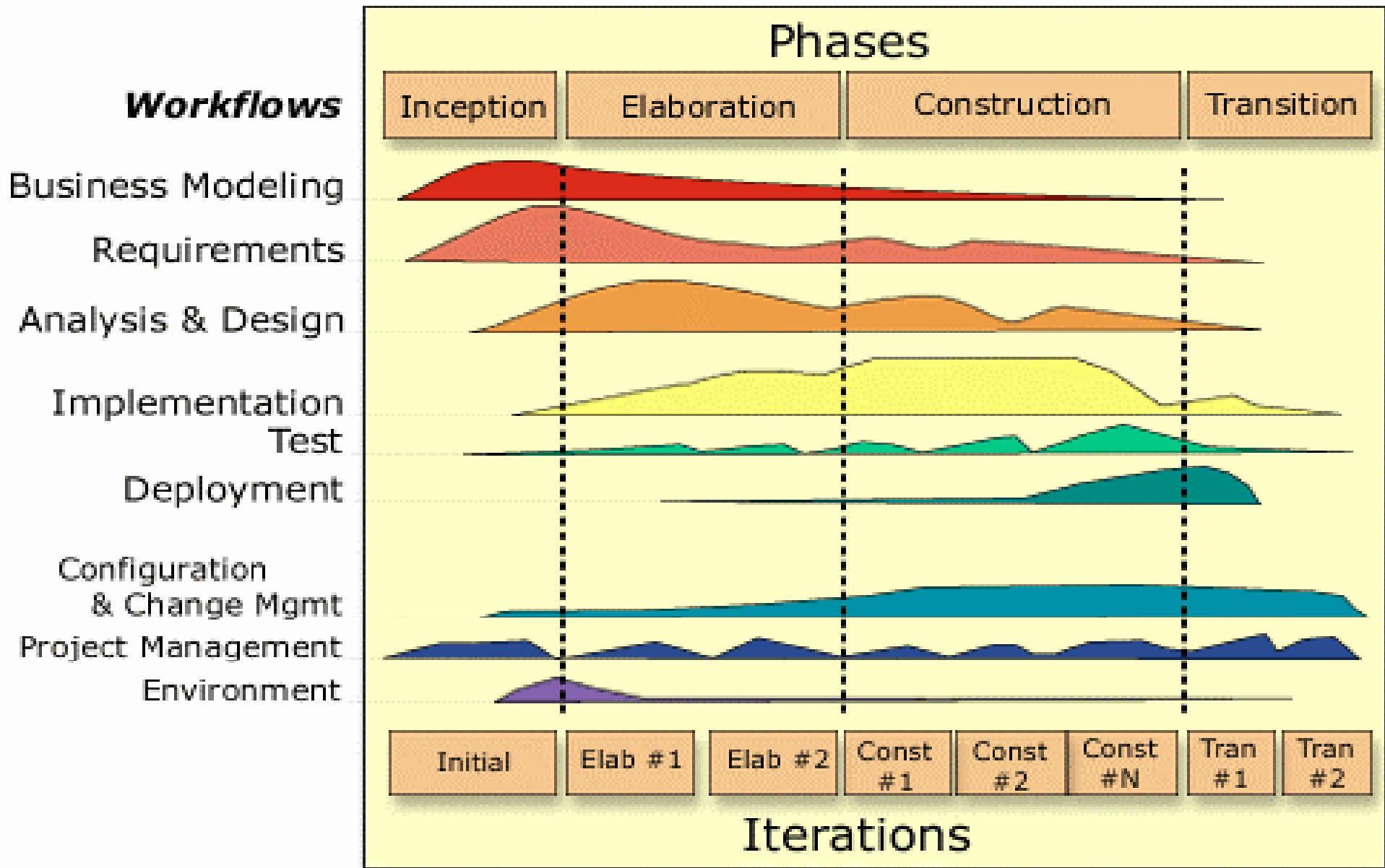
- Разделяет представление, данные и логику, и обработку событий UI
- Model: информация, бизнес-правила.
- View: элементы UI, получающие данные из модели
- Controller: обработка событий UI, часто через callback, передача изменений в модель

