не надо писать “нахера так много???”
если не нравится редактируйте сами, я не буду уменьшать объем
Вопросы к экзамену
1. Code Style

> **Станда́рт оформле́ния ко́да** (**станда́рт коди́рования**, **стиль программи́рования**) (англ. *coding standards*, *coding convention* или *programming style*) — набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования. Наличие общего стиля программирования облегчает понимание и поддержание исходного кода, написанного более чем одним программистом, а также упрощает взаимодействие нескольких человек при разработке программного обеспечения.

Обычно, стандарт оформления кода описывает:

- способы выбора названий и используемый регистр символов для имён переменных и других идентификаторов:
    - запись типа переменной в её идентификаторе (венгерская нотация) и
    - регистр символов (нижний, верхний, «верблюжий», «верблюжий» с малой буквы), использование знаков подчёркивания для разделения слов;
- стиль отступов при оформлении логических блоков — используются ли символы табуляции, ширина отступа;
- способ расстановки скобок, ограничивающих логические блоки;
- использование пробелов при оформлении логических и арифметических выражений;
- стиль комментариев и использование документирующих комментариев.

Вне стандарта подразумевается:

- отсутствие магических чисел;
- ограничение размера кода по горизонтали (чтобы помещался на экране) и вертикали (чтобы весь код файла держался в памяти), а также функции или метода в размер одного экрана.

Примеры готовых стилей:

★ Google Style
★ MS Венгерская нотация
★ Facebook Java Coding Standards
★ Python PEP8

2. Системы контроля версий

Система управления версиями (от англ. *Version Control System, VCS* или *Revision Control System*) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Централизованные системы контроля версий

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как: CVS, Subversion и Perforce, имеют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет.

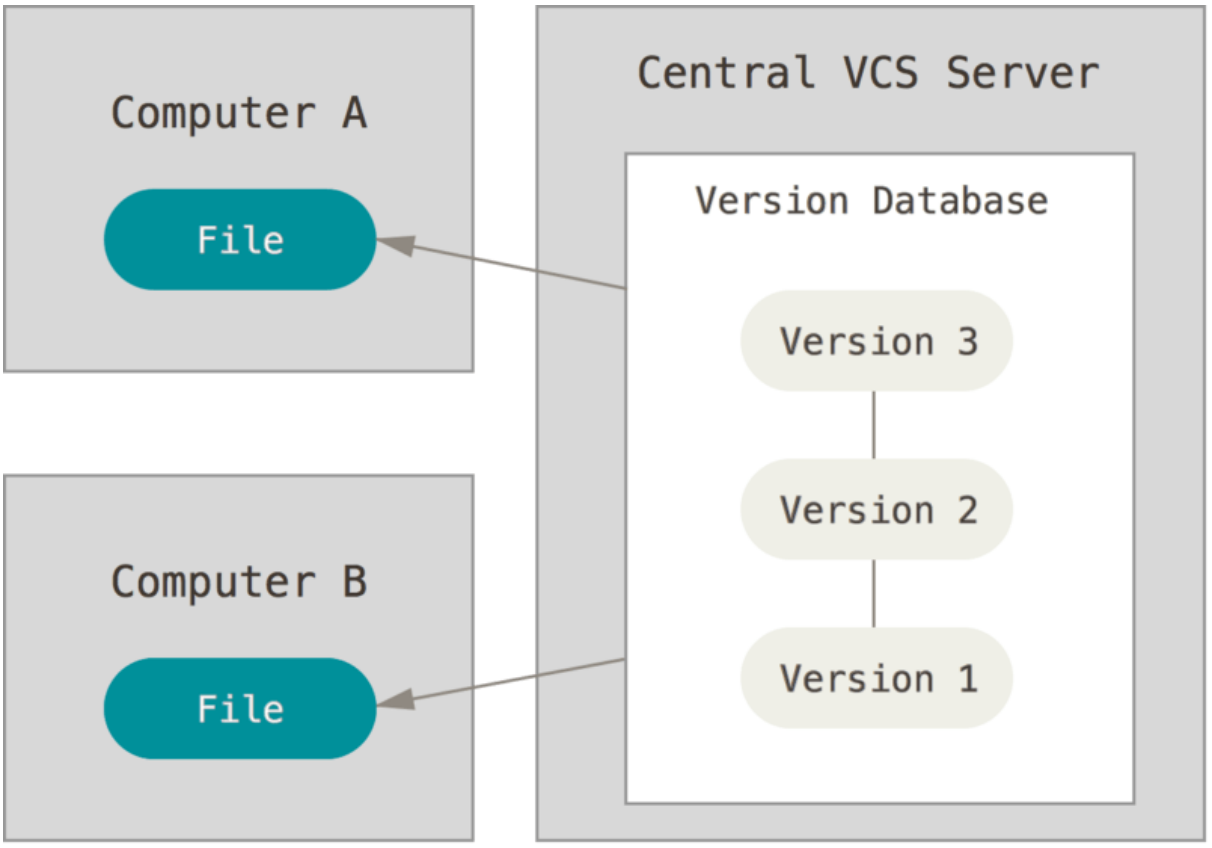


Figure 2. Централизованный контроль версий.

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми он работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы — когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

Децентрализованные системы контроля версий

Здесь в игру вступают децентрализованные системы контроля версий (ДСКВ). В ДСКВ (таких как Git, Mercurial, Bazaar или Darcs), клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени): они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

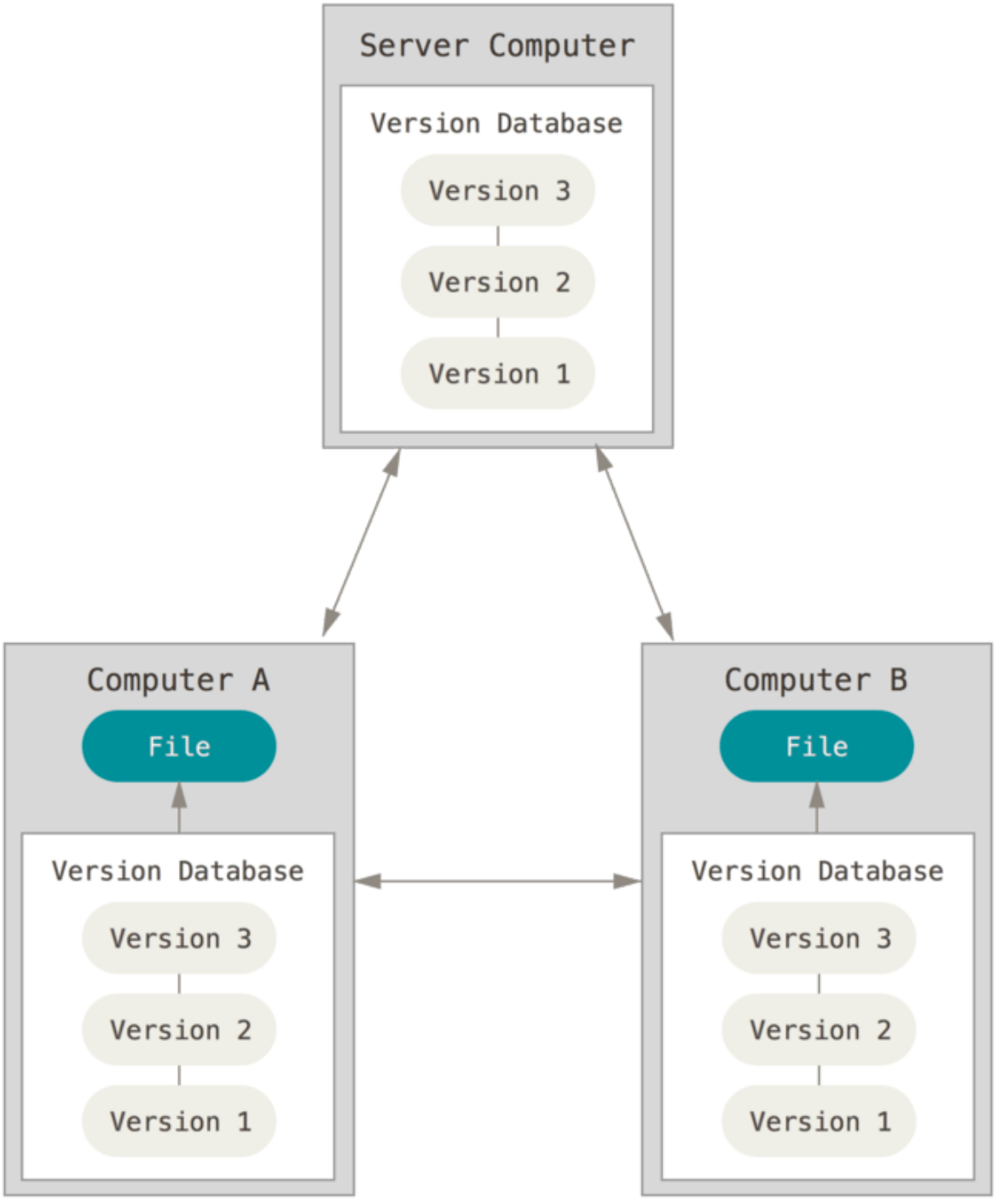


Figure 3. Децентрализованный контроль версий.

Более того, многие ДСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно, в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

Subversion(также известная как «**SVN**»)

Использование Subversion[

Рабочий цикл

Типичная итерация рабочего цикла с Subversion включает следующие этапы.

- Обновление рабочей копии из хранилища (svn update) или её создание (svn checkout).
- Изменение рабочей копии. Изменения директорий и информации о файлах производится средствами Subversion, в *изменении* же (содержимого) файлов Subversion никак не задействован — изменения производятся программами, предназначенными для этого (текстовые редакторы, средства разработкии т. п.):
    - новые (ещё не зафиксированные в хранилище) файлы и директории нужно *добавить* (команда svn add), то есть передать под управление версиями;
    - если файл или директорию в рабочей копии нужно *удалить*, *переименовать*, *переместить* или *скопировать*, необходимо использовать средства Subversion (svn mkdir, svn delete, svn move, svn copy);
    - просмотр состояния рабочей копии и локальных (ещё не зафиксированных) изменений (svn info, svn status, svn diff);
    - любые локальные изменения, если они признаны неудачными, можно *откатить* (svn revert).
- При необходимости — дополнительное обновление, для получения изменений, зафиксированных в хранилище другими пользователями и слияния этих изменений со своими (svn update).
- Фиксация своих изменений (и/или результатов слияния) в хранилище (svn commit).

Ветвление

Ветвление является важным аспектом работы систем управления версиями, поскольку типичные приёмы управления версиями (по крайней мере, при разработке программного обеспечения) включают в себя использование ветвей. Subversion обладает достаточно развитыми возможностями для ветвления и слияния (однако не поддерживает слияние переименованных файлов и директорий).

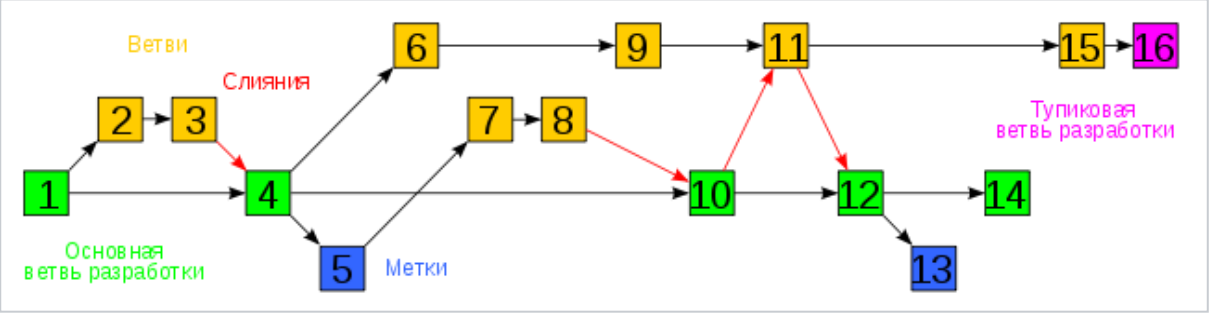


Рис. 3. Пример эволюции ветвей в Subversion

На рис. 3 условно показан пример эволюции ветвей в хранилище. Зелёным цветом показана основная линия разработки проекта (англ. *mainline, trunk*), жёлтым — ветви, синим — метки, пурпурным — ветвь, разработка которой прекращена. Красными стрелками показаны слияния изменений.

Создание ветвей

Новая ветвь (а также метка) создаётся командой svn copy, которая создаёт в хранилище копию с наследованием истории ревизий источника (операция A+). Для создания ветвей всегда следует использовать «удалённую» форму команды svn copy, например:

```
svn copy http://.../trunk/dir http://.../branches/branch_name -m "Creating a branch of dir"
```

Полученная копия будет ветвью (или меткой, в зависимости от способа работы с ней). В дальнейшем изменения, сделанные на ветви, могут быть внесены в источник, из которого была создана эта ветвь, такое распространение изменений называется *слияние* (англ. *merge*).

