

- ? SRP: Single Responsible Principle
- ? OCP: Open–Closed Principle
- ? LSP: Liskov Substitution Principle
- ? ISP: Interface Segregation Principle
- ? DIP: Dependency Inversion Principle





*Преждевременное выделение аспектов поведения в отдельные абстракции приводит к переусложнению кода.*

😬 Начальник поставил задачу

😊 Надо изменить аспект поведения

*Кто такой этот актер?*

🗨️ Модуль должен отвечать перед одним и только одним актором

*Не совместимо с переиспользованием кода,  
так как в разных местах использования в  
любой момент могут потребоваться  
разные изменения.*

😬 Начальник поставил задачу

😊 Нужно иное поведение в месте использования

*Что такое обязанность?*

💡 Объект должен брать на себя лишь одну обязанность

*Декомпозировать обязанность можно квазибесконечно, но принцип не даёт правила, когда нужно остановиться.*

- ▼ Передать данные со входа на выход
- ▼ IO, JSON, Logging
- ▼ Parse, Transform, Serialize
- ▼ Tokenize, AST, Validate, Parse



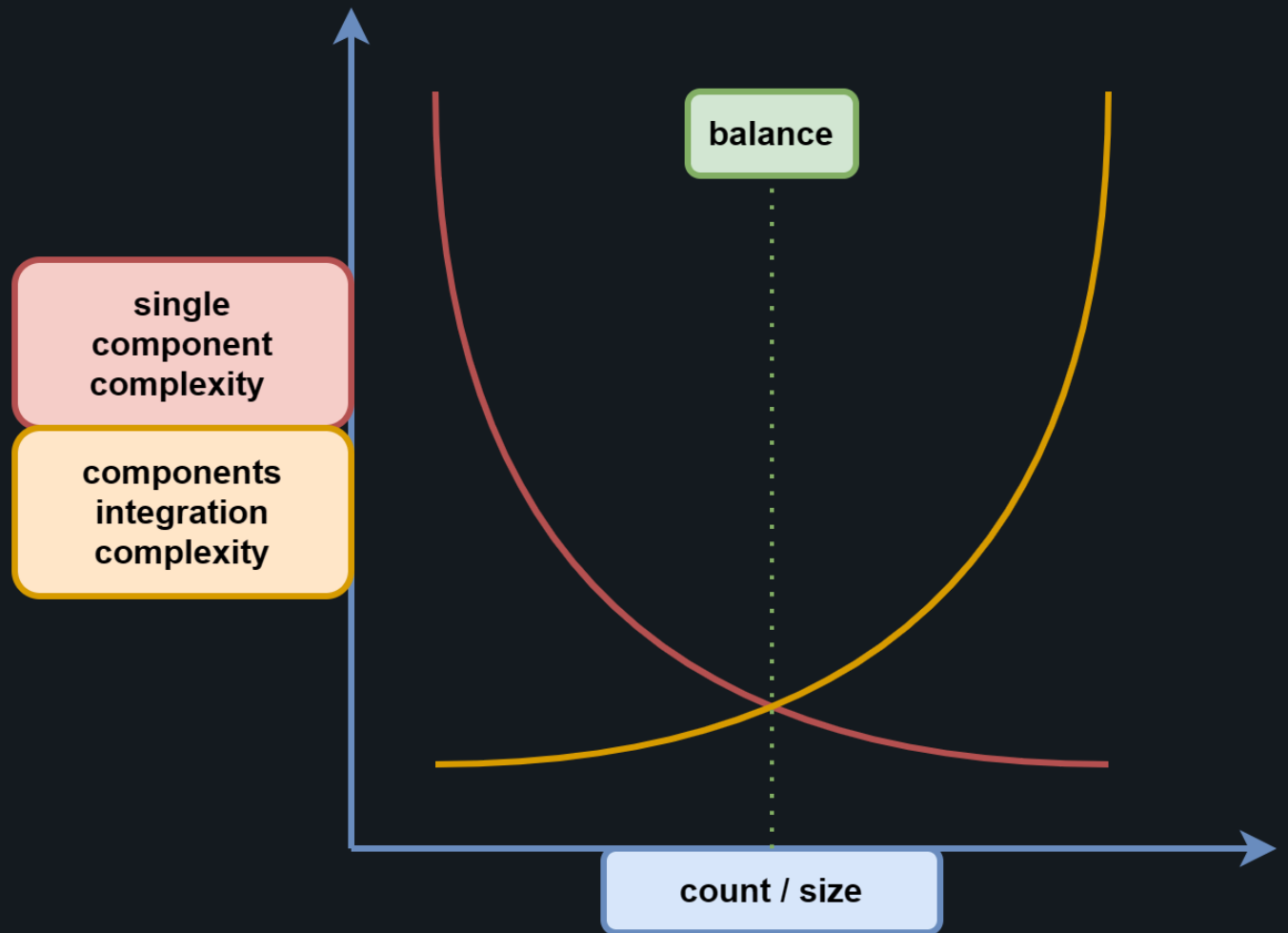
*Несколько размытых формулировок, не дающих понимания как правильно декомпозировать код. Без доказательств декларируется польза, но на деле несут только вред.*

- ✗ Три определения по цене одного
- ✗ Нет ясности в определениях
- ✗ Нет обоснования преимуществ
- ✗ Форсируются сомнительные практики

*Рекурсивное разделение большой задачи на слабо связанные друг с другом задачи поменьше, что позволяет работать над подзадачами, не думая о всей задаче.*


💡 Разделяй и властвуй

Уменьшение компонент приложения, приводит к увеличению их числа, а значит и усложнению их коммуникации. В какой-то момент упрощение от декомпозиции не перекрывает усложнения.

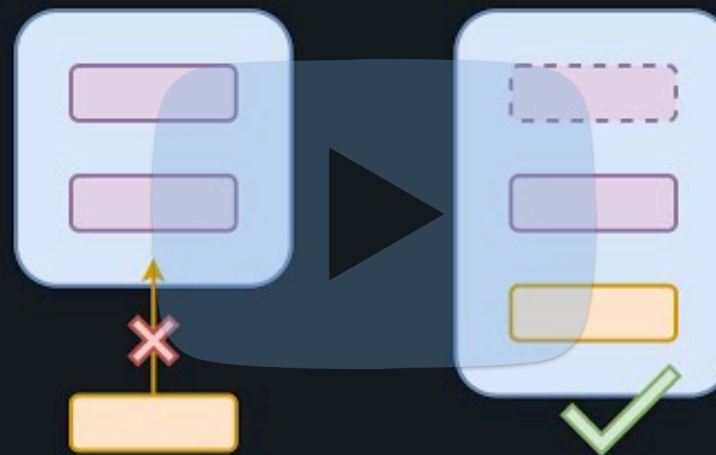


✗ SRP

✓ Умеренная декомпозиция

 SRP: глубокое погружение / AlephTav

Путаница с расширениями



[https://page.hyoo.ru/#! = qk8sq7\\_c388qt/#slide=4](https://page.hyoo.ru/#! = qk8sq7_c388qt/#slide=4)

В далёком 1988 году Бертран Мейер сформулировал свой принцип написания кода долгоживущих проектов под названием "Принцип открытости/закрытости" или ОСР.

Вкратце, он звучит так: "программные сущности должны быть открыты для расширения, но закрыты для изменения". И, как любой короткий принцип, он требует десятки статей для толкования. Но к чёрту всю воду, включаем нашу соковыжималку!

W Open–Closed Principle

Принцип ОСР может быть применён к разным типам сущностей, описываемых нашим кодом..

По мере развития проекта, требования к нему меняются, обнаруживаются дефекты, придумываются улучшения. Короче, появляется необходимость внесения изменений.