MVC





9. Rational Unified Process





Концепция (Vision)

- Vision – представляет собой общее описание проекта и является базисом для уточнения требований к системе
- Содержит:
 - Цели проекта
 - Stakeholders & Users (*описание инициаторов проекта и конечных пользователей*)
 - Перспективы и возможности системы
 - Особенности
 - Ограничения



RUP: Business modeling

- Задачи:
 - Идентификация бизнес-процессов (business use-cases)
 - Идентификация бизнес-актеров и сущностей (business entity)
 - Улучшение (refine) бизнес-процессов
- Модели:
 - business use-case model
 - business object model



RUP: Требования (Requirements)

Задачи:

- сбор и анализ требований к системе
- классификация use-cases
- оценки затрат и рисков

Модели:

- Use-case model



SRS (Спецификация требований)

- SRS (Software Requirements Specification) - полностью определяет требования к системе, зависит от Vision
- Содержит:
 - Функциональные требования (что должна делать система, роли пользователей, фактически, описание use-cases)
 - Нефункциональные требования (производительность, ограничения по используемым технологиям и т.д.)



RUP: Анализ и проектирование

Задачи:

- Трансформировать требования собранные на предыдущем этапе в дизайн системы
- Проработать архитектуру системы
- Адаптировать дизайн к среде исполнения

Модели:

- Analysis model
- Design model



SAD (Архитектурный документ)

- SAD (Software Architecture Document) – содержит полное описание архитектуры системы

- Содержит:
 - Use-case view
 - Logical View (архитектурно важные части Design model)
 - Process View
 - Deployment View (диаграмма размещения системы)
 - Implementation View
 - Open issues (список известных проблем, например, с производительностью или масштабируемостью, возможные пути решения)
 - Quality issues (любые проблемы в качестве)



RUP: Реализация (Implementation)

Задачи:

- Структурирование системы
- Реализация компонент системы

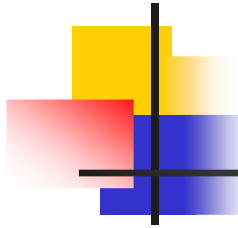
Артефакты:

- SAD – приведение в соответствие с реализацией
- Implementation model – модель реализации системы в терминах компонент и процессов



RUP: Ключевые роли

- Project Manager
- Analyst
- Test Designer
- System Architect
- Designer
- Implementer
- Tester



Роль RUP

- Высокоадаптивный процесс: минимум обязательных практик.
- Систематизация знаний в области процессов разработки ПО



