

Язык программирования C++

# Проектирование ООП программ

Преподаватели:

Пысин Максим Дмитриевич, ассистент кафедры ИКТ

Краснов Дмитрий Олегович, аспирант кафедры ИКТ

Лобанов Алексей Владимирович, аспирант кафедры ИКТ

Крашенинников Роман Сергеевич, аспирант кафедры ИКТ

**КАК ДОЛГО МОЖДНО**



**УЧИТЬ C++**

meme-generator.ru

# Подходы проектирования программ



Классы мы писали, мы писали, смысла мы всего не понимали... А как собственно строить программы на основе классов?

- SOLID
  - single responsibility principle (Принцип единственной ответственности)
  - open-closed principle (Принцип открытости/закрытости)
  - Liskov substitution principle (Принцип подстановки Лисков)
  - interface segregation principle (Принцип разделения интерфейса)
  - dependency inversion principle (Принцип инверсии зависимостей)
- Don't repeat yourself
- KISS – keep it simple, stupid
- YAGNI – You aren't gonna need it
- BDUF – Big Design Up Front
- Композиция вместо наследования
- Чем хуже, тем лучше
- Шаблоны проектирования
- Архитектурные шаблоны





# Принцип единства ответственности

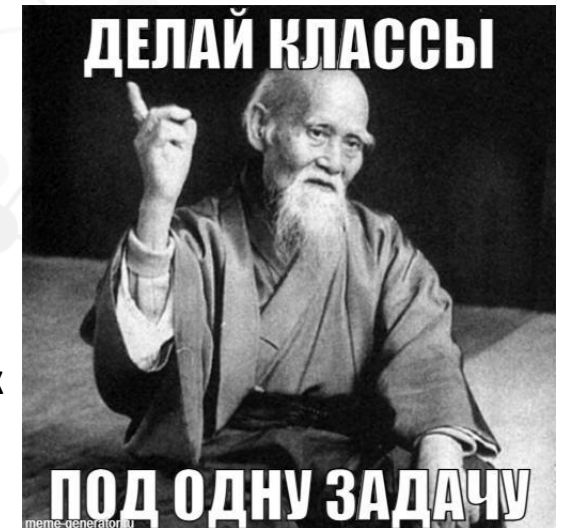


Это принцип, обозначающий, что каждый **объект должен иметь одну ответственность** и эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.

!Верным признаком необходимости применения принципа, можно считать ситуацию когда, при изменении кода, отвечающего за одну ответственность, в приложении появляются исправления кода, отвечающего за другую ответственность, то это первый сигнал о нарушении SRP.

**Слепое следование** принципу единственной ответственности **приводит к избыточной сложности приложения**, его поддержки и тестирования. SRP стоит **применять только тогда, когда это оправдано**. Принцип SRP можно применить только в том случае, когда:

- объекту класса становится позволительно слишком много;
- доменная логика концентрируется только в одном классе;
- любое изменение логики поведения объекта приводит к изменениям в других местах приложения, где это не подразумевалось изначально;
- приходится тестировать, исправлять ошибки, компилировать различные места приложения, даже если за их работоспособность отвечает третья сторона;
- невозможно легко отделить и применить класс в другой сфере приложения, так как это потянет ненужные зависимости.



# Принцип открытости/закрытости



Принцип, устанавливающий следующее положение: **«программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения».**

Принцип открытости/закрытости означает, что программные сущности должны быть:

- **открыты для расширения:** означает, что поведение сущности может быть расширено путём создания новых типов сущностей.
- **закрыты для изменения:** в результате расширения поведения сущности, не должны вноситься изменения в код, который эту сущность использует.

Термин «*принцип открытости/закрытости*» имеет два значения:

- Принцип открытости/закрытости Мейера (Оригинальный принцип предполагающий договорной запрет на добавление функционала путем изменения оригинального класса, но допускающего расширение его функционала в наследниках).
- Полиморфный принцип открытости/закрытости (Более актуальный вариант предполагающий наличие базового интерфейсного класса определяющего основу которую реализует каждый наследник перед добавлением функционала).



