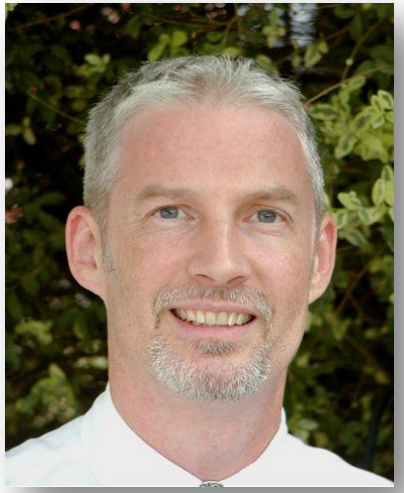
Проектирование ПО

Тема 5. Структурный проект

Практическая программная инженерия на основе учебного примера / Л.А. Мацяшек, Б.Л. Лионг. – М.: БИНОМ. Лаборатория знаний, 2009. – 956 с.

Особенности структурного проектирования

1) оно касается нефункциональных требований;
2) включает крупномасштабные, фундаментальные решения системного слоя;
3) занимается взаимозависимостями и компромиссами;
4) обеспечивает формирование и оценку альтернативных решений.



Крэг Ларман (Craig Larman)

Ларман К. Применение UML и шаблонов проектирования. 2-е издание. – М.: Издат. дом «Вильямс», 2002.- 624 с.

Структурные слои и управление зависимостями

Качественный структурный проект требует:

- *иерархического выделения слоёв* модулей ПО, которое уменьшает сложность и делает более понятной зависимость модуля, запрещая непосредственную связь объектов, находящихся не на соседних слоях, и
- *использования стандартов программирования*, которые делают зависимости модуля видимыми в компилируемых структурах программ и запрещают запутанные программные решения, использующие только структуры исполняемых программ.

Структурное проектирование – это управление зависимостями

Модуль А зависит от модуля В, если изменения в модуле В могут потребовать изменений в модуле А.

Структурные модули. Классы проекта

Анализ требований имел дело с *бизнес-объектами*, классифицируемыми как:

- бизнес-сущности;
- классы предметной области; и
- концептуальные классы.

В типичной программе нужны также:

- классы, ответственные за представление информации на экране компьютера,
- классы для доступа к БД,
- классы для выполнения алгоритмических вычислений и т. д.

Все эти классы будем называть *классами проекта*.

Структурные модули. Пакеты

Классы проекта группируются в *пакеты* (*packages*) согласно структурному шаблону, принятому для разрабатываемого проекта.

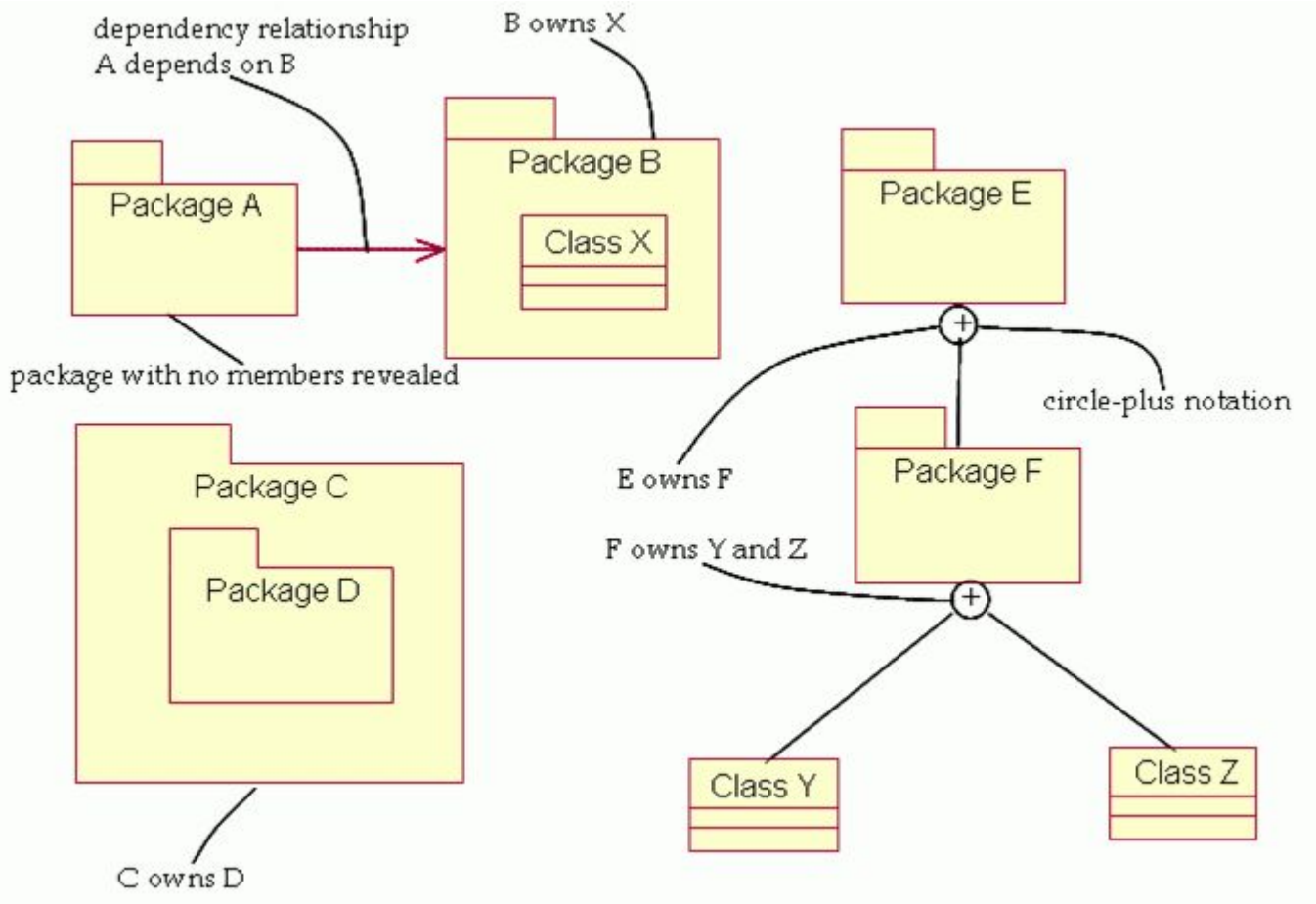
Пакет - группировка элементов моделирования под назначенным именем.

Пакет может содержать другие пакеты. Пакеты могут быть сгруппированы и структурированы в иерархию слоёв.

В UML пакет - логическая концепция проекта. В языках программирования поддержка пакета обеспечивается в форме пространства имен для классов и для импортирования других пакетов.

Пакет владеет своими элементами. Пакет может импортировать другие пакеты. Это означает, что пакет A или элемент пакета A может обратиться к пакету B или к его элементам. Следовательно, класс принадлежит только одному пакету, но он может быть импортирован в другие пакеты. Импорт представляет зависимость между пакетами и их элементами.

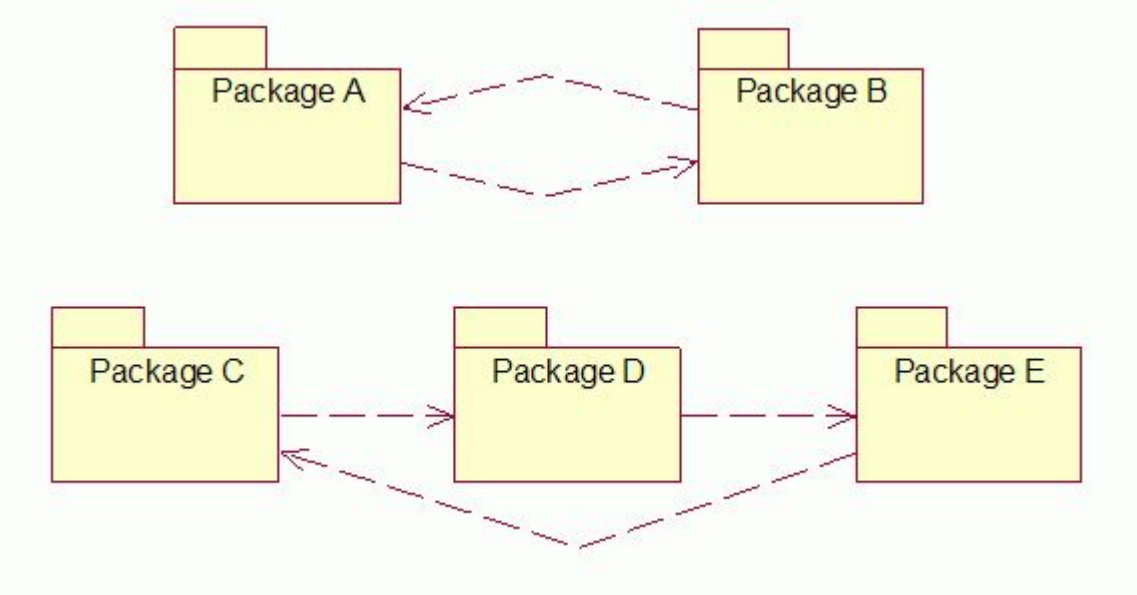
Нотация пакетов



Package A зависит от Package B. Это значит, что:

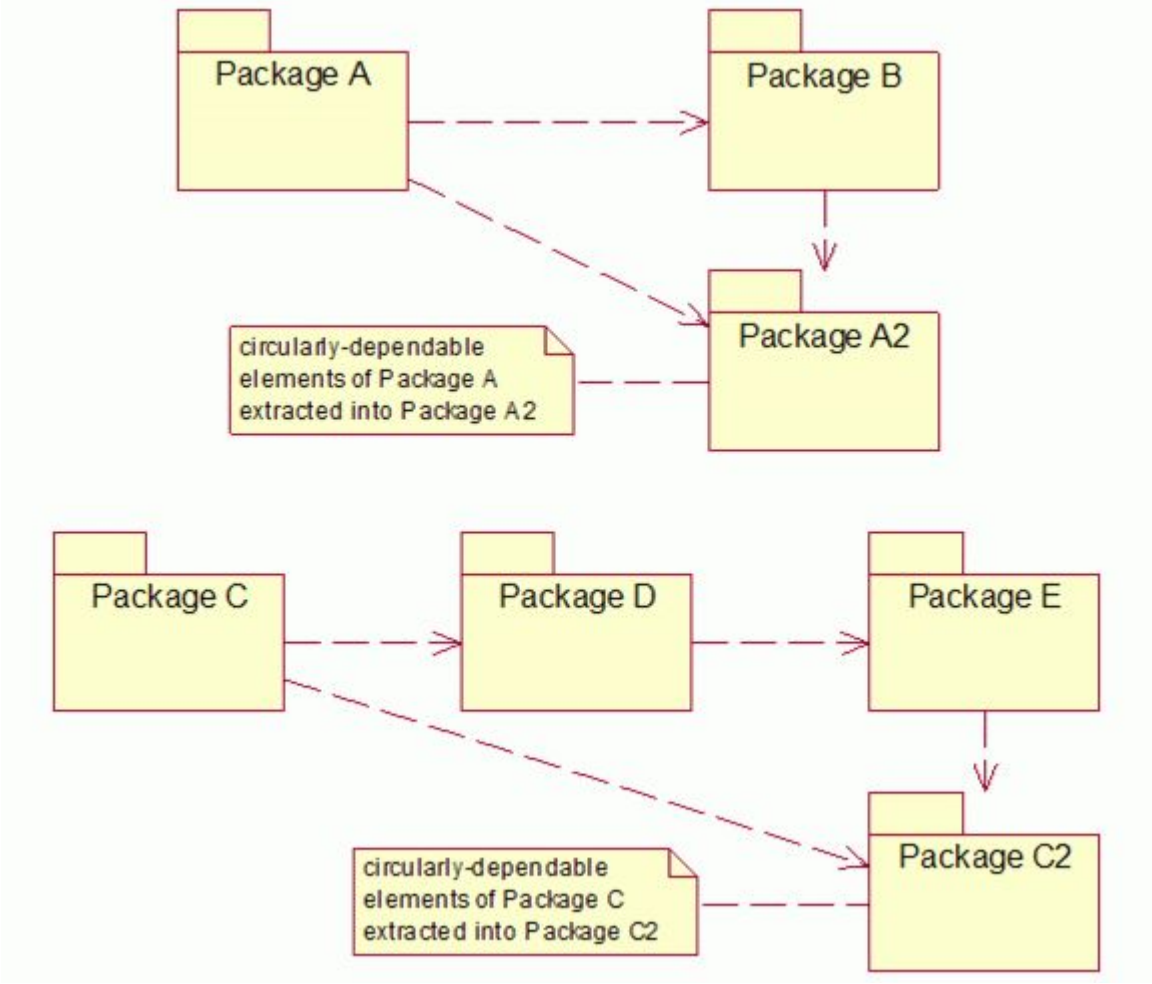
- Изменения в Package B могут воздействовать на Package A, обычно приводя к потребности перекомпилировать и перепроверить Package A.
- Package A может повторно использоваться только вместе с Package B.

Циклические зависимости между пакетами



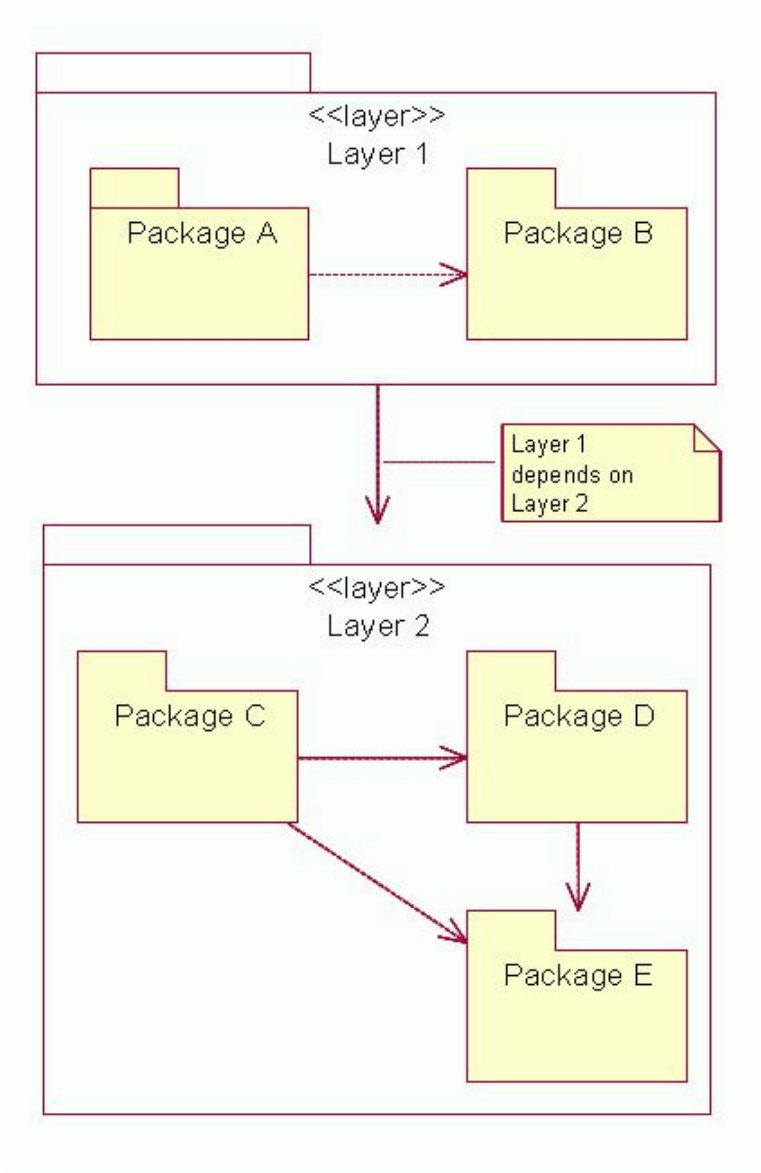
Циклические зависимости между объектами в разных пакетах приводят к нежелательным циклическим зависимостям между пакетами.