▲ Функции

▲ Объекты

▲ Классы

▲ Интерфейсы

▲ Модули

▲ Пакеты



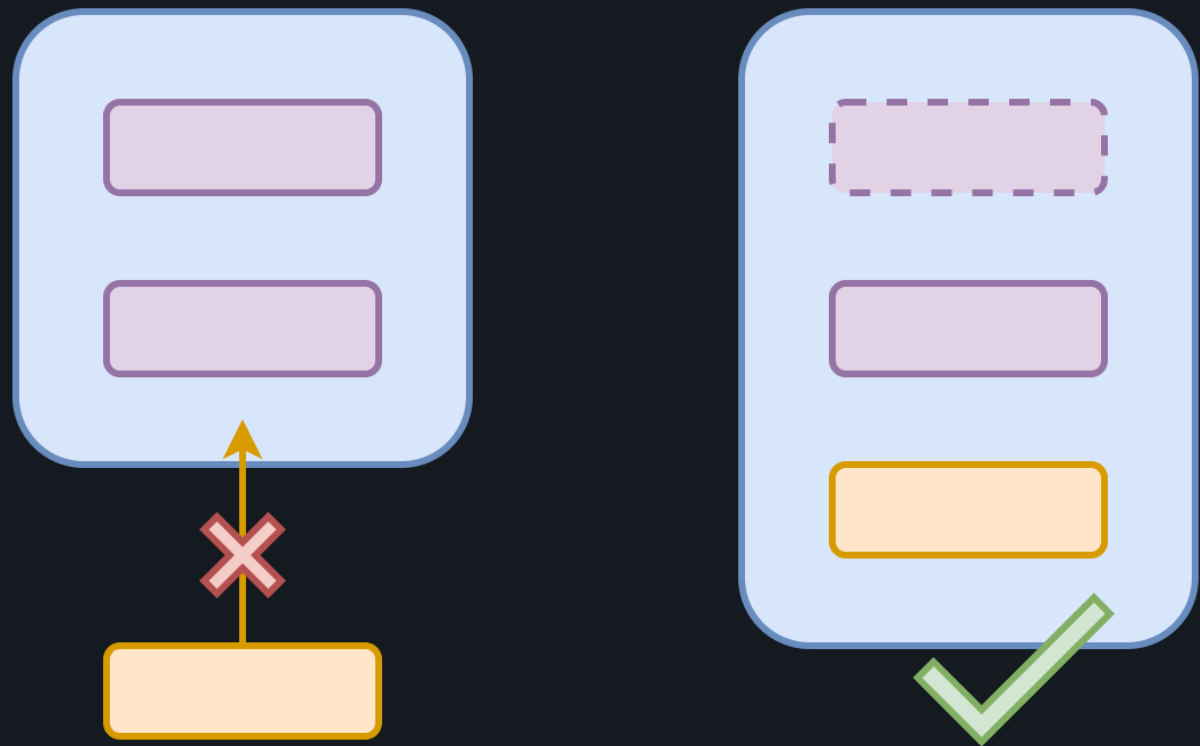
Простыми словами ОСР можно объяснить так: Работает? Не трогай! Создай новую сущность. Ну ладно, баги чинить можно. Но не более!

При этом, Бертран Мейер допускал использование наследования для снижения копипаста, но это неизбежно приводит к сложным и порой абсурдным иерархиям классов. А вот популяризовавший ОСР Роберт Мартин уже предлагает выделять абстракции заранее так, чтобы наследование реализации нам не требовалось. Ретроспективно это сделать, конечно, легко, но чтобы заранее ввести все абстракции на будущее, нужно выдающееся чутьё, если не сказать большего.

💡 Копипаст лучше рефакторинга!

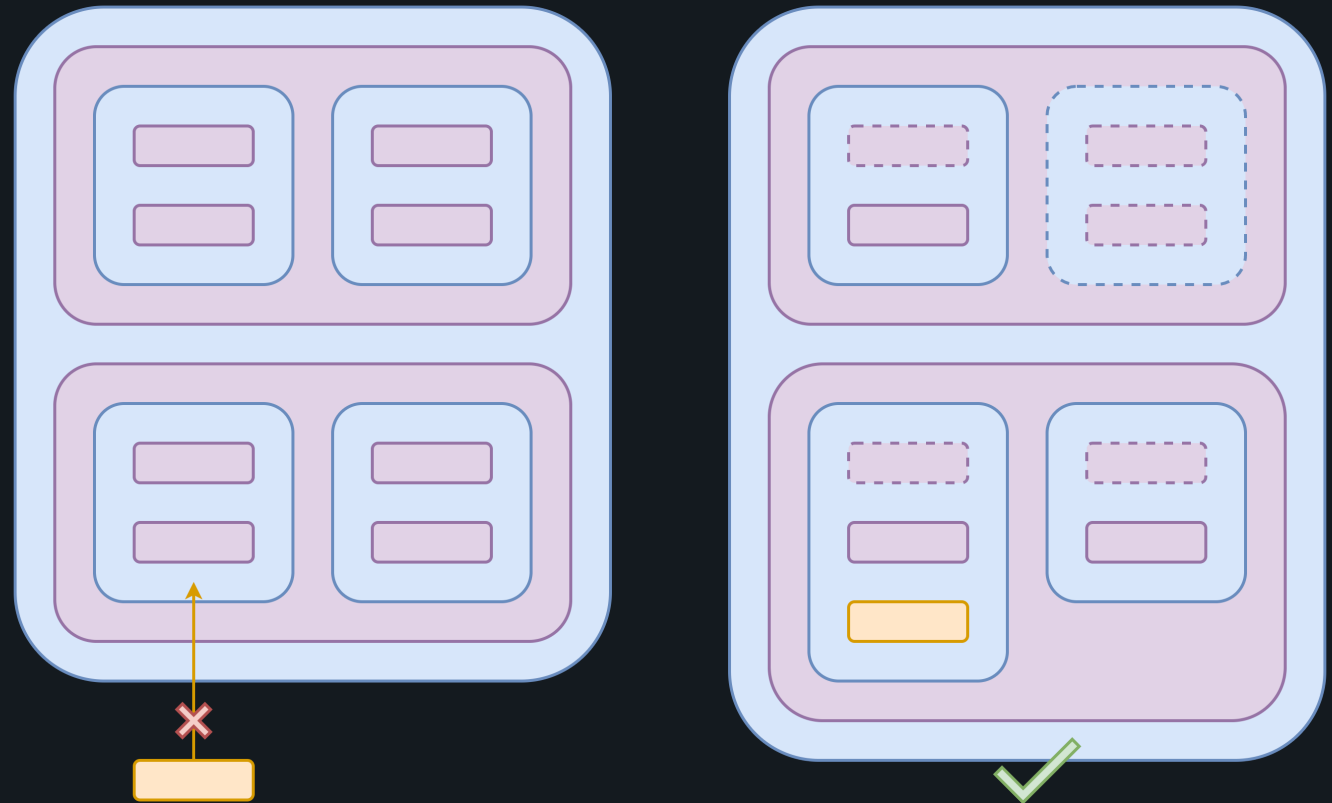
Стоит отдельно подчеркнуть неоднозначность формулировки. Может показаться, что обе половины принципа говорят об одном и том же предмете. Однако, подразумевают они на самом деле разные. Термин *изменение* относится к отдельной сущности - её нельзя *менять*. А вот термин *расширение* относится уже ко всему множеству сущностей - его можно лишь *расширять* добавлением новых сущностей.

Пунктиром на диаграмме обозначены реализации которые тем или иным способом могут быть унаследованы. Но это возможно далеко не всегда.



И тут у нас начинаются скользкие вопросы. Возьмём, такую сущность как функция. Добавить в неё новый параметр в соответствии с принципом ОСП нельзя, а надо создать новую функцию. Но если эта функция не в воздухе висит, а объявлена в рамках такой сущности как класс, то его тоже менять нельзя, и надо создать новый класс. А класс лежит в неймспейсе, который находится в модуле, который собирается в пакет. А пакет в яйце, яйцо в утке, утка в зайце, заяц в шоке.

Как далеко мы зайдём в этом расширении, стараясь ничего не менять?



Если мы изменим уже существующую, отлаженную, используемую кем-то сущность, то можем намеренно или случайно внести в неё несовместимые изменения. Это может привести к некорректной работе зависящих от изменённой сущностей, падению сборки проекта, необходимости каскадного внесения изменений в косвенно зависящие сущности, в том числе и во внешние потребители, доступа к которым у нас может и не быть.

Проблема ли это? Если в проекте не используется тестирование, статическая типизация и прочие практики контроля качества, то любое изменение - это мина, на которую мы прыгаем, с разбега, рыбкой. Внесение изменения - 5 минут, отладка его - 5 дней. Во времена Мейера это была проблема оного!

