Технологии программирования. Компонентный подход

В. В. Кулямин

Лекция 7. Образцы проектирования

Аннотация

Рассматривается понятие образца проектирования, классификация образцов проектирования и некоторые широко используемые примеры образцов анализа и архитектурных стилей.

Ключевые слова

Образец проектирования, архитектурный стиль, идиома, образец анализа, образец организации, образец процесса, архитектурный стиль «каналы и фильтры», архитектурный стиль «многоуровневая система».

Текст лекции

Образцы человеческой деятельности

Чем отличается работа опытного проектировщика программного обеспечения от работы новичка? Имеющийся у эксперта опыт позволяет ему аккуратнее определять задачи, которые необходимо решить, точнее выделять среди них наиболее важные и менее значимые, четче представлять ограничения, в рамках которых должна работать будущая система. Но важнее всего то, что эксперт отличается накопленными знаниями о приемлемых или не приемлемых в определенных ситуациях решениях, о свойствах программных систем, обеспечиваемых ими, и способностью быстро подготовить качественное решение сложной проблемы, опираясь на эти знания.

Давней мечтой преподавателей всех дисциплин является выделение таких знаний «в чистом виде» и эффективная передача их следующим поколениям специалистов. В области проектирования сложных систем на роль такого представления накопленного опыта во второй половине XX века стали претендовать *образцы проектирования (design patterns* или просто *patterns)*, называемые также типовыми решениями или шаблонами. Наиболее широко образцы применяются при построении сложных систем, на которые накладывается множество разнообразных требований. Одной из первых работ, которая систематически излагает довольно большой набор образцов, относящихся именно к разработке программ, стала книга [1].

На основе имеющегося опыта исследователями и практиками разработки ПО выделено множество образцов — типовых архитектур, проектных решений для отдельных подсистем и модулей или просто программистских приемов, — позволяющих получить достаточно качественные решения типовых задач, а не изобретать каждый раз велосипед.

Более того, люди, наиболее активно вовлеченные в поиск образцов проектирования в середине 90-х годов прошлого века, пытались создать основанные на образцах языки, которые, хотя и были бы специфичными для определенных предметных областей, имели бы более высокий уровень абстракции, чем обычные языки программирования. Предполагалось, что человек, знакомый с таким языком, практически без усилий сможет создавать приложения в данной предметной области, компонуя подходящие образцы нужным способом. Эту программу реализовать так и не удалось, однако выявленные образцы, несомненно, являются одним из самых значимых средств передачи опыта проектирования сложных программных систем.

Образец (pattern) представляет собой шаблон решения типовой, достаточно часто встречающейся задачи в некотором контексте, т.е. при некоторых ограничениях на ожидаемые решения и определенном наборе требований к ним.

В качестве примера рассмотрим такую ситуацию. Мы разработали большую программу из многих модулей. Так сложилось, что почти все они опираются на некоторый выделенный модуль

и часто используют его операции. В какой-то момент, однако, разработчик этого модуля решил поменять названия операций в его интерфейсе и порядок следования параметров (может быть и так, что его разработчиком является другая организация, у которой этот модуль был приобретен, и такие изменения в нем появились в очередной версии, в которой исправлены многие серьезные ошибки). Изменить код других модулей системы достаточно тяжело, так как вызовы операций данного модуля используются во многих местах. А если придется работать с несколькими разными версиями — не менять же код каждый раз!

Другим примером такой ситуации является разработка набора тестов для некоторых операций. Хотелось бы, чтобы с помощью этого набора можно было бы тестировать любую возможную реализацию функций, выполняемых этими операциями. Если функции достаточно часто встречаются, например, совместно реализуют очередь, хранящую некоторые элементы, то такая возможность очень полезна. Но у каждого набора операций может быть свой интерфейс, переделывать все тесты под который слишком трудоемко.

Если можно представить набор требуемых операций как интерфейс некоторого класса в объектно-ориентированном языке программирования, достойно выйти из такой ситуации поможет образец проектирования *адаптер (adapter)*.

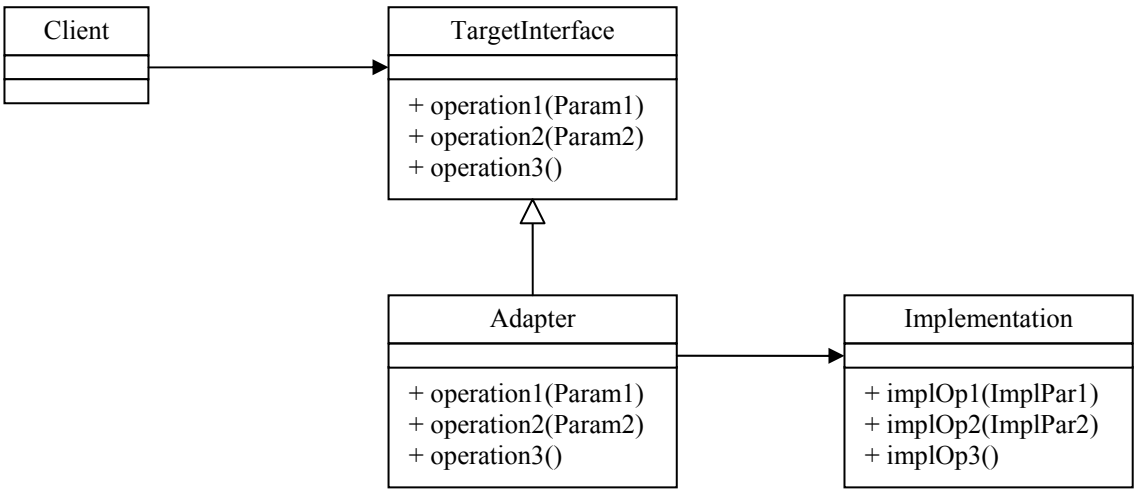

Рисунок 39. Структура классов-участников образца адаптер.

Предлагаемое решение состоит в следующем. Операции, которые необходимы для работы нашей системы, называемой *клиентом*, объединяются в некоторый класс или интерфейс (называемый *целевым*), и система пишется так, что она работает с объектом этого типа и его операциями. Получив *реализацию* тех же функций с отличающимися именами и типами параметров, мы определяем *адаптер* — класс-наследник целевого класса (или реализующий соответствующий интерфейс), в котором перегружаем нужные нам операции, выражая их через имеющуюся реализацию. При этом каждый раз объем дополнительной работы достаточно мал (если, конечно, полученная реализация действительно реализует нужные функции), а код клиента остается неизменным.