Исключение циклических зависимостей между пакетами



Добавление нового пакета может устранить циклические зависимости между пакетами.

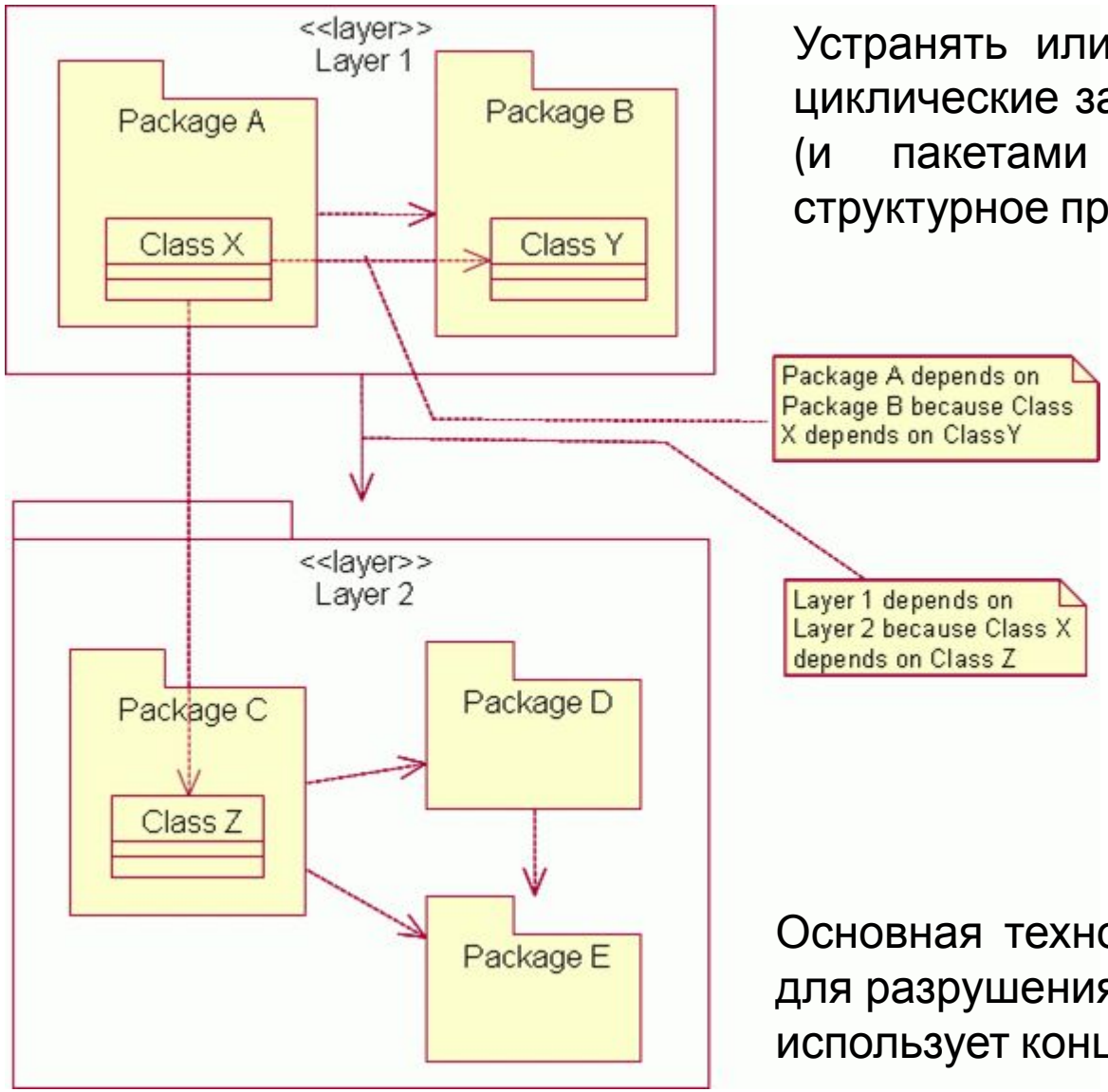
Слои в качестве пакетов



Хорошее структурное проектирование слоёв предусматривает, что иерархия слоёв:

1) не представляет сетевую структуру;
2) минимизирует зависимости между пакетами;
3) устанавливает стабильный шаблон для жизненного цикла разработки системы.

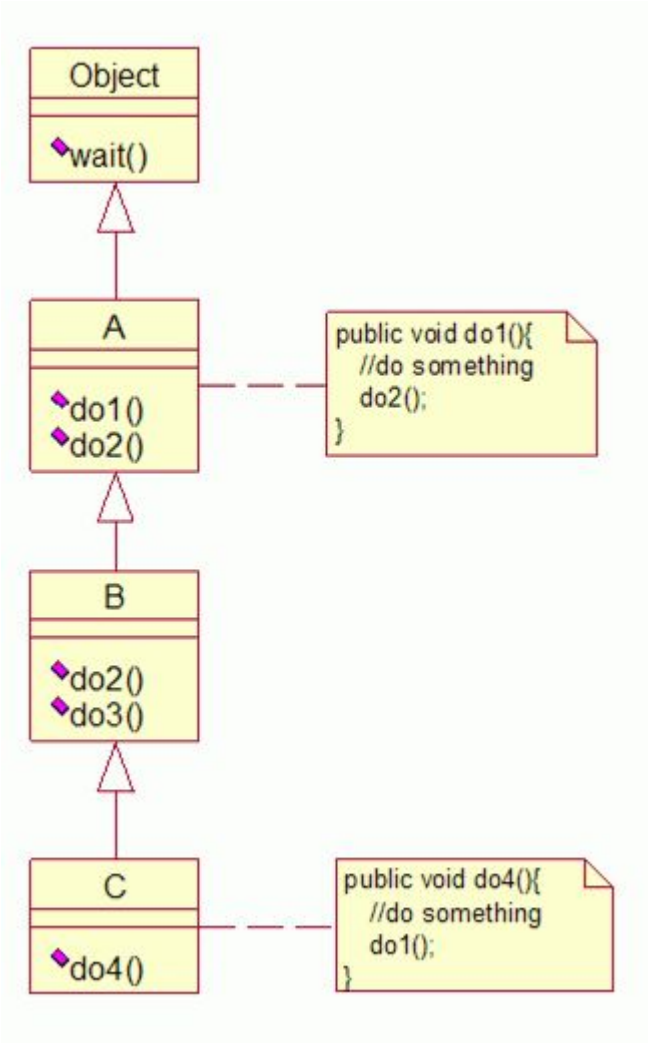
Зависимости классов и вытекающие из этого зависимости слоёв и пакетов



Устранять или выводить из структуры циклические зависимости между слоями (и пакетами в целом) позволяет структурное проектирование классов.

Основная технология, предназначенная для разрушения циклов между классами использует концепцию *интерфейса.*

1. Зависимости наследования времени компиляции

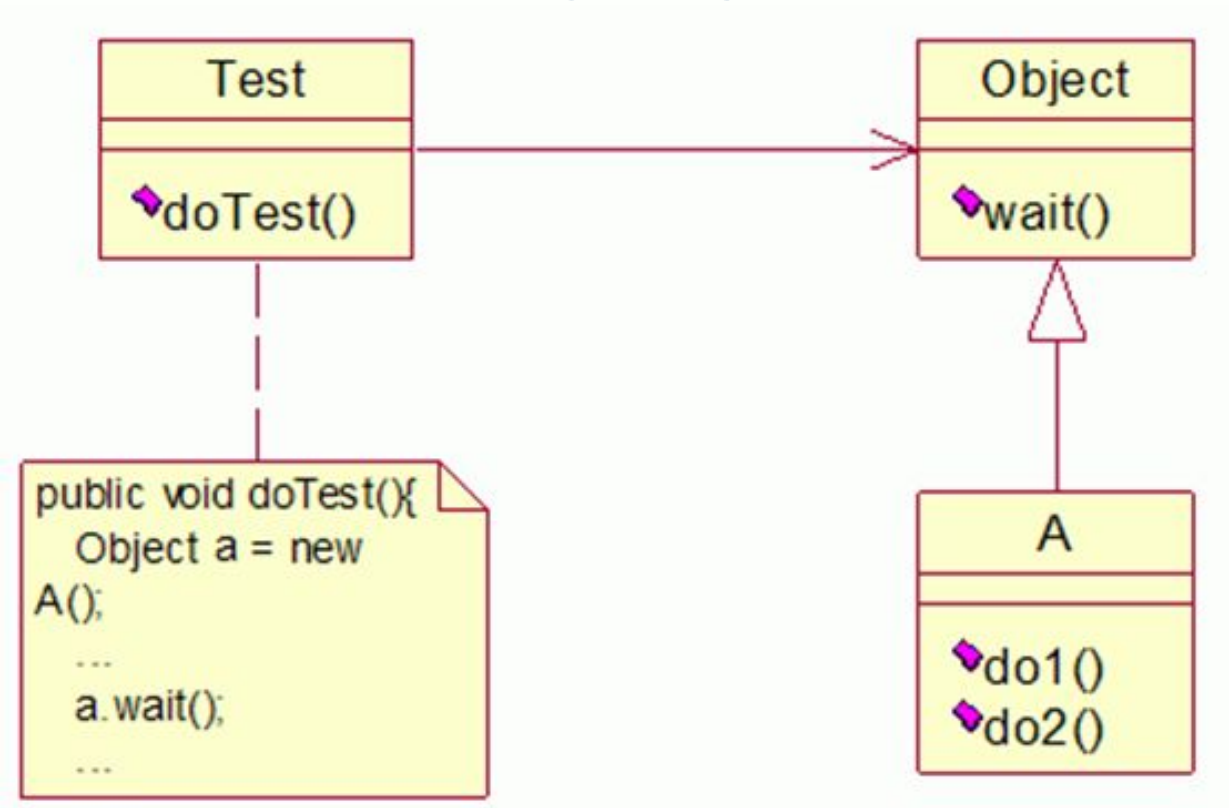


Все отношения обобщения в диаграмме представляют зависимости времени компиляции в направлении изображенных стрелок. Зависимости *транзитивны*: если C зависит от B и B зависит от A, то C зависит от A и т.д.

B наследует A и B переопределяет метод do2() объекта A. Это создает потенциально полиморфное поведение do2() и представляет вероятную зависи-мость времени выполнения в противоположном направлении от A к B. A будет зависеть от B, если метод do1() объекта A должен вызвать метод do2() объекта B, а не свой собственный.

Такими видами циклических зависимостей, полученных из-за наследования, трудно управлять.

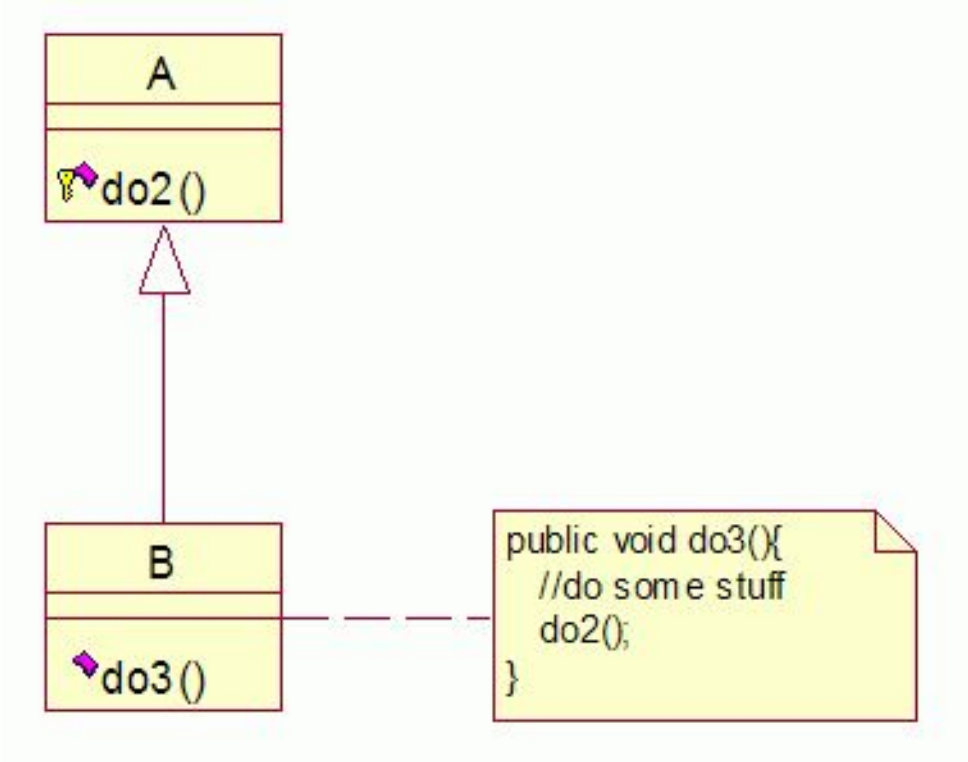
2. Зависимости наследования времени выполнения



Объект A инициализирован объектом Test, но объект Test не использует методы объекта A. В результате Test не имеет зависимости наследования времени выполнения от A.

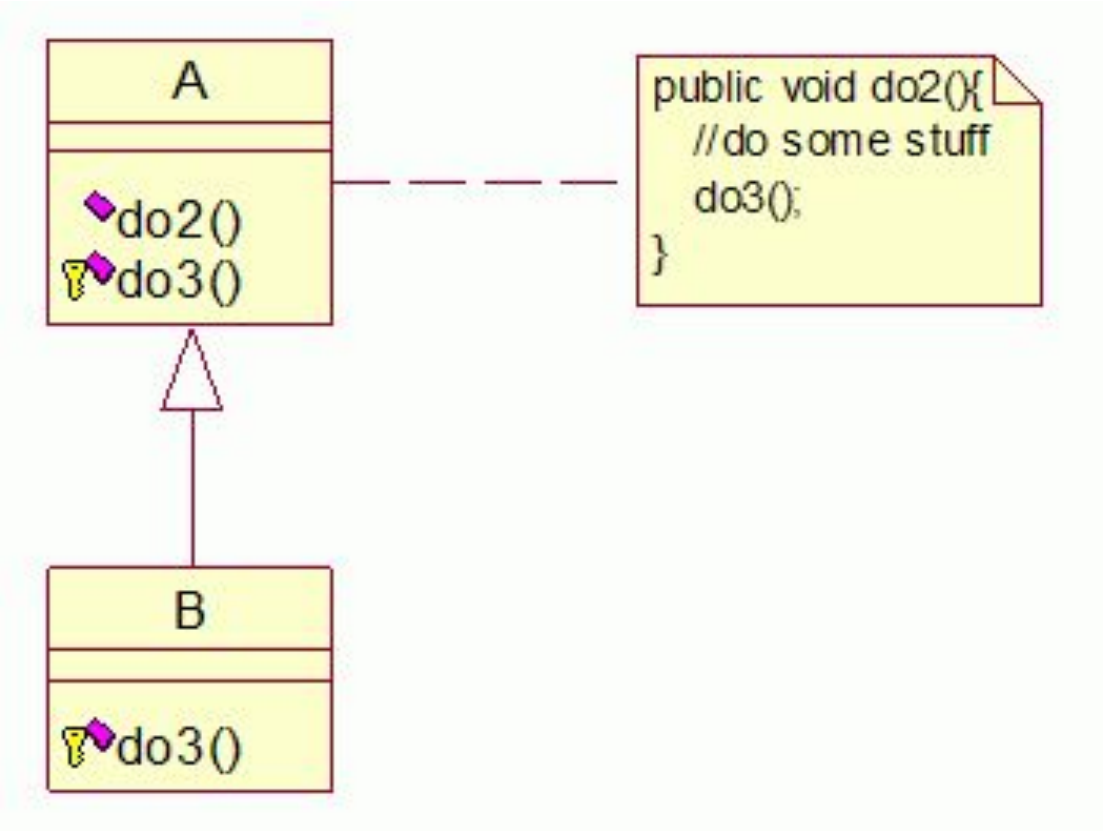
Если бы A переопределил метод wait(), то появилась бы зависимость времени выполнения от Test к Object и A.

