Композиция или наследование: как выбрать?

<https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose>

* [Анализ и проектирование систем](https://habr.com/hub/analysis_design/),
* [ООП](https://habr.com/hub/oop/),
* [Программирование](https://habr.com/hub/programming/),
* [Проектирование и рефакторинг](https://habr.com/hub/refactoring/),
* [Совершенный код](https://habr.com/hub/complete_code/)
* Перевод

В начале...

… не было ни композиции, ни наследования, только код.

И был код неповоротливым, повторяющимся, нераздельным, несчастным, избыточным и измученным.

Основным инструментом для повторного использования кода была копипаста. Процедуры и функции были редкостью, подозрительными новомодными штучками. Вызов процедур был дорогим удовольствием. Части кода, отделенные от основной логики, вызывали недоумение!

Мрачные были времена.

Но вот лучик ООП воссиял над миром… Правда, несколько десятилетий[1](https://habr.com/post/325478/#1) никто этого не замечал. Покуда не появился графический интерфейс[2](https://habr.com/post/325478/#2), которому, как выяснилось, очень-очень не хватало ООП. Когда нажимаешь на кнопку в окне, что может быть проще, чем отправить кнопке (или ее представителю) сообщение "Нажатие"[3](https://habr.com/post/325478/#3) и получить результат?

И вот тут ООП взлетел. Было написано множество[4](https://habr.com/post/325478/#4) книг, расплодились бесчисленные[5](https://habr.com/post/325478/#5) статьи. Так что сегодня-то каждый может в объектно-ориентированное программирование, так?

Увы, код (и интернет) говорит, что не так

Самые жаркие споры и наибольшее непонимание, похоже, вызывает выбор между композицией и наследованием, зачастую выраженный мантрой "[предпочитайте композицию наследованию](https://en.wikipedia.org/wiki/Composition_over_inheritance)". Вот об этом и поговорим.

Когда мантры вредят

В житейском плане "предпочитать композицию наследованию" в целом нормально, хоть [я и не любитель мантр](http://programmers.stackexchange.com/a/65209/906). Несмотря на то, что они зачастую и несут зерно истины, слишком легко поддаться соблазну и бездумно следовать лозунгу, не понимая, что за ним скрывается. А это всегда выходит боком.

Желтушные статьи с заголовками вроде "Наследование — зло"[6](https://habr.com/post/325478/#6) тоже не по мне, особенно если автор пытается обосновать свои набросы, сначала неправильно применяя наследование, а потом делая вывод, что оно во всем виновато. Ну типа "молотки — отстой, потому что ими нельзя завинтить шуруп."

Начнем с основ.

Определения

Далее в статье я буду понимать под ООП "классический" объектный язык, который поддерживает классы со свойствами, методами и простое (одиночное) наследование. Никаких вам интерфейсов, примесей, аспектов, множественного наследования, делегатов, замыканий, лямбд, — ничего, кроме самых простых вещей:

* Класс: именованная сущность из предметной области, возможно, имеющая предка (суперкласс), определенная как набор полей и методов.
* Поле: именованное свойство с определенным типом, которое может, в частности, ссылаться на другой объект (см. композиция).
* Метод: именованная функция или процедура, с параметрами или без них, реализующая какое-то поведение класса.
* Наследование: класс может унаследовать — использовать по умолчанию — поля и методы своего предка. Наследование транзитивно: класс может наследоваться от другого класса, который наследуется от третьего, и так далее вплоть до базового класса (обычно — Object), возможно, неявного. Наследник может переопределить какие-то методы и поля чтобы изменить поведение по умолчанию.
* Композиция: если поле у нас имеет тип Класс, оно может содержать ссылку на другой объект этого класса, создавая таким образом связь между двумя объектами. Не влезая в дебри различий между простой ассоциацией, агрегированием и композицией, давайте "на пальцах" определим: композиция — это когда один объект предоставляет другому свою функциональность частично или полностью.
* Инкапсуляция: мы обращаемся с объектами как с единой сущностью, а не как с набором отдельных полей и методов, тем самым скрываем и защищаем реализацию класса. Если клиентский код не знает ничего, кроме публичного интерфейса, он не может зависеть от деталей реализации.

Наследование фундаментально

Наследование — это *фундаментальное* понятие ООП. В языке программирования могут быть объекты и сообщения, но без наследования он не будет объектно-ориентированным (только основанным на объектах, но все еще полиморфным).

… как и композиция

Композиция это тоже фундаментальное свойство, причем любого языка. Даже если язык не поддерживает композицию (что редкость в наши дни), люди все равно будут мыслить категориями частей и компонентов. Без композиции было бы невозможно решить сложные задачи по частям.

(Инкапсуляция тоже вещь фундаментальная, но сейчас речь не о ней)

Так от чего весь сыр-бор?