# Принцип подстановки Лисков

Принцип который гласит, что функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.



Принцип подстановки (замещения) Лисков накладывает ряд ограничений:

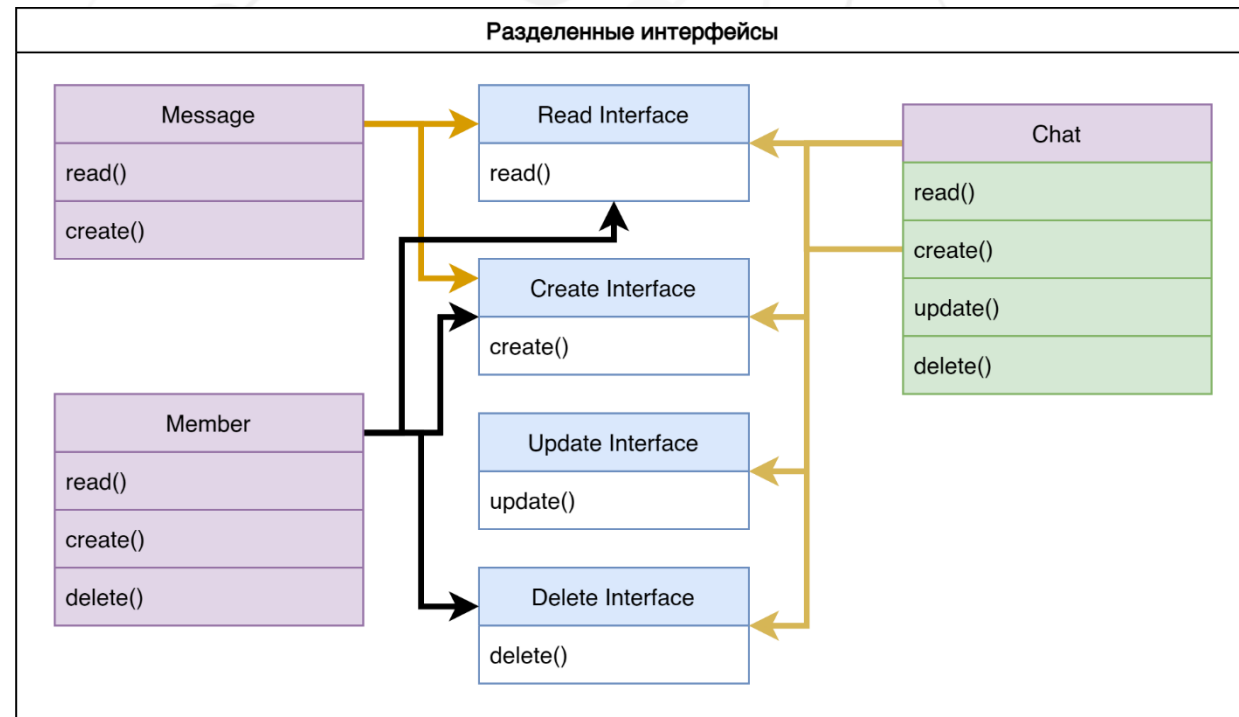
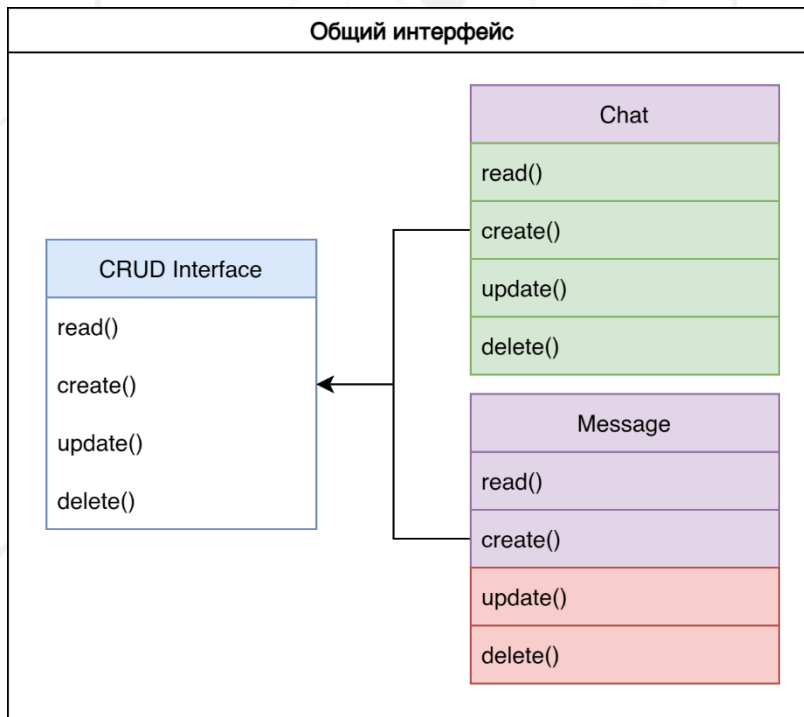
- Предусловия\* не могут быть усилены в подклассе (\*Действия по подготовке вызова функции).
- Постусловия\* не могут быть ослаблены в подклассе(\*Действия по подготовке ответа функции).
- Исторические ограничения— подкласс не должен создавать новых мутаторов свойств базового класса. Иными словами, неизменяемые данные базового класса не должны быть изменяемыми в подклассе.