Наследование без полиморфизма



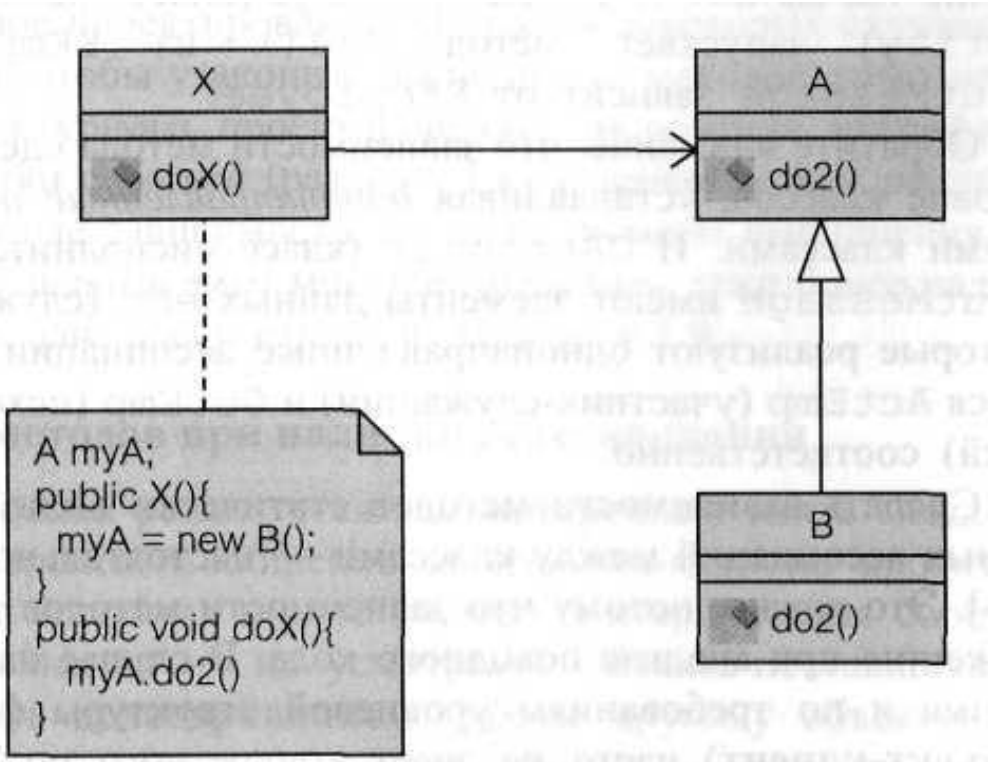
Если подкласс не переопределяет унаследованные методы, то это будет наследова-ние без полиморфизма.

Расширяющее наследование



Подкласс B наследует do2() и do3() и переопределяет do3(), возможно, *расширяя* функциональные возможности, полученные от унаследованного метода. Метод do3() может быть вызван как из объекта A, так и из объекта B. Он будет выполняться по-разному в зависимости от того, из какого объекта вызван.

Вызовы методов подкласса

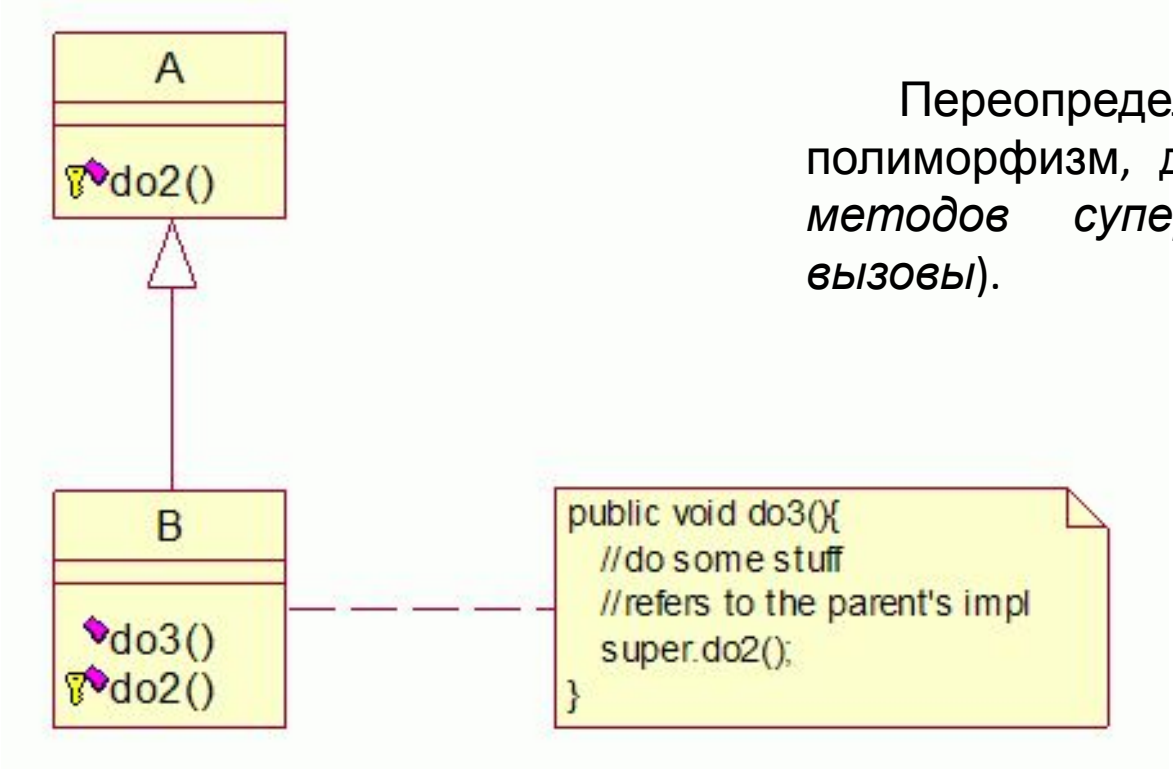


Переопределение, а следова-тельно, и полиморфизм, дает возможность использовать вызовы методов подкласса.

Даже притом, что класс X связан с классом A, тем не менее, когда X выполняет do2(), он фактически выполняет реализацию do2() класса B. Это происходит потому, что do2() выполняется на объекте myA, который является эк земпляром класса B.

Вызов метода подкласса на рисунке представляет зависимость наследования времени выполнения от X к B, которая не видна в статической структуре программы времени компиляции. X имеет связь с A, но не с B. Такими зависимостями времени выполнения очень трудно управлять

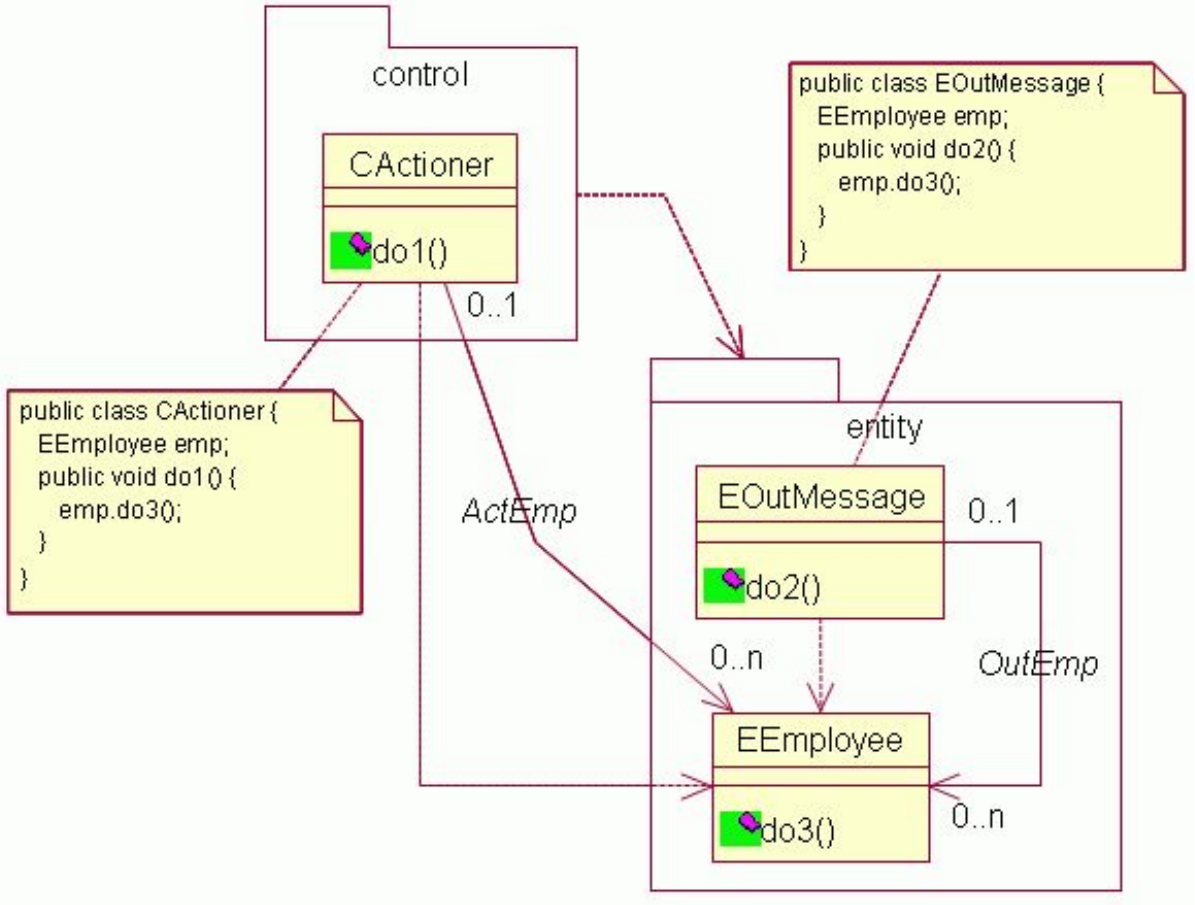
Вызовы методов суперкласса



Переопределение, а следовательно, и полиморфизм, допускает также и *вызовы методов суперкласса* (или *обратные вызовы*).

Подкласс B явно использует реализацию предка do2(), хотя фактически и имеет свою собственную переопределенную версию do2(). Хотя такой вызов через super (base) и можно объяснить, комбинация вызовов методов подкласса и суперкласса представляет неприятную проблему циклической зависимости вовлеченных в это классов.

Зависимости методов и вытекающие из этого зависимости классов и пакетов



Зависимости между мето-дами создают проблему, пото-му что многие зависимости методов нельзя проследить, анализируя только статичес-кую структуру программы времени компиляции.

CActioner использует метод do1(), чтобы послать сообщение do3() классу EEmployee. Поэтому do1() за висит от do3().

Зависимость распростра-няется на классы-владельцы и пакеты. CActioner зависит от

Сделать зависимости методов видимыми посредством явных ассоциаций между классами — настоятельно рекомендуемая практика.

Зависимости методов при наличии делегирования

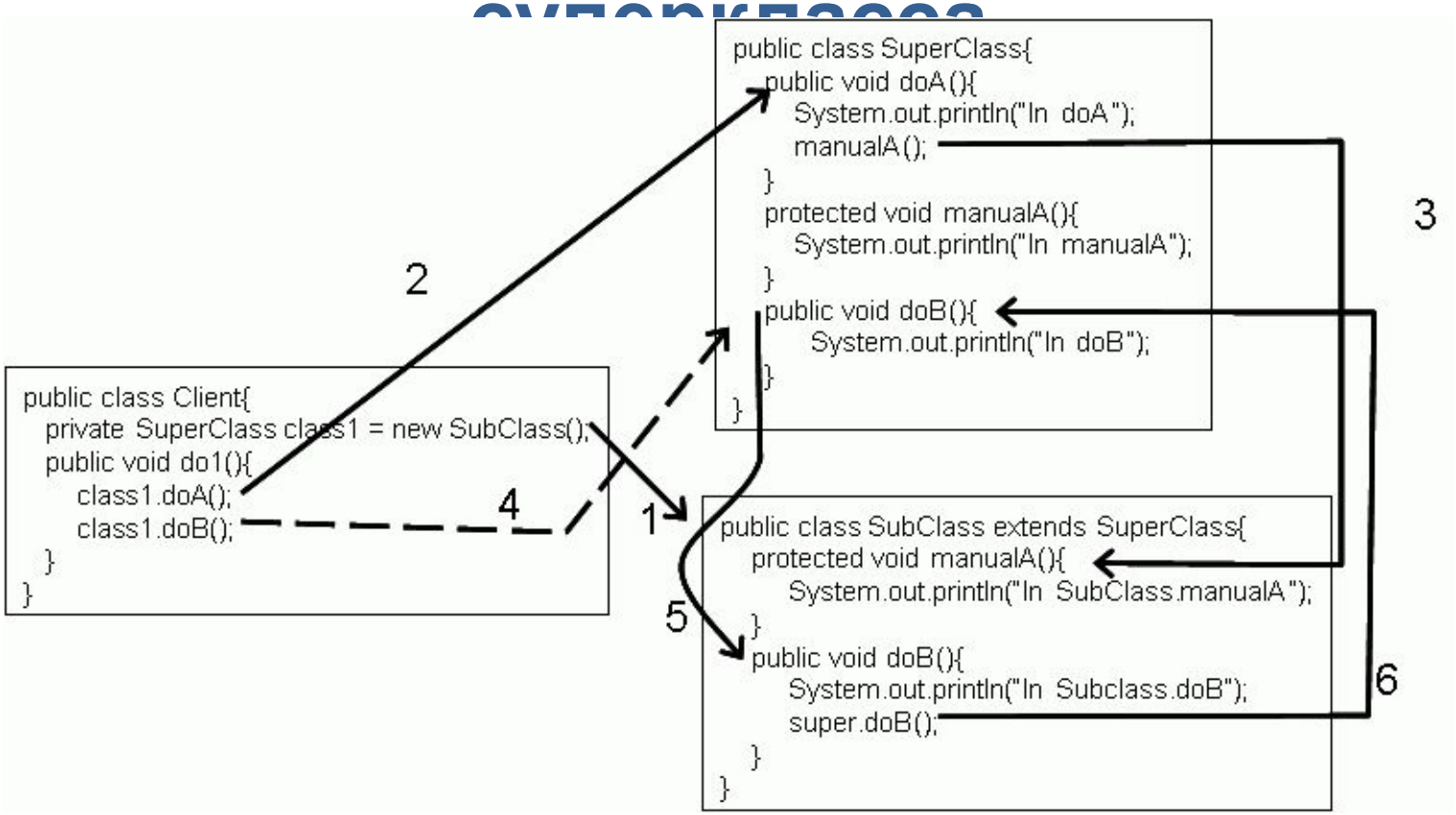
CActioner и EOutMessage запрашивают сервис do3() от EEmployee. EEmployee делеги-рует выполнение сервиса классу MBroker, а MBroker делегирует его далее классу FUpdater. FUpdater исполняет сервис.

