**СТИВ МАККОНЕЛ. СОВЕРШЕННЫЙ КОД.**

Инкапсуляция - скрытие интерфейса обьекта, класса или метода для его упрощенного понимания

Полиморфизм - поддержка языком возможности общих методов при отсутствии информации о конкретном обьекте

Наследование - создание обьекта на основе другого обьекта или класса

1. **ВЫБОР ПОДХОДА РАЗРАБОТКИ**

Последовательный подход (при котором вопросы решаются заблаговременно), если:

* требования довольно стабильны;
* проект приложения прост и относительно понятен;
* группа разработчиков знакома с прикладной областью;
* проект не связан с особым риском;
* важна долговременная предсказуемость проекта;
* на изменение требований, проекта приложения и кода скорее всего окажутся высокими.

Более итеративный подход (при котором вопросы решаются по мере работы) можно предпочесть, если:

* требования относительно непонятны или вам кажется, что они могут оказаться нестабильными по другим причинам;
* проект приложения сложен, не совсем ясен или и то и другое;
* группа разработчиков незнакома с прикладной областью;
* проект сопряжен с высоким риском;
* долговременная предсказуемость проекта не играет особой роли;
* затраты на изменение требований, проекта приложения и кода скорее всего будут низкими.

1. **ПРЕДВАРИТЕЛЬНЫЕ УСЛОВИЯ ЭТАПОВ РАЗРАБОТКИ**

**Определение проблемы** *-* это просто формулировка сути проблемы без каких либо намеков на ее возможные решения. Проблему следует формулировать на языке, понятном пользователю, а сама проблема должна быть описана с пользовательской точки зрения.

**Выработка требований**, она же спецификация - подробное описание того, что должна делать программа. Требования крайне редко бывают стабильны, но ключевые моменты необходимо определить до начала проектирования. Во время реализации среднего проекта требования во время разработки изменяются в среднем на 25%, на что приходится 70-85% обьема повторной работы над типичным проектом. Заказчика необходимо ставить в известность о стоимости изменения требований. *Контрольный список определения качества требований:* Совершенный код, 2 изд., стр 40.

**Проектирование архитектуры.** Архитектура - это высокоуровневая часть проекта приложения, каркас, состоящий из деталей проекта.

Типичные компоненты качественной архитектуры:

* *организация программы* - общее описание системы. Подтверждение того, что при разработке были рассмотрены альтернативные варианты и обоснован выбор окончательной организации системы. Определения основных компонентов программ, ответственность каждого из них, описание, какие другие компоненты данный компонент может использовать непосредственно, какие косвенно, а какие вообще не может.
* *Основные классы*. Определение основных классов приложения, их области ответственности и механизмы взаимодействия с другими классами.
* *Организация данных*. Описание основных видов формата файлов и таблиц. Описание видов доступа и их обоснование. Определение высокоуровневой организации и содержание всех используемых БД.
* *Бизнес*-правила. Архитектура, зависимая от специфических бизнес-правил, должна определять их и описывать их влияние на проект системы.
* *Пользовательский интерфейс*. Пользовательский интерфейс (GUI) часто проектируется на этапе выработки требований. Если это не так, его следует определить на этапе разработки архитектуры.
* *Управление ресурсами*. Архитектура должна включать план управления ограниченными ресурсами, такими как соединения с БД, потоки и дескрипторы. При разработке драйверов, встроенных систем и других приложений, которые будут работать в условиях ограниченной памяти, архитектура должна также определять способ управления памятью.
* *Безопасность.* Методы обработки буферов, cookie, конфигов и шифрования.
* *Производительность.* Оценки производительности и объяснение, почему разработчики архитектуры считают эти показатели достижимыми. Если они могут быть не достигнуты, это тоже должно быть отражено в архитектуре.
* *Масштабируемость -* возможность адаптации к росту требований
* *Взаимодействие с другими системами.*
* *Интернационализация/локализация*
* *Ввод-вывод*
* *Обработка ошибок.* Способы обнаружения и коррекции ошибок - стр. 47
* *Отказоустойчивость.* Стр. 48
* *Возможность реализации архитектуры -* подтверждение того, что система технически осщуствима.
* *Избыточная функциональность.* В мире ПО цепь не так крепка, как слабейшее звено; она так слаба, как все слабые звенья, вместе взятые. В спецификации архитектуры должно быть явно указано, могут ли программисты реализовать в своих блоках программы избыточную функциональность или они должны создать простейшую работоспособную систему.
* *Свои или сторонние подключенные библиотеки.* Объяснение, почему разработанные компоненты лучше готовых
* *Стратегия изменений.* Архитектура должна быть достаточно гибкой, что бы в нее легко было внести вероятные изменения