Бизнес-анализ

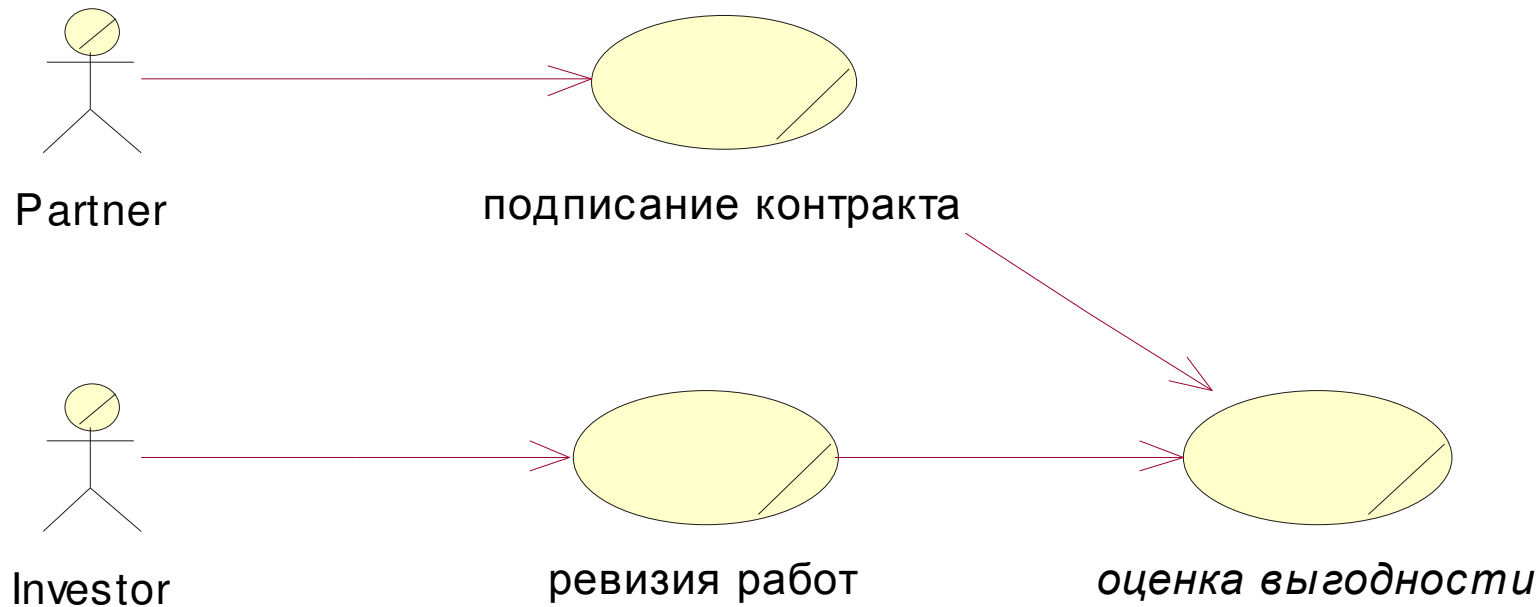
- Задачи:
 - Идентификация бизнес-процессов (use-cases)
 - Идентификация бизнес-актеров и сущностей (entity)
 - Улучшение (refine) бизнес-процессов
- Нужен для того, чтобы:
 - Лучше понять предметную область
 - Понять, как изменятся процессы в бизнесе в случае внедрения системы, описываемой в use-case model
- UML модели:
 - business use-case model
 - business object model



business use-case model

- Модель, описывающая бизнес процессы в терминах *business-actors* и *business use-cases*
- *Business actor* – некто или нечто **ВОВНЕ** бизнеса, взаимодействующее с ним
 - UML: класс со стереотипом <<*business actor*>>
- *Business use-case* – бизнес-процесс, представляющий ценность для *business actor*
 - UML: use-case со стереотипом <<*business use-case*>>