Клиент может и не знать своего реального *поставщи-ка*.

Без явных ассоциаций анализ воздействий из-за изменений в поставляемом коде может быть невозможен.

Делегирование часто необходимо для согласования вертикальной структуры слоёв, которая не допускает прямую связь между не соседними слоями.

Вызовы методов подкласса и суперкласса



Client имеет ссылку (class1) на Subclass - подкласс, но вместо этого хранит эту ссылку как тип Superclass - суперкласс (1). Назначение экземпляра Subclass ссылке на Superclass произойдет во время выполнения. Когда вызывается метод do1(), он выполняет doA () класса Superclass (2). Кажется, что doA() вызывает метод manualA(). Однако manualA класса Superclass переопределен, и поэтому вместо него вызывается переопределенный метод (3). Выполнение (2) и (3) — пример вызова метода подкласса.

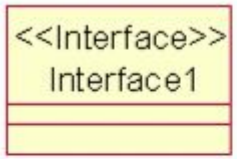
Интерфейсы

Интерфейс реализуется с помощью объекта класса, который обеспечивает пре-доставление структуры и поведения интерфейса для остальной части программы.

Интерфейсы и абстрактные классы, используемые в качестве основного входа в конкретные классы пакетов, обеспечивают механизмы, скрывающие внутреннюю сложность пакета и разрешающие расширение пакета без воздействия на объекты-клиенты в других пакетах.

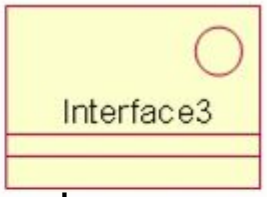
Понятие **доминирующего класса** может использоваться, чтобы реализовать эти механизмы в пределах пакета. Доминирующий класс реализует главные интерфейсы и абстрактные классы пакета.

интерфейс



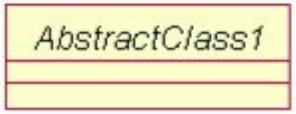
(явная метка)

(пиктограмма)

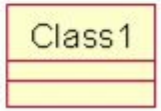


(«оформление»)

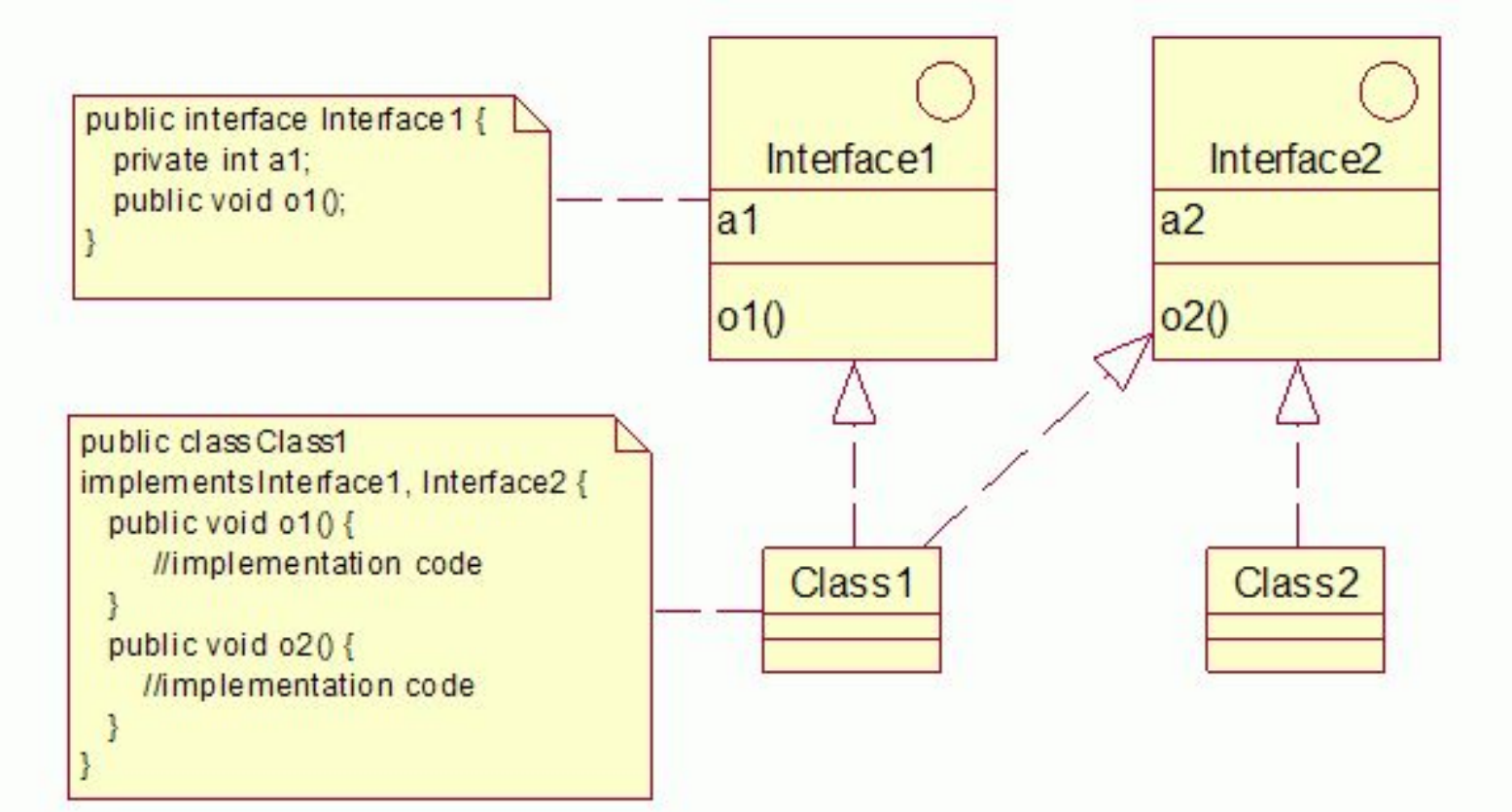
абстрактный класс



класс



Зависимость реализации



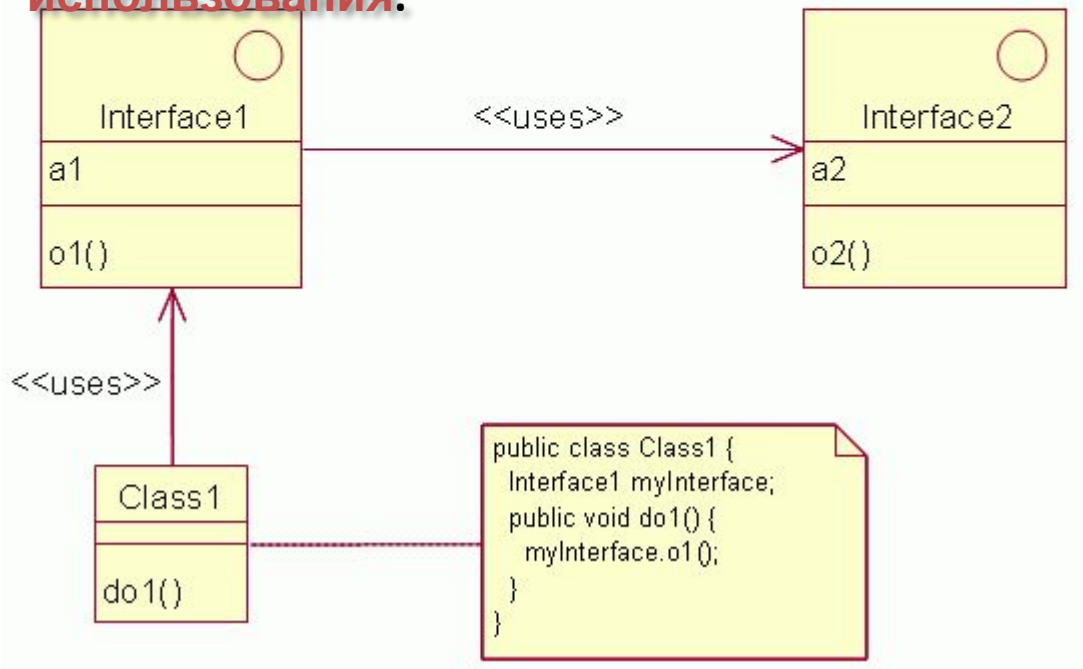
Interface1 и Interface2 называются предоставленным интерфейсом класса Class1.

Между классом и интерфейсом существует зависимость реализации.

Класс может реализовать много интерфейсов и один интерфейс может быть реализован более чем одним классом.

Зависимость использования

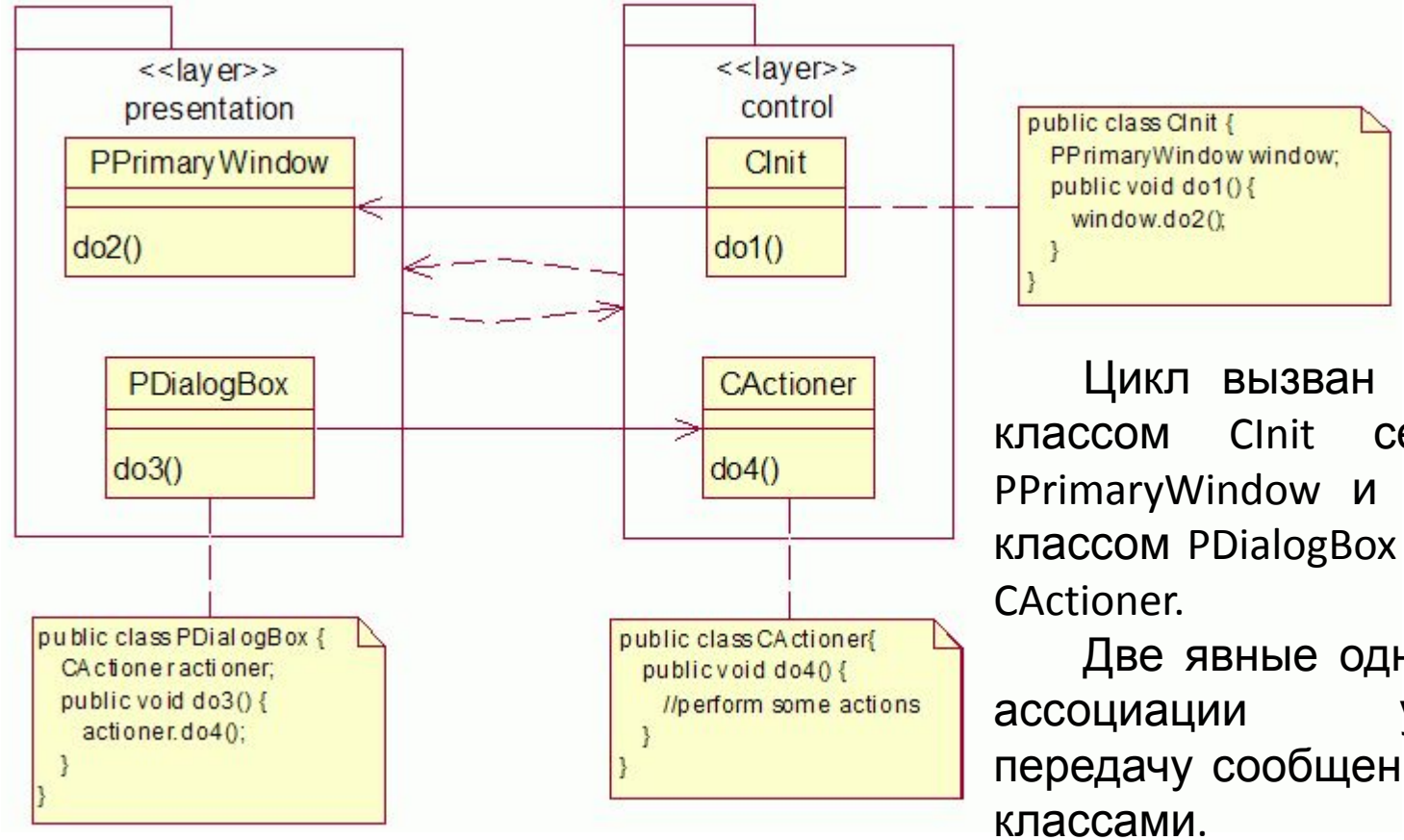
После своего объявления интерфейсы могут использоваться классами (или другими интерфейсами), которые затребуют их. Это называется **зависимостью использования**.



Class1 содержит метод do1(), который вызывает услуги операции o1(). В статическом коде неясно, какая реализация требуемого интерфейса будет выполнять услугу.

Это может быть экземпляр любого класса, который реализует Interface1. Точный экземпляр будет определен во время выполнения, когда выполняемый экземпляр класса Class1 задает величину элемента данных myInterface, чтобы обратиться к конкретному объекту конкретного класса.