Контрольный список вопросов по качеству архитектуры на стр. 51.

1. **ПРОЕКТИРОВАНИЕ ПРИ КОНСТРУИРОВАНИИ**

Проектировать надо так, что бы проблема решалась не на каком-то языке, а с использованием языка, обходя слабые стороны используемого языка.

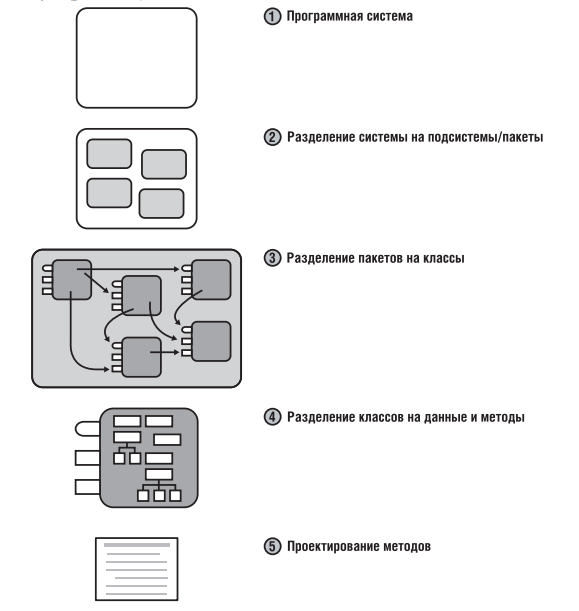
**Проблемы, связанные с проектированием ПО. Проектирование это:**

* Грязная проблема, т.е. та, которую можно ясно определить только путем полного или частичного решения.
* Неряшливый процесс, даже если он приводит к аккуратному результату
* Проектирование подразумевает ограничение возможностей, что бы были как можно более упрощенные решения.
* Неоднозначно недетерменированный процесс. Проектировать можно разными способами.
* Эвристический процесс. Это набор правил, которые "могут сработать".
* Постепенный процесс.

**Желательные характеристики проекта**

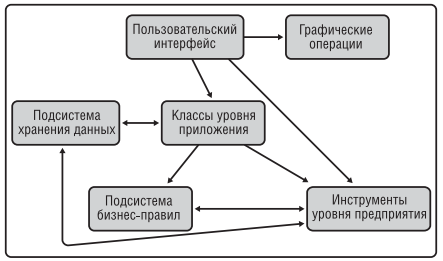
* *Главной целью проектирования должна быть минимизация сложности -* создавать простые и понятные проекты, разделенные на подсистемы
* *Простота сопровождения.* Постоянно думать о тех, кто будет проект сопровождать
* *Слабое сопряжение (loose coupling)* - сведение к минимуму числа соединеинй между разными частями программы.
* *Расширяемость* - возможность улучшение системы без нарушения ее основной структуры. Изменение одного фрагмента не должно влиять на ее остальные
* *Возможность повторного использования* - фрагменты системы можно было бы повторно использовать в других системах.
* *Высокий коэффициент объединения по входу* - к конкретному классу обращается среднее число других классов
* *Низкий или средний коэффициент разветвления по выходу* - конкретный класс обращается к малому или среднему числу других классов. Высокий коэффициент разветвления по выходу (fan#out) (более семи) говорит о том, что класс использует большое число других классов и, возможно, слишком сложен.
* *Портируемость* - возможность легко адаптировать к другой среде
* *Минимальная, но полная функциональность* - проект закончен, когда из него ничего нельзя выбросить. Опасайтесь вопроса: «Эту функцию реализовать легко — почему бы этого не сделать?
* *Стратификация* - возможность изучения системы на отдельных уровнях, игнорируя другие уровни. Например, если вы создаете современную систему, которая должна использовать большой объем старого, плохо спроектированного кода, напишите уровень, отвечающий за взаимодействие со старым кодом.
* *Соответствие стандартным (общепринятым) методикам проектирования*

