Технология

1. Преимущества и недостатки структурного программирования.

Логически-связанные функции находятся визуально ближе, а слабо связанные — дальше, что позволяет обходиться без блок-схем и других графических форм изображения алгоритмов (по сути, сама программа является собственной блок-схемой).

Сильно упрощается процесс тестирования и отладки программ.

Нисходящая разработка даёт возможность на ранних этапах согласовывать с заказчиком прототип. В случае недовольства, придётся переписывать минимум кода.

Нисходящая декомпозиция позволяет совмещать проектирование и кодирование.

Начиная разработку с верхних уровней, избавляемся от логических ошибок.

Нисходящая разработка, сквозной структурный контроль. Использование базовых логических структур.

Дейкстра, Милдс выделили три идеи структурного программирования:

1. нисходящая разработка
2. использование базовых логических структур
3. сквозной структурный контроль

Нисходящая разработка

Этапы создание программного продукта:

1. Анализ. *(Оцениваем задачу, переработка ТЗ)*
2. проектирование
3. кодирование
4. тестирование
5. сопровождение
6. модификация

2-4 используют нисходящий подход. Используются алгоритмы декомпозиции – разбиение задачи на подзадачи, выделенные подзадачи разбиваются дальше на подзадачи, формируется иерархическая структура (данные нисходящие, логика восходящая, разработка нисходящая). Логика поднимается на более высокий уровень. Данные на низком уровне, на высшем логика. Для каждой полученной подзадачи создаем отладочный модуль. Готовятся тестирующие пакеты (до этапа кодирования). Подзадача не принимает решения за модуль уровнем выше (функция отработала, вернула результат, а потом анализируется). Все данные должны передаваться явно. Блок, функция, файл – уровни абстракции. Ограничения вложенности – 3 (глубина вложенности), если больше, то выделить подфункции.

Базовые структуры

Майер: Любой алгоритм можно реализовать с помощью трех логических структур: следование, развилка, ветвление.

Развилка: выбор между двумя альтернативами, множественный выбор switch – ветви имеют const выражения.

Повторение: while, until, for, безусловный цикл loop

Сквозной структурный контроль

Организация контрольных сессий. На контрольную группу никогда не присутствует начальство (руководство) иногда приглашаются умные люди со стороны. Количество замечаний не влияет на программиста. Задачи контрольной сессии – выявить недостатки на ранних стадиях. Готовят плакаты с алгоритмами и архитектурными решениями.

Недостатки

1. Все глобальные переменные доступны всем функциям проекта. Большое число связей между функциями и данными, в свою очередь, также порождает несколько проблем. Во-первых, усложняется структура программы. Во-вторых, в программу становится трудно вносить изменения. Изменение структуры глобальных данных может потребовать переписывания всех функций, работающих с этими данными.
2. Разделение данных и функций, являющееся основой структурного подхода, плохо отображает картину реального мира - отделение данных от функций оказывается малопригодным для отображения картины реального мира. В реальном мире нам приходится иметь дело с физическими объектами, такими, например, как люди или машины. Эти объекты нельзя отнести ни к данным, ни к функциям, поскольку реальные вещи представляют собой совокупность свойств и поведения.

2. Технология ООП. Преимущества и недостатки ООП.

Объектно-ориентированное проектирование — парадигма программирования, основанная на принципах абстрагирования, инкапсуляции, модульности, иерархичности, типизации, параллелизма и устойчивости.

Чтобы не изменять все типы данных, давайте это сделаем изначально. Есть данные -> выделяем действия над этими данными. (Принцип *инкапсуляции*)
Не будем вносить изменение в рабочий код. Делаем надстройки над рабочим функционалом. (Принцип *наследования*.)
Проблема: программа развалилась на многие мелкие куски. Перенесём из физического мира взаимодействие между объектами. (*взаимодействие*). Акцессорное и событийное.

Объект – конкретная реализация абстрактного типа, обладающий характеристиками состояния, поведения, индивидуальности.
Состояние – один из возможных вариантов условий существований объекта.
Поведение – описание объекта в терминах изменения его состояния и передача сообщений (данных) в процессе воздействия.
Индивидуальность – сущность объекта, отличающееся от других объектов.

Модель Мура

- Состоит из множества *состояний*, каждое состояние представляет стадию в жизненном цикле типичного экземпляра.
- Из множества *событий*: каждое событие представляет собой инцидент или указание на то, что происходит эволюционирование.
- Из (множества) *правил перехода* определяет какое новое состояние получает в следствие какого-нибудь события (событие может и не изменять объект)
- Из *действий* – деятельность или операция который должен быть выполнены над объектом чтобы он мог достичь состояния (каждому действию соответствует состояние).

Категории объектов
Реальные объекты – абстракция фактического существующего объекта реального мира.
Роли – абстракции цели или назначения человека, части оборудования или организации.
Инциденты – абстракция чего-то происшедшего или случившегося (наводнение, скачёк напряжения, выборы).
Взаимодействия – объекты, получаемые из отношений между другими объектами (перекресток, договор, взятка).
Спецификации – используется для представления правил, критериев качества, стандартов (правила дорожного движения, распорядок дня).

Недостатки ООП
Производительность программ.

Преимущества ООП

«Более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку.
Сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями.
Увеличивается показатель повторного использования кода.

3. Понятия ООП: инкапсуляция, наследования, полиморфизм. Объекты, классы, домены, отношения между ними.

Инкапсуляция — это свойство системы, позволяющее объединить данные и методы, работающие с ними в классе, и скрыть детали реализации от пользователя. Разумная инкапсуляция позволяет локализовать части реализации программной системы, которые могут подвергнуться изменениям. Например, по мере развития программного продукта, разработчики могут принять решение изменить внутреннее устройство тех или иных. При этом, важным преимуществом наличия механизмов разграничения доступа (механизмов инкапсуляции), является возможность внесения изменений в реализацию объекта (класса) без изменения других объектов (классов).

Наследование — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом.

Полиморфизм — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Объект — это сущность, способная сохранять свое состояние (информацию) и обеспечивающая набор операций (поведение) для проверки и изменения этого состояния.

В ООП — это модель или абстракция реальной сущности в программной системе. Предмет моделирования при построении объекта в ООП может быть различным. Например, могут существовать следующие типы абстракции, используемые при построении объекта:

абстракция понятия: объект — это модель какого-то понятия предметной области;
абстракция действия: объект объединяет набор операций для выполнения какой-либо функции;
абстракция виртуальной машины: объект объединяет операции, которые используются другими, более высокими уровнями абстракции;
случайная абстракция: объект объединяет не связанные между собой операции.

Отношения между объектами

Отношения использования (*старшинства*) - каждый объект включается в отношения. Может играть 3 роли:

1. Объект воздействия – объект может воздействовать на другие объекты, но сам не поддается воздействию (активный объект).
2. Исполнение – объект может только подвергаться управлению, но не выступает в роли воздействующего (пассивный объект).
3. Посредники – такой объект может выступать в роли воздействующего, так и в роли исполнителя (создаются для помощи воздействующим). Чем больше посредников, тем легче модифицировать программу.

Все отношения между объектами могут быть сведены к двум типам:
Ассоциация (связь) — отношение, позволяющее реализовать взаимодействие клиент-сервер.
Агрегация (композиция) — отношение, служащее для определения понятия целое-часть и организации иерархий объектов.
Отношения *включения* – один объект включает другие объекты.

Класс – такая абстракция множества предметов реального мира, что все предметы в этом множестве имеют одни и те же характеристики, все экземпляры подчинены и согласованы с одними и те же набором правил и линией поведения.

Отношения между классами

Включение – класс может включать в себя другой класс, если он определяет атрибуты, являющиеся объектами другого класса.
Ассоциация – станавливает двустороннюю связь между объектами разных классов.
Наследование – на основе одного класса, мы строим новый класс, путем добавления новых характеристик и методов.
Использование – один класс вызывает методы другого класса. (акцессор) Простейший пример — использование объектов другого класса в качестве аргументов методов.
Метаклассы – класс, существующий для создания других классов (это класс, экземпляры которого в свою очередь являются классам).

Домены – отдельный, реальный, гипотетически и абстрактный мир населенный отчетливым набором объектов, которые ведут себя в соответствии с предусмотренным доменом правилами и линиями поведения. Каждый домен образует отдельное и связное единое целое. Класс определяется в одном соответствующем домене. Класс четко определен в одном домене. Классы в одном домене не требуют наличия классов в других доменах.

4. Этапы разработки ПО с использованием объектно-ориентированного подхода. Анализ, проектирование. Эволюция и модификация.

1. Анализ – построение модели системы. На основе построенной модели выполняется проектирование.
2. Проектирование – перевод модели в проектные документы; с помощью CASE-средств.
3. Эволюция – написание кода, тестирование, итерирование - ОО подход за счет рекурсивного дизайна, система разворачивается и усложняется.
4. Модификация – эволюционирование на готовом продукте, сданном заказчику.

Преимущества эволюции

1. Система постоянно развивается, но всегда готова – обеспечивается обратная связь с заказчиком
2. Предоставляются разные версии системы – можно откатываться на шаг назад; можно пускать разные версионные ветки
3. Заказчик постоянно видит результат
4. Серьёзных ошибок не возникает за счет постоянного взаимодействия с заказчиком
5. Хорошо распределяется время при проектировании
6. Размеры системы (кода) в принципе не ограничиваются; при структурном же подходе чрезвычайно большая программа при большом количестве разработчиков порождает проблемы их взаимодействия.

Изменения на этапе эволюции

1. Добавление класса
2. Изменение реализации класса
3. Изменение представления класса
4. Реорганизация структуры класса
5. Изменение интерфейса класса (самое страшное, тянет за собой кучу изменений в основном коде)

Задача анализа – довести модель до такого состояния, чтобы дальше не понадобилось изменять интерфейс.

5. Рабочие продукты объектно-ориентированного анализа и проектирования.

Результаты проектирования

- ПО
  - схема домена
  - проектная матрица
- класс
  - модель состояний
  - диаграмма потоков данных действий
- домен
  - модель связи подсистемы
  - модель взаимодействия подсистемы
  - модель доступа к подсистемам
- процесс
  - описание
  - псевдокод процесса
- подсистема
  - информационная модель (получаем описание объектов, атрибутов, связей).
  - модель взаимодействия объектов (получаем список событий).
  - модель доступа таблица процессов состояний для всей подсистемы.

6. Объектно-ориентированный анализ. Концепция информационного моделирования. Понятия классов, атрибутов и связей. Формализация связей.

