# Лекция 3 Архитектура программного обеспечения. Образцы проектирования.

**Анализ области решений**

Допустим, мы разобрались в предметной области, поняли, что требуется от будущей программной системы, даже зафиксировали настолько полно, насколько смогли, требования и пожелания всех заинтересованных лиц ко всем аспектам качества программы. Что делать дальше?

На этом этапе (а точнее, гораздо раньше, обычно еще в ходе анализа предметной области) исследуются возможные способы решения тех задач, которые поставлены в требованиях. Не всегда бывает нужно специальное изучение этого вопроса — чаще всего имеющийся опыт разработчиков сразу подсказывает, как можно решать поставленные задачи. Однако иногда, все-таки, возникает потребность сначала понять, как это можно сделать и, вообще, возможно ли это сделать и при каких ограничениях.

Таким образом, явно или неявно, проводится ***анализ области решений***. Целью этой деятельности является понимание, можно ли вообще решить стоящие перед разрабатываемой системой задачи, при каких условиях и ограничениях это можно сделать, как они решаются, если решение есть, а если нет — нельзя ли придумать способ его найти или получить хотя бы приблизительное решение, и т.п. Обычно задача хорошо исследована в рамках какой-либо области человеческих знаний, но иногда приходится потратить некоторые усилия на выработку собственных решений. Кроме того, решений обычно несколько и они различаются по некоторым характеристикам, способным впоследствии сыграть важную роль в процессе развития и эксплуатации созданной на их основе программы. Поэтому важно взвесить их плюсы и минусы и определить, какие из них наиболее подходят в рамках данного проекта, или решить, что все они должны использоваться для обеспечения большей гибкости ПО.

Когда определены принципиальные способы решения всех поставленных задач (быть может, в каких-то ограничениях), основной проблемой становится способ организации программной системы, который позволил бы реализовать все эти решения и при этом удовлетворить требованиям, касающимся нефункциональных аспектов разрабатываемой программы. Искомый способ организации ПО в виде системы взаимодействующих компонентов называют *архитектурой,* а процесс ее создания — *проектированием архитектуры ПО.*

**Архитектура программного обеспечения**

Под *архитектурой ПО* понимают набор внутренних структур ПО, которые видны с различных точек зрения и состоят из компонентов, их связей и возможных взаимодействий между компонентами, а также доступных извне свойств этих компонентов.

Под *компонентом* в этом определении имеется в виду достаточно произвольный структурный элемент ПО, который можно выделить, определив интерфейс взаимодействия между этим компонентом и всем, что его окружает. Обычно при разработке ПО термин «компонент» имеет несколько другой, более узкий смысл — это единица развертывания, самая маленькая часть системы, которую можно включить или не включить в ее состав. Такой

компонент также имеет определенный интерфейс и удовлетворяет некоторому набору правил, называемому компонентной моделью. Там, где возможны недоразумения, будет указано, в каком смысле употребляется этот термин. В этой лекции мы будем использовать преимущественно широкое понимание этого термина, а в далее, при необходимости - узкое.

В определении архитектуры упоминается набор структур, а не одна структура. Это означает, что в качестве различных аспектов архитектуры, различных взглядов на нее выделяются различные структуры, соответствующие разным аспектам взаимодействия компонентов. Примеры таких аспектов — описание типов компонентов и типов статических связей между ними при помощи диаграмм классов, описание композиции компонентов при помощи структур ссылающихся друг на друга объектов, описание поведения компонентов при помощи моделирования их как набора взаимодействующих, передающих друг другу некоторые события, конечных автоматов.

Архитектура программной системы похожа на набор карт некоторой территории. Карты имеют разные масштабы, на них показаны разные элементы (административно-политическое деление, рельеф и тип местности — лес, степь, пустыня, болота и пр., экономическая деятельность и связи), но они объединяются тем, что все представленные на них сведения соотносятся с географическим положением. Точно так же архитектура ПО представляет собой набор структур или представлений, имеющих различные уровни абстракции и показывающих разные аспекты (структуру классов ПО, структуру развертывания, т.е. привязки компонентов ПО к физическим машинам, возможные сценарии взаимодействий компонентов и пр.), объединяемых сопоставлением всех представленных данных со структурными элементами ПО. При этом уровень абстракции данного представления является аналогом масштаба географической карты.

Рассмотрим в качестве примера программное обеспечение авиасимулятора для командной тренировки пилотов. Задачей такой системы в целом является контроль и выработка необходимых для безопасного управления самолетом навыков у команд летчиков. Кроме того, отрабатываются навыки поведения в особых ситуациях, связанных с авариями, частичной потерей управления самолетом, тяжелыми условиями полета, и т.д.

Симулятор должен:

- Моделировать определенные условия полета и создавать некоторые события, к которым относятся следующие:
    - Скоростной и высотный режим полета, запас горючего, их изменения со временем.
    - Модель самолета и ее характеристики по управляемости, возможным режимам полета и скорости реакции на различные команды.
    - Погода за бортом и ее изменения со временем.
    - Рельеф и другие особенности местности в текущий момент, их изменения со временем.
    - Исходный и конечный пункты полета, расстояние и основные характеристики рельефа между ними.

- Исправность или неисправность элементов системы контроля полета и управления самолетом, показатели системы мониторинга и их изменение со временем.
- Наличие пролетающих вблизи курса самолета других самолетов, их геометрические и скоростные характеристики.
- Чрезвычайные ситуации, например, террористы на борту, нарушение герметичности корпуса, внезапные заболевания и «смерть» отдельных членов экипажа.

При этом вся совокупность условий должна быть непротиворечивой, выглядеть и развиваться подобно реальным событиям. Некоторые условия, например, погода, должны изменяться достаточно медленно, другие события — происходить внезапно и приводить к связанным с ними последствиям (нарушение герметичности корпуса может сопровождаться поломками каких-то элементов системы мониторинга или «смертью» одного из пилотов). Если приборы показывают наличие грозы по курсу и они исправны, через некоторое время летчики должны увидеть грозу за бортом и она может начать оказывать влияние на условия полета.

– Принимать команды, подаваемые пилотами, и корректировать демонстрируемые характеристики полета и работы системы управления самолетом в зависимости от этих команд, симулируемой модели самолета и исправности системы управления. Например, при повороте на некоторый угол вправо, показываемый пилотам «вид из кабины» должен переместиться на соответствующий угол влево со скоростью, соответствующей скорости реакции симулируемой модели самолета и исправности задействованных элементов системы управления.

Понятно, что одним из элементов симулятора служит система визуализации обстановки за бортом — она показывает пилотам «вид за окнами». Пилоты в ходе полета ориентируются по показателям огромного количества датчиков, представленных на приборной панели самолета. Вся их работа тоже должна симулироваться. Наличие и характеристики работы таких датчиков могут зависеть от симулируемой модели, но их расположение, форма и цвет служат слишком важными элементами выработки навыков управления самолетом, поэтому требуется поддерживать эти характеристики близкими к реальным. Представлять их только в виде изображений на некоторой панели неудобно, поскольку они должны располагаться и выглядеть максимально похоже на реальные прототипы. Значит, симулировать можно только небольшое семейство самолетов с практически одним и тем же наборов приборов на приборной панели.

Кроме того, все команды пилотов должны восприниматься соответствующими компонентами симулятора и встраиваться в моделируемые условия. В симулятор должен быть включен генератор событий, зависящий от текущей ситуации, а также интерфейс мониторинга и управления, с помощью которого внешний оператор мог бы создавать определенные события во время симуляции полета наблюдать за всем, что происходит. Все события и действия пилотов должны протоколироваться с помощью камер и микрофонов для дальнейшего разбора полетов.

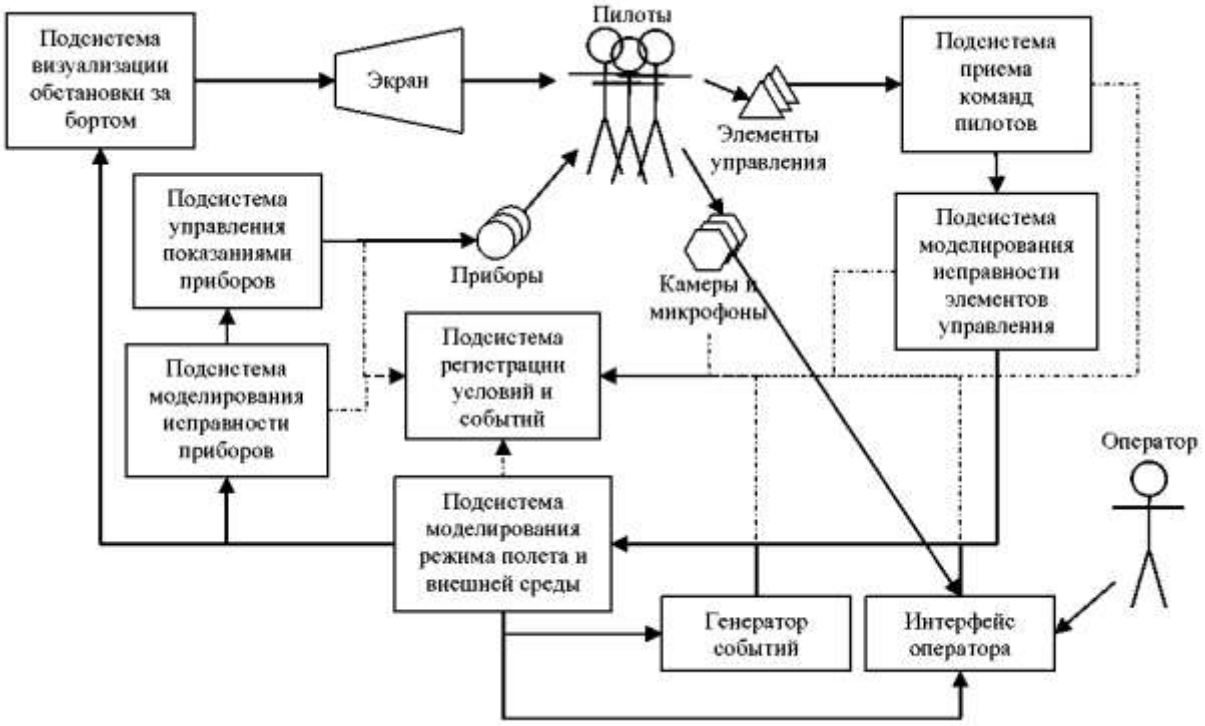


Рисунок 27**.** Примерная архитектура авиасимулятора**.**

Рис. 27 показывает набросок архитектуры такого авиасимулятора. Каждый из указанных компонентов решает свои задачи, которые необходимы для работы всей системы. В совокупности они решают все задачи системы в целом. Стрелками показаны потоки данных и управления между компонентами. Пунктирные стрелки изображают потоки данных, передаваемых для протоколирования.