Образец проектирования нельзя выдумать или изобрести. Некоторый шаблон решения можно считать кандидатом в образцы проектирования, если он неоднократное применялся для решения одной и той же задачи на практике, если решения на его основе использовались в нескольких (как минимум, трех) случаях, в различных системах.

Образцы проектирования часто сильно связаны друг с другом в силу того, что они решают смежные задачи. Поэтому часто наборы связанных, поддерживающих друг друга образцов представляются вместе в виде систем *образцов (pattern system)* или *языка образцов (pattern language)*, в которых указаны возникающие между ними связи и описываются ситуации, в которых полезно совместное использование нескольких образцов.

По типу решаемых задач выделяют следующие разновидности образцов.

- ***Образцы анализа (analysis patterns)***.
  Они представляют собой типовые решения при моделировании сложных взаимоотношений между понятиями некоторой предметной области. Обычно они являются представлением этих понятий и отношений между ними с помощью набора классов и их связей, подходящего для любого объектно-ориентированного языка. Такие представления обладают важными атрибутами качественных модельных решений — способностью отображать понятным образом большое многообразие ситуаций, возникающих в реальной жизни, отсутствием необходимости вносить изменения в модель при небольших изменениях в требованиях к основанному на ней программному обеспечению и удобством внесения изменений, вызванных естественными изменениями в понимании моделируемых понятий. В частности, небольшое расширение данных, связанных с некоторым понятием, приводит к небольшим изменениям в структуре, чаще всего, лишь одного класса модели. Образцы анализа могут относиться к определенной предметной области, как следует из их определения, но могут также и с успехом быть использованы для моделирования понятий в разных предметных областях.
  В отличие от образцов проектирования и идиом (см. ниже), образцы анализа используются при концептуальном моделировании и не отражают прямо возможную реализацию такой модели в виде конкретного кода участвующих в ней классов. Например, поле `X` класса концептуальной модели в реализации может остаться полем, а может превратиться в пару методов `getX()` и `setX()` или в один метод `getX()` (т.е. в *свойство*, *property*, в терминах C# и JavaBeans).
- ***Архитектурные образцы*** или ***архитектурные стили (architectural styles, architectural patterns)***.
  Такие образцы представляют собой типовые способы организации системы в целом или крупных подсистем, задающие некоторые правила выделения компонентов и реализации взаимодействий между ними.
- ***Образцы проектирования (design patterns)*** в узком смысле.
  Они определяют типовые проектные решения для часто встречающихся задач среднего уровня, касающиеся структуры одной подсистемы или организации взаимодействия двух-трех компонентов.
- ***Идиомы (idioms, programming patterns)***.
  Идиомы являются специфическими для некоторого языка программирования способами организации элементов программного кода, позволяющими, опять же, решить некоторую часто встречающуюся задачу.
- ***Образцы организации (organizational patterns)*** и ***образцы процессов (process patterns)***.
  Образцы этого типа описывают успешные практики организации разработки ПО или другой сложной деятельности, позволяющие решать определенные задачи в рамках некоторого контекста, который включает ограничения на возможные решения.

Для описания образцов были выработаны определенные шаблоны. Далее мы будем использовать один из таких шаблонов для описания архитектурных стилей, образцов проектирования и идиом. Этот шаблон включает в себя следующие элементы.

- Название образца, а также другие имена, под которыми этот образец используется.
- Назначение — задачи, которые решаются с помощью данного образца. В этот же пункт включается описание контекста, в котором данный образец может быть использован.
- Действующие силы — ограничения, требования и идеи, под воздействием которых вырабатывается решение.
- Решение — основные идеи используемого решения. Включает следующие подпункты.

- Структура — структура компонентов, принимающих участие в данном образце, и связей между ними. В рамках образца компоненты принято именовать исходя из ролей, которые они в нем играют.
- Динамика — основные сценарии совместной работы компонентов образца.
- Реализация — возможные проблемы при реализации и способы их преодоления, примеры кода на различных языках (в данном курсе мы будем использовать для примеров только язык Java). Варианты и способы уточнения данного образца.
- Следствия применения образца — какими дополнительными свойствами, достоинствами и недостатками, обладают полученные на его основе решения.

- Известные примеры использования данного образца.
- Другие образцы, связанные с данным.

Далее в этой лекции рассматриваются некоторые из известных образцов в соответствии с приведенной классификацией. Другие образцы будут упоминаться в последующих лекциях при рассмотрении способов решения тех или иных задач, а также библиотек языков Java и C#.

Образцы анализа

Образец анализа является типовым решением по представлению набора понятий некоторой предметной области в виде набора классов и связей между ними. Основной источник описаний выделенных образцов анализа — это работы Мартина Фаулера (Martin Fowler) [2,3].

В качестве примера образцов анализа рассмотрим группу образцов, связанных с представлением в программной системе данных измерений и наблюдений.

Наиболее простым образцом этой группы является образец *величина (quantity)*. Результаты большинства измерений имеют количественное выражение, однако, если представлять их в виде атрибутов числовых типов (рост — 182, вес — 83), часть информации пропадает. Пока все пользователи системы и разработчики, вносящие в нее изменения, помнят, *в каких единицах* измеряются все хранимые величины, все в порядке, но стоит хоть одному ошибиться — и результаты могут быть весьма серьезны. Такого рода ошибка в 1998 году вывела из строя американский космический аппарат Mars Climate Orbiter, предназначавшийся для исследования климата Марса. Данные о текущих параметрах движения аппарата поступали на Землю, обрабатывались, и результирующие команды отправлялись обратно. При этом процедуры мониторинга и управления движением на самом аппарате воспринимали величину импульса как измеренную в Ньтонах на секунду, а программы обработки данных на Земле — как значение импульса в фунтах силы на секунду. В итоге это привело к выходу на гораздо более низкую, чем планировалось, орбиту, потере управления и гибели аппарата стоимостью около 130 миллионов долларов [4].

Поэтому более аккуратное решение — использовать для хранения данных числовых измерений объекты специального класса `Quantity`, в полях которого хранится не только значение величины, но и единица ее измерения. Кроме того, весьма полезно определить операции сложения, вычитания и сравнения таких величин.