Система разбивается на домены – интерфейс, реализация, сама задача, сервисные функции (например, коммуникация) и так далее. В домене четко выделяются слабо связанные группы сильно связанных классов, таким образом разбивая его на подсистемы. Сначала проектируются подсистемы, затем реализуются связи внутри них, затем между ними.

Информационное моделирование. Атрибуты класса, связи.
Информационное моделирование – выделить сущности, с которыми надо работать, и их характеристики, описать и выявить связь между сущностями, попытаться её реализовать.

Для каждой подсистемы:

1. Строится описание классов, атрибутов, связей между классами.
2. Также строится модель взаимодействия объектов (событийная).
3. Модель доступа к объектам.
4. Таблица процессов состояний.

Для каждого класса - Выделяем модель состояний (модель переходов состояний).
Для каждого состояния каждой модели состояний - Строится диаграмма потоков данных действий (ДПДД).
Для каждого действия (процесса) - Делается описание процесса.

Атрибуты

Атрибут – каждая отдельная характеристика, являющаяся общей для всех возможных экземпляров классов. Каждый атрибут задаётся множеством значений, которое важно определить при анализе для определения в дальнейшем типе атрибута.

Каждый объект нужно идентифицировать. Идентификатор – то, что чётко определяет конкретный объект. Может состоять из одного или нескольких атрибутов. Изменение атрибута не приводит к изменению самого объекта – объект просто меняет «имя», но остаётся тем же самым.

Описательные атрибуты. – характеристика, присущая каждому экземпляру класса (вес студента).
Указывающие атрибуты − идентификатор или часть идентификатора (имя студента)
Вспомогательные атрибуты − формализуют связи одного объекта с другими объектами; для активных объектов это время жизни; атрибуты состояния объектов для динамических объектов.

Правила атрибутов:

1. Один экземпляр класса имеет одно единственное значение для каждого атрибута в любой момент времени. Не может быть два значения, или не быть совсем. Если появляется объект с неопределенным значением атрибута, то относить его к этому классу мы не можем – это уже объект другого класса.
2. Атрибут не должен содержать никакой внутренней структуры.
3. Когда объект имеет составной идентификатор, каждый атрибут, являющийся частью идентификатора, представляет характеристику всего объекта, а не его части, и тем более ничего другого.
4. Каждый атрибут, не являющийся частью идентификатора, представляет характеристику экземпляра, указанного идентификатором, а не характеристику другого атрибута-идентификатора. Не должно быть атрибутов, описывающих только часть объекта (атрибут описывающий другой атрибут).

Связи

Связь – это абстракция набора отношений, которые систематически возникают между различными видами реальных объектов.
Задаём связь из перспективы каждого участвующего объекта.
Каждой связи присваивается уникальный идентификатор, который состоит из буквы и номера.
Необходимо выявить множественность связей. С каждой стороны может участвовать один объект, один со многими, или многие со многими.

Иногда с какой-то стороны объект МОЖЕТ не участвовать в связи, тогда связь – условная (преподаватель не руководит студентами – условная связь со стороны преподавателя, т.к. у студента должен быть руководитель, а преподаватель не обязан руководить).

Если с обеих сторон объекты МОГУТ не участвовать в связи, то связь – биусловная (УчебныйКурс – Студент; курс не читается в этом семестре, или студент не выбрал этот курс).

Формализация связей

При формализации связи, вспомогательный атрибут добавляется к одному из объектов в связи.

При связи ОКМ добавляется атрибут со стороны многих – поскольку атрибут не может содержать внутреннюю структуру. Если один из объектов удаляется, то второй, участвующий в связи, тожк удаляется или изменяется, например, при смерти жены, муж становится вдовцом или холостяком.

При связи МКМ добавляется так называемый ассоциативный класс, который формализует связь, то есть хранит идентификаторы связанных классов.

Любую связь, имеющую динамическое поведение, нужно формализовывать ассоциативным объектом. Жена – муж, ассоциативный объект – свидетельство о браке

В результате ИМ получим

- Диаграмма информационных структур
- Описание объектов и их атрибутов
- Связи и их формализация

7. Динамическое поведение объектов, понятия состояний, событий, действий, жизненный цикл.

Модель Мура состоит из:

- Множество состояний
- Множество событий (инценденты)
- Правила перехода
- Действия

Множество состояний задается либо через ДПС и/или ТПС.

ДПС (Диаграмма переходов состояний) – состоит из прямоугольников, соединенных стрелками, над каждой стрелкой – событие, которое осуществляет переход из одного состояния в другое.

ТПС (Таблица переходов состояний) представляет собой матрицу: строки – это состояния, столбцы – это события. Вся матрица должна быть заполнена. Каждая ячейка – это новое состояние, в которое можно перейти из данного состояния по этому событию. Если событие игнорируется, ставят прочерк. Если событие в данном состоянии не может произойти, то ставят крестик.

Состояние – это положение объектов, в котором определяются определённый набор правил, линий поведений, предписаний, определённых законов. Каждому состоянию ставится в соответствие уникальные имя или номер, в соответствии с отношением.

Виды состояний:

- Создание
- Заключительное – экземпляр становится неподвижным (нет перехода в другие состояния) или объект прекращает своё существование. Рисуется пунктирной линией.
- Текущее (Состояние в котором объект может находиться, но которое не является заключительным или начальным.)

При составлении модели делают *анализ отказов* (то есть анализ аномального поведения объекта).

Нужно определить/проверить:

1. Какой инцидент в реальном мире мог бы вынудить объект к переходу в некорректное состояние.
2. Гарантируется ли наступление каждого события.
3. Если событие происходит, то нормально ли это.
4. Гарантируется ли что событие происходит своевременно (когда объект находится в нужном состоянии).

Событие – это абстракция инцендента или сигнала в реальном мире, которое сообщает нам о перемещении чего-либо в новое состояние.

- Значение события - короткая фраза, которая сообщает нам, что происходит с объектом в реальном мире.
- Предназначение - это модель состояний, которое принимает событие, может быть один единственный приёмник, для данного события.
- Метка – уникальная метка должна обеспечиваться для каждого события. Внешние события помечаются буквой «E»
- Данные события – данные, сопровождающие событие.

Существует ряд правил, связывающих события и данные.

1. Правило тех же данных – все события, которые вызывают переход в определенное состояние, должны нести одни и те же данные.
2. Правило состояний несоздания – если событие переводит объект из состояния в состояние, то оно должно переносить идентификатор объекта.
3. Правило состояния создания – событие, переводящее объект в состояние создания, не несёт его идентификатора.

Модель состояний скачкообразная. Плавный переход из состояния в состояние можно осуществить, добавив несколько промежуточных состояний.

Каждому событию ставится в соответствие действие.

Действие – это деятельность или операция, которая выполняется при достижении объектом состояния. С каждым состоянием связано только одно действие. Действие должно выполняться любым объектом одинаково.

Действие может:

1. Выполнять любые вычисления
2. Порождать события для любого объекта любого класса
3. Порождать события для чего-либо вне области анализа
4. Выполнять все действия над таймером - создавать\удалять\устанавливать\считывать.
5. Читать, записывать атрибуты собственного класса и других классов

Еще несколько правил:

1. Действие не должно оставлять данные противоречивыми.
2. Действие не должно оставлять противоречивыми связи (если был удалён объект, то нужно позаботиться и об удалении\изменении связанных с ним объектов).
3. Только одно действие может выполняться в данный момент для конкретного объекта; но для разных – действия могут выполняться одновременно.

Связь событий и действий:

1. События никогда не теряются.
2. Если событие порождено для объекта, который в этот момент уже выполняет действие, то это событие не будет принято, пока действие не будет завершено.
3. Каждое событие прекращается, когда оно представляется конечному автомату (?), само событие исчезает как событие.
4. Не все события обрабатываются, некоторые из них могут быть игнорированы.

<u>Формы жизненных циклов</u>

1. Циркуляционный
2. Линейный – (рождение-смерть)

Когда формируются жизненные циклы:

1. Создание или уничтожение во время выполнения

2. Миграция между подклассами
3. Накопление атрибутов – когда с изменением значения атрибута меняется поведение объекта (например, человек с возрастом меняет свое поведение)
4. Объект производится или возникает поэтапно (сборка машины на конвеере)
5. Если возникают объекты с жизненным циклом, и мы хотим реализовать асинхронную (т.е. событийную) схему взаимодействия, то задача или запрос тоже имеют жизненный цикл.
6. Если для какого-то объекта есть жизненный цикл, и он имеет безусловную связь с другим объектом (пассивным), то жизненный цикл нужно выделить и для этого пассивного объекта.

8. Динамика систем, модель взаимодейтсвия объектов. Схемы взаимводействия объектов в подсистеме. Каналы управления. Имитирование.

МВО (модель взаимодействия объектов) – графическое представление взаимодействия между моделями состояний и внешними сущностями. Каждая модель состояний – овал, сущность – прямоугольник (называется терминатором). События, которые порождаются одной моделью состояний для другой или терминатором, рисуются стрелкой. События могут быть направлены к терминаторам.

МВО формируется иерархически – объекты, наиболее осведомленные о всей системе (активные) располагаются вверху диаграммы. Если событие приходит извне к МВО, находящимся вверху, терминаторы рисуются вверху – терминаторы верхнего уровня. Если события уходят или приходят к МВО нижнего уровня, терминаторы рисуются снизу. Может быть схема верхнего и нижнего управления – система ограничена терминаторами сверху или снизу.

Типы событий в МВО

1. Внешние события (приходят от терминатора или уходят к нему)
    a. Незапрашиваемые события (не являются результатом предыдущих действий подсистемы, то есть это события управляющие, при этом не переводят объект в новое состояние)
    b. Запрашиваемые события (являются результатом действий, переводят объект в новое состояние)
2. Внутренние (соединяют одну модель состояний с другой)
    a. Схема верхнего управления (Эти события приходят от верхних терминаторов)
    b. Схема нижнего управления (Эти события приходят от нижних терминаторов)

Канал управления – последовательность действий и событий, которые происходят в ответ на поступление некоторого незапрашиваемого состояния, в то время как система находится в определённом состоянии. Если возникло событие к терминатору и это событие приведет к дальнейшим событиям от терминатора, то мы его тоже включаем в канал управления. Канал управления должен быть конечным.

2 времени имитирования:

1. Время выполнения действия
2. Время задержки – время, на протяжении которого объект должен находиться в состоянии (невозможен резкий переход из одного состояния в другое, мы должны учитывать время задержки)

Этапы имитирования (тесты)

1. Установить начальное состояние системы
2. В любом состоянии проверить, как подсистема будет реагировать на все незапрашиваемые события, и построить соответствующие каналы управления.
3. Оценить конечный результат

9. Диаграммы потоков данных действий. Понятия процессов и потоков управления. Модель доступа к объектам.