Операции копирования в Subversion *дешёвые* (англ. *cheap copy*), то есть требуют небольшого фиксированного количества времени и дискового пространства. Хранилище спроектировано таким образом, что при любом копировании происходит не дублирование данных, а создание ссылки на источник (аналогично жёсткой ссылке), однако этот механизм чисто внутренний — с точки зрения пользователя происходит именно создание копии. Благодаря высокой эффективности создания ветвей их можно создавать настолько часто,

насколько это необходимо (однако *слияние* — необходимое дополнение к ветвлению — выполняется в Subversion медленнее, чем в других современных системах управления версиями).

Работа с ветвями

В целом работа на ветви не отличается от работы на основной линии разработки (*trunk*). Специфичные команды требуются только для действий, в которых задействовано более одной ветви. К таким командам относятся (помимо команды создания ветви svn copy):

- svn switch — переключение рабочей копии на другую ветвь — используется для того, чтобы переключить имеющуюся рабочую копию на другую ветвь. В результате переключения служебные данные рабочей копии изменяются так, как будто эта рабочая копия получена операцией svn checkout из той ветви, на которую она переключена. При этом объём сетевого трафика меньше, чем при svn checkout, так как передается только разница между данными в рабочей копии и целевой ветвью;
- svn merge — копирование набора изменений между ветвями — используется для слияния.

Как правило, полный цикл работы с ветвями включает следующие этапы:

- создание ветви (svn copy);
- переключение имеющейся рабочей копии на ветвь (svn switch) или создание новой рабочей копии путём закачки (svn checkout);
- изменение файлов и директорий в рабочей копии, фиксация этих изменений (svn commit);
- копирование в ветвь свежих изменений из родительской ветви, сделанных после ветвления (svn merge, svn commit);
- копирование изменений из ветви в родительскую ветвь (svn merge, svn commit);
- удаление ветви (svn delete), если её жизненный цикл закончен.

Слияние

Копирование изменений между ветвями

Слияние в Subversion — это применение к ветви набора изменений, сделанных на другой (или той же самой) ветви. Для осуществления слияния необходимо использовать команду svn merge — она применяет набор изменений к рабочей копии; затем нужно зафиксировать внесённые изменения (svn commit).

Терминология, связанная со слиянием, несколько запутана. Термин *слияние* (англ. *merge*) является не совсем точным, поскольку как такового объединения

ветвей не происходит. Кроме того, не следует отождествлять *слияние* и команду svn merge: во-первых, для слияния нужно выполнить (помимо svn merge) разрешение конфликтов и фиксацию, во-вторых, применение svn merge не ограничивается слиянием.

Другие применения команды svn merge

Команду svn merge можно использовать не только для слияния. Фактически команда производит *внесение в рабочую копию изменений, равных разнице между двумя директориями или файлами в хранилище*, поэтому svn merge является универсальным средством для переноса изменений. Можно привести такие примеры использования команды svn merge:

- откат уже зафиксированных изменений, в том числе целого диапазона ревизий;
- удобный просмотр (в виде изменений в рабочей копии) разницы между двумя состояниями репозитория.

Централизованный SVN
Использование
Checkout, Update
Commit, Show log
Коммиты-большие, маленький коммит может сломать билд
Ветки кода
Большая фича == ветка
Много коммитов в ветку
1 коммит (merge) в основную

Создание хранилища

Для создания хранилища используется команда svnadmin create. Эта операция создаст пустое хранилище в указанной директории.

Вот вроде бы адекватное краткое описание гита
https://proglib.io/p/git-for-half-an-hour/

Децентрализованный Git

Команды:

- ➢ Init / Clone
- ➢ Add / Commit

- ➢ Pull / Push

Коммиты–маленькие

теперь это никому не помешает

Ветки – для организации

1 ветка == 1 фича

С поля боя

•master -тут живет стабильная версия кода

•develop - основная ветка для работы

•branches - для каждой фичи - своя ветка

•branches->develop - протестировали свою фичу.

•develop->master - протестировали конкретную версию develop и влили в мастер

3. Антипаттерны:
Анти-паттерны — полная противоположность паттернам. Если паттерны проектирования — это примеры практик хорошего программирования, то есть шаблоны решения определённых задач. То анти-паттерны — их полная противоположность, это — шаблоны ошибок, которые совершаются при решении различных задач. Частью практик хорошего программирования является именно избежание анти-паттернов.

1. Copy and Paste Programming

Когда от программиста требуется написание двух схожих функций, самым «простым» решением является написание одной функции, её копирование и внесение некоторых изменений в копию. Какие проблемы это сулит? Во-первых, ухудшается переносимость кода — если потребуется подобный функционал в другом проекте, то надо будет искать все места, где программист накопипастил и переносить их по отдельности. Во-вторых, снижается качество кода — часто программист забывает вносить требуемые изменения в скопированный код. В-третьих, усложняется поддержка кода — так, как если в изначальном варианте был баг, который в будущем надо будет исправить, то этот баг попал во все то, что, опять-таки, накопипастил программист. Это приводит также к возникновению различных множественных исправлений, которые будут возникать по мере нахождения бага в разных частях кода, для одного единственного бага. В-четвертых, code review значительно усложняется, так как кода становится больше, без видимой

значительной выгоды и роста производительности труда. Главными причинами возникновения подобных ошибок являются — отсутствие мыслей про будущее разработки (программисты не продумывают свои действия), недостаток опыта (программисты берут готовые примеры и модифицируют, адаптируя под свои нужды). Решение же, является очень простым — создание общих решений и их использование. Об этом следует думать еще при разработке решений небольших задач — возможно, что нам потребуется решить эту задачу где-нибудь ещё, или решить эту же задачу, но в другой интерпретации.

2. Cryptic code

Использование аббревиатур вместо мнемоничных имён (понятных, упрощающих понимание кода p.s это просто мое мнение) пример из лекции:

```
int p = 0;
int PP = p + 1;
double FrqFRDfs = FrqFRDfd;
```

3. Бездумное комментирование

Результат «работы» данного анти-паттерна — большое количество лишних и неинформативных комментариев.

4. Reinventing the wheel

Смысл этого анти-паттерна в том, что программист разрабатывает собственное решение для задачи, для которой уже существуют решения, очень часто лучшие чем придуманное программистом.

5. Programming by permutation

Многие начинающие программисты пытаются решать некоторые задачи методом перебора — не брутфорсом решения, а именно подбором параметров, порядка вызова функций и так далее. Все эти игры с +1, -1 к параметрам и подобные штучки устраняют только симптомы, и не дают понимания сути происходящего.

6. Boat anchor

Этот анти-паттерн означает сохранение неиспользуемых частей системы, которые остались после оптимизации или рефакторинга. Часто, после рефакторинга кода, который является результатом анти-паттерна, некоторые части кода остаются в системе, хотя они уже больше не используются. Так же некоторые части кода могут быть оставлены «на будущее», авось придётся ещё их использовать. Такой код только усложняет системы, не неся абсолютно никакой практической ценности. Эффективным методом борьбы с лодочными якорями является рефакторинг кода для их устранения, а так же процесс планирования разработки, с целью предотвращения возникновения якорей.

7. Blind faith

Недостаточная проверка корректности исправления ошибки или результата работы подпрограммы

8. Big ball of Mud

Система с нераспознаваемой структурой

9. Golden hammer

Золотой молоток — уверенность в полной универсальности любого решения. На практике, это — применение одного решения (чаще всего какого-либо одного паттерна проектирования) для всех возможных и невозможных задач.

10. Hard code

Жёсткое кодирование — внедрение различных данных об окружении в реализацию. Например — различные пути к файлам, имена процессов, устройств и так далее. Этот анти-паттерн тесно связан с магическими числами, частенько они переплетаются. Захардкодить — жёстко прописать значение каких-либо данных в коде. Главная опасность, исходящая от этого анти-паттерна — непереносимость. В системе разработчика код будет исправно работать до перемещения или переименования файлов, изменения конфигурации устройств. На любой другой системе код может вовсе не заработать сразу же. Как правило, программист практически сразу забывает где и что он захардкодил, даже если делает это в целях отладки кода.

11. Soft code

Мягкое кодирование — параноидальная боязнь жёсткого кодирования. Это приводит к тому, что незахардкожено и настраивается абсолютно всё, что делает конфигурацию невероятно сложной и непрозрачной. Этот анти-паттерн является опасным. Во-первых, при разработке много ресурсов уходит на реализацию возможности настроек абсолютно всего. Во-вторых, развёртывание такой системы повлечет так же дополнительные затраты. Перед началом решения определённой задачи следует определить, что должно быть настариваемым, а что является постоянным для различных систем или может быть настроено автоматически.

12. God Object

Божественный объект — анти-паттерн, который довольно часто встречается у ООП разработчиков. Такой объект берет на себя слишком много функций и/или хранит в себе практически все данные. В итоге мы имеем непереносимый код, в котором, к тому же, сложно разобраться. Так же, подобный код довольно сложно поддерживать, учитывая, что вся система зависит практически только от него. Бороться с таким подходом надо — разбивать задачи на подзадачи, с возможностью решения этих подзадач различными разработчиками.

13. Раздувание интерфейса

Making an interface so powerful that it is extremely difficult to implement. Задумать интерфейс слишком сложным для того, чтобы реализовать его.
https://en.wikipedia.org/wiki/Anti-pattern

14. Смешивание логики и UI-кода

Антипаттерн состоит из системы, разделенной на две части: пользовательский интерфейс и бизнес логика (создание, обработка, запись данных), которые связаны через единственную точку - нажатие “магической” кнопки.

https://en.wikipedia.org/wiki/Magic_pushbutton

15. Better call Super

    Суть в том, что

    https://en.wikipedia.org/wiki/Call_super

16.

более подробно https://habrahabr.ru/post/59005/ но не все=(

4. DRY

DRY - Don’t repeat yourself

Don’t repeat yourself, **DRY** (рус. *не повторяйся*) — это принцип разработки программного обеспечения, нацеленный на снижение повторения информации различного рода, особенно в системах со множеством слоёв абстрагирования. Принцип DRY формулируется как: «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы».

https://ru.wikipedia.org/wiki/Don%E2%80%99t_repeat_yourself

Если код не дублируется, то для изменения логики достаточно внесения исправлений всего в одном месте и проще тестировать одну (пусть и более сложную) функцию, а не набор из десятков однотипных. Следование принципу DRY всегда приводит к декомпозиции сложных алгоритмов на простые функции. А декомпозиция сложных операций на более простые (и повторно используемые) значительно упрощает понимание программного кода. Повторное использование функций, вынесенных из сложных алгоритмов, позволяет сократить время разработки и тестирования новой функциональности.

Следование принципу DRY приводит к модульной архитектуре приложения и к чёткому разделению ответственности за бизнес-логику между программными классами. А это — залог сопровождаемой архитектуры. Хотя чаще не DRY приводит к модульности, а уже модульность, в свою очередь, обеспечивает принципиальную возможность соблюдения этого принципа в больших проектах.

5. KISS

KISS - Keep it stupid simple

KISS — это принцип проектирования и программирования, при котором простота системы декларируется в качестве основной цели или ценности. Есть два варианта расшифровки аббревиатуры: «keep it simple, stupid» и более корректный «keep it short and simple».
В проектировании следование принципу KISS выражается в том, что:

- не имеет смысла реализовывать дополнительные функции, которые не будут использоваться вовсе или их использование крайне маловероятно, как правило, большинству пользователей достаточно базового функционала, а усложнение только вредит удобству приложения;
- не стоит перегружать интерфейс теми опциями, которые не будут нужны большинству пользователей, гораздо проще предусмотреть для них отдельный «расширенный» интерфейс (или вовсе отказаться от данного функционала);
- бессмысленно делать реализацию сложной бизнес-логики, которая учитывает абсолютно все возможные варианты поведения системы, пользователя и окружающей среды, — во-первых, это просто невозможно, а во-вторых, такая фанатичность заставляет собирать «звездолёт», что чаще всего иррационально с коммерческой точки зрения.

В программировании следование принципу KISS можно описать так:

- не имеет смысла беспредельно увеличивать уровень абстракции, надо уметь вовремя остановиться;
- бессмысленно закладывать в проект избыточные функции «про запас», которые может быть когда-нибудь кому-либо понадобятся (тут скорее правильнее подход согласно принципу YAGNI);
- не стоит подключать огромную библиотеку, если вам от неё нужна лишь пара функций;
- декомпозиция чего-то сложного на простые составляющие — это архитектурно верный подход (тут KISS перекликается с DRY);
- абсолютная математическая точность или предельная детализация нужны не всегда — большинство систем создаются не для запуска космических шаттлов, данные можно и нужно обрабатывать с той точностью, которая достаточна для качественного решения задачи, а детализацию выдавать в нужном пользователю объёме, а не в максимально возможном объёме.

6. YAGNI

You aren’t gonna need it

Если упрощенно, то следование данному принципу заключается в том, что возможности, которые не описаны в требованиях к системе, просто не должны реализовываться. Это позволяет вести разработку, руководствуясь экономическими критериями — Заказчик не должен оплачивать ненужные ему функции, а разработчики не должны тратить своё оплачиваемое время на реализацию того, что не требуется.