Циклическая зависимость

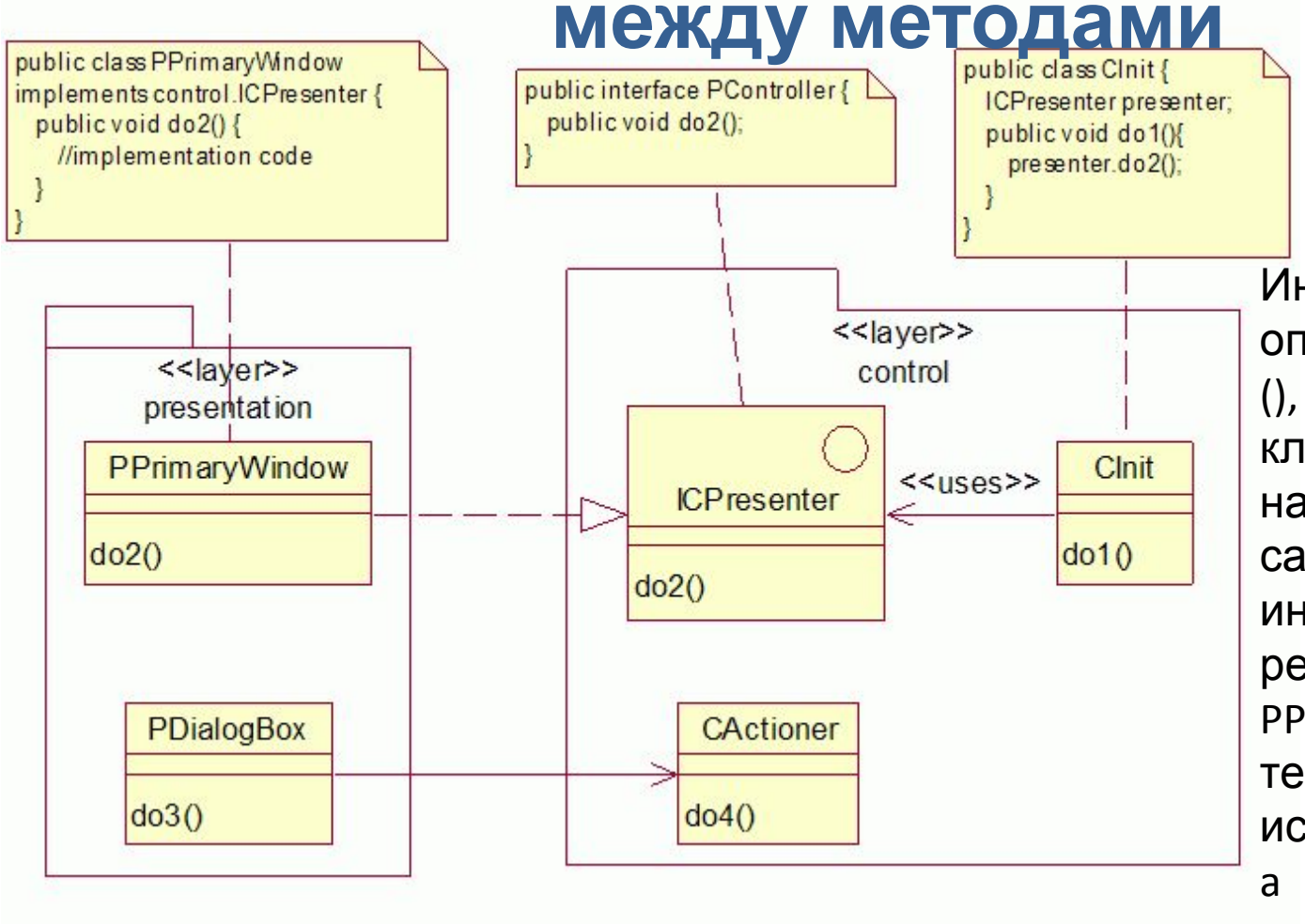


Цикл вызван использованием классом CInit сервисов класса PPrimaryWindow и использованием классом PDialogBox сервисов класса CActioner.

Две явные однонаправленные ассоциации устанавливают передачу сообщений между этими классами.

Интерфейсы могут успешно использоваться для уменьшения зависимостей в коде. Программирование с интерфейсами позволяет объектам-клиентам не знать конкретные классы объектов, которые они используют, и классы, которые фактически реализуют эти интерфейсы.

Использование интерфейса для исключения циклической зависимости между методами



Интерфейс определяет метод do2 (), необходи-мый для класса CInit, который находится в том же самом пакете. Но интерфейс фактически реализован классом PPrimaryWindow в паке-те presentation. CInit использует интерфейс, а PPrimaryWindow реали-зует его.

Для того чтобы нарушить цикл, интерфейс и класс, который реализует его, должны находиться в различных пакетах. Паттерн **Отделённого интерфейса**.

Обработка событий

Кроме *синхронных связей* между объектами бывают и **асинхронные связи**, где методы «инициализируются», чтобы обслужить асинхронные **события**.

В обработке событий происходит отделение создателя события (**объект-издатель**) от различных **приемников**/**наблюдателей** событий (обычно называемых **объектами-подписчиками**).

В больших системах выполнять подписку может отдельный **объект-регистратор**, который обеспечивает «взаимодействие» между издателем и подписчиками. Чтобы зарегистрировать подписчика у объекта-издателя, объект-регистратор, действующий в интересах подписчика, вызывает метод издателя addActionListener() (добавить приемник операций) с объектом-подписчиком в качестве аргумента.

Обычно, объект-издатель создает **объект-событие** — издатель переводит внутренний смысл события в объект-событие (называя его как-то вроде BCommandButtonEvent - объект «событие от командной кнопки B»). Объект-событие передается (в режиме *обратной связи)* всем подписчикам, которые зафиксировали свои интересы в отношении нажатия кнопки мышью.

Обработка событий и зависимости слоёв

Если при *синхронной передаче сообщения* объект-клиент A посылает сообщение объекту-поставщику B, то A зависит от B, потому что A ожидает некоторые результаты выполнения от B.

В *асинхронной обработке события* отправитель сообщения — объект-издатель, но передача сообщения обрабатывается как обратный вызов. При обратной связи издатель не имеет никакой информации как подписчик обрабатывает событие.

Зависимость существует, но она незначительна с точки зрения структурного проектирования.

Подтверждение связи подписчиков и издателей вызывает более сильную зависимость. Если объект-регистратор добивается подтверждения связи, то это зависит и от издателя, и от подписчика. Если объект-подписчик фиксирует подтверждение связи сам, то он (подписчик) зависит от издателя. Чтобы ослабить зависимости, связанные с подтверждением связи, подписчиков можно передавать методам регистрации в качестве аргументов, представ ленных как интерфейсы. В этом случае зависимости ослабляются, но будет требоваться дальнейший анализ программы, чтобы определить класс подписчика. Анализ включает определение классов, которые реализуют интерфейс подписчика.

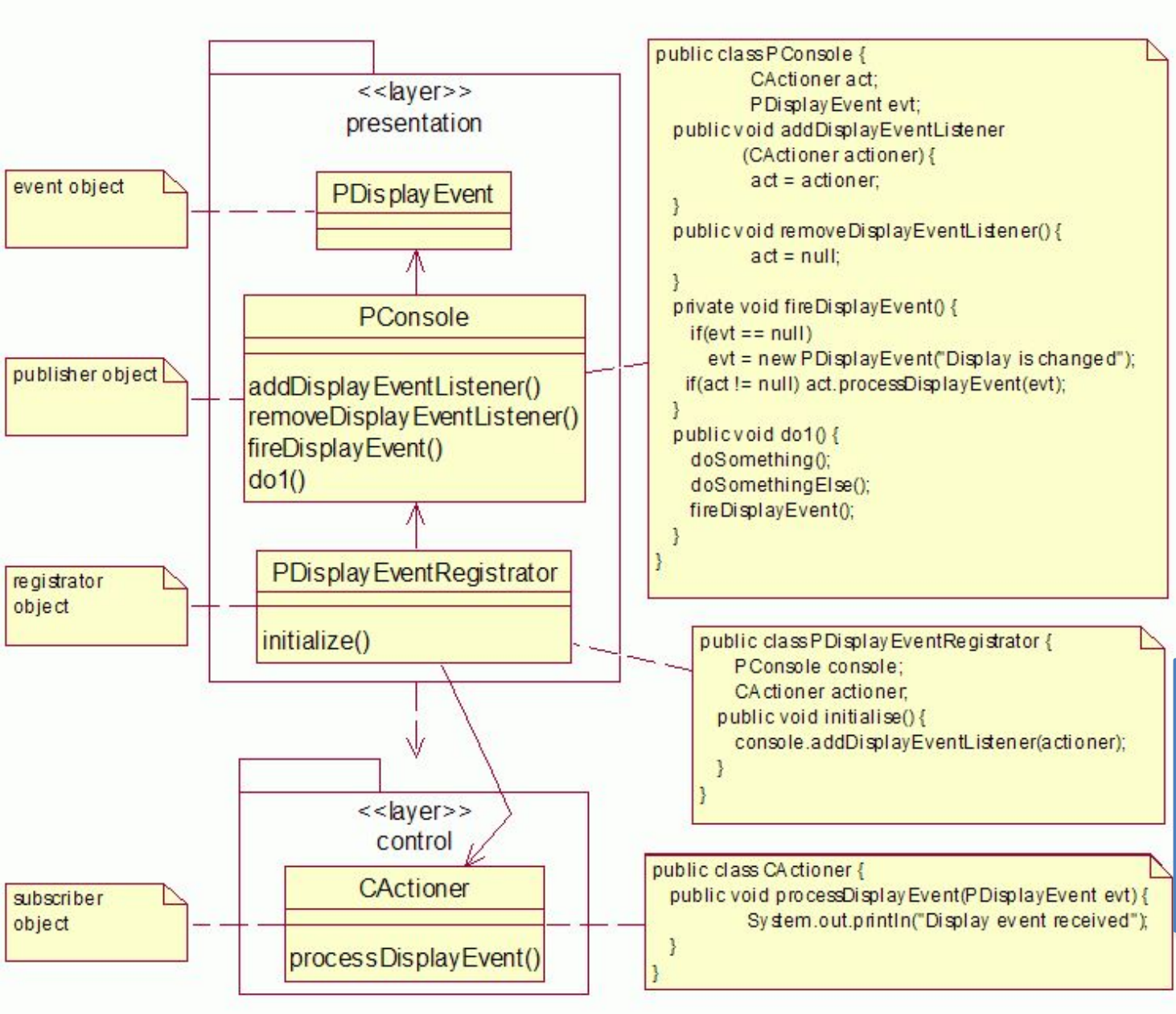
Соглашения именования

Методы, выполняющие регистрацию подписчиков для издателя, называются, начиная с фразы **add** (добавить) и кончая фразой **Listener** (приемник). Имя объекта-события (XXX) помещается между этими двумя фразами, например **addXXXListener**.

Класс-издатель будет обычно содержать «нормальные» методы, вместе с методами, инициализирующими события. Чтобы отличать последние, они начинаются с фразы **fire** (инициализировать), например: **fireCommandButtonEvent()** (инициализация события от командной кнопки).

Метод fire пробегает список подписчиков и для каждого подписчика вызывает его собственный метод process (обработка). Для удобства имя метода process может начинаться с фразы process, например, **processCommandButtonEvent()** (обработка события от командной кнопки).

Обработка событий и зависимости слоёв



CActioner — единственный подписчик на события класса Pconsole. PDisplayEventRegistrator формирует подтверждение связи классам PConsole и CActioner. Метод do1() перехватывает и интерпретирует события, что инициализирует объект fireDisplayEvent(). PConsole создает объект PDisplayEvent.

Помещение PDisplayEventRegistrator в пакет presentation создает приемлемую нисходящую зависи мость между пакетами presentation и control.

Использование интерфейсов для уменьшения зависимостей от обработки событий

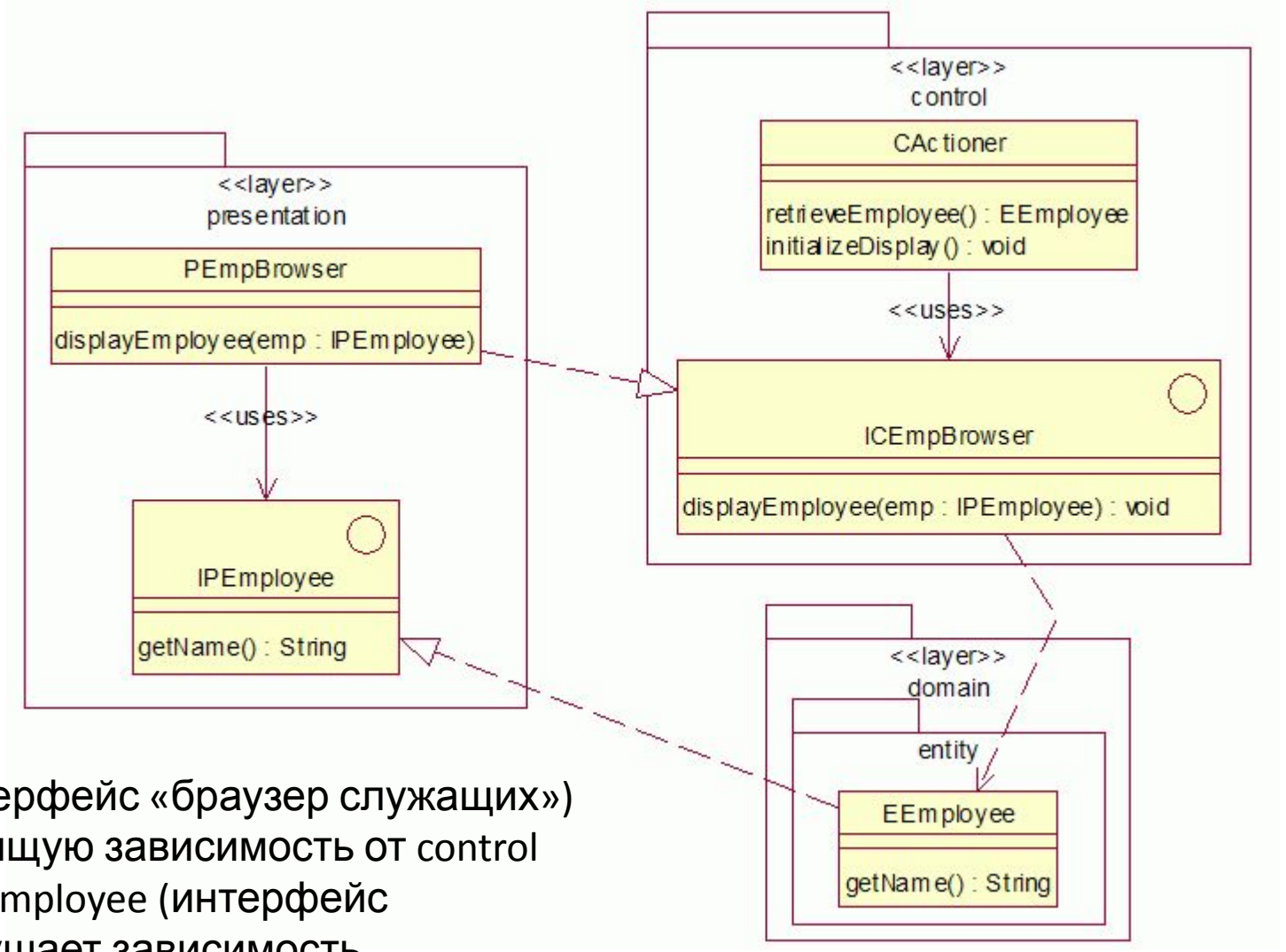
Интерфейс IPDisplayEventSubscriber (интерфейс «подписчик на события отображения») отделяет PConsole от CActioner. PConsole создает объект PDisplayEvent и использует интерфейс IPDisplayEventSubscriber, чтобы известить CActioner относительно объекта PDisplayEvent. CActioner предварительно обрабатывает PDisplayEvent с помощью своего метода processDisplayEvent () и создает объект PDecisionEvent.

Знакомство

Знакомство соответствует ситуации, когда объект передает другой объект как аргумент своего метода. Объект A знакомится с объектом B, если другой объект C передает B к A в качестве аргумента сообщения к A.

Зависимости знакомства - *зависимости методов,* возникающие динамически во время выполнения.