Архитектура определяет большинство характеристик качества ПО в целом. Архитектура служит также основным средством общения между разработчиками, а также между разработчиками и всеми остальными лицами, заинтересованными в данном ПО. Выбор архитектуры задает способ реализации требований на высоком уровне абстракции. Именно архитектура почти полностью определяет такие характеристики ПО как надежность, переносимость и удобство сопровождения. Она также значительно влияет на удобство использования и эффективность ПО, которые, однако, сильно зависят и от реализации отдельных компонентов. Значительно меньше влияние архитектуры на функциональность — обычно заданную функциональность можно реализовать, использовав совершенно различные архитектуры.

Поэтому выбор между той или иной архитектурой определяется в большей степени именно нефункциональными требованиями и необходимыми свойствами ПО с точки зрения удобства сопровождения и переносимости. При этом для построения хорошей архитектуры надо учитывать возможные противоречия между требованиями к различным характеристикам и уметь выбирать компромиссные решения, дающие приемлемые значения по всем показателям.

Так, для повышения эффективности в общем случае выгоднее использовать монолитные архитектуры, в которых выделено небольшое число компонентов (в пределе — единственный компонент). Этим обеспечивается экономия как памяти, поскольку каждый компонент обычно

имеет свои данные, а здесь число компонентов минимально, так и времени работы, поскольку возможность оптимизировать работу алгоритмов обработки данных имеется также только в рамках одного компонента.

С другой стороны, для повышения удобства сопровождения, наоборот, лучше разбить систему на большое число отдельных маленьких компонентов, с тем чтобы каждый из них решал свою небольшую, но четко определенную часть общей задачи. При этом, если возникают изменения в требованиях или проекте, их обычно можно свести к изменению в постановке одной, реже двух или трех таких подзадач и, соответственно, изменять только отвечающие за решение этих подзадач компоненты.

С третьей стороны, для повышения надежности лучше использовать либо небольшой набор простых компонентов, либо дублирование функций, т.е. сделать несколько компонентов ответственными за решение одной подзадачи. Заметим, однако, что ошибки в ПО чаще всего носят неслучайный характер. Они повторяемы, в отличие от аппаратного обеспечения, где ошибки связаны часто со случайными изменениями характеристик среды и могут быть преодолены простым дублированием компонентов без изменения их внутренней реализации. Поэтому при таком обеспечении надежности надо использовать достаточно сильно отличающиеся способы решения одной и той же задачи в разных компонентах.

Другим примером противоречивых требований служат характеристики удобства использования и защищенности. Чем сильнее защищена система, тем больше проверок, процедур идентификации и пр. нужно проходить пользователям. Соответственно, тем менее удобна для них работа с такой системой. При разработке реальных систем приходится искать некоторый разумный компромисс, чтобы сделать систему достаточно защищенной и способной поставить ощутимую преграду для несанкционированного доступа к ее данным и, в то же время, не отпугнуть пользователей сложностью работы с ней.

Список стандартов, регламентирующих описание архитектуры, которое является основной составляющей проектной документации на ПО, выглядит так.

- **IEEE 1016-1998 Recommended Practice for Software Design Descriptions** (рекомендуемые методы описаний проектных решений для ПО).
- **IEEE 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems** (рекомендуемые методы описания архитектуры программных систем).

Основное содержание этого стандарта сводится к определению набора понятий, связанных с архитектурой программной системы.

Это, прежде всего, само понятие архитектуры как набора основополагающих принципов организации системы, воплощенных в наборе ее компонентов, связях их друг с другом и между ними и окружением системы, а также принципов проектирования и развития системы.

Это определение, в отличие от данного в начале этой лекции, делает акцент не на наборе структур в основе архитектуры, а на принципах ее построения.

Стандарт IEEE 1471 определяет также *представление архитектуры* ***(architectural***

***description)*** как согласованный набор документов, описывающий архитектуру с точки зрения определенной группы заинтересованных лиц с помощью набора моделей. Архитектура может иметь несколько представлений, отражающих интересы различных групп заинтересованных лиц.

Стандарт рекомендует для каждого представления фиксировать отраженные в нем взгляды и интересы, роли лиц, которые заинтересованы в таком взгляде на систему, причины, обуславливающие необходимость такого рассмотрения системы, несоответствия между элементами одного представления или между различными представлениями, а также различную служебную информацию об источниках информации, датах создания документов и пр.

Стандарт IEEE 1471 отмечает необходимость использования архитектуры системы для решения таких задач, как следующие.

- Анализ альтернативных проектов системы.
- Планирование перепроектирования системы, внесения изменений в ее организацию.
- Общение по поводу системы между различными организациями, вовлеченными в ее разработку, эксплуатацию, сопровождение, приобретающими систему или продающими ее.
- Выработка критериев приемки системы при ее сдаче в эксплуатацию.
- Разработка документации по ее использованию и сопровождению, включая обучающие и маркетинговые материалы.
- Проектирование и разработка отдельных элементов системы.
- Сопровождение, эксплуатация, управление конфигурациями и внесение изменений и поправок.
- Планирование бюджета и использования других ресурсов в проектах, связанных с разработкой, сопровождением или эксплуатацией системы.
- Проведение обзоров, анализ и оценка качества системы.

**Разработка и оценка архитектуры на основе сценариев**

При проектировании архитектуры системы на основе требований, зафиксированных в виде вариантов использования, первые возможные шаги состоят в следующем.

1. Выделение компонентов
    a. Выбирается набор «основных» сценариев использования — наиболее существенных и выполняемых чаще других.
    b. Исходя из опыта проектировщиков, выбранного архитектурного стиля и требований к переносимости и удобству сопровождения системы, определяются компоненты, отвечающие за определенные действия в рамках этих сценариев, т.е. за решение определенных подзадач.
    c. Каждый сценарий использования системы представляется в виде последовательности обмена сообщениями между полученными компонентами.
    d. При возникновении дополнительных хорошо выделенных подзадач добавляются новые компоненты, и сценарии уточняются.
2. Определение интерфейсов компонентов

a. Для каждого компонента в результате выделяется его интерфейс — набор сообщений, которые он принимает от других компонентов и посылает им.

b. Рассматриваются «неосновные» сценарии, которые так же разбиваются на последовательности обмена сообщениями с использованием, по возможности, уже определенных интерфейсов.

c. Если интерфейсы недостаточны, они расширяются.

d. Если интерфейс компонента слишком велик, или компонент отвечает за слишком многое, он разбивается на более мелкие.

3. Уточнение набора компонентов

a. Там, где это необходимо в силу требований эффективности или удобства сопровождения, несколько компонентов могут быть объединены в один.

b. Там, где это необходимо для удобства сопровождения или надежности, один компонент может быть разделен на несколько.

4. Достижение нужных свойств.

Все это делается до тех пор, пока не выполнятся следующие условия:

a. Все сценарии использования реализуются в виде последовательностей обмена сообщениями между компонентами в рамках их интерфейсов.

b. Набор компонентов достаточен для обеспечения всей нужной функциональности, удобен для сопровождения или портирования на другие платформы и не вызывает заметных проблем производительности.

c. Каждый компонент имеет небольшой и четко очерченный круг решаемых задач и строго определенный, сбалансированный по размеру интерфейс.

На основе возможных *сценариев использования или модификации* системы возможен также анализ характеристик архитектуры и оценка ее пригодности для поставленных задач или сравнительный анализ нескольких архитектур. Это так называемый *метод анализа архитектуры ПО* (Software Architecture Analysis Method, SAAM). Основные его шаги следующие:

1. Определить набор сценариев действий пользователей или внешних систем, использующих некоторые возможности, которые могут уже планироваться для реализации в системе или быть новыми. Сценарии должны быть значимы для конкретных заинтересованных лиц, будь то пользователь, разработчик, ответственный за сопровождение, представитель контролирующей организации и пр. Чем полнее набор сценариев, тем выше будет качество анализа. Можно также оценить частоту появления и важность сценариев, возможный ущерб от невозможности их выполнить.

2. Определить архитектуру (или несколько сравниваемых архитектур). Это должно быть сделано в форме, понятной всем участникам оценки.

3. Классифицировать сценарии. Для каждого сценария из набора должно быть определено, поддерживается ли он уже данной архитектурой или для его поддержки нужно вносить в нее изменения. Сценарий может поддерживаться, т.е. его выполнение не потребует внесения изменений ни в один из компонентов, или же не поддерживаться, если его

выполнение требует изменений в описании поведения одного или нескольких компонентов или изменений в их интерфейсах. Поддержка сценария означает, что лицо, заинтересованное в его выполнении, оценивает степень поддержки как достаточную, а необходимые при этом действия — как достаточно удобные.