- ✗ Ошибки в поведении
- ✗ Ошибки при сборке
- ✗ Несовместимость контрактов

Однако, совсем другое дело, когда у нас есть современные практики контроля качества. Тогда мы не боимся вносить любые изменения, ведь уверены, что обо всех несовместимостях и дефектах мы довольно быстро узнаем благодаря автоматизированным инструментам: автотесты, авторефакторинги, автомиграции, тайпчекеры, линтеры, непрерывная интеграция и тд.

Получается, что серьёзные проблемы у нас могут быть с изменением лишь тех сущностей, на которые завязаны внешние потребители, для которых нам необходимо сохранять обратную совместимость.

- ✓ Тесты
- ✓ Миграции
- ✓ Статический анализ
- ✓ Непрерывная интеграция

А вот для внутренних, контролируемых нами, потребителей проблемы возникают, наоборот, если код не менять, ведь мы плодим горы похожего, но отличающегося в мелочах, кода. Его нужно поддерживать. Он потребляет больше тех или иных ресурсов. Короче, если в коде не прибираться, то он очень быстро тухнет.

А самое печальное, что существующие закрытые к изменению сущности, зачастую оказываются несоответствующими актуальным требованиям. Так что их использование становится мало того, что бессмысленным или нежелательным, так ещё и откровенно вредным.

Например, если раньше обращение к сущности не требовало авторизации, а потом вы решили, что авторизация таки нужна. Оставлять возможность работать без авторизации было бы очень опрометчиво. Это всё равно, что раскладывать грабли рядом с кроватью: с какой бы ноги ни встал с утра - весь день будешь потирать лобные извилины. Поэтому порой обратная совместимость должна быть сломана. И это нормально.

- ✗ Раздутие множества сущностей
- ✗ Поддержка разных реализаций одной задачи
- ✗ Несовместимость с бизнес требованиями

Стигматизация изменений приводит к специфическому влиянию на архитектуру. Вместо того, чтобы написать простой наивный код, легко понимаемый любым разработчиком, приходится городить кучу абстракций на всякий случай, чтобы потом можно было *изменить* поведение, не меняя кода, а только дописывая новый.

А так как программисты не славятся умением предвидеть будущее, то неизбежно одни точки расширения окажутся невостребованными, а других наоборот будет не хватать, что таки приведёт к вынужденному *изменению* кода.

✓ Гибкость и расширяемость

✗ Преждевременное усложнение

Что ж, давайте попробуем выцепить рациональное зерно из ОСР, и *изменим* его так, чтобы пользы он приносил больше, чем вреда. Итак..

В отличие от предыдущей, наша формулировка ОСР мало того, что разрешает обратно совместимые изменения сущностей, так ещё и не столь строга касательно ломающих изменений.

В своей песочнице ты - царь и бог. Но создавая публичный контракт, важно думать не только о том, как им будут пользоваться, но и о том, как ты сам в дальнейшем будешь развивать проект, сохраняя обратную совместимость. А если уж придётся её ломать, то следует предоставить как минимум средства автоматического выявления несовместимостей, а как максимум - средства автоматического их разрешения.

💡 Не ломай публичный контракт без необходимости



Как было показано ранее, в современных условиях следование OCP скорее вредно, чем полезно. Оно требует от программиста быть провидцем, иначе наказывает его протуханием кодовой базы. Поэтому требуется, если не полное отвержение, то хотя бы пересмотр принципа с использованием менее жёстких ограничений, разрешающих нам поддерживать проект в тонусе регулярными рефакторингами.

✗ OCP

✓ Обратная совместимость

В качестве десерта предлагаю вам статью Даниэля Норта, с критическим разбором принципов SOLID и ОСП в частности, в противовес которому он вводит свой *Принцип Снежного Кома*, по которому любой ваш код рассматривается не как что-то самоценное, а как постоянные затраты, которые всё множатся и преумножаются. Так что если есть возможность изменить код так, чтобы он стал проще, то именно так и стоит поступить. И чем раньше, тем лучше.

 История возникновения CUPID / Daniel Terhorst-North

Если данный разбор показался вам полезным, то дайте мне об этом знать посредством лайка или даже доната. А так же поделитесь ссылкой на него со своими коллегами. Особенно с теми, кто городит огород вокруг ОСР.

Если же вы не согласны с каким-либо тезисом или, наоборот, чувствуете некую недосказанность, и хотите дополнить своими идеями или иными материалами по теме, то жду ваших комментариев.

Наконец, подписывайтесь на канал, чтобы не пропустить дальнейшие разборы. Нам ещё много чего с вами нужно будет обсудить.


На этом пока что всё. С вами был открытый к расширению программист Дмитрий Карловский.


 Лайки

 Поддержка

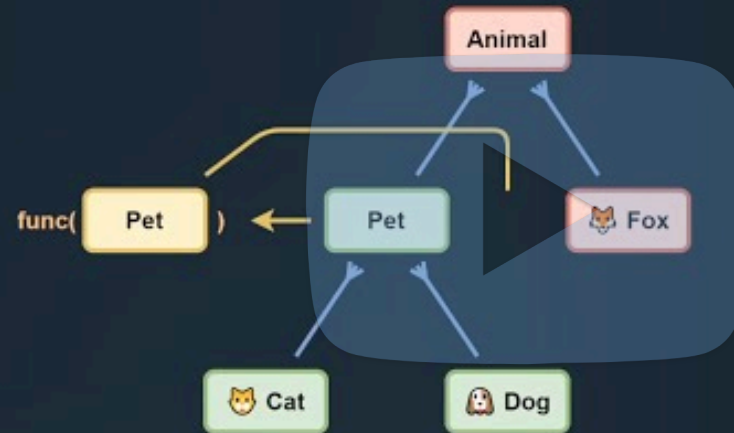
 Комментарии

 Подписка

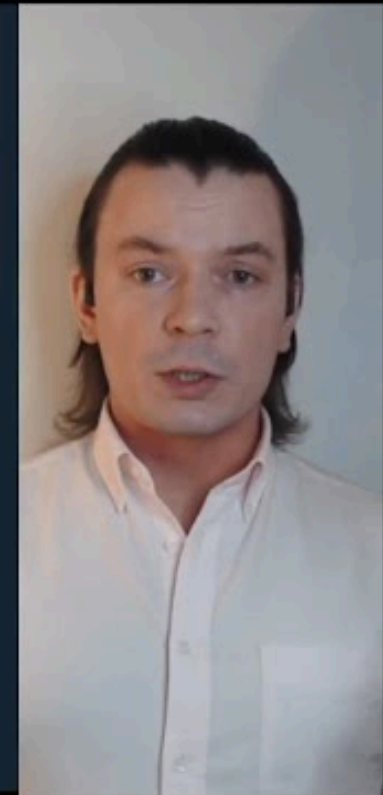
 habr.com

 pikabu.ru

Суть LSP - повсеместная ковариантность



<https://nin-jin.github.io/slides/abbr/#slide=15>



В далёком 1987 году Барбара Лисков сформулировала принцип разработки имени себя.

