ЭКЗАМЕН ПО ООП

Технология

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.

Преимущества и недостатки ООП. Основные понятия.

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.

Этапы разработки ПО с использованием объектно-ориентированного подхода.

1. Анализ – строим модель нашей программы.
2. Проектирование – можно полностью автоматизировать.
3. Эволюция – процесс создания продукта.
4. Модификация – после того, когда мы получаем готовый продукт.

Преимущества эволюции

1. пользователю предоставляется обширная обратная связь.
2. предоставляются различные версии структур системы (обеспечивает плавный переход от старой системы к новой)
3. меньше возможности отмены проекта

Изменения на этапе эволюции

1. Добавление класса
2. изменение реализации класса
3. изменение представления класса
4. реорганизация структуры класса
5. изменение интерфейса класса

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.
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.

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.

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

Наследование – на основе одного класса, мы строим новый класс, путем добавления новых характеристик и методов.
Использование – один класс вызывает методы другого класса. (акцессор)
Наполнение – это когда один класс содержит другие классы.
Метаклассы – класс существующий для создания других классов.

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.

Объектно-ориентированный анализ.

При анализе выделяем: домены (их разбиваем на подсистемы)

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.

Информационное моделирование. Атрибуты класса, связи.

- Выделение сущностей, с которыми мы работаем
- Описание или понятие этой сущности.
    - Выделение атрибутов, каждая характеристика которая является общей для всех возможных экземпляров классов выделяется как отдельный атрибут.
    - Идентификатор – это множество из одного или множества атрибутов, которое определяет класс.
    - Выделение привилегированных атрибутов.
- Графическое представление.
    - Все сущности мы номеруем. Для классов, связей и проч. мы создаём ключевой литерал (1-3 буквы)

Атрибуты

Описательные атрибуты. Какая-то характеристика, внутренне присущая каждому объекту (Zb пол). Если значение описательного атрибута меняется, то какой-то аспект изменяется, но объект остаётся.

Указывающие атрибуты. Которые используются как идентификатор, или как часть идентификатора (Zb имя студента)

Вспомогательные атрибуты. Для формализации связи одного объекта с другими объектами. Для активных объектов будем выделять время жизни. Атрибут состояния.

Правила атрибутов:

1. Один объект класса имеет одно единственное значение для каждого атрибута в любой момент времени.
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 ни нечего другого.
4. Каждый атрибут не явл частью идентификатора, представляет характеристику объекта указанного идентификатором, а не характеристику другого атрибута.

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.

В результате ИМ получим

- Диаграмма информационных структур
- Описание объектов и их атрибутов
- Выделили связи и формализовали их

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

Модель Мура состоит из:

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

ДПС (Диаграмма переходов состояний)

ТПС (Таблица переходов состояний) представляет собой матрицу: строки – это состояния, а столбцы – это события. Вся матрица должна быть заполнена.

Состояние – это положение объектов, в котором определяются определённый набор правил, линий поведений, предписаний, определённых законов. 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
- Текущее (Состояние в котором объект может находиться, которое не является заключительным или начальным.)

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 – события переносят данные. Все события, которые переносят объект из одного состояния в другое должны нести его идентификатор.

Каждому событию ставим в соответствие действие. Все события, которые вызывают переход в одно и тоже состояние должны нести одни и те же данные. Идентификатор события, к которому применяется событие, должен переноситься как данные.

Действие – это деятельность или операция, которая выполняется при достижении объектом состояния. Каждому состоянию ставится в соответствие одно действие. Действие должно выполняться любым объектом одинаково.

Действие может:

1. Выполнять любые вычисления
2. Порождать события для любого класса
3. Порождать события для чего-либо вне области анализа
4. Выполнять все действия над таймером
5. Читать, записывать атрибуты собственного класса и других классов
6. Гарантировать, что выполняя любые действия

Действие должно оставлять данные, описывающие собственный объект, непротиворечивыми

Действие-событие-время.

1. Только одно действие данного конечного автомата может выполняться в любой времени
2. Действия различных конечных автоматов могут выполняться параллельно.
3. События никогда не теряются.
4. Если событие порождено для объекта, который в настоящий момент времени выполняет действие, то данное событие не будет принято, пока действие не будет принято.

5. Каждое событие прекращается, когда оно представляется конечному автомату, само событие исчезает как событие.
6. Не все события всегда обрабатываются, событие может быть игнорировано.