ДПДД (Диаграмма потоков данных действий) – обеспечивает графическое представление модулей процесса в пределах действия и взаимодействия между ними. Строится для каждого действия каждого состония каждой модели состояний. На диаграмме каждый процесс рисуется овалом, внешняя сущность – прямоугольником. Процессы могут получать данные от других процессов и от каких-либо внешних сущностей.

Разбиваем действия на процессы, которые могут происходить:

1. Процесс проверки – условные переходы (У)
2. Процесс преобразования – выполняют вычисления (В)
3. Аксессоры (процесс, чья единственная цель состоит в том, чтобы получить доступ к данным одного архива данных): создание, чтение, запись, уничтожение (А)
4. Генераторы событий (создаёт лишь одно событие) (Г)

Каждый процесс нужно именовать и описывать. Аксессоры – какие атрибуты считывают или записывют, какие объекты создают или уничтожают. Генераторы событий – результат-событие, метка события. Преобразования – что делают. Проверки – «проверить, что...»

Правила выполнения для ДПДД:

1. Процес может выполняться, когда всех входы доступны.
2. Выводы процесса доступны, когда он завершает своё выполнение.
3. Данные событий (на диаграмме это стрелка сверху), данные из архивов данных и терминаторов всегда доступны.

Все процессы в подсистеме объединяются в единую таблицу.

Id процесса	Тип	Название процесса	Где используется	
			Модель состояний	Действие

Модель доступа

Модель доступа строится на основе выделенных аксессорных процессов. В ней модели состояний (объектов) рисуются вытянутыми овалами. Если А использует процесс-аксессор модели состояний В, то рисуется стрелка от А к В, А будет аксессором. Аксессоры реализуются добавлением в объект действий по записи и чтению атрибутов.

10. Модели доменного уровня, понятие мостов, клиентов, серверов.

Домен – отдельный, реальный, гипотетически и абстрактный мир населенный отчетливым набором объектов, которые ведут себя в соответствии с предусмотренным доменом правилами и линиями поведения. Каждый домен образует отдельное и связное единое целое.

Типы доменов:

1. Прикладные домены (основные домены, которые решают непосредственно задачу)
2. Сервисные домены (набор сервисов, которые будет использовать прикладной домен)
3. Архитектурные домены (обеспечивает общие механизмы и структуры для управления всей системой, как единым целым)
3. Реализации (библиотечные функции – взаимодействие по сети, протоколы взаимодействия по сети, дают возможность легкой замены одной реализации на другую)

Когда один домен использует возможности и механизмы другого, говорят, что существует *мост*.
Сервер – домен, предоставляющий эти возможности, *клиент* – использующий

Клиент рассматривает мост как набор предложений, которые будут представлены другим доменом.
Сервер подходит к мосту как к набору требований для выполнения.

Схема доменов.

Домены и мосты между ними. В овалах – домены. Стрелка – мост. Домен к задаче – внизу, сервисные – внизу. Диаграмма доменного уровня как правило содержит в верхней части домены, наиболее осведомленне о системе (прикладные), внизу – сервисные и реализации.

Строим 3 диаграммы для взаимодействия

1. модель связей подсистем (аналог информационной модели подсистем)

2. модель взаимодействия подсистем (по МВО)
3. модель доступа подсистем (по МДО)

Стрелочкой помечается идентификатор процесса. Модель взаимодействия – асинхронная, событийная модель. Модель доступа – синхронное взаимодействие (один объект может получить данные другого объекта) В итоге получаем модель доступа к объектам, таблицу процессов состояний, диаграмму потомков данных действий.

11. Объектно-ориентированное проектирование. Принцип проектирования. Архитектурный домен. Шаблоны для создания прикладных классов.

Диаграммы для ООП легко получаются из диаграмм ООА.

Диаграмма класса описывает внешнее представление данного класса.

Слева: принимаемые, возвращаемые, обрабатываемые данные
Ромб – обрабатывает исключительную ситуацию.
Пунктир – отсрочиваемая операция, callback выполняемый в результате события

Схема структуры класса конкретизирует внутреннее представление и структуру класса.
Класс реализуется по слоям. Есть публичные методы, есть приватные; доступ к нижним слоям напрямую снаружи недопустим, как и вызов пабликом паблика.

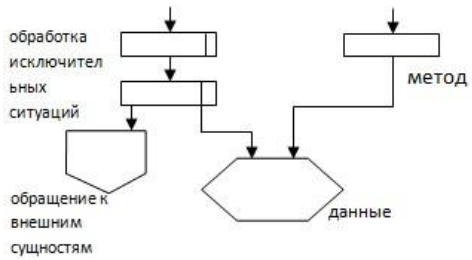


Диаграмма зависимостей – схема использования.
Двойная стрелка – дружественная связь.
Строится диаграмма класса,
только оставляем заголовок, а атрибуты перечисляем списком.

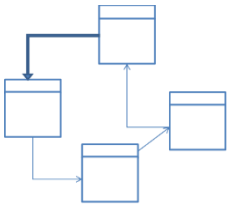
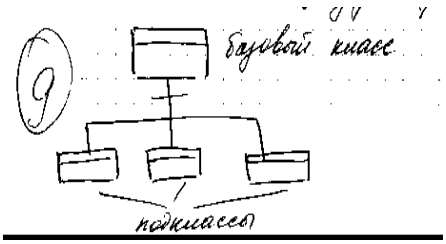
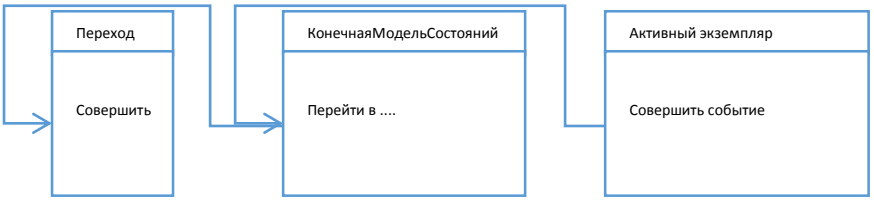


Диаграмма наследования – схема наследования классов. От базового класса подклассы. Внутри атрибуты и методы.



Архитектурный домен

Создаются несколько классов для архитектурного домена, задача которых – задать правила перехода из состояния в состояние при возникновении событий. Эти классы представляют активный экземпляр; все остальные будут производными от активного.



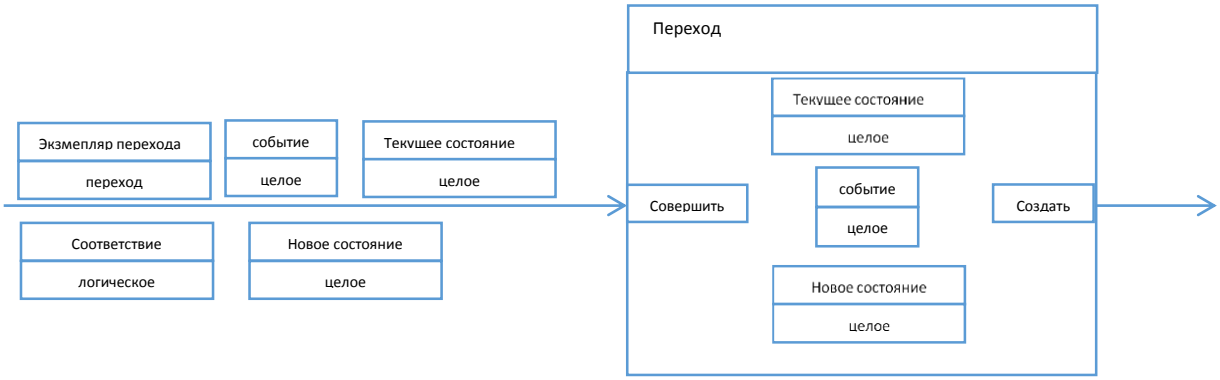
Задаются переходы: КМС будет состоять из возможных переходов.

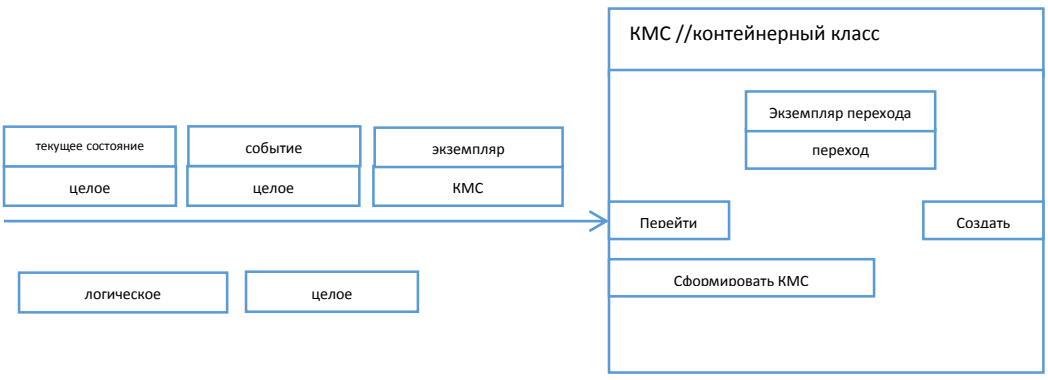
КМС делается шаблонным классом – позволяет сделать систему гибкой.

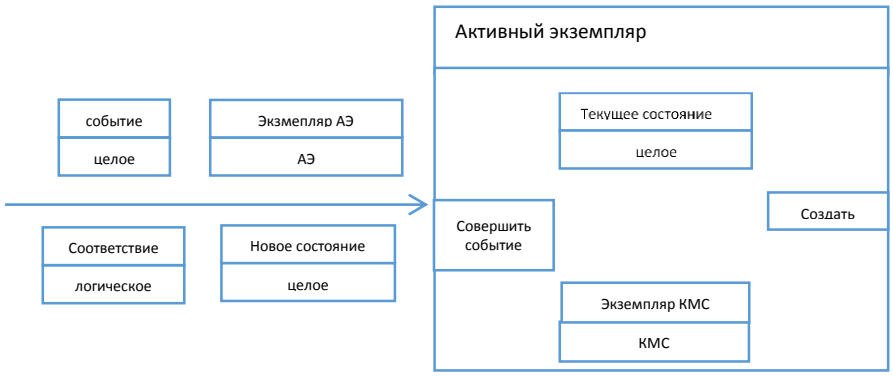
Все активные классы будут производными от «акт экз». Соответственно у АЭ должен быть конструктор, в который мы передаём начальное состояние и КМС.
В схему можно добавить параметр игнорирования перехода: возвращается флаг. Либо проверка происходит на уровне формирования активного класса.
Выделены пассивные и активные объекты. Для активных выделяются жизненные циклы.
Диаграмма пассивного класса должна включать в себя: атрибуты (компоненты экземпляра), аксессоры (сет и гет) (как объекта, так и класса).
Диаграмма активного: атрибуты, аксессоры (как правило), методы объекта (тейкеры событий) – действия, которые соответствуют состояниям в модели состояний объекта. «Затребовать событие» - реакция на событие.
Кроме этого, активный класс включает инициализатор КМС.

