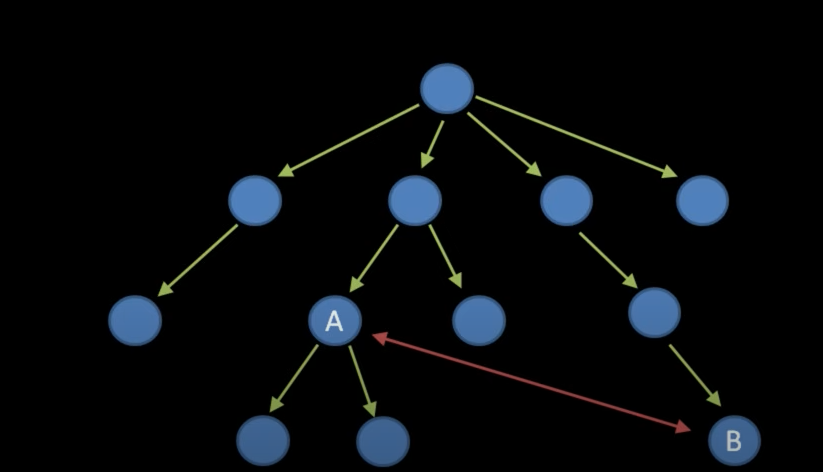
Немного об ООП, его обратная сторона

[Object-Oriented Programming is Bad (youtube.com) (ДА, НАЗВАНИЕ НЕОБЪЕКТИВНОЕ И СЛИШКОМ ГРОМКОЕ)](https://www.youtube.com/watch?v=QM1iUe6IofM)

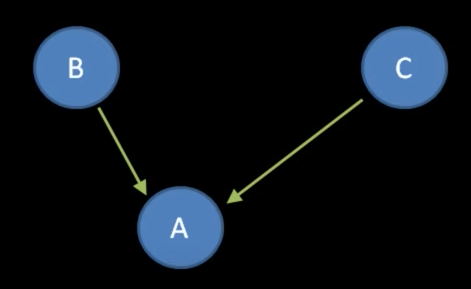
Есть глобально два подхода: императивный и декларативный. В императивный входят несколько видов программиронивания, как и в декларативный. Но все эти подходы строятся вокруг того **как управлять состоянием программы**. В процедурном программировании состояние глобальное, то есть там часто используют глобальные переменные. В ООП- состояние защищается инкапсуляцией, то есть состояние хранится «в одном месте» и имеет «свое апи для изменения» и ООП строится вокруг инкапсуляции. Но вот с инкапсуляцией есть проблема – **идеальный ее вид невозможен и если его и использовать то это очень трудно и время затратно, и такой вид инкапсуляции не подходят под реалии программы, которые постоянно меняются**

Вот пример. Синие кружочки – это классы, а стрелочки вызовы открытых методов других классов. Зеленая стрелочка – как должно быть в идеальной инкапсуляции, красная как не должно быть.



То есть если вы хотите из А класса использовать В класс, то нужно через родительные классы А по цепочки дойти до класса В, НО НЕ НА ПРЯМУЮ. Естественно когда код постоянно меняется число по человечески сложно поддерживать такую структуру – на это нет времени, и человек слишком «тупой», чтобы вот так вот писать. По моему мнению такая структура подходит когда программу написали один раз и не меняют. Когда один метод (в ооп) вызывает другой, даже через публичные методы с копией данных и без ссылочных типов в аргментах в методах(чего нет в c#), то состояния двух классов смешиваются.

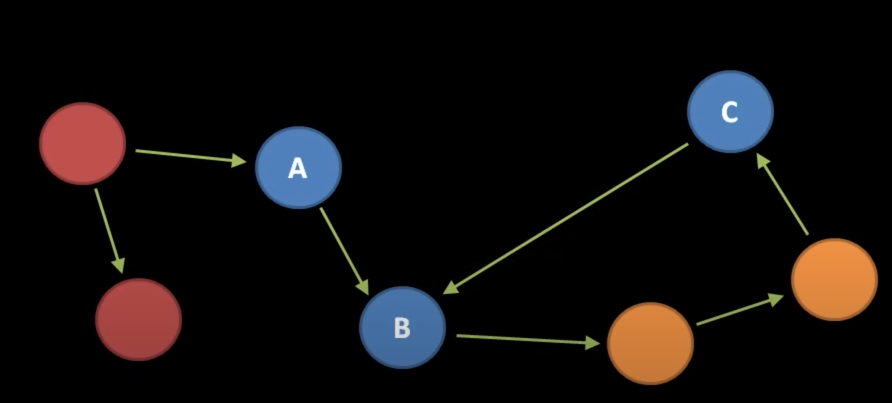
А если происходит следующее.