Основная проблема, которую решает принцип YAGNI — это устранение тяги программистов к излишней абстракции, к экспериментам «из интереса» и к реализации функционала, который сейчас не нужен, но, по мнению разработчика, может либо вскоре понадобиться, либо просто будет полезен, хотя в реальности такого очень часто не происходит.

https://web-creator.ru/articles/<и тут название принципа>
7.SOLID:

1. Single Responsibility

Принцип единственной обязанности / ответственности (single responsibility principle) обозначает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.

2. Open/Close Architecture

Принцип открытости / закрытости декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.

3. Liskov Substitution Principle

Принцип подстановки Барбары Лисков (Liskov substitution) в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом».
Принцип подстановки Барбары Лисков:

- Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T
- Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.
- Понятия замещения — если S является подтипом T, тогда объекты типа T в программе могут быть замещены объектами типа S без каких-либо изменений желательных свойств этой программы

4. Interface Segregation Principle

Принцип разделения интерфейса (interface segregation) в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

5. Dependency Inversion

Принцип инверсии зависимостей (dependency inversion) — модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.

https://habrahabr.ru/post/313796/ - это более понятное описание последнего принципа(если вам и так все понятно можете не смотреть)

https://web-creator.ru/articles/solid + презентация

8.Понятие ресурса

Примеры:

- Память (в языках без GC)
- Mutex, semaphore…
- Файлы
- Состояния внутри программы …

  =(мне кажется здесь можно что нибудь дописать

9.Утечка ресурсов и способы ее решения в разных языках

Утечка ресурсов:

- Незакрытые файлы
- Утекшая память
- Нарушение инвариантов программы

ОПЕРАЦИОННАЯ СИСТЕМА при аварийном завершении:

- ➢ Закроет файлы
- ➢ Освободит mutex
- ➢ Освободит память

TRY…FINALLY:

```
sr = SomeResource()
sr.lock() try:
        # Some Logic
finally:
        sr.unlock()
```

FINALLY Гарантирует выполнение «финализирующего» кода в случае возникновения exception

ПРОБЛЕМЫ FINALLY

- ➔ Постоянно писать finally - это утомительно
- ➔ Можно забыть закрыть ресурс (например, добавили еще один)
- ➔ Не везде есть (C++)

C++

unique_ptr – классический умный указатель

PYTHON

Менеджер контекста
https://lancelote.gitbooks.io/intermediate-python/content/book/context_managers.html (если вы не знаете что это)

Go:

```
func main() {
        f := createFile("/tmp/defer.txt")
        defer closeFile(f)
        writeFile(f)
}
```

```
C# // Context - наследуется от IDisposable
using(new Context()) {
        #SomeLogic
 }
```

https://msdn.microsoft.com/en-us/library/ms814165.aspx
(если непонятно что происходит)

```
        =(  //мне кажется здесь можно что нибудь дописать
```

10. Множественное владение объектом

МНОГО ВЛАДЕЛЬЦЕВ

- Счетчик ссылок

За что отвечает Garbage Collector ?

Garbage Collector должен делать всего две вещи:

- ❏ Обнаруживать мусор
- ❏ Очищать память от мусора

Reference counting

Суть подхода состоит в том, что каждый объект имеет счетчик. Счетчик хранит информацию о том, сколько ссылок указывает на объект. Когда ссылка уничтожается, счетчик уменьшается. Если значение счетчика равно нулю, - объект можно считать мусором и память можно очищать.

Главным минусом такого подхода является сложность обеспечения точности счетчика. Также при таком подходе сложно выявлять циклические зависимости (когда два объекта указывают друг на друга, но ни один живой объект на них не ссылается). Это приводит к утечкам памяти.

В общем, Reference counting редко используется из за недостатков. Во всяком случае HotSpot VM его не использует. По этому мы можем отложить в памяти, что такой подход есть, и продолжить дальше.

- shared_ptr
  std::shared_ptr – умный указатель, с разделяемым владением объектом через его указатель. Несколько указателей shared_ptr могут владеть одним и тем же объектом; объект будет уничтожен, когда последний shared_ptr, указывающий на него, будет уничтожен или сброшен. Объект уничтожается с использованием delete-expression или с использованием пользовательской функции удаления объекта, переданной в конструктор shared_ptr.

  shared_ptr может не владеть ни одним объектом, в этом случае он называется *пустым*.

  shared_ptr отвечает требованиям CopyConstructible и CopyAssignable.

- промежуточные объекты
  =(

11. Понятие ошибки и их виды
Баг — жаргонное слово, обычно обозначающее ошибку в программе или системе, из- за которой программа выдает неожиданное поведение и, как следствие, результат. Большинство багов возникают из-за ошибок, допущенных разработчиками программы в её исходном коде, либо в её дизайне. Также некоторые баги возникают из-за некорректной работы компилятора, вырабатывающего некорректный код. Программу, которая содержит большое число багов и/или баги, серьёзно ограничивающие её работоспособность, называют нестабильной или, на жаргонном языке, «глючной», «глюкнутой», «забагованной», «бажной», «баг(а)нутой»).

Ошибки:

- ★ ОШИБКИ ДО КОДИРОВАНИЯ
  - Ошибки, допущенные на этапе проектирования и сбора требований
- ★ ОШИБКИ КОДИРОВАНИЯ
  - Ошибки, допущенные на этапе разработки программы

12. “ОШИБКИ ДО КОДИРОВАНИЯ”

- Ошибки сбора требований
- Ошибки выявления боли пользователя
- Ошибки в составлении команды для проекта
- И т.п.

13. Assert и CompileTimeAssert

Для проверки ошибок в алгоритмах используются замечательные инструменты:

- Assert
  Приводит к аварийному Завершению программы С диагностикой ошибки
  Assert — это специальная конструкция, позволяющая проверять предположения о значениях произвольных данных в произвольном месте программы. Эта конструкция может автоматически сигнализировать при обнаружении некорректных данных, что обычно приводит к аварийному завершению программы с указанием места обнаружения некорректных данных.
  Какие виды assert'ов бывают?
  Assert'ы позволяют отлавливать ошибки в программах на этапе компиляции либо во время исполнения. Проверки на этапе компиляции не так важны — в большинстве случаев их можно заменить аналогичными проверками во время исполнения программы. Иными словами, assert'ы на этапе компиляции являются ничем иным, как синтаксическим сахаром.
- Presume
  Работает только в debug

ТОЛЬКО ДЛЯ ГУРУ

Для тех, кто знает, что некоторые вещи можно проверить на этапе компиляции

- compileTimeAssert(expr)

```
enum TColors {
        TC_Red,
        TC_Blue,
        TC_GReen,
        TC_Count
};
compileTimeAssert( TC_Count == 3 );
```

https://news.synopsys.com/2017-05-23-Synopsys-New-Superscalar-ARC-HS-Processors-Boost-RISC-and-DSP-Performance-for-High-End-Embedded-Applications?cmp=IP-emb-wa

14. Исключения

Обрабо́тка исключи́тельных ситуа́ций (англ. *exception handling*) — механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (*исключения*), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма. В русском языке также применяется более короткая форма термина: «**обработка исключений**».

Обработка исключительных ситуаций самой программой заключается в том, что при возникновении исключительной ситуации управление передаётся некоторому заранее определённому *обработчику* — блоку кода, процедуре, функции, которые выполняют необходимые действия.

Существует два принципиально разных механизма функционирования обработчиков исключений.

- **Обработка с возвратом** подразумевает, что обработчик исключения ликвидирует возникшую проблему и приводит программу в состояние, когда она может работать дальше по основному алгоритму. В этом случае после того, как выполнится код обработчика, управление передаётся обратно в ту точку программы, где возникла исключительная ситуация (либо на команду, вызвавшую исключение, либо на следующую за ней, как в некоторых старых диалектах языка BASIC) и выполнение программы продолжается. Обработка с возвратом типична для обработчиков асинхронных исключений (которые обычно возникают по причинам, не связанным прямо с выполняемым кодом), для обработки синхронных исключений она малопригодна.
- **Обработка без возврата** заключается в том, что после выполнения кода обработчика исключения управление передаётся в некоторое, заранее заданное место программы, и с него продолжается исполнение. То есть, фактически, при возникновении исключения команда, во время работы которой оно возникло, заменяется на безусловный переход к заданному оператору.

более подробно здесь:

https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B9

Исключение

- бросить // throw
- «пролетает» сквозь стек вызовов
- поймать // catch
- «отмотать стек», корректно вызвав деструкторы объектов на стеке

```
try{
        throw new A();
}
catch(B*  b ) {
        throw b;
}
catch(A* a) {
        throw;
}
catch( ... ) {
        throw new C();
}
```

ПРОБЛЕМЫ
ИСКЛЮЧЕНИЙ
•Бросать исключения - достаточно дорого (с точки зрения времени выполнения)

15. Класс - Данные и Класс - Механизм
Класс - данные

- Класс для хранения данных
- Внутренний инвариант, согласованное состояние
- Возможность сериализации

1. Рассмотрите возможность иммутабельности данных
2. Нет внутренних состояний методы - сеттеры можно вызывать в любом порядке
3. Возможно состояние IsInitialized

Класс - механизм

- ➢ Сложная операция или несколько похожих операций
  - Достаточно сложна, чтобы не быть методом класса -данных
- ➢ Создаётся прямо перед выполнением операции
- ➢ Основной интерфейс – единственный метод
- ➢ Возможно, метод вызывается для многих запросов
  - неизменные параметры – аргументы конструктора
  - параметры запроса – аргументы метода
- ➢ Между запросами не накапливает данных, влияющих на алгоритм
  - только на скорость
- ➢ Не зависит истории вызовов
- ➢ Не зависит неявно от глобального состояния

ЭТО ИЗ ЛЕКЦИЙ

Классы данных – это классы, которые содержат только поля и простейшие методы для доступа к ним (геттеры и сеттеры). Это просто контейнеры для данных, используемые другими классами. Эти классы не содержат никакой дополнительной функциональности и не могут самостоятельно работать с данными, которыми владеют.

https://refactoring.guru/ru/smells/data-class

16.Паттерны:

Шаблон проектирования или **паттерн** (англ. *design pattern*) в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Архитектура программного обеспечения (англ. *software architecture*) — совокупность важнейших решений об организации программной системы. Архитектура включает:

- выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов;
- соединение выбранных элементов структуры и поведения во всё более крупные системы;
- архитектурный стиль, который направляет всю организацию — все элементы, их интерфейсы, их сотрудничество и их соединение.

https://docs.google.com/file/d/0B6GuCegBf3X3Tm1rZl9BUTduQm8/edit
Это ссылка на хорошую книгу по паттернам

1. Observer

Название и классификация паттерна

➢ Наблюдатель - паттерн поведения объектов.

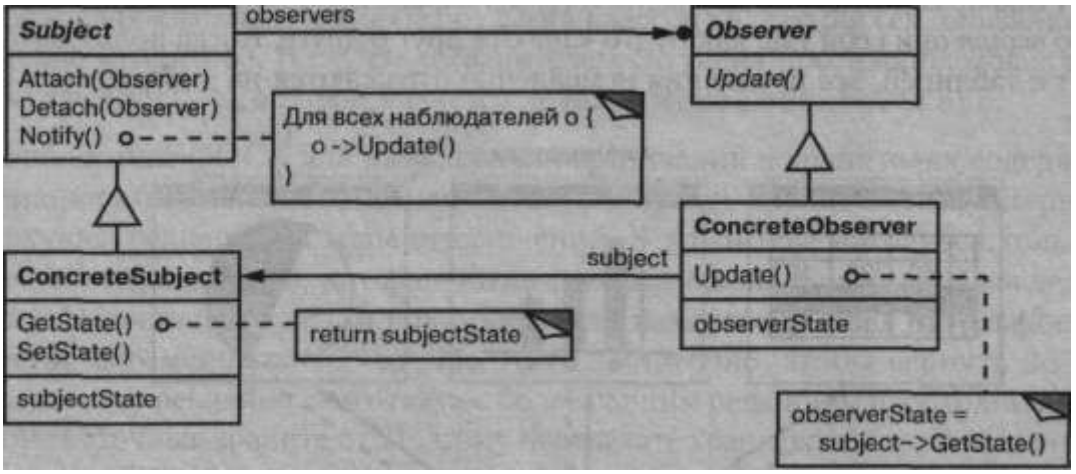
Назначение

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

Применимость

➢ Используйте паттерн наблюдатель в следующих ситуациях:
  - когда у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и по- вторно использовать их независимо; Паттерны поведения
  - когда при модификации одного объекта требуется изменить другие и вы не знаете, сколько именно объектов нужно изменить; а когда один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, вы не хотите, чтобы объекты были тесно связаны между собой.