Вместо 

рисую 







C++

12. Структура программы на языках C, C++. Отличия.

Набор файлов, включающих директивы препроцессора, объявления.

Main обязательно должна присутсвовать, не может вызывать саму себя.

Особенность: раздельная компиляция.

Чтобы собрать программу: создать заголовочные файлы. Они содержат: const,объявления переменных, типы объявленных функций (определять нельзя).

Определен если макрос, то включение заголовочного файла. Pragma once – не желательно использовать. Группу файлов с единым заголовочным файлом, объединяем в библиотеки.

Ссылка (кличка) – механизм передачи параметров в функцию и возврата из функции.

Перегрузка функций – множество функций с одним и тем же именем (список параметров разный)

13. Классы и объекты, ограничение доступа к членам класса.

Описание класса на основе структуры, объединения, класса. Закрыть доступ к данным при проектировании: защищенные уровни.

Появляется тип данных – класс. Класс в заголовочных классах. Реализация методов в форме реализации.

3 уровня доступа:

- private – нет доступа к этим членам извне.
- protected – доступ к членам для потомков.
- public – доступ извне.

По умолчанию struct – public,a class – private.

14. Создание и уничтожение объектов.

Выделили метод, который называли конструктор. Это метод вызываемый при инициализации объекта. У него отсутствует тип возврата. Конструктор можно перегружать. Конструктор не наследуется. Если конструктор private,то невозможно создание производных классов.

Когда вызывается конструктор:

1. При определении для статических и внутренних объектов. Выполняется до функции main ().
2. При определении локальных объектов. 3. При выполнении оператора new
4. Для временных объектов.

Конструктор должен быть всегда. Если нет конструктора, то всегда создаётся 2 конструктора

1 – по умолчанию, 2 – конструктор копирования.

Если мы указали хотя бы один конструктор, то конструктор по умолчанию не создаётся. Конструктор копирования создаётся всегда.

Конструктор копирования вызывается:

1. При инициализации одного объекта другим. 2. При передаче по значению параметров
3. При возврате по значению.

```
class Complex
{ private:
      double re, im; public:
        Complex ();                                    //1
        Complex (double r);                            //2
        Complex (double r, double i);            //3
        Complex (Complex &c);                    //4
};

Complex a(); a() - это функция без параметров
возвращающая тип Complex.
Complex b;               //1
Complex c (1.);  //2
Complex d = 2.;  //2
Complex e (3. ,4.); //3 f
= Complex (5., 6.); //3 g
(f), h = g;
```

Конструктор не может быть volatile, static, const

В C++ реализуется неявный вызов деструктора. Этот метод не принимает параметров. Нет типа возврата. Деструктор имеет такое же имя что и конструктор, но начинается со знака ~. Деструкторы вызываются в обратном порядке. Для локальных статических объектов вызывается деструктор до уничтожении глобальных статических объектов, но после

выполнении программы. Временные объекты уничтожаются, когда в них отпадает надобность. Деструктор не перегружается.

Деструктор не может быть `const`, `volatile`, `static`, но может быть `virtual`.

15. Наследование. Построение иерархии классов. Множественное наследование и неоднозначности в нём. Понятие доминирования.

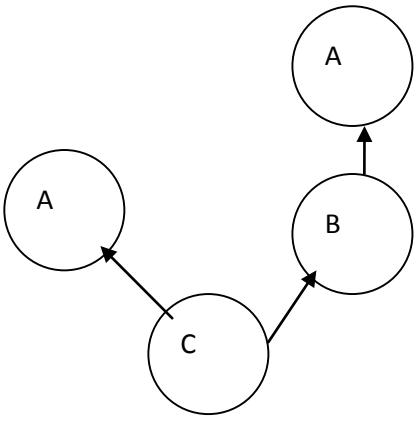
Расширение и выделение общей части из разных классов.

Причины выделения общей части:

1. Общая схема использования.
2. Сходство между наборами операций.
3. Сходство реализации.

Расщепление классов

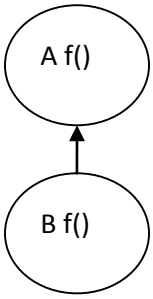
1. Два подмножества операций в классе используются в разной манере.
2. Методы класса имеют не связную реализацию.
3. Класс оперирует очевидным образом в 2-х несвязных обсуждениях проекта.



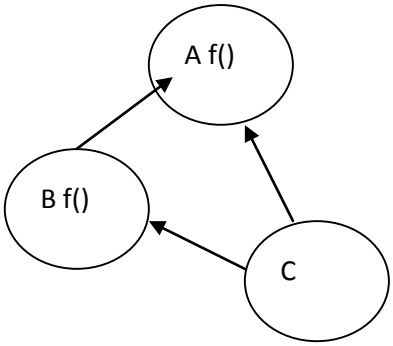
Прямая база – непосредственная база класса. Прямая база может входить только один раз

Косвенная база – A для C. Может быть несколько раз.

Тут A для C и косвенная и прямая база.



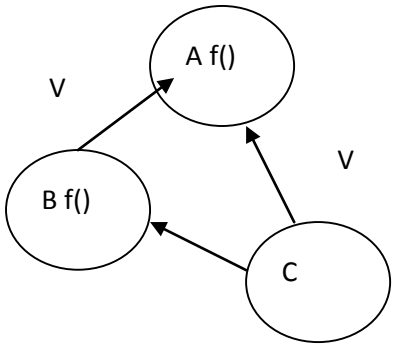
Метод f () в производном классе B доминирует над методом в доминантном классе A.

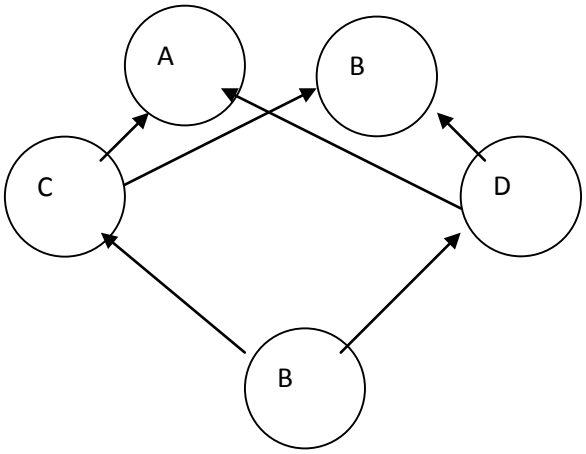


ОШИБКА!

(Неоднозначность)

Виртуальное наследование






```
class A
{
public:


        int a; int
        (*b)(); int
        f(); int
        f(int); int
        g();
};
  class B
{
private:
     int a;
     int b;
public:
     int f();
     int g;
     int h();
        int h(int);
};
class C:public A, public B{};
void f (C *pc)
{
        pc->a = 1;              //Error! Проверка на неоднозначность происходит
        pc->b ();               //Error! до проверки на степень доступа.
        pc->f ();               //Error!
        pc->f (1);              //Error!
        pc->g = 1;              //Error!
}
```

16. Полиморфизм, понятие абстрактного класса. Дружественные связи.

Дружба

Доступ ко всем членам класса методов других классов. В классе для объектов указываем что есть «друг».

```
class CA
{
        friend class CB;
}; class
CB{};
```

Схема зависимая, то есть при изменении CA приходиться менять CB. Но можно сделать по другому: свести «дружественное» отношение к методу.

```
class CA
{
        friend class CB;
        friend int CB::f(CA&);
}; class
CB {
public:
        int f(CA&);
};
```

Необходимо, чтобы дружественных отношений было как можно меньше. Дружба не наследуется (сын друга не друг), не транзитивна (друг моего друга не друг).

Перегрузка операторов.

Тип данных = множество значений + множество операций.

Перегружать нельзя: ”::”, ”.”, ”.*”, тернарный оператор (: ?), sizeof().

.* - выбор члена класса через указатель на классовые члены

Не меняется *арность*, приоритет и порядок выполнения (при перегрузке).

17. Шаблоны классов. Обработка исключительных ситуаций.

```
template <class T, int size>
class Vector {
      T  V[size];  //нельзя использовать массивы объектов, а можно массивы указателей
      int Q;       //количество элементов public:
        Vector(int &Z);
}
template <class, int size> Vector
<T,size>::Vector(int &Z)
{
        Q = &Z;
}

создание объектов по шаблону
Vector <double,10> V1(2);//создается класс с параметрами <double,10> Vector
<double,11> V2(3);
```

Идея: передавать управление как обработчик некой ошибки. Проблема в том, что нужно возвращаться в точку возникновения ситуации.

Каждый класс будет отвечать за свою исключительную ситуацию.

Классы, которые отвечают за обработку исключительных ситуаций, должны быть родственными. И тогда если мы создаём новый класс, то передаём указатель в класс-обработчик.

```
try {
        throw <выражение>; создание объекта класса
}
catch (<параметр>)
{
        …
}
```

CLR: делегаты, события, свойства (property).