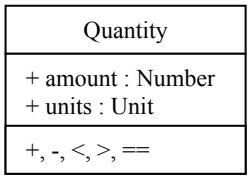

Рисунок 40. Класс для представления величин, имеющих разные единицы измерения.

Помимо измерений, использовать такое представление удобно и для сумм денег в финансовых системах. Аналогом единиц измерения в этом случае выступают различные валюты. От физических величин валюты отличаются изменяемым отношением, с помощью которого их

можно переводить одну в другую. Это отношение может зависеть от времени. Кроме того, существуют единицы измерения физических величин, которые преобразуются друг в друга более сложным, чем умножение на некоторое число, способом — например, градусы по Фаренгейту и по Цельсию.

Эти примеры могут быть охвачены образцом *преобразование*, который позволяет представлять в системе правила преобразования различных единиц измерения друг в друга. Для большинства преобразований достаточно величины отношения между единицами, быть может, зависящего от времени, поэтому стоит выделить класс для хранения этого отношения.

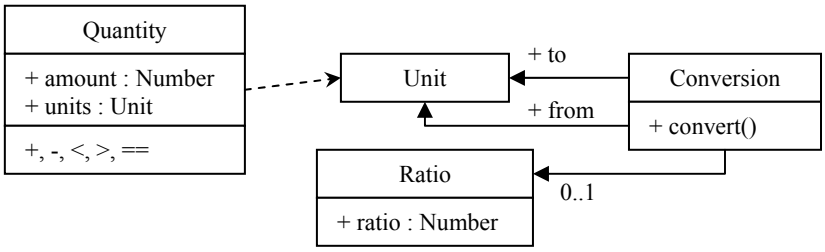

Рисунок 41. Представление возможных преобразований между единицами измерений.

Другой тип связи между различными единицами измерения — так называемые *составные единицы*, например Ньютон для измерения силы (1 Н = 1 кг*м/с^2). Разрешение подобного рода соотношений может быть реализовано, если определить два подкласса класса `Unit` — один для представления простых единиц, `PrimeUnit`, другой для представления составных, `CompoundUnit`, и определить две связи, сопоставляющие одной составной единице два мультимножества простых — те, что участвуют в ней в положительных степенях, и те, что участвуют в отрицательных.

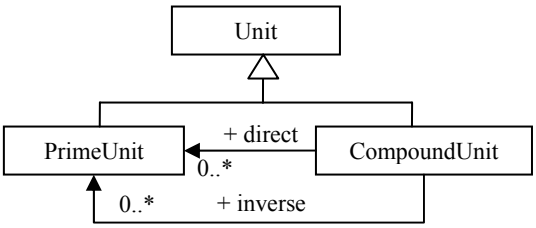

Рисунок 42. Представление составных единиц измерений.

В медицине, где хранение данных измерений имеет особое значение, измерения почти всегда связываются с пациентом, для которого они производились. К тому же, медицинских измерений, имеющих, например, значение длины, очень много. Для того, чтобы различать оба этих атрибута измерения — объект измерения и вид измерения (например, пациент Иванов Петр Сергеевич и окружность его талии), их нужно явно ввести в модель. Так возникает образец *измерение*. Этот образец становится полезным, если имеется очень много различных измерений для каждого объекта, группируемых в достаточно много видов измеряемых явлений.

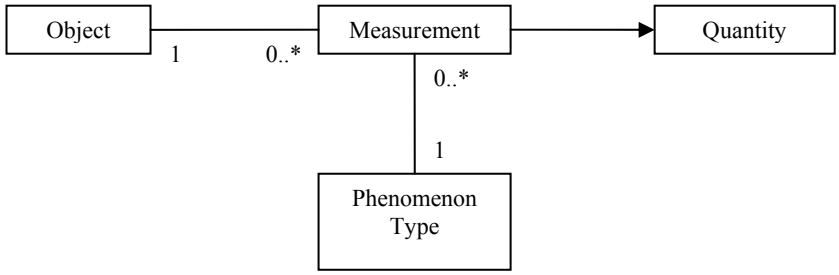

Рисунок 43. Набор классов для представления результатов измерений.

Бывает, однако, необходимо вести учет не только количественных измерений, но и качественных наблюдений, результат которых представляется не числом, а некоторым значением

перечислимого типа (группа крови II, ожог 3-й степени и пр.). При этом наблюдения очень похожи на измерения: относятся к некоторому объекту и определяют некоторое значение для какого-то вида наблюдений.

Для совместного представления результатов наблюдений и измерений можно использовать образец *наблюдение*, структура классов которого показана на Рис. 44. Требуется некоторая привычка, чтобы быстро разложить по этим классам какой-нибудь реальный пример. Например, группа крови — вид явлений, II — явление этого вида, наблюдение заключается в том, что у Петра Сергеевича Иванова была обнаружена именно такая группа крови. Эти усилия, однако, с лихвой окупаются огромным количеством фактов, которые без изменений можно уложить в эту схему.

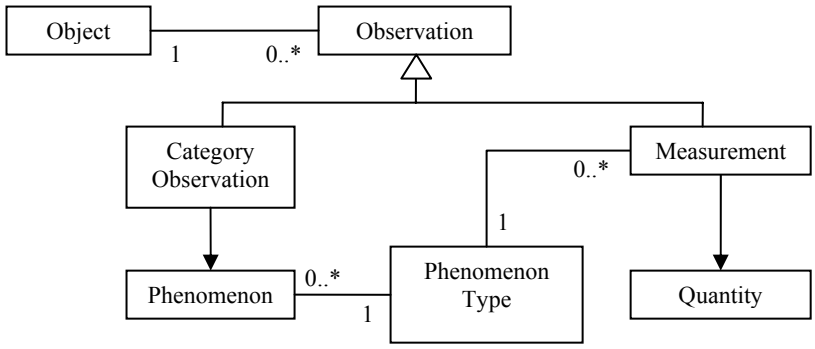

Рисунок 44. Набор классов для представления результатов как измерений, так и наблюдений.

Архитектурные стили

Архитектурный стиль определяет основные правила выделения компонентов и организации взаимодействия между ними в рамках системы или подсистемы в целом. Различные архитектурные стили подходят для решения различных задач в плане обеспечения нефункциональных требований — различных уровней производительности, удобства использования, переносимости и удобства сопровождения. Одну и ту же функциональность можно реализовать, используя разные стили.

