[1. Основные принципы построения объектной модели](#_376d81hpm6br)

[2. Способы использования языка UML](#_dempbb1g57u4)

[3. Нотация языка UML](#_lyyej43nd7pk)

[4. Виды диаграмм UML](#_vey2js6h1xk8)

[5. Диаграмма прецедентов](#_aanuqqnn5ojt)

[6. Диаграмма классов](#_hrrlnuci85zb)

[7. Диаграмма объектов](#_q4dd5yhcifk1)

[8. Диаграмма последовательностей](#_as3cfv6i3oqt)

[9. ДИАГРАММА ВЗАИМОДЕЙСТВИЯ](#_gt0ji8ml9gek)

[10. Диаграмма состояний](#_1v0o263i4w95)

[11. Диаграмма активности](#_4chqers6tul7)

[12. Диаграмма развёртывания](#_gkaabwi1ian8)

[13. Последовательность построения диаграмм](#_r40stsk2vclv)

[14. Отображение класса и его элементов на диаграмме UML](#_p5vfi3cr995t)

[15.Способы использования объектов класса](#_x5xzbcwivrst)

[16. Моделирование наследования в UML](#_mvqujjkwt18i)

[17. Отношения между классами](#_vf611xezpd99)

[18. Отношение зависимости между классами](#_jdfz235ymhtf)

[19. Отношение ассоциации между классами](#_9bzfc3cms8aq)

[20. Композиция и агрегация классов](#_s63c53510b8e)

[21. Сравнение диаграмм активностей и блок-схем](#_ukd6zoswj8mg)

[22. Моделирование процессов диаграммами д это бог всеактивности](#_tgvj0lhanpo6)

[23. Моделирование операций диаграммами активности](#_kp4222khdoqh)

[24. Правила построения диаграммам активности](#_i8pbyrkvjcyn)

[25. Диаграмма кооперации](#_8h9kv7je8mc1)

[26. Диаграмма последовательностей как диаграмма взаимодействия](#_ow0bjndn1ujq)

[28. Диаграмма кооперации как диаграмма взаимодействий объектов](#_x94svumtcct)

[29. Виды сообщений: синхронные и асинхронные, ответные, потерянные и найденные.](#_op9f3hihzgfe)

[30. Уровни экземпляров и спецификации в диаграммах кооперации.](#_kiueeiscdedw)

[31. Мультиобъекты, композитные и активные объекты в диаграммах кооперации.](#_z8l28ld5t5wq)

[32. Диаграммы взаимодействия с разветвленным потоком управления.](#_fw2tg873a2zn)

[33. Нефункциональные требования и их отображение на диаграммах прецедентов.](#_8f88sjkgclwk)

[34. Понятие эктора и отношения между экторами](#_7cby2epzvo6s)

[35. Отношения включения и расширения между экторами](#_25mwn2ou07r0)

[36. Причины использования прецедентов.](#_kiueeiscdedw)

[37. Прецеденты в прямом и обратном проектировании](#_1lri9uyygkoz)

[38. Обзор CASE-средств для построения диаграмм UML](#_ah7wg5j4t0kl)

[39. Критерии выделения прецедентов.](#_9xq6cie9hh5d)

[40. Понятие шаблона проектирования](#_olxumt51gz97)

[41. Основные шаблоны GRASP](#_t6j0mihcrm4i)

[42. Описание шаблонов проектирования (GoF)](#_hmld0gtnk60m)

[43. Классификация шаблонов проектирования GoF](#_gognkijd81z4)

[44. Структурные шаблоны проектирования](#_gognkijd81z4)

[45. Шаблоны централизованного управления](#_gognkijd81z4)

[Сценарий транзакций](#_yrq5asn2th11)

[Диспетчер](#_tytbn7pg9hlx)

[46. Шаблоны управления на основе событий](#_gognkijd81z4)

[Передача сообщений](#_ax3ysm2g109b)

[Управление прерываниями](#_7y9d5dq9uddm)

[47. Шаблоны взаимодействия с базами данных](#_gognkijd81z4)

[Активная запись](#_p0ih50lhm1no)

[Единица работы](#_85xlrrmif719)

[Загрузка по требованию](#_kxoh4prcyp5x)

[Количество объектов](#_2kt48f3btv55)

[Множество записей](#_ny53q6o8slxh)

[Шлюз записи данных](#_pxaosv1wh1xz)

[48. СТРУКТУРНЫЕ ШАБЛОНЫ ИНТЕГРАЦИИ](#_nx9jmwviryvj)

[49. ПОНЯТИЕ АНТИ-ШАБЛОНА ИЛИ ЛОВУШКИ ПРОЕКТИРОВАНИЯ](#_p3tl2rk49dvq)

[50. МЕТОДЫ НАСТРОЙКИ И ОБЕСПЕЧЕНИЯ ГИБКОСТИ ПРОЕКТОВ](#_rngxcmzhxokq)

[51. Понятие стратегии и классов стратегии](#_45ak1mrnl4o)

[52. Способы реализации классов стратегии](#_t6t5witjjysc)

[53. Неполная конкретизация классов стратегии](#_oybgrvuhyuej)

[54. Функторы и их использование](#_gt0yeljnowaa)

[55. Понятие рефакторинга программ](#_kt12ejxt6il4)

[56. Анти-шаблоны управления разработкой программ](#_gsnplzwcugjb)

[57. Анти -шаблоны разработки программ](#_yild0ublf9v9)

[59. Анти-шаблоны в программировании](#_obmwe6bfm6yk)

[63. СОЦИАЛЬНЫЕ АНТИ-ШАБЛОНЫ](#_czas96vfxbhg)

[64. Основные типы шаблонов проектирования](#_xb8gaj97dwlv)

[65. Шаблон делегирования](#_108fq4rebh1s)

[66. Шаблон функционального дизайна](#_e0yu17dxeu8c)

[67. Неизменяемый объект (шаблон проектирования)](#_3tkn2h62r8gv)

[68. Интерфейс (шаблон проектирования)](#_rbl6muz3h0a)

[69. Порождающие шаблоны проектирования](#_dzuqffz63xl7)

[70. Абстрактная фабрика (шаблон проектирования)](#_glx0ktlyjymz)

[71. Строитель (шаблон проектирования)](#_qde8nrqjopoq)

[72. Фабричный метод (ШАБЛОН ПРОЕКТИРОВАНИЯ)](#_6o9s9gih5224)

[73. Отложенная инициализация (ШАБЛОН ПРОЕКТИРОВАНИЯ)](#_azuvf8dwnlhj)

[74. Объектный пул (шаблон проектирования)](#_3vxsfc4052ju)

[75. Прототип (шаблон проектирования)](#_1y8qsxy5wx3k)

[Вкратце](#_lwt6kd73ujfd)

[Определение](#_llm4c7vuulym)

[Пример реализации:](#_sdjd6x7tm8ev)

[Применение](#_2ige97vy31jj)

[76. Получение ресурса есть инициализация (шаблон проектирования)](#_yxupu7qqq95o)

[Вкратце](#_nsx6czoxd9vn)

[Пример](#_8tc1tgq0yh7g)

[Плюсы и минусы](#_prwgv1yp6wto)

[77. Одиночка (шаблон проектирования)](#_pdqufucoo935)

[Вкратце](#_1760ao9ljx04)

[Определение](#_1hewh2mi755j)

[Пример](#_9n7af6x3lm5m)

[Проблемы](#_62lch4ak17ri)

[78. Структурные шаблоны](#_gu5r0wu4zsrt)

[Определение](#_bs2i1pgtcz2j)

[79. Адаптер (шаблон проектирования)](#_o0znt6duwv8x)

[Вкратце](#_jvlsb4veowfn)

[Определение](#_kxe9v6r52fnw)

[Пример](#_dpijfvpiv85t)

[80. Мост (шаблон проектирования)](#_vkccaxuev7h8)

[Вкратце](#_bnw9hionztlm)

[Определение](#_t3qd4zgvlrxh)

[Пример](#_z1umhbex99h3)

[81. Компоновщик (шаблон проектирования)](#_yiked69tdbh5)

[Вкратце](#_15w1wwbt8mi)

[Определение](#_o3uiuqle7l6z)

[Пример](#_p6anp18rk6ic)

[82. Декоратор (шаблон проектирования)](#_vyhkif7yx2qr)

[83. Фасад (шаблон проектирования)](#_7136v645ah48)

[84. Приспособленец (шаблон проектирования)](#_7136v645ah48)

[85. Заместитель (шаблон проектирования)  
Вкратце](#_7136v645ah48)

[86. Поведенческие шаблоны](#_7136v645ah48)

[87. Цепочка ответственности (шаблон проектирования)](#_7136v645ah48)

[Когда применяется цепочка обязанностей?](#_jv80ifxdiqo7)

[88. Команда (шаблон проектирования)](#_7136v645ah48)

[89. Интерпретатор (шаблон проектирования)](#_7136v645ah48)

[Обсуждение паттерна Interpreter](#_uevcewks7wnj)

[Структура паттерна Interpreter](#_nknkhdcevq3c)

[UML-диаграмма классов паттерна Interpreter](#_wiedi3764pvi)

[90. Итератор (шаблон проектирования)](#_7136v645ah48)

[Преимущества и недостатки](#_nped0evhcmi)

[91. Посредник (шаблон проектирования)](#_7136v645ah48)

[92. Хранитель (шаблон проектирования)](#_7136v645ah48)

[93. Наблюдатель (шаблон проектирования)](#_uufu7erhtm4g)

[94. Состояние (шаблон проектирования)](#_dy07isa7dqef)

[95. Стратегия (шаблон проектирования)](#_b2lcjxorvvix)

[96. Шаблоны параллельного программирования](#_b2lcjxorvvix)

[97. Модель-представление-контроллер (шаблон проектирования)](#_b2lcjxorvvix)

[98. Технология использования шаблонов проектирования](#_b2lcjxorvvix)

[99. Идиомы программирования и шаблоны проектирования](#_b2lcjxorvvix)

# 

# 1. Основные принципы построения объектной модели

Концептуальной основой объектно-ориентированного подхода является объектная модель. Основными принципами ее построения являются:

• абстрагирование;

• инкапсуляция;

• модульность;

• иерархия.

Абстрагирование — это выделение наиболее важных, существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы с точки зрения дальнейшего рассмотрения и анализа, и игнорирование менее важных или незначительных деталей.

Абстрагирование позволяет управлять сложностью системы, концентрируясь на существенных свойствах объекта. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности его поведения от деталей их реализации. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу ООП.

Инкапсуляция — физическая локализация свойств и поведения в рамках единственной абстракции, скрывающая их реализацию за общедоступным интерфейсом.

Инкапсуляция - это процесс отделения друг от друга отдельных элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать интерфейс объекта, отражающий его внешнее поведение, от внутренней реализации объекта. Объектный подход предполагает, что собственные ресурсы, которыми могут манипулировать только операции самого объекта, скрыты от внешней среды.

Инкапсуляция подобна понятию сокрытия информации. Это возможность скрывать многочисленные детали объекта от внешнего мира. Внешний мир объекта — это все, что находится вне его, включая остальную часть системы.

Модульность — это свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне сильно сцепленных, но слабо связанных между собой подсистем (модулей).

Модульность снижает сложность системы, позволяя выполнять независимую разработку отдельных модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

Иерархия — это ранжированная или упорядоченная система абстракций, расположение их по уровням.

Основными видами иерархических структур применительно к сложным системам являются структура классов и структура объектов. Примерами иерархии классов являются простое и множественное наследование, а иерархии объектов - агрегация.

# 2. Способы использования языка UML

Итак, UML можно использовать для рисования картинок, которые можно использовать для коммуникаций внутри команды и в ходе взаимодействия с заказчиком, т. е. он может служить средством обмена информацией. Кроме этого, UML является отличным средством спецификации систем, причем спецификации в процессе разработки. Разработанные архитектурные решения, задокументированные с помощью UML, могут быть использованы повторно.

Для чего UML использовать нельзя, вернее, чем он не является. Во-первых, UML не является языком программирования, хотя существуют средства выполнения UML-моделей как интерпретируемого кода и возможна кодогенерация. Несмотря на это, UML - средство не программирования, а моделирования, т. е. создания не программ, а моделей любого уровня абстракции для систем из любой предметной области. Во-вторых, UML не является и спецификацией какого бы то ни было инструмента моделирования, хотя такие инструменты имеются. Каким образом то или иное CASE-средство реализует UML-моделирование, никак не регламентируется и определяется самими разработчиками этих инструментов. В-третьих, UML не является и моделью какого-либо процесса разработки, даже Rational Unified Process, который был описан именно с помощью UML. UML можно использовать независимо от того, какую методологию разработки ПО вы используете, и даже если вы вообще не пользуетесь никакой методологией.

# 3. Нотация языка UML

Семантика языка UML. Представляет собой некоторую метамодель, которая определяет абстрактный синтаксис и семантику понятий объектного моделирования на языке UML.

Нотация языка UML. Представляет собой графическую нотацию для визуального представления семантики языка UML.

Нотация включает в себя описание отдельных семантических элементов, которые могут применяться при построении диаграмм.

Формальное описание самого языка UML основывается на некоторой общей иерархической структуре модельных представлений, состоящей из четырех уровней:

1. Мета-метамодель
2. Метамодель
3. Модель
4. Объекты пользователя

"Нотация" - это то, что в других языках называют "синтаксисом". Само слово "нотация" подчеркивает, что UML - язык графический и модели (а точнее диаграммы) не "записывают", а рисуют.

В UML используется четыре вида элементов нотации:

1. Фигуры;
2. Линии;
3. Значки;
4. Надписи.

Фигуры используются "плоские" - прямоугольники, эллипсы, ромбы и т. д. Но есть одно исключение - на диаграмме развертывания для обозначения узлов инфраструктуры применяется "трехмерное" изображение параллелепипеда.

О линиях стоит сказать лишь то, что своими концами они должны соединяться с фигурами. На UML диаграммах вы не встретите линий, нарисованных "сами по себе" и не соединяющих фигуры. Применяется два типа линий - сплошная и пунктирная. Линии могут пересекаться, таких случаев следует по возможности избегать, но в этом нет ничего страшного.

UML предоставляет исключительную свободу - можно рисовать что угодно и как вздумается, лишь бы можно было понять смысл созданных диаграмм. В изображении фигур и значков тоже нет каких-то жёстких требований, и разработчики CASE-средств для UML-проектирования вовсю используют эту свободу, применяя различные стили рисования, заливку фигур цветом, тени и т. д.

# 4. Виды диаграмм UML

UML 1.5 определял двенадцать типов диаграмм, разделенных на три группы:

1. Четыре типа диаграмм представляют статическую структуру приложения;
2. Пять представляют поведенческие аспекты системы;
3. Три представляют физические аспекты функционирования системы (диаграммы реализации).

Версия UML 2.1 внесла не слишком много изменений. Диаграммы слегка изменились внешне, немного усовершенствовалась нотация, некоторые диаграммы получили новые наименования. Количество типов диаграмм для конкретной модели конкретного приложения не является строго фиксированным, поэтому рассмотрим самые основные виды диаграмм. Для простых приложений нет необходимости строить все без исключения диаграммы. Важно понимать, что перечень диаграмм зависит от специфики разрабатываемого проекта и определяется самим разработчиком.

Итак, мы кратко рассмотрим такие виды диаграмм, как:

* диаграмма прецедентов;
* диаграмма классов;
* диаграмма объектов;
* диаграмма последовательностей;
* диаграмма взаимодействия;
* диаграмма состояний;
* диаграмма активности;
* диаграмма развертывания.

# 5. Диаграмма прецедентов

Любые системы проектируются с учетом того, что в процессе своей работы они будут использоваться людьми и/или взаимодействовать с другими системами. Сущности, с которыми взаимодействует система в процессе своей работы, называются экторами, причем каждый эктор ожидает, что система будет вести себя строго определенным образом.

Эктор (actor) - это множество логически связанных ролей, исполняемых при взаимодействии с прецедентами или сущностями. Эктором может быть человек или другая система, подсистема или класс, которые представляют нечто вне сущности.

Графически эктор изображается либо "человечком", либо символом класса с соответствующим стереотипом. Обе формы представления имеют один и тот же смысл и могут использоваться в диаграммах

Эктор образовано от слова action, что в переводе означает действие. Дословный перевод слова "эктор" - действующее лицо.

Прецедент (use case) - описание множества последовательных событий, выполняемых системой, которые приводят к наблюдаемому эктором результату. Прецедент представляет поведение сущности, описывая взаимодействие между экторами и системой. Прецедент не показывает, "как" достигается некоторый результат, а только "что" именно выполняется.

Прецеденты обозначаются очень простым образом - в виде эллипса, внутри которого указано его название. Прецеденты и экторы соединяются с помощью линий. Часто на одном из концов линии изображают стрелку, причем направлена она к тому, у кого запрашивают сервис, другими словами, чьими услугами пользуются.

Прецеденты могут включать другие прецеденты, расширяться ими, наследоваться и т. д.

Диаграммы прецедентов относятся к той группе диаграмм, которые представляют динамические или поведенческие аспекты системы. Это отличное средство для достижения взаимопонимания между разработчиками, экспертами и конечными пользователями продукта. Как мы уже могли убедиться, такие диаграммы очень просты для понимания и могут восприниматься и, что немаловажно, обсуждаться людьми, не являющимися специалистами в области разработки ПО.

Цели создания диаграмм прецедентов:

* определение границы и контекста моделируемой предметной области на ранних этапах проектирования;
* формирование общих требований к поведению проектируемой системы;
* разработка концептуальной модели системы для ее последующей детализации;
* подготовка документации для взаимодействия с заказчиками и пользователями системы.

# 6. Диаграмма классов

Класс (class) - категория вещей, которые имеют общие атрибуты и операции. Классы - это строительные блоки любой объектно-ориентированной системы. Они представляют собой описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. При проектировании объектно-ориентированных систем диаграммы классов обязательны. Классы используются в процессе анализа предметной области для составления словаря предметной области разрабатываемой системы. Это могут быть как абстрактные 17 понятия предметной области, так и классы, на которые опирается разработка и которые описывают программные или аппаратные сущности. Диаграмма классов - это набор статических, декларативных элементов модели.

Диаграммы классов могут применяться и при прямом проектировании, то есть в процессе разработки новой системы, и при обратном проектировании - описании существующих и используемых систем. Информация с диаграммы классов напрямую отображается в исходный код приложения - в большинстве существующих инструментов UML-моделирования возможна кодогенерация для определенного языка программирования (обычно Java или C++). Таким образом, диаграмма классов - конечный результат проектирования и отправная точка процесса разработки.

# 7. Диаграмма объектов

Объект (object) - конкретная материализация абстракции; сущность с хорошо определенными границами, в которой инкапсулированы состояние и поведение; экземпляр класса (вернее, классификатора - эктор, класс или интерфейс).

Объект уникально идентифицируется значениями атрибутов, определяющими его состояние в данный момент времени. Как же обозначается объект в UML? А очень просто - объект, как и класс, обозначается прямоугольником, но его имя подчеркивается. Под словом имя здесь мы понимаем название объекта и наименование его класса, разделенные двоеточием. Для указания значений атрибутов объекта в его обозначении может быть предусмотрена специальная секция. Еще один нюанс состоит в том, что объект может быть анонимным: это нужно в том случае, если в данный момент не важно, какой именно объект данного класса принимает участие во взаимодействии.

Для чего нужны диаграммы объектов? Они показывают множество объектов - экземпляров классов (изображенных на диаграмме классов) и отношений между ними в некоторый момент времени. То есть диаграмма объектов - это своего рода снимок состояния системы в определенный момент времени, показывающий множество объектов, их состояния и отношения между ними в данный момент. Таким образом, диаграммы объектов представляют статический вид системы с точки зрения проектирования и процессов, являясь основой для сценариев, описываемых диаграммами взаимодействия. Диаграмма объектов используется для пояснения и детализации диаграмм взаимодействия, например, диаграмм последовательностей.

# 8. Диаграмма последовательностей

Диаграмма последовательностей относится к диаграммам взаимодействия UML, описывающим поведенческие аспекты системы, но рассматривает взаимодействие объектов во времени. Другими словами, диаграмма последовательностей отображает временные особенности передачи и приема сообщений объектами. Диаграммы последовательностей можно (и нужно!) использовать для уточнения диаграмм прецедентов, более детального описания логики сценариев использования. Это отличное средство документирования проекта с точки зрения сценариев использования! Диаграммы последовательностей обычно содержат объекты, которые взаимодействуют в рамках сценария, сообщения, которыми они обмениваются, и возвращаемые результаты, связанные с сообщениями. Впрочем, часто возвращаемые результаты обозначают лишь в том случае, если это не очевидно из контекста.

Теперь о том, какие обозначения используются на диаграмме последовательностей. Как и ранее, объекты обозначаются прямоугольниками с подчеркнутыми именами (чтобы отличить их от классов), сообщения (вызовы методов) - линиями со стрелками, возвращаемые результаты - пунктирными линиями со стрелками. Прямоугольники на вертикальных линиях под каждым из объектов показывают "время жизни" (фокус) объектов. Впрочем, довольно часто их не изображают на диаграмме, все это зависит от индивидуального стиля проектирования

# 9. ДИАГРАММА ВЗАИМОДЕЙСТВИЯ

Диаграмма взаимодействия показывает поток сообщений между объектами системы и основные ассоциации между ними и по сути является альтернативой диаграммы последовательностей. Диаграмма же взаимодействия, как и диаграмма последовательностей, показывает взаимодействие объектов во времени, т. е. в динамике.На обозначениях, применяемых на диаграмме взаимодействия,все стандартно: объекты обозначаются прямоугольниками с подчеркнутыми именами , ассоциации между объектами указываются в виде соединяющих их линий, над ними может быть изображена стрелка с указанием названия сообщения и его порядкового номера. Необходимость номера сообщения объясняется очень просто - в отличие от диаграммы последовательностей, время на диаграмме взаимодействия не показывается в виде отдельного измерения. Поэтому последовательность передачи сообщений можно указать только с помощью их нумерации.

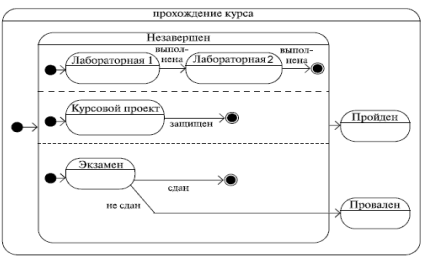
****

Как видите, эта диаграмма описывает (очень грубо) работу персонала библиотеки по обслуживанию клиентов: библиотекарь получает заказ от клиента, поручает сотруднику найти информацию по нужной клиенту книге, а после получения данных поручает еще одному сотруднику выдать книгу клиенту

# 10. Диаграмма состояний

Объекты характеризуются поведением и состоянием, в котором находятся. Например, человек может быть новорожденным, младенцем, ребенком, подростком или взрослым. Другими словами, объекты что-то делают и что-то "знают". Диаграммы состояний применяются для того, чтобы объяснить, каким образом работают сложные объекты. Состояние (state) - ситуация в жизненном цикле объекта, во время которой он удовлетворяет некоторому условию, выполняет определенную деятельность или ожидает какого-то события. Состояние объекта определяется значениями некоторых его атрибутов и присутствием или отсутствием связей с другими объектами. Диаграмма состояний показывает, как объект переходит из одного состояния в другое. Очевидно, что диаграммы состояний служат для моделирования динамических аспектов системы.

Диаграмма состояний полезна при моделировании жизненного цикла объекта . От других диаграмм диаграмма состояний отличается тем, что описывает процесс изменения состояний только одного экземпляра определенного класса - одного объекта, причем объекта реактивного, то есть объекта, поведение которого характеризуется его реакцией на внешние события. Понятие жизненного цикла применимо как раз к реактивным объектам, настоящее состояние (и поведение) которых обусловлено их прошлым состоянием. Но диаграммы состояний важны не только для описания динамики отдельного объекта. Они могут использоваться для конструирования исполняемых систем путем прямого и обратного проектирования. И они действительно с успехом применяются в таком качестве. Скругленные прямоугольники представляют состояния, через которые проходит объект в течение своего жизненного цикла. Стрелками показываются переходы между состояниями, которые вызваны выполнением методов описываемого диаграммой объекта. Существует также два вида псевдосостояний: начальное, в котором находится объект сразу после его создания (обозначается сплошным кружком), и конечное, которое объект не может покинуть, если перешел в него (обозначается кружком, обведенным окружностью).

****

Здесь мы видим составное состояние, включающее другие состояния, одно из которых содержит также параллельные подсостояния. Для того чтобы пройти курс, студент должен выполнить лабораторные работы, защитить курсовой проект и сдать экзамен.

# 11. Диаграмма активности

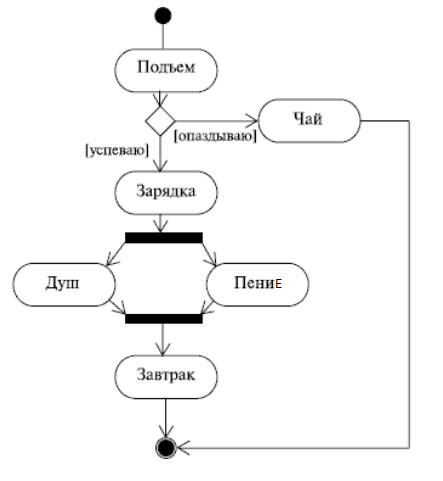
UML-диаграмма, на которой показаны действия. Она, как и диаграмма состояний, **отражает динамические аспекты поведения системы**. По существу, эта диаграмма представляет собой блок-схему, которая наглядно показывает, как поток управления переходит от одной деятельности к другой.

Активности на диаграмме “разбросаны” по беговым дорожкам, каждая из которых соответствует поведению одного из объектов (например, клиента, менеджера, веб-сервера, сервера БД и т.п.). Благодаря этому легко определить, каким из объектов выполняется каждая из активностей. Дорожка - часть области диаграммы деятельности, на которой отображаются только **те активности, за которые отвечает конкретный объект**. Предназначены дорожки для разбиения диаграммы в соответствии **с распределением ответственности за действия**. Имя дорожки может означать роль или объект, которому она соответствует.