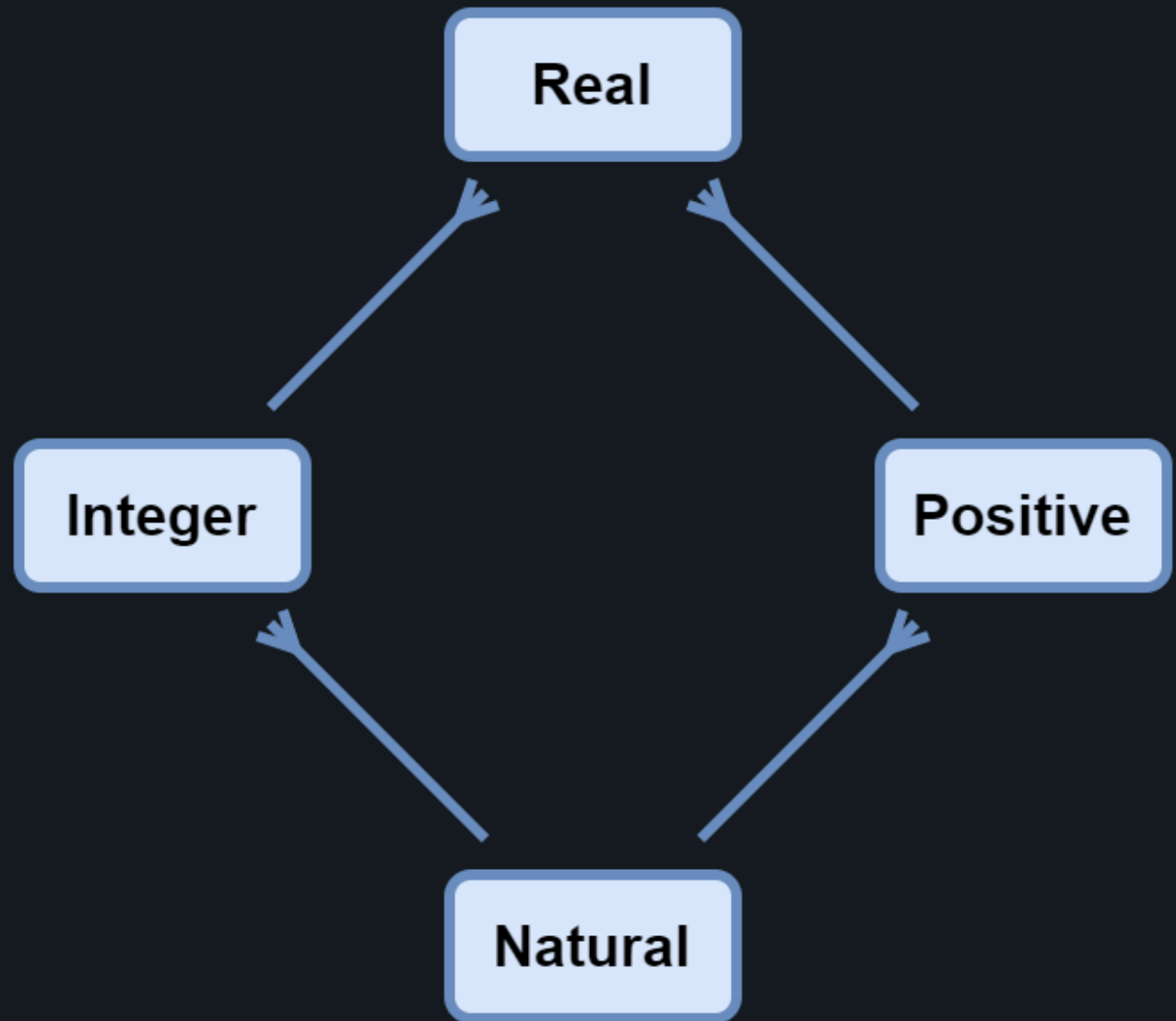
Он позволяет понять правильно вы написали полиморфный код или нет. Но прежде чем его сформулировать нам надо разобраться с некоторыми понятиями, которые входят в определение..

W Liskov Substitution Principle

Все данные в нашей программе принадлежат тому или иному типу. Тип определяет множество возможных значений и их семантику. Один тип может полностью включать в себя другой. В таком случае второй тип является подтипом первого. Таким образом типы могут образовывать иерархию. Рассмотрим пример с числами..

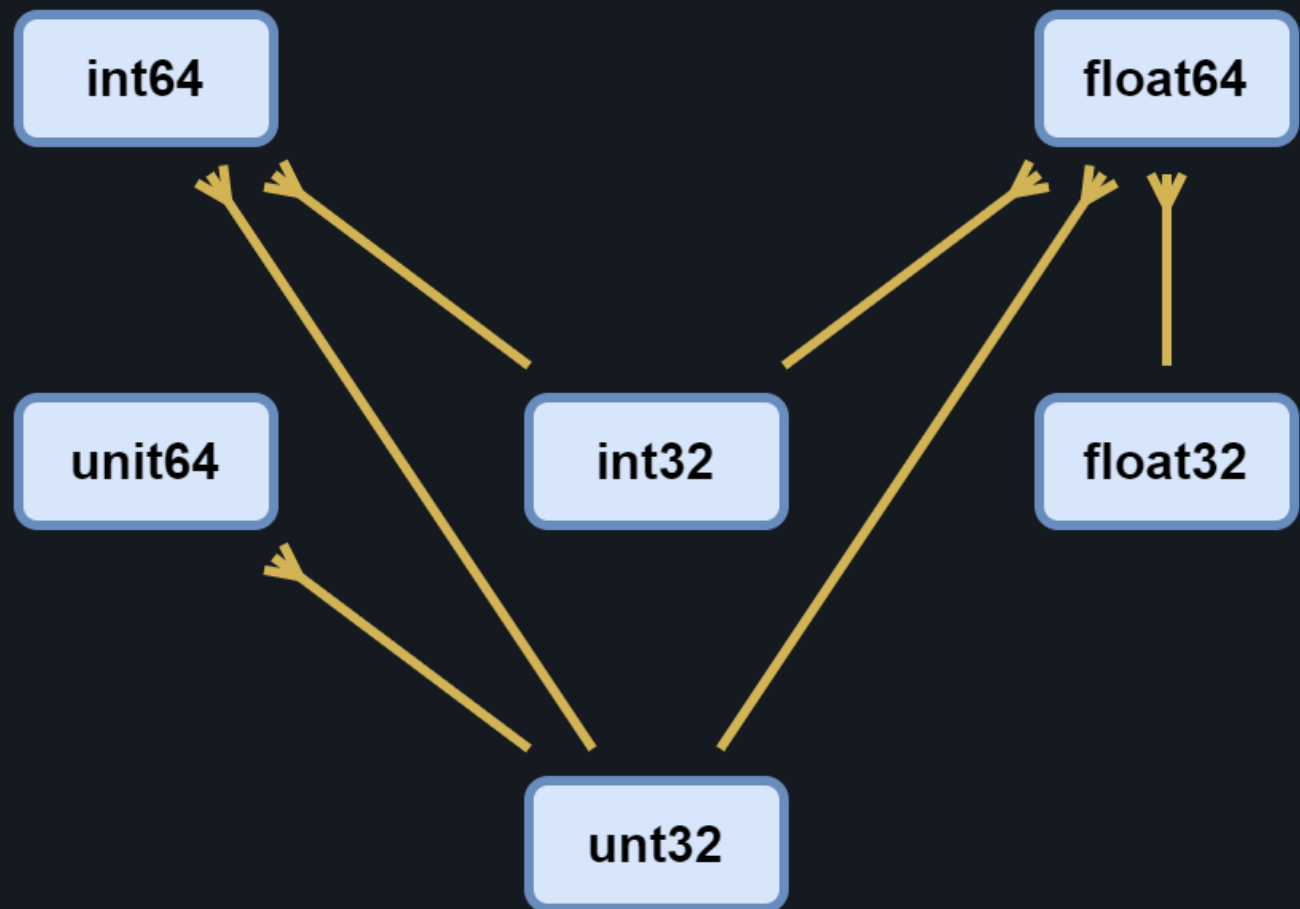
Как тип целых так и тип положительных чисел по отдельности являются частными случаями типа вещественных чисел, а значит являются его подтипами. В то же время целые не включают в себя все положительные. А положительные не включают в себя все целые. Поэтому эти типы не состоят друг с другом в отношении "супертип-подтип". А вот натуральные числа являются одновременно и целыми и положительными, поэтому тип натуральных чисел является подтипом и обоих типов.

Отношение "супертип-подтип" является транзитивным, то есть если один тип является подтипом другого, а другой - третьего, то и первый является подтипом третьего.



Тут правда необходимо иметь ввиду специфику представления значения разных типов в памяти. Например, эквивалентные значения целочисленного и вещественного типов обычно представляются разной последовательностью бит. Так что в некоторых языках эти типы не будут находиться в родственных отношениях. Однако, есть языки, умеющие автоматически преобразовывать значения из одного представления в другое - в них более узкий тип является подтипом более широкого, который позволяет хранить большее число вариантов значений.