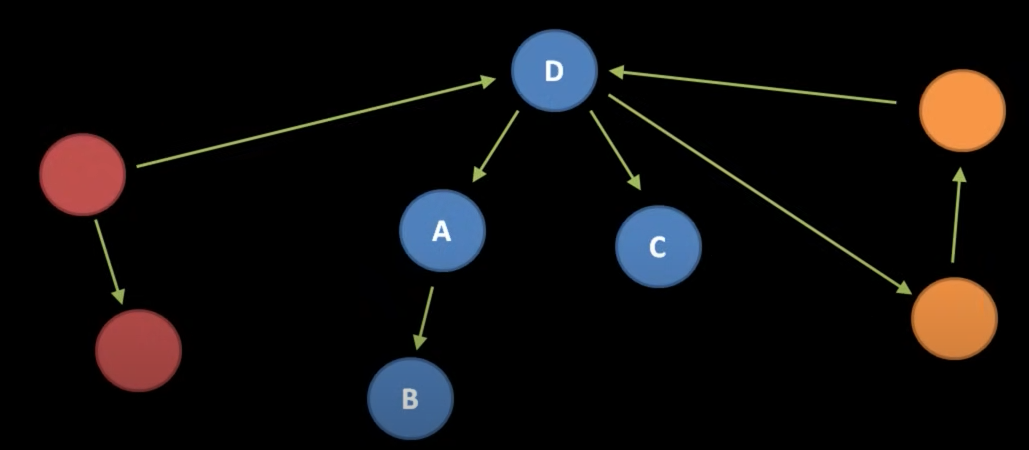
То состояние из B и C в А перемешается. Что нарушает инкапсудяюцию и приводит вообще к багам. На этой картинке еще безобидная ситуация.

И в реальности с классами происходит вот такое.

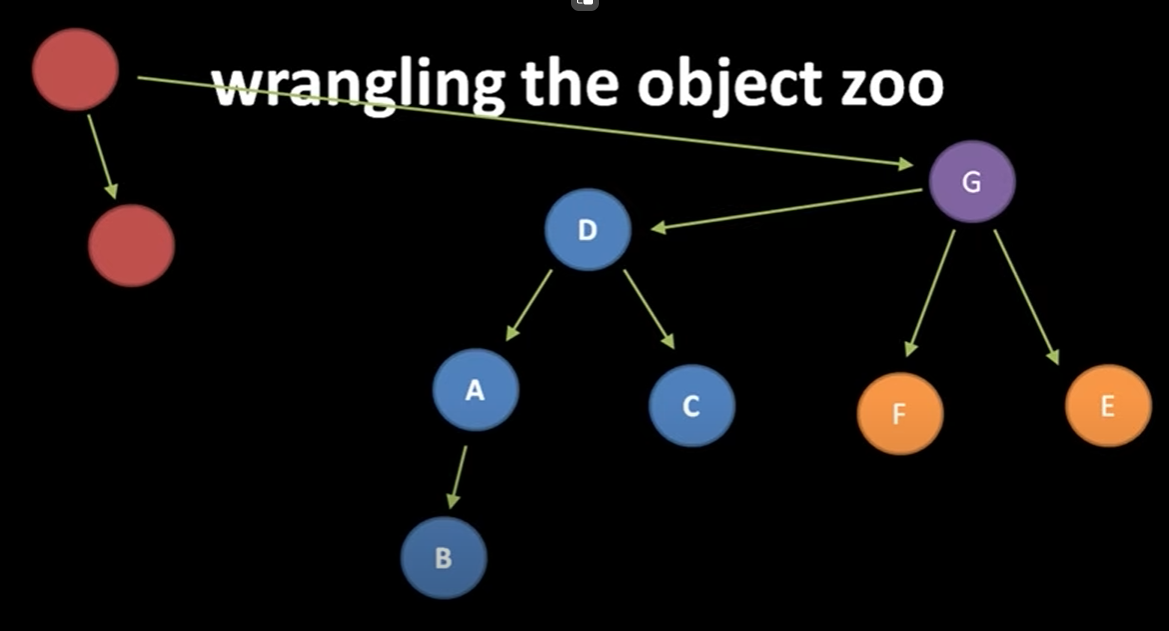
Сначала структура может быть такакая



Потом такая



А потом вообще такая



Вообщем программирование превращается жанглирование этими кружочками. И в примерах, еще мало классов. А в реальности классов больше и ситация очень печальная. Особенно с бизнес сущностями где у тебя десятки классов и у тебя смешивается состояние, и из-за этого поялвяются баги, точнее для тебя код превращается в черный ящик от которого хер пойми что ожидать. И ты пытаешь подобрать «самую идельную» комбинацю таких кружочков(классов), но это невозможно для больших проектов. Так как таких кружочков **в лучшем случае несколько десяток.** А может доходить и несколько сотен. И попробуй в уме понят как подобрать «ту самую идельную архитектуру». ЭТО ДАЖЕ КОМПЬЮТЕР БУДЕТ ДОЛГО ОБРАБАТЫВАТЬ ПРИ УСЛОВИИ ЕСЛИ НАУЧИТЬ ЕГО СТРОИТЬ АРХИТЕКТУРЫ ВЗАИМОДЕЙСТВИЙ КЛАССОВ, поэтому ты будет приходить по итогу «хорошей» структуре. Но так тогда – инкапсуляция улетает в окно, и у тебя просто смешан по всем классам стейт. А чтобы такого не было, нужно использовать иерархическое взаимодействие классов. Или подобрать самую идельную комбинацию классов, чтобы последствия от смешивания стейта были не так сильны.