Работа по выделению и классификации архитектурных стилей была проведена в середине 1990-х годов. Ее результаты представлены в работах [5,6]. Ниже приведена таблица некоторых архитектурных стилей, выделенных в этих работах.

Виды стилей и конкретные стили	**Контекст использования и основные решения**	**Примеры**
Конвейер обработки данных (data flow)	Система выдает четко определенные выходные данные в результате обработки четко определенных входных данных, при этом процесс обработки не зависит от времени, применяется многократно, одинаково к любым данным на входе. Обработка организуется в виде набора (не обязательно последовательности) отдельных компонентов-обработчиков, передающих свои результаты на вход другим обработчикам или на выход всей системы. Важными свойствами являются четко определенная структура данных и возможность интеграции с другими системами.	

	Пакетная обработка (batch sequential)	Один-единственный вывод производится на основе чтения некоторого одного набора данных на входе, промежуточные преобразования организуются в виде последовательности.	Сборка программной системы: компиляция, сборка системы, сборка документации, выполнение тестов.
	Каналы и фильтры (pipe-and-filter)	Нужно обеспечить преобразование непрерывных потоков данных. При этом преобразования инкрементальны и следующее может быть начато до окончания предыдущего. Имеется, возможно, несколько входов и несколько выходов. В дальнейшем возможно добавление дополнительных преобразований.	Утилиты UNIX
	Замкнутый цикл управления (closed-loop control)	Нужно обеспечить обработку постоянно поступающих событий в плохо предсказуемом окружении. Используется общий диспетчер событий, который классифицирует событие и отдает его на асинхронную обработку обработчику событий такого типа, после чего диспетчер снова готов воспринимать события.	Встроенные системы управления в автомобилях, авиации, спутниках. Обработка запросов на сильно загруженных Web-серверах. Обработка действий пользователя в GUI.
Вызов-возврат (call-return)		Порядок выполнения действий четко определен, отдельные компоненты не могут выполнять полезную работу, не получая обращения от других.	
	Процедурная декомпозиция	Данные неизменны, процедуры работы с ними могут немного меняться, могут возникать новые. Выделяется набор процедур, схема передачи управления между которыми представляет собой дерево с основной процедурой в его корне.	Основная схема построения программ для языков C, Pascal, Ada
	Абстрактные типы данных (abstract data types)	В системе много данных, структура которых может меняться. Важны возможности внесения изменений и интеграции с другими системами. Выделяется набор абстрактных типов данных, каждый из которых предоставляет набор операций для работы с данными такого типа. Внутреннее представление данных скрывается.	Библиотеки классов и компонентов
	Многоуровневая система (layers)	Имеется естественное расслоение задач системы на наборы задач, которые можно было бы решать последовательно — сначала задачи первого уровня, затем, используя полученные решения, — второго, и т.д. Важны переносимость и возможность многократного использования отдельных компонентов. Компоненты разделяются на несколько уровней таким образом, что компоненты данного уровня могут использовать для своей работы только соседей или компоненты предыдущего уровня.	Телекоммуникационные протоколы в модели OSI (7 уровней), реальные протоколы сетей передачи данных (обычно 5 уровней или меньше). Системы автоматизации предприятий (уровни интерфейса

		Могут быть более слабые ограничения, например, компонентам верхних уровней разрешено использовать компоненты всех нижележащих уровней.	пользователя-обработки запросов-хранения данных).
	Клиент-сервер	Решаемые задачи естественно распределяются между инициаторами и обработчиками запросов, возможно изменение внешнего представления данных и способов их обработки.	Основная модель бизнес-приложений: клиентские приложения, воспринимающие запросы пользователей и сервера, выполняющие эти запросы.
Интерактивные системы		Необходимость достаточно быстро реагировать на действия пользователя, изменчивость пользовательского интерфейса.	
	Данные–представление–обработка (model-view-controller, MVC)	Изменения во внешнем представлении достаточно вероятны, одна и та же информация представляется по-разному в нескольких местах, система должна быстро реагировать на изменения данных. Выделяется набор компонентов, ответственных за хранение данных, компоненты, ответственные за их представления для пользователей, и компоненты, воспринимающие команды, преобразующие данные и обновляющие их представления.	Наиболее часто используется при построении приложений с GUI. Document-View в MFC (Microsoft Foundation Classes) — документ в этой схеме объединяет роли данных и обработчика.
	Представление–абстракция–управление (presentation-abstraction-control)	Интерактивная система на основе агентов, имеющих собственные состояния и пользовательский интерфейс, возможно добавление новых агентов. Отличие от предыдущей схемы в том, что для каждого отдельного набора данных его модель, представление и управляющий компонент объединяются в агента, ответственного за всю работу именно с этим набором данных. Агенты взаимодействуют друг с другом только через четко определенную часть интерфейса управляющих компонентов.	
Системы на основе хранилища данных		Основные функции системы связаны с хранением, обработкой и представлением больших количеств данных.	
	Репозиторий (repository)	Порядок работы определяется только потоком внешних событий. Выделяется общее хранилище данных — репозиторий. Каждый обработчик запускается в ответ на соответствующее ему событие и как-то преобразует часть данных в репозитории.	Среды разработки и CASE-системы

	Классная доска (blackboard)	Способ решения задачи в целом неизвестен или слишком трудоемок, но известны методы, частично решающие задачу, композиция которых способна выдавать приемлемые результаты, возможно добавление новых потребителей данных или обработчиков. Отдельные обработчики запускаются, только если данные репозитория для их работы подготовлены. Подготовленность данных определяется с помощью некоторой системы шаблонов. Если можно запустить несколько обработчиков, используется система их приоритетов.	Системы распознавания текста

Таблица 7. Некоторые архитектурные стили.

Многие из представленных стилей носят достаточно общий характер и часто встречаются в разных системах. Кроме того, часто можно обнаружить, что в одной системе используются несколько архитектурных стилей — в одной части преобладает один, в другой — другой, или же один стиль используется для выделения крупных подсистем, а другой — для организации более мелких компонентов в подсистемах.

Более подробного рассмотрения заслуживают стили «Каналы и фильтры», «Многоуровневая система». Далее следуют их описания согласно [7].

Каналы и фильтры

Название. Каналы и фильтры (pipes and filters).