Модель бизнес-процессов





business object model

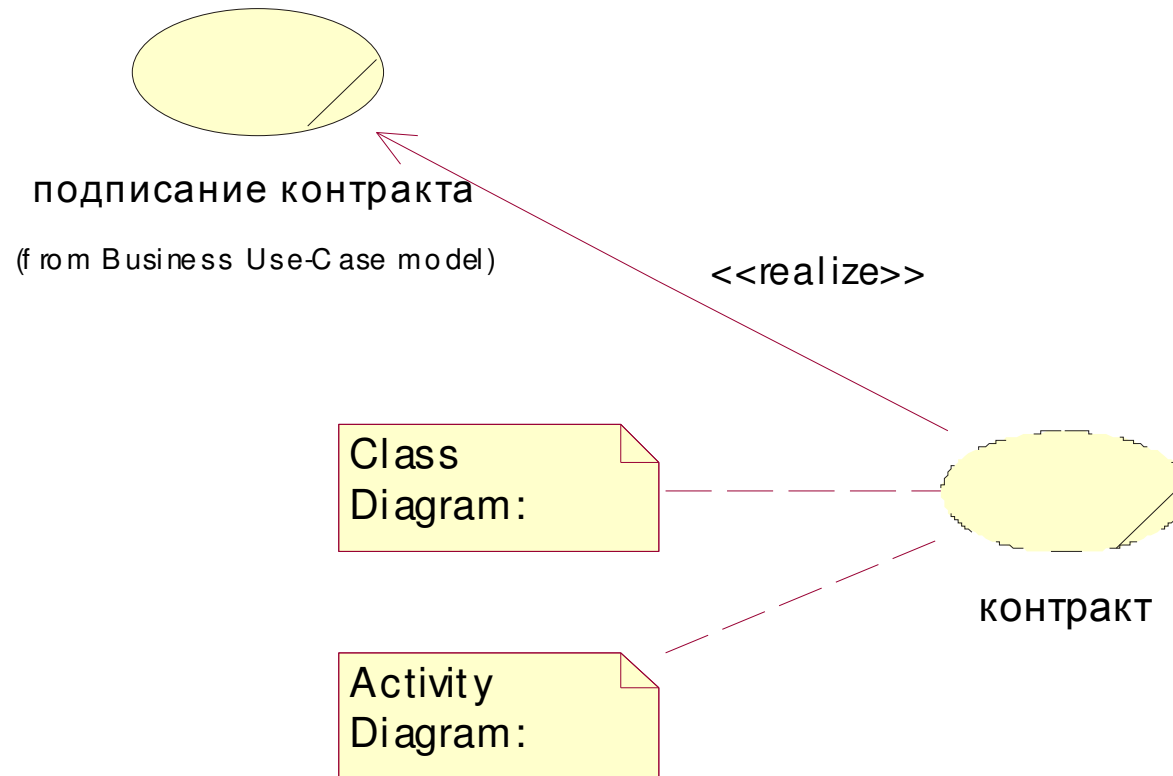
- Business object model - модель, описывающая реализацию business use-cases в терминах взаимодействующих объектов
 - UML: классы со стереотипами "business worker" и "business entity"
- Business use-case realization – часть *business object model*, коллаборация, описывающая при помощи *activity*, *sequence*, и *class* диаграмм, как данный *business use-case* реализован в *business-object model*.
 - UML: use-case со стереотипом "business use-case realization", классы со стереотипом "business actor"



Бизнес-объекты

- Business-worker — *исполнитель бизнес-процесса*
 - *UML: class со стереотипом <<business worker>>*
- Business-entity — *пассивная сущность, используемая в бизнесе*
 - *UML: class со стереотипом <<business entity>>*

Модель бизнес-объектов



use-case realization содержит набор диаграмм, описывающих КАК реализован исходный use-case

activity diagram для бизнес-процесса "контракт"