В декларативных стилях программирования как такого состояния нет, **в функциональном в частности вообще стараются не иметь состояния**, хотя это не везде возможно(так как есть файловая система и сеть с которой нужно иметь дело). Функциональное программирование строится на иммутабильности и чистых функция.

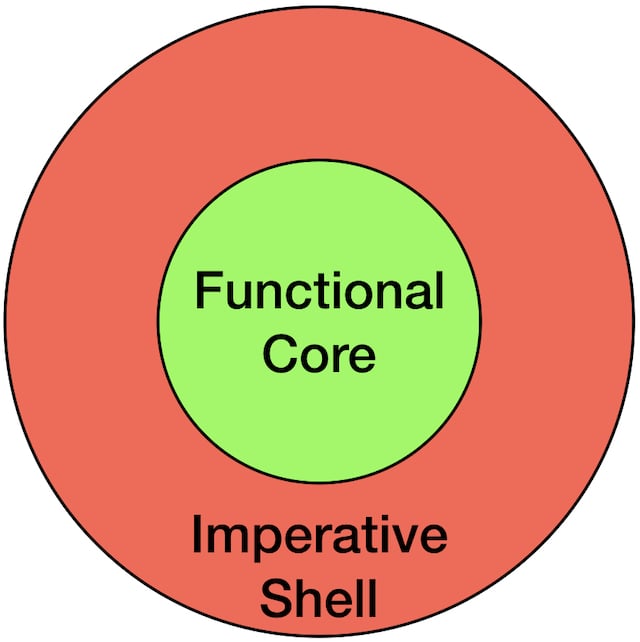
Чистая функция которая имеет в ходной параметр и выходной результат и не имеет никакого внешнего эффекта, ни логирования, ни записи в бд, ни изминения глобальной переменной/изменения переменной более высокого уровня в функции. Они позволяют писать относительно предсказуемый код. Но их минус – не оставляют сайд эффекта(их минус и плюс). То есть чистые функции не подходят для для работы с сетью и бд, но отлично подходит для всего остального есть логика и поведение.

[The Essence of Pure Functions (youtube.com)  
https://youtu.be/HlgG395PQWw?si=zi4md-Q5vevEBlUH](https://www.youtube.com/watch?v=s7E6o0djYWI)

А вот отличный пример как можно писать с рекордами

https://youtu.be/Z-fMkvIEmGM?si=-Qt3RURMeU0TBT15

Поэтому есть подход где ты Imperative Shell и Functional Core.



Но есть drawback виде того, что в чистую функцию нужно давать сразу все данные для логики и нельзя вот так вот постепенно делать вызовы к бд, делать логи. [Moving IO to the edges of your app: Functional Core, Imperative Shell - Scott Wlaschin (youtube.com)](https://www.youtube.com/watch?v=P1vES9AgfC4)

«Давайте поговорим об изменениях». Всегда не изменно наличие изменений. Главный источник изменений это требования бизнеса и его клиентов.