*как я понял Тассова, Qt-ные QEvent и Q_PROPERTY тоже прокатят, по поводу эквивалентности CLR-ных делегатов и Qtных slot, signal я не уверен+

Упрощённо делегаты можно рассматривать как узаконенные указатели на функции. Но всё же они, как и всё в CLR, являются объектами и обладают своей дополнительной функциональностью. Объявление делегатов производится с помощью ключевого слова __delegate. С помощью делегатов могут быть вызваны любые методы управляемых классов, как обычные, так и статические. Это принципиально отличает делегаты от указателей на функции, так как делегат хранит не только указатель на функцию, но и информацию о конкретном объекте, у которого эта функция должна быть вызвана. Единственное условие – прототип метода должен совпадать с типом делегата.

```
__delegate void DelegateSampl(int);
__gc class Foo { public:
    void TestDelegate1(int n) {
System::Console::WriteLine(n+1);
    }
    static void TestDelegate2(int n) {
System::Console::WriteLine(n+2);
    }
};
void test() {
    Foo *f = new Foo();
    DelegateSampl *d1 = new DelegateSampl(f,Foo::TestDelegate1);
d1(1); // вызов TestDelegate1
    d1 += new DelegateSampl(0,Foo::TestDelegate2);
    d1(2); // одновременный вызов TestDelegate1 и TestDelegate2 }
```

События объявляются с помощью ключевого слова __event.

```
__delegate void ClickEvent(int,int);
__gc class EventSource { public:
    __event ClickEvent *OnClick;
    void FireEvent() {
      OnClick(1,2);
    }
};
```

```
__gc class Foo { public:
    __property int  get_X() { return 0; }
    __property void set_X(int) {}
};

void test() {
    Foo *f = new Foo();      f->X
= 1;      // вызов set_X      int
i = f->X; // вызов get_X
}
```

Паттерны проектирования

Шаблоны: 1. Одиночка 2. Хранитель Структурные: 1. Адаптер 2. Композит 3. Декорат (заместитель) 4. Прокси 5. Мост 6. Фасад 7. Приспособленец	Порождающие: 1. Фабричный метод 2. Строитель Поведения: 1. Команда 2. Стратегия 3. Цепочка обязанностей 4. Интерпретатор 5. Посредник 6. Подписчик издатель

Шаблоны

Одиночка

Одиночка - паттерн, порождающий объекты. Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Используйте паттерн одиночка, когда должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам. В классе содержится статическая переменная-член instance, которая содержит указатель на уникальный экземпляр.

```
template <typename T> class
Singleton
{
private:
    static T *ptr;
protected:
Singleton();
public:
static T& instance()
    {
        return ptr?*ptr:*(ptr = new T);
    }
private:
    //Singleton(Singleton<T> const&);
    //Singleton<T>& operator=(Singleton<T> const&) { return *this; };
};
template<T> T* Singleton<T>::ptr=0;

Canvas &obj = Singletone::instance<Canvas>();
```

Хранитель

Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект. Иногда необходимо тем или иным способом зафиксировать внутреннее состояние объекта. Его необходимо где-то сохранить, чтобы позднее восстановить в нем объект.

```
template <typename T> class Holder;
template <typename T> class Trule
//TRansfer capsULE
//используется только для передачи;
чтобы при передачи в метод функции
или возврата функции из метода не возникло утечки
{
private:
	T* ptr;
public:
	Trule(Holder<T>&h) { ptr = h.release(); }
	~Trule() {delete ptr; }
private:
	Trule(Trule<T>&);
	Trule<T>& operator = (Trule<T>&);
	friend class Holder<T>; //для простоты обращения
};

template <typename T> class Holder
{
private:
	T* ptr;
public:
//когда создаётся холдер указателя, он не должен принимать указатель откуда-то извне - это может привести
к тому, что холдер может держать указатель на уже освобождённую память
	Holder(): ptr(0) {}
	explicit Holder(T* p): ptr(p) {} //нужен явный вызов конструктора, явное создание объекта
	~Holder() { delete ptr; }
	//холдер должен быть прозрачен: через него мы может обратиться ко всему, что он держит
	T* operator ->() const{return ptr;}
	//необходима работа со ссылкой на объект
	T& opetator *() const{return *ptr;}
	//обмен
	void exchange(Holder<T>& h)
	{
		swap(ptr, h.ptr); }
	}
	//холдер берёт на себя указатель капсулы
	Holder(Trule<T> const& t)
	{
		ptr = t.ptr;
		//т.птр - константа, поэтому приходится извращаться
		const_cast<Trule<T>&>(t).ptr = 0;
	}
	//присваивание
	Holder<T>& operator = (Trule<T> const& t)
	{
		delete ptr;
		ptr = t.ptr;
		const_cast<Trule<T>&>(t).ptr = 0;
		return *this;
	}
	T* release()
	{
		T* p = ptr;
		ptr = 0;
		return p;
	}
	//также нужны ещё конструктор копирования и оператор присваивания(холдер холдер)
	//при передаче объектов в функцию мы будем использовать труле
};
```

Структурные

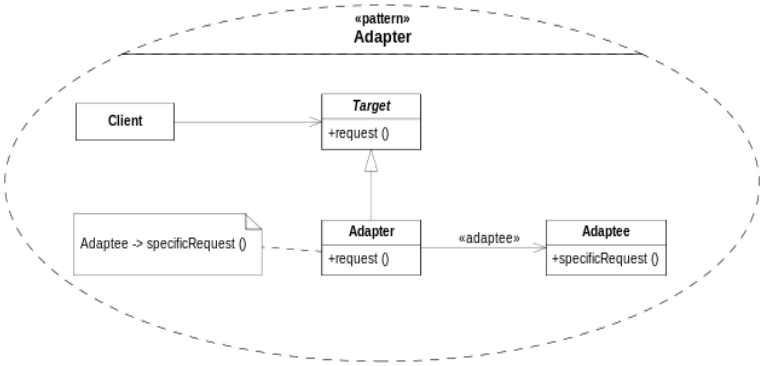
Адаптер

Назначение: для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс (приводит интерфейс класса (или нескольких классов) к интерфейсу требуемого вида)

Применяется: система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс. Чаще всего шаблон Адаптер применяется если необходимо создать класс, производный от вновь определяемого или уже существующего абстрактного класса.

Пусть класс, интерфейс которого нужно адаптировать к нужному виду, имеет имя Adaptee. Для решения задачи преобразования его интерфейса паттерн Adapter вводит следующую иерархию классов:

1. Виртуальный базовый класс Target. Здесь объявляется пользовательский интерфейс подходящего вида. Только этот интерфейс доступен для пользователя.
2. Производный класс Adapter, реализующий интерфейс Target. В этом классе также имеется указатель или ссылка на экземпляр Adaptee. Паттерн Adapter использует этот указатель для перенаправления клиентских вызовов в Adaptee. Так как интерфейсы Adaptee и Target несовместимы между собой, то эти вызовы обычно требуют преобразования.




```
class FahrenheitSensor {
public:
    float getFahrenheitTemp() {float t = 32.0;return t;}
};
class Sensor {
public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};
class Adapter : public Sensor {
public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {}
   ~Adapter() {delete p_fsensor;}
float getTemperature() {
      return (p_fsensor->getFahrenheitTemp()-32.0)*5.0/9.0;
    }
private:
    FahrenheitSensor* p_fsensor;
};
```

Достоинства:

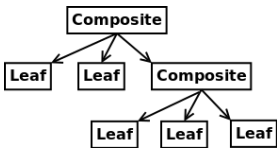
Паттерн Adapter позволяет повторно использовать уже имеющийся код, адаптируя его несовместимый интерфейс к виду, пригодному для использования.

Недостатки:

Задача преобразования интерфейсов может оказаться непростой в случае, если клиентские вызовы и (или) передаваемые параметры не имеют функционального соответствия в адаптируемом объекте.

Компоновщик (composite)

Composite (Компоновщик) относиться к классу структурных паттернов. Он используется для компоновки объектов в древовидные структуры для представления иерархий, позволяя одинаково трактовать индивидуальные и составные объекты.



Как можно догадаться, composite это составной объект, а leaf это конечный потомок дерева (лист), у которого не может быть потомков. Примером такой структуры может быть XML или DOM дерево.

Основным назначением паттерна, является обеспечение единого интерфейса как к составному, так и конечному объекту, что бы клиент не задумывался над тем, с каким объектом он работает. Общеизвестными примерами этого паттерна является SimpleXML и jQuery.

```
class Unit { public:
    virtual int getStrength() = 0;
virtual void addUnit(Unit* p) {}
virtual ~Unit() {}
};
class Archer: public Unit { public:
    virtual int getStrength() {return 1;} };
```

```
class Infantryman: public Unit { public:
    virtual int getStrength(){return 2;}
};
class CompositeUnit: public Unit { public:
    int getStrength() {
int total = 0;
        for(int i=0; i<c.size(); ++i)
total += c[i]->getStrength();
return total;
    }
    void addUnit(Unit* p){c.push_back(p);}
    ~CompositeUnit() {
        for(int i=0; i<c.size(); ++i)
delete c[i];
    } private:
    std::vector<Unit*> c;
};
CompositeUnit* createLegion() {
    CompositeUnit* legion = new CompositeUnit;
for (int i=0; i<3000; ++i)
        legion->addUnit(new Infantryman);
for (int i=0; i<1200; ++i)          legion-
>addUnit(new Archer);      return legion;
}
```

Достоинства

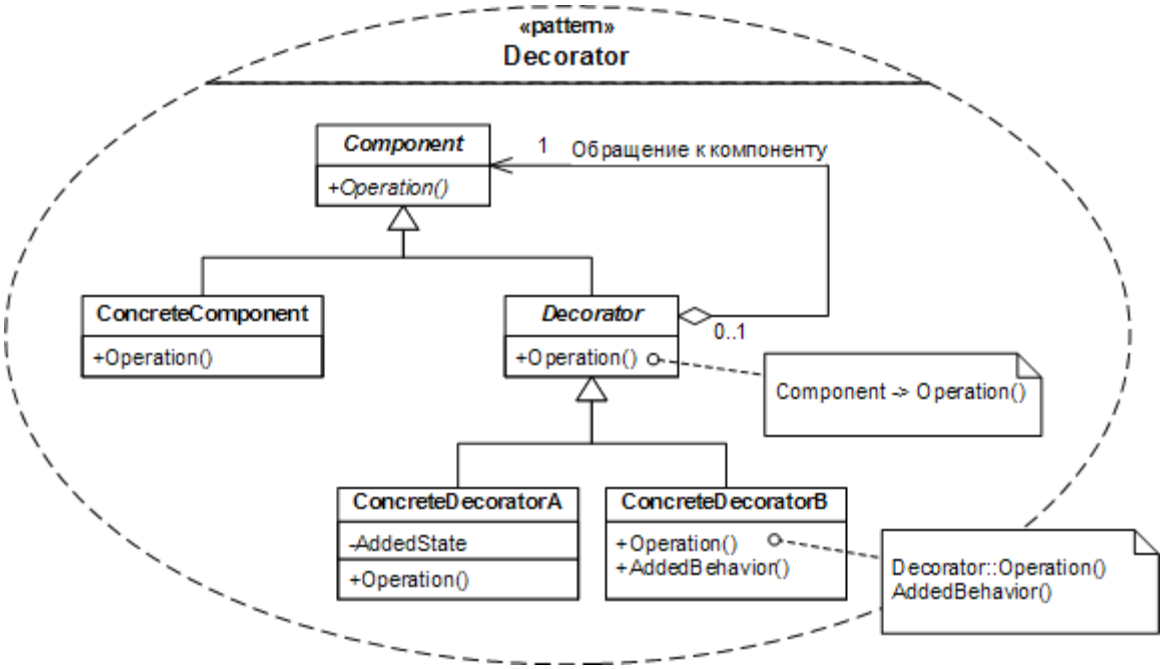
1. В систему легко добавлять новые примитивные или составные объекты, так как паттерн Composite использует общий базовый класс Component.
2. Код клиента имеет простую структуру – примитивные и составные объекты обрабатываются одинаковым образом.
3. Паттерн Composite позволяет легко обойти все узлы древовидной структуры

Декоратор (Decorate)

Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Иногда бывает нужно возложить дополнительные обязанности на отдельный объект, а не на класс в целом. Так, библиотека для построения графических интерфейсов пользователя должна «уметь» добавлять новое свойство, скажем, рамку или новое поведение (например, возможность прокрутки к любому элементу интерфейса). Добавить новые обязанности допустимо с помощью наследования. При наследовании классу с рамкой вокруг каждого экземпляра подкласса будет рисоваться рамка. Однако это решение статическое, а значит, недостаточно гибкое.