Формы жизненных циклов

1. Циркуляционный
2. Рождение-смерть

Когда формируются жизненные циклы:

1. Создание или уничтожение во время выполнения
2. Миграция между подклассами
3. Объект производится или возникает поэтапно.
4. Объект – задача или запрос
5. Динамическая связь

Динамика систем, схемы взаимодействия, каналы управления, имитирование.

МВО (модель взаимодействия объектов) – графическое представление взаимодействия. Каждая модель состояний – овал. Стрелочки – события.

События, которые приходят в систему – приходят извне, эта внешняя сущность – терминатор.

Типы событий

1. Внешние события (приходят от терминатора)
    a. Не запрашиваемые события (не являются результатом действия предыдущей действенности подсистемы)
    b. Запрашиваемые события
2. Внутренние (порождаются какой-либо моделью состояний нашей подсистемы)
    a. Схема верхнего управления (Те события приходят от терминаторов которые наверху)
    b. Схема нижнего управления (Объекты которые являются программной реализацией чего-либо существующего)

Этап имитирования. Мы генерим некоторое начальное состояние. Принимает незапрашиваемое состояние, смотрим какое состояние приняли все объекты в нашей системе. Процесс имитирования может быть очень сложным.

Канал управления – последовательность действий и событий, которые происходят в ответ на поступление некоторого незапрашиваемого состояния, когда система находится в определённом состоянии. Если возникло событие к терминатору, и эти событии приводят к дальнейшим событиям от терминатора, то мы их тоже включаем в канал управления.

2 времени имитирования:

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

3 этапа имитированя

- Установить начальное состояние системы
- Принять незапрашиваемое события и выполнить канал управления
- Оценить конечный результат

Потоки данных действия. Процессы. Модели доступа к объектам.

ДПДД (Диаграмма потоков данных действий) – обеспечивает графическое представление модулей процесса в пределах действия и взаимодействия между ними. Строится для каждого состояния каждого объекта класса.

Разбиваем действия на процессы, которые могут происходить:

- Процесс проверки
- Процесс преобразования
- Аксессоры (процесс, чья единственная цель состоит в том, чтобы получить доступ к данным одного архива данных)
    - Создание
    - Чтение
    - записи
    - уничтожение
- Генераторы событий (создаёт лишь одно событие как вывод)

Процесс может выполняться когда все входы доступны. Выводы доступны когда процесс завершит управления

Данные событий, архива данных и терминаторов всегда доступны.

Id процесса	Тип	Название процесса	Где используется	
			Модель состояний	Действие

Модели доменного уровня, понятие мостов, клиентов, серверов.

Домен – отдельный реальный или гипотетический отдельный населённый отчётливым набором объектом, который ведёт в соответствии с правилами.

- Прикладные домены
- Сервисные домены (набор сервисов, которые будет использовать прикладной домен)
- Архитектурные домены (обеспечивает единые механизмы и структуры для управления данными и управления всей программой, как единым целым)
- Реализаций (библиотечный класс – взаимодействие по сети, протоколы взаимодействия по сети)

Когда один домен, который использует механизм и возможность обеспечивающуюся другим, говорим, что существует мост. Использует – клиент, обеспечивает – сервер. Клиент рассматривает мост как набор предложений, которые будут, как он считает, представлены другим доменом. Сервер подходит к мосту, как к набору требований для выполнения.

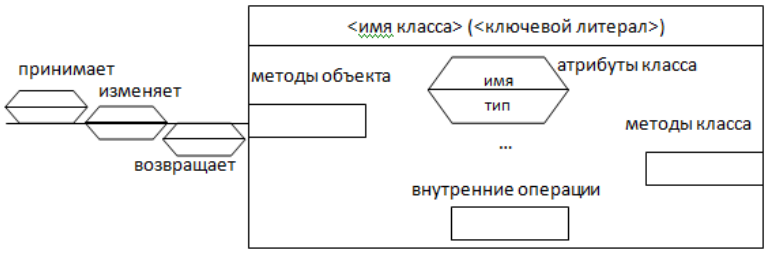
Схема доменов.
Домены и мосты между ними. В овалах – домены. Стрелка – мост. Домен к задаче – внизу, сервисные – внизу.

Строим 3 диаграммы для взаимодействия

- модель связей подсистем (по информационной модели)
- модель взаимодействия подсистем (по МВО)
- модель доступа подсистем (по МДО)
    - Модель доступа к объекта. 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, таблица процессов состояний, диаграмма потомков данных действий.