Ну хорошо, и композиция, и наследование фундаментальны, в чем дело-то?

А дело в том, что можно подумать, что одно всегда может заменить другое, или что первое лучше или хуже второго. Разработка ПО — это всегда выбор разумного баланса, компромисс.

С композицией все более-менее просто, мы с ней постоянно сталкиваемся в жизни: у стула есть ножки, стена состоит из кирпичей и цемента и тому подобное. А вот наследование, несмотря на свое простое определение, может все усложнить и запутать, если хорошенько не поразмыслить над тем, как его применять. Наследование это весьма абстрактная штука, о нем можно рассуждать, но так просто его не потрогаешь. Мы, конечно, можем сымитировать наследование, используя композицию, но это, как правило, слишком много возни. Для чего нужна композиция — очевидно: из частей собрать целое. А вот с наследованием сложнее, потому что оно сразу о двух вещах: о смысле и о механике.

Наследование смысловое

Как в биологии классификация таксонов организует их в иерархии, так наследование отражает иерархию понятий из предметной области. Упорядочивает их от общего к частному, собирает родственные идеи в ветви иерархического древа. Смысл (семантика) класса по большей части выражен в его интерфейсе — наборе сообщений, которые класс способен понять, но также определяется и теми сообщениями, которыми класс отвечает. Унаследовался от предка — будь добр не только понять все сообщения, которые мог понять предок, но также и уметь ответить как он (сохранить поведение предка — прим. пер.) И поэтому наследование связывает наследника с предком гораздо сильнее, чем если бы мы взяли просто экземпляр предка как компонент. Обратите внимание, даже если класс делает что-то совсем простое, почти не имеет логики, его имя несет существенную смысловую нагрузку, разработчик делает из него важные выводы о предметной области.

Наследование механическое

Говоря о наследовании в механическом плане, мы имеем в виду, что наследование берет данные (поля) и поведение (методы) базового класса и позволяет использовать их повторно или же дополнить в наследниках. С точки зрения механики, если потомок унаследует реализацию (код) предка, то неизбежно получит и его интерфейс.

Я уверен, что в недопонимании виновата именно эта двойственная природа наследования[7](https://habr.com/post/325478/#7) в большинстве ОО-языков. Многие считают, что наследование — это чтобы повторно использовать код, хотя оно не только для этого. Если придавать повторному использованию чрезмерное значение — жди беды в архитектуре. Вот пара примеров.

Как не надо наследовать. Пример 1

class Stack extends ArrayList {

public void push(Object value) { … }

public Object pop() { … }

}

Казалось бы, класс Stack, все хорошо. Но посмотрите внимательно на его интерфейс. Что должно быть в классе с именем Stack? Методы push() и pop(), что же еще. А у нас? У нас есть get(), set(), add(), remove(), clear() и еще куча барахла, доставшегося от ArrayList, которое стеку ну вообще не нужно.

Можно было бы переопределить все нежелательные методы, а некоторые (например, clear()) даже и адаптировать под наши нужды, но не многовато ли работы из-за одной ошибки в дизайне? На самом деле трех: одной смысловой, одной механической и одной комбинированной:

1. Утверждение "Stack это ArrayList" ложно. Stack [не является подтипом](https://en.wikipedia.org/wiki/Liskov_substitution_principle) ArrayList. Задача стека — обеспечить выполнение правила LIFO (последним пришел, первым ушел), которое легко удовлетворяется интерфейсом push/pop, но никак не соблюдается интерфейсом ArrayList.
2. Механически наследование от ArrayList нарушает инкапсуляцию. Клиентскому коду не должно быть известно, что мы решили использовать ArrayList для хранения элементов стека.
3. Ну и наконец, реализуя стек через ArrayList мы смешиваем две разные предметные области: ArrayList — это коллекция с произвольным доступом, а стек — это понятие из мира очередей, со строго ограниченным (а не произвольным)[8](https://habr.com/post/325478/#8) доступом.

Последний пункт — незначительная на первый взгляд, но важная вещь. Посмотрим на нее пристальнее.

Как не надо наследовать. Пример 2

Частая ошибка при наследовании — это создать модель из предметной области, унаследовав ее от готовой реализации. Вот, скажем, нам надо выделить некоторых наших клиентов (класс Customer) в определенное подмножество. Легко! Наследуемся от ArrayList<Customer>, называем это CustomerGroup и понеслась.

Не тут-то было. Поступив так мы опять спутаем две предметные области. Старайтесь избегать этого:

1. ArrayList<Customer> это уже наследник списка, утилиты типа "коллекция", готовой реализации.
2. CustomerGroup это совсем другая штука — класс из предметной области (домена).
3. Классы из предметной области должны использовать реализации, а не наследовать их.