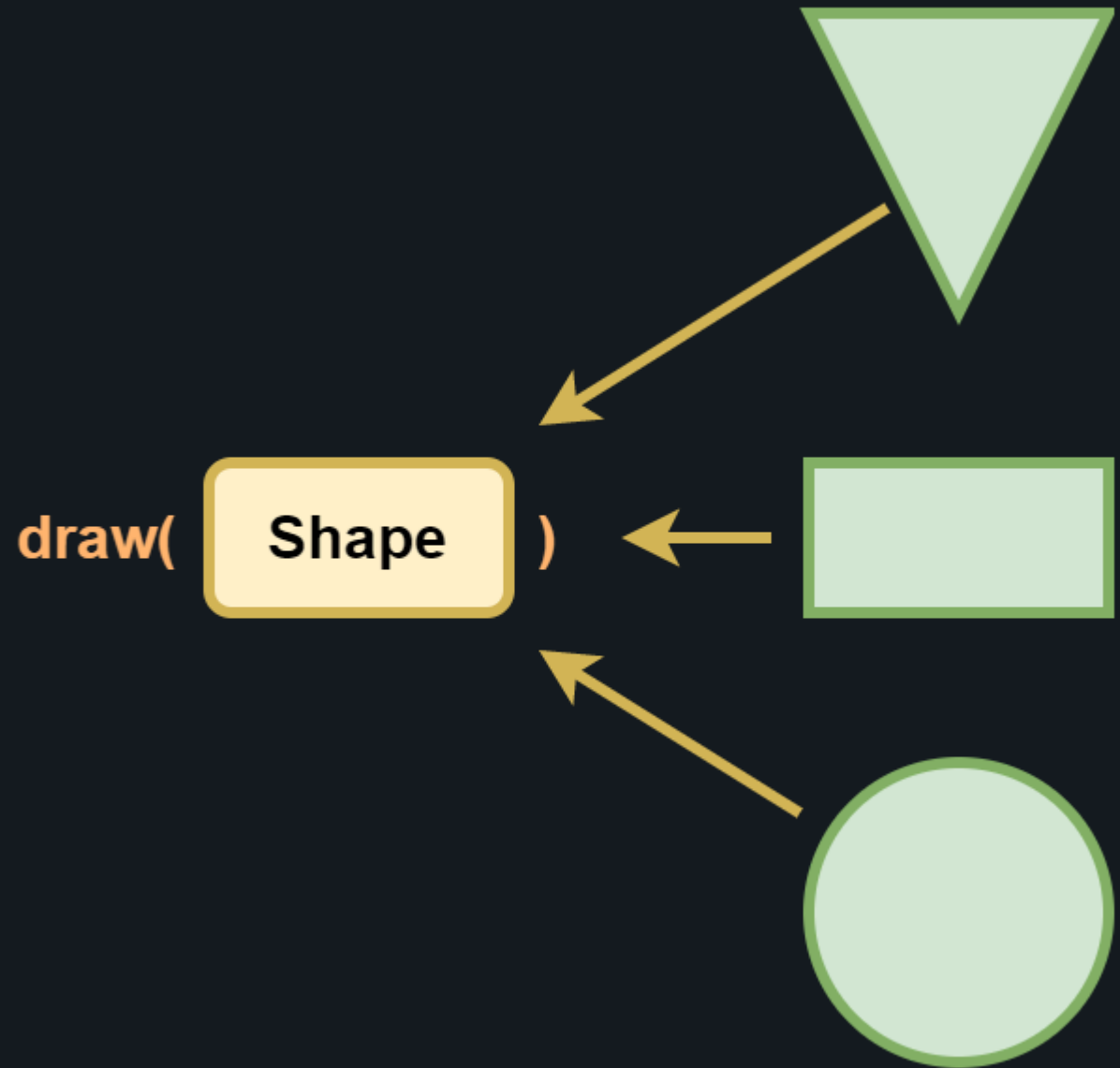
На иллюстрации мы видим систему числовых типов в слабо типизированном языке, который умеет неявно производить преобразование типа, если это не приводит к потере информации.





Полиморфизм - это способность одного и того же кода работать с аргументами разных типов.

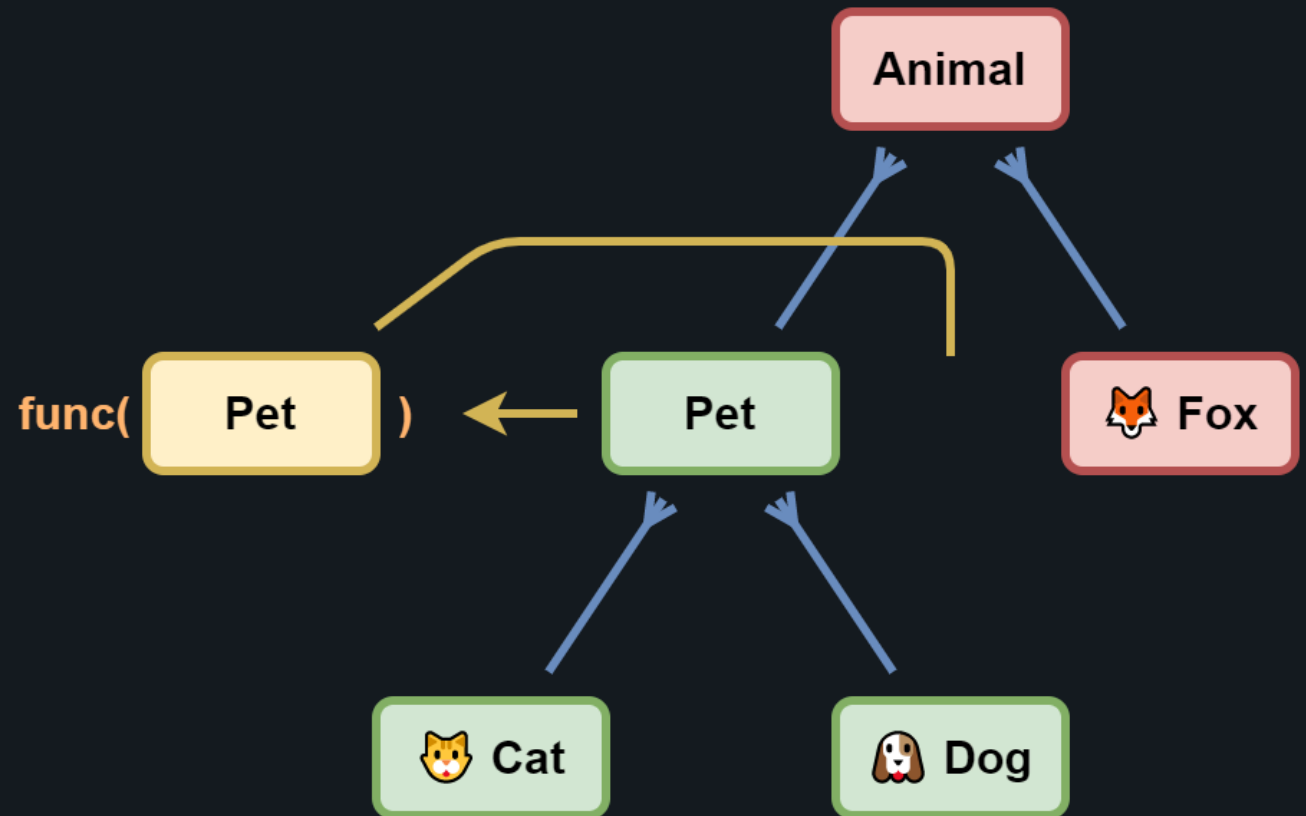
В данном примере, процедура draw у нас принимает на вход произвольную фигуру. И какую бы фигуру мы ей ни передали - процедура её всё равно нарисует.



Наконец, теперь мы можем сформулировать принцип подстановки Барбары Лисков: "любые функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом и не нарушая желаемых свойств программы".