4. Оценить сценарии. Определить, какие из сценариев полностью поддерживаются рассматриваемыми архитектурами. Для каждого неподдерживаемого сценария надо определить необходимые изменения в архитектуре — внесение новых компонентов, изменения в существующих, изменения связей и способов взаимодействия. Если есть возможность, стоит оценить трудоемкость внесения таких изменений.

5. Выявить взаимодействие сценариев. Определить какие компоненты требуется изменять для неподдерживаемых сценариев; если требуется изменять один компонент для поддержки нескольких сценариев — такие сценарии называют взаимодействующими. Нужно оценить смысловые связи между взаимодействующими сценариями. Малая связанность по смыслу между взаимодействующими сценариями означает, что компоненты, в которых они взаимодействуют, выполняют слабо связанные между собой задачи и их стоит декомпозировать. Компоненты, в которых взаимодействуют много (более двух) сценариев, также являются возможными проблемными местами.

6. Оценить архитектуру в целом (или сравнить несколько заданных архитектур). Для этого надо использовать оценки важности сценариев и степень их поддержки архитектурой.

Рассмотрим сравнительный анализ двух архитектур на примере индексатора — программы для построения индекса некоторого текста, т.е. упорядоченного по алфавиту списка его слов без повторений.

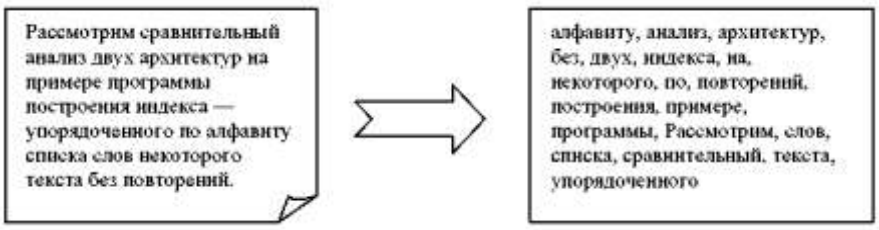


Рисунок 28**.** Пример работы индексатора текста**.**

1. Выделим следующие сценарии работы или модификации программы.
   a. Надо сделать так, чтобы индексатор мог работать в инкрементальном режиме, читая на входе одну фразу за другой и пополняя получаемый в процессе работы индекс.
   b. Надо сделать так, чтобы индексатор мог игнорировать предлоги, союзы, местоимения, междометия, частицы и другие служебные слова.
   c. Надо сделать так, чтобы индексатор мог обрабатывать тексты, подаваемые ему на вход в виде архивов.
   d. Надо сделать так, чтобы в индексе оставались только слова в основной грамматической форме — существительные в единственном числе и именительном падеже, глаголы в неопределенной форме и пр.

2. Определим две возможных архитектуры индексатора для сравнительного анализа.

a. В качестве первой архитектуры рассмотрим разбиение индексатора на два компонента. Один компонент принимает на свой вход входной текст, полностью прочитывает его и выдает на выходе список слов, из которых он состоит. Второй компонент принимает на вход список слов, а на выходе выдает его упорядоченный вариант без повторений. Этот вариант архитектуры построен в стиле «каналы и фильтры».

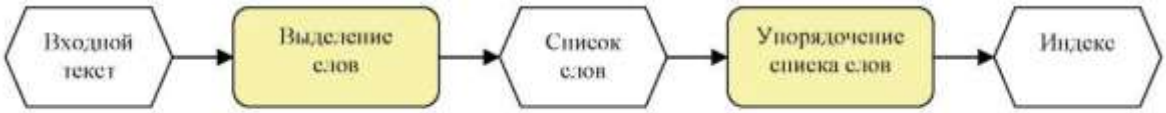


Рисунок 29. Архитектура индексатора в стиле каналов и фильтров**.**

b. Другой вариант архитектуры индексатора устроен следующим образом. Имеется внутренняя структура данных, хранящая подготовленный на настоящий момент вариант индекса. Он представляет собой упорядоченный список без повторений всех слов, прочитанных до настоящего момента. Кроме того, имеются две переменные — строка, хранящая последнее (быть может, не до конца) прочитанное слово, и ссылка на то слово в подготовленном списке, которое лексикографически следует за последним словом (соответственно, предшествующее этому слово в списке лексикографически предшествует последнему прочитанному слову).

В дополнение к этим данным имеются следующие компоненты.

i. Первый читает очередной символ на входе и передает его на обработку одному из остальных. Если это разделитель слов (пробел, табуляция, перевод строки), управление получает второй компонент. Если это буква — третий. Если входной текст кончается — четвертый.

ii. Второй компонент закачивает ввод последнего слова — оно помещается в список перед тем местом, на которое указывает ссылка; после чего последнее слово становится пустым, а ссылка начинает указывать на первое слово в списке.

iii. Третий компонент добавляет прочитанную букву в конец последнего слова, после чего, быть может, перемещает ссылку на следующее за полученным слово в списке.

iv. Четвертый компонент выдает полученный индекс на выход.

Эта архитектура построена в стиле «репозиторий».

Рисунок 30. Архитектура индексатора в стиле репозитория.

3. Определим поддерживаемые сценарии из выделенного набора.

a. Сценарий a.

Этот сценарий прямо поддерживается второй архитектурой. Чтобы поддержать его в первой, необходимо внести изменения в оба компонента так, чтобы первый компонент мог бы пополнять промежуточный список, читая входной текст фраза за фразой, а второй — аналогичным способом пополнять результирующий упорядоченный список, вставляя туда поступающие ему на вход слова.

b. Сценарий b.

Обе архитектуры не поддерживают этот сценарий. Для его поддержки в первой архитектуре необходимо изменить первый компонент или, лучше, вставить после него дополнительный фильтр, отбрасывающий вспомогательные части речи. Для поддержки этого сценария второй архитектурой нужно ввести дополнительный компонент, который перехватывает буквы, выдаваемые модулем их обработки (соответственно, этот модуль больше не должен перемещать указатель по итоговому списку) и сигналы о конце слова от первого компонента, после чего он должен отсеивать служебные слова.

c. Сценарий c.

Этот сценарий также требует изменений в обеих архитектурах. Однако в обоих случаях эти изменения одинаковы — достаточно добавить дополнительный компонент, декодирующий архивы, если они подаются на вход.

d. Сценарий d.

Этот сценарий также не поддерживается обеими архитектурами. Требуемые им изменения аналогичны требованиям второго сценария, только в этом случае дополнительный компонент-фильтр должен еще и преобразовывать слова в их основную форму и только после этого пытаться добавить результат к итоговому индексу.

Таким образом, требуется, как и во втором случае, изменить или добавить один компонент в первой архитектуре и изменить один и добавить новый во второй.

4. Мы уже выполнили оценку сценариев на предыдущем шаге. Итоги этой оценки приведены в Таблице 6.

| Архитектура | Сценарий a | Сценарий **b** | Сценарий c | Сценарий **d** |
|---|---|---|---|---|
| Каналы и фильтры | - - | + + * | + + * | + + * |
| Репозиторий | + + + + | + + - + * | + + + + * | + + - + * |

Таблица **6.** Итоги оценки двух вариантов архитектуры индексатора**:**

**+** обозначает возможность не изменять компонент**,**

**-** обозначает необходимость изменения компонента**,**

***** обозначает необходимость добавления одного компонента

5. Мы видели, что при использовании первого варианта архитектуры только для поддержки первого сценария пришлось бы вносить изменения в ее компоненты. В остальных случаях достаточно было добавить новый компонент, что несколько проще. При

использовании второго варианта нам в двух разных сценариях, помимо добавления нового компонента, потребовалось изменить компонент, обрабатывающий буквы.

6. В целом первая архитектура на предложенных сценариях выглядит лучше второй. Единственный ее недостаток — отсутствие возможности инкрементально поставлять данные на вход компонентам. Если его устранить, сделав компоненты способными потреблять данные постепенно, эта архитектура станет почти идеальным вариантом, поскольку она легко расширяется — для решения многих дополнительных задач потребуется только добавлять компоненты в общий конвейер. Вторая архитектура, несмотря на выигрыш в инкрементальности, проигрывает в целом. Основная ее проблема — слишком специфически построенный компонент-обработчик букв. Необходимость изменить его в нескольких сценариях показывает, что нужно объединить обработчик букв и обработчик конца слов в единый компонент, выдающий слова целиком, после чего полученная архитектура не будет ничем уступать исправленной первой.

## Образцы проектирования

### 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.

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).*

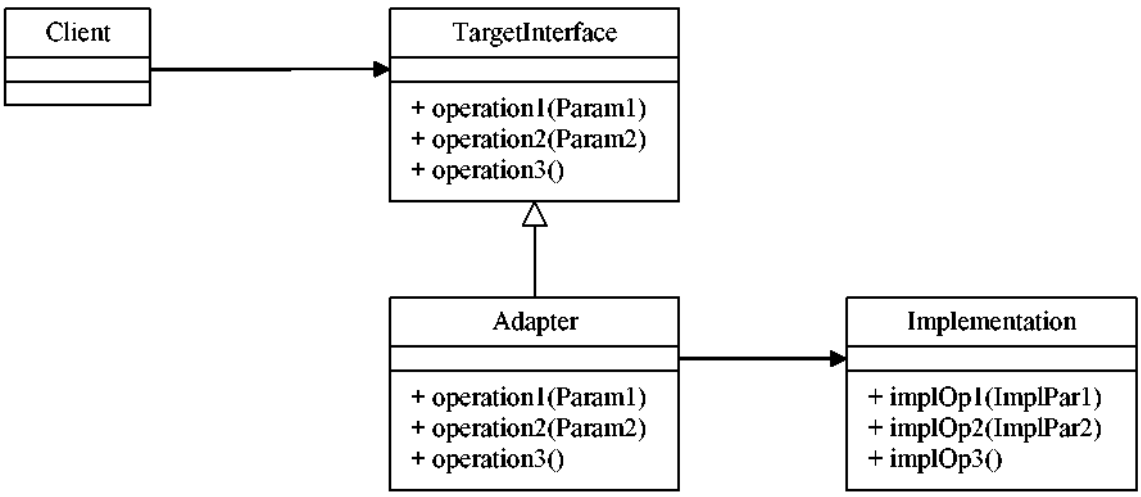


Рисунок 39**.** Структура классов-участников образца адаптер**.**

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#).