Диаграммы деятельности состоят из ограниченного количества фигур, соединённых стрелками. Основные фигуры:

1. Прямоугольники с закруглениями — действия
2. Ромбы — решения
3. Широкие полосы — начало (разветвление) и окончание (схождение) ветвления действий
4. Чёрный круг — начало процесса (начальный узел)
5. Чёрный круг с обводкой — окончание процесса (финальный узел)

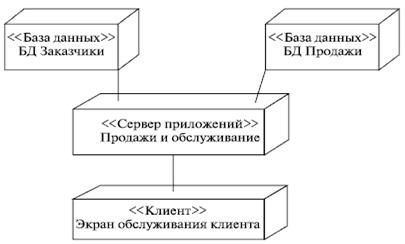
Стрелки идут от начала к концу процесса и показывают потоки управления или потоки объектов (данных).



# 12. Диаграмма развёртывания

Диаграмма развертывания (размещения) *– графическое представление инфраструктуры, на которую будет развернуто приложение*. Т*акие диаграммы есть смысл строить только для аппаратно-программных систем*.

Диаграмма развертывания показывает топологию системы и распределение компонентов системы по ее узлам, а также соединения - маршруты передачи информации между аппаратными узлами. Это единственная диаграмма, на которой применяются "трехмерные" обозначения: узлы системы обозначаются кубиками. Все остальные обозначения в UML - плоские фигуры.



На диаграммах развертывания можно обозначать компоненты системы, т. е. показывать их распределение по аппаратным узлам.

Диаграмма развертывания может помочь *рационально распределить компоненты системы по узлам сети*, от чего зависит производительность системы и ее безопасность. Пример диаграммы сетевой архитектуры (Network Architecture Diagram) – частный случай диаграммы развертывания.

# 13. Последовательность построения диаграмм

Прежде всего, вы должны ответить для себя на такие вопросы:

Какие именно виды диаграмм лучше всего отражают архитектуру системы и возможные технические риски, связанные с проектом?

Какие из диаграмм удобнее всего превратить в инструмент контроля над процессом (и прогрессом) разработки системы?

И еще одно - *никогда не выбрасывайте даже "забракованные" диаграммы*: они могут в дальнейшем оказаться полезными при анализе направления вашей мысли, поиске ошибок проектирования, да и просто для экспериментов по незначительному изменению системы.

В UML-проектировании важно уметь абстрагироваться от несущественных свойств системы. В этом плане очень полезными могут оказаться *коллективные упражнения на выявление и анализ прецедентов*.

Неплохой способ начать - *моделирование базовых абстракций или поведения одной из уже имеющихся у вас систем*.

Стройте *модели предметной области задачи в виде диаграммы классов*! Это хороший способ понять, как визуализировать множества взаимосвязанных абстракций.

Моделируйте *динамическую часть задачи с помощью простых диаграмм последовательностей и кооперации*. Хорошо начать с модели взаимодействия пользователя с системой - так вы сможете легко выделить наиболее важные прецеденты.

В итоге можно предложить такую последовательность построения диаграмм:

* диаграмма прецедентов,
* диаграмма классов,
* диаграмма объектов,
* диаграмма последовательностей,
* диаграмма кооперации,
* диаграмма состояний,
* диаграмма активности,
* диаграмма развертывания.

Конечно, это не единственная возможная последовательность и ее состав.

# 14. Отображение класса и его элементов на диаграмме UML

Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями:

* имя класса
* атрибуты (свойства) класса
* операции (методы) класса.

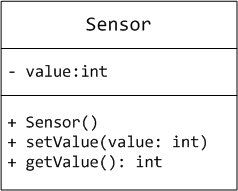
Для атрибутов и операций может быть указан один из трех типов видимости:

* **—** — private (частный)
* **#** — protected (защищенный)
* **+** — public (общий)

Видимость для полей и методов указывается в виде левого символа в строке с именем соответствующего элемента.

Каждый класс должен обладать именем, отличающим его от других классов. Имя класса может состоять из любого числа букв, цифр и знаков препинания (за исключением “:” и “.”) и может записываться в несколько строк.

На практике обычно используются краткие имена классов, взятые из словаря моделируемой системы и используют CamelCase.



**Атрибут** (свойство) – это именованное свойство класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Класс может иметь любое число атрибутов или не иметь ни одного. В последнем случае блок атрибутов оставляют пустым.

Для именования атрибута используются одно или несколько коротких существительных, выражающих некое свойство класса, к которому относится атрибут.

Можно уточнить спецификацию атрибута, указав его тип, кратность (если атрибут представляет собой массив некоторых значений) и начальное значение по умолчанию.

// Статические атрибуты класса обозначаются подчеркиванием.

**Операция** (метод) – это реализация метода класса. Класс может иметь любое число операций либо не иметь ни одной. Часто вызов операции объекта изменяет его атрибуты. Графически операции представлены в нижнем блоке описания класса.

Допускается указание только имён операций. Для именования операции используются короткие глагольные конструкции, описывающие некое поведение класса, которому принадлежит операция. Обычно каждое слово в имени операции пишется с заглавной буквы, за исключением первого, например move (переместить) или isEmpty (проверка на пустоту).

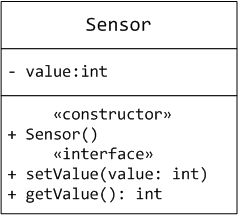
Можно специфицировать операцию, устанавливая ее сигнатуру, включающую имя, тип и значение по умолчанию всех параметров, а применительно к функциям – тип возвращаемого значения.

// Абстрактные методы класса обозначаются курсивным шрифтом.

// Статические методы класса обозначаются подчеркиванием.

Изображая класс, не обязательно показывать сразу все его атрибуты и операции. То есть для графического представления выбираются только некоторые из его атрибутов. Если помимо указанных существуют другие атрибуты и операции, вы даете это понять, завершая каждый список многоточием.

Чтобы легче воспринимать длинные списки атрибутов и операций, желательно снабдить префиксом (именем стереотипа) каждую категорию в них. В данном случае **стереотип** – это слово, заключенное в угловые кавычки, которое указывает то, что за ним следует.



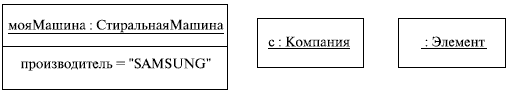
# 15.Способы использования объектов класса

Объект (object) - экземпляр класса.

· конкретная материализация абстракции;

· сущность с хорошо определенными границами, в которой инкапсулированы состояние и поведение;

· экземпляр класса (вернее, классификатора - эктор, класс или интерфейс). Объект уникально идентифицируется значениями атрибутов, определяющими его состояние в данный момент времени.



Пример изображения объекта

Диаграммы объектов представляют статический вид системы с точки зрения проектирования и процессов, являясь основой для сценариев, описываемых диаграммами взаимодействия. Говоря другими словами, диаграмма объектов используется для пояснения и детализации диаграмм взаимодействия, например, диаграмм последовательностей.

можно сказать, что объектные диаграммы используются для :

- Создания прототипа системы.

- Обратного инжиниринга.

- Моделирования сложных структур данных.

- Понимания системы с практической точки зрения.

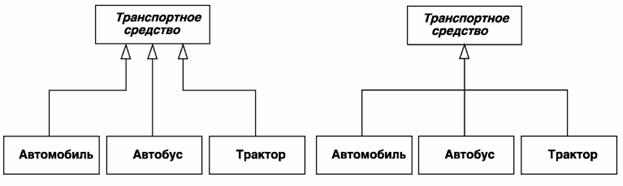
# 16. Моделирование наследования в UML

Отношения между классами в UML ориентированы на графическое отражение семантики и прагматики модели проектируемой системы до этапа реализации:

* отношение обобщения;
* отношение зависимости;
* отношение ассоциации;
* отношение «часть-целое» и др.

Отношение обобщения (наследования, генерализации) изображается линией с "незакрашенной" треугольной стрелкой на конце. Обобщение – отношение между предком (суперклассом) и потомком(подклассом), и стрелка всегда указывает на предка – в сторону того, от кого что-то требуют, чьими сервисами пользуются (рис1). Таким образом, экземпляры обобщенных прецедентов (потомков) сохраняют поведение, присущее обобщающему прецеденту (предку).

Объекты класса-потомка могут использоваться везде, где могут использоваться объекты класса-предка. Это свойство называют полиморфизмом по включению, имея в виду, что объекты потомка можно считать включаемыми во множество объектов класса-предка.



**рис.1 Альтернативные варианты изображения отношения обобщения классов**

В качестве **ограничений обобщения** могут быть использованы следующие ключевые слова языка UML:

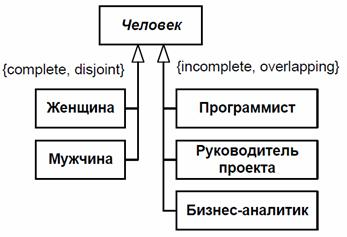
· **{complete}** – в данном отношении обобщения специфицированы все классы-потомки, и других потомков у данного класса-предка быть не может;

· **{incomplete}** – случай, противоположный {complete}, когда на диаграмме указаны не все классы-потомки, а в последующем возможно расширение их перечня, не изменяя уже построенную диаграмму;

· **{disjoint}** – классы-потомки не могут содержать объектов, одновременно являющихся экземплярами двух или более классов;

· **{overlapping**} ­– случай, противоположный {disjoint}, когда отдельные экземпляры классов-потомков могут принадлежать одновременно нескольким классам.

На диаграмме классов могут быть любые варианты ограничений обобщения (рис. 2).

****

**Рис 2. Пример вариантов ограничений обобщения**

# 17. Отношения между классами

На диаграммах классов UML могут быть представлены следующие отношения между классами: **зависимость, агрегация, ассоциация, реализация и наследование**.

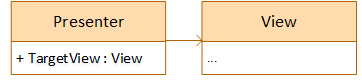
***Зависимость*** между объектами (или классами) – это такое отношение, при котором изменение одного объекта может повлечь за собой изменение другого объекта.

Отношение **зависимости** между классами на диаграммах UML отражается пунктирной линией со стрелкой, как показано на рисунке:



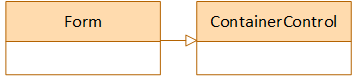
***Ассоциация*** между объектами показывает что один объект включает другой объект:

класс Presenter содержит ссылку на класс View

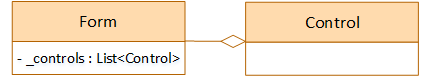


***Наследование*** (или обобщение) – показывает что один класс является подклассом другого, более общего класса.

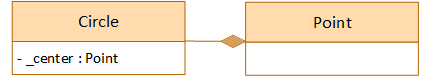
видно, что класс Form наследуется от класса ContainerControl:



***Агрегация*** – разновидность ассоциации при которой один объект является частью другого объекта как коллекция. Для примера реализации можно привести следующий рис:

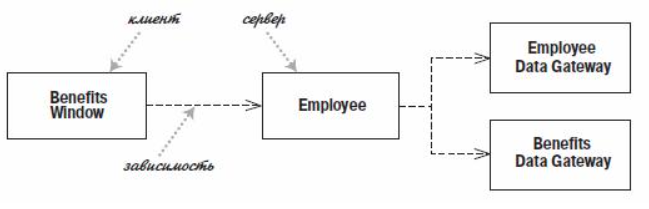


***Композиция*** – более строгий вариант агрегации. Если при агрегации возможно существование класса-контейнера без создания объектов коллекции, то при композиции невозможно существование класса-контейнера без создания экземпляра включенного класса.



# 18. Отношение зависимости между классами

Считается, что между двумя элементами диаграммы UML существует **зависимость** (dependency), если изменения одного элемента (**сервера**) могут вызвать изменения в другом элементе (**клиенте**). Это так называемая базовая зависимость, отражаемая линиями со стрелками (рис. 4.6).



**Рис. 4.6. Пример изображения зависимости**

В случае классов зависимости появляются по разным причинам:

o класс посылает сообщение другому классу;

o класс владеет другим классом как частью своих данных;

o класс использует другой класс в качестве параметра операции.

Если класс изменяет свой интерфейс, то сообщения, посылаемые этому классу, могут стать недействительными.

UML включает множество видов зависимостей, каждая с определенной семантикой и ключевыми словами*:*

**«call»** (вызывать) – источник вызывает операцию в цели;

**«create»** (создавать) – источник создает экземпляр цели;

**«derive»** (производить) – источник представляет собой производное цели;

**«instantiate»**(создать экземпляр) – источник является экземпляром цели (если источник является классом, то сам класс является экземпляром класса класс, то есть целевой класс – это метакласс);

**«permit»** (разрешать) – цель разрешает источнику доступ к ее закрытой функциональности;

**«realize»** (реализовать) – источник является реализацией спецификации или интерфейса, определенного целью;

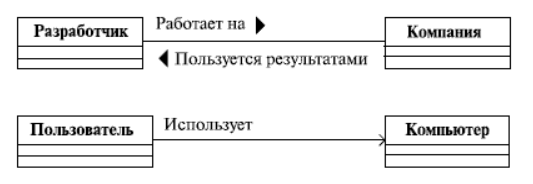
**«refine»** (уточнить) – конкретизация отношения между различными семантическими уровнями (например, источник может быть классом разработки, а цель – соответствующим классом анализа);

**«substitute»** (заместить) – источник может быть заменен целью;

**«trace»** (проследить) – используется, чтобы отследить такие моменты, как требования к классам или как изменения одной ссылки модели влияют на все остальное;

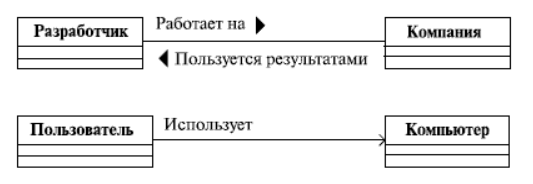
**«use»** (использовать) – для реализации источника требуется цель.

Некоторые типы зависимостей, такие как замещение, являются транзитивными, но в большинстве случаев существует разница между прямыми и обратными зависимостями (рис. 4.7).



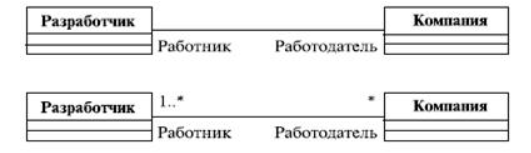
# 19. Отношение ассоциации между классами

Отношение **ассоциации –** это просто связь между объектами. Ассоциация может иметь имя, показывающее природу отношений между объектами, при этом в имени может указываться **направление**чтения связи при помощи треугольного маркера. Однонаправленная ассоциация может изображаться стрелкой (рис. 4.7):



**Рис. 4.7. Пример изображения ассоциаций**

Кроме направления ассоциации, можно указать на диаграмме **роли** класса в отношении, и **кратность *–*** количество объектов, связанных отношением (рис. 4.8):



**Рис. 4.8. Пример изображения роли и кратности**

Ассоциация может объединять три и более класса. В этом случае она называется **n-арной** и изображается ромбом на пересечении линий (рис. 4.9):



**Квалифицированная ассоциация** в языке UML эквивалентна таким понятиям, как ассоциативные массивы (associative arrays), проекции (maps), хеши (hashes) и словари (dictionaries). В случае представления ассоциации между классами **Order** (Заказ) и **Order Line** (Строка заказа) квалификатор **Product** указывает, что в соответствии с заказом для каждого экземпляра продукта может существовать только одна строка заказа.



**Рис. 4.10. Пример изображения квалифицированной ассоциации**

С точки зрения программной реализации квалифицированная ассоциация означает, что доступ к определенной строке заказа требует подстановки некоторого продукта в качестве аргумента для обращения к табличной структуре данных “ключ – значение”. Квалификатор – ключ (первичный или вторичный) отношения.

# 20. Композиция и агрегация классов

Особый вид ассоциаций между классами – связь типа **«часть-целое»**. При этом выделяют **агрегирование** и **композицию**.

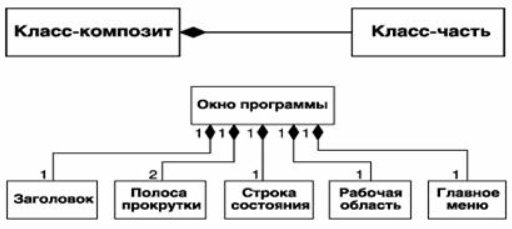
**Агрегирование** предполагает, что части, отделенные от целого, могут существовать независимо от него.



**Рис. 4.11. Примеры изображения агрегирования**

Жесткий диск можно вынуть из компьютера и установить в новый компьютер.

**Композиция** соответствует ситуации, когда целое владеет своими частями, их время жизни соответствует времени жизни целого, а независимо от целого части не существуют (рис. 4.12).



**Рис. 4.12.** **Примеры изображения композиции**

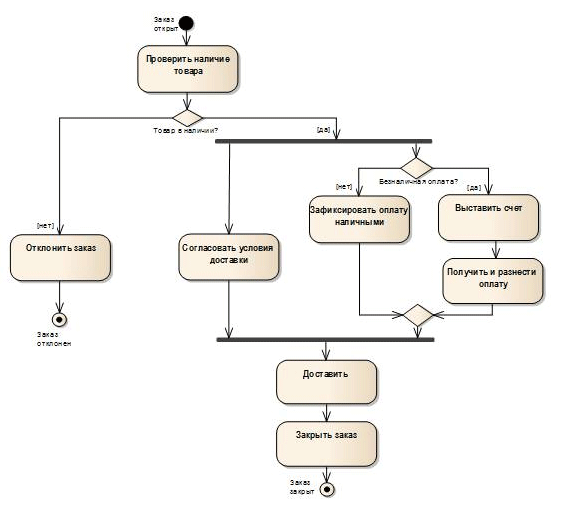
Заголовок окна не существует вне интервала существования окна.

**Агрегирование** и **композиция должны однозначно соответствовать понятию «часть-целое»**. Симметрия таких отношений не разрешена (рис. 4.13).



**Рис. 4.13. Примеры запрещенных связей (изображены красным цветом)**

# 21. Сравнение диаграмм активностей и блок-схем



Диаграммы деятельности напоминают блок-схемы. Блок-схемы реализуют концепцию конечного автомата, отражая поведение одного объекта. В отличие от блок-схем, диаграммы деятельности в UML 2.x допускают описание параллельных асинхронных процессов, используя в качестве формальной основы концепцию расширенных сетей Петри. В сетях Петри параллельные процессы с асинхронными взаимодействиями отражаются перемещениями так называемых маркеров между позициями в результате активной деятельности переходов. Понятие маркера в диаграммах деятельности UML 2.x – атрибут локализации элемента потока объектов и управления в отдельном узле.

# 22. Моделирование процессов диаграммами активности

В общем случае, моделируют процессы в несколько этапов, первым из которых является разбиение их на подпроцессы. Подпроцессы, являющиеся "участками большого процесса", описать легче. Дальше выделяют ключевые объекты (и создают для них дорожки), определяют предусловия и постусловия каждого процесса (т. е. его границы), описывают деятельности и переходы, отображают на диаграммах состояния ключевых объектов, в которые они переходят в ходе процесса.

Для того чтобы построить модель процесса, необходимо следующее:

1. Выделить какой-либо участок процесса. Проектируя непростые системы, невозможно отразить все представляющие интерес последовательности на одной диаграмме.

2. Выбрать объекты, на которые возложена ответственность высокого уровня за части всего процесса. Это могут быть реальные сущности, вошедшие в системный словарь, или более абстрактные объекты. В любом случае следует создать отдельную дорожку для каждого объекта.

3. Идентифицировать предусловия для начального состояния процесса и постусловия для его конечного состояния. Это поможет при моделировании границ процесса.

4. Начиная с исходного состояния описать деятельности и действия, выполняемые в различные моменты времени, а затем отразить их на диаграмме деятельности в виде состояний деятельности или действий.

5. Сложные действия или множества действий, встречающиеся многократно, следует свернуть в состояния деятельности и для каждого из таких состояний составить отдельную диаграмму деятельности.

6. Изобразить переходы, соединяющие состояния этих деятельностей и действий. Сначала нужно сосредоточиться на последовательных потоках, затем перейти к ветвлениям и в последнюю очередь рассмотреть разделения и слияния.

7. Если в процесс вовлечены важные объекты, изобразить их на диаграмме деятельности. В случае необходимости следует показать изменение значений и состояний таких объектов, чтобы прояснить суть траектории каждого.

# 23. Моделирование операций диаграммами активности

В этом случае диаграмма активностей превращается в "продвинутую" блок-схему, предоставляющую дополнительные возможности, например, отображение параллельно выполняющихся операций.

Моделирование операции состоит из следующих шагов:

1. Выявить абстракции, относящиеся к операции. Сюда относятся параметры операции (включая тип возвращаемого значения, если таковое имеется), атрибуты объемлющего класса и некоторых соседних классов.

2. Идентифицировать предусловия в начальном состоянии и постусловия в конечном состоянии операции. Следует идентифицировать также инварианты объемлющего класса, которые должны сохраняться во время выполнения операции.

3. Начиная с исходного состояния операции, специфицируйте деятельности и действия, протекающие во времени, и изобразите их на диаграмме деятельности в виде состояний деятельности или действий.

4. При необходимости используйте точки ветвления для описания условных переходов и итераций.

5. Лишь в том случае, если владельцем операции является активный класс, используйте точки разделения и слияния для описания параллельных потоков выполнения, если в этом возникает необходимость.

Если операция включает в себя взаимодействие сообщества объектов, ее реализацию можно моделировать с использованием коопераций.

# 24. Правила построения диаграммам активности

С одной стороны, на начальных этапах проектирования, когда детали реализации деятельностей в проектируемой системе неизвестны, построение диаграммы деятельности начинают с выделения под-деятельностей, которые в совокупности образуют деятельность подсистем. В последующем, по мере разработки диаграмм классов и состояний, эти под-деятельности уточняются в виде отдельных вложенных диаграмм деятельности компонентов подсистем, какими выступают классы и объекты.

С другой стороны, отдельные участки рабочего процесса в существующей системе могут быть хорошо отлаженными, и у разработчиков может возникнуть желание сохранить этот механизм выполнения действий в проектируемой системе. Тогда строится диаграмма деятельности для этих участков, отражающая конкретные особенности выполнения действий с использованием дорожек и объектов. В последующем такая диаграмма вкладывается в более общие диаграммы деятельности для подсистемы и системы в целом, сохраняя свой уровень детализации.

В случае типового проекта большинство деталей реализации действий могут быть известны заранее на основе анализа существующих систем или предшествующего опыта разработки систем-прототипов. Для этой ситуации доминирующим будет восходящий процесс разработки. Использование типовых решений может существенно сократить время разработки и избежать возможных ошибок при реализации проекта.

При разработке проекта новой системы, процесс функционирования которой основан на новых технологических решениях, ситуация представляется более сложной. А именно, до начала работы над проектом могут быть неизвестны не только детали реализации отдельных деятельностей, но и само содержание этих деятельностей становится предметом разработки. В данном случае доминирующим будет нисходящий процесс разработки от более общих схем к уточняющим их диаграммам. При этом достижение такого уровня детализации всех диаграмм, который достаточен для понимания особенностей реализации всех действий и деятельностей, может служить признаком завершения отдельных этапов работы над проектом.

# 25. Диаграмма кооперации

Цель кооперации состоит в том, чтобы специфицировать особенности реализации отдельных наиболее значимых операций в системе. Кооперация определяет структуру поведения системы в терминах взаимодействия участников этой кооперации.

Кооперация может быть представлена на двух уровнях:

* На уровне спецификации — показывает роли классификаторов и роли ассоциаций в рассматриваемом взаимодействии.
* На уровне примеров — указывает экземпляры и связи, образующие отдельные роли в кооперации.

Диаграмма кооперации уровня спецификации показывает роли, которые играют участвующие во взаимодействии элементы. Элементами кооперации на этом уровне являются классы и ассоциации, которые обозначают отдельные роли классификаторов и ассоциации между участниками кооперации.

Диаграмма кооперации уровня примеров представляется совокупностью объектов (экземпляры классов) и связей (экземпляры ассоциаций). При этом связи дополняются стрелками сообщений. На данном уровне показываются только релевантные объекты, т. е. имеющие непосредственное отношение к реализации операции или классификатора.

В кооперации уровня примеров определяются свойства, которые должны иметь экземпляры для того, чтобы участвовать в кооперации. Кроме свойств объектов на диаграмме кооперации также указываются ассоциации, которые должны иметь место между объектами кооперации. При этом вовсе не обязательно изображать все свойства или все ассоциации, поскольку на диаграмме кооперации присутствуют только роли классификаторов, но не сами классификаторы. Таким образом, в то время как классификатор требует полного описания всех своих экземпляров, роль классификатора требует описания только тех свойств и ассоциаций, которые необходимы для участия в отдельной кооперации.

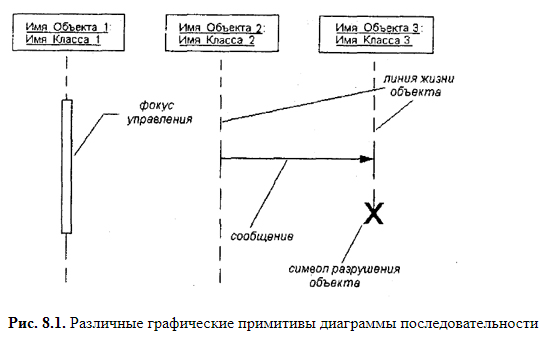
Отсюда вытекает важное следствие. Одна и та же совокупность объектов может участвовать в различных кооперациях. При этом, в зависимости от рассматриваемой кооперации, могут изменяться как свойства отдельных объектов, так и связи между ними. Именно это отличает диаграмму кооперации от диаграммы классов, на которой должны быть указаны все свойства и ассоциации между элементами диаграммы

# 26. Диаграмма последовательностей как диаграмма взаимодействия

Для моделирования взаимодействия объектов в языке UML используются соответствующие диаграммы взаимодействия. Говоря об этих диаграммах, имеют в виду два аспекта взаимодействия. Взаимодействия объектов можно рассматривать во времени, и тогда для представления временных особенностей передачи и приема сообщений между объектами используется диаграмма последовательности.