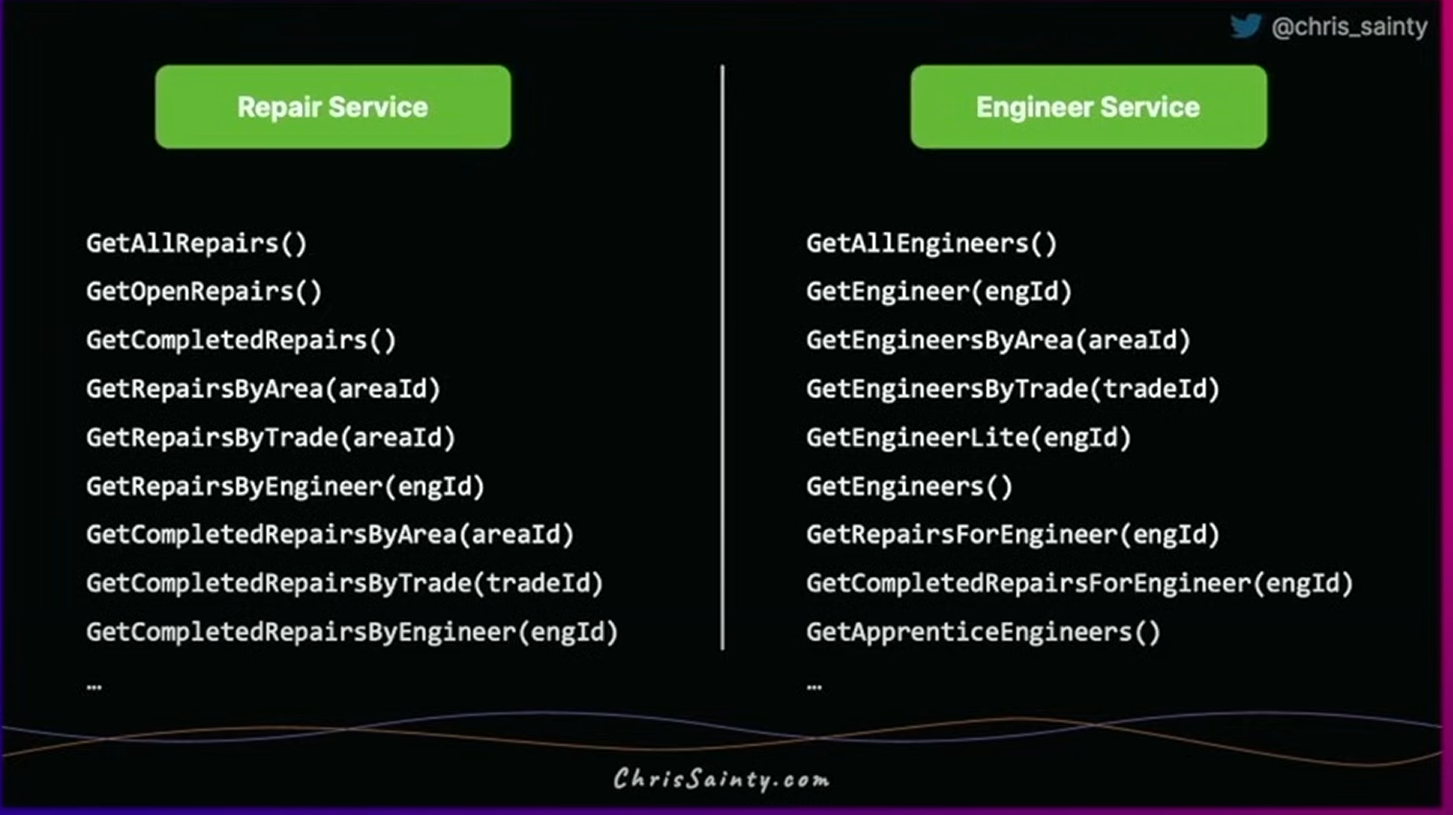
Обычно бизнес логика программистами(которые предпочитают ООП) строится вокруг сущностей, RepairsService, JobServices, EngineerService и тд. Когда ты начинаешь проиктировать архитектуру, ты берешь класс(сервис, сущность, контроллер) и добавляешь туда методы, которые делают действия, то есть ты строишь поведение вокруг сущностей. И ты смотришь на архитектуру и ты думаешь что она замечательная. Но есть проблема – давление со стороны бизнеса, и они часто влияют на то, каким у тебя будет приложение.

1. «Rushed Features» – фичи, которые ты должен сделать здесь и сейчас, причин много, но все привоводит к таким фичам
2. Кастомные фичи для отдельного человека
3. Добавление определенного поведение, которое решит определенную проблему: баг, трата большого времени сотрудника, тупые для разарботчика, но не тупые для пользователя ошибки и прочее, что связано с тем чтобы добавить какое-то поведение в программу.

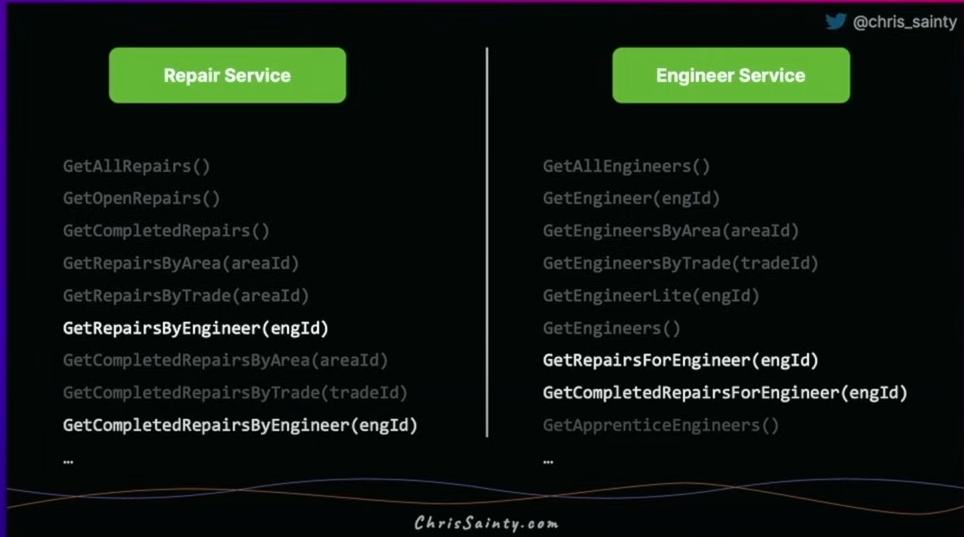
Системы для бизнеса – это в первую очередь про возможности, а не про сущности, про поведение, про ЛОГИКУ, когда происходит то – происходит это, на таком событии сделай то-то, раз в несколько дней делай вот это.