- ***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 будут рассмотрены некоторые из известных образцов в соответствии с приведенной

классификацией.

**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).

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.

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.

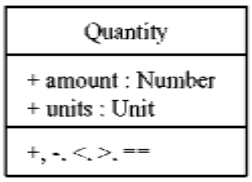


Рисунок 40. Класс для представления величин, имеющих разные единицы измерения.

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 (рис. 41).

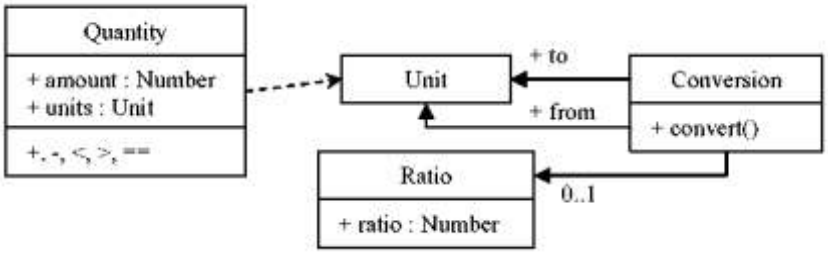


Рисунок 41. Представление возможных преобразований между единицами измерений.

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.

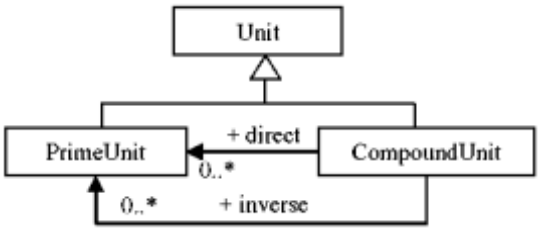


Рисунок 42. Представление составных единиц измерений.

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.

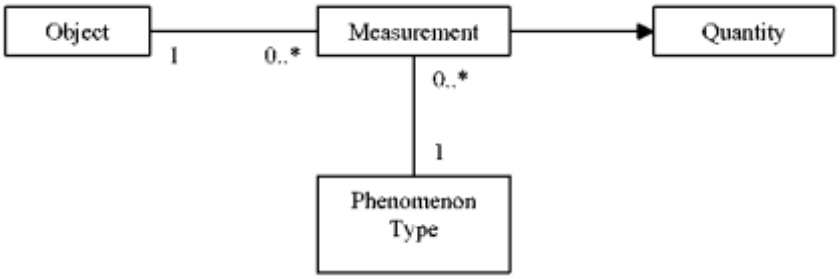

Рисунок 43. Набор классов для представления результатов измерений**.**

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.

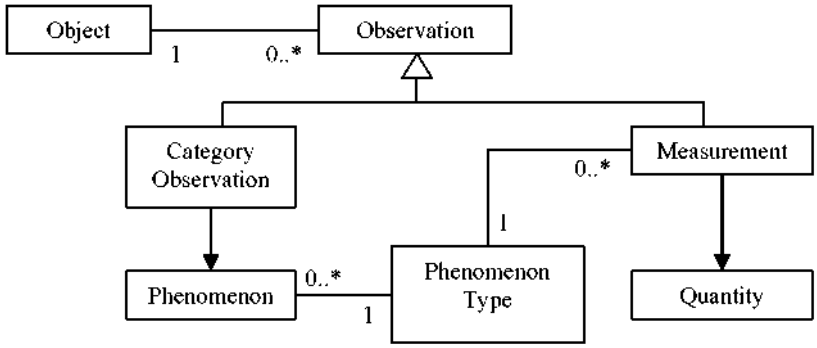


Рисунок 44. Набор классов для представления результатов как измерений**,** так и наблюдений**.**

**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. 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. | Основная схема построения программ |

| | | | |
|---|---|---|---|
| | | Выделяется набор процедур, схема передачи управления между которыми представляет собой дерево с основной процедурой в его корне. | 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.<br>Компоненты разделяются на несколько уровней таким образом, что компоненты данного уровня могут использовать для своей работы только соседей или компоненты предыдущего уровня. Могут быть более слабые ограничения, например, компонентам верхних уровней разрешено использовать компоненты всех нижележащих уровней. | Телекоммуникационные протоколы в модели OSI (7 уровней), реальные протоколы сетей передачи данных (обычно 5 уровней или меньше).<br>Системы автоматизации предприятий (уровни интерфейса 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.<br>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.<br>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.<br>Выделяется общее хранилище данных — репозиторий. Каждый обработчик запускается в ответ на соответствующее ему событие и как-то преобразует часть данных в репозиторий. | 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.<br>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.

**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.

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.

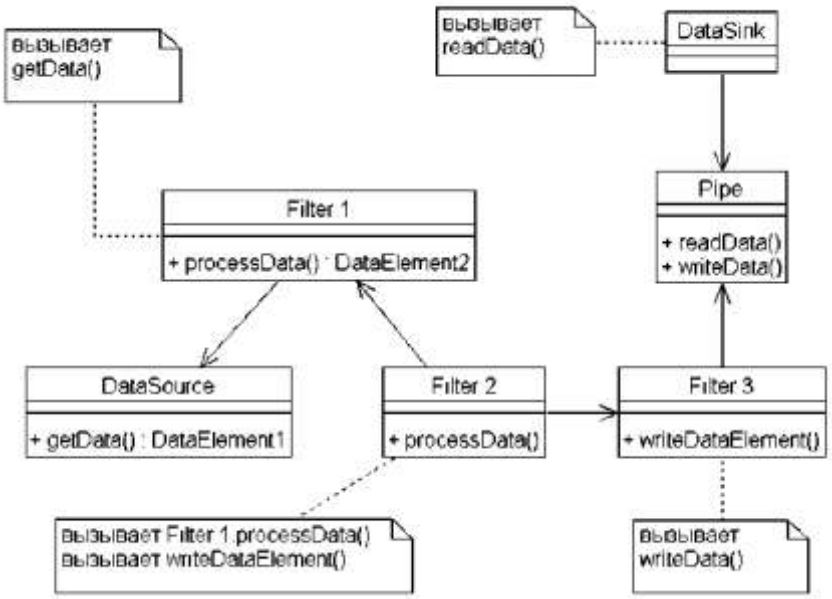


Рисунок 45. Пример структуры классов для образца каналы и фильтры.

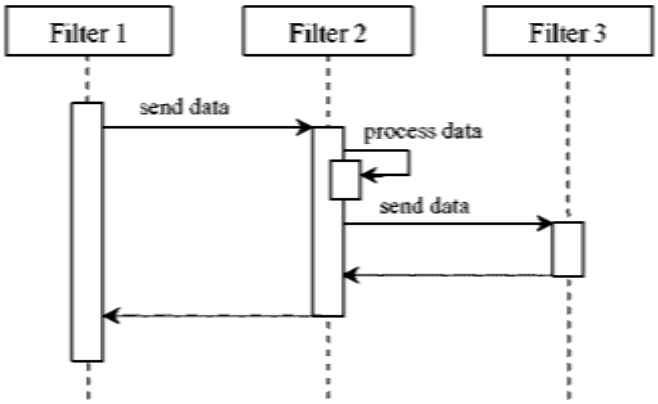


Рисунок 46. Сценарий работы проталкивающего фильтра.

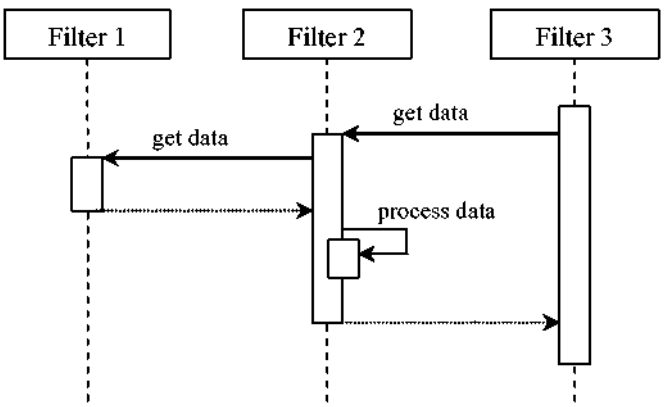


Рисунок 47. Сценарий работы вытягивающего фильтра.