При изучении диаграмм состояния и деятельности, было отмечено одно немаловажное обстоятельство. Хотя рассмотренные диаграммы и используются для спецификации динамики поведения систем, время в явном виде в них не присутствует. Однако временной аспект поведения может иметь существенное значение при моделировании синхронных процессов, описывающих взаимодействия объектов. Именно для этой цели в языке UML используются диаграммы последовательности.

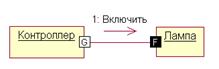
На диаграмме последовательности изображаются исключительно те объекты, которые непосредственно участвуют во взаимодействии и не показываются возможные статические ассоциации с другими объектами. Для диаграммы последовательности ключевым моментом является именно динамика взаимодействия объектов во времени



# 28. Диаграмма кооперации как диаграмма взаимодействий объектов

Диаграмма кооперации - диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения.

Главная особенность диаграммы кооперации заключается в возможности графически представить не только последовательность взаимодействия, но и все структурные отношения между объектами, участвующими в этом взаимодействии

.

Этот тип диаграмм позволяет описать взаимодействия объектов, абстрагируясь от последовательности передачи сообщений. На этом типе диаграмм в компактном виде отражаются все принимаемые и передаваемые сообщения конкретного объекта и типы этих сообщений.

**ЕСЛИ ЕСТЬ ВРЕМЯ МОЖНО ИЗ 25 ВОПРОСА ТРОХУ ВЗЯТЬ))00))**

# 29. Виды сообщений: синхронные и асинхронные, ответные, потерянные и найденные.

Взаимодействие между экземплярами моделируется через обмен сообщениями. Сообщения могут быть следующих видов:

-  – синхронное сообщение (англ. synchronous message). Клиент посылает сообщение серверу и ждет, пока тот примет и обработает сообщение. Как правило, один объект передает синхронное сообщение второму, второй – третьему и т.д., образуя вложенный поток сообщений. В любом случае клиент, инициирующий поток сообщений, должен дождаться его завершения, т.е. возврата управления. Это самый распространенный тип сообщений;

-  – асинхронное сообщение (англ. asynchronous message). Клиент посылает сообщение серверу и, не дожидаясь ответа, продолжает выполнять следующие операции;

- – возвращающее сообщение (англ. reply message), обозначающее возврат значения или управления от сервера обратно клиенту. Стрелки этого вида зачастую отсутствуют на диаграммах, поскольку неявно предполагается их существование после окончания процесса выполнения операции.

Возможны случаи, когда известен адресат сообщения, но неизвестен его отправитель. С примерами таких сообщений (в бумажном виде) в советские времена довольно часто встречались секретари госучреждений. Такие сообщения называют найденными. Или обратный случай: отправитель известен, а получатель - нет потерянными.

# 30. Уровни экземпляров и спецификации в диаграммах кооперации.

Диаграмма кооперации уровня спецификации

Кооперация на уровне спецификации изображается на диаграмме пунктирным эллипсом, внутри которого записывается имя этой кооперации (рис. 9.1). Такое представление кооперации относится к отдельному варианту использования и детализирует особенности его последующей реализации. Символ эллипса кооперации соединяется отрезками пунктирной линии с каждым из участников этой кооперации, в качестве которых могут выступать объекты или классы. Каждая из этих пунктирных линий помечается ролью (role) участника. Роли соответствуют именам элементов в контексте всей кооперации. Эти имена трактуются как параметры, которые ограничивают спецификацию элементов при любом их появлении в отдельных представлениях модели.



**Рис. 9.1.** Общее представление кооперации на диаграммах уровня спецификации

Простой класс на диаграмме кооперации обозначается прямоугольником класса, внутри которого записывается строка текста. Эта строка текста называется ролью классификатора (classifier role). Роль классификатора показывает особенность использования объектов данного класса. Обычно в прямоугольнике показывается только секция имени класса, хотя не исключается возможность указания секций атрибутов и операций.

# 31. Мультиобъекты, композитные и активные объекты в диаграммах кооперации.

***Мультиобъект*** - набор объектов, изображается как два прямоугольника, смещенных по отношению друг к другу, что создает впечатление "колоды карт".Сообщение, отправленное мультиобъекту, означает сообщение к набору объектов, например, операция выбора - поиска определенного объекта.

***Композитный объект*** - это высокоуровневый объект, состоящий из нескольких частей-объектов. Это экземпляр композитного класса, реализующего композитное агрегирование класса и его частей. Это конструкция, показывающая целое в виде взаимодействующих частей, но в основном с точки зрения композиции. Изображается композитный объект в виде прямоугольного символа объекта, но с некоторыми отличиями:

* имя объекта указывается в верхней части прямоугольника, отделенной от его остальной части горизонтальной линией;
* в нижней части прямоугольника размещаются части композитного объекта, также, естественно, изображаемые символами объектов;
* части композитного объекта могут (и даже должны) быть связаны между собой;
* допускается ситуация, когда некоторые части композитного объекта сами являются композитными объектами.

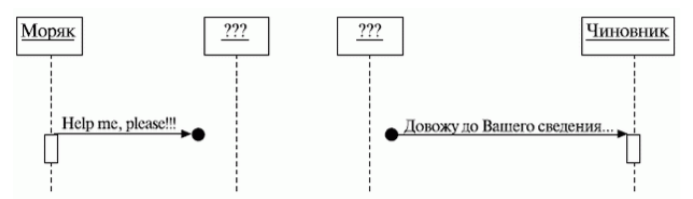
**Активными** называют объекты, которые владеют собственным потоком управления и могут инициировать выполнение действий. Выглядит на диаграмме как прямоугольный символ объекта, но с утолщенными границами. Часто активный объект изображается как композитный объект, содержащий объекты-части.

# 

# 32. Диаграммы взаимодействия с разветвленным потоком управления.

***Диаграмма взаимодействия*** - это диаграмма, на которой представлено взаимодействие, состоящее из множества объектов и отношений между ними, включая и сообщения, которыми они обмениваются. Этот термин применяется к видам диаграмм с акцентом на взаимодействии объектов (диаграммах кооперации, последовательности и деятельности).

Существуют различные типы сообщений: синхронные, асинхронные и ответные, потерянные и найденные. ***Синхронные*** сообщения приостанавливают поток выполнения до тех пор, пока не будет получен ответ. Синхронные сообщения изображаются сплошной линией с треугольной закрашенной стрелкой на конце. ***Асинхронные*** сообщения не ждут ответа, не приостанавливают поток выполнения - сразу после их посылки происходит немедленный переход к следующему шагу, и последовательность продолжается. Асинхронные сообщения изображаются сплошной линией с обычной (составленной из двух отрезков) стрелкой на конце. Возможны случаи, когда нам известен адресат сообщения, но неизвестен его отправитель. Или обратный случай: отправитель известен, а получатель - нет. Такие сообщения называют ***потерянными*** или ***найденными***. Пример:



# 33. Нефункциональные требования и их отображение на диаграммах прецедентов.

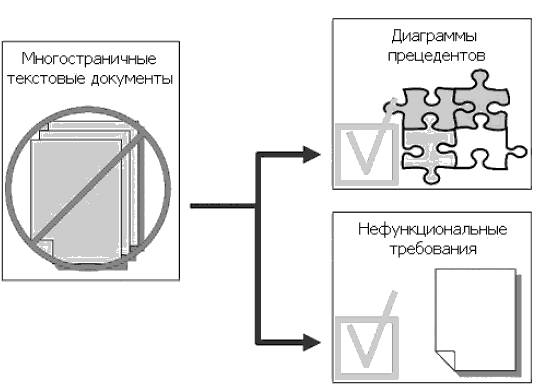
Требование - это желаемая функциональность, свойство или поведение системы. Именно со сбора требований начинается процесс разработки ПО. Если изобразить процесс разработки ПО в виде "черного ящика", на выходе которого мы получаем программный продукт, то на вход этого "черного ящика" будет подаваться именно набор требований к программному продукту (рис. 1).



* диаграммы прецедентов;
* нефункциональные требования

. **Диаграммы прецедентов** составляют модель прецедентов (вариантов использования, use-cases). Прецедент - это функциональность системы, позволяющая пользователю получить некий значимый для него, ощутимый и измеримый результат. Каждый прецедент соответствует отдельному сервису, предоставляемому моделируемой системой в ответ на запрос пользователя, т. е. определяет способ использования этой системы. Именно по этой причине use cases, или прецеденты, часто в русской терминологии фигурируют как варианты использования. Варианты использования чаще всего применяются для спецификации внешних требований к проектируемой системе или для спецификации функционального поведения уже существующей системы. Кроме этого, варианты использования неявно описывают типичные способы взаимодействия пользователя с системой, позволяющие корректно работать с предоставляемыми системой сервисами.

**Нефункциональные требования** - это описание таких свойств системы, как особенности среды и реализации, производительность, расширяемость, надежность и т. д. Часто нефункциональные требования не привязаны к конкретному варианту использования и потому выносятся в отдельный список дополнительных требований к системе (рис. 6.2).



Идентифицировать прецеденты и действующие лица - обязанность системного аналитика. И делает он это для того, чтобы:

- четко разграничить систему и ее окружение;

- определить, какие действующие лица и как именно взаимодействуют с системой, какой функционал (варианты использования) ожидается от системы;

- определить и описать в словаре предметной области (глоссарии) общие понятия, которые необходимы для детального описания функционала системы (прецедентов).

Подобный [вид деятельности](https://pandia.ru/text/category/vidi_deyatelmznosti/) обычно выполняется в такой последовательности:

1. Определение действующих лиц.

2. Определение прецедентов.

3. Составление описания каждого прецедента.

4. Описание модели прецедентов в целом (этот этап включает в себя создание словаря предметной области).

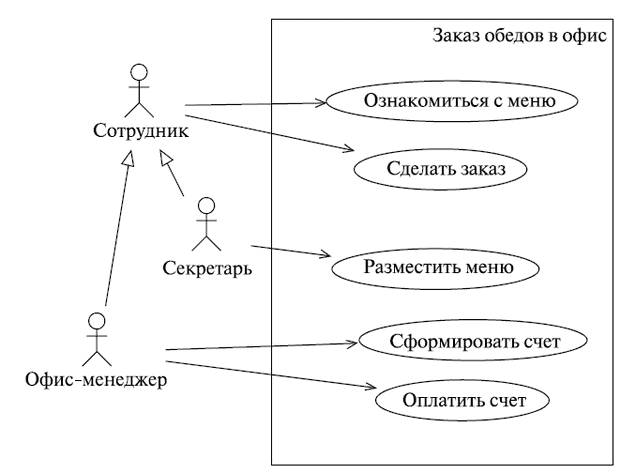
Вначале требования оформляются в виде обычного текстового документа, который создается или самим пользователем, или пользователем и разработчиком вместе. Далее требования оформляют в виде таблицы. В левую колонку помещают прецеденты, а в правую - действующих лиц, участвующих в прецеденте.

Рассмотрим пример.(**думаю, что нахрен не нужно, но пусть будет**) Секретарь размещает на сервере меню обеденных блюд на неделю. Сотрудники должны иметь возможность ознакомиться с меню и сделать заказ, выбрав блюда на каждый день следующей недели. Офис-менеджер должен иметь возможность сформировать счет и оплатить его. Система должна быть написана на . Такое вот нехитрое интернет-приложение для автоматизации заказов обедов в офис.

Думаем, здесь все понятно. Таблица с описанием требований может быть, например, такой:

|  |  |
| --- | --- |
| **Прецедент** | **Действующее лицо** |
| разместить меню | секретарь |
| ознакомиться с меню | сотрудник, секретарь, офис-менеджер |
| сделать заказ | сотрудник, секретарь, офис-менеджер |
| сформировать счет | офис-менеджер |
| оплатить счет | офис-менеджер |

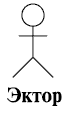
Здесь нигде не сказано о том, что система должна быть написана на . Почему - понятно: это ведь нефункциональное требование! И еще, очевидно, что секретарь и офис-менеджер тоже являются сотрудниками. Читатель, внимательно прочитавший предыдущие лекции, заподозрит, что в данном случае, создавая модель прецедентов, говоря о действующих лицах, можно бы применить генерализацию. Действительно, диаграмма прецедентов, построенная на основе этой таблицы, может быть, например, такой (рис. 3):



# 34. Понятие эктора и отношения между экторами

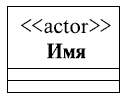
Действующее лицо (actor) – это роль, которую пользователь играет по отношению к системе. Действующие лица представляют собой роли, а не конкретных людей или наименования работ. Несмотря на то, что на диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок, действующее лицо может также быть внешней системой, которой необходима некоторая информация от данной системы. Показывать на диаграмме действующих лиц следует только в том случае, когда им действительно необходимы некоторые варианты использования. Действующие лица делятся на три основных типа – пользователи системы, другие системы, взаимодействующие с данной, и время. Время становится действующим лицом, если от него зависит запуск каких-либо событий в системе.

Экторы не могут быть связаны друг с другом.



**Рис. 6.4.**

Кстати, человечек ( "stick-person" ) - это не единственное обозначение эктора, используемое в UML. На диаграммах прецедентов обычно применяется именно "человекоподобная" форма эктора, но на других диаграммах, и особенно в случаях, когда эктор имеет атрибуты, которые важно показать, используется изображение эктора как класса со стереотипом <<actor>> (рис. 6.5):



С системой экторы общаются через сообщения, но если говорить на более высоком уровне абстракции, в терминах модели прецедентов, то взаимодействуют они с системой через прецеденты. Один и тот же эктор может быть связан с несколькими прецедентами, и наоборот, один прецедент может быть связан с несколькими разными экторами. Ассоциации между эктором и прецедентом всегда бинарные - т. е. представляют отношения типа "один к одному", использование кратности недопустимо. Это не противоречит сказанному выше: действительно, один эктор может быть связан с несколькими прецедентами, но только с помощью отдельных ассоциаций - по одной на каждый прецедент. Ассоциация, изображенная стрелкой - это направленная ассоциация и стрелка (как и на других диаграммах) всегда направлена в сторону той сущности, от которой что-то требуют, чьим сервисом пользуются и т. д. Единственное допустимое отношение между экторами - генерализация (наследование).

# 35. Отношения включения и расширения между экторами

Связь включения применяется в тех ситуациях, когда имеется какой-либо фрагмент поведения системы, который повторяется более чем в одном варианте использования. С помощью таких связей обычно моделируют многократно используемую функциональность. В примере АТМ варианты использования «Снять деньги» и «Положить деньги на счет» должны опознать (аутентифицировать) клиента и его идентификационный номер перед тем, как допустить осуществление самой транзакции. Вместо того чтобы подробно описывать процесс аутентификации для каждого из них, можно поместить эту функциональность в свой собственный вариант использования под названием «Аутентифицировать клиента». Связь расширения применяется при описании изменений в нормальном поведении системы. Она позволяет варианту использования только при необходимости использовать функциональные возможности другого.

А как же изображается включение? Да очень просто - как зависимость (пунктирная линия со стрелкой) со стереотипом <<include>>. При этом стрелка направлена, естественно, в сторону включаемого прецедента.

Расширение дополняет прецедент другими прецедентами, "срабатывающими" при некоторых условиях, - просто добавляет в исходный прецедент последовательность действий, содержащуюся в другом прецеденте. Отношение расширения прецедента А к прецеденту В означает, что экземпляр прецедента В может включать в себя (при определенных условиях, которые могут быть описаны в расширении) прецедент A.

# 36. Причины использования прецедентов.

Модель прецедентов, по сути, является концептуальной моделью системы. В ней в общих чертах описывается только поведение (функциональность) системы, о деталях реализации речь не идет - на данном этапе реализация не важна, гораздо важнее собрать требования к системе и оформить их в наглядном виде, понятном и разработчикам, и заказчику.

Т.о., мы можем сформулировать три причины использования прецедентов. Или, три способа использования прецедентов в ходе работы над системой:

1) Прецеденты дают возможность аналитикам, пользователям и разработчикам говорить на одном языке: используя прецеденты, аналитики (эксперты в предметной области) могут на основе пожеланий заказчика описать поведение системы с точки зрения пользователя с такой степенью детализации, что разработчики смогут без труда сконструировать "внутренности" системы. В то же время, нотация диаграмм прецедентов настолько проста, что даже неподготовленный пользователь (заказчик) способен понять их смысл и помочь в их уточнении.

2)Прецеденты позволяют разработчикам понять назначение элемента: система, подсистема или даже класс могут быть сложными образованиями, состоящими из большого числа составных частей и имеющими большое число атрибутов и операций. Моделирование прецедентов позволяет лучше представить себе поведение системы, понять, какие элементы модели играют какие роли в реализации этого поведения, в какие кооперации входят, и какой именно прецедент (функционал системы) реализуют.

3)Прецеденты являются основой для тестирования элемента в течение всей разработки: модель прецедентов описывает желаемое поведение системы (ее функционал) с точки зрения пользователя. Так что, постоянно сопоставляя предоставляемый элементом (фактический) функционал с имеющимися прецедентами, можно надежно контролировать корректность реализации элемента.

# **37. Прецеденты в прямом и обратном проектировании**

Прецеденты полезны и для прямого, и для обратного проектирования. При прямом проектировании мы, по сути, осуществляем "перевод" с UML на некий язык программирования. И тестировать созданное приложение следует, основываясь именно на потоках событий, описываемых прецедентами. Обратное проектирование предполагает перевод с языка программирования на язык UML-диаграмм. Такими вещами приходится заниматься в силу ряда причин:

1)С целью поиска ошибок и чтобы убедиться в адекватности дизайна. Хорошо после первого перевода с UML на язык программирования сделать обратный перевод и сравнить исходные и восстановленные UML-модели (желательно, чтобы эти переводы выполнялись разными командами). Такой подход называется обратной семантической трассировкой.

2)Когда отсутствует документация: иногда стоит задача модификации существующей системы, код которой плохо документирован. В таком случае перевод с языка программирования на язык UML-диаграмм - способ понять назначение системы и ее частей, функционал, предоставляемый ею, и т. д.

# 38. Обзор CASE-средств для построения диаграмм UML

UML - отличное средство моделирования, но, как уже говорилось выше, строить диаграммы на бумаге - не всегда удобно, хотя бы по причине сложностей с редактированием, распространением и т. д. Чтобы облегчить труд проектировщика, были созданы CASE-средства - программы специального вида. CASE-средства помогут вам построить профессионально выглядящие диаграммы.

CASE-средства (от Computer Aided Software/System Engineering) - позволяют проектировать любые системы на компьютере. Необходимый элемент системного и структурнофункционального анализа, CASE-средства позволяют моделировать бизнес-процессы, базы данных, компоненты программного обеспечения, деятельность и структуру организаций. Применимы практически во всех сферах деятельности. Результат использования CASE-средств - оптимизация систем, снижение расходов, повышение эффективности, снижение вероятности ошибок.

Существует немало подобных программ. Рассмотрим некоторые из них:

1. **IBM Rational Rose**

Rational Rose - современное и мощное средство анализа, моделирования и разработки программных систем. Rational Rose пригодится при решении практически любых задач проектирования информационных систем: от анализа бизнес-процессов до кодогенерации на определенном языке программирования. Есть следующие его версии:

/Rational Rose Modeler.

/Rational Rose Professional

/Rational Rose RealTime

/Rational Rose Enterprise

/Rational Rose DataModeler

В зависимости от поставки, в Rational Rose может быть расширен или сужен набор визуальных компонент (возможных диаграмм). Впрочем, Rational Rose и так достаточно функционален. Вот основные возможности продукта:

* /прямое и обратное проектирование на языках: ADA, Java, С, C++, Basic;
* /поддержка технологий COM, DDL, XML;
* /возможность генерации схем БД Oracle и SQL.

1. **Borland Together**

Borland Together ControlCenter - это интегрированная платформа разработки, позволяющая упростить и ускорить анализ, дизайн, разработку и развертывание комплексных корпоративных приложений. Эти возможности сочетаются в одном интегрированном решении с поддержкой UML, помогающем командно разрабатывать высококачественные системы быстрее и эффективнее.

Вот некоторые особенности Borland Together:

* /Поддержка XP ("экстремальное программирование").
* /Ускорение процессов разработки путем применения паттернов.
* /Развертывание на несколько серверов приложений выполняется быстро, без перекодирования.
* /Функция контроля качества облегчает жизнь разработчиков
* /Эффективнейший аудит и поддержка метрик качества разработки ПО позволяет команде контролировать качество продукта в ходе разработки.

Встроенное функциональное тестирование помогает обнаружить проблемы еще в процессе разработки, что действительно очень важно, поскольку стоимость исправления ошибок тем выше, чем позже они обнаружены.

1. **Microsoft Visio**

Visio - решение для построения диаграмм от Microsoft. Этот пакет из семейства Microsoft Office предназначен исключительно для

рисования диаграмм. Возможности Visio можно легко расширять, используя новые шаблоны бизнес-диаграмм. В Visio можно прототипировать интерфейс приложений с помощью встроенных шаблонов пользовательского интерфейса Microsoft Windows XP, что позволяет создавать модель пользовательского интерфейса в стандартном Windows XP-стиле. Можно легко рисовать диаграммы сетевых ресурсов, иллюстрирующие развертывание

нового ПО на существующие сетевые ресурсы.

Возможности Visio:

* /Документирование и анализ бизнес-процессов
* /Отслеживание комментариев членов команды
* /Поддержка Tablet PC
* /Создание календарей
* /Простое создание и использование технических диаграмм
* /Более быстрое создание и редактирование диаграмм
* /Visio поддерживает множество локальных языков
* /Отличная интеграция с другими приложениями MS Office

# 39. Критерии выделения прецедентов.

Кстати, вернемся к требованиям. Да, мы сказали, что на вход нашего "черного ящика" подается набор требований. Но в какой форме? Как их документируют, эти требования? Думаю, большинство читателей помнит, что такое техническое задание - основной документ, без составления которого не начинался в советские времена ни один проект. Документ это был большой, многостраничный, с четкой структурой, определяемой ГОСТами (государственными отраслевыми стандартами). И описывал он, по сути, не что иное, как требования к создаваемой системе!

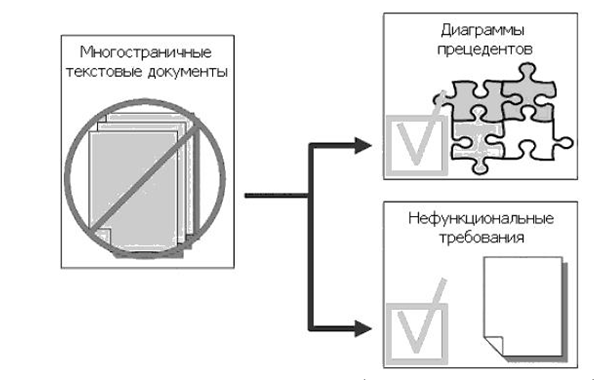
Техническое задание - вещь по-своему хорошая. Но время шло, менялись стандарты, нотации, способы описания требований. И вот постепенно техническое задание уступило место набору артефактов, состоящему из документов двух видов:

диаграммы прецедентов;

нефункциональные требования.

Диаграммы прецедентов составляют модель прецедентов(вариантов

использования, use-cases). Прецедент - это функциональность системы, позволяющая пользователю получить некий значимый для него, ощутимый и измеримый результат. Каждый прецедент соответствует отдельному сервису, предоставляемому моделируемой системой в ответ на запрос пользователя, т. е. определяет способ использования этой системы. Именно по этой причине use cases, или прецеденты, часто в русской терминологии фигурируют как варианты использования. Варианты использования чаще всего применяются для спецификации внешних требований к проектируемой системе или для спецификации функционального поведения уже существующей системы. Кроме этого, варианты использования неявно описывают типичные способы взаимодействия пользователя с системой, позволяющие корректно работать с предоставляемыми системой сервисами.

Нефункциональные требования - это описание таких свойств системы, как особенности среды и реализации, производительность, расширяемость, надежность и т. д. Часто нефункциональные требования не привязаны к конкретному варианту использования и потому выносятся в отдельный список дополнительных требований к системе (рис. 6.2).

Но вернемся же к прецедентам (вариантам использования). Идентифицировать прецеденты и действующие лица - обязанность системного аналитика. И делает он это для того, чтобы:

* четко разграничить систему и ее окружение;
* определить, какие действующие лица и как именно взаимодействуют с системой, какой функционал (варианты использования) ожидается от системы;
* определить и описать в словаре предметной области (глоссарии) общие понятия, которые необходимы для детального описания функционала системы (прецедентов).

Подобный вид деятельности обычно выполняется в такой последовательности:

* Определение действующих лиц.
* Определение прецедентов.
* Составление описания каждого прецедента.

Описание модели прецедентов в целом (этот этап включает в себя создание словаря предметной области).

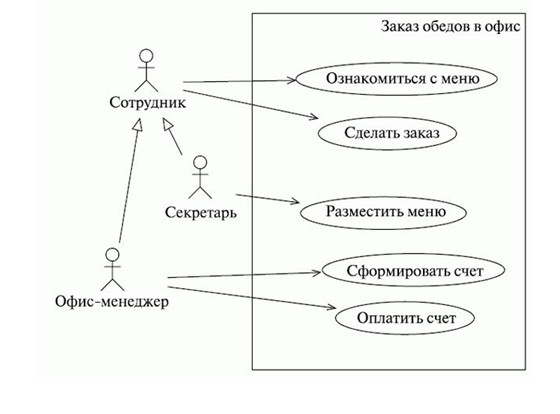
Вначале требования оформляются в виде обычного текстового документа, который создается или самим пользователем, или пользователем и разработчиком вместе. Далее требования оформляют в виде таблицы. В левую колонку помещают прецеденты, а в правую - действующих лиц, участвующих в прецеденте.