Используется для динамического, прозрачного для клиентов добавления обязанностей объектам;



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. A и B – уточнения Декоратора блоков ConcreteComponent, если они имеются.

Мост (Bridge)

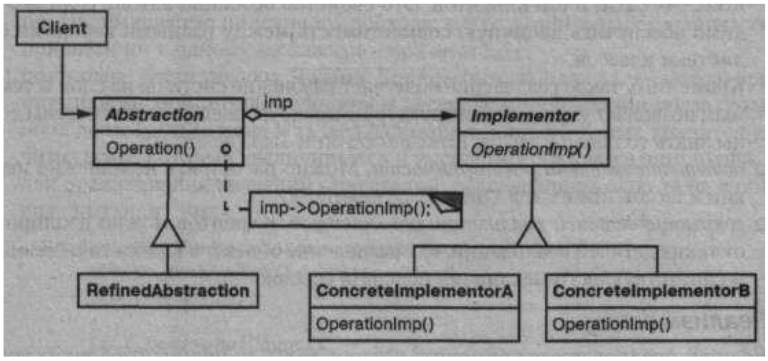
Назначение: Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

Если для некоторой абстракции возможно несколько реализаций, то обычно применяют наследование. Абстрактный класс определяет интерфейс абстракции, а его конкретные подклассы по-разному реализуют его. Но такой подход не всегда обладает достаточной гибкостью. Наследование жестко привязывает реализацию к абстракции, что затрудняет независимую модификацию, расширение и повторное использование абстракции и ее реализации.

Применяйте паттерн мост, когда:

- хотите избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы;
- и абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо;
- изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;

Участники



1. Abstraction (Window) - абстракция:
   - определяет интерфейс абстракции;
   - хранит ссылку на объект типа Implement or;
2. RefinedAbstraction (iconWindow) - уточненная абстракция:
   - расширяет интерфейс, определенный абстракцией Abstraction;
3. Implementor (Windowlmp) - реализатор:
   - определяет интерфейс для классов реализации. Он не обязан точно соответствовать интерфейсу класса Abstraction. На самом деле оба интерфейса могут быть совершенно различны. Обычно интерфейс класса Implementor предоставляет только примитивные операции, а класс Abstraction определяет операции более высокого уровня, базирующиеся на этих примитивах;
4. Concretelmplementor (XWindowlmp, PMWindowlmp) - конкретный реализатор:
   - содержит конкретную реализацию интерфейса класса Implementor.

Отношения

Объект Abstraction перенаправляет своему объекту Implementor запросы клиента.

```
class DrawingAPI {
public:
    virtual void drawCircle(double x, double y, double radius) = 0;      virtual
~DrawingAPI() {}
};
class DrawingAPI1: public DrawingAPI { public:
    DrawingAPI1() {}
virtual ~DrawingAPI1() {}
    void drawCircle(double x, double y, double radius) {
printf("nAPI1 at %f:%f %fn", x, y, radius);
    }
};
class Shape {
public:
    virtual void draw()= 0;
    virtual void resizeByPercentage(double pct) = 0;
virtual ~Shape() {
    }
};
class CircleShape: public Shape { public:
    CircleShape(double x, double y, double radius, DrawingAPI& drawingAPI) :
        x(x), y(y), radius(radius), drawingAPI(drawingAPI) {}     virtual
~CircleShape() {}
    void draw() { drawingAPI.drawCircle(x, y, radius); }
void resizeByPercentage(double pct) { radius *= pct; } private:
    double x, y, radius;
DrawingAPI& drawingAPI;
};
DrawingAPI1 api1;
CircleShape c1(1, 2, 3, api1);
Shape* shapes[1];
shapes[0] = &c1;
shapes[0]->resizeByPercentage(2.5); shapes[0]->draw();
```

Заместитель (Proxy)

Разумно управлять доступом к объекту, поскольку тогда можно отложить расходы на создание и инициализацию до момента, когда объект действительно понадобится. Рассмотрим редактор документов, который допускает встраивание в документ графических объектов. Затраты на создание некоторых таких объектов, например больших растровых изображений, могут быть весьма значительны. Но документ должен открываться быстро, поэтому следует избегать создания всех «тяжелых» объектов на стадии открытия (да и вообще это излишне, поскольку не все они будут видны одновременно).

В связи с такими ограничениями кажется разумным создавать «тяжелые» объекты по требованию. Это означает «когда изображение становится видимым». Но что поместить в документ вместо изображения? И как, не усложняя реализации редактора, скрыть то, что изображение создается по требованию? Например, оптимизация не должна отражаться на коде, отвечающем за рисование и форматирование.

Решение состоит в том, чтобы использовать другой объект – заместитель изображения, который временно подставляется вместо реального изображения. Заместитель ведет себя точно так же, как само изображение, и выполняет при необходимости его инстанцирование.

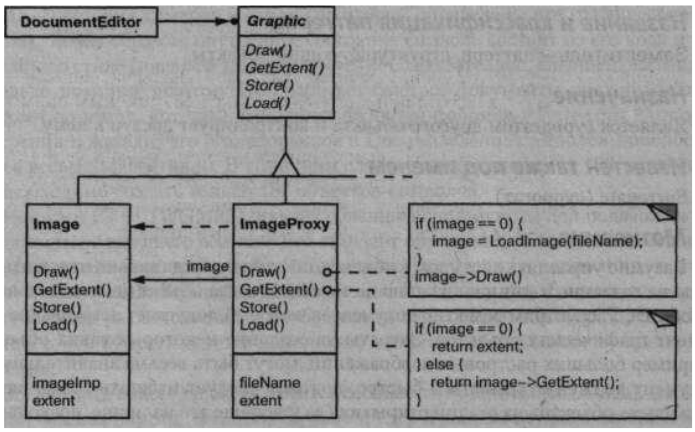
Предположим, что изображения хранятся в отдельных файлах. В таком случае мы можем использовать имя файла как ссылку на реальный объект. Заместитель хранит также размер изображения, то есть длину и ширину. «Зная» ее, заместитель может отвечать на запросы форматера о своем размере, не инстанцируя изображение.

Применение

С помощью паттерна заместитель при доступе к объекту вводится дополнительный уровень косвенности. У того подхода есть много вариантов в зависимости от вида заместителя:

- а удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве;
- а виртуальный заместитель может выполнять оптимизацию, например создание объекта по требованию;
- а защищающий заместитель и «умная» ссылка позволяют решать дополнительные задачи при доступе к объекту.

Редактор документов получает доступ к встроенным изображениям только через интерфейс, определенный в абстрактном классе Graphic. ImageProxy – это класс для представления изображений, создаваемых по требованию. В ImageProxy хранится имя файла, играющее роль ссылки на изображение, которое находится на диске. Имя файла передается конструктору класса ImageProxy.



Участники

1. Proxy (imageProxy) - заместитель:

- хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту. Объект класса Proxy может обращаться к объекту класса Subject, если интерфейсы классов RealSubj ect и Subj ect одинаковы;

- предоставляет интерфейс, идентичный интерфейсу Subject, так что заместитель всегда может быть подставлен вместо реального субъекта;

- контролирует доступ к реальному субъекту и может отвечать за его создание и удаление;

- прочие обязанности зависят от вида заместителя:

- удаленный заместитель отвечает за кодирование запроса и его аргументов и отправление закодированного запроса реальному субъекту в другом адресном пространстве;

- виртуальный заместитель может кэшировать дополнительную информацию о реальном субъекте, чтобы отложить его создание. Например, класс ImageProxy из раздела «Мотивация» кэширует размеры реального изображения;

- защищающий заместитель проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права;

2. Subject (Graphic) - субъект:

- определяет общий для RealSubject и Proxy интерфейс, так что класс Proxy можно использовать везде, где ожидается RealSubject;

3. RealSubject (Image) - реальный субъект:

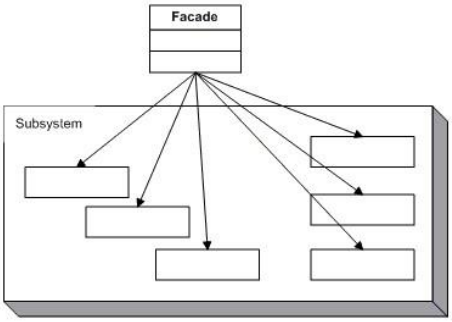
- определяет реальный объект, представленный заместителем.

Отношения

Proxy при необходимости переадресует запросы объекту RealSubject. Де тали зависят от вида заместителя.

Фасад (Facade)

Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.



Разбиение на подсистемы облегчает проектирование сложной системы в целом. Общая цель всякого проектирования - свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними. Один из способов решения этой задачи - введение объекта фасад, предоставляющий единый упрощенный интерфейс к более сложным системным средствам.

Фасад используют, когда:

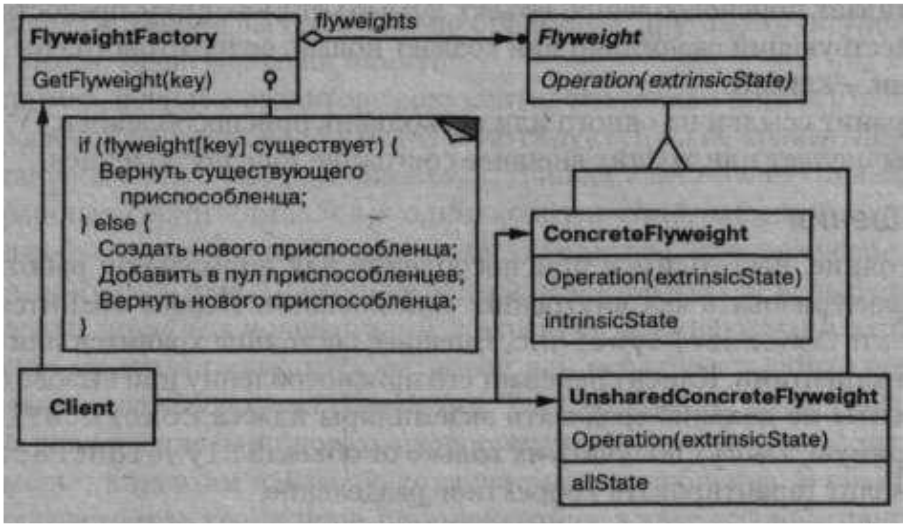
1. Нужно предоставить простой интерфейс к сложной подсистеме
2. Между клиентами и классами реализации абстракции существует много зависимостей
3. Нужно разложить подсистему на отдельные слои.