**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.

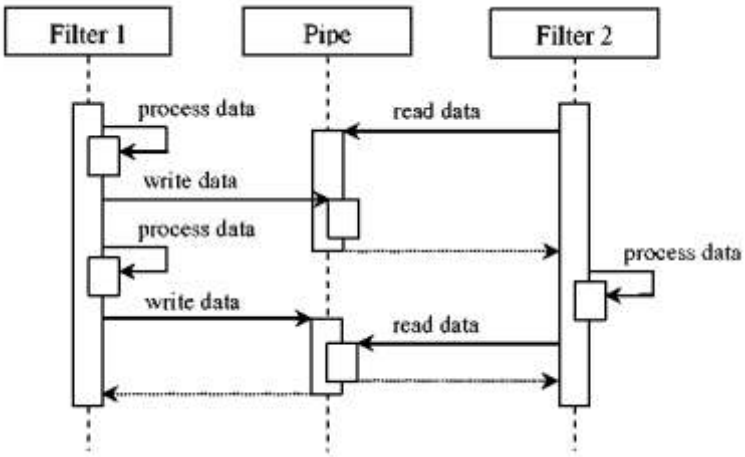


Рисунок 48. Сценарий работы буферизующего и синхронизующего канала**.**

**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, 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.

**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.

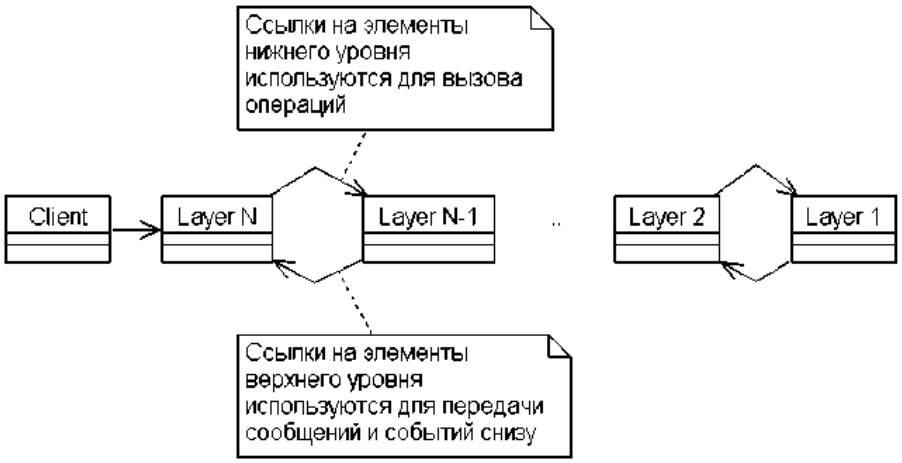


Рисунок 49. Пример структуры многоуровневой системы**.** Классы для уровней условные**,** они обычно декомпозируются на наборы более мелких компонентов**.**

**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.

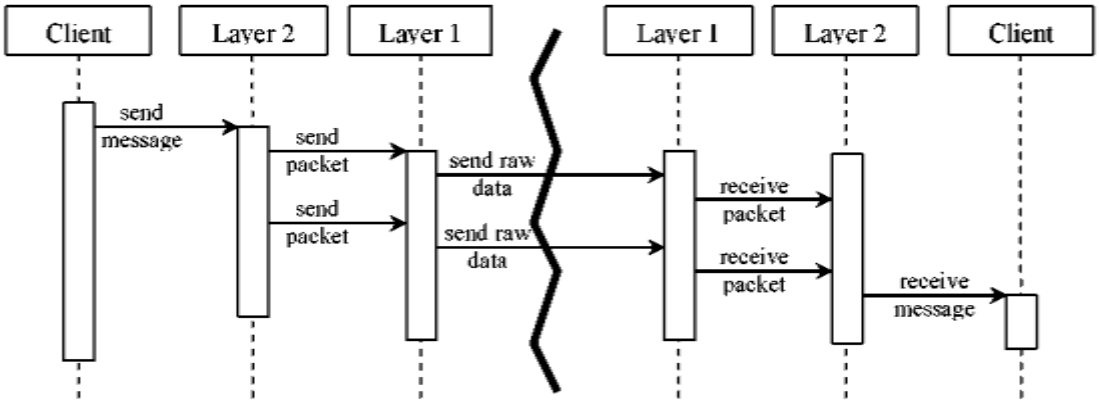


Рисунок 50. Составной сценарий пересылки сообщения по сети**.**

**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 большего, 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). 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:

- 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.

Далее мы продолжим разбирать примеры образцов — рассмотрим детально архитектурный стиль «Данные-представление-обработка», а также примеры образцов: идиом и образцов организации.

**Данные-представление-обработка**

**Название.** Данные-представление-обработка (model-view-controller, MVC).

**Назначение.** Интерактивные приложения с гибким интерфейсом пользователя. Требования к пользовательскому интерфейсу в интерактивных приложениях меняются чаще всего. Разные пользователи имеют разные наборы требований. В несколько меньшей степени это касается методов обработки данных, лежащих в основе таких приложений, — визуальное представление управляющих элементов может меняться вместе с интерфейсом, а сами выполняемые действия зависят от бизнес-логики и предметной области, и поэтому более устойчивы. Наименее подвержена изменениям модель данных, с которыми работает приложение.Поэтому для увеличения гибкости и удобства изменений в таких приложениях необходимо соответствующим образом разделить их компоненты. При этом нужно принимать во внимание следующие факторы.

**Действующие силы.**

- Одна и та же информация может быть представлена по-разному и в нескольких местах для удобства доступа к ней многих различных пользователей, имеющих разные привычки и разные навыки работы с информацией.
- Изменения в данных должны немедленно отображаться в различных представлениях этих данных.
- Внесение изменений в пользовательский интерфейс должно быть максимально простым, иногда оно даже должно быть возможно прямо во время работы приложения.
- Поддержка различных стандартов пользовательского интерфейса и его перенос между платформами не должны влиять на код, связанный с методами работы с данными и структурой данных приложения.

**Решение.** Выделяется три набора компонентов. Первый набор — *данные, модель данных* или просто *модель (model)* — соответствует структуре данных предметной области, в которой работает приложение. Обязанности этих компонентов: представлять в системе данные и базовые операции над ними. Компоненты второго набора — *представления (view)* — соответствуют различным способам представления данных в пользовательском интерфейсе. Для одних и тех же данных может иметься несколько представлений. Каждому компоненту представления соответствует один компонент из третьего набора, *обработчик (controller)* — компонент, осуществляющий обработку действий пользователей. Такой компонент получает команды, чаще всего нажатия клавиш и нажатия кнопок мыши в областях, соответствующих визуальным элементам управления — кнопкам, элементам меню и пр. Эти команды он преобразует в действия над данными. В результате каждого действия требуется обновить все представления всех данных, которые подверглись изменениям.

**Структура.** Основными ролями компонентов в данном стиле являются *модель, представление* и *обработчик.*

Компонент-модель моделирует данные приложения, реализует основные операции над ними и возможность регистрировать зависимые от него обработчики и представления. При

изменениях в данных модель оповещает о них все зарегистрированные компоненты.

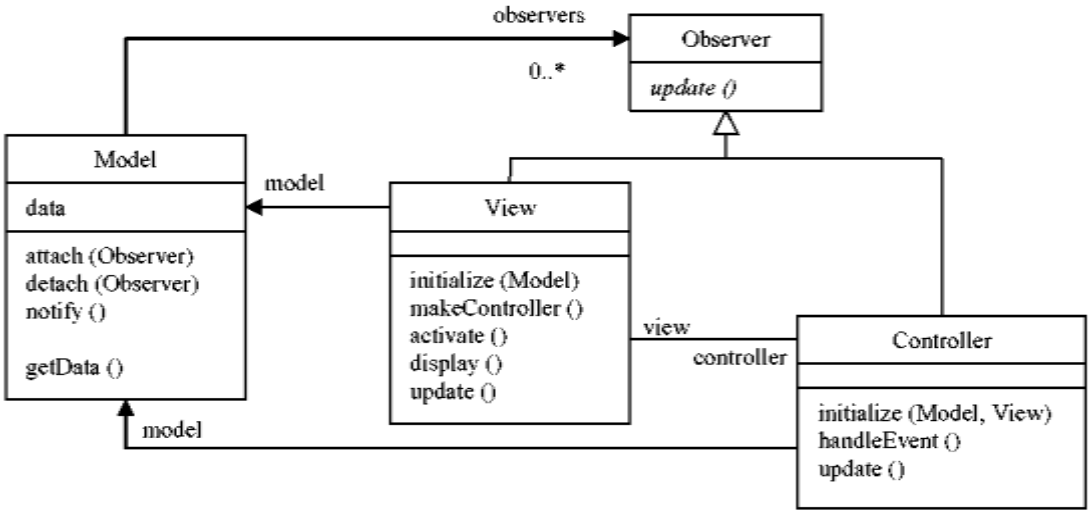


Рисунок 51**.** Структура классов модели**,** представления и обработчика**.**

Компонент-представление представляет данные в некотором виде для пользователей, читая их из модели при необходимости, т.е. при инициализации и после сообщений о произошедших изменениях. Кроме того, он инициализирует связанный с ним обработчик.

Компонент-обработчик обрабатывает действия пользователя, транслируя их в операции над моделью или запросы на показ некоторых элементов представлений. При оповещении об изменениях в модели он соответствующим образом изменяет собственное состояние, например, делая активными или отключая какие-нибудь кнопки и пункты меню.

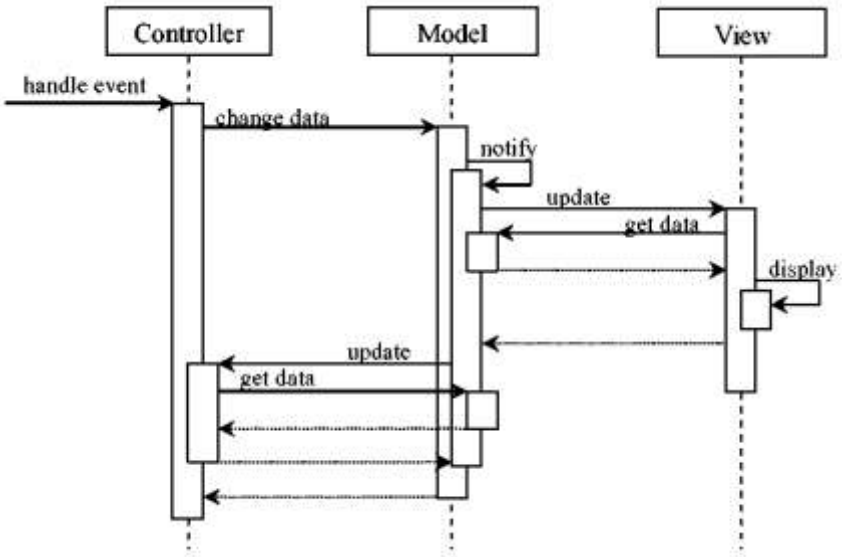


Рисунок 52. Сценарий обработки действия пользователя.