Структура



Участники

- Subject - субъект:
  - - располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей;
  - - предоставляет интерфейс для присоединения и отделения наблюдателей;
- Observer - наблюдатель:
  - - определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта;
- ConcreteSubject - конкретный субъект:
  - - сохраняет состояние, представляющее интерес для конкретного наблюдателя ConcreteObserver;
  - - посылает информацию своим наблюдателям, когда происходит изменение;
- ConcreteObserver - конкретный наблюдатель:
  - - хранит ссылку на объект класса ConcreteSubject;
  - - сохраняет данные, которые должны быть согласованы с данными субъекта;
  - - реализует интерфейс обновления, определенный в классе Observer, чтобы поддерживать согласованность с субъектом.

2. *Switcher, *Lock
   переключатель состояния(*Switcher, *Lock)

- Специфичный для C++
  - При создании объекта (на стеке) переключаемся в особое состояние
  - При разрушении (автоматически) переключаемся обратно
- «Внешний RAII»
- CMutexLock

- CCriticalSectionLock
- CMemoryManagerSwitcher

А теперь по человечески:

=(

ВИКИПЕДИЯ НЕ ЗНАЕТ ТАКОГО ПАТТЕРНА!!! T_T

3. Singleton

❖ Название и классификация паттерна

➢ Одиночка - паттерн, порождающий объекты.

❖ Назначение

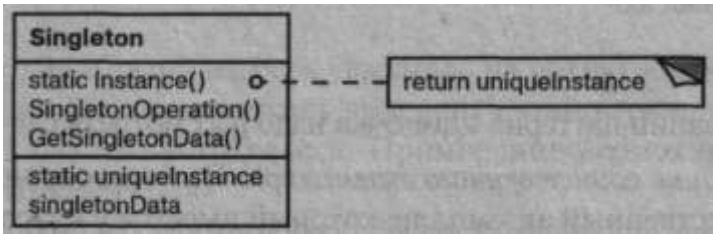
➢ Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

❖ Применимость

➢ Используйте паттерн одиночка, когда:

- должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
- единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

❖ Структура

➢ 

❖ Участники

➢ a Singleton - одиночка:

- - определяет операцию Instance, которая позволяет клиентам получать доступ к единственному экземпляру. Instance - это операция класса, то есть метод класса в терминологии Smalltalk и статическая функция-член в C++;
- - может нести ответственность за создание собственного уникального экземпляра.

4. Decorator

❖ Название и классификация паттерна

➢ Декоратор - паттерн, структурирующий объекты.

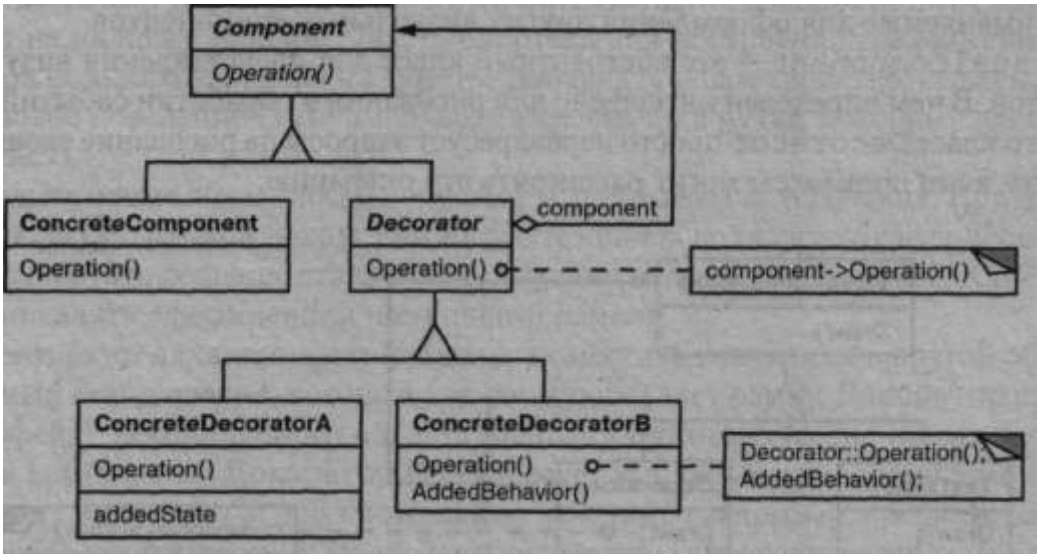
❖ Назначение

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

❖ Применимость

- Используйте паттерн декоратор:
  - для динамического, прозрачного для клиентов добавления обязанностей объектам;
  - для реализации обязанностей, которые могут быть сняты с объекта;
  - когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

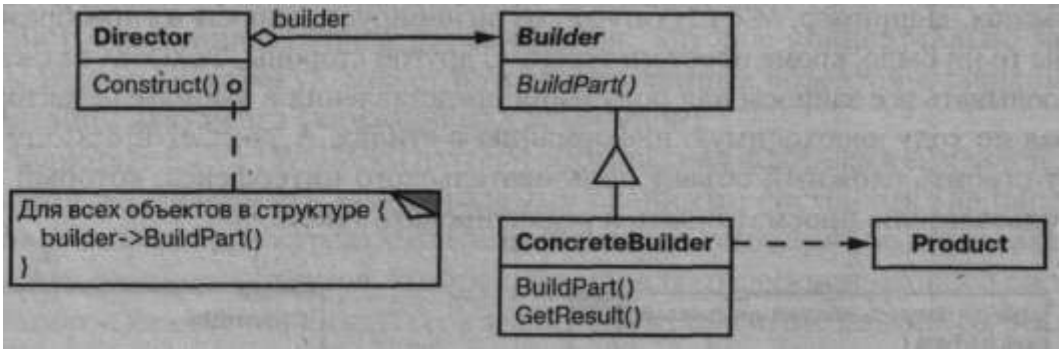
❖ Структура



❖ Участники

- Component (VisualComponent) - компонент:
  - - определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности;
- ConcreteComponent (TextView) - конкретный компонент:
  - определяет объект, на который возлагаются дополнительные обязанности;
- Decorator - декоратор:
  - - хранит ссылку на объект Component и определяет интерфейс, соответствующий интерфейсу Component;
- ConcreteDecorator (BorderDecorator, ScrollDecorator) - конкретный декоратор:
  - - возлагает дополнительные обязанности на компонент.

5. Builder

- ❖ Название и классификация паттерна
  - ➢ Строитель - паттерн, порождающий объекты.
- ❖ Назначение
  - ➢ Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.
- ❖ Применимость
  - ➢ Используйте паттерн строитель, когда:
    - ■ алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
    - ■ процесс конструирования должен обеспечивать различные представления конструируемого объекта.
- ❖ Структура
  - ➢ 

- ❖ Участники
  - ➢ Builder (TextConverter) - строитель:
    - ■ задает абстрактный интерфейс для создания частей объекта Product;
  - ➢ ConcreteBuilder(ASCIIConverter,TeXConverter,TextWidgetConverter) - 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 (например, GetASCI IText,GetTextWidget);
  - ➢ Director (RTFReader) - распорядитель:
    - ■ конструирует объект, пользуясь интерфейсом Builder;
  - ➢ Product (ASCIIText, TeXText, TextWidget) - продукт:

6. Шаблонный метод
   Шаблонный метод, Template Method

- Базовый класс определяет основу алгоритма, но остаётся абстрактным
- Наследники доопределяют поведения алгоритма в конкретных методах

Относится этот паттерн к категории паттернов поведения и служит одной простой цели — переопределению шагов некоторого алгоритма в семействе классов, производных от базового, определяющего структуру этого самого алгоритма.

Допустим, у мы пишем класс Crypt, который предназначен для шифрования некоторой строки текста. В классе определена функция шифрования:

```
void encrypt() {
    // Установка начальных параметров
    setupRnd();
    setupAlgorithm();

    // Получаем строку
    std::string fContent = getString();
    // Применяем шифрование
    std::string enc = applyEncryption(fContent);
    // Сохраняем строку
    saveString(fContent);

    // Подчищаем следы работы алгоритма
    wipeSpace();
}
```

С помощью паттерна «Шаблонный метод» мы можем использовать алгоритм, представленный в функции encrypt(), чтобы работать со строками, полученными из разных источников — с клавиатуры, прочитанные с диска, полученные по сети. При этом сама структура алгоритма и неизменные шаги (установка начальных параметров, подчистка следов работы, и при желании применение шифрования) остаются неизменными. Это позволяет нам:

1. Повторно использовать код, который не изменяется для различных подклассов;
2. Определить общее поведение семейства подклассов, используя единожды определённый код;
3. Разграничить права доступа — при реализации изменяемых шагов алгоритма мы будем использовать закрытые виртуальные функции. Это гарантирует, что такие операции будут вызываться только в качестве шагов модифицируемого алгоритма (или, скорее, не будут вызываться производными классами в неподходящих для этого местах).

Итак, дополним класс Crypt необходимыми членами:

```
private:
    void setupRnd() {
        // Некая инициализация алгоритма случайных чисел
        std::cout << "setup rnd\n";
    };
    void setupAlgorithm() {
        // Начальные установки алгоритма шифрования
        std::cout << "setup algorithm\n";
    };
    void wipeSpace() {
        // Удаление следов работы
        std::cout << "wipe\n";
    };

    virtual std::string applyEncryption(const std::string& content) {
        // Шифрование
        std::string result = someStrongEncryption(content);
        return result;
    }
    virtual std::string getString() = 0;
    virtual void saveString(const std::string& content) = 0;
```

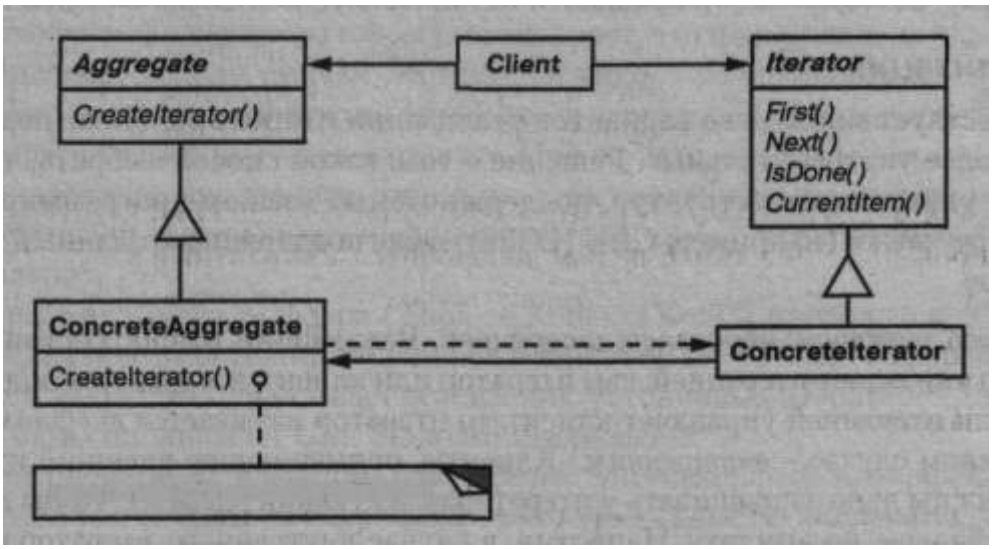
Обратите внимание, что функции закрытые. Это намеренное ограничение на их вызов, которое не мешает переопределить их в производном классе.
И, собственно, производный класс — шифрующий файл на диске:

```
class DiskFileCrypt : public Crypt {
public:
    DiskFileCrypt(const std::string& fName)
        : fileName(fName) {};
private:
    std::string fileName;

    virtual std::string getString() {
        std::cout << "get disk file named \"" << fileName << "\"\n";
        // Прочитать файл с диска и вернуть содержимое
        return fileContent;
    }
    virtual void saveString(const std::string& content) {
        std::cout << "save disk file named \"" << fileName << "\"\n";
        // Записать файл на диск
```

```
    }
};
```