Принцип Барбары Лисков заставляет задуматься о том, что такое «декларация типа» в терминах объектно-ориентированного языка программирования, который мы используем. Задавая себе ряд вопросов, можно спроектировать систему отвечающую этому принципу:

- Достаточно ли нам описать интерфейс объекта с помощью обычного абстрактного класса со списком методов, типами параметров и возвращаемого значения?
- Каким образом мы можем декларировать требования к значениям параметров метода и свойства, которыми будет обладать возвращаемое значение?
- Как нам описать исключения, которые может сгенерировать метод во время выполнения?
- Как нам описать изменение состояния объекта на разных этапах его жизненного цикла?

# Принцип разделения интерфейса

Принцип разделения интерфейсов говорит о том, что **слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические**, чтобы программные сущности маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться программные сущности, которые этот метод не используют.





# Принцип инверсии зависимостей

Принцип объектно-ориентированного программирования, суть которого состоит в том, что **классы должны зависеть от абстракций**, а не от конкретных деталей.

Принцип подразделяется на две части:

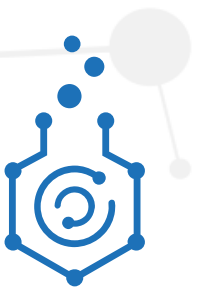
- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Термины:

- Модули (или классы) верхнего уровня = классы, которые выполняют операцию при помощи инструмента
- Модули (или классы) нижнего уровня = инструменты, которые нужны для выполнения операций
- Абстракции – представляют интерфейс, соединяющий два класса
- Детали = специфические характеристики работы инструмента

Что такое модули верхних уровней? Как определить этот уровень? Как оказалось, все очень просто. **Чем ближе модуль к вводу/выводу, тем ниже уровень модуля.** Т.е. модули, работающие с ВД, интерфейсом пользователя, низкого уровня. А модули, реализующие бизнес-логику — высокого уровня.

Что такое **зависимость** модулей? Это **ссылка на модуль в исходном коде**, т.е. `import`, `require` и т.п. С помощью динамического полиморфизма в runtime можно обратить эту зависимость.





# DRY - Don't Repeat Yourself



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»

Дублирование кода – **пустая трата времени и ресурсов**. Вам придется поддерживать одну и ту же логику и тестировать код сразу в двух местах, причем если вы измените код в одном месте, его нужно будет изменить и в другом.