**Динамика.** У системы два базовых сценария работы — инициализация всех компонентов и обработка некоторого действия пользователя с изменением данных и обновлением соответствующих им представлений и обработчиков.

**Реализация.** Основные шаги реализации следующие:

- Отделить взаимодействие человека с системой от базовых функций самой системы.

Для этого необходимо выделить структуру данных, с которыми система работает, и набор необходимых для функционирования системы операций над ними.

- Реализовать механизм передачи изменений. Для этого можно воспользоваться образцом проектирования Подписчик (иначе называемым Наблюдатель, Observer).
- Спроектировать и реализовать необходимые представления.
- Спроектировать и реализовать необходимые обработчики действий пользователя.
- Спроектировать и реализовать связь между обработчиком и представлением. Обычно представление должно инициализировать соответствующий обработчик. Для этого можно, например, использовать образец проектирования Метод порождения (Factory Method).
- Реализовать построение системы из компонентов и инициализацию компонентов. В качестве дополнительных аспектов реализации необходимо рассмотреть следующие.
- Динамические представления, создаваемые во время работы приложения.
- Подключаемые элементы управления, которые могут быть включены во время работы приложения. Например, переключение из режима «новичок» в режим «эксперт».
- Инфраструктура и иерархия представлений и обработчиков. Часто имеется готовая библиотека таких компонентов, на основе которых нужно строить собственные представления и обработчики. Эту задачу нужно решать с учетом семантики и возможностей библиотечных компонентов и связей между ними. Кроме того, одни представления могут визуально включать другие, а также элементы управления, с которыми связаны обработчики. Эта визуальная связь часто должна быть поддержана, например, возможностью переключения фокуса пользовательского ввода между отдельными элементами. Необходимо внимательно спроектировать (насколько это возможно, с учетом ограничений платформы и библиотек визуальных компонентов) стратегии обработки событий, особенно таких, в которых могут быть одновременно заинтересованы несколько компонентов, присутствующих на экране.
- Возможно, потребуется сделать систему еще более переносимой за счет отделения ее компонентов от конкретных библиотек и платформ. Для этого нужно разработать собственный набор абстрактных визуальных компонентов.

**Следствия применения образца.**

Достоинства:

- Возможность иметь несколько представлений одних данных, обновляемых по результатам воздействий пользователя на одно из них. Все такие представления синхронизированы, их показания соответствуют друг другу.
- Поддержка подключаемых и динамически изменяемых представлений и обработчиков.

- Возможность изменения стилей пользовательского интерфейса во время работы.
- Возможность построения каркаса (библиотек визуальных компонентов) для разработки многих интерактивных приложений.

Недостатки:

- Возрастание сложности разработки.
- Потери в производительности из-за необходимости обработки запросов пользователей сначала в обработчиках, затем в моделях, а затем во всех обновляемых компонентах.
- Если не оптимизировать производимые обновления аккуратно, чаще всего в ходе работы происходит много ненужных вызовов операций, обновляющих представления и обработчики.
- Представления и обработчики связаны очень тесно, из-за чего эти компоненты почти никогда нельзя переиспользовать по отдельности.
- И представления, и обработчики достаточно тяжело использовать без соответствующей им модели.
- Представления и обработчики наверняка потребуют изменений при их переносе на другую платформу или в другую библиотеку элементов графического интерфейса пользователя.
- Данный образец тяжело использовать в большинстве средств разработки GUI, поскольку они чаще всего определяют собственные стратегии обработки событий и стандартные обработчики для многих событий, например, для нажатия правой кнопки мыши.

**Примеры.** Впервые этот архитектурный стиль был использован при проектировании библиотеки разработки пользовательского интерфейса для языка Smalltalk. С тех пор создатели множества подобных каркасов и библиотек используют те же принципы.

Еще один пример библиотеки для разработки пользовательского интерфейса, построенного на основе данного стиля — библиотека MFC (Microsoft Foundation Classes) от Microsoft. В ней используется более простой вариант стиля — с объединенными представлениями и обработчиками. Такая схема получила название документ-представление (Document-View): документы соответствуют моделям, представления объединяют функции представлений и обработчиков.

Такое объединение часто имеет смысл, поскольку представления и обработчики тесно связаны и практически не могут использоваться друг без друга. Их разделение на отдельные компоненты должно обосновываться серьезными причинами.

Последний пример использования этого стиля — архитектура современных Web-приложений, т.е. бизнес-приложений с пользовательским интерфейсом на основе HTML и связью между отдельными элементами, построенной на базе основных протоколов Интернет. В ней роль модели играют компоненты, реализующие бизнес-логику и хранение данных, а роль представлений и обработчиков исполняется HTML-страничками и HTML-формами, статичными или динамически генерируемыми. Далее в этом курсе построение таких приложений будет

рассматриваться более детально, поэтому образец «данные-представление-обработка» имеет большое значение для дальнейшего изложения.

**Образцы проектирования**

*Образцы проектирования* в узком смысле являются типовыми проектными решениями, позволяющими удовлетворить часто встречающиеся требования к гибкости приложений и реализовать возможности их расширения за счет специальных форм организации классов и их связей.

Далее мы в деталях рассмотрим образец проектирования «подписчик». О другом примере, «адаптере», было рассказано в начале лекции.

**Подписчик**

**Название.** Подписчик (subscriber) или подписчик-издатель (publisher-subscriber). Известен также под названиями «наблюдатель» (observer), «слушатель» (listener) или «подчиненные» (dependents).

**Назначение.** Реализация системы, в которой нужно поддерживать согласованными состояния большого числа объектов. Чаще всего при этом достаточно много компонентов зависит от небольшого набора данных. Можно было бы связать их явно, введя обращения ко всем компонентам, которые должны знать об изменениях, при каждом внесении изменений, но полученная система станет очень жесткой. Добавление нового компонента потребует сделать изменения во всех местах, от которых он зависит. Предлагаемое в рамках данного образца решение основано на гибкой связи между субъектом (от которого зависят другие компоненты) и этими зависимыми компонентами, называемыми *подписчиками.* Здесь нужно принимать во внимание следующие факторы.

**Действующие силы.**

- Об изменениях в состоянии некоторого компонента должны узнавать один или несколько других компонентов.
- Количество и конкретный вид компонентов, которые должны оповещаться об этих изменениях, заранее не известны или могут быть изменены во время работы приложения.
- Проведение зависимыми компонентами время от времени явных запросов о произошедших изменениях неэффективно.
- Источник информации (субъект или издатель) не должен быть тесно связан со своими подписчиками или зависим от них.

**Решение.** Компонент, от которого зависят другие, берет на себя функции *издателя.* Он предоставляет интерфейс для регистрации *компонентов-подписчиков,* заинтересованных в информации о его изменениях, и хранит множество зарегистрированных подписчиков. При изменениях он оповещает их с помощью вызова предназначенного для этого метода update().

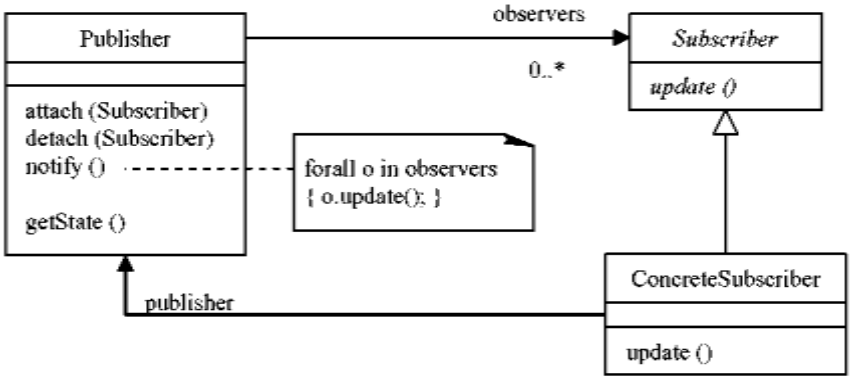


Рисунок 54. Структура классов подписчиков и издателя**.**

**Структура.** Основными компонентами являются *издатель* (или *субъект)* и *подписчики* (или *наблюдатели*). Подписчики реализуют общий интерфейс, у которого имеется метод update() для оповещения подписчика о том, что в состоянии издателя произошли изменения. Издатель хранит множество подписчиков, позволяет регистрировать или удалять их из этого набора. При возникновении изменений он оповещает все элементы этого множества при помощи метода update ().

**Динамика.** Можно использовать два вида обновления подписчиков: издатель сам может сообщать им о том, какие именно изменения произошли (схема проталкивания, push model), или после получения уведомления подписчик сам обращается к издателю, чтобы узнать, что именно изменилось (схема вытягивания, pull model). Вторая схема значительно более гибкая, она позволяет подписчикам получать только необходимую им информацию, в то время как согласно первой схеме каждый подписчик получает всю информацию о произошедших изменениях.