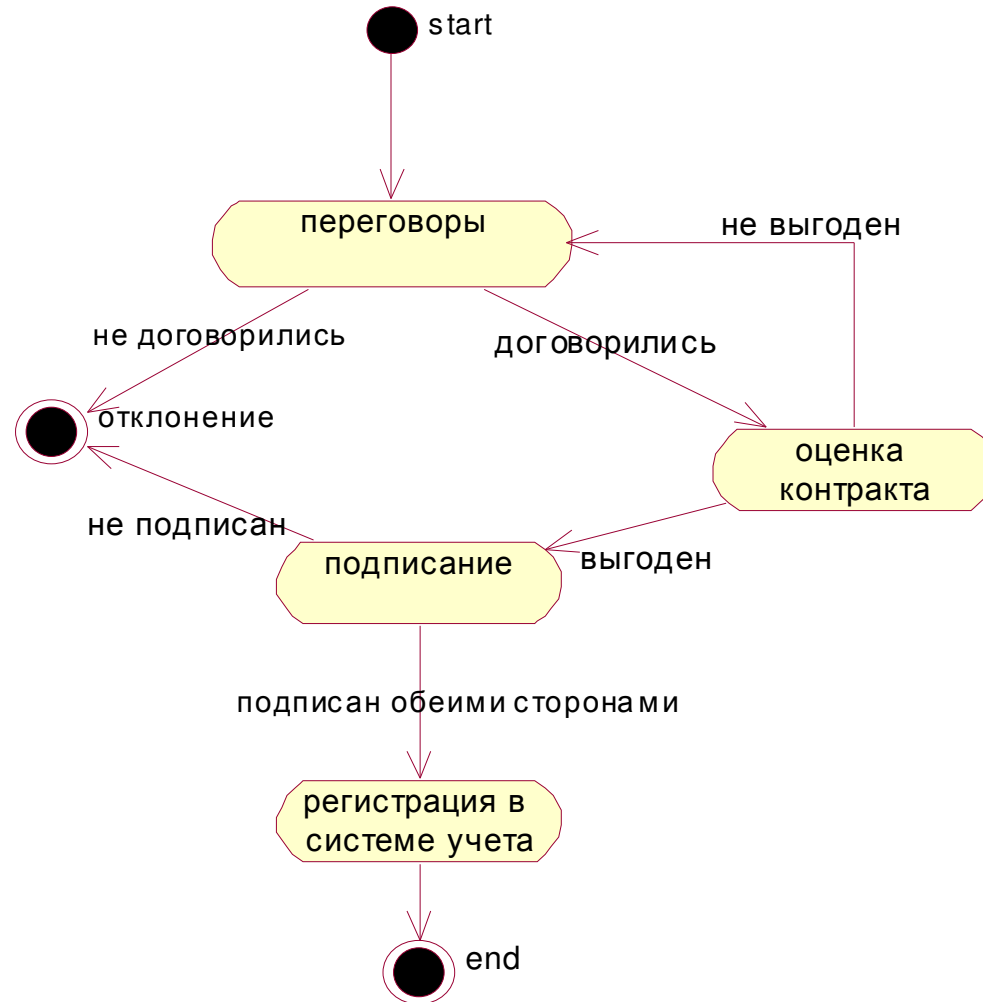


диаграмма классов для бизнес-процесса "контракт"