В большинстве случаев дублирование кода происходит из-за незнания системы. **Прежде чем что-либо писать, проявите прагматизм: осмотритесь**. Возможно, эта функция где-то реализована. Возможно, эта бизнес-логика существует в другом месте. Повторное использование кода – всегда разумное решение.

Когда вы разрабатываете крупный проект, часто приходится сталкиваться с избыточной общей сложностью реализации. Люди плохо справляются с управлением сложных систем, им лучше удастся находить необычные решения определенных задач. Самое простое решение по уменьшению сложности – разделить систему на мелкие, независимые модули, которыми проще управлять.

Следует обратить внимание, что DRY говорит не просто о коде, и не столько о коде, сколько о информации с которой он работает, у нас по сути **не должно существовать несколько источников описания одной и той же информации**.

# KISS – keep it simple, stupid



Принцип KISS утверждает, что большинство систем работают лучше всего, **если они остаются простыми, а не усложняются**. Поэтому в области проектирования простота должна быть одной из ключевых целей, и следует избегать ненужной сложности.

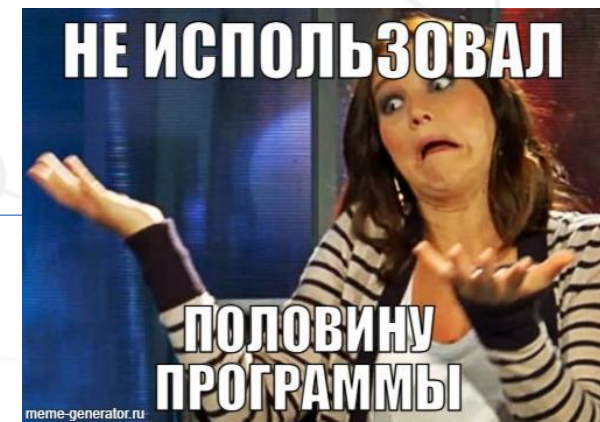
Простые рекомендации по упрощению:

- **Разбивайте задачи на подзадачи**, которые не должны, по вашему мнению, длиться более 4-12 часов написания кода
- Каждая задача должна решаться **одним или парой классов**
- **Сохраняйте ваши методы маленькими**. Каждый метод должен состоять не более чем из 30-40 строк. Каждый метод должен решать одну маленькую задачу, а не множество случаев. Это повысит читаемость, позволит легче поддерживать код и быстрее находить ошибки в нём. Вы полюбите улучшать код.
- **Сохраняйте ваши классы маленькими**. Здесь применяется та же техника, что и с методами.
- **Сначала придумайте решение задачи, потом напишите код**. Никогда не поступайте иначе. И не бойтесь переписывать код ещё, ещё и ещё... В счёт не идёт число строк, до тех пор пока вы считаете, что можно ещё меньше/ещё лучше.
- **Не бойтесь избавляться от кода**. Изменение старого кода и написание нового решения — два важных момента. Если вы столкнулись с новыми требованиями, или не были оповещены о них ранее, тогда порой лучше придумать новое, более изящное решение, решающее и старые, и новые задачи.

# YAGNI - You aren't gonna need it

Принцип проектирования ПО, при котором в качестве основной цели и/или ценности декларируется **отказ от избыточной функциональности**, — то есть отказ добавления функциональности, в которой нет непосредственной надобности. Считается, что не следование принципу приводит к:

- **Тратится время**, которое было бы затрачено на добавление, тестирование и улучшение необходимой функциональности.
- Новые функции **должны быть отлажены, документированы и сопровождаться**.
- Новая функциональность ограничивает то, что может быть сделано в будущем, — ненужные новые функции могут впоследствии **помешать добавить новые нужные**.
- Пока новые функции действительно не нужны, **трудно полностью предугадать, что они должны делать**, и протестировать их. Если новые функции тщательно не протестированы, они могут неправильно работать, когда впоследствии понадобятся.
- Это приводит к тому, что программное обеспечение **становится более сложным** (подчас чрезмерно сложным).
- Если вся функциональность не документирована, она может так и остаться неизвестной пользователям, но может создать различные **риски для безопасности** пользовательской системы.
- Добавление новой функциональности может привести к **желанию ещё более новой функциональности**, приводя к эффекту «снежного кома».