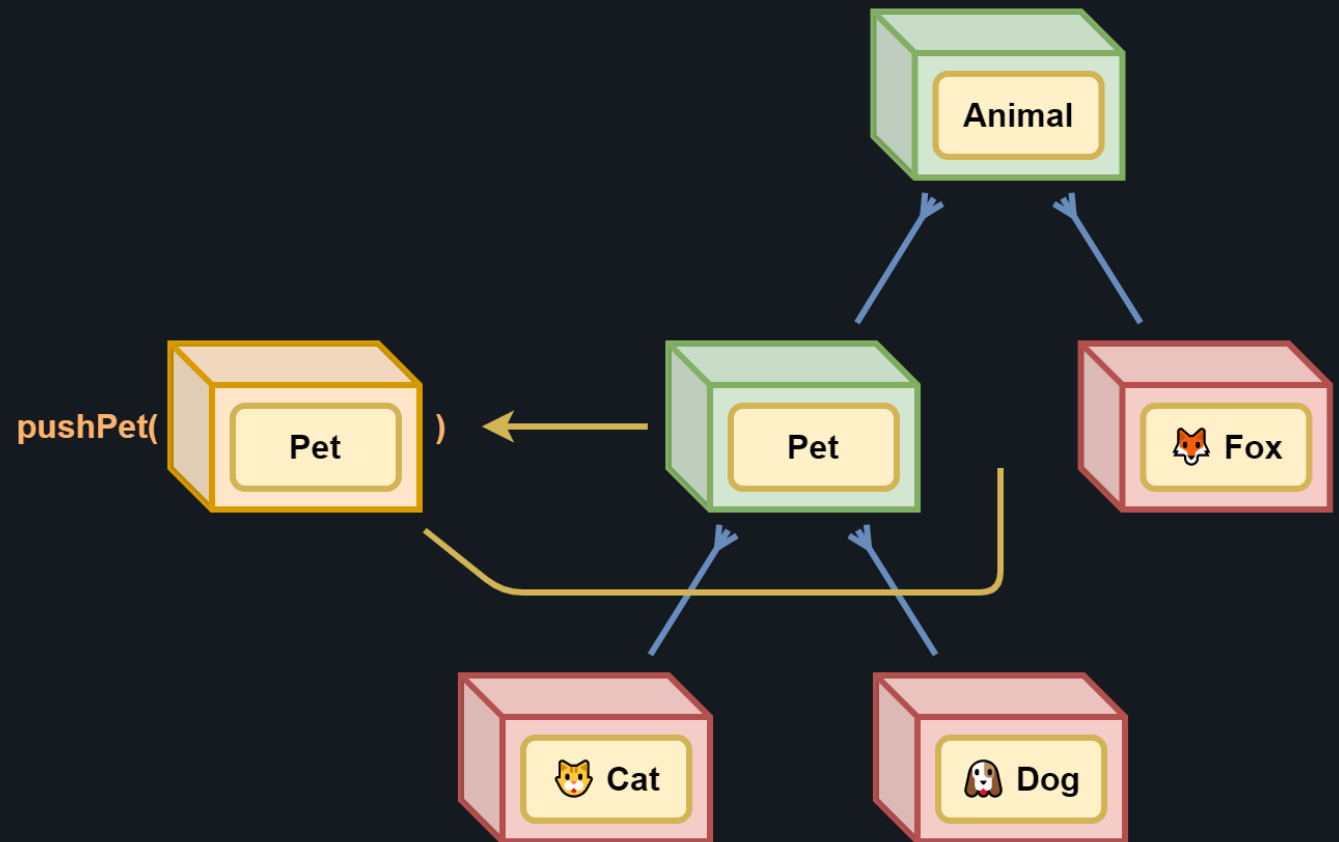
Например, если функция принимает на вход питомца, то передать ей можно хоть кошечку, хоть собачку, а вот дикую лису нельзя. Говоря современным языком, принцип LSP гласит: все параметры всех функций должны быть ковариантными, то есть ограничивающими дерево типов сверху относительно задекларированного для данного параметра.

Звучит вроде бы логично, однако...



Давайте рассмотрим функцию, которая принимает на вход коробку для питомца и засовывает в неё какого-то, нам не известного, питомца.

Если мы позволим передавать в неё клетку для собак, то вскоре можем столкнуться с абсурдной ситуацией, когда вытащим оттуда казалось бы собаку и скажем "Апорт", а она замяукает и насытит вас в тапки. То есть подтип передавать в эту функцию нельзя. А вот супертип очень даже можно, ведь в клетку для животных можно засовывать хоть домашнего, хоть дикого животного. Получается своего рода инверсия LSP. То есть иерархия типов ограничивается не сверху, а снизу. Такое ограничение имеет название "контравариантность".



Получается, что возможность подстановки одного типа вместо другого зависит не столько от типа параметра, сколько от того, что функция с этим параметром делает. И тут могут быть самые разные ограничения..

То есть, если мы хотим писать корректные программы, то мы вынуждены явно нарушать LSP во многих случаях.

- Только чтение - **ковариантность** (ограничение сверху)
- Только запись - **контравариантность** (ограничение снизу)
- Чтение и запись:
  - **инвариантность** (ограничение снизу и сверху)
  - **бивариантность** (без ограничений)

Все эти разные варианты появляются лишь когда мы изменяем какое-то состояние. Но если мы работаем лишь с чистыми функциями, которые ничего не изменяют, то ковариантность всех параметров и, как следствие, LSP мы получаем автоматически. Хотим мы того или нет - от нас это не зависит.

С другой стороны, любая небесполезная функциональная программа содержит не только чистую, но и грязную часть, где вопросы вариантности встают в полный рост.

W Функциональное программирование

Ладно, давайте пофантазируем и попробуем сформулировать LSP здорового человека, учитывающего все озвученные ранее нюансы...

"Функция может не знать конкретный передаваемый ей тип, но она обязана ограничить множество принимаемых типов так, чтобы ей нельзя было передать такой тип, который ломает ожидаемое поведение программы".

Звучит с одной стороны всё ещё так же размыто, как и оригинальная формулировка, а с другой - самоочевидно. Понятное дело, что если есть возможность правильно написать типы, то стоит это делать правильно.

✅ Статическая типизация с вариантностью

Как было показано ранее, если мы хотим писать корректные программы, то не можем следовать LSP. Ведь это принцип, а не, например, паттерн. Принципа либо придерживаются во всём, либо придерживаются чего-то другого, что лишь иногда соответствует тому принципу. Принципу нельзя придерживаться частично, так же как нельзя быть немножко беременной.

LSP, на текущий момент, - это устаревшая концепция, не учитывающая многие случаи. Поэтому применять её как руководство к действию ни в коем случае нельзя. А важно понимать вариантность ваших параметров и то, как конкретный язык программирования работает с типами.

✗ LSP


✓ Вариантность

Для лучшего понимания вопроса вариантности рекомендую свою статью на эту тему. Там я доступным языком рассказываю всю эту сложную теорию, и привожу примеры кода на разных языках программирования, иллюстрирующие разные типы вариантности.


К сожалению, в современных языках поддержка вариантности находится в зачаточном состоянии. Где-то она не поддерживается вообще. Где-то вариантности жёстко захардкожены. Где-то есть куцые механизмы явного указания вариантности. И в совсем редких языках можно встретить не только автоматическое выведение типов параметров, но и выведение их вариантности, что очень круто, так как ручное раззуливание вариантности порой слишком многословно.


 Теория программирования: Вариантность



Если данный разбор показался вам полезным, то дайте мне об этом знать посредством  лайка. А так же поделитесь ссылкой на него со своими коллегами. Особенно с теми, кто работает с ООП, придерживаясь LSP.

Если же вы не согласны с какой-либо мыслью или, наоборот, чувствуете какую-то недосказанность и хотите дополнить своими идеями, то жду ваших комментариев.

Если вы не боитесь подискутировать со мной в прямом эфире или даже готовы стать соавтором будущих разборов, то пишите  телеграммы.


Наконец, подписывайтесь на  канал, чтобы не пропустить дальнейшие разборы. Нам ещё много чего с вами нужно будет обсудить.

На этом пока что всё. С вами был немножко программер Дмитрий Карловский.