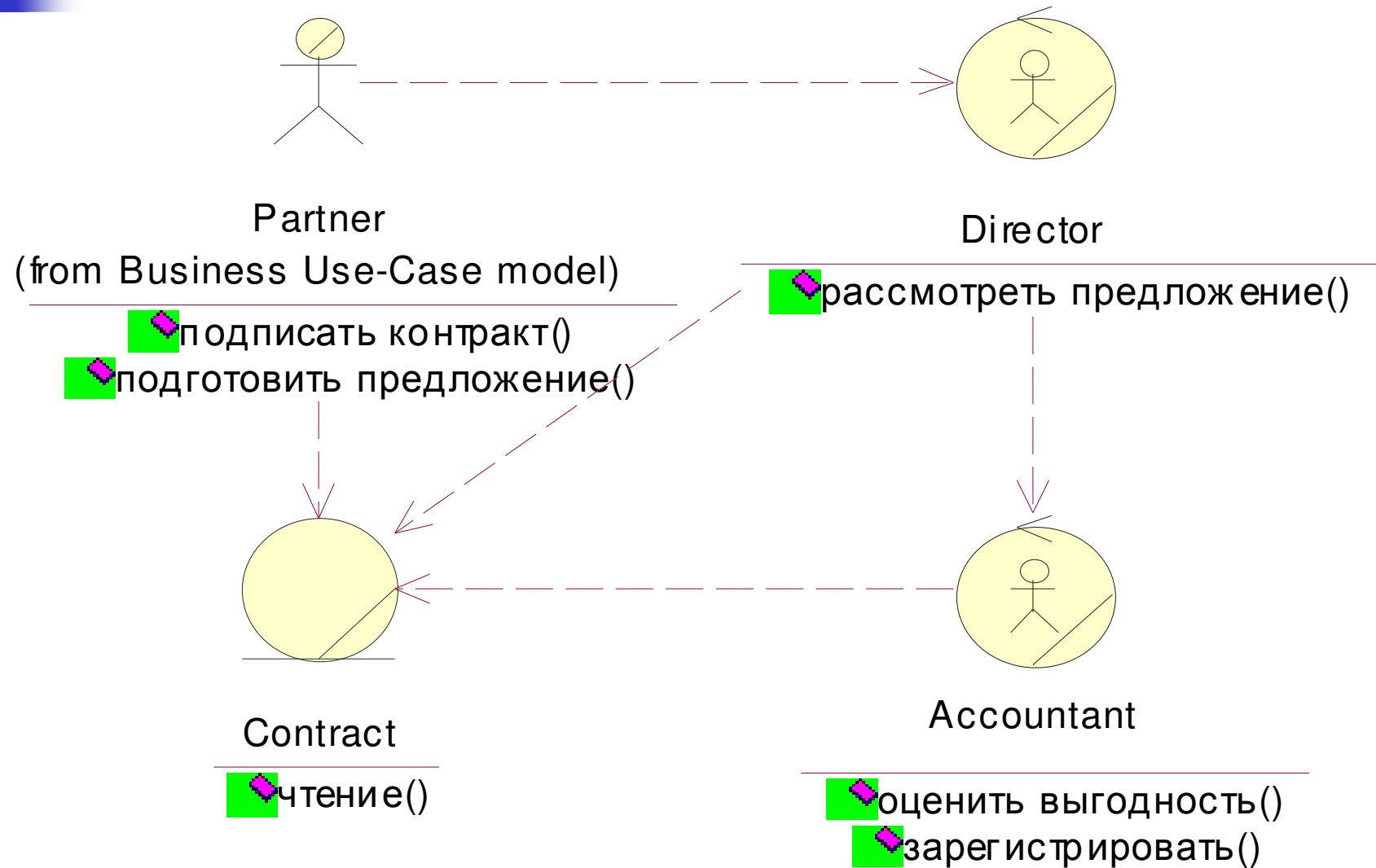