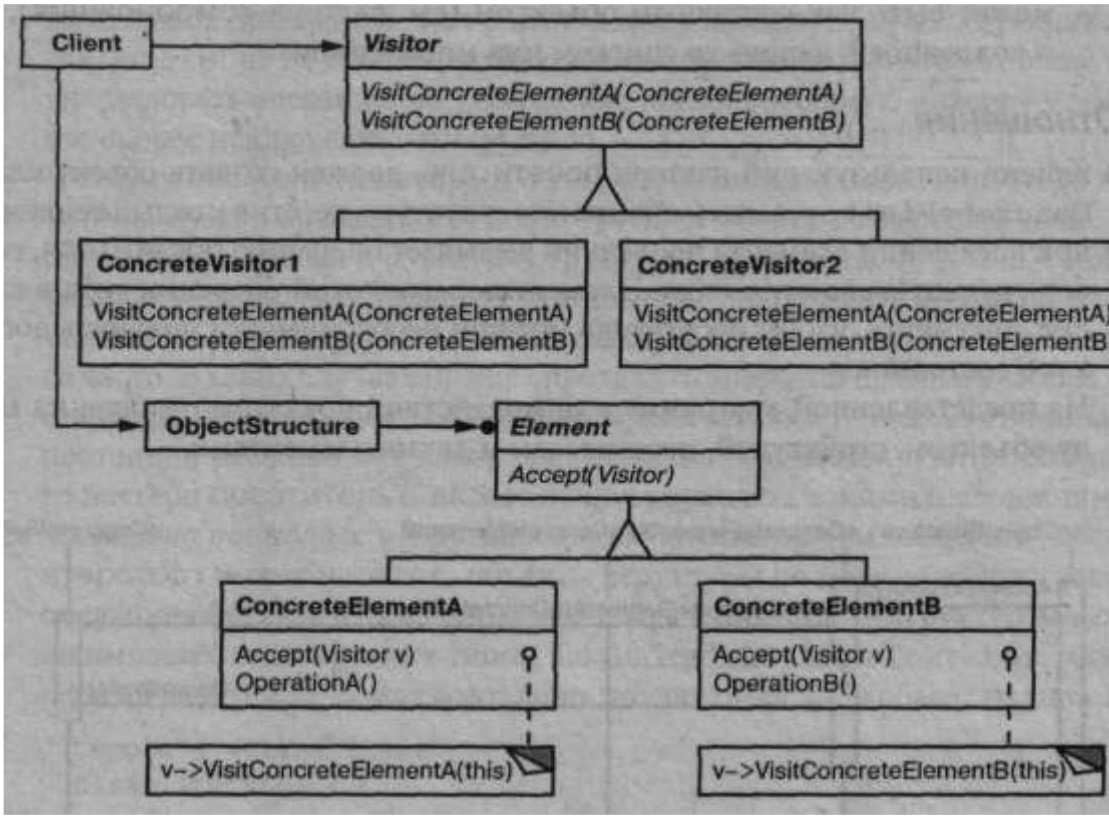
https://habrahabr.ru/post/277295/

7. Iterator
   - ❖ Название и классификация паттерна
     - ➢ Итератор - паттерн поведения объектов.
   - ❖ Назначение
     - ➢ Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления
   - ❖ Применимость
     - ➢ Используйте паттерн итератор:
       - ■ для доступа к содержимому агрегированных объектов без раскрытия их внутреннего представления;
       - ■ для поддержки нескольких активных обходов одного и того же агрегированного объекта;
       - ■ для предоставления единообразного интерфейса с целью обхода различных агрегированных структур (то есть для поддержки полиморфной итерации).
   - ❖ Структура
     - ➢ 

   - ❖ Участники
     - ➢ Iterator - итератор:
       - ■ определяет интерфейс для доступа и обхода элементов;
     - ➢ ConcreteIterator - конкретный итератор:
       - ■ реализует интерфейс класса Iterator;
       - ■ следит за текущей позицией при обходе агрегата;

- ➢ Aggregate - агрегат:
  - определяет интерфейс для создания объекта-итератора;
- ➢ ConcreteAggregate - конкретный агрегат:
  - реализует интерфейс создания итератора и возвращает экземпляр подходящего класса ConcreteIterator.

8. Visitor
   - ❖ Название и классификация паттерна
     - ➢ Посетитель - паттерн поведения объектов.
   - ❖ Назначение
     - ➢ Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.
   - ❖ Применимость
     - ➢ Используйте паттерн посетитель, когда:
       - в структуре присутствуют объекты многих классов с различными интерфейсами и вы хотите выполнять над ними операции, зависящие от конкретных классов;
       - над объектами, входящими в состав структуры, надо выполнять разнообразные, не связанные между собой операции и вы не хотите «засорять» классы такими операциями. Посетитель позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн посетитель позволит в каждое приложение включить только относящиеся к нему операции;
       - классы, устанавливающие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов, представленных в структуре, нужно будет переопределить интерфейсы всех посетителей, а это может вызвать затруднения. Поэтому если классы меняются достаточно часто, то, вероятно, лучше определить операции прямо в них.
   - ❖ Структура

➢ 

❖ Участники

➢ Visitor (NodeVisitor) - посетитель:

- объявляет операцию Visit для каждого класса ConcreteElement в структуре объектов. Имя и сигнатура этой операции идентифицируют класс, который посылает посетителю запрос Visit. Это позволяет посетителю определить, элемент какого конкретного класса он посещает. Владея такой информацией, посетитель может обращаться к элементу напрямую через его интерфейс;

➢ Concrete Visitor (TypeCheckingVisitor) - конкретный посетитель:

- реализует все операции, объявленные в классе Visitor. Каждая операция реализует фрагмент алгоритма, определенного для класса соответствующего объекта в структуре. Класс ConcreteVisitor предоставляет контекст для этого алгоритма и сохраняет его локальное состояние. Часто в этом состоянии аккумулируются результаты, полученные в процессе обхода структуры;

➢ Element (Node) - элемент:

- определяет операцию Accept, которая принимает посетителя в качестве аргумента;

- ➢ ConcreteElement (AssignmentNode, VariableRefNode) - конкретный элемент:
  - реализует операцию Accept, принимающую посетителя как аргумент;
- ➢ ObjectStructure (Program) - структура объектов:
  - может перечислить свои элементы;
  - может предоставить посетителю высокоуровневый интерфейс для посещения своих элементов;
  - может быть как составным объектом (см. паттерн компоновщик), так и коллекцией, например списком или множеством.

ЕСЛИ ЧТО ТО НЕПОНЯТНО ОБРАЩАЙТЕСЬ К ВЫШЕУПОМЯНУТОЙ КНИГЕ !

17. Консольные приложения vs Оконные Приложения

Консольные приложения

- Программа запрашивает ввод, обрабатывает его, запрашивает новый ввод и т.д.

```
#include <iostream>
int main()
{
	std::string cmd;
	while (true) {
		std::cin >> cmd;
		if (!processCmd(cmd)) {
			break;
		}
	}
	return 0;
}
```

- Синхронность – пользовательский ввод «вписан» в логику программы.
- Программа сама решает, когда запрашивать ввод.
- Простота программирования.
- «Захватывает» ввод.
- Сложность использования.

ОКОННЫЕ ПРИЛОЖЕНИЯ

- Приложение асинхронно реагирует на пользовательский ввод.
- Содержит более удобные элементы управления нежели консольное приложение.
- Ввод принадлежит операционной системе.
- Сложнее в разработке (ГОРАЗДО)

Консольные приложения

Консольное приложение - это программа, которая работает с**командной строкой**. То есть это обычное окно, где пользователь может ввести какую-то команду и получить результат. Здесь нет никаких кнопочек и прочих прелестей Windows.
Пример консольного приложения - это командный интерпретатор, который есть в любой операционной системе. В Windows 95/98/ME - это программа **command.com** (впрочем, он есть и в более поздних версиях Windows). В Windows 2000 и выше - это программа **cmd.exe**.

Оконные приложения

Оконное приложение - это привычная всем программа Windows. То есть это окошко с разными кнопочками и полями для ввода-вывода данных. На сегодняшний день это, пожалуй, самый распространённый вид программ. Именно оконные приложения создают большинство программистов.

18. Механизм сообщений и карта сообщений

СООБЩЕНИЯ

- Операционная система владеет вводом. Она должна
- каким-то образом уведомить приложение о нём (щелчок мышкой, клавиша клавиатуры).
- Сообщение – один из способов решения этой задачи.
- Цикл сообщений – код в программе, обрабатывающий сообщения от операционной системы.

ОБРАБОТКА СООБЩЕНИЙ

• Сообщения:

• Инициализация, деинициализация (WM_CREATE,

WM_DESTROY и тд)

• Ввод (WP_KEYUP, WM_KEYDOWN и тд)

• Отображение (WM_PAINT)

• Прочие (WM_TIMER и тд)

MFC

• Попытка «натянуть» ООП обертку над WinAPI

• Генерация большей части кода

• Программист только дописывает логику

• Карта сообщений

КАРТА СООБЩЕНИЙ

BEGIN_MESSAGE_MAP(CMyDoc, CDocument)

ON_COMMAND(ID_MYCMD, &CMyDoc::OnMyCommand)

END_MESSAGE_MAP()

еще бы понимать что это значит T_T

1. Механизм сообщений

Windows является операционной системой, управляемой событиями. Почти все главные и второстепенные события в среде Windows принимают форму сообщений и обрабатываются ОС и приложениями.

    1. Формат сообщений

Само по себе сообщение представляет собой структуру данных, описанную в файле WinUser.h:

typedef struct tagMSG

{

HWND hwnd; //идентификатор получателя

UINT message; //уникальный для Windows код сообщения

WPARAM wParam; //содержимое, зависит от конкретного сообщения

LPARAM lParam; //содержимое, зависит от конкретного сообщения

DWORD time;// время отправления сообщения

POINT pt; //позиция курсора в экранных координатах, когда сообщение было отправлено.

} MSG;

    1. Обработка сообщений

Все события, происходящие в системе, обретают форму сообщений. Например, когда вы нажимаете и затем отпускаете клавишу, формируется прерывание, которое обрабатывается драйвером. Он вызывает процедуру в модуле user.exe, которая формирует сообщение, содержащее информацию о событии. Аналогично сообщения создаются при перемещении мыши или в том случае, когда вы нажимаете кнопки на корпусе мыши. Можно сказать, что драйверы устройств ввода/вывода транслируют аппаратные прерывания в сообщения.

Следует отметить, что в Windows используется **многоуровневая система** сообщений.

Сообщения **низкого** уровня вырабатываются, когда вы перемещаете мышь или нажимаете клавиши на корпусе мыши или на клавиатуре. В эти сообщения входит информация **о текущих координатах курсора** мыши или кодах нажатых клавиш. Обычно приложения редко анализируют сообщения низкого уровня. Все эти сообщения передаются операционной системе Windows, которая на их основе формирует сообщения более **высокого** уровня. Когда вы нажимаете кнопку в диалоговом окне приложения Windows, приложение получает сообщение о том, что **нажата кнопка**. Вам не надо постоянно анализировать координаты курсора мыши – Windows сама вырабатывает для вас соответствующее сообщение высокого уровня.

Куда направляются сообщения, созданные драйверами?

Прежде всего, сообщения попадают в **системную очередь** сообщений Windows, реализованную в модуле user.exe. Системная очередь сообщений **одна**. Далее из нее сообщения распределяются в **очереди сообщений приложений**. Для каждого приложения создается своя очередь сообщений.

Очередь сообщения приложений может пополняться не только из системной очереди. Любое приложение может послать сообщение любому другому сообщению, в том числе и само себе.

Основная работа, которую должно выполнять приложение, заключается в **обслуживании собственной очереди** сообщений. Обычно приложение в цикле опрашивает свою очередь сообщений. Обнаружив сообщение, приложение с помощью специальной функции из программного интерфейса Windows распределяет его нужной функции окна, которая и выполняет обработку сообщения.

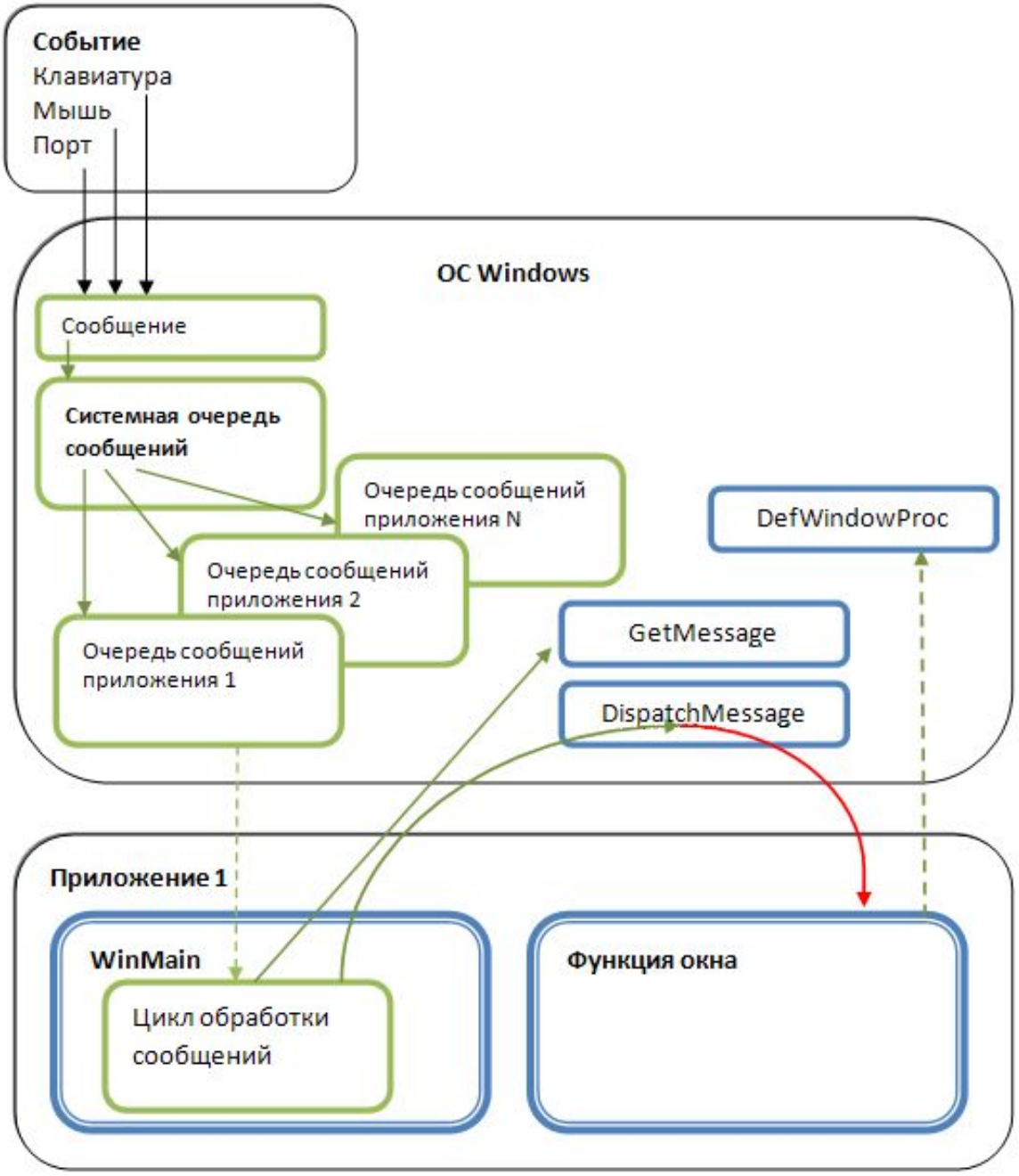


Рис.3 Обработка сообщений

Так как Windows многозадачная операционная система в ней разработан механизм использования несколькими параллельно работающими приложениями таких ресурсов, как мышь и клавиатура. Так как все сообщения, создаваемые драйверами мыши и клавиатуры, попадают в одну системную очередь сообщений, должен существовать способ распределения этих сообщений между различными приложениями.