Разбиение системы на компоненты позволяет снизить ее сложность. Ослабить связи между компонентами системы можно с помощью паттерна Facade. Объект "фасад" предоставляет единый упрощенный интерфейс к компонентам системы.

Приспособленец (Flyweight)

Использует разделение для эффективной поддержки множества мелких объектов.

Приспособленец - это разделяемый объект, который можно использовать одновременно в нескольких контекстах. В каждом контексте он выглядит как независимый объект, то есть неотличим от экземпляра, который не разделяется.



Flyweight используется для уменьшения затрат при работе с большим количеством мелких объектов. При проектировании приспособленца необходимо разделить его свойства на внешние и внутренние. Внутренние свойства всегда неизменны, тогда как внешние могут отличаться в зависимости от места и контекста применения и должны быть вынесены за пределы приспособленца. Внутреннее состояние хранится в самом приспособленце и состоит из информации, не зависящей от его контекста. Именно поэтому он может разделяться. Внешнее состояние зависит от контекста и изменяется вместе с ним, поэтому не подлежит разделению. Объекты-клиенты отвечают за передачу внешнего состояния приспособленцу, когда в этом возникает необходимость.

Пример использования: редактор текста может создавать по одному приспособленцу на каждую букву алфавита. Каждый приспособленец хранит код символа, но координаты положения символа в документе и стиль его начертания определяются алгоритмами размещения текста и командами форматирования, действующими в том месте, где символ появляется. Код символа - это внутреннее состояние, а все остальное - внешнее.

1. Flyweight - приспособленец: - объявляет интерфейс, с помощью которого приспособленцы могут получать внешнее состояние или как-то воздействовать на него;
2. ConcreteFlyweight - конкретный приспособленец:
   - реализует интерфейс класса Flyweight и добавляет при необходимости внутреннее состояние. Объект класса ConcreteFlyweight должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть не зависящим от контекста;
3. UnsharedConcreteFlyweight - неразделяемый конкретный приспособленец:
   - не все подклассы Flyweight обязательно должны быть разделяемыми. Интерфейс Flyweight допускает разделение, но не навязывает его. Часто у объектов UnsharedConcreteFlyweight на некотором уровне структуры приспособленца есть потомки в виде объектов класса ConcreteFlyweight.
4. FlyweightFactory - фабрика приспособленцев:
   - создает объекты-приспособленцы и управляет ими;
   - обеспечивает должное разделение приспособленцев.
   Когда клиент запрашивает приспособленца, объект FlyweightFactory предоставляет существующий экземпляр или создает новый, если готового еще нет;
5. Client - клиент:
   - хранит ссылки на одного или нескольких приспособленцев;
   - вычисляет или хранит внешнее состояние приспособленцев.

```
//работа с символами
#include <map>
//базовый класс:
class Character
{
protected:
	char Symbol;
	int Size;
public:
	virtual void view() = 0;
};

class CharacterFactory
{
private:
	//map – каждому символу char  в соответсвие ставится Character
	std::map<char,Character*> Characters;
	int size;
public:
	//приспособленец или возвращает новый объект, или указатель на существующий
	Character& GetCharacter(char key)
	{
		Characters::iterator it = Characters.find(key);
		if (Characters.end() == it)
		{
			Character* ch = new ConCharacter(key,size);
			Characters[key] = ch;
			return *ch;
		}
		else
		{
			return *it;   (здесь было написано *it->second, но это похоже не бред)
		 }
	}
};
```

Порождающие

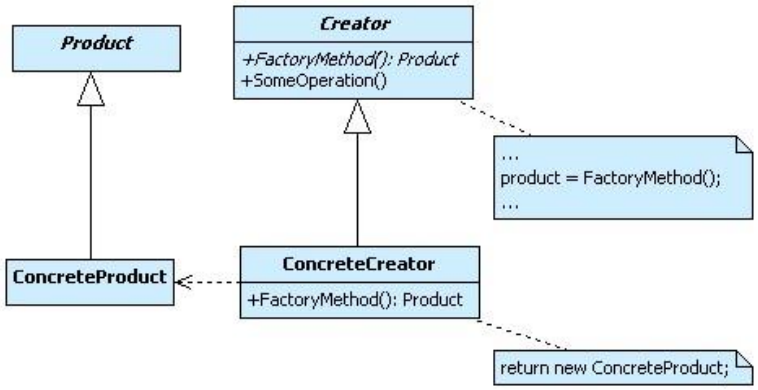
Фабричный метод (Factory Method)

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанциировать. Создает объекты разных типов, позволяя системе оставаться независимой как от самого процесса создания, так и от типов создаваемых объектов.

Фабричный метод позволяет классу делегировать создание подклассов. Используется, когда:

- классу заранее неизвестно, объекты каких подклассов ему нужно создавать.
- класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами.
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

1. Product — продукт определяет интерфейс объектов, создаваемых абстрактным методом
2. ConcreteProduct — конкретный продукт реализует интерфейс Product;
3. Creator — создатель объявляет фабричный метод, который возвращает объект типа Product.
4. ConcreteCreator — конкретный создатель переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса ConcreteProduct.



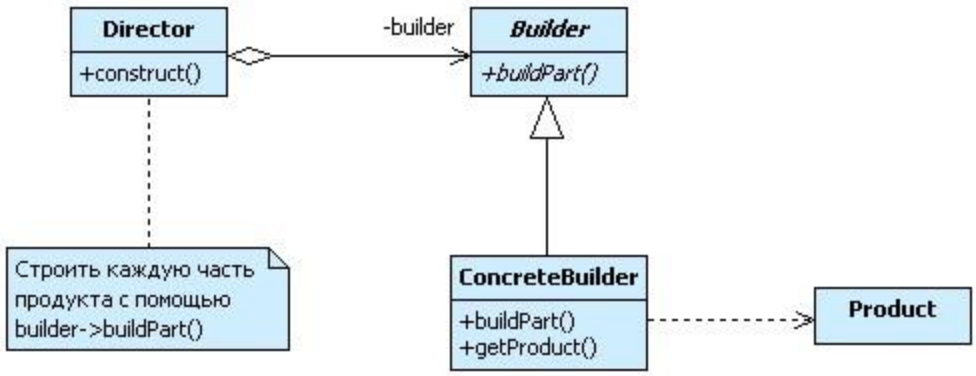

```
class Product;
class Creator
{
public:
    Creator():ptr(0) {}
    ~Creator() { delete ptr; }
    Product* getProduct(){ return ptr ? ptr : (ptr = createProduct()); }
protected:
    virtual Product* createProduct() = 0;

private:
    Product* ptr;
};

template <typename T> class ConCreator: public Creator
{
protected:
    Product* createProduct()
    {
        return new T;
    }
};
```

Строитель(Builder)

Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления. Строитель включает в себя набор методов поэтапного создания объекта и его частей + метод GetProduct, возвращающий окончательно созданный объект.

Используйте, когда:

1. В системе могут существовать сложные объекты, создание которых за одну операцию затруднительно или невозможно. Требуется поэтапное построение объектов с контролем результатов выполнения каждого этапа.
2. Данные должны иметь несколько представлений. Приведем классический пример. Пусть есть некоторый исходный документ в формате RTF (Rich Text Format), в общем случае содержащий текст, графические изображения и служебную информацию о форматировании (размер и тип шрифтов, отступы и др.). Если этот документ в формате RTF преобразовать в другие форматы (например, Microsoft Word или простой ASCII-текст), то полученные документы и будут представлениями исходных данных.



1. Builder (TextConverter) - строитель: задает абстрактный интерфейс для создания частей объекта Product;
2. ConcreteBuilder - конкретный строитель:
   - конструирует и собирает вместе части продукта посредством реализации интерфейса Builder;
   - определяет создаваемое представление и следит за ним;
   - предоставляет интерфейс для доступа к продукту;
3. Director (RTFReader) - распорядитель: - конструирует объект, пользуясь интерфейсом Builder;
4. Product (ASCIIText, TeXText, TextWidget) - продукт:
   - представляет сложный конструируемый объект;
   - включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

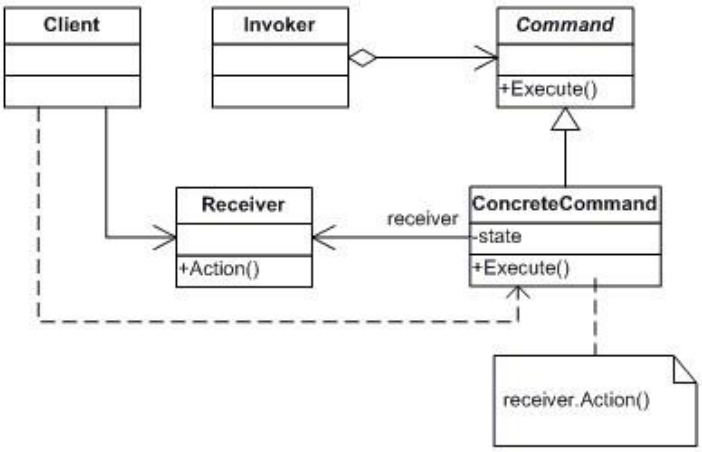
Поведение

Команда (Command)

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

Используйте, когда:

1. Система управляется событиями. При появлении такого события (запроса) необходимо выполнить определенную последовательность действий.
2. Необходимо параметризировать объекты выполняемым действием, ставить запросы в очередь или поддерживать операции отмены (undo) и повтора (redo) действий.
3. Нужен объектно-ориентированный аналог функции обратного вызова в процедурном программировании.



Пример событийно-управляемой системы – приложение с пользовательским интерфейсом. При выборе некоторого пункта меню пользователем вырабатывается запрос на выполнение определенного действия (например, открытия файла).