# BDUF - Big Design Up Front



Этот принцип говорит о том, что перед тем как программировать какую-либо систему, **требуется потратить достаточно усилий на ее проектирование**, а так же, решение локальных проблем не должно нарушать общей картины.

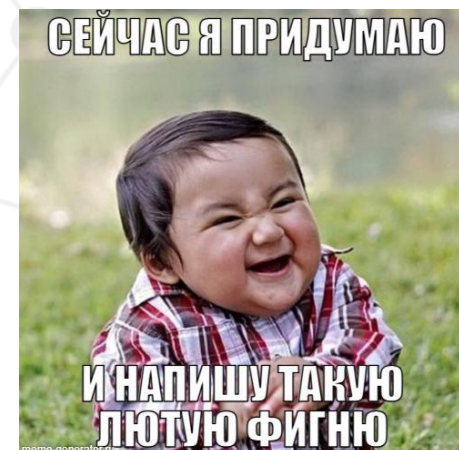
Многие разработчики считают, что если они не пишут код, то они не добиваются прогресса. Это неверный подход. **Составив план, вы избавите себя от необходимости раз за разом начинать с нуля.**

Иногда в недостатках и процессах разработки архитектуры **должны быть замешаны и другие люди**. Чем раньше вы все это обсудите, тем лучше будет для всех.

Очень распространенный контраргумент заключается в том, что стоимость решения проблем зачастую ниже стоимости времени планирования. Чем с меньшим количеством ошибок столкнется пользователь, тем лучше.

Один из способов применения принципа BDUF в проекте с изменяющимися требованиями:

Сначала создать общую архитектуру. Затем необходимо разделить требования на несколько этапов в соответствии с приоритетами. В процессе разработки начните с этапа с самым высоким приоритетом, постепенно опускаясь до самого низкого. На каждом этапе используйте этот принцип перед началом разработки.





# Композиция вместо наследования



Это принцип, согласно которому классы должны **достигать полиморфного поведения и повторное использование кода посредством их композиции** (путем включения экземпляров других классов, реализующих желаемую функциональность), а не наследования от базового или родительского класса. Принцип говорит, что программисту требуется использовать наследование в тех ситуациях для которых оно предназначено, и не использовать там, где можно обойтись простой композицией.