**Уровни проектирования**

****

**Уровень 1: программная система.**

**Уровень 2: разделение системы на подсистемы.**Типичные подсистемы - модуль GUI, бизнес-правил, доступа к БД, изоляции зависимостей от ОС. Этот уровень нужен для проектов, требующих нескольких недель или больше. Важный аспект - определение правил взаимодействия подсистем, ограничение свободного взаимодействия между собой.

Чтобы соединения подсистем были понятными и легкими в сопровождении, старайтесь поддерживать простоту отношений между подсистемами. Самым простым отношением является то, при котором одна подсистема вызывает методы другой. Более сложное отношение имеет место, когда одна подсистема содержит классы другой. Самое сложное отношение — наследование классов одной подсистемы от классов другой.

Придерживайтесь одного разумного правила: диаграмма системного уровня вроде той, что показана на рисунке, должна быть ациклическим графом. Иначе говоря, программа не должна содержать циклических отношений, при которых класс A использует класс B, класс B использует класс C, а класс C — класс А.

**Уровень 3: разделение подсистем на классы.**Определяются все классы системы. Определяются детали взаимодействия каждого класса с остальными элементами системы, особенно интерфейс класса.

Объект — это любая конкретная динамическая сущность, имеющая конкретные значения и атрибуты и существующая в период выполнения программы. Класс — это статическая сущность, с которой вы имеете дело, просматривая листинг программы. Например, вы можете объявить класс Person (человек), имеющий такие атрибуты, как фамилия, возраст, пол и т. д. В период выполнения вы будете работать с объектами nancy, hank, diane, tony и т. д. — иначе говоря, со специфическими экземплярами класса. Если вы знакомы с терминологией БД, различие между классом и объектом аналогично различию между «схемой» и «экземпляром». Класс можно рассматривать как форму для выпечки булочек, а объекты — как сами булочки.

**Уровень 4: разделение классов на методы.** Определение закрытых методов классов. При этом многие методы окажутся простыми, тогда как другие будут включать иерархии методов и потребуют доп. проектирования. Формально выполнять этот этап не обязательно, но хотя бы про себя выполнить его нужно.

**Уровень 5. Проектирования методов.** Детальное определение функциональности определенных методов. Уровень проектирования выполняется по всех случаях, но не всегда осознанно

**Эвристические принципы в проектировании**

**Определить обьекты реального мира:**

* обьекты и их атрибуты
* действия, которые могут быть выполненны над каждым обьектом
* действия, которые каждый обьект может выполнять над другими обьектами
* продумать интрефейс обьектов - что скрыть, а что показать

**Определить уровни абстракции** - дом это дом, а не набор кирпичей и коммуникаций

Инкапсулировать детали реализации, что бы представить сложный обьект в простой форме

Использовать наследование, если оно упрощает проектирование

**Для упрощения сокрытия информации:**

* избегать избыточного распространения информации, собирать информацию в одном месте
* Избегать круговой зависимости, когда метод класса A вызывает метод класса B, а метод класса B — метод класса A
* Не представлять класс как глобальный - Если программа использует огромные классы, включающие десятки методов, различие между данными класса и глобальными данными стирается. Лучше грамотно проектировать небольшие классы
* Почаще задавать вопрос "Что мне скрыть?", "Что этот класс должен скрывать?"