В паттерне Command может быть до трех участников:

1. Клиент, создающий экземпляр командного объекта.
2. Инициатор запроса, использующий командный объект.
3. Получатель запроса.

```
class Command //базовый класс команды
{
public:
	virtual bool execute() = 0;
};

template<typename Reciever> class SimpleCommand: public Command
{
private:
	typedef bool (Reciever::*Action)();
	Reciever* rec; //указатель на объект
	Action act; //указатель на метод
public:
	SimpleCommand(Reciever* r, Action a); //конструктор
	bool execute() {return (rec->*act)(); }
};
```

Стратегия (Strategy)

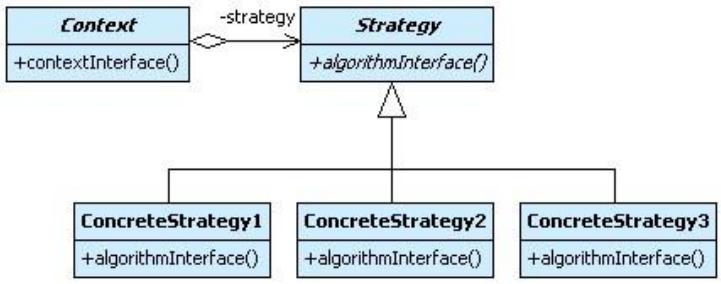
Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Используйте паттерн стратегия, когда:

1. имеется много родственных классов, отличающихся только поведением. Стратегия позволяет сконфигурировать класс, задав одно из возможных поведений;
2. вам нужно иметь несколько разных вариантов алгоритма.
3. в алгоритме содержатся данные, о которых клиент не должен «знать».
4. в классе определено много поведений, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Участники:

1. Strategy - стратегия: - объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy;
2. ConcreteStrategy (SimpleCompositor, TeXCompositor,
3. ArrayCompositor - конкретная стратегия: - реализует алгоритм, использующий интерфейс, объявленный в классе Strategy;
4. Context - контекст:
   - конфигурируется объектом класса ConcreteStrategy;
   - хранит ссылку на объект класса Strategy;
   - может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.



Пример: приложение, предназначенное для компрессии файлов использует один из доступных алгоритмов: zip, arj или rar.

```
class Compression{
public:
    virtual ~Compression() {}
    virtual void compress( const string & file ) = 0;
};
class ZIP_Compression : public Compression{
public:
    void compress( const string & file );
};
class RAR_Compression : public Compression{
public:
    void compress( const string & file );
};
class Compressor {
public:
    Compressor( Compression* comp): p(comp) {}
   ~Compressor() { delete p; }
    void compress( const string & file ) { p->compress( file); }
private:
    Compression* p;
};
Compressor* p = new Compressor( new ZIP_Compression);
```

Цепочка обязанностей (Chain of Responsibility)

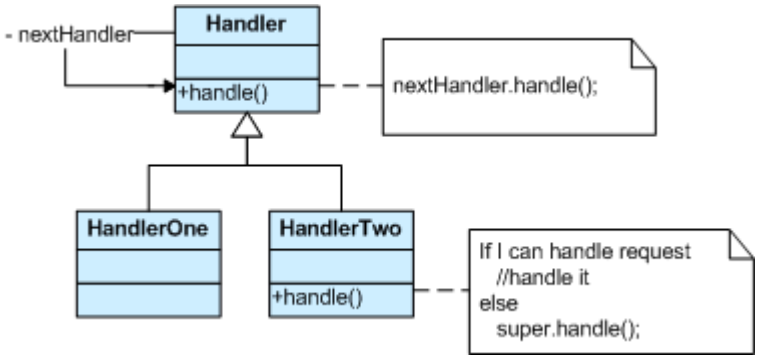
Инкапсулирует элементы по обработке запросов внутри абстрактного "конвейера". Клиенты "кидают" свои запросы на вход этого конвейера. Производные классы знают, как обрабатывать запросы клиентов. Если "текущий" объект не может обработать запрос, то он делегирует его базовому классу, который делегирует "следующему" объекту и так далее.

Используйте, когда:

1. в разрабатываемой системе имеется группа объектов, которые могут обрабатывать сообщения определенного типа;
2. все сообщения должны быть обработаны хотя бы одним объектом системы;
3. сообщения в системе обрабатываются по схеме «обработай сам, либо перешли другому», то есть одни сообщения обрабатываются на том уровне, где они получены, а другие пересылаются объектам иного уровня.

Участники

1. Handler (HelpHandler) - обработчик:
   - определяет интерфейс для обработки запросов;
   - (необязательно) реализует связь с преемником;
2. ConcreteHandler (PrintButton, PrintDialog) - конкретный обработчик:
   - обрабатывает запрос, за который отвечает;
   - имеет доступ к своему преемнику;
   - если ConcreteHandler способен обработать запрос, то так и делает, если не может, то направляет его своему преемнику;
3. Client - клиент:
   - отправляет запрос некоторому объекту ConcreteHandler в цепочке.



Интерпретатор (Interpreter)

Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.

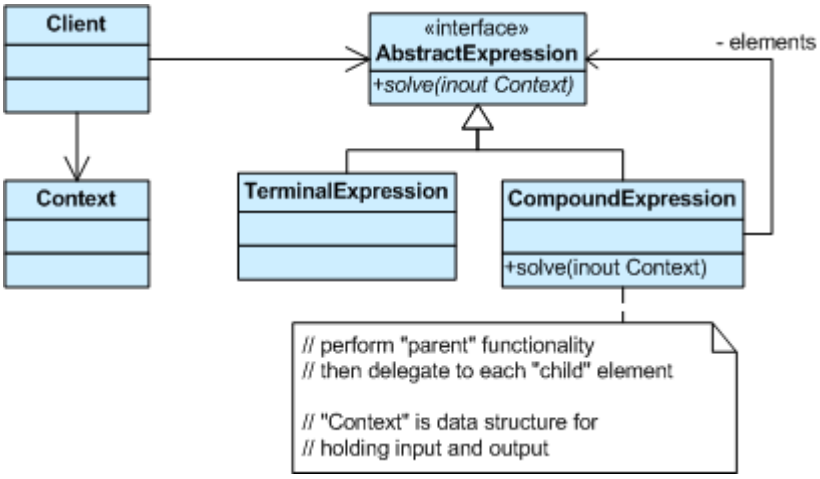
Пусть в некоторой, хорошо определенной области периодически случается некоторая проблема. Если эта область может быть описана некоторым “языком“, то можно будет создать интерпретатор, который решает задачу, анализируя предложения этого языка.

Пример: поиск строк по маске.

1. Определите “малый“ язык.
2. Разработайте грамматику для языка.
3. Для каждого грамматического правила (продукции) создайте свой класс.
4. Полученный набор классов организуйте в структуру с помощью паттерна Composite.
5. В полученной иерархии классов определите метод interpret(Context).

Объект Context инкапсулирует информацию, глобальную по отношению к интерпретатору. Используется классами во время процесса ”интерпретации”.

Лучше всего паттерн работает, когда грамматика языка проста и эффективность не является главным критерием.

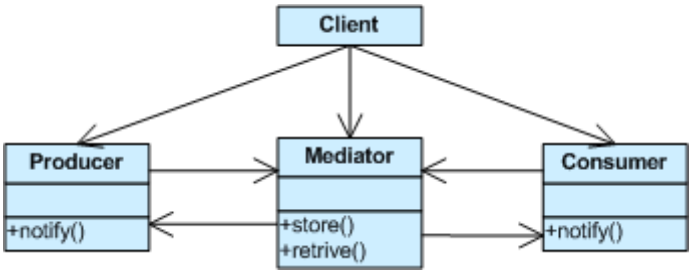


Посредник (Mediator)

Паттерн Mediator вводит посредника для развязывания множества взаимодействующих объектов, заменяет взаимодействие "многие ко многим" на"один ко многим".

Используйте паттерн посредник, когда:

1. имеются объекты, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания;
2. нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами;
3. поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

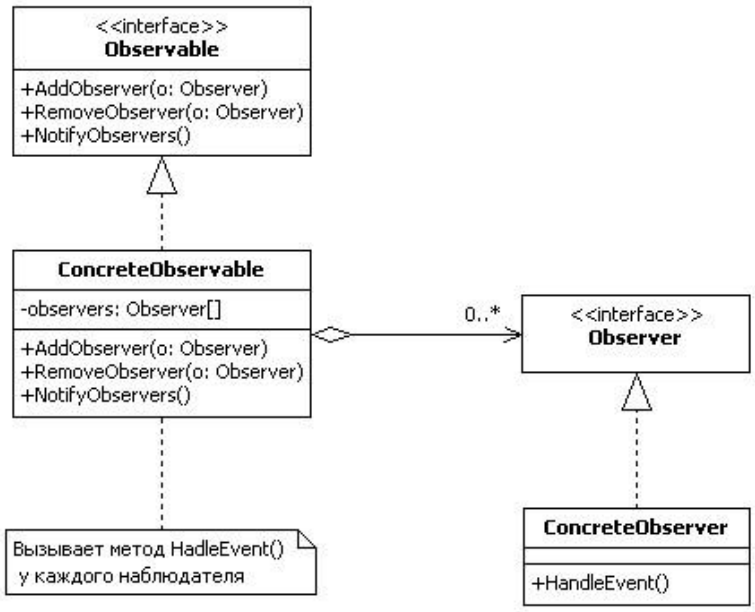


Участники

1. Mediator (DialogDirector) - посредник;
   - определяет интерфейс для обмена информацией с объектами Colleague;
2. ConcreteMediator (FontDialogDirector) - конкретный посредник:
   — реализует кооперативное поведение, координируя действия объектов Colleague;
   - владеет информацией о коллегах и подсчитывает их;
3. Классы Colleague (ListBox, EntryField) - коллеги:
   — каждый класс Colleague «знает» о своем объекте Mediator;
   - все коллеги обмениваются информацией только с посредником, так как при его отсутствии им пришлось бы общаться между собой напрямую.

Подписчик-издатель (Publisher and Subscriber)

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



Observable — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей.
Observer — интерфейс, с помощью которого наблюдатель получает оповещение.
ConcreteObservable — конкретный класс, который реализует интерфейс Observable.
ConcreteObserver — конкретный класс, который реализует интерфейс Observer.

```
delegate void EventHandler(Object^ source, double a);
public ref class Manager //издатель
{
public:
	event EventHandler^ OnHandler;
	void method()
	{
		OnHandler(this, 10.);
	}
}

public ref class Watcher //подписчик
{
public:
	Watcher(Manager^ m)
	{
		m->OnHandler += gcnew EventHandler(this, &Watcher::f); //& - потомучто метод класса
	}

//подписали метод f на возникновение события. Когда начнёт выполняться method, будут выполняться методы классов, подписанных на возникновение события
	void f(Object^ source, double a)  {…}
}
```