Рассмотрим пример. Секретарь размещает на сервере меню обеденных блюд на неделю. Сотрудники должны иметь возможность ознакомиться с меню и сделать заказ, выбрав блюда на каждый день следующей недели. Офис-менеджер должен иметь возможность сформировать счет и оплатить его. Система должна быть написана на ASP.NET. Такое вот нехитрое интернет-приложение для автоматизации заказов обедов в офис.

Думаем, здесь все понятно. Таблица с описанием требований может быть, например, такой:



Здесь нигде не сказано о том, что система должна быть написана на ASP.NET. Почему - понятно: это ведь нефункциональное требование! И еще, очевидно, что секретарь и офис-менеджер тоже являются сотрудниками. Читатель, внимательно прочитавший предыдущие лекции, заподозрит, что в данном случае, создавая модель прецедентов, говоря о действующих лицах, можно бы применить генерализацию. Действительно, диаграмма прецедентов, построенная на основе этой таблицы, может быть, например, такой (рис. 6.3):



# 40. Понятие шаблона проектирования

**Шаблон проектирования**(*design pattern)* – это многократно применяемая архитектурная конструкция, предоставляющая решение общей проблемы проектирования в рамках конкретного контекста и описывающая значимость этого решения.

Шаблон проектирования (паттерн) не является законченным образцом проекта, который может быть прямо преобразован в код. Это описание или образец того, как решить задачу таким образом, чтобы это можно было использовать в различных ситуациях. Шаблоны проектирования зачастую объектно-ориентированы. Они показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

В отличие от шаблонов проектирования, шаблоны функций и классов С++ непосредственно преобразуются в код на этапе компиляции. [Алгоритм](http://ru.wikipedia.org/w/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC)ыпо своей сути также являются шаблонами, но не проектирования, а вычисления, так как решают вычислительные задачи. В отличие же от идиом, шаблоны проектирования независимы от применяемого языка программирования.

**Идиома** – шаблон решения некоторой задачи, записи [алгоритма](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) или [структуры данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) путём комбинирования встроенных элементов языка.

Идиому можно считать самым низкоуровневым шаблоном проектирования, предписывающим конкретный способ реализации определённых деталей и отношений между ними средствами конкретного языка. Одна и та же идиома может выглядеть в двух разных языках по-разному, либо в ней может не быть надобности в одном из них..

Пример 1 идиомы C/C++:

|  |
| --- |
| **for** (;;) { **// Организация бесконечного цикла**  do\_something();  } |

Пример 2 идиомы C++:

|  |
| --- |
| **class nonaddressable { // Класс с запретом определения адреса объектов**  **public**:  **typedef** double useless\_type;  **private**:  useless\_type **operator**&() **const**;  **};**    **void f1() {**  **nonaddressable na;**  **nonaddressable \* naptr = &na;** ***// Compiler error here.***  **}**    **template** <**class T**> **T \* addressof(T & v)** **{** ***// Получение адреса объекта любого класса***  **return reinterpret\_cast**<T \*>(& **const\_cast**<char&>(**reinterpret\_cast**<**const volatile** char &>(v)));  **}**    **void f2() {**  **nonaddressable na;**  **nonaddressable \* naptr = addressof(na);** ***// No more compiler error.***  **}** |

Идиома может представлять собой воспроизведение в языке элементов семантически иного языка, которые в данном языке тоже могут быть применимы, но не провоцируются самим языком (не входят в число его идиом)..

|  |
| --- |
| **class** **MakeFinal** { // Запрет создания производных классов  MakeFinal() {} // private by default*.*  **friend** **class** **sealed**;  };    **class** **sealed** : **virtual** MakeFinal { };    **class** **Test** : **public** sealed { };    void main() {  Test t; *// Compilation error here.*  } |
| **class** **Base** **final** { }; // Запрет создания производных классов C++11    **class** **Test** : **public** Base { }; *// incorrect* |

С этой позиции многие шаблоны в объектно-ориентированном проектировании рассматриваются как идиоматическое воспроизведение *элементов* функциональных языков.

[**https://en.wikibooks.org/wiki/More\_C%2B%2B\_Idioms**](https://en.wikibooks.org/wiki/More_C++_Idioms)

[**http://qaru.site/questions/528/what-are-your-favorite-c-coding-style-idioms**](http://qaru.site/questions/528/what-are-your-favorite-c-coding-style-idioms)

Однако языки программирования не являются единственными источниками определения понятия шаблона проектирования. Реальный источник идей выделения шаблона – практика проектирования реальных систем (особенно операционных систем и инфраструктуры создания корпоративных систем).

Важнейший современный подход к проектированию систем – **проектирование на основе обязанностей** (responsibility driven design, RDD) (“Разделяй и властвуй”). В UML *обязанность* (responsibility) определяется как "контракт классификатора" и описывает поведение объекта в терминах его ролей. Диаграммы взаимодействий UML отражают распределение обязанностей между объектами. Обязанности отображаются на сообщения, отправляемые объектам различных классов.

В общем случае в RDD выделяют **два вида обязанностей: з*нание* (knowing) и *действие* (doing).**

Обязанности, относящиеся к ***знаниям***объекта:

1) наличие информации о закрытых инкапсулированных данных;

2) наличие информации о связанных объектах;

3) наличие информации о следствиях или вычисляемых величинах.

Обязанности, относящиеся к ***действиям***объекта:

1) выполнение некоторых действий самим объектом, например, создание эк­земпляра или выполнение вычислений;

2) инициирование действий других объектов;

3) управление действиями других объектов и их координирование.

Обязанности назначаются объектам в процессе проектирования. Например, можно сказать, что объект Sale отвечает за создание экземпляра SalesLineltems (действие) или что объект Sale отвечает за наличие информации о стоимости покупки (знание)(рис. 1.1).

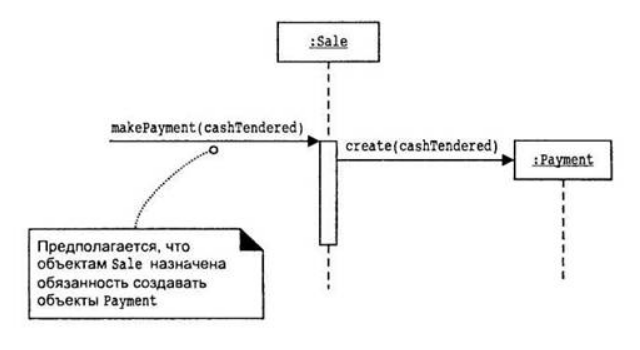


Рис. 1.1. Связь методов и обязанностей

Обязанно­сти, относящиеся к разряду "знаний", вытекают из модели предметной области, поскольку модель иллюстрирует атрибуты и ассоциации.

Возможность реализации обязанностей классами и методами зависит от подробности модели. Например, реализация обязанности "обеспечения доступа к реляционным базам данных" может потребовать создания десятков классов и сотен методов, а для реализации обязанности "создания экземпляра объекта Sale" достаточно одного или нескольких методов.

Методы и обязанности классов не эквивалентны. Обязанности реализуются посредством методов, действующих либо отдель­но, либо во взаимодействии с другими методами и объектами. Например, для класса Sale можно определить один или несколько методов вычисления стоимо­сти (например, метод getTotal). Для выполнения этой обязанности объект Sale должен взаимодействовать с другими объектами, в том числе передавать сооб­щения getSubtotal каждому объекту SalesLineltem о необходимости предос­тавления соответствующей информации.

На рис. 1.1 показано, что обязанностью объектов Sale является создание эк­земпляров Payment. Для выполнения этой обязанности передается сообщение, реализуемое посредством метода makePayment. Более того, для ее выполнения требуется взаимодействие с объектами SalesLineltem и вызов их конструктора.

В объектно-ориентированной технологии проектирования **шаблоном назы­вают именованное описание проблемы и ее решения, которые можно применить при разработке других систем**.

Далее считаем синонимами термины “шаблон проектирования” и “паттерн”.

Известны два de-facto основных семейства шаблонов:

1) **шаблоны GRASP** (General Responsibility Assignment Software Patterns – Общие шаблоны распределения обязанностей в программных систе­мах) (GRASP в переводе – схватывать, постигать);

2) **шаблоны GoF** (Gang Of Four – банда четырех: Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес).

Пример определения одного из шаблонов семейства GRASP:

|  |  |
| --- | --- |
| **Имя шаблона** | **Information Expert** |
| **Решаемая проблема** | **Каков основной принцип распределения обязанностей между объектами?** |
| **Решение** | **Обязанности назначаются классу, который имеет необходимую для их выполнения информацию** |

Шаблон может включать следующие элементы.

1. **Имя.**Сcылка на имя описывает проблему проектирова­ния, ее решения и их последствия. Присваивание имен шаблонам позволяет проектировать на более высоком уровне абстракции, с помощью их словаря можно вести обсуждение с коллегами, упоминать в документации, в тонкостях представлять проект системы.

2. **Задача.**Описание того, когда следует применять шаблон. Необходимо сформулировать задачу и ее контекст. Может описываться конкретная проблема проектирования (например, способ представления алгоритмов в виде объек­тов). Иногда отмечается, какие структуры классов или объектов свидетельствуют о негибком проекте. Также может включаться перечень условий, при выполнении которых имеет смысл применять данный шаблон.

3. **Решение.**Описание элементов проекта, отношений между ними, функций каждого элемента. Конкретный проект или реализация не имеются в виду, поскольку шаблон применим в самых разных ситуациях. Просто дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего обобщенного сочетания эле­ментов (классов и объектов).

4. **Результаты.**Следствия применения шаблона и разного рода компромиссы. О последствиях часто не упо­минают, но знать о них необходимо, чтобы можно было выбрать между различ­ными вариантами и оценить преимущества и недостатки данного шаблона. Здесь речь идет и о выборе языка и реализации. Поскольку в объектно-ориентированном проектировании повторное использование – важный фактор, то к результатам следует относить и влияние на степень гибкости, расширяемости и переносимости системы.

В идеале, шаблон содержит советы по поводу его применения в различных ситуациях, а также описание его преимуществ.

Таким образом, шаблон проектирования:

· именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, позволяющие применить его для со­здания повторно используемого проекта;

· вычленяет участвующие классы и экземпляры, их роль и отношения, а также функции для конкретной задачи объектно-ориентированно­го проектирования.

**Важно отметить, что шаблоны не содержат новых идей.** Термин "шаблон" означает стандартную, повторяющуюся сущность. Шаблоны не предназначены для изучения и выражения новых принципов разработки программ­ного обеспечения. Наоборот, они призваны систематизировать существующие знания, идиомы и принципы. Чем шире они используются, тем лучше.

# 41. Основные шаблоны GRASP

Шаблоны GRASP – хорошо документированные, стандартизированные и проверенные временем принципы объектно-ориентированного анализа.

Список шаблонов GRASP (Ларман): <https://ru.wikipedia.org/wiki/GRASP>

**Information Expert (Информационный эксперт)** – выполнение обязанностей назначается информационному эксперту – классу, имеющему информацию, которая необходима для выполнения обязанности.

Конечно, не всегда вся необходимая информация заключена в одном классе (это противоречит шаблону High Cohesion), чаще она распределена между различными классами, тогда каждый из этих классов будет являться частичным экспертом, предоставляя только доступную ему информацию, а при общем взаимодействии будет решена общая задача.

**Creator (Создатель)** – классу В назначается обязанность по созданию экземпляров класса А, если класс B удовлетворяет одному из следующих условий:

♦ содержит объекты класса А;

♦ агрегирует объекты класса А;

♦ обладает данными инициализации объекта класса А;

♦ записывает экземпляры объектов класса А;

♦ активно использует объекты класса А.

Смысл шаблона Creator интуитивно понятен, но за кажущейся простотой скрывается масса преимуществ: поддерживается шаблон Low Coupling, снижаются затраты на сопровождение и обеспечивается возможность повторного использования созданных компонентов в дальнейшем.

**Controller (Контроллер)** – обязанность по обработке системных сообщений назначается классу, удовлетво­ряющему одному из следующих условий:

1) класс представляет всю систему, устройство или подсистему в целом (внешний контроллер);

2) класс представляет некоторый сценарий прецедента, в процессе выполнения которого происходят системные события (контроллер прецедента или сеанса).

**Low Coupling** (Слабая связанность) – обеспечить низкую зависимость классов и повысить возможность повторного использования можно распределением обязанностей таким образом, чтобы степень связанности остава­лась низкой.

Связность можно определить как количество точек соприкосновения классов между собой. Известно, что чем ниже связность классов, тем меньше их взаимовлияние, тем выше возможность повторного использования. Не существует абсолютной меры для определения слишком высокой степени связывания. Важно понимать степень связанности на текущий момент и не упустить тот момент, когда дальнейшее повышение связанности может привести к появлению проблем. Следует руководствоваться таким соображением: классы, которые являются достаточно общими по своей природе и с высокой вероятностью будут повторно использоваться в дальнейшем, должны иметь минимальную степень связанности с другими классами. Высокая степень связывания сама по себе не является проблемой. Проблемой является жесткое связывание с неустойчивыми в некотором отношении элементами.

**High Cohesion** (Сильное зацепление) – обеспечить возможность управления сложностью можно распределением обязанностей таким образом, чтобы степень зацепления остава­лась высокой.

Разница между связностью и зацеплением очень тонка, и понять ее необходимо на более глубоком интуитивном уровне. Зацепление (cohesion) (точнее, функциональное зацепление) - это мера связанности и концентрации обязанностей класса. Считается, что элемент обладает высокой степенью зацепления, если его обязанности тесно связаны между собой и он не выполняет огромных объемов работы. Класс с низкой степенью зацепления выполняет много разнородных функций или несвязанных между собой обязанностей.

High Cohesion утверждает, что класс должен выполнять как можно меньше не специфичных для него задач, и иметь вполне определенную область применения. Только с опытом приходит понимание балансировки между High Cohesion и Low Coupling. Некорректное связывание порождает неправильное зацепление и наоборот.

**Polymorphism (Полиморфизм)** – если поведение объектов одного типа (класса) может изменяться, обязанности распределяются для различных вариантов поведения с использованием поли­морфных операций для этого класса.

Шаблон Polymorphism решает проблему обработки альтернативных вариантов поведения на основе типа. Решать эту проблему стоит с использование полиморфных операций, а не с помощью проверки типа и условной логики. Кроме того, при помощи полиморфизма легко создавать подключаемые компоненты. В итоге получаем следующие преимущества:

· легко расширять систему, добавляя новые вариации;

· новые вариации можно вводить без модификации клиентской части приложения.

**Pure Fabrication(Чистая выдумка)** – дляреализации шаблонов High Cohesion и Low Coupling без проблем создается искусственный класс, не представляющий конкретное понятие из предметной области, а только для поддержки высокого зацепления, слабого связывания и повторного использования

Например, классу X необходимо сохранять информацию в реляционной базе данных. Согласно шаблону Information Expert, эту обязанность можно присвоить самому классу X. Однако следует учесть, что это требует достаточно большого числа специализированных операций, поэтому класс X будет иметь низкую степень зацепления. Класс X будет связан с интерфейсом БД, что повысит связность, кроме того задача записи в БД является довольно общей, поэтому необходимо обеспечить повторное использование кода. Естественным решением данной проблемы является создание нового класса, ответственного за сохранение объектов некоторого вида, например в базе данных. Его можно назвать PersistentStorage. Это чисто синтетический объект, что полностью соответствует Pure Fabrication. В итоге решаются задачи зацепления, связности и повторного использования.

**Indirection(Посредник)** – для обеспечения связи между компонен­тами или службами, не связанными между собой напрямую, вводится промежуточный объект.

Пример, рассмотренный выше, с введением класса PersistentStorage является также и примером шаблона Indirection, т.к. дополнительный класс будет являться промежуточным звеном между БД и классом X. Примерами специализированных вариантов Indirection являются шаблоны Adapter, Facade, Observer (см. шаблоны GoF). Целью введения промежуточного звена является обеспечение слабой связности за счет отделения друг от друга различных компонентов.

**Protected Variations (Устойчивость к изменениям)** – дляраспределения обязанностей между объектами, подсистемами и системами так, чтобы вариации или нестабильность их элементов не оказывали отрицательного влияния на другие элементы, необходимо идентифицировать конкретные нестабильные точки, а обязанности распределить так, чтобы вокруг этих точек был создан "стабильный" интерфейс.

Ключевые принципы, на основе которых реализуются обеспечения гибкости и защиты системы от влияния изменений внешних систем – инкапсуляция данных, интерфейсы, полиморфизм, перенаправление.

Существует много способов обеспечения устойчивости интерфейса:

o проектирование на основе данных;

o использование служб;

o проектирование на основе интерпретаторов;

o рефлексивное проектирование или проектирование на метауровне;

o унифицированный доступ.

Пример шаблона **GRASP**

|  |  |
| --- | --- |
| **Имя шаблона** | **Информационный эксперт (Information Expert)** |
| **Проблема** | В системе должна аккумулироваться, рассчитываться и т. п. необходимая информация. |
| **Решение** | Назначить обязанность аккумуляции информации, расчета и т. п. некоему классу (информационному эксперту), обладающему необходимой информацией. |
| **Рекомендации** | Информационным экспертом может быть не один класс, а несколько. |
| **Пример** | Необходимо рассчитать общую сумму продажи. Имеются классы проектирования "Продажа", "ТоварПродажа" (продажа отдельного вида товара в рамках продажи в целом), "ТоварСпецификация" (описание конкретного вида товара).  Необходимо распределить обязанности по предоставлению информации и расчету между этими классами. Объект "Продажа" должен передать сообщение "Рассчитать промежуточную сумму" каждому экземпляру класса "ТоварПродажа" (которые, в свою очередь, передают сообщения "СообщитьЦену" объектам "ТоварСпецификация", с целью получения информации о цене экземпляра товара), и, затем, просуммировать полученные результаты. Промежуточную сумму рассчитывает объект "Товар Продажа". Таким образом, все три объекта являются информационными экспертами. |
| **Преимущества** | Шаблон Expert поддерживает инкапсуляцию. Для выполнения требуемых задач объекты используют собственные данные. Подобную возможность обеспечивает также шаблон Low Coupling (GRASP), применение которого приводит к созданию более надежных и легко поддерживаемых систем  Соответствующее поведение системы обеспечивается несколькими классами, содержащими требуемую информацию. Это приводит к определениям клас­сов, которые гораздо проще понимать и поддерживать. Кроме того, поддер­живается шаблон High Cohesion (GRASP). |
| **Недостатки** | Если объект, обладающий наиболее полной информацией, например, о продаже (*см. пример* - класс "Продажа"), будет отвечать и за сохранение этой информации в базе данных, то получится, что логика приложения (моделирование продажи) и логика связи с базой данных "помещаются" в один класс (нарушение принципа разделения обязанностей основных объектов системы, и, кроме того, логика связи с базой данных будет дублироваться во многих других классах. |
| **Cвязанные шаблоны** | Low Coupling (Слабая связанность) (GRASP).  High Cohesion (Сильное зацепление) (GRASP). |
| **Другие названия и аналогичные принципы** | "Хранение обязанностей вместе с дан­ными",  "Кто знает, тот и выполняет",  "Сделай сам",  "Размещайте службы вме­сте с их атрибутами". |

Пример другого шаблона **GRASP**

|  |  |
| --- | --- |
| **Имя шаблона** | **Создатель экземпляров класса (Creator)** |
| **Проблема** | "Кто" должен отвечать за создание экземпляров класса. |
| **Решение** | Назначить классу В обязанность создавать объекты другого класса А |
| **Р екомендации** | Логично использовать шаблон если класс В содержит, агрегирует, активно использует и т.п. объекты класса А. |
| **Пример** | Необходимо определить, какой объект должен отвечать за создание экземпляра "ТоварПродажа". Логично, чтобы это был объект "Продажа", поскольку он содержит (агрегирует) несколько объектов "ТоварПродажа". |
| **Преимущества** | Использование этого шаблона не повышает связанности, поскольку созданный класс, как правило, виден только для класса - создателя. |
| **Недостатки** | Если процедура создания объекта достаточно сложная (например выполняется на основе некоего внешнего условия), логично использовать шаблон "Абстрактная Фабрика", делегируя обязанность создания объектов специальному классу. |

Пример устаревшего шаблона GRASP:

|  |  |
| --- | --- |
| **Имя шаблона** | **Не разговаривайте с неизвестными (Don't talk to strangers)** |
| **Проблема** | Обеспечить связь клиентского объекта с непрямыми объектами (то есть известными другим объектам, а не самому клиенту). |
| **Решение** | Необходимо избегать проектных решений, предполагающих передачу сообщений с удаленными непрямыми объектами (незнакомцами). Решением может быть частный случай шаблона "Устойчивый к изменениям"**(Protected Variations)** - GRASP. Прямым объектам потребуются новые операции. |
| **Преимущества** | Обеспечивает устойчивость системы к изменению структуры объектов. |

Шаблон Don't talk to strangers заменен шаблоном Protected Variations

# 

# 42. Описание шаблонов проектирования (GoF)

Паттерны GoF (Gang of Four) включают в себя 23 паттерна. Эти паттерны не зависят от языка реализации, но их реализация зависит от области приложения.

За счёт шаблонов производится унификация терминологии, названий модулей и элементов проекта. Позволяет, отыскав удачное решение, пользоваться им снова и снова.

|  |  |
| --- | --- |
| Абстрактная фабрика | Класс, который представляет собой интерфейс для создания компонентов системы |
| Строитель | Класс, который представляет собой интерфейс для создания сложного объекта |
| Фабричный метод | Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанциировать |
| Прототип | Объект, инициализируемый во время первого обращения к нему |
| Одиночка | Класс, который может иметь только один экземпляр. |
| Посетитель | Описывает операцию, которая выполняется над объектами других классов. При изменении класса Visitor нет необходимости изменять обслуживаемые классы |
| Адаптер | Объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс |
| Мост | Структура, позволяющая изменять интерфейс обращения и интерфейс реализации класса независимо |
| Конпановщик | Объект, который объединяет в себе объекты, подобные ему самому |
| Дерекоратор | Класс, расширяющий функциональность другого класса, без использования наследования |
| Фасад | Объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое |
| Приспособленец | Это объект, представляющий себя как уникальный экземпляр в разных местах программы, но по факту не являющийся таковым |
| Цепочка ответственности | Предназначен для организации в системе уровней ответственности |
| Команда | Представляет действие. Объект команды заключает в себе само действие и его параметры |
| Интерпретатор | Решает часто встречающуюся, но подверженную изменениям, задачу |
| Итератор | Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящий в состав агрегации |
| Посредник | Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга |
| ангел Хранитель | Позволяет не нарушая инкапсуляцию зафиксировать и сохранить внутреннее состояния объекта так, чтобы позднее восстановить его в этом состоянии |
| Наблюдатель | Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии |
| Состояние | Используется в тех случаях, когда во время выполнения программы объект должен менять свое поведение в зависимости от своего состояния |
| Стратегия | Предназначен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости |
| Шаблонный метод | Определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом. |

# 43. Классификация шаблонов проектирования GoF

**Порождающие шаблоны** — шаблоны, создающие объекты. Избавляют от необходимости прямого создания объектов, придают программе гибкость при необходимости решать, какой 102 объект в данном случае должен быть создан. К ним относятся шаблоны:

Фабрика (Factory method) — простой класс принимающий решение о том, объект какого из классов наследников базового класса вернуть, основываясь на входных данных.

Абстрактная фабрика (Abstract Factory Method) — предоставляет интерфейс для создания одного из нескольких семейств объектов.

Строитель(Builder) — разделяет устройство сложного объекта и его представление, так, чтобы при потребности можно было создать несколько различных представлений.

Прототип (Prototype) — копирует или клонирует созданный и инициализированный экземпляр объекта, вместо создания новых экземпляров.

Одиночка(Singleton) — класс, который ген 90 точку доступа к этому экземпляру.

**Структурные шаблоны**описывают, как из классов или объектов могут быть собраны более крупные структуры. Разница между структурными шаблонами классов и структурными шаблонами объектов в том, что шаблоны классов определяют, как увеличить эффективность программных интерфейсов с помощью наследования.

Адаптер (Adapter) — используется для преобразования программного интерфейса одного класса к программному интерфейсу другого

Мост(Bridge) — используется для разделения интерфейса и класса, данные которого отображаются. Позволяет менять их раздельно и варьировать их взаимодействие.

Композиция(Composite) — набор объектов, каждый из которых может быть как примитивным объектом, так и композицией

Оформитель,Обертка(Decorator) — шаблон представляет собой класс, который «оборачивает» заданный класс добавляя к нему новые возможности, и не изменяя существующие методы.

Фасад(Facade) — шаблон группирует сложную иерархию и предоставляет упрощенный интерфейс для доступа к данным

Легковес(Flyweight) — шаблон обобщает множество мелких однородных объектов, путем извлечения части данных из них и хранения вне самих объектов. Сами сущности при этом становятся — легковесами и не требуют отдельного объекта на каждый экземпляр.

Заместитель(Proxy) — шаблон предоставляет простой класс-суррогат для более сложного и ресурсоемкого класса.

**Поведенческие шаблоны** помогают определить взаимодействие между объектами в системе или обработку данных в сложной программе.

Наблюдатель (Observer) — определяет способ, которым множество объектов получает извещение о изменениях в одном.

Посредник (Mediator) — определяет, как упростить взаимодействие между классами так, чтобы им не требовалось знать ничего друг о друге.

Цепочка ответственности(Chain of responsibility) — углубляет разделение между классами, передавая запрос между ними, пока он не будет обработан.

Шаблон (Template) — предоставляет абстрактное определение алгоритма

Интерпретатор(Interpreter) — определяет порядок включения элементов языка в программу. Определяет представление грамматики заданного языка, интерпретирует его предложения.

Стратегия(Strategy) — инкапсулирует алгоритм внутрь класса

Посетитель(Visitor) — добавляет функцию в класс

Состояние(State) — предоставляет память для переменных объекта. Определяет поведение объекта в зависимости от его состояния.