Поведение системы становится сложнение, а возможностей больше. Пронесемся на 5 лет

И когда такое происходит то с сущностями происходит следующее:



P.S это еще цветочки, я видел такие интерфейссы на 1000+ строчек кода! А сервис в 200к строчек кода! Первое что в голову приходит: это проблема в людях, которые слишком раздувают интрефейся/сервис/сущность, как раз идея сервиса в том чтобы туда что-то добавлять. Проблема тут в другом: НАРУШЕНИЯ ПРИНЦИПА SINGLE RESPOSOBILITIES. «One reason to change». Какая причина может быть в том чтобы изменить 200к строчек кода. Это должна быть серьезная причина! И внимание:

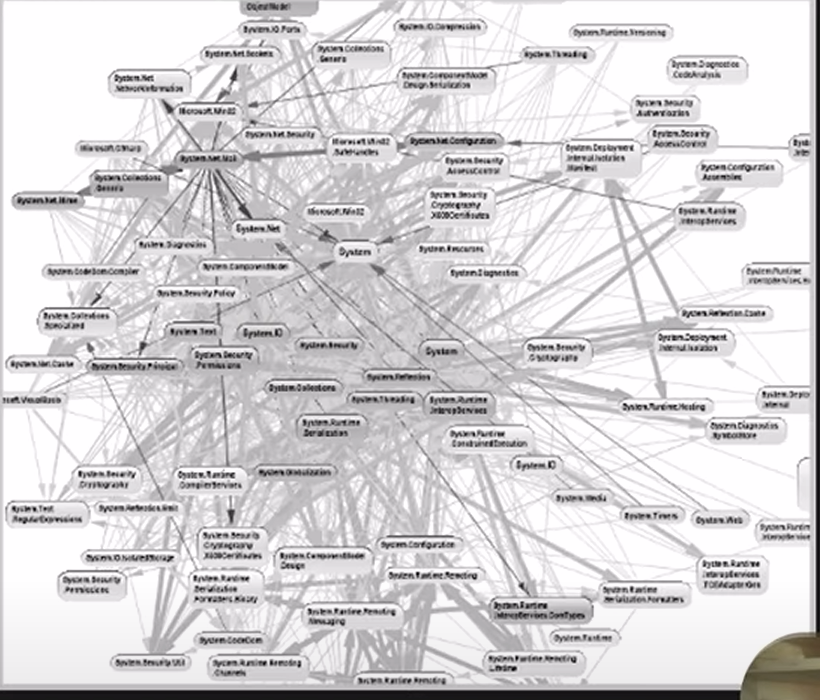


Эти методы очень похожи, и хрен разберешь чем они отличаются и почему находятся в разных сервисах! И потом с этими сущностями встает вопрос, а что из этого использоваться?! И тебе нужно тратить время на то чтобы разобрать 4 методы, которые делают одно и то же, чтобы понять, что тебе нужно, но ты можешь либо спешить, либо слишком плохо знать систему, чтобы разобраться хорошо, а что тебе нужно. А это лишь вершина айсберга. Другая проблема (в контексте DI), каждый этот метод может грузить свои зависимости, и по итогу один твой сервис с 1000 методами создает 100 классов чтобы выполнить 1 функцию!. Я не стороник преждевременных оптимизация, но все равно как-то не очень от того факта, что ты на ровном месте тратишь ресурсы, так еще и сверху прошлые проблемы имеешь. И другая проблема: человеческое внимание и его возможности – они ограничены. И тебе как человеку в таком классе сложно ореентироваться, ты можешь легко потерься в коде, проморгая свою функцию, и твой мозг начинает кипеть от того, что ты за раз видишь столько кода (ВОТ ТОЛЬКО ПОСМЕЙ СКАЗАТЬ ЧТО ЭТО У МЕНЯ ПРОБЛЕМЫ С ОПЕРАТИВКОЙ, я человек, а не компилятор!) А еще другая проблема «mental boilerplate» из разряда, а куда положит

Пример из нашего прода. В умном клиенте в Constructions, когда я лазил по сущностям и методам, мне сложно было понять, как методы внутри сущности связаны друг с другом, ну то есть да, они привязаны к сущности, но я не понимал как в плане поведения они связаны, они просто есть вместе но не вызывают друг друга а просто находятся рядом, и вот у тебя файлик в 2к строчек кода(при том это не сервисы, а сущности со стейтом, что вообще делают ситуацию ужасной!). Каждый метод имеет разные причины для изменений.