**Определить области вероятных изменений,** отделить и изолировать их. Частые области изменений:

* Зависимость от оборудования - интерфейсы между программой и железом, другими программами
* Вввод-вывод
* Нестандартные возомжности языка
* Сложные аспекты проектирования и конструирования
* Переменные статуса - например ошибку лучше определять не как булеву переменную, а сделать перечисление со значениями *ErrorType\_None, ErrorType\_Warning и ErrorType\_Fatal.* Вместо непосредственной проверки переменной используйте методы доступа. Так вы сохраните возможность реализации более сложного механизма определения состояния.
* Размер структуры данных - вместо хардкодинга чисел, лучше использовать переменные

**Предвосхищение изменений**

Хороший способ - сначала определить области, которые крайне необходимы пользователям. Это подмножество и составит ядро системы. Дальше определять минимальные инкрементные приращения системы

**Поддержка сопряжения слабым**

* Чем меньше соединений между модулями (классами или обьектами), тем лучше. Класс, имеющий 4 грамотно определенных открытых метода лучше, чем класс с 37 открытыми методами
* Видимость - передача данных через список параметров формирует очевидную связь, в отличии от передачи в глобальных данных, где не видно, что именно передается
* Гибкость. Чем проще вызывать модуль из других модулей, тем слабее он сопряжен. Если один метод работает с другим обьектом, который передает в него информацию, то лучше сделать передачу информации конкретными числами в виде аргументов, чем передавать целый обьект с параметрами, и обрабатывать этот обьект внутри

**Виды сопряжения**

* Сопряжение посредством даннных параметров - между модулями передаются только элементарные типы даных, причем передаются через списки параметров. 5\5.
* Посредством обьекта - модуль создает экземпляр обьекта. 5\5
* Передача обьекта-параметра. Принимающему обьекту нужно обладать информацией о поступающем обьекте, это хуже, 4\5
* Семантическое сопряжение - когда принимающий или отдающий модуль зависит от внутреннего устройства другого. Опасно тем, что изменение кода в отдающем модуле может нарушить работу принимающего модуля, так что ошибку будет сложно найти. 2\5

**Шаблоны проектирования (Design patterns)**

Надо стараться использовать один из шаблонов проектирования - потому что другие быстрее поймут и многие проблемы уже решены, это как подключение библиотек.

Хорошие книги по проектированию на стр. 116.

**Другие полезные эвристические принципы**

**Стремитесь к максимальной связанности - с**вязанность - насколько все методы класса или все фрагменты метода соответствуют главной цели Класса. Переклюкается с определением абстракций

**Формируйте иерархии -** иерархия устраняет детали, выталкивая их на другой уровень

**Формализовать контакты классов -** класс гарантирует выполнение четко определенных условий, если в него будут передаваться четко определенные данные

**Грамотно назначать сферы ответственности** - что скрыть, а что показать

**Проектировать систему для тестирования -** Проектирование для тестирования часто приводит к разработке более формализованных интерфейсов классов, что обычно выгодно

**Избегать неудач -** принимать во внимание неудачи прошлых проектов

**Тщательно выбирать время связывания** - это тот момент, когда переменной присваивается конкретное значение. Следует подумать, "А что если связать раньше? А если позже?"

**Создать центральные точки управления -** место, где можно вносить изменения во все модули, подключать дополнительные

**Решение в лоб** - бывает очень эффективным

**Рисование диаграм**

**Поддержка модульности проекта системы -** каждый класс должен быть похож на черный ящик - вы знаете, что в него поступает и что из него выходит, но не знает, что происходит внутри

**Методики проектирования**

**Использование итераций**

Проектирование итеративный процесс, потому пройди из точки А в точку Б, можно вернутся. Каждая последующая итерация проектирования делает проект лучше.

**Инкрементное улучшение**

Разделить программу на области спроектировать их по-отдельности

**Нисходящий и восходящий подходы к проектированию**

Нисходящее проектирование обычно начинается с простого, но иногда низкоуровневые сложности прорываются на вершину, и это может при-водить к усложнению системы, которого можно было избежать. Восходящее проектирование начинается со сложных аспектов, определение этой сложности на ранних этапах позволяет лучше спроектировать высокоуровневые классы...если к этому моменту сложность не потопит всю систему!

**Экспериментальное прототипирование**

Написание *абсолютно минимального* объема подлежащего выбрасыванию кода, нужного для ответа на отдельный вопрос проектирования. Причем задача должна быть очень конкретна:

«Будет ли эта организация базы данных поддерживать 1000 транзакций в секунду при условиях X, Y и Z?»