Хотя одновременно может быть запущено много приложений, активным является **только одно**. Принято говорить, что это приложение имеет **фокуса**

ввода. Все сообщения от клавиатуры и мыши идут в очередь приложения, имеющего фокус ввода.

Перемещать фокус ввода от одного окна к другому можно, нажимая определенные клавиши, или мышью.

Сообщения от драйвера мыши всегда передаются функции того окна, над которым находится курсор мыши. При необходимости приложение может выполнить операцию захвата (capturing) мыши. В этом случае все сообщения от мыши будут поступать в очередь приложения, захватившего мышь, вне зависимости от положения курсора мыши.

1. Цикл обработки сообщений

Простейший цикл обработки сообщений состоит из вызовов двух функций – GetMessage и DispatchMessage.

BOOL WINAPI GetMessage(

LPMSG lpMsg,// Указатель на структуру MSG, которая получает информацию об очередном сообщении из очереди сообщений.

HWND hWnd,// дескриптор окна, чьи сообщения должны быть извлечены.

UINT wMsgFilterMin, //наименьший номер сообщения

UINT wMsgFilterMax//наибольший номер сообщения

);

Если wMsgFilterMin и wMsgFilterMax равны нулю, GetMessage возвращает все доступные сообщения (то есть, фильтрация сообщений не выполняется).

Функция GetMessage предназначена для **выборки сообщения из очереди** приложения. Сообщение выбирается из очереди и записывается в область данных, принадлежащую приложению. Функция возвращает ненулевое значение, если очередное сообщение не WM_QUIT, и ноль в случае WM_QUIT.

Функция DispatchMessage предназначена для **распределения** выбранного из очереди сообщения **нужной функции окна**. Так как приложение обычно создает много окон, и эти окна используют различные функции окна, необходимо распределить сообщение именно тому окну, для которого оно предназначено. Windows оказывает приложению существенную помощь в

решении этой задачи – для распределения приложению достаточно вызвать функцию DispatchMessage.

Вот как выглядит простейший вариант цикла обработки сообщений:

```
MSG msg;

while (GetMessage(&msg, NULL, 0, 0))

{

DispatchMessage(&msg);

}

return (int) msg.wParam;
```

Завершение цикла обработки сообщений происходит при выборке из очереди сообщения WM_QUIT, в ответ на которое функция GetMessage возвращает нулевое значение.

Коды сообщений определены в файле WinUser.h, включаемом в исходные тексты любых приложений Windows.

1. Функции работы с сообщениями

Таблица 2

Функции для работы с сообщениями

	Имя функции	Назначение
1	GetMessage	Извлекает сообщение из очереди сообщений приложения.
2	DispatchMessage	Передает сообщение функции окна.
3	PostMessage	Посылает сообщение в очередь.
4	SendMessage	Посылает сообщение в функцию окна, возврат из функции происходит после обработки этого сообщения.
5	PostQuitMessage	Посылает сообщение о завершении (WM_QUIT) в очередь сообщений приложения.

6	TranslateMessage	Переводит сообщения виртуальных клавиш в символьные.

1. Классификация сообщений по функциональным признакам

Таблица 3

Системные сообщения

1	WM_SYSCOMAND	Выбран пункт меню System
2	WM_SYSKEYDOWN	Нажата системная клавиша
	и др.	

Таблица 4

Сообщения от мыши

1	WM_NCHITTEST	Это сообщение генерируется драйвером мыши при любых перемещениях мыши.
2	WM_MOUSEMOVE	Перемещение курсора мыши во внутренней области окна
3	WM_NCMOUSEMOVE	Перемещение курсора мыши во внешней области окна
4	WM_MOUSEACTIVATE	Активизация неактивного окна при помощи мыши.
От левой клавиши мыши		
1	WM_LBUTTONDBLCLK	Двойной щелчок левой клавишей мыши во внутренней области окна
2	WM_LBUTTONDOWN	Нажата левая клавиша мыши во внутренней области окна
3	WM_LBUTTONUP	Отпущена левая клавиша мыши во внутренней области окна

4	WM_NCLBUTTONDBLCLK	Двойной щелчок левой клавишей мыши во внешней области окна
5	WM_NCLBUTTONDOWN	Нажата левая клавиша мыши во внешней области окна
6	WM_NCLBUTTONUP	Отпущена левая клавиша мыши во внешней области окна
Аналогичные сообщения приходят от средней и правой клавиш мыши с префиксами WM_RBUTTON и WM_MBUTTON		

Пакет **Microsoft Foundation Classes** (MFC) — библиотека на языке C++, разработанная Microsoft и призванная облегчить разработку GUI-приложений для Microsoft Windows путём использования богатого набора библиотечных классов.

Принцип действия

Библиотека MFC, как и её основной конкурент, Borland VCL, облегчает работу с GUI путём создания *каркаса приложения* — «скелетной» программы, автоматически создаваемой по заданному макету интерфейса и полностью берущей на себя рутинные действия по его обслуживанию (отработка оконных событий, пересылка данных между внутренними буферами элементов и переменными программы и т. п.). Программисту после генерации каркаса приложения необходимо только вписать код в места, где требуются специальные действия. Каркас должен иметь вполне определенную структуру, поэтому для его генерации и изменения в Visual C++ предусмотрены мастера.

Кроме того, MFC предоставляет объектно-ориентированный слой обёрток (англ. *wrappers*) над множеством функций Windows API, делающий несколько более удобной работу с ними. Этот слой представляет множество встроенных в систему объектов (окна, виджеты, файлы и т. п.) в виде классов и опять же берёт на себя рутинные действия вроде закрытия дескрипторов и выделения/освобождения памяти.

Добавление кода в каркас приложения

Добавление кода приложения к каркасу реализовано двумя способами. Первый использует механизм наследования: основные программные структуры каркаса представлены в виде классов, наследуемых от библиотечных. В этих классах предусмотрено множество виртуальных функций, вызываемых в определенные моменты работы программы. Путём доопределения (в большинстве случаев необходимо вызвать функцию базового класса) этих функций программист может добавлять выполнение в эти моменты своего кода.

Второй способ используется для добавления обработчиков оконных событий. Мастер создает внутри каркасов классов, связанных с окнами, специальные массивы — *карты (оконных) сообщений* (англ. *message map*), содержащие пары «ID сообщения — указатель на обработчик». При добавлении/удалении обработчика мастер вносит изменения в соответствующую карту сообщений.

Карта сообщений

Библиотека классов MFC предоставляет альтернативу многочисленным операторам switch, используемым в традиционных Windows-программах для обработки сообщений, посылаемых окну. Взаимосвязь сообщений и их обработчиков может быть определена таким образом, что при поступлении сообщения в оконную процедуру соответствующий обработчик вызывается автоматически. Реализация такой взаимосвязи основана на понятии Карты (или Таблицы) сообщений (message map).

Карта сообщений представляет, собой механизм пересылки сообщений и команд Windows в окна, документы, представления и другие объекты приложения, реализованного на базе MFC. Такие карты преобразуют сообщения Windows, извещения элементов управления, а также команды меню, кнопок панелей инструментов, командных клавиш клавиатуры в функции соответствующих классов, которые их обрабатывают. Эта особенность карты сообщений реализована по аналогии с виртуальными функциями C++, но имеет дополнительные преимущества, не доступные для них. Каждый класс, который может получить сообщение, должен иметь свою карту сообщений для того, чтобы соответствующим образом обрабатывать сообщения. При этом следует иметь в виду, что карта сообщений должна определяться вне какой-либо функции или объявления класса. Она также не может размещаться внутри C-блока.

Для определения карты сообщений используются три макроса: BEGIN_ MESSAGE_MAP, END_MESSAGE_MAP и DECLARE_MESSAGE_MAP.

Макрос DECLARE_MESSAGE_MAP располагается в конце объявления класса, использующего карту сообщений:

```
class CTheClass : public CBaseClass

{

 //объявления компонентов класса

 protected:

//{{AFX_MSG_MAP(TheClass)

afx_msg void OnPaint();

//}}AFX_MSG_MAP,

DECLARE_MESSAGE_MAP()

};
```

Он используется для объявления трех специальных компонентов класса, о которых нужно знать, но которые практически никогда непосредственно не используются.

Структура карты сообщений достаточно проста и представляет собой набор макросов, заключенных в специальные "операторные скобки":

```
BEGIN_MESSAGE_MAP(CTheClass, CBaseClass))

// (AFX_MSG_MAP!CTheClass)

ON_WM_CREATE ()

ON_WM_DESTROY()

ON_COMMAND(ID_CHAR_BOLD, OnCharBold)

ON_UPDATE_COMMAND_UI(ID_CHAR__BOLD, OnUpdateCharBold)

// {{AFX_MSG_MAP

ON NOTIFY(FN_GETFORMAT, ID_VIEW_TORMATBAR, OnGetCharFormat)
ON_NOTIFY(FN_SETFORMAT, ID_VIEW_FORMATBAR, OnSetCharFormat)

 END_MESSAGE_MAP()
```

Примечание

При использовании специальных средств автоматической генерации программного кода — AppWizard и ClassWizard — все три описанных макроса, а также специальные "операторные скобки" для ClassWizard (//{{AFX_MSG_ MAP(CTheClass) и //{(AFX_MSG_MAP) формируются автоматически при создании оконного класса.

Как видите, начинается карта с выполнения макроса:

BEGIN_MESSAGE_MAP(CTheClass, CBaseClass)

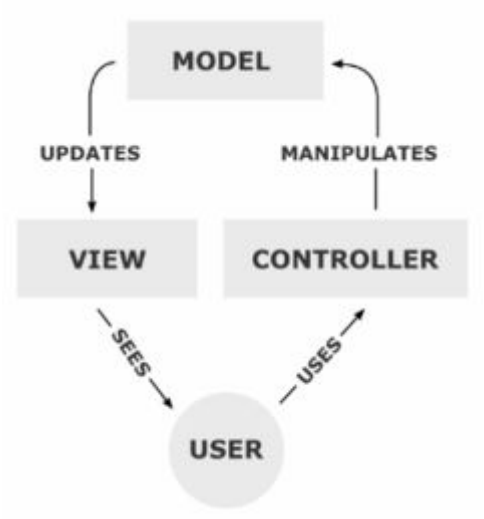
который имеет следующие параметры: CTheClass — задает имя класса, владельца карты сообщений; CBaseClass — определяет имя базового класса.

Заканчивается карта сообщений вызовом макроса END_MESSAGE_MAP, имеющего еще более простой формат:

END_MESSAGE_MAP()

Между этими двумя вызовами располагаются специальные макросы, называемые компонентами карты сообщений. Вот они-то, собственно, и позволяют сопоставить сообщение с конкретным обработчиком.

19. MVC



ПРИМЕР – ASP.NET MFC

• [HttpGet]

• public ActionResult Register() {

• var newUser = new User();

• return View(newUser);

• }

• [HttpPost]

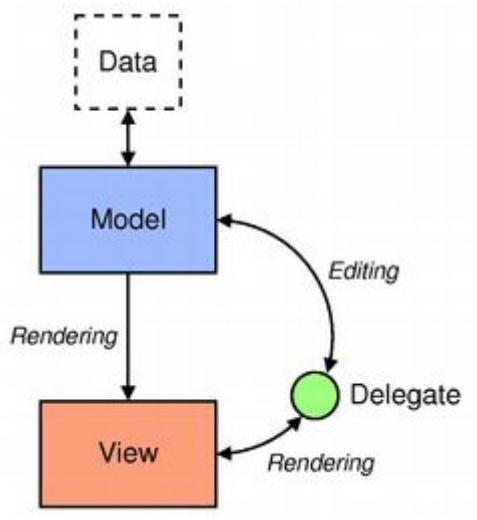
• public ActionResult Register(User user) {

• // Logic here

• }

МОДЕЛЬ/ПРЕДСТАВЛЕНИЕ

• Если объединить представление и контроллер в MVC.