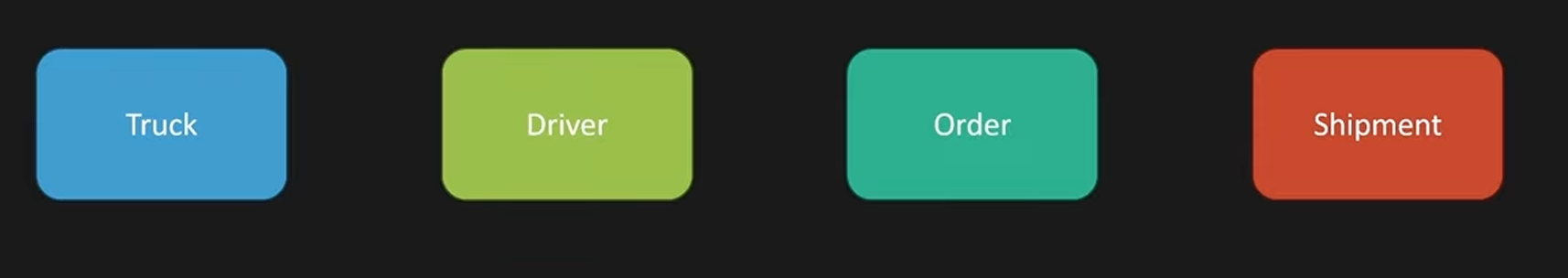
И мое любимое: НЕСКОЛЬКО ДЕСЯТОК СВОЙСТВ С ТАКИМ ЖЕ КОЛИЧЕСТВОМ ФУНКЦИЙ, КОТОРЫЕ СВЯЗНЫ В ОДИН STATE. Кода на 2к строчек. И ладно если бы это была бы прямая логика, где если произошло, то значит делай это, но нет, там были кучу публичных методов, которые меняеют глобальные свойства программы, которые чем-то похожи на глобальные переменные, даже если у них есть своя логика, только в контексте класса, и глядя на один метод ты не ожидаешь/не знаешь/не помнишь какой state будет у класса – буквально камок черного ящика, который очень сложно распутать, так как там за одним вызовов последует сотня мутаций данных, даже если ты прочитаешь весь код одного метода, и дай бог чтобы он был хотя бы не больше 30 строчек кода!

И по итогу вот что происходит! (Вообще это картинка из видео, что если использовать dry для сущностей, но здесь тоже хорошо подходит) Моя оперативка не способна столько переварить!

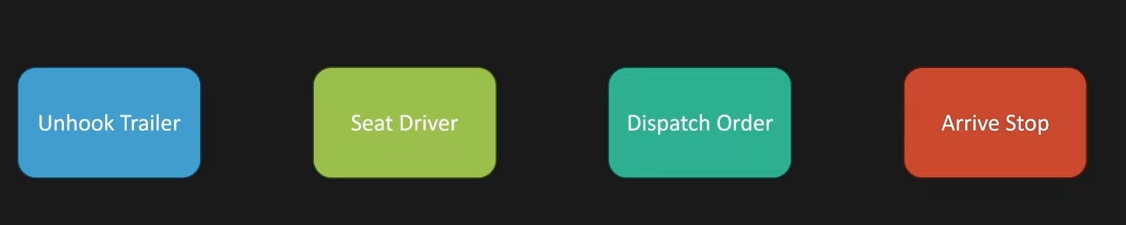


Напоминаю что каждая так сущность содержит десятки свойство и еще больше состояния, и по итогу программа будет вести себя не предсказуемо и искать почему произошел баг, это возможно, но слишком сложно. Это происходит если система строится вокруг сущностей и к сущностям добавляется поведение. И вспоминаем про инкапсуляцию, здесь она покинула чат, состояние в таких сущностях с свойствами перемешалось по всей программе.

Или вот пример. Ты смотришь на сущности и не понимаешь, что они делают, что происходит с программой. Рисунок ниже. А главная цель приложения что-то делать и давать результат.



Вот так намного понятней! Тв с ходу видишь возможности и фичи, которые есть в системе.



Абстракции и трейдоф

«Программирование – искусство про трейдоффы» – отличная фраза, которая подходит для описания всей сложности программирования и почему чертовски сложно. «И нет серебрянной пули.» Общепринятые и очевидные истины о которых стоит напомнить.