Команда(Command, Action,Transaction) — предоставляет простой способ отделения выполнения команды от интерфейса ее создавшего, посредством инкапсуляции ее в объект.

Итератор(Iterator, Cursor) — определяет и формализует способ последовательного доступа ко всем данным составного объекта, не раскрывая структуры данных

# 44. Структурные шаблоны проектирования

Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.

Использование Структурные шаблоны уровня класса используют наследование для составления композиций из интерфейсов и реализаций.

Простой пример — использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот шаблон, когда нужно организовать совместную работу нескольких независимо разработанных библиотек.

**Адаптер** (*Adapter*)— это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

**Шаблон мост** ( *Bridge*) — «разделять [абстракцию](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) и [реализацию](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) так, чтобы они могли изменяться независимо». Шаблон мост использует [инкапсуляцию](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), [агрегирование](https://ru.wikipedia.org/wiki/%D0%90%D0%B3%D1%80%D0%B5%D0%B3%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и может использовать наследование для того, чтобы разделить ответственность между классами.

**Компоновщик** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Composite pattern*) — шаблон проектирования объединяющий объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково.

**Декоратор** (*Decorator*) — [структурный](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) [шаблон проектирования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), предназначенный для динамического подключения дополнительного поведения к [объекту](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

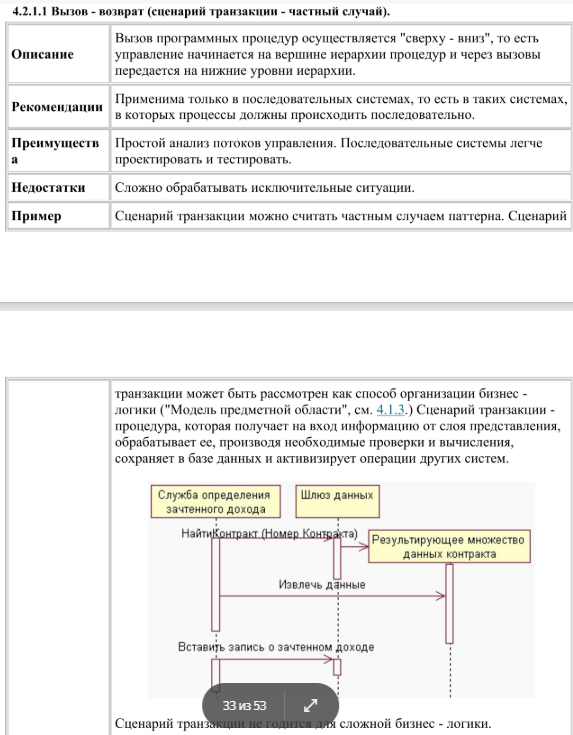
**Шаблон фасад** — позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному [объекту](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), делегирующему их соответствующим объектам системы.

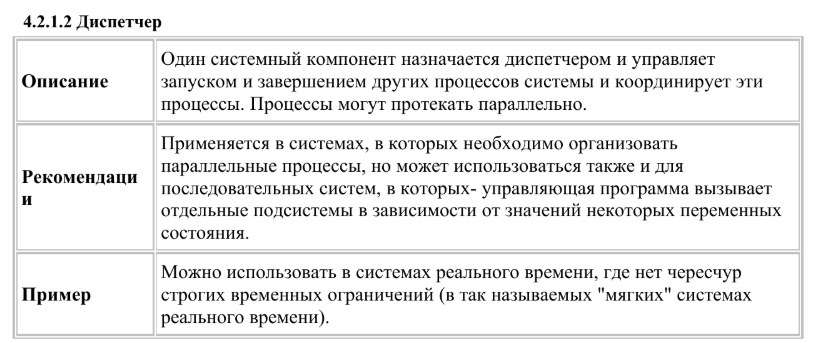
Font-controller - это « [контроллер,](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller) который обрабатывает все запросы на [веб-сайт](https://en.wikipedia.org/wiki/Website) », который является полезной структурой для разработчиков веб-приложений для достижения гибкости и повторного использования без избыточности кода.

**Приспособленец**— [шаблон проектирования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым.

**Заместитель** ( *Proxy*) — [шаблон проектирования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), предоставляющий [объект](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), который контролирует доступ к другому объекту, перехватывая все вызовы

# 45. Шаблоны централизованного управления





шаблоны централизованного управления относятся к типам шаблонов, которые предлагают централизованный способ организации логики выполняемых программных продуктов.

Паттерны этой категории содержат описание механизмов единого унифицированного управления, основанного на структурных особенностях поэтапного выполнения программ. Эти шаблоны выполняют роль "мозгового центра" проектируемого программного продукта.

#### **Сценарий транзакций**

Современные корпоративные информационные системы обладают большим количеством компонентов и громоздкой иерархической структурой их взаимосвязей.

Вызов процедур, как правило, осуществляется "сверху вниз". Управление начинается на вершине иерархии и через вызовы передается на нижние уровни.Подобный способ управления хорошо зарекомендовал себя в программных продуктах, в которых процессы запускаются и выполняются последовательно, в соответствии с условиями их функционирования.

Применение шаблона "Сценарий транзакций" способствует сохранению целостности, прозрачности, предсказуемости создаваемого программного обеспечения."Сценарий транзакции"целесообразно рассматривать как механизм организации бизнес-логики ("Модель предметной области").

*"Сценарий транзакций"– процедура, которая получает на вход информацию от слоя представления, обрабатывает ее, производя необходимые проверки и вычисления, сохраняет в базе данных и активизирует операции других слоев.*

В качестве основных недостатков можно выделить сложность выявления и обработки исключительных ситуаций. Сценарий транзакции не годится для организации "запутанной"бизнес-логики.

#### **Диспетчер**

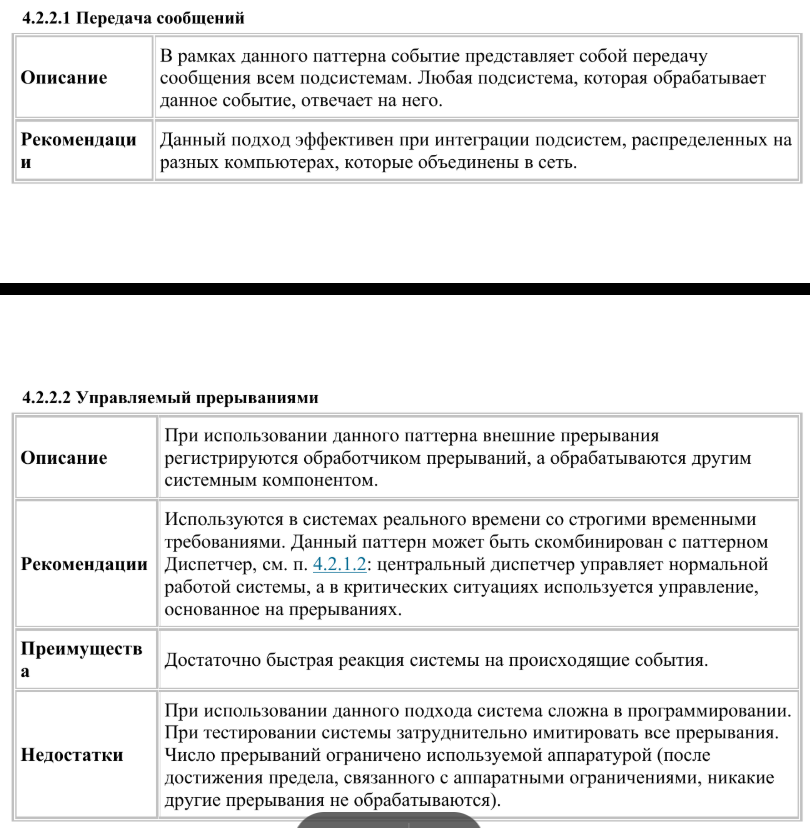
Этот шаблон организации систем предусматривает создание системного компонента, который назначается диспетчером, то есть отвечает за управление и координацию выполнения процессов, исполняемых в программном продукте. При этом нет ограничений на способы выполнения процессов ("последовательно", "параллельно" и пр.).

Паттерн, как правило, применяется в системах, для которых необходимо повысить их общую производительность за счет "запараллеливания" процессов, но может использоваться и для последовательной обработки (управляющая программа или диспетчер вызывает отдельные подсистемы в зависимости от значений некоторых переменных состояния).

Одним из главных недостатков применения шаблона "Диспетчер" является рекомендация по использованию в так называемых "мягких" системах реального времени (нет строгих временных ограничений).

*Применение "Диспетчера" целесообразно в тех случаях, когда группа разработки полностью понимает и имеет различные способы воздействия на все программные компоненты (система представляет собой "прозрачный" ящик).*

# 46. Шаблоны управления на основе событий



Управление по событиям является способом координации потока(ов) информации и ее последующей обработки в зависимости от факторов и условий, значимых для достижений конечного результата деятельности программного приложения.

Шаблоны управления по событиям помогают в разработке логики, автоматизирующей значимые функции, возникающие при наступлении или отсутствии определенных бизнес-сигналов.

#### **Передача сообщений**

Данный шаблон проектирования транслирует каждое обрабатываемое событие в виде передачи сообщений всем заинтересованным подсистемам, которые обрабатывают событие и при необходимости отвечают на него. Шаблон "Передача сообщений" получил особую популярность при реализации систем, обрабатывающих сложную и разрозненную бизнес-логику.

Логика исполнения основана на сетевом взаимодействии различных участников автоматизируемого процесса.Сферы ответственности участников достаточно независимы друг от друга. Конечный результат обработки определяется вкладом каждого из участников.

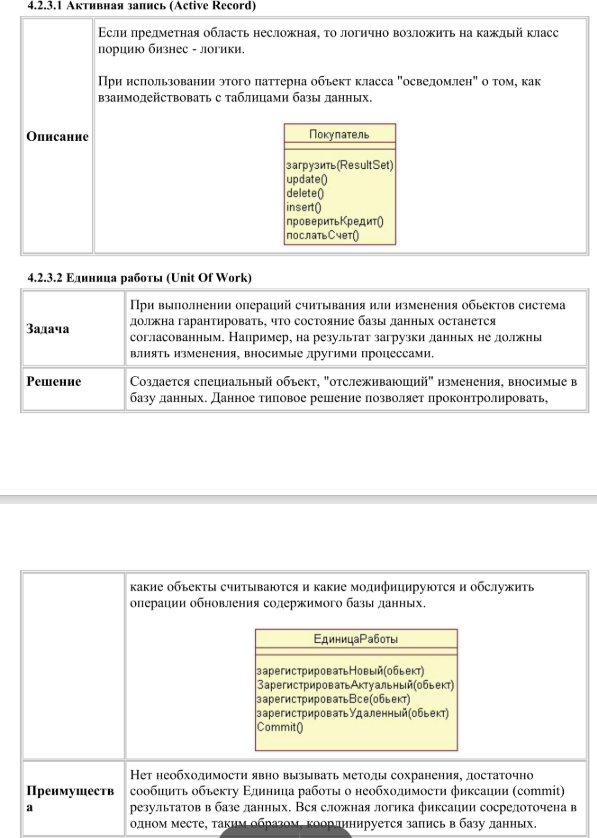
#### **Управление прерываниями**

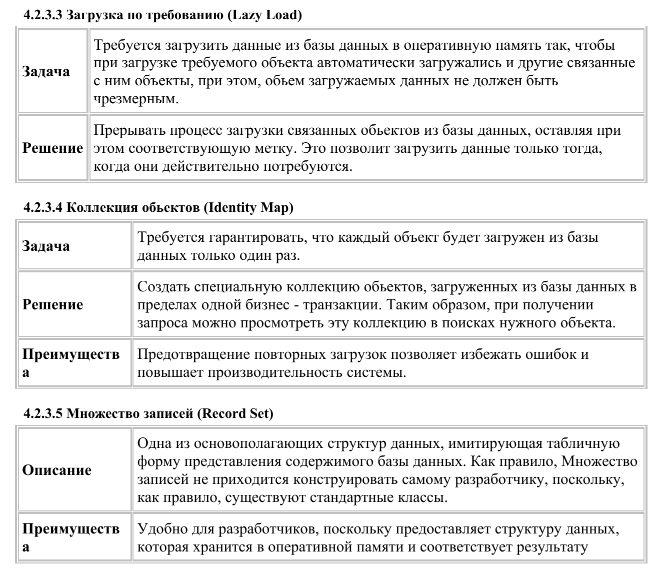
Шаблон "Управление прерываниями" применяется в основном в системах реального времени со строгими временными требованиями на обработку поступающих извне сообщений, когда необходима быстрая реакция системы на зафиксированное событие.

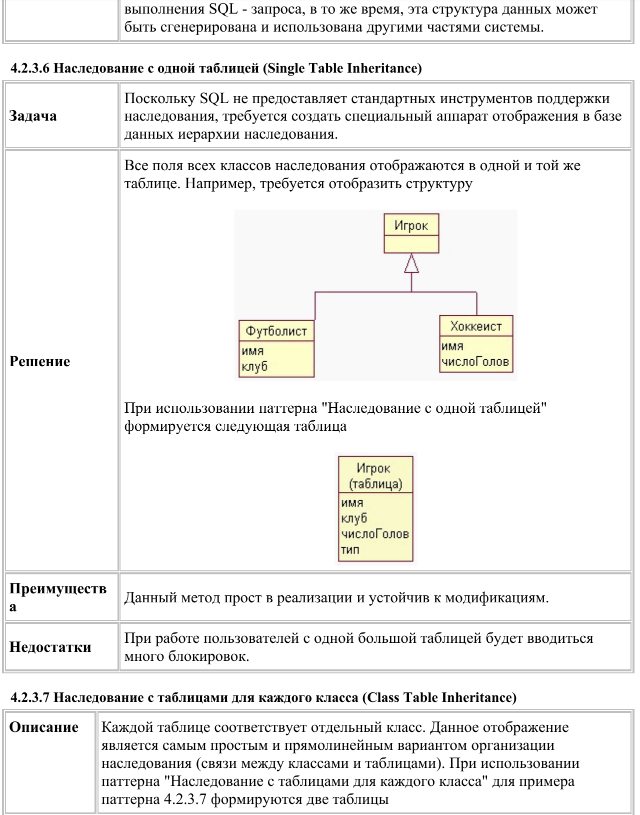
Использование данного шаблона предусматривает регистрацию возникающих прерываний специализированным обработчиком. После чего наступает стадия непосредственной обработки, которая должна выполняться другим системным компонентом.

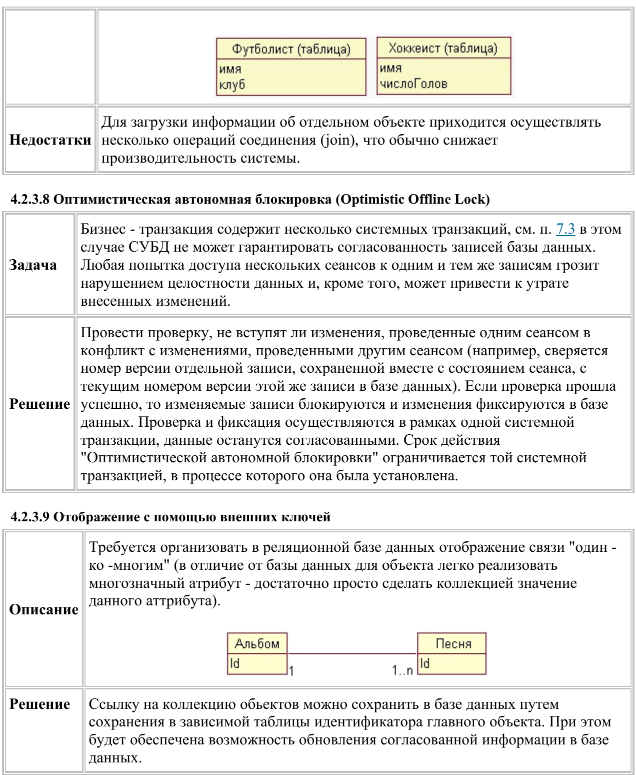
Шаблон "Управление прерываниями" очень часто используют вместе с шаблоном "Диспетчер". В случае штатного функционирования система управляется посредством "Диспетчера", а в критических ситуациях используется управление, основанное на прерываниях. Основной недостаток подобного комбинирования шаблонов состоит в том, что система сложна в программировании и последующем тестировании. Если уделить внимание основным сложностям, то они состоят в имитации всех возможных прерываний, которые должны быть известны априори.

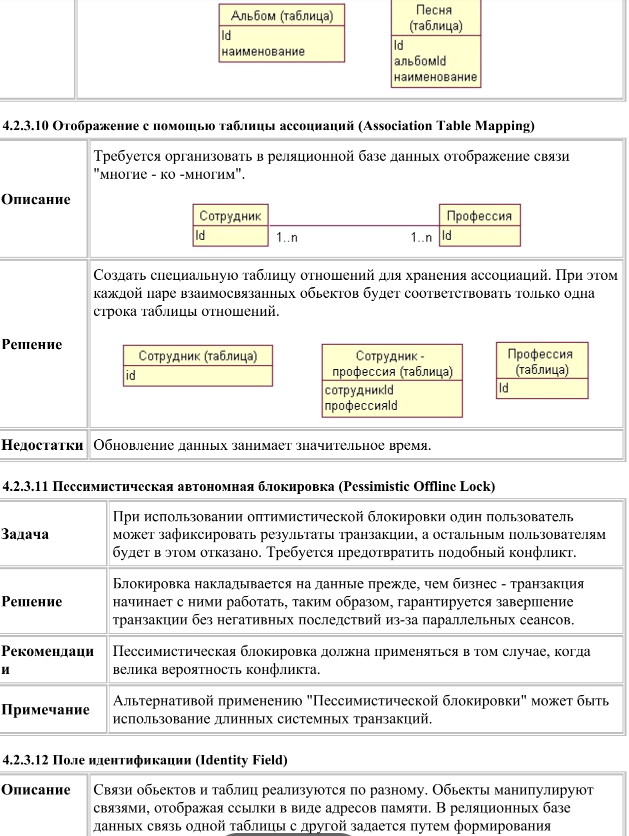
# 47. Шаблоны взаимодействия с базами данных

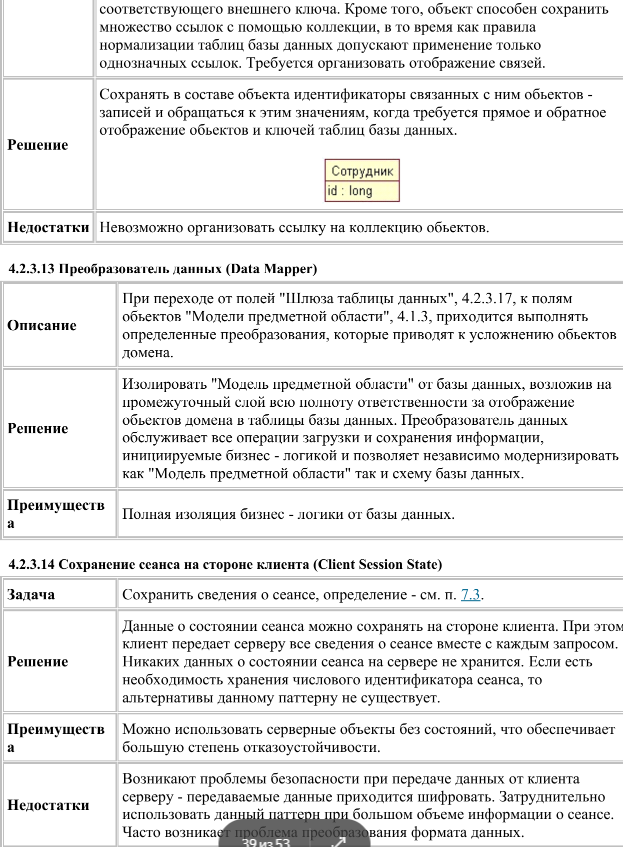


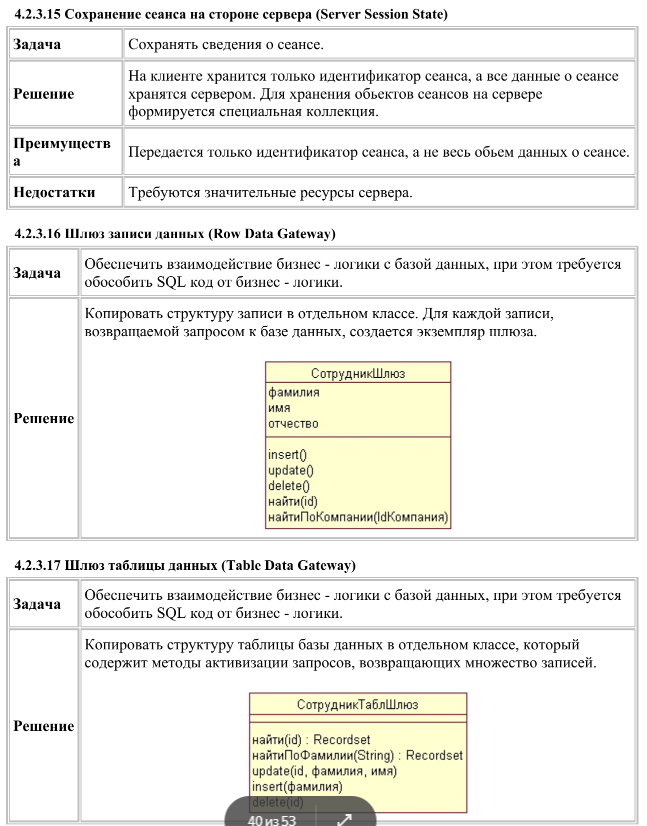












заимодействие с базой данных – неотъемлемая часть современных информационных систем. То, насколько оптимально организована связь между слоем логики и слоем данных, во многом предопределяет нефункциональные характеристики программных продуктов.

Шаблоны, описывающие данный способ взаимодействия, позволяют избежать многих проблем, которые возникают в работе над созданием программного обеспечения.

#### **Активная запись**

В случаях, когда предметная область несложная, то логично каждому классу делегировать и ответственность за конкретные объекты бизнес-логики.При использовании этого паттерна объект класса должен "понимать", как взаимодействовать с данными.

Для этого шаблона характерны следующие характеристики:

* Экземпляр класса соответствует определенной записи в таблице.
  + При создании нового экземпляра класса в таблицу добавляется новая запись.
* При чтении полей объекта считываются соответствующие значения записи таблицы баз данных.
* При изменении (удалении) какого-либо объекта изменяется (удаляется)

#### **Единица работы**

При выполнении операций (создание, чтение, изменение, удаление) над объектами базы данных система должна гарантировать, что данные останутся согласованными (консистентными).

Эта задача решается за счет специализированного объекта, который должен отслеживать вносимые изменения. Подобное решение позволит контролировать различное состояние объектов и соответствующим образом обслужить операции обновления содержимого базы данных. Не будет необходимости явно вызывать методы сохранения – достаточно сообщить объекту о необходимости фиксации результатов работы отдельных методов в базе данных.Вся сложная логика фиксации сосредоточена в одном месте.

#### **Загрузка по требованию**

Если требуется загрузить данные из базы в оперативную память так, чтобы при загрузке требуемого объекта автоматически загружались и другие связанные с ним объекты (при этом объем загружаемых данных не должен быть чрезмерным), используется шаблон "Загрузка по требованию".

При загрузке связанных объектов процесс прерывается при достижении первоначально необходимого объема данных,но остается метка о важности загрузки последующих данных. Это позволит загрузить данные только тогда, когда они действительно потребуются.

#### **Количество объектов**

Когда требуется гарантировать, что каждый объект будет загружен из базы данных только один раз, применяется шаблон "Количество объектов". Паттерн реализуется за счет создания специальной коллекции объектов, загруженных из базы данных в пределах одной транзакции.

При получении очередного запроса можно просмотреть эту коллекцию в поисках нужного объекта.Предотвращение повторных загрузок позволяет избежать ошибок и повышает производительность системы.

#### **Множество записей**

Шаблон "Множество записей" относится к одной из основополагающих "динамических" структур данных.Он имитирует табличную форму представления содержимого базы данных. "Множество записей" не приходится конструировать – как правило, существуют стандартные классы, реализующие этот шаблон данных.

Они удобны, поскольку предоставляют структуру данных, которая хранится в оперативной памяти и соответствует результату выполнения запроса к базе данных.В то же время эта структура данных может быть сгенерирована и использована другими частями системы.

#### **Шлюз записи данных**

В продолжение обсуждения "низкоуровневых" шаблонов, которые катализируют процесс разработки программного обеспечения за счет экономии ресурсов на повторном использовании наиболее эффективных программных элементов, следует затронуть следующий паттерн–"Шлюз записи данных".

Этот шаблон обеспечивает взаимодействие компонентов слоев бизнес-представления со слоями базы данных. Для его создания следует инкапсулировать код реализации компонента от кода бизнес-логики. Реализация этого шаблона заключается в копировании структуры записей в отдельном программном классе. Для каждой записи, возвращаемой запросом к базе данных, создается экземпляр шлюза.

# 48. СТРУКТУРНЫЕ ШАБЛОНЫ ИНТЕГРАЦИИ

Паттерны интеграции информационных систем - верхний уровень классификации паттернов проектирования.

Среди паттернов интеграции выделена группа структурных паттернов (описывают основные компоненты единой интегрированной метасистемы).

Для описания взаимодействия отдельных корпоративных систем, включенных в интегрированную метасистему, организована группа паттернов, объединенных в соответствии с тем или иным методом интеграции.

МЕТОДЫ ИНТЕГРАЦИИ:

1. Интеграция систем по данным (data-centric).
2. Функционально-центрический подход (function-centric) .
3. Объектно-центрический (object-centric).
4. Интеграция на основе единой понятийной модели предметной области (concept-centric).

Интеграция корпоративных информационных систем подразумевает тем или иным способом организованный обмен данными между системами.