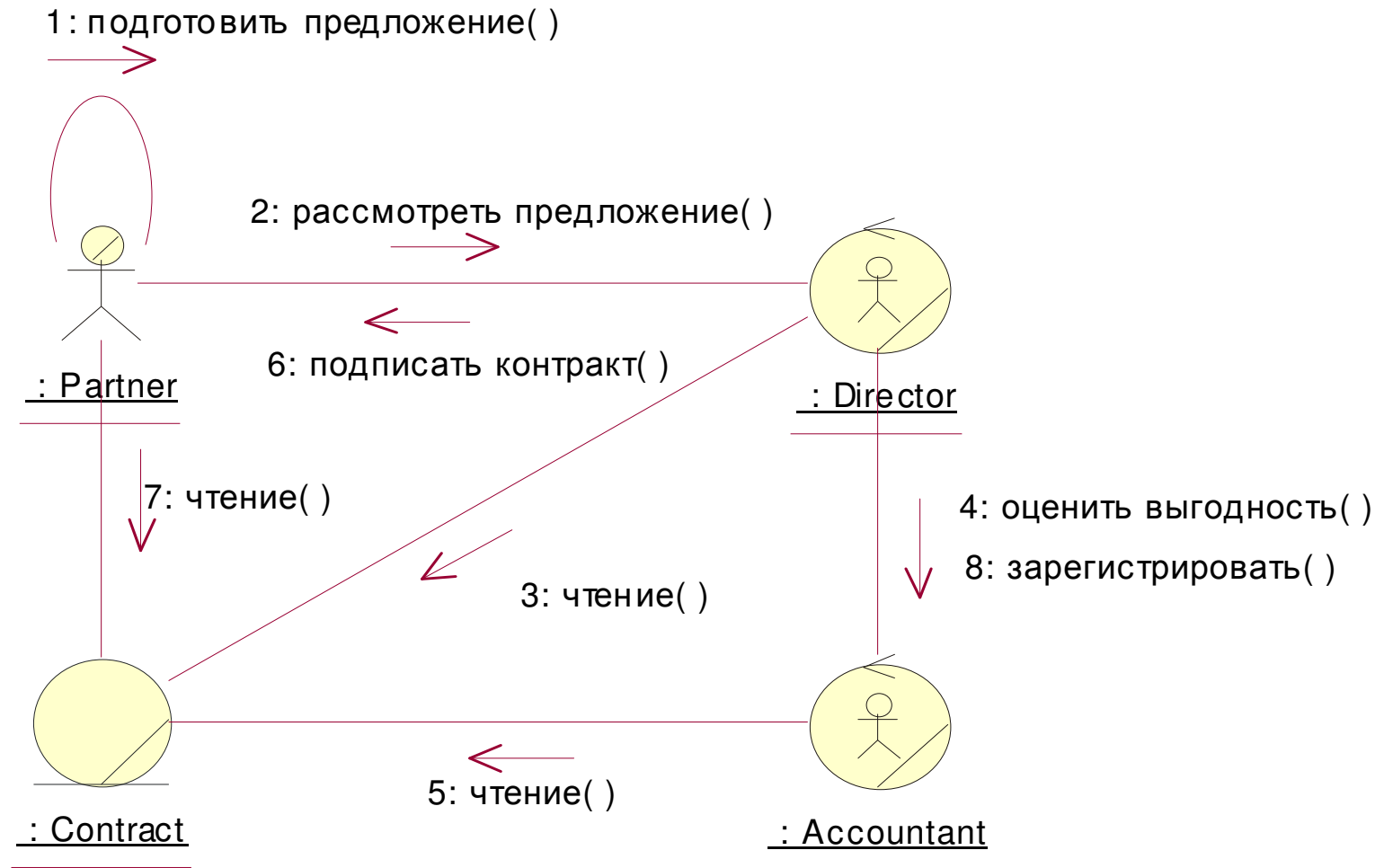