ПРИМЕР - QT

• Модель осуществляет соединение с источником данных, предоставляя необходимый интерфейс другим компонентам архитектуры. Характер связи зависит от типа источника данных и способа реализации модели.

• Из модели представление получает модельные индексы, являющиеся ссылками на элементы данных. Передавая модельные индексы модели, представление может получить элементы данных из источника данных.

• В стандартных представлениях делегат отображает элементы данных. Если элемент редактируемый, делегат связывается с моделью непосредственно используя модельные индексы.

Model-View-Controller (**MVC**, «Модель-Представление-Контроллер», «Модель-Вид-Контроллер») — схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо[1].

- ***Модель*** (*Model*) предоставляет данные и реагирует на команды контроллера, изменяя свое состояние[1].
- ***Представление*** (*View*) отвечает за отображение данных модели пользователю, реагируя на изменения модели[1].
- ***Контроллер*** (*Controller*) интерпретирует действия пользователя, оповещая модель о необходимости изменений[1].

Основная цель применения этой концепции состоит в отделении бизнес-логики (*модели*) от её визуализации (*представления*, *вида*). За счёт такого разделения повышается возможность повторного использования кода. Наиболее полезно применение данной концепции в тех случаях, когда пользователь должен видеть те же самые данные одновременно в различных контекстах и/или с различных точек зрения. В частности, выполняются следующие задачи:

1. К одной *модели* можно присоединить несколько *видов*, при этом не затрагивая реализацию *модели*. Например, некоторые данные могут быть одновременно представлены в виде электронной таблицы, гистограммы и круговой диаграммы;
2. Не затрагивая реализацию *видов*, можно изменить реакции на действия пользователя (нажатие мышью на кнопке, ввод данных) — для этого достаточно использовать другой *контроллер*;
3. Ряд разработчиков специализируется только в одной из областей: либо разрабатывают графический интерфейс, либо разрабатывают бизнес-логику. Поэтому возможно добиться того, что программисты, занимающиеся разработкой бизнес-логики (*модели*), вообще не будут осведомлены о том, какое *представление* будет использоваться.

Концепция MVC позволяет разделить модель, представление и контроллер на три отдельных компонента:

Модель

Модель предоставляет данные и методы работы с ними: запросы в базу данных, проверка на корректность. Модель не зависит от представления — не знает как данные визуализировать — и контроллера — не имеет точек взаимодействия с пользователем —, просто предоставляя доступ к данным и управлению ими.

Модель строится таким образом, чтобы отвечать на запросы, изменяя своё состояние, при этом может быть встроено уведомление «наблюдателей».

Модель, за счёт независимости от визуального представления, может иметь несколько различных представлений для одной «модели».

Представление

Представление отвечает за получение необходимых данных из модели и отправляет их пользователю. Представление не обрабатывает введённые данные пользователя.

Представление может влиять на состояние модели сообщая модели об этом.

Контроллер

Контроллер обеспечивает «связи» между пользователем и системой. Контролирует и направляет данные от пользователя к системе и наоборот. Использует модель и представление для реализации необходимого действия.

Условно-обязательные модификации

Для реализации схемы «Model-View-Controller» используется достаточно большое число шаблонов проектирования (в зависимости от сложности архитектурного решения), основные из которых — «наблюдатель», «стратегия», «компоновщик»[1]:

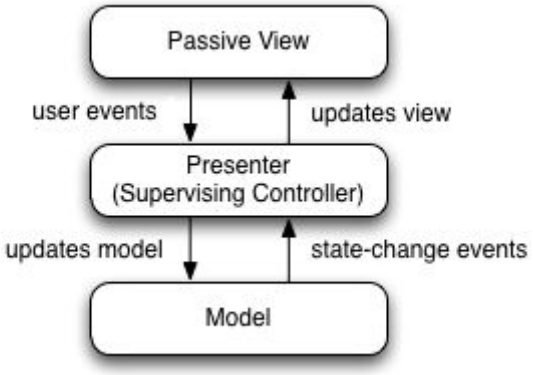
Наиболее типичная реализация — в которой вид отделён от модели путём установления между ними протокола взаимодействия, использующего «аппарат событий» (обозначение *«событиями»* определённых ситуаций, возникающих в ходе выполнения программы, — и рассылка уведомлений о них всем тем, кто подписался на получение): при каждом особом изменении внутренних данных в модели (обозначенном как «событие»), она оповещает о нём те зависящие от неё представления, которые подписаны на получение такого оповещения — и представление обновляется. Так используется шаблон «наблюдатель»;

При обработке реакции пользователя — представление выбирает, в зависимости от реакции, нужный *контроллер*, который обеспечит ту или иную

связь с моделью. Для этого используется шаблон «стратегия», или вместо этого может быть модификация с использованием шаблона «команда»;

Для возможности однотипного обращения с подобъектами сложно-составного иерархического вида — может использоваться шаблон «компоновщик». Кроме того, могут использоваться и другие шаблоны проектирования — например, «фабричный метод», который позволит задать по умолчанию тип контроллера для соответствующего вида.

20. MVP



Model-View-Presenter (MVP) — шаблон проектирования, производный от MVC, который используется в основном для построения пользовательского интерфейса.

Элемент Presenter в данном шаблоне берёт на себя функциональность посредника (аналогично контроллеру в MVC) и отвечает за управление событиями пользовательского интерфейса (например, использование мыши) так же, как в других шаблонах обычно отвечает представление.

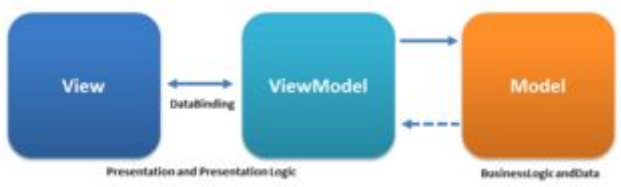
MVP — шаблон проектирования пользовательского интерфейса, который был разработан для облегчения автоматического модульного тестирования и улучшения разделения ответственности в презентационной логике (отделения логики от отображения):

- *Модель* (англ. *Model*) — хранит в себе всю бизнес-логику, при необходимости получает данные из хранилища.
- *Представление* (англ. *View*) — реализует *отображение* данных (из Модели), обращается к Presenter за обновлениями.
- *Presenter* — представитель; реализует взаимодействие между моделью и представлением.

Обычно экземпляр Представления создаёт экземпляр Presenter’а, передавая ему ссылку на себя. При этом Presenter работает с Представлением в абстрактном виде, через его интерфейс. Когда вызывается событие

Представления, оно вызывает конкретный метод Presenter’a, не имеющего ни параметров, ни возвращаемого значения. Presenter получает необходимые для работы метода данные о состоянии пользовательского интерфейса через интерфейс Представления и через него же передаёт в Представление данные из Модели и другие результаты своей работы.

21. MVVM



Шаблон **Model-View-ViewModel** (MVVM) — применяется при проектировании архитектуры приложения.

Назначение

MVVM используется для разделения модели и её представления, что необходимо для изменения их отдельно друг от друга. Например, разработчик задает логику работы с данными, а дизайнер соответственно работает с пользовательским интерфейсом.

Использование

MVVM удобно использовать вместо классического MVC и ему подобных в тех случаях, когда в платформе, на которой ведётся разработка, присутствует «связывание данных». В шаблонах проектирования MVC/MVP изменения в пользовательском интерфейсе не влияют непосредственно на Модель, а предварительно идут через Контроллер (*Controller*) или Presenter. В таких технологиях как WPF и Silverlight есть концепция «связывания данных», позволяющая связывать данные с визуальными элементами в обе стороны. Следовательно, при использовании этого приема применение модели MVC становится крайне неудобным из-за того, что привязка данных к представлению напрямую не укладывается в концепцию MVC/MVP.

Описание

Шаблон MVVM делится на три части:

- *Модель* (англ. *Model*), так же, как в классической MVC, Модель представляет собой бизнес логику и фундаментальные данные, необходимые для работы приложения.
- *Представление* (англ. *View*) — это графический интерфейс, то есть окно, кнопки и т. п. Представление является подписчиком на событие изменения значений свойств или команд, предоставляемых Моделью Представления. В случае, если в Модели Представления изменилось какое-либо свойство, то она оповещает всех подписчиков об этом, и Представление, в свою очередь, запрашивает обновленное значение свойства из Модели Представления. В случае, если пользователь воздействует на какой-либо элемент интерфейса, Представление вызывает соответствующую команду, предоставленную Моделью Представления.
- *Модель Представления* (англ. *ViewModel*) является, с одной стороны, абстракцией Представления, а с другой, предоставляет обёртку данных из Модели, которые подлежат связыванию. То есть, она содержит Модель, которая преобразована к Представлению, а также содержит в себе команды, которыми может пользоваться Представление, чтобы влиять на Модель.

22. Пользовательские приложения

Пользовательские приложения

- «Программирование для пользователя»
- Типичные задачи
- «Фронт» взаимодействия с заказчиком
- Постоянные мелкие правки
- Главное – возможность быстрой модификации
- Понятный, читаемый код
- Паттерны проектирования

хз что тут еще добавить=(

23. Библиотеки

- «Программирование для программиста»
- Максимально переиспользуемый код
- Максимально «настраиваемый» код
- BlackBox
- Главное - возможность оперативной поддержки и bugfix
- Актуальная документация!!!

- Все, что может быть сделано в библиотеке - должно быть сделано в библиотеке

короче мне нечего добавить=(

24. Технологические библиотеки

Технологии

- «Программирование для программиста»
- Решение сложной «наукоемкой» задачи
- Поддержка, развитие и улучшение решения
- Главное - качество и скорость работы
- Одна и та же кодовая база, развиваемая годами
- «Внутренняя документация»
- Нетривиальные решения

25. Тесты

если что см. конспект Даши

- Unit-тесты - Проверка мелких механизмов (классы, функции)
- Автотесты - проверка сценариев
- Интерактивный контроль

26. Метрики и логи

А зачем они вообще нужны?

- Понять, что есть проблема до «бизнеса»
- Понять, что именно произошло
- Предвосхитить проблему (метрики)

https://habrahabr.ru/company/intel/blog/106082/

Ме́трика програ́ммного обеспе́чения (англ. *software metric*) — мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций.

Поскольку количественные методы хорошо зарекомендовали себя в других областях, многие теоретики и практики информатики пытались перенести

данный подход и в разработку программного обеспечения. Как сказал Том ДеМарко, «вы не можете контролировать то, что не можете измерить».

Метрики

Набор используемых метрик включает:

- порядок роста (имеется в виду анализ алгоритмов в терминах асимптотического анализа и О-нотации),
- количество строк кода,
- цикломатическая сложность,
- анализ функциональных точек,
- количество ошибок на 1000 строк кода,
- степень покрытия кода тестированием,
- покрытие требований,
- количество классов и интерфейсов,
- метрики программного пакета от Роберта Сесиль Мартина,
- связность.

Критика

Потенциальные недостатки подхода, на которые нацелена критика:

- Неэтичность: Утверждается, что неэтично судить о производительности программиста по метрикам, введенным для оценки эффективности программного кода. Такие известные метрики, как количество строк кода и цикломатическая сложность, часто дают поверхностное представление об "удачности" выбора того или иного подхода при решении поставленных задач, однако, нередко они рассматриваются, как инструмент оценки качества работы разработчика. Такой подход достаточно часто приводит к обратному эффекту, приводя к появлению в коде более длинных конструкций и избыточных необязательных методов.
- Замещение «управления людьми» на «управление цифрами», которое не учитывает опыт сотрудников и их другие качества
- Искажение: Процесс измерения может быть искажён за счёт того, что сотрудники знают об измеряемых показателях и стремятся оптимизировать эти показатели, а не свою работу. Например, если количество строк исходного кода является важным показателем, то программисты будут стремиться писать как можно больше строк и не будут использовать способы упрощения кода, сокращающие количество строк.
- Неточность: Нет метрик, которые были бы одновременно и значимы и достаточно точны. Количество строк кода — это просто количество

строк, этот показатель не даёт представление о сложности решаемой проблемы. Анализ функциональных точек был разработан с целью лучшего измерения сложности кода и спецификации, но он использует личные оценки измеряющего, поэтому разные люди получат разные результаты.

27. Code Review

- А зачем?
- Доверие - корень всех бед
- Я доверяю коллегам, как специалистам, но не доверяю как людям
- Проверка кода «незамыленным взглядом»
- Поиск недостаточно документированных мест
- Выявление опискок/опечаток/ошибок
- Неправильный способ решения задачи?