Паттерны интеграции по типу обмена данными:

1. Файловый обмен
2. Общая база данных
3. Удаленный вызов процедур
4. Обмен сообщениями

# 49. ПОНЯТИЕ АНТИ-ШАБЛОНА ИЛИ ЛОВУШКИ ПРОЕКТИРОВАНИЯ

Анти-паттерны (anti-patterns), также известные как ловушки (pitfalls) — это классы наиболее часто внедряемых плохих решений проблем. Они изучаются, как категория, в случае когда их хотят избежать в будущем, и некоторые отдельные случаи их могут быть распознаны при изучении неработающих систем.

Анти-паттерны:

1. в разработке ПО
2. Инверсия абстракции (Abstraction inversion): Создание простых конструкций поверх сложных (спорный)
3. Неопределённая точка зрения (Ambiguous viewpoint): Представление модели без спецификации её точки рассмотрения
4. Большой комок грязи (Big ball of mud): Система с нераспознаваемой структурой
5. Блоб (Blob): см. Божественный объект (God object)
6. Бензиновая фабрика (Gas factory): Необязательная сложность дизайна
7. Затычка на ввод данных (Input kludge): Забывчивость в спецификации и выполнении поддержки возможного неверного ввода
8. Раздувание интерфейса (Interface bloat): Изготовление интерфейса очень мощным и очень трудным для осуществления
9. Магическая кнопка (Magic pushbutton): Выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса. Например, в системах типа Delphi это написание прикладной логики в обработчиках нажатий на кнопку
10. Перестыковка (компьютер) (Re-Coupling): Процесс внедрения ненужной зависимости
11. Дымоход (Stovepipe system): Редко поддерживаемая сборка плохо связанных компонентов
12. Гонки (Race hazard, Race condition): непредвидение возможности наступления событий в порядке, отличном от ожидаемого
13. в управлении разработкой ПО
14. Дым и зеркала (Smoke and mirrors): Демонстрация того, как будут выглядеть ненаписанные функции (название происходит от двух излюбленных способов, которыми фокусники скрывают свои секреты)
15. Раздувание ПО (Software bloat): Разрешение последующим версиям системы требовать всё больше и больше ресурсов
16. Функции для галочки: Превращение программы в конгломерат плохо реализованных и не связанных между собой функций (как правило, для того, чтобы заявить в рекламе, что функция есть)
17. в ООП
18. Базовый класс-утилита (BaseBean): Наследование функциональности из класса-утилиты вместо делегирования к нему
19. Вызов предка (CallSuper): Для реализации прикладной функциональности методу класса потомка требуется в обязательном порядке вызывать те же методы класса-предка
20. Ошибка пустого подкласса (Empty subclass failure): Создание класса (в Perl), который не проходит «проверку пустоты подкласса» («Empty Subclass Test») из-за различного поведения по сравнению с классом, который наследуется от него без изменений
21. Божественный объект (God object): Концентрация слишком большого количества функций в одной части системы (классе)
22. Объектная клоака (Object cesspool): Переиспользование объектов, находящихся в непригодном для переиспользования состоянии
23. Полтергейст (компьютер) (Poltergeist): Объекты, чьё единственное предназначение — передавать информацию другим объектам
24. Проблема йо-йо (Yo-yo problem): Чрезмерная размытость сильно связанного кода (например, выполняемого по порядку) по иерархии классов
25. Одиночество (антипаттерн) (Singletonitis): Избыточное использование паттерна одиночка
26. Методологические анти-паттерны
27. Анти-паттерны управления конфигурацией
28. Анти-паттерны в программировании

# 50. МЕТОДЫ НАСТРОЙКИ И ОБЕСПЕЧЕНИЯ ГИБКОСТИ ПРОЕКТОВ

# 51. Понятие стратегии и классов стратегии

Определим понятие стратегии как интерфейс обычного класса или шаблона класса. Как пример, рассмотрим стратегию создания объектов Creator, представленную классом шаблона с параметром T, где есть функция T \*Create():

template <class T> struct OpNewCreator // Операция new

{

static T\* Create() {

return new T;

}

};

template <class T> struct MallocCreator // Функция malloc для размещения объекта

{

static T\* Create() {

void \*p=std::malloc(sizeof(T));

return p? new(p) T:0;

}

};

template <class T> struct PrototypeCreator // Клонирование прототипа

{

PrototypeCreator(T \*pObj=0): pPrototype(pObj) {}

T\* Create() {

return pPrototype? pPrototype->Clone():0;

}

T\* Get() {return pPrototype; }

void Set(T \*pObj) { pPrototype=pObj; }

private:

T \*pPrototype;

};

Стратегия создания объектов Creator здесь представлена тремя реализациями – классами стратегии. Такие классы обычно отдельно не используются, а наследуются другими классами или включаются в них. Классы стратегий могут быть представлены и шаблонами классов. В отличие от шаблона проектирования Strategy GoF, классы стратегий связываются на этапе компиляции.

Классы стратегий, в отличие от чисто виртуальных функций, не имеют четкого определения. Стратегия Creator определяет лишь правильный вид синтаксиса класса стратегии – наличие функции T \*Create(), не указывая вид и набор других функций подобных классов.

Использование стратегии обеспечивается через наследование или включение ее классов в другие классы:

template <class Creator>

class AnyController: public Creator { // Наследование класса стратегии

//…

};

Класс, использующий одну или несколько стратегий, называют главным (host classes).

Выбранная при проектировании стратегия указывается при конкретизации шаблона главного класса:

class Flow;

typedef AnyController<OpNewCeator<Flow>> FlowController;

Очевидно, что такой оператор разумно поместить в удобном для настройки проекта – заголовочном файле.

# 52. Способы реализации классов стратегии

Реализация классов стратегии может проводится разными способами.

Первый способ: использование шаблонных параметров шаблонов

template <template <class> class Creator>

class AnyController:

public Creator<Flow> { // Наследование класса стратегии

//…

};

Конкретизация шаблона главного класса становится проще и безопаснее:

typedef AnyController<OpNewCeator> FlowController;

Второй способ: использование классов стратегий с шаблонными функциями-элементами

template <class T>

struct OpNewCreator // Операция new

{

static T\* Create() {

return new T;

}

};

используется обычный класс, но с шаблоном функции-элемента

struct OpNewCreator // Операция new

{

template <class T>

static T\* Create() {

return new T;

}

};

Рассматриваемый способ считается труднее объясняемым, определяемым, реализуемым и применяемым.

Ранее приводились примеры трех классов стратегии, различающихся набором открытых функций. Классы стратегии с дополнительными функциями называют расширенными. Можно заметить, что главные классы никогда не вызывают дополнительные функции по определению стратегии. Однако пользователь может их вызывать, зная о их наличии после конкретизации шаблона. Возможные ошибки обнаружатся на этапе компиляции.

Преимущества использования стратегий:

· стратегии можно менять извне, лишь меняя шаблонные аргументы при конкретизации класса;

· набор стратегий можно расширять;

· стратегии учитывают более подробную информацию о типе и статическом связывании, что повышает компактность и эффективность кода по сравнению с виртуальными функциями.

Недостаток:

· неудобства динамического связывания.

# 53. Неполная конкретизация классов стратегии

Если функция-элемент шаблона класса не используется, то согласно стандарту языка С++, она не конкретизируется, а просто игнорируется компилятором, но после проверки ошибок синтаксиса. Это позволяет главным классам задавать и применять так называемую неполную конкретизацию.

Рассмотрим класс стратегии:

template <class T>

struct PrototypeCreator // Клонирование прототипа

{

Prototype(T \*pObj=0): pPrototype(pObj) {}

T\* Create() {

return pPrototype? pPrototype->Clone():0;

}

T\* Get() {return pPrototype; }

void Set(T \*pObj) { pPrototype=pObj; }

private:

T \*pPrototype;

};

и его применение в шаблоне главного класса

template <template <class> class Creator>

class AnyController:

public Creator<Flow> { // Наследование класса стратегии

//…

void SwitchPrototype(Flow \*pNewPrototype) {

Creator<Flow>& MyController = \*this;

delete MyController.Get();

MyController.Set(pNewPrototype);

//…

}

};

Ситуация 1. Конкретизация фиксирует стратегию PrototypeCreator

typedef AnyController<PrototypeCreator> MyController;

Программа может вызывать функцию SwitchPrototype.

Ситуация 2. Конкретизация фиксирует стратегию

struct OpNewCreator // Операция new

{

static T\* Create() {

return new T;

}

};

typedef AnyController<OpNewCreator> MyController;

Ошибочный вызов функции SwitchPrototype, которой нет в классе OpNewCreator, будет обнаружен компилятором. В случае отсутствия такого вызова программа будет работать правильно. Неполная конкретизация ­– отражение использования принципа приоритета действия перед планом, не претендует на оптимизацию решения для конкретного случая, но максимально ориентируется на повторное использование общих решений.

# 54. Функторы и их использование

Функтор (сокр. от функциональный объект) — концепция программирования, позволяющая использовать объект класса как функцию. Как правило, функторы используются для написания callback-функций.

Функциональный объект — объект, который можно использовать как функцию, либо функция, обладающая свойствами объекта (атрибутами).

Перегрузка оператора вызова позволяет использовать объект подобно функции:

**class** Functor {

**public**:

**void operator**() (**double**, **int**) { }

};

…

Functor f;

f(3.14, 0);

Разумеется, объект с таким перегруженным оператором может иметь и другие

методы, а также поля. Если продемонстрированное использование объекта является основным, принято называть такой объект *функтором (functor).* Функторы активно используются в стандартной библиотеке языка С++.

# 55. Понятие рефакторинга программ

**Рефакторинг** — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших сохраняющих поведение преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и четкости.

**Цель рефакторинга** — сделать код программы легче для понимания. Рефакторинг следует отличать от оптимизации производительности. Как и рефакторинг, оптимизация обычно не изменяет поведение программы, а только ускоряет ее работу. Но оптимизация часто затрудняет понимание кода, что противоположно рефакторингу .

Основными **стимулы для проведения рефакторинга**:

1. необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение;

2. необходимо исправить ошибку, причины возникновения которой сразу не ясны;

3. преодоление трудности в командной разработке, которые обусловлены сложной логикой программы.

Наиболее употребимые **методы рефакторинга**:

Изменение сигнатуры метода (Change Method Signature)

Инкапсуляция поля (Encapsulate Field)

Выделение класса (Extract Class)

Выделение интерфейса (Extract Interface)

Выделение локальной переменной (Extract Local Variable)

Выделение метода (Extract Method)

**Инкапсуляция поля (Encapsulate field)**

В случае, если у класса имеется открытое поле, необходимо сделать его закрытым

и обеспечить методы доступа. После «Инкапсуляции поля» часто применяется

«Перемещение метода».

**Перемещение метода (Move Method)**

Перемещение метода применяется по отношению к методу, который чаще

обращается к другому классу, чем к тому, в котором сам располагается.

# 56. Анти-шаблоны управления разработкой программ

**Антипаттерн —** это распространённый подход к решению класса часто встречающихся проблем, являющийся неэффективным, рискованным или непродуктивным. В отличие от [шаблона проектирования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), рассмотрение антипаттерна включает в себя как неправильное решение проблемы с его признаками и последствиями, так и выход из ситуации

**Дым и зеркала(Smoke and mirrors)**: Демонстрация того, как будут выглядеть ненаписанные функции (название отражает способы скрытия секретов [фокусник](http://ru.wikipedia.org/w/index.php?title=%D0%A4%D0%BE%D0%BA%D1%83%D1%81%D0%BD%D0%B8%D0%BA)ами).

[**Раздувание ПО**](http://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B0%D0%B7%D0%B4%D1%83%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%9F%D0%9E) **(Software bloat)**: Разрешение последующим версиям системы требовать всё больше и больше ресурсов.

Технически раздутое программное обеспечение чаще всего ассоциируется с понятием «расползание возможностей», которое обозначает тенденцию разработчиков добавлять в программный продукт больше и больше возможностей в попытке «не отстать от конкурентов», однако приводящую фактически к созданию более медленного и менее эффективного изделия. Согласно озвученным в 2002 году данным исследований, только 20 %—25 % функций программных продуктов использовались всегда или часто, тогда как до 45 % функций вообще никогда не использовались. По мнению [Мартина Фаулера](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D1%83%D0%BB%D0%B5%D1%80,_%D0%9C%D0%B0%D1%80%D1%82%D0%B8%D0%BD), большинство проектов должны были бы иметь лишь четверть от их существующего размера.

Вклад в «раздувание возможностей» вносит также «эффект второй системы», описанный Фредериком Бруксом ещё в 1975 году: программист, разрабатывающий свою вторую систему, склонен добавлять все те возможности, которые он не смог добавить в свою первую систему (из-за нехватки времени), поэтому вторая система часто получается перегруженной возможностями.

**Функции для галочки(software bloat, bloatware)**: Превращение программы в конгломерат плохо реализованных и не связанных между собой функций (обычно для утверждения в [рекламе](http://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B5%D0%BA%D0%BB%D0%B0%D0%BC%D0%B0), что некоторая функция есть).

# 57. Анти -шаблоны разработки программ

**Анти-паттерны в разработке ПО:**

**Антипаттерн —** это распространённый подход к решению класса часто встречающихся проблем, являющийся неэффективным, рискованным или непродуктивным. В отличие от [шаблона проектирования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), рассмотрение антипаттерна включает в себя как неправильное решение проблемы с его признаками и последствиями, так и выход из ситуации

**1.** **Инверсия абстракции** (Abstraction inversion): Создание простых конструкций поверх сложных (спорный)

**2 Неопределённая точка зрения** (Ambiguous viewpoint): Представление модели без спецификации её точки рассмотрения

**3 Большой комок грязи** (Big ball of mud): Система с нераспознаваемой структурой

**4 Блоб** (Blob): см. Божественный объект (God object): объект, который делает слишком много, или хранит в себе слишком много

**5 Бензиновая фабрика** (Gas factory): Необязательная сложность дизайна

**6 Затычка на ввод данных** (Input kludge): Забывчивость в спецификации и выполнении поддержки возможного неверного ввода

**7 Раздувание интерфейса** (Interface bloat): Изготовление интерфейса очень мощным и очень трудным для осуществления

**8 Магическая кнопка** (Magic pushbutton): Выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса. Например, в системах типа Delphi это написание прикладной логики в обработчиках нажатий на кнопку

**9 Перестыковка** (компьютер) (Re-Coupling): Процесс внедрения ненужной зависимости

**10 Дымоход** (Stovepipe system): Редко поддерживаемая сборка плохо связанных компонентов

**11 Гонки** (Race hazard, Race condition)

58. Анти -шаблоны в объектно -ориентированном программировании

**Анти-паттерны в объектно-ориентированном программировании:**

**1 Базовый класс-утилита** (BaseBean): Наследование функциональности из класса-утилиты вместо делегирования к нему

**2 Вызов предка** (CallSuper): Для реализации прикладной функциональности методу классапотомка требуется в обязательном порядке вызывать те же методы класса-предка

**3 Ошибка пустого подкласса** (Empty subclass failure): Создание класса (в Perl), который не проходит «проверку пустоты подкласса» («Empty Subclass Test») из-за различного поведения по сравнению с классом, который наследуется от него без изменений

**4 Божественный объект** (God object): Концентрация слишком большого количества функций в одной части системы (классе)

**5 Объектная клоака** (Object cesspool): Переиспользование объектов, находящихся в непригодном для переиспользования состоянии

**6 Полтергейст** (компьютер) (Poltergeist): Объекты, чьё единственное предназначение — передавать информацию другим объектам

**7 Проблема йо-йо** (Yo-yo problem): Чрезмерная размытость сильно связанного кода (например, выполняемого по порядку) по иерархии классов

**8 Одиночество** (антипаттерн) (Singletonitis): Избыточное использование паттерна одиночка

**9 Каша из интерфейсов** (Interface soup[15]): Объединение нескольких интерфейсов, разделенных согласно принципу изоляции интерфейсов (Interface segregation), в один.

**10 Висящие концы**: Интерфейс, большинство методов которого бессмысленны и реализуются «пустышками».

**11 Заглушка** (Stub): Попытка «натянуть» на объект уже имеющийся малоподходящий по смыслу интерфейс, вместо создания нового.

**12** [**Френд-зона**](https://ru.wikipedia.org/w/index.php?title=%D0%A4%D1%80%D0%B5%D0%BD%D0%B4-%D0%B7%D0%BE%D0%BD%D0%B0_(%D0%B0%D0%BD%D1%82%D0%B8%D0%BF%D0%B0%D1%82%D1%82%D0%B5%D1%80%D0%BD)&action=edit&redlink=1)(Friend zone): Неуместное использование дружественных классов и дружественных функций в языке C++.

# 59. Анти-шаблоны в программировании

**Анти-паттерны в программировании:**

**1.Ненужная сложность** (Accidental complexity): Внесение ненужной сложности в решение

**2.Действие на расстоянии** (Action at a distance): Неожиданное взаимодействие между широко разделёнными частями системы

**3.Накопить и запустить** (Accumulate and fire): Установка параметров подпрограмм в наборе глобальных переменных

**4.Слепая вера** (Blind faith): Недостаточная проверка (a) корректности исправления ошибки или (b) результата работы подпрограммы

**5.Лодочный якорь** (Boat anchor): Сохранение более не используемой части системы

**6.Активное ожидание** (Busy spin): Потребление ресурсов ЦПУ (процессорного времени) во время ожидания события, обычно при помощи постоянно повторяемой проверки, вместо того, чтобы использовать асинхронное программирование (к примеру, систему сообщений или событий)

**7.Кэширование ошибки** (Caching failure): Забывать сбросить флаг ошибки после её обработки

**8.Воняющий подгузник** (The Diaper Pattern Stinks): Сброс флага ошибки без ее обработки или передачи вышестоящему обработчику

**9.Проверка типа вместо интерфейса** (Checking type instead of membership, Checking type instead of interface): Проверка того, что объект имеет специфический тип в то время, когда требуется только определённый интерфейс

**10.Инерция кода** (Code momentum): Сверхограничение части системы путём постоянного подразумевания её поведения в других частях системы

**11.Кодирование путём исключения** (Coding by exception): Добавление нового кода для поддержки каждого специального распознанного случая

**12.Таинственный код** (Cryptic code): Использование аббревиатур вместо мнемоничных имён 137

**13.Жёсткое кодирование** (Hard code): Внедрение предположений об окружении системы в слишком большом количестве точек её реализации

**14.Мягкое кодирование** (Soft code): Патологическая боязнь жёсткого кодирования, приводящая к тому, что настраивается всё что угодно, при этом конфигурирование системы само по себе превращается в программирование

**15.Поток лавы** (Lava flow): Сохранение нежелательного (излишнего или низкокачественного) кода по причине того, что его удаление слишком дорого или будет иметь непредсказуемые последствия

**16.Магические числа** (Magic numbers): Включение в алгоритмы чисел без объяснений их смысла

**17.Процедурный код** (Procedural code): Когда другая парадигма является более подходящей

**18.Спагетти-код** (Spaghetti code): Код с чрезмерно запутанным порядком выполнения

**19.Мыльный пузырь** (Soap bubble): Класс, инициализированный мусором, максимально долго притворяется, что содержит какие-то данные

# 60. Методологические анти-шаблоны

– **Программирование методом копирования-вставки** (Copy and paste pro-

gramming) – копирование (и лёгкая модификация) существующего кода вместо

создания общих решений. Симптом этого антипаттерна: после внесения измене-

ний программа в некоторых случаях ведёт себя также, как и раньше. Для устра-

нения антипаттерна требуется выделить повторяющийся код в отдельный метод.

– **Дефакторинг** (De-Factoring) – процесс уничтожения функциональности и

замены её документацией.

– **Золотой молоток** (Golden hammer) – сильная уверенность в том, что лю-

бимое решение универсально применимо. Название происходит от английской

поговорки «когда в руках молоток, все проблемы кажутся гвоздями».

– **Фактор невероятности** (Improbability factor) – предположение о невоз-

можности того, что сработает известная ошибка.

– **Преждевременная оптимизация** (Premature optimization) – оптимизация на

основе недостаточной информации.

– **Изобретение колеса** (Reinventing the wheel) – ошибка адаптации существу-

ющего решения.

– **Изобретение квадратного колеса** (Reinventing the square wheel) – создание

плохого решения, когда существует хорошее.

[**Шаблон проектирования**](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)**(Design pattern):** само по себе использование шаблонов считается анти-шаблоном – признак того, что система не воплощает достаточный уровень абстракции.

[**Самоуничтожение**](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%B0%D0%BC%D0%BE%D1%83%D0%BD%D0%B8%D1%87%D1%82%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1)**(Self-destruction):** Фатальная ошибка либо нестандартное поведение программы, приводящая к отказу в обслуживании, возникшая вследствие другой менее серьёзной ошибки. Например, при возникновении ошибки, приложение начинает очень быстро и много писать в журнал, вследствие чего заканчивается место на жёстком диске быстрее, чем это обнаружит мониторинг событий.

[**Два тоннеля**](https://ru.wikipedia.org/w/index.php?title=%D0%94%D0%B2%D0%B0_%D1%82%D0%BE%D0%BD%D0%BD%D0%B5%D0%BB%D1%8F&action=edit&redlink=1)**:** Вынесение новой функциональности в отдельное приложение вместо расширения уже имеющегося. Чаще всего применяется, когда по каким-либо причинам (в основном, при нехватке времени либо нежелании менеджмента) внесение изменений в уже имеющийся код требует больших затрат, чем создание нового. При этом у клиента в конечном итоге работают два приложения, запускаясь одновременно либо попеременно друг из друга.

[**Коммит-убийца**](https://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%BE%D0%BC%D0%BC%D0%B8%D1%82-%D1%83%D0%B1%D0%B8%D0%B9%D1%86%D0%B0&action=edit&redlink=1)**(Commit assasin):** Внесение отдельных изменений в систему контроля версий без проверки влияния их на другие части программы. Как правило, после подобных действий работа коллектива парализуется на время исправления проблем в местах, которые ранее работали безошибочно.

# 61. Анти-шаблоны управления конфигурацией

[**Ад зависимостей**](http://ru.wikipedia.org/w/index.php?title=%D0%90%D0%B4_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D0%B5%D0%B9) **(Dependency hell):** Проблемы с версиями требующихся продуктов, особенно в системах [UNIX](http://ru.wikipedia.org/w/index.php?title=UNIX)/[GNU/Linux](http://ru.wikipedia.org/w/index.php?title=GNU/Linux).  
Разрастание графа взаимных зависимостей программных продуктов и библиотек, приводящее к сложности установки новых и удаления старых продуктов. В сложных случаях различные установленные программные продукты требуют наличия разных версий одной и той же библиотеки. В наиболее сложных случаях один продукт может косвенно потребовать сразу две версии одной и той же библиотеки.

[**DLL-ад**](http://ru.wikipedia.org/w/index.php?title=DLL-%D0%B0%D0%B4) **(DLL hell):** Проблемы с версиями, доступностью и увеличением количества [DLL](http://ru.wikipedia.org/w/index.php?title=%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0), особенно в [Microsoft Windows](http://ru.wikipedia.org/w/index.php?title=Microsoft_Windows).

**Дым и зеркала** (Smoke and mirrors): Демонстрация того, как будут выглядеть ненаписанные функции (название происходит от двух излюбленных способов, которыми фокусники скрывают свои секреты).

**Раздувание ПО** (Software bloat): Разрешение последующим версиям системы требовать всё больше и больше ресурсов.

**Функции для галочки**: Превращение программы в конгломерат плохо реализованных и не связанных между собой функций (как правило, для того, чтобы заявить в рекламе, что функция есть).

**JAR-ад** : проблемы с различными версиями или местоположениями файлов JAR, обычно вызванные отсутствием понимания модели загрузки классов

**Конфликт расширений** : проблемы с различными расширениями Mac OS, пытающимися исправлять одни и те же части операционной системы.

# 62. Примеры организационных анти-шаблонов

• Аналитический паралич (Analysis paralysis): Выделение непропорционально больших усилий в фазе анализа проекта.

• Дойная корова (Cash cow): Закрытый продукт, приносящий выгоду, часто ведёт к самоуспокоенности относительно новых продуктов.

• Продолжительное устаревание (Continuous obsolescence): Выделение непропорционально больших усилий портированию системы в новые окружения.

• Сваливание расходов (Cost migration): Перенос расходов на проект к уязвимому отделу или бизнес-партнёру.

• Ползущий улучшизм (Creeping featurism): Добавление новых улучшений в ущерб качеству системы.

• Разработка комитетом (Design by committee): Результат того, что имеется много содействующих разработке, но не имеется единого видения.

• Эскалация обязательств (Escalation of commitment): Продолжение реализации решения в том случае, когда неправильность его доказана.

• Я тебе это говорил (I told you so): Когда игнорируется предупреждение эксперта, являющееся оправданным.

• Управление основанное на числах (Management by numbers): Уделение избыточного внимания численным критериям управления, когда они неважны или стоимость их получения слишком высока.

• Драконовские меры (Management by perkele): Военный стиль управления без толерантности к диссидентству.

• Управление грибами (Mushroom management): Удержание работников в неинформированном и занятом состоянии.

• Расползание рамок (Scope creep): Дозволение рамкам проекта расти без должного контроля.

• Замкнутость на продавце (Vendor lock-in): Изготовление системы, жёстко привязанной к одному поставщику.

• Тёплое тело (Warm body): Человек, чей вклад в проект под сомнением, особенно если рассмотрен в панике.

• Единственный знающий человек (Single head of knowledge): ЕЗЧ (SHOK) применим в том случае, когда единственная личность во всей организации контролирует жизненно-важную область ноу-хау или информации о внутренностях системы.

• Рыцарь на белом коне (Knight in shining armor): РНБК (KISA) происходит тогда, когда личность, которая не совершает ошибок, появляется на сцене и пытается починить всё, без сообщений о том, какие изменения он/она сделал/сделает и почему.