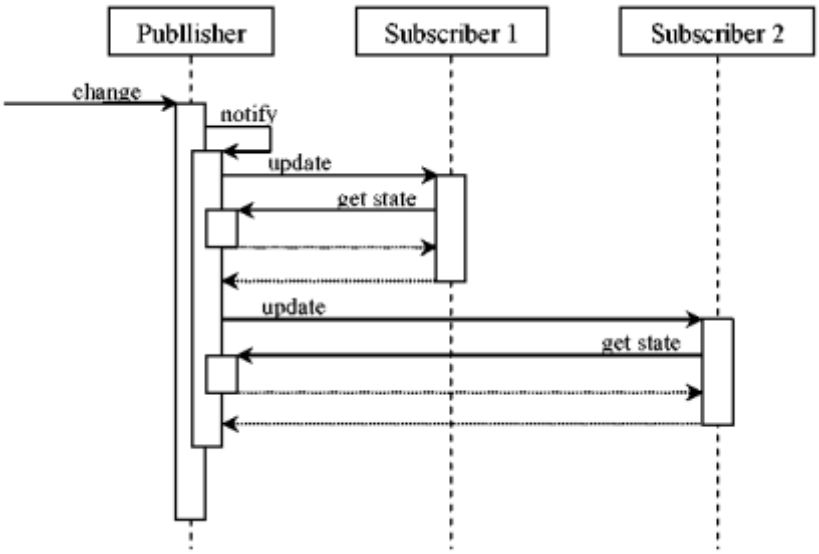


Рисунок 55. Сценарий оповещения об изменениях по схеме вытягивания.

**Реализация**. Основные шаги реализации следующие:

- Определить протокол обновления — будет ли использоваться простой метод update () или в качестве его параметров нужно будет передавать изменившийся

объект-издатель и данные произошедшего изменения.

- Определить схему обновления одного подписчика: на основе проталкивания или на основе вытягивания информации.
- Определить отображение издателей на подписчиков. Если издателей много, а подписчиков мало, то хранение множеств подписчиков для каждого издателя может быть неэффективным — для экономии памяти за счет времени поиска подписчиков можно использовать отдельную таблицу, отображающую издателей на подписчиков.
- Обеспечить гарантии целостности состояния издателя перед оповещением подписчиков.
- Обеспечить гарантии аккуратного удаления объекта-подписчика из системы — нужно, чтобы он был удален из всех списков оповещений.
- Если семантика обновлений сложна, например, если подписчик зависит от нескольких издателей, которые могут изменяться в рамках одной операции, то, возможно, потребуется выделить такие сложные связи в отдельный компонент, называемый менеджером изменений (change manager). Такой компонент должен сам хранить отображение между издателями и подписчиками, определять стратегию проведения обновления и обновлять всех зависимых подписчиков по запросу от издателя. В качестве стратегии обновления может выступать механизм, гарантирующий подписчику получение только одного уведомления, если изменяются несколько издателей, от которых он зависит.

**Следствия применения образца.**

Достоинства:

- Слабая связанность между издателем и подписчиками — издатель знает только, что у него есть несколько подписчиков с одинаковым интерфейсом.
- Удобным образом, не зависящим от числа участников, поддерживаются широковещательные оповещения.

Недостатки:

- Низкая производительность в случае большого количества подписчиков — даже небольшое изменение требует оповестить их всех, и каждый из них будет пытаться провести обновление своего состояния.
- Неэффективное использование информации из-за необходимости оповещать всех подписчиков, даже тех, для которых выполненные изменения несущественны. Кроме того, подписчик может зависеть от нескольких издателей и не знать, какой именно издатель оповещает его об изменении. Чтобы исправить эту ситуацию, необходимо внесение изменений в протокол оповещения — предоставление дополнительной информации, как об изменившемся издателе, так и о виде изменения.

**Примеры.** Один из примеров мы уже видели — в рамках более широкого стиля

«данные-представление-обработка» представления и обработчики являются подписчиками по отношению к модели-издателю.

Вариант этого образца с введением менеджеров изменений описан под названием «канал событий» (Event Channel) в спецификации службы оповещения о событиях в стандарте CORBA.

**Идиомы**

***Идиома*** представляет собой типовое решение, определяющее специфическую структуризацию элементов кода на некотором языке программирования. Чаще всего это некоторый «трюк», с помощью которого можно придать программе на данном языке нужные свойства. При этом идиома может оказаться специфичной для языка и не иметь аналога в других языках. Кроме того, очень часто удачные идиомы при развитии языков программирования превращаются в новые синтаксические конструкции, делающие такую идиому ненужной.

Далее мы увидим примеры таких идиом в виде соглашений о построении кода компонентов JavaBeans, превратившихся в C# в элементы языка.

В этой лекции мы рассмотрим в качестве примера идиому «шаблонный метод».

**Шаблонный метод**

**Название.** Шаблонный метод (template method).

**Назначение.** Фиксация общей схемы некоторого алгоритма, имеющего много вариантов, с предоставлением возможности реализовать эти варианты за счет переопределения отдельных его шагов, без изменения схемы в целом. При использовании этой идиомы нужно принимать во внимание следующие факторы.

**Действующие силы.**

- Инвариантные части алгоритма нужно записать один раз и переиспользовать, изменяя только варьирующиеся шаги и элементы.
- Иногда такие инвариантные части еще нужно выделить, чтобы сделать более понятным и более удобным для сопровождения код нескольких классов, реализующих близкие по назначению методы.
- Многие алгоритмы при их практическом использовании могут зависеть от большого числа факторов, изменяющихся гораздо чаще, чем общая схема такого алгоритма (вспомните принцип разделения политик и алгоритмов).

**Решение.** Общая схема алгоритма, которую нужно зафиксировать, помещается в абстрактный класс в виде метода, код которого реализует эту схему. В тех местах, где необходимо использовать какой-то варьирующийся элемент алгоритма, этот метод обращается к другому методу данного класса, который можно переопределить в классах-потомках.

**Структура.** Метод, реализующий основную схему алгоритма, называется *шаблонным методом.* Он пишется один раз в абстрактном базовом классе и не переопределяется в потомках. Методы, вызываемые им, делятся на следующие группы:

- Конкретные *операции,* реализация которых известна на момент написания метода и не должна изменяться.
- Абстрактные *операции,* которые представляют собой изменяемые части алгоритма.

Они не имеют реализации и должны определяться в каждом классе-потомке в соответствии с теми вариациями, которые он вносит в базовый алгоритм.

- Операции-*перехватчики,* или *зацепки (hook operations),* которые также представляют собой изменяемые элементы алгоритма, но имеют некоторую реализацию по умолчанию, записанную в базовом классе. Эти операции могут переопределяться в классах-потомках, если представляемые ими элементы алгоритма нужно изменить по сравнению с имеющейся реализацией по умолчанию.
- Фабричные *методы (factory methods),* предназначенные для создания объектов, которые связаны с работой конкретного варианта алгоритма. Они реализуются по образцу, также называемому «фабричный метод». Суть такого метода в том, что при его вызове мы точно не знаем, объект какого конкретного класса будет создан — это зависит от текущей конфигурации системы.

**Динамика.** Типичный сценарий работы шаблонного метода показан на Рис. 56. Для наглядности на этой диаграмме операции, выполняемые в одном объекте, разделены между двумя виртуальными объектами: первый представляет все действия, выполняемые в рамках абстрактного класса, определяющего шаблонный метод, а второй — те действия, которые (пере)определяются в конкретном подклассе.

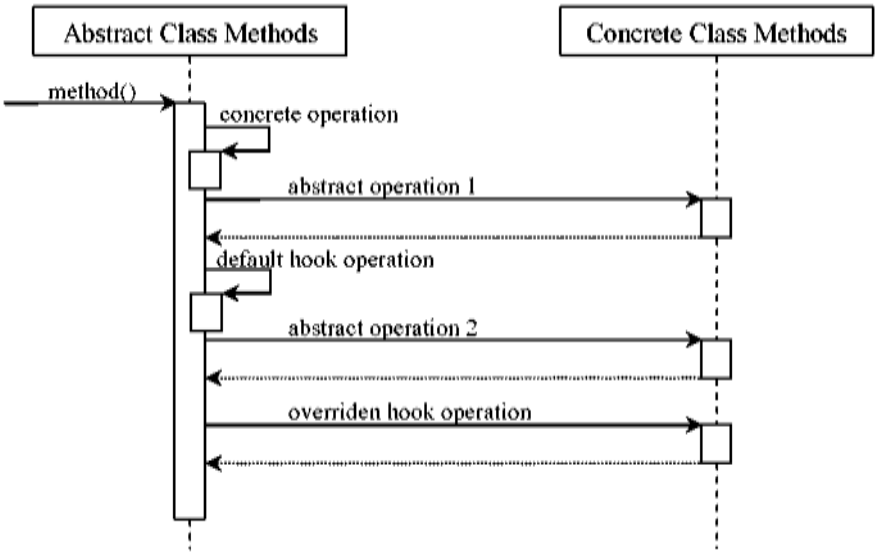


Рисунок 56. Сценарий работы шаблонного метода.

**Реализация.** Основные шаги реализации следующие.

- Определить основные шаги алгоритма. Определить набор данных, которыми он пользуется.
- Выделить среди шагов алгоритма те, которые зависят от конкретных политик. Определить для каждого такого шага абстрактный метод. Выделить среди данных, которыми пользуется алгоритм, те, чья структура зависит от политик или конкретного варианта алгоритма. Определить для порождения таких данных фабричные методы, а для операций над ними — абстрактные методы. Для их представления могут понадобиться дополнительные классы, но изменяемая часть

данных должна, по возможности, находиться в абстрактном классе, определяющем шаблонный метод.