Использование интерфейсов для понижения зависимостей

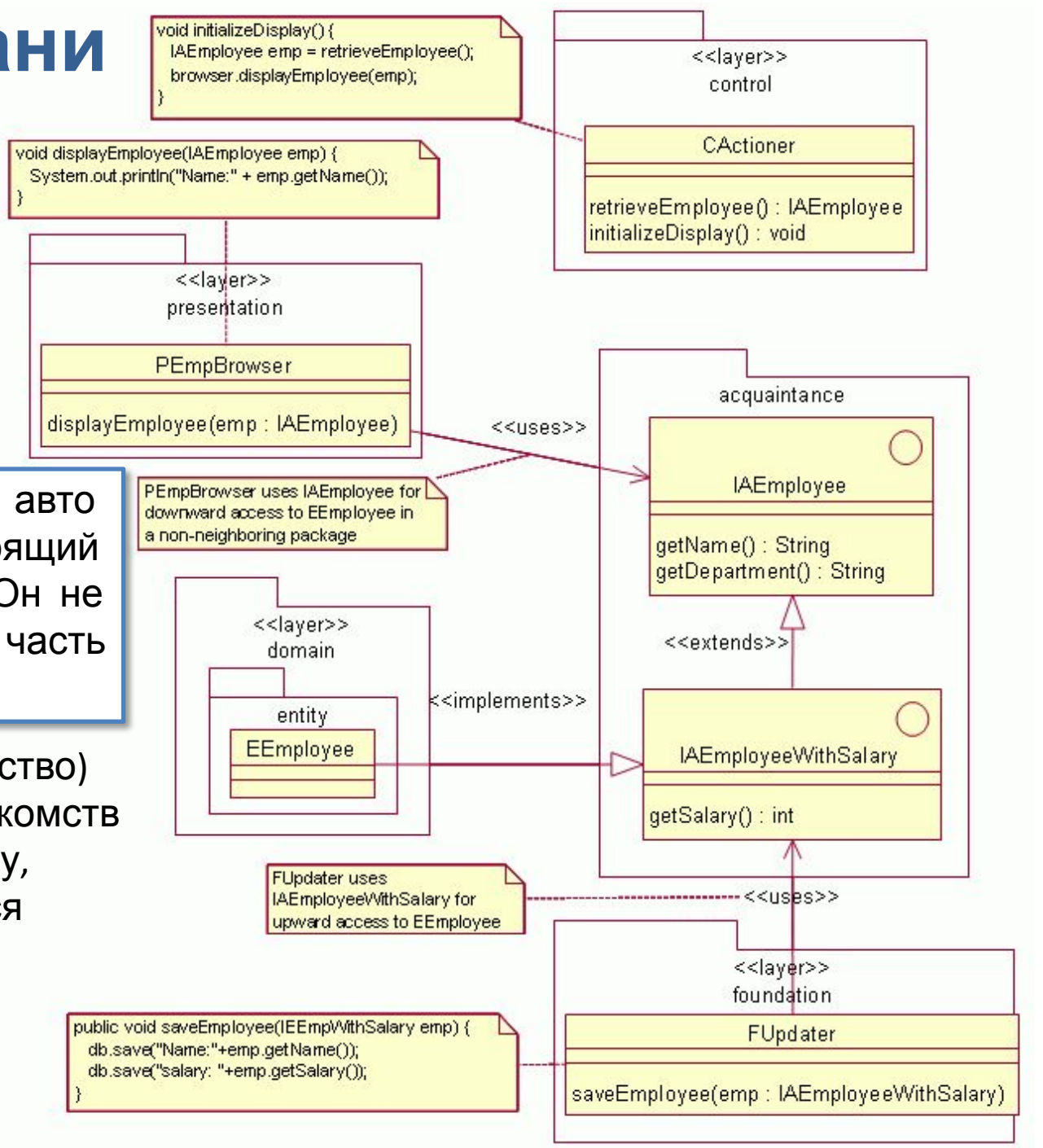


ICEmpBrowser (интерфейс «браузер служащих») нарушает восходящую зависимость от control до presentation. IPEmployee (интерфейс «сотрудник») нарушает зависимость знакомства от presentation до domain.

Использовани е пакета знакомств

Пакет знакомств — это авто номный пакет, состоящий только из интерфейсов. Он не представ-ляет ни слой, ни часть иерархии слоёв.

Пакет acquaintance (знакомство) отделяет зависимости знакомств в изолированную проблему, которая может управляться независимо.
Используется *принципа разделения задач.*



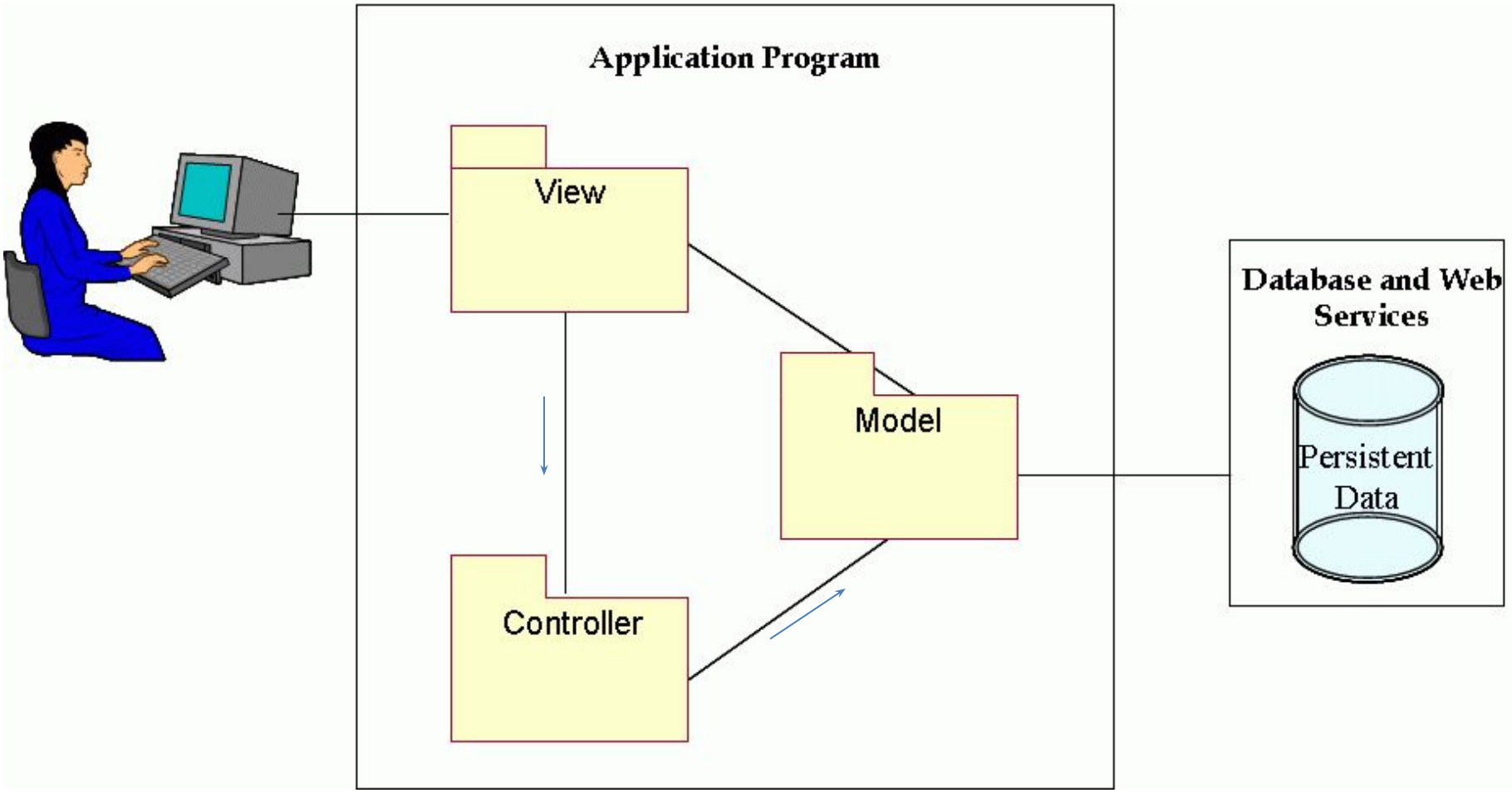
Структурные шаблоны

В объектно-ориентированной технологии **шаблон** — технология повторного использования проекта.

Паттерн проекта называет и объясняет лучшие и широко подтвержденные практику и знания, используемые для решения задач проектирования.

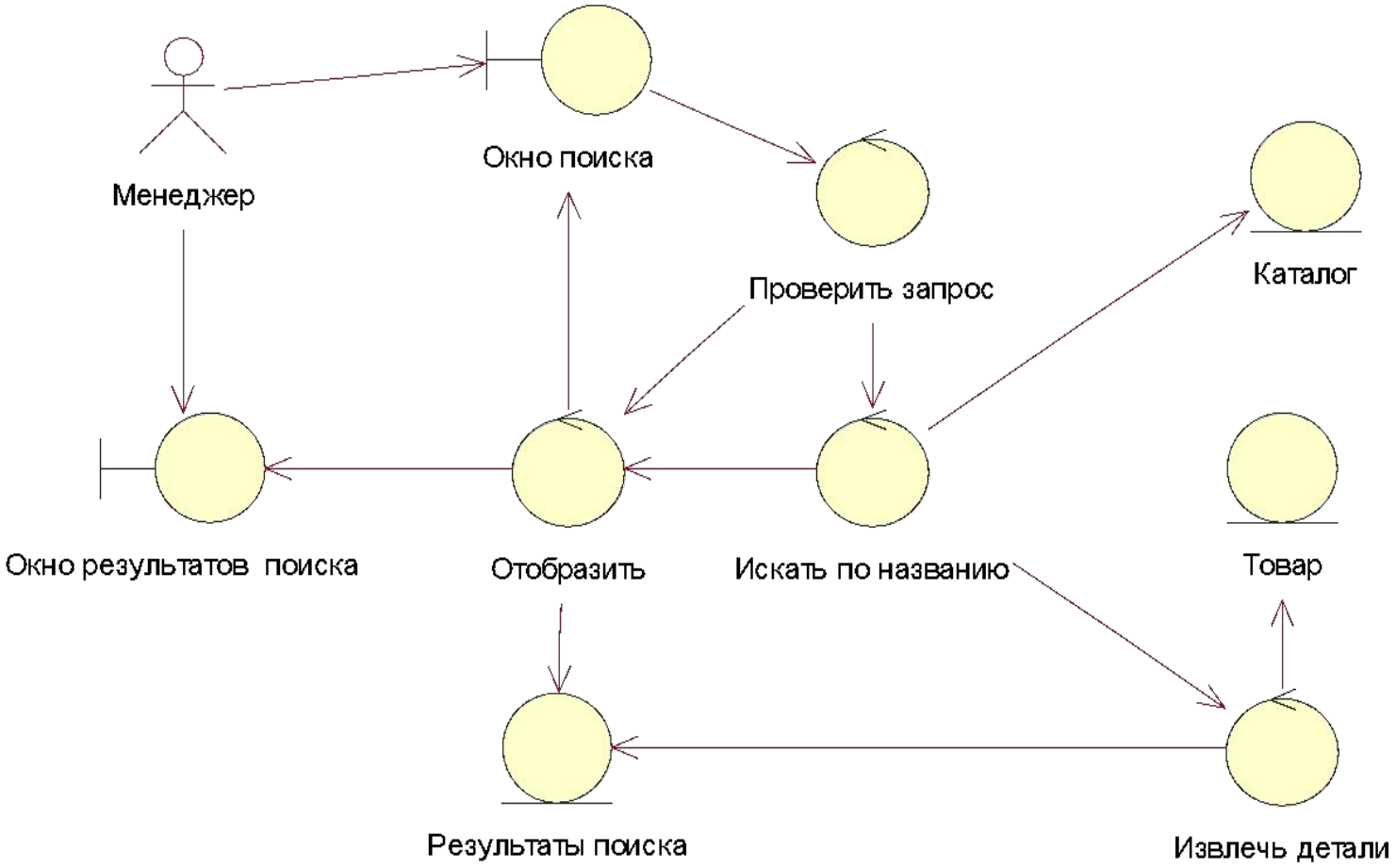
Шаблон обеспечивает скелет решения проблемы, который должен быть настроен и расширен, чтобы он мог выполнять полезную функцию. Настройка включает написание определенного кода, который «заполняет пробелы» в шаблоне (то есть, который реализует различные элементы шаблона, приспосабливая их к окончательному структурному и поведенческому проекту).

Шаблон MVC



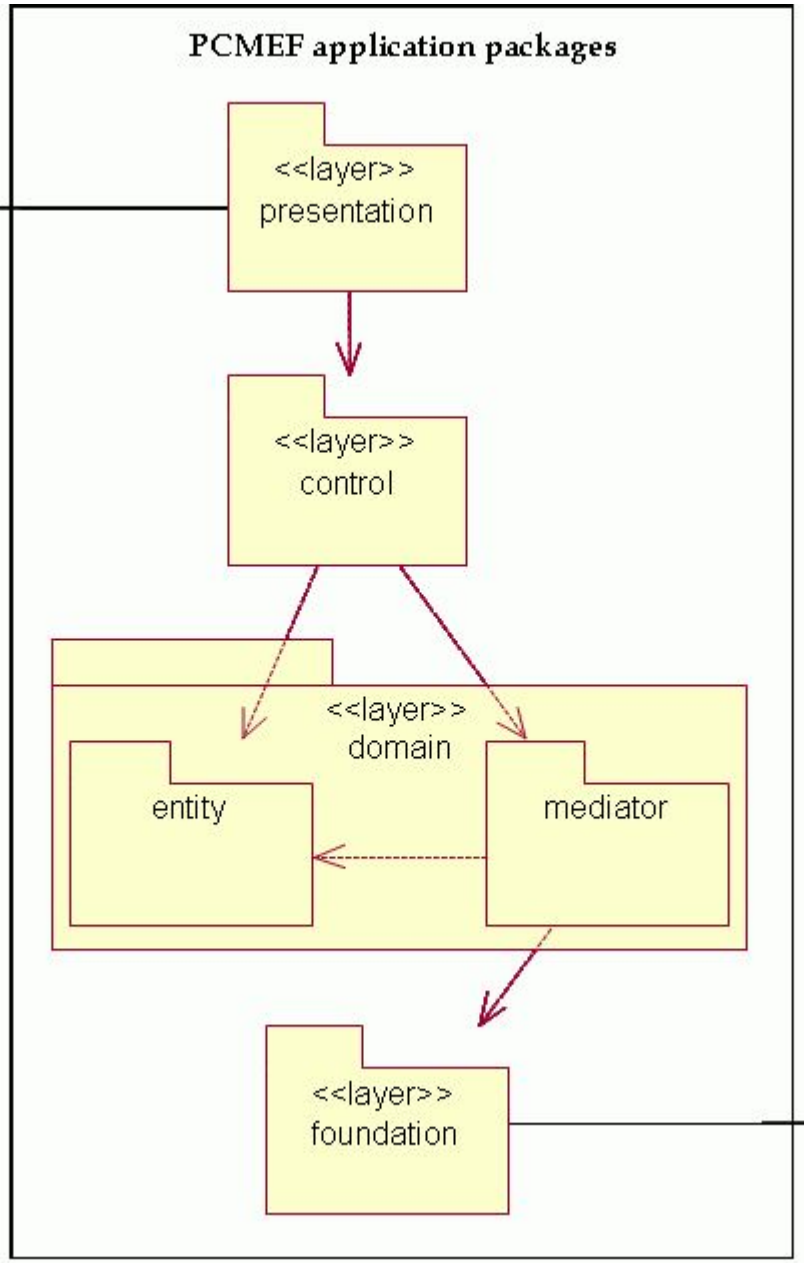
Пользователь активизирует команду меню, чтобы отобразить информацию о клиенте на экране. Объект View получает событие и передает его своему объекту Controller. Объект Controller просит модель обеспечить данные о клиенте. Объект Model возвращает данные объекту Controller, который передает их View, для отображения.