 Лайки

 Поддержка

 Комментарии

 Подписка

W Interface Segregation Principle

*Самый конкретный принцип из всех!*

💡 Сущности не должны зависеть от методов, которые они не используют

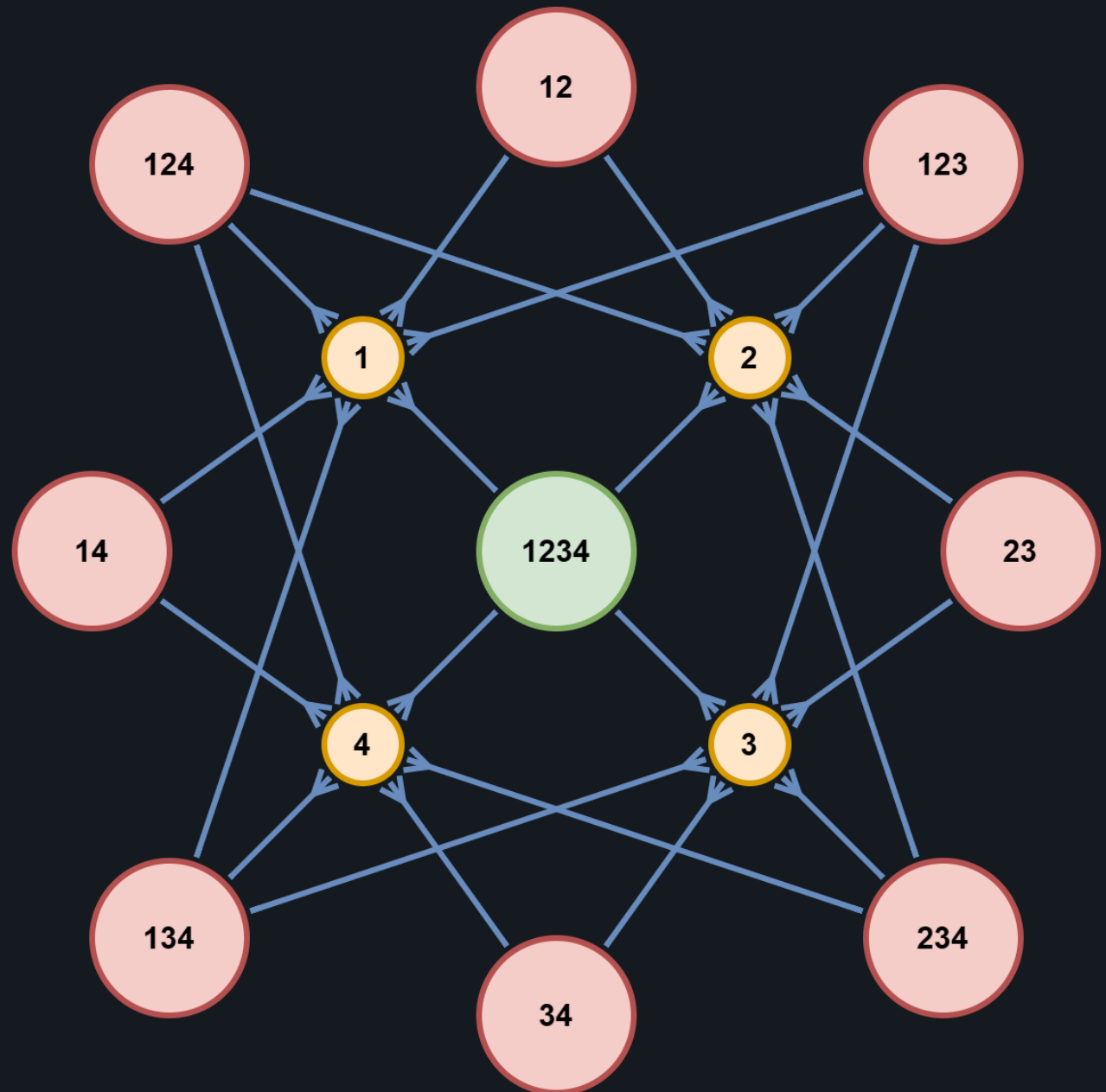
*Один общий интерфейс делится на кучу мелких. А потом из кучи мелких собирается куча наборов под разные места использования.*

▼ User

▼ UserFirstName, UserSecondName, UserNickName, UserBirthDay, ...

▼ UserForSignIn, UserForSignOut, UserForProfile, UserForLink, ...-

Число интерфейсов многократно увеличивается. В них сложно не запутаться. Их сложно поддерживать, ибо добавление вызова метода требует обновления интерфейсов в многих местах по пути к месту вызова.



✓ Достаточно минимального набора методов

✗ Раздутие кода на ровном месте

✗ Каскадные изменения интерфейсов

*Но заглядывать в будущее сложно. А если эти методы там так и так будут, то ничего страшного, если будет использован более общий интерфейс.*

💡 Не требуйте у зависимостей то, что вам точно никогда не понадобится

✗ ISP

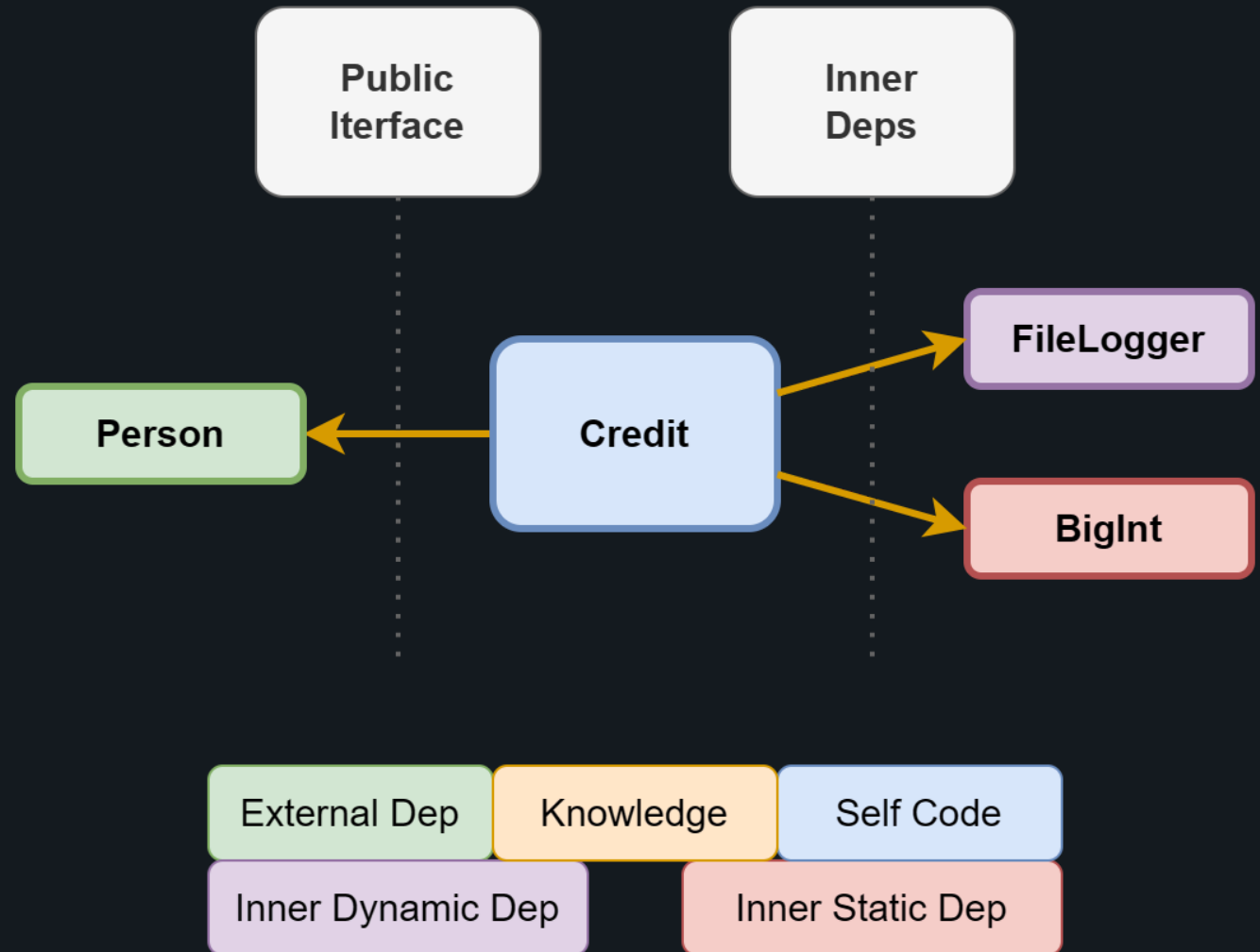
✓ Типовые интерфейсы



???