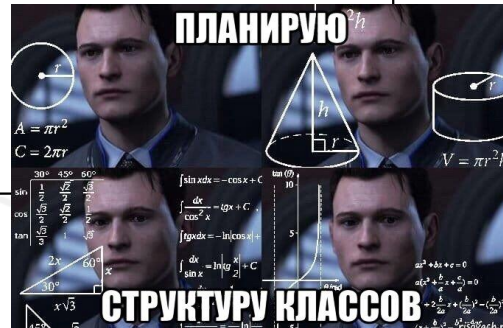
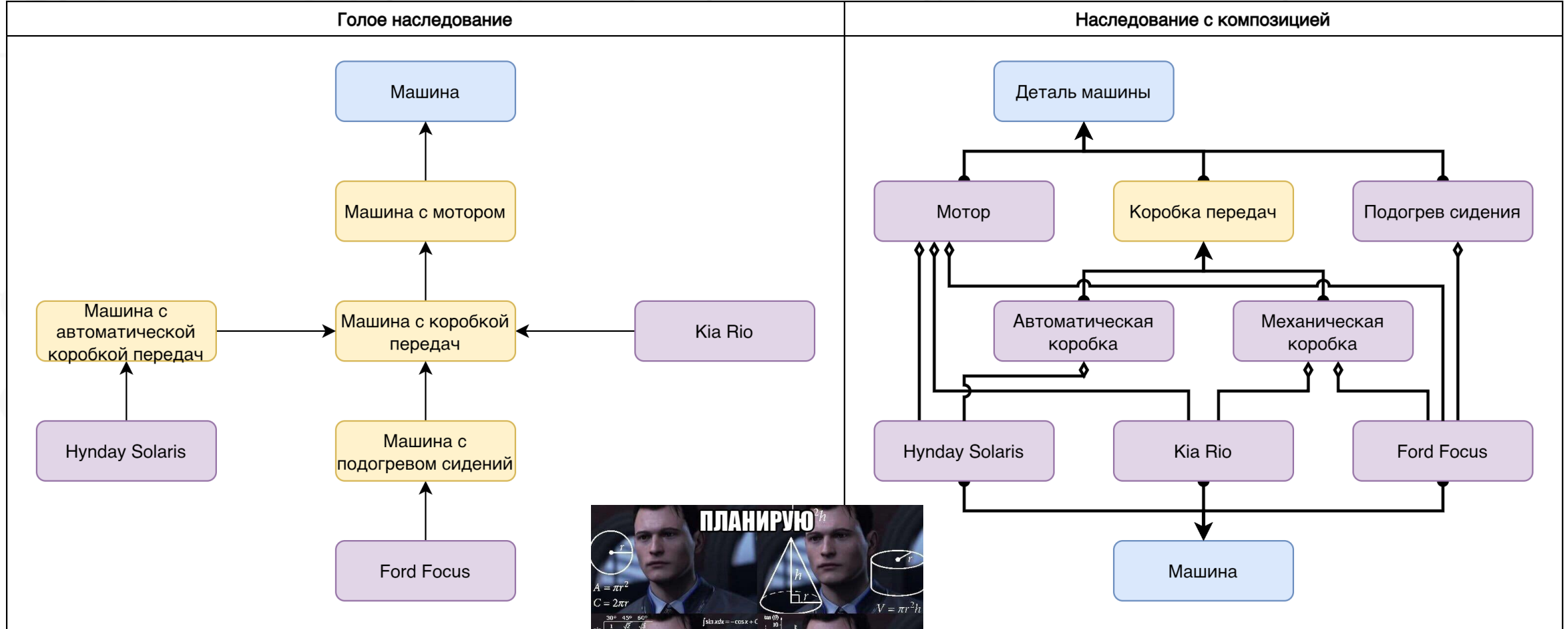
**Наследование это не вопрос повторного использования кода и методов**, наследование это **организация типологической и иерархической связи** между уровнями погружения в детализацию логики вашей программы. Так, если программа будет описывать биологические виды, наследование будет вестись по эволюционному дереву. В этом случае у нас будет появляться множество моментов дублирования кода одних и тех же методов, но будет сохраняться логика близости видов. Поэтому:

- Если мы просто хотим избавиться от дублирования алгоритма, мы вынесем его в отдельную функцию
- Если мы хотим логически связать два описываемых объекта как объекты одной иерархии, мы используем наследование от единого интерфейса.

Отношение **наследования не должно пересекать границы между предметными областями**: инструментальной (структуры данных, алгоритмы, сети) и прикладной (бизнес-логика)