Объектно-ориентированное проектирование, шаблоны для создания прикладных классов. Диаграмма класса, схема структуры класса, диаграмма зависимости, схема зависимости.

Диаграмма класса описывает внешнее представление данного класса.



Чтобы конкретизировать потоки данных, рисуется *схема структуры класса*.

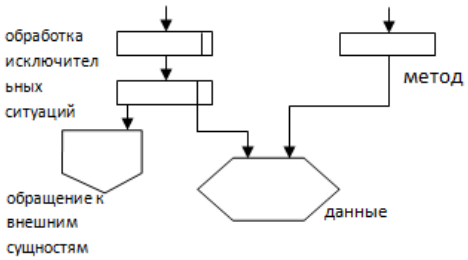


Диаграмма зависимостей – строится диаграмма класса, только оставляем заголовок, а атрибуты перечисляем списком.

Диаграмма наследования – от базового класса подклассы. Внутри атрибуты и методы.

C++

Структура программы на языках С, С++.

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)

Классы и объекты, ограничение доступа.

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.

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`.

Наследование, построение иерархии, множественное наследование и неоднозначности в нём.

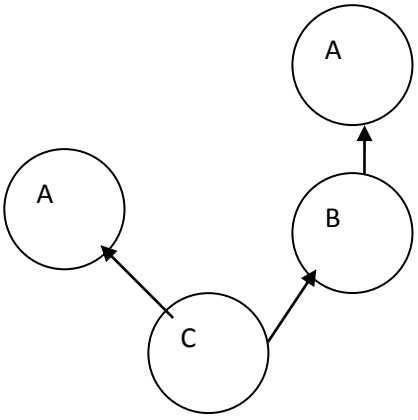
Расширение и выделение общей части из разных классов.

Причины выделения общей части:

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

Расщепление классов

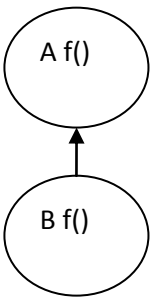
1. Два подмножества операций в классе используются в разной манере.
2. Методы класса имеют не связную реализацию.
3. Класс оперирует очевидным образом в 2-х несвязных обсуждениях проекта.



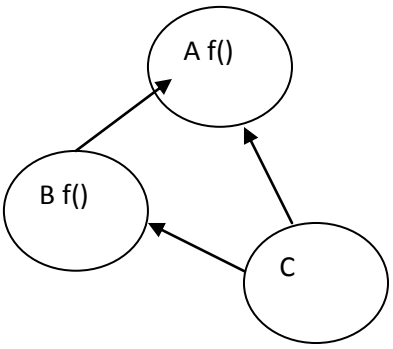
Прямая база – непосредственная база класса. Прямая база может входить только один раз

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

Тут A для C и косвенная и прямая база.



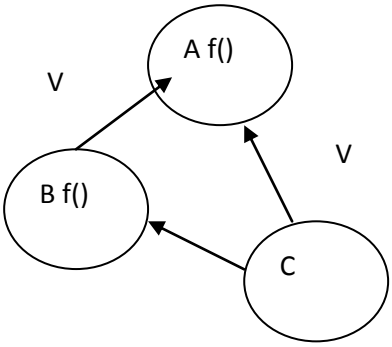
Метод f () в производном классе B доминирует над методом в доминантном классе A.

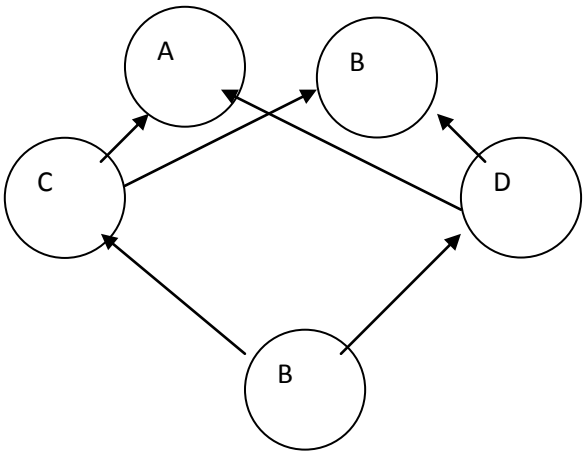


ОШИБКА!

(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!
}
```

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).

Шаблоны классов.