<https://zenstack.dev/blog/trade-off>

Любое действие имеет последсвтие – начиная, от создания лишней переменной, которая может привести к непредсказуемой работе программы и к сложном у понимаю другим программистом как код работает, заканчивая какой язык и библиотеку выбрать.

Абстракция в том числе имеет свои последсвтия – связывание, усложнения и утечка.

[The Law of Leaky Abstractions – Joel on Software](https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/)

1. Усложняет тем что, это еще один слой, с которым нужно взаимодействивовать, документировать, объяснять как он работает, поддерживать. Так же абстракция тебе диктует как писать код. Требуют времени чтобы их изучить и использовать. А если делать свою – то за этой абстракцией нужно следить, чтобы ее не сломали и она не утекла. СЛОЖНАЯ АБСТРАКЦИЯ ОЧЕНЬ ДОРОГО И ДЛЯ ЕЕ ИСПОЛЬЗОВАНИЯ ДОЛЖНЫ БЫТЬ ОЧЕНЬ ХОРОШИЕ ПРИЧИНЫ.
2. Связывание появляется, так как абстракция всегда создается, чтобы ее можно было использовать в нескольких местах. А связывание – причина, почему от одного изменения код ломается в нескольких местах. (Вообще тема больше об dry, но здесь тоже стоит это упомянуть). Связывание не плохо и не хорошо – это вещь которую стоит применять ОЧЕНЬ АККУРТАНО.

<https://youtu.be/_Al7qI4vMt0?si=Bzq0XhuVr77hvkJs>

1. Утечка абстракций – неизбежная вещь, когда абстракция прячет от сложных вещей, и чтобы этого не происходило – за абстракцией нужен жесткий дозор. Возьмом c# – большая абстракция над тем, что происходит с памятью(вспоминаем про GC), как работают вычисления и пр. Абстракции шарпа вытекают, утечка памяти(как каламбур) как раз утечка абстракции – **но есть люди которые следят за этим.** Любая сложная абстракция требует пристольного внимания и много ресурсов.

А если использовать уже готовую сложную абстракцию тебе нужно знать как она работает внутри, так как она неизбежно будет либо ломаться либо утекать. И возьмем тот же Ef Core – иронично, но не смотря на то, что он призван как упростить жизнь – он ее усложняет, так как он абстрагирует от очень и нет мега сложных вещей. Это абстракция над абстракцией(комбо). Тебе нужно потратить трилион времени, чтобы заставить делать ef core то что тебе надо. Как тот же csla – абстракция, которая используют http/wcf, которая абстрагирует еще от запросов к бд. И проблема еще абстракций в том – что они дают не всегда идельное апи, и тебе приходится идти против апи абстракции. К примеру прямые sql выражения в ef core. Мое любимое – использовать паттерн репозиторий поверх ef core, который является реализация паттерная Unit Of Work, который включает в себя репозитрий, что собственно делает csla. Такая абстракция будет утекать, апи репозиториев или dal часто будут не подходить и его нужно будет постоянно менять и дорабатывать, то есть добавлтяь новые и новые методы в интерфейс. И я заметил что люда любят использовать **DbContext в интерфейсе репозитрия.** Мягко говоря – это нонсент а прямо говоря – пиздец полный. Мало того, что идея абстракции нарушется – быть независимым от фреймворка и бд (хотя это невозоможно), так еще добавляет доп слой, с которым нужно иметь дело. (Забавно что мне такой репозиторий показал тим лид с 10 летным опыт тимлида). Или когда майкрософт – рекомендуют исопльзовать паттерн репозиторий/dal повверх ef core, для тестируемости(а почему ef core не выполняет этой функции, нет серьезно, тестирумость – базовое свойство, которым должен обладать ef core как реализация паттерна репозитория и Unit Of Work). Поэтому я считаю efcore мусором, который не следует использовать(если выражаться более корректно юзкейсов где сложности с efcore могут быть оправадны очень мало, единственное это использовать OData протокол в asp.net core, и то сомнительно как по мне). Каждая абстракция должна иметь ценность, так как имеет ОЧЕНЬ БОЛЬШУЮ ЦЕНУ. А еще ef core это RAD tool

[Why I Dislike Entity Framework for REST API Development - DEV Community](https://dev.to/webjose/why-i-dislike-entity-framework-for-rest-api-development-1lib)