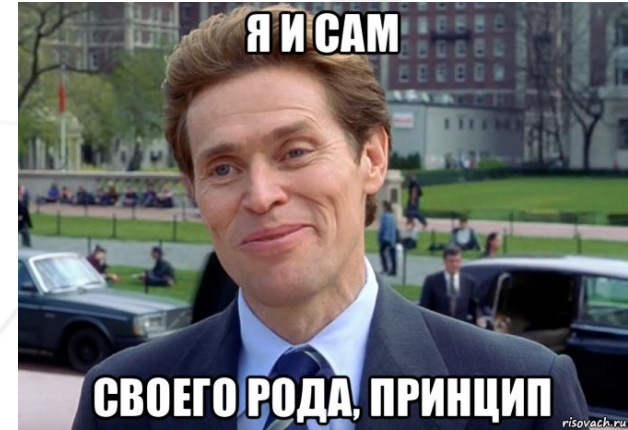
# Композиция вместо наследования



# Чем хуже, тем лучше



Подход к разработке программного обеспечения, объявляющий простоту реализации и простоту интерфейса более важными, чем любые другие свойства системы. Этот стиль описан Ричардом П. Гэбриелом (Richard P. Gabriel) в работе «Lisp: Good News, Bad News, How to Win Big» в разделе «The Rise of 'Worse is Better'» и часто перепечатывается отдельной статьёй.



Гэбриел описывает подход так:

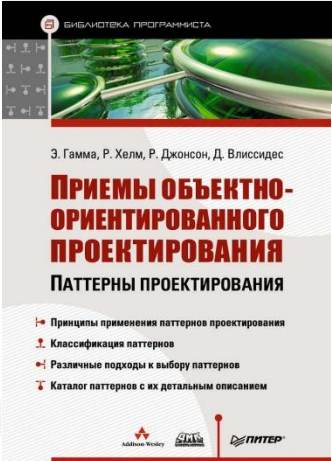

- **Простота:** реализация и интерфейс должны быть простыми. Простота реализации даже несколько важнее простоты интерфейса. Простота — самое важное требование при выборе дизайна.
- **Правильность:** дизайн должен быть правильным во всех видимых проявлениях. Простой дизайн немного лучше, чем правильный.
- **Логичность** (последовательность): дизайн не должен быть слишком нелогичным. Иногда можно пожертвовать логичностью ради простоты, но лучше отказаться от тех частей дизайна, которые полезны лишь в редких случаях, чем усложнить реализацию или пожертвовать логичностью.
- **Полнота:** дизайн должен охватывать как можно больше важных ситуаций. Полнотой можно жертвовать в пользу остальных качеств и обязательно нужно жертвовать, если она мешает простоте. Логичностью можно жертвовать в пользу полноты, если сохраняется простота (особенно бесполезна логичность интерфейса).

Гэбриел считает язык C и систему Unix примерами такого подхода.

# Шаблоны проектирования



Повторяемая кодовая конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Порождающие шаблоны	Структурные шаблоны	Поведенческие шаблоны
Абстрактная фабрика	Адаптер	Цепочка обязанностей
Строитель	Мост	Команда
Фабричный метод	Компоновщик	Интерпретатор
Прототип	Декоратор	Итератор
Одиночка	Фасад	Посредник
 	Приспособленец	Хранитель
	Заместитель	Наблюдатель
		Состояние
		Стратегия
		Шаблонный метод
		Посетитель



# Архитектурные шаблоны



Смысл архитектурного шаблона, аналогичен кодовым шаблонам проектирования, но находящихся на уровне построения всей программы в целом, а не решения отдельной группы задач.

- Многоуровневый шаблон – организация программы в слои: представление(интерфейса), приложение, предметной области, хранения данных.
- Клиент серверный шаблон – организация программы на две зависимые друг от друга части, где одна предоставляет данные и обрабатывает их, а вторая их демонстрирует и отправляет обратно их изменения.
- Ведущий ведомый – организация программы на основную или управляющую часть и множество управляемых.
- Каналы и фильтры – организация программы на
- Посредник – организация программы на разделенные и независимые части общающиеся через промежуточную прослойку, которая управляет доставкой их взаимодействия.
- Шина событий – организация программы на разделенные и зависимые части, которые выстраиваются в последовательность запуска через общую шину передачи событий.
- Модель-представление-контроллер – организация программы на 3 взаимодействующие части: модель для хранения данных, представления для их отображения, контроллер для их преобразования от модели к представлению и обратно.





**КОНЧЕНО**