```
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);
```

Обработка исключительных ситуаций.

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 Qt-ных 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

Одиночка

```
template <typename T>
class Singleton
{
public:
    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&);
};
template<T>
T* Singleton<T>::ptr=0;
```

Фабричный метод

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанциировать. 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.

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

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

Абстрактная фабрика

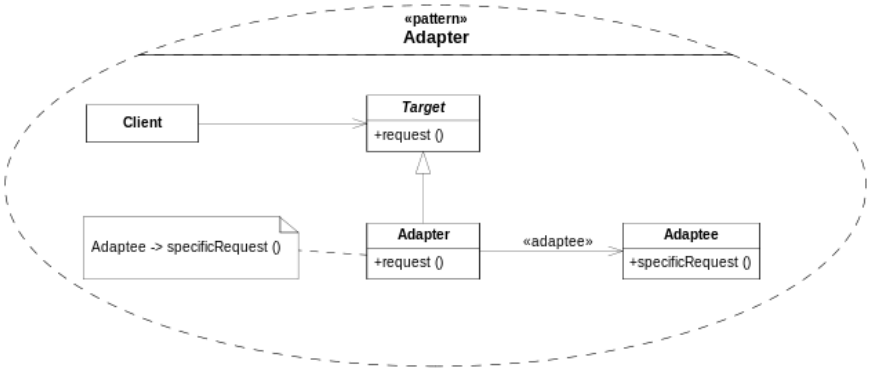
```
class Infantryman{
  public:
    virtual void info() = 0;
    virtual ~Infantryman() {}
};
class Archer{
  public:
    virtual void info() = 0;
    virtual ~Archer() {}
};
class RomanInfantryman: public Infantryman{
  public:
    void info() { cout << "RomanInfantryman" << endl;}
};

class RomanArcher: public Archer{
  public:
    void info() { cout << "RomanArcher" << endl;}
};
class ArmyFactory {
  public:
    virtual Infantryman* createInfantryman() = 0;
    virtual Archer* createArcher() = 0;
    virtual ~ArmyFactory() {}
};
class RomanArmyFactory: public ArmyFactory {
  public:
    Infantryman* createInfantryman() { return new RomanInfantryman;  }
    Archer* createArcher() { return new RomanArcher;  }
};
class Army {
  public:
   ~Army() {
      int i;
      for(i=0; i<vi.size(); ++i)  delete vi[i];
      for(i=0; i<va.size(); ++i)  delete va[i];
    }
    void info() {
      int i;
      for(i=0; i<vi.size(); ++i)  vi[i]->info();
      for(i=0; i<va.size(); ++i)  va[i]->info();
    }
    vector<Infantryman*> vi;
    vector<Archer*> va;
};
class Game {
  public:
    Army* createArmy( ArmyFactory& factory ) {
      Army* p = new Army;
      p->vi.push_back( factory.createInfantryman());
      p->va.push_back( factory.createArcher());
      return p;
    }
};
int main(){
    Game game;
    RomanArmyFactory ra_factory;
    Army * ra = game.createArmy( ra_factory);
    cout << "Roman army:" << endl;
    ra->info();
}
```

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)

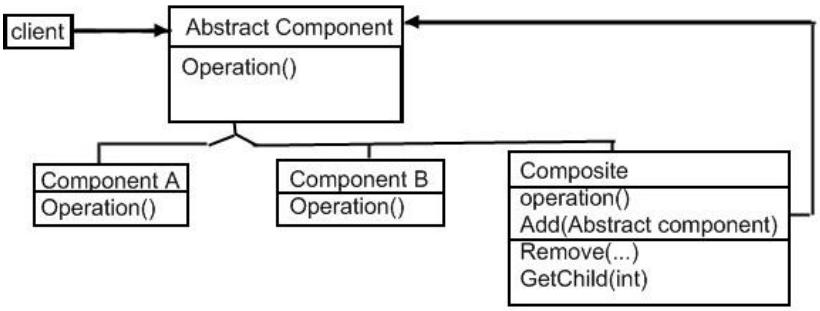
Применяется: система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс. Чаще всего шаблон Адаптер применяется если необходимо создать класс, производный от вновь определяемого или уже существующего абстрактного класса.