**Какую часть проектирования считать достаточной?**  
Проектирование можно остановить, когда проще написать сразу код, чем проектировать. Проектирование редко бывает достаточным

**КЛАССЫ**

**Класс** — это набор данных и методов, имеющих общую, целостную, хорошо определенную сферу ответственности. Данные — необязательный компонент класса: класс может включать только методы, предоставляющие целостный набор услуг.

Основы классов - **абстрактные типы данных** (АТД). Это набор, включающий данные и выполняемые над ними операции. Операции описывают данные для остальной части программы и позволяют их изменять. Слово «данные» используется в выражении «абстрактный тип данных» довольно условно. АТД может быть графическое окно со всеми влияющими на него операциями, файл с файловыми операциями, таблица страховых тарифов с соответствующими операциями и др.

**Преимущества АТД перед функциональными принципами:**

* Возможность сокрытия деталей реализации. Для возможности менять код в одном месте
* Ограничение области изменений. Можно изменять один фрагмент и он не будет влиять на остальную часть программы
* Более высокая информативность интерфейса. Код currentFont. size = 16 неоднозначен, так как число 16 может определять размер шрифта и в пикселах, и в пунктах.

currentFont.SetSizeInPoints( sizeInPoints )  
currentFont.SetSizeInPixels( sizeInPixels )

* Легкость оптимизации кода **-** поскольку можно переписать несколько методов, а не искать их по всей системе
* Легкость проверки кода.
* Удобочитаемость и понятность кода.
* Ограничение области использования данных.
* Возможность работы с сущностями реального мира

Представляйте в форме АТД распространенные низкоуровневые типы данных.

Обычно при обсуждении АТД речь идет о представлении в форме АТД популярных низкоуровневых типов данных. Как вы могли заметить по примерам, в форме АТД можно представить стек, список, очередь и почти любой другой популярный тип данных. Спросите себя: «Что представляет этот стек, список или эта очередь?» Если стек представляет набор сотрудников, рассматривайте АТД как набор сотрудников, а не как стек. Если список соответствует набору счетов, рассматривайте его как набор счетов. Если очередь представляет ячейки электронной таблицы, обращайтесь с ней как с набором ячеек, а не обобщенных элементов. Используйте как можно более высокий уровень абстракции

Представляйте в форме АТД даже простые элементы

Например, есть объект "Фонарь". Представление фонаря и его операций в форме АТД облегчает понимание и изменение кода, ограничивает потенциальные следствия изменений методов *TurnLightOn()* и *TurnLightOff()* и снижает число элементов данных, которые нужно передавать в методы.

Качественные интерфейсы классов

1. Хорошая абстракция. Та абстракция, методы которой служат одной цели. Все внутренние детали реализации скрыты. Это значит, что не надо примешивать в абстракции методы, которые слабо в него вписываются
2. Выражайте в интерфейса класса согласованный уровень абстракции. Это значит, что любые другие методы или обьекты, которые не должны быть включены в данную абстракцию, лучше вынести в отдельный класс. Например Работник и Зарплата - два разных класса, не должны включать методы один другого.
3. Убедитесь, что вы понимаете, реализацией какой абстракции является класс. Если класс реализует более одного АТД или если вам не удается определить, реализацией какого АТД класс является, самое время реорганизовать класс в один или несколько хорошо определенных АТД.
4. Предоставляйте методы вместе с противоположными им методами. Если есть метод Class.open(), должен быть и Class.close()
5. Убирайте постороннюю информацию в другие классы. Если половина методов класса работает с одними данными, а половина - с другими, пора разделить класс на 2 класса.
6. Делать интерфейсы классов программными, а не семантическими. Любой аспект класса, который не может быть проверен компилятором, является потенциальным источником ошибок. Старайтесь преобразовывать семантические элементы интерфейса в программные, используя утверждения (assertions) или иными способами
7. Опасайтесь нарушения целостности интерфейса при изменении класса.

Пример интерфейса, изуродованного при сопровождении

программы (C++)

class Employee {

public:

...