Назначение. Организация обработки потоков данных в том случае, когда процесс обработки распадается на несколько шагов. Эти шаги могут выполняться отдельными обработчиками, возможно, реализуемыми разными разработчиками или даже организациями. При этом нужно принимать во внимание следующие факторы.

Действующие силы.

- Должны быть возможны изменения в системе за счет добавления новых способов обработки и перекомбинации имеющихся обработчиков, иногда самими конечными пользователями.
- Небольшие шаги обработки проще переиспользовать в различных задачах.
- Не являющиеся соседними обработчики не имеют общих данных.
- Имеются различные источники входных данных — сетевые соединения, текстовые файлы, сообщения аппаратных датчиков, базы данных.
- Выходные данные могут быть востребованы в различных представлениях.
- Явное хранение промежуточных результатов может быть неэффективным, создаст множество временных файлов, может привести к ошибкам, если в его организацию сможет вмешаться пользователь.
- Возможно использование параллелизма для более эффективной обработки данных.

Решение. Каждая отдельная задача по обработке данных разбивается на несколько мелких шагов. Выходные данные одного шага являются входными для других. Каждый шаг реализуется специальным компонентом — *фильтром (filter)*. Фильтр потребляет и выдает данные инкрементально, небольшими порциями. Передача данных между фильтрами осуществляется по *каналам (pipes)*.

Структура. Основными ролями компонентов в рамках данного стиля являются фильтр и канал. Иногда выделяют специальные виды фильтров — *источник данных (data source)* и *потребитель данных (data sink)*, которые, соответственно, только выдают данные или

только их потребляют. Каждый поток обработки данных состоит из чередующихся фильтров и каналов, начинается источником данных и заканчивается их потребителем.

Фильтр получает на свой вход данные и обрабатывает их, дополняя их результатами обработки, удаляя какие-то части и трансформируя их в некоторое другое представление. Иногда фильтр сам требует входные данные и выдает выходные по их получении, иногда он, наоборот, может реагировать на события прихода данных на вход и требования данных на выходе. Фильтр обычно потребляет и выдает данные некоторыми порциями.

Канал обеспечивает передачу данных, их буферизацию и синхронизацию обработки их соседними фильтрами (например, если оба соседних фильтра активны, работают в параллельных процессах). Если никакой дополнительный буферизации и синхронизации не требуется, канал может представлять собой простую передачу данных в виде параметра или результата вызова операции.

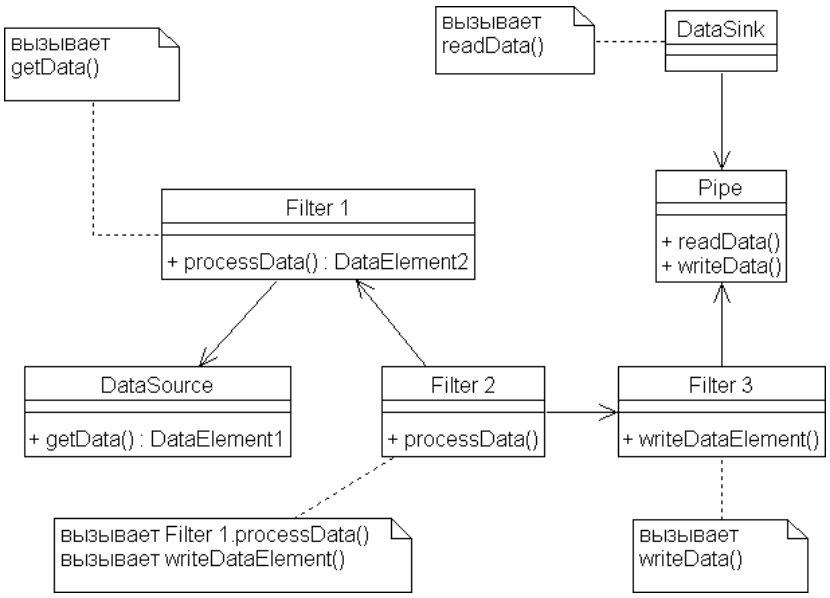

Рисунок 45. Пример структуры классов для образца каналы и фильтры.

На Рис. 45 показан пример диаграммы классов для данного образца, в котором 3 канала реализованы неявно — через вызовы операций и возвращение результатов, а один — явно. Из участвующих в этом примере фильтров источник и потребитель данных, а также Filter 1 запрашивают входные данные, Filter 3 сам передает их дальше, а Filter 2 и запрашивает, и передает данные самостоятельно.

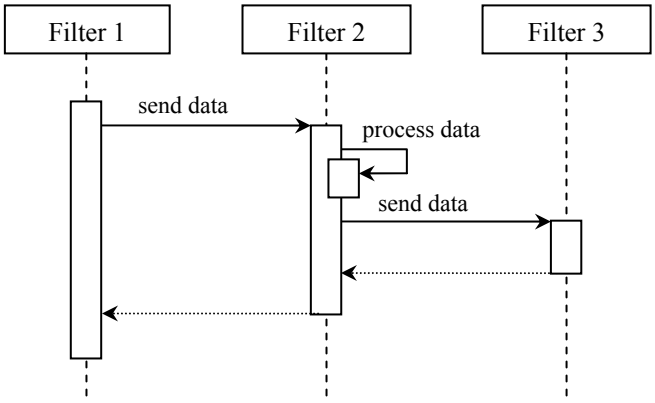

Рисунок 46. Сценарий работы проталкивающего фильтра.

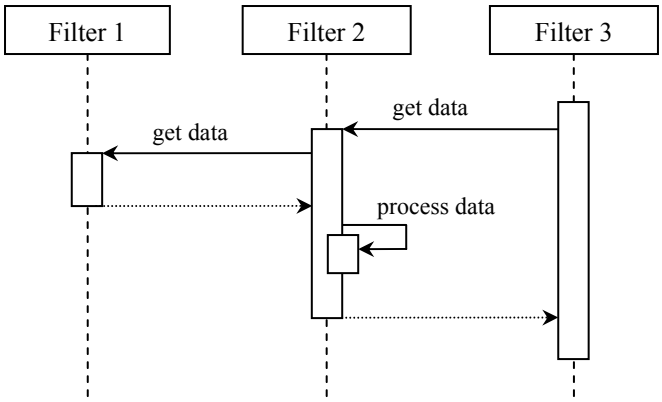

Рисунок 47. Сценарий работы вытягивающего фильтра.