```
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;
};
```

Компоновщик

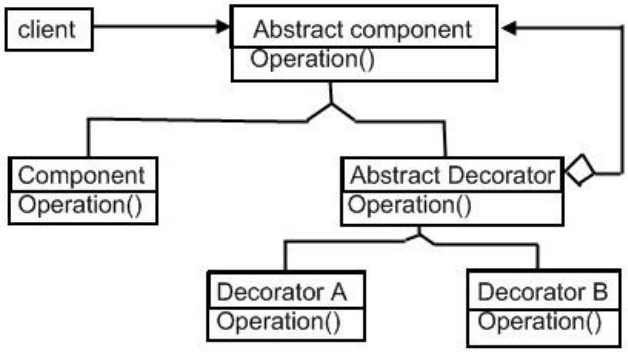
Паттерн определяет иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым.




```
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;
}
```

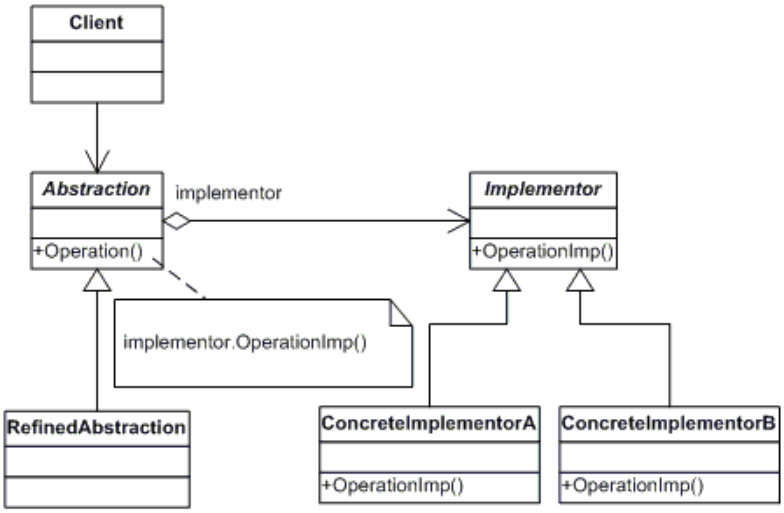
Декоратор




```
class Widget {
  public:
    virtual void draw() = 0;
};
class TextField: public Widget {
    int width, height;
  public:
    TextField(int w, int h) {
        width = w;
        height = h;
    }
    void draw() { cout << "TextField: " << width << ", " << height << 'n'; }
};
class Decorator: public Widget {
    Widget *wid;
  public:
    Decorator(Widget *w) {wid = w;}
    void draw() {wid->draw();}
};
class BorderDecorator: public Decorator {
  public:
    BorderDecorator(Widget *w): Decorator(w) {}
    void draw() {Decorator::draw();cout << "BorderDecorator" << 'n';}
};
Widget *aWidget = new BorderDecorator( new BorderDecorator( (new TextField(80, 24))));
aWidget->draw();
```

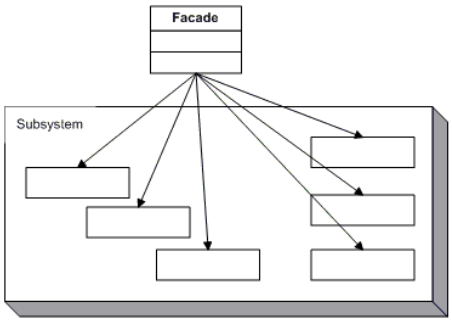
Мост

создать класс, который создает методы, зависимые от среды