Boundary – Control – Entity



Unified Process (UP) — унифицированный процесс — использует принципы MVC при разделении классов на *граничные объекты (boundary)*, *объекты управления (control)* и *объекты-сущности* (*entity*).

PCMEF-шаблон



пред
ставление
управление
посредник
сущность
основание

```
package presentation;
import control.*;

package control;
import domain.entity.*;
import domain.mediator.*;

package entity;

package mediator;
import entity.*;
import foundation.*;

package foundation;
```

Слои PCMEF в C#

```
using System;
using control;
namespace presentation {
// слой Представление
}

using System;
using domain.entity;
using domain.mediator;
namespace control {
// слой Управление
}

using System;
using entity;
using foundation;
namespace domain {
// слой Предметная область
  namespace mediator {
  // пакет Посредник
  }
}
```

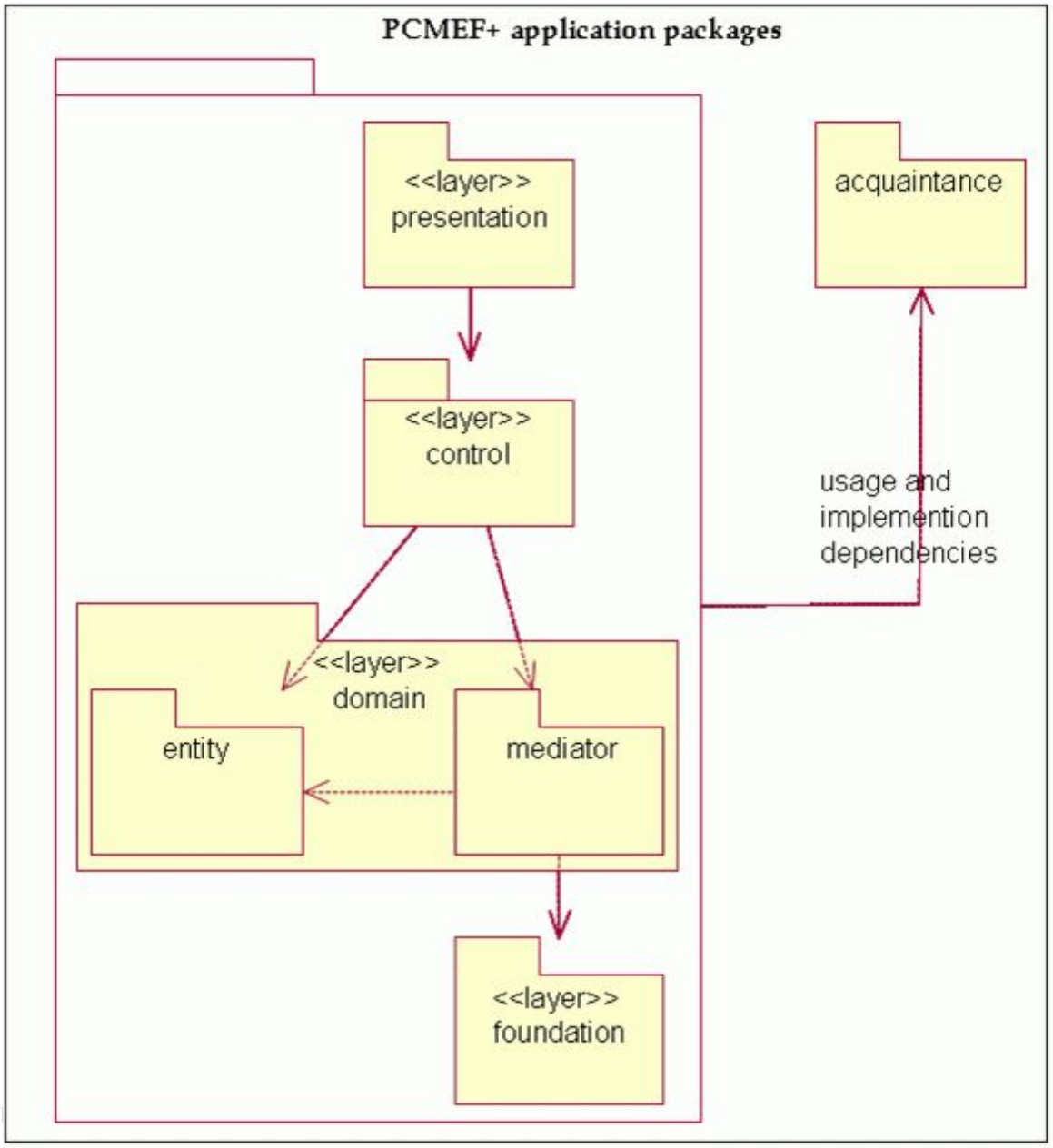
```
using System;
namespace domain {
// слой Предметная область
  namespace entity {
  // пакет Сущность
  }
}

using System;
namespace foundation {
// слой Основание
}
```

Принципы PCMEF

- принцип **нисходящей зависимости** (Downward Dependency Principle — DDP);
- принцип **восходящего уведомления** (Upward Notification Principle — UNP);
- принцип **соседней связи** (Neighbor Communication Principle —NCP);
- принцип **явной ассоциации** (Explicit Association Principle — EAP);
- принцип **устранения циклов** (Cycle Elimination Principle — CEP);
- принцип **именования классов** (Class Naming Principle — CNP);
- принцип **использования пакета знакомств** (Acquaintance Package Principle — APP).

Пакет знакомств в PCMEF+

Пакет acquaintance состоит из интерфейсов, которые пере-дают вместо конкретных объектов эти объекты в ка-честве аргумента при вызове методов. Интерфейсы могут быть реализованы в любом PCMEF-пакете. Это позволяет осуществлять **эффективную связь между несоседними па кетами** при централизации управления зависимостями в единственном пакете знакомств.

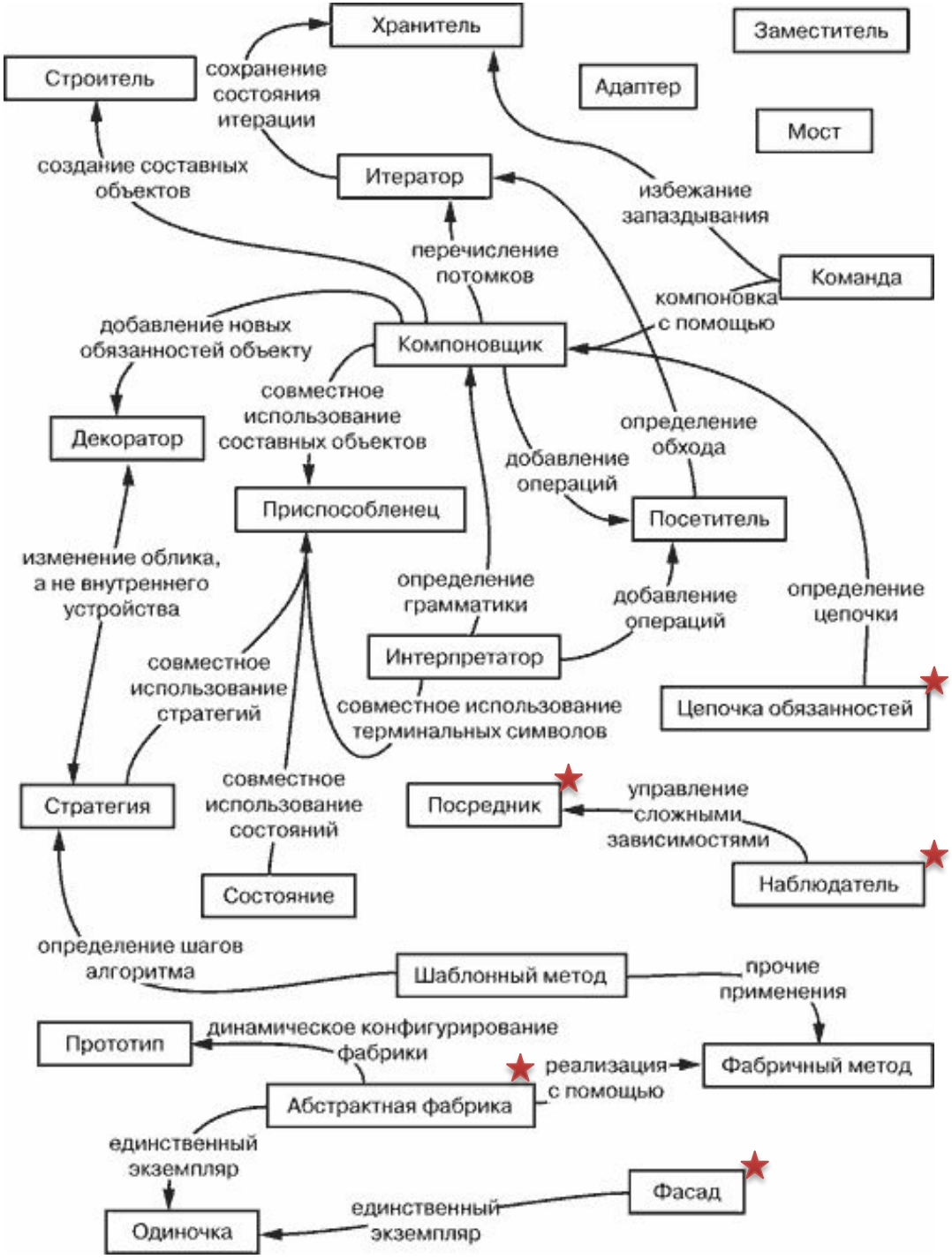


Развертывание PCMEF-слоёв

Развертывание PCMEF-слоёв как компонентов завершается некоторой **клиент-серверной структурой.**

Слои устроены так, что их можно размещать неза-висимо, как компоненты.

Компонент определя-ется в UML как «модуль-ная часть системы, ко-торая инкапсулирует ее содержание и чье объявление можно заменить в пределах среды этой системы». UML 2.0 обеспечивает также диаграмму развёртывания компо-нентов, чтобы показать ввод их в действие в среде



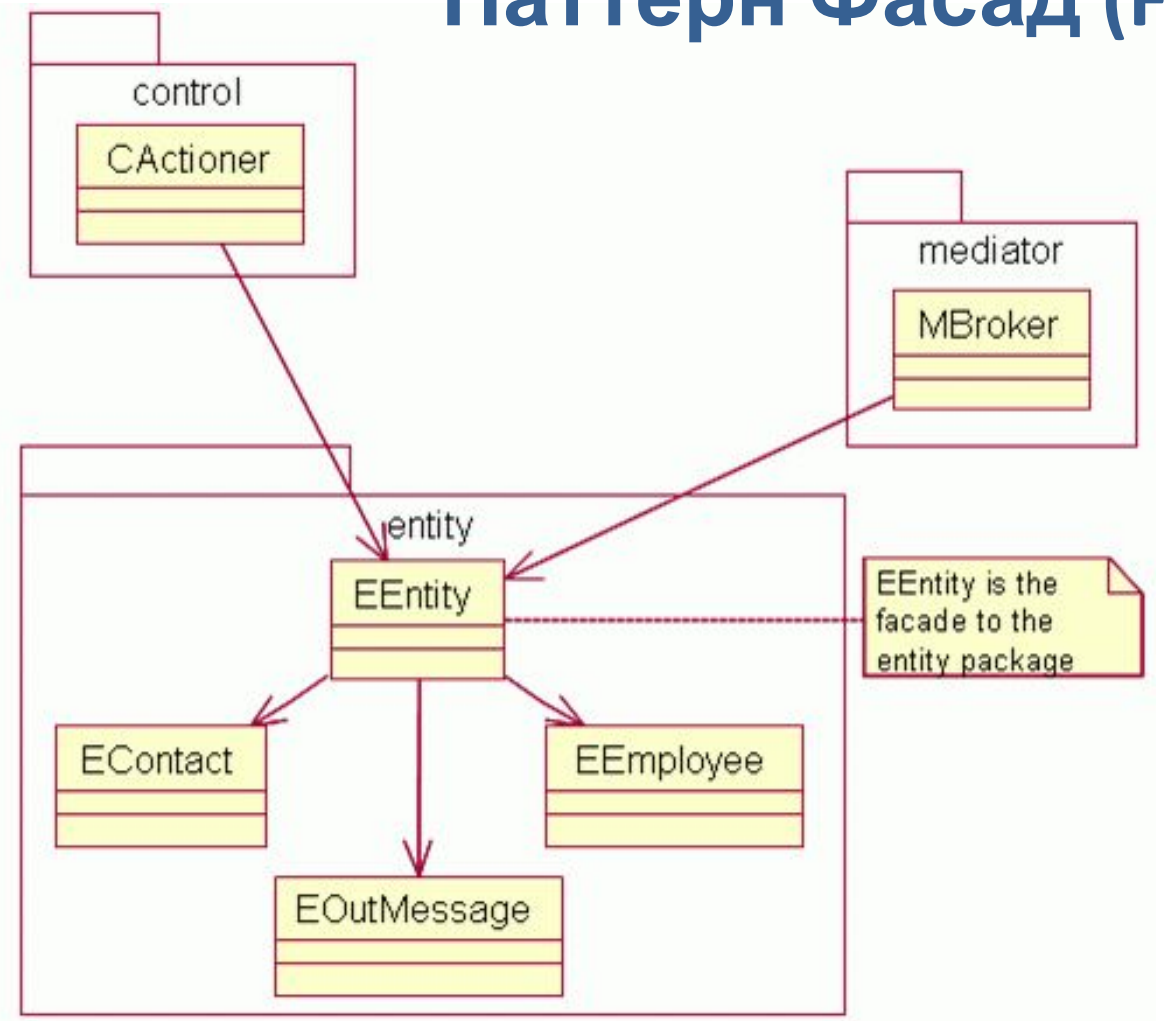
Структурные паттерны

Когда паттерн служит целям структурного проектирования, он называется *структурным паттерном.*

Паттерн структуры (pattern) **проекта** означает и объясняет лучшие и широко под-твержденные теорией и прак-тикой решения задач про-ектирования.