# 63. СОЦИАЛЬНЫЕ АНТИ-ШАБЛОНЫ

* Цензура (Censorship): Подавление дискуссии и запрещение определённых тем в рамках обсуждения системы, в результате которого система ухудшается по качеству, функциональности или другим показателям
* Концентрация власти (Political corruption, Concentrated power): Индивидуальное злоупотребление властью, даже с изначально хорошими помыслами
* Демократия (Democracy): Большая группа индивидов не может принимать аргументированные решения, а руководствуется лишь поверхностной информацией.
* Диктатура (Dictatorship): Не всегда один индивид имеет все навыки, необходимые для ведения проекта и грамотного управления.
* Дискриминация (Discrimination): Концентрация на неуместных особенностях усиливает экономическую неэффективность и социальную напряжённость
* Догма (Dogmatic religion): Догма подавляет индивидуальное мышление и тормозит прогресс
* Нетерпимость (Intolerance): Настаивание на изменении нежелательных, но безопасных особенностей других людей влечёт усиление напряжённости и также, очень часто является задачей, которая отнимает время от проекта, и решение которой не имеет значительных плюсов
* Монополия (Monopoly): Без соперничества большинство эффектов свободного рынка не работают, и частная компания не имеет стимула действовать максимально эффективно
* Система голосования на основе большинства (Plurality voting system): Политика при голосовании на основе большинства вырождается в две полярно-противоположные партии, результатом чего является подавление других политических воззрений 141
* Соревнование в популярности (Popularity contest): Популярность становится самодостаточной величиной и не сопоставима ни с каким другими параметрами или достоинствами
* Сегрегация (Racial segregation): Разделение по равноправию весьма редко, если вообще существует; ведёт к напряжённости
* Однопартийная система (Single-party system): Аналогично монополии, но внутри компании.
* Тоталитаризм (Totalitarianism): Нередко подавление индивидуальности с переходом за определённые рамки, ведёт к напряжённости, которая отрицательно влияет на эффективность.
* Преступление без жертв (Victimless crime): Подавление поведения создаёт субкультуру людей, постоянно живущих по другим законам, для которых эта правовая система является врагом
* Охота на ведьм (Witch hunt): Попытки отыскать козла отпущения, но если проблема никогда не решается в действительности, результатом будет являться поиск всё новых и новых козлов отпущения
* Нулевой Год (Year Zero): Социальное изменение является долгим процессом, ускорение его влечёт катастрофу

# 64. Основные типы шаблонов проектирования

|  |  |  |
| --- | --- | --- |
| [Шаблон делегирования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%B4%D0%B5%D0%BB%D0%B5%D0%B3%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) | Delegation pattern | Объект внешне выражает некоторое поведение, но в реальности передаёт ответственность за выполнение этого поведения связанному объекту. |

|  |  |  |  |
| --- | --- | --- | --- |
| [Шаблон функционального дизайна](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B4%D0%B8%D0%B7%D0%B0%D0%B9%D0%BD%D0%B0) | Functional design | Гарантирует, что каждый модуль компьютерной программы имеет только одну обязанность и исполняет её с минимумом побочных эффектов на другие части программы. | н/д |
| [Неизменяемый интерфейс](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B8%D0%B7%D0%BC%D0%B5%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81) | Immutable interface | Создание [неизменяемого объекта](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B8%D0%B7%D0%BC%D0%B5%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82). | н/д |
| [Интерфейс](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Interface | Общий метод для структурирования компьютерных программ для того, чтобы их было проще понять. | н/д |
| [Интерфейс-маркер](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81-%D0%BC%D0%B0%D1%80%D0%BA%D0%B5%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Marker interface | В качестве атрибута (как пометки объектной сущности) применяется наличие или отсутствие реализации интерфейса-маркера. В современных языках программирования вместо этого могут применяться атрибуты или аннотации. | н/д |
| [Контейнер свойств](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%82%D0%B5%D0%B9%D0%BD%D0%B5%D1%80_%D1%81%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Property container | Позволяет добавлять дополнительные свойства для класса в контейнер (внутри класса), вместо расширения класса новыми свойствами. | н/д |
| [Канал событий](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BD%D0%B0%D0%BB_%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D0%B9_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) | Event channel | Расширяет шаблон [Publish/Subscribe](https://ru.wikipedia.org/wiki/%D0%98%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C-%D0%BF%D0%BE%D0%B4%D0%BF%D0%B8%D1%81%D1%87%D0%B8%D0%BA_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), создавая централизованный канал для событий. Использует объект-представитель для подписки и объект-представитель для публикации события в канале. Представитель существует отдельно от реального издателя или подписчика. Подписчик может получать опубликованные события от более чем одного объекта, даже если он зарегистрирован только на одном канале. |  |

# 65. Шаблон делегирования

**Делегирование** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Delegation*) — [основной](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F&action=edit&redlink=1) [шаблон проектирования](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в котором [объект](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) внешне выражает некоторое [поведение](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5), но в реальности передаёт ответственность за выполнение этого поведения связанному объекту. Шаблон делегирования является фундаментальной [абстракцией](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), на основе которой реализованы другие шаблоны - [композиция](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%B7%D0%B8%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (также называемая агрегацией), [примеси](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BC%D0%B5%D1%81%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (mixins) и [аспекты](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D0%BF%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (aspects).

Плюсы: Возможность изменить поведение конкретного экземпляра объекта вместо создания нового класса путём наследования.

Минусы: Этот шаблон обычно затрудняет [оптимизацию по скорости](https://ru.wikipedia.org/w/index.php?title=%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D0%BE_%D1%81%D0%BA%D0%BE%D1%80%D0%BE%D1%81%D1%82%D0%B8_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F&action=edit&redlink=1) в пользу улучшенной чистоты абстракции.

Например, мы имеем класс, моделирующий автомобиль и еще один класс, описывающий его рулевую систему.

Мы можем воспользоваться композицией и включить рулевую систему просто в состав автомобиля, но можно поступить и немного иначе. Будет весьма логично потребовать от автомобиля двигаться вперед, остановиться или повернуть. Ведь именно такого поведения мы и ожидаем. Поэтому в классе **Car** можно создать экземпляр класса **ControlSystem** и также добавим в него те же методы — **start(), stop(), left(), right()**, в которых вызовем методы уже объекта **ControlSystem**. При этом внешне мы вроде бы управляем самим автомобилем, но на деле всю работу за нас выполняет объект класса ControlSystem.

# 66. Шаблон функционального дизайна

Шаблон функционального дизайна — это шаблон проектирования использующийся для упрощения проектирования ПО. Функциональный дизайн гарантирует, что каждый модуль компьютерной программы имеет только одну обязанность и исполняет ее с минимумом побочных эффектов на другие части программы. Функционально разработанные модули имеют очень маленькое сцепление.

**Преимущества**

Системы с функционально-спроектированными частями легче модифицировать, потому что каждая часть делает только то, для чего она предназначена. Так как поддержка программы занимает больше 3/4 жизни успешной системы, эта особенность является решающим преимуществом. Это также делает систему лёгкой для понимания и документирования, что также упрощает обучение. Результатом является то, что практическое время жизни функциональной системы больше.

Преимуществом для реализации является то, что если программный модуль имеет единственное предназначение, он будет проще и тем самым — легче и менее дорогостоящим для проектирования и реализации.

**Методика**

Стандартный способ обеспечения функционального дизайна — это обзор описания модуля. Если описание включает связи, такие как «и» или «или», тогда дизайн имеет 146 более, чем одно предназначение, и соответственно возможно будет иметь побочные эффекты. Предназначения должны быть разделены в отдельные модули для того, чтобы функциональный дизайн был бы достижим.

# 67. Неизменяемый объект (шаблон проектирования)

Неизменяемый объект — в объектно-ориентированном программировании объект, который не может быть изменён после своего создания.

* Объект может быть неизменяемым как полностью, так и частично.
* В некоторых случаях объект считается неизменяемым с точки зрения пользователя класса, даже если изменяются его внутренние поля.
* Как правило, неизменяемый объект получает все внутренние значения во время инициализации, либо значения устанавливаются в несколько этапов, но до того, как объект будет использован.
* Часто, неизменяемые объекты могут быть полезными потому, что они позволяют избежать некоторых дорогостоящих операций копирования и

сравнения.

* Таким образом упрощается исходный код программы, и ускоряется ее работа.
* Однако, в некоторых случаях, неизменяемость объекта может мешать, например, если объект содержит большое количество изменяемых.
* Многие языки программирования имеют возможности работы как с изменяемыми, так и с неизменяемыми объектами.

# **68. Интерфейс (шаблон проектирования)**

Шаблон интерфейса является общим методом для структурирования компьютерных программ для того, чтобы их было проще понять. В общем, интерфейс — это класс, который обеспечивает программисту простой или более программноспецифический способ доступа к другим классам.

Интерфейс может содержать набор объектов и обеспечивать простую, высокоуровневую функциональность для программиста (например, Шаблон Фасад); он может обеспечивать более чистый или более специфический способ использования сложных классов («класс-обёртка»); он может использоваться в качестве «клея» между двумя различными API (Шаблон Адаптер); и для многих других целей. Другими типами интерфейсных шаблонов являются: Шаблон делегирования, Шаблон компоновщик, и Шаблон мост.

# 

# 69. Порождающие шаблоны проектирования

Абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

*Инстанцирование*— создание экземпляра класса. В отличие от слова «создание», применяется не к объекту, а к классу. То есть, говорят: «(в виртуальной среде) создать экземпляр класса или инстанцировать класс». Порождающие шаблоны используют полиморфное инстанцирование.

**Использование**

Эти шаблоны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных.

Порождающие шаблоны инкапсулируют знания о конкретных классах, которые применяются в системе. Они скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, — это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие шаблоны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда.

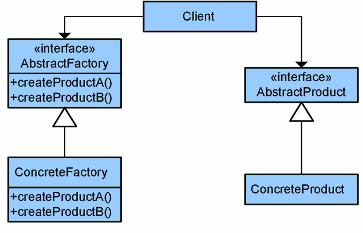
Иногда допустимо выбирать между тем или иным порождающим шаблоном. В других ситуациях порождающие шаблоны дополняют друг друга. Так, применяя строитель, можно использовать другие шаблоны для решения вопроса о том, какие компоненты нужно строить, а прототип часто реализуется вместе с одиночкой. Порождающие шаблоны тесно связаны друг с другом, их рассмотрение лучше проводить совместно, чтобы лучше были видны их сходства и различия.

**Перечень порождающих шаблонов:**

* абстрактная фабрика (abstract factory);
* строитель (builder);
* фабричный метод (factory method);
* ленивая инициализация (lazy initialization);
* объектный пул (object pool);
* прототип (prototype);
* одиночка (singleton).

# 70. Абстрактная фабрика (шаблон проектирования)

**Шаблон Abstract Factory** использует несколько фабричных методов и предназначен для создания целого семейства или группы взаимосвязанных объектов. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся наследующиеся от него классы, реализующие этот интерфейс.



**Преимущества и недостатки**

+ Гарантирует сочетаемость создаваемых продуктов.

+ Избавляет клиентский код от привязки к конкретным классам продуктов.

+ Выделяет код производства продуктов в одно место, упрощая поддержку кода.

+ Упрощает добавление новых продуктов в программу.

+ Реализует *принцип открытости/закрытости*.

* Усложняет код программы из-за введения множества дополнительных классов.
* Требует наличия всех типов продуктов в каждой вариации.

**Отношения с другими паттернами**

· Многие архитектуры начинаются с применения [Фабричного метода](https://refactoring.guru/ru/design-patterns/factory-method) (более простого и расширяемого через подклассы) и эволюционируют в сторону [Абстрактной фабрики](https://refactoring.guru/ru/design-patterns/abstract-factory), [Прототипа](https://refactoring.guru/ru/design-patterns/prototype) или [Строителя](https://refactoring.guru/ru/design-patterns/builder) (более гибких, но и более сложных).

· [Строитель](https://refactoring.guru/ru/design-patterns/builder) концентрируется на построении сложных объектов шаг за шагом. [Абстрактная фабрика](https://refactoring.guru/ru/design-patterns/abstract-factory) специализируется на создании семейств связанных продуктов. *Строитель* возвращает продукт только после выполнения всех шагов, а *Абстрактная фабрика* возвращает продукт сразу же.

· Классы Абстрактной фабрики чаще всего реализуются с помощью [Фабричного м](https://refactoring.guru/ru/design-patterns/factory-method)етода, хотя они могут быть построены и на основе [Прототипа](https://refactoring.guru/ru/design-patterns/prototype).

· [Абстрактная фабрика](https://refactoring.guru/ru/design-patterns/abstract-factory) может быть использована вместо [Фасада](https://refactoring.guru/ru/design-patterns/facade) для того, чтобы скрыть платформо-зависимые классы.

· [Абстрактная фабрика](https://refactoring.guru/ru/design-patterns/abstract-factory) может работать совместно с [Мостом](https://refactoring.guru/ru/design-patterns/bridge). Это особенно полезно, если у вас есть абстракции, которые могут работать только с некоторыми из реализаций. В этом случае фабрика будет определять типы создаваемых абстракций и реализаций.

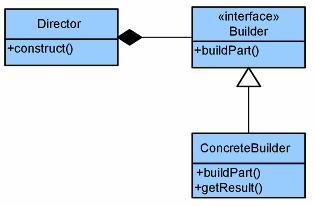
· [Абстрактная фабрика](https://refactoring.guru/ru/design-patterns/abstract-factory), [Строитель](https://refactoring.guru/ru/design-patterns/builder) и [Прототип](https://refactoring.guru/ru/design-patterns/prototype) могут быть реализованы при помощи [Одиночки](https://refactoring.guru/ru/design-patterns/singleton).

**Разница между Abstract Factory и Factory**

1. This is perhaps the easiest one but important certainly. Factory Method is used to *create one product* only but Abstract Factory is about *creating families of related or dependent products*.
2. Abstract Factory is one level *higher in abstraction* than Factory Method. Factory Method *abstracts the way objects are created*, while Abstract Factory also *abstracts the way factories are created* which in turn abstracts the way objects are created.
3. Factory Method depends on *inheritance* to decide which product to be created. In Creator class (in structure diagram), it has *other methods also* (implemented, to manipulate the product) which use the *only abstract method* *FactoryMethod() to create* the product and it *can only be implemented/changed by subclasses*.

# 71. Строитель (шаблон проектирования)

**Шаблон Builder** определяет процесс поэтапного конструирования сложного объекта, в результате которого могут получаться разные представления этого объекта.



**Преимущества и недостатки**

+ Позволяет создавать продукты пошагово.

+ Позволяет использовать один и тот же код для создания различных продуктов.

+ Изолирует сложный код сборки продукта от его основной бизнес-логики.

* Усложняет код программы из-за введения дополнительных классов.
* Клиент будет привязан к конкретным классам строителей, так как в интерфейсе строителя может не быть метода получения результата.

**Отношения с другими паттернами**

· Многие архитектуры начинаются с применения [Фабричного метода](https://refactoring.guru/ru/design-patterns/factory-method) (более простого и расширяемого через подклассы) и эволюционируют в сторону [Абстрактной фабрики](https://refactoring.guru/ru/design-patterns/abstract-factory), [Прототипа](https://refactoring.guru/ru/design-patterns/prototype) или [Строителя](https://refactoring.guru/ru/design-patterns/builder) (более гибких, но и более сложных).

· [Строитель](https://refactoring.guru/ru/design-patterns/builder) концентрируется на построении сложных объектов шаг за шагом. [Абстрактная фабрика](https://refactoring.guru/ru/design-patterns/abstract-factory) специализируется на создании семейств связанных продуктов. *Строитель* возвращает продукт только после выполнения всех шагов, а *Абстрактная фабрика* возвращает продукт сразу же.

· [Строитель](https://refactoring.guru/ru/design-patterns/builder) позволяет пошагово сооружать дерево [Компоновщика](https://refactoring.guru/ru/design-patterns/composite).

· Паттерн [Строитель](https://refactoring.guru/ru/design-patterns/builder) может быть построен в виде [Моста](https://refactoring.guru/ru/design-patterns/bridge): *директор* будет играть роль абстракции, а *строители* — реализации.

· [Абстрактная фабрика](https://refactoring.guru/ru/design-patterns/abstract-factory), [Строитель](https://refactoring.guru/ru/design-patterns/builder) и [Прототип](https://refactoring.guru/ru/design-patterns/prototype) могут быть реализованы при помощи [Одиночки](https://refactoring.guru/ru/design-patterns/singleton).

# 72. Фабричный метод (ШАБЛОН ПРОЕКТИРОВАНИЯ)

Порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс инстанциировать. Иными словами, Фабрика делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне. Также известен под названием виртуальный конструктор.

**Цель**:

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанциировать. Фабричный метод позволяет классу делегировать создание подклассов.

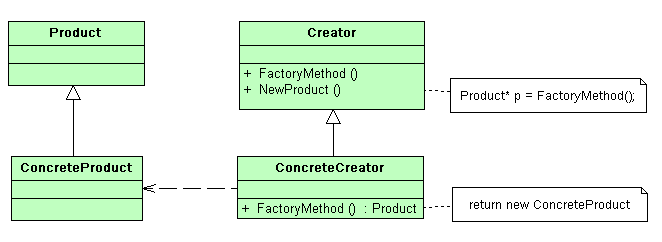
Используется, когда:

1) классу заранее неизвестно, объекты каких подклассов ему нужно создавать.

2) класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.

3) класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

**Структура:**

****

1. Product — продукт определяет интерфейс объектов, создаваемых абстрактным методом;
2. ConcreteProduct — конкретный продукт o реализует интерфейс Product;
3. Creator — создатель o объявляет фабричный метод, который возвращает объект типа Product. Может также содержать реализацию этого метода «по умолчанию»; o может вызывать фабричный метод для создания объекта типа Product;
4. ConcreteCreator — конкретный создатель o переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса ConcreteProduct.

Плюсы:

1. позволяет сделать код создания объектов более универсальным, не привязываясь к конкретным классам (ConcreteProduct), а оперируя лишь общим интерфейсом (Product);
2. позволяет установить связь между параллельными иерархиями классов.

Минусы:

1. необходимость создавать наследника Creator для каждого нового типа продукта (ConcreteProduct).

**Разница между Abstract Factory и Factory**

1. This is perhaps the easiest one but important certainly. Factory Method is used to *create one product* only but Abstract Factory is about *creating families of related or dependent products*.
2. Abstract Factory is one level *higher in abstraction* than Factory Method. Factory Method *abstracts the way objects are created*, while Abstract Factory also *abstracts the way factories are created* which in turn abstracts the way objects are created.
3. Factory Method depends on *inheritance* to decide which product to be created. In Creator class (in structure diagram), it has *other methods also* (implemented, to manipulate the product) which use the *only abstract method* *FactoryMethod() to create* the product and it *can only be implemented/changed by subclasses*.

# 73. Отложенная инициализация (ШАБЛОН ПРОЕКТИРОВАНИЯ)

Приём в программировании, когда некоторая ресурсоёмкая операция (создание объекта, вычисление значения) выполняется непосредственно перед тем, как будет использован её результат. Таким образом, инициализация выполняется «по требованию», а не заблаговременно. Аналогичная идея находит применение в самых разных областях: например, компиляция «на лету» и логистическая концепция «Точно в срок».

Частный случай ленивой инициализации — создание объекта в момент обращения к нему — является одним из порождающих шаблонов проектирования. Как правило, он используется в сочетании с такими шаблонами как Фабричный метод, Одиночка и Заместитель.

**Достоинства:**

1. Инициализация выполняется только в тех случаях, когда она действительно необходима;
2. ускоряется начальная инициализация.

**Недостатки:**

1. Невозможно явным образом задать порядок инициализации объектов;
2. возникает задержка при обращении к объекту, что может оказаться критичным при параллельном выполнении другой ресурсоёмкой операции. Вследствие этого требуется тщательно просчитывать целесообразность использования «ленивой» инициализации в многопоточных программных системах, особенно — [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0).

**Реализация:**

Реализовать данный шаблон можно несколькими различными вариантами:

* [Lazy Initialization](https://en.wikipedia.org/wiki/Lazy_loading#Lazy_initialization) – Инициализация по требованию. Это самый простой способ – реализовать проверку поля на null и в случае необходимости заполнять его данными.
* [Virtual Proxy](https://en.wikipedia.org/wiki/Lazy_loading#Virtual_proxy) – Виртуальный прокси-объект. Метод несколько усложнен проблемой идентификации объектов, т.к. вместо них, до инициализации, выступают заменители.
* [Ghost](https://en.wikipedia.org/wiki/Lazy_loading#Ghost) – Фиктивный объект, Призрак. Это реальный объект с неполным состоянием.
* [Value Holder](https://en.wikipedia.org/wiki/Lazy_loading#Value_holder) – Диспетчер значения. Объект является оболочкой для некоторого значения. Также не самый лучший вариант в связи с проблемами типизации.

# 74. Объектный пул (шаблон проектирования)

Порождающий шаблон проектирования, набор инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создаётся, а берётся из пула. Когда объект больше не нужен, он не уничтожается, а возвращается в пул.

**Применение:**

Объектный пул применяется для повышения производительности, когда создание объекта в начале работы и уничтожение его в конце приводит к большим затратам. Особенно заметно повышение производительности, когда объекты часто создаются/уничтожаются, но одновременно существует лишь небольшое их число.

**Переполнение:**

Если в пуле нет ни одного свободного объекта, возможна одна из трёх стратегий:

1. Расширение пула.

2. Отказ в создании объекта, аварийный останов. 3. В случае многозадачной системы, можно подождать, пока один из объектов не освободится.

**Примеры:**

1. Информация об открытых файлах в DOS.

2. Информация о видимых объектах во многих компьютерных играх (хорошим примером является движок Doom). Эта информация актуальна только в течение одного кадра; после того, как кадр выведен, список опустошается.

3. Компьютерная игра для хранения всех объектов на карте, вместо того, чтобы использовать обычные механизмы распределения памяти, может завести массив такого размера, которого заведомо хватит на все объекты, и свободные ячейки держать в виде связного списка. Такая конструкция повышает скорость, уменьшает фрагментацию памяти и снижает нагрузку на сборщик мусора (если он есть).

**Ловушки:**

1. После того, как объект возвращён, он должен вернуться в состояние, пригодное для дальнейшего использования. Если объекты после возвращения в пул оказываются в неправильном или неопределённом состоянии, такая конструкция называется объектной клоакой (англ. object cesspool).

2. Повторное использование объектов также может привести к утечке информации. Если в объекте есть секретные данные (например, номер кредитной карты), после освобождения объекта эту информацию надо затереть.

# 75. Прототип (шаблон проектирования)

## Вкратце

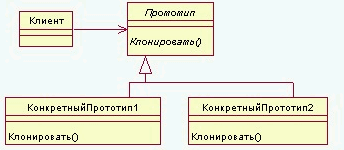
Объект создаётся посредством клонирования существующего объекта.

## Определение

Шаблон Прототип описывает способ построения новых объектов путём клонирования существующих объектов. Обычно объект создаётся при помощи вызова конструктора. Если же используется шаблон Прототип, то клиент знает только об интерфейсе или базовом классе, содержащем метод клонирования. Ре-альный класс объекта клиенту неизвестен. Прототипы для клонирования могут использоваться произвольное количество раз, сами они при операции клонирования меняться не должны. Хотя существуют различные варианты дизайна данного шаблона, наиболее гибким является вариант с менеджером прототипов, содержащим индексированный список доступных прототипов.

## Пример реализации:

Создавать новые объекты с помощью паттерна - прототипа. "Прототип" объявляет интерфейс для клонирования самого себя. "Клиент" создает новый объект, обращаясь к "Прототипу" с запросом клонировать "Прототип".



## Применение

Когда необходимый объект аналогичен уже существующему или когда создание с нуля дороже клонирования.

# 76. Получение ресурса есть инициализация (шаблон проектирования)

[Тыц](https://habr.com/en/sandbox/21603/)

## Вкратце

В конструкторе объект получает доступ к какому либо ресурсу (например, открывается файл или устанавливается соединение по сети к базе данных) и сохраняет описатель ресурса в закрытый члена класса, а при вызове деструктура этот ресурс освобождается (закрывается файл или соединение к БД). При объявлении объекта данного класса на стеке происходит и его инициализация с вызовом конструктора, захватывающий ресурс. При выходе из области видимости объект выталкивается из стека, но перед этим вызывается деструктор объекта, который и освобождает захваченный ресурс.

## Пример

Например, если мы хотим поговорить с кем-то по телефону, то при инициализации разговора мы снимаем трубку, при этом захватываем ресурс под названием “Телефонная Линия”. Вдоволь наговорившись с собеседником, мы прерываем разговор и кладем трубку и одновременно освобождаем телефонную линию.

## Плюсы и минусы

1. Обращение к ресурсу происходит в один этап. Либо мы получаем готовый полностью функциональный объект сразу, либо не получаем ничего.
2. Безопасность по отношению к исключению. Например, если после создания объекта и обращения к ресурсу произойдет исключение и мы перейдем к обработчику исключения, мы можем быть уверены что ресурс освободится без нашего участия. Даже если ресурсов несколько, мы уверены что все они будут корректно освобождены. В противном случае, если захватывать и освобождать ресурс вручную, то при возникновении исключения нужно учитывать, какие ресурсы уже захвачены, а какие – нет, и освобождать только использованные ресурсы, что не слишком просто.
3. Идиома очень удобна, когда нужно отслеживать важные ресурсы, а при этом сопровождение кода оставляет желать лучшего.
4. Часто при использовании нескольких ресурсов освобождать их следует в обратном порядке. При использовании идиомы RAII, вследствие того что объекты с захваченными ресурсами располагаются на стеке, их уничтожение происходит в обратном порядке, что как правило и является желательным.
5. Поддержка принципа DRY (Don’t Repeat Yourself). Код инициализации и освобождения ресурса содержится только в одном месте. Нет необходимости копировать и вставлять код инициализации в каждое место в программе где это необходимо. Достаточно просто создать объект.
6. При необходимости использовать дополнительные параметры для обращения к ресурсу (например, логин и пароль к БД) эти параметры могут быть переданы в качестве аргументов конструктора.
7. Накладные расходы при простейшей реализации обращения к ресурсу минимальны. В С++, как правило, при оптимизации компилятор реализует невиртуальные конструкторы и деструкторы в виде inline-функций.
8. Данная идиома применима только в языках с предсказуемым временем жизни объекта. Сюда относится, например, С++, а также языки с сборщиком мусора, где время жизни объекта определяется количеством ссылок на него, такие как Objective C.
9. Эта идиома неприменима в таких языках как Java или С#, где невозможно предсказать когда объект будет удален

# 77. Одиночка (шаблон проектирования)

## Вкратце

Шаблон позволяет удостовериться, что создаваемый объект — единственный в своём классе.