```
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();
```

Фасад

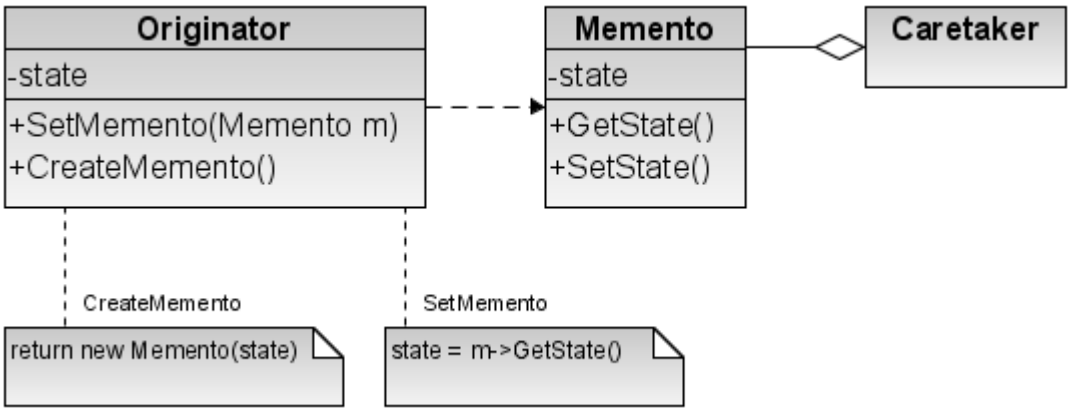


Позволяет скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Поведения

Хранитель

- необходимо сохранить снимок состояния объекта (или его части) для последующего восстановления
- прямой интерфейс получения состояния объекта раскрывает детали реализации и нарушает инкапсуляцию объекта



Originator - "Создатель"

Caretaker - "Опекун"

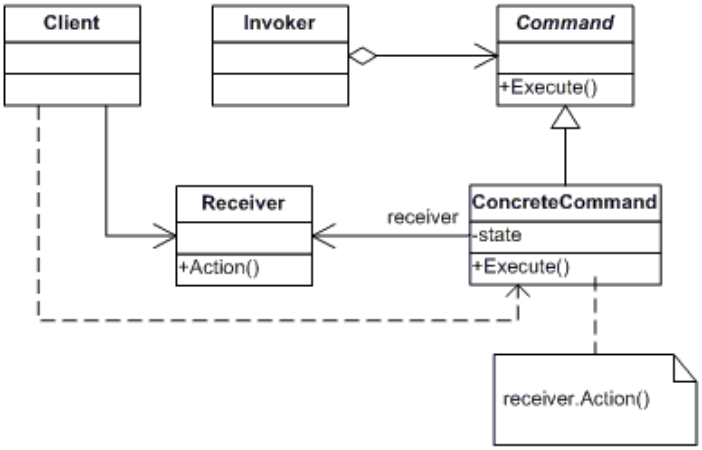
Memento - "Хранитель"

```
template <typename T>
class Holder;
template <typename T>
class Trule {
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>;
};
template <typename T>
class Holder {
private:
    T* ptr;
public:
    Holder() : ptr(0) {}
    explicit Holder(T* p) : ptr(p) {}
    ~Holder() {delete ptr;}
    T& operator *() const {return *ptr;}
    T& get() const {return *ptr;}
    T* operator ->() const {return ptr;}
    void exchange(Holder<T>& h);
    Holder(Trule<T> const& t) {
        ptr = t.ptr;
        const_cast<Trule<T>&>(t).ptr = 0;
    }
    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;
    }
 private:
    Holder(Holder<T> const&);
    Holder<T>& operator =(Holder<T> const&);
};
```

Команда

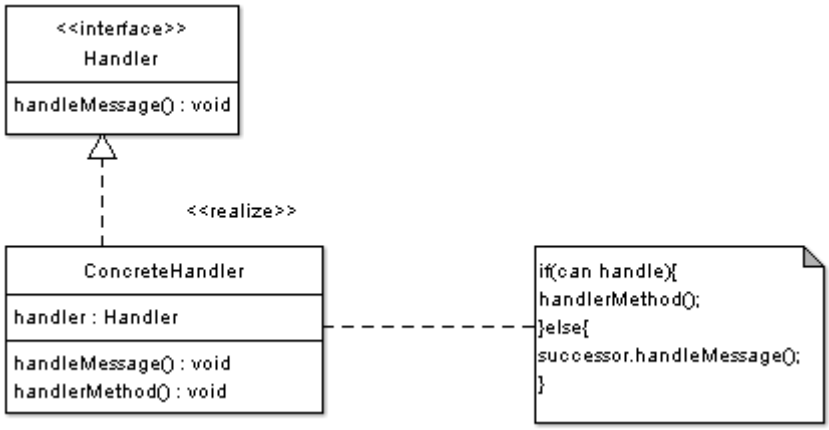
Паттерн поведения объектов, известен так же под именем Action(действие).

Обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве параметра методам, а также возвращать её в виде результата, как и любой другой объект.



Цепочка обязанностей

- в разрабатываемой системе имеется группа объектов, которые могут обрабатывать сообщения определенного типа;
- все сообщения должны быть обработаны хотя бы одним объектом системы;
- 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.



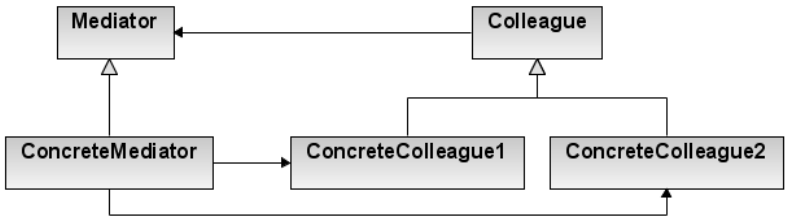
Итератор

Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящий в состав агрегации.