28. Definition of Done

- Набор правил по которым мы закрываем задачи
- Суть - не забыть сделать какой-то шаг

29. Преждевременная оптимизация

- Преждевременная оптимизация - корень всех бед!
- Может никогда не закончится
- То ли место мы оптимизируем
- Портит архитектуру

http://elizarov.livejournal.com/17873.html

30. Алгоритмы и оптимизация

- Знай тот код, которым ты пользуешься!
- Пример: C++ map - красно-черное дерево. Вставка O(log n), а не O(1)
- Пример python - named tuple вычисляется через eval
- Пример %Ваш язык программирования% - тоже имеет особенности
- Знай данные с которыми ты работаешь
- QSort vs Merge Sort Linear Search vs BinarySearch
- …

Что может тормозить?

- Лишние вычисления
- Повторяющиеся вычисления
- IO
- …

Лишние вычисления

a = «»

for i in range(100):

a += «a»

Лишние вычисления

- Лечатся алгоритмом
- Ранние отсечения веток кода

Повторяющиеся вычисления

def foo(var): // To long evaluation

a = foo(1)

…

b = foo(1)

Повторяющиеся вычисления

Мемоизация (кеширование) Особенно на чистых функциях Предвычисления

Предвычисления

Lazy - при первом обращении Проблема многопоточности Load - при загрузке программы / модуля Вычисляем «на будущее» - а вдруг не надо? Compile-Time - вычисляем «до запуска» и зашиваем данные Не всегда возможно

IO

Уменьшить количество IO Лучше читать/писать последовательно …

31. Мемоизация и предвычисления

Повторяющиеся вычисления

- Мемоизация (кеширование)
  - Особенно на чистых функциях
- Предвычисления

Предвычисления

- Lazy - при первом обращении
  - Проблема многопоточности
- Load - при загрузке программы / модуля
  - Вычисляем «на будущее» - а вдруг не надо?
- Compile-Time - вычисляем «до запуска» и зашиваем данные
  - Не всегда возможно

IO

- Уменьшить количество IO
- Лучше читать/писать последовательно
- …

Мемоизация (запоминание, от англ. memoization (англ.) в программировании) — сохранение результатов выполнения функций для предотвращения повторных вычислений. Это один из способов оптимизации, применяемый для увеличения скорости выполнения компьютерных программ. Перед вызовом функции проверяется, вызывалась ли функция ранее:

- если не вызывалась, функция вызывается и результат её выполнения сохраняется;
- если вызывалась, используется сохранённый результат.

Мемоизация может использоваться не только для увеличения скорости работы программы. Например, она используется при простом взаимно-рекурсивном нисходящем синтаксическом разборе в обобщённом алгоритме нисходящего синтаксического анализа.

Несмотря на связь с кешированием, мемоизация является особым видом оптимизации, отличающимся от таких способов кеширования, как буферизация и подмена страниц.

32. Статические vs Динамические библиотеки

- Статические библиотеки - *.lib – целиком влинковываются в код Вашей программы.
- Динамические библиотеки - *.dll – поставляются вместе с программой. Их код не влинковывается в exe.

- Статические
    - Удобный механизм для абстракции и инкапсуляции
- Динамические
    - Удобный механизм для абстракции и инкапсуляции
    - Возможно повторное использование кода
    - Возможно разделение одной dll между несколькими процессами
    - Возможна замена dll без изменения основного exe*

С точки зрения ОС статических библиотек не существует.

Статические библиотеки — это набор уже скомпилированных подпрограмм или объектов, которые подключаются к исходной программе в виде объектных файлов. Этот набор выглядит как архив из нескольких объектных файлов, который умеет распаковывать gcc.

Например:

```
gcc -c file.c
gcc file.o -o file
```

(первая команда даст file.o, вторая — исполняемый файл). На втором этапе подключается библиотека libc. При этом gcc выбирает компилятор по расширению файла (например, gcc -c file.pas и gcc -c file.cpp вызовут разные компиляторы). Однако по умолчанию подключается только библиотека libc. Если сказать

```
gcc file.o -l stdc++ -o file
```

подключится ещё и библиотека stdc++.

Такое подключение называется статической линковкой. Она плоха тем, что для одной маленькой программы могут вызываться огромные библиотеки. Например,

```
g++ -c file.cpp
g++ file.o -o file
```

Библиотека stdc++ подключится сама. Она включает в себя все необходимые функции, что делает её автономной, но и сильно увеличивает размер программы.

Кроме статической, есть ещё динамическая линковка.

Динамическая библиотека языка — это библиотека, которая загружается в ОС по запросу работающей программы непосредственно в ходе её выполнения. Это делает линковщик, который собирает эту библиотеку из программных модулей, и загрузчик, который при запуске проверяет наличие этих модулей на компьютере. Такие библиотеки имеют другое расширение: .so в linux и .dll в win (в отличие от .a и .lib соответственно для статических).

Достоинства динамической линковки в том, что подключаемые библиотеки занимают меньше места, и, если мы делаем какие-нибудь обновления, то исполняемый файл перекомпилировать не нужно. (Недостатки в том, что для работы динамических библиотек необходимо их иметь. Кроме того, они могут отличаться версиями.)

По умолчанию все библиотеки .so и .dll собираются динамически. Если сказать

```
gcc file.o <библиотека> -o file -static
```

то данная библиотека при компиляции этого файла скомпилируется статически.

33. Статическое и динамическое связывание

Как использовать api из dll в программе?

Два способа:

- Статическое связывание
  - Статическое связывание означает, что dll загружается одновременно с основным кодом
  - Если dll не найдена – программа будет аварийно завершена

- Для статического связывания dll используются libфайлы (не путать с обычными статическими библиотеками) - библиотеки импортирования (import libraries).
- В них содержится не сам код библиотеки, а только ссылки на все функции, экспортируемые из файла dll, в котором все и хранится.
- Для статического связывания с dll надо указать линкеру где брать lib файл.
- Два способа:
  - #pragma comment(lib,"MYLIBRARY")
  - В настройках проекта в разделе linker
- Так же необходим заголовочный файл(-ы) (*.h), где определен публичный api библиотеки

➢ Динамическое связывание
- Dll загружается в адресное пространство процесса специальным вызовом api операционной системы:
  - LoadLibrary
- Затем требуется получить доступ к её api
  - GetProcAddress
- В конце работы библиотеку можно отгрузить
  - FreeLibrary

При статическом связывании *DLL* загружается сразу, как только начинает выполняться приложение, которое будет ее использовать. Это наиболее простой способ использования *DLL*. Вызов функций в приложении при этом почти не отличается от вызова любых других функций. Здесь время загрузки увеличивается и нельзя выполнять приложение, если нет соответствующего файла *DLL*. Кроме того, без *DLL* приложение не может выполняться и в сокращенном, также рабочем варианте. Недостатком статического связывания является и то, что *DLL* занимает память все время, в течение которого выполняется приложение, независимо от того, вызываются ли в данном сеансе работы с приложением какие-то функции библиотеки, или нет.

Статическое связывание подразумевает, что для *DLL* создан специальный файл описаний импортируемых функций (*import libraryfile*). Этот файл имеет расширение **.lib** и то же имя, что и соответствующая *DLL*, и должен быть связан с приложением на этапе компиляции.

При динамическом связывании *DLL* загружается только в тот момент, когда необходимо выполнить какую-то хранящуюся в ней функцию. Затем *DLL* можно выгрузить из памяти. Но при более эффективном использовании памяти вызов функций *DLL*существенно усложняется, и время вызова увеличивается.

34. DLL main

- У каждой dll есть особая функция – DllMain

BOOL WINAPI DllMain(_In_ HINSTANCE hinstDLL, _In_ DWORD fdwReason, _In_ LPVOID lpvReserved);

- Зачем она нужна?
    - Во первых, её можно не использовать. И тогда все ок.
- Windows вызывает dll main при каждом подключении и отключении процесса или потока к dll => она нужна, если нужно подготовить какие то данные перед использованием библиотеки
- Второй параметр – fdwReason:
    - DLL_PROCESS_ATTACH
    - DLL_PROCESS_DETTACH
    - DLL_THREAD_ATTACH
    - DLL_THREAD_DETTACH
- Надо пользоваться с осторожностью, потому что:
    - А) Порядок загрузки dll весьма сложен => ограничен набор функций которыми можно пользоваться в dllMain. Например, вызов LoadLibrary может привести к dealLock'у
    - Б) Вызов функции dllMain не гаранитрован. Например, TerminateThread, не вызывает dllMain

35. Расположение Dll в памяти

- Базовый адрес – куда, предположительно, должна быть загружена библиотека. Он «вшивается» в dll
- При загрузке:
    - Windows пытается загрузить dll по базовому адресу
    - Если по этому адресу недостаточно места, то система перемещает dll в памяти
    - Это приводит к тому, что все вызовы к dll должны быть скорректированы (rebase)

36.Dll Hell

- Хотели
    - DLL совместимы и взаимозаменяемы от версии к версии.
- Получили:

- Реализация механизма DLL такова, что несовместимость и невзаимозаменяемость становится скорее правилом, чем исключением, что приводит к большому количеству проблем.
- Отсутствие стандартов на имена, версии и положение DLL в файловой структуре приводит к тому, что несовместимые DLL легко замещают или отключают друг друга
- Отсутствие стандарта на процедуру установки приводит к тому, что установка новых программ приводит к замещению работающих DLL на несовместимые версии
- Отсутствие поддержки DLL со стороны компоновщиков и механизмов защиты приводит к тому, что несовместимые DLL могут иметь одинаковые имя и версию
- Отсутствуют стандартные инструменты идентификации и управления системой DLL пользователями и администраторами
- Использование отдельных DLL для обеспечения связи между задачами приводит к нестабильности сложных приложений

НА ЭТОМ МЕСТЕ ЗАКАНЧИВАЮТСЯ ПРЕЗЕНТАЦИИ, YANDEX И МОИ СИЛЫ

ПРОШУ НЕ ПИСАТЬ ТУПЫХ ВОПРОСОВ ТИПА “ЧТО ТАКОЕ ГАРБИШ КОЛЛЕКТОР И Т.Д. И Т.П.?”. НЕ ЗНАЕТЕ - ЗАГУГЛИТЕ И ВПИШИТЕ В КОНСПЕКТ(“СКАТЫВАЛЬНИК”) !

37.Внешние зависимости

=(

38. CI/CD

Непрерывная интеграция (CI, англ. *Continuous Integration*) — это практика разработки программного обеспечения, которая заключается в слиянии рабочих копий в общую основную ветвь разработки несколько раз в день и выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоёмкость интеграции и сделать её более предсказуемой за счет наиболее раннего обнаружения и устранения ошибок и противоречий.

Требования к проекту

- Исходный код и всё, что необходимо для сборки и тестирования проекта, хранится в репозитории системы управления версиями;

- Операции копирования из репозитория, сборки и тестирования всего проекта автоматизированы и легко вызываются из внешней программы.

Организация

На выделенном сервере организуется служба, в задачи которой входят:

- получение исходного кода из репозитория;
- сборка проекта;
- выполнение тестов;
- развёртывание готового проекта;
- отправка отчетов.

Локальная сборка может осуществляться:

- по внешнему запросу,
- по расписанию,
- по факту обновления репозитория и по другим критериям.

Сборка по расписанию[править | править вики-текст]

В случае сборки по расписанию (англ. *daily build* — рус. *ежедневная сборка*), они, как правило, проводятся каждой ночью в автоматическом режиме — *ночные сборки* (чтобы к началу рабочего дня были готовы результаты тестирования). Для различия дополнительно вводится система нумерации сборок — обычно, каждая сборка нумеруется натуральным числом, которое увеличивается с каждой новой сборкой. Исходные тексты и другие исходные данные при взятии их из репозитория (хранилища) системы контроля версий помечаются номером сборки. Благодаря этому, точно такая же сборка может быть точно воспроизведена в будущем — достаточно взять исходные данные по нужной метке и запустить процесс снова. Это даёт возможность повторно выпускать даже очень старые версии программы с небольшими исправлениями.

Преимущества[править | править вики-текст]

- проблемы интеграции выявляются и исправляются быстро, что оказывается дешевле;
- немедленный прогон модульных тестов для свежих изменений;
- постоянное наличие текущей стабильной версии вместе с продуктами сборок — для тестирования, демонстрации, и т. п.
- немедленный эффект от неполного или неработающего кода приучает разработчиков к работе в итеративном режиме с более коротким циклом.

Недостатки[править | править вики-текст]

- затраты на поддержку работы непрерывной интеграции;
- потенциальная необходимость в выделенном сервере под нужды непрерывной интеграции;

- немедленный эффект от неполного или неработающего кода отучает разработчиков от выполнения периодических резервных включений кода в репозиторий.
    - в случае использования системы управления версиями исходного кода с поддержкой ветвления, эта проблема может решаться созданием отдельной «ветки» (англ. branch) проекта для внесения крупных изменений (код, разработка которого до работоспособного варианта займет несколько дней, но желательно более частое резервное копирование в репозиторий). По окончании разработки и индивидуального тестирования такой ветки, она может быть объединена (англ. merge) с основным кодом или «стволом» (англ. trunk) проекта.

39. Сборка в одну кнопку

=(

40. Сакральные знания в коде

=(