Динамика. Возможны три различных сценария работы одного фильтра — проталкивание данных (push model, фильтр сам передает данные следующему компоненту, а получает их только в результате передачи предыдущего), вытягивание данных (pull model, фильтр требует данные у предыдущего компонента, следующий сам должен затребовать данные у него) и смешанный вариант. Часто реализуется только один вид передачи данных для всех фильтров в рамках системы. Кроме того, канал может буферизовать данные и синхронизовать взаимодействующие с ним фильтры. Сценарии работы системы в целом строятся в виде различных комбинаций вариантов работы отдельных фильтров.

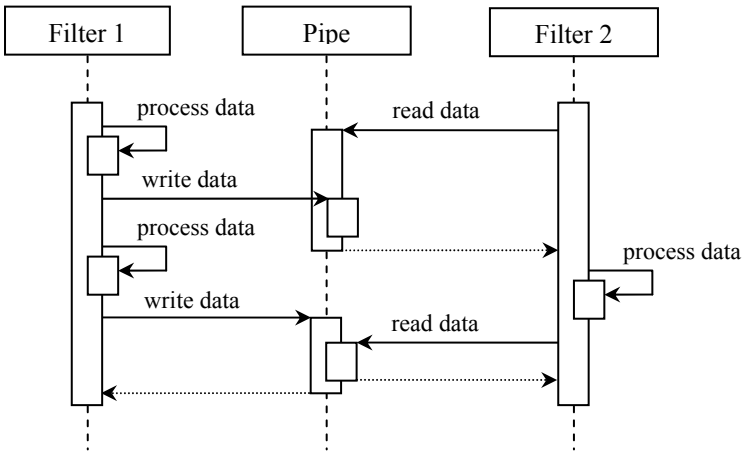

Рисунок 48. Сценарий работы буферизующего и синхронизующего канала.

Реализация. Основные шаги реализации следующие.

- Определить шаги обработки данных, необходимые для решения задач системы. Очередной шаг должен зависеть только от выходных данных предшествующего шага.
- Определить форматы данных при их передаче по каждому каналу.
- Определить способ реализации каждого канала, проталкивание или вытягивание данных, необходимость дополнительной буферизации и синхронизации.
- Спроектировать и реализовать необходимый набор фильтров. Реализовать каналы, если для их представления нужны отдельные компоненты.
- Спроектировать и реализовать обработку ошибок. Обработку ошибок при применении этого стиля достаточно тяжело организовать, поэтому ею часто пренебрегают. Однако, требуется, как минимум, адекватная диагностика случающихся на разных этапах ошибок.
  Могут быть выделены специальные каналы для передачи сообщений об ошибках.

При возникновении ошибок ввода соответствующий фильтр может игнорировать дальнейшие входные данные до получения определенного разделителя, гарантирующего, что после него идут данные, не связанные с предыдущими.

- Сконфигурировать необходимый конвейер обработки данных, собрав вместе нужные фильтры и соединяющие их каналы.

Следствия применения образца.

Достоинства.

- Промежуточные данные могут не храниться в файлах, но могут и храниться, если это необходимо для каких-то дополнительных целей.
- Фильтры можно легко заменять, переиспользовать, менять местами, переставлять и комбинировать, реализуя множество функций на основе одних и тех же компонентов.
- Конвейерные системы обработки данных могут быть разработаны очень быстро, если имеется богатый набор фильтров.
- Активные фильтры могут работать параллельно, давая в результате более эффективное решение на многопроцессорных системах.

Недостатки.

- Управление обработкой с помощью большого общего состояния, которое иногда необходимо, не может быть эффективно реализовано с помощью этого стиля.
- Часто параллельная обработка не приносит никакого повышения производительности, поскольку передача данных между фильтрами может быть достаточно дорогой, фильтры могут требовать всех входных данных, прежде чем выдадут хоть что-то, и их синхронизация с помощью каналов может приводить к значительным простоям.
- Часто фильтры большее время тратят на преобразование формата поступающих входных данных, чем на их обработку. Использование одного формата, например, текстового, также зачастую снижает эффективность их использования.
- Обработка ошибок в рамках данного стиля очень сложна. В том случае, если разрабатываемая система должна быть очень надежной, а возвращение к самому началу работы в случае обнаружения ошибки, так же как ее игнорирование не являются допустимыми сценариями, использовать этот стиль не стоит.

Примеры. Наиболее известный пример использования данного образца — система утилит UNIX [8], пополненная возможностями оболочки (shell) по организации каналов между процессами. Большинство утилит могут играть роль фильтров при обработке текстовых данных, а каналы строятся при помощи соединения стандартного ввода одной программы со стандартным выводом другой.

Другим примером может служить часто используемая архитектура компилятора как последовательности фильтров, обрабатывающих входную программу — лексического анализатора (лексера), синтаксического анализатора (парсера), семантического анализатора, набора оптимизаторов и генератора результирующего кода. Таким способом можно достаточно быстро построить прототипный компилятор для несложного языка. Более производительные компиляторы, нацеленные на промышленное использование, строятся по более сложной схеме, в частности, используя элементы стиля «Репозиторий».

Многоуровневая система

Название. Многоуровневая система (layers).

Назначение. Реализация больших систем, которые имеют большое количество разноплановых элементов, использующих друг друга. Некоторые аспекты работы таких

систем могут включать в себя много операций, выполняемых разными компонентами на разных уровнях (т.е. одна задача решается за счет последовательных обращений между элементами разных уровней, другая — тоже, но участвующие в решении этих задач элементы могут быть различны). При этом нужно принимать во внимание следующие факторы.

Действующие силы.

- Изменения в требованиях к решению одной из задач не должны приводит к изменениям в коде многочисленных компонентов, желательно, чтобы они сводились к изменениям внутри одного компонента. То же касается и изменений платформы, на которой работает система.
- Интерфейсы между компонентами должны быть стабильными или даже соответствовать имеющимся стандартам.
- Части системы должны быть заменяемы. Компоненты должны быть заменяемы другими, если те реализуют такие же интерфейсы. В идеале может даже потребоваться в ходе работы переключиться на другую реализацию, даже если при начале работы системы она не была доступна.
- Низкоуровневые компоненты должны позволять разрабатывать другие системы быстрее.
- Компоненты с похожими областями ответственности должны быть сгруппированы для повышения понятности системы и удобства внесения в нее изменений.
- Нет возможности выделить компоненты некоторого стандартного размера: одни из них решают достаточно сложные задачи, другие — совсем простые.
- Сложные компоненты нуждаются в дальнейшей декомпозиции.
- Использование большого числа компонентов может отрицательно сказаться на производительности, поскольку данным придется часто преодолевать границы между компонентами.
- Разработка системы должна быть эффективно поделена между отдельными разработчиками. При этом интерфейсы и зоны ответственности компонентов, передаваемых разным разработчикам, должны быть очень четко определены.