```
class Stack {
    int items[10];
    int sp;
  public:
    friend class StackIter;
    Stack() { sp =  - 1; }
    void push(int in) { items[++sp] = in; }
    int pop() { return items[sp--]; }
    bool isEmpty() { return (sp ==  - 1); }
};
class StackIter {
    const Stack &stk;
    int index;
  public:
    StackIter(const Stack &s): stk(s) { index = 0; }
    void operator++() { index++; }
    bool operator()() { return index != stk.sp + 1; }
    int operator *() { return stk.items[index]; }
};
bool operator == (const Stack &l, const Stack &r)
{
  StackIter itl(l), itr(r);
  for (; itl(); ++itl, ++itr)
    if (*itl !=  *itr)
      break;
  return !itl() && !itr();
}
```

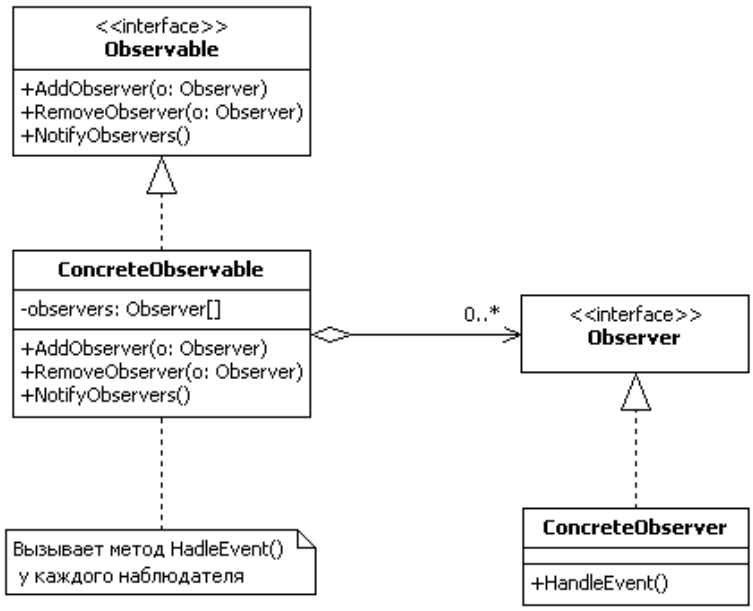
Посредник

Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.



Наблюдатель (подпсичик-издатель)

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.

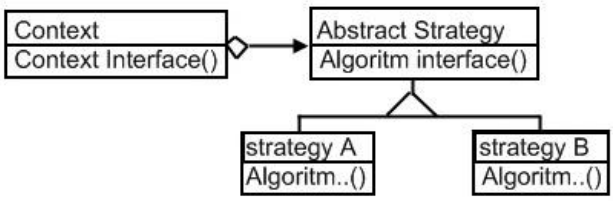
```
class Observer {
  public:
    virtual void update(int value) = 0;
};
class Subject {
    int m_value;
    vector m_views;
  public:
    void attach(Observer *obs) { m_views.push_back(obs); }
    void set_val(int value) { m_value = value; notify(); }
    void notify() {
        for (int i = 0; i < m_views.size(); ++i)
          m_views[i]->update(m_value);
    }
};
class DivObserver: public Observer {
    int m_div;
  public:
    DivObserver(Subject *model, int div) {
        model->attach(this);
        m_div = div;
    }
    void update(int v) { cout << v / m_div << 'n'; }
};
int main() {
  Subject subj;
  DivObserver divObs1(&subj, 4);
  DivObserver divObs2(&subj, 3);
  subj.set_val(14);
}
```

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. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.




```
class Compression{
  public:
    virtual ~Compression() {}
    virtual void compress( const string & file ) = 0;
};
class ZIP_Compression : public Compression{
  public:
    void compress( const string & file ) { cout << "ZIP" << endl;}
};
class RAR_Compression : public Compression{
  public:
    void compress( const string & file ) {cout << "RAR" << endl;}
};
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);
```