## Определение

Шаблон Одиночка гарантирует создание единственного экземпляра объекта некоторого класса и предоставляет точку доступа для получения этого экземпляра. Существование единственного объекта часто требуется при организации доступа к аппаратному обеспечению, реализации кэша, систем ведения отладочной информации и т. п.

## Пример

Создать класс и определить статический метод класса, возвращающий этот единственный объект.

public sealed class Singleton{

private static Singleton instance;

private Singleton(){ }

public static Singleton GetInstance(){

if (instance == null){

instance = new Singleton();

}

return instance;

}

}

## Проблемы

На самом деле шаблон «Одиночка» считается антипаттерном, не следует им слишком увлекаться. Он необязательно плох и иногда бывает полезен. Но применяйте его с осторожностью, потому что «Одиночка» вносит в приложение глобальное состояние, так что изменение в одном месте может повлиять на все остальные случаи использования, а отлаживать такое — не самое простое занятие. Другие недостатки шаблона: он делает ваш код сильно связанным (tightly coupled), а создание прототипа (mocking) «Одиночки» может быть затруднено.

# 78. Структурные шаблоны

## Определение

Показывают, как объекты и классы объединяются для образования сложных структур.Эти шаблоны в основном посвящены компоновке объектов (object composition). То есть тому, как сущности могут друг друга использовать. Еще одно объяснение: структурные шаблоны помогают ответить на вопрос «Как построить программный компонент?»

* адаптер (Adapter);
* мост (Bridge);
* компоновщик (Composite);
* декоратор (Decorator);
* фасад (Facade);
* приспособленец (Flyweight);
* заместитель (Proxy).

# 79. Адаптер (шаблон проектирования)

## Вкратце

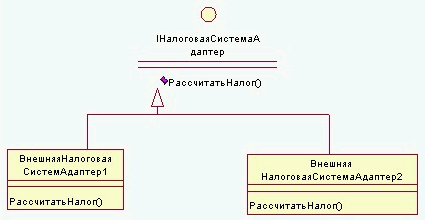
Шаблон «Адаптер» позволяет помещать несовместимый объект в обёртку, чтобы он оказался совместимым с другим классом.

## Определение

Адаптер — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

## Пример

Интеграция разрабатываемой системы с различными внешними системами учета налогов. Используются локальные программные объекты, обеспечивающие адаптацию (Адаптеры), при отправке сообщения к такому объекту выполняется обращение к внешней системе с использованием ее собственного программного интерфейса.



# 80. Мост (шаблон проектирования)

## Вкратце

Шаблон «Мост» — это предпочтение компоновки наследованию. Подробности реализации передаются из одной иерархии другому объекту с отдельной иерархией.

## Определение

Шаблон Мост позволяет разделить объект на абстракцию и реализацию так, чтобы они могли изменяться независимо друг от друга. При этом основной код, необходимый для функционирования объекта, переносится в реализацию. Всё остальное, включая взаимодействие с клиентом, содержится в абстракции. Методы абстракции при необходимости могут быть изменены или дополнены. Абстракция содержит экземпляр реализации и использует его для обработки поступающих от клиента запросов.

## Пример

Допустим, у вас есть сайт с несколькими страницами. Вы хотите позволить пользователям менять темы оформления страниц. Как бы вы поступили? Создали множественные копии каждой страницы для каждой темы или просто сделали отдельные темы и подгружали их в соответствии с настройками пользователей? Шаблон «Мост» позволяет реализовать второй подход.

# 81. Компоновщик (шаблон проектирования)

## Вкратце

Шаблон «Компоновщик» позволяет клиентам обрабатывать отдельные объекты в едином порядке.

## Определение

Шаблон Компоновщик позволяет упростить и стандартизировать взаимодействие между клиентом и группой объектов, представляющих древовидную структуру. Этот шаблон используется, если необходимо, чтобы клиент одинаково обращался как с составным объектом, так и с отдельными его частями.

## Пример

Необходимо учесть несколько скидок различных видов (зависят от времени, типа покупателя, типом выбранного продукта. Как применять политику ценообразования? Вырабатывается стратегия приоритета скидок, объект "Продажа" не должен обладать информацией о применяемых скидках, но можно было бы применить стратегию расчета скидок. Создается новый класс "РасчетСкидкиАлгоритмКомпозит".

# 82. Декоратор (шаблон проектирования)

**Задача**

Объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта.

**Способ решения**

Декоратор предусматривает расширение функциональности объекта без определения подклассов.

**Участники**

Класс ConcreteComponent — класс, в который с помощью шаблона Декоратор добавляется новая функциональность. В некоторых случаях базовая функциональность предоставляется классами, производными от класса ConcreteComponent. В подобных случаях класс *ConcreteComponent* является уже не конкретным, а абстрактным. Абстрактный класс *Component* определяет интерфейс для использования всех этих классов.

**Следствия**

1. Добавляемая функциональность реализуется в небольших объектах. Преимущество состоит в возможности динамически добавлять эту функциональность до или после основной функциональности объекта ConcreteComponent.

2. Позволяет избегать перегрузки функциональными классами на верхних уровнях иерархии

3. Декоратор и его компоненты не являются идентичными

**Реализация**

Создаётся абстрактный класс, представляющий как исходный класс, так и новые, добавляемые в класс функции. В классах-декораторах новые функции вызываются в требуемой последовательности — до или после вызова последующего объекта.

При желании остаётся возможность использовать исходный класс (без расширения функциональности), если на его объект сохранилась ссылка

## 

## 

# 83. Фасад (шаблон проектирования)

**Суть паттерна:**

Фасад — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

**Проблема**

Как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно сильное связывание с этой подсистемой или реализация подсистемы может измениться?

**Решение**

Определить одну точку взаимодействия с подсистемой — фасадный объект, обеспечивающий общий интерфейс с подсистемой, и возложить на него обязанность по взаимодействию с её компонентами. Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам. Фасадный объект обеспечивает реализацию GRASP паттерна Устойчивый к изменениям (Protected Variations) с точки зрения защиты от изменений в реализации подсистемы.

# 84. Приспособленец (шаблон проектирования)

**Вкратце**  
 Шаблон применяется для минимизирования использования памяти или вычислительной стоимости за счёт общего использования как можно большего количества одинаковых объектов.

**Определение**

Приспособленец — структурный шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым.

**Пример**

Приспособленец" объявляет интерфейс, с помощью которого приспособленцы могут получить внешнее состояние или как-то воздействовать на него, "КонкретныйПриспособленец" реализует интерфейс класса "Приспособленец" и добавляет при необходимости внутреннее состояние. Внутреннее состояние хранится в объекте "КонкретныйПриспособленец", в то время как внешнее состояние хранится или вычисляется "Клиентами" ("Клиент" передает его "Приспособленцу" при вызове операций).

Объект класса "КонкретныйПриспособленец" должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть независимым от контекста, "ПриспособленецФабрика" - создает объекты - "Приспособленцы" (или предоставляет существующий экземпляр) и управляет ими. "НеразделяемыйКонкретныйПриспособленец" - не все подклассы "Приспособленца" обязательно должны быть разделяемыми. "Клиент" - хранит ссылки на одного или нескольких "Приспособленцев", вычисляет и хранит внешнее состояние "Приспособленцев".

## 85. Заместитель (шаблон проектирования) Вкратце

С помощью шаблона «Заместитель» класс представляет функциональность другого класса.

**Определение**

Шаблон Заместитель позволяет контролировать доступ к заданному объекту, перехватывая все вызовы к этому объекту и прозрачно замещая его. Ни интерфейс, ни функциональность замещенного объекта с точки зрения клиента не меняются. Данный шаблон часто используется, если необходимо упростить или оптимизировать взаимодействие с объектом, скрывая несущественные для конкретной задачи подробности реализации.

Разновидности:

* Удалённый заместитель (remote proxy) обеспечивает связь с замещаемым
* объектом, который находится в другом адресном пространстве или на удалённой машине; Виртуальный заместитель (virtual proxy) реализует создание замещае-мого объекта только тогда, когда он действительно необходим;
* Защищающий заместитель (protection proxy) проверяет, имеет ли вызы-вающий объект необходимые для выполнения запроса права.
* Рассмотрим примера реализации виртуального заместителя

**Пример**

Есть несколько способов открыть дверь: например, она может быть открыта при помощи карты доступа или нажатия кнопки, которая обходит защиту. Основная функциональность двери — это открытие, но заместитель, добавленный поверх этого, добавляет функциональность: проверку пароля.

"Заместитель" может иметь и другие обязанности, а именно:

* удаленный "Заместитель" может отвечать за кодирование запроса и его аргументов и отправку закодированного запроса реальному "Субъекту",
* виртуальный "Заместитель" может кэшировать дополнительную информацию о реальном "Субъекте", чтобы отложить его создание,
* защищающий "Заместитель" может проверять, имеет ли вызывающий объект необходимые для выполнения запроса права.

# 86. Поведенческие шаблоны

Шаблоны поведения (behavioral patterns) – используются для организации, управления и объединения различных вариантов поведения объектов. Они связаны с присвоением обязанностей (responsibilities) объектам. От структурных шаблонов они отличаются тем, что не просто описывают структуру, но и очерчивают шаблоны передачи данных, обеспечения взаимодействия. То есть поведенческие шаблоны позволяют ответить на вопрос «Как реализовать поведение в программном компоненте?»

* [Цепочка ответственности](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfiles8fc3f4f858fc3f4f85cd240bf9628c2c6e5728e2cpng-cepochka-otvetstvennosti)
* [Команда](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfiles8d2e5082a8d2e5082afd747d4aee91ca0e48113c4png-komanda)
* [Итератор](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfiles625e57fc3625e57fc3cfa494da9b770f93f5322b0png-iterator)
* [Посредник](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfiles4358f59fc4358f59fcb634418a9dc22bce47b0720png-posrednik)
* [Хранитель](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfilesf1c4c72d2f1c4c72d22204ce08703d3a196fa9908png-hranitel)
* [Наблюдатель](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfilesd7bce8a7ed7bce8a7e9cd4ed989a08d5f21c16f75png-nablyudatel)
* [Посетитель](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfiles735be130d735be130d5514a59a834860596a6087bpng-posetitel)
* [Стратегия](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfiles0ff1d477c0ff1d477ce4d4e2ca81d427ce072a757png-strategiya)
* [Состояние](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfiles375200f5a375200f5aa9c444aa74eac8813c00af5png-sostoyanie)
* [Шаблонный метод](https://habrahabr.ru/company/mailru/blog/325492/#imagehttpshabrastorageorgfilesdc3e84492dc3e84492b0f47e0954fd7454fcad987png-shablonnyy-metod)

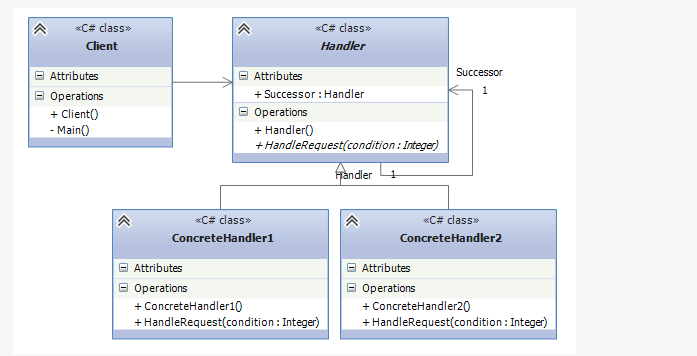
# 87. Цепочка ответственности (шаблон проектирования)

Цепочка Обязанностей (Chain of responsibility) - поведенческий шаблон проектирования, который позволяет избежать жесткой привязки отправителя запроса к получателю, позволяя нескольким объектам обработать запрос. Все возможные обработчики запроса образуют цепочку, а сам запрос перемещается по этой цепочке, пока один из ее объектов не обработает запрос. Каждый объект при получении запроса выбирает, либо обработать запрос, либо передать выполнение запроса следующему по цепочке.

## Когда применяется цепочка обязанностей?

* Когда имеется более одного объекта, который может обработать определенный запрос
* Когда надо передать запрос на выполнение одному из нескольких объект, точно не определяя, какому именно объекту
* Когда набор объектов задается динамически

UML-представление паттерна:



# 88. Команда (шаблон проектирования)

**Команда** (англ. *Command*) — поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

В объектно-ориентированном программировании шаблон проектирования Команда является поведенческим шаблоном, в котором объект используется для инкапсуляции всей информации, необходимой для выполнения действия или вызова события в более позднее время. Эта информация включает в себя имя метода, объект, который является владельцем метода и значения параметров метода.

Четыре термина всегда связаны с шаблоном Команда: команды (command), приёмник команд (receiver), вызывающий команды (invoker) и клиент (client). Объект Command знает о приёмнике и вызывает метод приемника. Значения параметров приёмника сохраняются в команде. Вызывающий объект (invoker) знает, как выполнить команду и, возможно, делает учёт и запись выполненных команд. Вызывающий объект (invoker) ничего не знает о конкретной команде, он знает только об интерфейсе. Оба объекта (вызывающий объект и несколько объектов команд) принадлежат объекту клиента (client). Клиент решает, какие команды выполнить и когда. Чтобы выполнить команду он передает объект команды вызывающему объекту (invoker).

Использование командных объектов упрощает построение общих компонентов, которые необходимо делегировать или выполнять вызовы методов в любое время без необходимости знать методы класса или параметров метода. Использование вызывающего объекта (invoker) позволяет ввести учёт выполненных команд без необходимости знать клиенту об этой модели учёта (такой учёт может пригодиться, например, для реализации отмены и повтора команд).

# 89. Интерпретатор (шаблон проектирования)

* Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.
* Отображает проблемную область в язык, язык – в грамматику, а грамматику – в иерархии объектно-ориентированного проектирования.

## Обсуждение паттерна Interpreter

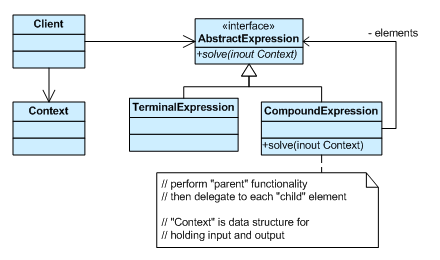
Паттерн Interpreter определяет грамматику простого языка для проблемной области, представляет грамматические правила в виде языковых предложений и интерпретирует их для решения задачи. Для представления каждого грамматического правила паттерн Interpreter использует отдельный класс. А так как грамматика, как правило, имеет иерархическую структуру, то иерархия наследования классов хорошо подходит для ее описания.

Абстрактный базовый класс определяет метод interpret(), принимающий (в качестве аргумента) текущее состояние языкового потока. Каждый конкретный подкласс реализует метод interpret(), добавляя свой вклад в процесс решения проблемы.

## Структура паттерна Interpreter

Паттерн Interpreter моделирует проблемную область с помощью рекурсивной грамматики. Каждое грамматическое правило может быть либо составным (правило ссылается на другие правила) либо терминальным (листовой узел в структуре ”дерево”). Для рекурсивного обхода ”предложений” при их интерпретации используется [паттерн Composite](http://cpp-reference.ru/patterns/structural-patterns/composite/).

### **UML-диаграмма классов паттерна Interpreter**



# 90. Итератор (шаблон проектирования)

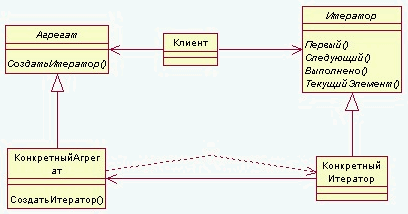
**Определение**

Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

**Пример**

Составной объект, например, список, должен предоставлять доступ к своим элементам (объектам), не раскрывая их внутреннюю структуру, причем перебирать список требуется по-разному в зависимости от задачи.

Создается класс "Итератор", коорый определяет интерфейс для доступа и перебора элементов, "КонкретныйИтератор" реализует интерфейскласса "Итератор" и следит за текущей позицией при обходе "Агрегата". "Агрегат" определяет интерфейс для создания объекта - итератора. "КонкретныйАгрегат" реализует интерфейс создания итератора и возвращает экземпляр класса "КонкретныйИтератор", "КонкретныйИтератор" отслеживает текущий объект в агрегате и может вычислить следующий объект при переборе.



## Преимущества и недостатки

* Упрощает классы хранения данных.
* Позволяет реализовать различные способы обхода структуры данных.
* Позволяет одновременно перемещаться по структуре данных в разные стороны.
* Не оправдан, если можно обойтись простым циклом.

# 91. Посредник (шаблон проектирования)

**Посредник** —паттерн, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

**Использование паттерна Mediator**

* Определите совокупность взаимодействующих объектов, связанность между которыми нужно уменьшить.
* Инкапсулируйте все взаимодействия в абстракцию нового класса.
* Создайте экземпляр этого нового класса. Объекты-коллеги для взаимодействия друг с другом используют только этот объект.
* Найдите правильный баланс между принципом слабой связанности и принципом распределения ответственности.
* Будьте внимательны и не создавайте объект-"контроллер" вместо объекта-посредника.

# 92. Хранитель (шаблон проектирования)

**Хранитель** — паттерн, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

**Применение**

**Шаблон Хранитель используется, когда:**

* необходимо сохранить снимок состояния объекта (или его части) для последующего восстановления
* прямой интерфейс получения состояния объекта раскрывает детали реализации и нарушает [инкапсуляцию](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F) объекта

***Особенности паттерна Хранитель***

*Сохранение границ инкапсуляции.* Хранитель позволяет избежать раскрытия информации, которой должен распоряжаться только хозяин, но которую тем не менее необходимо хранить вне последнего.

*Упрощение структуры хозяина.* При других вариантах дизайна, направленного на сохранение границ инкапсуляции, хозяин хранит внутри себя версии внутреннего состояния, которое запрашивали клиенты. Таким образом, вся ответственность за управление памятью лежит на хозяине.

*Значительные издержки при использовании Хранителей. С* Хранителями могут быть связаны заметные издержки, если хозяин должен копировать большой объем информации для занесения в память Хранителя или если клиенты создают и возвращают Хранителей достаточно часто.

*Определение «узкого» и «широкого» интерфейсов.* В некоторых языках сложно гарантировать, что только хозяин имеет доступ к состоянию Хранителя.

*Скрытая плата за содержание Хранителя.* Посыльный отвечает за удаление Хранителя, однако не располагает информацией о том, какой объем информации о состоянии скрыт в нем. Поэтому нетребовательный к ресурсам посыльный может расходовать очень много памяти при работе с Хранителем.

# 93. Наблюдатель (шаблон проектирования)

Наблюдатель определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

При реализации шаблона «наблюдатель» обычно используются следующие классы.

**Observable** — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей.

**Observer** — интерфейс, с помощью которого наблюдаемый объект оповещает наблюдателей.

**ConcreteObservable** — конкретный класс, который реализует интерфейс Observable.

**ConcreteObserver** — конкретный класс, который реализует интерфейс Observer.

**Область применения**

Шаблон «наблюдатель» применяется, когда система обладает свойствами:

* существует, как минимум, один объект, рассылающий сообщения
* имеется не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения.
* нет надобности очень сильно связывать взаимодействующие объекты, что полезно для повторного использования.
* Данный шаблон часто применяют в ситуациях, в которых отправителя сообщений не интересует, что делают получатели с предоставленной им информацией.

# 94. Состояние (шаблон проектирования)

**Состояние** — паттерн, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создается впечатление, что изменился класс объекта.

Паттерн состоит из 3 блоков:

**Widget** — класс, объекты которого должны менять свое поведение в зависимости от состояния.

**IState** — интерфейс, который должно реализовать каждое из конкретных состояний. Через этот интерфейс объект Widget взаимодействует с состоянием, делегируя ему вызовы методов. Интерфейс должен содержать средства для обратной связи с объектом, поведение которого нужно изменить. Для этого используется событие (паттерн Publisher — Subscriber). Это необходимо для того, чтобы в процессе выполнения программы заменять объект состояния при появлении событий. Возможны случаи, когда сам Widget периодически опрашивает объект состояние на наличие перехода.

**StateA** … **StateZ** — классы конкретных состояний. Должны содержать информацию о том, при каких условиях и в какие состояния может переходить объект из текущего состояния. Например, из StateA объект может переходить в состояние StateB и StateC, а из StateB — обратно в StateA и так далее. Объект одного из них должен содержать Widget при создании.

# 95. Стратегия (шаблон проектирования)

-поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

**Причины**:

1)Программа должна обеспечивать различные варианты алгоритма или поведения

2)Необходимо изменять поведение объектов на стадии выполнения

3)Введение интерфейса позволяет классам-клиентам ничего не знать о классах, реализующих этот интерфейс и инкапсулирующих в себе конкретные алгоритмы

4)Нужно изменять поведение каждого экземпляра класса

**Следствие:**

1)Шаблон Strategy определяет семейство алгоритмов.

2)Это позволяет отказаться от использования переключателей и/или условных операторов.

3)Вызов всех алгоритмов должен осуществляться стандартным образом (все они должны иметь одинаковый интерфейс).

**Результат:**

Класс, который использует алгоритм (Context), включает абстрактный класс (Strategy), обладающий абстрактным методом, определяющим способ вызова алгоритма. Каждый производный класс реализует один требуемый вариант алгоритма.

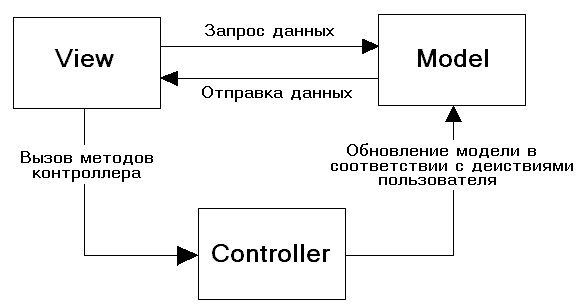
# 96. Шаблоны параллельного программирования

Используются для более эффективного написания многопоточных программ, и предоставляет готовые решения проблем синхронизации.

* Pipeline (конвейер)
* Master and Slave (хозяин и подчинённый)
* Divide and Conquer (разделяй и властвуй)
* Geometric Decomposition (геометрическая декомпозиция)
* Replicable (воспроизводимый)
* Repository (хранилище, контейнер)

# 97. Модель-представление-контроллер (шаблон проектирования)

Шаблон MVC описывает простой способ построения структуры приложения, целью которого является отделение бизнес-логики от пользовательского интерфейса. В результате, приложение легче масштабируется, тестируется, сопровождается и конечно же реализуется.

  
 **Модель** — содержит бизнес-логику приложения и включает методы выборки (это могут быть методы ORM), обработки (например, правила валидации) и предоставления конкретных данных, что зачастую делает ее очень толстой, что вполне нормально.

Модель не должна напрямую взаимодействовать с пользователем. Все переменные, относящиеся к запросу пользователя должны обрабатываться в контроллере.

Модель не должна генерировать HTML или другой код отображения, который может изменяться в зависимости от нужд пользователя. Такой код должен обрабатываться в видах.

Одна и та же модель, например: модель аутентификации пользователей может использоваться как в пользовательской, так и в административной части приложения. В таком случае можно вынести общий код в отдельный класс и наследоваться от него, определяя в наследниках специфичные для подприложений методы.

**Вид** — используется для задания внешнего отображения данных, полученных из контроллера и модели.

Виды cодержат HTML-разметку и небольшие вставки PHP-кода для обхода, форматирования и отображения данных.

Не должны напрямую обращаться к базе данных. Этим должны заниматься модели.

Не должны работать с данными, полученными из запроса пользователя. Эту задачу должен выполнять контроллер.

Может напрямую обращаться к свойствам и методам контроллера или моделей, для получения готовых к выводу данных.

Виды обычно разделяют на общий шаблон, содержащий разметку, общую для всех страниц (например, шапку и подвал) и части шаблона, которые используют для отображения данных выводимых из модели или отображения форм ввода данных.

**Контроллер** — связующее звено, соединяющее модели, виды и другие компоненты в рабочее приложение. Контроллер отвечает за обработку запросов пользователя. Контроллер не должен содержать SQL-запросов. Их лучше держать в моделях. Контроллер не должен содержать HTML и другой разметки. Её стоит выносить в виды.

В хорошо спроектированном MVC-приложении контроллеры обычно очень тонкие и содержат только несколько десятков строк кода. Чего, не скажешь о Stupid Fat Controllers (SFC) в CMS Joomla. Логика контроллера довольно типична и большая ее часть выносится в базовые классы.

Модели, наоборот, очень толстые и содержат большую часть кода, связанную с обработкой данных, т.к. структура данных и бизнес-логика, содержащаяся в них, обычно довольно специфична для конкретного приложения.

# 98. Технология использования шаблонов проектирования

Разбиение задачи на минимальные блоки и использование шаблонов для достижения определенных целей.

# 99. Идиомы программирования и шаблоны проектирования

Шаблон проектирования, или паттерн, в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования, в рамках некоторого часто возникающего контекста.

**Идиомы**, являясь низкоуровневыми паттернами, имеют дело с вопросами реализации какой-либо проблемы с учетом особенностей данного языка программирования. При этом часто одни и те же идиомы для разных языков программирования выглядят по-разному или не имеют смысла вовсе. Например, в C++ для устранения возможных утечек памяти могут использоваться интеллектуальные указатели. Интеллектуальный указатель содержит указатель на участок динамически выделенной памяти, который будет автоматически освобожден при выходе из зоны видимости. В среде Java такой проблемы просто не существует, так как там используется автоматическая сборка мусора. Обычно, для использования идиом нужно глубоко знать особенности применяемого языка программирования.