диаграмма коопераций для бизнес-процесса "контракт"





10. Управление конфигурацией

- Конфигурация ИТ проекта:
 - исходные тексты, документы, модели
 - требования и их изменения
 - все возможные артефакты проектной деятельности
 - используемые библиотеки
 - конфигурация средств разработки
 - правила кодирования и документирования кода
- Конфигурация подлежит управлению:
 - управлению версиями
 - контролю за соблюдением правил



Средства управления конфигурацией

- Системы контроля версий
 - CVS – вышел из моды
 - SVN – наиболее распространена в данный момент
 - Visual SourceSafe (Microsoft)
 - Rational ClearCase (IBM)
- Системы контроля требований и инцидентов (CRM, ITS, bug trackers)
 - Scarab – open source bug tracker
 - JIRA – гибкая недорогая коммерческая ITS
 - Rational ClearQuest (IBM)



Терминология: контроль версий

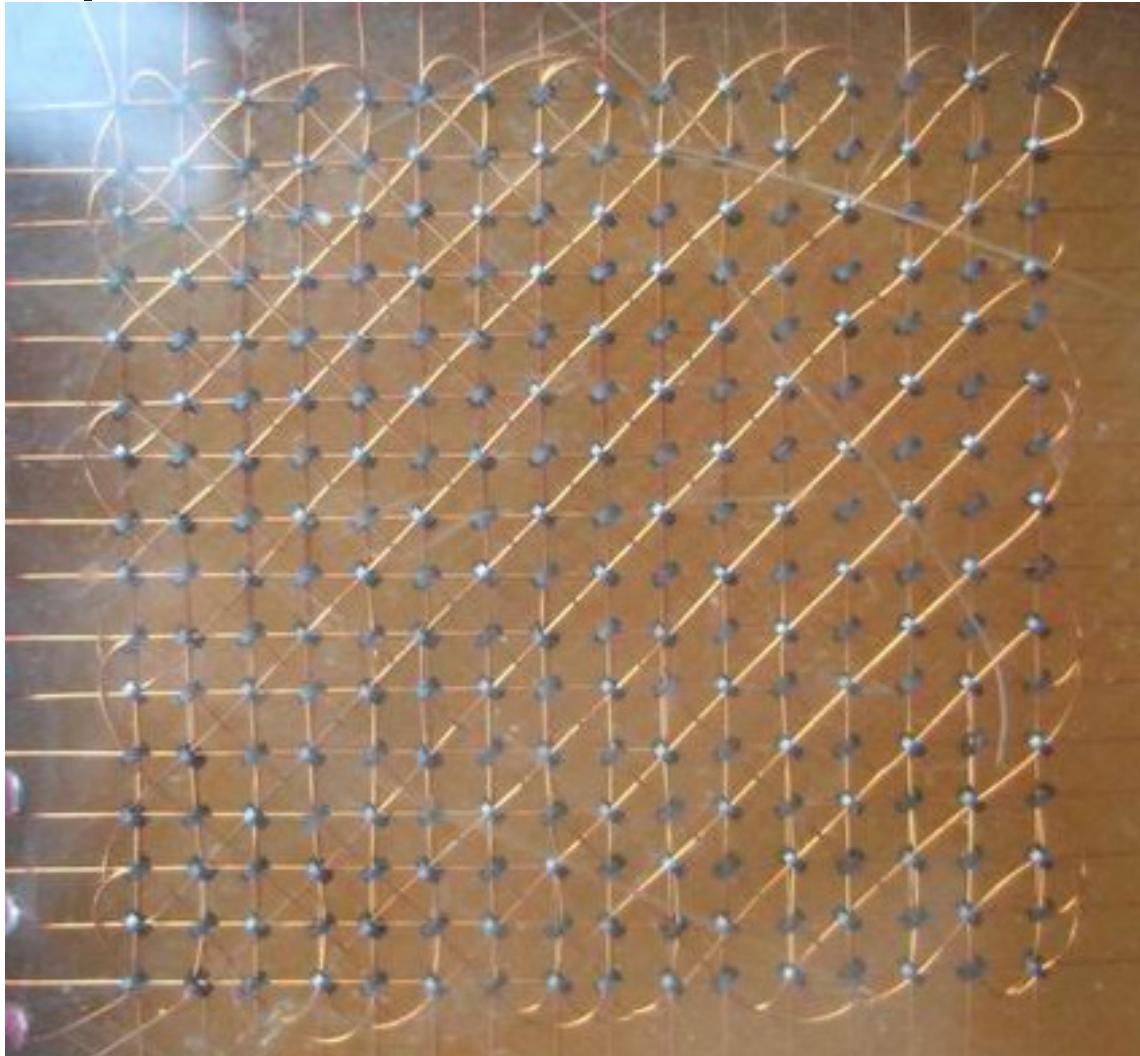
- Репозиторий – место хранения артефактов проекта в системе управления версиями
- Релиз – любая версия ПО, вышедшая за пределы проектной команды; подлежит обязательному присвоению номера версии и тега
- Бранч – версия ПО, начавшая изменяться параллельно с текущей версией
- Тег – моментальный снимок состояния репозитория



Терминология

- Требование (requirement) – любое функциональное или нефункциональное требование
- Изменение (change) – изменение исходного требования
- Уточнение (adjustment) – уточнение требования, не влияющее на оценку проекта
- Ошибка (bug) – подтвержденная ошибка
- Ticket, Issue – проблема, нерешенный вопрос, может стать ошибкой, может изменением

Термин Bug (жук)



- Есть мнение, что термин возник из-за настоящих жуков, иногда попадавших на проволочки памяти на ферритовых кольцах, и мешавших считывать информацию



Несколько советов

- Не путайте бранч с тегом, у них разные задачи
- Commit в репозиторий делается ежедневно.
No commit – no excuse.
- Автоматический бекап репозитория:
 - Ежедневно.
 - Обязательно на другой физический диск;
 - лучше всего – на диск, расположенный на другом сервере;
 - еще лучше – на диск на сервере, расположенном в другом здании.
 - Регулярное копирование бекапа на долговечный носитель (CD, DVD, streamer).



Несколько советов

- Используйте единый метод сборки проекта всеми участниками, желательно из командной строки (build.xml, makefile), т.к это позволит, при необходимости, легко внедрить регулярную автоматическую сборку
- Не допускайте использования разных сред разработки в одном проекте (то, что программисту «нравится» - пусть использует у себя дома, на работе он должен использовать то, что записано в описании проектной среды)