Слой предметной области не должен знать, как у нас там все внутри сделано. Рассуждая о том, что делает наша программа, мы оперируем понятиями из предметной области, и мы не хотим отвлекаться на нюансы внутреннего устройства. Если видеть в наследовании только инструмент повторного использования кода, мы раз за разом будем попадаться в эту ловушку.

Дело не в одиночном наследовании

Одиночное наследование пока остается самой популярной моделью ООП. Оно неизбежно влечет наследование реализации, которое приводит к сильному зацеплению (coupling — прим. пер.) между классами. Может показаться, что беда в том, что ветка наследования у нас только одна на обе потребности: и смысловую и механическую. Если использовали для одного, то для другого уже нельзя. А раз так, может быть множественное наследование все исправит?

*Нет.* Отношение наследования не должно пересекать границы между предметными областями: инструментальной (структуры данных, алгоритмы, сети) и прикладной (бизнес-логика). Если CustomerGroup будет наследовать ArrayList<Customer> и одновременно, скажем, DemographicSegment, то две предметные области переплетутся между собой, а "видовая принадлежность" объектов станет неочевидна.

Предпочтительно (по крайней мере, с моей точки зрения) делать так. Наследуемся от имеющихся в языке инструментальных классов по минимуму, ровно настолько, чтобы реализовать "механическую" часть вашей логики. Потом соединяем получившиеся части композицией, но не наследованием. Иными словами:

*От инструментов можно наследовать только другие инструменты.*

Это очень частая ошибка новичков. Что не удивительно, ведь так просто взять и унаследовать. Редко где встретишь обсуждения, почему именно это неправильно. Еще раз: бизнес-сущности должны пользоваться инструментами, а не быть ими. Мухи (инструменты) — отдельно, котлеты (бизнес-модели) — отдельно.

Так когда же нужно наследование?

Наследуемся как надо

Чаще всего — и при этом с наибольшей отдачей — наследование применяют для описания объектов, незначительно отличающихся друг от друга (в оригинале используется термин "differential programming" — прим. пер.) Например, нам нужна особенная кнопка с небольшими дополнениями. Нормально, наследуемся от существующего класса Кнопка. Потому что наш новый класс, это все еще кнопка, а мы полностью наследуем API класса Кнопка, его поведение и реализацию. Новая функциональность только *добавляется* к существующему. А вот если в наследнике часть функциональности убирается, это повод задуматься, а нужно ли наследование.

Наследование полезнее всего для группировки сходных сущностей и понятий, определения семейств классов, и вообще для организации терминов и понятий, описывающих предметную область. Зачастую, когда значительная часть предметной логики уже реализована, исходно выбранные иерархии наследования перестают работать. Если всё к тому идет, не бойтесь разобрать и заново сложить эти иерархии[9](https://habr.com/post/325478/#9) так, чтобы они лучше соответствовали и работали друг с другом.

Композиция или наследование: что выбрать?

В ситуации, когда вроде бы подходит и то и другое, взгляните на дизайн в двух плоскостях:

1. Структура и механическое исполнение бизнес-объектов.
2. Что они обозначают по смыслу и как взаимодействуют.

Пока наследование остается внутри одной плоскости, все нормально. Но если иерархия проходит через две плоскости сразу, это плохой симптом.

Например, у вас есть один объект внутри другого. Внутренний объект реализует значительную часть поведения внешнего. У внешнего объекта куча прокси-методов, которые тупо пробрасывают параметры во внутренний объект и возвращают от него результат. В этом случае посмотрите, а не стоит ли унаследоваться от внутреннего объекта, хотя бы частично.

Разумеется, никакие инструкции не заменят голову на плечах. Когда строишь объектную модель, вообще полезно думать. Но если вам хочется конкретных правил, то пожалуйста.

Наследуем, если:

1. Оба класса из одной предметной области
2. Наследник является корректным подтипом (в терминах [LSP](https://en.wikipedia.org/wiki/Liskov_substitution_principle) — прим. пер.) предка
3. Код предка необходим либо хорошо подходит для наследника
4. Наследник в основном добавляет логику

Иногда все эти условия выполняются одновременно:

* в случае моделирования высокоуровневой логики из предметной области
* при разработке библиотек и расширений для них
* при дифференциальном программировании (автор снова использует термин "differential programming", очевидно, понимая под ним нечто, отличное от [DDP](https://en.wikipedia.org/wiki/Differential_dynamic_programming) — прим. пер.)

Если это не ваш случай, то и наследование вам, скорее всего, будет нужно не часто. Но не потому, что надо "предпочитать" композицию наследованию, и не потому что она "лучше". Выбирайте то, что подходит наилучшим образом для конкретно вашей задачи.

Надеюсь, эти правила помогут вам понять разницу между двумя подходами.

Приятного кодинга!