// открытые методы

FullName GetName() const;

Address GetAddress() const;

PhoneNumber GetWorkPhone() const;

...

bool IsJobClassificationValid( JobClassification jobClass );

bool IsZipCodeValid( Address address );

bool IsPhoneNumberValid( PhoneNumber phoneNumber );

SqlQuery GetQueryToCreateNewEmployee() const;

SqlQuery GetQueryToModifyEmployee() const;

SqlQuery GetQueryToRetrieveEmployee() const;

...

private:

...

};

То, что начиналось как ясная абстракция, превратилось в смесь почти несогласованных методов. Между сотрудниками и методами, проверяющими корректность почтового индекса, номера телефона или ставки зарплаты (job classification), нет логической связи. Методы, предоставляющие доступ к деталям SQL запросов, относятся к гораздо более низкому уровню абстракции, чем класс Employee, нарушая общую абстракцию класса.

1. Добавляя в интерфейс класса новый метод, постоянно себя спрашивать "А хорошо ли согласуется этот метод с абстракцией класса?". Если нет - не включать его в класс.

**ХОРОШАЯ ИНКАПСУЛЯЦИЯ**

Инкапсуляция не позволит узнать детали реализации, даже если вы захотите.

1. **Минимизируйте доступность классов и их членов.** Если не знаете, каким делать метод - открытым, закрытым или защищенным, Макконел предлагает спросить себя, какой вариант лучше всего сохраняет целостность абстракции
2. **Не делайте данные-члены открытыми.** Все данные типа

var x = 100;

Следует скрыть в методы

Class.getX();

Class.setX();

1. **Не включайте в интерфейс класса закрытые детали реализации.** Если детали реализации скрыть проблематично (как в C++), лучше их вынести в отдельный заголовочный файл.
2. **Не делайте предположений о клиентах класса.** Класс не должен делать предположений о том, как интерфейс будет или не будет использоваться, все должно быть четко определено
3. **Избегайте использования дружественных классов.**
4. **Легкость чтения кода выше, чем удобство написания.** Даже если метод плохо согласуется с абстракцией итерфейса, иногда хочется его включить в интерфейс. Этого делать не стоит.
5. **Крайне осторожно относится к семантическим нарушениям инкапсуляции:**

* решить не вызывать метод InitializeOperations() Класса A, потому что метод PerformFirstOperation() Класса A вызывает его автоматически;
* использовать константу *MAXIMUM\_ELEMENTS* Класса B вместо константы *MAXIMUM\_ELEMENTS* Класса A, потому что знаете, что они имеют одинаковые значения.
* использовать указатель или ссылку на Объект B, созданный Объектом A, даже после выхода Объекта A из области видимости, потому что знаете, что Объект A хранит Объект B в статическом хранилище, в следствие чего Объект B все еще будет корректным;

*Со всеми этими примерами связана одна и та же проблема: зависимость клиентского кода от закрытой реализации класса, а не от его открытого интерфейса. Каждый раз, когда вы смотрите на реализацию класса, что бы узнать, как его использовать, вы программируете не в соответствии с интерфейсом, а сквозь интерфейс в соответствии с реализацией. Программирование сквозь интерфейс разрушает инкапсуляцию, а вскоре к ней присоединяется и абстракция.*

1. **Остерегаться слишком жесткого сопряжения.** Что бы сила связи (число соединений) между двумя классами было минимальным

**ПРОЕКТИРОВАНИЕ КЛАССА**

**Включение -** один класс содержит примитивный элемент или другой класс.

Относится настороженно к классам, в которых содержиться более семи элементов данных или методов. После увеличения этого числа класс становится сложным для удержания в голове