- Реализовать общую схему алгоритма в теле шаблонного метода. Выделить его элементы, используемые несколько раз или представляющие собой отдельные операции, в отдельные методы абстрактного класса.
- Определить, какие дополнительные возможности по вариации поведения алгоритма могут понадобиться. Определить для таких возможностей методы-перехватчики.
- Определить несколько наиболее часто используемых вариантов алгоритма. Реализовать их в виде подклассов определяющего шаблонный метод класса, определив в них методы, которые представляют абстрактные операции, и, если это необходимо, переопределив методы-перехватчики.

**Следствия применения образца.**

Достоинства:

- Общая часть алгоритма реализуется явно и может быть легко переиспользована.
- Изменяемые части алгоритма могут варьироваться удобным образом, не влияя друг на друга и давая в результате различные его модификации.
- Алгоритм может быть параметризован большим набором политик, для каждой из которых возможна реализация по умолчанию.

Недостатки

- Снижение понятности кода за счет сложного потока управления.
- Снижение производительности в случае большого числа параметров, из которых в каждом конкретном варианте алгоритма используется лишь несколько.

**Примеры.** Шаблонные методы очень часто используются при построении библиотечных классов и каркасов приложений.

Жизненный цикл компонентов EJB реализован в виде шаблонного метода, в котором абстрактной операцией служит создание объектов данного компонента. Имеется также несколько операций-перехватчиков, позволяющих разработчику компонента специфическим образом обрабатывать переход компонента из одного состояния в другое.

Другой пример — реализация метода start (), запускающего отдельный поток в Java. Инструкции, выполняемые в рамках потока, помещаются в метод run () объекта класса Thread или класса, реализующего интерфейс Runnable. Этот метод служит операцией-перехватчиком для метода start () — реализация метода run () по умолчанию ничего не делает.

**Образцы организации и образцы процессов**

Образцы организации работ и образцы процессов существенно отличаются от остальных видов образцов, рассматриваемых здесь. Они фиксируют успешные практики по организации деятельности, связанной с разработкой ПО (или другими сложными видами деятельности).

Такие образцы только поддерживают проектирование и разработку ПО, не давая вариантов самих проектных решений.

Образцы этого вида чаще всего извлекаются из форм организации работ и процессов, принятых в успешных компаниях-производителях ПО. Плодотворность их использования оценивается управленцами достаточно субъективно. При этом, однако, для признания некоторого вида организации работ образцом, необходимо успешное ее использование для решения одних и тех же задач в нескольких организациях.

Шаблон для описания таких образцов выглядит следующим образом:

- Название образца.
- Контекст использования, включающий основную решаемую задачу и начальные условия.
- Действующие силы — проблемы, ограничения, требования, рассуждения и идеи, под воздействием которых вырабатывается решение.
- Решение — описание используемой формы организации работ, выделяемых подзадач, выполняемых действий, используемых техник.
- Итоговый контекст — описание ожидаемых результатов использования образца, обоснование того, что его применение даст нужный эффект.

В качестве примера образца организации работ приведем процесс *инспекции программ (Fagan inspection process)*, определенный Майклом Фаганом (Michael Fagan) (похожий процесс, называется технической экспертизой, technical review).

**Инспекция программ по Фагану**

**Название.** Инспекция программ по Фагану (Fagan inspection process).

**Контекст использования.** Поиск ошибок на ранних этапах разработки программного обеспечения — при подготовке требований, проектировании, начальных этапах кодирования, планировании тестов.

**Действующие силы:**

- Усилия, необходимые для исправления ошибки, и, соответственно, ее стоимость возрастают в зависимости от этапа проекта, на котором она обнаружена. Из эмпирических данных известно, что каждый раз при переходе через границу между фазами (при использовании водопадной модели разработки) подготовка требований - проектирование - кодирование - тестирование - эксплуатация трудозатраты на исправление найденных на данном этапе ошибок возрастают в 3-5 раз. При использовании итеративных моделей затраты возрастают меньше, но не намного. Поэтому, чем раньше ошибки будут обнаруживаться, тем эффективней будет разработка в целом.
- Членам команды разработчиков надо понимать, над чем работает каждый из них и какие решения он использует. Это помогает значительно повысить эффективность собственной работы.
- Каждый артефакт — требования, проектные документы, код, тестовые планы —

должен быть подготовлен на нужном уровне качества, прежде чем он будет использован для дальнейшей работы.

- Знания о найденных ошибках позволяют членам команды избегать их повторения, а также обращать больше внимания на компоненты, которые оказались наиболее подвержены ошибкам на предыдущих этапах.

**Решение.** Несколько членов команды разработчиков проводят тщательную инспекцию результатов работы одного из них. Такие инспекции основываются на *первичных документах,* чтобы проверить соответствие им *вторичных документов.* Первичные и вторичные документы для каждого вида деятельности в ходе разработки, для которых проведение инспекций эффективно, представлены в Таблице 8.

Выделяются следующие роли участвующих в процессе инспекции лиц:

- Ведущий *(moderator).* Он руководит проведением инспекции, руководит собраниями, фиксирует обнаруженные ошибки, назначает время проведения собраний, сроки подготовки отчетов, следит за исправлением найденных ошибок. В качестве ведущего должен использоваться компетентный разработчик или архитектор, не вовлеченный в проект, материалы которого инспектируются.
- Автор *(author).* Это автор первичного документа или человек, имеющий достаточно полное представление о нем. Его обязанности — подготовить рассказ об основных положениях первичного документа и отвечать на вопросы, возникающие у членов инспектирующей команды по его поводу.
- Интерпретатор *(reader).* Это автор вторичного документа, который разработан в соответствии с первичным. Его обязанности — объяснить участникам инспекции основные идеи, лежащие в основе его интерпретации первичного документа, и отвечать на их вопросы по поводу вторичного документа.
- Инспектор *(tester).* В ходе всей инспекции он анализирует вторичный документ, проверяя его на соответствие первичному.

| **Вид деятельности** | **Первичные документы** | **Вторичные документы** |
|---|---|---|
| Анализ требований | Модели предметной области, составленные заказчиками и | Требования к ПО |
| Проектирование | Требования к ПО | Описание архитектуры, проектная документация |
| Кодирование | Проектная документация | Код, проектная документация на отдельные |
| Тестирование | Требования к ПО, проектная документация, | Тестовые планы и наборы тестовых |

Таблица 8. Первичные и вторичные документы на разных этапах разработки**.**

Обычно рекомендуется использовать не более 4-х человек в команде, проводящей инспекцию. Расширение ее возможно в особых случаях и только за счет разработчиков,

которым непосредственно придется иметь дело с инспектируемыми вторичными документами.

Сам процесс инспекции состоит из следующих шагов.

1. *Планирование (planning).*

На этом шаге ведущий должен убедиться в том, что первичный и вторичный документы готовы к проведению инспекции — они существуют, написаны достаточно понятно, с достаточной степенью детализации.

Кроме того, на этом шаге проводиться планирование всего хода инспекции — определяются участники, их роли, назначаются сроки проведения собраний и время, выделяемое на выполнение каждого шага.

2. *Обзор (review).*

Проводится собрание, на котором автор представляет наиболее существенные положения первичного документа и отвечает на вопросы участников о нем. Первичный и вторичный документы выдаются на руки участникам инспекции для дальнейшей работы.

Ведущий объясняет задачи данной инспекции, вопросы и моменты, на которые стоит обратить особое внимание, а также сообщает, какие ошибки были уже обнаружены в рассматриваемых документах, чтобы участники группы имели представление об их проблемных местах. "

3. *Подготовка (preparation).*

Каждый из участников тщательно изучает оба документа самостоятельно, пытаясь понять заложенные в них решения и проследить их реализацию.

Часто на этом этапе обнаруживаются ошибки, но гораздо меньше, чем на следующем.

4. *Совместная инспекция (inspection meeting).*

Проводится совместное собрание, на котором интерпретатор рассказывает об основных идеях и техниках, использованных во вторичном документе, а также объясняет, почему были приняты те или иные решения и почему они соответствуют первичному документу.

Участники задают вопросы и акцентируют внимание на проблемных местах. Как только ведущий по ходу собрания замечает ошибку (или кто-то обращает его внимание на нее), он сообщает о ней и убеждается, что все участники согласны с тем, что это именно ошибка, т.е. несоответствие между первичным и вторичным документами. Каждая ошибка фиксируется, описывается ее положение, она классифицируется по некоторой схеме, например, критическая (приводящая к ошибке в работе системы) или некритическая (связанная с опечатками, излишней сложностью или неудобством интерфейса и пр.).

5. *Доработка (rework).*

В ходе доработки интерпретатор исправляет обнаруженные ошибки.

6. *Контроль результатов (follow-up).*

Результаты доработки проверяются ведущим. Он проверяет, что все найденные ошибки были исправлены и что не было внесено новых ошибок. Если по результатам инспекции было переработано более 5% вторичного документа, следует провести полную инспекцию вновь.

Иначе ведущий сам определяет, насколько документ подготовлен к дальнейшему использованию.

Кроме того, ведущий подготавливает отчет обо всех обнаруженных ошибках для последующего использования в других инспекциях и при оценке качества результатов разработки.

**Итоговый контекст.** В результате проведения инспекций повышается качество проектных документов и кода, разработчики знакомятся ближе с работой друг друга и с задачами проекта в целом, углубляют понимание проблем проекта и используемых в нем решений. Кроме того, руководитель проекта получает надежные данные о качестве результатов разработки.

Руководитель должен понимать, что *результаты инспекций не должны использоваться как показатель качества работы разработчиков,* иначе все положительные эффекты от их проведения пропадают — разработчики начинают неохотно участвовать в инспекциях, скрывают детали своей работы, снисходительнее относятся к ошибкам других в расчете на взаимность и пр.

При выполнении этого условия инспекции являются эффективным средством обнаружения ошибок на ранних этапах. Статистика показывает, что они находят до 80% ошибок, обнаруживаемых за весь период разработки ПО.