**SOLID** принципи в описі структури мобільного телефона

1. Single-Responsibility Principle

В даному випадку моб. Телефон ми будемо розглядати як абстрактну сутність, тому що, телефони бувають різні по внутрішнім та зовнішнім характеристикам, можуть мати різний функціонал. Телефон, якщо говорити в загальному, як засіб зв’язку призначений першочергово для зв’язку між іншими телефонами. Ми не можем телефоном забивати цвяхи (не берем до уваги нокію 3110:)), телефон не приготує нам якусь страву, він не вилікую нас від застуди. Хоча сучасний прогрес розвитку технологій веде до створення універсального бійця, але все ж існують певні функції які краще розділити і покласти їх виконання на окрему сутність. Як раз перший принцип SOLID - “*клас повинен мати леше одну відповідальність*” і говорить про те щоб розділяти функціонал, щоб не створювати один GODClass.

1. Open/Close Principle

Отже при побудові класу моб.телефон можна сказати, що всі сучасні моб.телефони виконують функції зв’язку між собою, обробку інформації, здійснюють безконтактний розрахунок тощо. І в загальному це правильно, але з точки зору ООП і SOLID це призводить до нагромадження коду, тому що моб.телефон це складний функціональний пристрій в якому виконання різних процесів покладено на різні електронно-функціональні модулі. Так, за обробку інформації відповідає процесор, який має низку своїх власних характеристик за безконтактний розрахунок відповідає модуль NFC, не всі телефони оснащенні ним. І для того щоб збільшити продуктивність роботи телефона необхідно замінити процесор або добавити ОЗП., чи замінити ще якийсь модуль. Тому з точки ООП і другого принципу SOLID “*клас повинен бути відкритим до розширення (наслідування), але закритим для модифікації”* розробникам моб, телефонів необхідно його будувати на різних модулях а краще використати одну так звану материнську плату, де з легкістю можна перепаяти певну мікросхему не міняючи інші*.*

1. Liskov Substitution Principle

Як згадувалось у попередньому принципі моб.телефон будується на основі електронно-функціональних модулів, які вмонтовані та зібрані на одній електронній платі. Для прикладу третього принципу SOLID “*батьківський клас повинен легко замінюватись дочірніми класами без ламання програми*” будемо говорити, що плата це є наш супер клас, а від нього унаслідуються решта телефонів, тобто інший телефон це теж плата яка має процесор, пам'ять та інші модулі. Отже даний принцип говорить що необхідно будувати ієрархію класів таким чином щоб унаслідуванні класи могли замінити батьківський(не до кінця розумію який приклад навести правильний)

1. Уявімо собі ситуацію розробники телефонів комплектують готовий телефон: кладуть у коробку навушники для прослуховування аудіо; кладуть зарядний пристрій для підзарядки телефона; документацію і звісно сам телефон. Сучасні тенденції розвитку технологій диктують правила і тепер телефони заряджаються безконтактно. Тобто уявімо, що у нашого телефона відсутнє гніздо для підключення блоку живлення. А як говорилося в комплект поставки блок живлення входить. З цього випливає що він тепер непотрібний. Зведемо це на криворуких комплектувальників:). Так само і в ООП якщо провести абстрактні паралелі то комплектація виступає як інтерфейси. Відповідно принцип Interface Separation Principle говорить *“не потрібно примусово імплементувати непотрібні інтерфейси, які не використовуються класом”* в нашому випадку це непотрібний блок живлення, його краще було покласти до коробки з телефоном у якого є гніздо для підзарядки. Тобто в таких ситуаціях потрібно розділяти інтерфейси.
2. Уявімо собі, що процесор в моб.телефоні містить відео модуль який відповідає за графіку відображення на дисплеї, здебільшого від нього залежить можливість грати у високопродуктивні ігри або відтворювати відео з високою роздільною здатністю. Для простого споживача це класно, бо нас не цікавить яка всередині телефона архітектура, і для розробників з рештою теж підходить таке рішення адже дозволяє зекономити місце в малому корпусі. Але технічний прогрес не стоїть на місці, з’являються більш вимогливі до продуктивності телефону ігри, чи відео, чи програми які працюють з відео тощо. Що ж робити розробнику – виготовляти новий процесор, це дорого. В даному випадку потрібно розділити залежності, продуктивність відеогри неповинна напряму залежати від процесора, краще змінити архітектуру і відео процесор винести окремим модулем – зробити його інтерфейсом. Так як це говорить принцип Dependency Inversion *що абстракції не повинні залежати від реалізації, а реалізації повинні залежати від абстракції.*