**Наследование.**

* Если классы, которые производные от базового, не собираются *полностью* придержиться контракта, определенного интерфейсом базового класса, наследование выполнять не стоит.
* Проектировать классы с учетом возможности их наследования или запретить его совсем.
* Соблюдать принцип подстановки Лисков - все методы базового класса должны иметь в каждом производном классе ту же семантику. Иначе наследование лучше не использовать.
* Убедиться, что наследуется только то, что хочется наследовать. Если вам нужна реализация класса, но не его интерфейс, используйте включение, а не наследование.
* Не переопределять непереопределяемые методы-члены
* С подозрением относится к классам, обьекты которых создаются в единственном экземпляре.Может просто создать обьект вместо класса? Singleton - исключение
* С подозрением относится к базовым классам, имеющими только одни производный класс. Скорее всего, прокраммист проектировал "наперед", не понимая, че творит
* С подозрением относитесь к классам, которые переопределяют метод,оставляя его пустым. Как правило, это говорит о неудачном проектировании базового класса. Допустим, вы создали класс Cat, включающий метод Scratch() (царапать), но после обнаружили, что некоторые коты лишены когтей и не могут царапаться. Вы могли бы унаследовать от класса Cat класс ScratchlessCat, переопределив в нем метод Scratch() так, чтобы он ничего не делал.

Исправлять эту проблему следует не в базовом классе, а в первоначальном классе Cat. Создайте класс Claws (когти) и включите его в класс Cats.

* Не использовать более двух-трех уровней наследования, иначе их будет сложно удержать в пямяти
* Наличие в коде большого количества блоков case может указывать на то, что лучше было бы спроектировать, используя наследование, но только если код в блоках case не сильно отличается по смысловой нагрузке друг от друга (пример - рисование разных форм - круг, квадрат и т.д.)
* Делать все данные закрытыми а не защищенными.
* Применение множественного наследования полезно главным образом только при создании «миксинов» — простых классов, позволяющих добавить ряд свойств в другой класс.

Ключевые правила наследования:

* если несколько классов имеют общие данные, но не формы поведения, создайте общий объект, который можно было бы включить во все эти классы;
* если несколько классов имеют общие формы поведения, но не данные, сделайте эти классы производными от общего базового класса, определяющего общие методы;
* если несколько классов имеют общие данные и формы поведения, сделайте эти классы производными от общего базового класса, определяющего общие данные и методы;
* используйте наследование, если хотите, чтобы интерфейс определялся базовым классом, и включение, если хотите сами контролировать интерфейс.

**Методы-члены и данные-члены**

* включать в класс как можно меньше методов
* блокировать неявно сгенерированные методы и операторы, которые не нужны
* уменьшать число разных методов, вызываемых классом - ведут к ошибкам
* избегать вызовов методов обьекта в обьекте. In the account example above, that means account.ContactPerson() is OK, but account.ContactPerson().DaytimeContactInfo() is not.
* минимизировать сотрудничество класса с другими классами

**Конструкторы**

* Стараться инициализировать все данные в конструкторе, и в том порядке,в каком они обьявлены
* Предпочесть полное копирование (включая данные-члены) ограниченному

**Какие классы создавать НЕ надо**

- избегать создания GOD-классов

- удалять классы, которые содержат только данные, без поведения (методов)

- избегать классов, имена которых начинаются после глаголов, типа DatabaseInitialization() or StringBuilder(). Такие классы стоит сделать методами другого класса.

**ПРОЕКТИРОВАНИЕ МЕТОДОВ (Процедур)**

Причины создания функций и процедур такие же, как и классов: уменьшение сложности программы, увеличение читаемости кода, возможность спрятать очереди и указатели, улучшение читаемости булевых тестов. Функции не обязательно должны быть маленькими, как классы.

Один из основных стоп-блоков у людей - нежелание делать функцию, содержащую одну строчку. Но делать ее стоит! В будущем она разрастется до большей. Это повышает читаемость, а правильно подобранное имя функции сразу ее документирует.