W Dependency Inversion Principle

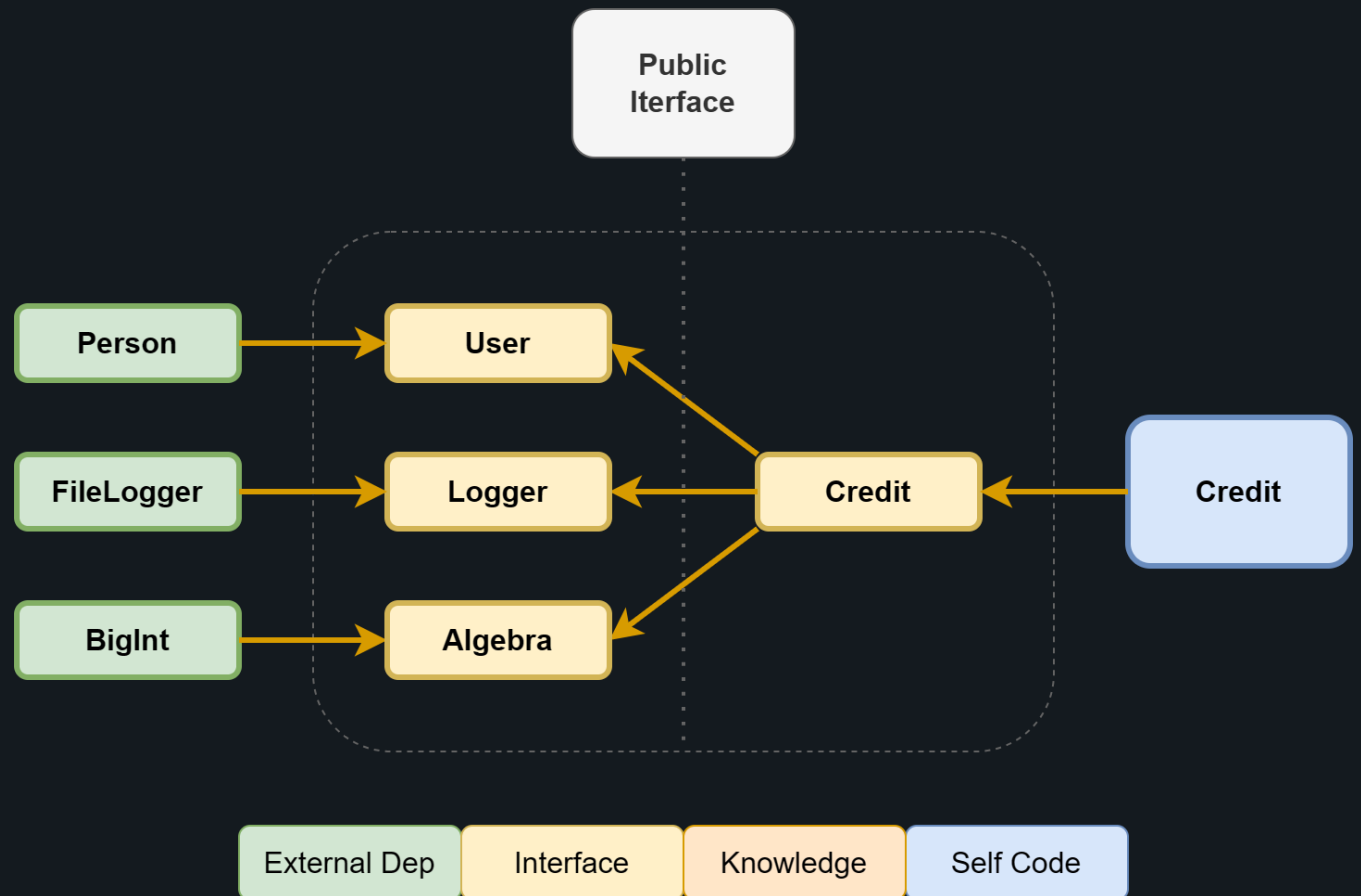
Зависимости бывают входящими, внутренними динамическими и внутренними статическими. Прямые зависимости о реализации не позволяют использовать тот же код с иными реализациями без копиясты.



"Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. А абстракции не должны зависеть от их конкретных реализаций."

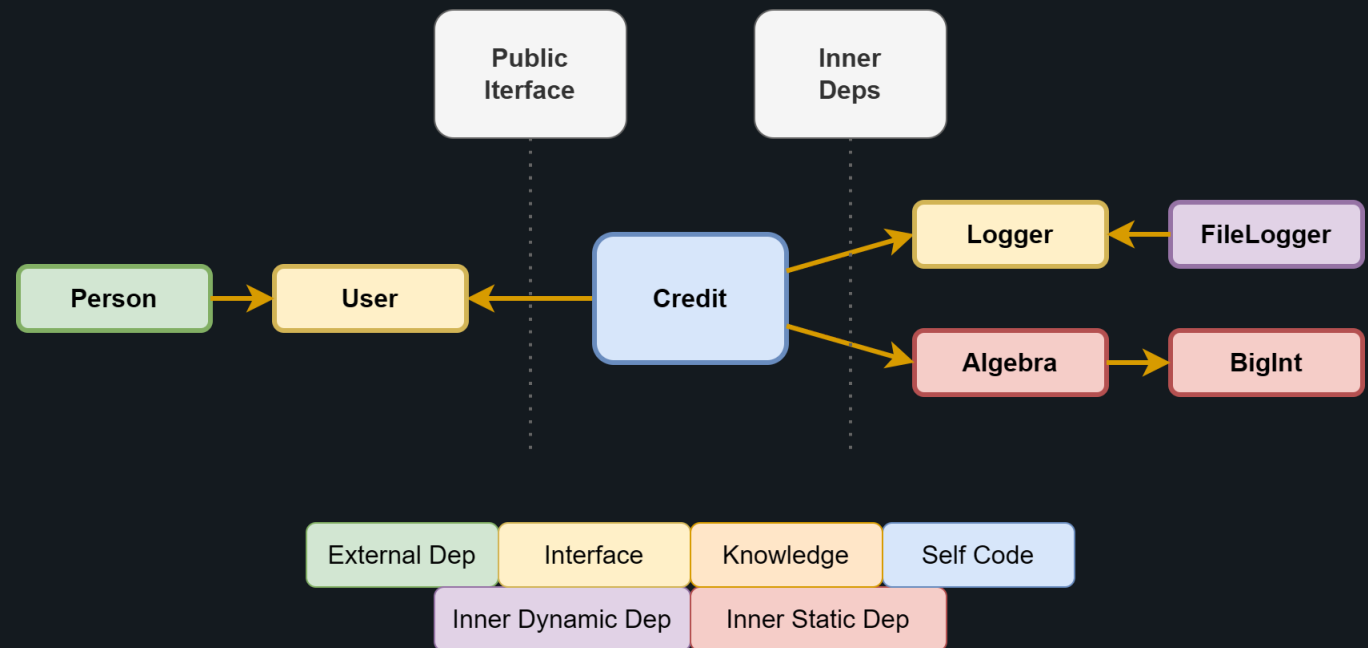
💡 Реализации должны зависеть только от абстракций, но не наоборот

Строится иерархия из абстракций, от которой уже зависят конкретные реализации, которые ничего не знают друг о друге.



- ✗ Раздутие кода
- ✗ Протекание абстракций
- ✗ Невозможность инлайнинга зависимостей
- ✗ Сложно навигироваться по развязанному коду


Используйте разнообразные способы резолвинга зависимостей в зависимости от потребностей. Одни инъецируются из вне. Другие лукаются по интерфейсу. Третье берутся напрямую, либо через адаптер.



✗ DIP

✓ Инструмент под задачу




 Критический взгляд на DIP / Сергей Тепляков

- |                                        |       |
|----------------------------------------|-------|
| ✗ SRP: Single Responsible Principle    | ? REP |
| ✗ OCP: Open–Closed Principle           | ? CCP |
| ✗ LSP: Liskov Substitution Principle   | ? CRP |
| ✗ ISP: Interface Segregation Principle | ? ADP |
| ✗ DIP: Dependency Inversion Principle  | ? SDP |
|                                        | ? SAP |


Корректно описывай вариантность в типах.  
Не требуй у зависимостей то, что никогда  
не понадобится. Разделяй и властвуй. Не  
ломай публичный контракт без  
необходимости.

- ✓ VTP: Variance Typing Principle
- ✓ ARP: Ascetic Requirements Principle
- ✓ LDP: Limited Decomposition Principle
- ✓ ICP: Interface Compatibility Principle
- ✓ DDP: Dependency Diversity Principle

 Почему я не преподаю SOLID / Brian Geihlsler

 Почему принципы SOLID не являются надёжным решением / Elye

 История возникновения CUPID / Daniel Terhorst-North

 core\_dump - основы

 mam\_mol - новости

 hyoo - донаты

 h\_y\_o\_o/28 - обсуждения

 nin\_jin - личка