Решение. Выделяется некоторый набор уровней, каждый из которых отвечает за решение своих собственных подзадач. Для этого он использует интерфейс, предоставляемый предыдущим уровнем, предоставляя, в свою очередь, некоторый интерфейс для следующего уровня.

Каждый отдельный уровень может быть в дальнейшем декомпозирован на более мелкие компоненты.

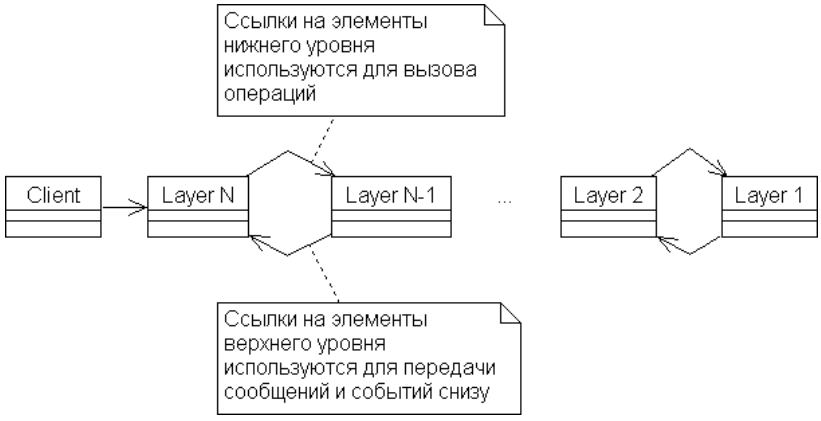

Рисунок 49. Пример структуры многоуровневой системы. Классы для уровней условные, они обычно декомпозируются на наборы более мелких компонентов.

Структура. Основными компонентами являются *уровни*. Иногда выделяют *клиентов*, использующих интерфейс самого верхнего уровня. Каждый уровень предоставляет интерфейс для решения определенного множества задач. Сам он решает их, опираясь на интерфейс предшествующего уровня.

На каждом уровне может находиться много более мелких компонентов.

Динамика. Сценарии работы системы могут быть получены компоновкой следующих четырех. Часто в виде многих уровней реализуются коммуникационные системы, две такие системы могут взаимодействовать через самый нижний уровень — при этом пара симметричных сценариев (по подъему-спуску обращений) выполняется в рамках одного общего сценария на разных машинах.

- Обращение клиента к верхнему уровню инициирует цепочку обращений с верхнего уровня до самого нижнего.
- Событие на нижнем уровне (например, приход сообщения по сети или нажатие на кнопку мыши) инициирует цепочку обращений, идущую снизу вверх, вплоть до некоторого события самого верхнего уровня, видимого клиентам.
- Обращение клиента к верхнему уровню приводит к цепочке вызовов, которая, однако, не доходит до самого низа. Такая ситуация реализуется, если, например, один из уровней кэширует ответы на запросы и может выдать ответ на ранее уже подававшийся запрос без обращения к более низким уровням.
- То же самое может произойти и с событием, которое передается с самого нижнего уровня. Дойдя до некоторого уровня, оно может поглотиться им (с изменением состояния каких-то компонентов), потому что не соответствует никакому событию на более высоких уровнях. Например, нажатие клавиши Capslock не приводит само по себе ни к каким реакциям программы, но изменяет значение нажимаемых после этого клавиш, меняя их регистр на противоположный.

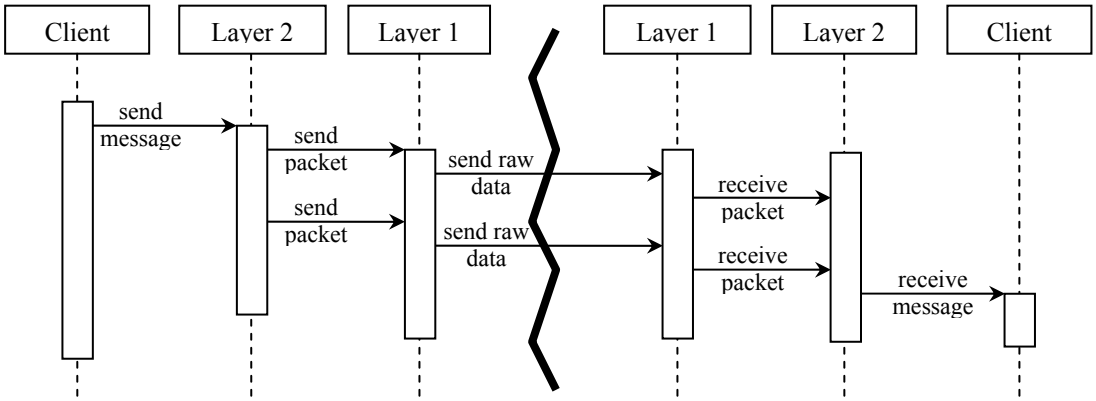

Рисунок 50. Составной сценарий пересылки сообщения по сети.

Реализация. Основные шаги реализации следующие.

- Определить критерии группировки задач по уровням. Это критически важный шаг. Неправильное распределение задач, скорее всего, приведет к необходимости перепроектировать систему.
- Определить количество уровней, которые будут реализованы, и их имена. Часто приходится объединять концептуально различные задачи, чтобы добиться большей эффективности системы. С другой стороны, произвольное смешение задач на уровне ведет к непонятной архитектуре и к системе, очень неудобной для сопровождения. При возможности поместить некоторую задачу на несколько уровней, стоит размещать ее на самом высоком уровне из тех, где она может быть решена с достаточной производительностью.