Самая важная составляющая успешного проектирования функции - ее связанность, в отличии от классов, где решающей является абстракция. Функция cosinus() хорошо связана, но функция cosinusAndTangens() уже хуже

1. **Функциональная связанность** – самая сильная и лучший тип связанности, возникающий, когда функция выполняет одну и только одну операцию. Например – sin(), getCustomerName(), eraseFile() и т.д. Важно подобрать ей показательное имя
2. **Последовательная связанность** – возникет, когда функция содержит ряд операций, которые должны быть выполенны в определенном порядке. Из последовательной функции можно сделать функциональную, если разделить последовательную функцию на две или больше функциональных, которые могут вызывать друг друга нужным образом
3. **Куммуникативная связанность** возникает, когда операции используют одни и те же данные, и не связаны между собой никоим образом. Лучше тоже разделить коммуникативную функцию на функциональные подфункции, которые будут вызываться общей коммуникативной функцией.
4. **Временная связанность** возникает, когда функции выполняются одновременно, например startup(), shutdown(). Пихать туда лапшу кода считается плохой практикой, лучше сделать управляющую функцию, которая будет вызывать другие функции

Другие виды связанности неприемлемы, и если такая функция сбоит, ее легче переписать, чем искать ошибку

**Создание отличных имен функций и методов**

* Описывать все, что функция делает. Если имя слишком длинное, значит функция с побочными эффектами, которые можно вынести в отдельные функции
* Подбирать выразительные имена. handleOutput() - очень общее имя, лучше подходит formatAndPrintOutput(). Если функции сложно подобрать конкретное имя, скрее целевое назначение функции так же неясно (см. функциональная связанность)
* Делать имена функции настолько длинным, насколько нужно
* Называть функцию, используя описание того, что она возвращает: sin(), currentColor()
* В функции нужно указывать, над каким обьектом она проиводит действие: printDocument(), calcMountlyProfit(). В нормально спроектированном классе, названием обьекта служит сам обьект: document.print(); mountlyProfit.calc()
* Использовать противоположные названия для противоположных функций:

add/remove, begin/end, create/destroy, first/last, get/put, get/set, increment/decrement, insert/delete, lock/unlock, min/max, next/previous, old/new, open/close, show/hide, source/target, start/stop, up/down

* Придерживаться соглашений про имена методов и переменных, что бы не получилось так: employee.id.Get(), dependent.GetId(), candidate.id()

Данные по длинне функций противоречивы, однако рекомендуется делать функции не длиннее 200 строк, т.к. дальше функцию уже сложно удержать в голове

**Использование аргументов методов**

Интерфейсы методов являются самой предрасположенной к ошибкам областью (40% всех ошибок). Для уменьшения вероятности ошибок:

* Располагать аргументы в таком порядке: сначала только входящие (input), потом входящие и исходящие (input & output), последние - только исходящие (output). Такая система в параметрах методов jQuery
* Если несколько методов принимают одни и те же аргументы, следует передавать их в одном и том же порядке - для упрощения запоминания
* Использовать все передаваемые аргументы, или не передавать неиспользуемых
* Передавать переменные статуса или кода ошибок последними
* Не использовать передаваемые аргументы методу в качестве переменных. Лучше сразу их переназвать своими, локальными именами в самом начале функции.
* Создавая переменные из аргументов, сразу писать к ним комментарии по поводу того, что они собираются делать, или что *предполагается*, что они будут делать. Так же можно комментировать аргументы функции в том месте, где она вызывается

Что конкретно указывать в комментах:

* аргумент только на вход, на выход или модифицируемый
* мера измерения (дюймы, метры и т.д.)
* Обозначения статусов кодов и значений ошибок
* Диапазон ожидаемых значений
* Специфичные значения, которые никогда не появятся
* Ограничить число принимаемых аргументов до 7, объект при этом может быть как отдельный аргумент, логическая структура со значениями внутри. Если постоянно приходится передавать слишком большое количество аргументов, значит методы слишком сильно сопряжены, и их стоит переделать.
* Передавать в метод те переменные или объекты, которые нужны ему для поддержания абстракции интерфейса:
* если абстракция подразумевает, что метод ожидает три конкретных элемента данных, которые по чистой случайности принадлежат одному объекту, передайте три элемента по отдельности.
* если же абстракция состоит в том, что элементы данных всегда принадлежат конкретному объекту, над которым метод должен выполнять ту или иную операцию, тогда, раскрывая три этих специфических элемента, вы на самом деле нарушаете абстракцию.

Обычно, если передаваемые параметры приходится собирать в обьект, а на выходе разбирать их назад, скорее всего метод плохо спроектирован.