Dal csla имееют выше указанные проблемы с сервисами. И одно дело если бы из всех доступных способ с общенией с бд была бы чистая ADO.NET (тот редкий случай поверх чего справедливо писать абстракции), но есть Ef Core и Dapper. И даже так – НЕ ВОЗМОЖНО НЕ ЗАВИСИТЬ ОТ БД. Это буквально задает вектор развития приложения на старте. А чтобы сделать бизнес логику независимой от логики приложения – самое главное, чтобы у бизнес логики и логики с бд – были отдельные сущности, даже если классы с ними полностью индетичные так как у них разные причины для измения, и этого достаточно – слои как таковы не нужны, главное чтобы код бизнес логики использовал свои сущности. А гланое – СУЩНОСТИ/ТАБЛИЦЫ БД НЕ ДОЛЖНЫ ДИКТОВАТЬ БИЗНЕС ЛОГИКУ. А когда ты строишь бизнес логику вокруг сущностей, к которым ты подсоединяешь поведение – твои сущности, даже которые относятся исключительно к бизнес логики и никак не свзяаны с бд, будт диктоваться твоей моделью данных в бд, это неизбежно, если ты строишь бизнес логику именно вокруг сущностей, потому что твои сущности в программе – это сущности факту из бд. Это причина по которой нужно при написании бизнес логики – использовать функциональное программирования

1. Строится все вокруг повединия – чем по существу и является бизнес логика. Бизнес логика – это поведение, а не сущности
2. Не нужно использовать никакие абстракции для бизнес логики. Создаешь функцию(чистую функцию). Делаешь нужные параметры либо виде параметров функций или класса(ОТДЕЛЬНОГО КЛАССА КОТОРЫЙ НЕ ИСПОЛЬЗУЕТСЯ ДЛЯ ОБЩЕНИЯ С БД). И описываешь. Из-за такого подхода у тебя бизнес логика никак вообще не будет зависить от остального когда. Так как там исключетльно поведение, чем и является бизнес логика, и нет ни логирования, ни записей к бд, ничего такого, только поведение и результат. (Imperative Shell и Functinal Core).
3. Бизнес логика будет более стабильней в повидении из-за природы функционального подхода(иммутабельность и чистые функции)

Еще абстракции нужно документировать, и хорошо документировать, чтобы можно было если понять, что не так. Документация должна пояснять суть апи, которое дает абстракция, каждую часть апи, каждую, хорошо и с примерами. **У csla нет документации, только примеры и книга**, которую нужно купить. А искать по гуглу и гитхабу через вопросы и ответы, что ты мог сделать не так долго.

А еще csla слишком свзяывает клиент и сервер. (Впомни как у тебя постоянно, что-то ломается от того, что ты поменя на сервере метод csla), так как у csla нет никакого версионирования, и по итогу нет никакой обратной совместимости, ЧТО ВООБЩЕ ЯВЛЯЮТСЯ ЗОЛОТЫМ СТАНДАРТОМ В ХОРОШЕМ ПРИЛОЖЕНИИ!

Я надеюсь ты прочитал статью

[Why I Dislike Entity Framework for REST API Development - DEV Community](https://dev.to/webjose/why-i-dislike-entity-framework-for-rest-api-development-1lib)

Потому что csla имеет такие же проблемы, но вот тольок ef core – стандард в .net, csla – на НЕГО НЕТ ДАЖЕ ДОКУМЕНТАЦИИ, НЕТ СЕРЬЕЗНО.

Материал, который повлиял на мое мышление

Название громкое

https://youtu.be/QM1iUe6IofM?si=hNe9eBJrOulz3q39

Вертикальные слайсы и почему слои могут не работать

https://youtu.be/\_1rjo2l17kI?si=7US3HsSEJsETWjH2

Dry(entities)

https://youtu.be/znpdlYgvU3M?si=hC-0AMoDT8HYE1uA

Screaming arhitecture

https://youtu.be/Stu8SqtW2bg?si=DICKwsPSNfMZuTKX

Чистые функции

https://youtu.be/P1vES9AgfC4?si=pJ8LRBxYrAs2yBu-

Как построить ужасную архитектуру(Не относится, просто интересное)

https://youtu.be/88\_LUw1Wwe4?si=ok2z9R6ZRT-ehIHv

Как не построить распределенный монолит(Не относится, просто интересное)

https://youtu.be/p2GlRToY5HI?si=tcGMLrZD1GT9ttk1

ABP NET Implementing ddd(book)

Еще про микросервисы

https://youtu.be/\_UN50hNZlx4?si=0jJimCcfAxnCu9Xr

Yagni

https://youtu.be/\_Al7qI4vMt0?si=Bzq0XhuVr77hvkJs

Отличный пример с имутабельностью

https://youtu.be/Z-fMkvIEmGM?si=-Qt3RURMeU0TBT15

Глава про приближенные алгоритмы

«Грокаем алгоритмы»

Абстракции

https://youtu.be/rQlMtztiAoA?si=XPV98pcw1wA8UhUU

Меньше багов

https://youtu.be/HlgG395PQWw?si=zi4md-Q5vevEBlUH

REPR пртмер про single resposobility

https://github.com/FastEndpoints/FastEndpoints

Трейд оффы

https://zenstack.dev/blog/trade-off

https://youtu.be/s7E6o0djYWI?si=2fCm81HGvrxgeMpq

Утечка абстракций

https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/