- Определить интерфейсы, предоставляемые нижними уровнями верхним. Здесь нужно помнить, что с помощью несколько б*о*льшего, чем минимально необходимый, набора интерфейсных операций нижнего уровня можно добиться значительного повышения производительности системы в целом.
- Определить компоненты и их взаимодействие в рамках каждого отдельного уровня.
- Определить способы взаимодействия соседних уровней. Можно использовать проталкивание, вытягивание данных или комбинацию этих подходов.
- Отделить соседние уровни. В идеале нижние уровни не должны знать ничего о верхних, каждый уровень должен знать только о непосредственно предшествующем ему. Для этого передачу данных с нижнего уровня можно организовать в виде обратных вызовов (callbacks) — указатель на функцию, которую нужно вызвать для передачи сообщения наверх, верхний уровень может передавать в качестве параметра при предшествующих запросах.
- Спроектировать и реализовать обработку ошибок. Ошибки лучше обрабатывать на самом нижнем уровне, который в состоянии их заметить.

Следствия применения образца.

Достоинства.

- Возможность легко заменять и переиспользовать компоненты одного уровня, не оказывая влияния на остальные уровни. Возможность отлаживать и тестировать уровни по отдельности.
- Поддержка стандартов. Многоуровневость системы делает возможной поддержку стандартных интерфейсов, таких как POSIX.

Недостатки.

- Изменение функциональности одного уровня может привести к каскадному изменению всех уровней. Существенный рост производительности нижнего уровня и требование обеспечить соответствующий рост производительности на более высоких уровнях также могут привести к переопределению всех интерфейсов.
- Падение производительности из-за необходимости все вызовы и данные проводить через все уровни.
- Часто уровни дублируют работу друг друга, например, при обработке ошибок, поскольку они разрабатываются независимо и не имеют информации о деталях реализации друг друга.
- Большое количество уровней может привести к существенному повышению сложности системы и падению ее производительности. С другой стороны, слишком малое число уровней (например, два) часто не позволяет обеспечить необходимую гибкость и переносимость.

Примеры. Наиболее известный пример использования данного образца — стандартная модель протоколов связи открытых систем (Open System Interconnection, OSI) [9]. Она состоит из 7-ми уровней.

- Самый нижний уровень — *физический*. Он отвечает за передачу отдельных битов по каналам связи. Основные его задачи — гарантировать правильное определение нуля и единицы разными системами, определить временные характеристики передачи (за какое время передается один бит), обеспечить передачу в одном или двух направлениях, и т.п.
- Второй уровень — *канальный* или *уровень передачи данных*. Его задача — предоставить верхним уровням такие сервисы, чтобы для них передача данных выглядела бы как посылка и прием потока байт без потерь и без перегрузок.

- Третий уровень — *сетевой*. Его задача — обеспечить прозрачную связь между компьютерами, не соединенными непосредственно, а также обеспечивать нормальную работу больших сетей, по которым одновременно путешествует очень много пакетов данных.
- Четвертый уровень — *транспортный*. Он обеспечивает надежную передачу данных верхних уровней словно по некоторой трубе — пакеты приходят обязательно в той же последовательности, в которой они были отправлены. Заметим, что канальный уровень решает такую же задачу, но только для непосредственно связывающихся друг с другом машин.
- Пятый, *сеансовый* уровень предоставляет возможность устанавливать сеансы связи (или сессии), содержащие некоторый набор передаваемых туда и обратно сообщений, и управлять ими.
- Шестой, *уровень представления*, определяет форматы передаваемых данных. Например, именно здесь определяется, что целое число будет представляться 4-мя байтами, причем старшие биты числа идут раньше младших, первый бит интерпретируется как знак, а отрицательные числа представляются в дополнительной системе (т.е. 0x0000000f обозначает 15, а 0x80000000f — $-2147483633 = -(2^{31}-15)$).
- Наконец, седьмой уровень — *прикладной* — содержит набор протоколов, которыми непосредственно пользуются программы и с которыми работают пользователи — HTTP, FTP, SMTP, POP3 и пр.

Модель OSI оказалась все же слишком сложна для использования на практике. Сейчас наиболее широко применяемые наборы протоколов строятся по урезанной схеме OSI — в ней отсутствуют пятый и шестой уровни, прикладные протоколы пользуются непосредственно службами протоколов транспортного уровня.

Другой пример многоуровневой архитектуры — архитектура современных информационных систем или систем автоматизации бизнеса. Она включает следующие уровни [3].

- Интерфейс взаимодействия с внешней средой.
  Чаще всего этот уровень рассматривается как интерфейс пользователя. В его рамках определяется представление данных для передачи другим системам или пользователям, набор экранов, форм и отчетов, с которыми имеют дело пользователи.
- Бизнес-логика. На этом уровне реализуются основные правила функционирования данного бизнеса, данной организации.
- Предметная область. Данный уровень содержит концептуальную схему данных, с которыми имеет дело организация. Эти же данные могут использоваться и другими организациями в своей работе.
- Уровень управления ресурсами.
  На нем находятся все ресурсы, которыми пользуется система, в том числе другие системы. Очень часто используемые ресурсы сводятся к набору баз данных, необходимых для работы организации. На этом уровне определяется структура используемых ресурсов и способы управления ими, в частности, конкретное размещение данных по таблицам реляционной базы данных или классам объектной базы данных и соответствующий набор индексов. Чаще всего схемы баз данных оптимизируются под конкретный набор запросов, и поэтому их структура несколько отличается от концептуальной схемы данных, находящейся на предыдущем уровне.

Часто два средних уровня объединяются в один — уровень функционирования приложений, что дает в результате широко используемую *трехзвенную архитектуру* информационных систем.

Литература к Лекции 7

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер-ДМК, 2001.
[2] M. Fowler. Analysis Patterns: Reusable Object Models. Addison-Wesley, 1997.
[3] М. Фаулер и др. Архитектура корпоративных программных приложений. М.: Вильямс, 2004.
[4] Mars Climate Orbiter Mishap Investigation Board Phase I Report.
Доступен по адресу ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf
[5] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Englewood Cliffs, NJ, 1996.
[6] M. Shaw and P. Clementz. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. Proceeding of COMPSAC, Washington, D.C., August 1997.
[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. A System of Patterns. Wiley, 2002.
[8] Э. Таненбаум. Современные операционные системы. 2-е издание. СПб.: Питер, 2002.
[9] Э. Таненбаум. Компьютерные сети. 4-е издание. СПб.: Питер, 2003.
[10] Л. Басс, П. Клементс, Р. Кацман. Архитектура программного обеспечения на практике. СПб.: Питер, 2006.
[11] Э. Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2004.