Отношения между паттернами Gof

Паттерн Фасад (Facade)

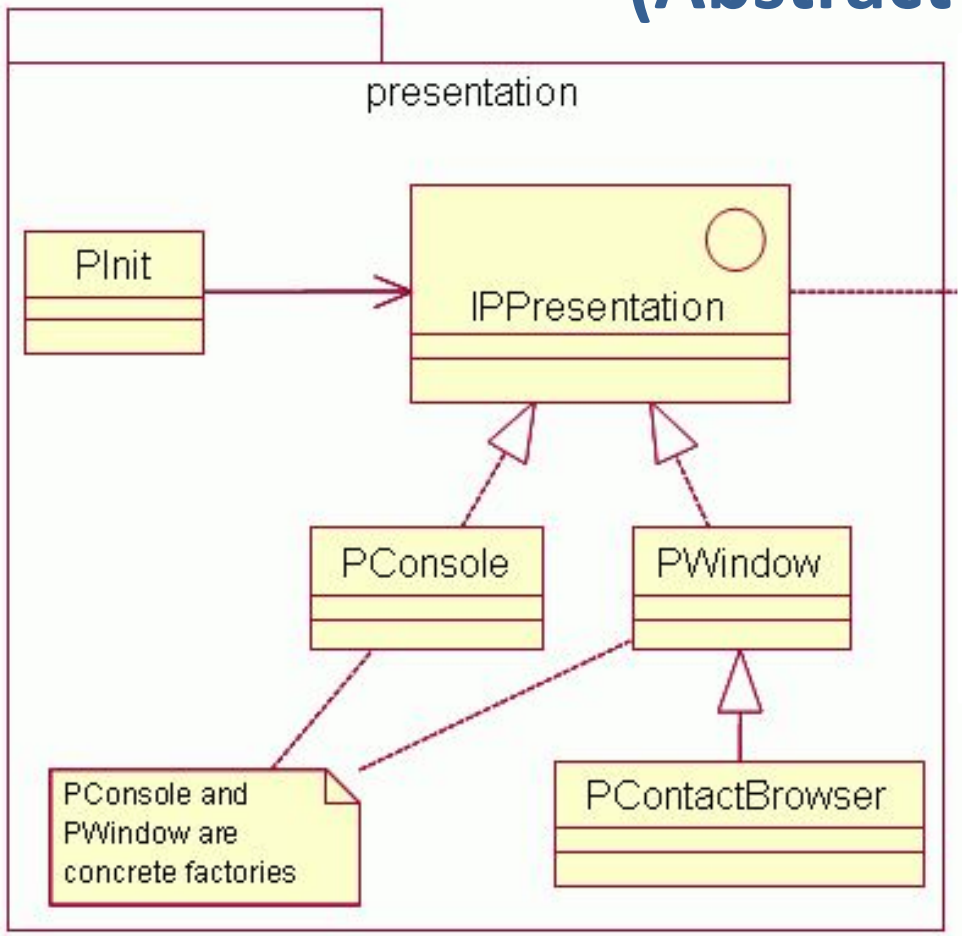


Объекты-клиенты связыва-ются с пакетом через объект Фасад. Объект Фасад делеги-рует свою работу другим объектам пакетов. Сокращаются пути связи между пакетами и уменьшается число объектов, с которыми клиенты пакета имеют дело. Фактически пакет оказывается скрытым за объектом Фасад.

EEntity — доминирующий класс. Делая его интерфейсом или абстрактным классом, можно далее уменьшить зависимости клиентов от пакета entity.

Интерфейс с более высоким слоем инкапсулирует главные функциональные возможности подсистемы (пакета) и обеспечивает основные или даже единственную точку входа для клиентов пакета.

Паттерн Абстрактная фабрика (Abstract Factory)

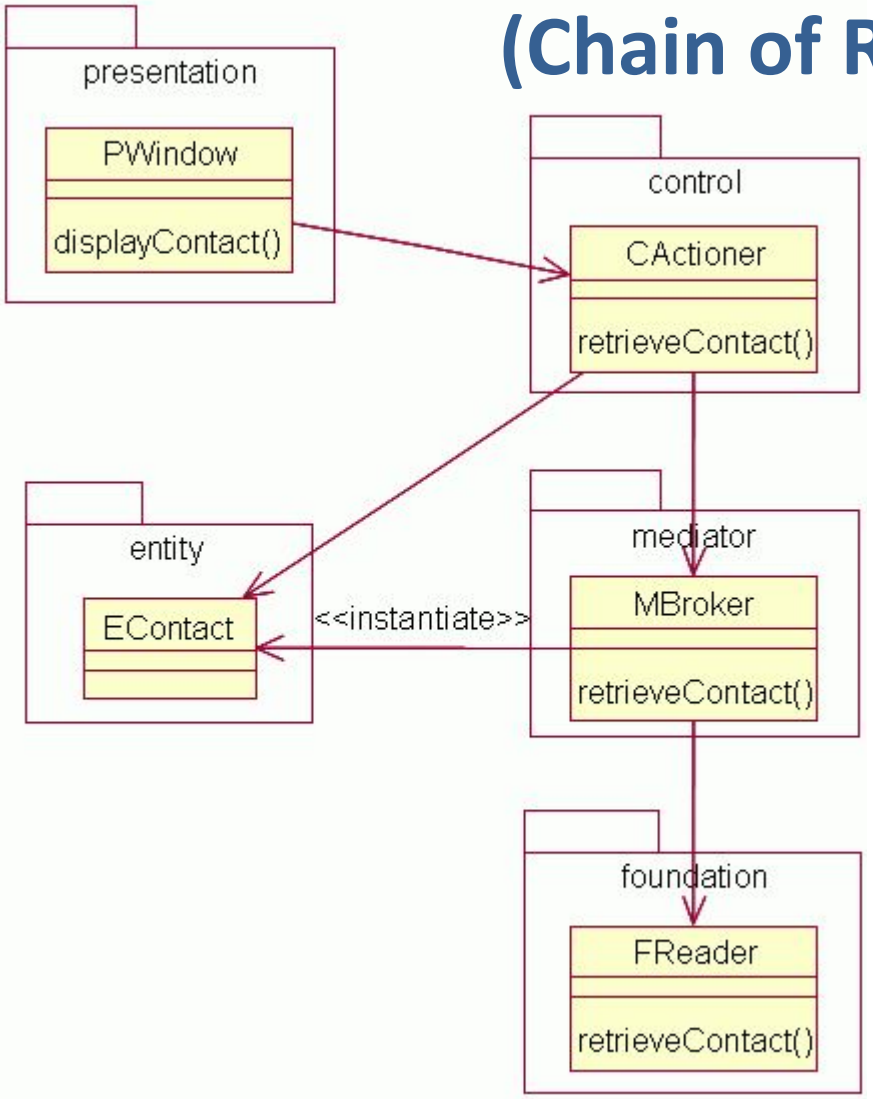


IPPresentation (интерфейс пакета presentation) - Абстрактная фабрика, которая передает создание конкретных объектов или классу PConsole или PWindow, являющимся конкрет-ными фабриками, в зависимости от того, как конфигурируется система.

Объекты-клиенты (типа PInit) обращаются к конкретным объектам через интерфейс (IPPresentation).

Обеспечивает «интерфейс для создания семейств связанных или зависимых объектов без определения их конкретных классов».

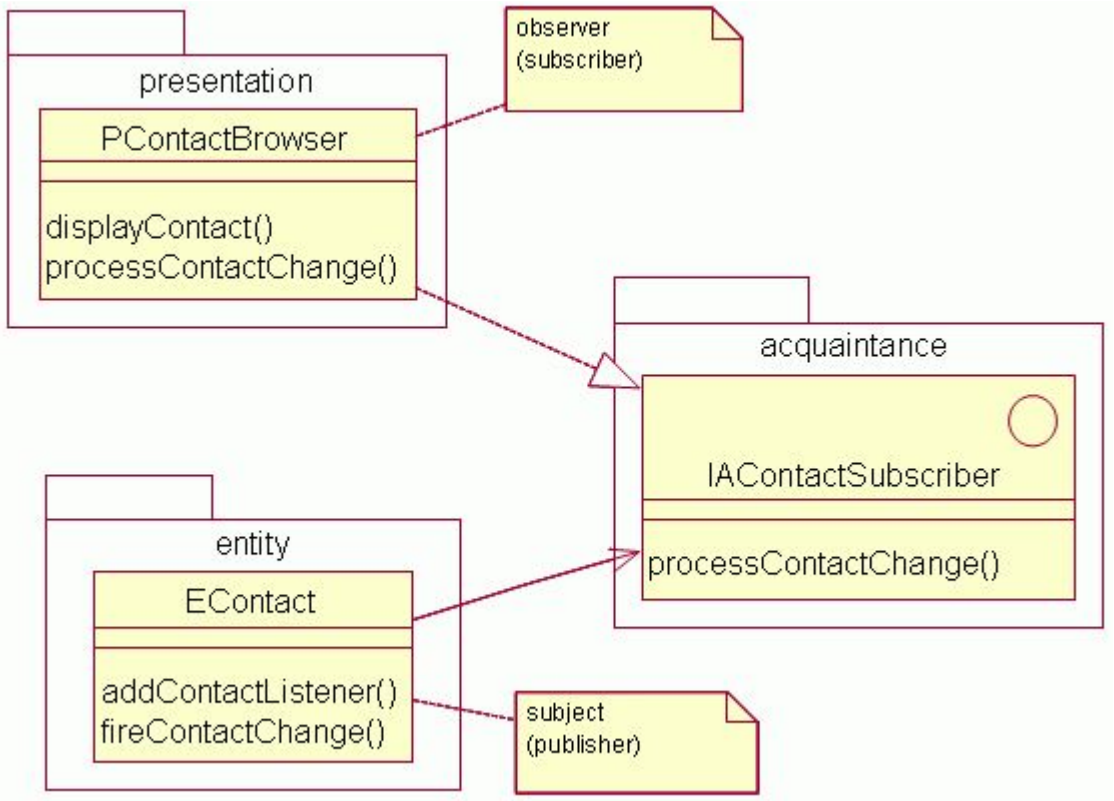
Паттерн Цепочка обязанностей (Chain of Responsibility)



Объект PWindow получает запрос displayContact() (отобразить делового партнера) и посылает сообщение retrieveContact() (извлечь делового партнера) объекту CActioner. CActioner пересылает запрос объекту EContact (если требуемый объект EContact был ранее инициализирован и существует в памяти программы) или к MBroker (если объект EContact должен быть извлечен из БД). Если объект находится в памяти, EContact обеспечит обслуживание, и объект будет возвращен объекту PWindow для отображения. Иначе MBroker делегирует запрос объекту FReader, чтобы объект мог быть извлечен из БД и инициализирован.

Цель паттерна — «избежать непосредственного соединения отправителя запроса с его получателем, давая возможность более чем одному объекту обработать запрос». Цепочка обязанностей — всего лишь другое название концепции **делегирования**.

Паттерн Наблюдатель (Observer)

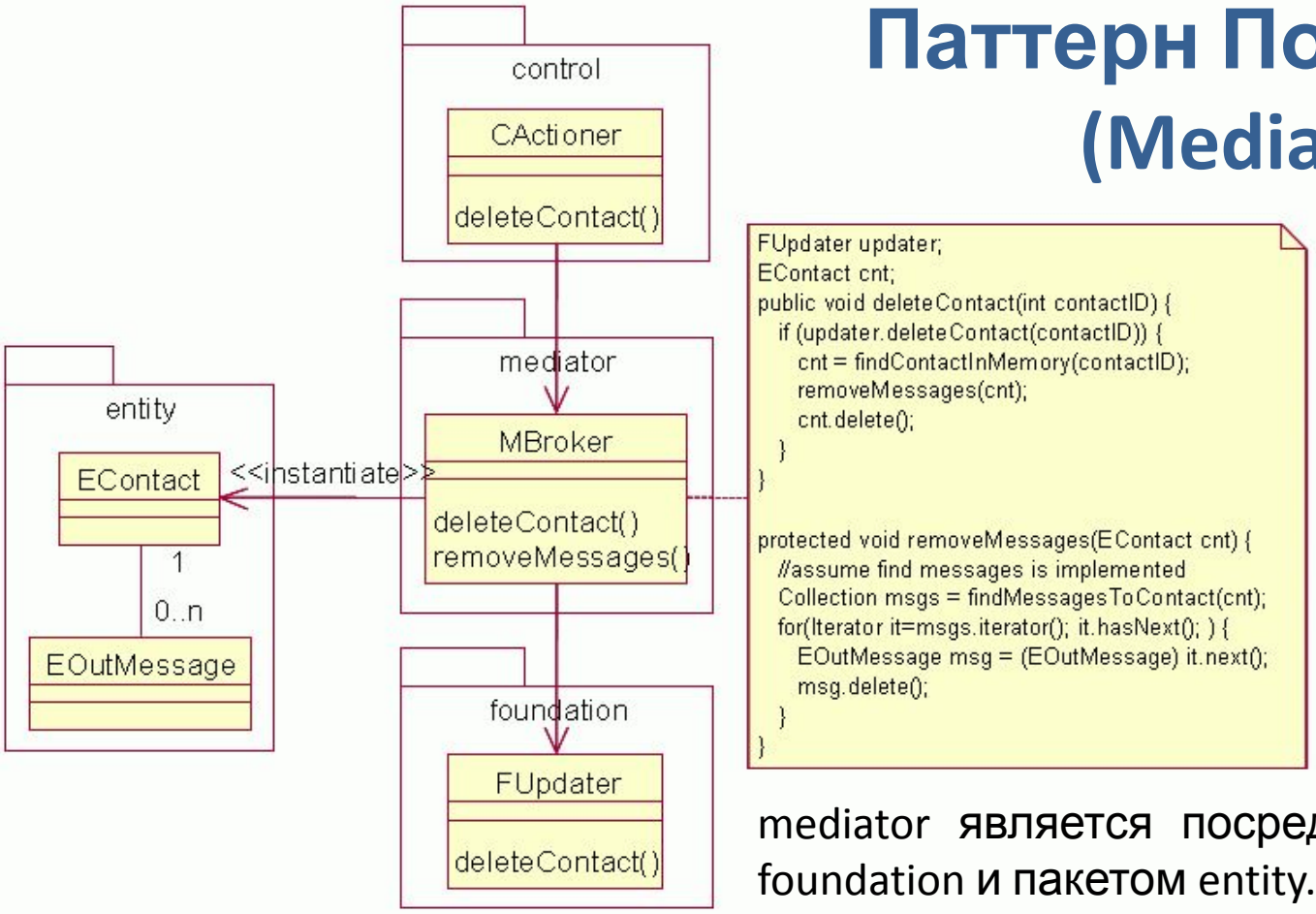


PContactBrowser подписан на EContact. EContact использует addContactListener() для регистрации IAContactSubscriber в качестве наблюдателя. В действительности наблюдателем является Pcontact-Browser, который реализует интерфейс IAContactSubscriber. Когда состояние EContact изменяется, fireContactChange() уведомляет об изменении интерфейс PContactBrowser, вызывая метод processContact-Change(). Метод processContact-Change() может затем запросить метод displayContact(), чтобы показать новое состояние EContact в окне браузера.

Паттерн **Издание-подписка** (publish/subscribe). Субъект может иметь много наблюдателей, которые подпишутся на него.

Назначение паттерна — «определить зависимость один ко многим между объектами так, чтобы в случае изменения состояния объекта все зависимые от него объекты были уведомлены и автоматически обновлены».

Паттерн Посредник (Mediator)



Паттерн Посредник определяет объекты, которые инкапсулиру-ют взаимную связь между другими объ-ектами, возможно, из различных слоёв.

Посредник заменяет связи «многие ко мно-гим» связями «один ко многим».

mediator является посредником между слоем foundation и пакетом entity.

Паттерн Посредник «обеспечивает свободную связь, храня явные ссылки объектов друг на друга, что позволяет независимо изменять их взаимодействие».

CActioner делегирует метод deleteContact () MBroker. Задача MBroker двойственная: 1) требуется делегировать метод deleteContact() далее объекту класса FUpdater, который удаляет делового партнера из БД. 2) если удаление выполнено, MBroker должен проверить, находится ли EContact в памяти программы вместе со свими EOutMessages. Если да, то MBroker посылает сообщение removeMessages(cnt), чтобы удалить EOutMessages, а затем метод cnt.delete(), чтобы удалить EContact.